Universität Ulm
Abt. Datenbanken und Informationssysteme
Leiter: Prof. Dr. P. Dadam

Dynamische Ablaufänderungen in Workflow-Management-Systemen

DISSERTATION
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Informatik
der Universität Ulm

vorgelegt von

MANFRED REICHERT
aus Ulm

Mai 2000

Amtierender Dekan:	Prof. Dr. U. Schöning
Gutachter:	Prof. Dr. P. Dadam
	Prof. Dr. M. Weber
Tag der Promotion:	27. Juli 2000

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als Wissenschaftlicher Mitarbeiter an der Abteilung Datenbanken und Informationssysteme (DBIS) der Universität Ulm. Die Ausgangsmotivation für die Bearbeitung des Themas „Dynamische Ablaufänderungen in Workflow-Management-Systemen“ bildeten praktische Erfahrungen und Erkenntnisse, die ich in interdisziplinären Forschungsprojekten mit klinischen Partnern gewonnen habe. Die Arbeit selbst ist im ADEPT-Workflow-Projekt entstanden, an dem in der Abteilung DBIS seit 1995 gearbeitet wird (ADEPT steht für Application DEvelopment Based on Pre-Modeled Process Templates).

Mein persönlicher Dank gilt meinem Betreuer Prof. Dr. Peter Dadam, der mich während meiner gesamten Promotionszeit tatkräftig unterstützt hat. In zahlreichen konstruktiven Diskussionen hat er zum inhaltlichen Gelingen dieser Arbeit wesentliche Hilfe geleistet. Darüber hinaus hat seine problemorientierte Sichtweise („What’s the message?“) entscheidend zum Erfolg meiner wissenschaftlichen Veröffentlichungen und Vorträge beigetragen.

Ebenso möchte ich Prof. Dr. Michael Weber herzlich für die Übernahme des Koreferats danken. Die fachlichen Gespräche mit ihm empfand ich stets als äußerst angenehm und motivierend. Insbesondere möchte ich ihm nochmals dafür danken, daß er für zahlreiche der von mir betreuten Diplomarbeiten als Gutachter zur Verfügung stand.

Ein ganz besonderer Dank gilt meinem Kollegen und Freund Clemens Hensinger. In stundenlangen, oftmals sehr „hitzigen“ Diskussionen hat er mich inhaltlich immer wieder ein großes Stück weitergebracht. Ich möchte mich bei ihm und bei Thomas Bauer für das sorgfältige Korrekturlesen der Arbeit bedanken. Mein persönlicher Dank gilt aber auch Thomas Strzeletz und allen anderen Mitgliedern des ADEPT-Teams und der Abteilung DBIS. Sie haben stets für ein angenehmes Arbeitsumfeld, eine positive Stimmung und viel Spaß gesorgt. Ausdrücklich danken möchte ich Rudi Seifert für die sehr gute technische Betreuung. Schließlich gilt mein Dank allen Diplomanden, Praktikanten und wissenschaftlichen Hilfskräften, die mich während meiner Zeit in der Abteilung begleitet haben.

Der größte Dank gebührt jedoch meiner Frau und meinen beiden Kindern. Ihre Liebe, Unterstützung und Geduld hat diese Arbeit erst ermöglicht.

Ulm, im Mai 2000

Kurzfassung

Workflow-Management-Systeme (WfMS) unterstützen die elektronische Abwicklung von Arbeitsprozessen (engl. *Workflow*). Heutige WfMS erzwingen meist die starre Ausführung des vormodellierten Workflow, d.h. von dem einmal geplanten Ablauf kann zur Ausführungszeit nicht oder nur unter erheblichen Problemen abgewichen werden. Einerseits ist dadurch die praktische Tauglichkeit dieser Systeme stark eingeschränkt, andererseits darf die Unterstützung von Ad-hoc-Abweichungen zur Laufzeit (z.B. dynamisches Hinzufügen, Löschen oder Überspringen von Arbeitsschritten) aber nicht dazu führen, daß es im weiteren Verlauf der Workflow-Ausführung zu Konsistenzproblemen (z.B. Lost Updates) oder gar Fehlern (z.B. Aufruf von Anwendungskomponenten mit fehlenden Parameterdaten) kommt. Dies sicherzustellen, wird mit zunehmender Komplexität und Ausdrucksmächtigkeit des verwendeten Workflow-Beschreibungsformalismus (Workflow-Metamodell) jedoch immer schwieriger.

Gegenstand dieser Arbeit bildet ein flexibles Workflow-Metamodell, das ADEPT-Modell. Es dient als formale Grundlage für die korrekte Spezifikation, die flexible Ausführung und die dynamische Änderung von Workflows. Hieraus hervorgegangen ist das ADEPT-WfMS, das seit 1997 an der Abteilung Datenbanken und Informationssysteme der Universität Ulm prototypisch entwickelt wird. Dabei handelt es sich um eine mächtige Entwicklungs- und Laufzeitumgebung für verteilte, vorgangsorientierte Anwendungssysteme. Oberstes Ziel des ADEPT-Ansatzes ist es, ein zuverlässiges und robustes Ausführungsverhalten, bei dem ungewollte „Überraschungen“ zur Laufzeit ausgeschlossen werden, mit der Möglichkeit spontaner Abweichungen vom geplanten Ablauf zu verknüpfen. Entsprechende Funktionen findet man in heutigen WfMS nicht vor.

In dieser Arbeit werden hierzu theoretische Grundlagen und teilweise auch algorithmische Umsetzungen behandelt. Ausgangspunkt bildet ein formales, auf markierten, attributierten Graphen basierendes Workflow-Metamodell (*ADEPT-Basismodell*), das die Konzepte für die Beschreibung von Kontroll- und Datenflüssen sowie verschiedener anderer Workflow-Aspekte bereitstellt. Das ADEPT-Basismodell definiert Regeln für die korrekte Strukturierung von Workflow-Modellen und für ein korrektes dynamisches Verhalten. Hiervon ausgehend wird die dynamische Änderung laufender Workflows mit Hilfe einer vollständigen Menge an Änderungsoperationen ($ADEPT_{flex}$-Kalkül) realisiert. Diese Operationen werden intern auf wohldefinierte Graphersetzungs- und Graphvereinfachungsregeln abgebildet. Mehrere Änderungsoperationen können in einer Änderungstransaktion atomar zusammengefaßt werden, wobei die Anwendung einer Änderungstransaktion nur zulässig ist, wenn dadurch die geforderten Struktureigenschaften der Workflow-Graphen erhalten bleiben und wenn statische bzw. dynamische Korrektheits- und Konsistenzeigenschaften des Workflow (z.B. Verklemmungsfreiheit, korrekte Parameterversorgung aufgerufener Arbeitsschrittprogramme) nicht verletzt werden. Darüber hinaus werden fortschrittliche Konzepte zur Durchführung dynamischer Workflow-Änderungen in WfMS (z.B. Änderungstransaktionen, Unterstützung alternativer Strategien zur Behandlung auftretender Probleme, Unterstützung von Änderungen unterschiedlicher Gültigkeitsdauer) vorgestellt. Abschließend wird ein Überblick zum aktuellen Stand des ADEPT-Prototypen gegeben.

Inhaltsverzeichnis

I Motivation

1 Einleitung 3

1.1 Prozeßorientierte Anwendungssysteme 3
1.2 Inadäquate Lösung: Konventionelle Implementierung 4
1.3 Adäquate Lösung: Workflow-Management-Systeme 5
1.3.1 Entwicklungsaspekte 5
1.3.2 Ausführungsaspekte 7
1.3.3 Potentielle Vorteile aus Anwendersicht 9
1.3.4 Potentielle Vorteile aus Entwicklersicht 9
1.3.5 Mangelhafte Flexibilität und Adaptivität heutiger WfMS 10
1.3.6 Schlußfolgerungen 12
1.4 Ziel und Aufbau der Arbeit 13

2 Adaptivität in existierenden Workflow-Management-Systemen 17

2.1 Dynamische Workflow-Änderungen und ihre Ursachen 17
2.1.1 Planbarkeit von Prozessen 17
2.1.2 Ausnahmen in prozeßorientierten Anwendungssystemen und ihre Behandlung 19
2.1.3 Änderung von Standardprozessen 23
2.1.4 Zusammenfassung 24
2.2 Realisierung flexibler, vorgangsorientierter Anwendungssysteme auf der Grundlage existierender WF-Technologie – Möglichkeiten und Grenzen 25
2.2.1 Prozeßorientierte Koordinationsmodelle und WfMS 26
2.2.2 Objektmigrationsmodelle und dokumentenorientierte WfMS 28
2.2.3 Informationsharing-basierte Koordinationsmodelle und formularbasierte WFs 30
2.2.4 Konversationsstrukturorientierte Koordinationsmodelle 33
2.3 Diskussion ausgewählter WfMS 34
2.3.1 Vertreter prozeßorientierter WfMS 34
2.3.2 Vertreter dokumentenorientierter WfMS 35
2.3.3 Vertreter formularorientierter WfMS 37
2.3.4 Vertreter konversationsstrukturorientierter WfMS 39
2.4 Standards 40
2.5 Zusammenfassung 41

II Theoretische Grundlagen des ADEPT-Modells

3 Das ADEPT-Basismodell 45

3.1 Entwurfsziele 45
3.2 Aktivitätenvorlagen – Elementare Bausteine für die WF-Modellierung 48
3.2.1 Wiederverwendbarkeit von Aktivitätenbeschreibungen 48
3.2.2 Vorlagenschnittstelle 48
3.2.3 Vorlagenimplementierung 53
3.3 Kontrollflußmodellierung im ADEPT-Basismodell 55
3.3.1 Strukturierte Modellierung von Workflows 56
3.3.2 ADEPT-Konstrukte für die Kontrollflußmodellierung 57
3.4 Formale Repräsentation von Kontrollflußgraphen 67
3.4.1 Mengenbasierte Darstellung 68
3.4.2 Nachfolger- und Vorgängerfunktion 71
3.4.3 KF-Graphklassen 73
3.5 Struktur von KF-Graphen 74
3.5.1 Wohlstrukturierte, azyklische KF-Graphen 74
3.5.2 Wohlstrukturierte, azyklische KF-Graphen mit Sync-Kanten 79
3.5.3 Wohlstrukturierte, zyklische KF-Graphen 82
3.6 Kontrollblöcke in KF-Graphen 83
3.6.1 Formale Definition eines Kontrollblocks 84
3.6.2 Bestimmung minimaler Kontrollblöcke 85
3.7 Hierarchische Strukturierung von KF-Graphen 86

4 Dynamisches Verhalten 89

4.1 Zustände von Instanzen elementarer Aktivitäten 89
4.2 Beschreibung von Ausführungszuständen durch Graphmarkierungen 91
4.3 Regeln zur Änderung von KF-Graphmarkierungen 92
4.3.1 Wohlstrukturierte, azyklische KF-Graphen 93
4.3.2 Wohlstrukturierte, azyklische KF-Graphen mit Sync-Kanten 98
4.3.3 Wohlstrukturierte, zyklische KF-Graphen 99
4.4 Bestimmung der Folgemarkierung nach Beendigung einer Aktivität 101
4.5 Ausführung von Sub-Workflows 101
4.5.1 Bearbeitungszustände einer (Sub-)Workflow-Instanz 101
4.5.2 Instanzabhängigkeitsgraph und Ausführung von Sub-Workflows 102
4.6 Dynamische Eigenschaften von KF-Graphen und deren Überprüfung 103
4.6.1 Grundlegende Definitionen 103
4.6.2 Korrektheitsforderungen 105
4.6.3 Azyklische KF-Graphen mit einfachen Sync-Kanten 106

4.6.4 Azyklische KF-Graphen mit einfachen und strikten Sync-Kanten 108
4.6.5 Wohlstrukturierte, zyklische KF-Graphen 113
4.7 Zusammenfassung 113

5 Datenflußmodellierung und -steuerung 114

5.1 Modellierung von Datenflüssen in ADEPT 115
5.1.1 Grundlegender Ansatz 115
5.1.2 Erweiterung um Vorschalt- und Nachschaltdienste 116
5.2 Formale Repräsentation von DF-Schemata 118
5.3 Struktur von DF-Schemata 119
5.3.1 Versorgung obligater Eingabeparameter 120
5.3.2 Ausschluß paralleler Schreibzugriffe auf Datenelementen 122
5.3.3 Vermeidung überflüssiger Schreiboperationen 123
5.4 Effiziente Verfahren zur Sicherstellung von DF-Strukturierungsregeln 124
5.4.1 Sicherstellung der Versorgung obligater Eingabeparameter 124
5.4.2 Ausschluß paralleler Schreibzugriffe auf Datenelementen 135
5.4.3 Vermeidung überflüssiger Schreiboperationen 136
5.4.4 Bildung logischer Blockparameter 136
5.5 Datenkontextmanagement und Datenflußsteuerung 137
5.5.1 Schreiben von Datenelementen 137
5.5.2 Lesen von Datenelementen 138
5.6 Zusammenfassung 140

6 Behandlung planbarer Abweichungen 141

6.1 Behandlung von Vorwärtssprüngen 142
6.1.1 Knotenprioritäten und Priorisierungskanten 142
6.1.2 Überspringen von Aktivitäten mit/ohne Nachholen (Shortcuts) 144
6.1.3 Abschließende Bemerkungen 151
6.2 Behandlung von Rückwärtssprüngen 151
6.2.1 Vormodellierung automatischer Rücksprünge 152
6.2.2 Vormodellierung benutzerinitiierter Rücksprünge 155
6.2.3 Ad-hoc-Rücksprünge in KF-Ausführungsgraphen 158
6.2.4 Rücksetzverfahren bei strukturell unverändertem KF-Ausführungsgraphen 159
6.2.5 Abschließende Bemerkungen 170
6.3 Sonstige Aspekte 170
6.4 Zusammenfassung 171

III Dynamische Workflow-Änderungen mit ADEPT$_{flex}$

7 Die ADEPT$_{flex}$-Änderungsoperationen 175

7.1 Entwurfsziele 176
7.2 Grundlagen 177
7.2.1 Graph- und Zustandstransformationen 177
7.2.2 Änderungsprimitive 178
7.2.3 Nullaktivitäten und Graphvereinfachungsregeln 182
7.3 Die Einfüge-Operation 183
7.3.1 Einfügen von Aktivitäten 184
7.3.2 Einfügen vorgefertigter Kontrollblöcke 198
7.4 Die Lösch-Operation 199
7.4.1 Löschen von Aktivitäten 200
7.4.2 Löschen von Kontrollblöcken 206
7.5 Zusammenfassung weiterer Basisänderungsoperationen 207
7.5.1 Verschieben von Aktivitäten und Kontrollblöcken 207
7.5.2 Hinzufügen und Löschen von Spezialkanten 209
7.5.3 Bildung und Auflösung physischer Blockstrukturen 211
7.5.4 Änderung von DF-Modellen 212
7.5.5 Änderung von Knoten- und Kantenattributen 213
7.6 Realisierung komplexer Änderungsoperationen in ADEPT$_{flex}$ 213
7.6.1 (Dynamisches) Einfügen einer Aktivität zwischen zwei Knotenmengen 213
7.6.2 Ad-hoc-Vorwärtssprünge in Ausführungsgraphen 217
7.7 Änderungstransaktionen 221
7.8 Zusammenfassung 223

8 Permanente und temporäre WF-Instanzänderungen 224

8.1 Motivation und Problemstellung 224
8.2 Verwaltung von Änderungen in einer Änderungshistorie 226
8.3 Kriterium für die Rücknahme temporärer Änderungen 227
8.3.1 Definition von Startbereichen für die ADEPT$_{flex}$ -Operationen 228
8.3.2 Bestimmung des Startbereichs einer Änderungstransaktion 229
8.4 Permanente und temporäre Abänderung einer WF-Instanz 230
8.4.1 Grundlagen 231
8.4.2 Durchführung permanenter Änderungen einer WF-Instanz 231
8.4.3 Durchführung temporärer Änderungen einer WF-Instanz 232
8.4.4 Beispiel für die Einbringung permanenter und temporärer Änderungen 237
8.5 Rücknahme von temporären Änderungen bei Schleifenrücksprüngen 239
8.5.1 Bestimmung der betroffenen Änderungstransaktionen 241

8.5.2 Verfahren zur korrekten Rücknahme temporärer Änderungen 243
8.5.3 Optimierungsmöglichkeiten 247
8.6 Anwendung des Verfahrens bei hierarchischen Workflows 247
8.7 Zusammenfassung 248

9 Der ADEPT$_{workflow}$ -Prototyp 250

9.1 Architektur und Implementierung von ADEPT$_{workflow}$ 250
9.1.1 Überblick 250
9.1.2 Schichten des Systemkerns 251
9.1.3 Ausgewählte Implementierungsaspekte 252
9.1.4 Realisierungsplattform 254
9.2 Durchführung einer Ad-hoc-Abweichung in ADEPT$_{workflow}$ 254
9.2.1 Definitionsphase 255
9.2.2 Analysephase 255
9.2.3 Durchführungsphase 256
9.3 Anwendungsentwicklung in ADEPT$_{workflow}$ 257
9.3.1 Grundlegende Vorgehensweise zur Entwicklung von WF-Anwendungen 257
9.3.2 Klienten zur Erstellung von Workflow- und Organisationsmodellen 258
9.3.3 Klienten zur Kontrolle, Überwachung und Ad-hoc-Änderung von Workflows 260

IV Diskussion und Zusammenfassung

10 Diskussion verwandter Ansätze 265

10.1 Vergleich mit Petri-Netz-basierten Ansätzen 266
10.1.1 Grundlagen 266
10.1.2 Adaption in Petri-Netz-basierten WF-Modellen 269
10.1.3 Zusammenfassende Bewertung 278
10.2 Vergleich mit Statechart-basierten Ansätzen 279
10.2.1 State- und Activitycharts 279
10.2.2 Dynamische Änderungen von Statecharts in HieraStates 279
10.3 Vergleich mit graph- und netzbasierten Ansätzen 281
10.3.1 Adaptives Workflow-Management in PoliFlow 281
10.3.2 Dynamische Änderung von Aktivitätennetzen in WASA 282
10.3.3 Dynamische Evolution von WF-Schemata in WIDE und TRAM 284
10.3.4 Zusammenfassung weiterer Ansätze 286
10.4 Vergleich mit Graphgrammatik-basierten Ansätzen 286
10.4.1 Graphgrammatiken und Graphersetzungssysteme 286
10.4.2 Unterstützung evolutionärer Softwareentwicklungsprozesse in DYNAMITE 287
10.4.3 Komposition von Prozeßschablonen in Obligations 289

10.5 Vergleich mit regelbasierten Ansätzen 289
10.5.1 Spezifikation von Workflows mittels ECA-Regeln 290
10.5.2 Regelbasierte Modellierung von automatischen Ausnahmebehandlungen 290
10.5.3 Regelbasierte Erweiterung von WF-Metamodellen in MOKASSIN 292
10.6 Vergleich mit deklarativen, planbasierten Ansätzen 293
10.6.1 Klassische Planungsverfahren in der Künstlichen Intelligenz 294
10.6.2 Anwendung von Planungstechniken auf den Workflow-Bereich 296
10.6.3 Zusammenfassende Bewertung 301
10.7 Vergleich mit transaktionalen WF-Ausführungsmodellen 302
10.7.1 Erweiterte Transaktionsmodelle und transaktionale Workflows 302
10.7.2 Restrukturierung von Workflows in TAM 303
10.8 Vergleich mit temporalen Logiken, Prozeßalgebren und -grammatiken 304
10.8.1 Temporale Logiken und Prozeßalgebren 304
10.8.2 Prozeßgrammatiken 305
10.9 Vergleich mit sonstigen Ansätzen 305
10.9.1 Deskriptive Workflow-Modellierung in MOBILE 305
10.9.2 Unterstützung von Design-Workflows in CONCORD 306
10.9.3 Tolerieren von Prozeßabweichungen in SENTINEL und PROSYT 306
10.10 Zusammenfassung der Diskussion 307

11 Zusammenfassung 309

Literaturverzeichnis 313

V Anhang

A Übersicht zu wichtigen Funktionen 333

B Beweise 334

C Algorithmen 340

Teil I

Motivation

1 Einleitung

Workflow-Management-Systeme (WfMS) [JaBu96, ReDa00] unterstützen die elektronische Abwicklung von *Geschäftsprozessen* (z.B. Bearbeitung eines Kundenauftrags) entsprechend fest vorgegebener Arbeitsabläufe (engl. *Workflows*). Konventionelle WfMS erzwingen dabei meist eine starre Ausführung des modellierten Workflow (kurz: WF). Spontane Abweichungen vom geplanten Ablauf, wie sie zur Behandlung von Ausnahmen oder zur Anpassung des WF an geänderte Realweltsituationen notwendig werden können, sind in heutigen WfMS dagegen nicht oder nur unter erheblichen Schwierigkeiten möglich. Gegenstand dieser Arbeit ist die Entwicklung eines ausdrucksstarken *WF-Metamodells* zur flexiblen Unterstützung komplexer Arbeitsabläufe. Dieses Metamodell soll eine angemessene Basis für die formale Spezifikation, die flexible Ausführung und die dynamische Änderung von Workflows bilden. Dabei soll ein zuverlässiges und robustes Ausführungsverhalten mit der Möglichkeit, zur Laufzeit spontan vom geplanten Ablauf abweichen zu können, verknüpft werden.

1.1 Prozeßorientierte Anwendungssysteme

Die Steigerung der Produktivität und Wirtschaftlichkeit, die Sicherung der Produktqualität sowie schnelles und flexibles Reagieren auf Marktveränderungen sind heute mehr denn je wesentliche Faktoren für den Erfolg eines Unternehmens [ReDa00]. Die Umsetzung dieser Ziele setzt voraus, daß die betrieblichen Strukturen und Abläufe erfaßt, analysiert und optimiert werden, also das getan wird, was man unter *Geschäftsprozeßreengineering* versteht [HaSt95, NiPi95, RKD96]. Der Einsatz von rechnergestützten Werkzeugen für die Prozeßmodellierung, -simulation und -analyse ist hierbei eine wichtige Hilfe [Sche98a, Sche98b]. Der Erfolg dieser Maßnahmen wird aber nur nachhaltig sein können, wenn es gelingt, die optimierten Prozesse anschließend durch vorgangsorientierte Anwendungssysteme intelligent und flexibel zu unterstützen.

Heutige betriebliche Informationssysteme bieten keine ausreichende Ablaufunterstützung. Die elektronische Informationsverarbeitung in Unternehmen erfolgt überwiegend funktions- und datenbezogen [Brei93]. In den einzelnen Bereichen gibt es Anwendungssysteme (mit oftmals eigener Datenverwaltung) für die Unterstützung spezieller Funktionen, wie zum Beispiel die Abfrage von Kundendaten, die Auftragserfassung, die Angebotserstellung, das Verbuchen von Warenbewegungen oder die Rechnungsschreibung. Diese Funktionen werden durch die Bereitstellung der Daten im jeweiligen Anwendungskontext unterstützt. Ihre logische Verknüpfung, d.h. der zugrundeliegende Workflow, existiert dagegen nur in den Köpfen der beteiligten Mitarbeiter oder in Form von Organisationsanweisungen, falls solche vorhanden sind. Eine durchgängige rechnerbasierte Prozeßunterstützung gibt es dagegen nicht, so daß meist keine direkten Zugriffsmöglichkeiten auf alle relevanten Informationen zu laufenden Geschäftsprozessen bestehen. Darüber hinaus müssen Anwender oftmals ein umständliches und zeitaufwendiges Navigieren durch Systemmenüs in Kauf nehmen, um einzelne Anwendungsfunktionen ausführen zu können. Mit zunehmender Komplexität der Arbeitsabläufe führt diese *passive* Form der Informationsbereitstellung dazu, daß zu erledigende Aufgaben übersehen oder wichtige Abhängigkeiten zwischen ihnen (z.B. Reihenfolgen) nicht beachtet werden. Die Folgen sind Unterlassungsfehler, falsche Entscheidungen, Verzögerungen, Leerlauf oder die unnötige Wiederholung von Aufgabenbearbeitungen [RKD96, RDMK00]. Hinzu kommen hohe Transport- und Liegezeiten (z.B. von Papierakten) sowie Doppelarbeit (z.B. redundante Dateneingaben).

Aus diesen Gründen besteht von Anwenderseite ein einigen Jahren ein zunehmendes Interesse an der Einführung prozeßorientierter Anwendungssysteme [DRK97, LeRo97, SaBa93]. Sie sollen die Mitarbeiter ablaufbezogen unterstützen, indem sie die richtige *Information* der richtigen *Person* zum richtigen *Zeitpunkt* und verknüpft mit der richtigen *Anwendungsfunktion* aktiv so anbieten, daß diese Person die anstehende *Teilaufgabe* möglichst effizient durchführen kann [DRK00]. Ein prozeßorientiertes Anwendungssystem, das die Ausführung von Arbeitsabläufen *aktiv* koordiniert und überwacht, das Anwender ablaufbezogen informiert und das auf potentielle Probleme (z.B. drohende Terminverletzungen) frühzeitig hinweist, kann hier ein weitaus größeres Potential ausschöpfen, als es heutige funktionsorientierte Informationssysteme in der Lage sind zu tun [RDMK00]. Die praktische Nutzung diese Potentials hängt jedoch ganz entscheidend davon ab, welcher Aufwand von Hersteller- bzw. Anwenderseite getrieben werden muß, um (verteilte) prozeßorientierte Anwendungen mit der erforderlichen *Zuverlässigkeit* und *Flexibilität* zu realisieren.

1.2 Inadäquate Lösung: Konventionelle Implementierung

Eine Möglichkeit zur Realisierung prozeßorientierter Anwendungen besteht darin, die Ablaufsteuerung der Prozesse direkt im Anwendungsprogramm zu implementieren, etwa indem die jeweilige Prozeßlogik mittels eines erweiterten endlichen Automaten abgebildet wird. Für jede Prozeßinstanz muß dann eine logische Programminstanz (Betriebssystemprozeß, Thread) erzeugt und ausgeführt werden. Da im allgemeinen nicht alle Funktionen eines Prozesses durch dasselbe Anwendungssystem, sondern durch verschiedene, verteilte Anwendungskomponenten bereitgestellt werden [Kuhn94, Kuhn94a], müssen zur Ausführung der Prozeßschritte ggf. entfernte Anwendungsfunktionen angesteuert werden, z.B. mittels entferntem Prozedur- oder Programmaufruf (vgl. Abbildung 1-1).

Bei „naiver“ Implementierung dieses Ansatzes ergeben sich zahlreiche Probleme: So macht jede Prozeßinstanz auf den Arbeitsplatzrechnern potentiell ein eigenes Fenster auf, wenn dort ein Prozeßschritt zur Ausführung ansteht. Dabei entscheidet nicht der Bearbeiter dieses Schrittes, wann die zugehörige Anwendungsfunktion bei ihm gestartet werden soll, sondern die Ablaufsteuerung. Es gibt auch keine Arbeitsliste, die für den Anwender die anstehenden Aufgaben zusammenfaßt, sondern diese sind implizit durch die Menge der geöffneten Fenster gegeben. Das versehentliche Schließen eines solchen Fensters oder der Absturz eines Anwendungsprogramms erfordert dem entsprechend aufwendige Fehlerbehandlungen, die sich noch schwieriger gestalten, wenn ein Arbeitsplatzrechner mit noch aktiven Anwendungsprogrammen oder die zentrale Ablaufsteuerung abstürzt. Problematisch ist hier zum Beispiel die Behandlung von Prozeßinstanzen, deren Steuerungsprogramm zum Zeitpunkt des Absturzes auf die Rückkehr entfernter Prozeduraufrufe wartet. (Ein Ausweg bildet die Verwendung nichtblockierender entfernter Prozeduraufrufe, was aber die Anwendungsprogrammierung erheblich schwieriger gestaltet.) Der Programmierer der zentralen Ablaufsteuerung muß nicht nur Vorkehrungen für den Absturz dieser zentralen Komponente treffen, sondern auch Situationen berücksichtigen, bei denen entfernte Anwendungen außerplanmäßig terminieren.

Das vielleicht größte Problem, das sich bei der beschriebenen Vorgehensweise ergibt, betrifft die Wartung prozeßorientierter Anwendungen. Während funktionsorientierte Applikationen tendenziell relativ stabil sind, d.h. selten abgeändert werden müssen, werden bei prozeßorientierten Anwendungen naturgemäß sehr viel häufiger Anpassungen erforderlich [EKR95, JST97, KRD95, RHD98, ShKo97]. Aus diesem Grund sind vorgangsorientierte Anwendungssysteme nur dann auf breiter Basis sinnvoll einsetzbar, wenn sie sich rasch und kostengünstig an sich ändernde Ablauf- und Organisationsstrukturen anpassen lassen. Für den skizzierten Ansatz würde dies bedeuten, daß aufgrund der „harten Verdrahtung“ der Prozeßlogik im Anwendungsprogramm, zur Durchführung entsprechender Änderungen in den Programmcode eingegriffen werden muß. Bei solchen Änderungen muß man auch prüfen, ob man hierdurch nicht ungewollt andere Abläufe im Anwendungssystem mit verändert hat. Deshalb sind Eingriffe dieser Art tendenziell zeitaufwendig und fehleranfällig. Dies gilt in besonderem Maße, wenn Änderungen nicht nur auf neu zu erzeugende (also zukünftige) Prozeßinstanzen, sondern – soweit sinnvoll und möglich – auch noch auf bereits laufende Prozesse angewendet werden sollen [CCPP98, EKR95, JoHe98]. Hinzu kommt, daß in vielen Anwendungsbereichen, etwa der Medizin [Dada95, RHD98] oder der Produktentwicklung [BeDa96, DeMa96], ein durch den Rechner erzwungenes, starres schematisches Vorgehen bei der täglichen Arbeit nur auf wenig Akzeptanz stößt. Um Ausnahmesituationen angemessen behandeln oder Anpassungen an geänderte Realweltsituationen vornehmen zu können, muß es ein prozeßorientiertes Anwendungssystem den Benutzern im Einzelfall erlauben, ad hoc vom implementierten Ablauf abzuweichen, etwa durch das Einfügen oder Überspringen von Schritten. Dies wiederum darf in der Folge nicht zu Problemen führen, wie dem Aufruf von Anwendungskomponenten mit fehlenden Eingabeparametern.

Aus diesen Gründen sind die konventionelle Implementierung der WF-Steuerung in den Anwendungsprogrammen und die Ausführung einzelner Prozeßschritte mittels entferntem Prozedur- oder Programmaufruf für die Realisierung prozeßorientierter Anwendungssysteme nur mit Einschränkungen geeignet. Abgesehen von trivialen Anwendungsfällen führt diese Vorgehensweise schnell zu komplexen und kaum mehr wartbaren Anwendungssystemen. Darüber hinaus bedarf es sehr systemnaher Kenntnisse, etwa in den Bereichen Rechnerkommunikation, Prozeßsynchronisation, Logging und Recovery [GrRe93], über die normale Anwendungsprogrammierer nicht verfügen.

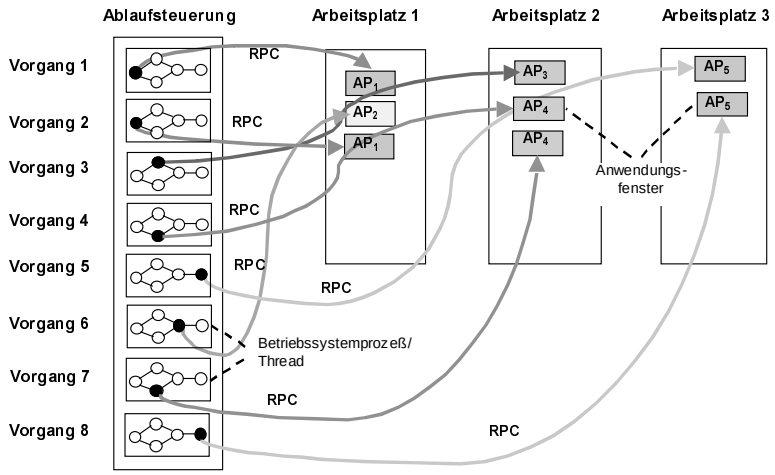


Abbildung 1-1: *Direkte zentrale Ablaufsteuerung (bei mehreren gleichzeitig aktiven Prozessinstanzen)*

Bei systematischer Betrachtung der Alternativen zur Implementierung verteilter, vorgangsorientierter Informationssysteme (siehe [ShRo93]) zeigt sich, daß solche Systeme nur dann mit vertretbarem Aufwand und mit der geforderten Zuverlässigkeit realisierbar sind, wenn die Definition der Ablauflogik der Arbeitsprozesse von der Implementierung ihrer Schritte getrennt wird. Dabei müssen systemnahe Aspekte, wie Synchronisation, Recovery (inkl. Logging), Interprozeßkommunikation oder Datenbereitstellung in heterogenen Umgebungen, vor den Anwendungsprogrammierern versteckt werden. Insbesondere sollte sich die Implementierung einzelner Prozeßschritte nicht komplexer darstellen, als die Programmierung rein funktionsorientierter Anwendungen, selbst wenn später ihre Aufrufreihenfolge geändert werden muß. Das bedeutet, daß es möglich sein sollte, einzelne Prozeßschritte bzw. Funktionen als isolierte, wiederverwendbare Anwendungskomponenten zu implementieren, deren Eingaben beim Aufruf von der Laufzeitumgebung korrekt versorgt werden und die bei ihrer Beendigung lediglich dafür sorgen müssen, daß korrekte Werte für die Ausgabeparameter erzeugt werden. Alle anderen Aufgaben der Ablaufsteuerung und -überwachung sollen von den Laufzeitkomponenten des prozeßorientierten Anwendungssystems übernommen werden [DRK97].

1.3 Adäquate Lösung: Workflow-Management-Systeme

Eine viel versprechende Technologie zur Realisierung prozeßorientierter Anwendungen sind *Workflow-Management-Systeme* (WfMS) [Härd97, Hsu93, Hsu95, JaBu96, ReDa00, VoBe96]. Konventionelle WfMS unterstützen die (teilweise) Abwicklung von *Geschäftsprozessen* entsprechend vorgegebener Arbeitsabläufe. Die elektronische Repräsentation eines solchen Ablaufs wird als *Workflow* (WF) bezeichnet. *Workflow Management* ist dem entsprechend die komplette oder teilweise Automatisierung von Prozessen, bei der Dokumente, Informationen und Aufgaben von einem Anwender zum anderen gemäß prozeßorientierter Regeln weitergeleitet werden [WFMC96]. Eine charakteristische Eigenschaft „echter“ WfMS ist, daß sie das Anwendungssystem in einen *Ablauf-* und in einen *Funktionsteil*, genauer eine Sammlung von Funktionsbausteinen, zerlegen (siehe Abbildung 1-2). Dazu werden die verschiedenen Aspekte des WF, wie *Tätigkeiten*, zuständige *Bearbeiter*, Ausführungsreihenfolgen und -bedingungen (*Kontrollflüsse*) sowie *Informationsflüsse*, von der Implementierung der Bearbeitungsschritte separiert und dem WfMS gegenüber explizit bekannt gemacht. Zu diesem Zweck stellt das WfMS Komponenten für die Spezifikation, Verifikation, Steuerung und Überwachung von WFs bereit. Typisch ist eine Aufteilung in *Modellierungs-* und *Laufzeitkomponenten*.

1.3.1 Entwicklungsaspekte

Die WfMS-basierte Entwicklung prozeßorientierter Anwendungssysteme erfolgt – vereinfacht ausgedrückt – in zwei Stufen:

1. Definition der einzelnen Prozeßschritte (inkl. ihrer Eigenschaften) und Modellierung des Kontroll- und Datenflusses zwischen ihnen (Prozeßlogik)

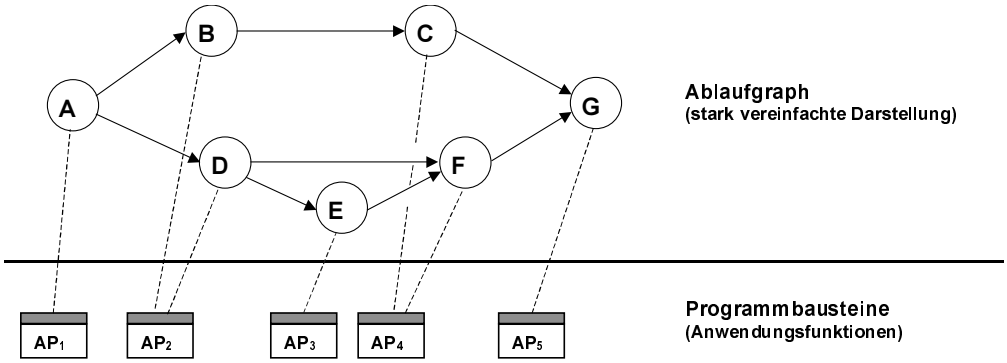


Abbildung 1-2: *Grundprinzip der Trennung von Ablauflogik und Anwendungscode in WfMS*

2. Auswahl geeigneter Softwarebausteine für die einzelnen Prozeßschritte bzw. Festlegung der geforderten Funktionalität und Schnittstellen für die Implementierung

Für jeden zu unterstützenden Prozeßtyp muß mit den Modellierungskomponenten des WfMS eine *Prozeßvorlage* (*WF-Schema, WF-Modell*) erstellt und im System hinterlegt werden. Eine solche Vorlage beschreibt einen Ausschnitt eines Geschäftsprozesses und legt fest, welche Tätigkeiten von welchen Personengruppen in welcher Reihenfolge und unter Nutzung welcher Informationen und Anwendungskomponenten rechnergestützt durchgeführt werden sollen. Die Grundlage hierfür bildet das vom WfMS bereitgestellte *WF-Metamodell* (*WF-Beschreibungssprache*), das die für die Beschreibung der verschiedenen WF-Aspekte verfügbaren Konstrukte vorgibt. Je nach Ausdrucksmächtigkeit können mehr oder weniger komplexe Prozesse abgebildet und vom WfMS unterstützt werden.

Die Mehrzahl der heutigen WfMS bietet für die Definition von Prozeßvorlagen *graphische Beschreibungssprachen* an, bei denen die einzelne Prozeßschritte durch Kreise, Rechtecke oder andere Symbole und die Bearbeitungsreihenfolgen zwischen ihnen durch Pfeile festgelegt werden. Man kann in diesen WF-Modellen zum Ausdruck bringen, ob Prozeßschritte sequentiell oder parallel ausführbar sein sollen und ob es alternative Ausführungspfade gibt. Für den letztgenannten Fall müssen zusätzlich qualitative (prädikative) Angaben gemacht werden (z.B. abhängig von Daten des Workflows), die es dem WfMS zur Ausführungszeit ermöglichen zu entscheiden, welche Alternative gewählt werden soll. Je nach Ausdrucksmächtigkeit der verwendeten Beschreibungssprache können noch weitere Kontrollabhängigkeiten abgebildet werden [Jabl95]. Neben der Definition von Ausführungsreihenfolgen und -bedingungen (*Kontrollflüsse*) haben sich in der Praxis die formale Beschreibung von Prozeßschritten (*Funktionen*), die Festlegung von *Datenflüssen* (durch die Verknüpfung der Ein- und Ausgabeparameter verschiedener Schritte), die Abbildung *organisatorischer Regelungen* (einschl. Zugriffs- und Ausführungsberechtigungen und Vertreterregelungen) und die Einbeziehung von *Zeitbeschränkungen* (z.B. Deadlines, Termine) als wichtige Aufgaben der WF-Modellierung herauskristallisiert [BaWa95a, CKO92, Grim97, JBS97, JaBu96]. Eine typische Option vieler WfMS ist in diesem Zusammenhang die Animation und Validation des Systemverhaltens, unabhängig vom Implementierungsstand der einzelnen Prozeßschritte [Roll96]. Hierdurch wird die Anwendungsentwicklung erheblich vereinfacht.

Die einzelnen Schritte einer Vorlage bilden die logischen Arbeitseinheiten des modellierten Prozesses. Sie werden als *Aktivitäten* bezeichnet. Ihre Bearbeitung wirkt auf ein oder mehrere *Arbeitsobjekte*, bei denen es sich sowohl um physische Objekte (Laborprobe, Werkstück, usw.) als auch um informationstragende Objekte (Datensatz, Dokument usw.) handeln kann. Um die Bearbeitung von Aktivitäten rechnerunterstützt durchführen zu können, müssen ihnen bei der WF-Definition oder dynamisch zur Laufzeit geeignete Anwendungs- bzw. Programmbausteine (sog. *Aktivitätenprogramme*) zugeordnet werden. Bei vielen WfMS kann festgelegt werden, wie die Eingabeparameter des Bausteins bei seinem Aufruf aus dem *WF-Datenkontext* versorgt und wie die Ausgabeparameter bei seiner Beendigung in den WF-Datenkontext übernommen werden sollen. Die Bausteine selbst werden, zumindest aus logischer Sicht, separat entwickelt. Es kann sich dabei um einfache Bildschirmmasken handeln, über die Informationen ein- und ausgegeben werden, es können aber auch bereits existierende Programme sein, die an dieser Stelle eine bestimmte Anwendungsfunktion (z.B. „Angebot schreiben“, „Bestellung aufnehmen“, usw.) realisieren. Einige WfMS unterstützen in diesem Zusammenhang die Verwaltung von *Aktivitätenvorlagen*. Dabei handelt es sich um Schablonen für einzelne Aktivitäten, die unabhän-

gig von einer konkreten Prozeßvorlage definiert und in verschiedenen WF-Modellen wiederverwendet werden können. Eine Aktivitätenvorlage beschreibt die durchzuführende Tätigkeit (z.B. Ein- und Ausgabeparameter) und ordnet ihr geeignete Anwendungsbausteine zu. Je nach Ansatz kann diese Zuordnung statisch oder dynamisch erfolgen. Im letztgenannten Fall werden bei der Vorlagendefinition lediglich Prädikate für die Auswahl geeigneter Anwendungsbausteine festgelegt.

Darüber hinaus gestatten es viele WF-Metamodelle, die Prozeßvorlagen hierarchisch zu strukturieren. Dem entsprechend wird zwischen elementaren und komplexen Aktivitäten unterschieden. *Elementare Aktivitäten* legen Arbeitseinheiten fest, die aus Sicht der Modellierung nicht mehr weiter unterteilt werden können. *Komplexe Aktivitäten* (sog. *Sub-Workflows*) dagegen umfassen ihrerseits eine Menge von Sub-Aktivitäten, deren Kontroll- und Datenflußbeziehungen durch ein separates (Sub-)WF-Modell beschrieben werden. In Bezug auf elementare Aktivitäten wird weiter zwischen automatischen und manuellen Aktivitäten unterschieden. *Automatische Aktivitäten* erfordern keine Benutzerinteraktionen und werden unter der Kontrolle des WfMS ausgeführt. Sobald ihre Vorbedingungen erfüllt sind, werden sie automatisch vom WfMS gestartet und das mit ihnen verknüpfte Aktivitätenprogramm ausgeführt. *Manuelle Aktivitäten* dagegen repräsentieren Arbeitsschritte, die von prozeßbeteiligten Akteuren, in der Regel unterstützt durch ein Anwendungsprogramm, bearbeitet werden. Beispiele sind das Ausfüllen einer Bildschirmmaske oder das Editieren eines Dokumentes. Eine manuelle Aktivität wird bei ihrer Aktivierung nicht automatisch gestartet, sondern allen zur Ausführung berechtigten Akteuren in ihren *Arbeitslisten* (engl. *Worklists*) angeboten. Sobald einer von ihnen die Bearbeitung der Aktivität übernimmt, werden die entsprechenden Arbeitseinträge (engl. *Work Items*) aus den Arbeitslisten der anderen Akteure wieder entfernt. Das Management der Arbeitslisten übernimmt die *Arbeitslistenverwaltungskomponente* (engl. *Worklist Handler*) des WfMS.

Um manuelle Aktivitäten zur Ausführungszeit potentiellen Bearbeitern in ihren Arbeitslisten anbieten zu können, müssen für sie bei der WF-Modellierung geeignete Festlegungen getroffen werden. In der Regel werden dabei keine konkreten Personen als Bearbeiter zugeordnet, sondern spezifische Ausführungsorgane, die bezüglich der Aufgabenzuteilung sog. *Rollen* übernehmen. Eine *Rolle* (z.B. Sachbearbeiter, Arzt, Sekretärin) faßt dabei eine Grundmenge von Fähigkeiten und Kompetenzen zusammen, welche die Benutzer mit dieser Rolle zur Durchführung bestimmter Tätigkeiten qualifizieren. Ausführungsorgane in derselben Rolle werden als austauschbar betrachtet, d.h. alle Personen, die sich über ihre Rolle für die Bearbeitung einer Tätigkeit qualifizieren, können diese auch ausführen. Ggf. kann ein Benutzer auch mehrere Rollen einnehmen. Einige WfMS gestatten es darüber hinaus, die Auswahl der Bearbeiter einer Aktivität nicht nur auf der Grundlage von Rollen, sondern auch in Abhängigkeit von ihrer Zugehörigkeit zu bestimmten organisatorischen Einheiten vorzunehmen. Zu diesem Zweck müssen zusätzlich zu den Prozeß- auch die Organisationsstrukturen des Unternehmens (z.B. Abteilungen und Personalstruktur) – soweit sie für die Ablaufsteuerung relevant sind – im WfMS modelliert und gespeichert werden [Kony96, Kubi98]. Neben der Beschreibung von Rollen, Organisationseinheiten und Benutzern müssen bei der Organisationsmodellierung auch Vertreterregelungen festgelegt werden. Auf ihrer Grundlage sorgt das WfMS bei der Abwesenheit von Personen selbständig dafür, daß zu erledigende Tätigkeiten nicht liegenbleiben, sondern an andere Personen zur Bearbeitung weitergeleitet werden [RuWe94]. Zu erwähnen bleibt, daß die Auswahl des Bearbeiters eines Prozeßschrittes auch in Abhängigkeit von den Bearbeitern vorausgehender Aktivitäten erfolgen kann. Ein *variable Bearbeiterzuordnung* könnte z.B. festlegen, daß eine bestimmte Aktivität vom selben Akteur bearbeitet werden soll, der eine bestimmte vorausgehende Aktivität durchgeführt hat.

Im Anschluß an die WF-Modellierung und die Anknüpfung der Anwendungsbausteine erhält man ein komplettes Anwendungssytem. Üblicherweise bieten WfMS Standardoberflächen an, mit denen sich autorisierte Benutzer beim System registrieren, neue Prozeßsinstanzen erzeugen sowie zur Ausführung anstehende Aktivitäten starten, beenden, unterbrechen oder wiederholen können. Die Generierung und Verwaltung der Arbeitslisten übernimmt das WfMS. Die Standardoberflächen können durch angepaßte, anwendungsspezifische Benutzeroberflächen ersetzt werden, wofür die meisten WfMS auch Programmierschnittstellen anbieten.

1.3.2 Ausführungsaspekte

Um einen neuen Vorgang in Gang zu setzen, muß ein berechtigter Akteur zuerst eine geeignete Prozeßvorlage (z.B. „Bearbeitung eines Kundenauftrags“) aus dem Repository auswählen. Nach der Fest-

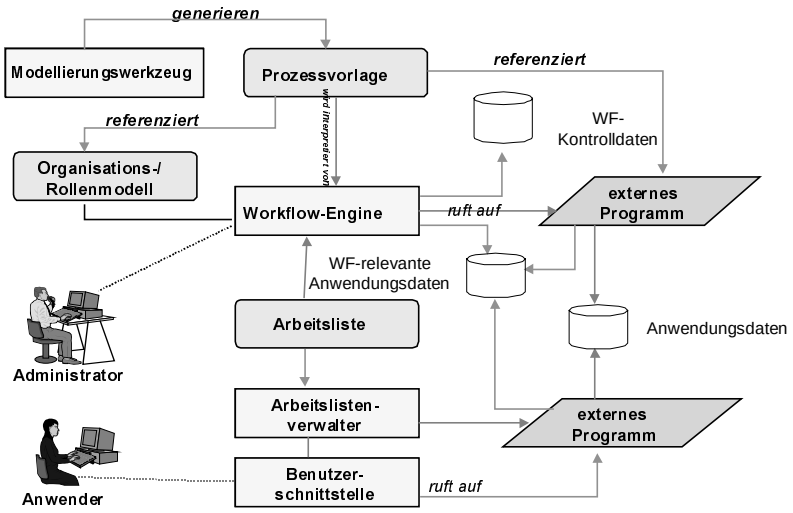


Abbildung 1-3: *Komponenten eines WfMS (Quelle: [WFMC96])*

legung vorgangsspezifischer Daten (z.B. Kunden- und Auftragsnummer) wird vom WfMS eine neue *WF-Instanz* (*Workflow*) erzeugt und gestartet. Sie wird dann über ihre komplette Lebensdauer hinweg, ggf. also auch über einen längeren Zeitraum (Tage, Wochen, Monate), vom System begleitet. Auf der Grundlage einer Prozeßvorlage können mehrere WF-Instanzen unabhängig voneinander erzeugt und in unterschiedlichem Kontext ausgeführt werden. Die Laufzeitkomponenten des WfMS (sog. *WF-Engine*) steuern und überwachen die Ausführung der WF-Instanzen, verwalten instanzspezifische Daten, bieten die zur Bearbeitung anstehenden manuellen Arbeitsschritte den zuständigen Akteuren in ihren Arbeitslisten an, führen automatische Schritte selbständig durch und rufen die für die Ausführung einer Aktivität passenden Anwendungsbausteine auf [ReDa00]. Den prinzipiellen Ablauf einer WF-Instanz zur Laufzeit zeigt Abbildung 1-4. Ein weiterer wichtiger Vorteil ist, daß die Behandlung von Laufzeitfehlern (z.B. Rechner- oder Programmabstürzen) durch das WfMS erfolgt.

Die Ausprägung einer elementaren WF-Aktivität wird als *Aktivitäteninstanz* bezeichnet. Sie repräsentiert einen einzelnen Arbeitsschritt zur Laufzeit und befindet sich zu jedem Zeitpunkt der Ausführung immer in einem wohldefinierten Bearbeitungsstatus (*Aktivitätenstatus*). Üblich ist die Unterscheidung zwischen Zuständen wie „nicht ausführbar“, „ausführbar“, „gestartet“ und „beendet“ [WFMC96]. Der Status einer WF-Instanz ergibt sich aus den Zuständen ihrer Aktivitäteninstanzen, d.h. den Informationen zu bisher durchgeführten bzw. sich in Ausführung befindlichen Schritten. Hierzu gehören nicht nur Statusinformationen, sondern auch die von den Aktivitätenprogrammen produzierten Ausgabedaten. Sie werden im *Datenkontext* der WF-Instanz persistent verwaltet.

Vor der Ausführung eines Aktivitätenprogramms werden seine Eingabeparameter aus dem *WF-Datenkontext* versorgt. Umgekehrt werden die Ausgabeparameter einer Aktivität – sie repräsentieren die Ergebnisse der Aufgabenbearbeitung – nach ihrer Beendigung in den WF-Datenkontext übernommen. Anschließend stehen sie für nachfolgende Aktivitätenausführungen zur Verfügung. Generell lassen sich die für die WF-Ausführung relevanten Daten in drei Kategorien einteilen [WFMC96] (vgl. Abbildung 1-3): *WF-Kontrolldaten* werden von der WF-Engine für die Steuerung und Überwachung von WF-Instanzen sowie für Recovery-Zwecke benötigt. Sie umfassen Informationen zum aktuellen Status des WF und seinen Aktivitäten sowie zu seinem bisherigen Verlauf (z.B. Ausführungshistorie). WF-Kontrolldaten können von Anwendern nicht direkt manipuliert werden bzw. dürfen von ihnen nur zu Informationszwecken gelesen werden. *Anwendungsdaten* werden von den in den WF eingebetteten Aktivitätenprogrammen erzeugt oder für deren Ausführung benötigt. Im Gegensatz zu Kontrolldaten können sie direkt von Anwendern, mit den dafür vorgesehenen Aktivitätenprogrammen, manipuliert werden, besitzen aber im allgemeinen keinen Einfluß auf die WF-Steuerung. Eine Sonderrolle nehmen *WF-relevante Anwendungsdaten* ein. Sie entsprechen Anwendungsdaten (bzw. daraus ableitbaren Informationen), die zur Steuerung des WF verwendet werden. Üblicherweise handelt es sich dabei um Aktivitätenausgaben, die als Grundlage für Verzweigungsentscheidungen dienen.

Zu erwähnen bleibt, daß ein WfMS eine unterschiedlich starke Kontrolle in Bezug auf Anwendungsdaten ausüben kann. Eine Möglichkeit besteht darin, die Schreib- und Lesezugriffe von Aktivitäten-

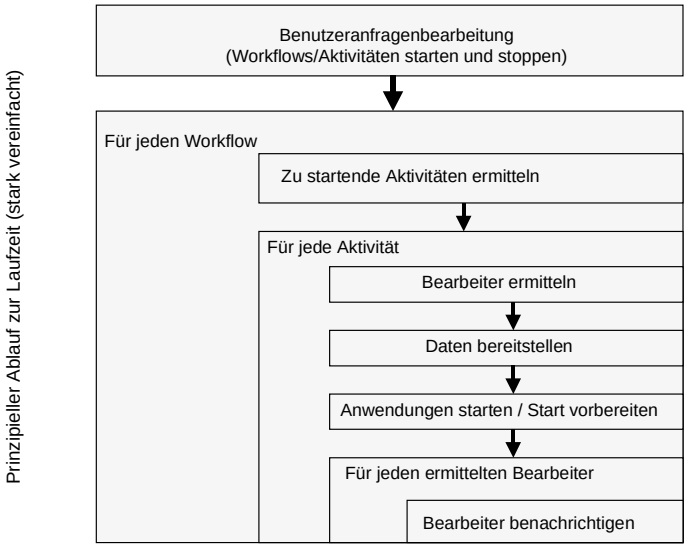


Abbildung 1-4: *Prinzipieller Ablauf eines Workflows zur Laufzeit*

programmen auf Anwendungsdaten außerhalb der Kontrolle des WfMS durchzuführen. Aufgrund der dann fehlenden Kenntnis der Datenabhängigkeiten zwischen Aktivitäten gestalten sich hier WF-Änderungen schwierig und fehleranfällig. Alternativ könnten Anwendungsdaten komplett über das WfMS ausgetauscht werden, was die vollständige Beschreibung der Datenflüsse im WF-Modell und die Verwaltung der Anwendungsdaten durch das WfMS erfordert. Diese Vorgehensweise ist bei datenintensiven Anwendungen nicht immer sinnvoll und kann sich nachteilig auf den Gesamtdurchsatz des WfMS auswirken. Aus diesen Gründen empfiehlt es sich, die Datenabhängigkeiten zwischen Aktivitäten dem WfMS gegenüber bekannt zu machen, für Anwendungsobjekte aber jeweils explizit zu entscheiden, ob ihre Werte oder nur Referenzen auf sie über das WfMS ausgetauscht werden sollen.

1.3.3 Potentielle Vorteile aus Anwendersicht

Aus Anwendersicht bietet der Einsatz eines WfMS viel versprechende Perspektiven (vgl. [RKD97]):

- Anstehende Tätigkeiten und die zu ihrer Bearbeitung benötigten Daten werden automatisch an die zuständigen Stellen übermittelt, d.h. unproduktive Liege- und Transportzeiten entfallen.
- Die Prozeßausführung wird vom System aktiv überwacht, d.h. Akteure werden auf (noch) nicht durchgeführte Tätigkeiten hingewiesen. Dadurch können Unterlassungsfehler reduziert werden.
- Das WfMS verwaltet für jeden Benutzer Arbeitslisten und räumt ihm unterschiedliche Sichten darauf ein (Anordnung der Arbeitslisteneinträge nach Kontext, Aufgabentyp, Dringlichkeit, usw.)
- Wählt ein Anwender eine Tätigkeit aus seiner Arbeitsliste aus, startet das WfMS das damit verknüpfte Anwendungsprogramm automatisch mit den richtigen Aufrufdaten. Das zeitaufwendige Navigieren durch Menüs, wie es bei funktionsorientierten Anwendungen notwendig ist, entfällt.
- Das WfMS besitzt Kenntnis über den Status laufender Vorgänge, so daß die beteiligten Akteure ablaufbezogen informiert werden können. Zu jeder WF-Instanz kann vom WfMS erfragt werden, welche Aktivitäten bereits abgeschlossen worden sind, welche gerade in Bearbeitung sind oder welche noch ausstehen. Auch die (Rück-)Verfolgung bereits beendeter Vorgänge ist möglich.
- Die im Verlauf eines Vorgangs durchgeführten Tätigkeiten, ihre Dauer und ihre Ergebnisse werden dokumentiert. Dadurch ergibt sich die Chance zu einer prozeßbegleitenden Betriebsdatenerfassung und damit zu einer prozeßorientierten Kostenrechnung und Qualitätssicherung.

1.3.4 Potentielle Vorteile aus Entwicklersicht

Im vorangegangenen Abschnitt wurden potentielle Vorteile prozeßorientierter Anwendungssysteme aus Anwendersicht aufgezeigt. Sie sehen dem Anwendungssystem allerdings nicht unmittelbar an, ob es mit Hilfe eines WfMS oder auf der Grundlage konventioneller Entwicklungsmethoden realisiert worden ist. Welche Vorteile bietet also die beschriebene Vorgehensweise – Trennung des Anwen-

dungssystems in Ablauf- und Funktionsteil – aus Sicht der Anwendungsentwicklung? Wesentlich ist, daß die Prozeßlogik dem WfMS gegenüber explizit durch die Modellierung bekannt gemacht und nicht im Programmcode „versteckt“ wird. Hieraus ergeben sich die folgenden Perspektiven:

1. Vor der Einführung eines WfMS muß eine gründliche Analyse der betrieblichen Prozeß- und Organisationsstrukturen durchgeführt und geeignet aufbereitet werden. Diese Ist-Aufnahme gibt einen Einblick in die aktuellen Prozesse und bildet die Basis für das Aufdecken von Schwachstellen und mögliche Prozeßoptimierungen. Der Einsatz von Geschäftsprozeßmodellierungswerkzeugen [Sche96] kann hierbei eine wichtige Hilfe sein. Durch die Umsetzung der Prozesse in formale Modelle sowie durch Prozeßsimulationen und -analysen können frühzeitig Leerlaufzeiten oder Kapazitätsengpässe aufgedeckt und beseitigt werden [RKD96].
2. Die Abbildung der modellierten Geschäftsprozesse auf WF-Modelle ist in der Regel relativ direkt möglich, wodurch sich Umsetzungsfehler vermeiden oder zumindest reduzieren lassen. Darüber hinaus kann der modellierte Ablauf mit den Anwendern „durchgespielt“ werden, etwa durch Animation [Roll96], noch bevor eine Zeile Programmcode geschrieben wird. Dadurch läßt sich frühzeitig feststellen, ob die Umsetzung der Vorgaben in das WfMS-Ablaufmodell korrekt erfolgt ist.
3. Ein weiterer Vorteil wird erkennbar, wenn eine Prozeßvorlage später geändert wird und das prozeßorientierte Anwendungssystem deshalb angepaßt werden muß. Wurde bei der Implementierung der Anwendungsfunktionen sorgfältig vorgegangen, so kann die Reihenfolge der Aktivitäten modifiziert oder es können neue Aktivitäten hinzugefügt werden, ohne daß hiervon die bereits existierenden Programmbausteine betroffen sind. Und selbst wenn dies der Fall wäre, sind die Änderungen bzw. deren Auswirkungen überschaubarer als bei konventionell implementierten Anwendungssystemen. Vom Prinzip her lassen sich WfMS-basierte Anwendungen deshalb einfacher an geänderte Geschäftsprozesse anpassen, d.h. Wartungskosten werden reduziert [DRK97].
4. Die explizite, formale Beschreibung der Prozeßlogik eröffnet die Möglichkeit, das zukünftige Systemverhalten vorab zu evaluieren. Hierdurch können Entwurfsfehler, wie nicht aktivierbare Prozeßschritte, noch vor der Inbetriebnahme des Anwendungssystems bzw. noch vor dessen eigentlicher Implementierung, entdeckt werden [Aals97, Elli79, OSS94].
5. Die Durchführung der Ablaufsteuerung durch das WfMS bietet die Möglichkeit, die Anwendungsentwicklung durch Bereitstellung entsprechender Systemdienste für die Fehler- und Ausnahmebehandlung sowie für die Überwachung von Zeitvorgaben für Aktivitäten drastisch zu vereinfachen. Dadurch erhöht sich die Zuverlässigkeit der resultierenden Anwendungssysteme.

In Verbindung mit dem zunehmenden Trend zur komponentenbasierten Software-Entwicklung („Component Ware“) [Adle95, NTMS91, NGT92] hat WF-Technologie das Potential, die Implementierung vorgangsorientierter Anwendungen nachhaltig zu verändern. Die Realisierung eines prozeßorientierten Anwendungssystems könnte auf mittlere Sicht dadurch erfolgen, daß der Entwickler aus einer Bibliothek eine geeignete Prozeßvorlage auswählt, sie an die Erfordernisse der gegebenen Anwendungsdomäne anpaßt und dann den einzelnen Aktivitäten vorgefertigte Anwendungsbausteine aus einer Komponentenbibliothek zuweist (im Sinne eines "Plug-and-Play"). Hiervon ausgehend könnte das WfMS ein neues Anwendungssystem generieren. Insgesamt können WfMS dazu beitragen, die vor Ort eingesetzten Anwendungssysteme prozeßorientiert zu integrieren und so eine gemeinsame Basis für die Verwaltung von Arbeitsabläufen zu schaffen.

1.3.5 Mangelhafte Flexibilität und Adaptivität heutiger WfMS

Das beschriebene Potential WfMS-basierter Anwendungssysteme ist bislang nur rudimentär nutzbar, da die am Markt verfügbaren WfMS noch zahlreiche Schwächen aufweisen. Die meisten von ihnen sind noch ein erhebliches Stück davon entfernt, betriebliche Abläufe in dem für einen breiteren Einsatz erforderlichen Maße unterstützen zu können. Zum einen fehlt es an einer methodischen Unterstützung für die fehlerfreie Modellierung und Implementierung von Workflows, zum anderen gibt es zahlreiche Anwendungsgebiete, deren Anforderungen bezüglich Modelliermächtigkeit, Funktionalität, Flexibilität und Performanz die Fähigkeiten derzeitiger WfMS bei weitem übersteigen.

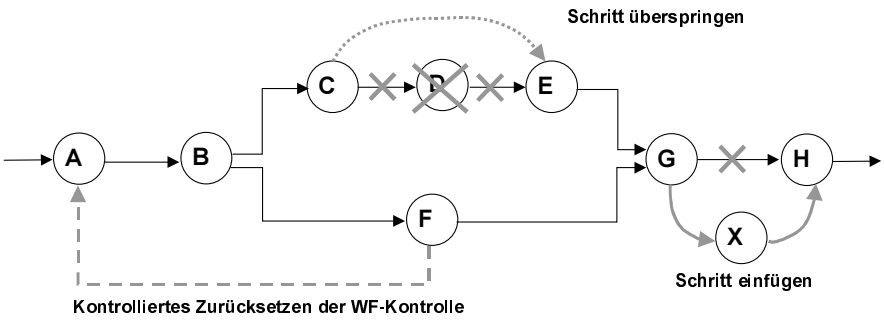


Abbildung 1-5: *Laufzeitabweichungen vom vormodellierten Workflow*

Viele WfMS erzwingen eine starre Ausführung der modellierten Workflows [ReDa98]. Dem gegenüber stehen Anwendungsgebiete, wie die Medizin [MüRa99, RDMK00] oder die Produktentwicklung [BDS98, DeMa96], bei denen Arbeitsabläufe eine derart hohe Variabilität und Dynamik aufweisen, daß eine exakte Abbildung aller Prozeßvarianten (inkl. Ausnahmen) in ein formales WF-Modell und eine starre rechnergestützte Ausführung häufig nicht möglich sind [RHD98]. Vielmehr kommt es hier immer wieder einmal vor, daß von dem geplanten Ablauf spontan abgewichen werden muß, etwa bei Verschlechterung des Zustands eines Patienten oder bei Fehlern im Produktentwurf. Solche Ausnahmen sind nicht vollständig vorhersehbar bzw. hätten, falls alle denkbaren Abweichungen vormodellierbar wären, sehr unübersichtliche WF-Modelle zur Folge [Dada95, StMi95]. Ein WfMS muß es deshalb berechtigten Akteuren im Einzelfall erlauben, spontan von dem modellierten, d.h. dem im Anwendungssystem implementierten Ablauf abzuweichen. Beispiele sind das dynamische Überspringen von Aktivitäten, das Ad-hoc-Einfügen nicht eingeplanter Schritte, die dynamische Vertauschung von Ausführungsreihenfolgen (z.B. durch Vorziehen der Bearbeitung noch nicht aktivierter Schritte) oder das Zurücksetzen der Ausführung in einen früheren Bearbeitungszustand (vgl. Abbildung 1-5). Auch die für Aktivitäten festgelegten Attribute (z.B. Bearbeiterzuordnungen) sowie die von Prozeßvorlagen referenzierten Modelle (z.B. Organisationsmodell) müssen dynamisch abänderbar sein.

Heutige WfMS bieten im Hinblick auf Ad-hoc-Änderungen kaum Unterstützung. Von dem einmal modellierten WF kann zur Ausführungszeit nicht oder nur unter erheblichen Schwierigkeiten abgewichen werden [Hein98b, KRD95, Nutt96]. Prozeßorientierte WfMS wie MQSeries Workflow [LeRo00] oder Staffware [KaDe96], die die in Abschnitt 1.3.1 beschriebene Trennung von Prozeßlogik und Anwendungscode noch am besten umsetzen, erlauben es erst gar nicht, in die Kontrolle laufender WF-Instanzen einzugreifen. Statt dessen müssen Ausnahmen außerhalb des Systems behandelt werden, wodurch eine Diskrepanz zwischen dem realen Arbeitsprozeß und seiner elektronischen Repräsentation entsteht. Aus diesem Grund sind diese WfMS nur auf einfache, relativ starre Prozesse anwendbar. Einige dokumentenorientierte WfMS, wie ProMInanD [Karb94], Workflow [JKMW97] oder InConcert [McSa93], gestatten es zwar, die Ausführung einzelner Arbeitsschritte auszulassen oder neue Arbeitsschritte direkt vor oder nach der Bearbeitung einer Aktivität einzufügen – bereits diese einfachen Änderungsmöglichkeiten werden aber mit einem hohen Preis bezahlt. So kann bei den meisten WfMS dieser Kategorie nicht ausgeschlossen werden, daß es aufgrund der vorgenommenen Laufzeitabweichungen in der Folge zu Konsistenzproblemen (z.B. Lost Updates, Verklemmungen) oder Fehlern (z.B. Aufruf von Aktivitätenprogrammen mit fehlenden Eingabeparametern) kommt. Zwar können diese Fälle bei den genannten WfMS zur Modellierungszeit systemseitig ebenfalls nicht erkannt und behandelt werden, hier hat aber zumindest der Entwickler die Möglichkeit korrigierend einzugreifen.

Auch die in Abschnitt 1.3.4 unter Punkt 3 diskutierte Wartungsfreundlichkeit ist derzeit noch nicht wirklich realisiert. Sie wäre in reiner Form nur dann möglich, wenn alle Anwendungsbausteine so gekapselt und autonom wären, daß sie in beliebiger Reihenfolge ausführbar sind. Da oftmals ein logischer Zusammenhang zwischen den Aktivitäten eines WF besteht und meist auf gemeinsame Anwendungsdaten zugegriffen wird, ist eine völlige Unabhängigkeit dieser Art meistens nicht gegeben. Besonders wichtig ist es deshalb, die zwischen den Aktivitäten bzw. ihren Anwendungskomponenten bestehenden Datenflüsse zu modellieren und dem WfMS gegenüber explizit bekannt zu machen [ReDa98]. Hierdurch kann bei Änderungen der Prozeßvorlage zumindest erkannt werden, welche begleitenden Anpassungen am Datenfluß und ggf. an den Aktivitätenprogrammen vorgenommen werden müssen, um nach der Änderung weiter einen fehlerfreien Ablauf zu garantieren. Bei den meisten WfMS werden solche Anpassun-

gen durch die implizite oder unvollständige Modellierung der Datenflüsse oder durch die nicht saubere Trennung von Kontroll- und Datenfluß erschwert. Auch bei WfMS, bei denen diese beiden Modellaspekte getrennt voneinander beschreibbar sind, mangelt es an einer methodischen Unterstützung für die *korrekte* Modellierung von Datenflüssen. So werden zur Modellierungszeit keine Vorkehrungen getroffen, die gewährleisten, daß zur Laufzeit die Eingabeparameter aufgerufener Aktivitätenprogramme vollständig versorgt sind. Dies sicherzustellen, bleibt die Aufgabe des WF-Entwerfers.

Unter Punkt 4 wurde das Potential einer WfMS-basierten Anwendungsentwicklung hinsichtlich einer formalen Validation erwähnt. Typische Probleme, die sich in Verbindung mit prozeßorientierten Anwendungen ergeben können, sind Blockierungen der Prozeßausführung zur Laufzeit, Ablaufzweige die aufgrund fehlerhafter Spezifikationen niemals durchlaufen werden oder der Aufruf von Aktivitätenprogrammen mit fehlenden Eingabedaten. Obwohl es im Hinblick auf Dynamik-Aspekte interessante Vorarbeiten gibt, unter anderem im Bereich der formalen Beschreibung dynamischer Systeme (z.B. mittels Petri-Netzen oder erweiterten Automaten [Part98]), sind bisher erst wenige ernsthafte Bemühungen unternommen worden [AAH98, Aals97, Ober96, WoWe97], sie auf den WF-Bereich zu übertragen. Besonders wichtig wird die formale Validation des Systemverhaltens, wenn man dynamische Änderungen einzelner Workflows zur Laufzeit, wie oben beschrieben, unterstützen will. Hier sind klare, formale Korrektheitskriterien notwendig, die systemseitig überprüft und ggf. auch erzwungen werden müssen [ReDa97]. – Eine Diskussion weiterer Limitationen derzeitiger WfMS findet man in [AbSa94, BeDa96, DRK00, JaKe97, JKMW97, MAGK95, VLP95, Wodt95].

1.3.6 Schlußfolgerungen

WfMS sind eine relativ junge Technologie, die auf lange Sicht das Potential hat, die Entwicklung vorgangsorientierter Anwendungssysteme nachhaltig zu verändern. Faktisch macht sie die Realisierung und den Betrieb derartiger Anwendungssysteme im größeren Stil überhaupt erst möglich. Wie erwähnt, werden WfMS derzeit hauptsächlich zur Unterstützung gut strukturierter, starrer Arbeitsabläufe, mit nur wenigen, vorhersehbaren Ausnahmefällen, verwendet. Der Grund dafür ist, daß die zugrundeliegenden WF-Metamodelle nicht ausdrucksstark und flexibel genug sind, um die betriebliche Abläufe mit ihren Ausnahmen und ihrer Variabilität angemessen abbilden zu können [JKMW97, KRD95, Nutt96]. Ein durch den Rechner erzwungenes, starres schematisches Vorgehen bei der täglichen Arbeit würde jedoch in vielen Anwendungsumgebungen – auch bei ansonsten idealer Prozeßunterstützung – nur auf geringe Akzeptanz stoßen [Nutt96, RHD98]. Ein fortschrittliches WfMS muß es deshalb Anwendern erlauben, bei Bedarf vom vormodellierten Ablauf dynamisch abzuweichen. Das bedeutet, daß die Bearbeitung einer WF-Instanz teilweise unterbrochen, die gewünschte Modifikation ihres Ausführungsgraphen, ihrer Attribute oder ihres Status vorgenommen und anschließend ihre Ausführung fortgeführt werden können müssen. Wichtige Aspekte dabei sind etwa:

- Flexibilität, insbesondere dynamische Anpaßbarkeit und Modifikation der Prozeßlogik, der Attribute und des Status laufender Workflows
- Unterstützung nur ungefähr beschreibbarer Workflows und dynamische Erweiterbarkeit von Workflows zur Laufzeit (unter Erhalt ihrer Konsistenz)
- Ausnahme- und Fehlerbehandlung (z.B. kontrollierter Abbruch und Zurücksetzen des WF)
- Berücksichtigung operativer Sicherheitsanforderungen (z.B. korrekte Terminierung laufender Workflows) bei der Modellierung und (dynamischen) Änderung von Workflows

Eine zentrale Forderung betrifft die Korrektheit und Konsistenz des WF [KaRa96, ReDa98]. So dürfen bereits abgeschlossene Prozeßschritte nicht mehr gelöscht oder ihre Attribute nachträglich verändert werden. Ebensowenig sollte das Überspringen eines Schrittes dazu führen, daß in der Folge Aktivitätenprogramme mit fehlenden Parameterdaten gerufen werden. Auch durch das WfMS selbst dürfen keine Fehler erzeugt werden, etwa durch die ungenügende Isolation der Anwender im Mehrbenutzerbetrieb. Hier muß sichergestellt werden, daß parallele Änderungen niemals zu Inkonsistenzen führen. Ein fortschrittliches WfMS muß ein *robustes* und *zuverlässiges* Ausführungsverhalten mit der Möglichkeit, zur Laufzeit *ad hoc* vom vormodellierten Workflow abweichen zu können, verbinden. Das bedeutet, daß nach der Durchführung von Laufzeitabweichungen für den WF dieselben Konsistenzeigenschaften erfüllt sein müssen, wie sie zur Modellierungszeit gefordert und durch geeignete Kon-

struktionsregeln und Analysen garantiert werden sollten. Um dem gerecht werden zu können, ist die Verwendung eines durchgängigen formalen Modells für die Definition, die Ausführung und die dynamische Änderung von Workflows unabdingbar – ein solches WfMS-Ablaufmodell, das den Anforderungen aus Theorie und Praxis gleichermaßen gerecht wird, gibt es derzeit nicht.

Mit zunehmender Komplexität des verwendeten WF-Metamodells gestalten sich dynamische WF-Änderungen immer schwieriger [ReDa98]. Um so wichtiger ist es, eine formale Basis bereitzustellen, die es einerseits erlaubt, Arbeitsabläufe möglichst natürlich und in einer für die Anwender *verständlichen* Form abzubilden und die andererseits *ausdrucksstark* genug ist, um die betrieblichen Prozesse mit ihren Ausnahmeregeln angemessen abbilden zu können. Weitere wichtige Anforderungen betreffen die *Ausführbarkeit* der Modelle und ihre *formale Validation*. Ersteres setzt voraus, daß die Ausführungszustände von WF-Instanzen und Informationen zu ihrem bisherigen Verlauf modellseitig abbildbar sind und daß es klare Regeln gibt, die festlegen, welche Zustandsübergänge beim Auftreten welcher Ereignisse stattfinden sollen. Wichtig ist, daß diese Regeln auch im Kontext dynamischer WF-Änderungen anwendbar bleiben, so daß nach ihrer Durchführung wieder konsistente WF-Zustände resultieren. Für die *formale Validation* von WF-Modellen sind Korrektheitsbedingungen notwendig, die sowohl bei der Modellierung als auch der Beschreibung dynamischer Änderungen als Bezugsbasis für formale Prüfungen dienen können. Dazu müssen (effizient überprüfbare) Regeln für die korrekte Strukturierung von WF-Modellen und für ein angemessenes dynamisches Verhalten vorliegen.

Um Anwender in die Lage zu versetzen, Änderungen an laufenden Workflows vornehmen und sie validieren zu können, ist es wichtig, daß das interne Ausführungsmodell semantisch nicht zu sehr vom „externen“ Modell abweicht. Dies wäre zum Beispiel nicht gegeben, wenn ein graphisches WF-Modell intern in eine ausführbare Regelmenge übersetzt wird. Eine solche Übersetzung würde nicht nur die Definition und Validation von Laufzeitabweichungen schwierig gestalten, sondern auch die Bereitstellung von Erklärungskomponenten erschweren (z.B. „Warum ist eine gewünschte Umformung nicht legal?“). Ein viel versprechender Ansatz ist die Bereitstellung geeigneter *Änderungsoperationen*, deren Anwendung die Korrektheits- und Konsistenzeigenschaften der Prozeßmodelle und -instanzen garantiert. Hierzu sind formale *Vor- und Nachbedingungen* für die Anwendung der einzelnen Operationen notwendig. Weitere wichtige Forderungen betreffen die *Vollständigkeit* der angebotenen Operatormenge und – soweit sinnvoll – ihre „Minimalität“. In Bezug auf die letztgenannte Forderungen ist allerdings im Einzelfall abzuwägen, ob ableitbare Operationen, die häufig benötigt werden, durch die kombinierte Anwendung von Basisoperationen oder – zur Steigerung der Performanz – ggf. gesondert implementiert werden sollen. – Im Zusammenhang mit Ablaufänderungen müssen schließlich noch Fragen der Autorisierung und Protokollierung geklärt sein [Kubi98].

Bei der Entwicklung eines WF-Metamodells stellt sich die Frage, ob als Grundlage ein existierender Beschreibungsformalismus verwendet werden soll oder nicht. Daß ein applikativer Ansatz nicht unbedingt vorteilhaft sein muß, wird am Beispiel von Petri-Netzen deutlich. Petri-Netze und ihre verschiedenen Erweiterungen [Ober96] werden von mehreren WfMS für die Modellierung und Steuerung von Workflows eingesetzt. Als entscheidende Vorteile werden ihre graphische Darstellbarkeit, ihre mathematische Fundierung und die Analysierbarkeit ihres dynamischen Verhaltens (z.B. auf Verklemmungsfreiheit) angegeben. Dennoch bieten Petri-Netze keine Konzepte im Hinblick auf die Unterstützung dynamischer WF- bzw. Netzänderungen. Deshalb müßte entweder versucht werden, alle denkbaren Ausnahmen bereits bei der Modellierung vorherzusehen oder das ursprüngliche Konzept muß um Möglichkeiten zur Verwirklichung dynamischer Änderungen erweitert werden. Ersteres scheitert oftmals am explosionsartigen Anwachsen der Komplexität der Modelle, wodurch die Vorteile der formalen Überprüfbarkeit nur noch theoretischer Natur sind. Auch die Erweiterung von Petri-Netz-Formalismen um Konzepte für dynamische Netzänderungen gestaltet sich äußerst schwierig und ist in der Praxis kaum anwendbar (vgl. hierzu Kapitel 10).

1.4 Ziel und Aufbau der Arbeit

Ziel dieser Arbeit ist es, fortschrittliche Konzepte für die Modellierung, die Ausführung und die dynamische Änderung von Workflows zu entwickeln. Neben wichtigen Fragestellungen, die sich in Verbindung mit der Unterstützung dieser Aspekte ergeben, soll auf grundlegende Funktionen flexibler WfMS und (teilweise) auf ihre algorithmische Umsetzung eingegangen werden. Im einzelnen ergeben sich die folgenden Problemstellungen, die im Verlauf der Arbeit sukzessive behandelt werden:

- Welche Beschreibungselemente muß ein Metamodell bereitstellen, um die Prozeßlogik (Kontrollfluß) und das Ausführungsverhalten komplexer WFs kompakt und präzise definieren zu können?
- Welche Konstruktionsregeln bzw. -prinzipien müssen dabei für die WF-Modellierung zugrunde gelegt werden, um zu korrekten und verständlichen WF-Modellen zu gelangen?
- Wie können WF-Zustände (inkl. Informationen zum bisherigen WF-Verlauf) und WF-Zustandsänderungen abgebildet werden? Welche Dynamikeigenschaften (z.B. Verklemmungsfreiheit) müssen für WF-Modelle gefordert werden, um ein korrektes Ausführungsverhalten zu erzielen. – Interessant ist, wann diese Eigenschaften bereits durch die Art der Konstruktion der WF-Modelle festliegen und wann zusätzliche Analysen durchzuführen sind.
- Wie können Datenflüsse zwischen WF-Aktivitäten (bzw. ihren Anwendungskomponenten) modelliert und zur Laufzeit gesteuert werden? – Hier bedarf es nicht nur zusätzlicher Beschreibungs- und Ausführungskonzepte, sondern auch formal prüfbarer Regeln, mit denen Inkonsistenzen (z.B. Lost Updates) und Fehler (z.B. fehlende Parameterdaten) ausgeschlossen werden können.
- Welche Erweiterungen müssen für das (interne) Ausführungsmodell vorgenommen werden, um planbare Ausnahmen (z.B. Scheitern einer Aktivität) und Aktionen zu ihrer Behandlung (z.B. partielles Zurücksetzen der Kontrolle) angemessen abzubilden? Wie sollen entsprechende Möglichkeiten den Anwendern zur Laufzeit angeboten werden?
- Wie lassen sich in einem adaptiven WfMS dynamische WF-Änderungen verwirklichen? – Hier sollen Änderungsoperationen (und ihre algorithmische Umsetzung) erarbeitet werden, deren Anwendung die für Prozeßmodelle und -instanzen geforderten Korrektheits- und Konsistenzeigenschaften garantiert. Dazu werden geeignete Vor- und Nachbedingungen festgelegt. Potentielle Probleme, die aus der Anwendung einzelner Operationen resultieren können, sollen effizient festgestellt und geeignete Strategien zu ihrer Behandlung entwickelt werden.
- Welche fortschrittlichen Änderungskonzepte (Änderungstransaktionen, Änderungen unterschiedlicher Gültigkeitsdauer, usw.) sind zur Unterstützung anspruchsvoller Anwendungen notwendig?
- Wie sind die behandelten Konzepte in einem adaptiven WfMS zu implementieren?

Bei diesen Punkten handelt es sich nicht nur um ein Problem der technischen Realisierung in einem WfMS, sondern es besteht noch grundsätzlicher Klärungsbedarf. Die genannten Aspekte bauen aufeinander auf und bilden deshalb eine sinnvolle Reihenfolge zur Bearbeitung der Gesamtfragestellung. Hieraus ergibt sich der Aufbau der Arbeit. Als Abschluß des Motivationsteils wird in Kapitel 2 dargestellt, warum ein WfMS dynamische WF-Änderungen unterstützen muß, um wirklich breit einsetzbar zu sein. Anschließend wird für existierende WfMS untersucht, ob sie die benötigte Flexibilität und Anpassungsfähigkeit bieten. Es wird gezeigt, daß das Spektrum heutiger WfMS sehr viel größer ist als bisher dargestellt. Die WfMS werden in verschiedene Kategorien unterteilt, und für jede Kategorie werden generelle Möglichkeiten und Grenzen diskutiert, bevor ausgewählte WfMS näher betrachtet werden. Dabei wird nicht nur auf Aspekte dynamischer Änderungen eingegangen, sondern es wird auch die generelle Eignung der Systeme im Hinblick auf die Unterstützung komplexer WFs bewertet.

Teil II dieser Arbeit (Kapitel 3 - Kapitel 6) befaßt sich mit dem *ADEPT-Basismodell*. Dieses Metamodell bildet die theoretische Grundlage für die Modellierung und Ausführung von WFs. Es ermöglicht die integrierte Beschreibung verschiedener WF-Aspekte und definiert formale Eigenschaften für ihre Korrektheit. Dadurch soll gewährleistet werden, daß zur Laufzeit keine Inkonsistenzen oder Fehlern auftreten. In dieser Arbeit erfolgt eine Konzentration auf Kontroll- und Datenflußaspekte. Dazu orthogonale WF-Aspekte, wie Zeit oder Ressourcen, bleiben ausgeklammert. Sie wurden in begleitenden Diplomarbeiten untersucht und bei der Implementierung des ADEPT-WfMS berücksichtigt.

Zu Beginn von Kapitel 3 werden die Entwurfsziele für das ADEPT-Basismodell diskutiert. Anschließend werden die Modellelemente, die ADEPT zur Beschreibung von *Kontrollflußstrukturen* bietet, vorgestellt. Es wird zunächst eine informelle Einführung gegeben, bevor formale Konstruktionsprinzipien und -regeln dargestellt werden. Zentral ist der strukturierte Modellierungsansatz, bei dem Sequenzen, Verzweigungen und Schleifen als ineineinander geschachtelte *Kontrollblöcke* mit jeweils genau einem Ein- und Ausgangsknoten modelliert werden. Hinzu kommen verschiedene Modellerweiterungen zur Erhöhung der Ausdrucksmächtigkeit. Beispielsweise können „Wartet-Auf“-Be-

ziehungen zwischen Aktivitäten paralleler Zweige sowie Fehlerrücksprünge abgebildet werden. Die Vorteile des gewählten Blockkonzepts werden im Verlauf der Arbeit sukzessive herausgearbeitet.

In Kapitel 4 wird das dynamische Verhalten von ADEPT-Prozeßmodellen näher beleuchtet. Auf Ausführungsebene wird für jede WF-Instanz, zumindest aus logischer Sicht, eine Kopie ihres Ursprungsgraphen verwaltet. Dieser Graph wird um *Knoten- und Kantenmarkierungen* angereichert, die jeweils den aktuellen WF-Zustand widerspiegeln. Die Zustandsmarkierungen bereits abgearbeiteter Bereiche des Ausführungsgraphen bleiben beim Fortschreiten der Kontrolle erhalten, zumindest solange sie nicht wiederholt durchlaufen werden. Dadurch können Informationen zum bisherigen Verlauf der Ausführung direkt aus den Markierungen abgeleitet werden. Für die WF-Steuerung werden *Ausführungs-* und *Markierungsregeln* definiert, die festlegen, unter welchen Graphmarkierungen eine Aktivität aktiviert werden darf bzw. welche Folgemarkierungen sich nach ihrer Ausführung ergeben. Diese Regeln bleiben im Kontext dynamischer WF-Änderungen weiter anwendbar, so daß nach der Durchführung von Laufzeitabweichungen wieder konsistente Prozeßzustände resultieren. Schließlich lassen sich präzise Aussagen zum *dynamischen Verhalten* von Prozeßmodellen treffen. Dazu wird gezeigt, daß – bei Zugrundelegen gewisser Fairneßannahmen – wichtige Dynamikeigenschaften in der Mehrzahl der Fälle bereits durch die Art der Konstruktion der WF-Modelle gewährleistet werden können.

Während in den Kapiteln 3 und 4 Kontrollflußaspekte behandelt werden, geht es in Kapitel 5 um die Modellierung und Steuerung von *Datenflüssen*. Zuerst wird gezeigt, wie Datenflüsse definiert werden und welche Freiheitsgrade dabei für den Modellierer bestehen. Im Anschluß daran werden – ähnlich wie bei der Definition von Kontrollflüssen – geeignete Regeln zur Strukturierung von Datenflußschemata angegeben. Sie sollen sicherstellen, daß es während der Ausführung nicht zu Inkonsistenzen (z.B. Lost Updates) oder Fehlern (z.B. Aufruf von Aktivitätenprogrammen mit fehlenden Eingabeparametern) kommt. Dies ist im Hinblick auf dynamische WF-Änderungen von großer Wichtigkeit, wird aber von heutigen WfMS kaum beachtet. Für die Überprüfung der Korrektheit modellierter Datenflüsse werden effiziente Analyseverfahren angegeben, die sich die Strukturierung der WF-Modelle konsequent zunutze machen. Abschließend werden operationale Aspekte, die sich im Zusammenhang mit der Steuerung von Datenflüssen und dem Management von WF-Daten ergeben, behandelt.

Kapitel 6 nimmt Erweiterungen vor, die nützlich sind, um *planbare Abweichungen* vom normalen Ablauf auf Modellebene abbilden und sie auf Ausführungsebene berücksichtigen zu können. Zuerst wird gezeigt, wie sich *Vorwärtssprünge* im Kontrollfluß vormodellieren lassen und wie Anwender zwischen präferierten und ausnahmebedingten Bearbeitungspfaden unterscheiden können. Anschließend werden Modellerweiterungen vorgenommen, um *Rückwärtssprünge* und das damit verbundene *partielle Zurücksetzen* der WF-Kontrolle in früherer WF-Zustände unterstützen zu können. In beiden Fällen wird skizziert, wie der Modellierer entsprechende Abweichungen auf semantisch hoher Ebene definieren kann und welche Einschränkungen getroffen werden müssen, um bei der Anwendung der Sprünge zur Laufzeit die Korrektheit und Konsistenz des WF-Ausführungsgraphen aufrechtzuerhalten.

Teil III dieser Arbeit (Kapitel 7 – Kapitel 9) beschäftigt sich mit theoretischen Grundlagen dynamischer Änderungen, ihrer algorithmischen Umsetzung und ihrer prototypischen Implementierung.

Kapitel 7 stellt mit $\text{ADEPT}_{\text{flex}}$ ein formales Grundgerüst für die Verwirklichung *dynamischer WF-Änderungen* vor, das auf dem in Teil II entwickelten ADEPT-Basismodell fußt. Dabei werden (dynamische) Änderungen auf eine Menge von *Basisänderungsoperationen* mit wohldefinierter Semantik zurückgeführt. Zuerst werden die Ziele für den Entwurf dieser Operationen diskutiert. Insbesondere soll ihre Anwendung die vom ADEPT-Basismodell geforderten Konsistenzeigenschaften garantieren. Weitere wichtige Realisierungsaspekte betreffen die *Vollständigkeit* der angebotenen Operatormenge und – soweit sinnvoll – ihre *Minimalität*. Unterstützt werden elementare Basisoperationen, wie das Hinzufügen, das Löschen oder das Verschieben einzelner Aktivitäten oder ganzer Kontrollblöcke, aber auch abgeleitete, semantisch höherwertige Operationen, etwa für die Umsetzung von Ad-hoc-Sprüngen. Für jede Änderungsoperation werden formale *Vor-* und *Nachbedingungen* angegeben. Es wird gezeigt, daß bei Einhaltung dieser Bedingungen die geforderten Korrektheitseigenschaften auch nach der Abänderung des WF gelten. Beispielsweise sorgt $\text{ADEPT}_{\text{flex}}$ dafür, daß nach Anwendung einer dynamischen Änderung wieder ein WF mit konsistentem Zustand resultiert. Ein wesentliches Kriterium hierbei ist, daß der Ausführungsgraph nach Durchführung der Änderung keine Zustandsmarkierungen aufweist, die nicht auch bei „normaler“ Ausführung des aus der Änderung resultierenden WF-Schemas erreichbar sind. Um dieser Forderung nachzukommen, wird die Anwendbarkeit von

Änderungsoperationen durch den aktuellen Zustand des WF (d.h. seinen Knoten- und Kantenmarkierungen) eingeschränkt. Entsprechende Überprüfungen werden *vor* der Anwendung einer Änderungsoperation durchgeführt. Darüber hinaus wird *nach* der strukturellen Abänderung eines Ausführungsgraphen sein Status neu bewertet und ggf. angepaßt, so daß wieder ein konsistenter Zustand resultiert.

In Kapitel 7 werden nicht nur formale Bedingungen für die Änderungsoperationen angegeben, sondern auch potentielle Probleme, die aus ihrer Anwendung resultieren können (z.B. fehlende Daten nach Löschen einer Aktivität), aufgezeigt. Des weiteren werden Strategien zur Behandlung dieser Probleme diskutiert. Können etwa infolge des Löschens einer Aktivität die Eingabeparameter nachfolgender Schritte nicht versorgt werden, muß diese Operation entweder unterbunden werden oder es sind begleitende Maßnahmen (z.B. kaskadierendes Löschen datenabhängiger Aktivitäten, Installation von „Datennachforderungsdiensten“) anzuwenden. Es wird gezeigt, daß sich die hierzu notwendigen Analysen die Eigenschaften der WF-Graphen (z.B. Blockstruktur, Graphmarkierungen) konsequent zunutze machen können und daß die Art und der Umfang der Überprüfungen in Abhängigkeit von der jeweiligen Änderungsoperation eingegrenzt werden kann. Die interne Umsetzung von Änderungsoperation(en) erfolgt durch Anwendung einer Menge von Graphtransformations- und Graphvereinfachungsregeln. Zu erwähnen bleibt, daß alle im Kontext einer Änderung stattfindenden Modifikationen innerhalb derselben Änderungstransaktion auf Ausführungsgraphen angewendet werden.

Das Kapitel 8 behandelt fortschrittliche Fragestellungen adaptiver WF-Modelle, die sich ergeben, wenn dynamische WF-Änderungen in Verbindung mit Schleifen- und Fehlerrücksprüngen unterstützt werden sollen. Insbesondere kann es hier sinnvoll sein, Änderungen auf Instanzebene bei Bedarf nur temporär einzubringen und sie später, etwa nach einem Schleifenrücksprung, wieder automatisch zu entfernen. Alternativ dazu kann eine auf einen Ausführungsgraphen angewandte Änderung permanent, d.h. für die Restdauer des WF, erhalten bleiben. Die Unterstützung dieser beiden Änderungsarten in Kombination miteinander ist nicht trivial. Beispielsweise muß bei der Einbringung permanenter Änderungen sichergestellt werden, daß sie nicht in Abhängigkeit von vorangehenden temporären Modifikationen vorgenommen werden. Wird dies nicht ausgeschlossen, können sich bei der späteren Rücknahme einer temporären WF-Änderung Inkonsistenzen oder Fehler ergeben.

In Kapitel 9 wird das ADEPT-WfMS vorgestellt, das seit 1997 an der Universität Ulm entwickelt wird. Dabei handelt es sich um den Prototypen eines flexiblen Prozeßunterstützungssystems für die Realisierung unternehmensweiter, vorgangsorientierter Anwendungssysteme. Das ADEPT-WfMS setzt die eingangs beschriebene Trennung von Prozeßlogik und Anwendungscode konsequent um. Durch die explizite, formale Beschreibung des Kontroll- und Datenflusses kann das zukünftige Systemverhalten evaluiert werden, noch bevor eine Zeile Programmcode geschrieben wird. Darüber hinaus kann zur Laufzeit die Prozeßlogik geändert werden, ohne daß Anwendungskomponenten hiervon betroffen sind. Umgekehrt können diese Komponenten unabhängig vom Kontext, in dem sie später verwendet werden, implementiert werden. Im ADEPT-WfMS werden zahlreiche der in dieser Arbeit behandelten Konzepte prototypisch umgesetzt. Kapitel 9 gibt einen Einblick in die Architektur dieses adaptiven WfMS und geht auf ausgewählte Implementierungsaspekte ein. Des weiteren wird gezeigt, wie dynamische Änderungen durchgeführt und welche Phasen dabei durchlaufen werden. Abschließend wird auf Aspekte der Anwendungsentwicklung eingegangen. Dazu werden Entwicklungswerkzeuge und die angebotenen Schnittstellen für die Anwendungsprogrammierung skizziert.

Teil IV (Kapitel 10 - Kapitel 11) beinhaltet die Zusammenfassung der Arbeit und ordnet sie in die bestehende Literatur ein. In Kapitel 10 beschreiben wir existierende WF-Metamodelle, die sich mit adaptivem Workflow-Management beschäftigen, und vergleichen sie mit dem ADEPT-Ansatz. Neben applikativen Ansätzen, die auf existierenden Formalismen (z.B. Petri-Netze oder Statecharts) basieren, gehen wir hier auf Vorschläge aus verschiedenen Bereichen der Informatik ein, etwa KI-Planungssysteme, Graphgrammatiken oder Prozeßalgebren, um nur einige Beispiele zu nennen. Kapitel 11 faßt die Hauptergebnisse der Arbeit zusammen und gibt einen Ausblick auf weiterführende Themen.

2 Adaptivität in existierenden Workflow-Management-Systemen

In diesem Kapitel zeigen wir, warum ein WfMS dynamische WF-Änderungen im laufenden Betrieb unterstützen können muß, um wirklich breit einsetzbar zu sein. Anschließend wird für existierende WfMS untersucht, ob sie die benötigte Flexibilität und Anpassungsfähigkeit bieten. Dazu werden in Abschnitt 2.1 unterschiedliche Szenarien für Ausnahmen und dynamische WF-Änderungen entwikkelt. In Abschnitt 2.2 wird gezeigt, daß das Spektrum heutiger WfMS größer ist als in Kapitel 1 dargestellt. Zu diesem Zweck werden die Systeme in verschiedene Kategorien unterteilt und ihre Möglichkeiten und Grenzen diskutiert. Der Abschnitt 2.3 behandelt ausgewählte WfMS sowie deren Stärken und Schwächen. Dabei wird nicht nur auf Aspekte dynamischer WF-Änderungen eingegangen, sondern es wird auch die generelle Eignung der Systeme im Hinblick auf die Realisierung prozeßorientierter Anwendungssysteme eingegangen. Abschnitt 2.4 skizziert aktuelle Standards im WF-Bereich.

2.1 Dynamische Workflow-Änderungen und ihre Ursachen

Unter *Prozeßadaption* fassen wir alle Maßnahmen zusammen, mit denen ein prozeßorientiertes Anwendungssystem an geänderte Realweltsituationen angepaßt werden kann [RHD98, Sieb97, HaSh98]. Hierunter fallen Änderungen an Prozeßvorlagen und –instanzen, ebenso wie Adaptionen des Organisationsmodells oder der verwendeten Anwendungsbausteine. Im allgemeinen muß eine Prozeßadaption auch *dynamisch*, d.h. während der Ausführung von Prozeßinstanzen, erfolgen können. Für diesen Fall muß ggf. die Bearbeitung von WF-Instanzen (teilweise) unterbrochen, die gewünschte Modifikation vorgenommen und anschließend mit der Ausführung fortgefahren werden. Ein WfMS, das solche dynamischen Änderungen zuläßt, bezeichnen wir als *adaptiv*. Im folgenden konzentrieren wir uns auf dynamische WF-Änderungen, d.h. Modifikationen des WF-Graphen, einzelner Attribute oder des Status von sich in Ausführung befindlichen Workflows. Für die Unterstützung solcher dynamischen *WF-Änderungen* ist es wichtig, die Ursachen und Folgen dieser Laufzeitanpassungen besser zu verstehen.

2.1.1 Planbarkeit von Prozessen

Ein wichtiges Unterscheidungsmerkmal ist die *Planbarkeit* von Prozessen [DHLS96]. Sie determiniert, welche der für eine rechnergestützte Ausführung benötigten Informationen bereits zum Zeitpunkt der Modellierung bekannt sind bzw. wie noch fehlende Informationen zur Laufzeit ermittelt und berücksichtigt werden sollen. Prinzipiell können WF-Eigenschaften bei der Modellierung von Prozeßvorlagen, bei ihrer Instanziierung oder dynamisch, während der Ausführung einzelner WF-Instanzen, festgelegt werden. Im folgenden unterscheiden wir zwischen vollständig planbaren, teilweise planbaren und nicht planbaren Prozessen (vgl. Abbildung 2-1). Die Übergänge zwischen den Kategorien sind fließend, zumal sich für Prozesse die Planbarkeit ihrer verschiedenen Merkmale differenziert darstellen kann [GHS95, KiUn94, NaOt96, PiRo95, Rein93, SAM91].

2.1.1.1 Vollständig planbare Prozesse

Bei vollständig planbaren Prozessen stehen alle für die WF-Ausführung benötigten Informationen bereits zur Modellierungszeit fest. Die Prozesse umfassen eine klar definierte Menge an Aufgaben, deren Bearbeitungsreihenfolgen und -bedingungen, zumindest für den Normalfall, fest vorgegeben sind. Ebenso stehen die Datenflußbeziehungen zwischen den Prozeßschritten a priori fest. Planbarkeit bedeutet aber nicht notwendigerweise, daß zur Modellierungszeit alle Prozeßmerkmale eindeutig determiniert werden können. Für Eigenschaften wie „Bearbeiter einer Aktivität“ macht eine statische Festlegung im allgemeinen wenig Sinn. Zwar ist hier die Grundmenge der für die Bearbeitung in Frage kommenden Personen meist bekannt, welcher Akteur die Aktivität konkret bearbeiten soll, variiert aber von Fall zu Fall. Ähnliches gilt für andere Aktivitätenattribute. Beispielsweise kann es sinnvoll sein, der Aktivität a priori kein festes Programm zuzuordnen, sondern es dynamisch beim Start zu binden [Han97]. Die statische Festlegung solcher Prozeßmerkmale würde zu inflexiblen, häufig abzuändernden WF-Modellen führen. In der Regel kann ihre Bestimmung automatisch durch das WfMS zur Laufzeit erfolgen, vorausgesetzt aus der Beschreibung des Workflows geht hervor, wodurch die Grundmenge möglicher Merkmalsausprägungen festgelegt ist (z.B. durch Angabe einer Rolle). Zum Einsatz kommen einfache Mechanismen wie *dynamische Rollenauflösung* [Kubi98] oder *spätes Bin-*

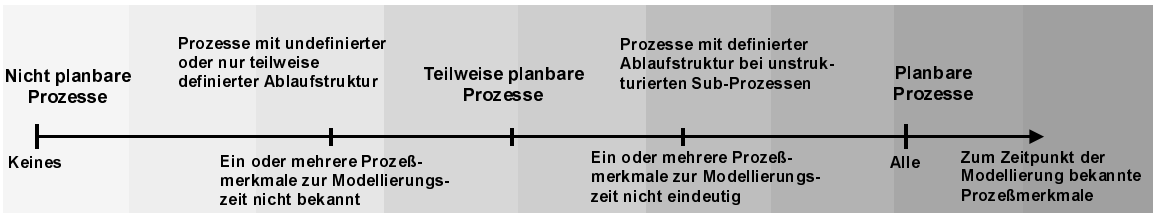


Abbildung 2-1: *Grobeinteilung von Prozessen nach dem Grad ihrer Planbarkeit*

den von Anwendungskomponenten [Han97, HHSW96, KwBa97]. Beispiele für planbare Prozesse sind die Schadensfallbearbeitung bei einer Versicherung oder die Bearbeitung eines Kreditantrags.

2.1.1.2 Teilweise planbare Prozesse

Für viele Workflows können die zu ihrer Ausführung benötigten Informationen, z.B. die Menge durchzuführender Aktivitäten und deren Reihenfolgebeziehungen, nicht vollständig a priori festgelegt werden, d.h. bei der Modellierung stehen ein oder mehrere Prozeßmerkmale noch nicht fest. In bestimmten Fällen ist es auch nicht erstrebenswert, WF-Modelle bis ins letzte Detail vorzugeben, etwa wenn Freiheitsgrade hinsichtlich der konkreten Prozeßgestaltung bestehen sollen. Die Unterstützung solcher teilweise planbaren Prozesse erfordert geeignete Anpassungsmöglichkeiten auf Instanzebene. In zahlreichen Anwendungsbereichen, wie der Produktentwicklung oder der Medizin, hat man es darüber hinaus häufig mit Prozessen zu tun, deren genaue Struktur sich erst im Verlauf der Ausführung ergibt [HJKW96, RHD98]. Je nach Art und Umfang der „offenen“ Modellaspekte lassen sich semistrukturierte Prozesse unterschiedlichen Subkategorien zuordnen [DHLS96, NaOt96]. Der interessanteste und zugleich schwierigste Fall ist gegeben, wenn sich die genaue Prozeßstruktur, d.h. die Menge der durchzuführenden Prozeßschritte sowie die zwischen ihnen bestehenden Kontroll- und Datenabhängigkeiten, erst im Verlauf der Prozeßdurchführung ergibt. Zur Abbildung solcher a priori nicht vollständig determinierbarer Prozesse muß das WfMS Konzepte für *evolutionäre* Workflows unterstützen, bei denen sich die Planung und Ausführung überlappen kann.

Beispiel 2-1 (Produktentwicklungsprozesse)

Bei Produktentwicklungsprozessen [Band95, BeDa96, BFG93, HJKW96, JaCo93, RiMi97] kann die Art und der Umfang der durchzuführenden Tätigkeiten von Zwischenergebnissen vorangehender Schritte (z.B. Produktentwurf) abhängig sein. Die Gesamtmenge der durchzuführenden Teilaufgaben und die genaue Prozeßstruktur stehen deshalb zu Beginn oftmals nur grob fest und entwickeln sich dynamisch während der Prozeßausführung.

Beispiel 2-2 (Behandlungsprozesse im Krankenhaus)

Der Behandlungsprozeß eines Patienten im Krankenhaus begleitet seinen Weg von der Einbestellung über die Aufnahme, Diagnostik und Therapie bis zur Entlassung [RHD98]. Üblicherweise kann ein solcher Prozeß Dutzende bis Hunderte von Arbeitsschritten umfassen und sich über eine lange Dauer (mehrere Wochen oder Monate) erstrecken. Selten findet man zwei identische Ausprägungen von Behandlungsprozessen vor. Hinzu kommt, daß die Faktoren, die die konkrete Prozeßgestaltung bestimmen, außerhalb der Kontrolle des WfMS liegen. Beispielsweise kann die Gesamtmenge der durchzuführenden medizinischen Untersuchungen und ihre Reihenfolge von Entscheidungen des Arztes, dem Zustand des Patienten, Ergebnissen vorangegangener Untersuchungen oder medizinischen Indikationen abhängig sein. Der Gesamtprozeß ergibt sich hier durch die dynamische Komposition von einfachen, gut strukturierbaren Prozessen (z.B. Abwicklung einzelner Untersuchungen). Diese Teilprozesse können in der Regel weder strikt sequentiell noch parallel durchgeführt werden. Vielmehr wird eine überlappende Ausführung, verbunden mit der Synchronisation einzelner Arbeitsschritte, erforderlich.

Aus der schrittweisen Evolution der Prozeßstruktur ergeben sich für adaptive WfMS weitergehende Anforderungen als im Fall vollständig planbarer Prozesse, bei denen diese Prozeßmerkmale – zumindest für den Normalfall – bereits zur Modellierungszeit festliegen. Die WF-Vorlage eines teilweise planbaren Prozesses bildet dagegen nur einen Teil des späteren Prozesses ab, legt diesen also nur grob fest. Auf Instanzebene müssen dem entsprechend Erweiterungen vorgenommen werden. In der Mehrzahl der Fälle müssen diese Adaptionen von den Akteuren manuell vorgenommen werden. Ein Sonderfall ist gegeben, wenn ein (planbarer) Prozeß einen Sub-Prozeß mit undefinierter Struktur besitzt, diese aber vor der Instanziierung des Sub-WF noch festgelegt werden kann. In einem solchen Fall können einfache Laufzeitmechanismen angewendet werden, wie das *späte Modellieren* von (Sub-)WFs (*late modeling*) [Goes97], die für viele Bereiche aber nicht ausreichend sind [DeMa96,

HJKW96, JaCo93, RiMi97]. Vielmehr muß es auch möglich sein, die Struktur eines nur teilweise vordefinierten WF-Graphen zur Ausführungszeit auf konsistente Art und Weise anzupassen, etwa durch die Hinzunahme oder das Löschen von Aktivitäten (*dynamische WF-Änderungen*).

2.1.1.3 Nicht planbare Prozesse

Viele fallbezogene Abläufe ergeben sich spontan, zeichnen sich durch eine schrittweise Evolution ihrer Prozeßstruktur aus und besitzen einmaligen Charakter. Im Extremfall ist keines der Prozeßmerkmale, die für die Abbildung in ein ausführbares WF-Modell benötigt werden, a priori bekannt bzw. planbar. Um wichtige Vorteile WfMS-basierter Anwendungssysteme (z.B. Arbeitslistenverwaltung und Prozeßdokumentation) dennoch nutzen zu können, sollte es das WfMS erlauben, die Struktur und die Eigenschaften eines WF ggf. auch erst im Verlauf seiner Ausführung zu definieren [HSS96 NaOt96, PiRo95]. Die Grundlage solcher *Ad-hoc-Workflows* bildet meist eine einfache WF-Vorlage mit einer informativen, oftmals rein textuellen Beschreibung des Ablaufs. Sie kann von Anwendern zur Laufzeit um neue Arbeitsschritte erweitert werden, wozu hinreichende Kenntnisse der Ablaufziele erforderlich sind [BhCr90, CrMa95, ElWa94]. In existierenden Implementierungen (z.B. [Karb94]) beschränken sich die angebotenen Möglichkeiten auf die Festlegung des jeweils nächsten Arbeitsschrittes und seines Bearbeiters, so daß sich lediglich sequentielle Abläufe abbilden lassen. Dies ist für viele Anwendungen nicht ausreichend [Meye96]. Vielmehr sollten bei der Ad-hoc-Definition eines WF dieselben Beschreibungsmöglichkeiten bestehen wie bei der Modellierung einer WF-Vorlage.

Die Ad-hoc-Definition eines Ablaufs verursacht einen hohen Kommunikationsaufwand zwischen den Akteuren [DHLS96]. Zu seiner Bewältigung wird vorgeschlagen, Workflow- mit Groupware-Technologie zu integrieren [AGSC95, BlNu95, Webe97]. Groupware-Systeme [Grei88, Grud94] unterstützen die Bearbeitung einer Aufgabe durch verschiedene Personen, wobei als Grundlage eine gemeinsame Informationsbasis dient [Gros97, Wulf97]. Je nach Systemkategorie werden synchrone oder asynchrone Bearbeitungsformen unterstützt, wobei die Kontrolle überwiegend von Benutzern ausgeht. Typisch ist auch die integrierte Nutzung verschiedener Medien. Der Einsatz von Groupware-Systemen bietet im Hinblick auf die Unterstützung unstrukturierter Gruppentätigkeiten (z.B. medizinisches Konzil einer Gruppe von Ärzten), für die auch zur Ausführungszeit keine klare Ablaufstruktur definierbar ist, interessante Perspektiven. Da solche Tätigkeiten oftmals im Kontext planbarer Prozesse stattfinden, wird von verschiedener Seite daran gearbeitet, Groupware-Funktionen an definierten Punkten eines WF zu integrieren [BlNu95, ShKo97, Webe97]. Die Bearbeitung einzelner Prozeßschritte besteht dann nicht aus dem Aufruf eines „normalen“ Aktivitätenprogramms, sondern aus der Durchführung einer Gruppentätigkeit, unterstützt durch ein Groupware-basiertes Anwendungssystem. Herausforderungen sind u.a. die Integration von Groupware-Systemen und WfMS [Webe97, Schn96], die Bereitstellung adäquater Kontextinformationen für die an der Bearbeitung der Gruppentätigkeit beteiligten Personen [BlNu95] sowie der Transfer von Informationen zwischen WfMS und Groupware-System [ShKo97].

2.1.1.4 Abschließende Bemerkungen

Für die rechnergestützte Ausführung von nicht oder nur teilweise planbaren Prozessen ist die Bereitstellung von Systemfunktionen zur dynamischen Festlegung von WF-Eigenschaften unerläßlich. Dies darf jedoch nicht zu dem Trugschluß führen, daß dynamische WF-Änderungen für die Unterstützung planbarer Prozesse keine Rolle spielen. Auch hier können situationsbedingt Abweichungen vom geplanten Ablauf notwendig werden [RHD98, Such87]. Hierauf gehen wir im folgenden Abschnitt näher ein. Eine strikte Trennung in verschiedene Prozeßkategorien ist für die Praxis ohnehin nicht möglich. So gibt es planbare WFs mit nur ungefähr beschreibbaren Sub-WFs, deren genauer Ablauf sich erst zur Laufzeit ergibt [HHJS97]. Umgekehrt können Prozesse, die nur teilweise planbar sind, Sub-WFs mit vordefinierter Struktur besitzen. Tabelle 2-1 faßt die Szenarien zusammen.

2.1.2 Ausnahmen in prozeßorientierten Anwendungssystemen und ihre Behandlung

Für Systeme, die in eine reale Umgebung eingebettet sind und mit denen Personen dieser Umgebung interagieren, können Divergenzen zwischen Realwelt und Modell niemals vollständig ausgeschlossen werden [Borg85, Dona89, ObSt91, Saas95]. Dies gilt in besonderem Maß für WF-basierte Anwendungssysteme, bei denen Abweichungen zwischen dem realen Arbeitsablauf und dem computer-

Planbarkeit	Mögliche Unterstützungsformen
1. Prozeßstruktur ist zur Modellierungszeit vollständig bekannt (*planbarer Prozeß*)	Erstellung einer mehrfach instantiierbaren WF-Vorlage und Steuerung der WF-Instanzen gemäß der durch diese Vorlage definierten Prozeßstruktur
2. Prozeßstruktur ist zur Modellierungszeit nicht bekannt (*nicht planbarer Prozeß*)	Definition einer wiederverwendbaren WF-Vorlage ist nicht möglich bzw. sinnvoll
a) Die genaue Prozeßstruktur ergibt sich während der Ausführung.	Ad-hoc-Definition des WF durch prozeßbeteiligte Akteure
b) Eine genaue Prozeßstruktur kann auch zur Ausführungszeit nicht festgelegt werden.	Einsatz Groupware-basierter Anwendungssysteme
3. Prozeßstruktur ist zur Modellierungszeit teilweise bekannt (*teilweise planbarer Prozeß*)	Beschreibung der planbaren Prozeßanteile in einer Prozeßvorlage, die den WF nur „ungefähr“ festlegt – bei der Ausführung einzelner WF-Instanzen müssen dynamische Erweiterungen vorgenommen werden können.
a) Die (noch) nicht definierbaren Prozeßanteile sind zur Modellierzeit bekannt.	
Beispiel: Ein strukturierter Prozeß besitzt einen Sub-Prozeß mit (noch) nicht bekannter Struktur. Die Bedingungen für die Ausführung des Sub-Prozesses stehen aber fest.	Kennzeichnung des entsprechenden Aktivitätenknotens im WF-Graphen
– Grundmenge möglicher Ausprägungen des Sub-Prozesses liegt a priori fest	Definition von Sub-WF-Vorlagen und *spätes Binden* während der WF-Ausführung
– Struktur des Sub-Prozesses kann vor seinem Start festgelegt werden	*Spätes Modellieren* des Sub-WFs (bzw. seiner Vorlage) vor seiner Instanziierung
– Struktur des Sub-Prozesses ergibt sich erst während seiner Ausführung	Ad-hoc-Definition des Sub-Workflows während seiner Ausführung (vgl. Fall 2a)
– Struktur des Sub-Prozesses kann auch zur Laufzeit nicht festgelegt werden	Unterstützung durch Groupware-basiertes Anwendungssystem (vgl. Fall 2b)
b) Die noch nicht definierten Prozeßanteile sind (noch) nicht bekannt.	Dynamische Abänderung bzw. Erweiterung des Workflows während seiner Ausführung

Tabelle 2-1: *Planbarkeit von Prozessen und mögliche Unterstützungformen*

unterstütztem WF häufig auftreten [AuLe88, Hein98a, RHD98, StMi95]. Während der WF-Ausführung können (Ausnahme-)Situationen eintreten, in denen eine WF-Instanz nicht mehr wie geplant durchführbar ist. Für die Einordnung und Behandlung von Ausnahmen ist es wichtig, die Entstehung und die Folgen dieser außerplanmäßigen Ereignisse besser zu verstehen.

2.1.2.1 Grundlegende Begriffe

In bezug auf WfMS wird der Ausnahmebegriff weit gefaßt. *Ausnahmen* sind Ereignisse, bei deren Auftreten der weitere Ablauf einer WF-Instanz nicht mehr wie geplant, d.h. wie im zugehörigen WF-Modell hinterlegt, durchführbar ist. Aus diesem Grund wird eine gesonderte Behandlung durch das WfMS, prozeßbeteiligte Akteure oder andere Personen (z.B. den Prozeß-/Systemadministrator) erforderlich. Gemäß dieser Definition decken Ausnahmen Fälle ab, wie die spontane Abweichung der Prozeßteilnehmer vom geplanten Ablauf, externe Ereignisse, infolge derer ein Prozeß (bzw. eine einzelne Tätigkeit) nicht mehr erfolgreich beendet werden kann, Fehler bei der Bearbeitung von Prozeßschritten, Nichtverfügbarkeit von Ressourcen oder Fehler in Software-/Hardware-Komponenten. Ausnahmen sind demnach häufig auftretende Ereignisse [SaWh95, StMi95], die nicht immer vorhersehbar bzw. planbar sind. Im folgenden unterscheiden wir zwischen *Fehlern* und *Abweichungen*.

Durch fehlerhafte Systemkomponenten können Verarbeitungsfehler oder Programmabstürze resultieren. Solche Ausnahmen bezeichnen wir als *Fehler*. Werden sie durch das WfMS erzeugt, liegt ein *interner* Fehler vor, der auf WfMS-Ebene behandelt werden muß. Dagegen werden *externe* Fehler durch Komponenten verursacht, die für die WF-Ausführung zwar notwendig, die aber nicht Bestandteil des WfMS sind [ScBi96]. Beispiele sind Hardware-Ausfälle oder Kommunikationsstörungen.

Solche Fehler können nicht immer transparent für das WfMS behandelt werden [Wäch96, WäRe92]. Nach Hardware- oder Betriebssystemausfällen ist ein Wiederanlauf erforderlich, wobei wieder ein konsistenter Systemzustand hergestellt werden muß. Befand sich das WfMS beim Auftreten des Fehlers im Übergang zwischen zwei Zuständen, muß entweder der alte korrekte Zustand wiederhergestellt (*Backward Recovery*) oder ein neuer korrekter Zustand erzeugt (*Forward Recovery*) werden [ScBi96, Wäch96]. Notwendige Voraussetzungen hierfür sind die persistente Speicherung von WF-Daten und die Synchronisation des internen Status eines WF bzw. einer Aktivität mit dem Status aufgerufener Anwendungskomponenten (z.B. mittels Zwei-Phasen-Commit-Protokoll [Dada96]).

Eine andere Art von Ausnahmen bilden *Abweichungen* vom implementierten WF zur Laufzeit. Sie können dynamische Änderungen der WF-Struktur (z.B. Einfügen / Löschen von Aktivitäten), der WF-Attribute (z.B. Bearbeiterrollen, Ressourcen) oder des WF-Zustands (z.B. partielles Zurücksetzen des WF) erforderlich machen. Abweichungen sind auf einer semantisch höheren Ebene als Fehler angesiedelt und können unterschiedliche Ursachen haben, wie das Eintreten unvorhersehbarer Situationen bei der Durchführung eines Arbeitsablaufs, das Überschreiten vorgegebener Fristen oder das Scheitern einer Aufgabenbearbeitung. In solchen Fällen muß für Akteure die Möglichkeit bestehen, Arbeitsabläufe *flexibel* umzugestalten, was Anpassungen der sie unterstützenden Workflows nach sich ziehen kann. Die Art einer Ausnahme determiniert dabei die verschiedenen Möglichkeiten ihrer Behandlung. Wichtige Parameter sind die *Ursache*, die *Vorhersehbarkeit* und der *Kontext* der Ausnahme.

2.1.2.2 Abweichungen von vormodellierten Workflows und ihre Ursachen

Daß ein WF nicht (mehr) wie geplant durchführbar ist, kann unterschiedliche Ursachen haben. Während der Bearbeitung eines Prozesses können Sonderfälle eintreten, die bei der Modellierung nicht berücksichtigt wurden. Des weiteren können in der Realwelt Ausnahmesituationen, Verarbeitungsfehler oder Zustandsänderungen auftreten, die Prozeßteilnehmer dazu zwingen, vom ansonsten üblichen Ablauf abzuweichen. Um die Entstehung und die Folgen solcher außerplanmäßigen Ereignisse besser zu verstehen, wurden im Projekt „Workflow-Management in klinischen Anwendungsumgebungen“ [RDMK00, RKD96, RSD97, RHD98] zahlreiche Ausnahmeszenarien, die bei der Durchführung realer Krankenhausabläufe auftreten können, systematisch untersucht [Meye96]. Für jeden betrachteten Ausnahmefall wurden Aspekte wie Ursache, Vorhersehbarkeit, Vermeidbarkeit, Auswirkungen auf den Prozeßverlauf und erforderliche Aktionen zur Ausnahmebehandlung eingehend beleuchtet.

Beispiel 2-3 (Abweichungen vom geplanten Standardablauf)

Ein einfacher Untersuchungsablauf im Krankenhaus umfaßt Arbeitsschritte wie „Untersuchung anordnen“, „Termin vereinbaren“, „Patient aufklären“, „Patient vorbereiten“, „Untersuchung durchführen“, „Befund schreiben“ und „Befund validieren“. Bereits für diesen einfachen Prozeß können sich Situationen ergeben, die Abweichungen vom geplanten Ablauf erforderlich machen. Beispielsweise muß in einem Notfall ein Patient auch ohne die ansonsten übliche Terminvereinbarung oder Vorbereitungen einer Untersuchungsstelle zugeführt werden können, d.h. es kann situationsbedingt notwendig werden, einzelne oder mehrere Arbeitsschritte auszulassen. Umgekehrt kann es in Einzelfällen wichtig sein, nicht eingeplante Tätigkeiten, z.B. eine zusätzliche Vorbereitung zu einer Untersuchung, ablaufbezogen durchzuführen (und entsprechend zu dokumentieren). Terminverschiebungen, Untersuchungsabsagen oder Fehler bei der Durchführung einzelner Tätigkeiten (z.B. unterlassene Vorbereitungen, Verlust einer Blutprobe, Entnahme des falschen Probematerials) können ebenfalls zu Abweichungen vom Standardablauf führen, etwa der Wiederholung von Tätigkeiten. Bei einer Untersuchungsabsage muß der Prozeß abgebrochen und bereits vereinbarte Termine müssen storniert werden.

In solchen Situationen kann der Fall eintreten, daß die vordefinierte, dem Rechner bekannte WF-Beschreibung den realen Arbeitsablauf nicht mehr angemessen abbildet. Divergenzen können hinsichtlich der Prozeßstruktur, dem Verarbeitungsstand und einzelner Prozeßmerkmale bestehen. Konkret können Änderungen des realen Arbeitsablaufs dazu führen, daß

- den Akteuren in ihren Arbeitslisten Aktivitäten zur Bearbeitung angeboten werden, deren Durchführung aufgrund der erfolgten Abweichung nicht mehr möglich bzw. sinnvoll ist.
- auf der Ebene des realen Ablaufs Tätigkeiten ausgeführt werden, die dem WfMS nicht bekannt sind und deshalb von ihm weder koordiniert noch dokumentiert werden.

Solche Diskrepanzen sind meist nicht tolerierbar. Sie erzwingen entweder die dynamische Anpassung der elektronischen Ablaufbeschreibung an die neuen Gegebenheiten der Realwelt oder aber die umge-

kehrte Vorgehensweise. Letztere ist für die Praxis oftmals nicht akzeptabel. Im Krankenhausbereich etwa, mit dem „Unsicherheitsfaktor“ Patient, bilden Variationen im Verlauf eines Prozesses den Normalfall und sind Ausdruck der inhärenten Variabilität der Prozesse [Dada95, DRK00]. Das unvorhersehbare Ereignis stellt in gewisser Weise ein normales Phänomen dar, auf das Mitarbeiter flexibel reagieren können müssen und auch in der Lage sind, dies zu tun. Ein WF-basiertes Anwendungssystem muß es deshalb berechtigten Akteuren in Ausnahmesituationen erlauben, in flexibler Form von dem im WfMS „implementierten“ WF abzuweichen. Weitere Gründe für Abweichungen können sein:

- Während der gesamten Bearbeitung des realen Arbeitsablaufs können unvorhersehbare Ereignisse eintreten (z.B. Tod eines Patienten, medizinischer Notfall).
- Die Annahmen, unter denen ein WF initiiert wurde, treffen ggf. nicht mehr zu. Verweigert ein Patient die Einwilligung zu einer Untersuchung (vgl. Beispiel 2-3), so sind die Voraussetzungen für die Fortsetzung des Ablaufs nicht mehr gegeben. In einem solchen Fall müssen der Prozeß abgebrochen und bereits festgelegte Termine storniert werden. Dies erfordert die systemseitige Unterstützung für den kontrollierten Abbruch und das semantische Zurücksetzen von WFs.
- Bei der Durchführung eines einzelnen Arbeitsschrittes können Situationen eintreten, in denen der Schritt nicht mehr erfolgreich bearbeitbar ist (z.B. Abbruch einer Untersuchungsdurchführung bei Verschlechterung des Gesundheitszustandes eines Patienten). Darüber hinaus können Verarbeitungsfehler (z.B. falsche Benutzereingaben) vorkommen, deren Auftreten erst im weiteren Verlauf des Prozesses festgestellt wird. Zur Behandlung solcher Fälle werden Aktionen, wie die Wiederholung oder das Auslassen von Prozeßschritten, die Durchführung alternativer Tätigkeiten oder das Wiederaufsetzen der Bearbeitung in einem früheren Zustand, erforderlich.
- Trotz aller Sorgfalt kann es bei der WF-Modellierung zu Mängeln kommen, die erst zur Laufzeit erkannt werden und deshalb im laufenden Betrieb beseitigt werden müssen. Sie sind meist die Folge ungenauer bzw. falscher Beobachtungen der Realwelt [Nutt96, StMi95]. So kennt der Modellierer in der Regel nicht alle Umweltgegebenheiten der abzubildenden Prozesse, wodurch bestimmte Informationen bei der Modellierung verloren gehen. Die falsche Interpretation einer Aufgabenbeschreibung oder Fehler bei der Modellbildung (z.B. Festlegung inkorrekter Verzweigungsbedingungen) können ebenfalls Laufzeitanpassungen nach sich ziehen. Dasselbe gilt für den Fall zu starrer oder zu vager WF-Beschreibungen [Nutt96].

Weitere Szenarien für Ausnahmen in computerunterstützten Prozessen werden in [BaWa95, KrWo87, MüRa99, RHD98, Saas95, SaWh95, Schu96, StMi95] behandelt. Im allgemeinen sind Ausnahmen die Folge von Mängeln bei der Modellierung oder Ausführung von Prozessen. Hinzu kommen unvorhersehbare Ereignisse oder geänderte Umweltbedingungen, infolge derer ein WF nicht mehr wie geplant fortgeführt werden kann. Zu erwähnen bleibt, daß viele Ausnahmesituationen auch die Folge einer unzureichenden Prozeßqualität sind. Sie lassen sich durch Qualitätssicherungsmaßnahmen und Prozeßverbesserungen zumindest teilweise vermeiden [HaSt95, StMi95]. Ein WfMS, das es Anwendern erlaubt, in flexibler Form vom vormodellierten Ablauf abzuweichen, bietet hier die Möglichkeit, durch die Auswertung der vorgenommenen Abweichungen zu verbesserten Prozeßmodellen zu gelangen.

2.1.2.3 Behandlung von Abweichungen

Für den Umgang mit Abweichungen in prozeßorientierten Anwendungssystemen ergeben sich zwei grundsätzliche Anforderungen: Sie müssen erkannt bzw. dem System mitgeteilt und ihre Auswirkungen angemessen behandelt werden. Je nach Art und Kontext der Abweichung gestalten sich diese beiden Aufgaben unterschiedlich. Ein wichtiger Faktor ist die *Planbarkeit* der Abweichung. Eine Abweichung ist *planbar*, wenn der Kontext ihres Auftretens und die zu ihrer Behandlung notwendigen Aktionen a priori bekannt sind. Für diesen Fall ist es prinzipiell möglich, die Abweichung vom Standardablauf bei der WF-Modellierung zu berücksichtigen, so daß zur Laufzeit eine *automatische Behandlung* durch das WfMS erfolgen kann. Eine Voraussetzung hierfür ist, daß dem WfMS das Auftreten der Ausnahme mitgeteilt und die zu ihrer Behandlung notwendigen Informationen im WF-Modell hinterlegt werden können. Dies ist für bestimmte Ausnahmen, die bei der Bearbeitung einzelner Aktivitäten auftreten können, teilweise möglich. Stehen etwa die Gründe für das potentielle Scheitern einer Aktivität a priori fest und ist auch bekannt wie in solchen Fällen verfahren werden soll (z.B. Wiederholung/Auslassen der Aktivitätenbearbeitung, kontrolliertes Zurücksetzen, Abbrechen des

WF), so ist eine Berücksichtigung auf Modellebene möglich. Voraussetzung hierfür ist, daß das verwendete WF-Metamodell entsprechende Beschreibungskonstrukte anbietet, wobei Normal- und Ausnahmefall getrennt voneinander beschreibbar sein sollten [Ober96].

Im allgemeinen ist es nicht möglich, alle Varianten eines Prozesses in einem WF-Modell abzubilden. Für komplexe Prozesse, die mehrere Dutzend Arbeitsschritte und mehrere Hierarchiestufen umfassen [JST97, Schu96], ist dies allein aus kombinatorischen Gründen nicht sinnvoll [RHD98, StMi95]. Aber selbst für überschaubare WFs können unvorhersehbare Situationen eintreten, die Abweichungen vom geplanten Ablauf erzwingen. Solche Fälle können nicht immer automatisch verarbeitet werden. Vielmehr ist ein situationsbedingtes Eingriffen notwendig, das Wissen zum Kontext und zur Art der aufgetretenen Ausnahme erfordert [Such87]. Aus diesem Grund müssen Anwender manuell in die Kontrolle des WF eingreifen können, um ihn an die geänderte Situation anzupassen. Dazu muß ein prozeßorientiertes Anwendungssystem berechtigten Akteuren die Möglichkeit einräumen, einzelne Aktivitäten oder Gruppen von Aktivitäten zu wiederholen, zu überspringen, vorzeitig zu bearbeiten oder durch andere Aktivitäten zu ersetzen. Um bei der WF-Modellierung nicht eingeplante Schritte ablaufbezogen ausführen zu können, müssen darüber hinaus dynamisch Aktivitäten in den Ablaufgraphen der Instanz eingefügt und durch das WfMS gesteuert werden können. Weitere wichtige Handlungsmöglichkeiten sind das Vertauschen von Aktivitätenreihenfolgen, die wiederholte Bearbeitung von Teilen des WF (z.B. durch kontrolliertes Zurücksetzen der Bearbeitung) oder der kontrollierte Abbruch des WF. Solche *Ad-hoc-Abweichungen* betreffen in der Regel einzelne WF-Instanzen.

Für langandauernde, zyklische Prozesse ergibt sich in Verbindung mit Abweichungen und ihrer Behandlung ein weiterer wichtiger Aspekt. So kann es sinnvoll sein, eine Modifikation des Ablaufgraphen einer WF-Instanz ggf. nur *temporär* vorzunehmen und sie später wieder zurückzunehmen. Der Grund dafür ist, daß Ad-hoc-Änderungen oftmals situationsabhängig, zur Behandlung einer aufgetretenen Ausnahme, erfolgen. Betreffen sie einen Bereich des Ablaufgraphen der mehrfach durchlaufen wird, so muß die Änderung vor dessen wiederholter Ausführung, ggf. wieder rückgängig gemacht werden, falls die Ausnahme nicht länger zutrifft. Andererseits gibt es aber auch WF-Instanzänderungen, die *permanent*, d.h. für die Restdauer der WF-Instanz, erhalten bleiben sollen. Verschlechtert sich zum Beispiel der Zustand eines Patienten im Verlauf eines (zyklischen) Behandlungsprozesses, so kann es erforderlich werden, die Gabe eines bestimmten Medikament für den aktuellen Behandlungszyklus auszulassen, sie aber in späteren Zyklen wieder durchzuführen, wenn es dem Patienten wieder besser geht (*temporäre Änderung*). Reagiert ein Patient dagegen auf ein Medikament allergisch, muß es ggf. für alle Behandlungszyklen abgesetzt werden (*permanente Änderung*).

2.1.3 Änderung von Standardprozessen

Betriebliche Strukturen und Abläufe unterliegen einem kontinuierlichen Verbesserungsprozeß und müssen häufig geändert werden [HaSt95, NiPi95, Schu96]. Aus diesem Grund sind prozeßorientierte Anwendungssysteme nur dann sinnvoll einsetzbar, wenn sie sich rasch und kostengünstig an geänderte Ablauf- und Organisationsstrukturen anpassen lassen. Gelingt dies nicht, wird die Diskrepanz zwischen realem Arbeitsablauf und rechnergestütztem WF auf Dauer zu groß, so daß eine optimale Prozeßunterstützung nicht mehr gegeben ist [StMi95]. Dies wiederum würde eine sinkende Benutzerakzeptanz mit sich bringen. Organisatorische Änderungen machen Anpassungen in der Definition von Prozeßvorlagen oder anderen Teilmodellen (z.B. Organisationsmodell) notwendig. Nach ihrer Freigabe müssen sie bei der Ausführung neu erzeugter WF-Instanzen berücksichtigt werden. Dies reicht jedoch nicht immer aus, insbesondere wenn langandauernde Prozesse unterstützt werden sollen. Idealerweise sollten hier Änderungen einer Vorlage nicht nur auf neu erzeugte WF-Instanzen wirken, sondern, soweit sinnvoll und möglich, auch auf bereits aktive WFs, die bisher gemäß der alten Vorlage ausgeführt worden sind. Es ergeben sich Querbezüge zu Ad-hoc-Änderungen einzelner WF-Instanzen (vgl. Abschnitt 2.1.2), nämlich die Unterstützung von dynamischen Änderungen an laufenden WFs. Da es hier aber um konzeptionelle Änderungen der Prozeßvorlagen geht, muß im allgemeinen von erheblich größeren Umformungen des Ablaufgraphen ausgegangen werden, als man es im Ad-hoc-Fall zulassen kann und wird. Es stellt sich hier in verschärfter Weise die Frage nach der Anpassung von laufenden Prozeßinstanzen an die neue Vorlage sowie der Koexistenz zwischen Prozeßinstanzen alter und neuer Form.

2.1.4 Zusammenfassung

Für die Akzeptanz prozeßorientierter Anwendungen ist es wichtig, daß die Systeminitiative nicht zu stark betont und die Benutzer in ihrem Handeln nicht zu sehr eingeschränkt werden. In vielen Umgebungen ist jedes starre System, auch bei ansonsten idealer Prozeßunterstützung, zum Scheitern verurteilt [DRK97, ElNu93]. Zur Erzielung eines flexiblen Ausführungsverhaltens gibt es verschiedene Ansatzpunkte für die WF-Modellierung und -Ausführung, die zum Teil komplementär sind:

- *Flexibilität durch weniger starre Ausführungsmodelle*: Um zu weniger starren Ausführungsmodellen zu gelangen, müssen Akteure zur Laufzeit flexibel vom vormodellierten WF abweichen können. Zur Verwirklichung solcher Ad-hoc-Abweichungen muß das WfMS geeignete Mechanismen zur dynamischen Modifikation von sich in Ausführung befindlichen WFs bereitstellen. Ergänzend sollten Konzepte zur Vormodellierung und automatischen Behandlung planbarer Abweichungen angeboten werden. Mechanismen zur dynamischen Änderung von WFs werden auch für die Konversion von WF-Instanzen an geänderte WF-Vorlagen benötigt.
- *Flexibilität durch weniger detaillierte (vage) Prozeßbeschreibungen*: Um Anwendern bei der Bearbeitung einer Aufgabe Freiheitsgrade einzuräumen, sollten Prozesse nicht bis ins letzte Detail beschrieben werden. Statt dessen kann es sinnvoll sein, bestimmte Teile des WF erst zur Laufzeit durch Benutzer festlegen zu lassen. Konzepte für die dynamische Erweiterung und das späte Modellieren von WFs bieten hier interessante Perspektiven. Wichtig ist die Wahl des richtigen Detaillierungsgrades bei der Modellierung. Einerseits sollte eine WF-Vorlage genügend Informationen umfassen, um die Akteure möglichst optimal zu unterstützen, andererseits machen zu detaillierte Ablaufpläne vermehrt Laufzeitabweichungen notwendig.
- *Flexibilität durch weniger präskriptive Modelle*: Ein WF-Modell heißt *präskriptiv*, wenn es Anwendern vorschreibt, wie eine Aufgabe bearbeitet werden soll. Das bedeutet, daß die WF-Aktivitäten sowie ihre Ausführungsreihenfolgen und –bedingungen festliegen. Laufzeitabweichungen sind dagegen nicht möglich. Dieser Ansatz erweist sich nicht nur in Ausnahmesituationen als zu starr, sondern er hindert Anwender auch daran, anstehende Aufgaben so zu verrichten, wie es für sie am effektivsten ist [ElWa94, Nutt96]. Aus diesem Grund muß ein WF-Modell auch unstrukturierte Anteile besitzen können, bei denen die Details bzgl. durchzuführenden Aufgaben lediglich in abstrakter Form vorliegen [BaWa95, CrMa95]. In [BlNu95] wird ein solcher Ansatz vorgestellt, bei dem unstrukturierte Teilaufgaben auf sog. Zielknoten im WF-Graphen abgebildet werden. Ein Zielknoten repräsentiert einen Teilausschnitt des Prozesses, für dessen Bearbeitung lediglich Vorgaben in Form von Zielen oder Leitlinien vorliegen. Ihre konkrete Gestaltung bleibt den Anwendern überlassen, die vom WfMS Informationen zum Kontext der Aufgabenbearbeitung erhalten. Ein Vorteil dabei ist, daß die Bedingungen für die Ausführung der unstrukturierten Tätigkeiten feststehen und deshalb bei der WF-Steuerung berücksichtigt werden können.

Üblich ist auch die Unterscheidung zwischen „A-priori-Flexibilität“ und „A-posteriori-Flexibilität“ (vgl. [Joer98]). *A-priori-Flexibilität* bezeichnet alle Maßnahmen, die der Entwerfer zur Modellierungszeit ergreifen kann, um das WF-Ausführungsverhalten flexibel zu gestalten. Hierunter fallen z.B. Konzepte für die Vormodellierung planbarer Abweichungen, für das späte Modellieren von WF-Aspekten [HHJS97, HSS96] oder für das späte Binden von WF-Ressourcen (z.B. dynamische Rollenauflösung, spätes Binden von Sub-WFs) [HHSW96, Han97]. Mit *A-posteriori-Flexibilität* werden dagegen alle Maßnahmen bezeichnet, die darauf zielen, Workflows zur Laufzeit in ihrer Struktur und ihrem Status dynamisch anzupassen (eine etwas allgemeinere Definition dieses Begriffs findet man in [BFG93, Joer98]). Wie gezeigt werden solche dynamischen Änderungen aus mehreren Gründen und in unterschiedlichem Kontext notwendig. Sie können eine oder mehrere WF-Instanzen betreffen, sich auf verschiedene WF-Aspekte beziehen, von unterschiedlichen Benutzergruppen definiert oder auf unterschiedliche Art und Weise festgelegt werden. Ein adaptives WfMS muß alle diese Fälle unterstützen, um wirklich breit einsetzbar zu sein. Für die Einteilung dynamischer WF-Änderungen sind u.a. die folgenden Fragestellungen hilfreich:

- Aus welchem Grund soll die WF-Änderung durchgeführt werden?
- Betrifft die Änderung eine Prozeßvorlage (und ggf. davon abgeleitete WF-Instanzen) oder eine einzelne WF-Instanz?

- Soll eine Prozeßvorlagen-Änderung nur für zukünftige WF-Instanzen gelten oder soll sie – soweit sinnvoll und möglich – auch auf laufende Instanzen propagiert werden?
- Welche WF-Aspekte (Kontrollfluß, Datenfluß, Aktivitäten, Zeit, Bearbeiterregelungen, Status, usw.) sollen abgeändert werden?
- Wer darf die Änderung anstoßen und wer darf sie vornehmen (prozeßbeteiligte Akteure, Prozeßadministrator, Systemadministrator, automatische Agenten)?
- Wie kann die Änderung festgelegt werden (graphischer Editor, menügesteuert, deklarativ, usw.)?
- Wann wird die Änderung vorgenommen (vor/bei Erzeugung der WF-Instanz oder im Verlauf ihrer Ausführung)? In welchem Modus soll die Änderung durchgeführt werden (Off-Line, On-the-fly)? Können währenddessen nebenläufige Änderungen stattfinden?
- Wie lange soll eine WF-Instanzänderung erhalten bleiben (temporär, permanent)?

2.2 Möglichkeiten und Grenzen heutiger Workflow-Management-Systeme

Das Spektrum heutiger WfMS ist in Wirklichkeit sehr viel breiter als in Kapitel 1 dargestellt. Es reicht von *nachrichtenbasierten* WfMS („Email-Attachment-Workflow“) bis hin zu „echt“ *prozeßorientierten* WfMS („Production Workflow“). Erstere unterstützen das elektronische Weiterleiten von Formularen oder Dokumenten. Eine systemseitige WF-Steuerung gibt es nicht oder nur in eingeschränkter Form. Aus diesem Grund lassen sich Ad-hoc-Abweichungen mit einfachen Mitteln verwirklichen. Dem gegenüber stehen prozeßorientierte WfMS. Sie gestatten die Integration autonomer und verteilter Anwendungskomponenten, bei allerdings starrer WF-Steuerung durch das WfMS. Ad-hoc-Abweichungen gestalten sich bei WfMS dieser Kategorie schwierig. Dazwischen gibt es WfMS unterschiedlicher Ausprägungsform [GHS95]. Abhängig von der Strukturierbarkeit und Komplexität der Prozesse ist die folgende Unterscheidung üblich (vgl. Abbildung 2-2): *Kollaborative WFs* bezeichnen unstrukturierte Gruppentätigkeiten. *Ad-hoc-WFs* unterstützen einfache, meist sequentielle Abläufe, deren Definition spontan durch die Anwender zur Laufzeit erfolgt. *Administrative WFs* beinhalten einfache Koordinierungsregeln zur Steuerung von Informationsprozessen niedriger Komplexität (z.B. Weiterreichen eines Dokumentes). Für ihre Unterstützung reicht in vielen Fällen eine Mail-Kommunikation aus. *Produktions-WFs* [LeRo00] dagegen beschreiben komplexe, langandauernde Prozesse, die den Zugang zu existierenden, heterogenen Informationssystemen erfordern. Typischerweise können hier zu einer Prozeßvorlage bis zu mehrere tausend WF-Instanzen gleichzeitig aktiv sein.

Heutige WfMS unterstützen WFs der genannten Kategorien mehr oder weniger gut. Sie unterscheiden sich im Hinblick auf die Ausdrucksmächtigkeit der verwendeten WF-Metamodelle, die Möglichkeiten zur (formalen) Validierung der WF-Modelle, die Art und Weise in der WFs implementiert und zur Laufzeit durch das WfMS koordiniert werden, die Wartbarkeit der resultierenden Anwendungssysteme oder die angebotenen Möglichkeiten für dynamische WF-Änderungen. Das Kennen und Verstehen der zugrundeliegenden Konzepte, des Entwicklungsstandes und des Potentials dieser Systeme hilft, ihre Eignung hinsichtlich einer gegebenen Aufgabenstellung besser einschätzen zu können. Im folgenden werden typische Systemkategorien behandelt und die prinzipiellen Möglichkeiten und Grenzen entsprechender WfMS diskutiert. Einen Schwerpunkt bildet die Untersuchung der Fragestellung, ob und – falls ja – in welcher Form dynamische WF-Änderungen unterstützt werden. Als primäres Unterscheidungsmerkmal dienen die den WfMS zugrundeliegenden Koordinationsmodelle, die die Art und Weise, in der WF-Aktivitäten durch das WfMS koordiniert werden, regeln [Schw93, Rein93]:

- Prozeßorientierte Koordinationsmodelle und WfMS
- Objektmigrationsmodelle und dokumentenorientierte WfMS
- Informationsharing-basierte Koordinationsmodelle und formularbasierte Workflows
- Konversationsstrukturorientierte Koordinationsmodelle

Im folgenden werden diese Modelle und darauf basierende WfMS beschrieben. Dabei erfolgt eine Konzentration auf die im Kontext dieser Arbeit relevanten Aspekte. Orthogonale Fragestellungen, z.B. die Architektur der Systeme betreffend [BaDa98, JBS97, Jabl97], bleiben ausgeklammert. Für jedes Modell wird skizziert, wie sich die Entwicklung von WF-Anwendungen darstellt und welche Möglichkeiten zur Durchführung dynamischer WF-Änderungen bestehen.

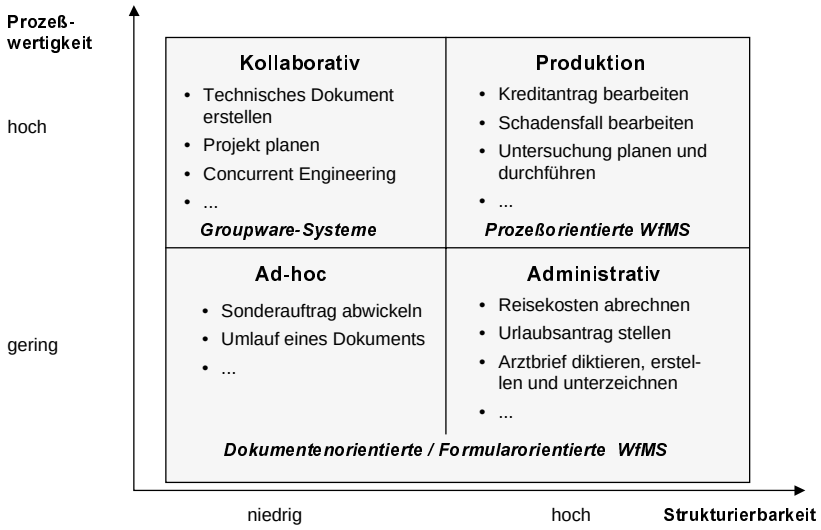


Abbildung 2-2: *Kategorien Workflow-basierter Anwendungen*

2.2.1 Prozeßorientierte Koordinationsmodelle und WfMS

Prozeßorientierte Koordinationsmodelle und darauf basierende WfMS strukturieren Abläufe aus zentraler Sicht. Die Modellbildung orientiert sich am Gesamtprozeß und an dessen Teilschritten. Jeder Prozeßtyp wird durch eine separate Prozeßvorlage beschrieben, wobei die Prozeßlogik (Kontroll- und Datenfluß) getrennt vom eigentlichen Applikationscode festgelegt und dem WfMS gegenüber explizit bekannt gemacht wird. Prozeßorientierte Koordinationsmodelle und die ihnen zugrundeliegenden Konzepte bilden die zentralen Ausgangspunkte dieser Arbeit. Auf ihre Stärken und Schwächen wurde bereits ausführlich in Abschnitt 1.3 eingegangen. An dieser Stelle erfolgt lediglich eine Einordnung in den Gesamtkontext, die eine Abgrenzung zu den nachfolgend vorgestellten Koordinationsmodellen ermöglichen soll.

2.2.1.1 WF-Modellierung und Anwendungsentwicklung

Für die WF-Modellierung werden üblicherweise graphische Beschreibungssprachen und Editoren verwendet, bei denen WF-Modelle durch gerichtete Graphen dargestellt werden, deren Knoten die Aktivitäten und deren Kanten die Kontroll- und Datenflußbeziehungen zwischen ihnen repräsentieren. Aktivitäten werden entweder Anwendungsprogramme zugeordnet oder sie werden weiter verfeinert, d.h. durch entsprechende Sub-WF-Modelle unterlegt. Es ist somit eine Top-Down-Vorgehensweise bei der Modellierung möglich. Für die Prozeßbeschreibung kommen Petri-Netze [AAH98, Baum96, Jens90, Ober96], hierarchische Automaten [Biec96, Hare87], Ereignis-Prozeßketten [Sche96] oder verschiedene Formen von Aktivitätennetzen [Elli79, LeAl94, Rein93] zum Einsatz.

Bei Petri-Netz-basierten Ansätzen orientiert sich die Modellierung an den Daten („Tokens“), die zwischen den Aktivitäten fließen [AAH98, Aals98, Gruh91, LöHa94, Ober96]. Kontrollflüsse werden dagegen nicht explizit modelliert, sondern ergeben sich implizit aus den festgelegten Datenabhängigkeiten. Eine Unterscheidung zwischen Kontroll- und Datenfluß ist somit – zumindest auf graphischer Ebene – nicht möglich (für weitere Details siehe Kapitel 10). Im Gegensatz dazu unterstützen Formalismen, wie State-/Activitycharts [Teeg96, Wodt95], ereignisgesteuerte Prozeßketten [Stro97, Wäch95] oder Aktivitätennetze [Elli79, Roll96], die getrennte Festlegung dieser beiden Modellaspekte. Dazu werden unterschiedliche Ansätze verfolgt: Für die Kontrollflußmodellierung werden entweder spezielle Knotentypen (Split-/Join-Knoten mit AND-, OR- oder XOR-Semantik) bereitgestellt [Elli79, LiPu97, RuSh94, Sche96] oder die Festlegung unterschiedlicher Verzweigungsarten erfolgt implizit, durch die Zuordnung von Transitionsbedingungen zu Kontrollkanten des WF-Graphen [CCPP95, Roll96, Wesk97]. Letztere entsprechen Prädikaten auf Daten aus dem WF-Datenkontext, die notwendige Bedingungen für die Durchführung der jeweiligen Zielaktivität beschreiben. Der Modellierer kann festlegen, ob zur Ausführung einer Aktivität die Transitionsbedingungen aller einmün-

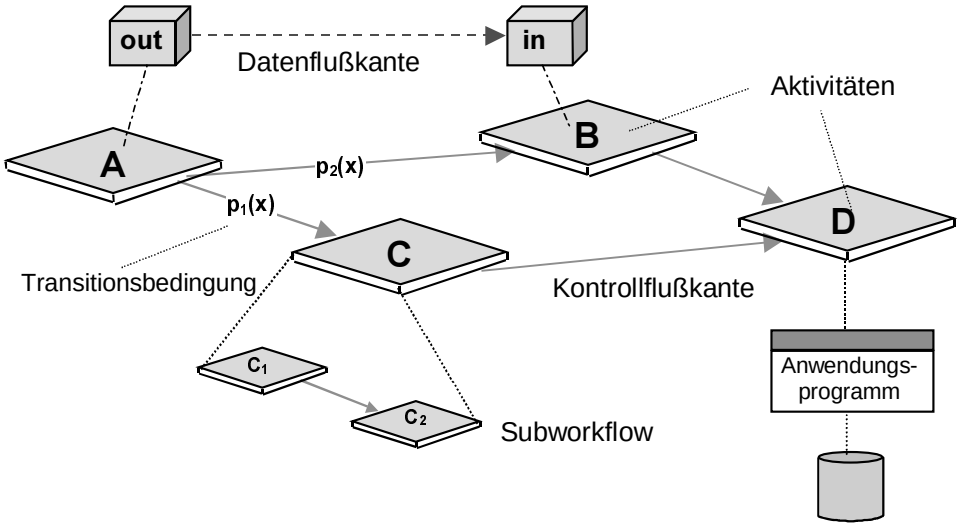


Abbildung 2-3: *Beispiel eines prozeßorientierten Ablaufmodells*

denden Kontrollkanten erfüllt sein müssen oder ob es genügt, wenn dies für eine Kante der Fall ist. Dadurch können, je nach Festlegung der Transitionsbedingungen, unterschiedliche Arten von Verzweigungen und Zusammenführungen modelliert werden. Die Datenflußmodellierung erfolgt entweder durch direkte Verknüpfung von Ausgabeparametern einer Aktivität mit Eingabeparametern nachfolgender Aktivitäten [Roll96] oder durch Beschreibung des lesenden und schreibenden Zugriffs der Aktivitäten auf gemeinsame genutzte Datenobjekte [Elli79, WäRe92]. In der Regel werden dabei die Strukturen und Typen der auszutauschenden Daten explizit beschrieben und dem WfMS bekannt gemacht [Gruh91, LeRo97, Ober96]. Die Entscheidung darüber, ob Anwendungsdaten direkt oder lediglich über entsprechende Referenzen zwischen Aktivitäten ausgetauscht werden, liegt beim Entwickler, und wird durch die Art und den Umfang der verwendeten Daten beeinflußt.

Der WF-Modellierung folgt die Implementierung der Arbeitsschritte, wobei dies auch die Anknüpfung existierender Anwendungskomponenten bedeuten kann. Für den Anwendungsentwickler werden API-Schnittstellen angeboten, um Eingabeparameter einer Komponente bei deren Aufruf mit Daten aus dem WF-Datenkontext zu versorgen und um die Werte ihrer Ausgabeparameter nach Beendigung der Programmausführung persistent in den WF-Datenkontext zu übernehmen [LeAl94, Rupi97]. Dadurch ist es möglich, daß derselbe Applikationsbaustein bei der Implementierung unterschiedlicher Prozeßvorlagen wiederverwendet werden kann [Krue92, NTMS91]. Auch der Aufruf entfernter Anwendungsdienste zur Ausführung von Prozeßschritten und die Integration von Groupware-Anwendungen ist prinzipiell möglich [AGSC95, Kuhn94, Schn96, Shet96, Webe97].

Prozeßorientierte WfMS bieten üblicherweise Standardoberflächen für die Arbeitslistenverwaltung an, mittels denen berechtigte Akteure neue WF-Instanzen erzeugen sowie zur Ausführung anstehende WF-Aktivitäten starten, beenden, abbrechen oder wiederholen können. Die Generierung und Verwaltung der Arbeitslisten übernimmt das WfMS. In der Regel werden auch Programmierschnittstellen angeboten, die die Realisierung eigener Bedieneroberflächen gestattet [RSD97, Saue96, Wolp97].

2.2.1.2 WF-Steuerung und dynamische WF-Änderungen

Die Ablaufsteuerung erfolgt logisch zentral[1] durch die Interpretation des WF-Graphen und durch Aufruf des jeweiligen Arbeitsschrittprogramms. Für Akteure besteht die Möglichkeit, den Prozeßverlauf mit zu verfolgen. Einzelne Arbeitsschritte bzw. die ihnen zugeordneten Anwendungskomponenten müssen über kein globales Ablaufwissen verfügen, sondern werden zur Laufzeit von der übergeord-

[1] Dies bedeutet nicht, daß die WF-Ausführung nicht auf mehrere Server verteilt sein kann, z.B. um den Gesamtdurchsatz oder die Verfügbarkeit des WfMS zu erhöhen. Allerdings wird dann – ähnlich wie bei verteilten Datenbanken [Dada96] – gefordert, daß das *Single-System-Image* für den Anwender erhalten bleibt. Ausführliche Untersuchungen zur Partitionierung von WF-Graphen und zur verteilten WF-Ausführung wurden in Meteor [Shet96, ShKo97], Mentor [Muth98, WoWe97, WWWK96] und $ADEPT_{distribution}$ [BaDa97, BaDa00, BRD00] angestellt. Einen Überblick zu verteilten WfMS-Architekturen gibt [BaDa98, BaDa99a].

neten Kontrollkomponente – der WF-Engine – gerufen. Dabei werden Eingabeparameter der gerufenen Applikation mit Daten aus dem WF-Datenkontext versorgt und nach ihrer Beendigung Ausgabeparameter übernommen.

Wie in Kapitel 1 gezeigt, besteht ein großer Mangel heutiger prozeßorientierter WfMS darin, daß von dem einmal modellierten WF zur Ausführungszeit nicht oder nur unter sehr eingeschränkten Bedingungen abgewichen werden kann. Das heißt Ad-hoc-Änderungen einzelner WF-Instanzen während ihrer Ausführung sind kaum möglich. Auch die Anpassung bereits aktiver WF-Instanzen an Änderungen in ihrer Prozeßvorlage sowie die Koexistenz zwischen Prozeßinstanzen alter und neuer Form werden in heutigen WfMS entweder überhaupt nicht oder nur sehr eingeschränkt unterstützt.

2.2.1.3 Bewertung

Prozeßorientierte WfMS bieten gute Voraussetzungen für die Modellierung, Realisierung und Wartung vorgangsorientierter Anwendungssysteme, insbesondere wenn es um die Unterstützung von gut strukturierten, repetitiven Prozessen geht. Die Trennung der Beschreibung verschiedener WF-Aspekte von der Implementierung der Arbeitsschritte bietet mehrere Vorteile (vgl. Kapitel 1): Sie vereinfacht die Anpaßbarkeit und Wartbarkeit vorgangsorientierter Anwendungssysteme und kann – durch Realisierung kleinerer, besser überschaubarer Programmbausteine – auch zu einer Erhöhung der Softwarequalität beitragen. Die explizite, formale Beschreibung von Kontroll- und Datenflüssen eröffnet zudem die Möglichkeit, das zukünftige Systemverhalten vorab zu evaluieren. Hierdurch können Entwurfsfehler, wie etwa zyklische Reihenfolgebeziehungen oder Programmteile, die aufgrund der festgelegten Transitionsbedingungen nie durchlaufen werden, noch vor Inbetriebnahme bzw. Implementierung des Anwendungssystems entdeckt werden. Die Durchführung der Ablaufsteuerung durch das WfMS bietet die Möglichkeit, die Anwendungsentwicklung durch Bereitstellung entsprechender Systemdienste für die Fehler- und Ausnahmebehandlung drastisch zu vereinfachen. Diese potentiell vorhandenen Möglichkeiten sind bislang allerdings nur rudimentär nutzbar, da die WF-Metamodelle heutiger prozeßorientierter WfMS nicht ausdrucksfähig und flexibel genug sind, um die betriebliche Realität mit ihren üblichen Ausnahmeregeln adäquat beschreiben zu können (vgl. Kapitel 1). Insbesondere die mangelhafte Unterstützung dynamischer WF-Änderungen schränkt ihre Anwendbarkeit auf relativ simple, a priori vollständig determinierte Abläufe ein. – Beispiele für prozeßorientierte WfMS sind *MQSeries Workflow* [LeRo00], *Staffware* [KaDe96] und *COSA* [JBS97].

2.2.2 Objektmigrationsmodelle und dokumentenorientierte WfMS

Bei Objektmigrationsmodellen und darauf basierenden dokumentenorientierten WfMS stehen die Dokumente (oder allgemeiner die Datenobjekte) eines Vorgangs und die Steuerung des Dokumentenflusses im Vordergrund und nicht, wie bei prozeßorientierten WfMS, die Koordination der einzelnen Arbeitsschritte. Den Ursprung dieser Systeme bilden Arbeiten aus dem Bereich der Büroautomatisierung [ElNu80, KRW90]. Ein Vorgang wird als eine Menge zusammengehörender Dokumente aufgefaßt, die während der Vorgangsbearbeitung von Arbeitsstation zu Arbeitsstation weitergeleitet werden. In diesem Zusammenhang wird häufig die Metapher der „elektronischen Umlaufmappe“ [KRW90] verwendet (vgl. Abbildung 2-4): Ein Vorgang entspricht dem Umlauf einer Handakte, die während der Vorgangsbearbeitung ein Netz von Bearbeitungsschritten durchläuft. Zur Durchführung einzelner Schritte können Akteure die Mappendokumente lesen oder bearbeiten, Haftnotizen anbringen sowie der Mappe neue Dokumente hinzufügen. Nach Abschluß einer Aktivitätenbearbeitung wird die Mappe an die nächste Arbeitsstation, gemäß dem vordefinierten Laufweg, weitergeleitet. Auch die Rückgabe der Mappe an frühere Bearbeiter sowie ihre Wiedervorlage oder Ablage sind einfach umsetzbar.

2.2.2.1 WF-Modellierung und Anwendungsentwicklung

Für jeden Vorgangstyp muß vom Modellierer eine Mappenvorlage definiert werden. Sie legt sowohl die Mappendokumente als auch den Laufweg der Mappe fest, d.h. es wird explizit beschrieben, welchen Akteuren in welcher Reihenfolge die Mappe zur Bearbeitung vorgelegt werden soll und auf welche Dokumente dabei mit welchen Anwendungen zugegriffen werden darf. Struktur und Inhalt der Dokumente werden dem WfMS gegenüber nicht bekannt gemacht, sondern obliegen der Kontrolle der Aktivitätenprogramme [KaRa90]. Die Modellierung des Datenflusses besteht darin, festzulegen, auf

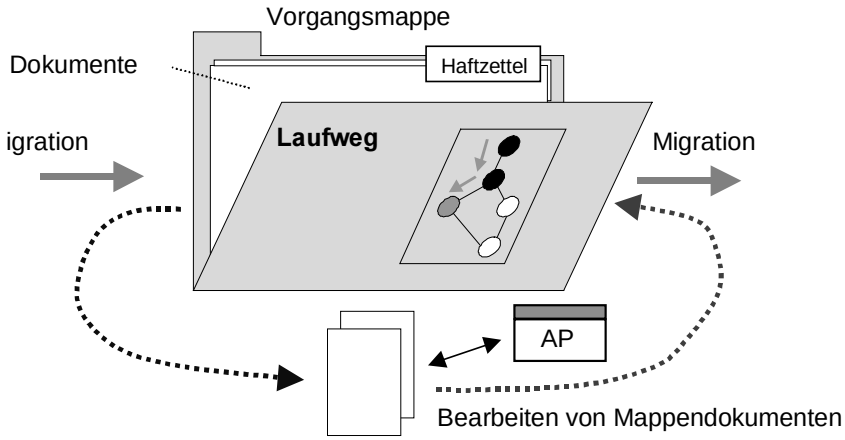


Abbildung 2-4: *Konzept der elektronischen Umlaufmappe*

welche Dokumente bei der Bearbeitung einzelner Aktivitäten zugegriffen werden darf. Für die Kontrollflußmodellierung werden graphische Sprachen verwendet [Karb94], deren Ausdrucksmächtigkeit geringer ist als bei prozeßorientierten WfMS. Da das WfMS die Inhalte der von ihm verwalteten Dokumente nicht kennt und keine Mechanismen zur Verwaltung WF-relevanter Daten (z.B. Mappenvariablen) existieren, ist es – bei strikter Auslegung des skizzierten Modells – nicht möglich, bedingte Ausführungen mittels Verzweigungsprädikaten (z.B. „$x \geq 40$“) zu definieren. Statt dessen basieren Verzweigungsentscheidungen meist auf Rückgabewerten der aufgerufenen Aktivitätenprogramme [Karb94, Rupi97], wodurch die Unabhängigkeit von Prozeßvorlage und Anwendungscode nicht mehr gegeben ist. Auch die Möglichkeiten, das Systemverhalten vorab zu evaluieren, sind beschränkt.

Da Mappen konzeptionell nicht teilbar sind, bleibt die Komplexität der beschreibbaren Abläufe beschränkt. Bei strikter Auslegung des skizzierten Modells könnten lediglich serielle und alternative Pfade abgebildet werden, wohingegen nebenläufige Bearbeitungen nicht modellierbar sind. Aus diesem Grund werden in realen Implementierungen meist Erweiterungen vorgenommen, die es erlauben, nebenläufige Bearbeitungen über *abhängige Vorgänge* zu bewerkstelligen [Karb94]. Hier legt der Anwender (dynamisch) einen parallelen Bearbeitungszweig fest, indem er eine Aktivität angibt, nach deren Beendigung ein abhängiger Vorgang gestartet werden soll. Dieser Vorgang kann freilaufend sein oder er wird mit einer Folgeaktivität synchronisiert. In letzterem Fall muß er vor deren Start beendet werden. Ein abhängiger Vorgang kann auf alle Dokumente zugreifen, die bei seinem Start in der Originalmappe enthalten sind bzw. die während seiner Bearbeitung entstehen. Zu diesem Zweck wird bei einer parallelen Aufsplittung eine neue Mappe angelegt. Sie umfaßt die Beschreibung des Vorgangs sowie Kopien aller bereits existierenden Dokumente. Problem ist nun, daß verschiedene Kopien desselben Dokuments in parallelen Ablaufzweigen unabhängig voneinander geändert werden dürfen. Die hieraus resultierenden Konflikte sind erst bei der Zusammenführung der Mappen erkennbar und müssen vom Anwender manuell aufgelöst werden. Zu ihrer Vermeidung wird vorgeschlagen, Nebenläufigkeit in Objektmigrationsmodellen durch die geeignete Partitionierung der Anwendungsdaten zu unterstützen [Adam96]. Die Bearbeiter von Aktivitäten verschiedener Zweige sollen zwar dieselbe Kopie der Originalmappe erhalten, aber mit jeweils disjunkten Schreibrechten auf den Dokumenten.

Im Gegensatz zu prozeßorientierten WfMS kann die Implementierung der Aktivitätenprogramme im allgemeinen nicht unabhängig vom Kontext ihrer Verwendung erfolgen. Dies gilt in besonderem Maße, wenn zwischen den Prozeßschritten strukturierte Daten ausgetauscht werden sollen. Da dies bei dokumentenorientierten Ansätzen nur indirekt, über Lese- und Schreibzugriffe auf die gemeinsamen Mappendokumente möglich ist, ist die Implementierung der Aktivitätenprogramme von der Struktur dieser Dateien abhängig. Das bedeutet, daß bei Änderungen der Dokumentstruktur ggf. auch Anpassungen der darauf zugreifenden Aktivitätenprogramme vorgenommen werden müssen.

2.2.2.2 WF-Steuerung und dynamische WF-Änderungen

Auf dem Objektmigrationsmodell basierende WfMS unterstützen meist eine dezentrale Kontrolle der WFs [KaRa90, KaRa91]. Die Mappenobjekte kontrollieren ihre Bearbeitung selbst und migrieren zwischen verschiedenen Stellen. Einen zentralen WF-Server, der die WF-Ausführung koordiniert, gibt es

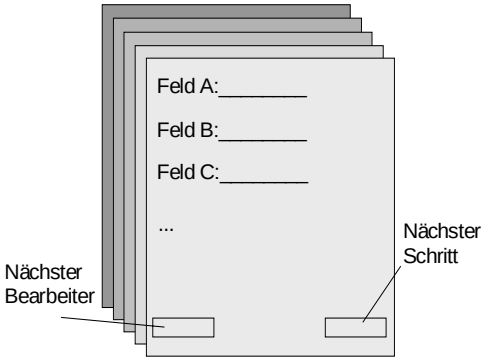


Abbildung 2-5: *Formularbasierter WF*

nicht. Dadurch wird die Bereitstellung globaler Übersichten bei parallelen Ausführungspfaden zumindest erschwert. Die Datenflußsteuerung reduziert sich auf den Austausch von Anwendungsdateien, die vom WfMS zwar meist verwaltet werden, deren Struktur und Inhalt ihm aber nicht bekannt sind [KaRa90]. Wird vom Benutzer eine Aufgabe aus der Arbeitsliste ausgewählt, ruft die lokale Verarbeitungskomponente das für die Bearbeitung vorgesehene Anwendungsprogramm auf. Dabei werden dem Programm Namen und Zugriffspfade der bearbeitbaren Mappendokumente übergeben [Karb94].

Im Prinzip genügt es, Informationen zum jeweils nächsten Bearbeitungsschritt erst bei der Beendigung einer Aktivität zu bestimmen. Dadurch können einfache Ad-hoc-Abweichungen, wie das Hinzufügen eines neuen Bearbeitungsschritts direkt nach der Beendigung einer Aktivität oder die Rückgabe der Mappe an den vorangehenden Bearbeiter mit anschließender Wiedervorlage, im Vergleich zu prozeßorientierten WfMS mit relativ geringem Aufwand realisiert werden. Dasselbe gilt für die Unterstützung von Ad-hoc-WFs, zumindest wenn die Bearbeitung unstrukturierter Dokumente im Vordergrund steht. Gegenwärtige Implementierungen unterstützen meist nur sequentielle Ad-hoc-WFs. Modifikationen einer Mappenvorlage und die Propagierung der vorgenommenen Änderungen auf bereits laufende WF-Instanzen sind in dokumentenorientierten WfMS gegenwärtig nicht möglich.

2.2.2.3 Bewertung

Objektmigrationsmodelle und dokumentenorientierte WfMS basieren auf einem einfachen Metamodell. Nichtsdestotrotz bieten sie für einige Anwendungen eine natürliche Sichtweise auf Arbeitsabläufe. In besonderem Maße gilt dies für Prozesse aus dem Bürobereich [ElNu80, Elli83, Tsich85], für die das Konzept ursprünglich entwickelt wurde [HEA90, KaRa90]. Bei Anwendung auf komplexe WFs zeigen dokumentenorientierte WfMS aber Schwächen. Zu nennen sind die geringe Ausdrucksmächtigkeit der verwendeten Metamodelle – viele praktisch relevante Arbeitsabläufe lassen sich nicht oder nur unter erheblichen Schwierigkeiten abbilden -, die mangelhaften Möglichkeiten zur Integration externer Applikationen, die schlechte Wartbarkeit der resultierenden Anwendungssysteme sowie die fehlende Sicherstellung der Datenkonsistenz bei Ablaufänderungen. Die letzten beiden Punkte sind Folge der starken Dokumentenorientierung. Sollen strukturierte Daten zwischen Aktivitäten ausgetauscht werden, können diese dem WfMS nicht explizit bekannt gemacht werden. Die Datenabhängigkeiten zwischen Aktivitätenprogrammen sind dem WfMS deshalb nicht bekannt, sondern ergeben sich implizit aus ihren Lese- und Schreiboperationen auf gemeinsamen Mappendokumenten. Dadurch kann bei Modifikationen des WF systemseitig nicht immer erkannt werden, welche Anpassungen am Datenfluß bzw. an den Anwendungskomponenten vorgenommen werden müssen, um anschließend einen fehlerfreien Ablauf und konsistente Anwendungsdaten garantieren zu können. Ähnliche Probleme ergeben sich bei Änderungen der Struktur einzelner Dokumente und der Implementierung von Anwendungskomponenten. Aus den genannten Gründen sind dokumentenorientierte WfMS vorwiegend für die Unterstützung administrativer WFs geeignet. Mit zunehmender Komplexität der Prozesse wird es jedoch immer schwieriger, noch zu fehlerfreien und wartbaren Anwendungssystemen zu gelangen [SAM91]. Ausführliche Darstellungen zu Objektmigrationsmodellen findet man in [BMR96, ScMa91, VoEr92]. Realisiert wurde das Konzept der elektronischen Umlaufmappe z.B. in den Systemen *ProMInanD* [Karb94] und *Workflow* [JKMW97].

2.2.3 Informationsharing-basierte Koordinationsmodelle und formularbasierte WFs

Im Zentrum *informationsharing-basierter* Koordinationsmodelle stehen – ähnlich wie beim Objektmigrationsansatz – die im Verlauf eines Vorgang anfallenden Anwendungsdaten. Sie werden in einem gemeinsamen Datenbereich verwaltet, auf den Akteure lesend und schreibend zugreifen können. Im Gegensatz zu prozeß- und dokumentenorientierten WfMS erfolgt keine explizite Beschreibung der Ablauflogik. Wer welche Aufgaben in welcher Reihenfolge zu bearbeiten hat, wird entweder gar nicht oder nur implizit festgelegt. Dem entsprechend passiv verhält sich das System. Im einfachsten Fall

werden vom Entwickler lediglich eine Gruppe miteinander kooperierender Personen und deren Zugriffsrechte auf die gemeinsamen Daten festgelegt [Gros97]. Durch die Koordination der Zugriffe und die Festlegung geeigneter Sichten können einfache Arbeitsabläufe abgebildet werden. Dieser Ansatz eignet sich besonders gut für die Realisierung einfacher Formularflüsse [ElNu80, Geha82, HEA90, Tsich82]. Eine passende Metapher ist hier die Bearbeitung eines Formularsatzes, dessen Durchschläge logisch den Aktivitäten entsprechen (vgl. Abbildung 2-5). Die Durchführung einer Aktivität erfolgt durch die Bearbeitung eines Formulars, in das Feldwerte aus zuvor ausgefüllten Formulardurchschlägen übernommen und ggf. abgeändert sowie Werte für neu hinzugekommene Felder festgelegt werden.

2.2.3.1 WF-Modellierung und Anwendungsentwicklung

Zur Realisierung formularbasierter WFs kommen häufig Groupware-Systeme, die Dienste zur Verwaltung semi-strukturierter Dokumente anbieten, zum Einsatz. Ein Vorgangstyp wird dabei logisch auf einen Dokumententyp abgebildet, dessen Felder die Vorgangsdaten beschreiben. Entsprechend repräsentiert ein Dokument, d.h. die Instanz eines Dokumententyps, eine konkrete Vorgangsinstanz, im folgenden *Vorgangsdokument* genannt. Formularflüsse werden durch die Koordination der Zugriffe auf Vorgangsdokumente sowie durch die Bildung aufgaben- bzw. bearbeiterspezifischer Sichten auf die Dokumentinhalte realisiert. Zu diesem Zweck müssen, zusätzlich zu den eigentlichen Anwendungsdaten, Steuer- und Statusfelder wie „nächster Bearbeitungsschritt“ oder „nächster Bearbeiter“ in die Dokumente aufgenommen werden. Die Belegung dieser Felder regelt zur Laufzeit, welche Akteure mit welchen Formularen auf das Vorgangsdokument zugreifen dürfen. Ein Formular zeigt im allgemeinen nur eine Teilmenge der Dokumentfelder an. Nach der Durchführung der gewünschten Aktionen wird das Dokument geschlossen, und die geänderten Feldwerte werden gespeichert. Zuvor müssen der nächste Schritt und Bearbeiter in den dafür vorgesehenen Steuerfeldern eingetragen werden, so daß mit der Bearbeitung der Nachfolgeaktivität fortgefahren werden kann (vgl. Abbildung 2-6).

Die Aufrufreihenfolge der Formulare wird nicht auf graphischer Ebene definiert, sondern in ihrer Beschreibung „versteckt“. Beim Schließen eines Formulars wird dem Feld „Nächster Bearbeitungsschritt“ ein konstanter Wert zugewiesen, der den jeweiligen Folgeschritt festlegt. Sollen nicht nur sequentielle, sondern auch alternative Bearbeitungen möglich sein, muß dieser Wert dynamisch definiert werden, etwa in Abhängigkeit von Feldwerten oder manuell durch Benutzer. In Bezug auf Parallelität ergeben sich ähnliche Probleme wie beim Objektmigrationsmodell. Bei fehlenden Mechanismen zur Synchronisation nebenläufiger Bearbeitungen können Schreibkonflikte auf Dokument- oder zumindest auf Feldebene resultieren. Existierende Systeme bieten keine oder nur eingeschränkte Unterstützung zur Vermeidung oder Auflösung solcher Konflikte. Wird beim Speichern eines Dokumentes ein Konflikt festgestellt, wird oftmals nur eine Kopie des Dokumentes oder zumindest einzelner Felder angelegt. Das Zusammenführen verschiedener Datenversionen ist die Aufgabe des Anwenders. Bei naiver Implementierung muß die Konfliktbereinigung auch dann manuell erfolgen, wenn der Konflikt zwar auf Dokument-, nicht aber auf Feldebene aufgetreten ist, d.h. die in parallelen Ausführungspfaden vorgenommenen Änderungen disjunkte Feldermengen betreffen. Auch die Zusammenführung paralleler Pfade gestaltet sich aufwendig, da sie explizit in den Formularen „programmiert" werden muß.

Arbeitsschritte werden durch einfache Formulare implementiert, wobei unterschiedliche Ansätze möglich sind. Für jede Tätigkeit kann ein Formular definiert werden, das die benötigte Teilsicht auf die Vorgangsdaten bietet. Um die Zahl an zu definierenden Formularen zu reduzieren, kann alternativ dazu für jeden Bearbeiter ein Formular entworfen werden, das auch zur Bearbeitung verschiedener Teilaufgaben verwendet werden kann. Für diesen Fall müssen, abhängig vom jeweiligen Arbeitsschritt, Datenfelder ein- bzw. ausgeblendet werden können. Die Wiederverwendung von Formularen und die Anknüpfung externer Anwendungskomponenten sind bei dem skizzierten Ansatz allerdings nicht oder nur unter erheblichen Schwierigkeiten möglich.

Die Erzeugung von Arbeitslisten erfolgt durch die Bildung von Teilsichten auf die Vorgangsdatenbank. Dabei wird eine Liste derjenigen Vorgangsdokumente erstellt, deren Feldwerte bestimmte Bedingungen erfüllen. Wird zur Verwaltung der Dokumente eine (Dokumenten-)Datenbank verwendet, lassen sich die Arbeitslisten durch einfache Anfragen gegen diese DB verwirklichen. Die Darstellung der Anfrageergebnisse erfolgt meist in Form einer Tabelle, deren Zeilen Vorgangsdokumenten und deren Spalten Feldwerten dieser Dokumente entsprechen. Möchte eine Akteur einen bestimmten Ein-

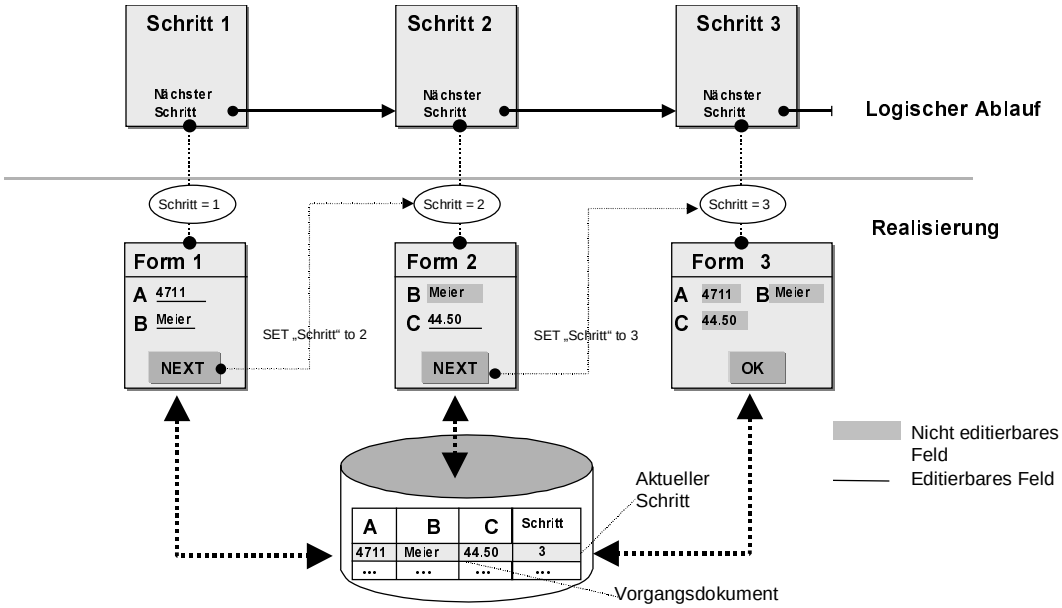


Abbildung 2-6: *Ablaufkoordination bei einem formularbasierten Workflow*

trag bearbeiten, wählt er ihn aus. Daraufhin wird das zugehörige Vorgangsdokument mit dem für den aktuellen Bearbeitungsstatus gültigen Formular geöffnet.

2.2.3.2 WF-Steuerung und dynamische WF-Änderungen

Während prozeß- und dokumentenorientierte WfMS die aktive Koordination von Arbeitsabläufen unterstützen, nimmt das skizzierte Koordinationsmodell eine passive Rolle bei der WF-Steuerung ein. Der Grund dafür ist, daß es keine WF-Engine gibt, die die Kontroll- und Datenflüsse des WF kennt und sie zur Laufzeit interpretiert. Für dynamische WF-Änderungen (bzw. Änderungen von Formularaufrufreihenfolgen) gibt es dem entsprechend keine Unterstützung. Bereits einfache Modifikationen gestalten sich aufwendig und fehleranfällig. Soll etwa die Reihenfolge zweier Schritte vertauscht werden, können Anpassungen in der Definition mehrerer Dokumente und Formulare notwendig werden. Welche Anpassungen konkret durchzuführen sind und welche Seiteneffekte sie bewirken, ist für das System selbst nicht feststellbar. Gegenüber prozeßorientierten WfMS sind WF-Änderungen bei formularbasierten WFs tendenziell unkritischer, da hier die Aktivitäten durch einfache Formulare implementiert sind. Einige Systeme erlauben es schließlich, Dokumente mit beliebigen Formularen zu öffnen [ReMo96]. Gibt es für einzelne Formularfelder keine entsprechenden Dokumentfelder, werden sie dynamisch der Dokumentbeschreibung hinzugefügt. Dadurch lassen sich Ad-hoc-Schritte realisieren.

2.2.3.3 Bewertung

Die skizzierte Vorgehensweise eignet sich zur Realisierung einfacher Formularflüsse. Sie wurde bereits Anfang der 80er Jahre in Ansätzen wie TLA [Tsich82, Tsich85] und Officetalk-Zero [ElNu80] verfolgt. Mittlerweile gibt es eine Reihe kommerzieller Produkte, wie *Notes* [ReMo96] oder *Exchange*, die in eine ähnliche Richtung zielen. Allerdings handelt es sich dabei nicht um WfMS im eigentlichen Sinne. Je komplexer Prozesse sind, desto aufwendiger gestaltet sich ihre Implementierung. Systemseitige Analysen sind infolge der impliziten WF-Definition nicht möglich. Aus demselben Grund müssen wichtige Elemente der Ablaufunterstützung, wie die Steuerung der Formularflüsse oder die Verwaltung von Arbeitslisten, vom Anwendungsentwickler selbst programmiert werden. Schließlich schränkt die fehlende Möglichkeit der Integration von Fremdapplikationen den Einsatz solcher Systeme im größeren Stil ein. Die Unterstützung unternehmensweiter Prozesse wäre allerdings, aufgrund der tendenziell schlechten Wartbarkeit und Anpaßbarkeit der resultierenden Anwendungssysteme, ohnehin kein realistisches Ziel. Ähnliche Erkenntnisse ergaben sich aus dem Projekt Officetalk-Zero [ElNu80], dessen Nachfolgeprojekt Officetalk-D [ElBe82] die implizite Beschreibung von Formularflüssen aufgegeben und mit ICN [Elli79] eine graphische Sprache für die Prozeßdefinition gewählt hat. Auch für Notes gibt es solche „On-top"-Aufsätze, wie Pavone Espresso (siehe Ab-

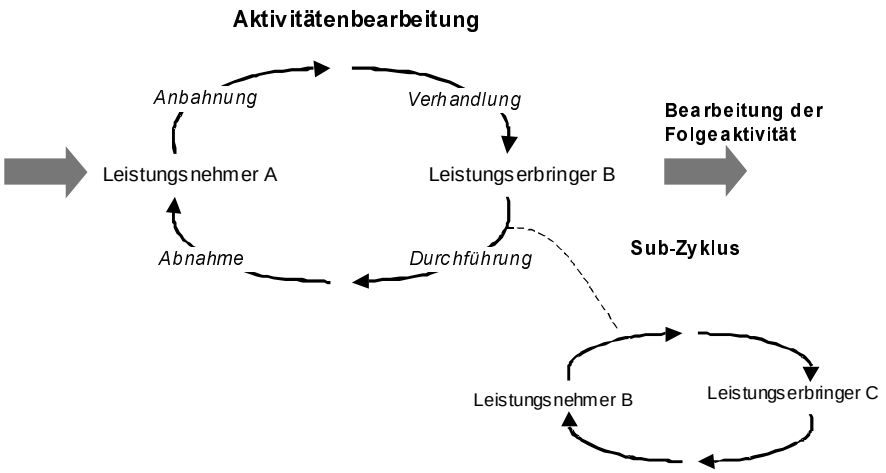


Abbildung 2-7: *Konversationsstrukturorientierte Ablaufmodelle*

schnitt 2.3.3.2) oder Domino Workflow, bei denen formularbasierte Workflows mit einer graphischen Sprache modelliert und anschließend auf eine Menge von Notes-Dokumenten abgebildet werden.

2.2.4 Konversationsstrukturorientierte Koordinationsmodelle

Im Zentrum *konversationsstrukturorientierter Koordinationsmodelle* [FGHW88, Schw96, WiFl86] steht die Kommunikation zwischen den am Vorgang beteiligten Akteuren. Grundlegende Annahme ist, daß es für jede WF-Aktivität zwingend einen Leistungsnehmer gibt, der den eigentlichen Bearbeiter der Aktivität zum Leistungserbringer macht, indem er ihn mit der Durchführung einer Tätigkeit beauftragt [KrZe96]. Aus diesem Grund müssen zur Bearbeitung einer einzelnen Aktivität mehrere Phasen durchlaufen werden, was eine Konversation zwischen den beiden Akteuren erforderlich macht. Diese Konversation wird rechnergestützt im Sinne der Sprechakttheorie [WiFl86] abgewickelt, mit dem Ziel eines geregelten und strukturierten Nachrichtenaustauschs zwischen Leistungsnehmer und -erbringer. Hierfür werden vordefinierte Konversationstypen bzw. -protokolle [Rein93] angeboten. Vorgänge ergeben sich durch Aneinanderreihung mehrerer solcher strukturierter Konversationen.

2.2.4.1 WF-Modellierung und Anwendungsentwicklung

Konversationsstrukturorientierte Modelle gehen davon aus, daß die zur Abwicklung einer WF-Aktivität notwendige Konversation durch einen Zyklus, bestehend aus den vier Phasen „Anbahnung“, „Verhandlung“, „Durchführung“ und „Abnahme“, beschrieben werden kann [MWFF92, Swen93]. Durch die graphische Verkettung mehrerer Zyklen ergeben sich Arbeitsabläufe, wobei einzelne Zyklusphasen durch Sub-Zyklen unterlegt werden können, d.h. eine hierarchische Strukturierung ist möglich (vgl. Abbildung 2-7). Ein WF-Modell besteht aus einer Menge zeitlich geordneter und geschachtelter Konversationen, bei denen die beteiligten Personen abwechselnd die Rolle des Leistungsnehmers oder -erbringers einnehmen. Die einzelnen Phasen einer Konversation und ihre Inhalte müssen auf Modellierungsebene definiert werden. Dabei ist für einzelne Aktivitäten festzulegen, welche Qualitätsmerkmale die zu erstellende Leistung aufweisen soll, welche Formulare zu ihrer Erbringung ausgefüllt werden sollen, welche Zugriffe auf gemeinsame Daten möglich sein sollen oder nach welchen Kriterien sie in der Abnahmephase bewertet werden soll. Hinsichtlich der Anknüpfung von Applikationsbausteinen bestehen zwischen konversationsstrukturorientierten WfMS Unterschiede. Während einige lediglich das Versenden von Nachrichten zur Abwicklung der einzelnen Zyklusphasen unterstützen [FGHW88], bieten andere formularbasierte Schnittstellen an [MWFF92, Swen93].

2.2.4.2 WF-Steuerung und dynamische WF-Änderungen

Die WF-Steuerung erfolgt durch Interpretation des WF-Graphen. Seine Knoten entsprechen Konversationen, d.h. die Durchführung einer Aktivität besteht nicht aus dem Aufruf eines Programms, der Bearbeitung eines Dokumentes oder dem Ausfüllen eines Formulars, sondern zerfällt in mehrere Phasen [KrZe96]: In der *Anbahnungsphase* bestimmt der Leistungsnehmer einen passenden Anbieter,

wozu er eine Kommunikation mit mehreren Akteuren eröffnen kann. Wird ein Leistungserbringer gefunden, beginnt die *Verhandlungsphase*, in der die zu erbringende Leistung näher spezifiziert wird. Anschließend führt der Erbringer die eigentliche Aufgabe werkzeugunterstützt durch (*Durchführungsphase*), wobei er ggf. Sub-Konversationen initiiert. Schließlich übermittelt er das Ergebnis an den Leistungsnehmer, der die Leistung abnimmt und Feedback gibt (*Abnahmephase*).

Dynamische Änderungen der modellierten Anordnungsbeziehungen zwischen „Aktivitätenzyklen" (z.B. Überspringen oder Vertauschen der Reihenfolge) werden nicht oder nur in engen Grenzen unterstützt. Im Grunde wird das Problem starrer Abläufe in Realisierungen dieses Ansatzes [MWFF92] sogar verschärft, da nun auch die Bearbeitung einzelner Aktivitäten ebenfalls nach unveränderlichen Protokollen abgewickelt werden muß. Mit wenigen Ausnahmen [KrZe96, Swen94] müssen in WfMS dieser Kategorie die einzelnen Phasen und ihre Inhalte bereits auf Modellierungsebene vollständig beschrieben werden. Des weiteren wird davon ausgegangen, daß eine Konversation zur Abwicklung einer Aufgabenbearbeitung stets alle vier Phasen umfaßt. Situationsbedingte Modifikationen im Ablauf einer Konversation, z.B. das Überspringen einer Phase, sind nicht möglich.

2.2.4.3 Bewertung

Bisherige Erfahrungen zeigen, daß sich konversationsstrukturorientierte Modelle meist als umständlich und kompliziert erweisen [Schw96]. Bereits die Darstellung einfacher WFs gestaltet sich aufwendig. Hinzu kommt, daß bei vielen Aktivitäten die beschriebenen vier Phasen nicht immer vollständig ausgeprägt sind – die Modellierung unterstellt aber genau dies. Ein weiteres Manko ist die geringe Flexibilität dieser Systeme, die dynamische WF-Änderungen und Änderungen einzelner Konversationen nicht zulassen. Beispiele für Implementierungen sind *ActionWorkflow* [MWFF92], *Regatta* [Swen94], *NEW* [KrZe96], *Obligations* [Bogi95], *SAMPO* [AHL92] und *WAM* [Adam96].

2.3 Diskussion ausgewählter WfMS

2.3.1 Vertreter prozeßorientierter WfMS

Vertreter prozeßorientierter WfMS sind *MQSeries Workflow* [LeRo00], *Staffware* [KaDe96], *LEU* [DGSZ94, Gruh93], *COSA* [JBS97] und – mit Abstrichen – *WorkParty* [Rupi97, RSD97]. In diesem Abschnitt wollen wir am Beispiel der beiden erstgenannten Systeme zeigen, daß dynamische WF-Änderungen in heutigen prozeßorientierten WfMS nicht adäquat unterstützt werden (vgl. Kapitel 1).

2.3.1.1 MQSeries Workflow (FlowMark)

In *MQSeries Workflow* (ehemals: *FlowMark* [LeAl94, Roll96]) wird eine Prozeßvorlage graphisch durch eine Menge von Aktivitäten sowie den zwischen ihnen bestehenden Kontroll- und Datenflußbeziehungen beschrieben. Alternativ dazu kann die skriptbasierte Sprache FDL (Flow Description Language) verwendet werden. Jede Aktivität besitzt einen Ein- und Ausgabedatencontainer, auf die Aktivitätenprogramme lesend bzw. schreibend zugreifen können [Roll96]. *MQSeries Workflow* trennt die Modellierung von Kontroll- und Datenfluß strikt voneinander [LeAl94]. Kontrollflüsse werden mittels gerichteten Kanten zwischen Aktivitäten und durch die Zuordnung von Transitionsbedingungen zu diesen Kanten beschrieben. Eine Transitionsbedingung wertet Daten des Ausgabecontainers der Quellaktivität aus (vgl. Abschnitt 2.2.1.1). Darüber hinaus kann jeder Aktivität eine Ausgangsbedingung zugeordnet werden, die bei ihrer Beendigung evaluiert wird. Ist diese Bedingung nicht erfüllt, wird die Ausführung der Aktivität mit den ursprünglichen Belegungen ihres Eingabedatencontainers wiederholt. Dadurch können, wenn auch in eingeschränkter Form, Schleifen abgebildet werden. Allerdings enthalten WF-Graphen selbst keine Zyklen. Datenflüsse werden über Datenflußkanten modelliert. Führt eine Datenflußkante von einer Aktivität X zu einer Nachfolgeaktivität Y, so können Felder des Ausgabecontainers von X auf Felder des Eingabecontainers von Y abgebildet werden. Eine Überprüfung auf Korrektheit oder Vollständigkeit findet dabei nicht statt.

Die Ausführung der WF-Instanzen einer Prozeßvorlage basiert auf demselben WF-Graphen. Die Konsequenz dieser Verfahrensweise ist, daß Ad-hoc-Änderungen einzelner Instanzen nicht möglich sind. Statt dessen müssen sämtliche Ausnahmen bereits auf Modellierungsebene berücksichtigt werden. Da dies nicht immer möglich ist, lassen sich unvorhersehbare Situationen nur außerhalb des Systems be-

handeln. Dies ist für viele Anwendungen schlichtweg nicht akzeptabel [DRK00]. Zumindest angedacht, aber noch nicht realisiert, sind Mechanismen für das kontrollierte Zurücksetzen von WFs und damit verbunden das Kompensieren von Aktivitätenausführungen [LeRo97, LeRo00, Leym97]. Ein weiteres Manko ist, daß Vorlagenänderungen nicht mehr auf bereits laufende WF-Instanzen angewendet werden können. Sie wirken sich ausschließlich auf zukünftige Instanzen aus, was insbesondere im Fall langandauernder Prozesse eine signifikante Einschränkung bedeutet.

2.3.1.2 Staffware

In *Staffware* [KaDe96, JKMW97] können WF-Modelle, ähnlich wie bei *MQSeries Workflow*, graphisch oder skriptbasiert beschrieben werden. Das verwendete WF-Metamodell umfaßt Konstrukte zur Abbildung sequentieller, paralleler und bedingter Pfade. Parallele Pfade werden über Warteschritte zusammengeführt, wobei der WF erst fortschreiten kann, nachdem alle einmündenden Zweige abgearbeitet worden sind. Verzweigungsentscheidungen basieren auf der Auswertung WF-relevanter Daten, wozu vom Modellierer geeignete Prädikate festzulegen sind. Zur Bearbeitung manueller Aktivitäten dienen Formulare, für deren Implementierung ein Maskeneditor bereitgestellt wird. Aktivitätenprogramme können aber auch auf der Grundlage höherer Programmiersprachen realisiert werden.

Aus denselben Gründen wie bei MQSeries Workflow werden Ad-hoc-Änderungen einzelner Instanzen nicht unterstützt. Für jede Prozeßvorlage kann eine Person bestimmt werden, die diese Vorlage abändern darf. Solche Änderungen können ggf. auch auf bereits aktivierte Instanzen wirken, unabhängig von deren aktuellen Bearbeitungsstatus. Der Grund dafür ist, daß die Ausführung aller Instanzen eines bestimmten WF-Typs auf demselben WF-Graphen basiert. Erscheint dieser Ansatz bei oberflächlicher Betrachtung sinnvoll, erweist er sich in der praktischen Anwendung als problematisch. Abhängig vom aktuellen Bearbeitungsstatus einer WF-Instanz wirken sich Vorlagenänderungen vollständig, überhaupt nicht oder nur teilweise auf die Ausführung dieser Instanz aus. Gerade der letzte Fall ist kritisch. Beispielsweise ist es möglich, daß in einer Vorlage zwei neue Schritte X und Y in verschiedenen Bereichen des WF-Graphen eingefügt werden, wobei Ausgabedaten von X zur Versorgung der Eingabedaten von Y dienen sollen. Obwohl diese Änderung aus Sicht des Modellierers korrekt ist, kann sie bei Anwendung auf Instanzebene zu Problemen führen. Angenommen für eine bestimmte Instanz wirkt sich das Einfügen von X infolge des zu weit fortgeschrittenen Bearbeitungsstatus nicht mehr aus, während Y noch zur Ausführung kommen kann. In diesem Fall können bei der Ausführung von Y Programmfehler infolge der nun fehlenden Aufrufdaten resultieren. Der einzige Ausweg, den Staffware in diesem Zusammenhang bietet, besteht darin, solche Instanzen entweder vorzeitig zu beenden oder im geänderten WF-Modell entsprechende Sonderbehandlungen vorzusehen.

2.3.2 Vertreter dokumentenorientierter WfMS

In diesem Abschnitt werden mit *ProMInanD* [Karb94] und *InConcert* [JKMW97, McSa93] zwei Vertreter dokumentenorientierter WfMS vorgestellt.

2.3.2.1 ProMInanD

ProMInanD basiert auf dem Konzept der elektronischen Umlaufmappe [KaRa90, KaRa91]. Das verwendete Metamodell [Karb94] stellt die für Objektmigrationsmodelle üblichen Beschreibungskonstrukte bereit. Verzweigungsentscheidungen basieren auf Rückgabewerten von Aktivitätenprogrammen, parallele Bearbeitungen können über abhängige Vorgänge realisiert werden. Zusätzlich wird die hierarchische Strukturierung von WF-Modellen durch die Bildung sogenannter Fragmente unterstützt, Schleifen sind nicht abbildbar. *ProMInanD* hebt sich von anderen WfMS durch die angebotenen Mechanismen zur Ausnahmebehandlung ab. Unterstützt werden Ad-hoc-Eingriffe in die Kontrolle einzelner WF-Instanzen, die zu Zustandsänderungen der Instanz oder Modifikationen ihres Ablaufgraphen führen können. Hierzu wird bei der Erzeugung einer WF-Instanz eine Kopie der Originalvorlage angelegt, die während der Ausführung modifiziert werden kann. Entsprechende Anpassungen haben keine Auswirkungen auf andere Instanzen derselben Prozeßvorlage.

Für Anwender bestehen verschiedene Möglichkeiten, um zur Laufzeit vom modellierten WF abzuweichen. Nach der Weiterleitung einer Mappe kann der letzte Bearbeiter die Kontrolle über den Vorgang zurückerlangen, zumindest solange die Folgeaktivität noch nicht gestartet worden ist. Umgekehrt kann

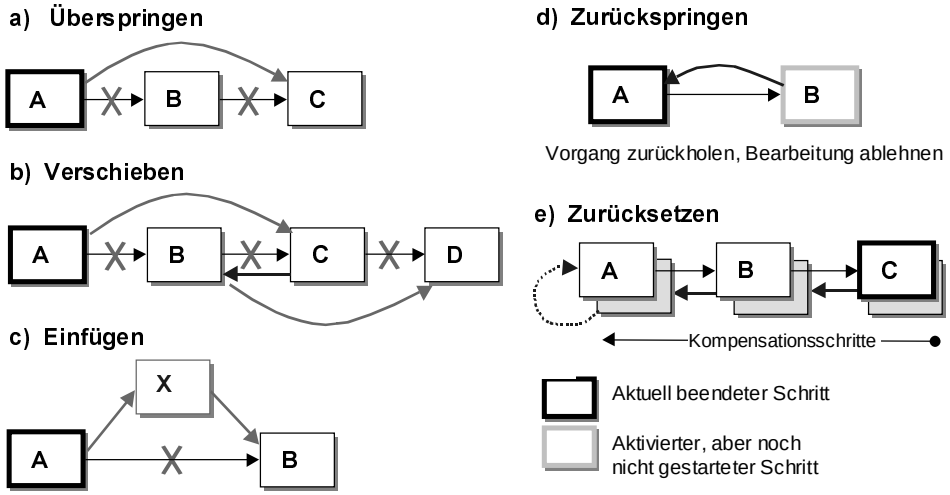


Abbildung 2-8: *Ad-hoc-Änderungen in ProMInanD (vgl. [Karb94])*

der Empfänger einer Mappe die Bearbeitung ablehnen und den Vorgang an den letzten Bearbeiter zurücksenden (Abbildung 2-8 d). Dabei wird der WF vor den Start der letzten manuellen Aktivität zurückgesetzt, wobei evtl. dazwischen liegende automatische Aktivitäten kompensiert werden [Karb94]. Neben verhaltensorientierten Änderungen sind einfache strukturelle Modifikationen möglich [Karb94]. Ein Akteur kann vor der Weitergabe einer Mappe einen neuen Schritt als direkten Nachfolger der gerade beendeten Aktivität einfügen (Abbildung 2-8 c). Des weiteren können Aktivitäten übersprungen (Abbildung 2-8 a) oder ersatzweise ihre Bearbeitung nach „hinten“ verschoben werden (Abbildung 2-8 b). Für den letztgenannten Fall muß eine Aktivität angegeben werden, vor die der übersprungene Schritt im WF-Graphen wieder eingefügt werden soll. Die Durchführung solcher Änderungen gestaltet sich aufgrund der geringen Komplexität des verwendeten Metamodells relativ einfach. Erwähnenswert ist noch, daß sich ein WF-Graph für Anwender als Sequenz von Prozeßschritten bzw. Fragmenten dargestellt. Beispielsweise werden bedingte Verzweigungen vor Anwendern durch geeignete Fragmentierung des WF-Graphen verborgen. Dem entsprechend limitiert sind die bestehenden Änderungsmöglichkeiten. Sie reduzieren sich darauf, in einer Sequenz einzelne Aktivitäten zu löschen bzw. direkt vor oder nach anderen Schritten einzufügen. Komplexere Modifikationen sind nicht möglich. Ein weiteres Problem ist die fehlende Sicherstellung der Datenkonsistenz bei dynamischen WF-Änderungen (vgl. Abschnitt 2.2.2).

Darüber hinaus unterstützt ProMInanD das Rücksetzen eines oder mehrerer Arbeitsschritte [VoEr92]. Im einfachsten Fall können die Auswirkungen einer gerade abgeschlossenen Aktivität wieder rückgängig gemacht werden. Durch Rücksetzen mehrerer Aktivitäten können Akteure, die zuvor in die Vorgangsbearbeitung involviert waren, die Kontrolle zurückerlangen (Abbildung 2-8 e). Zur Realisierung des Rücksetzverfahrens werden *Sagas* [GaSa87] verwendet, ein im Bereich erweiterter Transaktionsmodelle [Elma92, JaKe97] entwickelter Ansatz. Dabei werden in umgekehrter Reihenfolge der normalen Ausführung kompensierende Maßnahmen ergriffen. ProMInanD unterscheidet zwischen *passiver* und *aktiver* Kompensation von Aktivitäten. Bei der passiven Kompensation werden die an Mappendokumenten durchgeführten Änderungen rückgängig gemacht. Aktive Kompensation bedeutet zusätzlich, daß ein vom Entwickler festgelegtes Kompensationsprogramm ausgeführt wird, um externe Auswirkungen der Aktivität zu stornieren. Zuvor durchgeführte strukturelle Änderungen des WF-Instanzgraphen bleiben dagegen erhalten. Während dies in einigen Fällen wünschenswert sein kann, widerspricht diese Vorgehensweise zum Teil der Zielsetzung von ProMInanD, WF-Änderungen zur Behandlung von aufgetretenen Ausnahmen zu unterstützen. Hier wird fälschlicherweise angenommen, daß die Ausnahme, die zur Abänderung des WF-Graphen führte, sowie die Notwendigkeit der in diesem Zusammenhang durchgeführten Änderung auch nach dem Rücksetzen noch gegeben sind.

ProMInanD setzt die Metapher der elektronischen Umlaufmappe gut um [KaRa90, KaRa91]. Darüber hinaus bietet dieses WfMS interessante Möglichkeiten zur Durchführung dynamischer WF-Änderungen. Dies geht zu Lasten der Ausdrucksmächtigkeit des verwendeten Metamodells. Auch die fehlende Sicherstellung der Datenkonsistenz bei WF-Änderungen stellt ein Problem dar. In Bezug auf die

Art und den Umfang möglicher Änderungen bestehen ebenfalls Limitationen. Es werden lediglich einfache Ad-hoc-Änderungen einzelner WF-Instanzen unterstützt. Die angebotene Menge an Änderungsoperationen ist unvollständig, da nicht alle theoretisch möglichen bzw. praktisch relevanten Restrukturierungen unterstützt werden. Mit wenigen Ausnahmen besitzen Akteure, die aktuell nicht an der Bearbeitung beteiligt sind, keine Möglichkeiten, in den Vorgang einzugreifen.

2.3.2.2 InConcert

Das dokumentenorientierte WfMS *InConcert* [JKMW97, McSa93] ist aus dem in [SAM91] vorgestellten Ansatz hervorgegangen. Im Unterschied zu *ProMInanD* wird kein mappenbasiertes Modell, sondern ein Mischform zwischen prozeß- und dokumentenorientiertem Ansatz gewählt. Bei der WF-Definition werden durchzuführende Aufgaben sowie die Reihenfolgebeziehungen zwischen ihnen festgelegt. Schleifen sind nicht modellierbar. Jeder Aktivität wird eine Menge von Dokumenten zugeordnet, auf die während ihrer Bearbeitung zugegriffen werden kann. Durch die Zuordnung von Dokumenttypen ist für einzelne Dokumente geregelt, welche Applikation zu ihrer Bearbeitung ausgeführt werden muß. Der Inhalt der Dokumente ist Sache dieser Anwendungen und wird dem WfMS nicht bekannt gemacht [SAM91]. Die Ausführung einer Instanz erfolgt zentral durch die Interpretation ihres WF-Graphen. Nach der Aktivierung einer Aufgabe können assoziierte Dokumente mit den dafür vorgesehenen Applikationen bearbeitet werden. Falls erforderlich, können auch neue Dokumente hinzugefügt werden. Im Gegensatz zu *ProMInanD* dürfen Kopien desselben Dokumentes nicht nebenläufig geändert werden, was durch einen einfachen Check-In /Check-Out-Mechanismus bewerkstelligt wird.

Bei der Erzeugung einer WF-Instanz wird eine Kopie der Prozeßvorlage angelegt, die zur Laufzeit – ohne Auswirkungen auf andere Instanzen – abgeändert werden kann. Dabei stehen einfache Ad-hoc-Eingriffe wie das Zurückweisen, Wiederholen, Überspringen oder Delegieren einer Aufgabenbearbeitung allen Akteuren offen. Darüber hinaus besitzt der Prozeßverantwortliche (und nur dieser) die Möglichkeit, den Ausführungsgraphen dynamisch zu editieren, etwa durch das Einfügen neuer Aktivitäten oder das Löschen bzw. die Definition von Kontrollkanten. Zur Ausführung neu eingefügter Aktivitäten können Dokumente aus vordefinierten Dokumentvorlagen generiert werden. Schließlich erlaubt es *InConcert* für eine Aktivität – wie einige andere WfMS auch [HNOR95, HsKl96, RVW97] – zur Laufzeit ggf. mehrere Instanzen zu erzeugen, die dann nebenläufig bearbeitet werden. Ad-hoc-WFs können durch die Vorgabe einer „leeren Vorlage“, die zur Laufzeit individuell anpaßbar ist, realisiert werden [SAM91]. Neben den allgemeinen Beschränkungen dokumentorientierter WfMS, bestehen ähnliche Schwächen wie bei *ProMInanD*. Zu nennen sind die fehlenden Möglichkeiten zur Anpassung von Datenflüssen bei der Abänderung der Ablaufstruktur sowie die geringe Ausdrucksmächtigkeit des verwendeten WF-Metamodells. Darüber hinaus gibt es keine Unterstützung für das Zurücksetzen von WFs und die Sicherstellung seiner Konsistenz nach der Durchführung von Änderungen.

2.3.3 Vertreter formularorientierter WfMS

In diesem Abschnitt werden *Lotus Notes* [DiSa97], das als Plattform für die Realisierung formularbasierter WF-Anwendungen dient [ReMo96], sowie das darauf basierende *GroupFlow*-System [NaOt96, HNOR95] vorgestellt. Beide Ansätze gelten als flexibel und verdienen daher eine nähere Betrachtung.

2.3.3.1 Lotus Notes

Lotus Notes [DiSa97] ist ein Groupware-System [Grei88, Grud94], das Dienste für die Speicherung, die Replikation, den Zugriff, die (verteilte) Bearbeitung und das Versenden von semi-strukturierten Dokumenten bereitstellt. Obwohl *Notes* kein „echtes“ WfMS ist, wird es häufig zur Realisierung einfacher WF-Anwendungen eingesetzt [ReMo96, Hens00a]. Hierfür werden die notwendige Kommunikationsinfrastruktur und verschiedene Grundelemente (vgl. Abbildung 2-9) bereitgestellt. Darauf basierend lassen sich – entsprechend der in Abschnitt 2.2.3 skizzierten Vorgehensweise – formularbasierte Workflows realisieren.

Die wesentlichen Bausteine einer *Notes*-Anwendung sind Dokumente, Ansichten und Masken. Ein Dokument umfaßt eine Menge von Feldern, die dynamisch erweiterbar ist. Über Ansichten können ausgewählte Dokumente aufgelistet werden, die dann mit benutzerdefinierten Masken bearbeitbar sind. Welche Maske in welchem Status zum Öffnen des Dokumentes verwendet werden soll, ist über

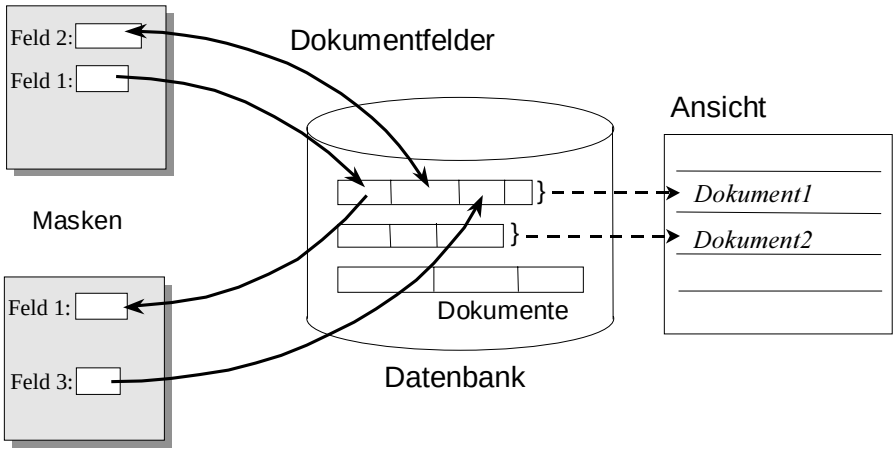


Abbildung 2-9: *Grundelemente einer Notes-Anwendung*

Formeln, abhängig von aktuellen Feldwerten, steuerbar. Für die Maskendefinition wird ein graphischer Editor bereitgestellt. Die Beschreibung einer Maske kann neben Feldern auch statische Gestaltungselemente (z.B. Text) und Interaktionsbausteine (z.B. Schaltflächen) umfassen, die abhängig von Feldwerten ein- bzw. ausblendbar sind. Mit Ereignissen wie „Schließen einer Maske“ oder „Aktivieren einer Schaltfläche" lassen sich benutzerdefinierte Aktionen (z.B. Berechnen von Feldwerten, Versenden von Dokumenten) verknüpfen. Zu ihrer Programmierung kann auf die bereits erwähnte Formelsprache zurückgegriffen werden oder alternativ auf eine Skriptsprache, welche die von höheren Programmiersprachen her bekannten Konstrukte anbietet. Darüber hinaus können Makros definiert werden, um programmgesteuert Änderungen an ausgewählten Dokumenten vorzunehmen [DiSa97].

Mit diesen Mitteln lassen sich formularbasierte WFs realisieren, wobei zwei Ansätze möglich sind [Hens00a]. Zum einen kann, wie in Abschnitt 2.2.3 skizziert, die Aufrufreihenfolge von Masken durch die Aufnahme zusätzlicher Steuer- und Statusfelder in die Dokumente koordiniert werden. Das Weiterschalten im WF erfolgt dann durch die Ausführung von Formeln, die mit Ereignissen wie „Schließen einer Maske“ verknüpft werden (*Pull-Modell*). Zum anderen sind mit dem in Notes integrierten Mailsystem nachrichtenbasierte WFs realisierbar, bei denen Vorgangsdokumente von Bearbeiter zu Bearbeiter weiter geleitet werden (*Push-Modell*). Prinzipiell sind auch einfache Ad-hoc-WFs möglich, wobei der Prozeßstatus für Dritte nicht einsehbar ist. Durch die implizite WF-Definition sind *Notes*-basierte Anwendungen nur mit großem Aufwand wartbar. Die Frage nach der Propagierung von „Vorlagenänderungen“ auf aktive Instanzen stellt sich infolge des impliziten Beschreibungsansatzes nicht. Hinzu kommen Mängel wie die fehlende Integration von Fremdapplikationen, die fehlende Transaktionsunterstützung und die fehlende Synchronisation nebenläufiger Dokumentzugriffe.

2.3.3.2 GroupFlow

GroupFlow (auch: *Pavone Espresso*) basiert auf Notes und nutzt die von diesem System angebotenen Möglichkeiten zur Realisierung formularbasierter WFs [NaOt96, OtNa97]. Um die WF-Implementierung weniger kryptisch zu gestalten, wird für den Entwickler ein graphischer Editor bereitgestellt mit dem er Abläufe modellieren und verschiedenen Analysen unterziehen kann [HNOR95]. Aus der erstellten WF-Definition generiert GroupFlow eine Notes-Anwendung mit entsprechenden Dokumenten, Ansichten und Masken. Die WF-Steuerung (Ermittlung zuständiger Bearbeiter, Maskenaktivierung) erfolgt in *Notes*, d.h. *GroupFlow* verfügt über keine eigene WF-Engine. Die bereits genannten Einschränkungen, etwa im Hinblick auf parallele Bearbeitungen, bleiben deshalb bestehen.

GroupFlow erhebt für sich den Anspruch, ein breites Spektrum an Prozessen – von gut strukturiert bis spontan – zu unterstützen. Ad-hoc-WFs sind auf der Grundlage des von *Notes* angebotenen Mailsystems möglich. Für die Unterstützung semi-strukturierter WFs bietet *GroupFlow* unterschiedliche Konzepte an [HNOR95, NaOt96]: Durch „offene Gruppenprozesse“ sollen unstrukturierte Gruppentätigkeiten, wie das Editieren eines Dokumentes durch mehrere Personen, innerhalb eines ansonsten vordefinierten WF ermöglicht werden. Sie werden mit Aktivitäten verknüpft, besitzen also einen definierten Ein- und Ausgang. Dasselbe gilt für „kontrollierte Gruppenprozesse“, die der Unterstützung

von Aufgaben dienen, für deren Bearbeitung eine vorgegebene Anzahl von Personen erforderlich ist (z.B. Abzeichnung eines Dokumentes durch mehrere Personen). Die konkrete Zusammensetzung der Gruppe erfolgt zur Laufzeit. Ad-hoc-Änderungen werden nur begrenzt unterstützt. Ein Bearbeiter kann Rückfragen stellen, eine weitere Person in die Bearbeitung einer Aktivität einbeziehen, (neue) Aufgaben an einen anderen Akteur delegieren, den Vorgang an den vorherigen Bearbeiter zurückgeben oder ihn abbrechen. Bei den meisten dieser Änderungen handelt es sich um einfache Kontrollflußeingriffe, komplexe strukturelle Änderungen sind nicht möglich. Insgesamt erleichtert *GroupFlow* zwar die Implementierung formularbasierter WF-Anwendungen mit Notes, weist ansonsten aber dieselben Nachteile auf. Auch fortschrittliche Funktionen, wie das Zurücksetzen von WFs oder die Möglichkeit der Propagierung von Änderungen einer Prozeßvorlage auf laufende Instanzen, fehlen.

2.3.4 Vertreter konversationsstrukturorientierter WfMS

In diesem Abschnitt stellen wir mit den Systemen *ActionWorkflow* [MWFF92], *NEW* [KrZe96] und *Regatta* [Swen94] Vertreter konversationsstrukturorientierter WfMS vor.

2.3.4.1 The Coordinator, ActionWorkflow und NEW

Eines der ersten konversationsstrukturorientierten System war *The Coordinator* [FGHW88, WiFl86]. Der Verlauf einer Konversation erfolgt hier auf Grundlage eines endlichen Automaten, bei dessen Zustandsübergängen es zum Versenden von Nachrichten (z.B. Anfrage, Zusage, Ablehnung) zwischen Akteuren kommt. Andere Formen des Informationsaustausches gibt es nicht. Es wird angenommen, daß Konversationen immer durch dasselbe Protokoll beschreibbar sind. Dem entsprechend kann ihr Ablauf zur Laufzeit nicht modifiziert werden. Hieraus hervorgegangen ist das WfMS *ActionWorkflow* [MWFF92], das nicht nur einzelne Konversationen regelt, sondern auch deren zeitliche Anordnung in einem Aktivitätennetz erlaubt. Zur Laufzeit wird die Ausführung von Konversationen koordiniert und überwacht. Eine Konversation selbst wird wieder durch einen Zyklus, bestehend aus den vier Phasen Anbahnung, Verhandlung, Durchführung und Abnahme, beschrieben. Die Bearbeitung der einzelnen Phasen erfolgt formularbasiert. Darüber hinaus können einzelne Phasen in gleicher Weise durch Subzyklen unterlegt sein. Wie bei *Coordinator* sind Änderungen im Ablauf einer Konversation, z.B. das Auslassen einer Phase, nicht möglich. Ebensowenig werden dynamische WF-Änderungen unterstützt.

Einen Vorstoß zur Unterstützung semi-strukturierter Abläufe unternimmt der in [KrZe96] vorgestellte Ansatz des *Negotiation Enabled Workflow* (*NEW*). Ein Arbeitsablauf wird hier durch eine Sequenz von Aktivitäten – den sog. „Kunden-Lieferanten-Beziehungen (KLB)“ – beschrieben. Eine einzelne KLB legt die Kooperation zwischen einem Leistungsnehmer (Kunde) und einem Leistungserbringer (Lieferant) fest und wird in die beschriebenen vier Phasen unterteilt. Im Gegensatz zu *ActionWorkflow* müssen der genaue Ablauf und die verschiedenen Phasen einer KLB nicht notwendigerweise im voraus modelliert werden, sondern können auch noch während der WF-Ausführung festgelegt werden. Beispielsweise kann ein Akteur, der in einer laufenden KLB die Rolle des Leistungsnehmers (Kunden) einnimmt, diese Konversation abbrechen und eine neue KLB mit einem anderen Leistungserbringer eingehen. Ersteres entspricht logisch dem Abbrechen einer Aktivität, letzteres dem Einfügen einer neuen Aktivität in den WF. Da von NEW lediglich sequentielle Abläufe unterstützt werden, stellen solche Ad-hoc-Eingriffe konzeptuell kein Problem dar, zumal Datenflußaspekte ausgeklammert werden. Auch die Struktur einzelner KLB kann dynamisch abgeändert oder definiert werden. Dies erfolgt in der Verhandlungsphase der KLB durch die beteiligten Akteure. Die Möglichkeit der Überlappung von WF-Modellierung und -Ausführung bedeutet gegenüber *ActionWorkflow* einen Fortschritt. Die geringe Ausdrucksmächtigkeit des verwendeten WF-Metamodells läßt eine Anwendung auf ein breiteres Prozeßspektrum allerdings nicht zu. Welche Änderungsoperationen konkret unterstützt werden, wie dynamische Änderungen durchgeführt werden, welche Probleme dabei auftreten können und wie diese Probleme behandelt werden geht aus den Veröffentlichungen zu NEW nicht hervor.

2.3.4.2 Regatta

Ein interessanter Ansatz ist aus dem Projekt *Regatta* [Swen93, Swen94, Swen94a] entstanden. Er stellt eine Kombination aus konversationsstuktur- und prozeßorientiertem Koordinationsmodell dar. Arbeitsabläufe (sog. *Plans*) werden mit einer graphischen Beschreibungssprache modelliert, die Kon-

strukte zur Festlegung serieller, paralleler und bedingter Aufgabenbearbeitungen umfaßt. Schleifen können nicht abgebildet werden. Wie bei *ActionWorkflow* werden einzelne Aktivitäten (sog. *Stages*) durch eine Konversation zwischen zwei Akteuren, auf der Grundlage formularbasierter Benutzerschnittstellen, abgebildet. Regatta unterstützt in begrenztem Umfang auch dynamische WF-Änderungen. Ebenso wie bei *InConcert* und *ProMInanD* wird bei der Erzeugung einer WF-Instanz eine Kopie des WF-Graphen angelegt, die instanzspezifisch modifiziert werden kann. Akteure können vor Durchführung einer Aktivität einen neuen Schritt hinzuzufügen, z.B. um eine Rückfrage an eine andere Person zu stellen. Die für den Normalfall anstehende Aufgabe wird dann solange blockiert, bis der neu eingefügte Schritt beendet ist. Interessant ist die Idee, häufig durchzuführende Änderungen eines Ablaufplans in Skripten vorzudefinieren, die während der Planausführung entweder vom Prozeßverantwortlichen oder automatisch durch das System gestartet werden. Dieser reflektive Ansatz setzt voraus, daß die notwendigen Anpassungen zur Modellierungszeit bekannt sind, was in der Mehrzahl der Fälle nicht zutrifft. Des weiteren können im Kontext derselben WF-Instanz ggf. mehrere Änderungsskripte zu Anpassungen des Plans führen. Inwieweit ein vordefiniertes Skript auf einen zuvor geänderten Plan noch anwendbar ist bleibt offen. Wichtige Forderungen an Änderungsoperationen, wie Korrektheit oder Vollständigkeit, werden von *Regatta* nicht aufgegriffen. Welche Änderungen konkret unterstützt werden, auf welche Art und Weise ihre Durchführung erfolgt und welche Voraussetzungen dabei zu beachten sind, wird in Veröffentlichungen dieses Projekts ebenfalls nicht erörtert.

2.4 Standards

Um eine breite Einsetzbarkeit für WfMS zu erzielen, ist die Definition einheitlicher Schnittstellen für die Anwendungsentwicklung unerläßlich. Wesentliche Arbeiten hierzu sind bisher von der *Workflow Management Coalition* (WfMC) ausgegangen. Dabei handelt es sich um ein 1993 gegründetes Konsortium, dem alle namhaften WfMS-Anbieter und zahlreiche Anwender angehören. Hauptziel ist die Förderung und Verbreitung des WF-Gedankens. Zu diesem Zweck wurden zunächst ein allgemeingültiges Referenzmodell [WFMC95] definiert und eine einheitliche Terminologie [WFMC96] geschaffen. Das Referenzmodell (vgl. Abbildung 2-10) beschreibt die grundlegenden Komponenten eines WfMS und legt standardisierte Schnittstellen zwischen ihnen in Form von APIs (Application Programming Interfaces) und Austauschformaten fest. Ziel ist die Schaffung einer einheitlichen Basis für die Anknüpfung von Anwendungskomponenten in den WF und die Implementierung von Anwendungen zur Arbeitslistenverwaltung. Darüber hinaus wird eine Verbesserung der Interoperabilität zwischen WfMS sowie zwischen WfMS und Werkzeugen verschiedener Hersteller angestrebt.

Die *Schnittstelle 1* des Referenzmodells definiert ein Format für den Austausch von Prozeßvorlagen zwischen Modellierungswerkzeugen und WfMS [WFMC98]. Dadurch soll die Übernahme von Modellen aus Modellierungswerkzeugen [Sche96] und ihre Weiterverarbeitung mit den Komponenten des WfMS ermöglicht werden. Die Kommunikation zwischen *WF-Klientenanwendungen* (kurz: WF-Klient) und WfMS soll durch die *Schnittstelle 2* vereinheitlicht werden [WFMC98a, WFMC96a]. Aufgabe eines WF-Klienten ist es, einem Benutzer anstehende Aktivitäten in seiner Arbeitsliste anzuzeigen und ihm geeignete Manipulationsmöglichkeiten (z.B. Starten oder Beenden) darauf einzuräumen. Um die Implementierung von WF-Klienten unabhängig von einem konkreten WfMS vornehmen zu können, wurde von der WfMC mit WAPI (=Workflow Application Programming Interface) eine Standard-Programmierschnittstelle geschaffen. Sie umfaßt Funktionen für den Verbindungsaufbau/-abbau, für die Abfrage von Metadaten zu WF-Vorlagen, für die Kontrolle von WF- und Aktivitäteninstanzen (Erzeugen, Starten, Beenden usw.), für die Verwaltung von Arbeitslisten und für das WF-Monitoring. Für den Aufruf spezieller Applikationen (z.B. Email-Dienste) durch das WfMS gibt es ebenfalls eine Standardschnittstelle (*Schnittstelle 3*), die mittlerweile mit der *Schnittstelle 2* verschmolzen worden ist [WFMC98a]. Die Verbesserung der Interoperabilität zwischen WfMS verschiedener Hersteller soll durch die *Schnittstelle 4* verwirklicht werden [WFMC96b, WFMC98b]. Hierzu wurden verschiedene Interoperabilitätsstufen für WfMS festgelegt [WFMC96b], die von einer Koexistenz auf derselben Hardware- bzw. Betriebssystem-Plattform (Level 1), über den Austausch von Modell- und Instanzdaten (Level 5) bis hin zu einem einheitlichen Look & Feel (Level 7) reichen. *Schnittstelle 5* definiert ein Standard-Austauschformat und -API, um auf die während einer WF-Ausführung protokollierten Daten (sog. *Audit File*) zugreifen zu können [WFMC98c].

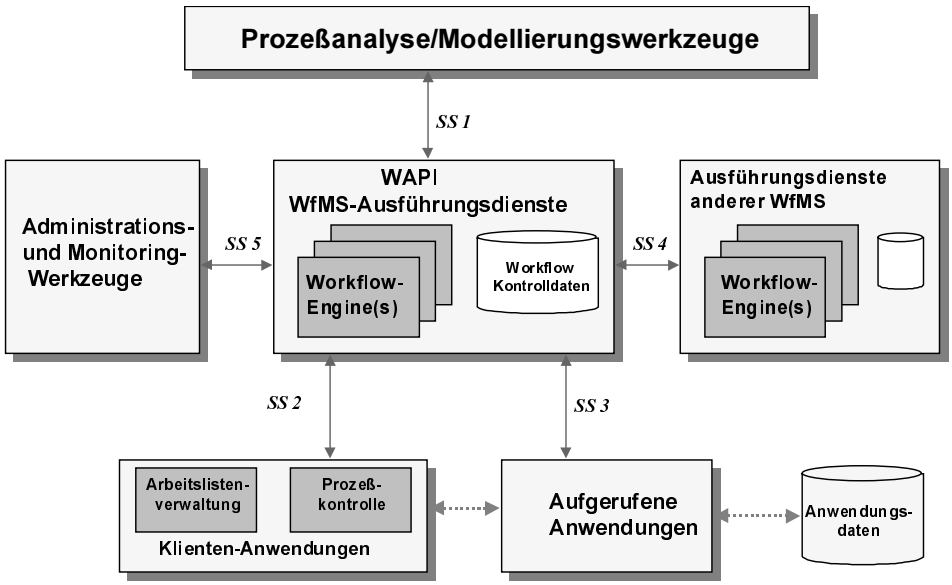


Abbildung 2-10: *Referenzmodell der WfMC (Quelle: [WFMC96])*

Neben den Arbeiten der WfMC gibt es einige weitere Standardisierungsbestrebungen (für einen Überblick siehe [Hast99]). Die SWAP-Gruppe (*Simple Workflow Access Protocol*) hat es sich zum Ziel gesetzt, WfMS-Interoperabilität auf Basis des Internets zu erreichen und nennt HTTP als mögliches Protokoll zur Kommunikation zwischen verschiedenen WfMS. Ein OMG-Protokoll zur Ansteuerung von WfMS in CORBA-Umgebungen ist *jFlow* (*Joint Workflow Management Facility Protocol*). In Bezug auf dynamische WF-Änderungen sind derzeit noch keine Standards absehbar. Ihre Unterstützung würde Erweiterungen sämtlicher Schnittstellen des WfMC-Referenzmodells notwendig machen.

2.5 Zusammenfassung

Heutige WfMS stehen noch am Anfang ihrer Entwicklung. Sie gehen von unterschiedlichen, oftmals falschen Annahmen in Bezug auf die zu unterstützenden Arbeitsabläufe aus, so daß ihre Einsatzgebiete entsprechend begrenzt sind. Bislang gibt es kaum fundierte Arbeiten über den unternehmensweiten Einsatz von WfMS und den sich hieraus stellenden Anforderungen. Ein Mangel bisheriger Studien ist die Beschränkung auf wenige Fragestellungen, die weitgehend getrennt voneinander und häufig nur sehr oberflächlich untersucht werden. Bei der Beschreibung der verschiedenen Systemkategorien wurde deshalb nicht nur auf Fragestellungen dynamischer WF-Änderungen eingegangen, sondern auch auf Aspekte wie Ausdrucksmächtigkeit des WF-Metamodells, Korrektheit der WF-Modelle oder Wartbarkeit der resultierenden Anwendungssysteme. Der Grund dafür ist, daß die Bewertung der Flexibilität von WfMS in keinem Fall isoliert von diesen Aspekten erfolgen sollte.

Bisherige Erfahrungen zeigen, daß konversationsstrukturorientierte WF-Modelle für die Unterstützung komplexer (WF-)Anwendungen eher ungeeignet sind und wohl auch in Zukunft keine Rolle spielen werden. Auf eine weitere Diskussion dieses Ansatzes wird deshalb im folgenden verzichtet. Darüber hinaus zeigen Erfahrungsberichte und der Markterfolg von Systemen wie *Notes* oder *InConcert*, daß Anwender bei kleineren, überschaubaren Abläufen die dokumenten- oder formularorientierte Vorgehensweise bevorzugen. Zwar können Formular- oder Dokumentenflüsse auch mit prozeßorientierten WfMS realisiert werden, allerdings ist hier der zu treibende Initialaufwand wesentlich höher.

Prozeßorientierte WfMS weisen derzeit gegenüber dokumenten- und formularorientierten Ansätzen ein Manko auf. Sie erzwingen die Ausführung von WF-Instanzen starr, d.h. von dem einmal modellierten WF kann zur Laufzeit nicht abgewichen werden. Im Hinblick auf WF-Änderungen sind dokumentenorientierte WfMS – zumindest was Ad-hoc-Änderungen einzelner WF-Instanzen betrifft – derzeit am flexibelsten. Die Mehrzahl dieser Systeme basiert auf dem Objektmigrationsansatz, dem ein simples WF-Metamodell zugrundeliegt. Dem entsprechend einfach gestalten sich WF-Änderungen in Systemen dieser Kategorie. Für einen großflächigen Einsatz kommen sie aus mehreren Gründen nicht in Frage. Die bereitgestellten WF-Metamodelle sind nicht ausdrucksmächtig genug, um ein

breites Spektrum an Prozessen adäquat unterstützen zu können. Hinzu kommt, daß sich die Validation von WF-Modellen aufgrund der impliziten Beschreibung einzelner WF-Aspekte (z.B. Datenflüsse) äußerst schwierig gestaltet. So ist systemseitig nicht feststellbar, welche Anpassungen am Datenfluß und an den Anwendungskomponenten vorgenommen werden müssen, um nach WF-Änderungen einen fehlerfreien Ablauf zu gewährleisten. Mit zunehmender Komplexität der WFs führt dies zu kaum mehr wartbaren Anwendungssystemen, die nur mit großem Aufwand an geänderte Umweltbedingungen anpaßbar wären. Spätestens dann ist der Übergang zu prozeßorientierten WfMS wünschenswert.

Auch wenn prozeßorientierte WfMS derzeit noch limitiert sind, besitzen sie technologisch das größte Potential im Hinblick auf die Modellierung, die Realisierung und die Wartung vorgangsorientierter Anwendungssysteme. Wie in Kapitel 1 diskutiert, eröffnet die explizite, formale Beschreibung von WF-Modellen die Chance, das Systemverhalten noch vor Inbetriebnahme des Anwendungssystems bzw. vor dessen eigentlicher Implementierung zu evaluieren. Dies ist gerade im Hinblick auf den Einsatz in verteilten Umgebungen [Dada96, GrRe93], die man heute immer häufiger vorfindet, ein sehr wichtiger Aspekt. Darüber hinaus trägt die Modellierung von Datenflüssen dazu bei, eine weitgehende Unabhängigkeit von WF-Modellen und Anwendungskomponenten zu erzielen. Spätere Änderungen in den Geschäftsprozessen und daraus resultierende Anpassungen in den Anwendungssystemen lassen sich somit sehr viel einfacher durchführen. Im Gegensatz zu den anderen Systemkategorien sind prozeßorientierte WfMS zudem in der Lage, die vorhandenen, oftmals heterogenen Anwendungssysteme zu integrieren und so eine gemeinsame Basis für die Verwaltung von Arbeitsabläufen zu schaffen. Schließlich eröffnet die Steuerung von Kontroll- und Datenflüssen durch das WfMS die Möglichkeit, die Entwicklung vorgangsorientierter Anwendungen durch die Bereitstellung entsprechender Systemdienste – z.B. für die Fehler- und Ausnahmebehandlung – drastisch zu vereinfachen.

Aus den genannten Gründen wandelt sich die oftmals diskutierte Fragestellung, „ob prozeßorientierte WfMS überhaupt geeignet sind, um Arbeitsabläufe *flexibel* zu unterstützen oder ob sie lediglich für starre Arbeitsabläufe einsetzbar sind“ zur Aussage „es gibt keine ernsthafte Alternative zu prozeßorientierten Koordinationsmodellen, wenn es um die Realisierung anspruchsvoller, vorgangsorientierter Anwendungssysteme geht“ und „wie können wir sicherstellen, daß diese Systeme den sich aus der Praxis stellenden Anforderungen, z.B. im Hinblick auf dynamische Ablaufänderungen, gerecht werden?“ Aus diesem Grund wird in dieser Arbeit ein prozeßorientierter Ansatz verfolgt, mit dem Ziel ein theoretisch fundiertes Modell für die Erstellung, Ausführung und dynamische Änderung von WF-Modellen zu gewinnen.

Teil II

Theoretische Grundlagen des ADEPT-Modells

3 Das ADEPT-Basismodell

Eine zentrale Aufgabe bei der Realisierung flexibler, prozeßorientierter Anwendungssysteme ist die Modellbildung. Arbeitsabläufe sollen in einer Form beschrieben werden, die es ermöglicht, die verschiedenen WF-Aspekte zu präzisieren, ohne die eigentliche Implementierung der WF-Aktivitäten vorwegzunehmen. Ziel dabei ist es, die Struktur und das Verhalten von WF-Modellen zu einem möglichst frühen Zeitpunkt studieren und validieren zu können. Im Gegensatz zu Modellierungsansätzen aus dem Bereich des Requirements Engineering – hier steht die Ermittlung, Formulierung und Analyse von Anforderungen an Software-gestützte Systeme im Vordergrund [Part98] – sind bei WfMS auch Laufzeitaspekte, wie die Parameterversorgung von Aktivitätenprogrammen, die Unterstützung dynamischer WF-Modifikationen und die Behandlung von Fehlern und Ausnahmen, von Bedeutung.

Voraussetzung für alles bildet die Bereitstellung eines formalen WF-Metamodells, das es erlaubt, die betrieblichen Abläufe mit ihrer üblichen Variabilität und ihren Ausnahmeregeln angemessen abzubilden. Dazu muß ein vernünftiger Kompromiß zwischen der Ausdrucksmächtigkeit dieses Metamodells einerseits und seiner praktischen Verwendbarkeit andererseits gefunden werden. Arbeitsabläufe müssen kompakt und auf semantisch hoher Ebene beschreibbar sein, so daß die resultierenden Modelle nicht nur für den Entwickler, sondern auch für den Endanwender verständlich sind, zumindest für die Anwender, die zur Laufzeit Ad-hoc-Änderungen vornehmen können sollen. Ebenso wichtig ist es, daß das WF-Metamodell geeignete Modelleigenschaften definiert, um Aussagen zur Korrektheit und zum dynamischen Verhalten von WF-Modellen und -Instanzen treffen zu können. Wichtig dabei ist, daß die geforderten Korrektheitseigenschaften sowohl bei der Erstellung von WF-Modellen als auch bei der dynamischen Änderung laufender WF-Instanzen beachtet werden. Dies setzt ein durchgängiges formales Modell für die Modellierung, Ausführung und Änderung von Workflows voraus.

Im Teil II dieser Arbeit wird mit dem *ADEPT-Basismodell* ein solches WF-Metamodell vorgestellt. Im Zentrum steht dabei, wie für prozeßorientierte Koordinationsmodelle (vgl. Kapitel 2) üblich, die Idee, die Beschreibung der verschiedenen Aspekte eines WF von der Implementierung seiner Arbeitsschritte zu trennen. Das ADEPT-Basismodell bildet die formale Grundlage für die Spezifikation und die Ausführung von Workflows, ebenso wie für die dynamische Abänderung von WF-Instanzen im laufenden Betrieb. Oberstes Ziel dabei ist es, ein korrektes und zuverlässiges Systemverhalten zu gewährleisten. Wir schränken uns im folgenden auf die im Kontext dieser Arbeit relevanten Beschreibungsaspekte ein, die im wesentlichen die funktionale Sicht des WF, d.h. die WF-Aktivitäten und deren Eigenschaften, sowie die zwischen den WF-Aktivitäten bestehenden Kontroll- und Datenabhängigkeiten festlegen. Wichtige Erweiterungen dieses Metamodells, etwa im Hinblick auf die Unterstützung temporaler oder organisatorischer Aspekte, die in begleitenden Diplomarbeiten vorgenommen wurden [Grim97, Kubi98], werden aus Platzgründen ausgeklammert. Ebenfalls nicht im Detail behandelt werden Fraugestellungen, die sich im Zusammenhang mit der Anknüpfung von Anwendungskomponenten, der Einbeziehung von Datenschutz- und Datensicherheitsaspekten, der Unterstützung globaler Geschäftsregeln oder der transaktionalen Ausführung von Workflows ergeben.

In Abschnitt 3.1 werden die wichtigsten Zielsetzungen beim Entwurf des ADEPT-Basismodells zusammengefaßt. Der Abschnitt 3.2 geht auf grundlegende Aspekte der Beschreibung und Implementierung von WF-Aktivitäten ein. Anschließend werden in Abschnitt 3.3 die für die Kontrollflußmodellierung bereitgestellten Elemente informell eingeführt, bevor sie in den Abschnitten 3.4, 3.5 und 3.6 einer formalen Betrachtung unterzogen werden. Dabei werden strukturelle Eigenschaften von Kontrollflußschemata näher beleuchtet. In Abschnitt 3.7 werden Erweiterungen diskutiert, die zur Unterstützung hierarchischer Workflows notwendig sind. Kapitel 4 geht dann auf die operationale Semantik des ADEPT-Basismodells und auf das daraus ableitbare dynamische WF-Verhalten ein.

3.1 Entwurfsziele

Der Entwurf des ADEPT-Basismodells orientiert sich an den folgenden Zielsetzungen:

1. **Ausdrucksmächtigkeit:** Ein WF-Metamodell muß ausdrucksstark genug sein, um ein breites Spektrum an Arbeitsprozessen abbilden zu können. Einfache Prozesse von kurzer Dauer und mit nur wenigen Beteiligten müssen ebenso unterstützt werden, wie langlaufende Abläufe mit vielen beteiligten Akteuren. Auch hinsichtlich der Komplexität der zu unterstützenden Prozesse bestehen

große Unterschiede. Im allgemeinen müssen Tätigkeiten sequentiell, bedingt, nebenläufig oder auch iterativ (zyklisch) bearbeitet werden können. Dabei sind zwischen den Bearbeitungsschritten eines Prozesses zum Teil komplexe Datenabhängigkeiten zu berücksichtigen. Ebenfalls charakteristisch für viele Anwendungsbereiche ist, daß es neben einfach strukturierten Abläufen mit einigen wenigen, planbaren Ausnahmen auch zahlreiche Abläufe gibt, bei denen aufgrund ihrer Variabilität nicht alle möglichen Ausführungspfade vormodelliert werden können (vgl. Kapitel 2). Die angemessene Unterstützung solcher, im voraus nur ungefähr beschreibbarer Workflows, ist aber von zentraler Bedeutung, wenn WF-Technologie auf breiterer Basis Fuß fassen soll.

2. **Strukturierbarkeit:** Aus der Modellierung realer Arbeitsabläufe können komplexe und umfangreiche WF-Modelle resultieren [BeDa96, JST97, RHD98]. Aus diesem Grund sollte ein WF-Metamodell geeignete Mittel zur Strukturierung dieser WF-Modelle bereitstellen. Neben der Möglichkeit der hierarchischen Strukturierung von Workflows – ähnlich wie man es von SA/RT-Ansätzen her kennt [Biec96] – sollten weitere Strukturierungshilfen angeboten werden. Beispielsweise kann es sinnvoll sein, unterschiedliche Abstraktionsstufen einzuräumen, etwa durch die Möglichkeit der Verfeinerung oder Vergröberung der Modelle. Darüber hinaus sollten unterschiedliche Modellaspekte (z.B. Kontroll- und Datenfluß) oder Ablaufsichten (z.B. Normalablauf vs. Ausnahmefall) getrennt voneinander beschreibbar sein [Ober96]. Erst durch die Bereitstellung solcher Strukturierungshilfen können Anwendern geeignete Sichten und Manipulationsmöglichkeiten auf WF-Modelle und -Instanzen eingeräumt werden. Ein weiterer wichtiger Aspekt stellt in diesem Kontext die Wiederverwendung von Prozeß- und Aktivitätenvorlagen dar.

3. **Ausführbarkeit:** Ein WF-Metamodell muß die Repräsentation von WF-Modellen und -Instanzen gleichermaßen gut unterstützen. Eine zentrale Anforderung betrifft die Ausführbarkeit der beschreibbaren WF-Modelle. Hierzu muß für die Beschreibungskonstrukte des Metamodells eine klare und eindeutige Ausführungssemantik definiert werden, die regelt unter welchen Bedingungen die Aktivitäten eines WF aktiviert (deaktiviert) werden sollen und welche Zustandsänderungen sich bei ihrer Beendigung ergeben. Des weiteren müssen auf Instanzebene zusätzliche Informationen zum WF-Status verwaltet werden, aus denen hervorgehen sollte, welche Aktivitäten aktuell ausführbar sind und welche nicht, und wie sich der bisherige Verlauf des WF darstellt. Letzteres ist im Kontext dynamischer WF-Änderungen wichtig, da ansonsten aufwendige Analysen notwendig werden können, um festzustellen, ob eine beabsichtige Änderung im aktuellen Zustand des WF korrekt anwendbar ist oder nicht. Die explizite Beschreibung von WF-Zuständen und die Festlegung einer operationalen Semantik bilden auch die Voraussetzung für die formale Analyse des dynamischen WF-Verhaltens.

4. **Formalisierbarkeit:** Ausgehend von einer anwendungsnahen, abstrakten Darstellung eines Arbeitsablaufs muß die schrittweise Formalisierung von WF-Modellen unterstützt werden. Die formale Beschreibung der Prozeßlogik ist unabdingbar, will man zu präzisen Aussagen über das zukünftige Systemverhalten gelangen. Besonders wichtig ist dabei, daß potentielle Fehler im Entwurf eines WF-Modells bereits zur Modellierungszeit, d.h. vor der Implementierung der WF-Aktivitäten, durch die Anwendung geeigneter Konstruktionsprinzipien und Fehleranalysen ausgeschlossen werden können. Dazu muß das WF-Metamodell überprüfbare Korrektheits- und Konsistenzeigenschaften definieren, die sowohl Aussagen zu strukturellen Eigenschaften der WF-Modelle (z.B. korrekter Versorgung der Aufrufparameter eines Aktivitätenprogramms, Zyklenfreiheit) als auch zu ihrem dynamischen Verhaltens (z.B. Verklemmungsfreiheit) gestatten.

5. **Analysier-** und **Validierbarkeit:** Um präzise Aussagen zu den statischen und dynamische Eigenschaften eines WF-Modells treffen zu können, müssen (effiziente) Verfahren für ihre Analyse und Validation bereitgestellt werden. Diese Verfahren müssen sowohl bei der Erstellung als auch bei der (dynamischen) Abänderung der WF-Modelle anwendbar sein. Dabei sollte ein vernünftiger Kompromiß zwischen der Ausdrucksmächtigkeit des Metamodells sowie dem Umfang und der Komplexität der notwendigen Korrektheitsanalysen gefunden werden. Idealerweise sollten bestimmte Modelleigenschaften, etwa in Bezug auf das dynamische WF-Verhalten, durch eine geeignete Konstruktion bzw. Strukturierung der Modelle erzielt werden können.

6. **Verständlichkeit:** Um WF-Modelle auf Benutzerebene (semantisch) validieren zu können, müssen WF-Beschreibungen nicht nur für Modellierer, sondern in gewissen Umfang auch für die Anwender verständlich sein – zumindest für diejenigen von ihnen, die (dynamische) Änderungen

vornehmen können sollen. Neben geeigneten Strukturierungs- und Visualisierungsmöglichkeiten (vgl. Punkte 2 und 7) sind hier verschiedene Aspekte nützlich, etwa die Kompaktheit der Modellierung oder die Überschaubarkeit der bereitgestellten Menge an Beschreibungskonstrukten.

7. **Visualisierbarkeit:** Eine nützliche Hilfe bei der Erstellung und (dynamischen) Änderung von Workflows bildet die Möglichkeit der visuellen Darstellung von WF-Modellen und -Instanzen. Dadurch können Anwender in die semantische Validation von WF-Modellen mit einbezogen werden. Die graphischen Darstellungsmöglichkeiten sind dabei eng mit den bereitgestellten Strukturierungsmitteln verknüpft. Werden unterschiedliche Konstrukte für die Beschreibung von Normal- und Ausnahmefällen angeboten, so ist z.B. eine unterschiedliche visuelle Darstellung möglich. Nützliche Techniken sind in diesem Zusammenhang das Ein- und Ausblenden von Modellelementen, die unterschiedliche graphische Darstellung verschiedener Knoten- und Kantentypen oder die Vergröberung und Verfeinerung von WF-Modellen.

8. **Änderbarkeit:** Wie bereits dargestellt, müssen WF-Modelle und -Instanzen im laufenden Betrieb änderbar sein. Um eine ausreichende Akzeptanz seitens der Benutzer zu erzielen, sollten sie jederzeit nachvollziehen können, was die Konsequenzen einer beabsichtigten WF-Änderung sind bzw. warum das System bestimmte Modifikationen erlaubt, während es andere verbietet. Insbesondere sollte eine vom Anwender definierte und vom WfMS akzeptierte Änderung in der Folge niemals zu Inkonsistenzen oder Fehlern führen, die möglicherweise sogar die Stabilität des Gesamtsystems beeinträchtigen. Die hierfür notwendigen Überprüfungen (z.B. im Hinblick auf die Parameterversorgung von Aktivitäten) dürfen nicht dem Benutzer aufgebürdet werden, sondern sind systemseitig durchzuführen. Die Grundlage hierfür bilden die explizite und formale Darstellung der WF-Modelle, die Abbildung von WF-Zuständen auf Instanzebene und die Bereitstellung von Verfahren zur effizienten Analyse von WF-Modellen und -Instanzen. Ein weiterer wichtiger Aspekt, der sich in Verbindung mit der Änderung von WF-Modellen und -Instanzen ergibt, ist die Bereitstellung eines Rechte- und Autorisierungsmodells für (dynamische) Änderungen (Wer darf welche Änderungen an welchen WF-Modellen und -Instanzen vornehmen?).

9. **Erweiterbarkeit:** Das ADEPT-Basismodell soll einen begrenzten, überschaubaren Satz an Modellkonstrukten anbieten, der als Grundlage für die Definition verschiedener WF-Beschreibungssprachen dienen kann. Komplexe Konstrukte einer WF-Sprache, die zur kompakten Modellierung bestimmter Sachverhalte (z.B. planbaren Abweichungen) notwendig werden, sollen auf einfache Konstrukte des Basismodells übersetzt werden können. Ein Vorteil ist die Erweiterbarkeit von Beschreibungssprachen, ohne Anpassungen des Basismodells vornehmen zu müssen.

Die beschriebenen Entwurfsziele sind teilweise gegensätzlich. So nimmt mit zunehmender Ausdrucksmächtigkeit des verwendeten WF-Metamodells die Komplexität von Analyseverfahren naturgemäß zu [HOR98, KwBa97]. Für diesen Fall wird es auch zunehmend schwieriger, die dynamische Änderbarkeit von Workflows und die sich daraus ergebenden Konsequenzen (z.B. Dateninkonsistenzen, Verklemmungen) theoretisch im Modell zu fassen. Weitere Probleme ergeben sich aus der Forderung, daß (dynamische) WF-Änderungen auch von Endanwendern vorgenommen werden können müssen. Die „Versuchung“ ist daher groß, bei der Konzeption flexibler WfMS die Komplexität des WF-Metamodells sowie die Art und den Umfang der erlaubten Änderungen soweit einzuschränken, daß die Anpassungen mit relativ einfachen Mitteln verwirklicht werden können (vgl. Kapitel 2). Dadurch würde die Anwendbarkeit des WfMS von vornherein auf einige wenige Anwendungsgebiete beschränkt bleiben. Die Unterstützung eines breiten Spektrums an Prozessen, die ausdrucksmächtige WF-Metamodelle verlangt [JaBu96, VoBe96], wäre hingegen nicht möglich.

Wie wir in Kapitel 10 ausführlich diskutieren werden, lassen sich die beschriebenen Zielsetzungen mit gängigen Beschreibungsformalismen, wie (erweiterten) Petri-Netzen [Elli79, Genr91, Jens90, Ober96], State- und Activity-Charts [Hare87, Wodt95] oder Netzplänen [Neum75] nur teilweise realisieren. ADEPT verfolgt statt dessen einen strukturierten, graphbasierten Modellierungsansatz, der Anleihen aus dem Bereich blockorientierter Programmiersprachen [Dijk68] nimmt. Hier werden Kontrollblöcke als strukturiertes programmiersprachliches Mittel zur expliziten Beschreibung von Nebenläufigkeit eingeführt, wobei die Verzweigung und Zusammenführung des Kontrollflusses an jeweils einem ausgezeichneten Punkt erfolgt [KKS97, Rein93]. Dadurch sind nur Kontrollflußstrukturen mit regelmäßiger Schachtelung erlaubt. Wir greifen diesen Ansatz in Abschnitt 3.3 auf, nehmen aber essentielle Erweiterungen vor, um den oben genannten Zielsetzungen gerecht zu werden.

3.2 Aktivitätenvorlagen – Elementare Bausteine für die WF-Modellierung

ADEPT unterstützt die Erstellung und Verwaltung von *Aktivitätenvorlagen*. Dabei handelt es sich um wiederverwendbare Schablonen für die Definition von WF-Aktivitäten. In diesem Abschnitt gehen wir auf die Beschreibung solcher Vorlagen ein. Wir beschränken uns dabei auf Aspekte, die für die Modellierung der Kontroll- und Datenflußbeziehungen zwischen Aktivitäten notwendig sind.

3.2.1 Wiederverwendbarkeit von Aktivitätenbeschreibungen

Im allgemeinen kann dieselbe Anwendungsfunktion (z.B. „Angebot erstellen“, „Rechnung schreiben“) im Kontext unterschiedlicher Arbeitsabläufe zur Ausführung kommen [Sche98a, Sche98b]. Dem entsprechend können verschiedene WF-Modelle auch Arbeitsschritte desselben Typs besitzen. Um zu pflegbaren Modellen zu gelangen, sollten wiederverwendbare Aufgaben- bzw. Aktivitätenbeschreibungen erstellt, in einem *Aktivitätenrepository* hinterlegt und zur Definition verschiedener WF-Vorlagen herangezogen werden können. Der ADEPT-Ansatz unterstützt in diesem Zusammenhang die Definition wiederverwendbarer *Aktivitätenvorlagen*.

Eine *Aktivitätenvorlage* beschreibt eine Aufgabe, die durch eine Schnittstelle und eine Implementierung charakterisiert wird. Die *Vorlagenschnittstelle* definiert Eigenschaften der durchzuführenden Aufgabe, wie Aufgabenziel, Aufgabenbeschreibung, Aufgabenkategorie, Aufgabenträger, Ausführungsmodus (automatisch/manuell) sowie benötigte Eingabe- und produzierte Ausgabedaten. Letztere können nicht nur normale Anwendungsdaten (z.B. „Kundennummer“, „Kundenname“) sein, sondern auch Informationen zu semantischen Fehlern (z.B. „fehlende Bonität des Kunden“), deren Auftreten zum Scheitern der Aufgabenbearbeitung führt. Im ADEPT-WfMS kann bei der Vorlagenbeschreibung noch eine Reihe weiterer Informationen (z.B. Unterbrechbarkeit / Abbrechbarkeit der Aufgabe) angegeben werden, auf deren Darstellung wir in der Folge der Einfachheithalber verzichten. Eine *Vorlagenimplementierung* legt die konkreten Aktionen fest, die zur Bearbeitung der Aufgabe durchzuführen sind. Im Fall einer elementaren Aktivität kann einer Aktivitätenvorlage entweder ein externes Anwendungsprogramm (*Aktivitätenprogramm*) oder eine Menge systeminterner Aktionen (z.B. Aufruf eines Formularprozessors, Auswertung einer prädikativen Bedingung) zugeordnet werden.

Eine Aktivitätenvorlage kann zur Definition mehrerer Aktivitäten derselben oder verschiedener WF-Vorlagen verwendet werden. Umgekehrt ist jede WF-Aktivität mit *genau* einer Vorlage verknüpft – eine Ausnahme bilden in ADEPT sog. *Nullaktivitäten*, auf die wir später eingehen werden. Eine WF-Aktivität, d.h. ein konkreter Schritt in einem WF-Modell, übernimmt jeweils die Eigenschaften (z.B. Ein- und Ausgabeparameter, Fehlerrückgabewerte) der ihr zugeordneten Vorlage. Aktivitätenvorlagen bilden somit die Bausteine für den WF-Entwickler, in Verbindung mit der dynamischen Definition bzw. Änderung laufender WF-Instanzen auch für den Anwender. Die Verfügbarmachung von Aktivitätenvorlagen bildet auch eine wichtige Grundlage dafür, daß Aktivitäten ad hoc zu einer WF-Instanz hinzugefügt werden können, ohne daß der Anwender hierfür aufwendige Spezifikationen für den neu einzufügenden Arbeitsschritt vornehmen muß.

3.2.2 Vorlagenschnittstelle

3.2.2.1 Allgemeine Attribute einer Vorlage

Im folgenden sei die Menge aller Vorlagenbezeichner eines prozeßorientierten Anwendungssystems durch $Templates \subseteq U$ gegeben. (U bezeichne die Menge aller atomaren Bezeichner.) Die Schnittstelle einer Prozeß- oder Aktivitätenvorlage mit Bezeichner $V \in Templates$ wird dann durch das folgende Tupel beschrieben: (Die einzelnen Komponenten entsprechen Funktionswerten. Für eine Abbildung $f: X \mapsto Y$ werden Werte $f(x)$ jeweils mit $f^x \in Y$ abgekürzt.)

$$(vVersion^V, vUsage^V, vDescr^V, vGoal^V, vCategory^V, vExecMode^V, vRole^V, vType^V, vParams^V, vFailures^V)$$

Dieses Tupel faßt die wichtigsten Beschreibungsattribute einer Vorlage zusammen: $vVersion^V$ gibt die Versionsnummer der durch V identifizierten Vorlage an. Die Komponente $vUsage^V$ legt fest, ob die Vorlage zur Definition mehrerer Aktivitäten derselben oder verschiedener WF-Modelle verwendet

werden darf ($vUsage^V$ = LOCAL_REUSABLE bzw. $vUsage^V$ = GLOBAL_REUSABLE) oder ob sie ausschließlich zur Festlegung einer bestimmten Aktivität innerhalb einer konkreten Prozeßvorlage dient ($vUsage^V$ = NON_REUSABLE). Das Attribut $vDescr^V$ legt die Aufgabenbeschreibung, $vGoal^V$ das Aufgabenziel und $vCategory^V$ die Aufgabenkategorie von V fest. Informationen zu Aufgabenkategorien können z.B. für das Retrieval von Vorlagen oder für die Darstellung von Arbeitslisten genutzt werden. Das Attribut $vExecMode^V \in$ {AUTOMATIC, INTERACTIVE} legt fest, ob durch die Vorlage eine automatische oder eine manuelle Aktivität beschrieben wird. Für den letztgenannten Fall muß zusätzlich eine Rolle $vRole^V$ angegeben werden, die ein Akteur einnehmen muß, um die Aktivität ausführen zu können[1]. Ob die Vorlage eine komplexe oder eine elementare Aktivität (vgl. Kapitel 1) beschreibt, wird durch $vType^V \in$ {SIMPLE, COMPLEX} festgelegt. Dem entsprechend ist die Menge aller Bezeichner für WF-Vorlagen durch $WT := \{V \in Templates \mid vType^V = \texttt{COMPLEX}\}$ und die Menge aller Bezeichner für Vorlagen elementarer Aktivitäten durch $AT := \{V \in Templates \mid vType^V = \texttt{SIMPLE}\}$ gegeben.

Für die Kontroll- und Datenflußmodellierung sind die Attribute $vParams^V$ und $vFailures^V$ von besonderer Wichtigkeit: Die Menge $vParams^V \subseteq Params$ (= Menge aller Parameterbezeichner) definiert die Ein- und Ausgabeparameter der durch V beschreibbaren Aktivitäten (vgl. Abschnitt 3.2.2.2). Im allgemeinen kann eine Aktivität nicht in jedem Fall erfolgreich bearbeitet werden, so daß das Aufgabenziel $vGoal^V$ nicht immer erreichbar ist. Für das Scheitern der Aufgabenbearbeitung kann es unterschiedliche Gründe (sog. *semantische Fehler*) geben, wobei je nach Art des Fehlers andere Ausnahmebehandlungen erforderlich werden können. Aus diesem Grund können a priori bekannte semantische Fehler in der Menge $vFailures^V \subseteq U$ zusammengefaßt werden. Unter Bezugnahme auf Fehlercodes aus $vFailures^V$ können dann bei der Definition von Prozeßvorlagen spezifische Ausnahmebehandlungen festgelegt werden.

Vorlage / Attribute	**V = „CT-Untersuchung durchführen“**	**V = „Rechnungsnummer ermitteln“**
$vVersion^V$	V2.0	V1.1
$vUsage^V$	GLOBAL_REUSABLE	GLOBAL_REUSABLE
$vDescr^V$	Durchführung einer CT-Untersuchung	Bestimmung einer Rechnungsnr.
$vGoal^V$	Diagnostik für Patienten	Eindeutige Rechnungssnummer
$vCategory^V$	Diagnostisches Procedere	Rechnungsstellung
$vExecMode^V$	INTERACTIVE	AUTOMATIC
$vRole^V$	RADIOLOGE **or** MTA	SYSTEM
$vType^V$	SIMPLE	SIMPLE
$vParams^V$	PATID ↓, PATALTER ↓, CT_BILD ↑ (vgl. Tabelle 3-2)	KUNDENID ↓, ANGEBOTSID ↓, BESTELLDATUM ↓, RECHNUNGSID ↑
$vFailures^V$	PATIENT_NICHT_NÜCHTERN	–

Tabelle 3-1: *Beispiele für Vorlagenbeschreibungen (stark vereinfacht)*

3.2.2.2 Ein- / Ausgabeparameter und ihre Eigenschaften

Eine wichtige Grundlage für die Wiederverwendung von Aktivitätenvorlagen bildet die formale Beschreibung ihrer Ein- und Ausgabeparameter. Die Eingabeparameter einer Vorlage legen Anwendungsdaten fest, die zur Bearbeitung der beschriebenen Aktivität benötigt werden. Ihre Ausgabeparameter fassen Ergebnisse der Aufgabenbearbeitung zusammen. Die formale Beschreibung von Para-

[1] Im ADEPT-WfMS (siehe Kapitel 10) können für Aktivitätenvorlagen auch komplexere Bearbeiterausdrücke festgelegt werden, wobei neben Rollen auch auf andere organisatorische Entitäten (z.B. Organisationseinheiten, Stellen, Kompetenzen) Bezug genommen werden kann. Des weiteren kann bei der Definition einer Aktivität die Vorgabe ihrer Vorlage konkretisiert werden, etwa wenn abhängige Bearbeiterzuordnungen (z.B. „gleicher Bearbeiter wie“) festgelegt werden sollen (für Details siehe [Kubi98]).

metern ist notwendig, um später die Datenflüsse zwischen den WF-Aktivitäten definieren und auf ihre Korrektheit hin überprüfen zu können. Wie bereits erwähnt, schafft die explizite Modellierung von Datenflüssen überhaupt erst die Voraussetzung dafür, (dynamische) WF-Änderungen korrekt durchführen zu können. Ad-hoc-Änderungen, wie das Hinzufügen oder Löschen einer Aktivität, sind nur dann sinnvoll unterstützbar, wenn formale Beschreibungen der Aktivitätenparameter vorliegen.

a) Parameterbeschreibung

Im folgenden sei durch *Params* die Menge aller Parameterbezeichner gegeben. Wie oben erwähnt, ist jeder Vorlage $V \in Templates$ eine endliche Menge $vParams^V \subseteq Params$ zugeordnet, die ihre Ein- und Ausgabeparameter festlegt. Für jeden einzelnen Parameter $par \in vParams^V$ werden seine „Richtung" (Ein- bzw. Ausgabeparameter), sein Wertebereich, ggf. ein Default-Wert sowie verschiedene Attribute definiert. Letztere geben z.B. an, ob für den Parameter zwingend ein Wert vorliegen muß oder ob er bei Fehlen anderweitig festgelegt werden kann (für ein Beispiel siehe Tabelle 3-2). Diese Informationen sind für die Modellierung von Datenflüssen (siehe Kapitel 5) ebenso nützlich, wie für den Erhalt der Datenkonsistenz bei (dynamischen) WF-Änderungen. Ein durch $par \in vParams^V$ festgelegter Parameter wird durch folgendes Tupel beschrieben

$$(pDir^{par}, pName^{par}, pDom^{par}, pDefault^{par}, pDescr^{par}, pDm^{par}, pRm^{par}, pTempl^{par})$$

Die einzelnen Komponenten haben die folgende Bedeutung:

- $pDir^{par} \in \{\texttt{IN}, \texttt{OUT}\}$ (engl. direction) gibt an, ob *par* einen Ein- oder Ausgabeparameter beschreibt. Für eine Vorlage V sind ihre Eingabeparameter durch $InParams^V := \{p \in vParams^V : pDir^p = \texttt{IN}\}$ und ihre Ausgabeparameter durch $OutParams^V := \{p \in vParams^V : pDir^p = \texttt{OUT}\}$ gegeben. Auf die explizite Darstellung von *InOut*-Parametern wird verzichtet. O.B.d.A. nehmen wir an, daß sie auf je einen Eingabe- und Ausgabeparameter gleichen Namens abgebildet werden.
- $pName^{par} \in U$ ist der logische Name des Parameters. Für eine Vorlage V müssen Parameternamen innerhalb der Mengen $InParams^V$ und $OutParams^V$ jeweils eindeutig sein, d.h. es soll gelten:
  $$\forall\ V \in Templates,\ \forall\ par_1, par_2 \in InParams^V\ (OutParams^V): pName^{par1} \neq pName^{par2}$$
- $pDom^{par} \in Dom$ (engl. domain) beschreibt den Wertebereich des Parameters. *Dom* bezeichne dabei die Menge aller atomaren Wertebereiche und enthalte die in einem Rechner direkt darstellbaren Basistypen (ganze Zahlen, Gleitkommazahlen, Zeichenketten, Bool'sche Werte, etc.). Notwendige Erweiterungen zur Unterstützung komplex strukturierter Parameterobjekte sind in [Blas96] beschrieben, werden hier aber der Einfachheithalber ausgeklammert. Für diese Arbeit vordergründig von Interesse ist die korrekte Modellierung von Datenflüssen und die Aufrechterhaltung ihrer Korrektheit bei der Durchführung von (dynamischen) WF-Änderungen.
- $pDefault^{par} \in pDom^{par}$ ist ein (optionaler) Default-Wert für den Parameter. Er dient der Versorgung des entsprechenden Eingabeparameters der gerufenen Anwendungskomponente, wenn durch den Aufrufkontext der Aktivität kein Wert festgelegt wird. Umgekehrt wird an den Aufrufkontext der Aktivität der Default-Wert eines Ausgabeparameters übergeben, wenn für diesen von der gerufenen Anwendungskomponente kein Wert geschrieben wird.
- $pDescr^{par}$ (engl. description) ist die textuelle Beschreibung des Parameters. Sie kann zum Beispiel von Formularprozessoren bei der (dynamischen) Generierung von Bildschirmmasken zur Anzeige oder Eingabe von Parameterwerten genutzt werden.
- $pDm^{par} \in \{\texttt{MANDATORY}, \texttt{OPTIONAL}\}$ (engl. demand mode) gibt für den Parameter an, ob er *obligat* oder *optional* ist (Default: MANDATORY). Zur korrekten Durchführung der Aufgabe müssen obligate *Eingabeparameter* zwingend mit einem Wert belegt sein. Dagegen können optionale Eingabeparameter auch „unversorgt" bleiben. Umgekehrt legt diese Einteilung für *Ausgabeparameter* fest, ob für sie nach erfolgreicher Bearbeitung der Aufgabe zwingend ein Ausgabewert vorliegen muß oder nicht. Im allgemeinen kann nur für obligate Ausgabeparameter davon ausgegangen werden, daß sie nach einer erfolgreichen Aufgabenbearbeitung versorgt sind.
- $pRm^{par} \in \{\texttt{NO_REQUEST}, \texttt{USER_REQUEST}, \texttt{SPECIAL_REQUEST}\}$ (engl. request mode) legt für den Parameter fest, ob sein Wert nachforderbar ist und – falls ja – ob die Nachforderung über eine

spezielle Aktivität (pRm^{par} = SPECIAL_REQUEST) oder über automatisch generierte Bildschirmmasken (pRm^{par} = USER_REQUEST) erfolgen kann (Default: NO_REQUEST). Falls pRm^{par} = SPECIAL_REQUEST gilt, wird durch $pTempl^{par} \in AT$ die Aktivitätenvorlage festgelegt, die einen speziellen Anwendungsdienst zur Nachforderung des Parameters beschreibt.

Die für die Attribute pDm^{par} und pRm^{par} vorliegenden Voreinstellungen *können* von WF-Modellierern geändert werden, *müssen* aber nicht. Das bedeutet, daß es ihnen überlassen bleibt, ob – und falls ja – in welchem Umfang sie von diesen Beschreibungsmöglichkeiten Gebrauch machen wollen. Wir werden später in Verbindung mit der Modellierung von Datenflüssen zeigen, wie die einzelnen Informationen dazu genutzt werden, um zu korrekten Datenflußschemata zu gelangen und wie sich deren Korrektheit auch nach Durchführung von (dynamischen) WF-Änderungen aufrechterhalten läßt.

Parameter / Attribute	par = PATALTER	par = CT_BILD
$pDir^{par}$	IN (Eingabeparameter)	OUT (Ausgabeparameter)
$pName^{par}$	*PatAlter*	*CT_Bild*
$pDom^{par}$	INTEGER	BYTESTRING
$pDefault^{par}$	NONE	NONE
$pDescr^{par}$	Alter eines Patienten	Digitales CT-Bild
pDm^{par}	MANDATORY	MANDATORY
pRm^{par}	USER_REQUEST (nachforderbar)	NO_REQUEST (nicht nachforderbar)

Tabelle 3-2: *Beispiele für Parameterbeschreibungen*

b) Obligate und optionale Parameter

Die Unterscheidung zwischen optionalen und obligaten Parametern läßt bei der Definition und Änderung von WF-Modellen und -Instanzen mehr Freiheitsgrade zu. Ist ein Eingabeparameter *par* als optional gekennzeichnet *(pDm^{par}* = OPTIONAL*)*, so ist die Vorlage auch dann noch korrekt verwendbar, wenn der Wert dieses Parameters aus dem Datenkontext des WF nicht oder – infolge einer Änderung (z.B. in Bezug auf die Reihenfolge von Arbeitsschritten) – nicht mehr versorgt werden kann (und auch kein Default-Wert vorliegt). Entsprechend muß für optionale Ausgabeparameter zur Ausführungszeit nicht notwendigerweise ein Ausgabewert geliefert werden. Diese Information kann nützlich sein, wenn Seiteneffekte einer WF-Änderung, wie fehlende Ausgabedaten nach dem Löschen einer Aktivität, behandelt werden müssen. Die Festlegung, welche Parameter einer Aktivität obligat und welche optional sind, hat deshalb direkte Auswirkungen auf ihre Einbettung in den WF.

Ein Extremfall wäre gegeben, wenn sämtliche Eingabe- und Ausgabeparameter einer Aktivität *X* (bzw. ihrer Vorlage) optional sind. Für diesen Fall könnten WF-Änderungen, wie das Vorziehen, Überspringen oder Verschieben von *X*, zumindest aus Datenflußsicht, ohne Schwierigkeiten durchgeführt werden. Weder in Bezug auf die Parameterversorgung nachfolgender Aktivitäten – alle Ausgaben von *X* sind optional – noch in Bezug auf die Versorgung der Aufrufparameter von *X* selbst, würden sich Probleme ergeben. Eine mögliche Anwendung ist z.B. ein formularbasierter WF, bei dem alle Aktivitäten durch Masken mit optionalen Ein- bzw. Ausgabefeldern implementiert sind. Ein anderer Sonderfall liegt vor, wenn alle Parameter einer Aktivität *Y* obligat sind. (Beispielsweise könnte die Durchführung der Aktivität *„Berechnung einer Medikamentendosis*" das Vorliegen der obligaten Eingabedaten *Alter*, *Gewicht* und *Größe* des Patienten erfordern.) Die Datenflußmodellierung muß dann so erfolgen, daß vor Aufruf der Aktivität *Y* alle Eingabeparameter versorgt und nach ihrer Ausführung alle Ausgabeparameter mit Werten belegt sind. Dadurch sind Laufzeitabweichungen, wie das Vorziehen, Überspringen oder Verschieben der Aktivität, im allgemeinen nicht mehr ohne Auswirkungen auf die Parameterversorgung anderer Arbeitsschritte (inkl. *Y* selbst) möglich.

c) Nachforderbarkeit von Parameterwerten

Nicht immer ist es möglich, die Aufrufdaten einer Aktivität direkt aus dem Datenkontext des WF, d.h. auf der Grundlage von Ausgabedaten vorangehender Aktivitäten, zu versorgen. Beispielsweise kann

es infolge des Überspringens einer Aktivität zum Fehlen obligater Ausgabeparameter kommen. Dies wiederum kann zur Folge haben, daß (obligate) Eingabeparameter nachfolgender Aktivitäten unversorgt bleiben. Werden solche Fälle nicht angemessen behandelt, können inkonsistente oder fehlerhafte Verarbeitungen, in Einzelfällen sogar schwerwiegende Programmfehler resultieren. Aus diesem Grund kann der Modellierer bei der Beschreibung von Parametern festlegen, ob und – falls ja – auf welche Art und Weise die fehlenden Parameterwerte in einem solchen Fall ersatzweise festgelegt werden können. Ist für einen Parameter *par* ein solches Nachversorgen nicht möglich, so muß für ihn das Attribut pRm^{par} auf den Wert `NO_REQUEST` gesetzt werden (Default-Wert). Kann der Parameter durch eine spezielle Aktivität, die durch eine separate Vorlage $pTempl^{par}$ beschrieben wird, versorgt werden, so wird dem Attribut pRm^{par} der Wert `SPECIAL_REQUEST` zugewiesen. In zahlreichen Fällen wird es genügen, die Nachforderung des Parameters über systemseitige Nachforderungsdienste (z.B. automatisch generierte Eingabemasken) zu bewerkstelligen. Für diesen Fall wird pRm^{par} der Wert `USER_REQUEST` zugewiesen. Die jeweilige Festlegung der Eigenschaften von Parametern kann von verschiedenen Faktoren abhängig sein, etwa dem Kontext, in dem die Nachversorgung erfolgen soll und der Art der durch die Vorlagen beschriebenen Tätigkeit.

Beispiel 3-1 (Nachforderbarkeit von Parameterwerten)

Ein Röntgenbild, das Ausgabe der Aktivität „Röntgen-Thorax-Untersuchung“ ist, kann nicht nachgefordert werden, da es nur von dieser Aktivität generiert wird (*pRm* = `NO_REQUEST`). Ein mit einem Textsystem zu erstellender Kurzarztbrief, der nach Auslassen einer Aktivität fehlt, kann durch die neu hinzugefügte Aktivität „Brief nachträglich erstellen“ nachgefordert werden (*pRm* = `SPECIAL_REQUEST`). Die Krankenkasse eines Patienten, die bei der Notaufnahme (ausnahmebedingt) nicht angegeben werden konnte, kann durch eine vom System generierte Abfrage nachträglich erfaßt werden (*pRm* = `USER_REQUEST`).

Wir werden auf den Aspekt der Nachforderbarkeit von Parametern in Kapitel 7, im Zusammenhang mit dynamischen WF-Änderungen (z.B. nach Löschen eines Schrittes mit obligaten Ausgabeparametern), nochmals zurückkommen. Dabei werden wir auch auf verschiedene Strategien bei der Nachforderung von Parametern (z.B. in Bezug auf den Zeitpunkt der Nachforderung) eingehen.

3.2.2.3 Status- und Fehlerparameter

Jede Vorlage $V \in Templates$ beschreibt eine in sich abgeschlossene Aufgabe. Bezogen auf das Ergebnis einer Aufgabenbearbeitung wird zwischen erfolgreicher und gescheiterter Durchführung unterschieden. Zur Ausführungszeit zeigt eine Aktivität diesen Status durch den Ausgabeparameter $status \in OutParams^V$ an. Dieser spezielle Parameter ist standardmäßig für jede Vorlage und damit für jede darauf basierende Aktivität definiert.

Besitzt der Statusparameter im Anschluß an die Ausführung einer Aktivität den Wert `COMPLETED`, war ihre Bearbeitung erfolgreich. Das bedeutet, daß das Aufgabenziel $vGoal^V$ erreicht werden konnte und – die korrekte Implementierung von Aktivitätenprogrammen vorausgesetzt – alle obligaten Ausgabeparameter der Aktivität mit Werten versorgt sind. Scheitert dagegen die Bearbeitung einer Aufgabe wird dem Statusparameter der Aktivität der Wert `FAILED` zugewiesen. Für diesen Fall bleiben Ausgabeparameter undefiniert, so daß nicht ohne weiteres mit der Bearbeitung des WF fortgefahren werden, insbesondere nicht dann, wenn obligate Ausgabeparameter unversorgt bleiben. Statt dessen muß eine gesonderte Behandlung erfolgen, bei der entweder die Ursachen für das Scheitern behoben werden (z.B. durch partielles Zurücksetzen des WF und Wiederholen bestimmter Aufgaben) oder eine vom normalen Ablauf abweichende Vorgehensweise (z.B. Änderung oder Abbruch des WF) gewählt wird. Eine wichtige Entscheidungshilfe bei der Wahl einer geeigneten Ausnahmebehandlung stellt eine Beschreibung der Fehlerursache, die zum Scheitern der Bearbeitung geführt hat, dar. In ADEPT können semantische Fehler, die zur Modellierungszeit bekannt sind, bei der Vorlagendefinition angegeben werden. Zu diesem Zweck kann jeder Vorlage eine endliche Menge von Fehlerwerten ($vFailures^V$) zugeordnet werden. Kann eine durch V beschriebene Aktivität zur Laufzeit nicht erfolgreich ausgeführt werden, wird ihre Bearbeitung abgebrochen, ihrem Statusparameter der Wert `FAILED` zugewiesen und dem speziellen Fehlerparameter $failure \in OutParams^V$ (Default: `UNDEFINED`) ein Fehlerwert $f \in vFailures^V$ zugeordnet. Unter Bezugnahme auf mögliche semantische Fehler können dann bei der WF-Modellierung spezifische Ausnahmebehandlungen für das Scheitern einer Aktivität (z.B. partielles Rücksetzen des WF) festgelegt werden (vgl. Kapitel 6).

Beispiel 3-2 (Semantische Fehler)

Fehlerursachen für das Scheitern der Aufgabe „Patienten untersuchen“ können sein: Fehlende Vorbefunde, unterlassene oder falsche Vorbereitung des Patienten, Rücknahme der Einwilligung des Patienten usw. Abhängig vom konkreten Grund können unterschiedliche Ausnahmebehandlungen notwendig werden.

3.2.3 Vorlagenimplementierung

Einer Vorlage, die eine elementare Aktivität beschreibt, kann mit einem Anwendungsprogramm verknüpft werden (sog. *Programmaktivitäten*). Dieses Programm wird zur Laufzeit, bei der Ausführung entsprechender Aktivitäteninstanzen, vom WfMS gerufen. Dabei werden seine Eingabeparameter mit Daten aus dem WF-Datenkontext versorgt und im Anschluß an seine Ausführung die Werte von Ausgabeparametern in den WF-Datenkontext übernommen. Alternativ zur Verknüpfung mit einem Anwendungsprogramm können einer Aktivitätenvorlage auch systeminterne Aktionen (z.B. Auswertung von Prädikaten auf WF-Daten) zugewiesen werden (sog. *Nicht-Programmaktivitäten*).

3.2.3.1 Programmaktivitäten

Die Art und Weise, in der externe Anwendungsprogramme an Aktivitätenvorlagen angeknüpft und zur Laufzeit vom WfMS angesteuert werden, ist für die nachfolgenden Betrachtungen nur von untergeordneter Bedeutung. Die Fragestellungen, die sich dabei ergeben sind orthogonal zu den eingangs diskutierten Zielsetzungen. Sie betreffen den Aufruf entfernter Anwendungskomponenten, das statische oder dynamische Binden von Anwendungskomponenten oder die Behandlung von Fehlerfällen (z.B. Konsistenthaltung von Anwendungsdaten und WF-Daten bei Programmabstürzen), um nur einige Beispiele zu nennen [JBS97]. Wir fassen an dieser Stelle lediglich einige wichtige Aspekte zusammen, die für ein grundlegendes Verständnis notwendig sind. Wird eine Vorlage zur Definition einer Programmaktivität verwendet, wird die mit ihr verknüpfte Anwendungskomponente zur Laufzeit vom WfMS aufgerufen, wenn neue Instanzen dieser Aktivität erzeugt und gestartet werden. Das WfMS übernimmt die Aktivierung des Programms und bietet Schnittstellen für die Versorgung und Übernahme seiner Aufrufparameter an. Die Programmausführung selbst erfolgt außerhalb seiner Kontrolle. Externen Anwendungsprogrammen (z.B. „Termin buchen“) können Kompensationsprogramme (z.B. „Termin stornieren“) zugeordnet werden, deren Ausführung die Auswirkungen der normalen Programmausführung semantisch rückgängig machen. Bei einem Kompensationsprogramm kann es sich auch um eine „Nullkompensation“ oder um die Komponente selbst handeln, die im Fall einer erforderlichen Kompensation mit „inversen“ Parametern gerufen wird (siehe Kapitel 6).

Die Implementierung eines Aktivitätenprogramms kann im Prinzip in einer beliebigen Programmiersprache erfolgen. Allerdings muß für jede unterstützte Sprache eine Programmierschnittstelle für die Anbindung der Programmparameter an den WF-Datenkontext bereitgestellt werden. Geeignet bedeutet hierbei, daß ein Aktivitätenprogramm bei seinem Aufruf die Werte seiner Eingabeparameter aus dem Datenkontext der WF-Instanz abfragen und nach seiner Beendigung die Werte seiner Ausgabeparameter in diesen Datenkontext schreiben können muß. Abgesehen davon sind Aktivitätenprogramme autonom ausführbar, machen also z.B. ihre eigene Bildschirmeingabe und -ausgabe. Dies erleichtert in erheblichem Umfang das isolierte Testen der Komponenten. Darüber hinaus können Anwendungskomponenten bei der Definition verschiedener Aktivitätenvorlagen wiederverwendet werden. Notwendige Voraussetzung hierfür ist, daß ihre Schnittstellen geeignet beschrieben werden. Im Kern müssen dabei ähnliche Informationen (z.B. in Bezug auf Ein- und Ausgabeparameter) wie in Abschnitt 3.2.2.2 festgelegt werden.

Eine wichtige Forderung für das korrekte „Einsetzen“ einer Komponente in eine Aktivitätenvorlage ist die nach *Plug-in-Kompatibilität* der beiden Schnittstellen. Sie gibt an, ob eine Komponente die von der Aktivitätenvorlage verlangten Minimalforderungen erfüllt. Insbesondere müssen alle obligaten Eingabeparameter der Anwendungskomponente aus den obligaten Eingabeparametern der Aktivitätenvorlage versorgt werden können. Umgekehrt muß die Komponente ihrerseits genügend zur Versorgung der obligaten Ausgabeparameter der Vorlage beitragen. Trivialerweise trifft dies zu, wenn die Vorlage und Anwendungskomponente dieselbe Signatur aufweisen. – Eine Anwendungskomponente C erfüllt die Minimalanforderungen einer Vorlage $V \in AT$, ist also plug-in-kompatibel zu V, wenn gilt:

(Wir berücksichtigen bei dieser Definition lediglich die Signaturen von Komponenten bzw. Vorlagen, und gehen davon aus, daß Parameter atomare Wertebereiche besitzen.)

$\forall\, x \in InParams^C$ mit $pDm^x = \texttt{MANDATORY} \Rightarrow$
$\exists\, p \in InParams^V$: $(pDm^p = \texttt{MANDATORY} \vee pDefault^p \neq \texttt{UNDEFINED}) \wedge pDom^p \subseteq pDom^x$
und

$\forall\, q \in OutParams^V$ mit $(pDm^q = \texttt{MANDATORY} \wedge pDefault^q = \texttt{UNDEFINED}) \Rightarrow$
$\exists\, y \in OutParams^C$: $pDm^y = \texttt{MANDATORY} \wedge pDom^y \subseteq pDom^q$

Vereinfacht ausgedrückt bedeutet Plug-in-Kompatibilität, daß C „höchstens die gleichen (obligaten) Eingaben benötigen darf“ und „mindestens die gleichen (obligaten) Ausgaben machen muß“ wie eine durch die Vorlage V beschriebene Aktivität. Eine Erweiterung dieser Definition wird erforderlich, wenn bei der Beschreibung von Parametern komplex strukturierte Datenobjekte oder semantische Beziehungen (z.B. Spezialisierung) berücksichtigt werden sollen. Bei obiger Definition sind auch Abbildungen (z.B. Typkonversionen) zwischen Aktivitäten- und Komponentenparametern ausgeklammert worden (siehe [Blas96]). Im folgenden gehen wir davon aus, daß einer Aktivitätenvorlage (bzw. Aktivität) stets plug-in-kompatible Anwendungskomponenten zugeordnet werden. Des weiteren nehmen wir an, daß eine Komponente bei korrektem Aufruf durch das WfMS auch korrekt ausgeführt wird. Korrekt bedeutet, daß die Komponente nach endlicher Zeit terminiert und entweder alle obligaten Ausgabeparameter in den WF-Datenkontext schreibt (*erfolgreiche* Ausführung) oder aber, daß sie das Auftreten eines semantischen Fehlers (vgl. Abschnitt 3.2.2.3) durch die Rückgabe eines entsprechenden Fehlerwertes anzeigt[2] (*gescheiterte* Ausführung).

Durch die Anknüpfung externer Programme an Vorlagen erreichen wir, daß die Implementierung dieser Komponenten weitgehend getrennt von der WF-Definition erfolgen kann. Zur Laufzeit muß das WfMS lediglich sicherstellen – die korrekte Ausführung der Aktivitätenprogramme vorausgesetzt –, daß die (obligaten) Eingabeparameter der Komponenten aus dem WF-Datenkontext korrekt versorgt und ihre Ausgabeparameter in den WF-Datenkontext übernommen werden können. Ersteres setzt voraus, daß obligate Eingabeparameter einer Aktivität zwingend von im Kontrollfluß vorausgehenden Aktivitäten geschrieben werden, unabhängig von den durchlaufenen Ausführungspfaden. Andernfalls können Dateninkonsistenzen oder möglicherweise sogar Programmfehler resultieren.

Der Einfachheithalber gehen wir im folgenden davon aus, daß die Zuordnung einer Anwendungskomponente zu einer Vorlage statisch erfolgt. In der Praxis macht es durchaus auch Sinn, Anwendungskomponenten erst dynamisch zu binden, etwa abhängig von WF-Daten oder Benutzerprofilen. Für diesen Fall ist eine Aktivitätenvorlage mehr oder weniger Platzhalter für das eigentliche Anwendungsprogramm. Bei der Vorlagendefinition müssen dann Prädikate oder Regeln für die Auswahl geeigneter Anwendungsbausteine zur Laufzeit vereinbart werden.

3.2.3.2 Nicht-Programmaktivitäten

Um einfache Operationen, wie die Initialisierung oder Inkrementierung von Prozeßvariablen, zu realisieren, müssen diesen Variablen Werte zugewiesen werden können. Solche Wertezuweisungen sollten mit einfachen Mitteln realisierbar sein, ohne hierfür – wie bei vielen WfMS üblich – spezielle Aktivitätenprogramme implementieren und zur Laufzeit ansteuern zu müssen. Dasselbe gilt in Bezug auf Operationen, die für die Kontrollflußsteuerung benötigt werden (z.B. Auswertung eines Verzweigungs-/Schleifenprädikats auf WF-relevanten Daten). Für die Definition solcher Nicht-Programmaktivitäten (bzw. ihrer Vorlagen) bietet ADEPT einen Satz eingebauter *Systemfunktionen* an, die anstelle eines externen Anwendungsprogramms gerufen werden können. Dabei müssen die Aufrufargumente einer Systemfunktion aus den Eingabeparametern der Aktivität ableitbar sein, ebenso wie ihre Ergebnisse auf Ausgabeparameter der Aktivität abzubilden sind. Dadurch können Nicht-Programmaktivitäten bei der WF-Modellierung wie gewöhnliche Aktivitäten behandelt werden. Beispiele für eingebaute Systemfunktionen sind

- die Zuweisung eines konstanten Wertes an einen Ausgabeparameter der Aktivität,

[2] Für Vorlagen $V \in AT$ und plug-in-kompatible Komponenten C fordern wir, daß gilt: $vFailures^V \supseteq vFailures^C$. Das bedeutet, daß die Komponente keine Fehlerwerte liefern darf, die auf Aktivitätenebene nicht bekannt sind.

- die Anwendung einer vordefinierten Funktion auf einen Eingabeparameter (z.B. Inkrementierung um 1) und die Zuweisung des Ergebnisses an einen Ausgabeparameter der Aktivität,
- die Anwendung einer Aggregationsfunktion (z.B. Summation, Maximum-/Minimumbestimmung) auf mehrere Eingabeparameter (gleichen Domains) und die Zuweisung des Ergebnisses an einen Ausgabeparameter der Aktivität
- die Auswertung eines Prädikates auf Eingabeparametern und die Zuweisung des Ergebnisses (*True* oder *False*) an einen speziellen Ausgabeparameter der Aktivität (*choice*)

Die Ausführung solcher Nicht-Programmaktivitäten kann – im Gegensatz zu Programmaktivitäten – vollständig unter der Kontrolle des WfMS erfolgen. Wichtig ist wieder die korrekte Parameterversorgung und -übernahme zur Laufzeit. So müssen zur Auswertung eines Verzweigungsprädikates die hierfür benötigten Daten aus dem WF-Datenkontext sicher versorgt werden können, unabhängig von zuvor durchlaufenen Ausführungspfaden oder vorgenommenen WF-Änderungen. Andernfalls können falsche Verzweigungsentscheidungen oder sogar Laufzeitfehler die Folge sein.

Bei der Definition einer Aktivität kann der WF-Modellierer entweder auf eine bereits vorhandene Vorlage aus dem Aktivitätenvorlagen-Repository zurückgreifen oder er muß, ggf. ausgehend von einer existierenden Vorlage, eine neue Beschreibung erstellen, die dann in das Repository übernommen wird. Dabei muß er festlegen, ob die neu definierte Vorlage auch zur Definition anderer Aktivitäten herangezogen werden darf oder nicht. Der eigentliche Arbeitsfluß ergibt sich durch die Festlegung von Ausführungsreihenfolgen und -bedingungen für die Aktivitäten (*Kontrollfluß*) sowie durch die geeignete Verknüpfung ihrer Ein- und Ausgabeparameter (*Datenfluß*).

3.3 Kontrollflußmodellierung im ADEPT-Basismodell

Die Aufgabe der Kontrollflußmodellierung ist die präzise Festlegung der Ausführungsreihenfolgen und -bedingungen für die einzelnen WF-Aktivitäten. Wichtig für die Anwendbarkeit einer Kontrollflußbeschreibungssprache sind die Ausdrucksmächtigkeit und Verständlichkeit der angebotenen Konstrukte. Es müssen nicht nur Möglichkeiten zur Modellierung sequentieller, paralleler oder alternativer Ausführungspfade vorhanden sein, sondern auch für die Festlegung iterativer Aufgabenbearbeitungen (Schleifen). Darüber hinaus müssen planbare Abweichungen vom „normalen“ Ablauf adäquat beschreibbar sein, etwa das partielle Zurücksetzen von Workflows beim Auftreten semantischer Fehler oder die Durchführung ausnahmebedingter Vorwärtssprünge im Kontrollfluß. Dies setzt voraus, daß der Normalablauf und die Behandlung von Ausnahmen getrennt voneinander modellierbar sind.

Im ADEPT-Basismodell können Kontrollflüsse zwischen Aktivitäten explizit beschrieben werden. Diese Beschreibung kann – im Gegensatz zu Petri-Netz-basierten WF-Modellen – (weitgehend) getrennt von der Festlegung anderer WF-Aspekte, wie Datenfluß oder Zeitabhängigkeiten [Gri97], erfolgen. Allerdings bestehen zwischen den verschiedenen Aspekten naturgemäß Interdependenzen, die auf der Modellierungs- und Ausführungsebene zu beachten sind. Beispielsweise kann eine Aktivität ihre Eingaben nicht von einer anderen, im Kontrollfluß nachfolgenden Aktivität erhalten (Ausnahme: Schleifen), woraus sich Restriktionen in Bezug auf die Datenflußmodellierung ergeben. Hierauf gehen wir ausführlich in Kapitel 5 ein. In diesem Kapitel konzentrieren wir uns auf Kontrollflußaspekte. Aus Gründen der besseren Lesbarkeit werden dabei noch nicht alle vom ADEPT-Basismodell bereitgestellten Beschreibungskonstrukte im Detail vorgestellt. Einige spezielle Modellelemente, z.B. Kantentypen zur Modellierung von Vorwärts- und Rückwärtssprüngen, werden später eingeführt, wenn die Motivation für ihre Verwendung klar ist und andere wichtige Grundlagen, z.B. die Datenflußmodellierung betreffend, behandelt worden sind.

In Abschnitt 3.3.1 wird die grundlegende Philosophie des von ADEPT verfolgten Ansatzes zur strukturierten Modellierung von Kontrollflußschemata vorgestellt. Im darauffolgenden Abschnitt werden die hierfür bereitgestellten Konstrukte zunächst informell eingeführt. Dabei wird ein graphbasiertes Modell verwendet. Eine Formalisierung der strukturellen Eigenschaften von Kontrollflußschemata erfolgt in Abschnitt 3.4. Hierbei steht vor allem die Untersuchung statischer Modelleigenschaften im Vordergrund, während in Kapitel 4 das dynamische Verhalten von Workflows und die Ausführungssemantik der Kontrollflußkonstrukte formalisiert werden. Hinter den letzten beiden Punkten steckt die

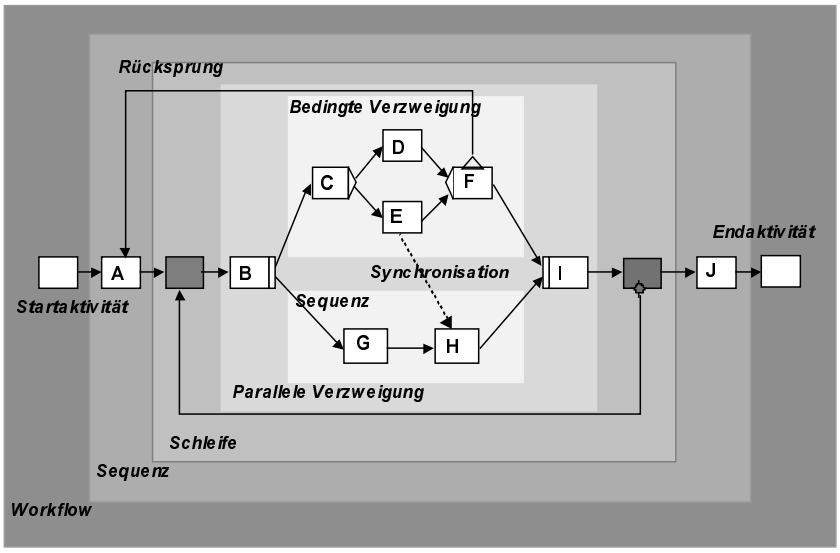


Abbildung 3-1: *Strukturierte Modellierung in ADEPT (Kontrollflußsicht).*

Erkenntnis, daß nur formale Darstellungen wirklich präzise und zuverlässige Aussagen über die Korrektheit von WF-Modellen und -Instanzen erlauben.

3.3.1 Strukturierte Modellierung von Workflows

Die Modellierung von Kontrollflüssen basiert in ADEPT auf einem strukturierten, graphbasierten Ansatz, der Anleihen aus dem Bereich blockorientierter Programmiersprachen [Dijk68] nimmt. In Zentrum dieses Ansatzes steht das Konzept der *regelmäßigen Blockstrukturierung*, bei dem Kontrollstrukturen wie Sequenzen, Verzweigungen und Schleifen auf Blöcke mit eindeutiger Start- und Endaktivität abgebildet werden. Solche Blockstrukturen – sie werden im folgenden als *Kontrollblöcke* bezeichnet – dürfen sich nicht überlappen, können aber beliebig ineinander verschachtelt sein (siehe Abbildung 3-1). Dadurch sind nur Kontrollflußstrukturen mit regelmäßiger Schachtelung möglich. Die Repräsentation von Verzweigungen als Kontrollblöcke führt zur „symmetrischen“ Strukturierung der Kontrollflußgraphen, da die Aufsplittung und Zusammenführung des Kontrollflusses an jeweils einem ausgezeichneten Aktivitätenknoten erfolgen. Dem entsprechend sind Verzweigungs- und Zusammenführungsknoten einander eindeutig zugeordnet. Schließlich bildet auch der Kontrollflußgraph des WF selbst einen Kontrollblock mit wohldefinierter *Start*- und *Endaktivität*.

Insgesamt bilden Kontrollblöcke ein wichtiges Mittel zur Strukturierung von Kontrollflußgraphen. Durch ihre Anwendung ergeben sich nicht nur Vorteile für die WF-Modellierung und -Analyse, sondern – wie wir später noch sehen werden – auch in Bezug auf die Bereitstellung blockbasierter Operationen zur Modifikation von WF-Schemata und -Instanzen. Je besser dabei die Strukturierungsmöglichkeiten des WF-Metamodells sind, desto mehr Unterstützung können die Anwender bei der Erstellung, Analyse und Änderung von WF-Modellen und -Instanzen erwarten. Ein wichtiger Aspekt ist auch die effiziente Durchführung von Korrektheitsüberprüfungen bei der Erstellung und Änderung von WF-Modellen. In jedem Fall bietet ein blockbasierter Beschreibungsansatz gegenüber Formalismen wie Petri-Netzen oder Statecharts verschiedene Verbesserungen, da Arbeitsabläufe übersichtlicher modelliert und Fehler leichter vermieden bzw. erkannt werden können. Die WF-Modellierung kann durch die Bereitstellung syntaxgesteuerter Graphikeditoren [Mich99] unterstützt werden. Dabei ist eine Vergröberung oder Verfeinerung der WF-Modelle unter Ausnutzung der zugrundeliegenden Blockstruktur möglich. Des weiteren muß sich der Modellierer nicht darum kümmern, vorgenommene Aufsplittungen des Kontrollflusses wieder korrekt zusammenzuführen. Ein weiterer Vorteil resultiert aus der Eindeutigkeit von Verzweigungsstrukturen, infolge derer die Zweige einer parallelen oder bedingten Verzweigung eindeutig festliegen. Dadurch lassen sich Operationen wie das Zurücksetzen einzelner Zweige oder das Hinzufügen neuer Zweige einfacher verwirklichen als bei unstrukturierten Modellen. Schließlich, und das ist ein ganz entscheidender Punkt, kann man sich die Blockstrukturierung für die effiziente Analyse und Verifikation von Kontroll- und Datenflußschemata zunutze machen. In zahlreichen Fällen kann die Korrektheit und Konsistenz von WF-Modellen bereits „per Kon-

struktion“ sichergestellt werden, so daß aufwendige Analysen entfallen. Hinzu kommt, daß eindeutig geregelt ist, wann der WF terminiert (Ausführung der Endaktivität). Diese und andere Vorteile des ADEPT-Blockkonzeptes werden im Verlauf dieser Arbeit sukzessive herausarbeitet.

Auf der Basis von Blockkonstrukten wie Sequenz, Parallelverzweigung, bedingte Verzweigung und Schleife lassen sich bereits viele Arbeitsabläufe (bzw. ihre Kontrollflüsse) adäquat modellieren. Allerdings ist die Ausdrucksmächtigkeit dieses Blockansatzes allein noch nicht ausreichend, um wirklich alle relevanten Arbeitsprozesse abbilden zu können. So müssen für die Modellierung komplexer Ablaufstrukturen auch Möglichkeiten vorgesehen werden, mit denen sich die Ausführung von Aktivitäten paralleler Kontrollblöcke synchronisieren läßt. Das bedeutet, daß zwischen einzelnen Aktivitäten verschiedener Parallelzweige (z.B. den Aktivitäten `E` und `H` in Abbildung 3-1) Kontrollabhängigkeiten festlegbar sein müssen. In Abbildung 3-1 etwa kann die Aktivität `H` erst ausgeführt werden, nachdem feststeht, daß `E` entweder beendet worden ist oder aber nicht mehr zur Ausführung kommt. Zur Behandlung von Aktivitätenfehlschlägen (vgl. Abschnitt 3.2.2.3) sollten Rücksprünge im Kontrollfluß vormodellierbar sein (z.B. Rücksprung vor die Bearbeitung von `A` nach Scheitern von `F`). Im allgemeinen muß eine WF-Ausführung in jeden bisher erreichten WF-Zustand zurückgesetzt werden können. Dabei muß es auch möglich sein, in *einzelne* Zweige einer parallelen Verzweigung zurückzuspringen. Sowohl in Verbindung mit Fehlerrücksprüngen als auch in Bezug auf die Synchronisation von Aktivitäten paralleler Kontrollblöcke kann das Prinzip der regelmäßigen Blockstrukturierung nicht aufrechterhalten werden. Um so bedeutsamer ist es, die zur Unterstützung dieser Konzepte notwendigen Erweiterungen des Metamodells so zu gestalten, daß die eingangs genannten Vorteile der Blockstrukturierung bestehen bleiben. ADEPT bietet hierfür spezielle Kantentypen, auf die wir später im Detail eingehen werden.

3.3.2 ADEPT-Konstrukte für die Kontrollflußmodellierung

In diesem Abschnitt werden die wichtigsten Kontrollkonstrukte des ADEPT-Basismodells informell vorgestellt. Dabei berücksichtigen wir zunächst nur elementare Aktivitäten. Der Kontrollfluß eines WF-Modells wird – vereinfacht ausgedrückt – durch einen markierten, attributierten und gerichteten Graphen *(N, E)* repräsentiert, dessen Knotenmenge *N* den WF-Aktivitäten entspricht und dessen Kantenmenge *E* die Kontrollabhängigkeiten zwischen ihnen beschreibt. Die Struktur eines solchen Graphen wird durch das eingangs beschriebene Blockkonzept und die angebotenen Kontrollkonstrukte determiniert. Zur ihrer Beschreibung werden auf Graphebene verschiedene Knoten- und Kantentypen verwendet. Im folgenden sei der Typ eines Knotens durch das Attribut *NT* (engl. *Node Type*) und der Typ einer Kante durch das Attribut *ET* (engl. *Edge Type*) gegeben.

Jeder WF besitzt in ADEPT je eine eindeutige *Start-* und *Endaktivität*. Die Startaktivität wird durch einen Knoten vom Typ *NT* = `STARTFLOW` repräsentiert, der im Kontrollflußgraphen als einziger keinen Vorgänger besitzt. Entsprechend wird die Endaktivität durch einen Knoten vom Typ *NT* = `ENDFLOW` festgelegt, der als einziger ohne Nachfolger ist. Jeder andere Knoten des Kontrollflußgraphen besitzt jeweils mindestens einen Vorgänger- und einen Nachfolgerknoten. Zwischen dem Start- und Endknoten befindet sich der Ablaufkontrollblock, der auf Basis der folgenden Konstrukte, die in Kombination miteinander anwendbar sind, gebildet wird:

- Sequenz
- Verzweigungen (unterschiedlicher Semantik)
- Schleife
- Synchronisationskante
- Fehlerkante
- Priorisierungskante
- Prozeß- und Blockaktivitäten (zur Bildung von Sub-Workflows)

Diese Konstrukte und ihre graphische Repräsentation werden in den folgenden Abschnitten informell eingeführt, wobei wir Fehler- und Priorisierungskanten vorerst ausklammern (siehe Kapitel 6).

3.3.2.1 Sequenz

Die einfachste Form der Koordination einer Menge von Aktivitäten (*NT* = ACTIVITY) (oder auch Kontrollblöcken) ist ihre sequentielle Bearbeitung. Die direkte Hintereinanderausführung zweier Aktivitäten X und Y wird über eine gerichtete *Kontrollkante* $X \rightarrow Y$ des Typs *ET* = CONTROL_E modelliert. Eine *Sequenz* entspricht dann einer nicht-leeren, endlichen Menge von Aktivitätenknoten (oder Kontrollblöcken) $\{A_1, A_2, ..., A_n\}$ mit zugehörigen Kontrollkanten $A_i \rightarrow A_{i+1}$, i = 1,..., n-1. Für ihre Elemente gilt – mit Ausnahme des Start- und des Endknotens –, daß sie jeweils genau eine eingehende und eine ausgehende Kontrollkante, d.h. genau einen Vorgänger- und einen Nachfolgerknoten, besitzen. Bei Start der Sequenz wird die erste Aktivität A_1 aktiviert. Umgekehrt ist die Bearbeitung der Sequenz abgeschlossen, wenn ihre letzte Aktivität A_n beendet wird. Für die Ausführung der Aktivitäten $A_2, ..., A_n$ gilt, daß A_i beendet sein muß, bevor A_{i+1} aktiviert werden darf (i = 2, ..., n). Die serielle Ausführung von Aktivitäten ist transitiv. Für die Verwendung von Kontrollkanten nehmen wir in der Folge stets an, daß die Gesamtmenge an Kanten vom Typ *ET* = CONTROL_E transitiv reduziert ist. Darüber hinaus sind zyklische Reihenfolgebeziehungen auf Basis dieses Kantentyps nicht erlaubt.

Beispiel 3-3 (Sequenz)

In Abbildung 3-2 werden die Aktivitäten A, B und C hintereinander ausgeführt. Um dies zu erreichen, wurden die beiden Kontrollkanten $A \rightarrow B$ und $B \rightarrow C$ eingeführt. Aufgrund der Forderungen nach transitiver Reduktion und Zyklenfreiheit nicht erlaubt wären dann die Kontrollkanten $A \rightarrow C$ oder $C \rightarrow A$.

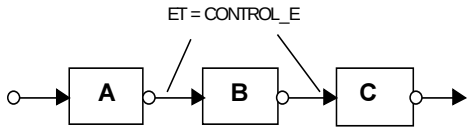


Abbildung 3-2: *Sequenz (ET: Kantentyp)*

Eine Sequenz bildet, ebenso wie die nachfolgend beschriebenen Verzweigungen, einen (logischen) *Kontrollblock* des Kontrollflußgraphen.

3.3.2.2 Verzweigungen

Die Unterstützung rein sequentieller Arbeitsabläufe reicht für anspruchsvolle Anwendungen nicht aus. Ebenso wichtig ist hier die Beschreibbarkeit paralleler und bedingter Aufgabenbearbeitungen, insbesondere wenn die Anzahl der zu koordinierenden WF-Aktivitäten groß ist. Aus diesem Grund unterstützt ADEPT verschiedene Arten von Verzweigungen. Unabhängig vom konkreten Verzweigungstyp gilt dabei, daß sowohl die Aufsplittung als auch die Zusammenführung des Kontrollflusses jeweils an einem ausgezeichneten Aktivitätenknoten erfolgen, d.h. jede Verzweigung besitzt einen eindeutigen Verzweigungsknoten (sog. *Split-Knoten*), einen eindeutigen Synchronisationsknoten (sog. *Join-Knoten*) sowie eine endliche, nichtleere Menge von Bearbeitungszweigen. Letztere werden ihrerseits wieder durch Kontrollblöcke beschrieben. Infolge der Blocksymmetrie ist jedem Split-Knoten ein eindeutiger Join-Knoten zugeordnet und umgekehrt. Die Verknüpfung eines Split-Knotens (Join-Knotens) mit den Startknoten (Endknoten) der einzelnen Zweige erfolgt über normale Kontrollkanten des Typs *ET* = CONTROL_E. Dabei gilt, daß die Anzahl der von einem Split-Knoten ausgehenden Kontrollkanten gleich der Anzahl der am korrespondierenden Join-Knoten einmündenden Kontrollkanten ist. Konkret unterstützt das ADEPT-Basismodell die folgenden drei Verzweigungsarten:

- Parallele Verzweigung (AND-Split / AND-Join)
- Bedingte Verzweigung (OR-Split / OR-Join)
- Parallele Verzweigung mit finaler Auswahl (AND-Split / OR-Join)

a) Parallele Verzweigung

Effiziente Arbeitsabläufe ergeben sich erst dann, wenn voneinander unabhängige Arbeitsschritte parallel bearbeitbar sind. Gegenüber sequentiellen Bearbeitungen kann dadurch die Gesamtvorgangsdauer erheblich verkürzt werden. Abbildung 3-3 a) zeigt, wie *parallele Verzweigungen* in ADEPT modelliert werden. Anhand dieses Beispiels wollen wir kurz die Semantik dieses Beschreibungskonstrukts informell vorstellen. Nach der Beendigung der Verzweigungsaktivität (AND-Split-Knoten A im Beispiel) werden die einzelnen Zweige aktiviert. Das bedeutet, daß ihre jeweiligen Startknoten (B, C und

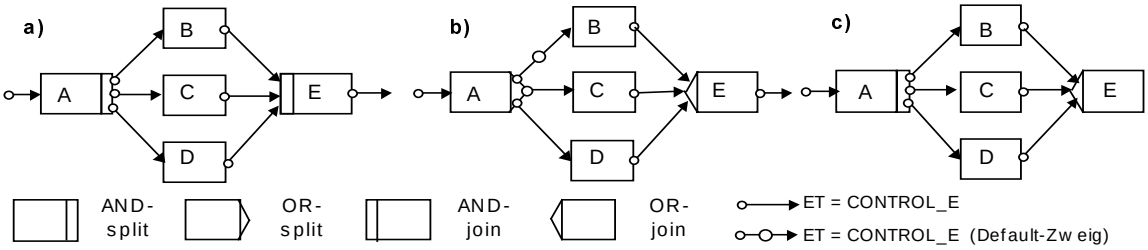


Abbildung 3-3: *(a) Parallele Verzweigung (b) Bedingte Verzweigung (c) Parallele Verzw. mit finaler Auswahl*

D im Beispiel) parallel (ggf. auch überlappend) ausführbar sind. Die Ausführung der Synchronisationsaktivität (AND-Join-Knoten E im Beispiel) wird solange blockiert, bis alle Zweige vollständig abgearbeitet worden sind (im Beispiel wird E nach Beendigung von B, C und D aktiviert).

b) Bedingte Verzweigung

Die graphische Darstellung einer *bedingten Verzweigung* zeigt Abbildung 3-3 b). Im Gegensatz zu parallelen Aufsplittungen des Kontrollflusses wird genau einer der vom OR-Split-Knoten (Knoten A im Beispiel) ausgehenden Zweige (entweder B, C oder D) abgearbeitet, während die anderen nicht zur Ausführung kommen. Dem entsprechend kann der OR-Join-Knoten (Aktivität E im Beispiel) erst nach dem Durchlaufen dieses Zweiges aktiviert werden. Prinzipiell sind bei bedingten Verzweigungen auch „leere“ Zweige – eine Kontrollkante führt direkt vom OR-Split- zum zugehörigen OR-Join-Knoten – zulässig.

Die Auswahl des Ausführungszweiges kann auf zwei unterschiedliche Arten erfolgen: Bei der *impliziten Auswahl* werden, wie im parallelen Fall, zunächst die Startaktivitäten aller Zweige aktiviert und den Akteuren in ihren Arbeitslisten angeboten. (Dies macht nur dann Sinn, wenn die jeweiligen Startknoten der einzelnen Zweige manuelle Aktivitäten beschreiben.) Sobald eine dieser Aktivitäten von einem Anwender zur Bearbeitung ausgewählt wird, ist der Ausführungszweig (implizit) determiniert, und die Aktivitäten der anderen Zweige werden wieder aus den Arbeitslisten entfernt. Bei der *expliziten Auswahl* wird der zu bearbeitende Zweig auf Basis eines dem Verzweigungsknoten zugeordneten *Auswahlparameters DP* (engl. *Decision Parameter*) ausgewählt. Für diesen Auswahlparameter muß zur Laufzeit vom Split-Knoten oder von einem Vorgängerknoten ein Wert festgelegt werden, der eine der ausgehenden Kontrollkanten eindeutig identifiziert. Zu diesem Zweck ist jeder Ausgangskante e eines OR-Split-Knotens ein eindeutiger *Auswahlwert* $SC^e \in Dom^{DP}$ (engl. *Selection Code*) zugeordnet. Um auch dann eine Verzweigungsentscheidung treffen zu können, wenn der Wert von *DP* keinem der Auswahlwerte der von der Verzweigungsaktivität ausgehenden Kontrollkanten entspricht, muß bei der Modellierung einer der Teilzweige als Default-Zweig festgelegt werden (in dem Beispiel aus Abbildung 3-3 b) erfolgt dies durch die Markierung der Kante A → B mit einem Kreis). Für den Rest dieser Arbeit betrachten wir ausschließlich bedingte Verzweigungen mit expliziter Auswahl des Ausführungszweiges. Die dabei getroffenen Aussagen, z.B. in Bezug auf die Korrektheit von Datenflußschemata oder in Bezug auf dynamische Strukturänderungen, lassen sich jedoch ohne Schwierigkeiten auch auf implizite Verzweigungen übertragen.

Einen wichtigen Spezialfall bilden *prädikative Verzweigungen*. Hier besteht die Ausführung des OR-Split-Knotens (oder eines Vorgängerknotens) aus der Auswertung eines Prädikats auf Daten aus dem WF-Datenkontext[3] (siehe unten). Diese Daten entsprechen logisch den Eingaben der Verzweigungsaktivität. Konzeptuell unterscheidet sich dieser Ansatz nur unwesentlich von der Zuordnung von Transitionsbedingungen zu Kontrollkanten[4]. Er erleichtert aber die Formulierung und Überprüfung von Regeln zur Korrektheit von Datenflußschemata erheblich (siehe Kapitel 5). Der wesentliche Vorteil

[3] Die Auswertung einer solchen Bedingung muß in ADEPT nicht notwendigerweise bei Ausführung des OR-Split-Knotens erfolgen. Sie kann auch bereits in einer vorangehenden Aktivität durchgeführt werden.

[4] Auf der Grundlage des ADEPT-Basismodells wurde in [Kirs96] eine Ablaufbeschreibungssprache definiert, bei der den Kontrollkanten sog. Transitionsbedingungen zugeordnet sind. Die erstellten WF-Modelle werden intern in eine Darstellung des ADEPT-Basismodells transformiert.

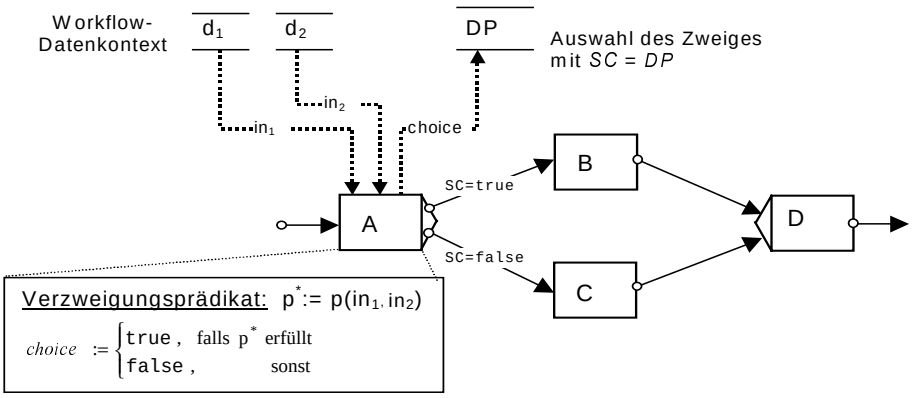


Abbildung 3-4: *Prädikative Verzweigung (choice ist Ausgabeparameter von A)*

ist, daß dieselben Mechanismen wie bei „normalen“ Aktivitäten angewendet werden können, um sicherzustellen, daß alle zur Ausführung (d.h. alle zur Auswertung des Prädikats) benötigten (obligaten) Eingaben versorgt sind (siehe Kapitel 5). Dies wird in heutigen Implementierungen entweder ignoriert, was bei mangelhafter Modellierung zu falschen Verzweigungsentscheidungen oder Laufzeitfehlern führen kann, oder aber es werden unnötige Einschränkungen vorgenommen. So referenziert in dem WfMS MQSeries Workflow (vgl. Kapitel 2) die einer Kontrollkante zugeordnete Transitionsbedingung zum Beispiel immer nur Daten, die Ausgabe der Quellaktivität dieser Kante sind. Übergangsbedingungen, die auf Ausgaben vorangehender Aktivitäten basieren, können bei diesem Ansatz nicht *direkt* formuliert werden.

Die Bildung von Prädikaten selbst basiert in ADEPT im wesentlichen auf den Werten von Eingabeparametern, Konstanten, Vergleichsoperatoren ($<, \leq, =, \geq, >, \neq$) sowie logischen Verknüpfungen ($\wedge$, $\vee$ und $\neg$) einfacher Prädikate[5]. Im folgenden nehmen wir stets an, daß es für prädikative Verzweigungen mindestens eine Wertebelegung für die Eingabeparameter des OR-Split-Knotens gibt, für die das Prädikat erfüllt ist. Prädikative Verzweigungen besitzen jeweils genau zwei Teilzweige und dem entsprechend ihr OR-Split-Knoten genau zwei ausgehende Kontrollkanten mit den Auswahlwerten *True* und *False*. Die Ausführung der Verzweigungsaktivität erfolgt als automatischer Schritt.

Beispiel 3-4 (Prädikative Verzweigung)

Ein Beispiel für eine prädikative Verzweigung zeigt Abbildung 3-4. Hier werden beim Start der Verzweigungsaktivität A zunächst die Eingabeparameter in_1 und in_2 mit Werten aus dem WF-Datenkontext (im Beispiel den aktuellen Werten der WF-Variablen d_1 und d_2) versorgt (siehe hierzu Kapitel 5). Anschließend wird die Aktivität A automatisch ausgeführt, indem das Prädikat p^* ausgewertet wird. Abhängig vom Ergebnis dieser Auswertung wird dem Auswahlparameter *DP* des OR-Split-Knotens entweder der Wert *True* oder der Wert *False* zugeordnet. Dadurch ist der zu aktivierende Zweig (B oder C) determiniert. Im allgemeinen kann der Auswahlparameter *DP* auch von Aktivitäten festgelegt werden, die Vorgänger von A im Kontrollfluß sind.

Wie bereits erwähnt, kann der einem Verzweigungsknoten zugeordnete Auswahlparameter auch durch die Ausgabeparameter externer Anwendungskomponenten (Aktivitätenprogramme) festgelegt werden. Dies kann sinnvoll sein, wenn die Entscheidung über den zu wählenden Ausführungszweig explizit durch einen Akteur vorgenommen werden soll. Für diesen Fall kann die Auswahl auch zwischen mehr als zwei Zweigen erfolgen. Die Modellierung bedingter Verzweigungen erfolgt im ADEPT-Basismodell somit nicht – wie bei einigen prozeßorientierten WfMS üblich – durch Zuordnung von Transitionsbedingungen zu Kontrollkanten, sondern auf Basis von expliziten OR-Split-Knoten. Wie gezeigt, kann dabei der Ausführungszweig auf unterschiedliche Arten festgelegt werden.

c) Parallele Verzweigung mit finaler Auswahl

Ein spezielles Verzweigungskonstrukt, das sowohl Ähnlichkeiten zu parallelen als auch zu bedingten Verzweigungen aufweist, ist die *Parallelverzweigung mit finaler Auswahl*. Ein Beispiel zeigt

[5] Ein einfaches Prädikat p ist eine Bedingung der Form p::= ($param$ θ $const$) mit $const \in \text{pDom}^{param}$ und $\theta \in \{<, \leq, =, \geq, >, \neq\}$.

Abbildung 3-3 c). Hier werden nach der Beendigung des AND-Split-Knotens (A im Beispiel) zunächst alle nachfolgenden Zweige (B, C und D im Beispiel) aktiviert und können dann nebenläufig bearbeitet werden. Im Gegensatz zur parallelen Verzweigung kann am OR-Join-Knoten (Aktivität E im Beispiel) mit der Kontrolle fortgefahren werden, sobald *einer* der Zweige (B, C oder D) vollständig abgearbeitet worden ist. Die anderen noch nicht beendeten Zweige werden dagegen abgebrochen und zurückgesetzt. Dabei können, je nach aktuellem Bearbeitungsstatus dieser Teilzweige, unterschiedliche Aktionen notwendig werden, wie etwa das Entfernen von Einträgen aus Arbeitslisten, das Ändern von Knoten- und Kantenzuständen, die Rücknahme von Schreiboperationen auf Datenelementen des WF-Datenkontexts oder die Ausführung externer Kompensationsschritte. Wir werden auf die einzelnen Maßnahmen später detailliert eingehen. Konzeptuell ist bei dem vorgestellten Konstrukt somit nur die Verzweigungssemantik äquivalent zu der paralleler Verzweigungen, nicht jedoch die Synchronisationssemantik am Join-Knoten. Letztlich bleibt immer genau einer dieser Zweige erhalten, während die anderen abgebrochen, zurückgesetzt und ggf. kompensiert werden. Aus Sicht der Modellausführung gelten sie dann als nicht durchgeführt, was bedeutet, daß die von ihnen erzeugten Daten nicht (mehr) als Eingaben für Folgeaktivitäten in Frage kommen.

Eine wichtige Anwendung dieses Konstrukts ist die Koordination von Aufgabenbearbeitungen, bei denen es zur Erreichung des Aufgabenziels mehrere Handlungsalternativen gibt, die aus Zeitgründen nebenläufig ausgeführt werden sollen. Sobald eine von ihnen zum Erfolg führt, d.h. das Aufgabenziel erreicht wird, sollen die alternativen Bearbeitungen abgebrochen und im Ablauf fortgefahren werden.

Beispiel 3-5 (Parallele Verzweigung mit finaler Auswahl)

Innerhalb eines Software-Entwicklungsprozesses (SEP) stellt die „Fehlersuche“ eine wichtige Funktion dar. Für zeitkritische SEPs kann es notwendig sein, bei Auftreten eines Software-Fehlers mehrere unabhängige Prüfstellen mit der Durchführung der Fehlersuche zu beauftragen. Sobald der Fehler von einer Stelle entdeckt wird, kann die Fehlersuche bei den anderen Prüfstellen abgebrochen und im SEP fortgefahren werden.

In Kapitel 6 wird gezeigt, daß sich mit dem vorgestellten Konstrukt – in Kombination mit anderen Kontrollkonstrukten – auch bestimmte Ausnahmen bzw. Abweichungen vom geplanten Ablauf (z.B. Vorwärtssprünge) vormodellieren lassen.

Bei der oben beschriebenen Synchronisationssemantik „gewinnt“ immer der *zuerst* beendete Zweig. Eine solche *implizite* Entscheidung ist jedoch nicht immer ausreichend. Eine Alternative besteht darin, die Auswahl des „Siegerzweiges“ durch einen Anwender vornehmen zu lassen, z.B. abhängig von den in den Teilzweigen produzierten Daten, d.h., es werden zunächst mehrere Teilzweige bearbeitet und anschließend einer von ihnen *explizit* als Siegerzweig bestimmt. Dies erfolgt über eine spezielle Auswahlaktivität, die berechtigten Anwendern angeboten wird, sobald ein Teilzweig durchlaufen ist.

Beispiel 3-6 (Explizite Auswahl des Siegerzweiges)

In einem Klinikum werden bestimmte medizinische Leistungen (z.B. Untersuchungen) häufig von mehreren Stellen erbracht. Im Kontext der Behandlung eines Patienten kann es sinnvoll sein, für die Durchführung einer bestimmten Leistung verschiedene Terminvorschläge von mehreren dieser Stellen einzuholen und dann den aus Sicht des Behandlungsprozesses günstigsten Termin zu wählen.

Wie oben beschrieben, werden die für die nachfolgenden Aktivitäten relevanten Informationen durch den Siegerzweig bestimmt, während die nicht gewählten Teilzweige abgebrochen und zurückgesetzt werden. Die Möglichkeit der expliziten Auswahl bringt daher gegenüber der impliziten Festlegung weder konzeptuell noch implementierungstechnisch neue Probleme mit sich. Wir werden sie aus Gründen der besseren Lesbarkeit bei der Formalisierung des Metamodells und der Festlegung seiner Ausführungssemantik nicht weiter berücksichtigen.

3.3.2.3 Schleifen

Bisher haben wir lediglich azyklische Kontrollflußgraphen betrachtet. Wie das folgende Beispiel zeigt, muß ein WF-Metamodell auch die Abbildung von Schleifen gestatten.

Beispiel 3-7 (Zyklische Prozesse)

In Krankenhäusern gibt es langandauernde Behandlungsprozesse, die in mehrere Behandlungszyklen unterteilt sind [RKD96]. Die einzelnen Zyklen laufen im Normalfall identisch ab, und können ihrerseits bereits eine kom-

plexe Ablaufstruktur besitzen. Ein Beispiel ist der Ablauf einer Chemotherapie, der mehrere Behandlungszyklen umfaßt und sich über einen Zeitraum von mehreren Monaten erstrecken kann.

Schleifen werden von den Metamodellen heutiger WfMS entweder gar nicht oder nur indirekt unterstützt (vgl. Kapitel 2). Letzteres ist bei Petri-Netz-basierten WF-Beschreibungssprachen der Fall. Hier werden iterative Aufgabenbearbeitungen auf Basis derselben Konstrukte (Stellen, Transitionen und Pfeile) modelliert, wie sie zur Beschreibung von Sequenzen und Verzweigungen eingesetzt werden. Eine solche Minimierung bzgl. der Anzahl an Beschreibungselementen ist für bestimmte Anwendungen (z.B. [Genr91, Jens90, Ober90]) zwar ausreichend, erweist sich aber in Bezug auf die Modellierung und Ausführung von Workflows als problematisch. Ein wichtiger, oftmals vernachlässigter Aspekt ist die mangelhafte Lesbarkeit bzw. Verständlichkeit der WF-Modelle. Bereits bei etwas komplexer strukturierten WF-Modellen ist für die Anwender auf graphischer Ebene nicht unmittelbar erkennbar, ob die Modelle Zyklen enthalten oder nicht. Ebensowenig sind direkte Aussagen darüber möglich, ob ein Zyklus eine Schleife modelliert oder ob er das Resultat einer fehlerhaften Modellierung ist. – Ein solch impliziter Beschreibungsansatz erschwert z.B. dynamische WF-Anpassungen zur Laufzeit. Des weiteren ist bei Petri-Netz-basierten Ansätzen beim Schalten einer Transition nicht ohne weiteres erkennbar, ob durch dieses Ereignis ein bereits zuvor durchlaufener Bereich erneut betreten wird oder nicht. Um dies festzustellen, müßten aufwendige Analysen durchgeführt und zusätzliche Laufzeitdaten verwaltet werden. Dadurch ließen sich fortschrittlichere Änderungsmöglichkeiten, wie das Zurücknehmen temporärer WF-Änderungen bei wiederholter Ausführung einer Schleife (siehe Kapitel 8), nicht adäquat und mit vertretbarem Aufwand verwirklichen.

Das ADEPT-Basismodell stellt ein explizites Schleifenkonstrukt bereit. Eine Schleife wird, ebenso wie Sequenzen oder Verzweigungen, als (logischer) Kontrollblock aufgefaßt. Sie besteht aus je einem eindeutigen Start- und Endknoten (Knotentypen `STARTLOOP` bzw. `ENDLOOP`), die über eine Schleifenrücksprungkante (Kantentyp `LOOP_E`) miteinander verknüpft sind. Zwischen diesen beiden Knoten liegt der Schleifenkörper, der seinerseits durch einen (geschachtelten) Kontrollblock beschrieben ist. Durch diesen Ansatz sind auch ineinander geschachtelte Schleifen modellierbar. Dem Startknoten einer Schleife kann optional eine interne Aktion (z.B. Initialisierung eines Schleifenzählers) zugeordnet sein, die bei erstmaligem Betreten der Schleife ausgeführt wird. Der Endknoten wird mit einer Schleifenbedingung verknüpft, die nach jedem Durchlauf ausgewertet wird – der Endknoten einer Schleife besitzt, wie andere Aktivitätenknoten auch, Eingabeparameter. Abhängig vom Ergebnis einer Auswertung wird die Schleife entweder erneut durchlaufen – auslösendes Ereignis aus Sicht der Kontrollflußsteuerung ist das Signalisieren der Schleifenrücksprungkante – oder aber sie wird verlassen. Die graphische Darstellung von Schleifen zeigt das Beispiel aus Abbildung 3-5. Zu erwähnen bleibt, daß es sich bei Schleifenanfangs- bzw. Schleifenendkonten um spezielle Knotentypen und bei Schleifenrücksprungkanten um einen speziellen Kantentyp handelt. Wie diese Information bei der Kontrollflußsteuerung konkret genutzt wird, zeigen wir in Abschnitt 4.3.3.

Um das ADEPT-Basismodell überschaubar zu halten, gibt es keine weiteren Schleifenkonstrukte. Allerdings lassen sich mit obigem Ansatz und den zuvor vorgestellten Konstrukten die von Programmiersprachen her bekannten Semantiken für Schleifen (`REPEAT-UNTIL`, `WHILE-DO`, `FOR-TO`) problemlos ausdrücken. Eine `WHILE-DO`-Schleife kann z.B. durch kombinierte Anwendung einer bedingten prädikativen Verzweigung und dem vorgestellten Schleifenkonstrukt formuliert werden. Die Semantik einer `FOR`-Schleife ergibt sich, wenn die mit Start- und Endknoten assoziierten Aktionen geeignet gewählt werden. Im Beispiel aus Abbildung 3-6 werden zu diesem Zweck von ADEPT bereitgestellten Systemfunktionen zur Initialisierung und Inkrementierung einer WF-Variablen verwendet.

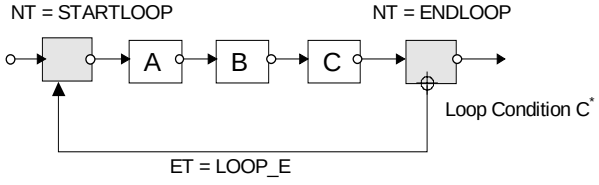


Abbildung 3-5: *Schleife*

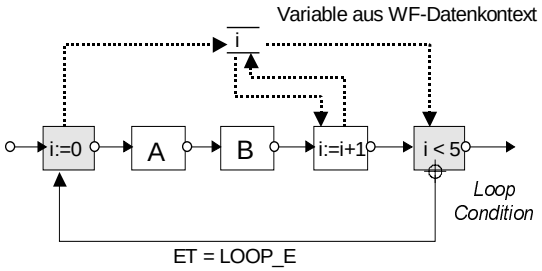


Abbildung 3-6: *FOR-Schleife*

Wir werden im Verlauf dieser Arbeit sehen, daß die Unterstützung von Schleifen in Verbindung mit dynamischen WF-Änderungen und Ausnahmebehandlungen essentielle Fragestellungen aufwirft, die in bisherigen Ansätzen, infolge des Fehlens dieses Konstrukts oder aufgrund fehlender Möglichkeiten zur Durchführung dynamischer Strukturänderungen, nicht erkannt bzw. nicht behandelt worden sind.

3.3.2.4 Synchronisationskanten

Durch die kombinierte und geschachtelte Anwendung der bisher vorgestellten Kontrollkonstrukte (Sequenz, parallele und bedingte Verzweigungen, Schleife) lassen sich zahlreiche Arbeitsabläufe adäquat beschreiben. Allerdings gibt es in anspruchsvollen Anwendungsumgebungen [RKD96, RKD97] auch Prozesse, für deren Modellierung die Ausdrucksmächtigkeit dieser Konstrukte entweder noch nicht ausreichend ist oder ihre Anwendung zu schwer verständlichen Modellen führen würde. Insbesondere können für die Ausführung von Aktivitäten unterschiedlicher paralleler Teilzweige gewisse Einschränkungen hinsichtlich ihrer Ausführungsreihenfolge bestehen.

Wir wollen dies an zwei einfachen Beispielen illustrieren. Angenommen für den in Abbildung 3-7 a) dargestellten Prozeßgraphen soll zusätzlich gelten, daß die Aktivität B beendet sein muß, bevor der Schritt D gestartet werden darf. Um diese Kontrollabhängigkeit abbilden zu können, reichen die bisher vorgestellten Blockkonstrukte nicht aus. Aus diesem Grund bietet das ADEPT-Basismodell mit den sog. *Synchronisationskanten* (kurz: *Sync-Kanten*) weitere Beschreibungselemente an, mit denen sich Reihenfolgebeziehungen zwischen Knoten paralleler Zweige modellieren lassen. Die Verwendung dieses Konstrukts zur Beschreibung der Kontrollabhängigkeit „B vor D“ zeigt Abbildung 3-7 b). Man kann leicht zeigen, daß es zu diesem Kontrollflußgraphen keine „äquivalente“ Modellierung gibt, die ohne Synchronisationskanten auskommt, d.h. die ausschließlich auf Basis der zuvor vorgestellten Blockkonstrukte erstellt werden kann.

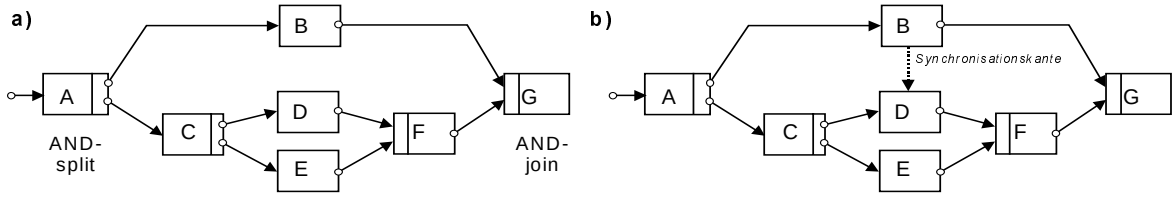


Abbildung 3-7: *Höhere Ausdrucksmächtigkeit durch Verwendung einer Sync-Kante*

Wie das folgende Beispiel zeigt, kann die Verwendung von Sync-Kanten auch aus einem anderen Grund sinnvoll sein. Angenommen in den WF-Graphen aus Abbildung 3-8 a) soll eine Aktivität X eingefügt werden, die erst aktiviert werden darf, wenn – abhängig vom gewählten Ausführungszweig – B, D oder F beendet worden ist (X soll vor Start von H beendet sein). Abbildung 3-8 b) zeigt, wie dieser Sachverhalt unter alleiniger Verwendung der Blockkonstrukte modelliert werden kann. (Die „Nullaktivitäten“ werden hier benötigt, um die geforderte Blockstrukturierung aufrecht zu erhalten. Hierbei handelt es sich um Knoten ohne assoziierte Aktion, die sofort nach ihrer Aktivierung beendet werden.)

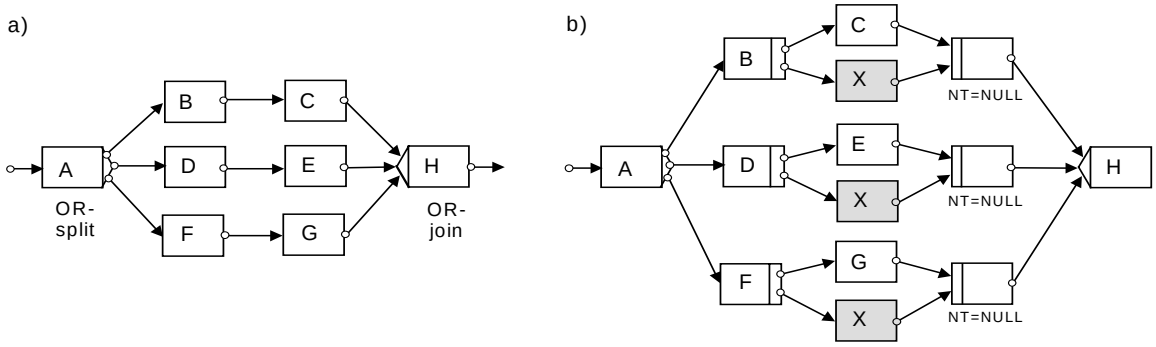


Abbildung 3-8: *Komplexe Modellierung bei ausschließlicher Verwendung der Blockkonstrukte*

Wie man leicht sieht, muß X in jeden der 3 Teilzweige eingefügt werden, obwohl es sich in allen Fällen semantisch um dieselbe Aktivität handelt. Für komplexer strukturierte Modelle würde sich deshalb

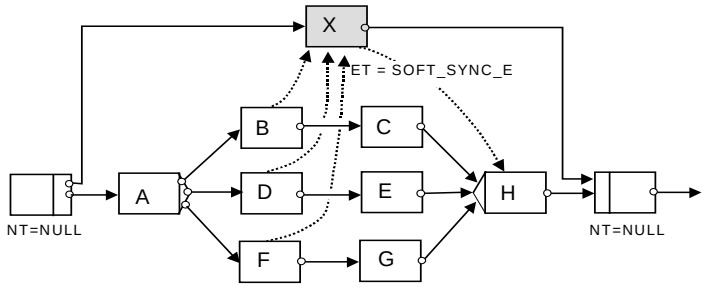


Abbildung 3-9: *Kompakte Modellierung durch Verwendung von Sync-Kanten*

die Definition entsprechender Änderungen schwierig gestalten. Abgesehen davon würden schwer verständliche und kaum mehr wartbare Modelle resultieren. Aus diesen Gründen kann in ADEPT die Aktivität X durch einen einzigen Knoten repräsentiert werden, der parallel zu dem durch (A, H) gebildeten Verzweigungsblock in den WF-Graphen eingebettet wird und dessen Reihenfolgebeziehungen zu den Knoten B, D, F und H über Sync-Kanten modelliert werden (vgl. hierzu Abbildung 3-9). Wie man leicht sieht, wird zur Ausführungszeit nur eine der Aktivitäten B, D oder F bearbeitet.

Das ADEPT-Basismodell unterscheidet zwei Arten von Sync-Kanten, die unterschiedliche Semantiken von „Wartet-Auf"-Beziehungen in nebenläufigen Ausführungen festlegen:

- Eine *„einfache" Sync-Kante* X → Y (*ET* = SOFT_SYNC_E) beschreibt eine *Verzögerung* zwischen zwei Aktivitäten X und Y: Y darf frühestens dann aktiviert bzw. ausgeführt werden, wenn X entweder zuvor erfolgreich beendet wird oder feststeht, daß X nicht mehr zur Ausführung kommt.
- Eine *„strikte" Sync-Kante* X → Y (*ET* = STRICT_SYNC_E) beschreibt eine *Erfolgsabhängigkeit* zwischen den Aktivitäten X und Y. Das bedeutet, daß Y nur dann aktiviert bzw. ausgeführt werden darf, wenn X zuvor erfolgreich beendet worden ist.

Im Falle einer einfachen Sync-Kante X → Y ist die erfolgreiche Beendigung von X nicht unbedingt notwendig, um Y ausführen zu können. Bei Verwendung einer strikten Sync-Kante X → Y dagegen ist sie zwingend für die Ausführung von Y. Wichtige Anwendungen für Sync-Kanten bei der WF-Definition sind, neben der Modellierung gewünschter Reihenfolgebeziehungen, die Unterstützung von Datenflüssen zwischen parallelen Ausführungszweigen (vgl. Kapitel 5) oder die explizite Synchronisation von parallelen Aktivitäten zur Vermeidung nebenläufiger Schreibzugriffe auf Daten aus dem WF-Datenkontext. Wie das Beispiel aus Abbildung 3-9 zeigt, ermöglicht die Verwendung einfacher Sync-Kanten zudem die kompakte Modellierung von WFs. Des weiteren wird in Kapitel 7 gezeigt, daß Sync-Kanten zur Festlegung dynamischer WF-Änderungen nützlich sein können.

Abbildung 3-10 a) zeigt ein typisches Beispiel für die Verwendung einer *einfachen* Sync-Kante (hier D → G mit Kantentyp *ET* = SOFT_SYNC_E). Die Aktivitäten D und G liegen in verschiedenen Zweigen der durch (A, H) gebildeten Parallelverzweigung. Des weiteren ist D ein Teilzweig einer bedingten Verzweigung, kommt also nicht in jedem Fall zur Ausführung. Die Aktivierung von G erfolgt unter diesen Voraussetzungen erst dann, wenn die Aktivität F beendet ist <u>und</u> wenn die Aktivität D entweder beendet ist oder aber feststeht, daß D nicht mehr aktiviert werden kann. Letzteres ist der Fall, wenn bei Beendigung der Verzweigungsaktivität B der Teilzweig C zur Ausführung ausgewählt wird. Die Verwendung einer strikten Sync-Kante (Kante D → G mit *ET* = STRICT_SYNC_E) wäre im vorliegenden Beispiel hingegen nicht zulässig, da bei Nichtauswahl des Zweiges D die Aktivität G (und damit auch H) überhaupt nicht mehr zur Ausführung kommen könnte.

Ein Beispiel für die Verwendung einer *strikten* Sync-Kante zeigt Abbildung 3-10 b). Die Quellaktivität C der Kante C → G (mit *ET* = STRICT_SYNC_E) bildet hier einen Teilzweig der bedingten Verzweigung (B, E) und wird deshalb nicht immer ausgeführt. Aufgrund der Semantik dieses Kantentyps ist die Ausführbarkeit des Zielknotens G entsprechend eingeschränkt. Da G einen Teilzweig der durch (F, I) festgelegten Parallelverzweigung (mit finaler Auswahl) bildet, kann dieser Knoten nur aktiviert werden, nachdem F und C erfolgreich abgeschlossen worden sind und bevor I aktiviert wird. Wird H

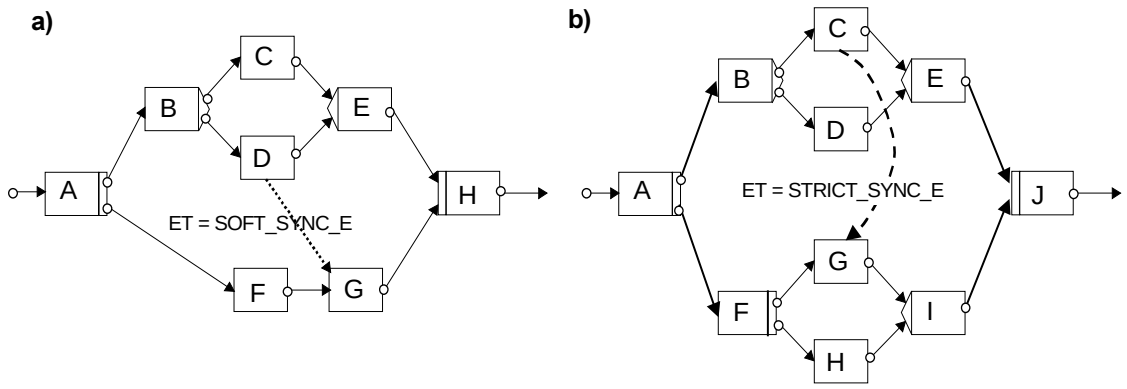


Abbildung 3-10: *Einfache Sync-Kante (a) und strikte Sync-Kante (b)*

allerdings vor der Beendigung von C erfolgreich abgeschlossen (und dadurch I aktiviert), kann G nicht mehr zur Ausführung kommen, unabhängig davon ob C ausgeführt wird oder nicht. In keinem Fall kann G aktiviert werden, wenn zur Laufzeit für die Verzweigung (B, E) der untere Teilzweig (mit dem Knoten D) gewählt wird. – Man kann zeigen, daß es für den vorliegenden Prozeßgraphen keine äquivalente Modellierung gibt, die allein auf Basis der vorgestellten Blockkonstrukte und einfachen Sync-Kanten vorgenommen werden kann.

Diese Beispiele deuten bereits an, daß Sync-Kanten nicht uneingeschränkt verwendbar sind. Ansonsten könnte es während der WF-Ausführung zu (ungewollten) Verklemmungssituationen kommen, in denen keine Aktivität mehr ausführbar ist. Im Beispiel aus Abbildung 3-10 a) wäre eine solche Verklemmung möglich, wenn es sich bei $D \rightarrow G$ um eine strikte Sync-Kante handeln würde oder wenn der dargestellte Kontrollflußgraph zusätzlich eine einfache oder strikte Sync-Kante $G \rightarrow B$ enthalten würde. Welche Restriktionen im ADEPT-Basismodell im allgemeinen für die Verwendung von Sync-Kanten bestehen und wie sich fehlerhafte Modelle ausschließen lassen, wird in den nachfolgenden Abschnitten gezeigt. In jedem Fall dürfen zwei Aktivitätenknoten nur dann über eine Sync-Kante miteinander verknüpft werden, wenn sie in unterschiedlichen Zweigen einer parallelen Verzweigung liegen. Des weiteren darf die Verwendung von Sync-Kanten niemals zu Zyklen im Kontrollflußgraphen führen, da es für diesen Fall zu Verklemmungen während der WF-Ausführung kommen kann. Wir werden in Kapitel 4 zeigen, daß bei der Verwendung strikter Sync-Kanten aufwendigere Verfahren für die Analyse des dynamischen Verhaltens von WF-Modellen erforderlich werden.

Zusammenfassend kann festgehalten werden, daß mit der Einführung von Sync-Kanten das WF-Metamodell an Ausdrucksmächtigkeit gewinnt. Bei Verwendung dieses Konstrukts erhöht sich aber der Umfang und die Komplexität der Analysen, die zur Sicherstellung der Korrektheit von Kontrollflußgraphen notwendig sind. Hilfreich für die effiziente Durchführung dieser Analysen ist die explizite Unterscheidbarkeit zwischen verschiedenen Knoten- und Kantentypen. So können für Kontrollflußgraphen, die ausschließlich normale Kontrollkanten umfassen, Zyklensuchen vollständig entfallen, da für diesen Fall „ungewollte“ Zyklen per Konstruktion ausgeschlossen sind. Andererseits kann auch bei der Verwendung von Konstrukten wie Sync-Kanten die Blockstrukturierung von Kontrollflußgraphen weiter für verschiedene Analyse- oder Änderungsoperationen genutzt werden. Dies wäre nicht der Fall, wenn anstelle von Kontroll-, Schleifenrücksprung- und Sync-Kanten nur noch ein Kantentyp für die Modellierung zur Verfügung stünde.

3.3.2.5 Kombinierte Anwendung der Konstrukte

Abbildung 3-11 zeigt die kombinierte Anwendung der verschiedenen Kontrollkonstrukte. Der dargestellte Kontrollflußgraph besitzt je eine Start- und Endaktivität. Zwischen diesen beiden Knoten liegt eine Sequenz, die sich aus der Aktivität A, dem durch die Schleife (L_S, L_E) gebildeten Kontrollblock und der Aktivität J konstituiert. Der Schleifenkörper selbst bildet ebenfalls wieder einen Kontrollblock, der seinerseits noch weiter geschachtelt wird. Insgesamt zeigt dieses Beispiel, wie Kontrollflußgraphen in ADEPT strukturiert werden und aus welchen Bestandteilen sie sich konstituieren. Dabei dient ein Kontrollblock vor allem der logischen Strukturierung von Kontrollflußgraphen. Seine innere Struktur ist auch für Aktivitäten sichtbar, die außerhalb des Blocks liegen. Dieser Aspekt ist

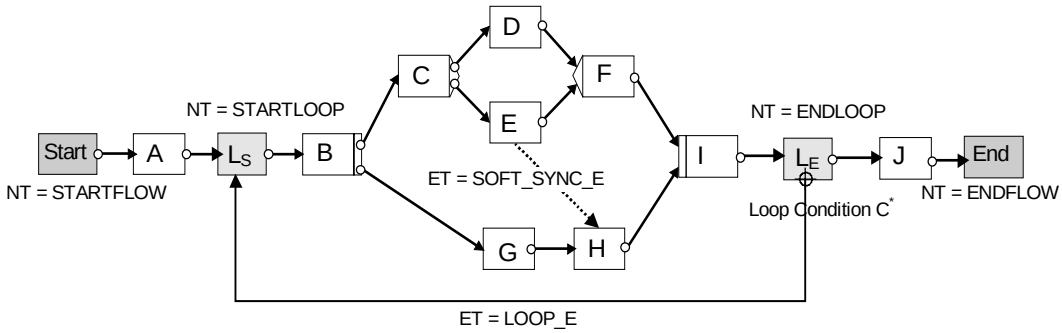


Abbildung 3-11: *Kombinierte Anwendung von ADEPT-Konstrukten*

unter anderem für die Datenflußmodellierung wichtig (siehe Kapitel 5). Die Bildung von Kontrollblöcken erfolgt nach bestimmten Strukturierungsregeln und basiert auf der Verwendung unterschiedlicher Knoten- und Kantentypen. Darunter leidet die Verständlichkeit der WF-Modelle aber keineswegs. Vielmehr wird es dadurch erst möglich, z.B. durch das Ein- und Ausblenden einzelner Kantentypen oder die Vergröberung von Kontrollblöcken, mehr Übersichtlichkeit zu erzielen. Des weiteren ergeben sich aus der Trennung von Knoten- und Kantentypen unterschiedlicher Semantik zahlreiche Vorteile, die in der Folge noch sukzessive aufgezeigt werden.

3.3.2.6 Hierarchische Strukturierung durch Prozeß- und Blockaktivitäten

Um komplexe Arbeitsabläufe angemessen strukturieren zu können, unterstützt das ADEPT-Basismodell – neben dem vorgestellten logischen Blockkonzept – die hierarchische Dekomposition von WF-Modellen. Dabei können, ähnlich wie bei SA/RT-Ansätzen [Biec96], die einzelnen WF-Aktivitäten bei der Modellierung schrittweise verfeinert werden. Zu diesem Zweck kann ihnen jeweils die Vorlage eines Sub-WF zugeordnet werden, dessen Schritte ggf. weiter verfeinert werden. Insgesamt ergibt sich durch diese Vorgehensweise eine hierarchische Anordnung der Aktivitäten bzw. ihrer Vorlagen. Den WF, der die oberste Hierarchiestufe bildet, bezeichnen wir auch als *Toplevel-Workflow*. Seine Vorlage besitzt, wie jede andere Prozeßvorlage, eine Schnittstelle und ein Schema (WF-Graph). Letzteres beschreibt den eigentlichen Prozeß, konstituiert sich also aus einer Menge komplexer oder elementarer Aktivitäten sowie den zwischen ihnen bestehenden Kontroll- und Datenflußbeziehungen. Eine komplexe Aktivität repräsentiert entsprechend einen Sub-WF, der durch eine separate Prozeßvorlage festgelegt ist. Elementare Aktivitäten sind dagegen nicht mehr weiter unterteilbare Arbeitseinheiten, die – mit Ausnahme von Nullaktivitäten – mit einer ausführbaren Aktion verknüpft werden.

Für die Definition einer komplexen Aktivität innerhalb einer Prozeßvorlage π gibt es im ADEPT-Basismodell zwei unterschiedliche Möglichkeiten:

1. *Prozeßaktivitäten*: Dem betreffenden Aktivitätenknoten aus π wird eine globale, wiederverwendbare Prozeßvorlage $\omega_{sub} \in WT$ (mit $vUsage^{\omega_{sub}}$ = GLOBAL_REUSABLE) aus dem Repository zugeordnet – wir bezeichnen entsprechende Aktivitäten im folgenden als *Prozeßaktivitäten*. Dabei ist die innere Struktur bzw. das Schema dieser globalen Vorlage für die übergeordnete Ebene, d.h. für die Prozeßvorlage π, selbst nicht sichtbar. Die hierarchische Dekomposition erfolgt durch die Wiederverwendung existierender Prozeßvorlagen. Je nach Kontext, kann eine solche wiederverwendbare Prozeßvorlage einen Top-Level- oder einen Sub-Workflow festlegen.

2. *Blockaktivitäten*: Es wird eine neue, nur im Kontext von π gültige Sub-WF-Vorlage $\omega_{sub}^* \in WT$ definiert (mit $vUsage^{\omega_{sub}^*} \in$ {LOCAL_REUSABLE, NON_REUSABLE}) und dem betreffenden Aktivitätenknoten zugeordnet – wir bezeichnen entsprechende Aktivitäten im folgenden als *Blockaktivitäten*. Die Vorlage einer Blockaktivität ist weder als Top-Level-WF instantiierbar, noch kann sie zur Definition der Aktivitäten anderer Prozeßvorlagen verwendet werden. Wie im Fall von Prozeßaktivitäten bleibt ihre innere Struktur für die jeweils übergeordnete Ebene verborgen. Allerdings können im ADEPT-Ansatz Blockaktivitäten ggf. dynamisch gebildet bzw. aufgelöst werden, was im Kontext bestimmter Ausnahmebehandlungen nützlich sein kann.

Durch diesen Top-Down-Ansatz ergeben sich zum einen verbesserte Möglichkeiten zur Strukturierung komplexer Workflows, zum anderen wird die Wiederverwendung von Prozeßvorlagen bei der Aktivi-

tätendefinition unterstützt. Vom Prinzip her bestehen Ähnlichkeiten zu den Konzepten blockorientierter Programmiersprachen. Insbesondere kann es durch die Verwendung von Prozeß- und Blockaktivitäten auch Prozeßvariablen mit unterschiedlichem Scope, d.h. unterschiedlicher Sichtbarkeit, geben (vgl. Kapitel 5). Zu erwähnen bleibt, daß bei der Schachtelung von WF-Modellen – im Gegensatz zu den meisten Programmiersprachen – rekursive Aufrufstrukturen ausgeschlossen werden (siehe hierzu Abschnitt 3.7).

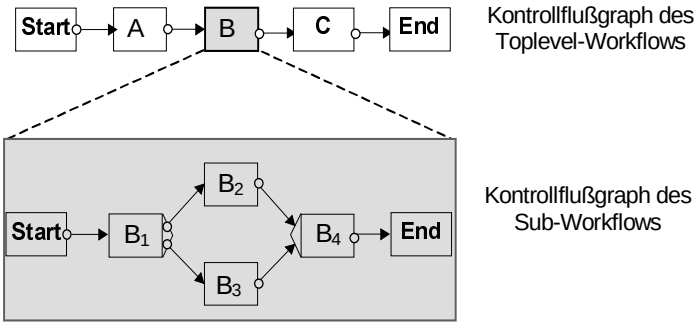


Abbildung 3-12: *Hierarchische Dekomposition von Kontrollflußgraphen*

3.3.2.7 Ergänzende Bemerkungen

Aus verschiedenen Gründen ist es nicht immer möglich bzw. sinnvoll, alle Schritte eines Arbeitsablaufs zu automatisieren, d.h. durch die Bereitstellung entsprechender Anwendungskomponenten zu unterstützen. Beispiele sind manuelle Tätigkeiten, wie z.B. die Beratung eines Kunden oder die körperliche Untersuchung eines Patienten. Des weiteren kann bei der Einführung vorgangsorientierter Anwendungssysteme nicht immer davon ausgegangen werden, daß alle beteiligten Bereiche bzw. deren Funktionen in computerunterstützte Abläufe eingebunden werden können. Dennoch kann es sich bei der Abbildung eines semantischen Geschäftsprozeßmodells [Sche98a, Sche98b] auf ein WF-Modell als sinnvoll erweisen, auch für solche (vorerst) nicht automatisierbaren Arbeitsschritte entsprechende Platzhalter (inkl. Schnittstellenbeschreibungen) im WF-Modell vorzusehen. In ADEPT ist dies problemlos möglich, da auch Aktivitäten bzw. Aktivitätenvorlagen modellierbar sind, die mit keiner Aktion verknüpft werden. Eine solche Vorgehensweise erleichtert den späteren Übergang von manuellen zu automatisierten Aktivitäten bzw. Teilprozessen und sie trägt zu einer Verringerung der Diskrepanz zwischen realem Arbeitsablauf und WF-Modell bei. Dieser Aspekt ist für dynamische WF-Änderungen, die üblicherweise von Anwendern vorgenommen werden müssen, sehr wichtig. Die Darstellung manueller Arbeitsschritte im Modell schafft auch die Grundlage für die Nutzung weitergehender Systemfunktionen, wie der zeitlichen Überwachung von Aktivitätenbearbeitungen [Grim97].

Bisher haben wir zwei wichtige Beschreibungselemente des ADEPT-Basismodells ausgeklammert, die für die Modellierung von Vorwärts- und Rückwärtssprüngen innerhalb von Workflows nützlich sind: Fehler- und Priorisierungskanten. Wir werden sie in Kapitel 6, im Zusammenhang mit Konzepten zur Vormodellierung planbarer Abweichungen, einführen. Auf Basis der vorgestellten Konstrukte und dieser Erweiterungen lassen sich komplexe Arbeitsabläufe angemessen beschreiben.

3.4 Formale Repräsentation von Kontrollflußgraphen

In Abschnitt 3.3 haben wir verschiedene Kontrollflußkonstrukte des ADEPT-Basismodells informell eingeführt und ihre graphische Darstellung anhand von Beispielen illustriert. In diesem Abschnitt folgt die formale Beschreibung von Kontrollflußgraphen und ihrer strukturellen („statischen“) Eigenschaften. Nur formale Repräsentationen erlauben wirklich präzise Aussagen über das zukünftige Systemverhalten sowie über die Korrektheit der WF-Modelle. Einerseits ist dies für die statische Analyse und Verifikation von WF-Modellen nützlich, etwa um frühzeitig Modellierungsfehler aufdecken bzw. vermeiden zu können, andererseits schafft ein formaler Ansatz erst die Grundlage dafür, WF-Modelle bzw. -Instanzen abändern zu können, ohne daß es dadurch zu Dateninkonsistenzen oder Programmfehlern kommen kann. Auch die Semantik von Änderungsoperationen läßt sich, von formalen Beschreibungen ausgehend, präziser fassen (vgl. Kapitel 7).

3.4.1 Mengenbasierte Darstellung

Für die formale Repräsentation von Kontrollflußgraphen verwenden wir eine mengenbasierte Darstellung: Kontrollflußgraphen werden auf Mengen von Knoten und Kanten unterschiedlichen Typs abgebildet. Darauf basierend formulieren wir in Abschnitt 3.5 Regeln zur korrekten Strukturierung der Graphen. Diese Regeln werden von uns in Kapitel 4 zur Ableitung dynamischer Eigenschaften herangezogen. Formal definieren wir Kontrollflußgraphen wie folgt:

Definition 3-1 (Kontrollflußgraphen, KF-Graphen)

Ein ADEPT-Kontrollflußgraph (kurz: KF-Graph) ist ein Tupel

$$\boldsymbol{CFS = (N, E, D, NT, NP, V_{out}, V_{in}, DP, EC, Template)}$$

mit Knotenmenge N, Kantenmenge E und Datenelementen D sowie den Abbildungen:

NT: $N \mapsto$ *NodeTypes* (*Knotentypen*)

NP: $N \mapsto$ *Priority* (*Knotenprioritäten* der Aktivitätenknoten)

V_{out}: $N \mapsto$ *OutBehaviour* (*Ausgangssemantik* der Knoten)

V_{in}: $N \mapsto$ *InBehaviour* (*Eingangssemantik* der Knoten)

DP: $N \mapsto D \cup$ {`UNDEFINED`} (*Entscheidungsparameter* bei bedingten Verzweigungen)

EC: $E \mapsto$ *EdgeCode* $\cup$ {`UNDEFINED`} (*Auswahlcode* der Kanten bei bedingten Verzweig.)

Template: $N \mapsto$ *Templates* $\cup$ {`NONE`} (Zugeordnete *Aktivitätenvorlage* der Knoten)

Die einzelnen Komponenten von *CFS* (*Control Flow Schema*) haben dabei die folgende Bedeutung:

- N („Nodes“) ist eine endliche Menge von eindeutigen (Aktivitäten-) Knotenbezeichnern
- $E \subseteq N \times N \times$ *EdgeTypes* („Edges“) ist eine endliche Menge von gerichteten Kanten. *EdgeType* definiert dabei die Menge der möglichen Kantentypen:

  EdgeTypes:= { `CONTROL_E`, `LOOP_E`, `SOFT_SYNC_E`, `STRICT_SYNC_E`, `FAILURE_E`, `PRIORITY_E` }

  Für die Definition von Kontrollflußbeziehungen zwischen Aktivitäten gibt es sechs verschiedene Kantentypen. Kontrollkanten des Typs `CONTROL_E` legen normale Ausführungsreihenfolgen zwischen Aktivitäten fest. Für die Modellierung von Schleifenrücksprüngen werden Kanten des Typs `LOOP_E` verwendet. Die Synchronisation paralleler Aktivitäten wird durch einfache oder strikte Sync-Kanten mit Kantentyp `SOFT_SYNC_E` bzw. `STRICT_SYNC_E` beschrieben. Bisher noch nicht eingeführt haben wir Fehler- und Priorisierungskanten. Erstere haben den Typ `FAILURE_E` und dienen der Modellierung ausnahmebedingter Rücksprünge im Kontrollfluß. Priorisierungskanten – sie definieren keine Kontrollabhängigkeiten im eigentlichen Sinne – besitzen den Typ `PRIORITY_E` und können zur Festsetzung von Ausführungsprioritäten für Aktivitäten verwendet werden. Die Bedeutung dieser beiden Kantentypen wird in Kapitel 6 erklärt.

Kantentyp	Graphische Darstellung
`CONTROL_E`	
`CONTROL_E` (Default-Zweig)	
`SOFT_SYNC_E`	
`STRICT_SYNC_E`	
`LOOP_E`	
`FAILURE_E`	
`PRIORITY_E`	

Tabelle 3-3: *Kantentypen und ihre graphische Repräsentation*

Wir definieren auf der Kantenmenge E eines KF-Graphen die folgende Projektionsfunktionen:

ET : $E \mapsto EdgeTypes$ mit $ET((n_1, n_2, etype)) = etype$ („Kantentypfunktion“)

Id_{source} : $E \mapsto N$ mit $ET((n_1, n_2, etype)) = n_1$ („Quellaktivität“ der Kante)

Id_{dest} : $E \mapsto N$ mit $ET((n_1, n_2, etype)) = n_2$ („Zielaktivität“ der Kante)

- *D* beschreibt eine endliche Menge von Bezeichnern für Prozeßvariablen (sog. Datenelemente). Sie bilden das Schema des WF-Datenkontexts. Jedes Datenelement besitzt einen Wertebereich dom^d und einen Defaultwert $default^d \in dom^d$ (kann auch ein sog. *Nullwert* sein). Zur Laufzeit dienen Datenelemente der Versorgung und Übernahme der Parameterdaten von Aktivitäteninstanzen. Ihre Werte werden dem entsprechend persistent im WF-Datenkontext verwaltet. Wir kommen hierauf in Kapitel 5 zurück. Wichtig ist an dieser Stelle der Hinweis, daß auf Elemente der Menge *D* auch bereits bei der Kontrollflußdefinition Bezug genommen werden kann, um zur Laufzeit Auswahlentscheidungen für bedingte Verzweigungen (z.B. Ergebnis der Auswertung eines Verzweigungsprädikats) im WF-Datenkontext speichern zu können (siehe unten)

- *NT*: $N \mapsto NodeTypes$ ist eine Funktion, die jedem Knoten einen Typ aus *NodeTypes* zuordnet. Diese Menge ist wie folgt definiert:

  NodeTypes:= {`STARTFLOW`, `ENDFLOW`, `ACTIVITY`, `NULL`, `STARTLOOP`, `ENDLOOP`}

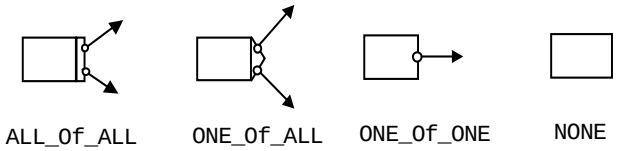
  Jeder KF-Graph besitzt je einen Knoten vom Typ `STARTFLOW` und `ENDFLOW`. Sie definieren die (logische) Start- bzw. Endaktivität des modellierten Workflow. Knoten vom Typ `ACTIVITY` („Aktivitätenknoten“) werden am häufigsten verwendet. Sie entsprechen elementaren oder komplexen Aktivitäten, definieren also die eigentlichen Arbeitsschritte des WF. Aktivitätenknoten ohne assoziierte Vorlage – sog. *Nullaktivitäten* – wird der Typ `NULL` zugeordnet. Zur Festlegung von Schleifenanfangs- und Schleifenendknoten dienen schließlich die Knotentypen `STARTLOOP` und `ENDLOOP` herangezogen.

- *NP:* $N \mapsto Priority$ ordnet jedem Knoten eine Ausführungspriorität zu. Wie die Menge *Priority* definiert ist und welche Semantik Knotenprioritäten besitzen, werden wir in Kapitel 6 sehen.

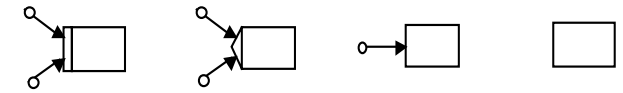
- V_{out}: $N \mapsto OutBehaviour$ ist eine Funktion, die jedem Knoten eine Ausgangssemantik (Verzweigungssemantik) zuordnet. Die Menge *OutBehaviour* ist wie folgt definiert:

  OutBehaviour:= { `ONE_Of_ONE`, `ONE_Of_ALL`, `ALL_Of_ALL`, `NONE`}

  Knoten mit Ausgangssemantik `ONE_Of_ONE` besitzen genau eine ausgehende Kante des Typs `CONTROL_E` (haben also einen direkten Nachfolger bzgl. dieses Kantentyps). Knoten mit Ausgangssemantik `ONE_Of_ALL` (`ALL_Of_ALL`) können mehrere ausgehende Kanten dieses Typs besitzen (mindestens aber eine). Sie definieren einen ODER-/UND-Verzweigungsknoten (OR-/AND-Split). Die Endaktivität besitzt als einzige Aktivität des KF-Graphen die Ausgangssemantik `NONE`, hat also keine ausgehenden Kontrollkanten.



- V_{in}: $N \mapsto InBehaviour$ ist eine Funktion, die jedem Knoten eine Eingangssemantik zuordnet. Die Menge *InBehaviour* ist analog zur Menge *OutBehaviour* definiert, allerdings sind ihre Elemente mit einer anderen Semantik versehen: Knoten mit Eingangssemantik `ONE_Of_ONE` besitzen genau eine einmündende Kante des Typs `CONTROL_E`, d.h. sie haben genau einen direkten Vorgänger bzgl. dieses Kantentyps). Knoten mit Eingangssemantik `ONE_Of_ALL` (`ALL_Of_ALL`) können mehrere einmündende Kanten dieses Typs besitzen (mindestens aber eine). Sie definieren einen ODER-/UND-Zusammenführungsknoten (OR-/AND-Join). Die Startaktivität besitzt als einzige Aktivität die Eingangssemantik `NONE`, hat also keine einmündende Kontrollkante.



- *DP*: $N \mapsto D \cup$ {UNDEFINED} („Decision Parameter“) ist eine Funktion, die jedem ODER-Verzweigungsknoten den eindeutigen Bezeichner eines Datenelements aus der Menge *D* zuordnet. Zur Laufzeit legt die Ausprägung dieses Datenelementes den auszuwählenden Zweig fest (Anmerkung: Prädikativen ODER-Verzweigungen ist ein Datenelement zugeordnet, das die Werte *True* und *False* annehmen kann). Für Nicht-ODER-Verzweigungsknoten wird *DP* der Wert UNDEFINED zugewiesen.

- *EC:* $E \mapsto$ *EdgeCode* $\cup$ {UNDEFINED} ordnet Kontrollkanten einen Code (aus dem Domain *EdgeCode)* zu. Er muß für ausgehende Kontrollkanten (mit Typ *ET* = CONTROL_E) eines OR-Split-Knotens, für Fehlerkanten (mit Typ *ET* = FAILURE_E) und für Priorisierungskanten (mit Typ *ET* = PRIORITY_E) definiert sein. In ersterem Fall legt er den logischen Auswahlwert („*selection code*“) für die Kante fest. Auf seiner Basis werden Verzweigungsentscheidungen getroffen. Im zweiten Fall definiert der Code den mit der Fehlerkante assoziierten Fehlerwert („*failure code*“). Für Priorisierungskanten schließlich legt er eine Ausführungspriorität fest, die dem Zielknoten der Kante zur Laufzeit zugewiesen werden soll (siehe hierzu Kapitel 6). Für alle anderen Kontrollkanten bleibt der Auswahlcode undefiniert (UNDEFINED). Zur besseren Unterscheidbarkeit von Auswahl- und Fehlerwerten definieren wir für eine Kante $e \in E$: $SC^e := EC^e$ für ET^e = CONTROL_E und $FC^e := EC^e$ für ET^e = FAILURE_E.

  Für die Menge $E_{out}^x \subset E$ der von einem OR-Split-Knoten *x* ausgehenden Kontrollkanten des Typs CONTROL_E muß gelten: $\forall\, e_1, e_2 \in E_{out}^x$ mit $e_1 \neq e_2 \Rightarrow SC^{e_1} \neq SC^{e_2}$. Diese Forderung ist notwendig, um zur Laufzeit den Bearbeitungszweig einer bedingten Verzweigung eindeutig bestimmen zu können – Verzweigungen mit impliziter Auswahl des Bearbeitungszweiges (vgl. Abschnitt 3.3.2.2) werden der Einfachheithalber ausgeklammert. Entsprechend müssen für Knoten mit mehreren ausgehenden Fehlerkanten, die zugeordneten Fehlerwerte unterscheidbar sein.

- *Template:* $N \mapsto$ *Templates* $\cup$ {NULL} ist eine Funktion, die – mit Ausnahme von Nullaktivitäten – jedem Knoten eine Prozeß- oder Aktivitätenvorlage (vgl. Abschnitt 3.2) zuordnet. Diese Vorlage legt u.a. die Ein- und Ausgabeparameter des Knotens sowie durchzuführende Aktionen fest. Je nachdem, ob einem Knoten eine Prozeß- oder eine Aktivitätenvorlage zugeordnet wird, definiert er einen Sub-Workflow oder eine elementare Aktivität. Prinzipiell kann auch einem Schleifenknoten oder dem Start- bzw. Endknoten des WF eine Vorlage zugewiesen werden.

- S: $N \mapsto$ *Services* $\times$ *{PRE, POST}* ist eine Funktion, die jedem Knoten eine Menge vor- und nachgeschalteter Hilfsdienste bzw. Aktionen zuordnet. Sie werden unmittelbar vor Start bzw. nach Beendigung der eigentlichen Aktivitäteninstanz ausgeführt (z.B. zur interaktiven Festlegung von Aktivitäteneingabedaten). Auf ihre genaue Bedeutung gehen wir in Abschnitt 5.1.2 ein.

Zur übersichtlicheren Schreibweise werden wir in der Folge Funktionswerte F(x) durch F^X abkürzen.

Beispiel 3-8 (Formale Darstellung von KF-Graphen)

Wir wenden die eingeführten Begriffe auf den KF-Graphen aus Abbildung 3-11 an. Ihm liegen die folgenden Knoten- und Kantenmengen zugrunde:

N = {Start, End, A, B, C, D, E, F, G, H, I, J, L_S, L_E}

E = { (Start, A, CONTROL_E), (A, L_S, CONTROL_E), {L_S, B, CONTROL_E), (B, C, CONTROL_E), (B, G, CONTROL_E), (C, D, CONTROL_E), (C, E, CONTROL_E), (D, F, CONTROL_E), (E, F, CONTROL_E), (F, I, CONTROL_E), (G, H, CONTROL_E), (H, I, CONTROL_E), (I, L_E, CONTROL_E), (L_E, J, CONTROL_E), (J, End, CONTROL_E), (E, H, SOFT_SYNC_E), (L_E, L_S, LOOP_E) }

D = {d} (in Abbildung 3-11 nicht dargestellt)

Knotentypen sowie Ein- und Ausgangssemantik der verschiedenen Knoten sind durch die Abbildungen NT, V_{out}, V_{in} und DP festgelegt (die Komponente *Template* wird hier ausgeklammert):

Knotentypen NT:

NT^{Start} = STARTFLOW, NT^{End} = ENDFLOW, NT^{L_S} = STARTLOOP, NT^{L_E} = ENDLOOP,
NT^n = ACTIVITY $\forall\, n \notin$ {Start, End, L_S, L_E}

Ausgangssemantik V_{out} und Eingangssemantik V_{in}:

$V_{out}{}^B$ = ALL_Of_ALL, $V_{out}{}^C$ = ONE_Of_ALL, $V_{out}{}^{End}$ = NONE,
$V_{out}{}^n$ = ONE_Of_ONE $\forall\, n \in$ N – {B, C, End}

$V_{in}{}^I$ = ALL_Of_ALL, $V_{in}{}^F$ = ONE_Of_ALL, $V_{in}{}^{Start}$ = NONE,
$V_{in}{}^n$ = ONE_Of_ONE $\forall\, n \in$ N – {I, F, Start}

Entscheidungsparameter DP (in Abbildung 3-11 nicht dargestellt):

DP^C = d $\in D$, DP^n = UNDEFINED für $n \neq$ C

$$EC^e = \begin{cases} \text{true} & \text{für } e=(C,D,CONTROL_E) \\ \text{false} & \text{für } e=(C,E,CONTROL_E) \\ \text{UNDEFINED,} & \text{sonst} \end{cases}$$

Im vorliegenden Beispiel besitzen lediglich die beiden Kontrollkanten (C, D, CONTROL_E) und (C, E, CONTROL_E) einen Auswahlcode *EC* bzw. *SC* (in Abbildung 3-11 nicht dargestellt).

Die graphische Darstellung ist übersichtlicher im Vergleich zur reinen Aufzählung von Mengen. Letztere ist aber für die präzise Festlegung von Struktur- und Dynamikeigenschaften unerläßlich.

3.4.2 Nachfolger- und Vorgängerfunktion

In diesem Abschnitt definieren wir Vorgänger- und Nachfolgerfunktionen auf KF-Graphen, die für die nachfolgenden Betrachtungen benötigt werden. Sie geben für jeden Aktivitätenknoten die Menge seiner direkten oder indirekten Nachfolger- bzw. Vorgängerknoten (derselben Hierarchiestufe im KF-Graphen) an. Entsprechende Funktionen werden sowohl für die Festlegung von Analyseverfahren benötigt, als auch für die präzise Formulierung von Aussagen zu statischen und dynamischen Eigenschaften eines KF-Graphen. Wie wir später sehen werden, ist man – je nach Fragestellung – an unterschiedlichen Nachfolger- bzw. Vorgängerknotenmengen einer Aktivität interessiert. Dem entsprechend werden nicht immer alle Kantentypen bei der Bestimmung dieser Mengen berücksichtigt. Aus diesem Grund erlauben die folgenden Definitionen eine kantentypbezogene Parametrisierung:

Definition 3-2 (Direkte Nachfolger/Vorgänger von Aktivitätenknoten):

Sei (N, E, ...) ein KF-Graph. Ferner bezeichne 2^M die Potenzmenge einer beliebigen Menge M.

(1) Für eine Kante $e \in E$ mit $e = (n_1, n_2, etype)$ heißt n_2 direkter Nachfolger von n_1 bzgl. dem Kantentyp *etype*. Umgekehrt heißt n_1 direkter Vorgänger von n_2 bzgl. *etype*.

(2) Die Nachfolgerfunktion *var_succ* (*succ* steht für *successor*) definiert für Knoten aus N jeweils die Menge ihrer direkten Nachfolger bzgl. einer gegebenen Kantentypmenge *Edges*:

$var_succ: N \times 2^{EdgeTypes} \mapsto 2^N$

mit

$var_succ(n, Edges) = \{ n' \in N \mid \exists\, e \in E: e = (n, n', etype) \wedge etype \in Edges \}$

Für $n \in N$, $Edges \subseteq EdgeTypes$ definiert $var_succ(n, Edges)$ die Knotenmenge, deren Elemente direkte Nachfolger von n bzgl. einem der durch *Edges* festgelegten Kantentypen sind.

(3) Auf Basis von *var_succ* definieren wir die folgenden beiden spezielleren Funktionen:

$succ: N \mapsto 2^N$ mit
$succ(n) := var_succ(n, \{$CONTROL_E, SOFT_SYNC_E, STRICT_SYNC_E$\})$

$c_succ: N \mapsto 2^N$ mit
$c_succ(n) := var_succ(n, \{\texttt{CONTROL_E}\})$

Für $n \in N$ ist $succ(n)$ die Menge aller direkten Nachfolger von n bzgl. normalen Kontroll- und Sync-Kanten; $c_succ(n)$ bestimmt lediglich die direkten Nachfolger von n bzgl. Kontrollkanten.

(4) Entsprechend lassen sich die Vorgängerfunktionen var_pred ($\rightarrow$ alle direkten Vorgänger bzgl. einer parametrisierbaren Menge an Kantentypen), $pred$ (alle direkten Vorgänger bzgl. Kontroll- und Sync-Kanten) und c_pred (alle direkten Vorgänger bzgl. Kontrollkanten) definieren ($pred$ steht für *predecessor*).

Beispiel 3-9 (Menge direkter Nachfolger)

Für den KF-Graphen aus Abbildung 3-11 gilt z.B. $succ(\texttt{E}) = \{\texttt{F}, \texttt{H}\}$ und $c_succ(\texttt{E}) = \{\texttt{F}\}$.

Bemerkung: Für $n \in N$ und $Edges_1 \subseteq Edges_2 \subseteq EdgeTypes$ gelten trivialerweise die folgenden Aussagen: $var_succ(n, Edges_1) \subseteq var_succ(n, Edges_2)$. Insbesondere gilt: $c_succ(n) \subseteq succ(n)$.

In den meisten Fällen interessiert man sich nicht nur für die direkten Nachfolger (Vorgänger) eines Aktivitätenknotens $n \in N$, sondern für alle Nachfolger (Vorgänger), die von n aus über eine Kanten- bzw. Pfeilfolge erreichbar sind. Je nach Fragestellung sind bei der Bestimmung dieser Mengen alle oder nur bestimmte Kantentypen zu berücksichtigen.

Definition 3-3 (Nachfolger/Vorgänger von Aktivitätenknoten):

Sei durch $(N, E, ...)$ ein KF-Graph gegeben.

(1) Die allgemeine Nachfolgerfunktion var_succ^* legt für Knoten aus N jeweils die Menge aller direkten oder indirekten Nachfolger bzgl. einer parametrisierbaren Kantentypmenge fest:

$var_succ^*: N \times 2^{EdgeTypes} \mapsto 2^N$

mit

$$var_succ^*(n, Edges) = \{n^* \in N \mid n^* \in var_succ(n, Edges) \vee (\exists\, n^{**} \in var_succ(n, Edges): n^* \in var_succ^*(n^{**}, Edges))\}$$

Für $n \in N$, $Edges \subseteq EdgeTypes$ definiert $var_succ^*(n, Edges)$ die Knotenmenge, deren Elemente direkte oder indirekte Nachfolger von n bzgl. der durch $Edges$ festgelegten Kantentypen sind.

(2) Unter Verwendung von var_succ^* können durch geeignete Auswahl der zu berücksichtigten Kantentypen wieder speziellere Funktionen definiert werden:

$succ^*: N \mapsto 2^N$ mit
$succ^*(n) = var_succ^*(n, \{\texttt{CONTROL_E}, \texttt{SOFT_SYNC_E}, \texttt{STRICT_SYNC_E}\})$

$c_succ^*: N \mapsto 2^N$ mit
$c_succ^*(n) = var_succ^*(n, \{\texttt{CONTROL_E}\})$

Für $n \in N$ ist $succ^*(n)$ die Menge aller über eine Folge von Kontroll- oder Sync-Kanten erreichbaren direkten oder indirekten Nachfolger von n. Die Menge $c_succ(n)$ gibt die über eine Folge von normalen Kontrollkanten erreichbaren Nachfolger von n an.

(4) Entsprechend lassen sich die Vorgängerfunktion var_pred^*, $pred^*$ und c_pred^* definieren.

Die Funktion var_succ^* entspricht dem *transitiven Abschluß* der Funktion var_succ. Die algorithmische Bestimmung der Mengen $var_succ^*(n, Edges)$, $n \in N$, $Edges \subseteq EdgeTypes$ kann über eine Breitensuche erfolgen. Entsprechende Algorithmen werden in der gängigen Literatur zur Graphentheorie [NeMo93] beschrieben, so daß wir an dieser Stelle auf eine Darstellung verzichten. Statt dessen geben wir ein einfaches Beispiel an.

Beispiel 3-10 (Nachfolgerknotenmenge)

Bezogen auf den KF-Graph aus Abbildung 3-11 ergibt sich z.B.: $succ^*$(E) = {F, H, I, L_E, J, End} und c_succ^*(E) = {F, I, L_E, J, End}. Bezieht man zusätzlich zu Kontroll- und Sync-Kanten auch Schleifenrücksprungkanten mit ein, so erhält man var_succ^*(E, *EdgeTypes*) = {L_S, B, C, D, E, F, G, H, I, L_E, J, End}.

3.4.3 KF-Graphklassen

Die Unterscheidbarkeit von Knoten- und Kantentypen macht es möglich, die Menge der theoretisch erzeugbaren KF-Graphen in verschiedene Graphklassen einzuteilen. Eine solche Einteilung ist in mehrfacher Hinsicht vorteilhaft. Zum einen vereinfacht sie die Diskussion struktureller Eigenschaften von KF-Graphen, zum anderen erlaubt sie eine stufenweise Vorgehensweise bei der Einführung von Analyseverfahren für KF-Graphen, deren Anwendung – ausgehend von einfacheren Kontrollflußstrukturen – schrittweise auf komplexer strukturierte Graphen ausgedehnt werden kann. In Bezug auf das ADEPT-Basismodell bietet sich z.B. die folgende Einteilung von KF-Graphen in Klassen an:

Definition 3-4 (KF-Graphklassen):

Sei *CFS* = (*N*, *E*, *D*, *NT*, ...) ein KF-Graph und seien durch die Mengen $SelEdges \subseteq EdgeTypes$ und $SelNodes \subseteq NodeTypes$ ausgewählte Kanten- und Knotentypen festgelegt. Besitzt CFS nur Knoten der Typen $NT \in SelNodes$ sowie Kanten der Typen $ET \in SelEdges$ (d.h. es gilt: $\forall\, n \in N$: $NT^n \in SelNodes$, $\forall\, e \in E$: $ET^e \in SelEdges$), so heißt CFS ein durch (*SelNodes*, *SelEdges*) erzeugter KF-Graph.

Schreibweise: $CFS \in CFS(SelNodes, SelEdges)$, wobei $CFS(SelNodes, SelEdges)$ die Klasse aller durch (*SelNodes*, *SelEdges*) erzeugbaren KF-Graphen beschreibt.

Bemerkungen:

- Obige Definition setzt nicht voraus, daß ein KF-Graph $CFS \in CFS(SelNodes, SelEdges)$ immer Knoten (Kanten) jeden Typs aus *SelNodes* (*SelEdges*) besitzen muß.
- Für $SelEdges_1 \subseteq SelEdges_2 \subseteq EdgeTypes$ und $SelNodes_1 \subseteq SelNodes_2 \subseteq NodeTypes$ gilt trivialerweise: $CFS(SelNodes_1, SelEdges_1) \subseteq CFS(SelNodes_2, SelEdges_2)$ (*)

Wir wollen kurz einige spezielle Graphklassen benennen, die wir für in den nachfolgenden Abschnitten aufgreifen werden.

Definition 3-5 (Spezielle KF-Graphklassen)

(1) Für *SelNodes*:= {STARTFLOW, ENDFLOW, ACTIVITY, NULL} und *SelEdges*:= {CONTROL_E} bildet CFS_{simple}:= $CFS(SelNodes, SelEdges)$ die Klasse der wohlstrukturierten, azyklischen KF-Graphen.

(2) Für *SelNodes*:= {STARTFLOW, ENDFLOW, ACTIVITY, NULL} und *SelEdges*:= {CONTROL_E, SOFT_SYNC_E, STRICT_SYNC_E } bildet CFS_{sync} := $CFS(SelNodes\ SelEdges)$ die Klasse der wohlstrukturierten, azyklischen KF-Graphen mit Sync-Kanten.

(3) Für *SelNodes*:= *NodeTypes* und *SelEdges*:= {CONTROL_E, LOOP_E, STRICT_SYNC_E, SOFT_SYNC_E} bildet CFS_{sync_loop}:= $CFS(SelNodes\ SelEdges)$ die Klasse der wohlstrukturierten, zyklischen KF-Graphen (mit Sync-Kanten).

(4) CFS_{all}:= $CFS(NodeTypes, EdgeTypes)$ schließlich bildet die Klasse der allgemeinen KF-Graphen. Hier bestehen keine Einschränkungen bzgl. erlaubter Knoten- und Kantentypen.

Bemerkung: Aufgrund von Bemerkung (*) gilt: $CFS_{simple} \subseteq CFS_{sync} \subseteq CFS_{sync_loop} \subseteq CFS_{all}$. Im nachfolgenden Abschnitt zeigen wir, daß Regeln für die Verwendung von Kanten- und Knotentypen, die für eine bestimmte KF-Graphklasse gültig sind, auch für allgemeinere KF-Graphklassen zutreffen.

Wohlstrukturierte, azyklische KF-Graphen beschreiben Prozeßstrukturen, die durch die kombinierte Verwendung von Sequenzen und Verzweigungen (unterschiedlicher Semantik) modelliert werden können. Für ihre Konstruktion sind normale Kontrollkanten ausreichend. Bei wohlstrukturierten, azy-

klischen KF-Graphen mit Sync-Kanten kommen zwei weitere Kantentypen hinzu, um Reihenfolgebeziehungen zwischen Aktivitäten paralleler Zweige definieren zu können. Sollen auch zyklische Aufgabenbearbeitungen (Schleifen) abbildbar sein, kommen noch die Knotentypen `STARTLOOP` und `ENDLOOP` sowie der Kantentyp `LOOP_E` hinzu. Zur Konstruktion allgemeiner KF-Graphen können dann Knoten und Kanten beliebigen Typs (inkl. Fehlerrücksprungkanten) verwendet werden.

3.5 Struktur von KF-Graphen

Wir wollen die in Abschnitt 3.3.2 diskutierten strukturellen Eigenschaften von KF-Graphen nun formal fassen. Im Mittelpunkt stehen dabei die Blockstrukturierung der KF-Graphen sowie einfache Regeln für die korrekte Verwendung der verschiedenen Knoten- und Kantentypen. Wichtig ist an dieser Stelle der Hinweis, daß Anwender weder bei der Modellierung noch bei der Änderung von WF-Modellen direkt auf entsprechenden Knoten- bzw. Kantenmengen operieren. Dem entsprechend benötigen sie auch keine genaue Kenntnis über die Strukturierung dieser Mengen. Vielmehr wird ihnen eine Menge von Änderungsoperationen an die Hand gegeben, mit denen sie WF-Modelle bzw. -Instanzen korrekt definieren bzw. ändern können. Eine wichtige Grundlage hierfür bilden die nun beschriebenen statischen und dynamischen Eigenschaften von KF-Graphen

Im folgenden sei $CFS = (N, E, D, NT, V_{out}, V_{in}, DP, EC, NP, Template)$ ein KF-Graph.

3.5.1 Wohlstrukturierte, azyklische KF-Graphen

Wir betrachten zuerst wohlstrukturierte, azyklische KF-Graphen, für deren Bildung die Blockkonstrukte Sequenz, bedingte Verzweigung, parallele Verzweigung und parallele Verzweigung mit finaler Auswahl verwendet werden können. Die für diese Klasse abgeleiteten Regeln zur Verwendung von Knoten- und Kantentypen lassen sich auf allgemeinere KF-Graphklassen übertragen.

Jeder KF-Graph besitzt einen ausgezeichneten Knoten, der keine einmündende und genau eine ausgehende Kontrollkante hat (Eingangssemantik `NONE` und Ausgangssemantik `ONE_Of_ONE`). Dieser Knoten wird als *Startknoten* (Startaktivität) des WF bezeichnet, und er besitzt den Knotentyp `STARTFLOW`. Entsprechend gibt es in jedem KF-Graphen genau einen Knoten, der keine ausgehende und genau eine einmündende Kontrollkante aufweist (Ausgangssemantik `NONE` und Eingangssemantik `ONE_Of_ONE`). Er heißt *Endknoten* (Endaktivität) des WF, und besitzt den Knotentyp `ENDFLOW`. O.B.d.A. wollen wir in der Folge stets annehmen, daß der Startknoten eines KF-Graphen den eindeutigen Bezeichner *Start* und der Endknoten den eindeutigen Bezeichner *End* besitzt.

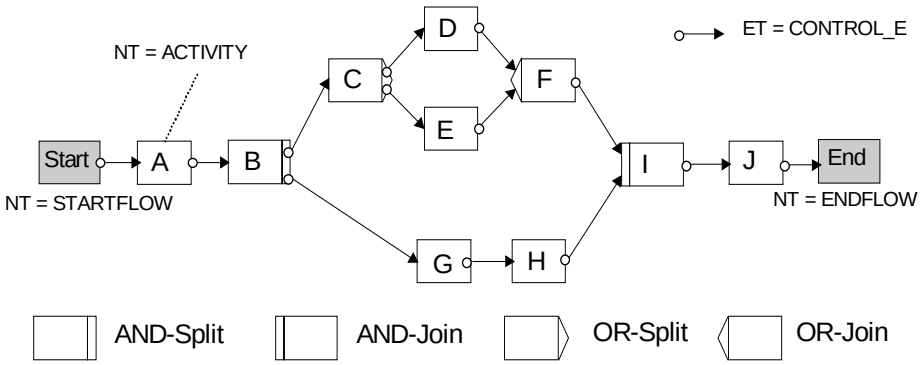


Abbildung 3-13: *Beispiel eines wohlstrukturierten, azyklischen KF-Graphen.*

Formal:

Strukturierungsregel KF-1 (Eingangs- und Ausgangssemantik von Start- bzw. Endknoten)

(1) *Startknoten eines KF-Graphen*

$\exists!\ Start \in N : NT^{Start} =$ `STARTFLOW` – Der Knoten *Start* besitzt die Eingangssemantik $V_{in}^{Start} =$ `NONE` und die Ausgangssemantik $V_{out}^{Start} =$ `ONE_Of_ONE`. Somit gilt:

$(\forall\ e \in E:\ Id_{dest}^{e} \neq Start)$ und $(\exists!\ e \in E$ mit $Id_{source}^{e} = Start$ und $ET^{e} =$ `CONTROL_E`$)$

(2) *Endknoten eines KF-Graphen*

$\exists!\ End \in N : NT^{End}$ = `ENDFLOW` – Der Knoten *End* besitzt die Ausgangssemantik V_{out}^{End} = `NONE` und die Eingangssemantik V_{in}^{End} = `ONE_Of_ONE`. Somit gilt:

$(\forall\ e \in E: Id_{source}^{e} \neq End)$ und $(\exists!\ e \in E$ mit $Id_{dest}^{e} = End$ und ET^{e} = `CONTROL_E`)

Hieraus folgt unmittelbar: $var_pred(Start, EdgeTypes) = \emptyset$ und $var_succ(End, EdgeTypes) = \emptyset$

Mit Ausnahme des Start- und Endknotens besitzt jede Aktivität eines KF-Graphen mindestens eine einmündende und eine ausgehende Kontrollkante vom Typ ET = `CONTROL_E`. Dem entsprechend hat jeder dieser Aktivitätenknoten mindestens einen direkten Vorgänger und einen direkten Nachfolger bzgl. Kanten des Typs `CONTROL_E`. Formal läßt sich diese Forderung durch die folgende Regel ausdrücken (vgl. Definition 3-1):

Strukturierungsregel KF-2 (Ein- und Ausgangssemantik von Aktivitätenknoten)

Für $n \in N - \{$`Start`, `End`$\}$ gilt stets:
$V_{out}^{n}\ (V_{in}^{n}) \in \{$`ONE_Of_ONE`, `ONE_Of_ALL`, `ALL_Of_ALL`$\}$

Die vom ADEPT-Basismodell geforderte Blockstrukturierung wird durch die Strukturierungsregel KF-3 ausgedrückt. Dazu definieren wir zunächst die Mengen der Verzweigungs- bzw. Zusammenführungsknoten eines KF-Graphen *CFS* wie folgt:

$$SplitNodes^{CFS} := \{n \in N \mid V_{out}^{n} \in \{\texttt{ONE_Of_ALL}, \texttt{ALL_Of_ALL}\}\} \subset N$$

$$JoinNodes^{CFS} := \{n \in N \mid V_{in}^{n} \in \{\texttt{ONE_Of_ALL}, \texttt{ALL_Of_ALL}\}\} \subset N$$

Strukturierungsregel KF-3 („Symmetrische“ Blockstrukturierung von KF-Graphen)

Es existiert eine <u>bijektive</u> Abbildung $join^{CFS} : SplitNodes^{CFS} \mapsto JoinNodes^{CFS}$ mit folgenden Eigenschaften:

(1) $s \in SplitNodes^{CFS} \Rightarrow \forall\ z_i \in c_succ(s): \left(join^{CFS}(s) \in c_succ^{*}(z_i)\right) \vee \left(join^{CFS}(s) = z_i\right)$[6]

(2) $s \in SplitNodes^{CFS} \wedge |c_succ(s)| > 1 \not\Rightarrow \exists\ j \in JoinNodes^{CFS}$:
$\left(j \in c_pred^{*}(join^{CFS}(s))\right) \wedge \left(\forall\ z_i \in c_succ(s): j \in c_succ^{*}(z_i) \vee j = z_i\right)$

(3) Für $s_1, s_2 \in SplitNodes^{CFS}$ mit $s_1 \neq s_2$ gilt:
$s_2 \in \left(c_succ^{*}(s_1) \cap c_pred^{*}(join^{CFS}(s_1))\right) =: Z \Leftrightarrow join^{CFS}(s_2) \in Z$

(4) $s \in SplitNodes^{CFS}$ mit $j = join^{CFS}(s)$: V_{out}^{s} = `ONE_Of_ALL` $\Rightarrow V_{in}^{j}$ = `ONE_Of_ALL`

Bemerkung: Entsprechende Eigenschaften sollen auch für die inverse Funktion $split^{CFS} := (join^{CFS})^{-1}$ gelten ($join^{CFS}$ ist invertierbar, da bijektiv). Regel (1) lautet dann zum Beispiel:

$$j \in JoinNodes^{CFS} \Rightarrow \forall\ z_i \in c_pred(j): (split^{CFS}(j) \in c_pred^{*}(z_i)) \vee (split^{CFS}(j) = z_i)$$

Die Strukturierungsregel KF-3 besagt, daß jede an einem Split-Knoten $s \in SplitNodes^{CFS}$ vorgenommene Aufsplittung des Kontrollflusses wieder „symmetrisch“ an einem eindeutigen Join-Knoten $join^{CFS}(s) \in JoinNodes^{CFS}$ zusammengeführt wird, so wie am Beispiel aus Abbildung 3-13 dargestellt. Die symmetrische Blockstrukturierung wird durch die folgenden Regeln sichergestellt:

Der Knoten $join^{CFS}(s)$ ist ein gemeinsamer Nachfolger aller von s ausgehenden Zweige (bzw. von deren Startknoten) oder – im Fall von leeren Zweigen – auch ein direkter Nachfolger von s selbst (1). Wegen (2) gibt es (für Verzweigungen mit mehr als einem ausgehenden Zweig) keinen anderen Join-Knoten, der Vorgänger von $join^{CFS}(s)$ im KF-Graphen ist und der dieselbe Eigenschaft aufweist.

[6] Gilt $join^{CFS}(s) \in c_succ(s)$, liegt ein leerer Zweig vor. Leere Zweige sind bei bedingten Verzweigungen erlaubt.

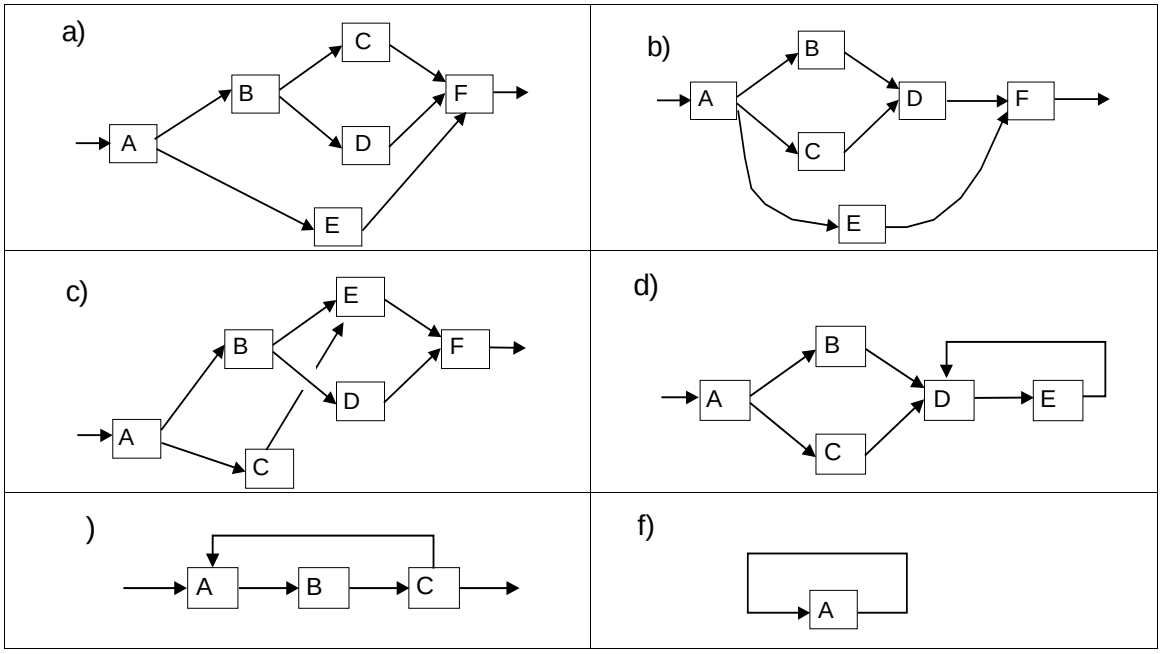


Tabelle 3-4: *Nicht zulässige Kontrollflußstrukturen für wohlstrukturierte, azyklische KF-Graphen (zur Vereinfachung der Darstellung wurden die Ein- und Ausgangssemantiken der Knoten weggelassen)*

Darüber hinaus kann, aufgrund der Injektivität[7] von $join^{CFS}$, die Zusammenführung zweier unterschiedlicher Split-Knoten [im Sinne der Regeln (1) und (2)] nicht durch ein und denselben Join-Knoten erfolgen. Damit ist für wohlstrukturierte, azyklische KF-Graphen eine Struktur, wie sie in Tabelle 3-4 a) dargestellt ist, ausgeschlossen. Ebenso muß es, aufgrund der Surjektivität[8] von $join^{CFS}$, zu jedem Join-Knoten der Menge $JoinNodes^{CFS}$ einen eindeutigen Split-Knoten aus $SplitNodes^{CFS}$ geben, der die Eigenschaften (1) und (2) erfüllt. Eine Graphstruktur, wie in Tabelle 3-4 b) dargestellt, ist deshalb nicht möglich. Wie das Beispiel aus Tabelle 3-4 c) zeigt, reichen diese Festlegungen allein noch nicht aus, um die geforderte „symmetrische" Strukturierung der KF-Graphen garantieren zu können[9]. Sie wird erst durch die Anwendung von Regel (3) erreicht. Durch die Regel (4) schließlich kann sichergestellt werden, daß ein OR-Split nicht durch einen AND-Join wieder zusammengeführt wird. Die drei anderen Kombinationen für Split-Join-Semantiken (OR-Split/OR-Join, AND-Split/AND-Join, AND-Split/OR-Join) sind dagegen zulässig.

Durch die Anwendung der Strukturierungsregel KF-3 sind zyklische Kontrollflußstrukturen nicht ausgeschlossen, wie das Beispiel d) aus Tabelle 3-4 zeigt. Hier sind die von dieser Regel geforderten Eigenschaften erfüllt, und dennoch tritt ein Zyklus im KF-Graph auf. In Bezug auf Kontrollkanten des Typs `CONTROL_E` fordern wir deshalb, daß es im KF-Graphen keine Folge von Kanten dieses Typs geben darf, innerhalb der ein bestimmter Knoten mehrfach auftritt. Der Hintergrund dieser Forderung ist, daß, mit Ausnahme von Schleifen- und Fehlerkanten, in KF-Graphen keine zyklischen Kontrollstrukturen erzeugt werden dürfen. Die Unterscheidbarkeit zwischen normalen Kontrollkanten und Rücksprungkanten werden wir uns später mehrfach zunutze machen. Formal:

Strukturierungsregel KF-4 (Zyklenfreiheit bzgl. normalen Kontrollkanten)

CFS darf keine Zyklen enthalten, d.h. $\forall\, n \in N$ gilt: $n \notin c_succ^*(n)$ ($\Leftrightarrow n \notin c_pred^*(n)$)

Zur Erinnerung: $c_succ^*(n)$ ($c_pred^*(n)$) ist die Menge aller direkten und indirekten Nachfolger (Vorgänger) von n bzgl. Kanten des Typs `CONTROL_E`)

[7] Die Funktion $join^{CFS}$ ist nach Voraussetzung bijektiv und deshalb auch *injektiv*. Für $s_1, s_2 \in SplitNodes^{CFS}$ gilt somit: $join^{CFS}(s_1) = join^{CFS}(s_2) \Rightarrow s_1 = s_2$

[8] Die Funktion $join^{CFS}$ ist nach Voraussetzung bijektiv und deshalb auch *surjektiv*, d.h. $\forall\, j \in JoinNodes^{CFS}$ gilt: $\exists\, s \in SplitNodes^{CFS}$ mit $j = join^{CFS}(s)$

[9] Wählt man $SplitNodes^{CFS}$:= {A, B} und $JoinNodes^{CFS}$:= {E, F} mit $join^{CFS}$(A):={E} und $join^{CFS}$(B):= F, so sind die Regeln (1) und (2) erfüllt – der dargestellte Graph ist aber nicht symmetrisch strukturiert.

Durch diese Regel sind auch Schlingen, wie in Tabelle 3-4 f) dargestellt, ausgeschlossen. Andernfalls könnte ein KF-Graph isolierte Knoten besitzen, die nur mit sich selbst, aber ansonsten mit keinem anderen Knoten verbunden sind. Solche isolierten Knoten würden nie zur Ausführung kommen.

Wichtig ist an dieser Stelle der Hinweis, daß die meisten der in Tabelle 3-4 dargestellten Fälle im ADEPT-Basismodell im Prinzip abbildbar sind. Allerdings muß dazu eine etwas andere Modellierung gewählt werden, bei der ggf. zusätzliche Knoten und Kanten (unterschiedlichen Typs) erforderlich werden. Die Fälle a) und b) aus Tabelle 3-4 etwa können im ADEPT-Basismodell, wie in Abbildung 3-14 dargestellt, modelliert werden.

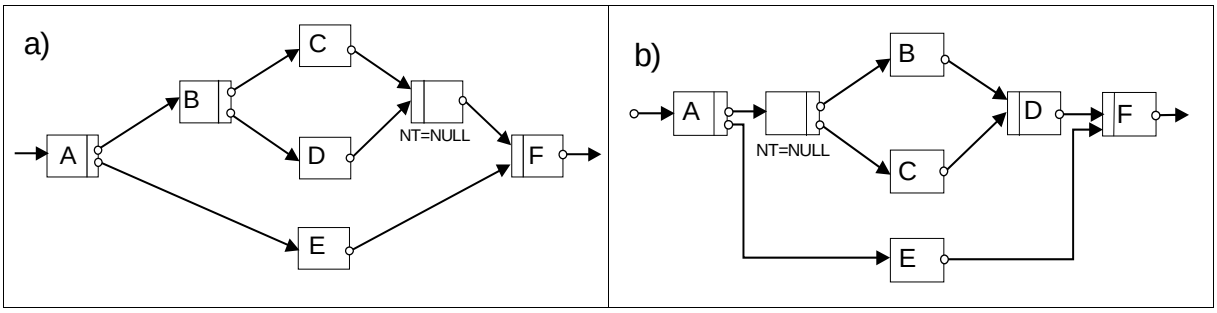


Abbildung 3-14: *Modellierung der Graphen a) und b) aus Tabelle 3-4 im ADEPT-Basismodell*

Wie man leicht sieht, wurde in beiden Fällen eine sog. *Nullaktivität* hinzugefügt, um die geforderte Blockstrukturierung zu erzielen. Auch der in Tabelle 3-4 c) dargestellte Fall läßt sich im ADEPT-Basismodell abbilden, wenn für die Modellierung zusätzlich Sync-Kanten zur Verfügung stehen. Hierauf wird in Abschnitt 3.5.2 eingegangen. Die notwendigen Erweiterungen zur Unterstützung von Schleifen werden in Abschnitt 3.5.3 diskutiert. Ihre Verwendung gestattet die Modellierung des zyklischen KF-Graphen aus Tabelle 3-4 e). Allerdings wird für die Modellierung des Rücksprungs ein anderer Kantentyp als `CONTROL_E` verwendet.

Definition 3-6 (Korrektheit von wohlstrukturierten, azyklischen KF-Graphen)

Ein wohlstrukturierter, azyklischer KF-Graph, der die Strukturierungsregeln KF-1 bis KF-4 befolgt, heißt *korrekt*.

Im folgenden geben wir statische Eigenschaften wohlstrukturierter, azyklischer KF-Graphen an, die sich aus den Strukturierungsregeln KF-1 bis KF-4 ableiten lassen. Einige von ihnen werden wir später dazu verwenden, um Aussagen zum dynamischen Verhalten von Workflows zu treffen.

Aus den Regeln KF-1 bis KF-4 kann unmittelbar gefolgert werden, daß KF-Graphen bzgl. Kanten des Typs `CONTROL_E` transitiv reduziert sind (vgl. Satz 3-1). Für zwei Knoten $n, n^* \in N$, für die es eine Kantenfolge (mit Länge ≥ 2) von n nach n^* gibt und für die $n^* \neq join(n)$ gilt, kann es also keine Kontrollkante geben, die direkt von n nach n^* führt. Durch die Blockstrukturierung vermeiden wir somit „überflüssige“ Kontrollkanten, was sich positiv auf die Übersichtlichkeit der Modelle auswirkt.

Satz 3-1 (Transitive Reduktion von KF-Graphen)

Sei $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter wohlstrukturierter, azyklischer KF-Graph. Seien ferner $n, n^* \in N$ zwei Aktivitätenknoten für die gelte:

(V1) Es gibt eine Kantenfolge (n, n_1, `CONTROL_E`), (n_1, n_2, `CONTROL_E`),.., (n_{k-1}, n^*, `CONTROL_E`) von n nach n^* mit $n_1 \neq n^*$ (d.h. die Kantenfolge besitzt mindestens die Länge 2)

(V2) Falls $n \in SplitNodes^{CFS}$, so gelte $n^* \neq join^{CFS}(n)$.

Dann folgt: $n^* \notin c_succ(n)$, d.h. es gibt keine Kontrollkante aus E die direkt von n nach n^* führt. Wir nennen CFS transitiv reduziert bzgl. Kanten des Typs ET = `CONTROL_E`.

Der formale Beweis zu Satz 3-1 findet sich, ebenso wie die Beweise zu den nachfolgenden Sätzen, in Anhang B. Eine ebenfalls wichtige Eigenschaft von KF-Graphen beschreibt Satz 3-2. Er sagt aus, daß

jeder Aktivitätenknoten eines KF-Graphen vom Startknoten aus über eine Kantenfolge erreichbar ist. Umgekehrt kann der Endknoten von beliebigen anderen Aktivitätenknoten des KF-Graphen aus erreicht werden (d.h. er ist ein Nachfolger aller Aktivitätenknoten des KF-Graphen).

Satz 3-2 (Verbundenheit von Aktivitätenknoten mit Start- und Endknoten)

Sei $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter wohlstrukturierter, azyklischer KF-Graph.

a) Für $n \in N$, $n \neq$ `Start` gilt stets: $n \in c_succ^*$(`Start`), d.h. der Knoten n ist vom Startknoten `Start` aus über eine Kantenfolge erreichbar.

b) Für $n \in N$, $n \neq$ `End` gilt stets: `End` $\in c_succ^*(n)$; d.h. der Endknoten `End` ist von jedem anderen Knoten n über ein Kantenfolge erreichbar. Insbesondere gilt: `End` $\in c_succ^*$(`Start`).

Für korrekte wohlstrukturierte, azyklische KF-Graphen gilt weiter, daß die Zusammenführung zweier vom selben Split-Knoten s ausgehenden Zweige nicht vor dem zugehörigen Join-Knoten $join^{CFS}(s)$ erfolgen kann. Dies folgt unmittelbar aus Satz 3-3. Er besagt, daß es zwischen zwei beliebigen Knoten verschiedener Ausführungszweige keine Kante des Typs `CONTROL_E` geben kann. Wie in Abschnitt 3.3.2.4 informell gezeigt, können solche Abhängigkeiten im ADEPT-Basismodell ausschließlich mittels Sync-Kanten modelliert werden.

Satz 3-3 (Unabhängigkeit von Teilzweigen in wohlstrukturierten, azyklischen KF-Graphen)

Sei $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter KF-Graph. Sei ferner $s \in SplitNodes^{CFS}$ ein Split-Knoten aus N mit zugehörigem Join-Knoten $j := join^{CFS}(s)$. Dann gilt für zwei beliebige Knoten $n_1, n_2 \in N$, die unterschiedlichen Zweigen der durch s und j gebildeten Verzweigung angehören, stets: (n_1, n_2, `CONTROL_E`) $\notin E$. Das bedeutet, daß die Knoten n_1 und n_2 nicht über eine normale Kontrollkante miteinander verbunden sein können.

Unmittelbar aus Satz 3-3 kann die folgende Aussage gewonnen werden:

Satz 3-4 (Nachfolger von Aktivitätenknoten verschiedener Teilzweige)

Seien $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter KF-Graph und $s \in SplitNodes^{CFS}$ ein Split-Knoten mit zugehörigem Join-Knoten $j := join^{CFS}(s)$. Für $z_1, z_2 \in c_succ(s)$, $z_1 \neq z_2$ und $z_1, z_2 \neq j$ gilt stets:

$$n \in c_succ^*(z_1) \cap c_succ^*(z_2) \Rightarrow n \in c_succ^*(j) \cup \{j\}$$

Dieser Satz besagt, daß jeder gemeinsame Nachfolger n (bzgl. Kanten des Typs ET = `CONTROL_E`) zweier auf einen Split-Knoten s folgenden Knoten z_1 und z_2 auch ein Nachfolger des zu s gehörigen Join-Knotens $join^{CFS}(s)$ (inkl.) ist. Eine ähnliche Aussage macht Satz 3-5. Er besagt, daß es von einem Nachfolgerknoten eines Split-Knotens s, der auch Vorgänger des zugehörigen Join-Knotens j ist, keinen Weg zu beliebigen Nachfolgerknoten von j geben kann, in dem j nicht enthalten ist.

Satz 3-5 (Zusammenführung von Kontrollflußaufsplittungen)

Seien $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter KF-Graph, $s \in SplitNodes^{CFS}$ ein Split-Knoten und $j := join^{CFS}(s)$ der zugehörige Join-Knoten. Sei ferner $x \in c_succ^*(s) \cap c_pred^*(j)$ ein beliebiger Knoten eines Teilzweigs der durch (s, j) gebildeten Verzweigung.

Traversiert eine vom Knoten x ausgehende Kantenfolge (x, n_1, `CONTROL_E`), (n_1, n_2, `CONTROL_E`),.., (n_{k-1}, n_k, `CONTROL_E`) einen Knoten n_x, $x \in \{2, ..., k\}$ mit $n_x \in c_succ^*(j)$, gilt: $j \in \{n_1, n_2, ..., n_{x-1}\}$

Die Sätze 3-1 bis 3-5 unterstreichen, daß KF-Graphen, die den Strukturierungsregeln KF-1 bis KF-4 genügen, die in Abschnitt 3.3.2 informell diskutierten strukturellen Eigenschaften aufweisen. Nennenswert ist nochmals die Eindeutigkeit von Verzweigungsstrukturen, die ein Reihe von Vorteilen mit sich bringt. Insbesondere lassen sich Status- und Strukturänderungsoperationen, wie das Zurücksetzen

einzelner Zweige (Statusänderung) oder das Hinzufügen, Entfernen oder Verschieben von Teilzweigen (strukturelle Änderung), erheblich einfacher verwirklichen, als bei konventionellen Beschreibungsformalismen (z.B. Petri-Netzen). Die Knotenmenge der Teilzweige einer Verzweigung sind eindeutig und überlappen sich nicht mit den Knotenmengen anderer Teilzweige.

Für korrekte KF-Graphen $CFS = (N, E, ...) \in CFS_{simple}$ definieren wir:

Definition 3-7 (Teilzweige einer Verzweigung)

Seien s ein Split-Knoten und $j = join^{CFS}(s)$ der zugehörige Join-Knoten eines KF-Graphen *CFS*. Sei ferner $z \in c_succ(s)$ ein direkter Nachfolger von s (bzgl. CONTROL_E) mit $z \neq j$. Dann bildet der durch die Knotenmenge $Z := (c_succ^*(z) \cup \{z\}) \cap c_pred^*(j)$ induzierte Teilgraph von CFS einen *Teilzweig* der durch (s, j) festgelegten Verzweigung. Der Knoten z heißt Startknoten dieses Zweiges.

Mit Satz 3-3 ergibt sich für $z_1, z_2 \in c_succ(s), z_1 \neq z_2$: $Z_1 \cap Z_2 = \emptyset$, d.h. die Knotenmengen verschiedener Teilzweige sind disjunkt. Die Anzahl der Teilzweige der durch (s, j) gebildeten Verzweigung wird dementsprechend durch die Anzahl der vom Split-Knoten s ausgehenden Kanten mit Typ CONTROL_E determiniert. Sie entspricht auch der Anzahl der in j einmündenden Kanten dieses Typs.

Beispiel 3-11 (Knotenmengen von Teilzweigen)

Im KF-Graphen *CFS* aus Abbildung 3-13 wird durch die Knoten (B, I) eine parallele Verzweigung mit zwei Teilzweigen definiert. Für deren Knotenmengen gilt: $Z_1 = (c_succ^*(C) \cup \{C\}) \cap c_pred^*(I) = \{C, D, E, F\}$ und $Z_2 = (c_succ^*(G) \cup \{G\}) \cap c_pred^*(I) = \{G, H\}$. Die Teilzweige selbst entsprechen den durch Z_1 bzw. Z_2 induzierten Teilgraphen von *CFS*. Offensichtlich gilt $Z_1 \cap Z_2 = \emptyset$. Des weiteren beschreibt der durch Z_1 induzierte Teilgraph selbst Verzweigung, deren Teilzweige aus jeweils einem Aktivitätenknoten (D bzw. E) bestehen.

3.5.2 Wohlstrukturierte, azyklische KF-Graphen mit Sync-Kanten

Wir betrachten in diesem Abschnitt wohlstrukturierte, azyklische KF-Graphen, zu deren Bildung zusätzlich die in Abschnitt 3.3.2.4 eingeführten Sync-Kanten verwendet werden dürfen. Dabei gelten die zuvor vorgestellten KF-Strukturierungsregeln und die aus ihnen ableitbaren Eigenschaften weiter uneingeschränkt. Ein Beispiel für einen KF-Graphen dieser Klasse zeigt Abbildung 3-15. Hier ist über eine einfache Sync-Kante festgelegt, daß Aktivität H frühestens dann aktiviert werden darf, wenn entweder Aktivität E beendet worden ist oder aber feststeht, daß E nicht mehr zur Ausführung kommt.

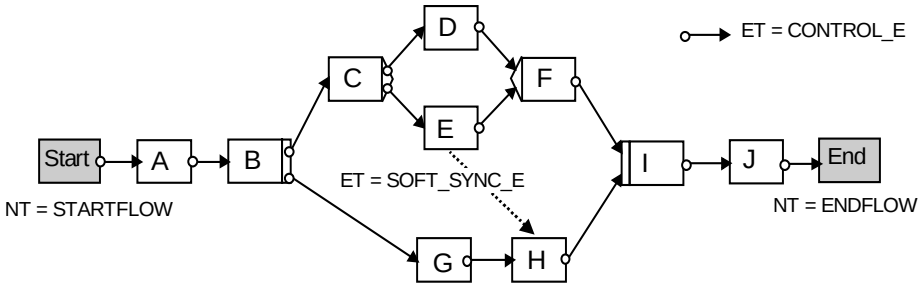


Abbildung 3-15: *Beispiel eines wohlstrukturierten, azyklischen KF-Graphen mit Sync-Kante.*

Wie in Abschnitt 3.5.1 gezeigt, können zwischen den Aktivitäten der verschiedenen Teilzweige einer Parallelverzweigung keine Reihenfolgebeziehungen über Kanten des Typs CONTROL_E bestehen (vgl. Satz 3-3). Genau zu diesem Zweck werden im ADEPT-Basismodell Sync-Kanten angeboten. Ihre Verwendung ist an bestimmte Regeln geknüpft, die nun kurz vorgestellt werden. Abbildung 3-16 zeigt einige Beispiele für die nicht korrekte Anwendung dieses Kantentyps.

Wenig sinnvoll und deshalb auch nicht zulässig ist die Verwendung von Sync-Kanten entlang des „normalen“ Kontrollflusses (Kante J→L im Beispiel). Entsprechende Reihenfolgebeziehungen können im ADEPT-Basismodell bereits durch Kontrollkanten des Typs CONTROL_E festgelegt werden, so daß die Verwendung von Sync-Kanten hier nur Redundanzen und damit unübersichtliche Modelle schaffen würde. Ebensowenig erlaubt sind „rückwärts“ gerichtete Sync-Kanten (Kante C→A im Bei-

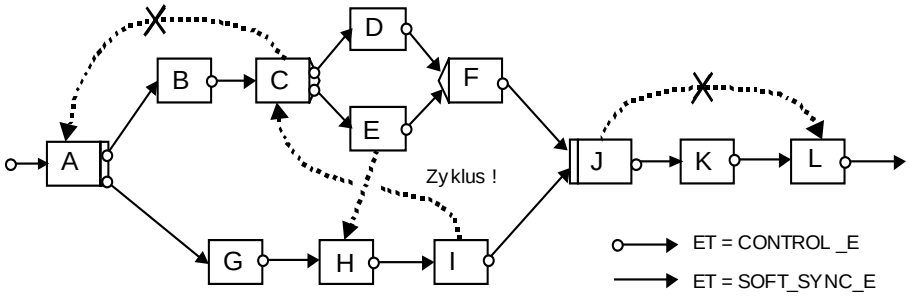


Abbildung 3-16: *Durch Strukturierungsregeln ausgeschlossene Verwendung von Sync-Kanten*

spiel), die zu Zyklen im KF-Graphen führen. Für die Modellierung solcher Rücksprünge sind in ADEPT Schleifen- und Fehlerrücksprungkanten vorgesehen. Schließlich dürfen Sync-Kanten keine Aktivitätenknoten verbinden, die in unterschiedlichen Teilzweigen einer bedingten Verzweigung oder einer parallelen Verzweigung mit finaler Auswahl liegen. Im Beispiel aus Abbildung 3-16 wäre deshalb z.B. eine Sync-Kante von `D` nach `E` nicht erlaubt (ihre Verwendung würde keinen Sinn ergeben).

Diese unerwünschten Fälle werden durch die nun vorgestellte Strukturierungsregel KF-5 ausgeschlossen. Die grundlegende Forderung dabei ist, daß zwei Aktivitätenknoten nur dann über eine Sync-Kante miteinander verknüpft werden dürfen, wenn sie in unterschiedlichen Teilzweigen einer parallelen Verzweigung liegen[10]. Um diese Regel formulieren zu können, definieren wir zunächst die Funktion *BranchNodes*, die wir auch an anderer Stelle nochmals verwenden werden. Sie bestimmt für zwei Aktivitätenknoten den Split- und den Join-Knoten der „innersten", sie umschließenden Verzweigung (vorausgesetzt, die beiden Knoten befinden sich in unterschiedlichen Teilzweigen des KF-Graphen).

Definition 3-8 (BranchNodes)

Sei $CFS = (N, E, ...) \in CFS_{sync}$ ein wohlstrukturierter, azyklischer KF-Graph mit Sync-Kanten. Dann definieren wir

$$BranchNodes^{CFS}: N \times N \mapsto (SplitNodes^{CFS} \cup \{\texttt{UNDEFINED}\}) \times (JoinNodes^{CFS} \cup \{\texttt{UNDEFINED}\})$$

mit

$$BranchNodes^{CFS}(n_1, n_2) := \begin{cases} (\texttt{UNDEFINED}, \texttt{UNDEFINED}) & \text{falls } n_1 \in c_pred^*(n_2) \vee n_2 \in c_pred^*(n_1) \vee n_1 = n_2 \\ (s^*, join^{CFS}(s^*)) & \text{sonst} \end{cases}$$

Der Split-Knoten $s^* = s^*(n_1, n_2) \in SplitNodes^{CFS}$ ist dabei wie folgt definiert: Es gilt:
$s^* \in c_pred^*(n_1) \cap c_pred^*(n_2) =: P \wedge \forall s \in SplitNodes^{CFS}$ mit $s \in P \Rightarrow s \in c_pred^*(s^*)$

Beispiel 3-12 (BranchNodes)

Wir beziehen uns auf den KF-Graphen aus Abbildung 3-15. Hier gilt z.B. *BranchNodes*(*CFS*, `E`, `H`) = (`B`, `I`), d.h. der durch die Knoten `B` und `I` gebildete Verzweigungskontrollblock entspricht dem minimalen, die beiden Knoten `E` und `H` umschließenden Verzweigungsblock.

Ein Algorithmus zur Implementierung dieser Funktion findet sich in Anhang C. Das Verfahren überprüft zuerst, ob die beiden Knoten n_1 und n_2 im KF-Graphen aufeinander folgen. Trifft dies nicht zu, wird anschließend – unter Ausnutzung der Blockstrukturierung – der am nächsten liegende, gemeinsame Vorgänger von n_1 und n_2 bestimmt.

Unter Verwendung von *BranchNodes* können wir die Strukturierungsregel KF-5 formal definieren:

[10] Da Verzweigungen geschachtelt sein können, ist nicht ausgeschlossen, daß der Quellknoten der Sync-Kante einem Teilzweig einer bedingten Verzweigung angehört, ihr Zielknoten aber außerhalb dieser bedingten Verzweigung liegt (vgl. Abbildung 3-15).

Strukturierungsregel KF-5 (Verwendung von Sync-Kanten)

Sei $CFS = (N, E, ...) \in CFS_{sync}$ gegeben. Die Menge der Sync-Kanten $E_{sync} \subset E$ des KF-Graphen CFS ist festgelegt durch

E_{sync}:= $\{e \in E \mid e = (n_1, n_2, stype)$ mit $n_1, n_2 \in N$, $stype \in$ {SOFT_SYNC_E, STRICT_SYNC_E}}

Für die Quell- und Zielknoten einer Sync-Kante $e = (n_{source}, n_{dest}, stype) \in E_{sync}$ fordern wir, daß sie in unterschiedlichen Teilzweigen einer parallelen Verzweigung liegen müssen, d.h. es soll gelten:

$BranchNodes^{CFS}(n_{source}, n_{dest}) = (s, j) \neq$ (UNDEFINED, UNDEFINED) mit

V_{out}^{s} = ALL_Of_ALL und V_{in}^{j} = ALL_Of_ALL

Die Regel KF-5 definiert eine notwendige, aber noch keine hinreichende Bedingung für die korrekte Verwendung von Sync-Kanten. Dies zeigt das Beispiel aus Abbildung 3-16. Hier ist für die beiden Kanten I→C und E→H die Strukturierungsregel KF-5 erfüllt, d.h. es gilt: $BranchNodes^{CFS}$(I, C) = $BranchNodes^{CFS}$(E, H) = (A, J) mit $V_{out}^{A} = V_{in}^{J}$ = ALL_Of_ALL. Allerdings resultiert aus der gemeinsamen Verwendung dieser beiden Kanten, zusammen mit den anderen Kontrollkanten, ein „unerwünschter“ Zyklus im KF-Graph (Kantenfolge C→E→H→I→C). Dadurch würde es zur Laufzeit zu einer Verklemmung kommen, infolge derer das Fortschreiten des Kontrollflusses nicht mehr möglich wäre. (Im Beispiel könnten die Aktivitäten C und H niemals aktiviert werden.) Aus diesem Grund darf die gemeinsame Verwendung von normalen Kontrollkanten (ET = CONTROL_E) und Sync-Kanten (ET ∈ {SOFT_SYNC_E, STRICT_SYNC_E}) niemals zu Zyklen im KF-Graphen führen. Wir erweitern deshalb die im vorangegangenen Abschnitt vorgestellte Strukturierungsregel KF-4 (→ Zyklenfreiheit bzgl. Kanten des Typs CONTROL_E) wie folgt ($succ^*$ bzw. $pred^*$ bezeichnen die Nachfolger- bzw. Vorgängerfunktion bzgl. Kontroll- und Sync-Kanten).

Strukturierungsregel KF-4* (Zyklenfreiheit bzgl. Kontroll- und Sync-Kanten)

$CFS \in CFS_{sync}$ darf keine Zyklen enthalten, d.h. $\forall\, n \in N$ gilt: $n \notin succ^*(n)$ ($\Leftrightarrow n \notin pred^*(n)$)

Wir verzichten auf die Darstellung eines Algorithmus zur Bestimmung von Zyklen in gerichteten Graphen. Entsprechende Verfahren werden in der gängigen Literatur zur Graphentheorie [NeMo93, Neum75]) behandelt, und können ohne Schwierigkeiten auch auf KF-Graphen angewendet werden.

Durch die Strukturierungsregeln KF-5 und KF-4* ist die in Abbildung 3-16 dargestellte Verwendung von Sync-Kanten nicht zulässig. Für strikte Sync-Kanten fordern wir zusätzlich zu diesen beiden Strukturierungsregeln, daß der Quellknoten der Sync-Kante nur dann einem Teilzweig einer parallelen Verzweigung mit finaler Auswahl angehören darf, wenn ihr Zielknoten in demselben Teilzweig enthalten ist. Der Hintergrund dieser Regel ist die Vermeidung kaskadierender Effekte beim Zurücksetzen einzelner Teilzweige, wie sie ansonsten bei Verwendung dieses Kantentyps in Verbindung mit diesem Konstrukt auftreten könnten. Praktisch gesehen resultieren hieraus keine wesentlichen Einschränkungen. Vielmehr vereinfacht sich dadurch die formale Festlegung des Ausführungsverhaltens von KF-Graphen und die Verfahren zur Analyse ihres dynamischen Verhaltens (siehe Kapitel 4):

Strukturierungsregel KF-6 (Verwendung von strikten Sync-Kanten)

Sei $CFS = (N, E, ...) \in CFS_{sync}$ gegeben. Die Menge der strikten Sync-Kanten $E_{strict} \subset E$ des KF-Graphen CFS ist definiert durch

E_{strict}:= $\{e \in E \mid e = (n_1, n_2,$ STRICT_SYNC_E$), n_1, n_2 \in N\}$

Sei ferner durch (s, j) eine parallele Verzweigung mit finaler Auswahl festgelegt (d.h. es gelte: $s \in SplitNodes^{CFS}$, $j := join^{CFS}(s) \in JoinNodes^{CFS}$ mit V_{out}^{s} = ALL_Of_ALL und V_{in}^{j} = ONE_Of_ALL). Die einzelnen Teilzweige seien durch die Knotenmengen $Z_v := (c_succ^*(z_v) \cup \{z_v\}) \cap c_pred^*(j)$, $z_v \in c_succ(s)$ ($v \in \{1,...,k\}$) festgelegt.

Dann muß für den Quellknoten n_1 und den Zielknoten n_2 einer strikten Sync-Kante $e \in E_{strict}$ gelten: $[n_1 \in Z_v$ für ein $v \in \{1,...,k\} \Rightarrow n_2 \in Z_v\ ]$

In Kapitel 4 werden wir zeigen, daß für einfache Sync-Kanten die Strukturierungsregeln KF-4* und KF-5 ausreichend sind, um wichtige Dynamikeigenschaften, wie das Fortschreiten des Kontrollflusses oder die Aktivierbarkeit von Knoten, gewährleisten zu können. Das gilt jedoch nicht mehr bei Verwendung strikter Sync-Kanten, selbst wenn hier zusätzlich die Regel KF-6 zur Anwendung kommt. Probleme können dann auftreten, wenn der Quellknoten der Sync-Kante dem Teilzweig einer bedingten Verzweigung angehört, ihr Zielknoten dagegen nicht. Während dies bei kombinierter Anwendung mehrerer strikter Sync-Kanten durchaus sinnvoll sein kann, um bestimmte Sachverhalte modellieren zu können, können sich in anderen Fällen unerwünschte Zustände während der WF-Ausführung ergeben, in denen Aktivitäten nicht mehr aktivierbar sind. Aus diesem Grund wäre zum Beispiel in Abbildung 3-15 die Verwendung einer strikten Sync-Kante von `E` nach `H` nicht erlaubt. (Würde `E` nicht ausgeführt werden, so käme es zu einem „Stillstand“ im WF.) Wir werden in Kapitel 4 sehen, wie sich solche Fälle zur Modellierungszeit ausschließen bzw. erkennen lassen.

3.5.3 Wohlstrukturierte, zyklische KF-Graphen

Bisher haben wir KF-Graphen mit *azyklischer* Struktur betrachtet. In diesem Abschnitt werden KF-Graphen behandelt, die, zusätzlich zu den bereits vorgestellten Knoten- und Kantentypen, auch die Abbildung von Schleifen gestatten (vgl. Abschnitt 3.3.2.3). Die in den beiden vorangegangenen Abschnitten eingeführten Strukturierungsregeln gelten weiterhin, müssen aber um Regeln für die Verwendung von Schleifenknoten und -kanten ergänzt werden. Dabei ist sicherzustellen, daß die regelmäßige Blockstrukturierung der KF-Graphen erhalten bleibt und, daß Schleifenrücksprungkanten nur von Schleifenend- zu Schleifenanfangsknoten führen. Ein Beispiel zeigt Abbildung 3-11.

Eine Schleife besitzt einen End- und einen Anfangsknoten, die über eine Schleifenrücksprungkante miteinander verknüpft sind. Dabei bildet der Teilgraph zwischen dem Start- und Endknoten einen Kontrollblock, der als *Schleifenkörper* bezeichnet wird. Im folgenden sei $CFS = (N, E, \ldots)$ ein wohlstrukturierter, zyklischer KF-Graph. Wir definieren die Mengen der Schleifenanfangs- und Schleifenendknoten von *CFS* wie folgt:

$$LoopStartNodes^{CFS} := \{n \in N \mid NT^n = \texttt{STARTLOOP}\} \subset N$$
$$LoopEndNodes^{CFS} := \{n \in N \mid NT^n = \texttt{ENDLOOP}\} \subset N$$

Für die Verwendung von Schleifenknoten und -kanten gilt die folgende Strukturierungsregel:

Strukturierungsregel KF-7 (Verwendung von Schleifenknoten und -kanten)

Es existiert eine <u>bijektive</u> Abbildung $endloop^{CFS}$: $LoopStartNodes^{CFS} \mapsto LoopEndNodes^{CFS}$, die jedem Anfangsknoten $L_S \in LoopStartNodes^{CFS}$ einer Schleife einen Schleifenendknoten $L_E := endloop^{CFS}(L_S)$ zuordnet. Der von der Knotenmenge $L_{body} := c_succ^*(L_S) \cap c_pred^*(endloop^{CFS}(L_S))$ induzierte Teilgraph von *CFS* heißt dem entsprechend *Schleifenkörper*. Für L_S, L_E und L_{body} sowie für die Verwendung von Schleifenrücksprungkanten gelten im ADEPT-Basismodell die folgenden Regeln:

(1) $L_E \in c_succ^*(L_S) \wedge \exists!\, e \in E$ mit $e = (L_E, L_S, \texttt{LOOP_E})$

(2) $V_{out}^{L_S} = V_{in}^{L_E} = \texttt{ONE_Of_ONE}$

(3) Für $s \in SplitNodes^{CFS}$ gilt:
$s \in L_{body} \Leftrightarrow join^{CFS}(s) \in L_{body}$

(4) Für $L_{S'} \in LoopStartNodes^{CFS}$, $L_{S'} \neq L_S$ gilt:
$L_{S'} \in L_{body} \Leftrightarrow endloop^{CFS}(L_{S'}) \in L_{body}$

(5) Für $e = (n_1, n_2, \texttt{LOOP_E}) \in E$ gilt stets:
$n_1 \in LoopEndNodes^{CFS} \wedge n_2 \in LoopStartNodes^{CFS} \wedge n_1 = endloop^{CFS}(n_2)$

(6) Für eine beliebige Sync-Kante $e \in E$ gilt stets:
$Id_{source}^{e} \in L_{body} \Leftrightarrow Id_{dest}^{e} \in L_{body}$ (und O.B.d.A. $Id_{source}^{e} \notin \{L_E, L_S\}$)

Diese Regel definiert die Bedingungen für die Verwendung von Schleifen in KF-Graphen. Jede Schleife besteht aus einem eindeutigen Anfangs- und Endknoten. Letzterer ist vom Anfangsknoten aus

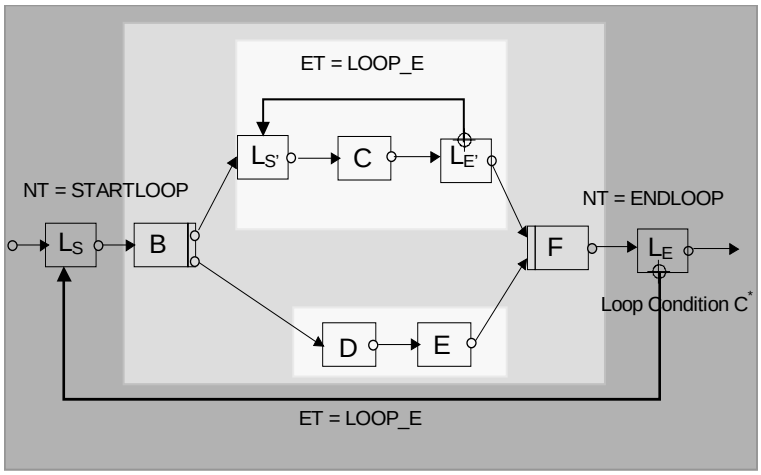


Abbildung 3-17: *Geschachtelte Schleifenstrukturen*

über normale Kontrollkanten erreichbar. Umgekehrt führt vom Endknoten aus eine Schleifenrücksprungkante zum Anfangsknoten (1). Des weiteren besitzt der Schleifenanfangsknoten genau eine ausgehende und der Schleifenendknoten genau eine einmündende Kontrollkante (2). Zwischen diesen beiden Knoten kann ein beliebiger Kontrollblock liegen, der den Körper der Schleife bildet. Grundlegend dabei ist wieder das Prinzip der regelmäßigen Blockstrukturierung, das durch die Eigenschaften (3) und (4) gewährleistet wird. Die Eigenschaft (3) schließt die Überlappung von Schleifen- und Verzweigungsstrukturen und (4) die Überlappung von Schleifenstrukturen aus. Erlaubt sind dagegen regelmäßig geschachtelte Strukturen (inkl. ineinander geschachtelter Schleifen, wie in Abbildung 3-17 dargestellt). Ferner wird durch (5) festgelegt, daß Schleifenkanten nur zur Verknüpfung von Schleifenendknoten und -anfangsknoten verwendet werden dürfen. Für KF-Graphen mit Sync-Kanten und Schleifen schließlich fordern wir, daß eine Sync-Kante nicht in einen Schleifenkörper hinein bzw. aus einem Schleifenkörper heraus führen darf (6). Die letztgenannte Regel stellt für die Praxis keine Einschränkung dar, vereinfacht aber die Realisierung fortschrittlicher Systemfunktionen, wie das partielle Zurücksetzen der WF-Kontrolle innerhalb einzelner Teilzweige einer parallelen Verzweigung (siehe hierzu Kapitel 6).

Bei Beachtung der vorgestellten Strukturierungsregeln erhält man korrekte KF-Graphen. Wie wir in Kapitel 7 sehen werden, bietet ADEPT für die Modellierung und Änderung von KF-Graphen eine Menge von Strukturänderungsoperationen an, die – ausgehend von einem korrekten KF-Graphen – wieder einen korrekten KF-Graphen erzeugen. Beispielsweise können Schleifenrücksprungkanten nur zusammen mit den entsprechenden Schleifenanfangs- und Schleifenendknoten in den KF-Graphen eingefügt bzw. aus ihm gelöscht werden. Dagegen ist z.B. das Hinzufügen oder Entfernen einzelner Sync-Kanten zulässig, sofern die für diesen Kantentyp festgelegten Strukturierungsregeln (z.B. Zyklenfreiheit) eingehalten werden. Die zugrundeliegende Philosophie ist es, die genannten Strukturierungsregeln per Konstruktion von Anfang an sicherzustellen und nicht – wie bei vielen Ansätzen üblich – im Nachhinein durch aufwendige Analyseverfahren zu überprüfen.

3.6 Kontrollblöcke in KF-Graphen

Wie bereits dargestellt, nehmen im ADEPT-Basismodell sog. *Kontrollblöcke* eine zentrale Rolle im Zusammenhang mit der Strukturierung der WF-Modelle ein. Kontrollblöcke sind dabei nicht zu verwechseln mit den bereits erwähnten Prozeß- und Blockaktivitäten, bei denen die Ablaufstruktur eines Sub-WF vor dem übergeordneten Super-WF verborgen werden kann. Ein Kontrollblock beschreibt einen Teilgraphen eines KF-Graphen, der – vereinfacht ausgedrückt – nur vollständige Verzweigungs- und Schleifenstrukturen enthält. Im KF-Graphen aus Abbildung 3-17 trifft dies zum Beispiel auf die grau umrahmten Teilgraphen zu, nicht jedoch z.B. auf den Teilgraphen $B \rightarrow D \rightarrow E$. Er enthält den AND-Split-Knoten B, nicht aber dessen zugehörigen Join-Knoten F. In diesem Abschnitt wird eine formale Definition für Kontrollblöcke angegeben.

3.6.1 Formale Definition eines Kontrollblocks

Formal sind Kontrollblöcke wie folgt definiert:

Definition 3-9 (Kontrollblock)

Sei $CFS = (N, E, ...) \in CFS_{all}$ ein KF-Graph und sei $N_{sub} \subseteq N - \{\texttt{Start}, \texttt{End}\}$ eine Knotenmenge mit folgenden Eigenschaften:

1. $N_{sub} \neq \emptyset$ enthält zwei Knoten n_{start} und n_{end} für die gilt:
   $N_{sub} = (c_succ^*(n_{start}) \cap c_pred^*(n_{end})) \cup \{n_{start}, n_{end}\}$
   (Somit gilt: $\forall\, n \in N_{sub} - \{n_{start}\}: n_{start} \in c_pred^*(n) \wedge \forall\, n \in N_{sub} - \{n_{end}\}: n_{end} \in c_succ^*(n)$).
2. Für $s \in SplitNodes^{CFS}$ gilt: $s \in N_{sub} - \{n_{end}\} \Leftrightarrow join^{CFS}(s) \in N_{sub} - \{n_{start}\}$
3. Für $L_S \in LoopStartNodes^{CFS}$ gilt: $L_S \in N_{sub} \Leftrightarrow endloop^{CFS}(L_S) \in N_{sub}$

Der durch die Knotenmenge N_{sub} induzierte Teilgraph $CFS_{sub} = (N_{sub}, E_{sub}, ...)$ von CFS heißt *Kontrollblock* von CFS mit Startknoten n_{start} und Endknoten n_{end}. Gilt $n_{start} \in SplitNodes^{CFS}$ und $n_{end} = join^{CFS}(n_{start})$, so liegt ein *Verzweigungskontrollblock* vor. Von einem *Schleifenkontrollblock* spricht man, falls gilt: $n_{start} \in LoopStartNodes^{CFS}$ und $n_{end} = endloop^{CFS}(n_{start}) \in LoopEndNodes^{CFS}$.

Gemäß dieser Definition ist ein einzelne Aktivität n (mit $NT^n = \texttt{ACTIVITY}$ oder $NT^n = \texttt{NULL}$) ebenfalls ein Kontrollblock (mit $n_{start} = n_{end} = n$). Dasselbe gilt für Sequenzen, Verzweigungen, Teilzweige einer Verzweigung, Schleifen (inkl. Anfangs- und Endknoten) oder Schleifenkörper. In jedem Fall ist ein Kontrollblock eindeutig durch seinen Start- und Endknoten determiniert. Zu beachten ist, daß Definition 3-9 (2. Eigenschaft) nicht ausschließt, daß der Startknoten (Endknoten) eines Kontrollblocks ein Join- (Split-)Knoten ist. Für diesen Randfall ist der zugehörige Split- (Join-) Knoten nicht in N_{sub} enthalten. In Bezug auf die Nachfolger- und Vorgängerknoten eines Kontrollblocks läßt sich die folgende Aussage treffen:

Satz 3-6 (Vorgänger / Nachfolgerknoten eines Kontrollblocks)

Zu dem KF-Graphen $CFS = (N, E, ...)$ sei durch $CFS_{sub} = (N_{sub}, E_{sub}, ...)$ ein Kontrollblock mit Startknoten n_{start} und Endknoten n_{end} gegeben. Für $n_1 \in N_{sub}$, $n_2 \in N - N_{sub}$ gilt:

$$n_1 \in c_succ^*(n_2) \Leftrightarrow n_{start} \in c_succ^*(n_2) \ \underline{\text{und}}\ n_1 \in c_pred^*(n_2) \Leftrightarrow n_{end} \in c_pred^*(n_2)$$

Dieser Satz besagt, daß ein (beliebiger) Knoten n_1 eines Kontrollblocks genau dann ein Nachfolger (Vorgänger) eines außerhalb dieses Blockes gelegenen Knotens n_2 ist, wenn der Startknoten (Endknoten) des Kontrollblocks ebenfalls ein Nachfolger (Vorgänger) von n_2 ist. Diese Aussage folgt unmittelbar aus der Definition 3-9 und den zuvor vorgestellten KF-Strukturierungsregeln.

Für die Manipulation von KF-Graphen ist es wichtig, Aussagen über den Kontext, in dem Kontrollblöcke eingebettet sind, treffen zu können. Der Kontext eines Kontrollblockes wird gebildet durch eine Menge von Kanten und Knoten: *Kontextkanten* entsprechen Kontrollkanten, die Knoten des Kontrollblockes mit außerhalb des Blockes liegenden Knoten verknüpfen. Entsprechend bezeichnen wir Knoten, die direkte Vorgänger oder Nachfolger von Knoten des Kontrollblockes sind, diesem aber selbst nicht angehören, als *Kontextknoten*.

Definition 3-10 (Kontext / Abgeschlossenheit eines Kontrollblocks)

Sei $CFS_{sub} = (N_{sub}, E_{sub}, ...)$ ein Kontrollblock des KF-Graphen $CFS = (N, E, ...)$ mit Startknoten n_{start} und Endknoten n_{end}. Die Menge der *Kontextkanten* und *-knoten* von CFS_{sub} ist dann definiert durch:

$$E_{context} := \{e \in E \mid (Id_{source}{}^e \in N - N_{sub} \wedge Id_{dest}{}^e \in N_{sub}) \vee (Id_{source}{}^e \in N_{sub} \wedge Id_{dest}{}^e \in N - N_{sub})\}$$

$$N_{context} := \{n \in N - N_{sub} \mid \exists\, e \in E_{context} : Id_{source}{}^e = n \vee Id_{dest}{}^e = n\}$$

Kontextknoten von CFS_{sub} sind diejenigen Knoten aus N, die außerhalb des Kontrollblockes CFS_{sub} liegen und die Quell- oder Zielknoten einer Kante aus $E_{context}$ sind. Das Paar $(N_{context}, E_{context})$ bezeichnen wir als *Kontext* von CFS_{sub}.

Des weiteren wird ein Kontrollblock, der nur Kontextkanten des Typs `CONTROL_E` besitzt, als *abgeschlossen* bezeichnet. Andernfalls sprechen wir von einem *offenen Kontrollblock*.

Wie man leicht sieht, entsprechen die Kontextknoten eines abgeschlossenen Kontrollblockes genau denjenigen Knoten des KF-Graphen, die direkte Vorgänger (bzw. Nachfolger) des Startknotens bzw. des Endknotens des Kontrollblockes sind (bzgl. Kanten des Typs `CONTROL_E`).

Beispiel 3-13 (Kontext von Kontrollblöcken)

Im KF-Graphen aus Abbildung 3-17 bildet die durch (`B`, `F`) definierte Parallelverzweigung einen Verzweigungskontrollblock mit Startknoten `B` und Endknoten `F`. Sein Kontext ist definiert durch $N_{context}$ = {L_S, L_E} und $E_{context}$ = {(L_S, `B`, `CONTROL_E`), (`F`, L_E, `CONTROL_E`)}. Gemäß obiger Definition ist dieser Kontrollblock abgeschlossen. Ein Beispiel für einen offenen Kontrollblock zeigt Abbildung 3-15. Hier bildet der durch die Knotenmenge {`C`, `D`, `E`, `F`} induzierte Teilgraph einen Verzweigungskontrollblock, dessen Kontext gegeben ist durch $N_{context}$ = {`B`, `I`, `H`} und $E_{context}$ = {(`B`, `C`, `CONTROL_E`), (`F`, `I`, `CONTROL_E`), (`E`, `H`, `SOFT_SYNC_E`)}. Da hier von einem Knoten des Kontrollblocks (Aktivität `E`) eine Sync-Kante zu einem außerhalb des Blocks liegenden Knoten (Aktivität `H`) führt, ist der Block nicht abgeschlossen.

Im allgemeinen erfordern Änderungsoperationen, wie das Verschieben oder Löschen eines Kontrollblocks, bei Abgeschlossenheit weniger Überprüfungen als im Falle offener Blöcke. Bei letzteren können durch die Anwendung einer Änderungsoperation zum Beispiel Sync-Kanten in einen anderen Kontext gerückt werden, in dem sie die geforderten Strukturierungsregeln nicht mehr erfüllen. Wir kommen hierauf im Zusammenhang mit (dynamischen) WF-Änderungen zurück. Wie wir im nachfolgenden Kapitel ebenfalls zeigen werden, ist die Abgeschlossenheit von Kontrollblöcken nützlich, um bestimmte Analysen des dynamischen Verhaltens von KF-Graphen effizient zu gestalten.

3.6.2 Bestimmung minimaler Kontrollblöcke

Kontrollblöcke bilden logische Strukturen innerhalb von KF-Graphen, die bestimmte Eigenschaften aufweisen und deren Zusammensetzung sich je nach Kontext bzw. Fragestellung anders darstellen kann. Zentrale Bedeutung kommt in diesem Zusammenhang der Bestimmung des kleinsten, eine vorgegebene Knotenmenge umschließenden Kontrollblocks (sog. *minimaler Block*) zu. Sie wird z.B. in Verbindung mit der Durchführung struktureller Änderungen, etwa beim Einfügen einer neuen Aktivität zwischen zwei Knotenmengen (siehe Kapitel 7) erforderlich, um minimale Änderungsbereiche so festlegen zu können, daß die regelmäßige Blockstrukturierung nach der Durchführung der Änderung erhalten bleibt. Formal definieren wir einen minimalen Block bzgl. einer Knotenmenge N^* wie folgt:

Definition 3-11 (Minimaler Block, MinBlock-Funktion)

Seien $CFS = (N, E, ...) \in CFS_{all}$ ein KF-Graph und $\emptyset \neq N^* \subseteq N - \{\texttt{Start}, \texttt{End}\}$ eine beliebige nichtleere Knotenmenge. Ein Kontrollblock $CFS_{min} = (N_{min}, E_{min}, ...)$ von CFS heißt *minimaler Block* bzgl. N^*, falls gilt:

$N^* \subseteq N_{min}$ und es gibt keinen anderen Kontrollblock $CFS_{sub} = (N_{sub}, E_{sub}, ...)$ mit $N^* \subseteq N_{sub} \subset N_{min}$.

Die Abbildung $MinBlock^{CFS}$: $2^{N - \{\texttt{Start}, \texttt{End}\}} \mapsto N \times N$, die jeder nicht-leeren Teilmenge von $N - \{\texttt{Start}, \texttt{End}\}$ den minimalen, sie umschließenden Kontrollblock – er liegt durch die Angabe seines Start- und Endknotens eindeutig fest – zuordnet, heißt *MinBlock*-Funktion.

Ein Trivialfall ist gegeben, wenn die Knotenmenge N^* genau eine Aktivität umfaßt. Hier besteht der minimale Block aus genau diesem Knoten. Ein Beispiel, bei dem die Knotenmenge für die der minimale Block ermittelt werden soll, aus zwei Knoten n_1 und n_2 besteht, zeigt Abbildung 3-18.

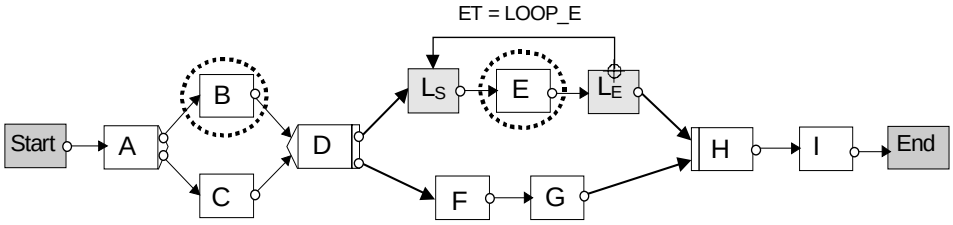


Abbildung 3-18: *Bestimmung des minimalen, zwei Knoten umschließenden Kontrollblockes*

Bei der Bestimmung von $\text{MinBlock}^{CFS}(\{n_1, n_2\})$ wird zwischen zwei Fällen unterschieden: (1) Die beiden Knoten n_1 und n_2 gehören unterschiedlichen Teilzweigen einer Verzweigung an, oder aber sie folgen – wie im Beispiel aus Abbildung 3-18 – im Kontrollfluß aufeinander. Im Fall (1) entspricht der gesuchte Block der „innersten", die Knoten n_1 und n_2 umschließenden Verzweigung. Dieser Verzweigungskontrollblock kann mit der bereits vorgestellten Funktion $BranchNodes^{CFS}$ (siehe Definition 3-8) bestimmt werden. Im Fall (2) muß ein umschließender Teilgraph ermittelt werden, der – vereinfacht ausgedrückt – keine „offenen" Schleifen- und Split bzw. Join-Knoten enthält. Dies erfolgt im wesentlichen durch Auszählen der zwischen n_1 und n_2 liegenden „offenen" Knoten (z.B. Split-Knoten ohne zugehörigen Join-Knoten) und durch geeignete Expandierung des zwischen n_1 und n_2 liegenden Teilgraphen. – Ein Verfahren zur Bestimmung minimaler Kontrollblöcke findet sich in Anhang C.

Beispiel 3-14 (Bestimmung des minimalen, zwei Knoten umschließenden Blockes)

Gegeben sei der KF-Graph aus Abbildung 3-18. Die Anwendung von *MinBlock* auf {B, E} liefert dann:

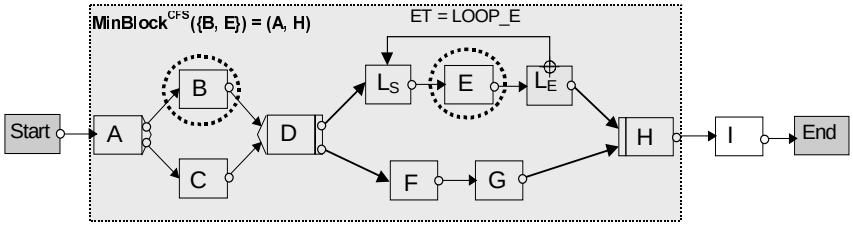


Abbildung 3-19: *Minimaler Block (bzgl. $N^* = \{B, F\}$)*

Beispiel 3-15 (Bestimmung des minimalen, mehrere Knoten umschließenden Blockes)

Wie bereits erwähnt, kann die MinBlock-Funktion auch auf eine beliebige Knotenmenge angewendet werden. Ein Beispiel hierzu zeigt die folgende Abbildung.

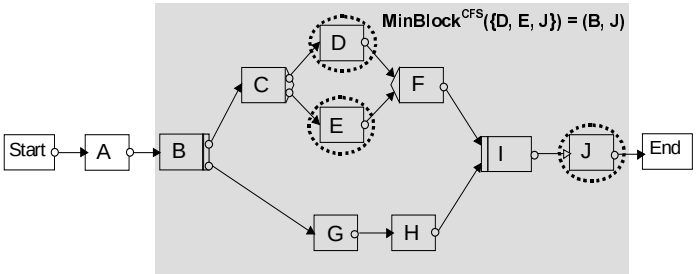


Abbildung 3-20: *Minimaler Block (bzgl.. $N^* = \{D, E, J\}$).*

3.7 Hierarchische Strukturierung von KF-Graphen

Wie bereits erwähnt, unterstützt ADEPT, neben dem beschriebenen *logischen* Blockkonzept, die *hierarchische* Strukturierung eines WF-Modells durch die Definition von Sub-WFs. Sie erfolgt durch die Zuordnung geeigneter Prozeßvorlagen zu einzelnen Knoten des KF-Graphen. Insgesamt kann sich ein Top-Level-WF aus mehreren elementaren oder komplexen Aktivitäten zusammensetzen. Dabei wird eine komplexe Aktivität solange verfeinert, bis nur noch elementare Aktivitäten vorliegen. Das Ergebnis dieser Dekomposition ist ein *Aktivitätenbaum*, dessen Wurzel die Vorlage des Top-Level-WF abbildet. Die anderen Nichtblattknoten repräsentieren die Vorlagen komplexer Aktivitäten einer

bestimmten Hierarchiestufe, die jeweils die Ausführung von Aktivitäten der nächst tieferen Ebene koordinieren. Die Blätter repräsentieren die Vorlagen elementarer Aktivitäten.

Bei der Definition eines WF-Modells kann eine existierende Prozeßvorlage $\omega \in WT$ prinzipiell zur Festlegung mehrerer komplexer Aktivitäten derselben oder unterschiedlicher Hierarchiestufen verwendet werden. Die gewählte Schachtelungstiefe – sie entspricht der längsten Pfeilfolge von der Wurzel des Aktivitätenbaums bis zu einem Blattknoten – unterliegt keinen Begrenzungen. Nicht zulässig sind *rekursive Aufrufstrukturen*, die für die Modellierung realer Arbeitsabläufe auch nicht relevant sind [Meye96]. Ihre Unterstützung würde einen unnötigen Zusatzaufwand mit sich bringen, sowohl in Bezug auf die Analyse des dynamischen WF-Verhaltens, als auch im Hinblick auf die Implementierung der WF-Steuerungskomponente. Rekursive Aufrufstrukturen werden deshalb im ADEPT-Basismodell ausgeschlossen. Hierzu führen wir die folgende *Abhängigkeitsrelation* ein:

Die Prozeßvorlage $\omega \in WT$ verwendet die Prozeß- oder Aktivitätenvorlage $\upsilon \in WT \cup AT$:⇔ Der ω zugeordnete KF-Graph $CFS_\omega = (N, E, ..., Template)$ besitzt einen Knoten $n \in N$ mit $Template(n) = \upsilon$. Das heißt mindestens ein Knoten des KF-Graphen CFS_ω wird durch die Vorlage υ beschrieben. Formal kann diese Relation durch die folgende Funktion ausgedrückt werden:

Definition 3-12 (children und children*):

Templates:= $WT \cup AT$ bezeichne die Menge aller im Vorlagen-Repository definierten Prozeß- und Aktivitätenvorlagen.

Für die Prozeßvorlage $\omega \in WT$ sei der KF-Graph gegeben durch $CFS_\omega = (N, E, ..., Template)$. Dann sind die Funktionen *children* und *children** wie folgt definiert ($2^{Templates}$: Potenzmenge von *Templates*):

(1) $children: WT \mapsto 2^{Templates}$ mit
$$children(\omega) := \left\{ \omega^* \in Templates \;\middle|\; \exists n \in N: \quad Template(n) = \omega^* \right\}$$

(2) $children^*: WT \mapsto 2^{Templates}$ mit
$$children^*(\omega) := \left\{ \omega^* \in Templates \;\middle|\; \omega^* \in children(\omega) \vee \left(\exists\, \omega^{**} \in children(\omega) : \omega^* \in children^*(\omega^{**})\right) \right\}$$

Für eine Prozeßvorlage $\omega \in WT$ beschreibt $children(\omega)$ diejenigen Prozeß- und Aktivitätenvorlagen aus dem Repository, die zur Definition der Knoten des KF-Graphen CFS_ω verwendet werden. Die transitive Hülle $children^*(\omega) \supseteq children(\omega)$ umfaßt dem entsprechend alle Vorlagen, die direkt oder indirekt zur Definition von CFS_ω oder untergeordneter KF-Graphen verwendet werden.

Die durch $\upsilon \in children(\omega)$ beschriebene Abhängigkeitsrelation kann durch einen Digraphen beschrieben werden, dessen Knotenmenge V der Menge $children^*(\omega)$ entspricht und für dessen Pfeilmenge A gilt: $(x, y) \in A \Leftrightarrow y \in children(x)$. Formal:

Definition 3-13 (Vorlagenabhängigkeitsgraph, VAG):

Sei $\omega \in WT$ eine Prozeßvorlage. Ein Digraph $G_\omega = [V, A]$ mit Knotenmenge
$V := children^*(\omega) \cup \{\omega\}$ und Kantenmenge $A := \left\{ e_{x,y} = (x, y) \;\middle|\; x, y \in V \wedge y \in children(x) \right\}$
heißt der von ω erzeugte *Vorlagenabhängigkeitsgraph* (VAG)

Auf Grundlage dieser Definition kann das folgende, einfach überprüfbare Kriterium zur Feststellung rekursiver Aufrufbeziehungen angegeben werden:

Zwischen den durch $\{\omega\} \cup \left(children^*(\omega) \cap WT\right)$, $\omega \in WT$ festgelegten Prozeßvorlagen bestehen keine rekursive Aufrufbeziehungen ⇔ Der von ω erzeugte VAG G_ω ist *zyklenfrei*.

Wir verzichten an dieser Stelle auf einen formalen Beweis und auf die Darstellung von Algorithmen zur Erkennung von Zyklen in Digraphen. Entsprechende Verfahren findet man in [Neum75]. Eine Beispiel für einen azyklischen VAG zeigt Abbildung 3-21.

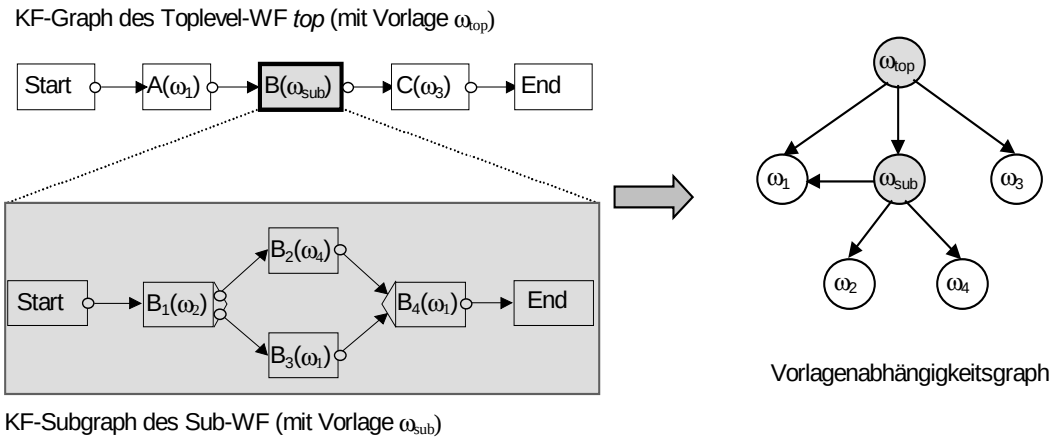


Abbildung 3-21: *Beispiel eines hierarchischen Workflows und davon abgeleiteter VAG (in Klammern sind die zugeordneten Vorlagen der Aktivitätenknoten angegeben).*

Basierend auf diesen Überlegungen formulieren wir für hierarchische Workflows die folgende Strukturierungsregel:

Strukturierungsregel KF-8 (Vermeidung rekursiver Aufrufbeziehungen)

Für eine WF-Vorlage $\omega \in WT$ fordern wir stets, daß ihr VAG G_ω zyklenfrei ist.

Diese Regel muß nicht nur bei der Modellierung von KF-Graphen beachtet werden, sondern auch im Zusammenhang mit dem dynamischen Hinzufügen komplexer Aktivitäten zu einer WF-Instanz während der Laufzeit (siehe Kapitel 7).

Wir haben in diesem Kapitel strukturelle Eigenschaften von KF-Graphen behandelt, die sowohl für die Erstellung als auch die Änderung von WF-Modellen wichtig sind. Im nun folgenden Kapitel wird auf das dynamische Verhalten von Workflows und die den verschiedenen Kontrollkonstrukten zugrundeliegende Ausführungssemantik eingegangen. Dabei wird gezeigt, daß sich bestimmte Dynamikeigenschaften eines WF-Modells, bei Zugrundelegung gewisser Fairneßannahmen, in der Mehrzahl der Fälle direkt aus den strukturellen Modelleigenschaften ableiten lassen.

4 Dynamisches Verhalten

Für die WF-Ausführung ist eine präzise Definition der Ausführungssemantik der verschiedenen Kontrollkonstrukte des ADEPT-Basismodells unerläßlich. Sie muß regeln, unter welchen Bedingungen WF-Aktivitäten aktiviert werden und welche Zustandsänderungen sich im Anschluß an ihre Ausführung ergeben sollen. Eine notwendige Voraussetzung bildet die explizite Darstellung von Zuständen auf WF-Instanzebene sowie die formale Beschreibung der Regeln zur Durchführung von Zustandstransformationen. Darauf basierend müssen Aussagen zum dynamischen WF-Verhalten getroffen werden können. Bevor wir auf diese Aspekte eingehen, beschreiben wir in Abschnitt 4.1 die Zustände und Zustandsübergänge, die sich bei der Ausführung der Instanz einer elementaren Aktivität ergeben können. In Abschnitt 4.2 diskutieren wir, wie diese Zustände auf KF-Graphebene abgebildet werden. Davon ausgehend wird in den Abschnitten 4.3, 4.4 und 4.5 eine formale Ausführungssemantik für KF-Graphen festgelegt. Der Abschnitt 4.6 behandelt wichtige Dynamikeigenschaften eines WF und formale Voraussetzungen bzw. Analyseverfahren zu ihrer Sicherstellung.

4.1 Zustände von Instanzen elementarer Aktivitäten

Der Statechart aus Abbildung 4-1 zeigt – vereinfacht dargestellt – die Zustände und Zustandsübergänge, die bei der Ausführung einer elementaren Aktivität *a* auftreten können. Initial befindet sich eine Instanz von *a* im Zustand `NOT_ACTIVATED`. In diesem Zustand ist die von der Aktivität festgelegte Aufgabe (noch) nicht bearbeitbar. Sobald alle Vorbedingungen für ihre Ausführung erfüllt sind (vgl. Abschnitt 4.3), wird die Instanz aktiviert (→ Wechsel in Zustand `ACTIVATED`). Welche Aktionen dabei konkret durchgeführt werden, hängt davon ab, ob eine automatische oder eine interaktive Aktivität vorliegt (vgl. Abschnitt 3.2.2.1). Im ersteren Fall erfolgt ein sofortiger Wechsel in den Zustand `STARTED`. Interaktive Aktivitäten dagegen werden nicht automatisch ausgeführt. Statt dessen wird bei ihnen die durchzuführende Aufgabe bei Betreten des Zustandes `ACTIVATED` zunächst allen berechtigten Akteuren[11] in ihren Arbeitslisten zur Bearbeitung angeboten. Die Instanz verbleibt noch solange im Zustand `ACTIVATED`, bis ein Benutzer sie zur Bearbeitung auswählt. Tritt dieser Fall ein, werden die entsprechenden Einträge aus den Arbeitslisten der anderen Akteure wieder entfernt, und die Aktivitäteninstanz wechselt in den Zustand `SELECTED`. Hiervon ausgehend kann der verantwortliche Bearbeiter die Ausführung der Aktivität starten, was einen Wechsel in den Zustand `STARTED` bewirkt. Solange eine Instanz noch nicht gestartet wurde, d.h. sich in einem der Zustände `ACTIVATED` oder `SELECTED` befindet, kann sie jederzeit wieder deaktiviert[12] werden. In diesem Fall werden die betroffenen Akteure benachrichtigt und ihre Arbeitslisten aktualisiert. Aktivitäteninstanzen, die sich in einem der Zustände `NOT_ACTIVATED`, `ACTIVATED` oder `SELECTED` befinden, wird im folgenden auch der übergeordnete Zustand `WAITING` zugeordnet. Wir werden uns immer dann auf übergeordnete Zustände beziehen, wenn eine Differenzierung zwischen den verschiedenen Sub-Zuständen nicht erforderlich ist. Betritt die Aktivitäteninstanz den Zustand `STARTED`, werden die mit ihr verknüpften Aktionen initiiert. (Falls es sich um die Instanz einer Nullaktivität handelt, erfolgt ein sofortiger Wechsel in den Zustand `COMPLETED`.) Vor der eigentlichen Bearbeitung der Aufgabe (z.B. durch Aufruf und Ausführung eines Aktivitätenprogramms) werden ggf. noch vorgeschaltete Aktionen durchgeführt, etwa um Dateneingaben vom Benutzer (siehe Abschnitt 5.1.2) zu verlangen. Entsprechend können nach ihrer Beendigung nachgeschaltete Aktionen zur Ausführung kommen.

Für interaktive Aktivitäten, wie dem Editieren eines Dokumentes, ist es nicht immer möglich, die Bearbeitung der Aufgabe in einem Arbeitsvorgang abzuwickeln. Um solche Aufgaben adäquat unterstützen zu können, kann in ADEPT die Bearbeitung einer Aktivitäteninstanz unterbrochen (→ Wechsel in den Zustand `SUSPENDED`) und zu einem späteren Zeitpunkt wieder aufgenommen (→ Rückkehr in

[11] Die Bestimmung der potentiellen Bearbeiter einer Aktivitäteninstanz erfolgt im ADEPT-WfMS auf der Grundlage komplexer Bearbeiterausdrücke [Kubi98]. An dieser Stelle nehmen wir der Einfachheithalber an, daß ein Akteur zur Bearbeitung einer Aktivität berechtigt ist, wenn er die von ihr geforderte Rolle einnehmen darf.

[12] Wie wir später sehen werden, kann eine solche Deaktivierung in Verbindung mit dynamischen WF-Änderungen notwendig werden, etwa wenn eine neue Aktivität vor einem bereits aktivierten, aber noch nicht gestarteten Arbeitsschritt eingefügt werden soll.

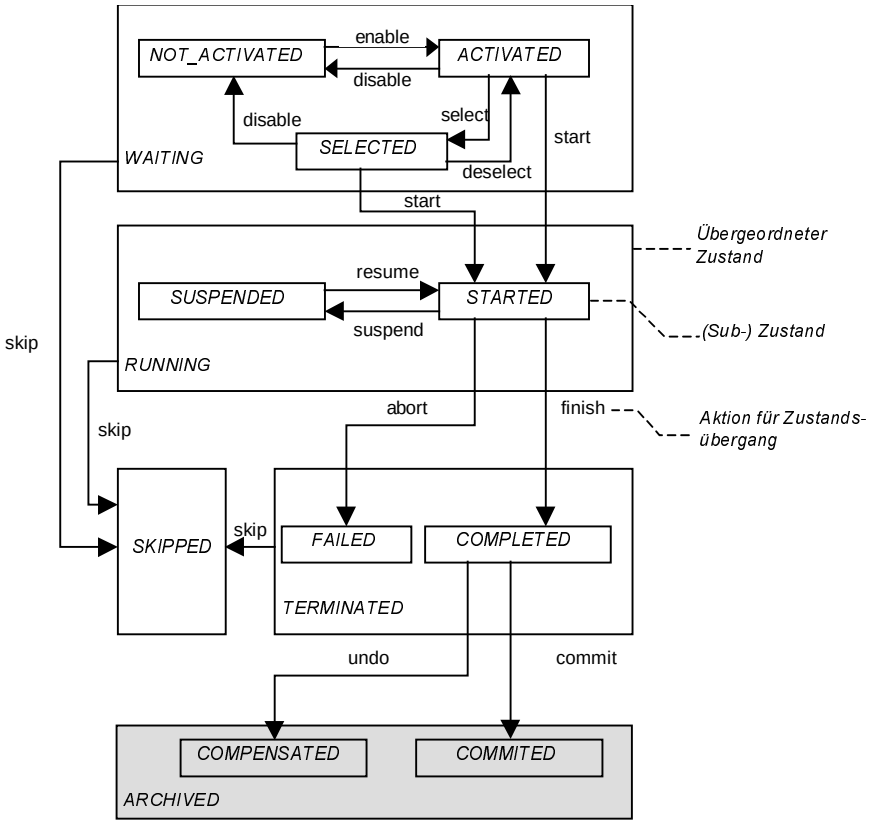


Abbildung 4-1: *Statechart einer Aktivitäteninstanz (vereinfacht)*

den Zustand STARTED) werden[13]. – Im folgenden wird Instanzen, die sich in einem der beiden Zustände STARTED oder SUSPENDED befinden, auch der übergeordnete Status RUNNING zugeordnet.

Ausgehend vom Zustand STARTED kann die Bearbeitung der Aktivitäteninstanz beendet oder abgebrochen werden. Im erstgenannten Fall wird ihr Status auf COMPLETED gesetzt. Das bedeutet, daß ihre Bearbeitung erfolgreich war und alle obligaten Ausgabeparameter $p \in OutParams^a$ der Aktivität mit einem Wert $val^p \in pDom^p$ belegt sind. Wird die Instanz dagegen abgebrochen, etwa weil die Aufgabe infolge externer Einflüsse nicht (mehr) bearbeitbar ist, wechselt sie in den Zustand FAILED. Die genaue Fehlerursache wird in dem Fehlerparameter $failure \in OutParams^a$ vermerkt, dessen mögliche Ausprägungen durch die Aktivitätenvorlage vorgegeben sind (vgl. Abschnitt 3.2). Andere Ausgabeparameter bleiben, mit Ausnahme des Statusparameters $status \in OutParams^a$, der den Wert FAILED zugewiesen bekommt, undefiniert.

Für Aktivitäteninstanzen im Zustand COMPLETED steht erst mit Beendigung des WF fest, ob die externen Auswirkungen der Aktivitätenausführung (z.B. Eintrag eines Datums in eine externe Datenbank) dauerhaft erhalten bleiben (→ Wechsel in Zustand COMMITED) oder nicht (→ Wechsel in Zustand COMPENSATED). Der Wechsel in den Zustand COMPENSATED kann notwendig werden, wenn die WF-Instanz abgebrochen oder zurückgesetzt wird und in diesem Zusammenhang die internen und externen Auswirkungen der Aktivität rückgängig gemacht werden sollen (vgl. Kapitel 6).

Eine Sonderrolle nimmt der Zustand SKIPPED ein. In diesen Zustand wechseln z.B. Aktivitäteninstanzen, die sich bisher im Zustand WAITING befanden, für die sich aber herausstellt, daß die Bedingungen für ihre Ausführung nicht mehr gegeben sind. Das ist z.B. der Fall, wenn die Aktivität in einem alternativen Bearbeitungszweig des WF enthalten ist und feststeht, daß dieser Zweig nicht mehr

[13] Ob eine Aktivität implizit, mit Beendigung des gerufenen Anwendungsbausteins, abgeschlossen werden soll oder ob dies explizit („manuell“) durch den Anwender erfolgt, kann bei ihrer Beschreibung festgelegt werden. Vorgeschaltete Aktionen werden nur bei erstmaligem Betreten des Zustands *STARTED* ausgeführt, nachgeschaltete Aktionen nur bei einem Wechsel in den Zustand *COMPLETED*.

zur Ausführung kommt. Bei parallelen Verzweigungen mit finaler Auswahl kann dieser Fall aber auch für bereits laufende (→ Zustand `RUNNING`) oder beendete Aktivitäteninstanzen (→ Zustand `TERMINATED`) eintreten, da die Voraussetzungen für ihre Ausführung im nachhinein nicht mehr erfüllt sind. Hier erfolgt ebenfalls ein Wechsel in den Zustand `SKIPPED`, wobei ggf. begleitende Aktionen (z.B. Abbruch oder Kompensation der Instanz) notwendig werden.

Bestimmte Aktivitäten können innerhalb derselben WF-Instanz mehrfach zur Ausführung kommen, beispielsweise nach Schleifen- oder Fehlerrücksprüngen. In solchen Fällen wird für jede Ausführung eine separate Aktivitäteninstanz erzeugt. In den folgenden Abschnitten zeigen wir, wie es im Verlauf der WF-Ausführung zur Erzeugung neuer Aktivitäteninstanzen kommt und welche Zustandsänderungen sich im Anschluß an ihre Beendigung ergeben.

4.2 Beschreibung von Ausführungszuständen durch Graphmarkierungen

Ein KF-Graph legt die Ausführungsreihenfolgen und -bedingungen für die Bearbeitung der Aktivitäten eines WF fest. Eine konkrete Ausprägung des WF wird als *WF-Instanz* bezeichnet. Neben der Zuordnung eines ausführbaren KF-Graphen erfordert die Repräsentation von WF-Instanzen die explizite Beschreibung von Aktivitätenzuständen auf Graphebene. Zu diesem Zweck nehmen wir Erweiterungen des im vorangegangenen Kapitel vorgestellten Metamodells vor, die Zustandsmarkierungen sowohl für Knoten als auch Kanten vorsehen. Wie diese Markierungen zustande kommen und wie sie sich im Verlauf der WF-Ausführung ändern, werden wir in Abschnitt 4.3 zeigen.

Im folgenden sei $CFS = (N, E, ...)$ ein (allgemeiner) KF-Graph, der das Kontrollflußschema für einen WF definiert. Wir bezeichnen eine Abbildung $NS: N \mapsto NodeStates$, die jedem Aktivitätenknoten einen Zustand $NS(n) \in NodeStates$ zuordnet, als *Knotenmarkierung* von CFS. Sie beschreibt den aktuellen Status der Aktivitäten einer WF-Instanz, deren Ausführung durch den KF-Graphen *CFS* gesteuert wird. Dabei definiert die Menge *NodeStates* im wesentlichen die in Abschnitt 4.1 beschriebenen Aktivitätenzustände. Zustände des Typs `ARCHIVED` (vgl. Abbildung 4-1) werden ausgeklammert, da sie für die nachfolgenden Betrachtungen nicht relevant sind. Des weiteren werden die Zustände `SUSPENDED` bzw. `STARTED` zu dem übergeordneten Zustand `RUNNING` zusammengefaßt. Somit ist die Menge *NodeStates* wie folgt definiert:

NodeStates := { `NOT_ACTIVATED`, `ACTIVATED`, `SELECTED`, `RUNNING`, `COMPLETED`, `SKIPPED`, `FAILED` }

Der *Ausführungsstatus* einer Aktivität $n \in N$ wird durch die aktuelle Markierung $NS(n)$ ihres Knotens sowie durch die Anzahl ihrer bisherigen Durchführungen $It^n \in \{0, 1, 2, ...\}$ ($It: N \mapsto \{0, 1, 2, ...\}$) beschrieben. Die wiederholte Ausführung einer Aktivität ist z.B. möglich, wenn ihr Knoten in einem Schleifenkörper enthalten ist. (It^n wird als Iterationszähler von n bezeichnet. Er wird jeweils bei Beendigung der Aktivität um eins erhöht.)

Für die formale Definition der Ausführungssemantik des ADEPT-Basismodells ist es von Vorteil, nicht nur Aktivitätenknoten, sondern auch Kanten (beliebigen Typs) mit Zustandsmarkierungen zu versehen. Der Status der Kanten eines sich in Ausführung befindlichen KF-Graphen $CFS_{instance}$ hängt vom Ausführungsstatus ihrer Quellaktivitäten und von getroffenen Verzweigungsentscheidungen ab. Er besitzt jeweils einen der Werte `NOT_SIGNALED`, `TRUE_SIGNALED` oder `FALSE_SIGNALED`. Diese drei Kantenzustände werden in der Menge *EdgeStates* zusammengefaßt. Eine Kante besitzt genau dann den Zustand `NOT_SIGNALED`, wenn ihre Quellaktivität sich in einem Zustand ungleich `TERMINATED` befindet (vgl. Abbildung 4-1), also noch nicht beendet worden ist. Wie es zur Festlegung der beiden anderen Kantenzustände kommt, werden wir in Abschnitt 4.3 sehen.

Eine Abbildung $ES: E \mapsto EdgeStates$, die jeder Kante $e \in E$ eine Markierung $ES(e) \in EdgeStates$ zuordnet, heißt *Kantenmarkierung* von *CFS*. Sie beschreibt die aktuellen Kantenzustände eines sich in Ausführung befindlichen KF-Graphen. Abbildung 4-2 zeigt die graphische Darstellung der verschiedenen Knoten- und Kantenmarkierungen. – Der KF-Ausführungsgraph einer WF-Instanz wird somit durch ein 4-Tupel $CFS_{instance} = (CFS, NS, ES, It)$ beschrieben. Dabei definiert *CFS* den KF-Graphen der WF-Instanz, dessen Knoten- und Kantenmarkierungen durch die Funktionen *NS* und *ES* beschrieben werden. *It* ordnet jedem Knoten seinen aktuellen Ausführungszähler zu.

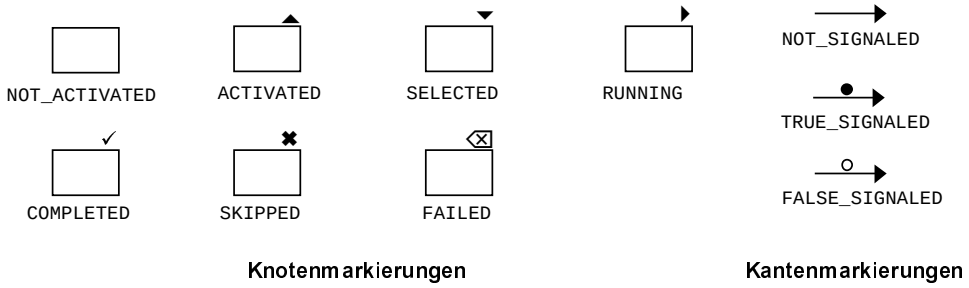


Abbildung 4-2: *Graphische Darstellung von Knoten- und Kantenmarkierungen*

4.3 Regeln zur Änderung von KF-Graphmarkierungen

Eine WF-Instanz besteht aus einer Menge von Aktivitäten, deren Koordination auf Basis des ihr zugeordneten KF-Graphen und seiner Zustandsmarkierungen erfolgt. In Abschnitt 4.1 haben wir bereits die internen Zustandsübergänge bei der Ausführung elementarer Aktivitäten kennengelernt. In diesem Abschnitt beschreiben wir, unter welchen Bedingungen Aktivitäten im Kontext einer WF-Instanz aktiviert bzw. ausgeführt werden. Die Grundlage für die Definition entsprechender *Ausführungsregeln* bilden die zuvor beschriebenen Knoten- und Kantenmarkierungen. Ebenso wichtig ist es festzulegen, wie sich diese Markierungen nach der Ausführung (→ Wechsel in den Zustand `COMPLETED`) bzw. Nichtausführung von Aktivitäten (→ Wechsel in einen der Zustände `SKIPPED` oder `FAILED`) ändern sollen, um im Kontrollfluß fortschreiten zu können (*Markierungsregeln*).

Im folgenden sei der Ausführungsgraph einer WF-Instanz durch $CFS_{instance} = (CFS, NS, ES, It)$ gegeben. Die Markierungen *NS, ES* und *It* ändern sich – ausgehend von einer *Anfangsmarkierung* – dynamisch im Verlauf der WF-Ausführung. Wir beschreiben nun sukzessive die Regeln, auf deren Basis diese Zustandsänderungen erfolgen.

Ausgehend von einem KF-Graphen wird eine neue WF-Instanz erzeugt, indem eine (logische) Kopie dieses Graphen angelegt wird. Dabei werden Knoten und Kanten des Graphen, der im folgenden auch als *KF-Ausführungsgraph* (kurz: Ausführungsgraph) bezeichnet wird, mit einer *Anfangsmarkierung* $M_0 = (NS_0, ES_0)$ versehen (NS_0 und ES_0 sind feste Markierungen), bei der alle Knoten den Zustand `NOT_ACTIVATED` und alle Kanten den Zustand `NOT_SIGNALED` erhalten (vgl. Abbildung 4-3 a). Ferner wird dem Ausführungszähler aller Knoten der Wert 0 zugewiesen:

$$NS(n) := NS_0(n) = \texttt{NOT_ACTIVATED},\ It(n) := \texttt{0}\ \forall\, n \in N$$

$$ES(e) := ES_0(e) = \texttt{NOT_SIGNALED}\quad \forall\, e \in E$$

Startet ein berechtigter Akteur die WF-Instanz, so wird der Startknoten ihres Ausführungsgraphen als `COMPLETED` und die von ihm ausgehende Kontrollkante (vgl. Regel KF-1 aus Abschnitt 3.5.1) als `TRUE_SIGNALED` markiert (vgl. Abbildung 4-3 b). Ferner wird der Ausführungszähler des Startknotens auf eins gesetzt. Ausgehend von (NS_0, ES_0) ergibt sich die Folgemarkierung (NS_1, ES_1) mit:

$$NS_1(n) := \begin{cases} \texttt{COMPLETED} & \text{für } n \in N \text{ mit } NT^n = \texttt{STARTFLOW} \\ N_0(n) & sonst \end{cases}$$

$$ES_1(e) := \begin{cases} \texttt{TRUE_SIGNALED} & \text{für } e \in E \text{ mit } ET^e = \texttt{CONTROL_E} \wedge Id^e_{source} = Start \\ E_0(e) & sonst \end{cases}$$

Im allgemeinen führt die Neumarkierung einer beliebigen Kontrollkante im KF-Ausführungsgraphen zur Neubewertung des Graphen. Grundlegend hierfür sind sog. *Ausführungsregeln*, die für die verschiedenen Knotentypen definiert sind. Sie legen – vereinfacht ausgedrückt – fest, unter welchen Knoten- und Kantenmarkierungen eine Aktivität aktiviert werden soll. Nach der Anwendung von Ausführungsregeln kann sich der Zustand einzelner Aktivitäten ändern, wodurch sie z.B. aktiviert und damit ausführbar werden. Umgekehrt führt die Beendigung von Aktivitäten (→ Wechsel in einen der

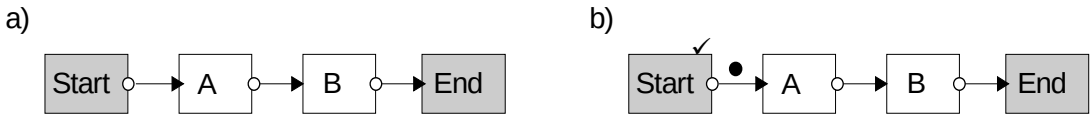


Abbildung 4-3: *a) Anfangsmarkierung eines Ausführungsgraphen und b) Markierung nach Start des WF*

Zustände COMPLETED, SKIPPED oder FAILED) zur Neubewertung der Markierungen ihrer ausgehenden Kanten. Welche Folgemarkierung sich für Kanten ergibt, ist abhängig vom Typ des beendeten Aktivitätenknotens und seiner Ausgangssemantik. Grundlage für die Kantenneubewertung bildet eine Menge von *Markierungsregeln*.

Die Ausführungs- und Markierungsregeln des ADEPT-Basismodells werden nun vorgestellt.

4.3.1 Wohlstrukturierte, azyklische KF-Graphen

Wir betrachten zunächst wohlstrukturierte, azyklische KF-Graphen (vgl. Abschnitt 3.5.1). Sie bestehen aus Sequenzen und Verzweigungen, und besitzen ausschließlich Kontrollkanten des Typs CONTROL_E.

4.3.1.1 Ausführungs- und Markierungsregeln für Sequenzen

Bei einer reinen Sequenz besitzt jeder Aktivitätenknoten (mit Ausnahme des Start- und Endknotens des KF-Graphen) genau eine einmündende und eine ausgehende Kontrollkante des Typs CONTROL_E. Dem entsprechend einfach ist die Festlegung von Ausführungs- und Markierungsregeln für dieses Konstrukt. Notwendige und hinreichende Bedingung für die Ausführung von Aktivitätenknoten innerhalb einer Sequenz ist die Markierung ihrer Eingangskontrollkante. Entsprechend wird nach ihrer Beendigung jeweils die Ausgangskontrollkante markiert. Dies wird durch die beiden folgenden Ausführungs- und Markierungsregeln verwirklicht:

Ausführungsregel $A_1^{(seq)}$ (Ausführung normaler Knoten)

Wird die einmündende Kontrollkante $e = (*, n, \texttt{CONTROL_E}) \in E$ eines Knotens n mit Eingangssemantik V_{in}^n = ONE_Of_ONE als $ES(e)$ = TRUE_SIGNALED markiert und befindet sich dieser Knoten aktuell im Zustand $NS(n)$ = NOT_ACTIVATED, so setze die Knotenmarkierung $NS(n)$ auf ACTIVATED (d.h. der Knoten n wird ausführbar).

Markierungsregel $M_1^{(seq)}$ (Beendigung normaler Knoten)

Die Zustandsmarkierung eines Knotens $n \in N$ mit $V_{out}^n \in$ {ONE_Of_ONE, NONE} ändert sich nach seiner erfolgreichen Beendigung von $NS(n)$ = RUNNING nach $NS(n)$ = COMPLETED.

Falls es sich bei n um den WF-Endknoten (NT^n = ENDFLOW) handelt, wird die WF-Instanz beendet. Andernfalls markiere die von n ausgehende Kontrollkante $e = (n, *, \texttt{CONTROL_E}) \in E$ mit $ES(e)$ = TRUE_SIGNALED und wende anschließend die Ausführungsregeln auf den modifizierten KF-Ausführungsgraphen an.

Auf Basis dieser beiden Regeln kann die Ausführung sequentieller Aktivitäten gesteuert werden. Ein Beispiel für ihre Anwendung zeigt Abbildung 4-4. An diesem Beispiel ist auch bereits erkennbar, daß in azyklischen KF-Ausführungsgraphen die Markierungen von Knoten und Kanten beim Fortschreiten des Kontrollflusses erhalten bleiben. Dadurch können Informationen zum bisherigen Verlauf der WF-Ausführung direkt dem Ausführungsgraphen entnommen werden. Erwähnenswert ist noch, daß interne Zustandswechsel einer Aktivitäteninstanz (z.B. Wechsel vom Zustand ACTIVATED in einen der Zustände SELECTED oder RUNNING) nicht immer zu einer Neubewertung von Kontrollkanten führen, sondern lediglich die Markierung des betreffenden Knotens ändern. Kantenneubewertungen werden nur dann erforderlich, wenn die betreffende Aktivität in einen (vorläufigen) Endzustand (COMPLETED oder SKIPPED) gelangt.

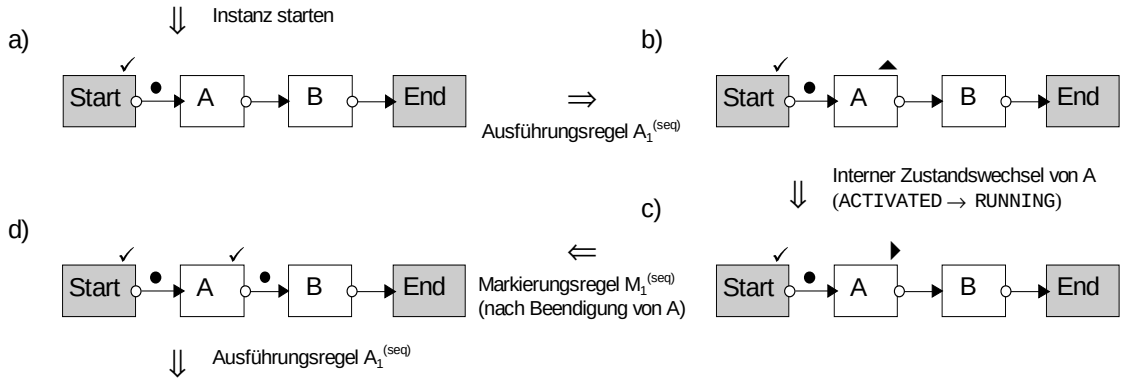


Abbildung 4-4: *Ausführungssemantik einer Sequenz*

4.3.1.2 Ausführungs- und Markierungsregeln für Verzweigungen

Eine Verzweigung wird durch kombinierte Anwendung eines AND-/OR-Split-Knotens mit einem AND-/OR-Join-Knoten gebildet, wobei die Kombination OR-Split/AND-Join ausgeschlossen ist (vgl. Regel KF-3, Eigenschaft 4). Für die Definition der Ausführungssemantik genügt es, Ausführungsregeln für Join-Knoten und Markierungsregeln für Split-Knoten festzulegen.

Wir betrachten zuerst *parallele Verzweigungen* (AND-Split/AND-Join). Hier erhalten alle ausgehenden Kontrollkanten nach Beendigung des AND-Split-Knotens den Status `TRUE_SIGNALED`. Umgekehrt kann der AND-Join-Knoten erst dann aktiviert werden, wenn alle Teilzweige abgearbeitet und dem entsprechend alle einmündenden Kontrollkanten als `TRUE_SIGNALED` markiert worden sind. Für die Festlegung dieses Verhaltens führen wir keine neuen Regeln ein, sondern ergänzen die vorangehend vorgestellten Ausführungs- und Markierungsregeln $A_1^{(seq)}$ und $M_1^{(seq)}$ entsprechend:

Markierungsregel M_1 (Beendigung von normalen Knoten oder AND-Split-Knoten)

Die Zustandsmarkierung eines Knotens $n \in N$ mit $V_{out}^{n} \in$ {`ONE_Of_ONE`, `NONE`, **`ALL_Of_ALL`**} ändert sich nach seiner erfolgreichen Beendigung von *NS(n)* = `RUNNING` nach *NS(n)* = `COMPLETED`.

Falls es sich bei *n* um den WF-Endknoten (NT^n = `ENDFLOW`) handelt, wird die WF-Instanz beendet. Andernfalls markiere **alle** von *n* ausgehenden Kontrollkanten $e \in E$ (mit $Id_{source}^{e} = n$) mit *ES(e)* = `TRUE_SIGNALED` und wende anschließend die Ausführungsregeln auf den modifizierten KF-Ausführungsgraphen an.

Ausführungsregel A_1 (Ausführung von normalen Knoten oder AND-Join-Knoten)

Sind **alle** einmündenden Kontrollkanten $e \in E$ eines Knotens *n* mit $n = Id_{dest}^{e}$ und Eingangssemantik $V_{in}^{n} \in$ {`ONE_Of_ONE`, **`ALL_Of_ALL`**} als *ES(e)* = `TRUE_SIGNALED` markiert und befindet sich der Knoten *n* aktuell im Zustand *NS(n)* = `NOT_ACTIVATED`, so setze die Knotenmarkierung *NS(n)* auf `ACTIVATED`.

Die Anwendung dieser beiden Regeln illustriert das Beispiel aus Abbildung 4-5.

Für *bedingte Verzweigungen* (OR-Split / OR-Join) – wir betrachten hier nur bedingte Verzweigungen mit expliziter Auswahl des Bearbeitungszweiges (vgl. Abschnitt 3.3.2.2 b) – muß je eine zusätzliche Markierungs- und Ausführungsregel definiert werden. Wie bereits beschrieben, besitzt jede von einem OR-Split-Knoten ausgehende Kontrollkante einen eindeutigen Kantenauswahlwert (z.B. einen Bool'schen Wert *True* oder *False*). Für einen OR-Split *s* wird der auszuführende Teilzweig zur Laufzeit auf Basis des Entscheidungsparameters DP^s bestimmt (vgl. Abschnitt 3.3.2.2 b). Entspricht dessen Wert keinem der Kantenauswahlwerte, wird ein zur Modellierungszeit festgelegter Default-Zweig zur Ausführung herangezogen. (Die Behandlung von Default-Zweigen wird in der nachfolgenden Markierungsregel M_2 aus Gründen der besseren Lesbarkeit ausgeklammert.)

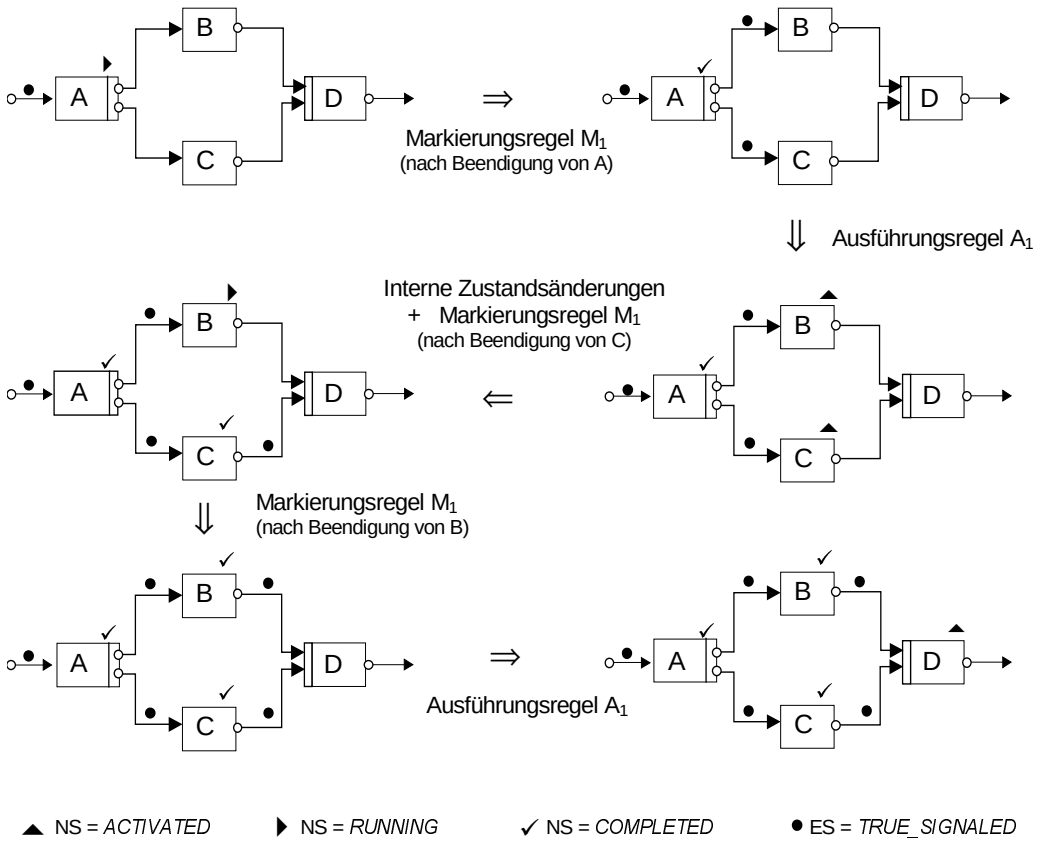


Abbildung 4-5: *Ausführungssemantik bei parallelen Verzweigungen*

Markierungsregel M_2 (Beendigung von OR-Split-Knoten)

Die Markierung eines OR-Split-Knotens $s \in N$ (mit V_{out}^{s} = `ONE_Of_ALL`) ändert sich nach seiner erfolgreichen Beendigung von $NS(s)$ = `RUNNING` nach $NS(s)$ = `COMPLETED`.

Anschließend wird diejenige Ausgangskontrollkante e des OR-Splits als $ES(e)$ = `TRUE_SIGNALED` markiert, deren Auswahlwert SC^e dem Wert des durch den Entscheidungsparameter DP^s festgelegten Datenelements entspricht (vgl. Definition 3-1). Alle anderen von s ausgehenden Kontrollkanten $e^* \in E$, $e^* \neq e$ (mit $Id_{source}^{e*} = s$) erhalten die Markierung $ES(e^*)$ = `FALSE_SIGNALED`, was bedeutet, daß die von diesen Kanten ausgehenden Teilzweige nicht bearbeitet werden sollen. Dem entsprechend werden alle Knoten dieser Teilzweige – sie liegen aufgrund der Blockstrukturierung eindeutig fest – mit `SKIPPED` und die von ihnen ausgehenden Kontrollkanten mit `FALSE_SIGNALED` markiert.

Formal:

Sei $E_{out}^{n} := \{ e \in E \mid Id_{source}^{e} = n \wedge ET^e = \texttt{CONTROL_E} \}$ die Menge aller von einem Knoten n ausgehenden normalen Kontrollkanten.

Wechselt der OR-Split-Knoten s in den Zustand $NS(s)$ = `COMPLETED`, so werden im KF-Ausführungsgraphen die folgenden Neumarkierungen für Knoten und Kanten vorgenommen:

1. Für $e \in E_{out}^{s}$ mit $SC^e = DP^s$: $ES(e)$:= `TRUE_SIGNALED`
2. $\forall\, e \in E_{out}^{s}$ mit $SC^e \neq DP^s$:

   $ES(e)$:= `FALSE_SIGNALED` und

   Für $x := Id_{dest}^{e}$ und $Z_x := (\{x\} \cup c_succ^*(x)) \cap c_pred^*(join^{CFS}(s))$:

   $\forall\, n \in Z_x$: $NS(n)$:= `SKIPPED` und $\forall\, e \in E_{out}^{n}$: $ES(e)$:= `FALSE_SIGNALED`

   (Anm: Z_x beschreibt jeweils die Knotenmenge eines nicht bearbeiteten Teilzweigs mit Startknoten x)

Wie man leicht sieht, bleibt die Markierungsregel M_2 auch bei Vorliegen „leerer“ Teilzweige korrekt anwendbar. Bei der Formulierung der Regel haben wir angenommen, daß der Wert des durch den Entscheidungsparameter festgelegten Datenelements zur Laufzeit immer einem der Auswahlwerte der vom OR-Split ausgehenden Kontrollkanten entspricht. Der ADEPT-Ansatz gestattet darüber hinaus die Festlegung von Default-Zweigen. Für jeden OR-Split kann ein solcher Default-Zweig festgelegt werden, der zur Laufzeit gewählt wird, wenn der genannte Wert keinem der Auswahlwerte entspricht. Die Berücksichtigung von Default-Zweigen macht geringfügige Modifikationen der obigen Markierungsregel erforderlich, auf deren Darstellung wir an dieser Stelle verzichten.

Nach der Markierung der Eingangskante des Bearbeitungszweiges wird, sofern es sich nicht um einen leeren Zweig handelt, sein Startknoten gemäß der Ausführungsregel A_1 aktiviert. Den Knoten und Kanten nicht ausgewählter Zweige werden dagegen die Markierungen `SKIPPED` bzw. `FALSE_SIGNALED` zugewiesen (vgl. Abbildung 4-6). Um die Ausführung am OR-Join-Knoten fortsetzen zu können, muß für ihn eine geeignete Ausführungsregel definiert werden. (Die Ausführung von OR-Join-Knoten einer parallelen Verzweigung mit finaler Auswahl wird hier noch ausgeklammert):

Ausführungsregel A_2 (Ausführung von OR-Join-Knoten – Einfacher Fall)

Ist genau **eine** einmündende Kontrollkante $e \in E$ (ET^e = `CONTROL_E`) eines OR-Join-Knotens j (mit V_{in}^j = `ONE_Of_ALL`) mit $ES(e)$ = `TRUE_SIGNALED` markiert und wurde dieser Knoten noch nicht aktiviert (d.h. es gilt $NS(j)$ = `NOT_ACTIVATED`), so setze den Knotenstatus $NS(n)$ auf `ACTIVATED`.

(Anmerkung: Im Falle einer bedingten Verzweigung sind alle anderen einmündenden Kontrollkanten des Typs `CONTROL_E` mit `FALSE_SIGNALED` markiert, vgl. Markierungsregel M_2)

Die Anwendung der Regeln M_2 und A_2 zeigt das Beispiel aus Abbildung 4-6:

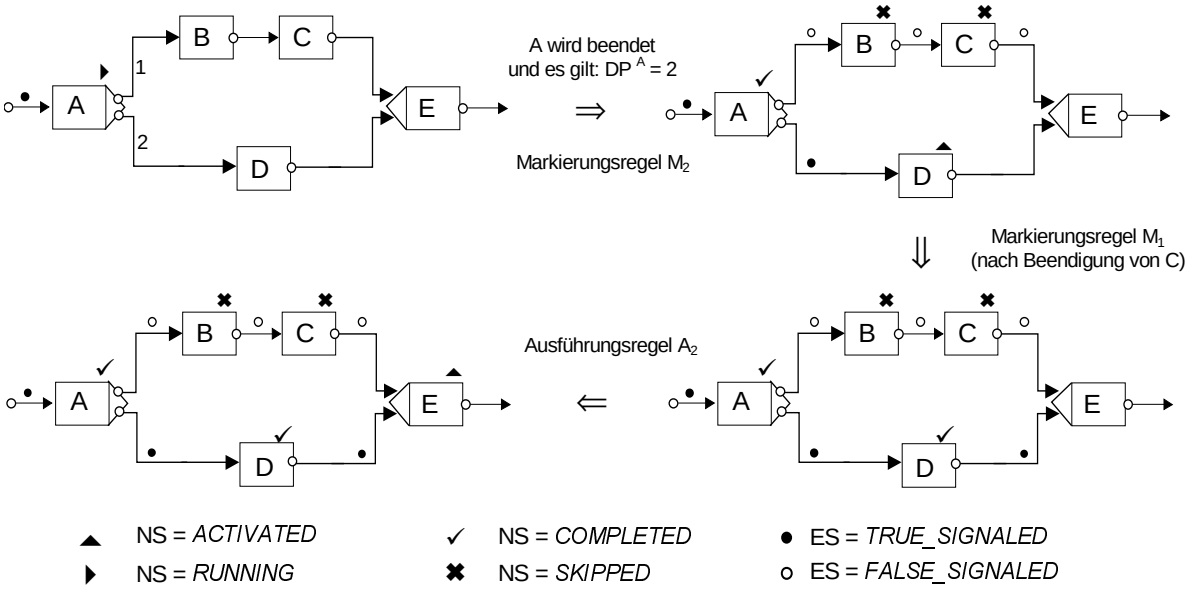


Abbildung 4-6: *Ausführungssemantik bei bedingten Verzweigungen*

Für die Festlegung der Ausführungssemantik einer *Parallelverzweigung mit finaler Auswahl* muß die Ausführungsregel A_2 für OR-Join-Knoten ergänzt werden, wohingegen die Markierungsregel M_1 für AND-Split-Knoten auch in Verbindung mit diesem Konstrukt anwendbar bleibt.

Ausführungsregel A_2 (Ausführung von OR-Join-Knoten – Allgemeiner Fall)

Ist genau **eine** einmündende Kontrollkante $e^* \in E$ (ET = `CONTROL_E`) eines OR-Join-Knotens j (mit V_{in}^j = `ONE_Of_ALL`) mit $ES(e^*)$ = `TRUE_SIGNALED` markiert und ist der Knoten j aktuell nicht markiert (d.h. es gilt $NS(j)$ = `NOT_ACTIVATED`), so setze den Knotenstatus $NS(n)$ auf `ACTIVATED`.

Besitzen die anderen in j einmündenden Kontrollkanten $e \in E$, $e \neq e^*$ ($Id_{dest}^{e} = j$) eine Markierung ungleich `FALSE_SIGNALED` – dies trifft genau für den Fall einer parallelen Verzweigung mit finaler Auswahl zu –, so werden die zugehörigen Teilzweige bis zum gemeinsamen Split-Knoten zurückgesetzt. Ihre Knoten werden mit `SKIPPED` und ihre Kanten mit `FALSE_SIGNALED` markiert.

Formal:

Sei $E_{in}^{n} := \{ e \in E \mid Id_{dest}^{e} = n \wedge ET^{e} = \texttt{CONTROL_E} \}$ die Menge aller in den Knoten $n \in N$ einmündenden normalen Kontrollkanten.

Sei j ein noch nicht aktivierter OR-Join-Knoten (d.h. es gilt $NS(j)$ = `NOT_ACTIVATED`) mit zugehörigem Split-Knoten $s = split^{CFS}(j)$. Wird eine Kante $e^* \in E_{in}^{j}$ mit $ES(e^*)$ = `TRUE_SIGNALED` markiert, so werden im KF-Ausführungsgraphen die folgenden Neumarkierungen vorgenommen:

1. Dem Knoten j wird die Markierung $NS(j)$ = `ACTIVATED` zugeordnet.
2. Gilt V_{out}^{s} = `ALL_Of_ALL`, liegt eine parallele Verzweigung mit finaler Auswahl vor. Für diesen Fall müssen die „Verliererzweige“ zurückgesetzt werden:

   $\forall\, e \in E_{in}^{j}$ mit $e \neq e^*$:

   $ES(e)$:= `FALSE_SIGNALED` und

   Für $x := Id_{source}^{e}$ und $Z_x := (\{x\} \cup c_pred^*(x)) \cap c_succ^*(s)$

   $\forall\, n \in Z_x$: $NS(n)$:= `SKIPPED` und $\forall\, e \in E_{in}^{n}$: $ES(e)$:= `FALSE_SIGNALED`

   (Anm: Z_x beschreibt jeweils die Knotenmenge des zurückzusetzenden Teilzweigs)

Ein Beispiel zeigt Abbildung 4-7.

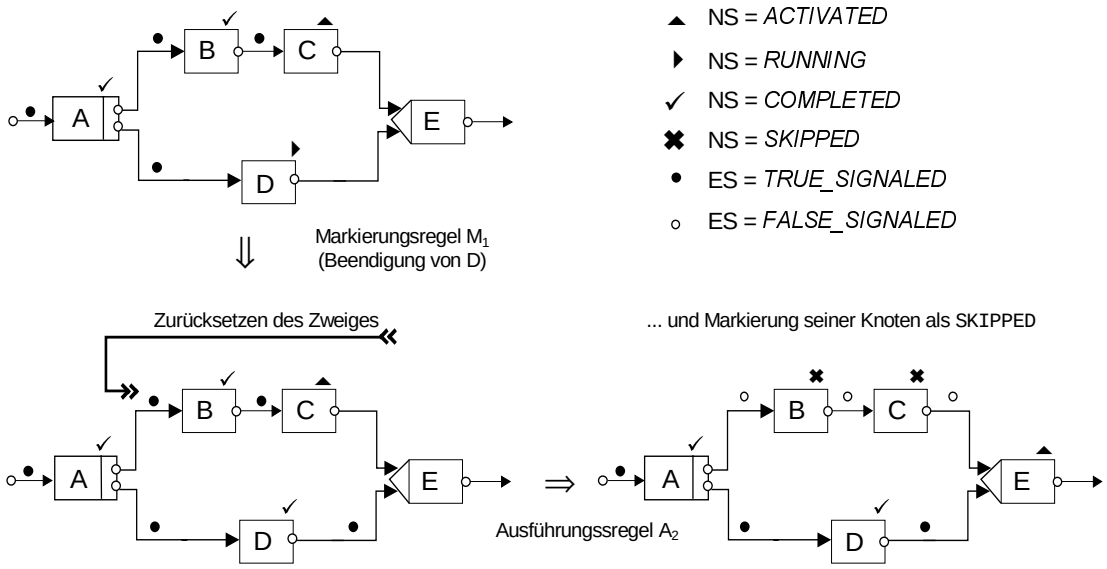


Abbildung 4-7: *Ausführungssemantik einer parallelen Verzweigung mit finaler Auswahl*

In Bezug auf das Zurücksetzen von Teilzweigen (im Beispiel der Zweig mit den Knoten `B` und `C`) macht die Ausführungsregel M_2 nur Aussagen zu den notwendigen Anpassungen von Knoten- und Kantenmarkierungen. Je nach Status eines betroffenen Aktivitätenknotens können mit dem Zurücksetzen weitere Aktionen verbunden sein, wie das Entfernen von Arbeitslisteneinträgen, der Abbruch einer laufenden Aktivität oder die Ausführung einer Kompensationsaktivität. Wir klammern diese Aspekte an dieser Stelle aus und kommen auf sie in Kapitel 6 zurück.

Wie bereits in Abschnitt 3.3.2.2 c) erörtert, ist es bei Parallelverzweigungen mit finaler Auswahl auch möglich, zuerst mehrere Teilzweige terminieren zu lassen, so daß der Anwender den „Siegerzweig“ explizit wählen kann. Diese explizite Auswahl haben wir in der obigen Ausführungsregel der Einfachheithalber nicht berücksichtigt. Wird eine solche benutzerbezogene Auswahlmöglichkeit gewünscht,

kann in ADEPT der WF-Modellierer den OR-Join-Knoten entsprechend kennzeichnen. Berechtigten Akteuren wird dann zur Laufzeit eine sog. „Auswahlaktivität“ in ihren Arbeitslisten angeboten, die aktiviert wird, sobald mindestens einer der Teilzweige abgearbeitet worden ist. Der Siegerzweig kann dann explizit ausgewählt werden, die anderen Zweige werden, wie oben definiert, zurückgesetzt. Wir wollen diese explizite Auswahl im folgenden nicht mehr gesondert berücksichtigen.

Insgesamt kann die Ausführung wohlstrukturierter, azyklischer KF-Graphen auf Basis der Markierungsregeln M_1 und M_2 sowie den beiden Ausführungsregeln A_1 und A_2 erfolgen.

4.3.2 Wohlstrukturierte, azyklische KF-Graphen mit Sync-Kanten

Werden bei der WF-Modellierung zusätzlich die in Abschnitt 3.3.2.4 vorgestellten Sync-Kanten verwendet, müssen die zuvor eingeführten Ausführungs- und Markierungsregeln ergänzt werden:

Ergänzung der Markierungsregeln M_1 und M_2
(Markierung von Sync-Kanten nach Ausführung bzw. Nichtausführung von Knoten):

Nach der erfolgreichen Ausführung eines Knotens $n \in N$ (d.h. nach seinem Wechsel in den Zustand $NS(n)$ = `COMPLETED`) muß allen von diesem Knoten ausgehenden Sync-Kanten die Markierung `TRUE_SIGNALED` zugewiesen werden. Umgekehrt müssen ausgehende Sync-Kanten eines Knoten $n^* \in N$ als `FALSE_SIGNALED` markiert werden, wenn die Markierung dieses Knotens in $NS(n^*)$ = `SKIPPED` abgeändert wird (vgl. Markierungsregel M_2).

Formal:

Neumarkierung von $n \in N$ mit $NS(n)$ = `COMPLETED`, $\Rightarrow$
$\forall\ e \in E$ mit $Id_{source}^{e} = n$ und $ET^e \in$ {`SOFT_SYNC_E`, `STRICT_SYNC_E`}: $ES(e)$:= `TRUE_SIGNALED`

Neumarkierung von $n^* \in N$ mit $NS(n)$ = `SKIPPED`, $\Rightarrow$
$\forall\ e \in E$ mit $Id_{source}^{e} = n^*$ und $ET^e \in$ {`SOFT_SYNC_E`, `STRICT_SYNC_E`}: $ES(e)$:= `FALSE_SIGNALED`

Bei der Neumarkierung einer Sync-Kante muß ihr Zielknoten ebenfalls neu bewertet werden, d.h. es muß überprüft werden, ob dieser Knoten infolge der Neumarkierung der Sync-Kante ausführbar wird. Zu diesem Zweck erweitern wir die vorangehend eingeführten Ausführungsregeln A_1 und A_2 (d.h. die nachfolgende Regel beschreibt einen Zusatz zu A_1 und A_2 und keine eigene Ausführungsregel):

Ergänzung der Ausführungsregeln A_1 und A_2:
(Aktivierbarkeit von Knoten bei einmündenden Sync-Kanten):

Eine notwendige Bedingung dafür, daß die Markierung eines noch nicht aktivierten Knotens $n \in N$ (d.h. es gilt: $NS(n)$ = `NOT_ACTIVATED`) in $NS(n)$ = `ACTIVATED` geändert werden darf, ist die Markierung aller in diesen Knoten einmündenden strikten Sync-Kanten mit `TRUE_SIGNALED` und aller einmündenden einfachen Sync-Kanten mit `TRUE_SIGNALED` oder `FALSE_SIGNALED`.

Formal:

Eine notwendige Voraussetzung für die Ausführung eines noch nicht aktivierten Aktivitätenknotens $n \in N$ ($NS(n)$ = `NOT_ACTIVATED`):

1. $\forall\ e \in E$ mit $Id_{dest}^{e} = n$ und ET^e = `STRICT_SYNC_E`, $\Rightarrow$
   $ES(e)$ = `TRUE_SIGNALED`
2. $\forall\ e \in E$ mit $Id_{dest}^{e} = n$ und ET^e = `SOFT_SYNC_E`, $\Rightarrow$
   $ES(e) \in$ {`TRUE_SIGNALED`, `FALSE_SIGNALED`}

Ein Beispiel für die Anwendung dieser Regelerweiterungen zeigt Abbildung 4-8.

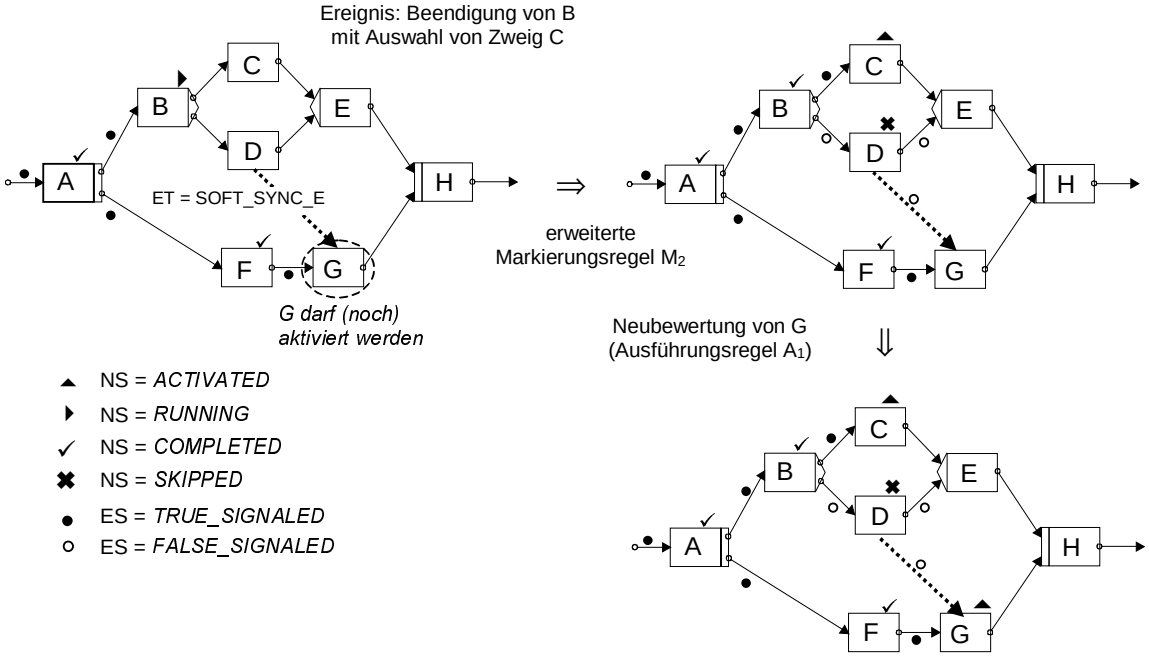


Abbildung 4-8: *Ausführungssemantik bei einfachen Sync-Kanten*

4.3.3 Wohlstrukturierte, zyklische KF-Graphen

Für die Unterstützung von Schleifen sind weitere Markierungs- und Ausführungsregeln erforderlich. Im Unterschied zu azyklischen Ausführungsgraphen werden hier bestimmte Kontrollblöcke des KF-Graphen im Verlauf der WF-Ausführung ggf. mehrfach durchlaufen – ein Umstand der, wie wir später sehen werden, auch im Zusammenhang mit dynamischen WF-Änderungen interessante Fragestellungen aufwirft. Aus diesem Grund bleiben bestimmte Knoten- und Kantenmarkierungen nicht, wie im Fall azyklischer KF-Ausführungsgraphen, bis zur Terminierung des WF erhalten. Vielmehr muß vor dem erneuten Durchlaufen der Schleife eine Neumarkierung von Knoten und Kanten des Schleifenkörper vorgenommen werden, um anschließend im Kontrollfluß fortschreiten zu können.

Im folgenden sei $CFS = (N, E, ...)$ ein wohlstrukturierter, zyklischer KF-Graph und $CFS_{instance}$ ein hiervon abgeleiteter Ausführungsgraph. Dann kommen zu den bereits vorgestellten Markierungs- und Ausführungsregeln die beiden folgenden Regeln hinzu:

Markierungsregel M_3 (Beendigung von Schleifenendknoten)

Sei durch $L_S \in LoopStartNodes^{CFS}$ der Anfangsknoten einer Schleife und durch $L_E := endloop^{CFS}(L_S)$ der zugehörige Endknoten gegeben.

Bei der Beendigung des Schleifenendknoten $n := L_E \in LoopEndNodes^{CFS}$ wird die Schleifenbedingung ausgewertet und das Ergebnis in dem durch *Loop* festgelegten Datenelement (mit Wertebereich `BOOLEAN`) festgehalten. Abhängig davon, ob dieses Datenelement den Wert *False* oder *True* annimmt, sind zwei Fälle möglich:

<u>Fall 1:</u> *Loop = False*

Die Schleifenbedingung ist nicht mehr erfüllt. Dem entsprechend wird die Schleifenausführung abgebrochen. Dabei wird der Schleifenrücksprungkante $e = (L_E, L_S, \texttt{LOOP_E})$ die Markierung `FALSE_SIGNALED` zugewiesen und die vom Schleifenendknoten ausgehenden normalen Kontrollkanten werden, entsprechend den vorangehend vorgestellten Markierungsregeln, markiert.

<u>Fall 2:</u> $LOOP^n = True$

Die Schleifenbedingung ist weiter erfüllt, d.h. die Schleife kann erneut durchlaufen werden. Zu diesem Zweck wird zuerst eine *Neumarkierung* der Knoten und Kanten des Schleifenkörpers (inkl. dem Start- und Endknoten der Schleife) vorgenommen. Formal:

$\forall\, n \in \{L_S, L_E\} \cup (c_succ^*(L_S) \cap c_pred^*(L_E))$:
$NS(n)$:= NOT_ACTIVATED und $\forall\, e \in E$ mit $Id_{source}^{\,e} = n$: $ES(e)$:= NOT_SIGNALED

Im Anschluß an diese Neumarkierung wird der Schleifenrücksprungkante $e = (L_E, L_S,$ LOOP_E) die Markierung $ES(e)$ = TRUE_SIGNALED zugewiesen.

Die Neumarkierung des Schleifenkörpers sowie des Start- und Endknotens der Schleife wird erforderlich, um beim wiederholten Durchlaufen die vorgestellten Markierungs- und Ausführungsregeln wieder korrekt anwenden zu können. Da die Knoten eines Schleifenkontrollblocks keine ausgehenden Sync-Kanten besitzen, deren Zielknoten außerhalb dieses Blocks liegen (vgl. Regel KF-7, Eigenschaft 6), wirkt sich diese Neumarkierung auch nicht auf Kontextknoten dieses Blocks aus.

Was die Ausführung von Schleifenendknoten anbetrifft, sind gegenüber „normalen“ Aktivitätenknoten keine gesonderten Regelungen notwendig. Ein Schleifenendknoten wird aktiviert, sobald seine einmündenden Kontroll- und Sync-Kanten, entsprechend der Ausführungsregel A_1, markiert sind. Die Ausführung selbst besteht aus der Auswertung der Schleifenbedingung, deren Ergebnis im Datenelement *LOOP* festgehalten wird. Auch die erstmalige Aktivierung eines Schleifenanfangsknotens erfolgt auf Basis der bereits vorgestellten Ausführungsregeln. Das gilt jedoch nicht mehr für die wiederholte Aktivierung eines solchen Knotens, die beim Signalisieren der einmündenden Schleifenrücksprungkante erfolgt. Für diesen Fall kommt die Ausführungsregel A_3 zur Anwendung:

Ausführungsregel A_3 (Wiederholte „Ausführung“ von Schleifenanfangsknoten)

Ein Schleifenanfangsknoten $L_S \in LoopStartNodes^{CFS}$ mit der Markierung $NS(L_S)$ = NOT_ACTIVATED kann aktiviert werden, d.h. seine Markierung darf in $NS(L_S)$ = ACTIVATED abgeändert werden, wenn die in L_S einmündende Schleifenrücksprungkante $e = (L_E, L_S,$ LOOP_E), $L_E := endloop^{CFS}(L_S)$ neu mit $ES(e)$ = TRUE_SIGNALED markiert wird.

Die Anwendung der Regeln M_3 und A_3 zeigt Abbildung 4-9:

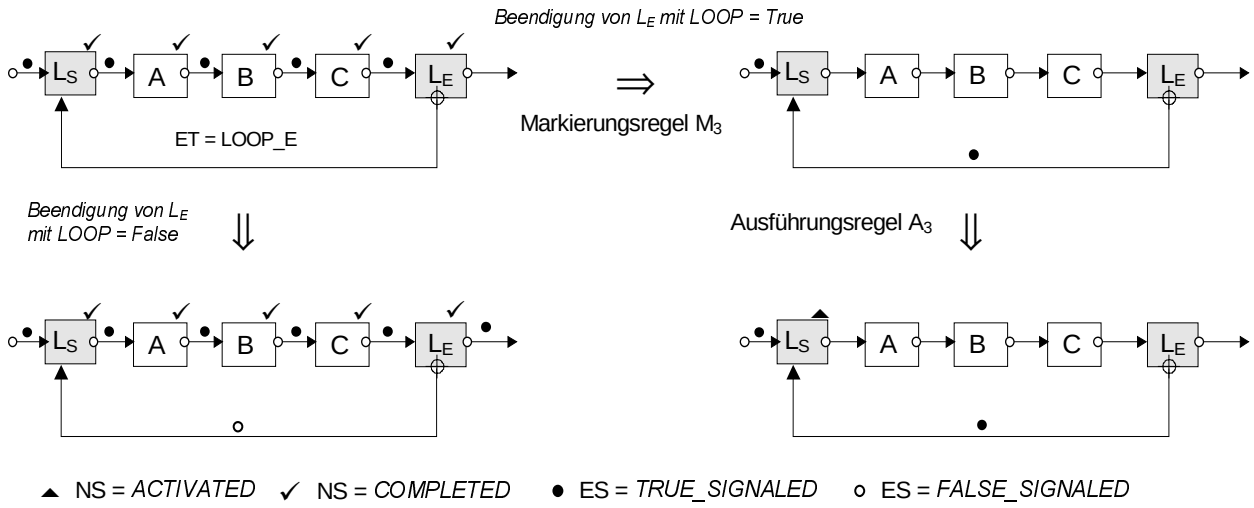


Abbildung 4-9: *Ausführung von Schleifen*

Die Ausführung von WF-Instanzen basiert somit insgesamt auf drei Ausführungs- und drei Markierungsregeln – diese Regelmenge muß noch erweitert werden, wenn das WF-Metamodell hierarchische Kontrollflußstrukturen und Fehlerkanten unterstützen können soll. Die Regeln wirken bei oberflächlicher Betrachtung etwas komplizierter als z.B. einfache Schaltregeln bei (konventionellen) Petri-Netzen, sind aber leicht implementierbar (vgl. Abschnitt 4.4) und bieten – wie wir noch zeigen werden – einige Vorteile im Hinblick auf dynamische Strukturänderungen. Durch den gewählten Markierungsansatz kann, ohne aufwendige Analyseverfahren anwenden zu müssen, vermieden werden, daß Änderungen in bereits abgearbeiteten oder in nicht mehr ausführbaren Bereichen des KF-Ausführungsgraphen vorgenommen werden. Des weiteren läßt sich nach der Durchführung struktu-

reller Modifikationen der WF-Zustand auf Basis der vorgestellten Markierungs- und Ausführungsregeln mit einfachen Mitteln automatisch neu bestimmen – unabhängig von der Art und dem Umfang der durchgeführten Änderungen.

4.4 Bestimmung der Folgemarkierung nach Beendigung einer Aktivität

Wie im vorangehenden Abschnitt gezeigt, müssen nach der Beendigung einer Aktivität ggf. mehrere Knoten- und Kantenmarkierungen angepaßt werden, um mit der Kontrolle fortfahren zu können. In Abbildung 4-8 wurden z.B. nach der Beendigung des Knotens `B` die beiden Knoten `C` und `G` gemeinsam aktiviert. Hierzu wurde je eine Markierungs- und Ausführungsregel angewendet. Generell muß beim Aktivitäten-Scheduling, ausgehend von der aktuellen Markierung (NS_{old}, ES_{old}) des Ausführungsgraphen, seine *Folgemarkierung* (NS_{new}, ES_{new}) bestimmt werden, die sich nach der Beendigung einer bestimmten Aktivitäteninstanz ergibt. ADEPT stellt hierfür das *NextMarking*-Verfahren bereit, das die vorgestellten Markierungs- und Ausführungsregeln nicht nur isoliert, sondern auch in Kombination miteinander anwendet. Dadurch können die mit der Beendigung einer Aktivitäteninstanz verknüpften Aktionen, wie z.B. die Aktualisierung von Arbeitslisten, effizient verwirklicht werden.

Wir verzichten an dieser Stelle auf die Darstellung dieses Verfahrens und verweisen auf den in Anhang C beschriebenen Algorithmus. Er bestimmt für einen gegebenen Ausführungsgraphen seine Folgemarkierung, die sich im Anschluß an die Ausführung einer bestimmten Aktivität ergibt. Um festzustellen, welche der von einem beendeten Aktivitätenknoten ausgehenden Kanten neu markiert werden müssen, wird – abhängig vom Knotentyp – eine geeignete Markierungsregel angewendet. Anschließend wird für die Zielknoten neu markierter Kanten überprüft, ob sie infolge der Änderung des Kantenstatus nun aktiviert werden dürfen. Trifft dies zu, erhalten sie die Markierung `ACTIVATED`. Der Algorithmus implementiert alle Markierungs- und Ausführungsregeln des ADEPT-Basismodells, berücksichtigt also auch komplexere Fälle, wie das Zurücksetzen oder die Nichtauswahl von Teilzweigen, bei denen es zur Neumarkierung mehrerer Knoten und Kanten kommt. – Nach der Bestimmung der Folgemarkierung können notwendige Aktionen, wie die Aktualisierung von Arbeitslisten, zusammenhängend vorgenommen werden. Die neu in Arbeitslisten einzutragenden Aufgaben entsprechen zum Beispiel genau denjenigen Aktivitäten, deren Markierung von `NOT_ACTIVATED` in `ACTIVATED` abgeändert wurde. Entsprechend lassen sich, durch den Vergleich von alter und neuer Markierung oder durch die explizite Kennzeichnung der geänderten Markierungen, die Aktivitäten ermitteln, die aus Arbeitslisten entfernt, abgebrochen oder zurückgesetzt werden sollen.

4.5 Ausführung von Sub-Workflows

Die Kontrollflußsteuerung einer WF-Instanz erfolgt durch die Interpretation ihres Ausführungsgraphen, wofür die vorgestellten Ausführungs- und Markierungsregeln die formale Grundlage bilden. Bisher haben wir gezeigt, wie auf der Basis eines solchen Ausführungsgraphen die Bearbeitung der Aktivitäten einer bestimmten Hierarchiestufe koordiniert wird. In diesem Abschnitt gehen wir darauf ein, wie bei der WF-Ausführung mit *Sub-Workflows*, d.h. Instanzen komplexer Aktivitäten, verfahren wird. Die Ausführung eines Sub-WF kann wiederum aus der Bearbeitung einer Menge von Instanzen elementarer oder komplexer Aktivitäten besteht.

4.5.1 Bearbeitungszustände einer (Sub-)Workflow-Instanz

Die Bearbeitungszustände einer (Sub-)WF-Instanz stellen sich aus Sicht der übergeordneten WF-Instanz (*Super-Workflow*) ähnlich dar, wie bei Instanzen elementarer Aktivitäten (vgl. Abschnitt 4.1). Allerdings werden die Zustandsübergänge intern teilweise durch andere Ereignisse ausgelöst, und es werden andere Aktionen beim Betreten bzw. Verlassen der Zustände ausgeführt.

Eine komplexe Aktivität wechselt nach ihrer Aktivierung (→ Betreten des Zustands `ACTIVATED`) entweder direkt in den Zustand `STARTED` oder sie wird berechtigten Benutzern in Arbeitslisten angeboten, etwa wenn fehlende Eingabedaten manuell festgelegt werden sollen. Beim Wechsel in den Zustand `STARTED` wird der Startknoten der Sub-WF-Instanz beendet und seine Nachfolgeaktivität aktiviert. Im Anschluß daran können die Aktivitäten des Sub-WF, entsprechend den durch seinen KF-Graphen festgelegten Ausführungsbedingungen, bearbeitet werden. Bei Bedarf kann die Bearbeitung

des (Sub-)WF unterbrochen (→ Zustand SUSPENDED) und später wieder fortgesetzt werden (→ Rückkehr in den Zustand STARTED). Während einer solchen Suspendierung dürfen keine Aktivitäten mehr aktiviert oder gestartet werden, noch nicht beendete Aktivitäteninstanzen können aber noch abgeschlossen werden. Wie im Fall elementarer Aktivitäten wird die Bearbeitung des (Sub-)WF entweder erfolgreich abgeschlossen (→ Zustand COMPLETED) oder sie scheitert (→ Zustand FAILED). Der Wechsel in den Zustand COMPLETED tritt ein, wenn der Endknoten des Sub-WF als COMPLETED markiert wird. Tabelle 4-1 faßt die wesentlichen Beziehungen zusammen, die zwischen dem Zustand einer (Sub-) WF-Instanz und den Zuständen ihrer Start- bzw. Endaktivität bestehen.

Zustand der (Sub-) Workflow-Instanz	Zustand ihrer Startaktivität	Zustand ihrer Endaktivität
NOT_ACTIVATED	NOT_ACTIVATED	NOT_ACTIVATED
ACTIVATED	ACTIVATED	NOT_ACTIVATED
STARTED	COMPLETED	NOT_ACTIVATED
COMPLETED	COMPLETED	COMPLETED

Tabelle 4-1: *Beziehungen zwischen dem Status des WF und den Zuständen seiner Start-/Endaktivität*

4.5.2 Instanzabhängigkeitsgraph und Ausführung von Sub-Workflows

Für die WF-Ausführung müssen genaue Informationen zu den Aufrufbeziehungen zwischen Super- und Sub-Workflows vorliegen. Insbesondere muß bei der Beendigung eines Sub-WF feststehen, welche übergeordnete WF-Instanz davon betroffen ist und für welchen Aktivitätenknoten ihres Ausführungsgraphen eine entsprechende Markierungsänderung vorgenommen werden soll. In diesem Abschnitt skizzieren wir, wie solche Abhängigkeiten in ADEPT verwaltet werden. Dabei beschränken wir uns auf eine informelle Darstellung und klammern Aspekte, die das Datenkontextmanagement und den Datenfluß zwischen Super- und Sub-WF betreffen, zunächst aus (siehe Kapitel 5).

Jeder Instanz einer elementaren oder komplexen Aktivität wird bei ihrer Erzeugung ein eindeutiger Bezeichner $instanceId \in Instances$ zugeordnet. Ihre Ausführung erfolgt in einem konkreten Kontext, der durch den übergeordneten Super-WF $parentInst(instanceId) \in Instances$ eindeutig festliegt. (Für einen Toplevel-WF $topId \in Instances$ ist der Kontext undefiniert, d.h. es gilt: $parentInst(topId) =$ UNDEFINED). Dem entsprechend wird ein Toplevel-WF auf einen Baum von Aktivitäteninstanzen (sog. *Instanzabhängigkeitsgraph*; kurz: IAG) abgebildet, dessen Blätter den Instanzen elementarer und dessen Nichtblätter den Instanzen komplexer Aktivitäten entsprechen. Die Struktur eines IAG selbst ergibt sich dynamisch zur Laufzeit, wobei neue Knoten jeweils bei der Erzeugung von Aktivitäteninstanzen hinzukommen. Bei der Verwendung von Schleifen ist es möglich, daß eine Aktivität mehrfach instantiiert wird. Die Struktur von Vorlagen- und Instanzabhängigkeitsgraph ist deshalb im allgemeinen nicht deckungsgleich. Allerdings gilt die folgende Beziehung (vgl. Abschnitt 3.7) :

$$parentInst(id_{child}) = id_{parent} \Rightarrow Template(id_{child}) \in children(Template(id_{parent})).$$

Mit der in Abschnitt 3.7 definierten Regel KF-8 (→ Ausschluß rekursiver Aufrufbeziehungen) folgt sofort, daß ein IAG zyklenfrei ist.

Die Ausführung eines Sub-WF erfolgt nach folgendem Prinzip: Eine Sub-WF-Instanz wird erzeugt, wenn auf Ebene des übergeordneten Super-WF *superWId* die Bedingungen für die Ausführung dieser komplexen Aktivitäten – gemäß den ADEPT-Ausführungsregeln – erfüllt sind. In diesem Fall wird dem IAG ein neuer Knoten *subWId* als direkter Nachfolger des Knotens *superWId* (d.h. $parentInst(subWId) = superId$) dynamisch hinzugefügt. Anschließend wird der Sub-WF aktiviert und ggf. gestartet, was im IAG zur Hinzunahme weiterer Knoten „unterhalb“ von *subWId* führt (vgl. Abbildung 4-10). Der Teilbaum mit Wurzel *subWId* wird während der Ausführung des Sub-WF sukzessive erweitert, bis dieser terminiert, d.h. bis im zugehörigen Ausführungsgraphen die Endaktivität als COMPLETED markiert wird. In diesem Fall wird im KF-Ausführungsgraphen der übergeordneten WF-Instanz (sofern kein Toplevel-WF vorliegt), der betreffende Aktivitätenknoten ebenfalls als COMPLETED markiert, so daß der WF auf dieser Ebene ebenfalls fortschreiten kann.

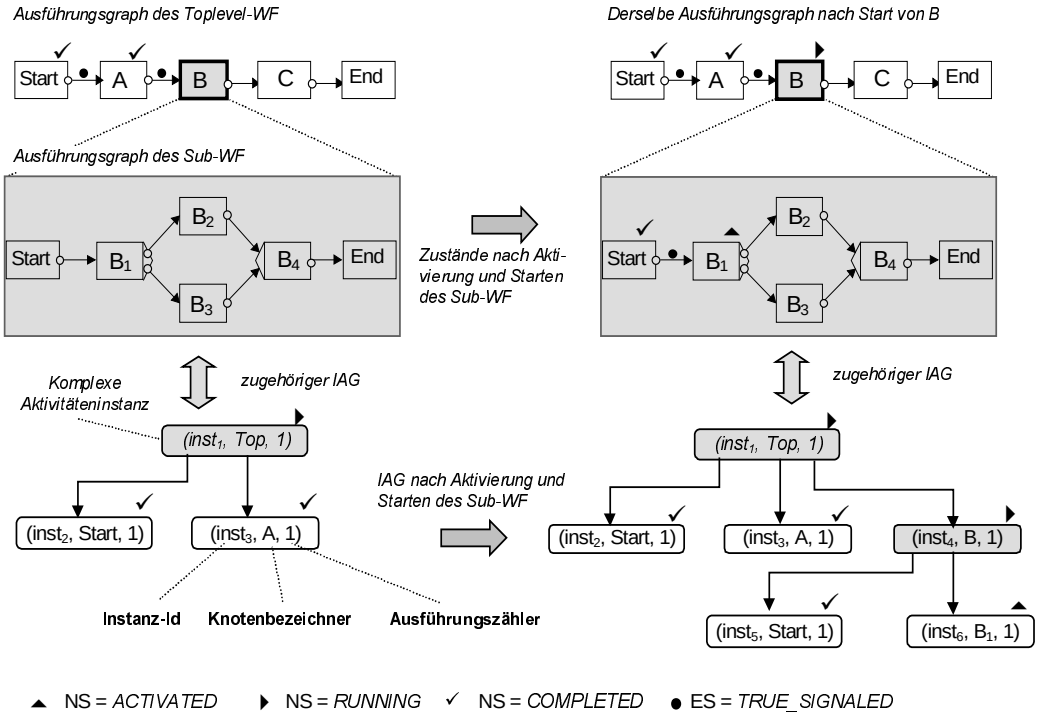


Abbildung 4-10: *Ausführung eines hierarchischen WF im ADEPT-Basismodell*

Das einer komplexen Aktivität (bzw. ihrer Vorlage) zugeordnete WF-Modell wird erst bei ihrer Instanziierung gebunden und nicht – wie bei einigen WfMS üblich – bereits zur Modellierungszeit. Dadurch erreichen wir, daß Änderungen in der Definition von WF-Schemata bei der Ausführung übergeordneter WFs berücksichtigt werden können, solange der Sub-WF noch nicht instantiiert wurde.

4.6 Dynamische Eigenschaften von KF-Graphen und deren Überprüfung

Bereits bei der Modellierung eines Arbeitsablaufs sollte seitens des WfMS oder seiner Modellierungskomponenten sichergestellt werden, daß bei der späteren WF-Ausführung keine undefinierten oder unerwünschten Systemzustände (z.B. Verklemmungen) auftreten können. Dies wäre z.B. der Fall, wenn unter einer erreichbaren Markierung des KF-Ausführungsgraphen kein Knoten mehr aktiviert werden kann, der WF aber noch nicht terminiert ist. Das Auftreten solcher Fälle wird im ADEPT-Ansatz bereits im Vorfeld der WF-Implementierung ausgeschlossen. In vielen Fällen ist dies durch die Art der Konstruktion der KF-Graphen möglich, so daß aufwendige Analysen entfallen. Eine Ausnahme bilden strikte Sync-Kanten, bei deren Verwendung ggf. aufwendigere Überprüfungen erforderlich sind. Hierauf kommen wir am Ende dieses Abschnitts zurück.

Bei hierarchischen WFs (vgl. Abschnitte 3.7 und 4.5) müssen die gewünschten Dynamikeigenschaften für *alle* KF-Graphen bzw. KF-Subgraphen sichergestellt werden, die zur Ausführung des WF bzw. seiner Sub-Workflows herangezogen werden. Kann z.B. ein Sub-Workflow nicht terminieren, so kann es auch auf der übergeordneten Ebene zu einem „Stillstand“ im WF kommen. D.h. wir müssen Aussagen zum Dynamik-Verhalten auf jeder Hierarchiestufe treffen. Ansonsten ergeben sich im Kontext hierarchischer WFs – nicht zuletzt aufgrund des Ausschlusses rekursiver Aufrufstrukturen – keine weiteren Besonderheiten in Bezug auf ihr dynamisches Verhalten. Wir können uns daher im folgenden auf die Untersuchung der Dynamikeigenschaften einzelner KF-Graphen konzentrieren.

4.6.1 Grundlegende Definitionen

Im folgenden sei durch $CFS_{instance} = (CFS, NS, ES, It)$ der KF-Ausführungsgraph einer WF-Instanz gegeben. Eine Aktivität $n \in N$ heißt *aktiviert* unter der Markierung $M = (NS, ES)$, wenn gilt: $NS(n) = \texttt{ACTIVATED}$. Wir nehmen im folgenden an, daß die Ausführung einer Aktivität nach ihrer Aktivierung in endlicher Zeit erfolgt, es sei denn, diese wird wieder zurückgenommen (z.B. infolge

des Zurücksetzens von Teilzweigen). Nach Beendigung der Aktivität werden die Markierungen des KF-Ausführungsgraphen neu bewertet (vgl. Abschnitt 4.4), woraus die Folgemarkierung $M_{new} = (NS_{new}, ES_{new})$ resultiert. In Anlehnung an Petri-Netze [Baum96] sagen wir "*n schaltet* von $M = (NS, ES)$ nach $M_{new} = (NS_{new}, ES_{new})$" und wir schreiben:

$$M\ [n\rangle\ M_{new}\ \ (\text{bzw. } (NS, ES)\ [n\rangle\ (NS_{new}, ES_{new}))$$

Handelt es sich bei n um einen OR-Split-Knoten, sind mehrere direkte Folgemarkierungen möglich, abhängig von der Wahl des Ausführungszweiges bzw. des ihm zugeordneten Auswahlwertes *sc*. Wir verwenden in diesem Fall die Notation $M\ [n\rangle_{sc}\ M_{new}$ und sagen „*n* schaltet von *M* nach M_{new} bei Auswahl von *sc*“. Dasselbe gilt auch in Verbindung mit der Ausführung von Schleifenendknoten. Die Bestimmung der Folgemarkierung, die sich nach dem Schalten einer Aktivität ergibt, kann auf Basis des in Abschnitt 4.4 erwähnten *NextMarking*-Verfahrens erfolgen. Da interne Zustandswechsel der Aktivität (z.B. Wechsel vom Zustand `ACTIVATED` nach `SELECTED` oder `RUNNING`) für die nachfolgenden Betrachtungen nicht relevant sind, werden sie hier der Einfachheithalber weggelassen, d.h. wir betrachten im folgenden nur die Zustände `NOT_ACTIVATED`, `ACTIVATED`, `COMPLETED` und `SKIPPED`.

Definition 4-1: ***Erreichbarkeitsmenge, Erreichbarkeitsgraph***

Sei $M_0 = (NS_0, ES_0)$ die Anfangsmarkierung eines KF-Ausführungsgraphen (oder eines abgeschlossenen Kontrollblocks eines solchen Graphen). Dann heißt die Markierung $M^* = (NS^*, ES^*)$ von M_0 aus *erreichbar*, wenn es ein Folge von Aktivitäten $n_0, n_1, ..., n_{k-1}$ gibt, so daß gilt:

$$M_0\ [n_0\rangle\ M_1\ [n_1\rangle\ M_2\ ....\ [n_{k-1}\rangle\ M^* \text{ (sog. Schaltfolge)}^{14}.$$

Ferner bezeichne $[M_0\rangle$ die Menge aller möglichen Markierungen, die – ausgehend von M_0 – unter Verwendung der vorgestellten Markierungs- und Ausführungsregeln erreicht werden können ($[M_0\rangle$ heißt auch *Erreichbarkeitsmenge* des KF-Ausführungsgraphen (unter M_0)).

Ein gerichteter Graph $G(M_0) = (V, E)$ mit Knotenmenge $V = [M_0\rangle$ und Kantenmenge $E = \{(M_1, M_2, n) \mid M_1, M_2 \in V, n \in N$ und $M_1\ [n\rangle\ M_2\ \}$ heißt *Erreichbarkeitsgraph* eines KF-Ausführungsgraphen.

Bei der Festlegung dieser Definition haben wir uns an Begriffen aus dem Petri-Netz-Bereich [Baum96, Ober96] orientiert. Die Erreichbarkeitsmenge eines KF-Ausführungsgraphen ist allerdings, im Gegensatz zu S/T-Netzen bzw. -Systemen, stets endlich, da seine Knotenmenge endlich ist und jeder Knoten nur endlich viele Markierungen[15] besitzen kann. Bevor wir auf die Verwendung von Erreichbarkeitsmenge und -graph eingehen, stellen wir in Abschnitt 4.6.2 wichtige Forderungen im Hinblick auf das dynamischen Verhalten von KF-Graphen auf. Dazu benötigen wir die Definition 4-2:

Definition 4-2: ***Aktivierbarkeit von Aktivitäten, Endmarkierung***

Eine Aktivität $n \in N$ heißt von der Markierung $M = (NS, ES)$ aus *aktivierbar*, wenn n unter M aktiviert ist, oder aber wenn es eine Folgemarkierung M^* von M gibt unter der n aktiviert wird:

$NS(n)$ = `ACTIVATED` oder $(\exists\ M^* = (NS^*, ES^*) \in [M\rangle$ mit $NS^*(n)$ = `ACTIVATED` $)$

Eine Markierung $M = (NS, ES) \in [M_0\rangle$ heißt *Endmarkierung* von $CFS_{instance}$, falls unter M der Endknoten des WF als NS(`End`) = `COMPLETED` markiert ist. $M_{end}(M_0) \subset [M_0\rangle$ bezeichne die Menge aller von M_0 aus erreichbaren Endmarkierungen.

Basierend auf dieser Definition können wir die folgenden Grundaussagen zu den erreichbaren Markierungen von KF-Ausführungsgraphen treffen:

---

[14] Wir haben an dieser Stelle Schaltungen bei OR-Split- oder Schleifenendknoten nicht explizit markiert.

[15] Relevant sind dabei die 4 Knotenmarkierungen `NOT_ACTIVATED`, `ACTIVATED`, `COMPLETED` und `SKIPPED`. Bei endlicher Knotenmenge N sind somit theoretisch $4^{|N|}$ Markierungen möglich. Dieser Fall wäre allerdings nur gegeben, wenn alle Knoten parallel ausführbar sind.

Satz 4-1: ***Knotenzustände bei Erreichen von Endmarkierungen***

Sei $CFS_{instance}$ ein KF-Ausführungsgraph mit korrektem Kontrollflußschema $CFS = (N, E, ...)$. Unter der Anfangsmarkierung M_0 von $CFS_{instance}$ sei der Startknoten aktiviert und alle anderen Knoten als `NOT_ACTIVATED` markiert.

a) Dann gilt für jede Folgemarkierung $M = (NS, ES) \in [M_0\rangle$ von $CFS_{instance}$ und beliebiges $n^* \in N$:
$[\, NS(n^*) \in \{\texttt{COMPLETED}, \texttt{SKIPPED}\} \Rightarrow \forall\, n \in pred^*(n): NS(n^*) \in \{\texttt{COMPLETED}, \texttt{SKIPPED}\}\,]$

b) Für eine Endmarkierung $M \in M_{end}(M_0)$ gilt: $\forall\, n \in N: NS(n) \in \{\texttt{COMPLETED}, \texttt{SKIPPED}\}$

Satz 4-1 a) besagt, daß wenn ein Knoten n^* eines Ausführungsgraphen als beendet bzw. als nicht mehr ausführbar markiert ist, dies auch für alle seine Vorgänger (bzgl. Kontroll- und Sync-Kanten) zutrifft. Man kann also aus den aktuellen Markierungen ableiten, welche Bereiche des Ausführungsgraphen bereits abgearbeitet worden sind bzw. im Fall von Schleifen welche Aktivitäten des Schleifenkörpers zuletzt bearbeitet worden sind. Liegt eine Endmarkierung vor, so besitzen gemäß Satz 4-1 b) alle Knoten des Ausführungsgraphen eine der beiden Zustandsmarkierungen `COMPLETED` oder `SKIPPED`. Das bedeutet, daß beim Erreichen einer Endmarkierung keine Aktivität mehr ausgeführt wird bzw. ausgeführt werden kann und der WF somit erfolgreich terminiert ist.

Beweis zu Satz 4-1:

Satz 4-1 a) kann direkt aus der Markierung des Knotens n^* als `COMPLETED` oder `SKIPPED`, den strukturellen Eigenschaften von KF-Graphen und den vorgestellten Ausführungs- und Markierungsregeln abgeleitet werden. Wir verzichten an dieser Stelle auf eine technische Beweisführung. Satz 4-1 b) ist ein Spezialfall von Satz 4-1 a): Unter der Endmarkierung M ist der Endknoten `End` des Ausführungsgraphen gemäß Definition 4-2 mit $NS(\texttt{End}) = \texttt{COMPLETED}$ markiert. Demzufolge muß wegen Satz 4-1a) für alle Knoten der Menge $pred^*(\texttt{End}) = N - \{\texttt{End}\}$ (vgl. Satz 3-2 b) gelten, daß sie als `COMPLETED` oder `SKIPPED` markiert sind.

q.e.d.

Im allgemeinen kann es zu einem KF-Ausführungsgraphen, abhängig von den jeweils durchlaufenen Ausführungszweigen, mehrere Endmarkierungen geben. Sofern die Ausführung einer WF-Instanz nicht abgebrochen werden soll, sollte gewährleistet sein, daß stets eine solche Endmarkierung erreicht wird – unabhängig von den bei der WF-Ausführung getroffenen Verzweigungsentscheidungen.

4.6.2 Korrektheitsforderungen

Sei $CFS_{instance}$ ein KF-Ausführungsgraph oder ein abgeschlossener Kontrollblock (vgl. Definition 3-10) eines KF-Ausführungsgraphen. Unter der Anfangsmarkierung M_0 von $CFS_{instance}$ sei der Startknoten aktiviert, ansonsten kein weiterer Knoten. Für die Ausführung bzw. das dynamische Verhalten von Workflows bestehen die folgenden Korrektheitsforderungen:

1. Jeder beliebige Knoten $n \in N$ (inkl. dem Endknoten von $CFS_{instance}$) muß, ausgehend von der Anfangsmarkierung M_0, mindestens über eine zulässige Schaltfolge aktivierbar sein.
2. Für jede von M_0 aus erreichbare Markierung M muß gelten, daß der WF von M aus nach endlich vielen Schaltungen terminiert, d.h. eine Endmarkierung erreicht wird (Eine Endmarkierung eines Kontrollblockes liegt vor, wenn sein Endknoten als `COMPLETED` markiert ist):

   $\forall\, M \in [M_0\rangle - M_{end}(M_0) : \exists\, M' \in [M\rangle \text{ mit } M' \in M_{end}(M_0)$

Die 1. Forderung soll sicherstellen, daß es keine Knoten gibt, die überhaupt nie zur Ausführung kommen können. Die 2. Forderung zielt darauf, daß der WF von jeder erreichbaren Markierung aus in der Folge terminiert. Dies wäre nicht gesichert, wenn der Ausführungsgraph in einen Zustand gelangen kann, unter dem kein Knoten mehr ausführbar ist, der Endknoten aber noch nicht die Markierung $NS(\texttt{End}) = \texttt{COMPLETED}$ besitzt (sog. *Deadlock*). Wir zeigen nun, unter welchen Voraussetzungen diese beiden Forderungen erfüllt sind bzw. wann zusätzliche Analysen zu ihrer Überprüfung notwendig werden. Dabei gehen wir davon aus, daß wir es mit korrekten KF-Graphen, die den in Abschnitt 3.5 vorgestellten Strukturierungsregeln genügen, zu tun haben.

Des weiteren treffen wir gewisse „*Fairneßannahmen*“ hinsichtlich der WF-Ausführung:

(FA1) Eine aktivierte Aktivität wird, wenn sie nicht infolge des Rücksetzens von Ausführungszweigen als `SKIPPED` markiert wird, in endlicher Zeit bearbeitet. Insbesondere werden Benutzerinteraktionen in endlicher Zeit abgeschlossen und aufgerufene Aktivitätenprogramme korrekt ausgeführt.

(FA2) Es gibt keine Endlosschleifen, d.h. jede Schleife terminiert nach endlich vielen Durchläufen.

(FA3) Der Workflow wird nicht abgebrochen oder anderweitig irregulär beendet.

(FA4) Jeder Teilzweig einer bedingten Verzweigung ist zur Laufzeit prinzipiell wählbar. (Auf Modellierungsebene kann lediglich sichergestellt werden, daß die Auswahlwerte der von einem OR-Split-Knoten n ausgehenden Kontrollkanten im Wertebereich des diesem Knoten zugeordneten Entscheidungsparameters DP^n liegen.)

4.6.3 Azyklische KF-Graphen mit einfachen Sync-Kanten

Wir betrachten zuerst wohlstrukturierte, azyklische KF-Graphen, zu deren Bildung Sequenzen, Verzweigungen (unterschiedlichen Typs) und einfache Sync-Kanten verwendet werden dürfen. Wie in Abschnitt 3.3.2 diskutiert, läßt sich damit bereits eine Vielzahl von Arbeitsabläufen adäquat modellieren. KF-Graphen mit strikten Sync-Kanten oder Schleifen werden zunächst ausgeklammert. Ein zentrale Aussage zum dynamischen Verhalten entsprechender KF-Ausführungsgraphen macht Satz 4-2.

Satz 4-2: ***Aktivierbarkeit / Termination bei wohlstrukturierten, azyklischen Ausführungsgraphen***

Sei $CFS_{instance}$ ein KF-Ausführungsgraph mit Kontrollflußschema $CFS = (N, E, ...)$ und Anfangsmarkierung M_0. Unter M_0 sei der Startknoten von $CFS_{instance}$ aktiviert und alle anderen Knoten seien als `NOT_ACTIVATED` markiert. Ferner besitze CFS nur Kanten der Typen `CONTROL_E` und `SOFT_SYNC_E`. Schleifenkanten und strikte Sync-Kanten seien dagegen ausgeschlossen. Bei Zugrundelegung der getroffenen Fairneßannahmen gilt dann:

a) Unter jeder Folgemarkierung $M_t \in [M_0\rangle$ von $CFS_{instance}$, die ungleich einer Endmarkierung ist, gibt es mindestens eine aktivierte (bzw. selektierte oder sich in Ausführung befindliche) Aktivität.

b) Ausgehend von einer beliebigen Markierung $M_t \in \in [M_0\rangle$ wird nach endlich vielen Schaltungen und in endlicher Zeit eine Endmarkierung erreicht, d.h. der WF terminiert.

c) Jeder beliebige Knoten $n \in N$ ist, ausgehend von der Anfangsmarkierung M_0, über mindestens eine zulässige Schaltfolge aktivierbar.

Satz 4-2 besagt unter anderem, daß wohlstrukturierte, azyklische Ausführungsgraphen, für deren Bildung auch einfache Sync-Kanten verwendet werden dürfen, in endlicher Zeit terminieren. Das bedeutet, daß, ausgehend von der Anfangsmarkierung, die Menge nicht mehr aktivierbarer Knoten (Zustand `COMPLETED` oder `SKIPPED`), sukzessive zunimmt. Wir sprechen in diesem Zusammenhang vom *Fortschreiten* des Kontrollflusses. Grundlage hierfür bilden die strukturellen Eigenschaften von KF-Graphen dieser Klasse, wie Blockstrukturierung und Zyklenfreiheit, sowie die in Abschnitt 4.3 festgelegte Ausführungssemantik des ADEPT-Basismodells. Interessant ist in diesem Zusammenhang, daß sich Aussagen zum dynamischen Verhalten von KF-Graphen direkt aus ihren strukturellen Eigenschaften und somit auf Basis einfacher graphentheoretischer Betrachtungen gewinnen lassen (vgl. Beweis zu Satz 4-2). Weitere Überprüfungen (z.B. durch Erreichbarkeitsanalyse) müssen im vorliegenden Fall nicht durchgeführt werden. Hierauf werden wir später, bei der Analyse des dynamischen Verhaltens allgemeiner KF-Graphen, zurückgreifen. Wie wir noch sehen werden, treffen die Aussagen von Satz 4-2 unter den gemachten Fairneßannahmen auch für KF-Graphen mit Schleifen zu.

<u>Beweis zu Satz 4-2 :</u>

Im folgenden bezeichne $M_t = (NS_t, ES_t) \in [M_0\rangle$ eine Markierung von $CFS_{instance}$ zu einem (beliebigen) Zeitpunkt $t > 0$ ($t = 0$ bezeichne den Startzeitpunkt der Ausführung). Die Menge der unter M_t nicht mehr aktivierbaren Knoten sei durch $R_t := \{ n \in N \mid NS_t(n) \in \{\texttt{SKIPPED}, \texttt{COMPLETED}\}\} \subseteq N$ de-

finiert. Diese Menge umfaßt alle Aktivitätenknoten, die zum Zeitpunkt t eine der beiden Markierungen SKIPPED oder COMPLETED besitzen. Mit den Markierungsregeln M_1 und M_2 folgt sofort, daß alle ausgehenden Kontroll- und Sync-Kanten eines Knotens $m \in R_t$ als FALSE_SIGNALED bzw. TRUE_SIGNALED markiert sind (*).

Beweis von Satz 4-2 a):

Wir führen den Beweis von Satz 4-2 a) durch Widerspruch. Dazu nehmen wir an, daß es eine Folgemarkierung M_t gibt, die keine Endmarkierung darstellt und unter der kein Knoten aktiviert (bzw. selektiert oder sich in Ausführung befindlich) ist.

Annahme:

$NS_t(End) \neq$ COMPLETED und $\forall\, n \in N - R_t : NS_t(n) \neq$ ACTIVATED (bzw. $NS_t(n) =$ ACTIVATED) (**)

O.B.d.A. können wir – wie bereits erwähnt – die Markierungen für Knotenzwischenzustände (SELECTED, RUNNING) außer Betracht lassen, da sie für Aussagen zum dynamischen Verhalten nicht relevant sind. Dem entsprechend besitzen Knoten der Menge $N - R_t$ eine der beiden Markierungen NOT_ACTIVATED oder ACTIVATED. Für einen Knoten $n \in N - R_t$ mit der Markierung NOT_ACTIVATED gilt darüber hinaus, daß mindestens eine seiner einmündenden Kontroll- oder Sync-Kanten noch nicht signalisiert worden ist. Dies folgt unmittelbar aus den Ausführungsregeln A_1 und A_2 und der Nichtberücksichtigung strikter Sync-Kanten. Mit (*) folgt dann, daß es für einen noch nicht aktivierten Knoten $n \in N - R_t$ mindestens einen Vorgängerknoten geben muß, der eine Markierung ungleich SKIPPED und ungleich COMPLETED besitzt. Wendet man diese Aussage auf die Annahme (**) an, so ergibt sich:

$\forall\, n \in N - R_t$: $\exists\, x \in pred(n)$ mit $x \notin R_t$.

Sei nun $n^* \in N - R_t$,
$\Rightarrow \exists\, x_1 \in pred(n^*)$: $x_1 \notin R_t$ (d.h. $x_1 \in N - R_t$), $\Rightarrow \exists\, x_2 \in pred(x_1)$: $x_2 \notin R_t$ (d.h. $x_2 \in N - R_t$), usw.

Nach diesem Prinzip ließe sich eine unendliche Rückwärtspfeilfolge $n^* \leftarrow x_1 \leftarrow x_2 \leftarrow ....$ konstruieren, deren Knoten keine der Markierungen COMPLETED, SKIPPED oder ACTIVATED tragen. Da per Definition die Anzahl der Knoten eines KF-Graphen endlich ist, könnte eine solche unendliche Pfeilfolge nur auftreten, wenn der Graph einen Zyklus enthält. Da dies jedoch durch Strukturierungsregel KF-4* für Kontroll- und Sync-Kanten ausgeschlossen ist, muß die Annahme (**) falsch sein.

Beweis von Satz 4-2 b):

Sei $M_t = (NS_t, ES_t) \in [M_0\rangle$ eine beliebige von M_0 aus erreichbare Markierung von $CFS_{instance}$, aber noch keine Endmarkierung dieses Graphen. Zu zeigen ist, daß ausgehend von M_t nach endlich vielen Schaltungen eine Endmarkierung M_{t^*} ($t < t^*$) erreicht wird.

Dazu bezeichne R_t wieder die Menge der unter M_t nicht mehr aktivierbaren Knoten mit Markierungen SKIPPED bzw. COMPLETED. Die Menge R_t ist monoton wachsend, d.h. es gilt $R_t \subseteq R_{t^*}$ für $t < t^*$ (***) (Begründung: Ein als COMPLETED markierter Knoten bleibt entweder in diesem Zustand oder er wechselt beim Zurücksetzen eines Zweiges einer Parallelverzweigung mit finaler Auswahl in den Zustand SKIPPED. Ein als SKIPPED markierter Knoten ändert seine Markierung überhaupt nicht mehr. Schleifen können nach den Voraussetzungen des Satzes nicht auftreten).

Da M_t keiner Endmarkierung entspricht, muß es wegen Satz 4-2 a) mindestens einen aktivierten Knoten $n \notin R_t$ geben. Gemäß Fairneßannahme FA1 wird dieser Knoten, wenn er nicht infolge des Rücksetzens eines Zweiges als SKIPPED markiert wird, in endlicher Zeit (d.h. zu einem Zeitpunkt t_1 mit $t < t_1 < \infty$) beendet und erhält dabei die Markierung COMPLETED. Mit (***) folgt dann, daß die Menge Menge R_{t_1} mindestens ein Element mehr als R_t umfaßt, unabhängig davon, welche Markierung der Knoten n (SKIPPED oder COMPLETED) erhält, d.h. es gilt: $R_t \subset R_{t_1}$. Entspricht M_{t_1} der Endmarkierung, so sind wir fertig. Andernfalls läßt sich nach demselben Grundprinzip eine endliche Folge von Zeitpunkten $t < t_1 < t_2 < ... < \infty$ konstruieren, so daß gilt: $R_t \subset R_{t_1} \subset R_{t_2} \subset R_{t_3} \subset ...$ Da die Gesamtknotenmenge N endlich ist, ergibt sich nach endlich vielen Schaltungen eine Endmarkierung.

Beweis von Satz 4-2 c):

Aussage c) kann direkt aus den Struktureigenschaften entsprechender KF-Graphen und den festgelegten Ausführungs- und Markierungsregeln abgeleitet werden. Eine wichtige Voraussetzung bildet auch die Fairneßannahme FA4 (siehe Abschnitt 4.6.2), bei der wir davon ausgehen, daß bei einer bedingten Verzweigung jeder Teilzweig zur Ausführung kommen kann. Wir verzichten an dieser Stelle auf eine technische Beweisführung.

q.e.d.

4.6.4 Azyklische KF-Graphen mit einfachen und strikten Sync-Kanten

Wie das nachfolgende Beispiel aus Abbildung 4-11 zeigt, treffen die Aussagen von Satz 4-2 bei Verwendung strikter Sync-Kanten nicht immer zu. So kommt es im Ausführungsgraphen aus Abbildung 4-11 b) zu einer Verklemmung (*Deadlock*), die ohne Verwendung der strikten Sync-Kante `D` → `F` nicht aufgetreten wäre. In Abbildung 4-11 b) sind `F` und damit auch der Endknoten `G` unter der gegebenen Markierung nicht mehr aktivierbar. Solche unerwünschten Zustände lassen sich allein durch strukturelle Einschränkungen im allgemeinen nicht vollständig ausschließen. Im KF-Graphen aus Abbildung 3-10 b) etwa, wird ebenfalls eine strikte Sync-Kanten verwendet, deren Quellaktivität dem Teilzweig einer bedingten Verzweigung angehört. Im Gegensatz zu dem gerade beschriebenen Beispiel sind hier jedoch die genannten Korrektheitsbedingungen erfüllt, d.h. der WF terminiert stets und jeder Knoten ist – zumindest über eine Schaltfolge – aktivierbar.

4.6.4.1 Grundlegende Überlegungen

Mit zunehmender Komplexität eines KF-Graphen wird es für den Anwender naturgemäß immer schwieriger, noch zu überblicken, ob es infolge der Verwendung strikter Sync-Kanten zu Verklemmungen oder zur Nichtaktivierbarkeit von Aktivitäten kommen kann. Hier bedarf es zusätzlicher, systemseitiger Analysen, auf die nun eingegangen wird. Im Kern wird dabei ähnlich wie bei Petri-Netz-basierten Ansätzen verfahren, indem für jeden Ausführungsgraphen seine Erreichbarkeitsmenge – sie ist, unabhängig von der Struktur des KF-Graphen, stets endlich – und sein Erreichbarkeitsgraph ermittelt werden (vgl. Definition 4-1). Liegen diese beiden Größen vor, lassen sich grundlegende Fragestellungen bzgl. des dynamischen Verhaltens eines Ausführungsgraphen einfach beantworten, etwa „Kann eine gegebene Markierung erreicht werden?“, „Gibt es für jeden Knoten zumindest eine Markierung, unter der er aktiviert ist?“ oder „Können Verklemmungen auftreten, d.h. gibt es Nicht-Endmarkierungen, unter denen kein Knoten mehr ausführbar ist?“.

Bevor wir einen Algorithmus zur Durchführung von *Erreichbarkeitsanalysen* und verschiedene Optimierungen vorstellen, gehen wir kurz auf die wesentlichen Zusammenhänge ein, die zwischen Erreichbarkeitsgraph und den in Abschnitt 4.6.2 geforderten Dynamikeigenschaften bestehen:

Satz 4-3 (*Beziehungen zwischen Erreichbarkeitsgraph und Dynamikeigenschaften)*

Sei $CFS_{instance}$ ein KF-Ausführungsgraph mit Schema $CFS = (N, E, ...)$ und Anfangsmarkierung $M_0 = (NS_0, ES_0)$. $G(M_0) = (V, E)$ bezeichne den Erreichbarkeitsgraphen von $CFS_{instance}$. Dann gilt:

a) Ein Knoten $n \in N$ kann während der WF-Ausführung aktiviert werden ⇔ der Erreichbarkeitsgraph $G(M_0)$ besitzt mindestens einen Knoten $M = (NS, ES) \in V = [M_0\rangle$, für den gilt: $NS(n)$ = `ACTIVATED` (M ist dann vom Knoten M_0 aus über eine Pfeilfolge erreichbar).

b) Es gibt (mindestens) eine WF-Ausprägung bzw. -Ausführung, nach deren Beendigung der Knoten $n \in N$ den Status `COMPLETED` besitzt ⇔ der Erreichbarkeitsgraph $G(M_0)$ besitzt einen Knoten $M = (NS, ES) \in V \cap M_{end}(M_0)$ ohne ausgehende Kante und es gilt: $NS(n)$ = `COMPLETED`.

c) Es gibt eine von M_0 aus erreichbare Nicht-Endmarkierung M, unter der kein Knoten mehr ausführbar ist, d.h. der WF kann nicht mehr fortschreiten ⇔ der Erreichbarkeitsgraph $G(M_0)$ besitzt einen Knoten $M \in V$, der keine ausgehende Kante besitzt und für den gilt: $M \notin M_{end}(M_0)$.

Diese Beziehungen folgen unmittelbar aus den Definitionen des Abschnitts 4.6.1. Wir verzichten an dieser Stelle auf einen formalen Beweis. Aussagen zum dynamischen Verhalten von Aus-

führungsgraphen, zu deren Bildung strikte Sync-Kanten verwendet werden, lassen sich somit durch Analysen ihres Erreichbarkeitsgraphen gewinnen. Wir werden hierzu später ein Beispiel präsentieren.

4.6.4.2 Erreichbarkeitsanalyse

Wir stellen nun einen Algorithmus vor, der für einen abgeschlossenen Kontrollblock (vgl. Definition 3-10) eines KF-Ausführungsgraphen (ggf. auch den kompletten Ausführungsgraphen) seine Erreichbarkeitsmenge sowie mögliche Schaltfolgen für Aktivitäten bestimmt. Hieraus kann unmittelbar der zugehörige Erreichbarkeitsgraph konstruiert werden. Der Algorithmus geht davon aus, daß unter der Anfangsmarkierung des Kontrollblocks sein Startknoten aktiviert ist, während alle anderen Knoten (Kanten) die Markierung `NOT_ACTIVATED` (`NOT_SIGNALED`) tragen.

Ermittlung der Erreichbarkeitsmenge (Auszug)

```
Erreicht:= {M0}; Erledigt:= ∅; Schaltungen:= ∅;
while Erreicht ≠ Erledigt do
    wähle ein M = (NS, ES) ∈ Erreicht – Erledigt;
    Aktiviert:= {n ∈ N | NS(n) = ACTIVATED}        // Menge aller unter M aktivierten Knoten
    // Bestimmung aller unmittelbaren Folgemarkierungen, die bei Schalten von x resultieren
    forall x ∈ Aktiviert do
        // Falls x kein OR-Split-Knoten ist, so gibt es genau eine unmittelbare
        // Folgemarkierung (Schleifen sind an dieser Stelle noch ausgeschlossen)
        if V_out^x ≠ ONE_Of_ALL then  // x ist kein OR-Split-Knoten → 1 Folgemarkierung
            Bestimme Folgemarkierung M‘ nach Schalten von x   (d.h. M [x⟩ M‘)
            // vgl. hierzu Abschnitt 4.4
            Erreicht:= Erreicht ∪ {M‘}; // Menge ändert sich nicht, falls M‘ schon enthalten
            Schaltungen:= Schaltungen ∪ { (M, x, M‘)}
        else  // x ist OR-Split-Knoten
            SC:= Menge der Auswahlwerte aller von x ausgehenden Kontrollkanten
                des Typs ET = CONTROL_E
            forall code ∈ SC do
                // Bestimmung der Folgemarkierung bei Auswahl des Teilzweiges, dessen
                // Eingangs-Kontrollkante den Auswahlwert code besitzt
                Bestimme Folgemarkierung M‘ nach Schalten von x mit DP^x = code
                (d.h. M [x⟩_code M‘)
                Erreicht:= Erreicht ∪ {M‘};
                Schaltungen:= Schaltungen ∪ { (M, x:code, M‘)}
            done
        endif
    done
    Erledigt:= Erledigt ∪ {M}
done
```

Dieser Algorithmus terminiert nach endlich vielen Schritten, da die Erreichbarkeitsmenge eines KF-Ausführungsgraphen immer endlich ist.

Beispiel 4-1: ***Erreichbarkeitsanalyse***

Gegeben sei der in Abbildung 4-11 a) dargestellte Kontrollblock (mit aktiviertem Startknoten A) eines Ausführungsgraphen. Der in b) abgebildete Graph zeigt eine Folgemarkierung dieses Blockes, unter der kein Knoten mehr ausgeführt werden kann, obwohl der Endknoten `G` noch nicht markiert worden ist. In diesem Fall kommt es zum „Stillstand“ des Workflows, d.h. der Kontrollfluß kann nicht mehr wie gewünscht fortschreiten. Verursacht wird diese Verklemmung durch die „ungeschickte“ Verwendung der strikten Sync-Kante von `D` nach `F`. Wendet man den oben skizzierten Erreichbarkeitsalgorithmus auf den Graphen aus Abbildung a) an, so ergeben sich die in Tabelle 4-2 dargestellten Markierungen und Schaltungen (Kantenmarkierungen sind hier nicht dargestellt). Unter diesen Markierungen befindet sich mit M_6 auch der in Abbildung b) dargestellte Zustand.

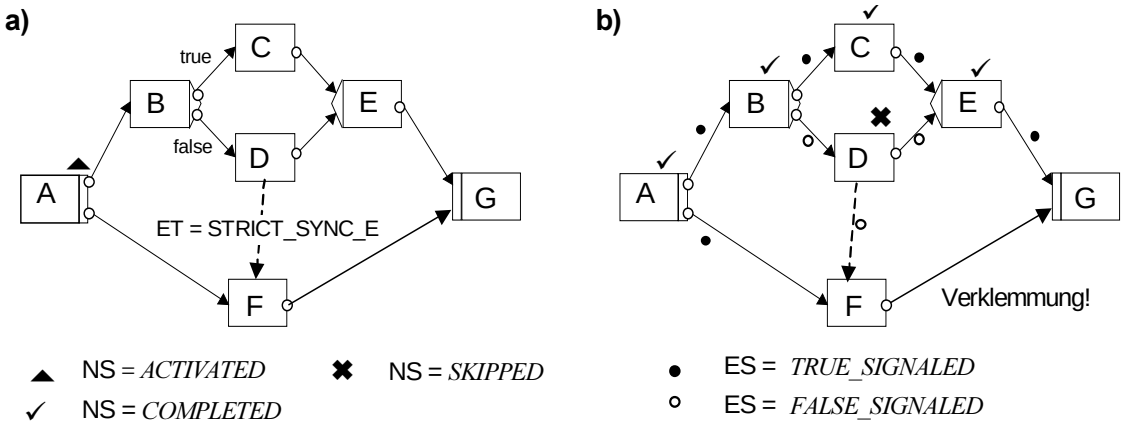


Abbildung 4-11: *(a) Anfangsmarkierung eines Verzweigungskontrollblocks und (b) mögliche Folgemarkierung, unter der kein Knoten mehr aktivierbar ist*

Markierungs-Nummer	Knotenmarkierungen							Schaltungen
	A	B	C	D	E	F	G	
M_0	▲	–	–	–	–	–	–	M_0 [A⟩ M_1
M_1	✓	▲	–	–	–	–	–	M_1 [B⟩$_{True}$ M_2 M_1 [B⟩$_{False}$ M_3
M_2	✓	✓	▲	✖	–	–	–	M_2 [C⟩ M_4
M_3	✓	✓	✖	▲	–	–	–	M_3 [D⟩ M_5
M_4	✓	✓	✓	✖	▲	–	–	M_4 [E⟩ M_6
M_5	✓	✓	✖	✓	▲	▲	–	M_5 [E⟩ M_7 M_5 [F⟩ M_8
M_6	✓	✓	✓	✖	✓	–	–	**Kein Knoten aktivierbar!**
M_7	✓	✓	✖	✓	✓	▲	–	M_7 [F⟩ M_9
M_8	✓	✓	✖	✓	▲	✓	–	M_8 [E⟩ M_9
M_9	✓	✓	✖	✓	✓	✓	▲	**Endknoten aktiviert**

Tabelle 4-2: *Erreichbarkeitsanalyse*

Der zur Erreichbarkeitsmenge $[M_0\rangle = \{M_0, M_1, ..., M_9\}$ korrespondierende Erreichbarkeitsgraph ist in Abbildung 4-12 dargestellt. Der mit einem Kreis versehene Knoten M_9 entspricht dabei der einzigen von M_0 aus erreichbaren Endmarkierung. Wie man dem Erreichbarkeitsgraphen ebenfalls entnehmen kann, terminiert der Workflow z.B. durch Anwendung der Schaltfolge (A, B_{False}, D, E, F). Allerdings ist die Endmarkierung M_9 nicht von jeder Nicht-Endmarkierung des Graphen über eine Kantenfolge erreichbar. So besitzt der Knoten M_6 keine ausgehende Kante, was bedeutet, daß unter dieser Markierung keine Aktivität mehr ausführbar ist (vgl. Abbildung 4-11 b). Da es sich bei M_6 um keine Endmarkierung handelt, muß ein Fehler in der Modellierung vorliegen. Aufgrund der im vorangegangenen Abschnitt angestellten Überlegungen kann dieser Fehler im vorliegenden Beispiel nur aus der Verwendung der strikten Sync-Kante resultieren.

Würde man für den KF-Graphen aus Abbildung 3-10 b) eine Erreichbarkeitsanalyse durchführen (ausgehend von der Aktivierung des Knotens A), so ergäben sich ausschließlich Markierungen M_i, unter denen jeweils mindestens ein Knoten schalten kann und von denen aus stets eine Endmarkierung erreicht wird (bzw. M_i selbst eine Endmarkierung darstellt).

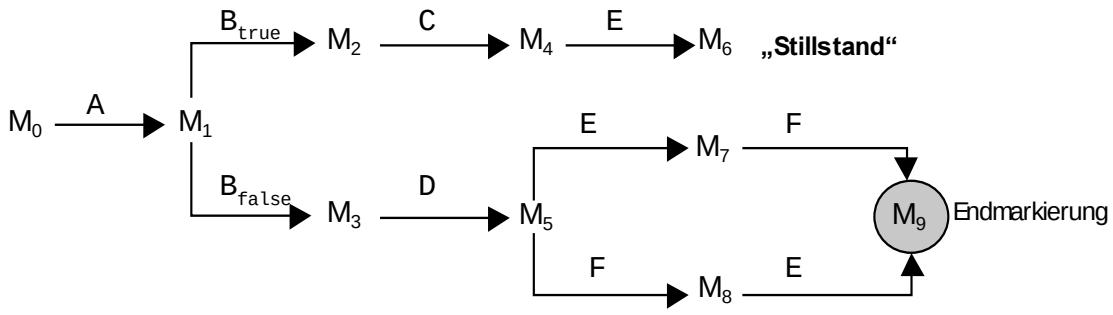


Abbildung 4-12: *Erreichbarkeitsgraph*

4.6.4.3 Reduktion möglicher Markierungen

Die Berechnung der Erreichbarkeitsmenge kann sich bereits für einfache Ausführungsgraphen aufwendig gestalten. Der Grund dafür liegt in der kombinatorischen Vielfalt von Graphmarkierungen, die sich bei parallelen Ausführungszweigen ergeben kann. Es muß deshalb das Ziel sein, die Anzahl möglicher Markierungen vor der Durchführung der eigentlichen Erreichbarkeitsanalyse geeignet einzuschränken (z.B. durch Reduktion der zu betrachtenden Knotenmenge), ohne daß dadurch Aussagen zum dynamischen Verhalten „verfälscht“ werden. Tatsächlich ist zur Überprüfung der genannten Dynamikeigenschaften (Aktivierbarkeit von Knoten, Terminierung des WF) in der Regel keine vollständige Erreichbarkeitsanalyse des kompletten Ausführungsgraphen notwendig. Vielmehr genügt es zu zeigen, daß bei Verwendung strikter Sync-Kanten die geforderten Eigenschaften erhalten bleiben. Zu diesem Zweck werden genau diejenigen (disjunkten) Verzweigungsblöcke ermittelt, bei denen Kanten dieses Typs vorkommen (sog. *relevante Kontrollblöcke*). Dadurch reduzieren sich die jeweils zu betrachtenden Knotenmengen und folglich auch die Mengen möglicher Graphmarkierungen. Durch die Analyse erreichbarer Markierungen wird für jeden dieser Kontrollblöcke separat untersucht, ob er im Anschluß an seine Aktivierung nach endlicher Zeit terminiert und ob alle enthaltenen Aktivitäten über mindestens eine Schaltfolge aktivierbar sind. Ist dies für alle relevanten Kontrollblöcke der Fall, so gelten diese beiden Qualitätseigenschaften auch für den Gesamt-WF (vgl. hierzu Satz 4-2).

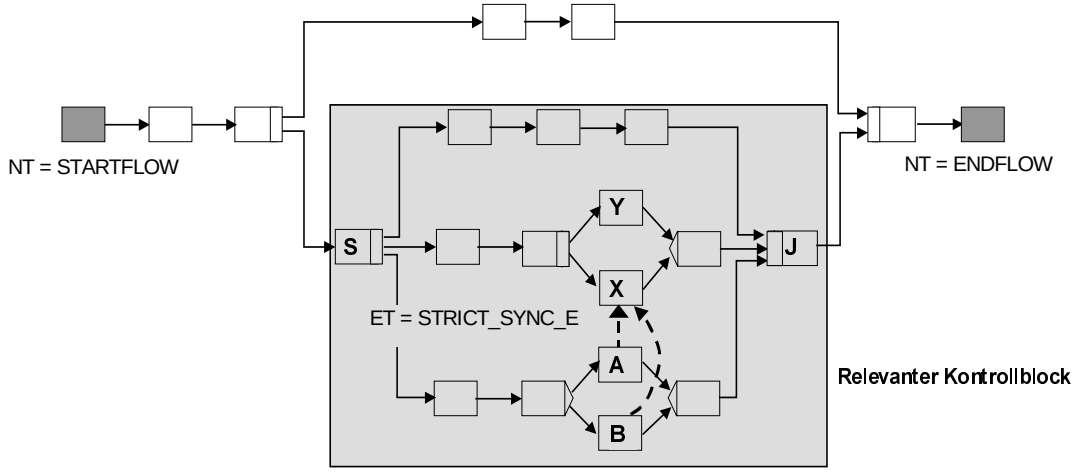


Abbildung 4-13: *Ermittlung der für Analysen des dynamischen Verhaltens relevanten Kontrollblöcke*

Ein Beispiel zeigt Abbildung 4-13. Der dargestellte KF-Graph besitzt zwei strikte Sync-Kanten, deren umschließender Kontrollblock jeweils dem durch (S, J) gebildeten Verzweigungsblock entspricht. Dementsprechend besitzt der KF-Graph genau einen für die Analyse des dynamischen Verhaltens relevanten Kontrollblock (→ grau dargestellter Teilgraph). Wir können uns somit die Blockstrukturierung auch für Analysen des dynamischen WF-Verhaltens zunutze machen.

Die Ermittlung der für die Analyse des dynamischen Verhaltens relevanten Kontrollblöcke basiert auf einem einfachen Verfahren, das auf die in Abschnitt 3.5 b) vorgestellte Funktion *BranchNodes* (→ Bestimmung des kleinsten Verzweigungsblockes, der zwei gegebene Knoten enthält) zurückgreift:

Bestimmung relevanter Blöcke für Überprüfung des dynamischen Verhaltens

CFS bezeichne einen korrekten KF-Graphen (N, E, ...), dessen strikte Sync-Kanten durch die Menge E_{strict}:= {e ∈ E | ET^e = `STRICT_SYNC_E`} gegeben sind. Die Menge relevanter Kontrollblöcke (→ RelevantBlocks) kann dann nach dem folgenden Verfahren ermittelt werden:

```
RelevantBlocks:= ∅
forall e ∈ E_strict do
        B_new:= BranchNodes(CFS, Id_source^e, Id_dest^e);      // Bestimmung des umschließenden Blocks
        addBlock:= True;
        forall B ∈ RelevantBlocks do            // Ausschluß redundanter Blöcke
                if B ⊃ B_new then                       //  „⊃“ bedeute, daß der Kontrollblock B den Block
                        addBlock:= False; break;        //  B_new vollständig enthält.
                elseif  B ⊆ B_new then
                        RelevantBlocks:= RelevantBlocks – {B}
                endif
        done
        if addBlock = True then
                RelevantBlocks:= RelevantBlocks ∪ {B_new}
        endif
done
```

Bei der Formulierung dieses Algorithmus wurde ausgenutzt, daß es keine überlappenden Verzweigungskontrollblöcke geben kann. – Wie das Beispiel aus Abbildung 4-13 zeigt, können die so ermittelten Verzweigungsblöcke ihrerseits noch Kontrollblöcke (oder Knoten) enthalten, deren Vorhandensein keinen Einfluß auf die erwähnten Dynamikeigenschaften (Terminierung des WF, Aktivierbarkeit von WF-Aktivitäten) ausübt. Dabei handelt es sich – vereinfacht ausgedrückt – um Kontrollblöcke ohne einmündende und ausgehende Sync-Kanten (anderer Teilzweige). Beispielsweise wird der obere Teilzweig des von (`S`, `J`) gebildeten Parallelverzweigungsblockes nach der Aktivierung von `S` in jedem Fall durchlaufen, unabhängig von der Ausführung der anderen Teilzweige (vgl. Satz 4-2). Dasselbe gilt für die Startknoten der beiden unteren Teilzweige. Um die Komplexität der Analysen weiter zu reduzieren, werden diese für qualitative Aussagen nicht relevanten Blöcke „eliminiert“. Für den in Abbildung 4-13 grau hervorgehobenen Kontrollblock ergibt sich nach der Durchführung entsprechender Vereinfachungen der Kontrollblock aus Abbildung 4-14 b). Durch die Analyse der für diesen Block erreichbaren Markierungen kann gezeigt werden, daß er nach seiner Aktivierung stets terminiert. Folglich trifft dies auch für den in Abbildung 4-13 dargestellten KF-Graphen zu. Des weiteren ergibt die Erreichbarkeitsanalyse, daß der Knoten `X` niemals aktivierbar ist. Der Grund hierfür ist, daß die Quellaktivitäten der in `X` einmündenden strikten Sync-Kanten (Knoten `A` und `B`) niemals gemeinsam ausgeführt werden. Auch diese Aussage ist auf den KF-Graphen aus Abbildung 4-13 übertragbar.

Im allgemeinen können die für Analysen des dynamischen Verhaltens relevanten Kontrollblöcke, bei denen es sich um parallele Verzweigungsblöcke handelt, nach folgenden Regeln vereinfacht werden:

- Teilzweige ohne einmündende oder ausgehende Sync-Kanten anderer Teilzweige können vollständig entfernt werden
- Teilzweige mit einmündenden oder ausgehenden Sync-Kanten anderer Teilzweige können ggf. noch vereinfacht werden, indem enthaltende Kontrollblöcke und Aktivitätenknoten entfernt werden, wenn sie keine einmündende oder ausgehende Sync-Kanten anderer Teilzweige besitzen.

Im Beispiel aus Abbildung 4-14 wurden beide Regeln angewendet. Im allgemeinen läßt sich die Komplexität der Erreichbarkeitsanalysen durch die Vorabbestimmung relevanter Kontrollblöcke und deren Vereinfachung erheblich verringern. Das gilt in besonderem Maße, wenn im KF-Graphen viele Parallelpfade auftreten, die Anzahl strikter Sync-Kanten dagegen vergleichsweise gering ist. Bei der Ermittlung der zu analysierenden Graphen erweisen sich das gewählte Blockkonzept und die Unterscheidbarkeit von Kantentypen als nützlich. Durch den vorgestellten Ansatz sind qualitative Aussagen zum dynamischen Verhalten eines WF-Graphen möglich, ohne daß seine Erreichbarkeitsmenge vollständig bestimmt werden muß. Wichtig ist der Hinweis, daß entsprechende Analysen erst in Verbindung mit

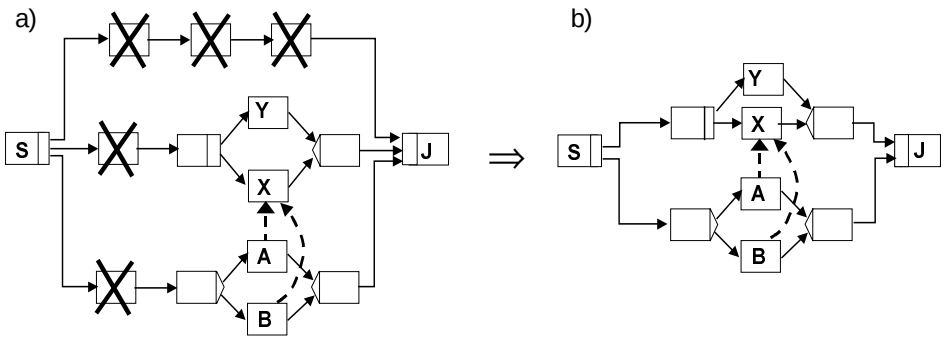


Abbildung 4-14: *Vereinfachung durch Weglassen nicht-relevanter Zweige und Knoten*

strikten Sync-Kanten anfallen. In der Mehrzahl der Fälle können die geforderten Dynamikeigenschaften bereits durch die Art der Konstruktion der KF-Graphen erzielt werden (vgl. Satz 4-2).

4.6.4.4 Weitere Optimierungen

Nicht immer ist eine Erreichbarkeitsanalyse erforderlich, um für KF-Graphen mit strikten Sync-Kanten die geforderten Dynamikeigenschaften nachzuweisen. Insbesondere bleiben die Aussagen von Satz 4-2 auch dann gültig, wenn – zusätzlich zu normalen Kontrollkanten und einfachen Sync-Kanten – strikte Sync-Kanten verwendet werden, die dem folgenden Kriterium genügen:

> Die Quellaktivität Id_{source}^{e} der strikten Sync-Kante e gehört keinem Teilzweig einer Verzweigung mit OR-Join-Knoten (bedingte Verzweigung oder Parallelverzweigung mit finaler Auswahl) an, der nicht auch den Zielknoten Id_{dest}^{e} von e enthält.

Die Verwendung einer strikten Sync-Kante ist unkritisch, falls ihre Quell- und Zielaktivität entweder gemeinsam oder gar nicht zur Ausführung kommen. Der Nachweis der gewünschten Dynamikeigenschaften kann ähnlich wie bei der Beweisführung zu Satz 4-2 erfolgen. Das vorgestellte Kriterium ist durch einfache Graphanalysen effizient überprüfbar, d.h. aufwendige Erreichbarkeitsanalysen können vermieden werden. Der genaue Algorithmus (→ *CheckSyncRelation*) wird in Anhang C vorgestellt.

4.6.5 Wohlstrukturierte, zyklische KF-Graphen

Man kann zeigen, daß die geforderten Dynamikeigenschaften auch für KF-Graphen mit Schleifen zutreffen. Grundlegend hierfür sind die Regel KF-7 zur Verwendung von Schleifenblöcken und die in Abschnitt 4.6.2 getroffene Fairneßannahme FA2, nach der jede Schleife nach endlich vielen Durchläufen terminiert (vorausgesetzt ihr Schleifenkörper erfüllt die geforderten Dynamikeigenschaften). Demzufolge terminiert eine Schleife, wenn der Schleifenkörper nach seiner Aktivierung stets eine Endmarkierung erreicht. Um qualitative Aussagen zum dynamischen Verhalten eines WF treffen zu können, können Schleifenkanten bei der Durchführung von Erreichbarkeitsanalysen deshalb ignoriert werden.

4.7 Zusammenfassung

Für KF-Graphen können die gewünschten Dynamikeigenschaften größtenteils durch die Art ihrer Konstruktion, d.h. durch die in Kapitel 3 eingeführten Strukturierungsregeln, gewährleistet werden. Diese positive Eigenschaft werden wir uns auch in Verbindung mit strukturellen Änderungen eines WF während seiner Ausführung zunutze machen. Grundlage für die Kontrollflußsteuerung bilden die in diesem Kapitel vorgestellten Ausführungs- und Markierungsregeln. Aus Kontrollflußsicht wird eine WF-Instanz durch eine Menge von geschachtelten Ausführungsgraphen (bei hierarchischen WF) repräsentiert, wobei jeder Ausführungsgraph Knoten- und Kantenmarkierungen besitzt, die Auskunft zum aktuellen Status der WF- bzw. Sub-WF-Instanz geben. Diese Kontrolldaten werden im ADEPT-WfMS persistent gespeichert, zusammen mit den später noch eingeführten Ausführungs- und Änderungshistorien. Letztere werden u.a. für das kontrollierte Zurücksetzen des WF benötigt. – Im nun folgenden Kapitel stellen wir vergleichbare Überlegungen für die Modellierung und Steuerung von Datenflüssen an. Auch hier ist die Definition geeigneter Korrektheitsbedingungen wieder essentiell.

5 Datenflußmodellierung und -steuerung

Wie in Kapitel 2 erörtert, ist Wartungsfreundlichkeit in Bezug auf WF-Vorlagen- und WF-Instanzänderungen in heutigen WfMS noch nicht wirklich realisiert. Sie wäre in vollem Umfang nur dann gegeben, wenn alle Aktivitätenprogramme so gekapselt und autonom wären, daß sie in beliebiger Reihenfolge ausführbar sind. Ein völlige Unabhängigkeit dieser Art ist bei vorgangsorientierten Anwendungen aber schon per Definition nicht gegeben.

Ein wichtiger Aspekt bei der Realisierung WF-basierter Anwendungssysteme ist deshalb die konsequente und vollständige Modellierung des Datenflusses zwischen Aktivitäten. Vollständig bedeutet, daß auch solche Datenabhängigkeiten explizit gemacht werden müssen, die infolge des gemeinsamen Lesens bzw. Schreibens externer Anwendungsdaten (z.B. Dokumenten) bestehen. Aufgabe für den Modellierer ist es, die Ein- und Ausgabeparameter von WF-Aktivitäten geeignet miteinander zu verknüpfen. Eine wichtige Forderung dabei ist, daß obligate Eingaben einer Aktivität bei ihrem Start vollständig versorgt werden können, unabhängig von zuvor durchlaufenen Ausführungspfaden. Umgekehrt müssen nach Beendigung der Aktivität ihre Ausgaben persistent in den WF-Datenkontext übernommen werden, um anschließend für Nachfolgeschritte verfügbar zu sein.

Heutige WfMS bieten keine oder nur eingeschränkte Möglichkeiten zur Validierung der Parameterversorgung bzw. -übernahme von Aktivitäten(programmen). Einige dieser Systeme erlauben es, die modellierten Datenflüsse zu animieren [Roll96] oder fehlende Daten beim Start von Aktivitäten vom Benutzer zu erfragen. Eine weiterreichende methodische Unterstützung bei der Modellierung oder Analyse von Datenflüssen gibt es dagegen nicht. Statt dessen bleibt es dem Modellierer überlassen, dafür zu sorgen, daß zur Laufzeit keine Überraschungen auftreten. Diese Sichtweise ist für einfache, überschaubare WF-Anwendungen ausreichend, führt aber mit zunehmender Komplexität der abzubildenden Kontroll- und Datenflüsse zu erheblichen Schwierigkeiten. Dies gilt in besonderem Maße, wenn strukturelle Änderungen an sich in Ausführung befindlichen Workflows durch Endanwender vorgenommen werden sollen. Spätestens dann ist die explizite Beschreibung von Datenflüssen, unter Beachtung gewisser Korrektheitsbedingungen, eine notwendige Voraussetzung für einen fehlerfreien Ablauf zur Laufzeit. Darüber hinaus schafft sie die Grundlage dafür, bei beabsichtigten WF-Modifikationen erkennen zu können, welche Anpassungen des Datenflusses und ggf. der Anwendungskomponenten vorgenommen werden müssen, um anschließend wieder einen fehlerfreien Ablauf zu erhalten.

ADEPT unterstützt die explizite Modellierung von Datenflüssen zwischen WF-Aktivitäten. Entsprechende Beschreibungen sind vollständig in die WF-Schemata integriert, wodurch der Modellierer in zweierlei Hinsicht profitiert. Zum einen kann durch systemseitige Analyse und Validation sichergestellt werden, daß der modellierte Datenfluß korrekt ist. So darf es zur Ausführungszeit nicht zum Aufruf von Aktivitätenprogrammen mit fehlenden oder unvollständigen obligaten Eingabedaten (bzw. -parametern) kommen. Ebensowenig dürfen durch parallele Schreibzugriffe nebenläufiger Aktivitäteninstanzen auf dem WF-Datenkontext Datenverluste durch Überschreiben (*Lost Updates*) resultieren. Zum anderen kann der Modellierer bei der Bildung und Abänderung von Datenflußschemata methodisch unterstützt werden, z.B. durch semi-automatisches Verknüpfen von Aktivitätenparametern [Blas96]. In dieser Arbeit ist vor allem der erstgenannte Aspekt von Interesse, d.h. die Frage, unter welchen Voraussetzungen ein gegebenes Datenflußschema korrekt ist. Dies erfordert Verfahren zur Aufrechterhaltung von Korrektheitseigenschaften bei der Bildung und Änderung des WF-Modells bzw. seiner Instanzen. Wir vereinfachen im folgenden die Sicht des Modellierers, indem wir nur Aktivitätenparameter mit atomaren Wertebereichen betrachten. Erweiterungen des Metamodells, z.B. im Hinblick auf die Unterstützung komplex strukturierter Parameterobjekte, finden sich in [Blas96].

In Abschnitt 5.1 beschreiben wir den grundlegenden Ansatz zur Modellierung von Datenflüssen in ADEPT. Die formale Repräsentation von Datenflußschemata ist Gegenstand von Abschnitt 5.2. Sie bildet die Grundlage für die nachfolgend vorgestellten Regeln zur korrekten Strukturierung von Datenflußschemata. Effiziente Verfahren zu ihrer Sicherstellung bzw. Überprüfung werden in Abschnitt 5.4 behandelt. In Abschnitt 5.5 beschreiben wir Aspekte der Datenflußsteuerung und des Datenkontextmanagements im Zusammenhang mit der Ausführung von WF-Modellen zur Laufzeit.

5.1 Modellierung von Datenflüssen in ADEPT

5.1.1 Grundlegender Ansatz

Der Datenaustausch zwischen Aktivitätenknoten eines KF-Graphen (d.h. Aktivitäten derselben Hierarchiestufe; sog. *horizontaler Datenfluß*) basiert in ADEPT auf globalen Prozeßvariablen, den sog. *Datenelementen*. Die Gesamtmenge D der Datenelemente eines WF-Modells bildet das Schema des WF-Datenkontexts. Jedes Datenelement $d \in D$ besitzt einen eindeutigen Bezeichner, einen atomaren Wertebereich dom^d und ggf. einen Default-Wert $default^d$. Der Datenfluß zwischen WF-Aktivitäten wird modelliert, indem ihre Ein- und Ausgabeparameter über Datenflußkanten (kurz: Datenkanten) mit Datenelementen aus D verknüpft werden. Soll die Ausgabe einer Aktivität A den Wert eines Eingabeparameters einer im Kontrollfluß nachfolgenden Aktivität B versorgen, so müssen die beiden Parameter über Datenkanten an dasselbe Datenelement gebunden werden. Nach welchen Regeln Verknüpfungen zwischen Parametern und Datenelementen vorzunehmen sind, um zu korrekten und vollständigen Datenflußschemata zu gelangen, wird später gezeigt. Eingabeparameter des WF selbst werden logisch als Ausgabeparameter seiner Startaktivität und seine Ausgabeparameter als Eingabeparameter seiner Endaktivität behandelt. Dem Start- und Endknoten des WF werden entsprechende Schnittstellenbeschreibungen in Form von Aktivitätenvorlagen zugeordnet. Durch diese Konvention kann die Behandlung der Ein-/Ausgaben des WF aus Anwendersicht nach denselben Prinzipien erfolgen wie im Fall „normaler“ Aktivitäten. Des weiteren vereinfacht sich dadurch die Formulierung von Korrektheitsbedingungen für Datenflußschemata.

Aufgabe der Datenflußmodellierung ist es zum einen, das Schema für den WF-Datenkontext (d.h. die Datenelemente und ihre Eigenschaften) zu definieren, zum anderen muß festgelegt werden, durch welche Datenelemente die Eingabeparameter von WF-Aktivitäten versorgt werden und in welche Datenelemente ihre Ausgaben fließen sollen. Dabei *müssen* für obligate Eingabeparameter einer Aktivität bei ihrem Start stets Werte vorliegen, unabhängig von zuvor durchlaufenen Ausführungspfaden, Werte für optionale Eingabeparameter *können* vorhanden sein. Wie in Abschnitt 3.2.2.2 erörtert, erfolgt die Festlegung, ob ein Aktivitätenparameter obligat oder optional ist, bei der Vorlagendefinition. Diese Parametereigenschaft kann bei der Bildung von WF-Modellen, bei denen die entsprechende Aktivitätenvorlage verwendet werden soll, noch abgeändert werden. Die Anknüpfung eines Eingabeparameters an das Datenkontext-Schema erfolgt durch Definition einer Lese-Datenkante, deren Quelle das versorgende Datenelement und deren Senke der Parameter ist. Wichtig ist, daß ein Datenelement erst nach mindestens einmaligem Beschreiben gültige Werte liefert, die als Eingaben von Aktivitäteninstanzen gelesen werden dürfen. (Eine Ausnahme bilden Lesezugriffe auf Datenelemente mit Default-Wert.) Umgekehrt werden nach erfolgreicher Beendigung einer Aktivitäteninstanz Ausgabedaten erzeugt, auf die nachfolgende Arbeitsschritte zugreifen können sollen. Sie werden durch die Ausgabeparameter der Aktivität definiert. Einem Ausgabeparameter kann vom Modellierer über eine Schreib-Datenkante eine Senke in Form eines Datenelements zugeordnet werden, das die entsprechenden Ausgabedaten aufnimmt. Auch in Bezug auf Ausgabeparameter wird wieder zwischen optionalen und obligaten Ausgaben unterschieden. Die Gesamtheit aller einem KF-Graphen zugeordneten Datenelemente und Datenkanten bilden sein *Datenflußschema* bzw. *-modell* (kurz: DF-Schema bzw. DF-Modell). Bevor wir hierzu eine formale Definition angeben, präsentieren wir ein einfaches Beispiel.

Beispiel 5-1 (DF-Schema):

Das in Abbildung 5-1 dargestellte DF-Schema umfaßt insgesamt 8 Datenkanten (5 Schreib- und 3 Lesekanten), die jeweils einen Aktivitätenparameter mit einem Datenelement des Datenkontextschemas verknüpfen. Der Ausgabeparameter par_1 des Startknotens entspricht dabei dem Eingabeparameter des WF. Dementsprechend liegt für das Datenelement d_1 (im Falle eines obligaten Parameters) bei Start der WF-Instanz ein entsprechender Wert vor, so daß alle WF-Aktivitäten lesend auf dieses Datenelement zugreifen können. Im Beispiel erfolgt ein solcher Zugriff durch die Aktivität B. Umgekehrt korrespondiert der Eingabeparameter par_8 des Endknotens zum Ausgabeparameter des WF. Sein Wert kann bei Beendigung der WF-Instanz an den Aufrufkontext des WF (z.B. den Datenkontext einer übergeordneten WF-Instanz) übergeben werden.

Wie aus dem Beispiel 5-1 ersichtlich wird, kann ein Datenelement mehrere einmündende bzw. ausgehende Datenkanten besitzen – im Beispiel trifft dies auf d_1 und d_3 zu. Im allgemeinen bedeutet dies nicht, daß es während der WF-Ausführung auch zu mehreren Schreib- bzw. Lesezugriffen auf die be-

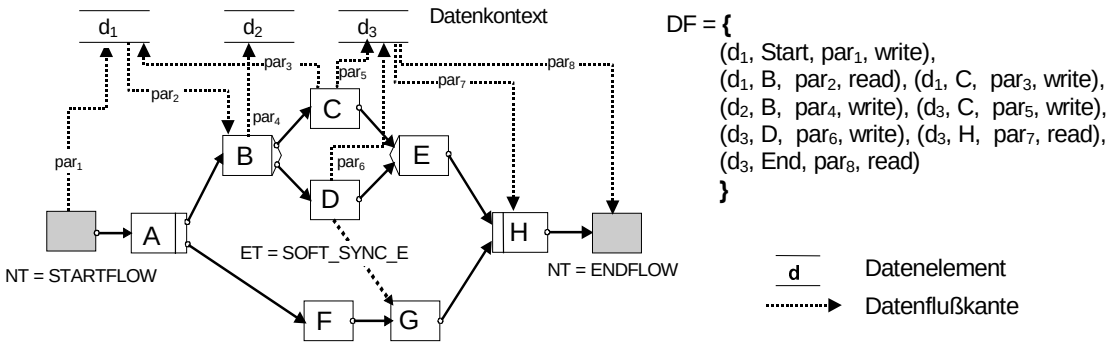


Abbildung 5-1: *Modellierung von Datenflüssen in ADEPT*

treffenden Datenelemente kommt. Beispielsweise besitzt d_3 zwei einmündende Schreibkanten, deren Quellaktivitäten `C` und `D` jedoch niemals gemeinsam zur Ausführung kommen. Mehrfach-Schreibzugriffe auf ein Datenelement können die Folge von Schreiboperationen verschiedener, im Kontrollfluß aufeinanderfolgender Aktivitäten (im Beispiel beschreiben die Aktivitäten `Start` und `C` beide das Datenelement d_1) oder derselben Aktivität innerhalb verschiedener Iterationen einer Schleifenausführung sein. Dabei erzeugt jeder Schreibzugriff auf ein Datenelement immer eine zusätzliche Version für ein entsprechendes Datenobjekt, d.h. alte Werte werden niemals überschrieben. Dies ist notwendig, um das kontextabhängige Lesen von Daten und das korrekte Zurücksetzen des WF im Fehlerfall unterstützen zu können. Ein Eingabeparameter liest normalerweise immer das aktuellste, im Ausführungskontext gültige Datum (siehe Abschnitt 5.5).

In Bezug auf *hierarchische Workflows* (vgl. Kapitel 3) gelten die folgenden Vereinbarungen: Ein WF-Modell koordiniert immer die Aktivitäten exakt einer Hierarchiestufe. Dem entsprechend ist ihm, unabhängig davon ob es einen Toplevel- oder einen Sub-WF beschreibt, genau ein KF- und ein DF-Schema zugeordnet. Der Datenfluß zwischen einem übergeordnetem Workflow W_{super} und einem untergeordneten Sub-Workflow W_{sub} (*vertikaler Datenfluß*) wird festgelegt, indem die Ein- und Ausgabeparameter der komplexen Aktivität W_{sub} an den Datenkontext des übergeordneten Workflows W_{super} über entsprechende Datenflußkanten geknüpft werden. Da die Eingaben des Sub-WF logisch als Ausgaben seiner Startaktivität und seine Ausgaben logisch als Eingaben seiner Endaktivität behandelt werden, können sie unmittelbar an den Datenkontext des Sub-WF gebunden werden. Wie bereits erwähnt, bestehen vom Prinzip her Ähnlichkeiten zu den Konzepten blockorientierter Programmiersprachen. Insbesondere führt die Verwendung von Prozeß- bzw. Blockaktivitäten zu Datenelementen (Prozeßvariablen) mit unterschiedlichem Scope, d.h. unterschiedlicher Sichtbarkeit. – Aufgrund dieser Festlegungen müssen vertikale Datenflüsse in der Folge nicht gesondert behandelt werden.

5.1.2 Erweiterung um Vorschalt- und Nachschaltdienste

Spätestens wenn Aktivitätenvorlagen durch autonome, heterogene Anwendungskomponenten implementiert werden, ist eine klassische Top-Down-Vorgehensweise bei der WF-Modellierung nicht mehr möglich. In diesem Fall stellt die Verknüpfung von Aktivitätenparametern eine nicht-triviale Aufgabe dar [Blas96]. Unterschiede in der Repräsentation von Parameterdaten verschiedener Aktivitäten müssen ebenso behandelt werden wie der Umstand, daß Eingabedaten einer Aktivität nicht immer vollständig durch Ausgaben vorangehender Aktivitäten bzw. ihrer Anwendungskomponenten versorgbar sind. Der letzte Fall ist im Zusammenhang mit Ad-hoc-Abänderungen des WF besonders relevant. So kann das Löschen von Aktivitäten und ihren Schreibkanten zu fehlenden Eingabedaten nachfolgender Schritte führen. Für diese Fälle muß ein WF-Ausführungsmodell einfache Mechanismen vorsehen, um die notwendigen Anpassungen bewerkstelligen zu können.

In ADEPT kann der Ausführung einer WF-Aktivität eine Reihe von Aktionen (sog. Hilfsdienste) vor- oder nachgeschaltet werden. Beispiele für solche Dienste sind die dynamische Generierung von Bildschirmmasken zur interaktiven Festlegung fehlender Parameterdaten oder die Konvertierung von Parameterwerten zwischen verschiedenen Datenformaten. Entsprechende Aktionen können auf der Basis eingebauter Systemfunktionen (z.B. dem ADEPT-Formularprozessor) mit einfachen Mitteln verwirklicht werden. Logisch gesehen hat jeder Hilfsdienst eine Aufrufschnittstelle, besitzt also seine eigenen

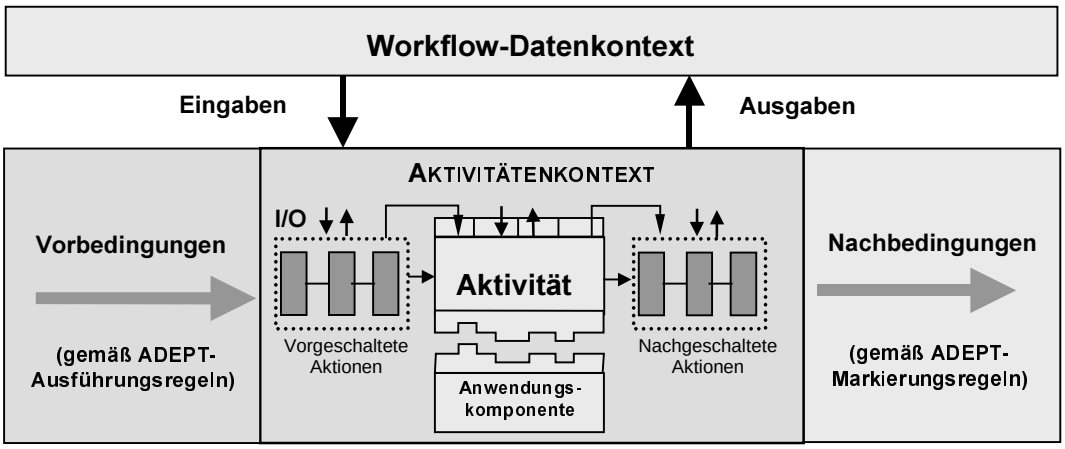


Abbildung 5-2: *Vorgeschaltete / nachgeschaltete Aktionen einer Aktivitätenbearbeitung*

Ein- und Ausgabeparameter. Diese können, ebenso wie „normale“ Aktivitätenparameter, über Datenkanten an den WF-Datenkontext angeknüpft werden. Dementsprechend können zwischen vor- bzw. nachgeschalteten Hilfsdiensten und der Aktivität selbst Daten fließen (siehe unten). Ebenso sind die von einem Hilfsdienst geschriebenen Kontextdaten für nachfolgende Aktivitäten (und deren Hilfsdienste) lesbar. Die entsprechenden Datenzugriffe müssen deshalb bei der Definition von DF-Schemata sowie der Überprüfung ihrer Korrektheit berücksichtigt werden. Prinzipiell können auch zwischen den einer Aktivität vor- bzw. nachgeschalteten Hilfsdiensten Daten ausgetauscht werden. Für die nachfolgenden Betrachtungen klammern wir diese Möglichkeit jedoch aus, um uns bei der Formulierung von Bedingungen hinsichtlich der Korrektheit von Datenflüssen auf die für diese Arbeit wesentlichen Aspekte konzentrieren zu können.

Die Ausführung der mit einer Aktivität assoziierten Hilfsdienste ist eng mit der Aktivitätenbearbeitung verknüpft. Sie erfolgt entweder direkt beim Start der Aktivität (d.h. nachdem ein berechtigter Akteur den entsprechenden Arbeitseintrag aus seiner Arbeitsliste selektiert hat) oder unmittelbar nach ihrer Beendigung. Dem entsprechend resultieren keine zusätzlichen Arbeitslisteneinträge. Besitzt eine Aktivität mehrere vorgeschaltete (nachgeschaltete) Hilfsdienste, so muß deren Ausführungsreihenfolge über eine eindeutige Positionsnummer festliegen. Erst wenn alle vorgeschalteten Hilfdienste einer Aktivitäteninstanz erfolgreich abgeschlossen sind, kann die eigentliche Aktion (z.B. Aufruf einer Anwendungskomponente) gestartet werden. Scheitert die Bearbeitung der Aktivitäteninstanz, z.B. infolge eines semantischen Fehlers, so können die Effekte zuvor durchgeführter Hilfsdienste (z.B. Schreiboperationen auf dem WF-Datenkontext) wieder zurückgesetzt werden, wenn dies gewünscht wird (siehe Kapitel 6). Die Maßnahmen zur Durchführung vor- und nachgeschalteter Hilfsdienste bildet zusammen mit der Aktivitätenausführung die logische „Hülle“ der Aktivität (siehe Abbildung 5-2).

Mit dem vorgestellten Konzept können die Werte der Ein- und Ausgabeparameter einer Aktivität auch noch beim Aktivitätenstart bzw. -ende festgelegt werden[16]. Wählt ein Anwender eine entsprechende Aufgabe aus seiner Arbeitsliste aus, können vor Ausführung der eigentlichen Anwendungskomponente fehlende, d.h. aus dem Datenkontext nicht versorgbare Eingabedaten über interaktive Dienste (z.B. generierte Bildschirmmasken) nachgefordert werden. Diese Möglichkeit der Parameterversorgung ist, wie wir später zeigen werden, in Verbindung mit dynamischen WF-Modifikationen relevant. Ebenso wichtig ist, daß Hilfsdienste auf Ebene einzelner Workflow- bzw. Aktivitäteninstanzen dynamisch hinzugenommen bzw. entfernt werden können. Das vorgestellte Konzept stellt somit ein wichtiges Instrument zur flexiblen Anpassung von Datenflußschemata dar. Entsprechende Änderungen werden z.B. erforderlich, wenn nach dem Überspringen bzw. Löschen von Aktivitäten (inkl. ihrer Datenkanten) Eingabedaten für nachfolgende Arbeitsschritte unversorgt bleiben. Die Verwendung von Vor-

[16] Bei vorgeschalteten Diensten handelt es sich um logische Dienste, deren Parameterschnittstellen dynamisch gebildet werden. Im ADEPT-WfMS etwa werden beim Fehlen obligater, nachforderbarer Aktivitäteneingaben die betreffenden Parameter markiert. Auf Basis dieser Markierungen und den vorliegenden Parameterbeschreibungen werden die fehlenden Daten dann beim Start der Aktivität über dynamisch generierte Bildschirmmasken vom Benutzer nachgefordert.

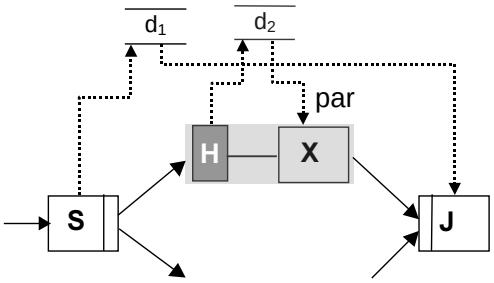


Abbildung 5-3: *Vorschalten eines Hilfsdienstes*

Beispiel 5-2 (Vorschalten eines Hilfsdienstes):

Ein Beispiel für ein DF-Schema mit Schreibzugriff durch einen Hilfsdienst zeigt Abbildung 5-3. Hier wird der Eingabeparameter `par` *der Aktivität* `X` *durch einen Ausgabeparameter des vorgeschalteten Hilfsdienstes* `H` *versorgt. (H schreibt das Datenelement* d_2*, das von X gelesen wird.)* `H` *wird dementsprechend bei Start von* `X` *ausgeführt und beendet bevor die mit* `X` *assoziierte Anwendungskomponente gerufen wird.*

schalt- und Nachschaltdiensten ist allerdings nicht immer möglich. So darf einer automatischen Aktivität nicht ohne weiteres ein interaktiver Hilfsdienst vor- bzw. nachgeschaltet werden. Des weiteren sind nur diejenigen Ein- bzw. Ausgabeparameter nachversorgbar, für die auf Vorlagenebene entsprechende Festlegungen getroffen wurden (vgl. Abschnitt 3.2.2.2).

Aus Datenflußsicht werden einzelne Aktivitäten und die ihnen vor- bzw. nachgeschalteten Hilfsdiensten wie eine Sequenz behandelt. Allerdings sind Hilfsdienste, im Vergleich zu normalen Aktivitäten, „leichtgewichtige“ Aktionen, deren Ausführung auf eingebauten Funktionen basiert und in unmittelbarem Kontext einer Aktivitätenbearbeitung stattfindet. Dadurch lassen sich Aktionen, wie die interaktive Versorgung von Eingaben, benutzergerecht verwirklichen.

5.2 Formale Repräsentation von DF-Schemata

Um Regeln für die Strukturierung von DF-Schemata präzise formulieren zu können, ist eine formale Darstellung des Datenflusses unerläßlich. Sie ist Gegenstand dieses Abschnitts. Korrektheitsbedingungen und Verfahren zu ihrer Sicherstellung werden in den beiden folgenden Abschnitten vorgestellt.

Im folgenden sei durch $CFS = (N, E, D, NT, ..., Template)$ ein allgemeiner KF-Graph gegeben. Wie in Kapitel 3 beschrieben, ist jedem Knoten $n \in N$ – ausgenommen „leere“ Knoten bzw. Nullaktivitäten – eine Vorlagenbeschreibung $Template^n \in Templates$ zugeordnet. Sie legt u.a. die Ein- und Ausgabeparameter der Aktivität sowie deren Eigenschaften fest. Zur sprachlichen Vereinfachung bezeichnen wir diese Parameter im folgenden als Ein-/Ausgabeparameter der Aktivität n und fassen sie entsprechend in den Mengen $InParams^n$ und $OutParams^n$ zusammen. Des weiteren sei für einen Knoten $n \in N$ die Menge der ihm zugeordneten Hilfsdienste durch $S^n := PreS^n \cup PostS^n$ gegeben[17]. $PreS^n$ ($PostS^n$) bezeichne dabei die Menge der dem Aktivitätenknoten n vorgeschalteten (nachgeschalteten) Hilfsdienste. Die Ein- und Ausgabeparameter eines Hilfsdienstes $s \in S^n$ seien, wie im Fall von Aktivitäten, durch die zwei Mengen $InParams^s$ und $OutParams^s$ gegeben.

Definition 5-1 (DF-Schema, DF-Modell):

Sei durch $CFS = (N, E, D, ...)$ ein allgemeiner KF-Graph gegeben (D bezeichnet die Menge aller definierten Datenelemente und bildet das Datenkontext-Schema).

Eine *Datenflußkante dfc* (engl. data flow connector) beschreibt die Verknüpfung eines Aktivitäten-/Dienstparameters mit einem Datenelement aus D und wird durch ein 4-Tupel

$$dfc = (d, x, par, access_mode)$$

beschrieben. Für die einzelnen Komponenten gilt

$$d \in D,\ x \in N \cup S \text{ mit } S := \bigcup_{n \in N} S^n,\ par \in InParams^x \cup OutParams^x,\ access_mode \in \{\texttt{read}, \texttt{write}\}$$

wobei die folgenden Einschränkungen bestehen:

[17] Die genaue Ausführungsreihenfolge der Hilfsdienste ist wegen der eingangs gemachten Einschränkung (→ kein Datenfluß zwischen vorgeschalteten / nachgeschalteten Hilfsdiensten einer Aktivität) hier nicht relevant.

(1) $dfc = (d, x, par, \texttt{read}), \Rightarrow par \in InParams^x \wedge pDom^{par} \supseteq dom^d$

(2) $dfc = (d, x, par, \texttt{write}) \Rightarrow par \in OutParams^x \wedge pDom^{par} \subseteq dom^d$

Dem entsprechend wird zwischen Lese- und Schreibkanten differenziert. *Lesekanten* verknüpfen die Eingabeparameter und *Schreibkanten* die Ausgabeparameter einer Aktivität bzw. der ihnen vor- und nachgeschalteten Aktionen (Hilfsdiensten) mit Datenelementen aus D. Des weiteren nennen wir $dfc = (d, x, par, access_mode)$ Datenflußkante mit obligatem (optionalen) Zugriff auf d (kurz: *obligate* (*optionale*) *Datenflußkante*), falls der referenzierte Parameter par als obligat (optional) gekennzeichnet ist, d.h. es gilt: $pDM^{par} = \texttt{MANDATORY}$ ($\texttt{OPTIONAL}$).

Die Menge DFS (= DFS^{CFS}; *Data Flow Schema*) aller auf CFS definierten Datenflußkanten heißt das dem KF-Graphen CFS zugeordnete *Datenflußschema* (kurz: *DF-Schema*) bzw. *Datenflußmodell* (kurz: *DF-Modell*). $WS = (CFS, DFS^{CFS})$ wird WF-Schema (WF-Modell) genannt.

Die intuitive Bedeutung einer Lesekante $dfc = (d, x, par, \texttt{read})$ ist, daß der Eingabeparameter par bei Start von x mit dem aktuellen Wert des Datenelements d versorgt wird. Umgekehrt wird durch eine Schreibkante $dfc = (d, x, par, \texttt{write})$ zum Ausdruck gebracht, daß der Wert des Ausgabeparameters par nach Beendigung von x in das (persistente) Datenelement d geschrieben wird. Ein Beispiel für ein Datenflußschema haben wir bereits in Abbildung 5-1 vorgestellt. An diesem Beispiel haben wir auch gezeigt, daß Eingaben des WF über Schreibkanten seiner Startaktivität und seine Ausgaben über Lesekanten seiner Endaktivität an den Datenkontext (logisch) angeknüpft sind. Wichtig für die nachfolgenden Betrachtungen sind noch die folgenden Festlegungen

Definition 5-2 (Prädikate zur Beschreibung von Lese-/Schreibzugriffen):

Sei DFS das dem KF-Graphen $CFS = (N, E, D, ...)$ zugeordnete DF-Schema. Dann definieren wir für Aktivitäten oder ihnen zugeordneten Hilfsdiensten $x \in N \cup S$ die folgenden beiden Prädikate:

$Reads^{DFS}(x, d, mode) :\Leftrightarrow \exists\, par \in InParams^x$ mit $pDM^{par} = mode \wedge (d, x, par, \texttt{read}) \in DFS$

$Writes^{DFS}(x, d, mode) :\Leftrightarrow \exists\, par \in OutParams^x$ mit $pDM^{par} = mode \wedge (d, x, par, \texttt{write}) \in DFS$

$(mode \in \{\texttt{MANDATORY}, \texttt{OPTIONAL}, \texttt{ANY}\})$

Das Prädikat $Reads^{DFS}(x, d, \texttt{MANDATORY})$ etwa ist erfüllt, wenn es in DFS eine Lesekante gibt, die das Datenelement d mit einem obligaten Eingabeparameter von x verknüpft. Wird $mode = \texttt{ANY}$ gewählt, so werden sowohl obligate als auch optionale Datenzugriffe berücksichtigt. Hierauf basierend definieren wir für Aktivitätenknoten $n \in N$ die folgenden Prädikate:

$WritesBefore^{DFS}(n, d, mode) :\Leftrightarrow \exists\, s \in PreS^n : Writes^{DFS}(s, d, mode)$

$WritesAfter^{DFS}(n, d, mode) :\Leftrightarrow \exists\, s \in PostS^n : Writes^{DFS}(s, d, mode)$

sowie

$ActivityWrites^{DFS}(n, d, mode) :\Leftrightarrow WritesBefore^{DFS}(n, d, mode) \vee WritesAfter^{DFS}(n, d, mode) \vee Writes^{DFS}(n, d, mode)$

Für $n \in N$ und $d \in D$ ist das Prädikat $WritesBefore^{DFS}(n, d, mode)$ [$WritesAfter^{DFS}(n, d, mode)$] genau dann erfüllt, wenn die Aktivität einen vorgeschalteten [nachgeschalteten] Hilfsdienst $s \in PreS^n$ [$s \in PostS^n$] besitzt, von dem aus eine obligate Schreibkante zu d führt.

Das Prädikat $ActivityWrites^{DFS}(n, d, mode)$ ist erfüllt, wenn von der Aktivität oder einem ihrer vor- bzw. nachgeschalteten Hilfsdienste eine obligate Schreibkante nach d führt. Auf dieselbe Art und Weise können Prädikate $ReadsBefore^{DFS}$, $ReadsAfter^{DFS}$ und $ActivityReads^{DFS}$ definiert werden.

5.3 Struktur von DF-Schemata

Der Datenaustausch über globale Prozeßvariablen wird auch in einigen existierenden WfMS (z.B. [Rupi97, WäRe92]) angewendet. Neu hinzu kommt im Kontext von ADEPT eine Menge von Regeln

zur Strukturierung von DF-Schemata und effiziente Verfahren zu ihrer Überprüfung. Das Ziel dabei ist, die Verwendung von Datenflußkanten bzw. die Menge zulässiger Datenverknüpfungen bei der Modellierung des DF-Schemas soweit einzugrenzen, daß es während der WF-Ausführung nicht zu Problemen kommen kann, wie Dateninkonsistenzen (z.B. infolge von Datenverlusten durch Überschreiben) oder Programmfehlern (z.B. beim Aufruf einer Anwendungskomponente mit fehlenden Eingabedaten). Solche Fälle müssen sowohl bei der Modellbildung als auch zur Laufzeit, im Zusammenhang mit der Durchführung dynamischer WF-Änderungen, ausgeschlossen werden. Dabei sollte die Verantwortung für die Vermeidung solcher Konsistenzprobleme oder Laufzeitfehler nicht auf den Endanwender übertragen werden. Statt dessen muß das WfMS in die Lage versetzt werden, darüber entscheiden zu können, ob eine beabsichtigte Änderung die Korrektheits- und Konsistenzeigenschaften des modellierten Datenflußes bewahrt oder – falls nein – welche begleitenden Änderungen (z.B. dynamisches Hinzufügen von Hilfsdiensten zur Versorgung fehlender Parameterwerte) ggf. notwendig werden. Letzteres ist essentiell, um die Anwender bei der Durchführung von Änderungen angemessen unterstützen zu können. In dieser Arbeit beschränken wir uns auf die Definition grundlegender Strukturierungsregeln. Ein weitergehender Ansatz, der auch Semantikaspekte bei der Verknüpfung von Parametern und Datenelementen berücksichtigt, wird in [Blas96] vorgestellt.

Im folgenden gehen wir davon aus, daß für die korrekte Ausführung einer WF-Aktivität alle obligaten Eingabeparameter versorgt sein müssen. Umgekehrt nehmen wir an, daß nach ihrer erfolgreichen Ausführung alle obligaten Ausgabeparameter mit Werten belegt sind. Darauf basierend betrachten wir nun wichtige Regeln zur Strukturierung von DF-Schemata.

5.3.1 Versorgung obligater Eingabeparameter

Um den Aufruf von Aktivitäten bzw. Hilfsdiensten bei fehlenden Werten für (obligate) Eingabeparameter zu vermeiden, fordern wir die Einhaltung der folgende Regel:

Strukturierungsregel DF-1 (Sichere Versorgung obligater Eingabeparameter)

Sei durch $CFS = (N, E, D, NT, ...)$ ein allgemeiner KF-Graph gegeben und durch DFS das diesem KF-Graphen zugeordnete DF-Schema.

Generell fordern wir, daß jeder obligate Eingabeparameter eines Aktivitätenknotens (bzw. eines zugeordneten Hilfsdienstes) an genau ein Datenelement aus dem WF-Datenkontext angeknüpft wird, d.h. es soll gelten:

$$\forall\, x \in N \cup S, \forall\, par \in InParams^x \text{ mit } pDM^{par} = \texttt{MANDATORY}: \exists!\, d \in D \text{ mit } (d, x, par, \texttt{read}) \in DFS$$

Für einen Aktivitätenknoten oder Hilfsdienst $x \in N \cup S$ bezeichne A^x die Menge aller zulässigen Aktionsmengen, die zur Aktivierung von x führen. Ein einzelnes Element V von A^x beschreibt dem entsprechend eine Menge von im Kontrollfluß <u>vorausgehenden</u> Aktivitätenknoten und/oder Hilfsdiensten von x, deren Ausführung zur Aktivierung von x führt[18] (siehe Beispiel 5-3).

Für $x \in N \cup S$, $d \in D$ fordern wir dann:

$$Reads^{DFS}(x, d, \texttt{MANDATORY}) \Longrightarrow \forall\, V \in A^x: \exists\, x^* \in V \text{ mit } Writes^{DFS}(x^*, d, \texttt{MANDATORY}) \qquad (*)$$

Wenn x das Datenelement d obligat liest, muß es bei der Aktivierung von x, unabhängig vom zuvor durchlaufenen Ausführungspfad, immer mindestens einen vorausgehenden Aktivitätenknoten oder Hilfsdienst x^* geben, der das Datenelement d obligat schreibt. Wir sagen dann, daß d bei der Aktivierung von x *sicher versorgt* ist.

Wir geben kurz ein Beispiel für die Anwendung dieser Regel an. Ein effizientes Verfahren zu ihrer Sicherstellung wird in Abschnitt 5.4 vorgestellt.

[18] Das bedeutet insbesondere, daß in der Menge V keine Knoten oder Hilfsdienste enthalten sein dürfen, deren Ausführung parallel zu x erfolgen kann.

Beispiel 5-3 (Sichere Versorgung obligater Eingabeparameter)

Für das Beispiel aus Abbildung 5-1 nehmen wir an, daß alle Datenflußkanten obligat sind. Für den Knoten H erhalten wir dann:

A^H = {{Start, A, B, C, E, F, G}, {Start, A, B, D, E, F, G}}

Für jede Aktionsmenge $V \in A^H$ gilt, daß das von H gelesene Datenelement d_3 von mindestens einer Aktivität aus V geschrieben wird (im Beispiel sind dies die Aktivitäten C bzw. D).

Würde man zu dem gegebenen DF-Schema eine obligate Lesekante, die d_3 mit einem obligaten Eingabeparameter der Aktivität G verknüpft, hinzufügen, so wäre Strukturierungsregel DF-1 nicht mehr erfüllt. Begründung: Es gilt A^G = {{Start, A, B, D, F}, {Start, A, B, F}} und das Element d_3 wird für V = {Start, A, B, F} von keiner Aktivität geschrieben. Folglich ist eine obligate Lesekante von G nach d_3 unzulässig, da ansonsten zur Laufzeit die G zugeordnete Anwendungskomponente möglicherweise mit fehlenden Eingaben gerufen wird.

Die durch die Regel DF-1 festgelegte Bedingung (*) ist für einen Aktivitätenknoten $n \in N$, $NT^n \neq$ STARTFLOW und für ein von n gelesenes Datenelement $d \in D$ trivialerweise erfüllt, wenn das Datenelement d über eine obligate Schreibkante mit einem Ausgabeparameter der WF-Startaktivität verknüpft ist; d.h. der Wert von d wird aus den Aufrufdaten des WF versorgt. Umgekehrt stellt die Einhaltung von Regel DF-1 sicher, daß alle obligaten Ausgabeparameter des WF – sie entsprechen logisch den obligaten Eingabeparameter der WF-Endaktivität – bei erfolgreicher WF-Terminierung vollständig versorgt sind. Des weiteren garantiert die Regel DF-1 auch die Datenversorgung von Aktivitätenknoten, deren Ausführung aus der Auswertung eines Verzweigungsprädikates besteht (vgl. Abschnitt 3.3.2.2); d.h. alle zur Auswertung des Prädikates benötigten Daten werden durch den WF-Datenkontext bereitgestellt.

Optionale Schreibzugriffe haben wir bei der Formulierung von Regel DF-1 nicht berücksichtigt, da durch sie die Versorgung von Datenelementen zur Laufzeit nicht sichergestellt ist. Umgekehrt muß für *optionale Lesezugriffe* einer Aktivität nicht geprüft werden, ob für die betreffenden Datenelemente beim Start der Aktivität Werte vorliegen. Entsprechende Überprüfungen können dennoch sinnvoll sein, um z.B. Warnhinweise für den Anwender generieren zu können. Optionale Lesekanten können auch an Datenelemente geknüpft werden, die nicht in allen möglichen Ausführungspfaden geschrieben werden oder die nur eingehende optionale Schreibkanten besitzen. Aus Gründen der besseren Lesbarkeit haben wir bei der Formulierung von Regel DF-1 *Default-Werte* für Datenelemente und Eingabeparameter nicht berücksichtigt. Besitzt ein Datenelement d einen nicht nullwertigen Default-Wert $default^d$, so muß die Versorgung von Lesezugriffen auf d nicht notwendigerweise überprüft werden. Dasselbe gilt für eine Lesekante (d, x, par, read) $\in DFS$, wenn der zugehörige Parameter par seinerseits einen nicht nullwertigen Default-Wert $pDefault^{par}$ besitzt (vgl. Abschnitt 3.2.2.2). Der Einfachheithalber werden wir diese Möglichkeiten in der Folge nicht mehr explizit erwähnen.

Schließlich muß die Regel DF-1 noch ergänzt werden, um sicherzustellen, daß nach der Beendigung eines OR-Split-Knotens ein Datenwert für die Auswahl des durchzuführenden Teilzweig vorliegt:

Ergänzung zu Strukturierungsregel DF-1

Für jeden OR-Split-Knoten $s \in N$ (mit V_{out}^s = ONE_Of_ALL) fordern wir zusätzlich, daß das ihm zugeordnete Verzweigungsdatenelement $DP^n \in D$, das den zu wählenden Teilzweig festlegt, spätestens nach der Beendigung von s geschrieben worden ist:

$$ActivityWrites^{DFS}(s, DP^s, \text{MANDATORY}) \vee \left(\forall V \in A^s: \exists x^* \in V \text{ mit } Writes^{DFS}(x^*, DP^s, \text{MANDATORY})\right)$$

Regel DF-1 ist für die Realisierung zuverlässiger und flexibler WF-Anwendungen essentiell. Sie könnte noch etwas weniger restriktiv formuliert werden. Dazu nehmen wir an, daß im KF-Graphen aus Abbildung 5-4 a) den beiden OR-Split-Knoten B und F derselbe Entscheidungsparameter zugeordnet ist und daß er (bzw. das von ihm referenzierte Datenelement) nach erstmaliger Festlegung nicht mehr geändert wird. Dann wird zur Laufzeit für die beiden Verzweigungen entweder immer der obere Teilzweig (Aktivitäten C und G) oder der untere Teilzweig (Aktivitäten D und H) durchlaufen. Aus diesem Grund sind die Lesezugriffe von G auf d_1 bzw. von H auf d_2 zur Laufzeit stets versorgt, obwohl die Regel DF-1 nicht erfüllt ist – diese Regel fordert die Versorgung des Lesezugriffs für alle

theoretisch möglichen Ausführungspfade (also z.B. auch für die Kombination D und G, die der Modellierer ausgeschlossen hat). Im allgemeinen ist es schwierig, solche vom Modellierer mehr oder weniger implizit festgelegten Abhängigkeiten zwischen Verzweigungsentscheidungen systemseitig zu erkennen, insbesondere wenn das Datenelement, auf dem die Verzweigungsentscheidung basiert, mehrfach geschrieben wird. Vom Prinzip her aber sind Erweiterungen des Basismodells denkbar (z.B. gleiche Färbung korrespondierender Teilzweige), um die skizzierten Abhängigkeiten explizit zu machen. Wir wollen darauf im folgenden verzichten, um die Diskussionen nicht unnötig zu überfrachten. Für das vorliegende Beispiel stellt die Formulierung von Regel DF-1 keinen wirklichen Nachteil dar, da für den WF-Modellierer die Möglichkeit besteht, die Abhängigkeiten explizit im Modell abzubilden. Wie man leicht sieht, kann der KF-Graph aus Abbildung 5-4 a) unter den gegebenen Annahmen, wie in Abbildung 5-4 b) dargestellt, modelliert werden.

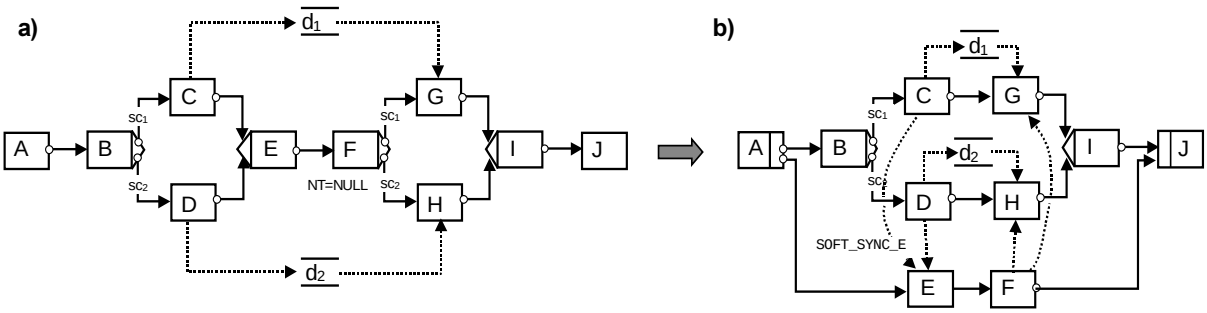


Abbildung 5-4: *Alternative Modellierung abhängiger Verzweigungsentscheidungen*

Auf Instanzebene kann die Regel DF-1 weniger rigide formuliert werden, da hier zusätzliche Statusinformationen vorliegen (siehe Kapitel 7).

5.3.2 Ausschluß paralleler Schreibzugriffe auf Datenelementen

Für parallele Verzweigungen (AND-Split/AND-Join) fordern wir der Einfachheithalber (siehe untenstehende Diskussion), daß ein bestimmtes Datenelement nicht von parallelen Knoten verschiedener Teilzweige geschrieben werden darf. Dadurch steht für nachfolgende Aktivitäten immer eindeutig fest, welcher Wert des Datenelements zur Ausführungszeit gelesen werden soll. Des weiteren ist im Fehlerfall das partielle Zurücksetzen der WF-Kontrolle (inkl. Schreibzugriffen auf dem WF-Datenkontext) innerhalb einzelner Teilzweige möglich, ohne daß davon die Ausführung der anderen Zweige betroffen ist (siehe Kapitel 6). Dieser Forderung wird durch die Strukturierungsregel DF-2 nachgekommen. Für ihre Formulierung verwenden wir die in Abschnitt 3.5.2 definierte Funktion *BranchNodes*, die für zwei Aktivitätenknoten, falls existent, den kleinsten, sie umschließenden Verzweigungskontrollblock bestimmt.

Strukturierungsregel DF-2 (Vermeidung paralleler Schreibzugriffe)

$CFS = (N, E, D, ...)$ sei ein allgemeiner KF-Graph und DFS das diesem Graphen zugeordnete Datenflußschema. Ferner seien $n_1, n_2 \in N$, $n_1 \neq n_2$ zwei Aktivitätenknoten und $d \in D$ gegeben mit

$$ActivityWrites^{DFS}(n_1, d, \texttt{ANY}) \wedge ActivityWrites^{DFS}(n_2, d, \texttt{ANY})$$

Das heißt das Datenelement d wird sowohl von n_1 (oder einem n_1 vor- bzw. nachgeschalteten Hilfsdienst) als auch von n_2 (oder einem n_2 vor- bzw. nachgeschalteten Hilfsdienst) geschrieben.

Dann dürfen die beiden Aktivitätenknoten n_1 und n_2 nicht in verschiedenen Zweigen einer parallelen Verzweigung mit AND-Join-Knoten liegen, es sei denn, ihre Ausführungsreihenfolge ist durch die Verwendung von Sync-Kanten eindeutig festgelegt. Formal:

Falls $(split, join) := BranchNodes^{CFS}(n_1, n_2) \neq (\texttt{UNDEFINED}, \texttt{UNDEFINED})$ gilt (d.h. die Knoten x_1 und x_2 liegen in unterschiedlichen Zweigen der durch $(split, join)$ gebildeten Verzweigung), so muß ebenfalls gelten:

$$(V_{in}^{join} \neq \texttt{ALL_Of_ALL}) \vee (n_1 \in succ^*(n_2) \vee n_1 \in pred^*(n_2)).$$

Im Gegensatz zu Regel DF-1 haben wir bei der Formulierung von Regel DF-2 auch optionale Schreibkanten berücksichtigt, da es durch sie potentiell zu Schreibzugriffen auf das betreffende Datenelement kommen kann.

Beispiel 5-4 (Ausschluß paralleler Schreibzugriffe auf Datenelementen)

Im Beispiel aus Abbildung 5-1 würde die Erweiterung des gegebenen DF-Schemas *DFS* um die Schreibkante (d_3, `G`, `par`, `write`) zu einer Verletzung von Strukturierungsregel DF-2 führen. Die Aktivitäten `C` und `G` könnten dann beide das Element d_3 schreiben, ohne daß ihre Ausführungsreihenfolge eindeutig festliegt. Dagegen ist die Verwendung der beiden Schreibkanten (d_3, `C`, par_5, `write`) und (d_3, `D`, par_6, `write`) korrekt, da sie in unterschiedlichen Zweigen einer bedingten Verzweigung liegen; das heißt zur Laufzeit schreibt immer nur eine der beiden Aktivitäten das Element d_3.

Bei Parallelverzweigungen mit finaler Auswahl sind für ein Datenelement nebenläufige Schreibzugriffe durch Knoten verschiedener Teilzweige dennoch erlaubt. Die Zweige arbeiten dann ggf. auf unterschiedlichen Versionen des Datenelements. Bei Beendigung der Verzweigung werden nur die Daten des Siegerzweiges übernommen, während die restlichen Teilzweige und die von ihren Aktivitäten durchgeführten Schreibzugriffe auf dem WF-Datenkontext zurückgesetzt werden (vgl. Abschnitt 5.5).

Soll dasselbe Datenelement durch Aktivitäten unterschiedlicher Zweige einer parallelen Verzweigung (mit AND-Join) geschrieben werden, muß gemäß Regel DF-2 zwischen ihnen eine eindeutige Ausführungsreihenfolge festliegen. Dies kann in ADEPT durch die Verwendung von Sync-Kanten erreicht werden. Allerdings ist eine solche „statische“ Serialisierung der Schreibzugriffe auf Datenelementen für bestimmte Anwendungen nicht immer ausreichend. Deshalb sollte ein WfMS generell Möglichkeiten für den konkurrierenden Zugriff auf Datenelemente bieten. Da hieraus gegenüber den in dieser Arbeit behandelten Themen überwiegend orthogonale Fragestellungen resultieren, verzichten wir an dieser Stelle auf entsprechende (nicht-triviale) Erweiterungen. Statt dessen skizzieren wir kurz einige wichtige Aspekte, die für die Erzielung eines höheren Grades an Parallelität relevant sind.

Um konkurrierende Schreibzugriffe auf Datenelementen angemessen behandeln zu können, sollte zwischen kommutativen und nicht kommutativen, serialisierten Zugriffen unterschieden werden. In beiden Fällen sind die Zugriffe auf das Datenelement durch die Datenverwaltungskomponente des WfMS zu *serialisieren*, d.h. das Lesen und Zurückschreiben des Wertes für das Datenelement muß durch eine Transaktion gekapselt werden [Dada96, GrRe93]. Dadurch kann sich ein Datenelement niemals gleichzeitig im Zugriff mehrerer Aktivitätenprogramme befinden, so daß Datenverluste durch Überschreiben (*Lost Updates*) nicht auftreten. Bei *kommutativen Zugriffen* (z.B. Erhöhung oder Erniedrigung eines Kontostandes oder Zählers) wird zusätzlich angenommen, daß die Reihenfolge der Zugriffsoperationen vertauschbar ist. Dem entsprechend können die Aktivitätenausführungen für den Fall des Zurücksetzens in beliebiger Reihenfolge und ggf. auch nur partiell kompensiert werden. Bei *nicht kommutativen* Zugriffen wäre die Kompensation nur in der zur Aktivitätenausführung umgekehrten Reihenfolge möglich.

Wird von einem Datenelement ein externes, d.h. ein nicht vom WfMS kontrolliertes Anwendungsobjekt (z.B. ein Dokument) referenziert, sind (auch bei der Zugrundelegung von Regel DF-2) konkurrierende Schreibzugriffe auf dieses Objekt durch die Aktivitäten paralleler Zweige möglich. Ihre Synchronisation erfolgt dann nicht durch das WfMS, sondern durch die externe Datenverwaltungskomponente (z.B. ein DBMS oder ein Dokumenten-Management-System) oder die Benutzer. Regel DF-2 stellt deshalb hinsichtlich konkurrierenden Schreibzugriffen auf referenzierte Anwendungsdaten keine gravierende Einschränkung dar. Allerdings sollte auch hier feststehen, ob die Schreibzugriffe kommutativ sind oder nicht und dem entsprechend ob die Reihenfolge eventueller Kompensationsaktionen vorgegeben ist oder nicht. Abhängig davon kann sich das Rücksetzen des WF mehr oder weniger aufwendig gestalten. Weiterführende Diskussionen zu diesem Thema finden man z.B. in [Elma93, Leym95, Leym97]. Auch im ADEPT-Projekt wird derzeit an Erweiterungen gearbeitet, die allerdings nicht Gegenstand dieser Arbeit sind.

5.3.3 Vermeidung überflüssiger Schreiboperationen

Erfolgt direkt nach dem Schreiben eines Datenelements durch eine Aktivität *X* ein weiterer Schreibzugriff durch einen Nachfolgerknoten *Y*, wird der erste Wert ungenutzt überschrieben. Dieser Fall sollte

bei der WF-Modellierung bzw. -Änderung erkannt und der Anwender darauf hingewiesen werden, muß aber nicht unbedingt ausgeschlossen werden. Während der WF-Ausführung kann das kontrollierte Überschreiben von zuvor geschriebenen Datenelementen sogar explizit gewünscht sein, etwa wenn fehlerhafte Ausgaben vorangegangener Prozeßschritte korrigiert werden sollen.

Strukturierungsregel DF-3 (Vermeidung direkt aufeinanderfolgender Schreibzugriffe)

$CFS = (N, E, D, ...)$ sei ein allgemeiner KF-Graph und DFS das diesem Graphen zugeordnete Datenflußschema. Des weiteren seien $x_1, x_2 \in N \cup S$, $x_2 \in succ^*(x_1)$ zwei im KF-Graphen aufeinanderfolgende Aktivitäten oder Hilfsdienste[19] und $d \in D$ ein Datenelement mit $Writes^{DFS}(x, d, \texttt{ANY})$ für $x = x_1, x_2$. Dann soll gelten:

$$\exists\, x_3 \in N \cup S \text{ mit } x_3 \in \big((succ^*(x_1) \cap pred^*(x_2)) \cup \{x_2\}\big) \wedge Reads^{DFS}(x_3, d, \texttt{ANY})$$

Innerhalb von Schleifenkörpern kann es nach Setzen der Schleifenrücksprungkante ebenfalls zu einem „Schreiben nach Schreiben“ kommen. Auch hier genügt es wieder, den Anwender auf entsprechende Konstellationen hinzuweisen, ohne sie jedoch zu verbieten.

Insgesamt haben wir uns in diesem Abschnitt auf einige wenige, dafür aber sehr wichtige Regeln zur Strukturierung von DF-Schemata beschränkt. Erweiterungen würden z.B. notwendig werden, wenn neben Parametern bzw. Datenelementen mit atomaren Wertebereichen – wie hier vereinfachend angenommen – auch komplex strukturierte Datenobjekte (Listen, Mengen, Record-Strukturen) abbildbar sein sollen [Ober96]. Hier würde z.B. das Schreiben eines mengenwertigen Datenelements (i.S. des Hinzufügen von Elementen) durch Aktivitäten paralleler Zweige durchaus Sinn machen [Blas96].

5.4 Effiziente Verfahren zur Sicherstellung von DF-Strukturierungsregeln

Wir betrachten in diesem Abschnitt Verfahren, mittels denen für ein gegebenes DF-Schema die Strukturierungsregeln DF-1, DF-2 und DF-3 überprüft werden können. Das Ziel dieser Verfahren ist, durch Ausnutzen der Struktureigenschaften von KF-Graphen die notwendigen Überprüfungen möglichst effizient durchführen zu können. Dies ist insbesondere für dynamische WF-Änderungen wichtig.

5.4.1 Sicherstellung der Versorgung obligater Eingabeparameter

Um für Aktivitätenknoten und für die ihnen vor- bzw. nachgeschalteten Hilfsdienste eine vollständige Versorgung obligater Eingabeparameter gewährleisten zu können, muß die in der Regel DF-1 angegebene Bedingung (*) für alle *obligaten* Lesezugriffe auf den WF-Datenkontext überprüft werden. Grundlage hierfür bildet das im folgenden beschriebene *WriterExists*-Verfahren, das für einen Lesezugriff einer Aktivität n_{read} überprüft, ob das gelesene Datenelement d bei der Aktivierung von n_{read} sicher versorgt werden kann oder nicht, d.h. ob d in jeder möglichen Ausführungskonstellation durch mindestens einen Vorgängerknoten von n_{read} obligat geschrieben wird. Trifft dies zu, dürfen von den Eingabeparametern des Knotens n_{read} (inkl. der ihm vor- und nachgeschalteten Hilfsdienste) obligate Lesekanten zum Datenelement d führen. Andernfalls darf die Aktivität n_{read} den Wert d nur dann lesen, wenn er von einem direkt vorgeschalteten Hilfsdienst geschrieben wird.

Eine naive Lösung wäre, ausgehend von der Aktivierung des Startknotens, eine vollständige Erreichbarkeitsanalyse für den KF-Graphen durchzuführen, d.h. alle erreichbaren Markierungen für Aktivitätenknoten zu bestimmen (vgl. Abschnitt 4.6.4). Unser Problem würde sich dann auf die Fragestellung reduzieren, ob es für jede erreichbare Markierung, unter welcher der Aktivitätenknoten n_{read} aktiviert ist ($NS(n_{read}) = \texttt{ACTIVATED}$), immer mindestens einen Knoten mit der Markierung `COMPLETED` gibt, der das Datenelement d obligat schreibt (inkl. Schreibzugriffen seiner vor- und nachgeschalteten Hilfsdienste). Dieser Ansatz würde bei parallelen Ausführungszweigen, aufgrund der kombinatorischen Vielfalt erreichbarer Knotenmarkierungen, aber schnell zu sehr aufwendigen Analysen mit ex-

[19] Korrekterweise müsste hier die Definition der Nachfolgerfunktion und ihres transitiven Abschlusses so erweitert werden, daß sie auch Reihenfolgen zwischen Aktivitäten und/oder Hilfsdiensten abbildet. Grundlage hierfür bilden Reihenfolgenbeziehungen zwischen Aktivitäten und die Positionsnummern der Hilfsdienste.

ponentieller Komplexität führen. Das Ziel muß daher sein, zu Verfahren zu gelangen, durch deren Anwendung sich solche aufwendigen DF-Analysen weitgehend vermeiden lassen.

Wir stellen in Abschnitt 5.4.1.1 ein solches Analyseverfahren für azyklische KF-Graphen ohne Sync- und Schleifenkanten vor. Anschließend werden wir es in Abschnitt 5.4.1.2 so erweitern, daß es auch für allgemeine KF-Graphen anwendbar wird. Dabei müssen wir uns mit der Fragestellung auseinandersetzen, unter welchen Bedingungen obligate Lesezugriffe des Knotens n_{read}, die nicht von Vorgängern bzgl. normaler Kontrollkanten versorgt werden können, infolge der Verwendung von Sync-Kanten dennoch zulässig sind. Hier muß ein Verfahren gefunden werden, das für beliebig strukturierte KF-Graphen anwendbar bleibt und das auch transitive Reihenfolgebeziehungen berücksichtigt. Insgesamt bilden die nun vorgestellten Verfahren eine zentrale Säule des ADEPT-Basismodells.

5.4.1.1 Wohlstrukturierte, azyklische KF-Graphen (ohne Sync-Kanten)

Wir betrachten zuerst einfache KF-Graphen ohne Synchronisations- und Schleifenkanten. Prinzipiell kommen hier alle Aktivitäten (bzw. Hilfsdienste), die Vorgänger von n_{read} im KF-Graphen sind und die das Datenelement *d* schreiben, als potentieller Versorger für den Lesezugriff in Frage. Um das Schreiben von *d* unabhängig von den jeweils durchlaufenen Ausführungspfaden gewährleisten zu können, müssen Schreibzugriffe, die innerhalb von Verzweigungen mit OR-Join-Knoten erfolgen, gesondert behandelt werden. Der Grund dafür ist, daß bei der Aktivierung eines OR-Joins jeweils nur die Aktivitäten eines Teilzweiges als ausgeführt gelten und dem entsprechend auch nur deren Ausgabedaten für Nachfolgeschritte zur Verfügung stehen. Wird das Datenelement *d* ausschließlich durch Aktivitäten geschrieben, die innerhalb einer solchen Verzweigung liegen und soll es von Knoten obligat gelesen werden, die der Verzweigung nachfolgen (inkl. dem OR-Join-Knoten), muß demzufolge für alle Teilzweige sichergestellt werden, daß *d* dort mindestens einmal geschrieben wird.

Die nun vorgestellte einfache Variante des *WriterExists*-Verfahrens basiert auf der folgenden Grundidee: Jeder Aktivitätenknoten $n \in N$ wird mit einer Marke $Supplied(n) \in \{True, False\}$ versehen, die initial den Wert *False* besitzt. Das Verfahren ist so konstruiert, daß nach seiner Beendigung exakt diejenigen Knoten *n* mit $Supplied(n) = True$ markiert sind, die das Datenelement *d* obligat lesen dürfen. Des weiteren gilt nach seiner Ausführung:

(1) Ein Nicht-OR-Join-Knoten $n \in c_pred^*(n_{read}) \cup \{n_{read}\}$ (mit einer Eingangssemantik $V_{in}^n \in$ {`ONE_Of_ONE`, `ALL_Of_ALL`} ist genau dann als „versorgt“ markiert (d.h. es gilt $Supplied(n) = True$), wenn mindestens einer seiner direkten Vorgänger *x* (bzgl. normaler Kontrollkanten) das Datenelement *d* obligat schreibt (inkl. Schreibzugriffen durch assoziierte Hilfsdienste) oder seinerseits mit $Supplied(x) = True$ markiert worden ist, d.h. *d* obligat lesen darf (**). Mit anderen Worten: Ist ein obligater Lesezugriff eines Knotens *n* auf ein Datenelement *d* sicher versorgt, so gilt dies unmittelbar auch für seine direkten Nachfolger, wenn sie eine Eingangssemantik ungleich `ONE_Of_ALL` besitzen.

(2) Ein OR-Join-Knoten *n* (mit Eingangssemantik V_{in}^n = `ONE_Of_ALL`) ist dagegen nur dann als $Supplied(n) = True$ markiert, d.h. er darf das Datenelement *d* obligat lesen, wenn die Bedingung (**) für *alle* direkten Vorgänger von *n* (bzgl. normaler Kontrollkanten) zutrifft. Um dies feststellen zu können, ordnen wir jedem Knoten n^* einen Zähler $Counter(n^*)$ zu, der angibt, über wieviele der in n^* einmündenden Ausführungszweige *d* versorgt ist. Dieser Zähler, der initial für jeden Knoten den Wert `0` besitzt, wird im Verlauf des Verfahrens immer dann um eins erhöht, wenn ein direkter Vorgänger des betreffenden Knotens neu markiert wird. Für einen OR-Join *n* ist die Versorgung von *d* folglich sichergestellt, wenn der Wert seines Zählers $Counter(n)$ gleich der Anzahl der in diesen Knoten einmündenden Zweige bzw. Kontrollkanten ist.

Das *WriterExists*-Verfahren arbeitet nach folgendem Grundprinzip: Initial werden für jeden Knoten des KF-Graphen die Attribute *Supplied* und *Counter* auf *False* bzw. `0` gesetzt. Dann wird zuerst für diejenigen Aktivitätenknoten, die das Datenelement *d* obligat schreiben (inkl. Schreibzugriffen assoziierter Hilfsdienste) und die direkter oder indirekter Vorgänger von n_{read} im KF-Graphen (bzgl. normaler Kontrollkanten) sind, überprüft, ob ihre direkten Nachfolgerknoten *d* obligat lesen dürfen. Dazu wird zuerst der Zähler *Counter* des betrachteten Nachfolgerknotens um eins erhöht. Grundlegend für die Überprüfung sind dann die in (1) und (2) angegebenen Kriterien. Kann ein betrachteter Knoten neu

als versorgt markiert werden, wird in der nächsten Iteration des Verfahrens wiederum für dessen direkte Nachfolger geprüft, ob sie ihrerseits *d* obligat lesen und damit als versorgt markiert werden können. Nach demselben Prinzip wird die Menge der Knoten, die *d* obligat lesen dürfen, sukzessive expandiert, bis sie entweder den Knoten n_{read} enthält oder aber keine neuen Elemente mehr aufgenommen werden können. Algorithmus 5-1 faßt die wesentlichen Schritte zusammen, die zur Überprüfung der Versorgung obligater Lesezugriffe bei azyklischen KF-Graphen ohne Sync- und Schleifenkanten anfallen. Eine formale Darstellung dieses Algorithmus kann Anhang C entnommen werden. Dort werden auch Schreibzugriffe über Sync-Kanten berücksichtigt (vgl. Abschnitt 5.4.1.2).

// *Überprüfung, ob der Knoten n_{read} das Datenelement d obligat lesen darf – Variante 1: Es werden*
// *nur Schreibzugriffe von Vorgängern über normale Kontrollkanten berücksichtigt*

Algorithmus 5-1 (WriterExists(CFS, DFS, n_{read}, d) → {*True*, *False*})

1. Bestimme die Menge $Pred = c_pred^*(n_{read})$ aller Vorgänger des Knotens n_{read} (bzgl. Kontrollkanten des Typs `CONTROL_E`). Initialisiere für alle Knoten $n \in N$ das Attribut *Supplied*(*n*) mit *False* und den Zähler *Counter*(*n*) mit `0`.

2. Initialisiere die Menge der aktuell zu betrachtenden Knoten mit der leeren Menge: *NodeList*:= ∅.

   Für jeden obligaten Schreibzugriff auf das Datenelement *d* durch einen Knoten $n \in Pred$ (bzw. einen vor-/nachgeschalteten Hilfsdienst von *n*) führe die folgende Anweisung aus:

   Inkrementiere für jeden direkten Nachfolgerknoten n^* von *n* (bzgl. Kanten des Typs `CONTROL_E`), der innerhalb der Menge $Pred \cup \{n_{read}\}$ liegt, den Zähler $Counter(n^*)$ um `1` und füge den Knoten n^* der Menge *NodeList* hinzu.

   (*NodeList* enthält im Anschluß nur Knoten, für die gilt: *d* wird in mindestens einem der in diesen Knoten einmündenden Ausführungszweige sicher versorgt).

3. Falls *NodeList* = ∅ gilt, gehe zu Schritt 5. Andernfalls: Initialisiere die Menge der in der nächsten Iteration zu betrachtenden Knoten mit der leeren Menge (*NewNodeList*:= ∅).

   Überprüfe für jeden Knoten der Menge *NodeList*, ob bei seiner Aktivierung das Datenelement *d* sicher versorgt ist: Für einen Knoten $n \in NodeList$ trifft dies genau dann zu, wenn er entweder eine der beiden Eingangssemantiken `ONE_Of_ONE` bzw. `ALL_Of_ALL` aufweist [vgl. (1)], oder wenn er die Eingangssemantik `ONE_Of_ALL` besitzt und sein Zähler *Counter(n)* gleich der Anzahl der in *n* einmündenden Kontrollkanten bzw. Teilzweige ist [vgl. (2)]. Ist eine dieser beiden Bedingungen erfüllt, so setze das Attribut *Supplied(n)* auf den Wert *True*, und führe für jeden direkten Nachfolger n^* von *n* wiederum die folgende Anweisung aus:

   Falls $n^* \in Pred \cup \{n_{read}\}$ und $Supplied(n^*) = False$ gilt, so erhöhe den Zähler $Counter(n^*)$ um `1` und füge den Knoten n^* der Menge *NewNodeList* (→ duplikatfrei) hinzu.

4. Falls *NewNodeList* = ∅ oder $Supplied(n_{read}) = True$ gilt, so gehe zu Schritt 5. Andernfalls setze *NodeList*:= *NewNodeList* und wiederhole Schritt 3 (d.h. propagiere die Markierungen weiter entlang des Kontrollflusses).

5. Die Datenversorgung des obligaten Lesezugriffs von n_{read} auf *d* ist genau dann sicher gestellt, wenn für die Markierung dieses Knotens $Supplied(n_{read}) = True$ gilt (→ Algorithmus liefert Rückgabewert $WriterExists(CFS, DFS, n_{read}, d) = True$). Andernfalls kann der Lesezugriff nicht in jedem Fall von Vorgängerknoten (bzgl. normaler Kontrollkanten) versorgt werden (→ Rückgabewert $WriterExists(CFS, DFS, n_{read}, d) = False$).

Wir wollen die Anwendung dieses Verfahrens anhand eines Beispiels illustrieren.

Beispiel 5-5 (DF-Analyse bei wohlstrukturierten, azyklischen KF-Graphen)

Im KF-Graphen aus Abbildung 5-5 soll die Versorgung des Lesezugriffs von Aktivität `I` auf dem Datenelement `d` überprüft werden – die Schreibzugriffe der Knoten `C` und `D` seien obligat.

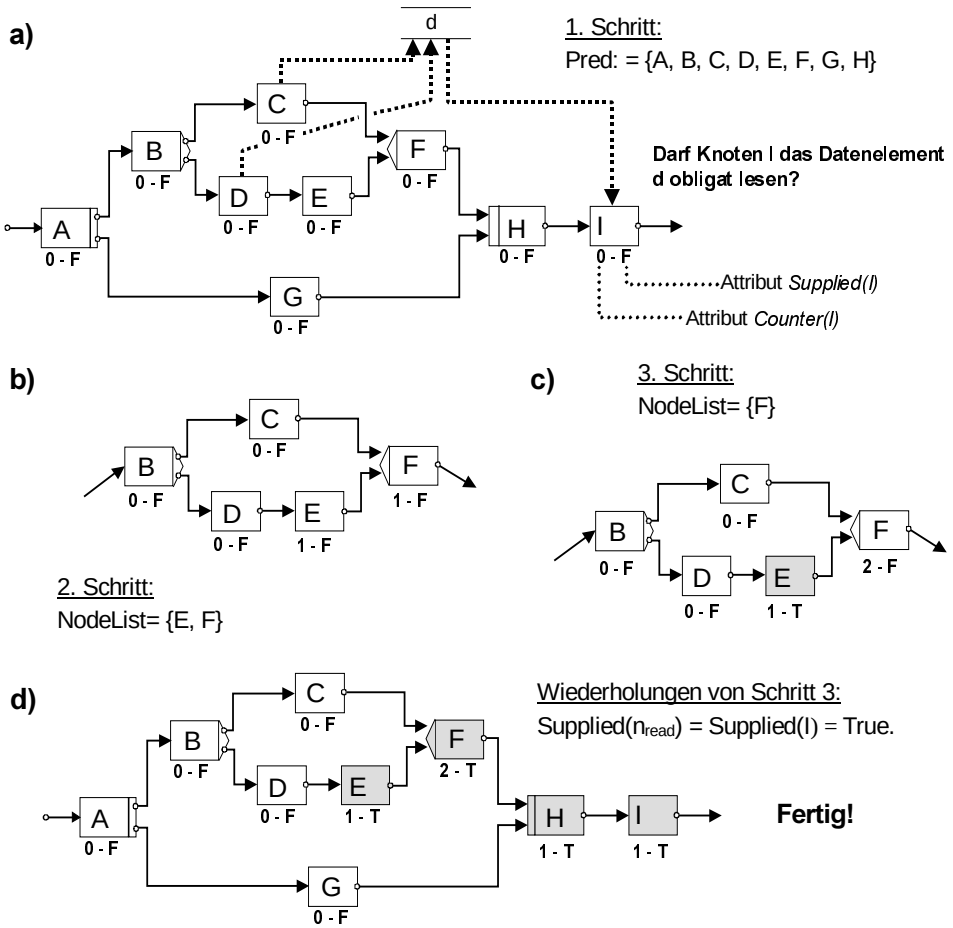


Abbildung 5-5: *Überprüfung der Parameterversorgung einer Aktivität* [*hier Knoten I; unterhalb der Knoten sind jeweils die aktuellen Werte der Attribute Counter und Supplied dargestellt (T: `True`, F: `False`) – die als versorgt markierten Knoten sind zusätzlich grau dargestellt*]

1. Schritt: Bestimme die Menge *Pred* aller Vorgänger von `I`: *Pred* = {`A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`}. Initialisiere die Attribute *Supplied* und *Counter* für Elemente dieser Menge und für `I`. (→ Abbildung 5-5 a)

2. Schritt: Bestimme diejenigen Aktivitäten aus *Pred*, die `d` obligat schreiben (→ Knoten `C` und `D`). Erhöhe für alle direkten Nachfolger von `C` und `D` (im Beispiel sind dies die Knoten `F` und `E`) das Attribut *Counter* um 1 und füge die Knoten der Menge *NodeList* hinzu. Wir erhalten: *NodeList* = {`E`, `F`} (→ Abbildung 5-5 b)

3. Schritt: Prüfe für jedes Element aus *NodeList* = {`E`, `F`}, ob die angegebene Bedingung erfüllt ist:

- Für `F` gilt: V_{in}^{F} = `ONE_Of_ALL` und *Counter*(`F`) < 2, d.h. die Bedingung ist (noch) nicht erfüllt.
- Für `E` gilt: V_{in}^{E} = `ONE_Of_ONE`, d.h. die Bedingung ist erfüllt, ⇒ Setze *Supplied*(`E`):= *True*, erhöhe für den direkten Nachfolgerknoten `F` den Zähler *Counter*(`F`) um `1` (neuer Wert: *Counter*(`F`) = 2) und füge `F` der Menge *NewNodeList* hinzu (→ Abbildung 5-5 c). Wir erhalten: *NewNodeList* = {`F`} ≠ ∅. Wegen *Supplied*(`I`) = `False`, wiederhole Schritt 3 mit *NodeList*:= *NewNodeList* = {`F`}

Wiederholung 3. Schritt: Prüfe für jedes Element aus *NodeList* = {`F`}, ob die angegebene Bedingung erfüllt ist:

- Für den Knoten `F` gilt: V_{in}^{F} = `ONE_Of_ALL` und *Counter*(`F`) = 2, d.h. die Bedingung ist erfüllt, ⇒ Setze *Supplied*(`F`):= *True*, erhöhe für den direkten Nachfolgerknoten `H` von `F` den Zähler *Counter*(`H`) um `1` (neuer Wert: *Counter*(`H`) = 1) und füge `H` der Menge *NewNodeList* hinzu. – Wegen *Supplied*(`I`) = *False*, wiederhole Schritt 3 mit *NodeList*:= *NewNodeList* = {`H`}

Wiederholung 3. Schritt: Prüfe für jedes Element aus *NodeList* = {`H`}, ob die angegebene Bedingung erfüllt ist:

- Für den Knoten `H` gilt: V_{in}^{H} = `ALL_Of_ALL`, d.h. die geforderte Bedingung ist erfüllt, ⇒ Setze *Supplied*(`H`):= *True*, erhöhe für den direkten Nachfolgerknoten `I` von `H` den Zähler *Counter*(`I`) um `1` (neuer Wert: *Counter*(`I`) = 1) und füge `I` der Menge *NewNodeList* hinzu, Wegen *Supplied*(`I`) = *False*, wiederhole Schritt 3 mit *NodeList*:= *NewNodeList* = {`I`}

Wiederholung 3. Schritt: Prüfe für jedes Element aus *NodeList* = {I}, ob die angegebene Bedingung erfüllt ist:

- Für den Knoten I gilt: V_{in}^{I} = ONE_Of_ONE, d.h. die geforderte Bedingung ist erfüllt, ⇒ Setze *Supplied*(I):= *True* (→ Abbildung 5-5 d)

Da *Supplied*(I) = True gilt, bricht das Verfahren anschließend ab, und wir erhalten *WriterExists*(I, d) = *True*. Der obligate Lesezugriff von I auf d wird also sicher versorgt. Dies wäre z.B. nicht der Fall, wenn man in dem gegebenen DF-Schema die obligate Schreibkante von D nach d entfernen würde, oder wenn eine der beiden Schreibkanten optional wäre.

Algorithmus 5-1 ist auf beliebig geschachtelte KF-Graphen anwendbar und terminiert immer nach endlich vielen Schritten. In jeder Iteration des Verfahrens (→ *Schritt 3*) wird entweder mindestens ein Aktivitätenknoten $n \in N$ neu als *Supplied*(n) = *True* markiert oder aber der Algorithmus bricht ab. Wichtig dabei ist, daß ein einmal als „versorgt" markierter Knoten in der Folge nicht mehr betrachtet werden muß. Da die Anzahl |N| der Knoten eines KF-Graphen endlich ist, terminiert das Verfahren somit spätestens nach |N| Durchläufen[20]. Des weiteren muß die Versorgung des obligaten Lesezugriffs eines bestimmten Knotens nur dann mehrfach überprüft werden, wenn der Knoten die Eingangssemantik ONE_OF_ALL besitzt. In allen anderen Fällen kann ein innerhalb von Schritt 3 betrachteter Knoten aus der Menge *NodeList* nach der Konstruktion des Verfahrens als versorgt markiert werden. Bezogen auf die Anzahl der Vergleichsoperationen, die zur Überprüfung der Versorgung eines obligaten Lesezugriffs notwendig werden, beträgt die Komplexität demnach $O(|N|^2)$.

Die bisher vorgestellte Variante des *WriterExists*-Verfahrens berücksichtigt bei der Prüfung der Parameterversorgung des Knotens n_{read} lediglich Schreibzugriffe von Vorgängern bzgl. normalen Kontrollkanten. Im folgenden Abschnitt erweitern wir das Verfahren dahingehend, daß auch Schreibzugriffe von Vorgängern bzgl. Sync-Kanten zur Versorgung des Lesezugriffs beitragen können.

5.4.1.2 Wohlstrukturierte, azyklische KF-Graphen mit Sync-Kanten

Das im vorangehenden Abschnitt vorgestellte Verfahren nutzt die regelmäßige Blockstrukturierung der KF-Graphen konsequent aus. Es bleibt in der Mehrzahl der Fälle auch für die Überprüfung der Datenversorgung in KF-Graphen mit Sync-Kanten anwendbar, zumindest wenn die Eingabeparameter einer Aktivität wieder durch Vorgängerknoten bzgl. normaler Kontrollkanten versorgt werden können. Wie am Beispiel aus Abbildung 5-1 erkennbar wird, ergeben sich bei der Verwendung von Sync-Kanten allerdings noch weiterreichende Möglichkeiten für die Parameterversorgung einer Aktivität. In dem dargestellten DF-Schema könnte zum Beispiel die Aktivität G das Datenelement d_2 (gemäß Regel DF-1) obligat lesen, obwohl der Schreiber B kein Vorgänger von G bzgl. normaler Kontrollkanten ist. Der Lesezugriff ist trotzdem möglich, da G vom Knoten B aus über Kontroll- und Sync-Kanten erreichbar ist und B in jeder Ausführungskonstellation vor G beendet wird. Interessanterweise darf G das von D – der Knoten D liegt auf dem Weg von B nach G – geschriebene Datenelement d_3 nicht obligat lesen, da hier im Gegensatz zum gerade skizzierten Fall nicht sichergestellt ist, daß der Schreiberknoten D stets vor der Aktivierung von G abgeschlossen wird. Um solche Fälle bei der DF-Analyse korrekt behandeln zu können, muß das oben vorgestellte Verfahren geeignet erweitert werden. Im Mittelpunkt steht dabei die Frage, wann obligate Eingabeparameter einer Aktivität n_{read} durch Knoten paralleler Zweige, die Vorgänger von n_{read} bzgl. Sync- und Kontrollkanten sind, versorgt werden können.

Bei der Durchführung solcher erweiterten DF-Analysen erweisen sich *strikte Sync-Kanten* aufgrund ihrer Ausführungssemantik als unproblematisch. Schreibt der Quellknoten einer strikten Sync-Kante das Datenelement *d* obligat oder darf er seinerseits *d* obligat lesen, so sind (obligate) Lesezugriffe ihres Zielknotens auf *d* aufgrund der Semantik strikter Sync-Kanten in jedem Fall versorgt, unabhängig von zuvor getroffenen Verzweigungsentscheidungen. Schwieriger verhält es sich mit *einfachen Sync-Kanten*, da hier die Ausführung der Quellaktivität keine notwendige Bedingung für das Signalisieren der Sync-Kante und damit für die Aktivierung des Zielknotens darstellt. Beispielsweise kann die Quellaktivität einer einfachen Sync-Kante in einem Teilzweig einer bedingten Verzweigung oder

[20] Dieser „worst case" wäre z.B. bei einem sequentiellen WF gegeben, wenn ein obligater Lesezugriff des WF-Endknotens durch einen obligaten Schreibzugriff des WF-Startknotens versorgt werden soll.

einer Parallelverzweigung mit finaler Auswahl enthalten sein, dem der Zielknoten nicht angehört (vgl. Abschnitt 3.3.2.4). Dem entsprechend können, je nach Kontext der Kante, Schreibzugriffe der Quellaktivität und ihrer Vorgänger zur Versorgung von Lesezugriffen des Zielknotens (und seiner Nachfolger) herangezogen werden oder auch nicht. Im allgemeinen muß sichergestellt werden (vgl. Regel DF-1), daß für jede mögliche Kombination von Aktivitäten, die bei den verwendeten Kontroll- und Sync-Kanten vor der Aktivierung von n_{read} zur Ausführung kommen können, es immer mindestens einen Knoten gibt, der *d* obligat schreibt (inkl. Schreibzugriffen assoziierter Hilfsdienste). Andernfalls sind obligate Lesezugriffe auf *d* durch den Knoten n_{read} nicht zulässig.

Das nun vorgestellte Verfahren zur Überprüfung des (obligaten) Lesezugriffs eines Knotens n_{read} auf ein Datenelement *d* kann auf beliebig geschachtelte KF-Graphen mit einfachen und strikten Sync-Kanten angewendet werden. Das Verfahren durchläuft zwei Phasen. In der *1. Phase* wird überprüft, ob der Lesezugriff von n_{read} auf *d* bereits durch Schreibzugriffe von Vorgängerknoten bzgl. normaler Kontrollkanten versorgt werden kann. Dazu wird der in Abschnitt 5.4.1.1 vorgestellte Algorithmus angewendet. Für den Fall, daß die Versorgung des Lesezugriffs auf diesem Weg nicht gewährleistet werden kann, wird in einer *2. Phase* geprüft, ob hierfür Schreiber von *d* aus parallelen Zweigen in Frage kommen. Dabei werden nur solche Knoten berücksichtigt, die *Vorgänger* von n_{read} (bzgl. Kontroll- und Sync-Kanten) sind.

Diese zusätzliche Überprüfung basiert auf der folgenden Idee: Für jeden Aktivitätenknoten *n* des KF-Graphen werden die in der 1. Phase ermittelten Knotenmarkierungen *Supplied(n)* und *Counter(n)* übernommen. Anschließend wird die Menge der Aktivitätenknoten, die in parallelen Ausführungszweigen von n_{read} liegen und die Vorgänger von n_{read} bzgl. Kontroll- und Sync-Kanten sind, bestimmt. Befindet sich unter diesen Knoten kein Schreiber von *d*, so kann die Versorgung des Lesezugriffs nicht gewährleistet werden und wir sind fertig. Andernfalls wird – ausgehend von diesen Schreiberknoten – wieder versucht, die Menge der Knoten, die *d* obligat lesen dürfen, sukzessive zu expandieren, bis sie entweder den Knoten n_{read} enthält oder aber kein neues Element mehr hinzugenommen werden kann. Hierzu verwenden wir ein gegenüber Abschnitt 5.4.1.1 erweitertes Markierungsverfahren, bei dem zuerst für die direkten Nachfolger (bzgl. Kontroll- und Sync-Kanten) der ermittelten Schreiberknoten überprüft wird, ob sie das Datenelement *d* obligat lesen dürfen oder nicht. (Diese Überprüfung findet nur für diejenigen direkten Nachfolger statt, die Vorgänger von n_{read} (inkl.) bzgl. Kontroll- und Sync-Kanten sind.) Falls ja, wird der betreffende Knoten entsprechend markiert und in der nächsten Iteration des Verfahrens wiederum für seine direkten Nachfolger geprüft, ob sie *d* ebenfalls obligat lesen dürfen, usw. Dieser Vorgang wiederholt sich solange, bis entweder der Knoten n_{read} als versorgt (d.h. mit $Supplied(n_{read}) = True$) gekennzeichnet oder aber kein weiterer Knoten mehr entsprechend markiert werden kann.

Interessant ist in diesem Kontext die Fragestellung, wann der direkte Nachfolger eines Knotens, der *d* obligat schreibt oder der *d* obligat lesen darf, auf dieses Datenelement lesend zugreifen darf. Für direkte Nachfolger bzgl. normaler Kontrollkanten wird wie in Abschnitt 5.4.1.1 verfahren. Der Zielknoten einer strikten Sync-Kante kann als „versorgt“ markiert werden, wenn ihr Quellknoten das Datenelement obligat schreibt oder es seinerseits obligat lesen darf. Die Verwendung einfacher Sync-Kanten macht ggf. aufwendigere Überprüfungen erforderlich. Wird die Quellaktivität n_{source} einer einfachen Sync-Kante neu mit $Supplied(n_{source}) = True$ markiert oder schreibt sie das Datenelement *d* obligat, bedeutet dies nicht zwangsläufig, daß der Zielknoten ebenfalls als versorgt markiert werden darf. Wie bereits erwähnt ist der Grund dafür, daß bei einfachen Sync-Kanten die Ausführung des Quellknotens (bzw. des Teilzweiges der diesen Knoten enthält) keine notwendige Bedingung für die Aktivierung des Zielknotens darstellt. Bei der DF-Analyse genügt es in der Regel auch nicht, einzelne Sync-Kanten isoliert zu betrachten, da die Parameterversorgung eines Knotens ggf. erst durch die kombinierte Anwendung mehrerer Sync-Kanten gewährleistet werden kann. Ein Beispiel hierzu zeigt Abbildung 5-8 b). Würde man in dem hier dargestellten KF-Graphen eine der beiden Sync-Kanten entfernen, so wäre der Lesezugriff des Knotens n_{read} = G auf d nicht mehr sicher versorgt. Schließlich darf der Zielknoten einer einfachen Sync-Kante ein Datenelement *d* ggf. auch dann obligat lesen (im Sinne von Regel DF-1), wenn es von keinem Vorgänger bzgl. normaler Kontrollkanten geschrieben wird und auch kein direkter Vorgänger bzgl. Sync-Kanten es schreibt bzw. als versorgt markiert ist (vgl. Beispiel 5-6 c). Vielmehr bestehen hier transitive Abhängigkeiten zwischen den Aktivitäten verschiedener Parallelzweige, die bei der DF-Analyse zu berücksichtigen sind.

Im allgemeinen kann der Zielknoten n_{dest} einer Sync-Kante nach folgendem Kriterium bewertet werden (vorausgesetzt er wurde nicht bereits anderweitig mit $Supplied(n_{dest}) = True$ markiert):

OR-Split-Kriterium (Datenversorgung der Zielknoten von Sync-Kanten)

Sei $CFS = (N, E, D, ...)$ ein KF-Graph mit Sync-Kanten und *DFS* das zugehörige Datenflußschema. Ferner sei durch $e = n_{source} \rightarrow n_{dest} \in E$ eine Sync-Kante gegeben. Die Menge *providerNodes* umfasse alle Knoten aus *N*, die in einem parallelen Zweig von n_{dest} liegen und die Vorgänger dieses Knotens bzgl. Sync- und Kontrollkanten sind. Der minimale, die Knotenmenge $providerNodes \cup \{n_{dest}\}$ umschließende Kontrollblock sei durch das Knotenpaar (*split*, *join*) bestimmt.

Der Zielknoten n_{dest} der Sync-Kante kann dann als „versorgt", d.h. mit $Supplied(n_{dest}) = True$, markiert werden (d.h. er darf *d* obligat lesen), wenn es einen Weg *Y* vom Knoten *split* zum Knoten n_{source} (über Kontrollkanten des Typs `CONTROL_E`) mit der folgenden Eigenschaft gibt: Für jeden OR-Split *sp*, der auf dem Weg *Y* liegt und dessen zugehöriger OR-Join nicht in *Y* enthalten ist, trifft (mindestens) eine der beiden folgenden Bedingungen zu:

(1) Der Knoten *sp* ist bzgl. *d* als „versorgt" markiert oder er schreibt das Datenelement *d* obligat.

(2) Jeder von *sp* ausgehender Teilzweig enthält mindestens einen Knoten *n*, der Vorgänger von n_{dest} im KF-Graphen (bzgl. Kontroll- und Sync-Kanten) ist und der das Datenelement *d* ebenfalls obligat lesen darf (d.h. es gilt $Supplied(n) = True$) oder es obligat schreibt.

Wir wollen die Anwendung des OR-Split-Kriteriums anhand einfacher Beispiele illustrieren.

Beispiel 5-6 (OR-Split-Kriterium)

In den unten abgebildeten KF-Graphen sind jeweils diejenigen Knoten grau dargestellt, die das Datenelement d obligat lesen dürfen, wenn das in Abschnitt 5.4.1.1 vorgestellte einfache WriterExists-Verfahren angewendet wird. Wie man leicht sieht, kann in keinem der drei Fälle der obligate Lesezugriff von `X` auf `d` durch Schreibzugriffe von Vorgängerknoten bzgl. normaler Kontrollkanten versorgt werden. Dem entsprechend besitzt der Knoten `X` jeweils die Markierungen *Counter*(`X`) = 0 und *Supplied*(`G`) = *F*. In den beiden oberen KF-Graphen ist der Quellknoten der einfachen Sync-Kante `D` → `X` als versorgt markiert [→ a)] oder er schreibt das Datenelement obligat [→ b)]. Wendet man das OR-Split-Kriterium auf diese beiden KF-Graphen (mit ihren aktuellen Markierungen) an, ergibt sich: Für den in a) dargestellten KF-Graphen kann der Zielknoten `X` der Kante `D` → `X` ebenfalls als versorgt markiert werden (Weg *Y* = `A`→ `B`→ `D` mit *ActivityWrites*(`B`, d, `MANDATORY`) = *True*), wohingegen dies für den KF-Graphen aus b) nicht möglich ist. Ein komplexeres Beispiel zeigt der KF-Graph aus c), der drei einfache Sync-Kanten (`E` → `K`, `F` → `I`, `I` → `X`) besitzt. In diesem KF-Graphen wären weder für den Knoten `I` noch für den Knoten K obligate Lesezugriffe auf `d` zulässig. (Für keinen der beiden Knoten ist das OR-Split-Kriterium erfüllt.). Für den Knoten X dagegen wäre ein Lesezugriff auf `d` sicher versorgt, obwohl seine direkten Vorgänger `I` und `K` selbst dieses Datenelement nicht lesen dürfen. Wie man leicht sieht, wird aufgrund der verwendeten Sync-Kanten, stets einer der beiden Knoten E oder F vor der Aktivierung von `X` beendet (Das OR-Split-Kriterium ist erfüllt: Y = `A` → `B` → `C` → `D` → `F` → `I` mit *ActivityWrites*(`E`, `d`, `MANDATORY`) = *True* und *ActivityWrites*(`F`, `d`, `MANDATORY`) = *True*, `E`, `F` ∈ $pred^*$(`X`).

Zur Überprüfung des OR-Split-Kriteriums dient ein einfachen Verfahren (→ SyncDestSupplied, vgl. Anhang C), bei dem zuerst alle relevanten Schreiberknoten des betreffenden Datenelements und alle als versorgt gekennzeichneten Knoten mit einer Markierung *Checked* = *True* versehen werden – vorausgesetzt sie sind Vorgänger des Zielknotens der Sync-Kante (bzgl. Kontroll- und Sync-Kanten). (Für alle anderen Knoten des KF-Graphen wird dieses Attribut auf den Wert *False* gesetzt.) Hiervon ausgehend wird versucht, entsprechende Markierungen auf die jeweiligen Vorgängerknoten zu propagieren. Dabei werden Knoten mit Ausgangssemantik `ONE_Of_ONE` oder `ALL_Of_ALL` genau dann als *Checked* = *True* markiert, wenn dies für mindestens einen ihrer direkten Nachfolger zutrifft. Für OR-Split-Knoten mit Ausgangssemantik `ONE_Of_ALL` erfolgt dies dagegen nur, wenn alle direkten Nachfolger eine entsprechende Markierung besitzen. Das Verfahren bricht ab, wenn kein weiterer Knoten mehr markiert werden kann oder der Knoten *split* mit *Checked*(*split*) = *True* markiert wird. In letzterem Fall ist der Lesezugriff des Zielknotens der Sync-Kante versorgt. Eine formale Darstellung dieses Verfahrens findet man in Anhang C.

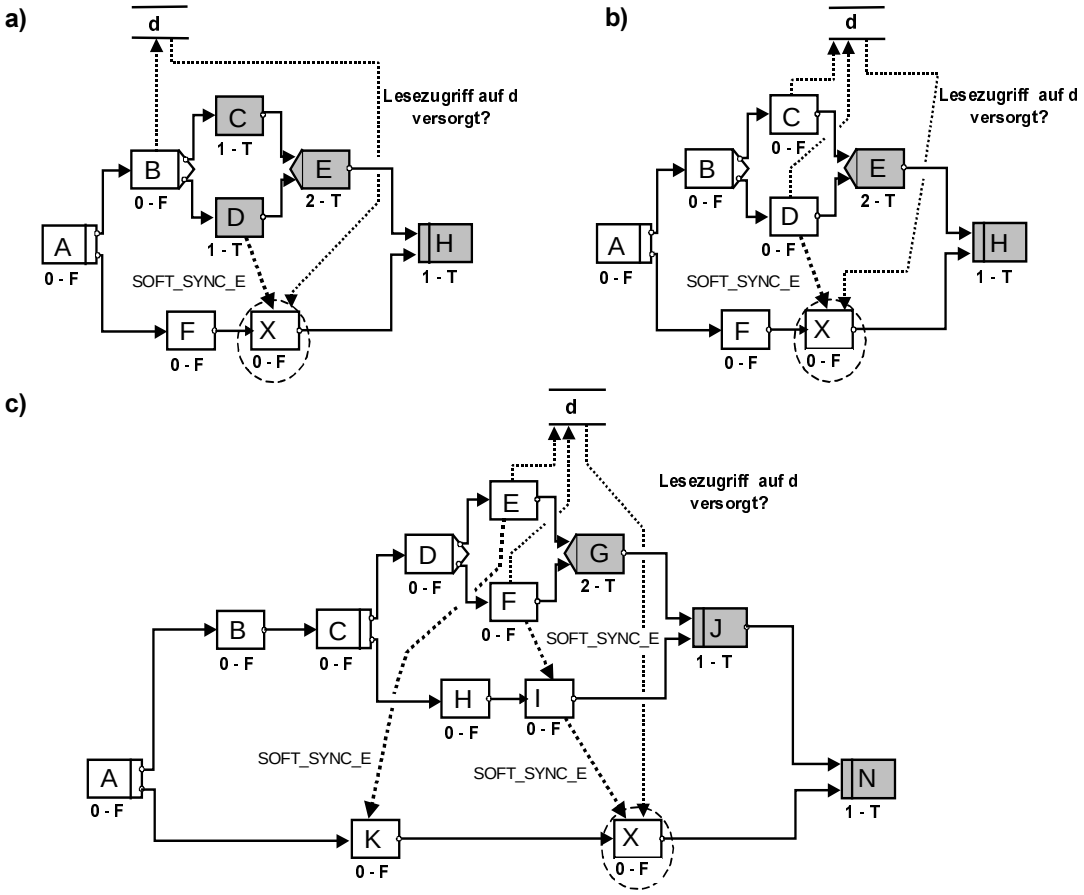


Abbildung 5-6: *Beispiele für Anwendung des OR-Split-Kriteriums*

Wir skizzieren nun die wesentlichen Schritte, die zur Überprüfung der Versorgung obligater Lesezugriffe bei der Verwendung von Sync-Kanten durchzuführen sind. Eine formale Darstellung dieses erweiterten *WriterExists*-Verfahrens kann Anhang C entnommen werden. Hier werden weitere Details berücksichtigt, auf die wir an dieser Stelle aus Platzgründen nicht näher eingehen. Zu nennen sind transitive Abhängigkeiten, die zwischen den Aktivitäten paralleler Teilzweige (über Sync-Kanten) bestehen können (vgl. Beispiel 5-6 c), die Behandlung paralleler Verzweigungen mit finaler Auswahl[21] und verschiedene Optimierungen.

// *Überprüfung, ob der Knoten n_{read} das Datenelement d obligat lesen darf. Variante 2: Es werden*
// *Schreibzugriffe von Vorgängern bzgl. normaler Kontrollkanten und Sync-Kanten berücksichtigt*

Verallgemeinerung von Algorithmus 5-1: (WriterExists(CFS, DFS, n_{read}, d) → {*True*, *False*}**)**

Phase I: Überprüfung der Datenversorgung durch Vorgänger bzgl. Kontrollkanten

1.-5. wie in Abschnitt 5.4.1.1 beschrieben. Besitzt das Attribut *Supplied*(n_{read}) nach Beendigung von Schritt 4 den Wert *True*, so ist der obligate Lesezugriff von n_{read} auf *d* bereits durch Schreibzugriffe von „normalen" Vorgängerknoten (bzgl. Kanten des Typs `CONTROL_E`) versorgt. Für diesen Fall werden keine zusätzlichen Überprüfungen notwendig, und das Verfahren terminiert mit Schritt 5. Andernfalls werden die Menge *NodeList* sowie die bisher vorgenommenen Markierungen (Attribute *Counter* und *Supplied*) übernommen und mit Schritt 6 fortgefahren.

---

[21] Schreibzugriffe von Knoten aus Teilzweigen einer Parallelverzweigung mit finaler Auswahl werden nur berücksichtigt, wenn der Leserknoten n_{read} in dem entsprechenden Teilzweig liegt oder wenn er ein Nachfolger des OR-Join-Knotens dieser Verzweigung ist.

Phase II: Überprüfung der Datenversorgung durch Vorgänger bzgl. Kontroll- und Sync-Kanten

6. Bestimme die Menge *SyncPred*: = $pred^*(n_{read})$ aller direkten und indirekten Vorgänger von n_{read} (bzgl. normaler Kontrollkanten und Sync-Kanten).

   Falls es unter den Vorgängerknoten aus parallelen Zweigen (d.h. den Knoten der Menge *SyncPred – Pred*) keinen Schreiberknoten von *d* gibt, bricht das Verfahren mit dem Rückgabewert *False* ab, d.h. der Lesezugriff von n_{read} auf *d* kann nicht (sicher) versorgt werden.

   Andernfalls: Initialisiere für jeden Knoten $n \in SyncPred \cup \{n_{read}\}$ das Attribut *SyncSupplied(n)* mit *False* (das Attribut zeigt an, ob *n* „über eingehende Sync-Kanten versorgt werden kann").

7. Für jeden obligaten Schreibzugriff auf *d* durch einen Knoten $n_{write} \in SyncPred - Pred$ (diese Schreibzugriffe wurden in der Phase I des Verfahrens noch nicht berücksichtigt), überprüfe, ob direkte Nachfolger von n_{write} bzgl. Kontroll- und Sync-Kanten das Datenelement *d* obligat lesen dürfen. (Die Überprüfung ist nur für die Nachfolgerknoten notwendig, die der Menge $SyncPred \cup \{n_{read}\}$ angehören und die noch nicht als versorgt markiert worden sind.)

   - Für direkte Nachfolger n^* von n_{write} bzgl. normaler Kontrollkanten (mit *ET* = `CONTROL_E`), erhöhe den Zähler $Counter(n^*)$ um 1 und füge n^* der Menge der noch zu betrachtenden Knoten *NodeList* hinzu.
   - Für direkte Nachfolger n^* von n_{write} bzgl. strikter Sync-Kanten, setze das Attribut $SyncSupplied(n^*)$ auf *True* und füge den Knoten der Menge *NodeList* hinzu.
   - Für direkte Nachfolger n^* von n_{write} bzgl. einfacher Sync-Kanten prüfe, ob das beschriebene OR-Split-Kriterium erfüllt ist. Falls ja, setze das Attribut $SyncSupplied(n^*)$ auf *True* und füge den Knoten der Menge *NodeList* hinzu.

   Gilt im Anschluß an Schritt 7 *NodeList* = ∅, so konnte kein direkter Nachfolgerknoten als „versorgt" markiert werden. Für diesen Fall muß für noch nicht betrachtete Sync-Kanten, deren Zielknoten in $SyncPred \cup \{n_{read}\}$ enthalten ist, noch geprüft werden, ob diese Zielknoten das Datenelement *d* ggf. doch lesen dürfen, z.B. infolge transitiver Abhängigkeiten zu vorangehenden Schreiberknoten paralleler Zweige (vgl. Beispiel 5-6 c). Falls dem so ist, werden die betreffenden Knoten der Menge *NodeList* hinzugefügt. Wir verzichten an dieser Stelle auf die Darstellung dieser zusätzlichen Überprüfung und verweisen auf Anhang C.

8. Falls *NodeList* = ∅ gilt, gehe zu Schritt 10. Andernfalls: Initialisiere die Menge der in der nächsten Iteration zu betrachtenden Knoten: *NewNodeList*:= ∅.

   Überprüfe für jeden Knoten $n \in NodeList$, ob er unter den aktuellen Markierungen das Datenelement *d* obligat lesen darf. Ein obligater Lesezugriff ist möglich, wenn *mindestens* eine der folgenden Bedingungen erfüllt ist:

   - Der Knoten *n* besitzt eine der Eingangssemantiken `ONE_Of_ONE` bzw. `ALL_Of_ALL` und es gilt: $Counter(n) \geq 1$.
   - Der Knoten *n* besitzt die Eingangssemantik `ONE_Of_ALL` und der Zähler *Counter(n)* ist gleich der Anzahl der in *n* einmündenden Kontrollkanten.
   - Der Knoten *n* kann durch Schreibzugriff von Vorgängerknoten paralleler Zweige versorgt werden, d.h. es gilt: *SyncSupplied(n) = True*.

   Trifft mindestens eine dieser Bedingungen zu, wird der Knoten *n* als versorgt (*Supplied(n)*:= *True*) markiert. Des weiteren werden für jeden direkten Nachfolger n^* von *n* (bzgl. Kontroll- und Sync-Kanten) – analog zu Schritt 7 – die folgenden Überprüfungen durchgeführt (Die Überprüfung ist nur für die Nachfolgerknoten n^* nötig, die der Menge $SyncPred \cup \{n_{read}\}$ angehören und die noch nicht als versorgt markiert worden sind.):

   - Für Nachfolger n^* von *n* bzgl. normaler Kontrollkanten erhöhe den Zähler $Counter(n^*)$ um 1 und füge den Knoten n^* der Menge der in der Folgeiteration zu betrachtenden Knoten *NewNodeList* (duplikatfrei) hinzu.
   - Für Nachfolger n^* von *n* bzgl. strikter Sync-Kanten, setze das Attribut $SyncSupplied(n^*)$ auf *True* und füge den Knoten der Menge *NewNodeList* hinzu.

- Für Nachfolger n^* von n bzgl. einfacher Sync-Kanten prüfe, ob sie gemäß dem obigen OR-Split-Kriterium als versorgt markiert werden können. Falls ja, setze das Attribut *Sync-Supplied(n^*)* auf *True* und füge den Knoten der Menge *NewNodeList* hinzu.

Falls nach dem Durchlaufen der Menge *NodeList* die Menge *NewNodeList* leer ist und der Knoten n_{read} noch nicht als versorgt markiert worden ist, muß für noch nicht betrachtete Sync-Kanten, deren Zielknoten in der Menge $SyncPred \cup \{n_{read}\}$ liegen und noch nicht als versorgt markiert worden sind, geprüft werden, ob sie das Datenelement d ggf. doch lesen dürfen (vgl. Schritt 7 bzw. Anhang C). Falls dem so ist, werden die betreffenden Knoten der Menge *NewNodeList* hinzugefügt.

9. Falls $NewNodeList = \emptyset$ oder $Supplied(n_{read}) = True$ gilt, gehe zu Schritt 10. Andernfalls setze *NodeList* gleich *NewNodeList* und wiederhole Schritt 8.

10. Die Datenversorgung eines obligaten Lesezugriffs des Knotens n_{read} auf das Datenelement d ist sichergestellt, falls $Supplied(n_{read}) = True$ gilt. In diesem Fall liefert der Algorithmus den Rückgabewert $WriterExists(n_{read}, d) = True$. Andernfalls kann der Lesezugriff nicht sicher versorgt werden. Rückgabewert ist für diesen Fall $WriterExists(n_{read}, d) = False$).

Wir illustrieren das Grundprinzip des erweiterten *WriterExists*-Verfahrens anhand zweier Beispiele:

Beispiel 5-7 (DF-Analyse bei wohlstrukturierten, azyklischen KF-Graphen mit Sync-Kanten)

Für den KF-Graphen aus Abbildung 5-7 a) soll überprüft werden, ob die Aktivität `F` das Datenelement `d` obligat lesen darf. Dazu nehmen wir an, daß die Schreibzugriffe der Knoten `C` und `G` obligat sind.

In Phase I des Verfahrens wird zunächst überprüft, ob der Lesezugriff von `F` auf `d` bereits durch Vorgängerknoten bzgl. normaler Kontrollkanten (→ Aktivitätenknoten `A`, `B`, `C`, `D` und `E`) sicher versorgt werden kann. Dazu wird wie in Abschnitt 5.4.1.1 beschrieben vorgegangen. Abbildung 5-7 b) zeigt das Ergebnis dieser Überprüfung. Wie man leicht sieht, ist keiner der Vorgänger von `F` (inkl.) mit *Supplied = T(rue)* markiert, d.h. für keinen dieser Knoten wäre ein obligater Lesezugriff auf `d` durch die Schreibzugriffe von Vorgängerknoten bzgl. normaler Kontrollkanten sicher versorgt. Lediglich der OR-Join-Knoten `E` besitzt mit *Counter*(`E`)=1 einen Zähler größer als 0, was bedeutet, daß `d` in einem der in `E` einmündenden Teilzweige sicher geschrieben wird.

Nachdem feststeht, daß eine (vollständige) Versorgung des Lesezugriffs von `F` auf `d` noch nicht gesichert ist, wird in der Phase II des Verfahrens eine erweiterte DF-Analyse durchgeführt, die zusätzlich Schreibzugriffe von Knoten paralleler Zweige berücksichtigt. Berücksichtigt werden allerdings nur die Schreiberknoten, die Vorgänger von `F` bzgl. Kontroll- und Sync-Kanten sind. Im Schritt 7 des Verfahrens wird mit `G` ein solcher Schreiberknoten ermittelt und sein direkter Nachfolger `H` der Menge *NodeList* hinzugefügt (nachdem zuvor sein Zähler um 1 erhöht wurde). In der 1. Iteration von Schritt 8 ergibt sich, daß `H` sicher versorgt werden kann (`H` besitzt die Eingangssemantik `ONE_Of_ONE` und es gilt: *Counter*(`H`) = 1) und der Knoten wird entsprechend mit der Markierung *Supplied*(`H`) = *T(rue)* versehen (vgl. Abbildung 5-7 c). Daraufhin wird für alle direkten Nachfolger von `H` geprüft, ob sie in der Folgeiteration bewertet werden sollen. Der Knoten `I` muß nicht bewertet werden, da er kein Vorgänger von `F` im KF-Graphen ist. Bleibt als einziger Nachfolger der Zielknoten `D` der einfachen Sync-Kante e = (`H`, `D`, `SOFT_SYNC_E`). Dieser Knoten erfüllt das eingangs dieses Abschnitts vorgestellte OR-Split-Kriterium und kann deshalb mit *SyncSupplied*(`D`) = *T(rue)* markiert werden. Dem entsprechend wird er in der nächsten Iteration des Verfahrens neu mit *Supplied*(`D`) = *T(rue)* markiert und der Zähler seines Nachfolgerknotens `E` um `1` erhöht. Im Anschluß daran kann der OR-Join-Knoten `E` mit *Supplied*(`E`) = *T(rue)* markiert werden, da er über alle einmündenden Zweige versorgt werden kann (*Counter*(`E`) = 2). Nach Terminierung des Verfahrens schließlich erhalten wir die Markierungen, wie sie in Abbildung 5-7 d) dargestellt sind. Alle grau gefärbten Knoten dürfen das Datenelement `d` obligat lesen, insbesondere also auch der Knoten `F`.

Beispiel 5-8 (DF-Analyse bei wohlstrukturierten, azyklischen KF-Graphen mit Sync-Kanten)

Zwei interessante Fälle zeigt Abbildung 5-8. Der in b) dargestellte KF-Graph besitzt gegenüber dem in a) abgebildeten KF-Graphen die zusätzliche Sync-Kante (`C`, `F`, `SOFT_SYNC_E`), ist ansonsten aber identisch. Darüber hinaus besitzen beide Graphen dasselbe DF-Schema (bestehend aus dem Datenelement `d`, den beiden Schreiberknoten `C` und `D` sowie dem Leserknoten `G`). Wird auf beide Graphen das erweiterte *WriterExists*-Verfahren angewendet, so ergeben sich die in den Abbildungen dargestellten Markierungen (die grau dargestellten Flächen umfassen jeweils die Knoten, die Vorgänger von `G` bzgl. Kontroll- und Sync-Kanten sind

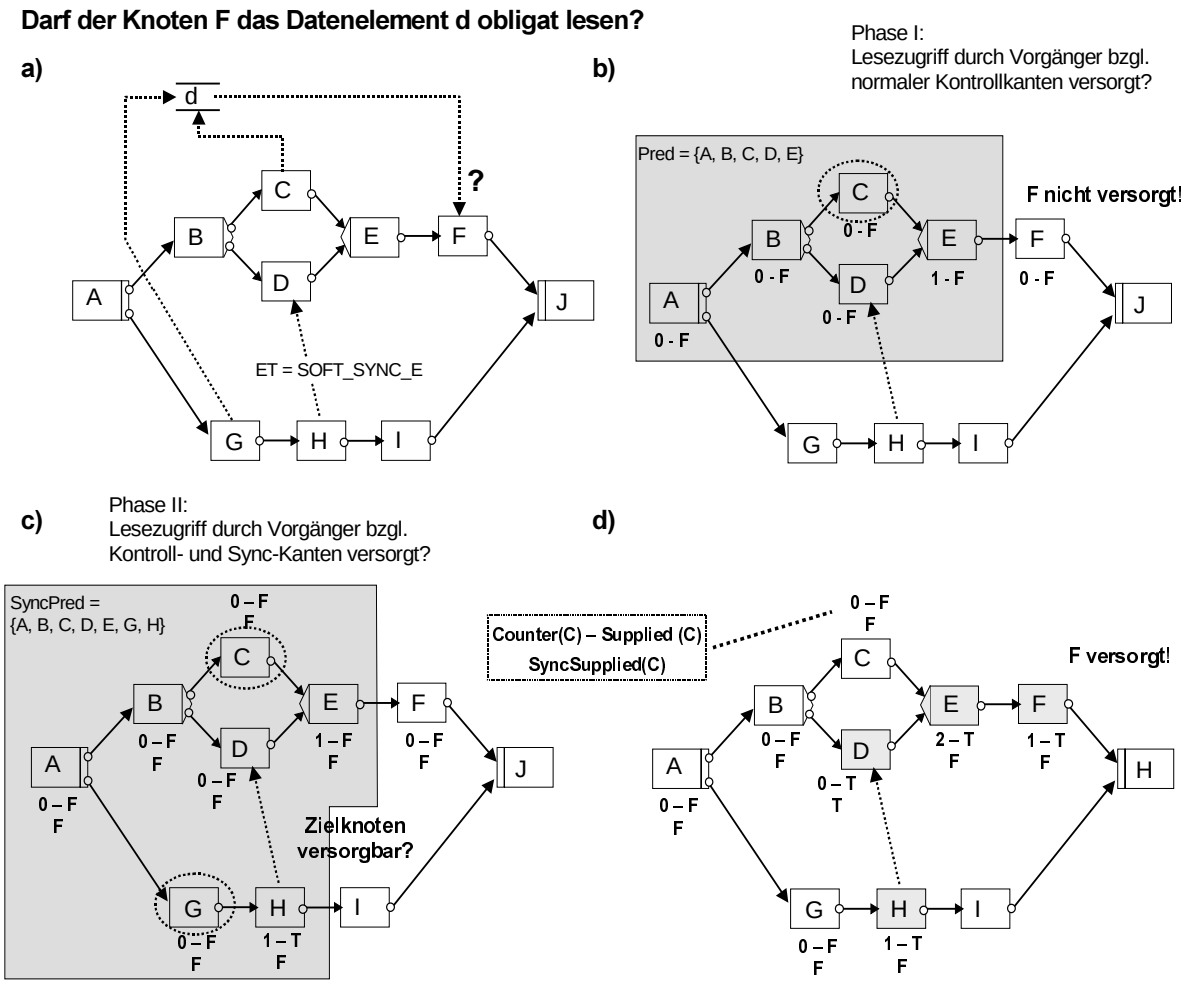


Abbildung 5-7: *DF-Analyse in wohlstrukturierten, azyklischen KF-Graphen mit Sync-Kanten (Unterhalb der Knoten sind jeweils die aktuellen Werte der Attribute Counter, Supplied und SyncSupplied dargestellt. Die grau invertierten Flächen zeigen jeweils die Menge der betrachteten Vorgängerknoten an).*

und deren Schreibzugriffe auf d somit berücksichtigt werden müssen). Demnach kann nur für den KF-Graphen aus Abbildung b) der Lesezugriff von G sicher versorgt werden [*Supplied*(G) = *T(rue)*]. Der Grund dafür ist, daß das OR-Split-Kriterium in Bezug auf den Knoten F nur für den KF-Graphen aus b) erfüllbar ist.

Das vorgestellte Verfahren ist auf beliebig geschachtelte KF-Graphen mit einfachen und strikten Sync-Kanten anwendbar. Es terminiert immer nach endlich vielen Schritten, da die Knotenmenge N endlich ist und in jeder Iteration (→ *Schritte 8* bzw. *3*) immer mindestens ein Aktivitätenknoten $n \in N$ neu mit *Supplied*(n) = *True* markiert wird oder aber der Algorithmus abbricht. Die Komplexität des Verfahrens ist polynomial bzgl. der Kardinalität der Knotenmenge N.

Durch die beschriebene zweistufige Vorgehensweise – Überprüfung der Versorgung durch Vorgänger über normale Kontrollkanten und im Bedarfsfall Überprüfung der Versorgung durch Vorgänger über Kontroll- und Sync-Kanten – erreichen wir zum einen, daß die Mehrzahl der Lesezugriffe mit einfachen Mitteln analysierbar bleibt, zum anderen ermöglicht dieser Ansatz eine vollständige und korrekte Analyse der DF-Schemata gemäß Strukturierungsregel DF-1.

5.4.1.3 Allgemeine KF-Graphen mit Schleifen

Aus der Verwendung von Schleifen resultieren im Hinblick auf die Sicherstellung von Strukturierungsregel DF-1 keine neuen Problemstellungen gegenüber azyklischen KF-Graphen. Der Grund dafür ist, daß wir die Versorgung obligater Eingabeparameter bereits für die erste Iteration einer Schleifenausführung sicherstellen müssen, unabhängig davon, ob in Folgeiterationen Ausgaben vorangegan-

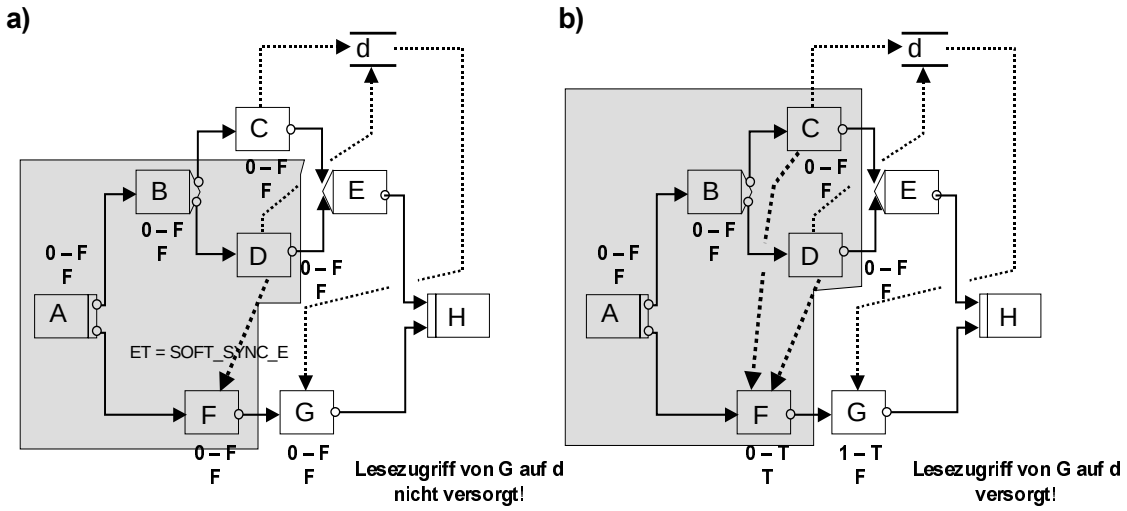


Abbildung 5-8: *Datenversorgung durch kombinierte Anwendung von Sync-Kanten*

gener Schleifendurchläufe zur Parameterversorgung von Aktivitäten herangezogen werden sollen oder nicht. Damit bleibt das in b) vorgestellte Verfahren auch für KF-Graphen mit Schleifen anwendbar, wobei Schleifenrücksprungkanten bei der Analyse nicht berücksichtigt werden müssen.

5.4.2 Ausschluß paralleler Schreibzugriffe auf Datenelementen

In Abschnitt 5.3.2 haben wir mit Strukturierungsregel DF-2 ein Kriterium eingeführt, das sicherstellen soll, daß ein bestimmtes Datenelement aus dem WF-Datenkontext nicht von Knoten unterschiedlicher paralleler Teilzweige geschrieben wird, es sei denn die Reihenfolge dieser Knoten liegt durch die Verwendung von Sync-Kanten eindeutig fest. Die Überprüfung dieser Regel gestaltet sich im Vergleich zur Regel DF-1 relativ einfach und kann auf Basis der in Abschnitt 3.5.2 definierten Funktion *BranchNodes* erfolgen. Zu diesem Zweck wird bei der Modellbildung bzw. -änderung für jede innerhalb einer parallelen Verzweigung hinzugefügte Schreibkante eines Knotens $n \in N$ auf das Datenelement $d \in D$ überprüft, ob es andere Schreibzugriffe auf d durch Aktivitäten paralleler Zweige gibt ($\rightarrow$ *ParallelWriterExists*(*CFS*, *DFS*, *n*, *d*) $\rightarrow$ {True, False}). Um dies festzustellen, werden für jeden Knoten $n^* \neq n$, der das Datenelement d schreibt, die folgenden Schritte durchgeführt:

1. Bestimme mit Hilfe der Funktion $BranchNodes^{CFS}$ die kleinste, die beiden Schreiberknoten n und n^* umschließende Verzweigung: (*split*, *join*):= $BranchNodes^{CFS}(n, n^*)$. Falls es keine solche Verzweigung gibt (d.h. es gilt: *split* = *join* = `UNDEFINED`), gehe zu Schritt 3.

2. Ist der Knoten *join* kein AND-Split–Knoten (d.h. $V_{in}^{join} \neq$ `ALL_Of_ALL`), gehe zu Schritt 3. Andernfalls überprüfe, ob zwischen den Knoten n und n^* eine eindeutige Reihenfolge (über Kontroll- und Sync-Kanten) festliegt. Falls ja, gehe zu Schritt 3, andernfalls gehe zu Schritt 4.

3. Die beiden Knoten folgen im KF-Graphen aufeinander, d.h. Strukturierungsregel DF-2 wird durch den gemeinsamen Schreibzugriff der Knoten n^* und n auf d nicht verletzt.

4. Die beiden Knoten liegen in unterschiedlichen Teilzweigen einer parallelen Verzweigung und ihre Ausführungsreihenfolge ist nicht eindeutig. Regel DF-2 ist nicht erfüllbar, d.h. die Schreibkante von n nach d darf nicht hinzugefügt werden (*ParallelWriterExists*(*CFS*, *DFS*, *n*, *d*) = *True*).

Falls das Verfahren für keinen Schreiberknoten in Schritt 4 abgebrochen hat, setzen wir den Rückgabewert von *ParallelWriterExists*(*CFS*, *DFS*, *n*, *d*) auf *False*, d.h. das Einfügen der Schreibkante von n auf d führt zu keiner Verletzung von Regel DF-2. Zu erwähnen bleibt, daß von diesem Verfahren – im Gegensatz zur Regel DF-1 – auch *optionale* Schreibkanten berücksichtigt werden. Seine Komplexität wird durch die der Funktion *BranchNodes* bestimmt und beträgt demnach $O(|N|^2)$ (vgl. Anhang C). An dieser Stelle sei nochmals erwähnt, daß weiterführende Konzepte, die einen höheren Grad an Parallelität erlauben, nicht Gegenstand dieser Arbeit sind.

5.4.3 Vermeidung überflüssiger Schreiboperationen

In Abschnitt 5.3.3 haben wir mit Regel DF-3 noch eine weitere (nicht zwingende) Strukturierungsregel für DF-Schemata kennengelernt. Sie soll sicherstellen, daß es für zwei aufeinanderfolgende Schreibzugriffe auf einem Datenelement *d* immer mindestens einen potentiellen Lesezugriff auf *d* gibt, der zwischen den beiden Schreiboperationen stattfindet. Zur Überprüfung dieser Regel müssen wir für je zwei aufeinanderfolgende Schreiberknoten x_1 und x_2 von *d* (O.B.d.A. gelte $x_2 \in succ^*(x_1)$) die folgenden Schritte ausführen:

- Bestimme die Menge *M* aller Knoten des KF-Graphen, die Nachfolger von x_1 und Vorgänger von x_2 sind (inkl. dem Knoten x_2) (M:= $(succ^*(x_1) \cap pred^*(x_2)) \cup \{x_2\}$)
- Falls es innerhalb von *M* einen Knoten x_3 gibt, der das Datenelement *d* liest, bleibt die Regel DF-3 erfüllt. Andernfalls wird der von x_1 geschriebene Wert in jedem Fall ungenutzt überschrieben.

5.4.4 Bildung logischer Blockparameter

Das ADEPT-Blockkonzept legt es nahe, sich die Blockstruktur auch für die Modellierung und Analyse von Datenflüssen zunutze zu machen. Hierzu wurden, ergänzend zu dieser Arbeit, in [Blas96] verschiedene Untersuchungen angestellt, auf die wir an dieser Stelle kurz eingehen werden.

Die Grundidee ist, bei der WF-Modellierung für abgeschlossene Kontrollblöcke (vgl. Definition 3-10) sog. *logische Blockparameter* (und die von den Blöcken gelesenen und geschriebenen Datenelemente) zu bestimmen und sie in der Folge bei der Durchführung von Datenflußanalysen zu nutzen – zur Modellier- oder zur Laufzeit. Die Parameter eines Kontrollblocks *B* können dabei aus seiner internen Kontroll- und Datenflußstruktur abgeleitet werden. Für die obligaten Eingabeparameter der in *B* enthaltenen Aktivitäten (oder Sub-Kontrollblöcke) etwa wird geprüft, ob sie durch die Ausgabeparameter vorangehender Aktivitäten (bzw. Kontrollblöcke) des Blocks *B* sicher versorgt werden. Ist dies nicht der Fall, muß der betreffende Eingabeparameter bzw. das von ihm referenzierte Datenelement bereits beim Eintritt in den Block, d.h. bei der Aktivierung seines Startknotens, zwingend versorgt sein (*B* ist abgeschlossen). Dem entsprechend bildet dieser Parameter – logisch gesehen – einen obligaten Eingabeparameter des Blockes. Umgekehrt leiten wir aus den Ausgabeparametern der im Block *B* enthaltenen Aktivitäten eine Menge von logischen Blockausgabeparametern (und die von *B* geschriebenen Datenelemente) ab. Liegt zum Beispiel ein Verzweigungskontrollblock mit OR-Join vor, sind die obligaten Blockausgabeparameter durch die Datenelemente bestimmt, die in allen Teilzweigen oder durch den Split- bzw. Join-Knoten des Blocks obligat geschrieben werden.

Beispiel 5-9: Für das WF-Schema aus Abbildung 5-5 a) ergibt sich, daß das Datenelement d von dem durch (A, H) gebildeten Verzweigungsblock obligat geschrieben wird. Da der Knoten I im dargestellten KF-Graphen ein Nachfolger dieses Blockes ist, folgt unmittelbar, daß sein Lesezugriff auf d sicher versorgt ist.

Im allgemeinen gibt es für die verschiedenen Arten von Kontrollblöcken (Verzweigungsblöcke, Schleifen) wohldefinierte Regeln, auf deren Basis sich die Blockparameter und ihre Eigenschaften (z.B. optionale vs. obligate Parameter) bestimmen lassen (vgl. [Blas96], S. 42 ff.). Die Ableitung der Blockparameter (und der zugeordneten Datenelemente) erfolgt von „innen“ nach „außen“, wodurch bei der Modellbildung auch die Ein- und Ausgabeparameter des WF selbst bestimmt werden können.

Sind die Parameter (bzw. die mit ihnen verknüpften Datenelemente) der Kontrollblöcke eines KF-Graphen einmal ermittelt, kann in der Folge die Komplexität von DF-Analysen zum Teil erheblich reduziert werden. Dies ist insbesondere im Zusammenhang mit dynamischen WF-Änderungen, die unter Umständen auf eine große Anzahl von WF-Instanzen angewendet werden sollen, von Bedeutung, da hier die DF-Analysen zur Laufzeit durchgeführt werden müssen. Soll z.B. ein bedingter Verzweigungsblock gelöscht werden, steht ohne weitere Analysen fest, welche Datenelemente dadurch nicht mehr sicher versorgt sind. Umgekehrt kann für dynamisch hinzugefügte Aktivitäten die Überprüfung ihrer obligaten Lesezugriffe beschleunigt werden, wenn das in Abschnitt 5.4.1 beschriebene *WriterExists*-Verfahren nicht mehr auf der Markierung einzelner Aktivitätenknoten, sondern ganzer Kontrollblöcke basiert. Dies gilt in besonderem Maße, wenn im KF-Graphen (ggf. geschachtelte) Blöcke bedingter Verzweigungen auftreten. Nicht ignoriert werden darf hier allerdings, daß bei der Durchführung struktureller WF-Änderungen die Blockparameter ggf. neu bestimmt werden müssen.

Es besteht somit ein gewisser Trade-Off zwischen Nutzen und Aufwand des beschriebenen Ansatzes. Tendenziell gilt, daß die Bestimmung von Blockparametern vor allem für abgeschlossene Kontrollblöcke und mehrfach geschachtelte Verzweigungsblöcke nützlich ist (vgl. [Blas96]).

5.5 Datenkontextmanagement und Datenflußsteuerung

In den vorangegangenen Abschnitten haben wir uns mit der Strukturierung von DF-Schemata auseinandergesetzt. In diesem Abschnitt skizzieren wir, wie auf Basis solcher Schemata die Verwaltung von Daten aus dem WF-Datenkontext und die Datenflußsteuerung während der WF-Ausführung bewerkstelligt werden. Ziel dabei ist es, ebenso wie bei der Definition der formalen Ausführungssemantik für die Kontrollflußsteuerung, eine klare Semantik für das Lesen und Schreiben von Datenelementen aus dem WF-Datenkontext vorzugeben.

5.5.1 Schreiben von Datenelementen

Bei der Erzeugung einer neuen (Sub-)WF-Instanz (vgl. Abschnitt 4.3) wird jedem Datenelement aus dem Datenkontext initial der Wert *NULL* (oder ein konstanter Default-Wert) zugewiesen. Das bedeutet, daß von Aktivitäten des WF noch keine entsprechenden Datenwerte geschrieben wurden[22]. Schreibzugriffe auf dem Datenkontext (bzw. auf Datenelementen) erfolgen beim Start der (Sub-) WF-Instanz durch die Eingabeparameter des WF oder nach der Beendigung von Aktivitäten durch deren Ausgabeparameter. Jedes Datenelement verwaltet verschiedene Versionen eines Datenobjektes, die durch das mehrmalige Schreiben verschiedener Aktivitätenknoten oder eines Aktivitätenknotens innerhalb verschiedener Schleifeniterationen während der (Sub-)WF-Ausführung entstehen. Das bedeutet, daß jeder Schreibzugriff auf das Datenelement immer eine zusätzliche Version des Datenobjektes erzeugt, es aber niemals überschreibt[23]. Dies wird notwendig, um das kontextabhängige Lesen von Datenelementen (siehe unten) und das korrekte Zurücksetzen des WF (siehe Kap. 6) zu ermöglichen.

Die verschiedenen Versionen eines Datenelementes werden persistent durch die Datenverwaltungskomponente des ADEPT-WfMS verwaltet. Dabei wird jede Version auf Ebene eines bestimmten WF durch die Zuordnung des Schreiberknotens n, seines aktuellen Iterationszähler It^n (vgl. Abschnitt 4.2) sowie die Angabe, ob der Schreibzugriff durch den Knoten selbst oder durch vor- bzw. nachgeschaltete Aktionen erfolgt ist, eindeutig identifiziert. Zur Repräsentation der Versionen eines Datenelementes verwenden wir in der Folge eine geordnete Liste, deren Elemente entsprechend der zeitlichen Abfolge der Schreibzugriffe angeordnet sind (O.B.d.A. stehe der aktuellste Wert am Listenanfang).

Definition 5-3 (WF-Datenkontext, Datenelement-Historie):

Sei durch $CFS = (N, E, D, ...)$ ein allgemeiner KF-Graph mit zugeordnetem DF-Schema *DFS* gegeben. $CFS_{instance}$ bezeichne den Ausführungsgraphen einer davon abgeleiteten (Sub-)WF-Instanz.

Dann ist der *Datenkontext* der Instanz durch eine Abbildung *CtxtValues* festgelegt, die jedem Datenelement $d \in D$ eine geordnete Liste $CtxtValues^d$ zuweist, deren Einträge die verschiedenen Versionen des verwalteten Datenobjektes enthalten. Wir bezeichnen $CtxtValues^d$ auch als *Historie des Datenelements d*. Ein einzelner Eintrag v einer solchen Historie besitzt die folgende Struktur:

$$v = (nodeId^v, nodeIt^v, val^v, mode^v)$$

Dabei ist val^v ein von der Aktivität $nodeId^v$ ($mode^v$ = `MAIN`) oder einem vor-/nachgeschalten Hilfsdienst ($mode^v$ = `PRE`/`POST`) in der Iteration $nodeIt^v$ geschriebener Datenwert von d.

[22] Ob ein bestimmtes Datenelement nullwertig ist oder nicht, kann während der Laufzeit einfach festgestellt werden. Beispielsweise kann die Ausführung vor- oder nachgeschalteter Hilfsdienste (zur Versorgung von Parameterdaten) davon abhängig gemacht werden, ob bestimmte Datenelemente den Wert *NULL* besitzen oder nicht. Für optionale Eingaben kann die aufgerufene Komponente über entsprechende Indikatorvariablen feststellen, ob sie nullwertig sind oder nicht.

[23] Wir verfolgen hier einen ähnlichen Ansatz wie im ConTracts-Modell [WäRe92, ReSc95]. Aufgrund der höheren Ausdrucksmächtigkeit von ADEPT gestaltet sich das kontextabhängige Lesen von Daten (siehe nachfolgenden Abschnitt) allerdings schwieriger als bei ConTracts. Des weiteren stellt ADEPT sicher, daß obligate Lesezugriffe stets versorgt sind.

5.5.2 Lesen von Datenelementen

Zur Versorgung von Aktivitäteneingaben zur Laufzeit müssen Datenelemente aus dem WF-Datenkontext gelesen werden. Die Bereitstellung einer entsprechenden Leseoperation stellt spezielle Anforderungen an die Datenverwaltungskomponente des WfMS, da die dem Kontext des Leserknotens entsprechenden (laufzeitabhängigen) Daten geliefert werden müssen. Klammert man parallele Verzweigungen und parallele Verzweigungen mit finaler Auswahl zunächst aus, so wird immer der aktuellste, d.h. der zuletzt geschriebenen Wert des Datenelementes gelesen. Dies ist jedoch in Verbindung mit den beiden genannten Verzweigungsarten nicht immer sinnvoll. Um eine klare Lesesemantik zu erhalten, muß die Aktualität der gelesenen Datenversion dadurch eingeschränkt werden, daß ihr Schreiberknoten im Kontrollfluß vor dem Leserknoten liegt. Die Existenz eines solchen Schreibers ist durch die Regel DF-1 sichergestellt. Praktisch bedeutet diese Einschränkung, daß Schreibzugriffe auf ein Datenelement *d*, die innerhalb eines Teilzweiges einer Parallelverzweigung stattfinden, im allgemeinen nur für nachfolgende Knoten (bzgl. Kontroll- und Sync-Kanten) sichtbar sind, nicht dagegen für „echt“ parallele Knoten. Eine Sonderrolle kommt, wie wir noch sehen werden, den Lese- und Schreibzugriffen der Aktivitäten eines Schleifenkörpers zu.

Bei Parallelverzweigungen mit finaler Auswahl darf ein bestimmtes Datenelement auch innerhalb mehrerer Teilzweige geschrieben werden. In diesem Fall arbeiten die Aktivitäten dieser Zweige auf unterschiedlichen Versionen des Datenelementes, was aufgrund der Semantik dieses Konstrukts (vgl. Abschnitt 3.3.2.2) wünschenswert ist. Nach Beendigung der Verzweigung, d.h. nach Auswahl des Siegerzweiges, bleiben für nachfolgende Knoten lediglich die Schreibzugriffe dieses Zweiges erhalten, wohingegen die Schreiboperationen anderer Teilzweige zurückgesetzt werden. Für Lesezugriffe, die innerhalb eines Schleifenkörpers stattfinden, muß ein weiterer Aspekt berücksichtigt werden. Ein in einer bestimmten Schleifeniteration von einem Knoten n_{write} geschriebener Werte eines Datenelements, darf in späteren Iterationen auch von Knoten gelesen werden, die innerhalb desselben Schleifenkörpers liegen und die Vorgänger von n_{write} (inkl.) bzgl. normaler Kontroll- und Sync-Kanten sind. Dasselbe gilt in Bezug auf Lesezugriffe durch Knoten, die im KF-Graphen parallel zu n_{write} liegen.

Basierend auf diesen Überlegungen geben wir nun die Funktion *ReadDataElement* an, die für den Lesezugriff eines Knotens *n* auf ein Datenelement *d* den kontextabhängigen Wert liefert. Voraussetzung für ihre Anwendung ist, daß es im DF-Schema eine Lesekante von *n* nach *d* gibt. Des weiteren muß sich der Knoten *n* im Zustand `ACTIVATED` bzw. `RUNNING` befinden. Sind diese beiden Bedingungen erfüllt, stellt die Regel DF-1 für obligate Lesezugriffe sicher, daß das gelesene Datenelement zuvor geschrieben wurde, d.h. ein entsprechender Wert auch vorliegt. Für optionale Parameter liefert die Funktion *ReadDataElement* den Wert `NULL`, wenn kein entsprechender Wert geschrieben wurde.

Im folgenden bezeichne $pred^*(n)$ wieder die Menge aller Vorgängerknoten, von denen aus der Knoten *n* über eine Kantenfolge (bestehend aus Kontroll- und Sync-Kanten) erreicht werden kann. Die Menge $l_pred^*(n) \supseteq pred^*(n)$ umfasse zusätzlich diejenigen Knoten, die Vorgänger von *n* über Kantenfolgen sind, die neben Kontroll- und Sync-Kanten auch Schleifenrücksprungkanten umfassen dürfen. Das bedeutet, daß $l_pred^*(n)$ möglicherweise auch Knoten enthält, die Nachfolger von *n* bzgl. normaler Kontroll- und Sync-Kanten sind, die aber innerhalb desselben Schleifenkörpers liegen. Formal ist die Menge $l_pred^*(n)$ wie folgt definiert (vgl. hierzu Abschnitt 3.4.2):

$$l_pred^*(n) := var_pred^*(n, \{\texttt{CONTROL_E}, \texttt{SOFT_SYNC_E}, \texttt{STRICT_SYNC_E}, \texttt{LOOP_E}\})$$

Für den KF-Graphen aus Abbildung 4-9 gilt zum Beispiel: $l_pred^*(\texttt{B}) = \{\texttt{L}_S, \texttt{A}, \texttt{B}, \texttt{C}\}$. Insbesondere kann der betrachtete Knoten (im Beispiel `B`) selbst in der Vorgängermenge enthalten sein. (Die Ausgaben eines Knotens *n* können in einer Folgeiteration Eingaben desselben Knotens sein.) Wir formulieren nun die Funktion *ReadDataElement* für das kontextabhängige Lesen von Datenelementen:

Algorithmus 5-2 (ReadDataElement) // *Kontextabhängiges Lesen von Datenelementen*

function ReadDataElement($CFS_{instance}$, n_{read}, d) → value

input

$CFS_{instance}$:	Ausführungsgraph der WF-Instanz
n_{read}:	Knoten, von dem aus eine Lesekante nach d führt
d:	Gelesenes Datenelement

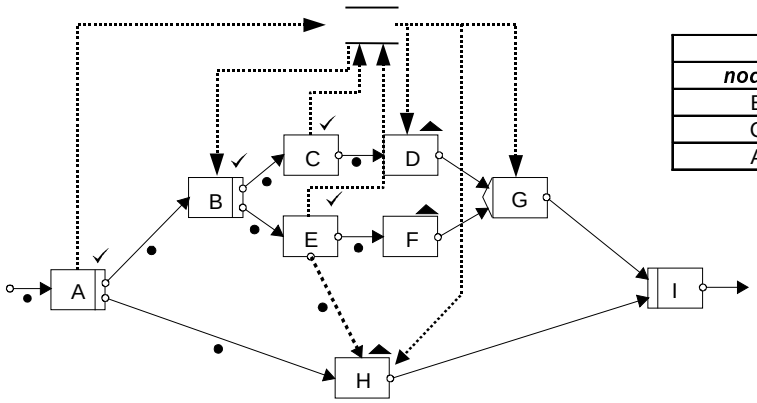
```
    CtxtValues^d:  Historie des Datenelements d (aktuellster Wert am Listenanfang)
output
    value:         Kontextabhängiger Wert des gelesenen Datenelementes
begin
    // Bestimmung der relevanten Vorgängermengen (siehe oben)
    Pred:=          pred*(n_read) ∪ {n_read};
    LoopPred:=      l_pred*(n_read) ∪ {n_read};

    // Historie des Datenelements, beginnend mit dem aktuellsten Schreibzugriff – CtxtValues^d ist
    // geordnet – durchlaufen, bis entweder der gesuchte Wert ermittelt ist oder aber kein weiterer
    // Listeneintrag vorliegt.
    forall v ∈ CtxtValues^d do
        n_write:= nodeId^v;        // Knoten der den aktuell betrachteten Wert geschrieben hat

        // Falls der Schreiberknoten n_write des Historieneintrags v ein Vorgänger von n_read bzgl. normaler
        // Kontrollkanten und Sync-Kanten ist, darf n_read den von n_write geschriebenen Wert val^v lesen, mit
        // einer Ausnahme: Gehört n_write einem Teilzweig einer noch nicht beendeten parallelen Ver-
        // zweigung mit finaler Auswahl an, der den Knoten n_read nicht enthält, darf der Wert von n_read
        // nicht gelesen werden (*).
        if (n_write ∈ Pred ) then          // n_write ist Vorgänger von n_read bzgl. Kontroll- und Sync-Kanten
            (split, join):= BranchNodes(CFS_instance, n_read, n_write);
            if (split = UNDEFINED)  then   // Knoten liegen nicht in verschiedenen Teilzweigen
                return value:= val^v                // eines Verzweigungskontrollblocks
            else                                    // Fall (*) muß ausgeschlossen werden
                // Es muß geprüft werden, ob n_write einem Teilzweig einer Parallelverzweigung mit
                // finaler Auswahl angehört, welcher den Knoten n_read nicht enthält. Für diesen Fall
                // darf n_read den von n_write geschriebenen Wert nur lesen, wenn der Join-Knoten der
                // Verzweigung bereits aktiviert wurde, d.h. n_write dem gewählten Siegerzweig ange-
                // hört, so daß der Schreibzugriff von n_write auf d nicht mehr zurückgesetzt wird
                // (vgl. hierzu das Beispiel aus Abbildung 5-9 und Kapitel 6)

                // Menge M der zwischen n_write (inkl.) und join (exkl.) liegenden Knoten ermitteln
                // (bei Berücksichtigung normaler Kontrollkanten) – Da n_read kein Nachfolger von
                // n_write bzgl. normaler Kontrollkanten ist, gehört dieser Knoten der Menge M nicht an
                M:= (c_succ*(n_write) ∩ c_pred*(join)) ∪ {n_write};

                if  (Es gibt keinen noch nicht aktivierten Knoten n ∈ M, für den gilt:
                    V_in^n = ONE_Of_ALL and V_out^split(n) = ALL_Of_ALL and split(n) ∉ M) then
                            return value:= val^v
                endif
            endif
        endif
        if (n_write ∈ LoopPred) then    // n_write und n_read befinden sich innerhalb (mind.) einer Schleife
            // Prüfen, ob Schreibzugriff in einem früheren Schleifendurchlauf erfolgte oder aber
            // innerhalb eines parallelen Teilzweiges derselben Iteration (In letzterem Fall ist das
            // Lesen nicht zulässig). Für die Überprüfung werden Informationen aus der
            // Ablaufhistorie (siehe hierzu Kapitel 6) herangezogen.
            if (  Es gibt in der Ablaufhistorie einen Eintrag für die Beendigung der i.ten Ausführung eines
                Schleifenendknotens L_E ∈ c_succ*(n_write) ∩ c_succ*(n_read) und es gilt: Die nodeIt^v-te
                Ausführung von n_write, in welcher der Datenwert val^v geschrieben wurde, erfolgte
                gemäß der Ablaufhistorie vor der i.ten Ausführung des Knotens L_E ) then
                        return value:= val^v
            endif
        endif
    done
    return value:= NULL        // Kein passender Historieneintrag (→ bei optionalem Zugriff möglich)
end  // ReadDataElement
```



Historie von d		
nodeId	**nodeIt**	**val**
E	1	*value3*
C	1	*value2*
A	1	*value1*

x	ReadDataElement(x,d) →
H	*value1*
D	*value2*
G	*value2* oder *value3*[1]

[1]abhängig vom gewählten Siegerzweig

Abbildung 5-9: *Datenkontextmanagement in ADEPT*

Ein Beispiel für die Anwendung dieser Funktion zeigt Abbildung 5-9. – Die Komplexität des vorgestellten Leseverfahrens ist polynomial bzgl. der Kardinalität $|N|$ der Knotenmenge N. Einflußfaktoren sind die Kardinalität der Historie des gelesenen Datenelements sowie die Größe und Gestaltung der Ablaufhistorie. Die angegebenen Überprüfungen können durch verschiedene Maßnahmen, z.B. durch Unterstützung von Direktzugriffen auf abgeschlossene Schleifenendknoten, optimiert werden.

5.6 Zusammenfassung

Die explizite, formale Beschreibung von Datenflüssen zwischen Aktivitäten ist eine wichtige Voraussetzung für einen fehlerfreien Ablauf zur Laufzeit. In diesem Kapitel haben wir grundlegende Aspekte der Modellierung, Analyse und Validation von Datenflüssen behandelt. Darauf basierend haben wir gezeigt, wie die Steuerung von Datenflüssen und das Datenkontext-Management zur Laufzeit erfolgen. Die vorgestellten Konzepte garantieren auf Modellebene, daß zur Ausführungszeit keine Aktivitätenprogramme mit fehlenden Eingaben gerufen werden können. Dabei wird nicht nur die Versorgung obligater Eingaben von Anwendungskomponenten gewährleistet, sondern z.B. auch die Bereitstellung WF-relevanter Daten für die Auswertung von Verzweigungsprädikaten. Darüber hinaus werden Dateninkonsistenzen bzw. -verluste durch paralleles Schreiben von Datenelementen bereits auf Modellebene ausgeschlossen. (Wie erwähnt, sind weiterführende Konzepte, die einen höheren Grad an Parallelität erlauben, denkbar. Sie werden jedoch im Rahmen dieser Arbeit nicht behandelt.) Diese Aspekte sind nicht nur für die Modellierung relevant, sondern auch für die dynamische Änderung von WF-Instanzen zur Laufzeit. Insgesamt schafft der beschriebene Ansatz erst die Grundlage dafür, in Bezug auf beabsichtigte WF-Änderungen erkennen zu können, welche begleitenden Anpassungen vorgenommen werden müssen, um anschließend fehlerfrei mit dem Ablauf fortfahren zu können.

Für die Überprüfung der Parameterversorgung von Aktivitäten in blockbasierten KF-Graphen (ohne/mit Sync-Kanten) haben wir den *WriterExists*-Algorithmus entwickelt. Dabei handelt es sich um einen Vertreter einer Familie von Verfahren für die effiziente Durchführung von DF-Analysen. Durch geringfügige Modifikationen sowie durch geeignete Parametrisierung dieses Algorithmus lassen sich weitere Verfahren zur Untersuchung verwandter Problemstellungen gewinnen, wie z.B.

- An welche Datenelemente aus dem WF-Datenkontext können Eingabeparameter einer bestimmten Aktivität potentiell angeknüpft werden (vgl. Kapitel 7)?
- Welche Aktivitätenknoten dürfen ein bestimmtes Datenelement *d* obligat lesen?
- Sind alle obligaten Ausgaben des WF bei seiner Terminierung sicher versorgt?
- Sind alle Daten, die zur Auswertung eines Verzweigungsprädikates oder einer Schleifenbedingung benötigt werden, immer versorgt?
- Welche obligaten Lesezugriffe eines DF-Schemas können infolge des Löschens eines Knotens (inkl. seiner Datenkanten) nicht mehr versorgt werden (vgl. Kapitel 7)?

Die Beantwortung solcher und vergleichbarer Fragestellungen ist in ADEPT sowohl bei der Bildung als auch bei der Abänderung von WF-Modellen von zentraler Bedeutung.

6 Behandlung planbarer Abweichungen

Bei der Realisierung vorgangsbasierter Anwendungssysteme müssen bereits zur Modellierungszeit Vorkehrungen getroffen werden, um zur Laufzeit eine flexible Gestaltung der WF-Ausführung zu ermöglichen. Ein adaptives WfMS sollte, neben der dynamischen Abänderung von WFs, in gewissem Umfang auch die Vormodellierung von Ausnahmen und ihrer Behandlung unterstützen. Hinter dieser Forderung steckt die Erkenntnis, daß Abweichungen von „normalen“ Ausführungsreihenfolgen, wie das Überspringen von Aktivitäten oder das kontrollierte Zurücksetzen des WF, in vielen Fällen a priori bekannt sind und deshalb bei der Modellbildung berücksichtigt werden können [Meye96, RHD98]. Zum einen läßt sich dadurch ein höheres Maß an Flexibilität bei der Bearbeitung von Prozessen erzielen, zum anderen kann damit der Umfang der vergleichsweise teureren Modifikationen auf WF-Instanzebene und damit verbundener Konsistenzprüfungen von vornherein reduziert werden.

Für den Umgang mit planbaren Abweichungen bzw. Ausnahmen ergeben sich zwei generelle Probleme: Ausnahmen müssen erkannt, d.h. dem System mitgeteilt (*Ausnahmedetektion*), und ihre Auswirkungen beseitigt werden (*Ausnahmebehandlung*). Notwendige Voraussetzung für die Vormodellierung von Ausnahmen ist zum einen die Vorhersehbarkeit des Kontexts, in dem sie auftreten, sowie der zu ihrer Behandlung notwendigen Aktionen, zum anderen die ausreichende Ausdrucksmächtigkeit des verwendeten WF-Metamodells. Auf Basis der bisher eingeführten Modellkonstrukte läßt sich die Mehrzahl an Arbeitsabläufen und die ihnen zugrundeliegende Variabilität bereits adäquat abbilden. Allerdings sind die angebotenen Modellelemente noch nicht ausdrucksmächtig genug, um alle Arten ausnahmebedingter Abweichungen vormodellieren zu können. Eine wichtige Anforderung aus Sicht des *WF-Modellierers* ist die getrennte Beschreibbarkeit „normaler“, d.h. präferierter Ausführungsreihenfolgen von ausnahmebedingten Abweichungen. Darüber hinaus darf durch die Einbeziehung von Ausnahmebehandlungen nicht die Übersichtlichkeit der Modelle leiden oder sich der Aufwand für ihre Erstellung wesentlich erhöhen. Aus Sicht des *Endanwenders* ist die Unterscheidbarkeit normaler Aktivitäten von ausnahmebedingten Aktionen, deren Ausführung eine Abweichung vom Standardablauf bedeutet, eine wichtige Voraussetzung dafür, daß sie die Konsequenzen ihrer Interaktionen mit dem System besser verstehen und nachvollziehen können. Dieser Aspekt wird in heutigen Ansätzen nahezu ignoriert, ist aber gerade im Hinblick auf die Bedienung des Systems durch "normale" Endanwender, die keine Computerexperten sind, von zentraler Bedeutung [Saue96]. Insgesamt ist die Art und Weise, in der planbare Ausnahmen vormodelliert und die damit verbunden Handlungsmöglichkeiten dem Endanwender angeboten werden, entscheidend für die Praxistauglichkeit entsprechender Konzepte.

In diesem Kapitel diskutieren wir Erweiterungen des ADEPT-Basismodells, welche die Vormodellierung bestimmter Arten von Ausnahmen erlauben sollen. Wir konzentrieren uns dabei auf ausnahmebedingte Vorwärts- und Rückwärtssprünge in KF-Graphen und gehen auf andere Mechanismen, wie die Behandlung nur „ungefähr“ beschreibbarer WF-Anteile, nur am Rande ein. Ebenso klammern wir Ausnahmen, die andere WF-Aspekte betreffen (z.B. Nichtauflösbarkeit von Rollen, Nichtverfügbarkeit von Ressourcen usw.), weitgehend aus. Ihre Behandlung wirft überwiegend orthogonale Fragestellungen auf (vgl. [Kubi98]). Abweichungen vom „normalen“ Kontrollfluß werden entweder durch den Benutzer oder durch vorhersehbare Ereignisse, z.B. das Scheitern einer Aktivitätenbearbeitung, initiiert. Erfahrungsgemäß läßt sich durch Vorwärts- und Rückwärtssprünge unterschiedlicher Semantik bereits die Mehrzahl der auf Ebene der Kontrollflußsteuerung vorhersehbaren Ausnahmen behandeln [Meye96, StMi95]. Darüber hinaus eignen sich entsprechende Konzepte gut, um grundlegende Aspekte der Vormodellierung von Ausnahmen und ihrer (automatischen) Behandlung zur Laufzeit diskutieren zu können. Dazu gehört auch die Klärung der Fragestellung, wo die Grenzen einer solchen Vormodellierung liegen. Beispielsweise sind aufgrund nicht-deterministischer Ausführungssequenzen nicht alle zur Laufzeit möglichen Rückwärtssprünge im Kontrollfluß sinnvoll vormodellierbar. Um auch solchen Fällen begegnen zu können, bietet ADEPT vergleichbare Sprungoperationen für den dynamischen Fall an (siehe Kapitel 7). Dadurch wird ein direkter Vergleich zwischen vormodellierten Abweichungen und Ad-hoc-Abweichungen zur Laufzeit möglich.

Dieses Kapitel gliedert sich wie folgt: In Abschnitt 6.1 behandeln wir vormodellierbare *Vorwärtssprünge*. Ein Beispiel ist das Überspringen einer Gruppe von Aktivitäten mit bzw. ohne Nachholen der übersprungenen Schritte. Wir beschreiben Vorwärtssprünge unterschiedlicher Semantik sowie ihre Umsetzung im ADEPT-Basismodell. Um entsprechende Ausführungsalternativen dem Anwender adäquat präsentieren zu können, führen wir als weiteres Knotenattribut *Prioritäten* und als weiteren

Kantentyp *Priorisierungskanten* ein. (Diese beiden Beschreibungselemente wurden, ebenso wie die nachfolgend behandelten *Fehlerkanten*, in Kapitel 3, bei der Definition von KF-Graphen, bereits berücksichtigt.) Durch Verwendung von Prioritäten und Priorisierungskanten können präferierte von ausnahmebedingten Ausführungsreihenfolgen unterschieden werden. In Abschnitt 6.2 diskutieren wir Konzepte zur Modellierung und Behandlung ausnahmebedingter *Rückwärtssprünge* in KF-Graphen. Dabei unterscheiden wir zwischen planbaren, d.h. vormodellierbaren Rücksprüngen sowie Ad-hoc-Rücksetzoperationen. In beiden Fällen wird das kontrollierte Zurücksetzen des WF unter Wahrung bestimmter Integritätseigenschaften notwendig. Dabei stellen wir im wesentlichen dieselben Korrektheits- und Konsistenzforderungen wie im Fall von Vorwärtssprüngen. Zur Vormodellierung planbarer Rücksprünge führen wir als weiteren Kantentyp *Fehlerkanten* ein, die – wie wir zeigen werden – eine andere Ausführungssemantik besitzen als Schleifenrücksprungkanten. Mit der Vorstellung von Priorisierungs- und Fehlerkanten ist die Beschreibung der Konstrukte des ADEPT-Basismodells vollständig.

6.1 Behandlung von Vorwärtssprüngen

In Ausnahmesituationen kann es erforderlich werden, die Bearbeitung einzelner oder mehrerer Aktivitäten zu überspringen bzw. Aktivitäten vorzeitig auszuführen. Damit verbunden sind Abkürzungen (sog. *Shortcuts*) im Ausführungsgraphen. Entsprechende Abweichungen sind Ausdruck der Variabilität von Arbeitsabläufen und treten in der Praxis häufig auf. Um so wichtiger ist es, Konzepte bzw. Mechanismen zu ihrer Behandlung nicht nur auf WF-Ausführungsebene, sondern – soweit sinnvoll und möglich – auch bereits bei der Modellbildung vorzusehen. Ist beispielsweise a priori bekannt, daß die Bearbeitung einer bestimmten Aktivität in Ausnahmesituationen vorgezogen werden können muß, so sollte dieser Sachverhalt auch im WF-Modell adäquat abbildbar sein. Dadurch könnte das WfMS in die Lage versetzt werden, die betreffende Aktivität dem verantwortlichen Benutzer ggf. vorzeitig als „Ausnahmeschritt“ zur Bearbeitung anzubieten, ohne daß er oder sie aufwendige Status- und/oder Strukturänderungen des KF-Ausführungsgraphen vornehmen muß, wie sie im Fall von Ad-hoc-Abänderungen des WF notwendig werden. Im Kern sollten dabei dieselben Mechanismen und Regeln für die Ausführung von WF zur Anwendung kommen wie bisher. Wichtig ist allerdings, daß die Zielaktivität eines möglichen Vorwärtssprungs dem Anwender in einer anderen Form präsentiert wird als es für „normale“ Arbeitsschritte der Fall ist. Zu diesem Zweck muß es möglich sein, bei der Modellbildung zwischen präferierten Ausführungsreihenfolgen und ausnahmebedingten Abweichungen zu unterscheiden. Im nun folgenden Abschnitt gehen wir auf die hierzu notwendigen Erweiterungen des ADEPT-Basismodells ein. Anschließend behandeln wir planbare Vorwärtssprünge (sog. *Shortcuts*) unterschiedlicher Semantik (Überspringen mit/ohne Nachholen der übersprungenen Schritte) und diskutieren ihre Umsetzung im ADEPT-Basismodell.

6.1.1 Knotenprioritäten und Priorisierungskanten

KF-Graphen, die auf Basis der bisher eingeführten Konstrukte beschrieben werden, legen die Aktivitäten eines Arbeitsablaufs sowie deren „normale“ Ausführungsreihenfolgen und -bedingungen fest. In diesem Abschnitt diskutieren wir Erweiterungen des ADEPT-Basismodells, die darauf zielen, präferierte Ausführungsreihenfolgen von ausnahmebedingten Abweichungen (z.B. Vorwärtssprüngen) unterscheiden zu können. Sie beinhalten zum einen die Zuordnung von Ausführungsprioritäten zu Aktivitätenknoten, zum anderen die Einführung eines weiteren Kantentyps zur Änderung solcher Ausführungsprioriäten während der Laufzeit. Durch kombinierte Anwendung dieser beiden Möglichkeiten sowie durch Verwendung der Standardkonstrukte des ADEPT-Basismodells lassen sich Vorwärtssprünge adäquat vormodellieren.

6.1.1.1 Knotenprioritäten

Um zur Ausführungszeit unterscheiden zu können, ob eine WF-Aktivität dem Benutzer als normaler Arbeitsschritt oder als Ausnahmeschritt angeboten werden soll, führen wir Knotenprioritäten ein. Jedem Aktivitätenknoten $n \in N$ wird eine der beiden Ausführungsprioritäten $NP(n)$ = `REGULAR` oder $NP(n)$ = `EXCEPTIONAL` zugewiesen, wobei als Voreinstellung der Wert `REGULAR` verwendet wird. Soll eine Aktivität bei ihrer Aktivierung dem Benutzer als Ausnahmeschritt angeboten werden, so muß ihrem Knoten bei der Modellierung die Priorität `EXCEPTIONAL` zugeordnet werden. Abgesehen da-

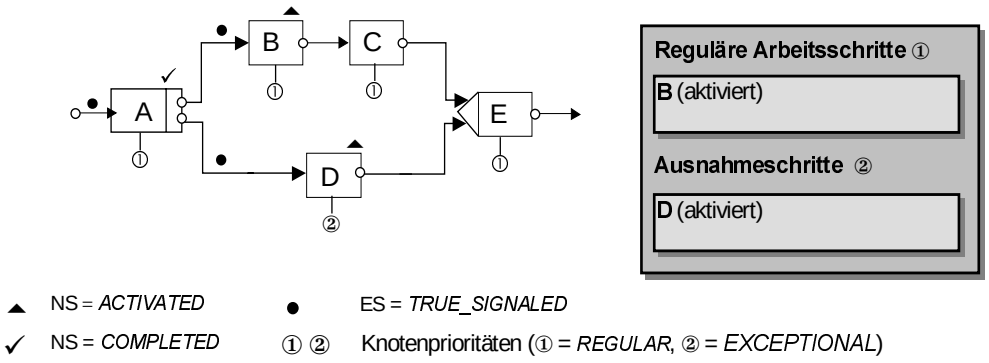


Abbildung 6-1: *Verwendung von Knotenprioritäten in ADEPT*

von erfolgt die Aktivierung und Ausführung der Aktivität nach denselben Regeln wie im Fall regulärer Arbeitsschritte. Für die Semantik von Knotenprioritäten gilt somit die folgende einfache Festlegung:

> Sei $CFS_{instance}$ = (CFS, NS, ES, It) ein KF-Ausführungsgraph. Dann gilt für aktivierte Knoten $n \in N$ mit $NS(n)$ = `ACTIVATED`:
>
> Besitzt der Knoten n die Priorität $NP(n)$ = `REGULAR`, so wird er wie ein normaler Arbeitsschritt behandelt. Gilt dagegen $NP(n)$ = `EXCEPTIONAL`, so wird die betreffende Aktivitäteninstanz als Ausnahmeschritt gekennzeichnet und kann daher an der Benutzerschnittstelle entsprechend hervorgehoben werden. Die Ausführung einer Ausnahmeaktivität bedeutet semantisch eine Abweichung vom „Standardablauf", ist aber ansonsten mit keinen weiteren Konsequenzen verbunden.

Die Art und Weise, in der Ausnahmeschritte an der Benutzerschnittstelle präsentiert werden, bleibt dem Anwendungsentwickler überlassen. Denkbar ist hier die Anwendung von Techniken wie dem Ein- bzw. Ausblenden von Ausnahmeschritten, der Darstellung von Arbeitslisteneinträgen in unterschiedlichen Farben oder die Präsentation von normalen Arbeitsschritten und Ausnahmeschritten in unterschiedlichen Arbeitslisten. Beispiele hierzu finden sich in [Saue96]. Wichtig ist an dieser Stelle der Hinweis, daß der Anwendungsentwickler, d.h. der Programmierer von WfMS-Klienten-Anwendungen, mit einfachen Mitteln feststellen kann, ob es sich bei einer aktivierten Aktivität um einen normalen Arbeitsschritt oder einen Ausnahmeschritt handelt.

Die Verwendung von Knotenprioritäten erweist sich in Verbindung mit Parallelverzweigungen bei finaler Auswahl als nützlich. Hier kommt es zur gleichzeitigen Aktivierung mehrerer paralleler Ausführungszweige, wobei für das Fortschreiten des Kontrollflusses immer nur die Ausführung eines Teilzweiges notwendig ist. Dadurch lassen sich, bei kombinierter Anwendung mit Knotenprioritäten, präferierte Ausführungszweige von „Ausnahmepfaden“ unterscheiden. Ein Beispiel zeigt Abbildung 6-1: Im dargestellten Ausführungsgraphen sind die beiden Aktivitäten `B` und `D` gleichzeitig aktiviert. Infolge der verwendeten Knotenprioritäten wird `B` als regulärer Arbeitsschritt angeboten, während `D` als Ausnahmeschritt behandelt wird. Dementsprechend wird als Ausführungssequenz die Bearbeitung der Schritte `A`, `B`, `C` und `E` in der angegebenen Reihenfolge präferiert. Prinzipiell kann aber auch Aktivität `D` anstelle von `B` und `C` ausgeführt werden[24], etwa wenn bestimmte Ausnahmesituationen die Ausführung dieser Aktivität zwingend erforderlich machen. Wir werden später zeigen, daß sich nach demselben Prinzip auch bestimmte Arten von Vorwärtssprüngen modellieren lassen.

6.1.1.2 Priorisierungskanten

Abhängig vom Status einer WF-Instanz muß es möglich sein, eine aktivierte Aktivität das eine Mal als Ausnahmeschritt und das andere Mal als regulären Arbeitsschritt zu behandeln. Soll zum Beispiel die

[24] Der Schritt *D* muß nicht notwendigerweise vom selben Bearbeiter wie `B` ausgeführt werden. Im allgemeinen werden Bearbeiter für Ausnahmeschritte wieder auf Basis von Rollen ermittelt. Prinzipiell kann deshalb die Bearbeitung einer Ausnahmeaktivität auch durch Akteure erfolgen, die im Normalfall von der Kontrolle des WF ausgeschlossen sind.

Bearbeitung der Aktivität X in Ausnahmefällen vorzeitig erfolgen können, im Normalfall allerdings erst nach Abschluß bestimmter anderer Schritte, so stellt die Ausführung des Knotens X zu gewissen Zeitpunkten der WF-Ausführung eine Abweichung vom Standardablauf dar, während sie zu einem späteren Zeitpunkt regulären Charakter besitzt und dementsprechend als normaler Arbeitsschritt angeboten werden sollte. Solche Sachverhalte lassen sich allein auf Basis statischer Knotenprioritäten noch nicht modellieren. Was wir zusätzlich benötigen ist die Möglichkeit, die Ausführungspriorität von Aktivitäten bei Eintreten bestimmter Ereignisse dynamisch zu ändern. Zu diesem Zweck führen wir als weiteren Kantentyp des ADEPT-Basismodells sog. *Priorisierungskanten* (ET = `PRIORITY_E`) ein. Sie dienen der Veränderung der Ausführungsprioritäten von Aktivitäteninstanzen während der Laufzeit. Eine Priorisierungskante e ist mit einer Kantenpriorität $EC(e)$ verknüpft, die einen der beiden Werte `REGULAR` oder `EXCEPTIONAL` besitzt. Beim Signalisieren der Kante (siehe unten) wird die Ausführungspriorität ihres Zielknotens entsprechend abgeändert.

Priorisierungskanten besitzen zur Laufzeit – wie andere Kontrollflußkanten eines Ausführungsgraphen auch – eine der drei Markierungen `NOT_SIGNALED`, `TRUE_SIGNALED` oder `FALSE_SIGNALED`. Ihre Festlegung erfolgt nach denselben Regeln wie bei Sync-Kanten, d.h. initial besitzt eine Priorisierungskante den Status `NOT_SIGNALED`. Im Verlauf der WF-Ausführung wird sie auf `TRUE_SIGNALED` (`FALSE_SIGNALED`) gesetzt, wenn ihr Quellknoten als `COMPLETED` (`SKIPPED`) markiert wird. Die Semantik von Priorisierungskanten wird durch die folgende Regel festgelegt:

Regel zur Änderung von Knotenprioritäten:

Sei $CFS_{instance}$ = (CFS, NS, ES, It) der KF-Ausführungsgraph einer WF-Instanz und sei NP: $N \mapsto$ {`REGULAR`, `EXCEPTIONAL`} die Knotenprioritätenfunktion von CFS. Bei Neumarkierung einer Priorisierungskante $e \in E$ mit ET^e = `PRIORITY_E` gilt:

Wird der Status der Kante von $ES(e)$ = `NOT_SIGNALED` in $ES(e)$ = `TRUE_SIGNALED` abgeändert, so setze die Ausführungspriorität des Zielknotens n^*:= $Id_{dest}{}^e$ auf $NP(n^*)$:= $EC(e)$, d.h. gleich der durch die Kante festgelegten Priorität. (Wird eine Priorisierungskante mit `FALSE_SIGNALED` neu markiert, so hat dies keine Auswirkungen auf die Ausführungspriorität ihres Zielknotens.)

Besitzt ein aktivierter Knoten n die Ausführungspriorität `EXCEPTIONAL` (d.h. die Aktivität wird aktuell als Ausnahmeschritt angeboten) und wird eine in n einmündende Priorisierungskante e mit Priorität $EC(e)$ = `REGULAR` neu signalisiert, so wird die Ausführungspriorität von n ebenfalls auf `REGULAR` gesetzt. Das bedeutet, daß die Ausführung von n nun keine Abweichung im Sinne einer Ausnahme mehr darstellt und die betreffende Aktivität dementsprechend wie ein normaler Arbeitsschritt behandelt werden kann. Ein Beispiel hierzu zeigt Abbildung 6-2. Hier wird Aktivität `D` zunächst als Ausnahmeschritt (Abbildung 6-2 a) und später – nach Setzen der Priorisierungskante `C` → `D` auf `TRUE_SIGNALED` – als regulärer Schritt angeboten (Abbildung 6-2 b).

Die strukturellen Einschränkungen für die Verwendung von Priorisierungskanten sind weitgehend identisch mit den Strukturierungsregeln für einfache Sync-Kanten. Für Priorisierungskanten gilt zusätzlich, daß sie auch zwei Knoten aus verschiedenen Teilzweigen einer parallelen Verzweigung mit finaler Auswahl verbinden dürfen. Wir verzichten an dieser Stelle auf weitere Details und auf eine formale Darstellung. Aufgrund ihrer einfachen Semantik haben Priorisierungskanten keinen Einfluß auf qualitative Aussagen zum dynamischen Verhalten von WF. Insgesamt erlauben es die vorgestellten Erweiterungen, präferierte von ausnahmebedingten Ausführungsreihenfolgen zu unterscheiden. Wir zeigen nun, wie sich darauf basierend Vorwärtssprünge im Kontrollfluß modellieren lassen.

6.1.2 Überspringen von Aktivitäten mit/ohne Nachholen (Shortcuts)

In bestimmten Fällen ist von vornherein bekannt, daß einzelne Aktivitäten eines WF nicht bei jeder Ausführung benötigt werden, um das angestrebte Prozeßziel zu erreichen. In anderen Fällen kann es situations- bzw. ausnahmebedingt notwendig werden, Aktivitäten früher als normal durchzuführen, um die Prozeßabwicklung erfolgreich zu gestalten.

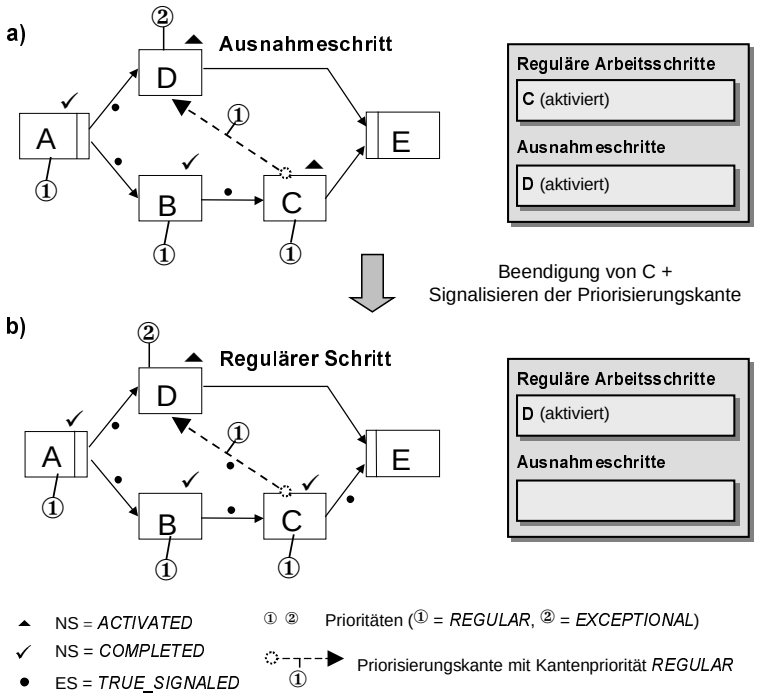


Abbildung 6-2: *Änderung der Ausführungspriorität von Aktivitätenknoten*

Beispiel 6-1 (Überspringen von Aktivitäten):

Die Abwicklung einer medizinischen Untersuchung im Krankenhaus umfaßt im Regelfall mehrere Arbeitsschritte [KRD95, RKD97]: Anordnung der Untersuchung, Planung/Festlegung des Untersuchungstermins, Vorbereitung und Aufklärung des Patienten, Durchführung der eigentlichen Intervention (z.B. endoskopische Untersuchung, radiologische Untersuchung), Befunderstellung und -übermittlung sowie Durchsicht des Befundes durch die anfordernde Stelle. Bereits bei dieser einfachen Ablaufkette müssen in medizinischen Anwendungsumgebungen Benutzer bei Bedarf in flexibler Form vom Standardablauf abweichen können. Beispielsweise muß in Ausnahmesituationen die medizinische Intervention auch ohne die ansonsten übliche elektronische Terminvereinbarung sowie die standardmäßigen Vorbereitungen notfallmäßig durchgeführt werden können. Das heißt Anwender müssen diese Arbeitsschritte situationsbedingt auslassen können. Zuvor vereinbarte Termine müssen dann ggf. storniert werden. Weitere Beispiele finden sich in [Dada95, Meye96].

Um solche ausnahmebedingten Abweichungen vom Standardablauf optimal unterstützen zu können, muß autorisierten Anwendern die Möglichkeit eingeräumt werden, einzelne Aktivitäten oder Folgen von Aktivitäten zu überspringen und sie entweder gar nicht mehr oder aber zu einem späteren Zeitpunkt zu bearbeiten. Solche Vorwärtssprünge lassen sich in ADEPT durch Verwendung von Shortcuts vormodellieren. Voraussetzung hierfür ist, daß die Prozeßzustände in denen eine entsprechende Abweichung erfolgen darf und der Zielknoten des Vorwärtssprungs (d.h. die Aktivität deren Bearbeitung im Ausnahmefall vorgezogen werden soll) a priori bekannt sind.

Aus Sicht des Modellierers erfolgt die Festlegung von Shortcuts auf Basis einer speziellen „Sprungkante“ (*Shortcut-Kante*). Intern wird der Shortcut in eine Darstellung des ADEPT-Basismodells übersetzt. Das bedeutet, daß zu seiner Repräsentation die bisher eingeführten Modellelemente ausreichend sind, wodurch sich auch eine klare Ausführungssemantik ergibt. Generell unterscheiden wir zwischen zwei Arten von Vorwärtssprüngen, solchen mit Nachholen der übersprungenen Schritte (Vorziehen einer Aktivität) und solchen ohne Nachholen der übersprungenen Schritte (Auslassen von Aktivitäten). Auch Mischformen (Überspringen von Aktivitäten mit teilweisem Nachholen) sind möglich, werden an dieser Stelle aber der Einfachheithalber nicht weiter behandelt.

Abbildung 6-3 zeigt die Sicht des Modellierers. Die Shortcut-Kante von A nach E beschreibt einen vorgeplanten Vorwärtssprung, dessen genaue Ausführungssemantik von der Art der Behandlung übersprungener Schritte abhängig ist (siehe unten). Für die Aktivierung des Shortcuts gilt, unabhängig von seiner konkreten Semantik, die folgende Regel: Nach erfolgreicher Beendigung seiner Quellaktivität n_s^{SC} (Knoten A im Beispiel) werden Akteuren in ihren Arbeitslisten nicht nur die normalen, im Kon-

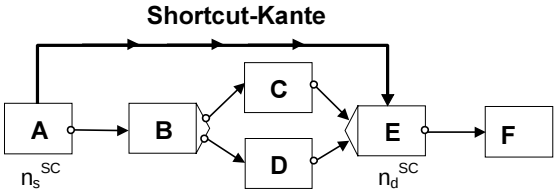


Abbildung 6-3: *Shortcut (Sicht des Modellierers)*

trollfluß nachfolgenden Aktivitäten von n_s^{SC} (B im Beispiel) zur Bearbeitung angeboten, sondern auch die Zielaktivität n_d^{SC} des Vorwärtssprungs (Knoten E im Beispiel). Sie wird zunächst wie ein Ausnahmeschritt behandelt und in Arbeitslisten besonders hervorgehoben. Diese Sonderstellung bleibt solange erhalten, bis n_d^{SC} auf normalen Wege aktiviert wird (In unserem Beispiel trifft dies zu, wenn entweder der Knoten C oder D beendet wird, je nach durchlaufenem Zweig). Tritt dieser Fall ein und wurde n_d^{SC} zuvor noch nicht gestartet, wird diese Aktivität in der Folge wieder als normaler Arbeitsschritt behandelt und in Arbeitslisten entsprechend dargestellt. Wird dagegen die Aktivität n_d^{SC} vorzeitig gestartet, d.h. noch während sie den Status eines Ausnahmeschrittes besitzt, müssen die dadurch übersprungenen Schritte (Knoten B, C oder D) gesondert behandelt werden (siehe unten).

Für die Verwendung eines Shortcuts (n_s^{SC}, n_d^{SC}) (bzw. einer Shortcut-Kante von n_s^{SC} nach n_d^{SC}) gelten, unabhängig von der konkreten Ausführungssemantik, die folgenden beiden Einschränkungen:

- Der Startknoten n_s^{SC} der Shortcut-Kante muß Vorgänger des Zielknotens n_d^{SC} bzgl. normaler Kontrollkanten (des Typs CONTROL_E) sein, d.h. es soll gelten: $n_d^{SC} \in c_succ^*(n_s^{SC})$
- Schleifenstrukturen und Shortcuts dürfen sich nicht überschneiden, sondern nur gegenseitig enthalten, d.h. für $L_S \in LoopStartNodes^{CFS}$, $L_E := endloop^{CFS}(L_S) \in LoopEndNodes^{CFS}$ muß gelten: $n_s^{SC} \in c_succ^*(L_S) \cap c_pred^*(L_E) =: L_{body} \Leftrightarrow n_d^{SC} \in L_{body}$

Weitere Restriktionen für die Verwendung von Shortcuts werden nachfolgend, abhängig von ihrer Semantik, vorgestellt. Wichtig ist an dieser Stelle der Hinweis, daß Shortcut-Kanten nicht Bestandteil des ADEPT-Basismodells sind, sondern in eine Darstellung dieses Meta-Modells übersetzt werden. Wir beschreiben nun Vorwärtssprünge unterschiedlicher Semantik (Vorwärtssprünge mit/ohne Nachholen der übersprungenen Schritte), und zeigen, wie sie sich auf Konstrukte des ADEPT-Basismodells abbilden lassen. In diesem Zusammenhang geben wir weitere Vorbedingungen an, die für eine korrekte Umsetzung der Shortcuts und für eine eindeutige Ausführungssemantik benötigt werden.

6.1.2.1 Überspringen ohne Nachholen

Eine Möglichkeit zur Behandlung der bei einem Shortcut übersprungenen Aktivitäten besteht darin, auf ihre Bearbeitung komplett zu verzichten. Wählt ein berechtigter Akteur den Zielknoten des Shortcuts zur Bearbeitung aus, während dieser den Status eines Ausnahmeschrittes besitzt, werden alle Aktivitäten, die zwischen dem Start- und dem Endknoten des Shortcuts liegen (im Beispiel sind dies B und C bzw. D) – abhängig von ihrem aktuellen Ausführungsstatus – kompensiert, abgebrochen oder deaktiviert. Für jede Aktivität muß festgelegt sein, ob sie kompensierbar oder abbrechbar ist. Anschließend wird mit der Ausführung am Sprungziel (im Bsp. mit der Bearbeitung von E) fortgefahren.

Das Auslassen von übersprungenen Aktivitäten wird durch Verwendung einer parallelen Verzweigung mit finaler Auswahl verwirklicht. Ein Beispiel zeigt die Abbildung 6-4: Auf der linken Seite ist ein Shortcut dargestellt, wie er vom Modellierer festgelegt wird. Die rechte Seite zeigt eine mögliche Umsetzung des Vorwärtssprungs im ADEPT-Basismodell. Sie basiert auf der Verwendung von Knotenprioritäten sowie dem Konstrukt der parallelen Verzweigung mit finaler Auswahl. Einer der Teilzweige enthält den Kontrollblock (sog. Sprungbereich), dessen Aktivitäten bei Anwendung des Shortcuts übersprungen und damit – abhängig von ihrem aktuellen Ausführungsstatus – kompensiert, abgebrochen oder ausgelassen werden sollen. Der andere Teilzweig besteht aus einer vordefinierten Aktivität, die berechtigten Akteuren nach Beendigung von A unter einer Bezeichnung <“Springe zu“ + Sprungknoten> als Ausnahmeschritt in ihren Arbeitslisten angeboten wird. Wählt ein Benutzer diesen speziellen Arbeitsschritt zur Ausführung aus, wird er ohne weitere Aktionen sofort wieder beendet. Der obere Teilzweig der durch die Knoten A und E gebildeten Verzweigung wird dadurch abgeschlossen. Aufgrund der speziellen Ausführungssemantik der verwendeten Verzweigungsart wird daraufhin der

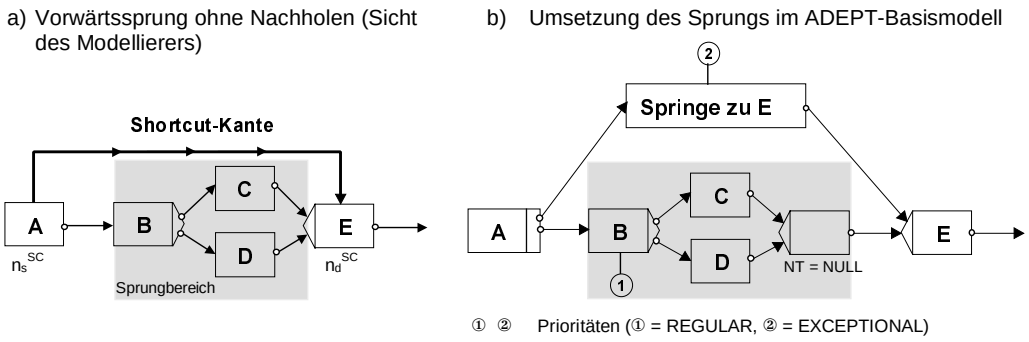


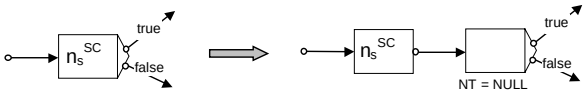
Abbildung 6-4: *Modellierung von Vorwärtssprungen ohne Nachholen und Umsetzung in ADEPT*

untere Teilzweig abgebrochen und zurückgesetzt und mit der Kontrolle am Knoten `E` fortgefahren. Werden dagegen die Aktivitäten des unteren Zweiges auf normalem Wege beendet, ohne daß zuvor die modellierte „Sprungaktivität" zur Ausführung kommt, wird umgekehrt diese Ausnahmeaktivität deaktiviert und aus den Arbeitslisten der Benutzer entfernt. In diesem Fall stellt die Ausführung der Aktivität `E` dann keine Abweichung im Sinne einer Ausnahme mehr dar.

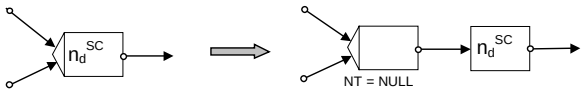
Um den für den Normalfall präferierten Bearbeitungszweig von der ausnahmebedingten Sprungaktivität unterscheiden zu können, werden bei der Umsetzung des Shortcuts die in Abschnitt 6.1.1.1 eingeführten Knotenprioritäten verwendet. Durch die Kennzeichnung mit der Priorität `EXCEPTIONAL` kann der Abweichungsschritt <"Springe zu" + Sprungknoten> Benutzern gegenüber besonders hervorgehoben werden. Im allgemeinen erfolgt die Abbildung eines Shortcuts (n_s^{SC}, n_d^{SC}) (bei Auslassen der übersprungenen Aktivitäten) in das ADEPT-Modell auf Basis der folgenden Umsetzungsschritte:

Umsetzung eines Vorwärtssprungs mit Auslassen in das ADEPT-Basismodell

1. Falls die Ausgangssemantik V_{out} der Shortcut-Quellaktivität n_s^{SC} ungleich `ONE_Of_ONE` ist (d.h. n_s^{SC} ist ein Split-Knoten), wird zunächst ein leerer Knoten (Nullaktivität) n_1 eingefügt, der die Ausgangssemantik und die Ausgangskanten (außer Schleifen- und Fehlerkanten) von n_s^{SC} übernimmt. n_s^{SC} seinerseits erhält die Ausgangssemantik V_{out} = `ONE_Of_ONE` sowie eine ausgehende Kontrollkante mit Zielknoten n_1 zugewiesen.



2. Die Ausgangssemantik des Knotens n_s^{SC} wird nun von V_{out} = `ONE_Of_ONE` nach V_{out} = `ALL_Of_ALL` abgeändert.
3. Falls die Eingangssemantik V_{in} der Shortcut-Zielaktivität n_d^{SC} ungleich `ONE_Of_ONE` ist (d.h. n_d^{SC} ist ein Join-Knoten), wird zunächst ein leerer Knoten (Nullaktivität) n_2 eingefügt, der die Eingangssemantik und die Eingangskanten (außer Schleifen- und Fehlerkanten) von n_d^{SC} übernimmt. n_d^{SC} seinerseits erhält die Eingangssemantik V_{in} = `ONE_Of_ONE` sowie eine einmündende Kontrollkante mit Quellknoten n_2 zugewiesen.



4. Die Eingangssemantik des Knotens n_d^{SC} wird von V_{in} = `ONE_Of_ONE` nach V_{in} = `ONE_Of_ALL` abgeändert.
5. Zwischen den Knoten n_s^{SC} und n_d^{SC} wird nun als zweiter Teilzweig die Sprungaktivität mit Bezeichner <"Springe zu" + n_d^{SC}> eingefügt. Der neu eingefügte Knoten erhält die Ausführungspriorität `EXCEPTIONAL`, während alle anderen Knoten ihre bisherige Priorität beibehalten.

Die Durchführung der Schritte 1-5 erzeugt eine neue Parallelverzweigung mit finaler Auswahl. Um die regelmäßige Blockstrukturierung aufrechterhalten zu können, unterliegt die Verwendung von Shortcuts (n_s^{SC}, n_d^{SC}) der beschriebenen Semantik, zusätzlich zu den bereits genannten Einschränkungen (vgl. oben), den beiden folgenden Restriktionen:

- Der durch die Knotenmenge N_{skip}:= ($c_succ^*(n_s^{SC}) \cap c_pred^*(n_d^{SC})) \cup \{n_s^{SC}, n_d^{SC}\}$ induzierte Teilgraph („Sprungbereich“) bildet einen Kontrollblock. Das bedeutet insbesondere, daß Verzweigungen und Schleifen entweder gar nicht oder aber vollständig im Sprungbereich liegen.
- Verschiedene Shortcuts (n_s^{SC}, n_d^{SC}) und (m_s^{SC}, m_d^{SC}) dürfen sich nicht überlappen, sondern nur gegenseitig enthalten. Formal: $n_s^{SC} \in c_succ^*(m_s^{SC}) \cap c_pred^*(m_d^{SC}) =: M_{skip} \Leftrightarrow n_d^{SC} \in M_{skip}$

Basierend auf diese Voraussetzungen führen die Schritte 1-5, bei Anwendung auf einen korrekten KF-Graphen, wieder zu einem korrekten KF-Graphen, d.h. Struktureigenschaften wie regelmäßige Blockstrukturierung und Zyklenfreiheit gelten weiterhin. Auch die geforderten Dynamikeigenschaften (vgl. Kapitel 4) werden bei korrekter Verwendung von Shortcuts nicht beeinträchtigt.

Für das dem KF-Graphen zugeordnete Datenflußschema muß nach Umsetzung des Shortcut geprüft werden, ob die Regeln DF-1 und DF-2 (vgl. Kapitel 5) erfüllt sind. Trifft dies bereits vor der Hinzunahme des Shortcuts zu, reduzieren sich die notwendigen Datenflußanalysen auf die folgende Fragestellung: Sind obligate Lesezugriffe auf Datenelementen, die von Knoten des Sprungbereichs geschrieben werden, auch dann sicher versorgt, wenn diese Knoten im Ausnahmefall ausgelassen werden? Es genügt, entsprechende Lesezugriffe von Nachfolgerknoten des Shortcut-Sprungziels (inkl. diesem Knoten) zu betrachten. Ergibt sich dabei, daß ein obligater Zugriff infolge der Verwendung des Shortcut nicht mehr sicher versorgt werden kann, muß die Shortcut-Kante wieder entfernt oder die Korrektheit des DF-Schemas durch andere Maßnahmen wiederhergestellt werden. Letzteres ist z.B. möglich, indem betreffenden Aktivitäten sog. Nachforderungsdienste vorgeschaltet werden, die zur Laufzeit genau dann aktiviert werden, wenn ein von ihnen obligat gelesenes Datenelement nullwertig ist. Dadurch erreichen wir, daß bei Nichtausführung des Vorwärtssprungs die Parameterversorgung auf normalem Wege (d.h. auch durch Schreibzugriffe von Aktivitäten des Sprungbereichs) sichergestellt wird, während im Fall eines Vorwärtsprungs die ausnahmebedingte Versorgung über Nachforderungsdienste erfolgt. Die in diesem Zusammenhang notwendigen DF-Analysen können auf Basis des in Kapitel 5 vorgestellten *WriterExists*-Verfahren erfolgen, wobei der umgesetzte Vorwärtssprung (mit Auslassen) wie ein normaler Verzweigungskontrollblock gehandhabt wird. Ein wesentlicher Vorteil der Vormodellierung von Vorwärtssprüngen besteht darin, daß die DF-Analysen vollständig zur Modellierungszeit durchgeführt werden können und nicht zur Laufzeit erfolgen müssen (vgl. Kapitel 7).

6.1.2.2 Überspringen mit Nachholen

Im Gegensatz zu dem gerade beschriebenen Szenario sollen bei der nun vorgestellten Shortcut-Variante die übersprungenen Arbeitsschritte nicht storniert, abgebrochen oder ausgelassen, sondern parallel weiter bearbeitet bzw. nachgeholt werden. Das bedeutet für Knoten des Sprungbereichs N_{skip}:= $c_succ^*(n_s^{SC}) \cap c_pred^*(n_d^{SC})$, daß die Effekte bereits beendeter Aktivitäten erhalten, begonnene Aktivitäten fortgeführt und alle noch nicht aktivierten Aktivitäten ebenfalls ausführbar bleiben sollen, wenn der Vorwärtssprung vollzogen wird. Zu diesem Zweck muß vom Modellierer bei der Definition des Shortcuts (n_s^{SC}, n_d^{SC}) ein weiterer Knoten n_n^{SC} festgelegt werden, bis vor dessen Aktivierung die übersprungenen Schritte „nachzuholen“ sind (Anm.: Hier kann auch eine Knotenmenge angegeben werden). Dieser Knoten muß in jedem Fall Nachfolger von n_d^{SC} (bzgl. Kontrollkanten des Typs `CONTROL_E`) im KF-Graphen sein, d.h. es soll gelten: $n_n^{SC} \in c_succ^*(n_d^{SC})$. Wichtig ist in diesem Zusammenhang die Definition des Nachholbereichs des Vorwärtssprungs, der durch die Knotenmenge N_{jump}:= $\{n_d^{SC}\} \cup (c_succ^*(n_d^{SC}) \cap c_pred^*(n_n^{SC}))$ festgelegt wird. Er definiert den Subgraphen des KF-Graphen, zu dessen Ausführung übersprungene Knoten im Ausnahmefall parallel nachgeholt werden dürfen. Ein Beispiel zeigt Abbildung 6-5.

Um einen solchen Vorwärtssprung in eine korrekte Darstellung des ADEPT-Basismodells transformieren zu können, müssen wieder gewisse Einschränkungen in Bezug auf die Festlegung der Knoten n_s^{SC}, n_d^{SC} und n_n^{SC} getroffen werden:

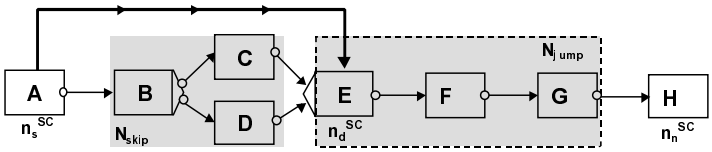


Abbildung 6-5: *Modellierung von Vorwärtssprüngen mit Nachholen (Sicht des WF-Modellierers)*

- Der Zielknoten n_d^{SC} des Shortcuts muß vom Startknoten n_s^{SC} aus deterministisch erreichbar sein, d.h. Vorwärtssprünge dürfen – wenn überhaupt – nur in Teilzweige einer parallelen Verzweigung erfolgen (d.h. für $s \in SplitNodes^{CFS}, j := join^{CFS}(s) \in JoinNodes^{CFS}$ soll stets gelten: $s \in N_{skip} \wedge j \notin N_{skip} \cup \{n_d^{SC}\} \Rightarrow V_{out}^{s} = \texttt{ALL_Of_ALL} \wedge V_{in}^{j} = \texttt{ALL_Of_ALL}$). Verboten sind dagegen Sprünge in Teilzweige einer bedingten Verzweigung hinein.
- Vorwärtssprünge aus parallelen Verzweigungen heraus werden ausgeschlossen, da hier die Behandlung noch nicht beendeter Teilzweige semantisch undefiniert ist, d.h. es soll stets gelten: Für $s \in SplitNodes^{CFS}$, $j := join^{CFS}(s) \in JoinNodes^{CFS}$ mit $s \notin N_{skip} \cup \{n_s^{SC}\} \wedge j \in N_{skip} \Rightarrow V_{in}^{j} \neq \texttt{ALL_Of_ALL}$) (Anm.: Diese Einschränkung kann bei Verallgemeinerung des nachfolgend vorgestellten Ansatzes ggf. auch entfallen).
- Der durch den Nachholbereich N_{jump} induzierte Teilgraph – in obigem Beispiel ist dies die Sequenz $\texttt{E} \rightarrow \texttt{F} \rightarrow \texttt{G}$ – bildet einen Kontrollblock. Startknoten dieses Blocks ist der Knoten n_d^{SC}.
- Für die Nachholbereiche $N_{jump}^{(1)}$ und $N_{jump}^{(2)}$ zweier verschiedener Shortcuts fordern wir stets, daß gilt: $(N_{jump}^{(1)} \subseteq N_{jump}^{(2)}) \vee (N_{jump}^{(1)} \supset N_{jump}^{(2)}) \vee (N_{jump}^{(1)} \cap N_{jump}^{(2)} = \emptyset)$, d.h. die teilweise Überlappung von Nachholbereichen ist nicht zulässig. Sind die Zielknoten zweier Shortcuts identisch, so müssen ihre Nachholbereiche übereinstimmen (Anmerkung: Dieser Fall kann z.B. auftreten, wenn Vorwärtssprünge aus verschiedenen Teilzweigen einer bedingten Verzweigung zum selben Zielknoten modelliert werden.)

Diese Restriktionen dienen im wesentlichen dazu, um bei der Shortcut-Umsetzung wieder KF-Graphen mit regelmäßiger Blockstrukturierung zu erhalten. Das Überspringen von Aktivitäten mit Nachholen wird durch Standardkonstrukte des ADEPT-Basismodells sowie den in Abschnitt 6.1.1 vorgestellten Erweiterungen verwirklicht. Im einzelnen sind die folgenden Schritte durchzuführen, um Shortcuts der beschriebenen Semantik in eine Darstellung des ADEPT-Basismodells umzusetzen:

Umsetzung eines Vorwärtssprungs mit Nachholen in das ADEPT-Basismodell

1. Es wird der minimale Kontrollblock (n_{start}, n_{end}) bestimmt, der die Knoten n_s^{SC}, n_d^{SC} und n_n^{SC} umschließt (Hierzu kann das in Kapitel 3 vorgestellte *MinBlock*-Verfahren verwendet werden).
2. Für diesen Kontrollblock (n_{start}, n_{end}) wird ein neuer, „leerer“ Startknoten n_1 (d.h. eine Nullaktivität) mit Ausgangssemantik $V_{out} = \texttt{ALL_Of_ALL}$ erzeugt. Er übernimmt die Eingangssemantik und Eingangskanten (des Typs `CONTROL_E`) des Knotens n_{start} und erhält n_{start} als direkten Nachfolger zugewiesen. Nach demselben Prinzip wird ein neuer, „leerer“ Endknoten n_2 mit Eingangssemantik $V_{in} = \texttt{ALL_Of_ALL}$ erzeugt. Im Anschluß gilt die folgende Beziehung: $n_2 = join^{CFS}(n_1)$.
3. Der durch den Nachholbereich N_{jump} induzierte Kontrollblock (mit Startknoten n_d^{SC}) wird aus dem Kontext, in den er bisher im KF-Graphen eingebettet ist, entfernt und als neuer Teilzweig der durch n_1 und n_2 gebildeten parallelen Verzweigung wieder in den KF-Graphen eingefügt[25]. Um die gewünschten Bearbeitungsreihenfolgen zu erzwingen, wird der bisherige Vorgängerknoten dieses Kontrollblocks und sein bisheriger Nachfolgerknoten n_n^{SC} über eine normale Kontrollkante verknüpft. Zusätzlich werden im KF-Graphen einfache Sync-Kanten von n_s^{SC} nach n_d^{SC} sowie vom Endknoten des „neuen“ Teilzweiges nach n_n^{SC} eingefügt.
4. Der Zielknoten des Shortcuts n_d^{SC} wird mit der Ausführungspriorität `EXCEPTIONAL` markiert. Darüber hinaus wird eine Priorisierungskante vom Endknoten des Sprungbereichs zum Knoten

---

[25] Dabei wird für Randknoten dieses Blockes, abhängig von ihrer Ein- bzw. Ausgangssemantik, ggf. eine gesonderte Behandlung notwendig. Auf die Darstellung entsprechender Details verzichten wir an dieser Stelle.

n_d^{SC} eingefügt, bei deren Signalisierung die Ausführungspriorität von n_d^{SC} in REGULAR abgeändert werden soll.

5. Ggf. lassen sich auf den resultierenden KF-Graphen noch gewisse Vereinfachungsregeln anwenden (Beispiele für solche Regeln werden in Kapitel 7 vorgestellt)

Die Umsetzung des in Abbildung 6-5 dargestellten Vorwärtssprungs in das ADEPT-Basismodell zeigt Abbildung 6-6 a). Dieser KF-Graph kann noch nach gewissen Regeln vereinfacht[26] werden, wodurch sich der in Abbildung 6-6 b) dargestellte KF-Graph ergibt. Für WF-Instanzen, deren Ausführung auf Basis dieses KF-Graphen erfolgt, werden zur Laufzeit nach Beendigung von Schritt A die Nachfolgeaktivität B als reguläre Aktivität und E als Ausnahmeschritt berechtigten Akteuren angeboten. Wird der WF „standardmäßig" ausgeführt, d.h. werden jeweils nur Aktivitäteninstanzen mit Ausführungspriorität REGULAR bearbeitet, so wird der untere Teilzweig der durch (A, H) gebildeten Verzweigung beendet, bevor E gestartet wird. Der Knoten E erhält dann erst nach Markierung der Priorisierungskante den Status einer regulären Aktivität. Führt ein Benutzer die Aktivität E davor aus, so liegt eine Abweichung i.S. einer Ausnahme vor. In jedem Fall werden die Knoten des Sprungbereichs (B sowie C bzw. D) vor Aktivierung der Aktivität H beendet.

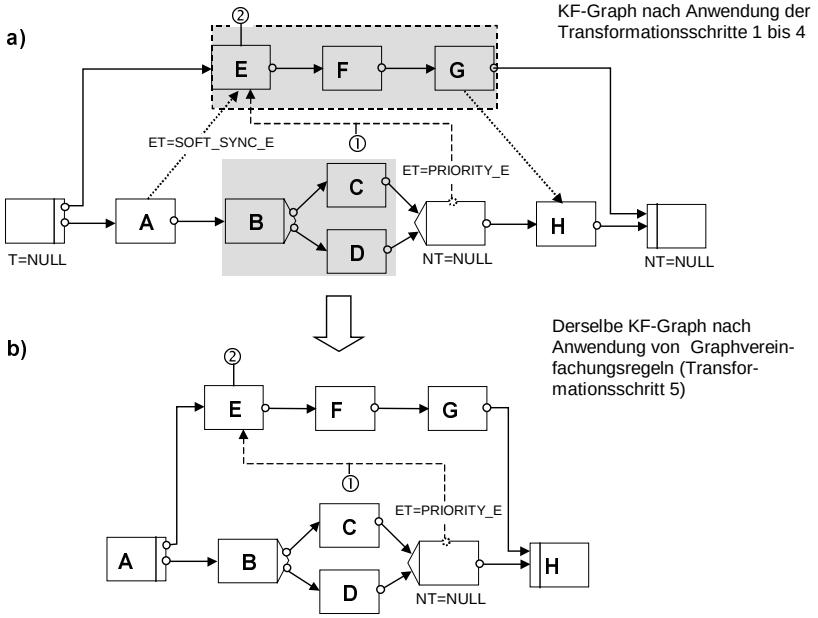


Abbildung 6-6: *Umsetzung von Vorwärtssprungen mit Nachholen im ADEPT-Basismodell (Modellierungszeit)*

Wie im Fall des Überspringens von Aktivitäten ohne Nachholen basieren Vorwärtssprünge mit Nachholen auf Standardkonstrukten des ADEPT-Modells, wobei unter den festgelegten Einschränkungen die gewünschten Struktur- und Dynamikeigenschaften erhalten bleiben. Voraussetzung dafür ist, daß der Ausgangsgraph, auf den der Shortcut angewendet werden soll, korrekt ist. Beispielsweise resultieren aus der Hinzunahme der beiden einfachen Sync-Kanten (→ Schritt 3) keine Zyklen, d.h. Strukturierungsregel KF-4 bzw. KF-4* gilt weiterhin. Grundlegend sind hierbei die für die betroffenen Knoten bestehenden Reihenfolgenbeziehungen: $n_n^{SC} \in succ^*(n_d^{SC})$ und $n_d^{SC} \in succ^*(n_s^{SC})$.

Für das dem Ausgangsgraphen zugeordnete Datenflußschema muß nach Umsetzung des Shortcut geprüft werden, ob die Strukturierungsregeln DF-1 und DF-2 (vgl. Kapitel 5) weiterhin zutreffen. Durch die beschriebenen Graphrestrukturierungen kann es potentiell zur Unterversorgung obligater Aktivitäteneingaben (→ Verletzung der Regel DF-1) sowie zu parallelen Schreibzugriffen auf Datenelementen (→ Verletzung der Regel DF-2) kommen. Solche Fälle müssen erkannt und der Modellierer darauf

[26] Grundlage für die Durchführung entsprechender Vereinfachungen bildet die spezielle Ausführungssemantik von Nullaktivitäten. Wir werden hierauf in Kapitel 7 zurückkommen.

hingewiesen werden. Ist z.B. das dem Ausgangsgraphen zugeordnete DF-Schema korrekt, müssen bei der Umsetzung des Shortcut lediglich die folgenden Überprüfungen durchgeführt werden:

- **Sicherstellung von Regel DF-1**: Sind obligate Lesezugriffe von Aktivitäten des Nachholbereichs N_{jump} auch dann sicher versorgt, wenn ihre Ausführung vorgezogen wird? Die hierzu notwendigen Überprüfungen können auf Basis des *WriterExists*-Verfahrens erfolgen. Es müssen lediglich Lesezugriffe auf die Datenelemente geprüft werden, auf die Aktivitäten des Sprungbereichs N_{skip} schreibenden Zugriff haben. Zeigt sich dabei, daß obligate Datenzugriffe nicht mehr versorgbar sind, so kann wie bei Vorwärtssprüngen ohne Nachholen verfahren werden.
- **Sicherstellung von Regel DF-2**: $D_{skip} \subseteq D$ bezeichne Datenelemente des WF-Datenkontexts auf die Knoten der Menge N_{skip} Schreibzugriff haben und D_{jump} bezeichne die entsprechende Menge für Schreibzugriffe durch Knoten der Menge N_{jump}. Dann können parallele Schreibzugriffe ausgeschlossen werden, falls gilt: $D_{skip} \cap D_{jump} = \emptyset$. Trifft dies nicht zu, muß die Shortcut-Kante entweder entfernt oder aber es müssen zusätzliche Sync-Kanten eingeführt werden.

Falls für einen KF-Graphen mehrere Vorwärtssprünge definiert werden, müssen bei ihrer Umsetzung „semantische“ Konflikte ausgeschlossen werden. Überlappen sich zum Beispiel die Sprungbereiche von Shortcuts unterschiedlicher Semantik (d.h. mit / ohne Nachholen der übersprungenen Schritte) teilweise, so muß einer von ihnen wieder entfernt werden, um eine klare und eindeutige Ausführungssemantik zu erhalten. In den meisten Fällen können solche und vergleichbare Konflikte bereits bei der Definition der Shortcuts auf Basis einfacher Mengenvergleiche erkannt werden.

6.1.3 Abschließende Bemerkungen

Die behandelten Vorwärtssprünge können noch verallgemeinert und ggf. weniger restriktiv gestaltet werden. Beispielsweise sollten in einem adaptiven WfMS auch solche Vorwärtssprünge vormodellierbar sein, bei denen die übersprungenen Schritte wahlweise nachgeholt oder ausgelassen werden können. Konzeptuell ergeben sich hieraus keine neuen Fragestellungen, so daß wir an dieser Stelle auf die Darstellung entsprechender Verallgemeinerungen verzichten. Trivialerweise lassen sich in ADEPT auch automatische Vorwärtssprünge modellieren. Die Grundlage hierfür bilden OR-Aufsplittungen des Kontrollflusses sowie die Verwendung geeigneter Verzweigungsprädikate.

Die Ausführungen zu planbaren Vorwärtssprüngen und ihrer Vormodellierung haben verdeutlicht, welche Vorteile Entwickler und Anwender durch die Bereitstellung entsprechender Möglichkeiten erwarten können. Zu nennen ist hier die Vermeidung aufwendiger Anpassungen bzw. WF-Modifikationen sowie der damit verbundenen Konsistenzprüfungen zur Laufzeit. Dies trifft in besonderem Maße zu, wenn zur Behandlung von Ausnahmen nicht nur einzelne Schritte, sondern ggf. auch mehrere Aktivitäten übersprungen werden müssen. Darüber hinaus zeigen die Beispiele zur Modellierung von Shortcuts eindrucksvoll, daß sich in ADEPT mit den bereitgestellten Konstrukten selbst komplexe Sachverhalte adäquat modellieren lassen. Allerdings wird es aus den eingangs dieser Arbeit genannten Gründen nicht immer möglich sein, alle möglichen Vorwärtssprünge von vornherein im WF-Modell zu berücksichtigen. Spätestens dann werden zur Laufzeit vergleichbare Mechanismen benötigt, um Ad-hoc-Vorwärtssprünge in KF-Ausführungsgraphen durchführen zu können. Dabei können, ebenso wie im gerade beschriebenen statischen Fall, komplexe Restrukturierungen des Ausführungsgraphen sowie Anpassungen des ihm zugeordneten DF-Schemas notwendig werden. Wir werden Ad-hoc-Vorwärtssprünge im Zusammenhang mit dynamischen Strukturänderungen von WFs näher betrachten.

6.2 Behandlung von Rückwärtssprüngen

Ebenso wie es möglich sein muß, vom Standardablauf abweichende Vorwärtssprünge durchzuführen, sollten umgekehrt zur Behandlung bestimmter Ausnahmen, wie z.B. Aktivitätenfehlschlägen (sog. *semantische Fehler*), Rückwärtssprünge im Kontrollfluß erlaubt sein [Meye96, EdLi98]. Die Semantik von Schleifenrücksprungkanten reicht im allgemeinen nicht aus, um alle relevante Arten vorhersehbarer Rücksprünge bei der Modellbildung beschreiben zu können. Zusätzlich benötigen wir Mechanismen für das kontrollierte Zurücksetzen von WF-Instanzen. In ADEPT kann die Rücknahme laufender oder bereits abgeschlossener Arbeitsschritte eines WF in unterschiedlichem Kontext erfolgen:

- Bei Parallelverzweigungen mit finaler Auswahl werden im Anschluß an die Selektion des Siegerzweiges nicht beendete Teilzweige abgebrochen und zurückgesetzt.
- Ausnahmen, die zum Scheitern einer Aktivitätenbearbeitung führen, können durch partielles Zurücksetzen des WF und erneute Ausführung bestimmter WF-Regionen behandelt werden.
- Berechtigte Akteure können unter gewissen Voraussetzungen in die Kontrolle des WF eingreifen, indem sie seine Ausführung unterbrechen und ihn komplett oder partiell zurücksetzen. Dadurch können zuvor getroffene Verzweigungsentscheidungen revidiert oder Fehler vorausgegangener Arbeitsschritte korrigiert werden.

Diese Beispiele deuten die Vielfalt möglicher Rücksprünge an: Das eine Rücksetzoperation auslösende Ereignis kann a priori bekannt sein, muß aber nicht. Dem entsprechend sollten Rücksprünge sowohl automatisch durch das WfMS auf Basis der ihm vorliegenden Informationen, als auch von berechtigten Benutzern initiiert werden können. Auch in Bezug auf das Ziel eines Rücksprungs bzw. den Zustand in den der WF zurückgesetzt werden soll, muß zwischen Rücksetzoperationen unterschieden werden, bei denen diese Parameter bereits zum Zeitpunkt der Modellbildung feststehen, und solchen, bei denen sie sich erst zur Laufzeit ergeben (sog. Ad-hoc-Zurücksetzen). ADEPT berücksichtigt alle relevanten Fälle, unterstützt also sowohl die automatische Behandlung semantischer Fehler auf Aktivitätenebene als auch das Ad-hoc-Zurücksetzen des WF während seiner Ausführung.

Bei der Durchführung eines (partiellen) Rücksprungs wird die WF-Bearbeitung (teilweise) unterbrochen und in einen Zustand zurückgesetzt, der vor dem Start einer bestimmten Aktivitäteninstanz gültig war. Dazu müssen einerseits die ursprünglichen Markierungen des WF-Graphen und die Werte seiner Datenelemente wiederhergestellt werden (sog. *interne Kompensation*), andererseits müssen die externen Effekte laufender bzw. bereits abgeschlossener Aktivitäteninstanzen, die in den „Rücksetzbereich“ (siehe unten) fallen, zurückgenommen werden (sog. *externe Kompensation*). Konkret sind im KF-Ausführungsgraphen Anpassungen der Knoten- und Kantenmarkierungen derart vorzunehmen, daß am Rücksetzknoten korrekt aufgesetzt und mit der Kontrolle fortgefahren werden kann. Dasselbe gilt in Bezug auf Datenelemente aus dem WF-Datenkontext. Sie müssen anschließend wieder die Werte besitzen, die vor Ausführung der zurückgesetzten Aktivitäteninstanzen gültig waren. Grundlage für die Durchführung dieser internen Anpassungen von WF-Kontrolldaten und WF-relevanten Daten – im folgenden auch als *interne Kompensation* bezeichnet – bilden die Ausführungshistorie der WF-Instanz sowie die Historien der einzelnen Datenelemente.

Neben der internen Kompensation von Aktivitäteninstanzen können, soweit sinnvoll und möglich, auch externe Auswirkungen ihrer Ausführung (z.B. Änderungen externer Anwendungsdaten) zurückgenommen werden (sog. *externe Kompensation*). Dies erfolgt in ADEPT, in Anlehnung an Ansätze aus dem Bereich erweiterter Transaktionsmodelle [Elma92, Hsu93, JaKe97, KaRa96], durch Aufruf und Ausführung externer Kompensationsprogramme [GaSa87]. Sie können bei der Definition von Aktivitätenvorlagen festgelegt werden. Im Kontext dieser Arbeit sind die externe Kompensation von Aktivitätenausführungen und damit zusammenhängende Fragestellungen (z.B. Parameterversorgung von Kompensationsaktivitäten, Kompensationssphären [Leym95, Leym97, LeRo97]) nur am Rande von Interesse. Statt dessen konzentrieren wir uns stärker auf Aspekte der internen Kompensation von WFs, wie der Anpassung von Graphmarkierungen nach Zurücksetzen, unter Wahrung der geforderten Integritätsbedingungen. Sie bilden die Grundlage für weiterführende Betrachtungen, wie dem Zurücksetzen von WF-Instanzen bei strukturell modifiziertem Ausführungsgraphen. Hier ergeben sich interessante Fragestellungen, wie sie von den heutigen starren Ausführungsmodellen [Alon96] her, bei denen sich die Ablaufstruktur im Verlauf der Ausführung nicht ändert, nicht bekannt sind. Aber auch bereits für den statischen Fall, d.h. dem Zurücksetzen des WF bei strukturell unverändertem Ausführungsgraphen, ergeben sich im ADEPT-Ansatz interessante Aspekte, die vor allem aus der Verwendung von Konstrukten wie Parallelität oder Schleifen resultieren.

6.2.1 Vormodellierung automatischer Rücksprünge

Für die Vormodellierung ausnahmebedingter Rückwärtssprünge bietet das ADEPT-Basismodell sog. Fehlerrücksprungkanten (kurz: *Fehlerkanten*) an. Auf ihre Verwendung gehen wir in diesem Abschnitt ein. Jedem Aktivitätenknoten $n \in N$ ist durch seine Vorlage eine Menge von Fehlercodes (→

aFailures^n) zugeordnet. Sie beschreibt a priori bekannte Ausnahmen („semantische“ Fehler), die bei der Bearbeitung von Instanzen der Aktivität auftreten können und die zu ihrem Scheitern führen.

Beispiel 6-2 (Scheitern von Aktivitätenbearbeitungen):

Wir beziehen uns auf das eingangs von Abschnitt 6.1.2 vorgestellte Beispiel der Abwicklung einer medizinischen Untersuchung (→ Beispiel 6-1). Bei diesem Ablauf kann die Durchführung der medizinischen Untersuchung aus den unterschiedlichsten Gründen fehlschlagen: unterlassene oder falsche Vorbereitung des Patienten, fehlende Vorbefunde, Rücknahme der Einwilligung des Patienten zur Untersuchung usw. Abhängig vom konkreten Grund des Scheiterns werden unterschiedliche Ausnahmebehandlungen notwendig.

In solchen Ausnahmefällen wird von der aufgerufenen Anwendungskomponente (oder vom Benutzer) bzw. vom ausgeführten Sub-WF ein entsprechender Fehlercode gesetzt und die Aktivitätenbearbeitung abgebrochen. Aktivitätenausgaben bleiben demzufolge undefiniert, so daß mit der WF-Bearbeitung nicht ohne weiteres fortgefahren werden kann, insbesondere dann nicht, wenn obligate Ausgaben fehlen. Aufgabe des WF-Modellierers ist es, für solche a priori bekannten Ausnahmefälle auf Modellebene geeignete Behandlungsformen festzulegen. Mögliche Reaktionen auf den Fehlschlag einer Aktivitätenbearbeitung sind in ADEPT u.a. die Wiederholung der Bearbeitung, die Ausführung einer alternativen Tätigkeit, das Überspringen der Aktivität, das partielle Zurücksetzen des WF in einen früheren Bearbeitungszustand mit anschließender Wiederaufnahme der Bearbeitung oder der kontrollierte Abbruch des WF. In komplexeren Fällen können auch aufwendigere Ausnahmebehandlungen, wie das Zurücksetzen des WF verbunden mit der dynamischen Änderung seines Ausführungsgraphen, durchgeführt werden. Unabhängig von der konkreten Form der Ausnahmebehandlung muß die Konsistenz und Integrität des WF nach Durchführung entsprechender Maßnahmen gewahrt bleiben.

Im Zentrum dieses Abschnitts steht die Behandlung semantischer Fehler auf Aktivitätenebene durch (partielles) Zurücksetzen der WF-Instanz und – damit ggf. verbunden – dem erneuten Ausführen von KF-Teilgraphen. Für die Beschreibung solcher Rücksprünge werden dem Modellierer *Fehlerkanten*, die nicht mit Schleifenrücksprungkanten zu verwechseln sind, angeboten: Eine solche Kante wird mit einem Fehlercode ihrer Quellaktivität n_{fail} verknüpft und besitzt die folgende Semantik: Schlägt bei der WF-Ausführung die Bearbeitung der Aktivität n_{fail} fehl und wird als Fehlerursache der entsprechende Code gesetzt, so wird die Bearbeitung der WF-Instanz (teilweise) unterbrochen und vor den Zielknoten n_{bwd} der Fehlerkante zurückgesetzt. Damit verbunden ist zum einen die Anpassung von Knoten- und Kantenmarkierungen im KF-Ausführungsgraphen, so daß anschließend am Rücksetzknoten n_{bwd} mit der Kontrolle fortgefahren werden kann, zum anderen werden für Aktivitäteninstanzen, die in den „Rücksetzbereich“ (siehe unten) fallen, interne Effekte (z.B. Schreiboperationen auf dem WF-Datenkontext) zurückgenommen und ggf. externe Auswirkungen durch Aufruf zuvor festgelegter Kompensationsaktivitäten aufgehoben.

Definition 6-1 (Fehlerkanten, Rücksprungbereich)

Sei $CFS = (N, E, ...) \in CFS_{all}$ ein allgemeiner KF-Graph. Dann bezeichnen wir eine Kontrollkante $e = (n_{fail}, n_{bwd}, \texttt{FAILURE_E}) \in E$ des Typs $ET^e = \texttt{FAILURE_E}$ als Fehlerrücksprungkante (kurz: *Fehlerkante*). Jede von einem Knoten $n \in N$ ausgehende Fehlerkante e ist immer an einen eindeutigen *Fehlercode* $FC^e \in aFailures^n$ dieses Knotens geknüpft. Besitzt ein Knoten mehrere ausgehende Fehlerkanten, so müssen die ihnen zugeordneten Fehlercodes jeweils voneinander verschieden sein.

Die Knotenmenge $N_{bwd} := (succ^*(n_{bwd}) \cap pred^*(n_{fail})) \cup \{n_{bwd}, n_{fail}\}$ bezeichnen wir als *Rücksprungbereich* der Fehlerkante *f*.

Einige Beispiele für die Verwendung von Fehlerkanten und ihre graphische Darstellung zeigt Abbildung 6-7: Die Fehlerkante `I` → `H` beschreibt einen Rücksetzoperation innerhalb eines parallelen Ausführungszweiges des dargestellten KF-Graphen, und die Kante `K` → `F` definiert einen Rücksprung in einen Teilzweig einer parallelen Verzweigung hinein (Die symmetrische Strukturierung trifft für Fehlerkanten aufgrund ihrer speziellen Semantik nicht zu). Eine besondere Wirkungsweise besitzt die Fehlerkante `B` → `Start`: Soll bei Auftreten eines bestimmten Fehlers (hier dem Scheitern der Aktivität `B`) die WF-Instanz vollständig zurückgesetzt und die Bearbeitung abgebrochen werden, so kann der Modellierer dies ausdrücken, indem er als Zielknoten des Rücksprungs die WF-Startaktivität wählt. Zur Laufzeit werden dann bei Signalisieren der Fehlerkante alle laufenden Aktivitäteninstanzen

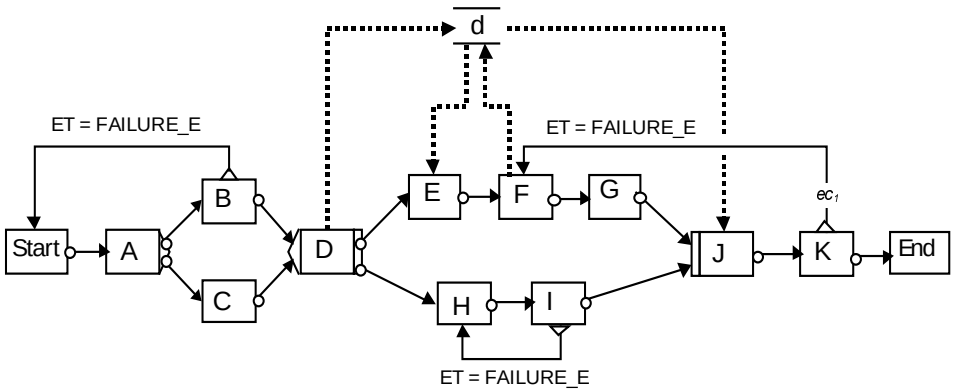


Abbildung 6-7: *Modellierung von Rücksprüngen durch Fehlerkanten*

abgebrochen bzw. bereits abgeschlossene Bearbeitungen zurückgesetzt. Dementsprechend gilt die Bearbeitung der WF-Instanz als gescheitert, und die konkrete Ursache für den Fehlschlag wird der aufrufenden Umgebung über einen Fehlercode mitgeteilt. Dabei kann – Eindeutigkeit vorausgesetzt – der Fehlercode der gescheiterten Aktivität übernommen oder aber ein auf Vorlagenebene explizit hinterlegter Wert verwendet werden. Handelt es sich bei der abgebrochenen Instanz um einen Sub-WF, kann die aufgetretene Ausnahme nun auf der übergeordneten Ebene behandelt werden. Durch diesen Ansatz ist – falls vom Modellierer gewünscht – das kaskadierende Zurücksetzen des WF über mehrere Hierarchiestufen hinweg möglich.

Um eine eindeutige Ausführungssemantik für Fehlerkanten zu erhalten, unterliegt ihre Verwendung – ebenso wie die anderer Kantentypen – wieder gewissen strukturellen Einschränkungen. Sie werden durch die folgende KF-Strukturierungsregel zusammengefaßt:

Strukturierungsregel KF-9 (Verwendung von Fehlerrücksprungkanten)

Sei $CFS = (N, E, ...) \in CFS_{all}$ ein allgemeiner KF-Graph. Dann unterliegt die Verwendung einer Fehlerkante $f = (n_{fail}, n_{bwd}, \texttt{FAILURE_E}) \in E$ den folgenden Regeln:

(1) Die Zielknoten n_{bwd} der Fehlerkante muß im KF-Graphen ein Vorgänger ihres Quellknotens n_{fail} (bzgl. Kontroll- und/oder Sync-Kanten) sein, d.h. es soll gelten: $n_{bwd} \in pred^*(n_{fail})$.

(2) Der Zielknoten n_{bwd} der Fehlerkante darf in keinem Teilzweig einer Verzweigung mit OR-Join-Knoten enthalten sein, dem der Quellknoten n_{fail} nicht ebenfalls angehört. Ansonsten wäre das Zurücksetzen in einen nicht ausgeführten Bereich des Ausführungsgraphen möglich, was jedoch semantisch keinen Sinn ergibt. Formal:

Für $s \in SplitNodes^{CFS}, j := join^{CFS}(s) \in JoinNodes^{CFS}$ mit $V_{in}^{j} = \texttt{ONE_Of_ALL}$:
$[n_{bwd} \in Z := c_succ^*(s) \cap c_pred^*(join^{CFS}(s)) \Rightarrow n_{fail} \in Z]$

(3) Zwischen dem Zielknoten n_{bwd} (exklusive) der Fehlerkante und ihrem Quellknoten n_{fail} (inklusive) darf sich kein Schleifenendknoten befinden, dessen zugehöriger Anfangsknoten nicht im Rücksprungbereich liegt. D.h. Rücksprünge von außerhalb einer Schleife in eine Schleife hinein sind nicht vormodellierbar, da die Schleifeniteration in die zurückgesetzt werden soll, undefiniert wäre. Formal:

Für $L_E \in LoopEndNodes^{CFS} \cap (N_{bwd} \neg \{n_{bwd}\}) \Rightarrow startloop^{CFS}(L_E) := (endloop^{CFS})^{-1}(L_E) \in N_{bwd}$

Die Strukturierungsregel KF-9, (3) ist nicht unbedingt notwendig. Bei ihrem Verzicht benötigen wir allerdings eine klare Semantik für die Durchführung von Fehlerrückspüngen in eine Schleife hinein.

Bevor wir nun auf Aspekte des Zurücksetzens eingehen, definieren wir jeweils eine neue Markierungs- und Ausführungsregel für die Berücksichtigung von Fehlerkanten auf Ausführungsebene. Sie komplettieren die in Kapitel 4 vorgestellten Markierungs- und Ausführungsregeln M_1-M_3 bzw. A_1-A_3 des ADEPT-Basismodells. Im folgenden sei $CFS = (N, E, ...)$ ein allgemeinen KF-Graph (inkl. Fehlerkanten) und $CFS_{instance}$ ein hiervon abgeleiteter Ausführungsgraph.

Markierungsregel M_4 (Scheitern einer Aktivitäteninstanz und Markierung von Fehlerkanten)

Schlägt die Bearbeitung der Instanz einer Aktivität $n \in N$ fehl, so wird ihr Status von $NS(n)$ = `RUNNING` in $NS(n)$ = `FAILED` abgeändert. Zusätzlich wird über den Ausgabeparameter *failure* der Aktivität ein Fehlercode *err_code* gesetzt, der die genaue Ursache des Scheiterns anzeigt.

Falls es eine Kante $e \in E$ mit ET^e = `FAILURE_E` und FC^e = *err_code* gibt, so wird sie mit $ES(e)$ = `TRUE_SIGNALED` markiert. Andernfalls werden andere mit dem Fehlercode assoziierte Aktionen (siehe oben) zur Behandlung des Fehlschlags initiiert.

Ausführungsregel A_4 (Zurücksetzen nach Markierung von Fehlerkanten)

Wird in einem KF-Ausführungsgraphen der Status einer Fehlerkante $f = (n_{fail}, n_{bwd}$, `FAILURE_E`$) \in E$ von $ES(f)$ = `NOT_SIGNALED` in $ES(f)$ = `TRUE_SIGNALED` abgeändert, so werden alle Knoten n des Rücksprungbereichs von f mit $NS(n)$ = `NOT_ACTIVATED` und alle von ihnen ausgehenden Kontrollkanten e (inkl. der Fehlerkante f) mit $ES(e)$ = `NOT_SIGNALED` neu markiert. Zusätzlich zur Anpassung der Knoten- und Kantenmarkierungen werden die Effekte abgeschlossener Aktivitäteninstanzen (z.B. Schreiboperationen auf dem Datenkontext), deren Knoten im Rücksprungbereich liegt, zurückgesetzt. Im Zusammenhang mit Schleifen können davon ggf. mehrere Ausführungen eines Knotens betroffen sein. Das genaue Rücksetzverfahren ist in Abschnitt 6.2.4 beschrieben.

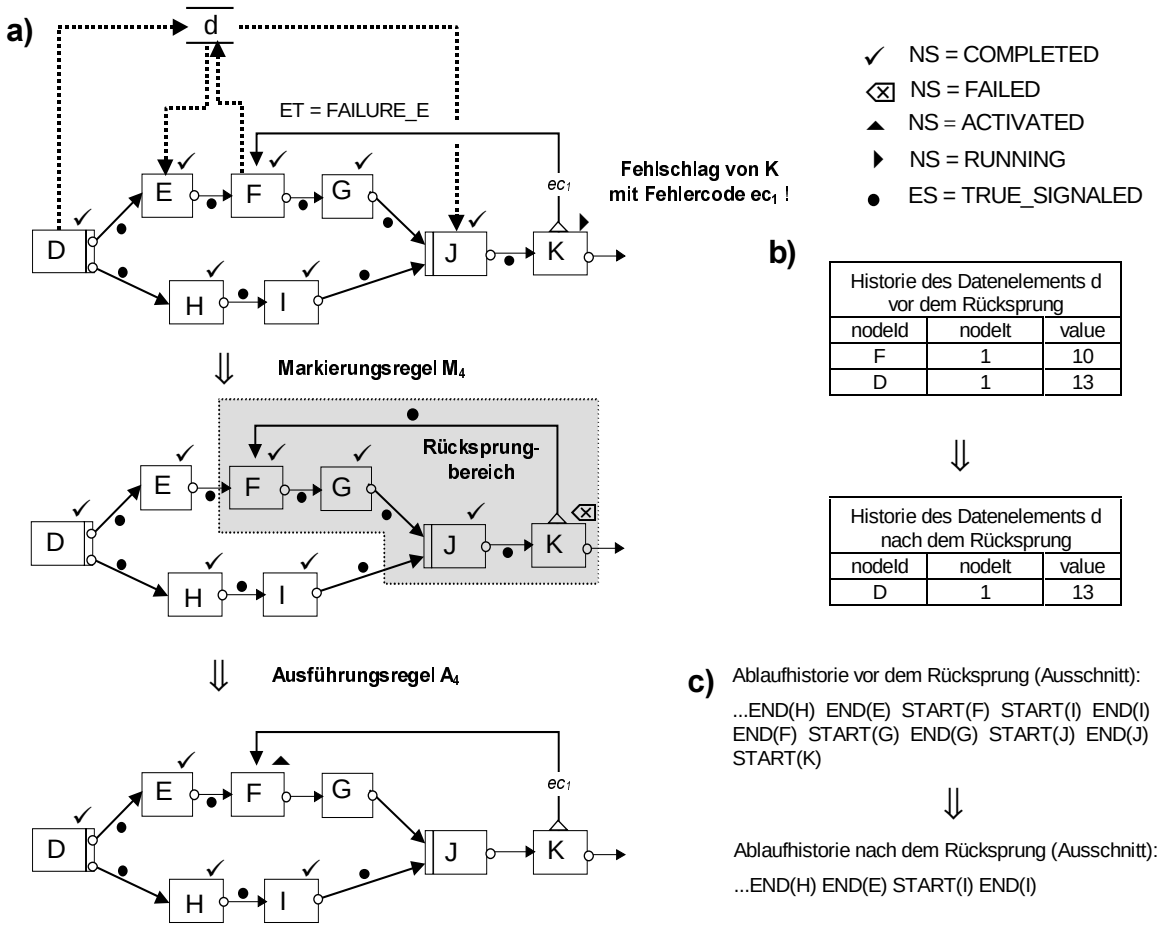
Im Anschluß an diese Maßnahmen wird der Zielknoten der Fehlerkante neu bewertet bzw. aktiviert, so daß mit der WF-Ausführung fortgefahren werden kann.

Ein Beispiel für die Anwendung dieser beiden Regeln zeigt Abbildung 6-8. Im dargestellten KF-Ausführungsgraphen wird nach dem Fehlschlag der Aktivitäteninstanz `K` in einen Teilzweig einer zuvor bereits abgearbeiteten parallelen Verzweigung zurückgesprungen. Wichtig dabei ist, daß durch den Rücksprung der Status von Aktivitäteninstanzen paralleler Teilzweige nicht beeinträchtigt wird. Des weiteren zeigt Abbildung 6-8 wie die Ablauf- und Datenelementhistorien der WF-Instanz beim Zurücksetzen angepaßt werden. Das Verfahren für die Durchführung der notwendigen Rücksetzoperationen wird in Abschnitt 6.2.4 beschrieben.

Fehlerkanten sind ein Konstrukt des ADEPT-Basismodells. Ähnlich wie bei Schleifen führt ihre Verwendung zu erlaubten Zyklen im KF-Graphen. Hieraus resultieren aber keine Verklemmungen, da das Markieren von Fehlerkanten und dadurch ausgelöste Rücksetzoperationen unabhängig von anderen Kantenmarkierungen erfolgen. Vielmehr garantiert die korrekte Verwendung von Kanten dieses Typs, d.h. die Einhaltung der Regel KF-9, daß die geforderten Struktur- und Dynamikeigenschaften des WF (z.B. Verbundenheit von Knoten, WF-Terminierung; vgl. Kapitel 3 und 4) erhalten bleiben und daß bei der Durchführung von Rücksprüngen keine undefinierten Systemzustände entstehen. Als Grundlage für den formalen Nachweis der Terminierung des WF muß die Annahme getroffen werden, daß die Zahl der Fehlschläge von Aktivitätenbearbeitungen während der WF-Ausführung endlich ist. Darauf basierend läßt sich dann zeigen, daß der WF in endlicher Zeit terminiert oder aber zuvor kontrolliert abgebrochen (= Rücksetzen der Bearbeitung vor den Startknoten) wird. Schließlich wird auch die Integrität der Daten bei der Durchführung von Rücksetzoperationen gewahrt (siehe Abschnitt 6.2.4).

6.2.2 Vormodellierung benutzerinitiierter Rücksprünge

Im vorangegangenen Abschnitt haben wir gezeigt, welche Vorkehrungen auf Modellierungsebene getroffen werden müssen, um Fehlschläge einzelner Aktivitäten zur Laufzeit automatisch erkennen und behandeln zu können. Als wesentlich haben sich dabei die Festlegung von Fehler- bzw. Ausnahmecodes auf Aktivitätenebene und ihre Verknüpfung mit Fehlerrücksprungkanten erwiesen. Im allgemeinen sollten ausnahmebedingte Rücksprünge nicht ausschließlich an das Auftreten semantischer Fehler bzw. Ausnahmen auf Ebene einzelner Aktivitäten gebunden sein. Vielmehr muß es für berechtigte Akteure unter gewissen Voraussetzungen auch möglich sein, direkt in die Kontrolle der WF-Ausführung einzugreifen, indem sie den Ablauf (teilweise) unterbrechen und ihn in einen früheren Bearbeitungszustand zurücksetzen. Solchen benutzerinitiierten Rücksprüngen liegen in der Mehrzahl der Fälle exogene Ursachen zugrunde, so daß der genaue Kontext ihres Auftretens a priori nicht oder nur ungefähr



Historie des Datenelements d vor dem Rücksprung		
nodeId	nodeIt	value
F	1	10
D	1	13

Historie des Datenelements d nach dem Rücksprung		
nodeId	nodeIt	value
D	1	13

Abbildung 6-8: *Partielles Zurücksetzen nach Neumarkierung einer Fehlerkante: (a) Anpassung von Knoten- und Kantenmarkierungen des KF-Ausführungsgraphen; (b) Zurücknahme von Schreiboperationen auf dem Datenkontext; (c) Entfernen von Einträgen aus der Ablaufhistorie (vereinfachte Darstellung)*

beschreibbar ist. Abhängig davon, ob eine Vormodellierung möglich ist oder nicht, unterscheiden wir – wie schon bei Vorwärtssprüngen – zwischen planbaren Rücksprüngen und Ad-hoc-Rücksprüngen des Anwenders. Ad-hoc-Rücksprünge werden in den nachfolgenden beiden Abschnitten erörtert. In diesem Abschnitt behandeln wir die Umsetzung planbarer, benutzerinitiierter Rücksprünge.

Als Ergebnis einer Aktivitätenbearbeitung (z.B. der Anordnung einer medizinischen Untersuchung) können Ausgaben (z.B. Art der durchzuführenden Untersuchung und notwendige Vorbereitungen) resultieren, die die konkrete Ausprägung und Abfolge der nachfolgenden Arbeitsschritte wesentlich beeinflussen. Das heißt der Benutzer kann in der Regel gezielt Einfluß auf den konkreten Verlauf eines Prozesses (inkl. Verzweigungsentscheidungen) ausüben. Um so wichtiger ist es, einmal getroffene Entscheidungen und damit verbundene Konsequenzen in Ausnahmesituationen wieder rückgängig machen zu können. Selbst wenn die von einer Aktivität ausgelösten Folgeschritte von anderen Akteuren bearbeitet werden, sollte es möglich sein, daß ihr Bearbeiter (z.B. der eine Untersuchung anordnende Arzt) die Kontrolle über die Durchführung dieser Nachfolgeschritte behält und im Bedarfsfall den Ablauf wieder in den Zustand vor Ausführung seiner Aktivität zurückversetzen kann.

Beispiel 6-3 (Benutzerinitiierte Rücksprünge)

Wir beziehen uns auf das Beispiel 6-1 (Abwicklung einer medizinischen Untersuchung). Hat eine anfordernde Krankenstation nach Anordnung einer Untersuchung nachträglich Änderungswünsche, läßt der aktuelle Zustand des Patienten die Untersuchung nicht zu oder sollen zuvor getroffene Entscheidungen revidiert werden, muß es, solange die eigentliche Intervention noch nicht stattgefunden hat, für das Stationspersonal jederzeit möglich sein, in die Kontrolle des Untersuchungsablaufs einzugreifen und ihn in den Zustand vor Anordnung der Untersuchung zurückzusetzen. Dazu müssen ggf. laufende Tätigkeiten (z.B. Vorbereitungen des Patienten) unterbrochen und zuvor abgeschlossene Schritte (z.B. Festlegung des Untersuchungstermins) storniert werden. Hat die Intervention dagegen bereits stattgefunden ist ein Zurücksetzen des Ablaufs nicht mehr möglich [KRD95].

Das Beispiel zeigt, daß benutzerinitiierte Rücksprünge planbar sein können. Es sollten daher Möglichkeiten vorgesehen werden, sie im WF-Modell abzubilden. In ADEPT lassen sich benutzerinitiierte Rücksprünge durch Verwendung des *RegainControl*-Konstrukts (engl.: Zurückerlangen der Kontrolle) vormodellieren: Einem Akteur kann die Möglichkeit eingeräumt werden, zwischen der Beendigung einer von ihm bearbeiteten Aktivität n_d^{RC} und vor Aktivierung eines im Ablauf nachfolgenden Knotens n_s^{RC}, den WF in den Zustand vor Ausführung von n_d^{RC} zurückzusetzen. Aus Sicht des Modellierers erfolgt die Festlegung eines solchen Rücksprungs, wie im Fall von Shortcuts, auf Basis einer speziellen Kante. Intern wird dieses Konstrukt wieder in eine Darstellung des ADEPT-Basismodells transformiert, so daß wieder eine klare Ausführungssemantik resultiert und wir ohne Erweiterungen des Basismodells auskommen. Abbildung 6-9 zeigt die Sicht des Modellierers: Die *RegainControl*-Kante von E nach A beschreibt einen planbaren Rücksprung, der nach Beendigung der Zielaktivität n_d^{RC} (im Beispiel Aktivität A) aktiviert und bis vor Start der Quellaktivität n_s^{RC} (im Beispiel Aktivität E) ausführbar bleibt. Bei Anwendung des Rücksprungs wird der WF in den Zustand zurückgesetzt, der vor Ausführung der Aktivität A Gültigkeit besaß.

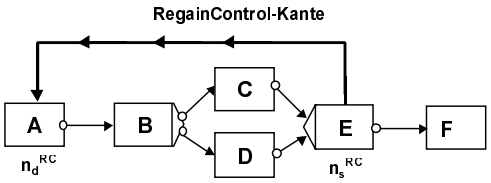


Abbildung 6-9: *RegainControl (Sicht des Modellierers)*

Bevor wir zeigen, wie dieses Konstrukt im ADEPT-Basismodell umgesetzt wird, geben wir einige Einschränkungen für die korrekte Verwendung einer *RegainControl*-Kante von n_s^{RC} nach n_d^{RC} an:

- Der Startknoten n_s^{RC} der RegainControl-Kante muß Nachfolger des Zielknotens n_d^{RC} bzgl. Kontroll- und Sync-Kanten sein, d.h. es soll gelten: $n_d^{RC} \in pred^*(n_s^{RC})$
- Schleifenstrukturen und RegainControl-Kanten dürfen sich nicht überschneiden, sondern nur gegenseitig enthalten.
- Zwischen dem Zielknoten n_d^{RC} des Rücksprungs und seinem Quellknoten n_s^{RC} dürfen sich keine offenen OR-Split-Knoten befinden, d.h. n_s^{RC} muß von n_d^{RC} aus deterministisch erreichbar sein.

Auf die zuletzt genannte Einschränkung kann verzichtet werden, wenn die beschriebene Semantik des *RegainControl*-Konstrukts so abgeändert wird, daß der Rücksprung solange anwendbar bleibt, bis die Aktivität n_s^{RC} entweder aktiviert wird oder aber feststeht, daß sie nicht mehr ausführbar ist (d.h. der Knoten als SKIPPED markiert wird). Die genannten Restriktionen stellen sicher, daß bei der Umsetzung von *RegainControl*-Konstrukten in Darstellungen des ADEPT-Basismodells wieder KF-Graphen mit regelmäßiger Blockstrukturierung und den geforderten Struktureigenschaften resultieren. Dabei verwenden wir Standardkonstrukte des ADEPT-Basismodells sowie die im vorangegangenen Abschnitt vorgestellten Fehlerkanten. Im einzelnen sind die folgenden Transformationsschritte durchzuführen, um eine *RegainControl*-Kante $n_s^{RC} \rightarrow n_d^{RC}$ in eine Darstellung des Basismodells umzusetzen.

Umsetzung eines benutzerinitiierten Rücksprungs (RegainControl) in das ADEPT-Basismodell

1. Es wird der minimale Kontrollblock (n_{start}, n_{end}) bestimmt, der die Knoten n_s^{RC} und n_d^{RC} umschließt.
2. Für diesen Kontrollblock wird ein neuer, „leerer“ Startknoten n_1, d.h. eine Nullaktivität ohne assoziierte Aktion, mit Ausgangssemantik V_{out} = ALL_Of_ALL erzeugt. Er übernimmt die Eingangssemantik und Eingangskanten (des Typs CONTROL_E) des Knotens n_{start} und erhält n_{start} als direkten Nachfolger zugewiesen. Nach demselben Prinzip wird ein neuer, „leerer“ Endknoten n_2 mit Eingangssemantik V_{in} = ALL_Of_ALL erzeugt. Im Anschluß gilt: $n_2 = join^{CFS}(n_1)$.
3. In die durch n_1 und n_2 gebildete parallele Verzweigung wird ein vorgefertigter Kontrollblock – die sog. *RegainControl*-Komponente (RCK) – als neuer Zweig eingefügt. Die RCK wird durch eine parallele Verzweigung mit finaler Auswahl gebildet, deren Split- und Join-Knoten Nullaktivitäten

darstellen. Einer der beiden Teilzweige der RCK wird durch die Rücksprungaktivität (kurz: RC-Aktivität), die die Ausführungspriorität EXCEPTIONAL besitzt, festgelegt. Dabei handelt es sich um eine spezielle Aktivität, die nach ihrem Start sofort wieder mit einem Fehlercode fc^{RC} abgebrochen wird. Der andere Teilzweig besteht aus einer Nullaktivität n_3, nach deren Aktivierung die RC-Aktivität nicht mehr ausführbar ist.

4. Das gewünschte Ausführungsverhalten wird durch folgende Maßnahmen verwirklicht: Um zu gewährleisten, daß der Rücksprung nicht vor Beendigung der Aktivität n_d^{RC} möglich ist, wird eine einfache Sync-Kante e_{S1} von n_d^{RC} zum Knoten der RC-Aktivität hinzugefügt. Umgekehrt wird durch die Hinzunahme einer einfachen Sync-Kante e_{S2} vom direkten Vorgängerknoten von n_s^{RC} zum Knoten n_3 sichergestellt[27], daß der Rücksprung nach Aktivierung des Knotens n_s^{RC} nicht mehr erfolgen kann. Schließlich wird an den Fehlercode fc^{RC} der RC-Aktivität eine Fehlerkante (ET = FAILURE_E) mit dem Zielknoten (bzw. Rücksprungknoten) n_d^{RC} geknüpft. – Die RC-Aktivität kann während des Zeitraums ihrer Aktivierung jederzeit von berechtigten Akteuren[28] gestartet werden, was zum Signalisieren der ausgehenden Fehlerkante und damit zum Zurücksetzen der WF-Instanz vor die Ausführung von n_d^{RC} führt.
5. Ggf. lassen sich auf den resultierenden KF-Graphen noch gewisse Vereinfachungsregeln anwenden (Beispiele für solche Regeln werden in Kapitel 7 vorgestellt)

Ein Beispiel für die Umsetzung eines benutzerinitiierten Rücksprungs zeigt Abbildung 6-10. Für WF-Instanzen deren Ausführung auf Basis dieses KF-Graphen erfolgt, werden zur Laufzeit nach Beendigung von Aktivität A die Arbeitsschritte B und D als reguläre Aktivitäten und die Rücksprungaktivität RC als Ausnahmeschritt berechtigten Akteuren angeboten. Wird die WF-Instanz auf normalem Wege ausgeführt, so wird nach Beendigung von B der Knoten n_3 (Nullaktivität) aktiviert und sofort wieder beendet, was dann zur Deaktivierung der Rücksprungaktivität führt. Startet dagegen ein Akteur die Aktivität RC, bevor B abgeschlossen bzw. C aktiviert wird, so wird die WF-Instanz in den Zustand vor Ausführung von A zurückgesetzt. Das Beispiel zeigt auch, daß Quell- und Zielknoten einer Fehlerkante in unterschiedlichen Teilzweigen einer parallelen Verzweigung liegen können, sofern ihre Reihenfolge über Sync-Kanten festliegt. Die von Strukturierungsregel KF-9 geforderten Bedingungen für die Verwendung von Fehlerkanten werden bei der Umsetzung des *RegainControl*-Konstrukts berücksichtigt. Ebenso bleibt das dem Ausgangsgraphen zugeordnete Datenflußschema (im Beispiel nicht dargestellt) nach der Durchführung der beschriebenen Transformationsschritte in jedem Fall korrekt, da keiner der neu hinzugefügten Knoten eigene Eingaben besitzt bzw. Ausgaben macht und die Anordnungsbeziehungen der schon vorhandenen Knoten nicht geändert werden.

Im folgenden gehen wir auf die von ADEPT angebotenen Möglichkeiten zur Durchführung von Ad-hoc-Rücksprüngen ein. Dabei handelt es sich nicht um planbare bzw. vormodellierbare Rücksprünge, sondern um Abweichungen, deren Festlegung dynamisch zur Laufzeit erfolgt. Nichtsdestotrotz werden Ad-hoc-Rücksprünge in diesem Kapitel behandelt, da sie sich auf dieselben Basismechanismen wie vormodellierbare Rücksprünge zurückführen lassen.

6.2.3 Ad-hoc-Rücksprünge in KF-Ausführungsgraphen

Im allgemeinen lassen sich Rücksprünge, die zur Behandlung aufgetretener Ausnahmen notwendig werden, nicht immer vormodellieren. So ist es für komplexere Anwendungen praktisch unmöglich, alle entsprechenden Ausnahmefälle vorherzusehen und im Modell abzubilden. Das gilt erst recht, wenn der WF zur Laufzeit strukturell abgeändert wird und das Ziel des Rücksprungs z.B. eine Aktivität darstellt, die erst dynamisch während der WF-Ausführung in den Ausführungsgraphen eingefügt worden ist. Aber selbst wenn diese Fälle nicht gegeben sind, ist eine Vormodellierung nicht immer

[27] Gemeint ist hier der direkte Vorgänger bzgl. normaler Kontrollkanten. Handelt es sich bei n_s^{RC} um eine Join-Aktivität, so kann direkt vor diesen Knoten wieder eine Nullaktivität eingefügt werden, die dann die Eingangssemantik und -kontrollkanten von n_s^{RC} übernimmt.

[28] Je nach Festlegung des Modellierers wird diese Aktivität dem Bearbeiter der Aktivität n_d^{RC} oder beliebigen Akteuren einer festgelegten Rolle bei ihrer Aktivierung als Ausnahmeschritt angeboten.

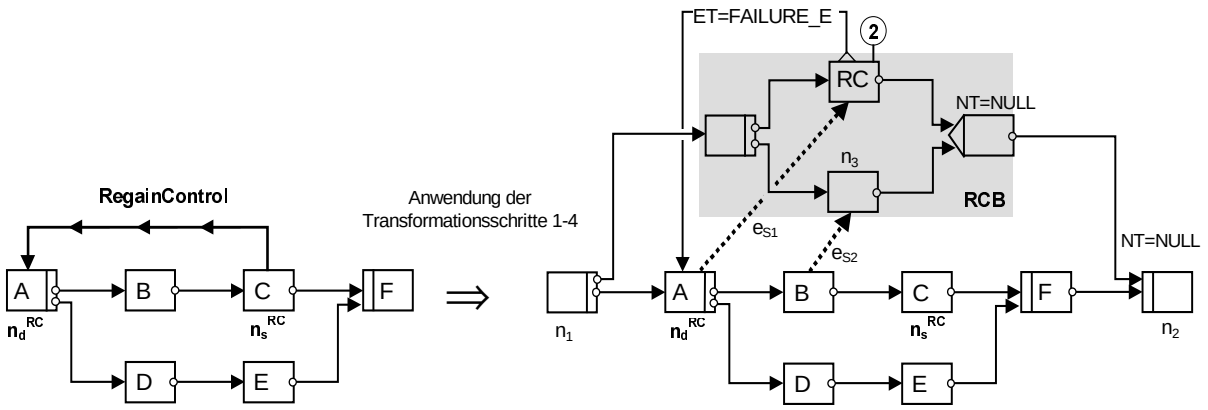


Abbildung 6-10: *Umsetzung von RegainControl-Rücksprüngen in das ADEPT-Basismodell (Modellierungszeit)*

möglich. Beispielsweise können Rücksprünge in Teilzweige einer bedingten Verzweigung hinein auf Modellebene nicht festgelegt werden (vgl. Strukturierungsregel KF-9, 2), da hier a priori nicht bekannt ist, welcher der Teilzweige zur Laufzeit wirklich ausgeführt wird – das Zurücksetzen der Bearbeitung in einen zuvor nicht durchlaufenen Zweig würde semantisch keinen Sinn ergeben. Ähnliches gilt im Zusammenhang mit Schleifen, wenn die Bearbeitung vor die Ausführung einer bestimmten Aktivitäteninstanz einer beliebigen früheren Schleifeniteration zurückgesetzt werden soll.

Aus diesen Gründen erlaubt es ADEPT berechtigten Akteuren, Ad-hoc-Rücksprünge im Ablauf durchzuführen. Dazu kann der Benutzer aus einer Liste bereits beendeter oder laufender Arbeitsschritte einen auswählen, vor dessen Ausführung der WF zurückgesetzt werden soll. Dabei kann es sich im allgemeinen auch um eine Aktivitäteninstanz handeln, die in einer früheren Iteration einer Schleife ausgeführt wurde. Für diesen Fall müssen ggf. mehrere Iterationen einer Schleife zurückgenommen werden. Wichtig ist an dieser Stelle der Hinweis, daß nur berechtigten Personen Zugriff auf solche dynamischen Rücksetzoperationen gewährt wird. Hierzu können in ADEPT Änderungsrechte (→ Wer darf welche Rücksprünge bzw. welche dynamischen Änderungsoperationen auf welche WF-Instanzen in welchen Prozeßzuständen anwenden?) bei der Definition von Organisations- und Prozeß-Modellen festgelegt werden. Diese Aspekte klammern wir in dieser Arbeit aus. Details hierzu finden sich in [Kubi98, Kony96]. Zusätzlich kann für einzelne Aktivitätenknoten angegeben werden, ob der WF überhaupt in den Zustand vor seiner Ausführung zurückgesetzt werden darf oder nicht. Dadurch kann die Menge potentieller Rücksprungziele bereits zur Modellierungszeit ggf. eingegrenzt werden.

Die Umsetzung von spontanen und vormodellierten Rücksetzoperationen (über Fehlerkanten) basiert auf dem im folgenden Abschnitt vorgestellten Verfahren.

6.2.4 Rücksetzverfahren bei strukturell unverändertem KF-Ausführungsgraphen

Im folgenden stellen wir ein Verfahren für das kontrollierte Zurücksetzen eines sich in Ausführung befindlichen WF vor. Auf seiner Basis können auf Ausführungsebene alle zuvor beschriebenen Rücksetzoperationen – vormodellierte Rücksprünge wie auch Ad-hoc-Rücksprünge – verwirklicht werden. Darüber hinaus kann es für das Rücksetzen einzelner „Verliererzweige“ bei Parallelverzweigungen mit finaler Auswahl herangezogen werden. Wir zeigen, welche Anpassungen in Bezug auf Zustandsmarkierungen des Ausführungsgraphen durchgeführt und welche Schreibzugriffe auf dem WF-Datenkontext zurückgenommen werden müssen, um im Anschluß an einen Rücksprung mit der Bearbeitung des WF in einem konsistenten Zustand fortfahren zu können. Das in diesem Abschnitt beschriebene Rücksetzverfahren ist zunächst nur für WF-Instanzen anwendbar, deren KF-Graph während der Ausführung strukturell nicht verändert wird, d.h. immer noch der Prozeßvorlage entspricht. Aspekte der Umsetzung von Rücksprüngen bei dynamisch modifizierten Ausführungsgraphen (z.B. Rücknahme temporärer Strukturänderungen) werden in Kapitel 8 diskutiert. Bevor wir das ADEPT-Rücksetzverfahren vorstellen (Abschnitt 6.2.4.3), gehen wir kurz auf wichtige Grundlagen ein. Sie betreffen zum einen die Repräsentation von Ablauf- und Datenhistorien einer WF-Ausführung (Abschnitt 6.2.4.1), zum anderen das Zurücksetzen einzelner Aktivitäteninstanzen (vgl. Abschnitt 6.2.4.2).

6.2.4.1 Ablauf- und Datenhistorien

a) Ablaufhistorie

Für das korrekte Zurücksetzen einer WF-Instanz benötigen wir Informationen zu ihrem bisherigen Verlauf. Die Angaben zu aktuellen Knoten- und Kantenmarkierungen reichen hierfür im allgemeinen nicht aus, da – bei Verwendung von Schleifen – bestimmte Aktivitäten ggf. mehrfach ausgeführt werden. Aus diesem Grund wird für jede (Sub-)WF-Instanz eine *Ablaufhistorie* verwaltet, in der relevante Ereignisse zu ihrer Ausführung (i.w. Informationen zum Start und zur Beendigung von Aktivitäteninstanzen) protokolliert werden. Nicht protokolliert werden dagegen Aktionen wie die Markierung nicht mehr ausführbarer Knoten als `SKIPPED` (bei der Nichtauswahl von Ablaufzweigen). Vereinfacht dargestellt weisen Historieneinträge die folgende Struktur auf:

- **START**(*<nodeId>*, *<It>*, *<repeated>*): Historieneintrag für den Start der *<It>*.ten Ausführung des Aktivitätenknotens *<nodeId>*. Das Feld *<repeated>* ist nur für Schleifenanfangsknoten definiert und zeigt an, ob der Knoten über normale Kontrollkanten (*<repeated>* = *False*) oder über eine Schleifenrücksprungkante (*<repeated>* = *True*) aktiviert wurde[29]. Diese Information wird von dem nachfolgend beschriebenen Rücksetzverfahren benötigt.
- **END**(*<nodeId>*, *<It>*, *<state>*, *<code>*): Historieneintrag für die Beendigung der *<It>*.ten Ausführung des Aktivitätenknotens *<nodeId>*. Das Feld *<state>* $\in$ {`COMPLETED`, `FAILED`} zeigt an, ob die Ausführung erfolgreich war oder nicht, und *<code>* speichert – soweit definiert – den von der Aktivitäteninstanz gesetzten Auswahlwert oder Fehlercode.

Die Einträge der Ablaufhistorie einer WF-Instanz sind geordnet. Grundlegend hierfür sind die Zeitpunkte, zu denen die zugehörigen Aktivitäteninstanzen gestartet bzw. beendet werden. Dabei steht der aktuellste Eintrag jeweils am Ende der Historie. Besteht zwischen zwei Historieneinträgen $H_{end}(n_1, i)$:= END(n_1, i, $*$) und $H_{start}(n_2, j)$:= START(n_2, i, $*$) ($n_1, n_2 \in N$; $i, j > 0$) die Anordnungsbeziehung $H_{end}(n_1, i) \prec H_{start}(n_2, j)$, so wurde die i.te Ausführung des Knotens n_1 beendet, bevor mit der j.ten Bearbeitung von n_2 begonnen wurde[30]. Liegen n_1 und n_2 im KF-Graphen nicht parallel zueinander, muß folglich n_1 ein Vorgänger von n_2 sein oder aber die beiden Knoten gehören dem Körper einer Schleife an und die i.te Ausführung von n_1 erfolgte in einer früheren Iteration als die j.te Bearbeitung von n_2. Für parallel ausführbare Aktivitäteninstanzen kann die Reihenfolge ihrer Historieneinträge dagegen beliebig sein. Dieser Umstand muß beim Zurücksetzen einzelner Teilzweige einer parallelen Verzweigung beachtet werden.

Auf Basis der Ablaufhistorie sowie dem der WF-Instanz zugrundeliegenden KF-Graphen läßt sich ihr KF-Ausführungsgraph (d.h. die Markierungen von Knoten und Kanten) jederzeit rekonstruieren. Kantenmarkierungen müssen nicht explizit gespeichert werden, da sie sich aus den Historiendaten und den ADEPT-Markierungsregeln ableiten lassen.

b) Datenhistorien

Um den WF-Datenkontext korrekt zurücksetzen zu können, werden für die Datenelemente eines WF nicht nur die aktuell gültigen Werte, sondern auch die Werte früherer Versionen verwaltet. Hierauf sind wir bereits in Abschnitt 5.5 eingegangen. Dort haben wir gezeigt, daß jedes Datenelement $d \in D$ eine Historie $CtxtValues^d$ besitzt, in der die verschiedene Versionen gespeichert werden. Die Historieneinträge sind geordnet, wobei der zuletzt geschriebene Wert am Anfang steht. Ein einzelner Historieneintrag $v \in CtxtValues^d$ besitzt die Struktur v=(*nodeId*, *nodeIt*, *val*, *mode*), wobei *val* dem von der Aktivität $nodeId^v$ ($mode^v$ = `MAIN`) (bzw. einem ihrer vor- oder nachgeschalteten Hilfsdienste [$mode^v$ = `PRE`/`POST`]) bei ihrer $nodeIt^v$-ten Ausführung geschriebenen Wert von d entspricht.

---

[29] Selbst wenn ein Schleifenanfangsknoten `L` mehrfach aktiviert bzw. ausgeführt wird, kann daraus nicht geschlossen werden, daß die betreffende Schleife auch mehrfach durchlaufen wurde; bei geschachtelten Schleifen könnte der Knoten `L` seinerseits in einer umgebenden Schleife enthalten sein.

[30] Trivialerweise gilt für die bei der i.ten Ausführung ($i > 0$) eines Knotens $n \in N$ erzeugten Historieneinträge: $H_{start}(n, i) \prec H_{end}(n, i)$, d.h. $H_{start}(n, i)$ steht in der Ablaufhistorie vor $H_{end}(n, i)$.

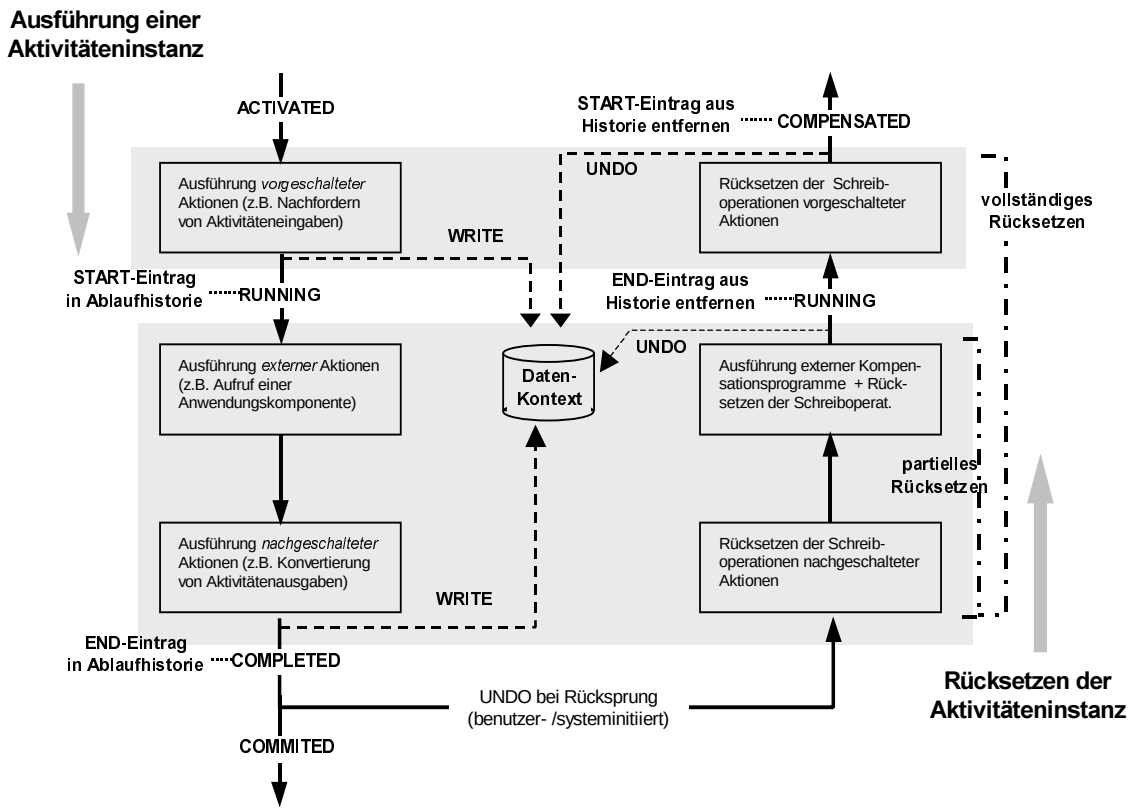


Abbildung 6-11: *Zurücksetzen einer elementaren Aktivitäteninstanz*

6.2.4.2 Rücksetzen elementarer Aktivitäteninstanzen

Die Aktivitäteninstanzen eines WF *inst* (mit Ausführungsgraph CFS_{inst} und Datenkontext DF_{inst}) werden auf logischer Ebene durch die Angabe ihres Knotens $n \in N$ und ihres Iterationszählers $It > 0$ [kurz: $n(It)$] repräsentiert. Soll eine laufende oder eine bereits abgeschlossene Aktivität zurückgesetzt werden, müssen ihre Effekte auf WF-Kontrolldaten, WF-relevanten Daten und ggf. externen Anwendungsdaten (vgl. Kapitel 2) rückgängig gemacht werden. Die konkret durchzuführenden Maßnahmen einer solch elementaren Rücksetzoperation *RollbackActivity*(*inst*, CFS_{inst}, DF_{inst}, *n*, *It, mode*) hängen dabei vom aktuellen Bearbeitungsstatus der Aktivität ab (vgl. Abbildung 6-11).

Die Rücknahme einer *aktivierten*, aber noch nicht gestarteten Aktivitäteninstanz $n(It)$ (vgl. Tabelle 6-1 a) erfordert lediglich die Neumarkierung des Knotens *n* (als `NOT_ACTIVATED`) im Ausführungsgraphen CFS_{inst} und – damit einhergehend – das Entfernen entsprechender Einträge aus Arbeitslisten. Für eine *gestartete*, aber noch nicht beendete Aktivitäteninstanz $n(It)$ (vgl. Tabelle 6-1 b) bedeutet das Rücksetzen, daß ihre Bearbeitung abgebrochen, der entsprechende START-Eintrag aus der Ablaufhistorie entfernt und der Status des Knotens *n* auf `NOT_ACTIVATED` zurückgesetzt werden müssen. Wir nehmen vereinfachend an, daß laufende Aktivitätenprogramme abbrechbar sind und daß bereits erfolgte Änderungen auf externen Anwendungsdaten rückgängig gemacht werden können. Des weiteren müssen ggf. Schreiboperationen auf Elementen des Datenkontexts DF_{inst}, die durch vorgeschaltete Aktionen von $n(It)$ erfolgt sind, zurückgenommen werden (siehe unten). Weitere Schreibzugriffe auf dem Datenkontext kann es aufgrund des aktuellen Status der Aktivität nicht geben – die Ausgabeparameter der Aktivität (bzw. der von ihr gerufenen Komponente) und ihrer nachgeschalteten Aktionen werden erst beim Wechsel in den Zustand `COMPLETED` in den Datenkontext geschrieben.

Für eine *beendete* Aktivität $n(It)$ müssen beim *vollständigen* Zurücksetzen (vgl. Tabelle 6-1 c) zusätzlich die Schreibzugriffe auf dem Datenkontext DF_{inst} unwirksam gemacht werden, die durch Ausgaben der Aktivität und der ihr nachgeschalteten Aktionen verursacht wurden. Darüber hinaus muß neben dem START-Eintrag auch der entsprechende END-Eintrag aus der Ablaufhistorie entfernt und der Status des Aktivitätenknotens *n* und seiner ausgehenden Kontrollkanten neu gesetzt (→ `NOT_ACTIVATED` bzw. `NOT_SIGNALED`) werden. Sollen auch externe Auswirkungen der Aktivitätenbearbeitung (z.B. Schreibzugriffe der gerufenen Anwendungskomponente auf externen Daten-

banken) rückgängig gemacht werden, besteht die Möglichkeit, beim Zurücksetzen der Instanz eine *Kompensationsaktivität* auszuführen. Zu diesem Zweck kann bei der Vorlagendefinition ein *Kompensationsprogramm* angegeben werden. Dabei kann es sich um das eigentliche Aktivitätenprogramm, das dann mit „inversen" Parametern gerufen wird, oder eine beliebige Anwendungskomponente handeln. Wie bereits erwähnt, vertiefen wir das hier nicht weiter und konzentrieren uns im folgenden ausschließlich auf Aspekte der internen Kompensation einer WF-Ausführung.

Für eine *abgeschlossene* Aktivitäteninstanz ist es auch möglich, sie in den Zustand zurückzusetzen, in dem sie sich vor (erstmaliger) Ausführung des ihr zugeordneten Aktivitätenprogramms befand (vgl. Tabelle 6-1 d). In diesem Fall bleiben die Effekte vorgeschalteter Aktionen (z.B. Nachforderungsdiensten) erhalten und die Aktivität kann nochmals ausgeführt werden. Dies kann sinnvoll sein, wenn die Bearbeitung einer gerade beendeten Aktivität wiederholt werden soll, ohne daß der Anwender Aufrufdaten, die er bereits zuvor (d.h. bei Start der letzten Aktivitätenausführung) festgelegt hat, nochmals eingeben muß. Die nachfolgende Tabelle faßt die verschiedenen Fälle nochmals zusammen:

RollbackActivity (***inst***, ***CFS***$_{inst}$, ***DF***$_{inst}$, ***n***, ***It***, ***mode***) **Rücksetzen einer Aktivitäteninstanz *n*(*It*) mit aktuellem Bearbeitungsstatus *NS*(*n*)**	**... erfordert Rücksetzen von Markierungen im Ausführungsgraphen *CFS***$_{inst}$ **und externe Maßnahmen**	**... erfordert Rücksetzen von Schreibzugriffen auf dem WF-Datenkontext *DF***$_{inst}$
a) $NS(n)$ = `ACTIVATED`	$NS(n)$:= `NOT_ACTIVATED`	-
b) $NS(n)$ = `RUNNING`	$NS(n)$:= `NOT_ACTIVATED` (Kontrollierter Abbruch und ggf. Kompensation der ausgeführten Anwendungskomponente)	$\forall\, d \in D$ mit $ActivityWrites(n, d,$ `ANY`$)$: Entferne alle Einträge (*nodeId*, *nodeIt*, *val*, *mode*) aus der Datenhistorie von *d*, für die gilt: *nodeId* = *n*, *nodeIt* = *It* und *mode* = `PRE`.
c) $NS(n)$ = `COMPLETED` (*mode* = `BEFORE_START`)	$NS(n)$:= `NOT_ACTIVATED` <u>und</u> $\forall\, e \in E$ mit $Id_{source}(e) = n$: Setze ES(e) = `NOT_SIGNALED` (Kompensation der ausgeführten Anwendungskomponente)	$\forall\, d \in D$ mit $ActivityWrites(n, d,$ `ANY`$)$: Entferne alle Einträge (*nodeId*, *nodeIt*, *val*, *mode*) aus der Datenhistorie von *d*, für die gilt: *nodeId* = *n*, *nodeIt* = *It* und *mode* $\in$ {`PRE`, `MAIN`, `POST`}
d) $NS(n)$ = `COMPLETED` (*mode* = `AFTER_START`)	$NS(n)$:= `RUNNING` <u>und</u> $\forall\, e \in E$ mit $Id_{source}(e) = n$: Setze ES(e) = `NOT_SIGNALED` (Kompensation der ausgeführten Anwendungskomponente)	$\forall\, d \in D$ mit $ActivityWrites(n, d,$ `ANY`$)$: Entferne alle Einträge (*nodeId*, *nodeIt*, *val*, *mode*) aus der Datenhistorie von *d*, für die gilt: *nodeId* = *n*, *nodeIt* = *It* und *mode* $\in$ {`MAIN`, `POST`}

Tabelle 6-1: *Rücksetzen einzelner Aktivitäteninstanzen*

6.2.4.3 Kontrolliertes Zurücksetzen einer WF-Instanz

Das nun beschriebene Rücksetzverfahren erlaubt es, den Bearbeitungsstatus eines WF bzw. eines WF-Ausschnittes in einen Zustand zurückzusetzen, der vor der Ausführung einer Aktivitäteninstanz $n_{bwd}(It)$ Gültigkeit besaß (Anmerkung: Liegt n_{bwd} nicht in einem Schleifenkörper, so wird der Knoten maximal einmal ausgeführt; es gilt dann: $It = 1$). Anschließend kann mit der Kontrolle am Rücksetzknoten n_{bwd} fortgefahren werden. Insbesondere ist es möglich, einzelne Teilzweige einer parallelen Verzweigung oder mehrere Iterationen einer Schleifenausführung vollständig oder teilweise zurückzunehmen, ohne daß dadurch die Ausführung paralleler Bearbeitungszweige beeinträchtigt wird. In Verbindung mit hierarchischen WFs kann zudem die Bearbeitung des Toplevel-WF vor die Ausführung von Aktivitäteninstanzen untergeordneter Hierarchiestufen zurückgesetzt werden. In allen diesen Fällen wird der Status von WF-Kontrolldaten und WF-relevanten Daten derart angepaßt, daß anschließend eine korrekte Wiederaufnahme der Bearbeitung möglich ist und die von ADEPT geforderten Dynamikeigenschaften des WF (vgl. Kapitel 4) erhalten bleiben.

a) Zurücksetzen von Aktivitäteninstanzen innerhalb derselben Hierarchiestufe

Wir betrachten zunächst WFs mit beliebiger, nicht-hierarchischer Struktur und zeigen wie sich für sie das Zurücksetzen in den Zustand vor Ausführung einer Aktivitäteninstanz $n_{bwd}(It)$ gestaltet.

Vorüberlegungen

Liegen keine parallelen Bearbeitungszweige vor, kann das (partielle) Zurücksetzen der WF-Instanz im wesentlichen auf Basis der Informationen aus der Ablaufhistorie und den Datenelementhistorien erfolgen. Der Grund hierfür ist, daß in diesem Fall aus der Anordnung der Einträge in der Ablaufhistorie unmittelbar auf die Reihenfolge der zugehörigen Aktivitätenknoten im KF-Ausführungsgraphen geschlossen werden kann. Die Menge der zurückzusetzenden Aktivitäteninstanzen ist eindeutig festgelegt durch diejenigen Einträge der Ablaufhistorie, die dem START-Eintrag von $n_{bwd}(It)$ (inklusive) nachfolgen. Das Zurücksetzen läßt sich dementsprechend einfach verwirklichen, indem die Ablaufhistorie – beginnend mit dem aktuellsten Eintrag – in umgekehrter Reihenfolge bis zum START-Eintrag von $n_{bwd}(It)$ durchlaufen wird und für jeden traversierten Eintrag die notwendigen internen und externen Kompensationsaktionen durchgeführt werden (vgl. Abschnitt 6.2.4.2). Durch diese Vorgehensweise lassen sich sukzessive alle Zwischenzustände des WF in umgekehrter Reihenfolge wiederherstellen, bis schließlich der gewünschte Zielzustand erreicht wird.

Für KF-Ausführungsgraphen mit parallelen Verzweigungen – Historieneinträge nebenläufig ausgeführter Aktivitäteninstanzen können beliebig angeordnet sein (vgl. Abschnitt 6.2.4.1) – ist dieses Verfahren in der beschriebenen Form nicht anwendbar. Ansonsten wären vom Zurücksetzen einzelner Bearbeitungszweige unter Umständen auch Aktivitäteninstanzen unabhängiger, paralleler Zweige betroffen. Beispielsweise würde die Anwendung des skizzierten Verfahrens auf die WF-Instanz aus Abbildung 6-8 dazu führen, daß die nicht in den Rücksetzbereich fallende Aktivitäteninstanz `I(1)` ebenfalls zurückgenommen wird. Um solche unerwünschten Seiteneffekte ausschließen zu können, müssen beim Zurücksetzen der WF-Instanz zusätzlich Informationen zu den Anordnungsbeziehungen von Aktivitätenknoten im KF-Ausführungsgraphen berücksichtigt werden.

Ein Sonderfall ist gegeben, wenn die Bearbeitungsreihenfolge zwischen Aktivitätenknoten paralleler Zweige über Sync-Kanten festliegt: Wird der Quellknoten n einer Sync-Kante zurückgesetzt, so müssen die Markierungen aller ausgehenden Kontroll- und Sync-Kanten und demzufolge alle Nachfolgerknoten von n bzgl. dieser beiden Kantentypen (transitiv) zurückgenommen werden. Etwas anders verhält es sich mit Priorisierungskanten. Wird die Markierung einer solchen Kante aufgehoben, muß für ihren Zielknoten lediglich geprüft werden, ob sich dadurch seine Ausführungspriorität (Ausnahmeschritt, regulärer Schritt) ändert oder nicht (vgl. Abschnitt 6.1.1). Auch in Verbindung mit Schleifen ergeben sich einige Besonderheiten, da es im allgemeinen – wie bereits erwähnt – möglich sein muß, ggf. mehrere Iterationen einer Schleifenausführung vollständig oder teilweise zurückzusetzen.

Für alle diese Fälle gilt es zu beachten, daß die Knoten- und Kantenmarkierungen des Ausführungsgraphen beim Zurücksetzen korrekt angepaßt werden. Dabei genügt es im allgemeinen nicht, nur die Markierungen der zurückgesetzten Aktivitätenknoten und ihrer ausgehenden Kanten aufzuheben (vgl. Abschnitt 6.2.4.2). Wie in Kapitel 4 bereits erörtert, werden im Zusammenhang mit bedingten Verzweigungen auch die Knoten und Kanten nicht ausgewählter Teilzweige nach Beendigung des OR-Splits markiert (als `SKIPPED` bzw. `FALSE_SIGNALED`)[31]. Dasselbe trifft auf die „Verliererzweige“ paralleler Verzweigungen mit finaler Auswahl zu, wenn der zugehörige OR-Join gestartet und damit der „Siegerzweig“ festgelegt wird. Wird nun die Ausführung der WF-Instanz vor einen OR-Split-Knoten zurückgesetzt, so müssen zuvor getroffene Verzweigungsentscheidungen wieder aufgehoben und dementsprechend für die von diesem Knoten ausgehenden Teilzweige – unabhängig davon welche Markierung sie tragen – sämtliche Knoten- und Kantenmarkierungen zurückgenommen werden[32] (vgl.

[31] Dieser Markierungsansatz wurde gewählt, um mit einfachen Mitteln sicherstellen zu können, daß eine dynamische Änderung nicht in Bereichen des Ausführungsgraphen vorgenommen wird, die infolge ihres aktuellen Status nicht mehr (oder zumindest nicht mehr in der aktuellen Schleifeniteration) durchlaufen werden können.

[32] Aufgrund der von ADEPT festgelegten Markierungs-/Ausführungsregeln sind im Verlauf der WF-Ausführung nur Markierungen generierbar, für die gilt: Besitzt ein Knoten $n \in N$ eine Markierung $NS(n) \notin$ {`COMPLETED`, `SKIPPED`}, so weisen alle Nachfolgerknoten bzgl. Kontroll- und/oder Sync-Kanten $n^* \in succ^*(n)$ den Status $NS(n^*)$ = `NOT_ACTIVATED` auf (vgl. Satz 4-1). Auf diesen Aspekt ist beim Zurücksetzen zu achten.

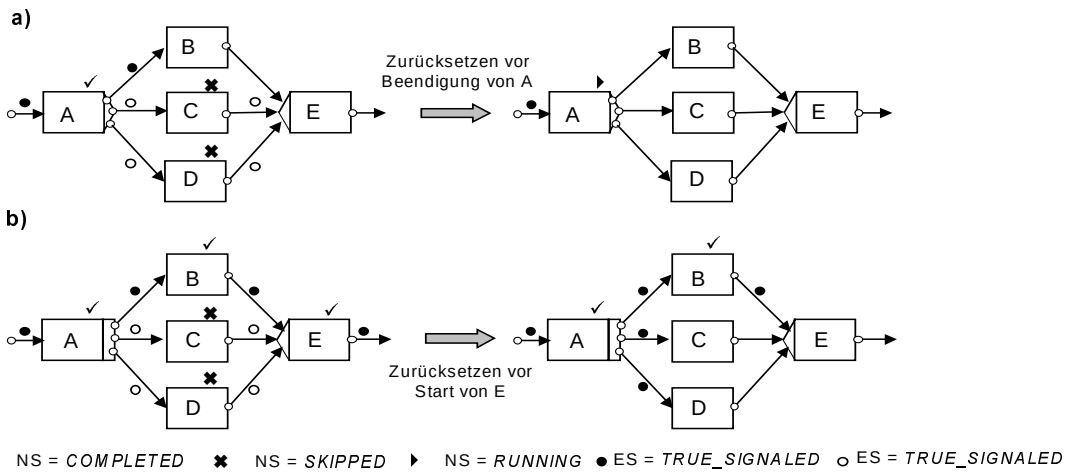


Abbildung 6-12: *Rücksetzen eines OR-Split-Knotens* [*(a) Das Zurücksetzen der Beendigung eines OR-Split-Knotens führt zur Demarkierung von zuvor nicht ausgewählten Teilzweigen (b) Das Zurücksetzen des Starts eines OR-Join-Knotens führt bei parallelen Verzweigungen mit finaler Auswahl zur Demarkierung von Verliererzweigen.*]

Abbildung 6-12 a). Wird umgekehrt der OR-Join einer Parallelverzweigung mit finaler Auswahl zurückgesetzt, wird die vorausgegangene Auswahl des „Siegerzweiges“ unwirksam. Dem entsprechend müssen die zuvor als „Verlierer“ markierten Zweige reinitialisiert und neu „gestartet“ werden (vgl. Abbildung 6-12 b). Durch diese Maßnahmen erzielen wir eine klare und eindeutige Semantik für Rücksetzoperationen in Ausführungsgraphen. Insbesondere können beim Rücksetzen keine anderen Markierungen für KF-Graphen erzeugt werden, als im Fall der „normalen“ WF-Ausführung. Dieser Forderung muß auch beim vollständigen oder teilweisen Zurücksetzen von Schleifenausführungen, z.B. infolge eines Ad-hoc-Rücksprungs in eine frühere Schleifeniteration hinein, nachgekommen werden. Wie wir nachfolgend an einem Beispiel zeigen werden, können sich die Markierungen der Knoten und Kanten des Schleifenkörpers nach einem Rücksprung komplett anders darstellen als zuvor.

Grundidee des Verfahrens

Basierend auf diesen Überlegungen formulieren wir das Verfahren für das partielle Zurücksetzen einer WF-Ausführung. Dabei klammern wir hierarchische Workflows zunächst aus. Die Grundidee ist, die Ablaufhistorie – beginnend mit dem aktuellsten Eintrag – rückwärts bis zum START-Eintrag von $n_{bwd}(It)$ zu durchlaufen. Für jeden traversierten Historieneintrag, der sich auf einen Knoten des Rücksetzbereichs bezieht, wird die Markierung des betreffenden Knotens zurückgesetzt. Das Verfahren durchläuft im allgemeinen mehrere Iterationen, wobei die WF-Instanz in jeder Iteration bis zum START-Eintrag eines bestimmten *Fixpunktes* der Ablaufhistorie zurückgesetzt wird. Dieser Fixpunkt wird beim Rückwärtslaufen durch die Historie ermittelt. Es handelt sich dabei entweder um den START-Eintrag der Aktivitäteninstanz $n_{bwd}(It)$, vor deren Ausführung der WF zurückgesetzt werden soll, oder aber um den ersten beim Rückwärtslaufen gefundenen START-Eintrag der Ausführung $L_S(k)$ eines Schleifenanfangsknotens L_S für den gilt: Der Rücksetzknoten n_{bwd} ist im Körper der durch die Knoten L_S und L_E:= *endloop*(L_S) gebildeten Schleife enthalten und $L_S(k)$ wurde über eine Schleifenrücksprungkante aktiviert (*<repeated> = True*).

Wurde als Fixpunkt der START-Eintrag eines solchen Schleifenanfangsknoten ermittelt, bedeutet das, daß die Ausführung der Aktivitäteninstanz $n_{bwd}(It)$ in einer früheren Iteration dieser Schleife erfolgt ist. Dementsprechend können alle Aktivitäten der aktuell betrachteten Schleifeniteration zurückgesetzt werden, unabhängig davon, in welcher Anordnungsbeziehung sie zum Rücksetzknoten n_{bwd} stehen[33]. Konkret werden genau diejenigen Instanzen zurückgenommen, deren Historieneinträge dem START-Eintrag von $L_S(k)$ (inklusive) nachfolgen und deren zugehöriger Knoten innerhalb des betreffenden Schleifenkörpers (inkl. Anfangs- und Endknoten) liegt, d.h. parallel ausgeführte Aktivitäteninstanzen

[33] Die Anordnungsbeziehungen zwischen Knoten des Schleifenkörpers und dem Knoten n_{bwd} spielen dann eine Rolle, wenn eine Schleifeniteration nur teilweise zurückgenommen werden soll. Im vorliegenden Fall trifft diese für die Schleifeniteration zu, innerhalb der die Aktivitäteninstanz $n_{bwd}(It)$ ausgeführt wurde.

sind von der Rücksetzoperation nicht betroffen. Bevor mit der Bestimmung des nächsten Fixpunktes fortgefahren werden kann, müssen für den Schleifenkörper die Knoten- und Kantenmarkierungen der vorhergehenden Iteration wiederhergestellt werden. Wurde als Fixpunkt der START-Eintrag von $n_{bwd}(It)$ ermittelt – dieser Fall tritt genau in der letzten Iteration des Verfahrens ein –, werden diejenigen Aktivitäteninstanzen zurückgesetzt, deren Historieneinträge dem START-Eintrag von $n_{bwd}(It)$ nachfolgen und deren Knoten Nachfolger von n_{bwd} (bzgl. Kontroll- und Sync-Kanten) im KF-Graphen sind. Dabei werden die Knoten- und Kantenmarkierungen des Ausführungsgraphen sowie Knotenprioritäten wie oben beschrieben angepaßt. Insgesamt stellt die Konstruktion des Verfahrens sicher, daß Rücksetzoperationen innerhalb eines bestimmten Bearbeitungszweiges keine Auswirkungen auf den Status paralleler Teilzweige haben.

// Das Verfahren RollbackWorkflow setzt die Ausführung des Workflows in den Status zurück, der
// vor Start der Aktivitäteninstanz $n_{bwd}(it)$ Gültigkeit besaß. Für die Formulierung des Algorithmus
// seien die folgenden einfache Funktionen verfügbar:

// ResetNode(n): Setzt den Status des Knotens n auf NS(n):= `NOT_ACTIVATED` *und*
// den Status der von ihm ausgehenden Kanten auf `NOT_SIGNALED`

// ResetPriority(node): Wertet für einen Knoten seine Ausführungspriorität neu aus, nachdem
// eine einmündende Priorisierungskante zurückgesetzt wurde

Algorithmus 6-1 (RollbackWorkflow)

function RollbackWorkflow(InstId, $CFS_{instance}$, $DF_{instance}$, n_{bwd}, It_{bwd})

input

InstId	:	Eindeutiger Bezeichner der WF-Instanz
$CFS_{instance}$	:	KF-Ausführungsgraph (*CFS, NS, ES, It*) der WF-Instanz
$DF_{instance}$	:	Datenkontext der WF-Instanz
n_{bwd}	:	Rücksetzknoten im Ausführungsgraphen
It_{bwd}	:	Zurücksetzen der Bearbeitung vor die It_{bwd}.-te Ausführung von n_{bwd}

begin

// Bestimme die Anfangsknoten aller Schleifen von $CFS_{instance}$, die den Knoten n_{bwd} enthalten
// (Bei geschachtelten Schleifen kann diese Menge mehr als ein Element umfassen.
// Liegt n_{bwd} in keiner Schleife, ist die Menge leer)

$$\text{LoopStartNodes} := \{L_S \in \text{LoopStartNodes}^{CFS} \mid n_{bwd} \in \text{c_succ}^*(L_S) \cap \text{c_pred}^*(L_E) \text{ mit } L_E := \text{endloop}^{CFS}(L_S)\}$$

// Zurücksetzen in mehreren Durchläufen; in jeder Iteration wird bis zu einem bestimmten
// Fixpunkt zurückgesetzt

rollbackCompleted:= False;

repeat

// Fixpunkt (bzw. Aktivitäteninstanz) bestimmen, bis zu dem im aktuellen Durchgang
// zurückgesetzt werden soll. Dazu wird die Ablaufhistorie – beginnend mit dem
// letzten Eintrag – rückwärts durchlaufen bis der nächste Fixpunkt gefunden wird;
// es genügt, nur START-Einträge von Aktivitäteninstanzen zu berücksichtigen

forall$_1$ entry ∈ ExecutionHistory(InstId) **with** (entry=`START`(n_{entry}, It_{entry}, $repeated_{entry}$) **do**$_1$

if ((n_{entry}, It_{entry}) = (n_{bwd}, It_{bwd}) **or** (n_{entry} ∈ LoopStartNodes **and** $repeated_{entry}$ = True)) **then**
fixpoint:= n_{entry}; *// Fixpunkt ist ermittelt*
exit forall-loop$_1$
endif

done$_1$

// Nachfolgerknoten des Fixpunktes bzgl. Kontroll- und/oder Sync-Kanten bestimmen
// (Wird benötigt, um sicherzustellen, daß keine parallelen Aktivitäteninstanzen zurück-
// gesetzt werden)

Succ_Of_Fixpoint:= succ^*(fixpoint) ∪ {fixpoint};

// WF-Instanz bis zum Fixpunkt zurücksetzen. Dazu: Ablaufhistorie erneut – beginnend mit
// letztem Eintrag – rückwärts bis zum `START`*- Eintrag des Fixpunktes durchlaufen; dabei*

```
// alle Einträge kompensieren, deren Knoten in der Menge Succ_Of_Fixpoint enthalten ist.
forall₂ entry ∈ ExecutionHistory(InstId) with (n_entry ∈ Succ_Of_Fixpoint) do₂
    // Aktueller Historieneintrag referenziert Knoten im Rücksetzbereich
    if entry = START(n_entry, It_entry, repeated_entry) then    // Eintrag für Aktivitätenstart
        // Zurücknahme des Starts der Aktivitäteninstanz n_entry(It_entry): Markierung des
        // Knotens in NOT_ACTIVATED abändern und Schreiboperationen vorge-
        // geschalteter Aktionen auf dem Datenkontext zurücknehmen (vgl. Tabelle 6-1)
        RollbackActivity(InstId, CFS_instance, DF_instance, n_entry, It_entry, BEFORE_START);
        // Falls es sich bei dem zurückzusetzenden Knoten um den OR-Join-Knoten
        // einer parallelen Verzweigung mit finaler Auswahl handelt, müssen Ver-
        // liererzweige reinitialisiert und neu gestartet werden (vgl. Abbildung 6-12 b)
        if (V_in^n_entry = ONE_OF_ALL and V_out^split(n_entry) = ALL_OF_ALL) then
            branchNodes:= c_succ*(split(n_entry)) ∩ c_pred*(n_entry);
            forall (n ∈ branchNodes with NS(n) = SKIPPED) do
                ResetNode(n)
            done
            // Alle ausgehenden Kanten des AND-Split-Knotens setzen
            forall e ∈ E with Id_source^e = split(n_entry) do
                ES(e):= TRUE_SIGNALED
            done
        endif
        Entferne den START-Eintrag entry aus der Ablaufhistorie
        // Ist der aktuelle Fixpunkt erreicht?
        if (n_entry = fixpoint) then
            if (n_entry , It_entry) = (n_bwd , It_bwd) then
                rollbackCompleted:= True;
            end
            exit-forall-loop₂
        endif
    else if entry = END(n_entry, It_entry, state_entry, code_entry) then    // Eintrag für Aktivitätenende
        // Zurücknahme des Abschlusses der Aktivitäteninstanz n_entry(It_entry): Markierung
        // des Knotens in RUNNING abändern und Status ausgehender Kanten auf
        // NOT_SIGNALED setzen; Schreiboperationen aufgerufener Anwendungskom-
        // ponenten und nachgeschalteter Aktionen rückgängig machen (vgl. Tabelle 6-1)
        RollbackActivity(InstId, CFS_instance, DF_instance, n_entry, It_entry, AFTER_START)
        // Ausführungsprioritäten anpassen
        forall e ∈ E with (Id_source^e = n_entry and ET^e = PRIORITY_E ) do
            ResetPriority(Id_dest^e)
        done
        // Handelt es sich bei dem zurückzusetzenden Knoten um einen OR-Split-Knoten
        // müssen die zuvor als SKIPPED markierten Zweige neu initialisiert werden.
        // (vgl. Abbildung 6-12 a)
        if (V_out^n_entry = ONE_OF_ALL) then
            forall e ∈ E with (Id_source^e = n_entry and SC^e ≠ code_entry) do
                branchNodes:= c_succ*(Id_dest^e) ∩ c_pred*(join(n_entry));
                forall (n ∈ branchNodes) do
                    ResetNode(n)
                done
            done
        endif
        Entferne den END-Eintrag entry aus der Ablaufhistorie
    endif
done₂
```

```
        // Nach dem Zurücksetzen der letzten Schleifeniteration werden für den Schleifenkörper
        // diejenigen Knoten- und Kantenmarkierungen wiederhergestellt, die nach Beendigung
        // der vorletzten (noch nicht zurückgesetzten) Iteration Gültigkeit besaßen.

        if rollbackCompleted = False then
            Bestimme in der Ablaufhistorie den letzten START-Eintrag des Schleifenanfangs-
            knotens L_S:= fixpoint (ein solcher Eintrag existiert in jedem Fall).

            Durchlaufe die Ablaufhistorie – beginnend mit diesem START-Eintrag – vorwärts bis
            zum END-Eintrag des Knotens L_E:= endloop^CFS(L_S): Führe dabei für alle END-Einträge
            von Aktivitäten, die zwischen den Knoten L_S und L_E liegen (beide inkl.), die resultieren-
            den Folgemarkierungen erneut auf dem KF-Ausführungsgraphen aus; dabei werden nicht
            nur beendete Knoten (als COMPLETED) und ihre ausgehenden Kanten entsprechend mar-
            kiert, sondern – bei Auftreten bedingter Verzweigungen oder paralleler Verzweigungen
            mit finaler Auswahl – auch die Markierungen von Knoten und Kanten nicht gewählter
            Zweige (SKIPPED bzw. NOT_SIGNALED) wiederhergestellt.
        endif

    until rollbackCompleted = True

end // RollbackWorkflow
```

Wir wollen die Anwendung dieses Verfahrens kurz anhand von zwei Beispielen erörtern.

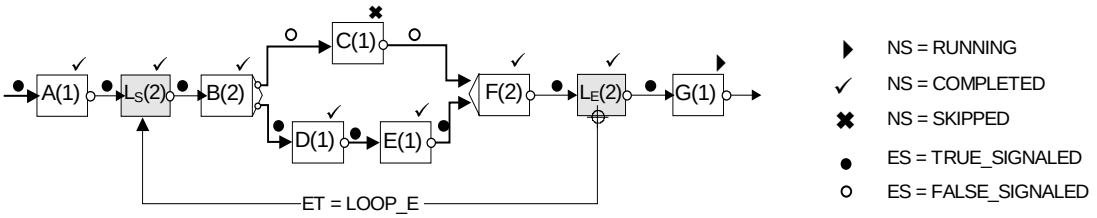
Beispiel 6-4 (Umsetzung eines automatischen Fehlerrücksprungs):

Ein einfaches Beispiel zeigt Abbildung 6-8. Der dargestellte Fehlerrücksprung kann durch den Aufruf von *RollbackWorkflow*(*, F, 1) verwirklicht werden. Da der Rücksprungknoten F in keiner Schleife liegt, ergibt sich als erster Fixpunkt F(1). Das Verfahren terminiert bereits nach einem Durchlauf. Das Beispiel zeigt auch, daß durch das Zurücksetzen einzelner Teilzweige der Status paralleler Bearbeitungszweige nicht beeinträchtigt wird.

Beispiel 6-5 (Umsetzung eines Ad-hoc-Rücksprungs):

Wir geben nun ein Beispiel für die Umsetzung eines Ad-hoc-Rücksprungs. Dazu sei der folgende KF-Ausführungsgraph gegeben (In Klammern steht jeweils der aktuelle Iterationszähler der Knoten).

Dynamisches Zurücksetzen der WF-Instanz vor die erstmalige Ausführung von Aktivität C!



Infolge eines Ad-hoc-Rücksprungs soll die Bearbeitung in den Prozeßzustand zurückgesetzt werden, der vor erstmaliger Ausführung des Knotens C Gültigkeit besaß. Dazu wird das teilweise Zurücksetzen der Schleifenausführung notwendig (vollständiges Zurücksetzen der 2. und partielles Zurücksetzen der 1. Iteration). Die Umsetzung des Rücksprungs erfolgt durch Aufruf von *RollbackWorkflow*(*, C, 1). Dazu sei die folgende Ablaufhistorie gegeben:

Ablaufhistorie (vor Durchführung des Rücksprungs; vereinfachte Darstellung):

... END(A,1), START(L_S,1, False), END(L_S,1), START(B,1), END(B,1, Auswahl oberer Zweig), START(C,1), END(C,1), START(F,1), END(F,1), START(L_E,1), END(L_E,1, Schleifenbedingung erfüllt), START(L_S,2, True), END(L_S,2), START(B,2), END(B,2, Auswahl unterer Zweig), START(D,1), END(D,1), START(E,1), END(E,1), START(F,2), END(F,2), START(L_E,2), END(L_E,2, Schleifenbedingung nicht erfüllt), START(G,1)

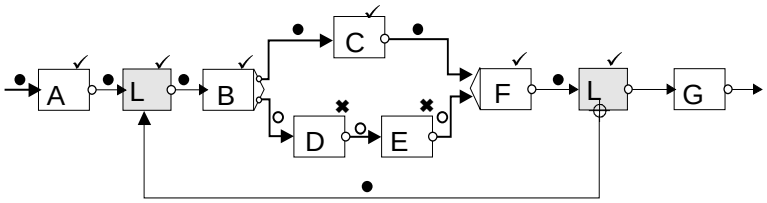
Das Rücksetzverfahren läuft dann in folgenden Schritten ab:

1. Anfangsknoten aller den Knoten C umschließenden Schleifen bestimmen: {L_S}
2. Fixpunkt für den ersten Durchlauf des Rücksetzverfahrens ermitteln: L_S (bzw. L_S(2))
3. Nachfolgerknoten des Fixpunktes (über Kontroll- und Sync-Kanten) bestimmen: Succ_Of_Fixpoint = {L_S, B, C, D, E, F, L_E, G}

4. Rücksetzen der Ausführung bis vor $L_S(2)$ (Historieneinträge der Knoten aus Succ_Of_Fixpoint zum Rücksetzen verwenden und anschließend entfernen). Es ergibt sich die folgende Ablaufhistorie:

Ablaufhistorie: ... END(A,1), START(L_S,1, False), END(L_S,1), START(B,1), END(B,1, Auswahl oberer Zweig), START(C,1), END(C,1), START(F,1), END(F,1), START(L_E,1), END(L_E,1, Schleifenbedingung erfüllt)

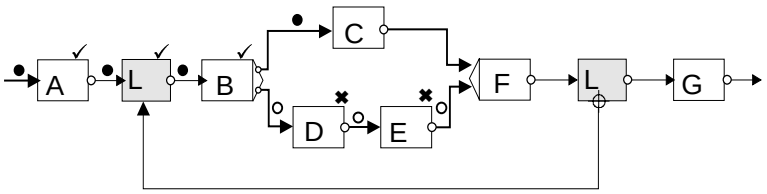
5. Knoten- und Kantenmarkierungen, die nach Beendigung der vorletzten (hier der 1. Schleifeniteration) Gültigkeit besaßen, wiederherstellen:



6. Neuen Fixpunkt für den zweiten Durchlauf des Rücksetzverfahrens bestimmen: **C** (bzw. C(1))
7. Nachfolgerknoten des Fixpunktes (über Kontroll- und Sync-Kanten) ermitteln: Succ_Of_Fixpoint = {C, F, L_E, G}
8. Rücksetzen der Ausführung bis vor C(1) (Historieneinträge der Knoten aus Succ_Of_Fixpoint zum Rücksetzen verwenden und anschließend entfernen). Es ergibt sich die folgende Ablaufhistorie:

Ablaufhistorie:... END(A,1), START(L_S,1, False), END(L_S,1), START(B,1), END(B,1, Auswahl oberer Zweig)

Der Ausführungsgraph stellt sich nach Durchführung von Schritt 8 wie folgt dar (bevor mit der Ausführung fortgefahren werden kann, muß der Graph unter Verwendung der ADEPT-Markierungsregeln neu bewertet werden; im Anschluß daran ist der Knoten C aktiviert).



Das Verfahren würde unverändert anwendbar bleiben, wenn der dargestellte KF-Graph nur den Teilzweig einer parallelen Verzweigung bildet, von einer äußeren Schleife umgeben wird oder beides davon zutrifft.

Die beiden Beispiele verdeutlichen nochmals, daß beim Rücksetzen einer WF-Instanz Anpassungen von WF-Kontrolldaten und WF-relevanten Daten vorgenommen werden müssen, um anschließend mit der Kontrolle fortfahren zu können. Als wesentliches Kriterium für die Korrektheit von Rücksetzoperationen haben wir gefordert, daß durch das Zurücksetzen einer WF-Instanz keine anderen Markierungen und Zustände entstehen können als im Verlauf der normalen WF-Ausführung. Dadurch ergibt sich eine eindeutige und klare Rücksetzsemantik, die sowohl die Umsetzung von vormodellierten Rücksprüngen wie auch von Ad-hoc-Rücksprüngen zuläßt. Das vorgestellte Rücksetzverfahren ist für beliebig strukturierte KF-Ausführungsgraphen, bei denen Schleifen, Verzweigungen und Sequenzen geschachtelt und in Kombination miteinander auftreten können, anwendbar.

b) Rücksetzen bei hierarchischen Workflows

Abschließend diskutieren wir einige wichtige Aspekte, die wir bei Anwendung von Rücksetzoperationen auf hierarchische WFs beachten müssen. Im allgemeinen sollte es möglich sein, die Bearbeitung eines Toplevel-WF vor die Ausführung einer beliebigen Aktivitäteninstanz der obersten oder einer untergeordneten Ausführungsebene zurückzusetzen. Dabei können interne und ggf. auch externe Kompensationsmaßnahmen auf mehreren Hierachiestufen notwendig werden.

Wie wir bereits in Kapitel 4 beschrieben haben, wird ein Toplevel-WF auf einen Baum von Instanzen elementarer und komplexer Aktivitäten abgebildet, den sog. Instanzabhängigkeitsgraphen (IAG). Der IAG wird jeweils bei Erzeugung neuer Aktivitäteninstanzen um neue Knoten erweitert, wächst also dynamisch im Verlauf der WF-Ausführung. Jeder Nichtblattknoten beschreibt eine (Sub-) WF-Instanz, und seine direkten Nachfolger entsprechen den von ihr koordinierten elementaren und/oder komplexen Aktivitäteninstanzen, d.h. hierarchische Aufrufbeziehungen spiegeln sich direkt im IAG wider (vgl.

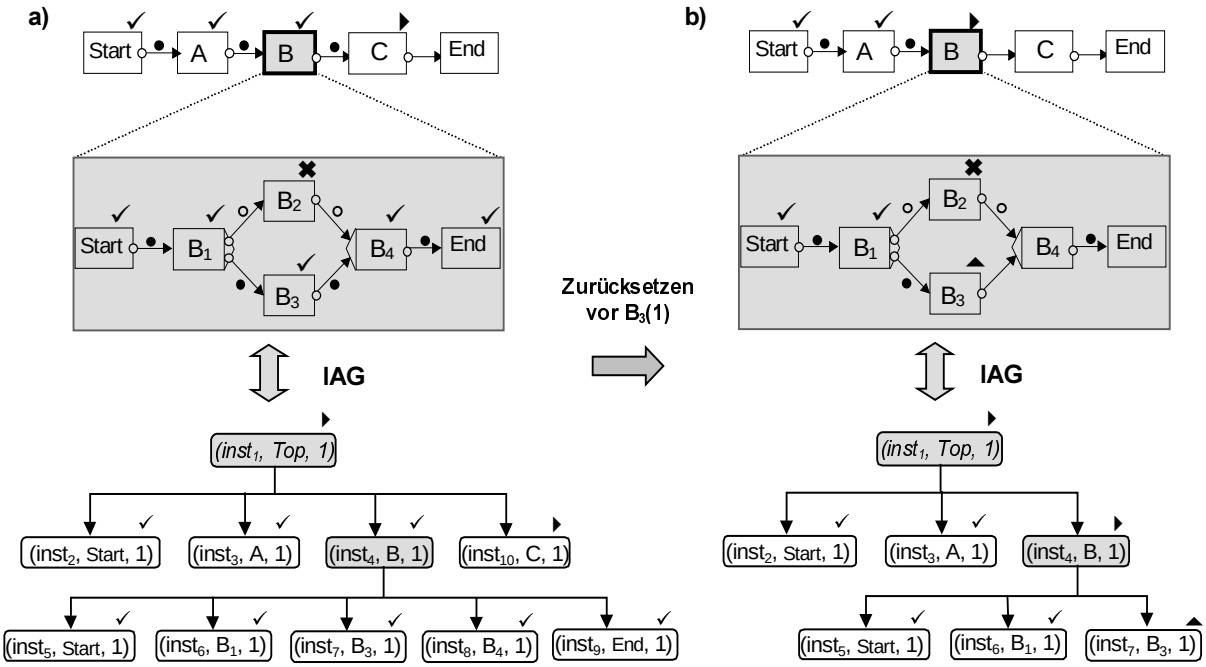


Abbildung 6-13: *Zurücksetzen bei hierarchischen Workflows*

Abbildung 6-13 a). Abgesehen davon besitzt jede (Sub-)WF-Instanz mit Kennung *inst* ihren eigenen Ausführungsgraphen CFS_{inst} (inkl. Ablaufhistorie) und Datenkontext DF_{inst}.

Soll für den Ausführungsgraphen CFS_{inst} einer noch laufenden (Sub-)WF-Instanz die Bearbeitung partiell zurückgesetzt werden, kann prinzipiell wie in Abschnitt a) beschrieben vorgegangen werden. Allerdings müssen beim Zurücksetzen von Instanzen komplexer Aktivitäten nicht nur deren Effekte in Bezug auf die Ausführung des übergeordneten WF zurückgenommen werden, sondern auch die Sub-WF-Bearbeitungen selbst. Grundlage hierfür bilden die Ausführungsgraphen der jeweiligen Sub-Workflows und ihre Ablauf- und Datenhistorien. Rücksetzoperationen pflanzen sich somit – ähnlich wie man es von *Nested Transactions* [JaKe97] her kennt – von Aktivitäteninstanzen höherer auf Instanzen niedrigerer Ebenen fort, bis schließlich der gewünschte Zielzustand erreicht ist. Im IAG wird dies zum Ausdruck gebracht, indem die Knoten zurückgesetzter Aktivitäteninstanzen und ihre ausgehenden Kanten sukzessive entfernt werden. Das heißt der IAG verkleinert sich durch das Rücksetzen.

ADEPT erlaubt es, die Bearbeitung eines WF in einen Zustand zurückzuversetzen, der vor Ausführung einer untergeordneten Aktivitäteninstanz beliebiger Hierarchiestufe gültig war. Das bedeutet, daß – mit Ausnahme der Wurzel – jeder beliebige Knoten des IAG das Ziel eines (Ad-hoc-) Rücksprungs sein kann. Folglich muß auch ein bereits abgeschlossener Sub-WF partiell zurückgesetzt werden können, was wiederum nur möglich ist, wenn auch der Status der übergeordneten WF-Instanz entsprechend abgeändert wird. Wir skizzieren anhand eines einfachen Beispiels, welche Erweiterungen des vorgestellten Verfahrens notwendig werden, um solche mehrstufigen Rücksetzoperationen korrekt durchführen zu können. Der in Abbildung 6-13 a) dargestellte Toplevel-WF $inst_1$ kann in jeden Bearbeitungszustand zurückgesetzt werden, in dem er sich vor Start einer der untergeordneten Aktivitäteninstanzen $inst_2$ - $inst_{10}$ befand. Abbildung 6-13 b) zeigt dieselbe WF-Instanz, nachdem sie in den Zustand vor (erstmaliger) Bearbeitung der untergeordneten Aktivität B_3 zurückgesetzt wurde. Dabei wurden nicht nur der Datenkontext und die Markierungen des Ausführungsgraphen der Sub-WF-Instanz $inst_4$ angepaßt, sondern es mußten auch entsprechende Zustandsmodifikationen auf Ebene der übergeordneten Instanz $parent(inst_4)$ (= $inst_1$) vorgenommen werden (Abbruch und Zurücksetzen von `C` und Änderung der Zustandsmarkierung von `B` in `RUNNING`). Andernfalls hätte sich durch das Zurücksetzen des Sub-WF ein Gesamtzustand ergeben, der so im Verlauf der normalen WF-Ausführung nie erreichbar gewesen wäre: Der Super-WF könnte fortschreiten und ggf. terminieren, noch bevor die zurückgesetzte Sub-WF-Instanz $inst_4$ beendet wird. Dies würde jedoch im Widerspruch zu dem von ADEPT geforderten Korrektheitskriterium für Rücksetzoperationen stehen

Allgemein erfolgt das Zurücksetzen des Gesamt-WF vor die Ausführung einer beliebigen Aktivitäteninstanz ($instId_{bwd}$, n_{bwd}, It_{bwd}) des IAG nach folgendem Grundprinzip: Zunächst wird im IAG der Weg vom Rücksetzknoten ($instId_0$, n_0, It_0) zum Wurzelknoten ($instId_k$, n_k, It_k) des Baumes bestimmt (sog. Rücksetzpfad):

$$(instId_{bwd}, n_{bwd}, It_{bwd}) \leftarrow (instId_1 := parent(instId_0), n_1, It_1) \leftarrow ... \leftarrow (instId_k := parent(instId_{k-1}), n_k, It_k).$$

Dann wird zunächst der Ausführungsgraph der Sub-WF-Instanz $instId_1$ vor Start der It_{bwd}-ten Bearbeitung des Knotens n_{bwd} zurückgesetzt. (Im Beispiel: Zurücksetzen des Sub-WF $inst_4$ vor die erstmalige Bearbeitung von B_3.) Befand sich der Sub-WF $instId_1$ zuvor im Status `COMPLETED`, d.h. war er bei Anwendung der Rücksetzoperation bereits abgeschlossen, muß die Ausführung auf der nächst höheren Ebene (d.h. der Ausführungsgraph der WF-Instanz $instId_2$: = $parent(instId_1)$) ebenfalls partiell zurückgesetzt werden, und zwar in den Zustand der nach Start des Sub-WF $instId_1$ (d.h. nach Start der It_1-ten Ausführung des Knotens n_1 der WF-Instanz $instId_2$) gültig war[34]. (Im Beispiel: Zurücksetzen des Toplevel-WF $inst_1$ in den Zustand nach erstmaligem Start von B.) Nach demselben Prinzip pflanzen sich Rücksetzoperation entlang des Rücksetzpfades fort, bis schließlich eine (Sub-)WF-Instanz erreicht wird, die sich noch nicht im Zustand `COMPLETED` befindet. Dies trifft in jedem Fall für den Endknoten des Rücksetzpfades, der die Instanz des Toplevel-WF repräsentiert, zu.

6.2.5 Abschließende Bemerkungen

Auf Basis der vorgestellten Konzepte läßt sich ein breites Spektrum an Rücksetzoperationen umsetzen. Darüber hinaus können mehrere Ad-hoc-Rücksetzoperationen und/oder andere dynamische WF-Änderungen in einer Änderungstransaktion zusammengefaßt werden (siehe Kapitel 7). Dadurch lassen sich zur Laufzeit komplexe Ausnahmebehandlungen verwirklichen, wie das gleichzeitige Zurücksetzen mehrerer Teilzweige einer parallelen Aufsplittung oder das Zurücksetzen des WF, seine strukturelle Abänderung und die anschließende Wiederaufnahme der Bearbeitung auf Basis des modifizierten Ausführungsgraphen. Wie wir später zeigen werden, müssen die vorgestellten Konzepte in mehrerlei Hinsicht erweitert werden, wenn sie auch für das Zurücksetzen bei dynamisch abgeändertem Ausführungsgraphen anwendbar bleiben sollen.

6.3 Sonstige Aspekte

Die in den beiden vorangegangenen Abschnitten beschriebenen Konzepte basieren auf der Idee, dem Modellierer spezielle Konstrukte in die Hand zu geben, auf deren Basis er oder sie ein flexibles Ausführungsverhalten modellieren kann. Um eine klare und eindeutige Ausführungssemantik zu erhalten und um die Ausführungskomponente des WfMS „schlank“ zu gestalten, werden diese Konstrukte in eine ausführbare Darstellung des ADEPT-Basismodells übersetzt. Ergänzend zu dieser Vorgehensweise gibt es weitere Ansatzpunkte, mit denen sich die A-priori-Flexibilität [Joer98] in WfMS erhöhen läßt. Wir wollen einige von ihnen der Vollständigkeitshalber kurz erwähnen.

Zum Teil läßt sich ein flexibles Ausführungsverhalten auch bereits dadurch erzielen, daß verschiedene *Ausführungsvarianten*, von denen der Anwender zur Laufzeit eine auswählen kann, explizit in das Kontrollflußschema hineinmodelliert werden (z.B. als Teilzweige einer bedingten Verzweigung). Dies ist trivialerweise auch in ADEPT, unter Verwendung der Konstrukte des Basismodells, möglich. Darüber hinaus kann die Flexibilität in der Bearbeitung einzelner Aktivitäten erhöht werden, indem die zu ihrer Ausführung herangezogenen Anwendungsprogramme bzw. Sub-Workflow-Modelle erst dynamisch während der WF-Ausführung gebunden werden. Dieses *späte Binden* kann z.B. benutzergesteuert oder abhängig von systemseitig überprüfbaren Bedingungen (z.B. Wertebelegungen von Datenelementen) erfolgen. Die hierarchische Struktur und Schachtelungstiefe eines WF ergeben sich dann erst im Verlauf seiner Ausführung, wodurch ein höheres Maß an Flexibilität erreichbar ist als bei statischem Binden [Han97a]. Darüber hinaus können Änderungen in der Implementierung von Aktivitätenprogrammen oder der Definition von (Sub-)WF-Modellen bei der Ausführung eines WF ohne zusätzliche Überprüfungen berücksichtigt werden, solange die betreffende Komponente noch nicht gebunden bzw. aufgerufen wurde. Aufgrund der gewählten Trennung von Vorlagenschnittstelle und

[34] Das bedeutet, daß die Effekte vorgeschalteter Aktionen (z.B. Schreiboperationen) nicht zurückgenommen werden sollen. Dies erreicht man durch geringfügige Modifikationen des in a) beschriebenen Verfahrens.

-implementierung (vgl. Kapitel 3) ist das späte Binden von Aktivitätenimplementierungen in ADEPT vom Prinzip her einfach umsetzbar. Voraussetzung dafür ist allerdings, daß die Komponenten plug-in-kompatibel (siehe Kapitel 3) zur statisch festgelegten Vorlagen-/Aktivitätenschnittstelle sind, also keine davon abweichenden obligaten Eingaben verlangen bzw. Ausgaben machen.

In der betrieblichen Praxis trifft man häufig auf Arbeitsabläufe, die sowohl determinierbare als auch nicht bzw. nur ungefähr beschreibbare Anteile besitzen. Falls diese bei der WF-Modellierung klar voneinander abgrenzbar sind, kann zur Unterstützung nur ungefähr beschreibbarer Teilabläufe auf das Konzept des *späten Modellierens* zurückgegriffen werden [HHJS97]. Hier können bei der Modellierung des WF komplexe Aktivitäten (sog. *Planungsaktivitäten*) vereinbart werden, deren zugehöriges Sub-WF-Modell zunächst nicht oder zumindest nicht vollständig spezifiziert sein muß. Vielmehr kann die Erstellung bzw. Erweiterung dieses Modells instanzspezifisch auch noch während der Laufzeit durch berechtigte Akteure erfolgen. Der Start des Sub-WF wird dann solange hinausgezögert, bis der Vorgang des späten Modellierens abgeschlossen ist. Um Planungsaktivitäten von normalen Aktivitäten unterscheiden zu können, müssen ihre Knoten auf Modellebene entsprechend gekennzeichnet werden. Ebenso muß zur Ausführungszeit durch ein Attribut angezeigt werden, ob die Aktivität bereits gestartet werden kann oder nicht. Grundlage für das Konzept des späten Modellierens, das nicht mit dynamischen Änderungen laufender WF-Instanzen zu verwechseln ist, bildet das späte Binden von Sub-Workflow-Komponenten.

Ein weiterer wichtiger Aspekt der WF-Modellierung betrifft die Festlegung der *Freiheitsgrade*, die Akteuren im Hinblick auf die spätere WF-Ausführung eingeräumt werden sollen. Beispielsweise können im ADEPT-Ansatz bei der Definition einzelner Aktivitäten Angaben dazu gemacht werden, ob ihre Bearbeitung zur Laufzeit übersprungen, wiederholt, abgebrochen oder zurückgesetzt werden darf. Entsprechende Informationen werden von der Ausführungskomponente berücksichtigt und schränken die Anwendbarkeit dynamischer Änderungsoperationen (vgl. Kapitel 7) ein. Dasselbe gilt auch in Bezug auf die bei der Definition von Vorlagen festgelegten Eigenschaften von Ein- und Ausgabeparametern (obligate vs. optionale Parameter, Nachforderbarkeit von Parametern). Sie haben maßgeblichen Einfluß darauf, welche begleitenden Modifikationen im Kontext einer beabsichtigten WF-Änderung (z.B. Löschen einer Aktivität) notwendig werden.

6.4 Zusammenfassung

In diesem Kapitel haben wir Möglichkeiten für die flexible Gestaltung der Definition und Ausführung von WFs kennengelernt. Den Schwerpunkt bildeten dabei Konzepte zur Vormodellierung ausnahmebedingter Abweichungen vom Standardablauf. Solche Abweichungen sind Ausdruck der vielen Arbeitsabläufen zugrundeliegenden inhärenten Variabilität [RHD98, StMi95] und treten in der Praxis deshalb häufig auf. Je besser es gelingt, sie bereits auf Modellebene zu fassen, desto weniger Ad-hoc-Anpassungen werden zur Laufzeit notwendig.

Die vorgestellten Konzepte stellen deshalb einen wichtigen Beitrag zur Erhöhung der A-priori-Flexibilität in adaptiven WfMS dar. Darüber hinaus tragen sie zu einer verbesserten Strukturierung der WF-Modelle bei, indem sich Normalfälle von seltener auftretenden Sonder- und Ausnahmefällen trennen lassen [Ober96]. Wir haben gezeigt, wie und unter welchen Voraussetzungen sich Vorwärts- und Rückwärtssprünge unterschiedlicher Semantik mit den Konstrukten des ADEPT-Basismodells abbilden und später an der Benutzerschnittstelle adäquat umsetzen lassen: Berechtigte Akteure, die nicht notwendigerweise aktuell prozeßbeteiligt sein müssen, können in die Kontrolle des Ablaufs eingreifen, indem sie zukünftige Arbeitsschritte vorzeitig ausführen (Vorziehen von Arbeitsschritten) oder den Ablauf partiell zurücksetzen, um die Bearbeitung in einem früheren Prozeßzustand wiederaufzunehmen (Zurückerlangen der Kontrolle). Auch das Überspringen einzelner Aktivitäten oder einer Gruppe von Aktivitäten (mit oder ohne Nachholen der Bearbeitung) ist vormodellierbar, vorausgesetzt die sie verursachenden Ausnahmen und der Kontext ihres Auftretens sind a priori bekannt. Sprünge können nicht nur durch Benutzer ausgelöst werden, sondern auch an vorhersehbare Ereignisse, wie dem Fehlschlag einer Aktivität, geknüpft werden. In einem solchen Fall kann die Ausnahmebehandlung weitgehend automatisch erfolgen. Die vorgestellten Abweichungen und Sprünge sind nicht nur für die Modellierung von WFs relevant, sondern nehmen auch im Kontext dynamischer WF-Änderungen eine wichtige Rolle ein, da zur Laufzeit auftretende unvorhersehbare Ausnahmen oder Benutzerwünsche ähnliche Behandlungsformen erforderlich machen können.

Wichtige Aspekte im Umgang mit planbaren Abweichungen betreffen ihre Abbildung auf Modellebene und die anschließende Umsetzung in eine ausführbare Darstellung des ADEPT-Basismodells. Der Modellierer beschreibt Abweichungen vom „normalen“ Kontrollfluß in einer abstrakten und verständlichen Notation. Hierzu tragen „High-Level“-Konstrukte wie *Shortcut* oder *RegainControl* bei. Die für die Verwendung dieser Konstrukte festgelegten Restriktionen garantieren eine klare und eindeutige Ausführungssemantik. Des weiteren resultiert aus der getrennten Beschreibung von Normalablauf und ausnahmebedingten Abweichungen eine verbesserte Strukturierung der Modelle. Bei der Modellbildung wird der WF-Modellierer durch syntaxgesteuerte, graphische Editoren der ADEPT-Entwicklungsumgebung [Kirs96, Mich99] unterstützt. Sie bieten neben den Standardkonstrukten des ADEPT-Basismodells auch die beschriebenen Ausnahmekonstrukte (Shortcut-, RegainControl- und Fehlerkanten) an. Dabei können verschiedene Kantentypen (inkl. Ausnahmekanten) dynamisch ein- oder ausgeblendet werden. Es lassen sich ausschließlich syntaktisch korrekte Modelle mit regelmäßiger Blockstrukturierung und korrekter Verwendung von Spezialkanten erzeugen. Darüber hinaus können bestimmte Arten von logischen Fehlern (z.B. semantische Konflikte zwischen unterschiedlichen Sprüngen) sowie Integritätsverletzungen WF-relevanter Daten vorzeitig erkannt und damit vermieden werden. Schließlich werden auch die geforderten Dynamik-Eigenschaften wie Terminierung des WF oder Aktivierbarkeit von Knoten per Konstruktion bzw. zusätzliche Analysen der Modelle gewährleistet (vgl. Kapitel 4).

Eine wichtige Voraussetzung hierfür bildet die Vorübersetzung der modellierten Ablaufvorlage in eine ausführbare Darstellung des Basismodells. Die Integration von Konstrukten wie *Shortcut* oder *RegainControl* in das Basismodell, würde zu hohe Anforderungen an die Ausführungskomponente des WfMS stellen: Unterschiedliche Semantiken müßten berücksichtigt und Abhängigkeiten oder Inferenzen aufgelöst werden. Jedes einzelne Konstrukt müßte aufwendig interpretiert werden. Zusätzlich würde sich die Komplexität von Modellanalysen, z.B. die Korrektheit von DF-Schemata betreffend, drastisch erhöhen. Während dies zur Modellierungszeit noch tolerierbar wäre, würden sich spätestens bei der Durchführung dynamischer WF-Änderungen, wo vergleichbare Analysen notwendig sind, erhebliche Performanzprobleme ergeben. Abgesehen davon wären dynamische Ablaufänderungen bei einer zu großen Vielfalt an Konstrukten für den „normalen“ Anwender kaum mehr beherrschbar. Aus diesen Gründen verfahren wir ähnlich wie im Bereich der Programmiersprachen. Komplexe Anweisungen einer (erweiterbaren) Ablaufbeschreibungssprache werden in einen begrenzten Satz einfacher Konstrukte übersetzt, die schnell und korrekt ausführbar sind.

Die korrekte Umsetzung komplexer Konstrukte (wie Shortcut-Kanten) wird durch die für ihre Verwendung festgelegten Restriktionen sichergestellt. Die zusätzlich notwendigen DF-Analysen können gezielt, unter Ausnutzung der Semantik der verschiedenen Sprungkanten, erfolgen. Um auch zur Ausführungszeit zwischen regulären Arbeitsschritten und Ausnahmeschritten unterscheiden zu können – dies ist ein wichtiger Aspekt für die praktische Tauglichkeit der beschriebenen Konzepte –, haben wir das ADEPT-Basismodell um einfache Möglichkeiten zur Abbildung präferierter Ausführungsreihenfolgen erweitert. Ebenso war zum Zweck der Vormodellierung von Rücksetzoperationen ein neuer Kantentyp aufzunehmen. Ansonsten sind die Konstrukte des Basismodells ausreichend, um das gewünschte Verhalten ausdrücken zu können. Insbesondere kann auch für vormodellierte Abweichungen das zukünftige Systemverhalten bereits auf Entwicklungsebene analysiert, animiert und validiert werden.

Die vollständige Berücksichtigung aller möglichen Ausnahmesituationen auf Modellebene ist im allgemeinen nicht möglich. Hauptgrund ist, daß nicht immer alle Ausnahmen vorhersehbar und damit planbar sind. Hinzu kommt die hohe Variabilität vieler Prozesse, die eine exakte Vormodellierung schwierig, wenn nicht gar unmöglich gestalten. Spätestens für solche Fälle benötigen wir Operationen die es uns gestatten, WF-Instanzen auch noch während ihrer Ausführung dynamisch abändern zu können. Hierauf gehen wir in den folgenden Kapiteln ein.

Teil III

Dynamische Workflow-Änderungen mit $ADEPT_{flex}$

7 Die ADEPT$_{flex}$-Änderungsoperationen

Ein adaptives WfMS muß verschiedene Arten von WF-Änderungen im laufenden Betrieb zulassen [RHD98]. Im allgemeinen sollten sowohl Modifikationen einer WF-Vorlage und – soweit sinnvoll und möglich – ihre Propagierung auf davon abgeleitete, bereits laufende WF-Instanzen, als auch *Ad-hoc-Änderungen* einzelner WF-Instanzen unterstützt werden. *Vorlagenänderungen* werden vom WF-Modellierer durchgeführt, wenn die Vorlage an Änderungen in den Geschäftsprozessen angepaßt werden soll. Laufzeitabweichungen bezogen auf einzelne WF-Instanzen erfolgen dagegen durch prozeßbeteiligte Akteure, etwa zur Behandlung von Ausnahmesituationen, zum späten Modellieren zuvor offengelassener WF-Anteile oder automatisch durch das WfMS beim Auftreten bestimmter Ereignisse. Die integrierte Unterstützung dieser verschiedenen Änderungsarten und -anforderungen in einem WfMS ist nur möglich, wenn für ihre Anwendung eine gemeinsame, formale Basis vorhanden ist. Andernfalls resultieren spezielle Lösungen für bestimmte Arten von Änderungen, die für sich isoliert betrachtet zwar anwendbar sind, nicht aber in Kombination miteinander funktionieren.

In Teil III dieser Arbeit stellen wir mit ADEPT$_{flex}$ ein formales Grundgerüst für die Verwirklichung von WF-Änderungen – auf Vorlagen- und auf Instanzebene – vor. Ausgangspunkt bildet das in Teil II eingeführte ADEPT-Basismodell. Hiervon ausgehend definieren wir eine *vollständige* Menge an Basisänderungsoperationen, mit denen sich sowohl Vorlagen- als auch Instanzänderungen definieren lassen. Durch Anwendung dieser Operationen können die *Struktur*, die *Attribute* und ggf. der *Status* von WF-Graphen bzw. WF-Ausführungsgraphen modifiziert werden. Oberstes Ziel dabei ist der Erhalt der *Korrektheit* und *Konsistenz*. Unterstützt werden Änderungsoperationen für

- das Hinzufügen, Löschen oder Verschieben einzelner Aktivitäten oder ganzer Kontrollblöcke,
- die Bildung und Auflösung von Blockstrukturen,
- das Hinzufügen und Entfernen von Synchronisations-, Priorisierungs- und Fehlerkanten,
- das Hinzufügen und Entfernen von Datenelementen und Datenflußkanten
- die Modifikation von WF-Attributen (z.B. Knotenprioritäten, Aktivitätenprogramme).

Diese *Basisänderungsoperationen* können in Kombination miteinander angewendet werden, um komplexe Änderungsoperationen zu verwirklichen. Sämtliche von ADEPT$_{flex}$ bereitgestellten Änderungsoperationen sind nicht nur auf die Ausführungsgraphen einzelner WF-Instanzen anwendbar, sondern können auch für Änderungen auf Vorlagenebene herangezogen werden. Sie bilden damit die Grundlage für die eventuelle Propagierung von Vorlagenänderungen auf bereits laufende WF-Instanzen [RHD98]. Im vorliegenden Kapitel stellen wir die von ADEPT$_{flex}$ angebotenen Änderungsoperationen vor. Auf ihrer Grundlage lassen sich weiterführende Konzepte für die Realisierung vorgangsorientierter, adaptiver Anwendungssysteme realisieren. So können Änderungen auf Instanzebene bei Bedarf nur temporär eingebracht und später – z.B. nach Ausführung von Schleifenrücksprüngen – wieder automatisch entfernt werden. Dem gegenüber stehen Änderungen, die für die komplette Restdauer der WF-Instanz erhalten bleiben sollen. Beide Konzepte sind essentiell für die Unterstützung dynamischer WF-Änderungen in Verbindung mit Schleifen- und Fehlerrücksprüngen. Sie wurden in bisherigen Ansätzen allerdings aufgrund des Fehlens oder der impliziten Realisierung dieses Konstruktes noch nicht einmal als Problem erkannt. Weitere wichtige Leistungsmerkmale des ADEPT$_{flex}$-Ansatzes, die hier aus Platzgründen nicht im Detail dargestellt werden, betreffen die Behandlung konkurrierender Änderungen [Weil97], das korrekte Zurücksetzen von WF-Instanzen bei strukturell modifiziertem Ausführungsgraphen [Weil97] und die Bereitstellung von Programmierschnittstellen für die Anwendung von Änderungsdiensten [Hens97].

Abschnitt 7.1 faßt wesentliche Ziele des Entwurfs der ADEPT$_{flex}$-Änderungsoperationen zusammen. Anschließend behandeln wir in Abschnitt 7.2 wichtige Grundlagen, die wir für eine formale Darstellung benötigen. In den Abschnitten 7.3 und 7.4 diskutieren wir am Beispiel des (dynamischen) Einfügens und Löschens von Aktivitäten und Kontrollblöcken grundlegende Konzepte und Problemstellungen dynamischer WF-Änderungen. Die weiteren Basisänderungsoperationen des ADEPT$_{flex}$-Ansatzes werden in Abschnitt 7.5 zusammengefaßt. Ihre kombinierte Anwendung, die zur Verwirklichung komplexer Änderungen notwendig wird, ist Gegenstand von Abschnitt 7.6.

7.1 Entwurfsziele

Bevor wir die Änderungsoperationen im einzelnen vorstellen, fassen wir wichtige Entwurfsziele des ADEPT$_{flex}$-Ansatzes zusammen. Strukturelle Änderungen an laufenden WF-Instanzen dürfen in der Folge niemals zu Inkonsistenzen (z.B. *Lost Updates*) oder Programmfehlern (z.B. Aufruf von Anwendungskomponenten bei fehlenden Parameterdaten) führen, die möglicherweise sogar die Stabilität und Robustheit des Anwendungssystems beeinträchtigen. Unser Hauptaugenmerk beim Entwurf der ADEPT$_{flex}$-Änderungsoperationen gilt deshalb der Aufrechterhaltung der *Korrektheits*- und *Konsistenzeigenschaften* von WF-Vorlagen und -Instanzen. Ebenso, wie wir es für die statische Modellbildung und das daraus ableitbare dynamische Verhalten von WFs gefordert haben, muß auch für die Anwendung von Änderungsoperationen auf laufende WF-Instanzen gewährleistet werden, daß der resultierende WF stets ein korrekt strukturiertes Schema (Kontroll- und Datenfluß) und einen zulässigen Status aufweist. Ersteres soll u.a. den Erhalt der regelmäßigen Blockstrukturierung garantieren, was die Verwendung blockbasierter Änderungsoperationen nahelegt. Des weiteren müssen die anderen, für KF-Graphen geforderten Struktureigenschaften (z.B. Zyklenfreiheit bzgl. Kontroll- und Sync-Kanten) auch für den aus der Änderung hervorgehenden KF-Graph zutreffen. Idealerweise sollten auftretende Verletzungen (siehe Beispiel 7-1) nicht erst nachträglich durch aufwendige Analysen feststellbar sein, sondern durch Festlegung geeigneter Vorbedingungen für die Anwendung von Änderungsoperationen von vornherein ausgeschlossen werden können. Auch in Bezug auf Datenflüsse fordern wir, daß die gewünschten Eigenschaften DF-1, DF-2 und ggf. DF-3 (vgl. Kapitel 5) nach der Abänderung des WF weiterhin gelten. Wie das Beispiel 7-2 zeigt, ist dies bei Anwendung einzelner Änderungsoperationen nicht immer der Fall, so daß entsprechende Änderungswünsche entweder zurückzuweisen sind oder aber die Korrektheit des Datenflusses durch begleitende Maßnahmen (z.B. durch dynamisches Hinzufügen von Nachforderungsdiensten) wiederhergestellt werden muß.

Die Überführung einer WF-Instanz von einem konsistenten in einen neuen konsistenten Zustand stellt ebenfalls eine essentielle Anforderung für die korrekte Durchführung dynamischer WF-Änderungen dar. Es ist offensichtlich, daß bestimmte Modifikationen eines WF-Ausführungsgraphen nicht in jedem Bearbeitungsstatus erlaubt werden dürfen. Wie das Beispiel 7-3 zeigt, könnten sich ansonsten WF-Zustände ergeben, die bei normaler Ausführung des WF nicht erzeugbar wären. Dies wiederum hätte eine unklare Änderungssemantik und ggf. sogar undefinierte Systemzustände zur Folge. Tritt ein solcher Fall auf, muß die beabsichtigte Änderung entweder verworfen werden, oder es sind begleitende bzw. alternative Maßnahmen durchzuführen. Die einfachste Möglichkeit besteht darin, die WF-Bearbeitung in einen früheren Zustand zurückzusetzen (vgl. Kapitel 6), in dem die Änderung dann angewendet werden kann. Dies ist in der Praxis nicht immer sinnvoll. Alternativ kann versucht werden, die beabsichtige Änderung (z.B. das Einfügen einer Aktivität) in anderen Bereichen des Aus-

Beispiel 7-1: Verletzung von KF-Strukturierungsregeln bei Abänderung von WF-Schemata

Angenommen, in einem KF-Graphen ist eine Aktivität Y in der Nachfolgermenge $succ^*(X)$ einer Aktivität X enthalten. Möchte ein Akteur in diesem KF-Graphen eine neue Aktivität als Nachfolger von Y und als Vorgänger von X einfügen, so würde dies unmittelbar zu einem Zyklus und damit zu einer Verletzung von Strukturierungsregel KF-4* führen. Dem Änderungswunsch kann deshalb nicht entsprochen werden.

Beispiel 7-2: Verletzung von DF-Strukturierungsregeln bei Abänderung von WF-Schemata

Das Löschen einer WF-Aktivität und ihrer assoziierten DF-Kanten darf nicht dazu führen, daß es in der Folge zur Ausführung von WF-Aktivitäten mit unversorgten Eingabeparametern kommt (Regel DF-1). Ebensowenig dürfen durch das Einfügen von Aktivitäten und ihrer DF-Schreibkanten parallele Schreibzugriffe auf Datenelementen resultieren, d.h. Strukturierungsregel DF-2 darf nicht verletzt werden.

Beispiel 7-3: Erzeugung von inkonsistenten Zuständen bei Abänderung von WF-Instanzen

Das Einfügen einer neuen Aktivität in einen als bearbeitet markierten Bereich des KF-Ausführungsgraphen (hierzu zählen auch Teilzweige bedingter Verzweigungen, die als nicht ausgeführt markiert worden sind) ist nicht zulässig, da ansonsten eine inkonsistente Zustandsmarkierung des Graphen resultiert. Schlimmstenfalls wäre es dann sogar möglich, daß eine Endmarkierung erreicht wird, d.h. der WF terminiert, noch bevor die neu eingefügte Aktivität beendet wird. Ebenso wenig dürfen bereits beendete Aktivitätenknoten oder geschriebene Datenelemente aus dem WF-Schema der Instanz gelöscht werden. Auch hier wären Inskonsistenzen (z.B. *Dirty Reads* auf dem WF-Datenkontext, unversorgte Lesezugriffe) unvermeidbar.

Tabelle 7-1: *Beispiele für nicht korrekte Änderungen von Workflows*

führungsgraphen einzubringen. In Verbindung mit Schleifen kann es auch sinnvoll sein, Änderungen, die in der aktuellen Schleifeniteration infolge des zu weit fortgeschritten WF-Status nicht mehr anwendbar sind, vorzumerken und sie ggf. vor Beginn der nächsten Iteration „verspätet“ einzubringen.

Beispiel 7-3 verdeutlicht nochmals, daß Änderungsoperationen, die auf WF-Vorlagen korrekt anwendbar sind, nicht notwendigerweise für davon abgeleitete WF-Instanzen zulässig sind. Aus diesem Grund müssen für jede Änderungsoperation klare Vorbedingungen festgelegt werden. Sie betreffen zum einen das Schema der abzuändernden WF-Instanz, zum anderen ihren aktuellen Bearbeitungsstatus (d.h. ihre Graphmarkierungen). Darüber hinaus soll Änderungsoperationen eine eindeutige Semantik zugrundegelegt werden, so daß Anwender in der Lage sind nachzuvollziehen, warum das System bestimmte Änderungen erlaubt, während es andere verbietet.

Weitere wichtige Forderungen betreffen die Vollständigkeit und – soweit sinnvoll – die Minimalität der angebotenen Menge an Basisänderungsoperationen. *Vollständigkeit* bedeutet, daß jede denkbare und syntaktisch korrekte Restrukturierung eines WF-Schemas auf Basis von Operationen dieser Menge verwirklichbar sein muß. Dies ist notwendig, um auch Vorlagenänderungen sinnvoll unterstützen zu können. Wie bereits erwähnt, werden auf Instanzebene die zulässigen Änderungen weiter durch den aktuellen Bearbeitungsstatus eingeschränkt, so daß hier dem Begriff der Vollständigkeit eine etwas andere Bedeutung zukommt. Die Forderung nach *Minimalität* soll sicherstellen, daß die Effekte einzelner Basisoperationen nicht durch die kombinierte Anwendung einer oder mehrerer anderer Basisoperationen erzielbar sind. Allerdings sollte im Einzelfall abgewägt werden, ob ableitbare Operationen, die häufig benötigt werden, durch die kombinierte Anwendung von Basisoperationen oder – zur Steigerung der Performanz – ggf. gesondert implementiert werden sollen (ähnlich wie im Fall der Join-Operation bei relationalen DBMS). Auch in $ADEPT_{flex}$ gibt es einige Basisoperationen, die bei strikter Auslegung des Minimalitätsbegriffs nicht notwendig wären. Abgesehen davon sollten auf der Grundlage der angebotenen Basisänderungsoperationen häufig benötigte, komplexe Änderungsoperationen definiert werden können. Beispiele hierfür sind Operationen für das Einfügen von Aktivitäten zwischen zwei Knotenmengen oder für die Durchführung von Ad-hoc-Vorwärtssprüngen.

Weitere Entwurfsziele, die eher praktischer Natur sind, betreffen die effiziente Anwendbarkeit von Änderungsoperationen (→ Eingrenzung der notwendigen Überprüfungen, effiziente Analysen), die Behandlung konkurrierender Änderungen, Sicherheitsaspekte (→ Wer darf welche Änderungsoperationen auf welche WF-Instanzen in welchen Zuständen anwenden? Welche Informationen müssen dokumentiert werden?) sowie Benutzeraspekte (→ Wie sollen entsprechende Änderungsmöglichkeiten dem Anwendungsentwickler bzw. dem Endanwender angeboten werden?). Die meisten dieser Aspekte sind im Kontext dieser Arbeit von untergeordneter Bedeutung, wurden aber begleitend im Rahmen verschiedener Diplomarbeiten eingehend untersucht [Kubi98, Saue96, Weil97].

7.2 Grundlagen

7.2.1 Graph- und Zustandstransformationen

Basisänderungsoperationen müssen sowohl auf WF-Vorlagen als auch auf WF-Instanzen anwendbar sein. Wir wiederholen nochmals kurz, was die wesentlichen Bestandteile einer WF-Vorlage sind und welche Informationen auf Instanzebene hinzukommen. Jeder WF-Vorlage w ist eine Aufrufschnittstelle (siehe Kapitel 3) und ein WF-Graph zugeordnet. Dieser WF-Graph definiert die Aktivitäten und die zwischen ihnen bestehenden Kontroll- und Datenflüsse. Er konstituiert sich aus einem KF-Graphen $CFS_w = (N, E, D, ...)$ und einem Datenflußmodell DFS_w. Letzteres umfaßt eine Menge von Datenkanten, die Ein- und Ausgabeparameter von Aktivitäten aus N (bzw. ihrer vor-/nachgeschalteten Aktionen) mit Datenelementen aus D verknüpfen. Soll zu einer gegebenen WF-Vorlage w eine neue WF-Instanz erzeugt werden, wird zuerst eine (logische) Kopie $WS = (CFS, DFS)$ ihres WF-Graphen (CFS_w, DFS_w) angelegt. Dieser „WF-Instanzgraph“, der auch als *Ausführungsgraph* bezeichnet wird, wird dann um zusätzliche Informationen „angereichert“, die für die WF-Steuerung benötigt werden. Konkret werden für die WF-Instanz die in den Kapiteln 4, 5 und 6 beschriebenen Kontrolldaten und WF-relevanten Daten verwaltet. Sie umfassen die aktuellen Zustandsmarkierungen (NS, ES, It) der Knoten und Kanten des Ausführungsgraphen CFS (kurz: $CFS_{instance} := (CFS, NS, ES, It)$) und den Datenkontext $CtxtValues$, der für jedes Datenelement $d \in D$ die Historie seiner bisher geschriebenen

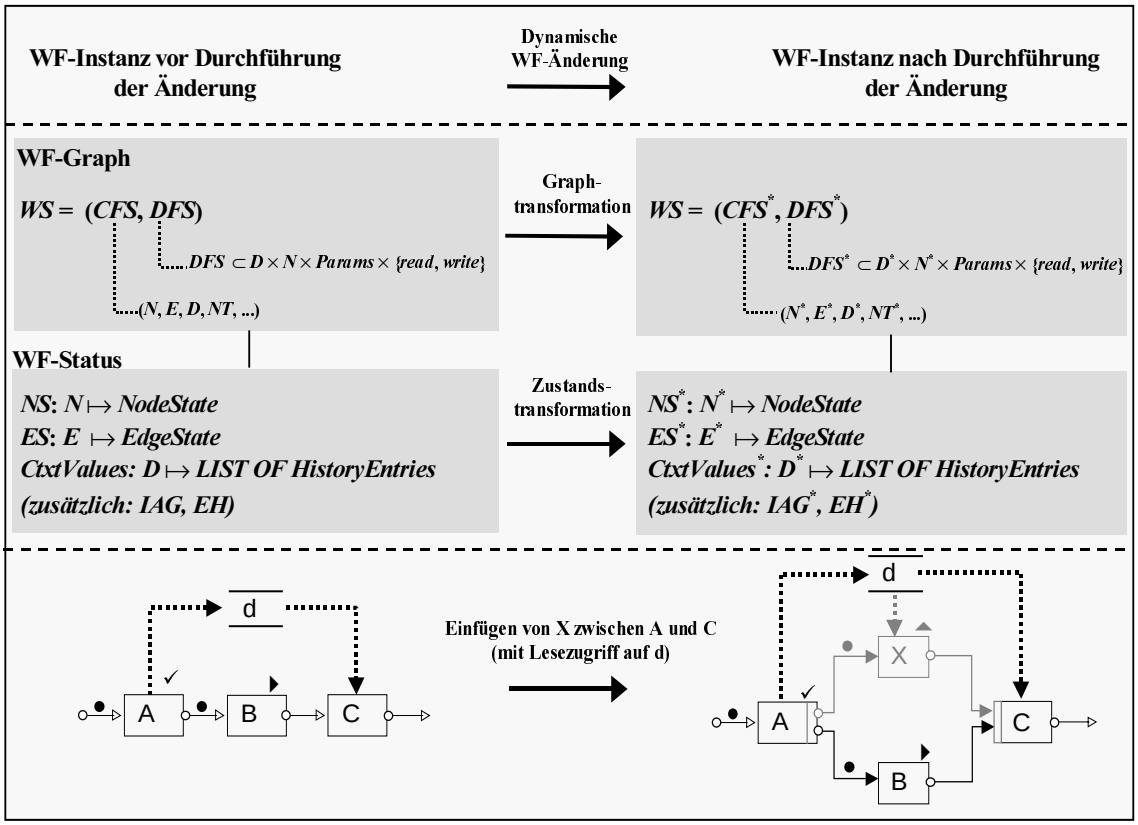


Abbildung 7-1: *Graph- und Zustandtransformationen bei dynamischen WF-Änderungen*

Werte verwaltet. Hinzu kommen der Instanzabhängigkeitsgraph *IAG*, der hierarchische Aufrufbeziehungen verwaltet, und die Ablaufhistorie *EH*.

Eine sich in Ausführung befindliche WF-Instanz kann strukturell modifiziert werden, ohne daß dies Auswirkungen auf andere, aus derselben Vorlage hervorgegangene WF-Instanzen hat. Im allgemeinen erfordert eine solche Instanzmodifikation zweierlei Maßnahmen (vgl. Abbildung 7-1): Zum einen muß der bisherige Ausführungsgraph $WS = (CFS, DFS)$ der WF-Instanz auf den gewünschten Zielgraphen $WS^* = (CFS^*, DFS^*)$ transformiert werden (*Graphtransformation*), zum anderen müssen die Zustandsmarkierungen dieses Graphen, der Datenkontext, der Instanzabhängigkeitsgraph und die Ablaufhistorie ggf. angepaßt bzw. erweitert werden (*Zustandstransformation*). Anpassungen, die ausschließlich eine Zustandsmodifikation, aber keine Graphtransformation bewirken, werden im folgenden auch als *Statusänderungen* bezeichnet. Hierzu gehört in seiner einfachen Form das in Kapitel 6 vorgestellte Verfahren zum kontrollierten Zurücksetzen von Workflows.

7.2.2 Änderungsprimitive

Die strukturelle Änderung von WF-Graphen und WF-Ausführungsgraphen erfolgt durch die Hinzunahme und das Entfernen von Knoten, Kanten (inkl. Datenkanten) und Datenelementen sowie durch die Abänderung ihrer Attribute. In diesem Zusammenhang stellt sich die Frage, wie die Semantik der Graph- und Zustandstransformationen, die sich bei Anwendung einzelner Änderungsoperationen ergeben, präzise beschrieben werden kann. In der gängigen Literatur zu Graphgrammatiken und Graphersetzungssystemen etwa werden auf Ebene der zugrundeliegenden Kalküle unterschiedliche Ansätze gewählt [EhTa96, Gött88, Nagl79, Roze97]: *Mengentheoretische Ansätze* beschreiben Graphen durch Knoten- und Kantenmengen. Die Wirkung von Graphtransformationen wird durch die üblichen Mengenoperationen wie Vereinigung, Differenz, usw. festgelegt. Bei *logikbasierten Ansätzen* werden sowohl Graphen als auch Graphregeln auf eine Menge von Formeln abgebildet. Die Anwendung einer Graphersetzungsregel wird dann als Manipulation dieser Formelmenge aufgefaßt. Beide Ansätze gestatten es, komplexe Regeln zum Aufbau von Graphen zu formulieren und zur Überprüfung der Korrektheit von Graphregeln zu verwenden. Allerdings erschließen sich logikbasierte Ansätze dem intui-

tiven Verständnis schwerer. Dasselbe trifft auch auf *kategorientheoretische Ansätze* zu, bei denen zudem größere Beschränkungen im Hinblick auf beschreibbare Graphregeln bestehen.

Das ADEPT$_{flex}$ zugrundeliegende Kalkül verfolgt einen mengenbasierten Ansatz. Wie in Teil II dieser Arbeit beschrieben, werden KF- und DF-Modelle durch Mengen attributierter Knoten und Kanten beschrieben. Darauf basierend haben wir Regeln zu ihrer korrekten Strukturierung formuliert und zur Ableitung dynamischer WF-Eigenschaften verwendet. Von daher liegt es nahe, auch die Wirkung der ADEPT$_{flex}$-Operationen (bzw. der mit ihnen verbundenen Graphersetzungen) mit Hilfe mengentheoretischer Operationen zu beschreiben. Eine solche Vorgehensweise ist zum einen intuitiv verständlich und erlaubt es zum anderen, selbst komplexe Graphersetzungen formal zu fassen. Zu diesem Zweck geben wir nun eine Menge von *Änderungsprimitven* an, die zur Manipulation der Bestandteile von WF-Graphen und WF-Ausführungsgraphen (Knotenmenge, Kantenmengen, Datenelemente, usw.) verwendet werden können. Die Semantik dieser Primitive wird durch einfache Mengenoperationen festgelegt. Hinzu kommen Primitive zur Änderung von Knoten- und Kantenattributen und zur Anpassung von Zustandsmarkierungen (bei Ausführungsgraphen). Die Änderungsprimitive werden in der Folge dazu verwendet, die ADEPT$_{flex}$-Basisänderungsoperationen und ihre Auswirkungen auf die genannten Mengen formal festzulegen. Wichtig ist an dieser Stelle der Hinweis, daß die Anwendung einzelner Primitive für sich allein im allgemeinen keinen Sinn ergibt und auch die geforderten Konsistenzeigenschaften des WF nicht aufrechterhält. Die nachfolgenden Tabellen fassen die verschiedenen Änderungsprimitive, notwendige Vorbedingungen für ihre Anwendung und ihre Semantik zusammen (E steht dabei für Erklärung, V für Vorbedingung und S für Semantik):

AddNodes(CFS, nodeLabels)

E: Fügt die Knotenbezeichner aus *nodeLabels* der Knotenmenge *N* des KF-Graphen *CFS* hinzu und initialisiert Knotenattribute mit Standardvorgabewerten (siehe unten)

V: nodeLabels $\cap$ N = $\emptyset$ ($\rightarrow$ Eindeutigkeit von Knotenbezeichnern)

S: N^*:= N $\cup$ nodeLabels

$\forall$ n $\in$ nodeLabels (Standardvorgaben):

$NT^*(n)$ = `NULL`; $V_{out}^*(n)$ = `ONE_Of_ONE`, $V_{in}^*(n)$ = `ONE_Of_ONE`

$DP^*(n)$ = `UNDEFINED`, $NP^*(n)$:= `REGULAR` (Priorität)

$Template^*(n)$ = `UNDEFINED` ($\rightarrow$ $InParams^*(n) = OutParams^*(n) = \emptyset$)

$PreS^*(n) = \{S_{pre}^n\}$ mit $OutParams(S_{pre}^n)$: = $\emptyset$ (siehe hierzu Tabelle 7-4)

$PostS^*(n) = \{S_{post}^n\}$ mit $OutParams(S_{post}^n)$: = $\emptyset$

Bei Einfügen auf *Instanzebene* zusätzlich:

$NS^*(n)$ = `NOT_ACTIVATED`, $It^*(n) = 0$

(Anm.: Alle anderen Knoten und ihre Attribute werden unverändert übernommen)

DeleteNodes(CFS, nodeLabels)

E: Löscht Knoten der Menge *nodeLabels* aus der Knotenmenge *N* des KF-Graphen *CFS*.

V: nodeLabels $\subseteq$ N

S: N^*:= N – nodeLabels

SetNodeAttr(CFS, nodeLabel, attr, val)

E: Setzt für den Knoten mit Bezeichner *nodeLabel* das Knotenattribut *attr* auf den Wert *val*.
Beispiel: SetNodeAttr(CFS, A, "NT", `ACTIVITY`)

V: nodeLabel $\in$ N, attr $\in$ {NT, V_{out}, V_{in}, NP}, val $\in$ dom^{attr}

S: $attr^*$(nodeLabel):= val

SetNodeTemplate(CFS, nodeLabel, templ)

E: Ordnet dem Knoten mit Bezeichner *nodeLabel* die neue Vorlage *templ* zu.

V: nodeLabel $\in$ N, templ $\in$ Templates (= Menge aller im Vorlagenrepository definierten Vorlagen)

S: $Template^*$(nodeLabel):= templ

Eigenschaften der Vorlage werden vom Knoten „übernommen“, d.h. es gilt anschließend z.B. $InParams^*$(nodeLabel): = $InParams^{templ}$ und $OutParams^*$(nodeLabel): = $OutParams^{templ}$

SetNodeDP (CFS, nodeLabel, dp) E: Ordnet dem OR-Split-Knoten *nodeLabel* das Datenelement *dp* als Entscheidungsparameter zu. V: nodeLabel $\in$ N, dp $\in$ D $\cup$ {`UNDEFINED`}, V_{out}(nodeLabel) = `ONE_Of_ALL` S: DP^*(nodeLabel):= dp
AddEdges(CFS, edgeSet) E: Fügt die angegebene Kantenmenge der Menge *E* des KF-Graphen *CFS* hinzu und initialisiert Kantenattribute mit Standardvorgabewerten (siehe unten) V: edgeSet $\subset$ N $\times$ N $\times$ EdgeType, edgeSet $\cap$ E = $\emptyset$ S: E^*:= E $\cup$ edgeSet $\forall$ e $\in$ edgeSet (Standardvorgabe): EC^*(e):= `UNDEFINED` (zur Erinnerung: *EC* definiert - abhängig vom Kantentyp - einen Auswahlwert, einen Fehlercode oder eine Priorität) Bei Einfügen auf Instanzebene (d.h. in Ausführungsgraphen) zusätzlich: ES^*(e) = `NOT_SIGNALED`
DeleteEdges(CFS, edgeSet) E: Löscht alle durch *edgeSet* festgelegten Kanten aus der Kantenmenge *E* des KF-Graphen *CFS*. V: edgeSet $\subseteq$ E S: E^*:= E – edgeSet
SetEdgeAttr(CFS, edge, attr, val) E: Setzt für die angegebene Kante *edge* das Kantenattribut *attr* auf den Wert *val*. (Das Attribut *EC* definiert – je nach Kantentyp – den Auswahlwert, den Fehlercode oder die Priorität der Kante) <u>Beispiel</u>: SetEdgeAttr(CFS, (n_1, n_2, `CONTROL_E`), "EC" , `True`) V: edge $\in$ E, attr $\in$ {EC}, val $\in$ dom^{attr} S: $attr^*$(edge):= val

Tabelle 7-2: *Primitive zur Manipulation von Knoten- und Kantenmengen sowie der Attribute ihrer Elemente*

In gleicher Weise sind die Primitive zur Manipulation von Datenelementen und -kanten definiert:

AddDataElements(CFS, DFS, dataLabels, dom, defaultVal) E: Fügt die angegebene Menge an Datenelementen bzw. -bezeichnern der Menge *D* hinzu. Den Elementen wird der Domain *dom* und der Defaultwert *defaultVal* (kann auch der Nullwert sein) zugewiesen. V: dataLabels $\cap$ D = $\emptyset$, dom $\in$ Dom, defaultVal $\in$ dom $\cup$ {`null`} S: D^*:= D $\cup$ dataLabels $\forall$ d $\in$ dataLabels: dom^*(d):= dom, $default^*$(d):= defaultVal
DeleteDataElements(CFS, DFS, elementSet) E: Löscht die angegebenen Datenelemente aus der Menge *D*. V: dataLabels $\subseteq$ D S: D^*:= D – elementSet
AddDataEdges(CFS, DFS, dataEdges) E: Fügt die angegebene Menge von Datenkanten dem Datenflußschema *DFS* hinzu. V: dataEdges $\cap$ DFS = $\emptyset$, dataEdges $\subseteq$ D $\times$ (N $\cup$ S) $\times$ Params $\times$ {read, write); $\forall$ dfc $\in$ dataEdges: [dfc=(d, x, par, read), $\Rightarrow$ par $\in$ $InParams^x$ $\wedge$ $pDom^{par}$ $\supseteq$ dom^d] und [dfc=(d, x, par, write) $\Rightarrow$ par $\in$ $OutParams^x$ $\wedge$ $pDom^{par}$ $\subseteq$ dom^d] S: DFS^*:= DFS $\cup$ dataEdges
DeleteDataEdges(CFS, DFS, dataEdges) E: Entfernt die angegebene Menge an Datenelementen aus dem DF-Schema *DFS*. V: dataEdges $\subseteq$ DFS S: DFS^*:= DFS – dataEdges

Tabelle 7-3: *Primitive zur Änderung der Menge definierter Datenelemente und -kanten*

Für die Verwendung vor- und nachgeschalteter Hilfsdienste (vgl. Kapitel 5) treffen wir aus Gründen der besseren Lesbarkeit die folgende Vereinfachung: Jeder Knoten besitzt genau einen vor- und einen nachgeschalteten Dienst, der zur (interaktiven) Festlegung fehlender Ein- bzw. Ausgabeparameter herangezogen werden kann. Die Menge der interaktiv zu versorgenden, nachforderbaren Ein- bzw. Ausgabeparameter (initial: leer) kann für Knoten dynamisch erweitert bzw. reduziert werden, je nach den konkreten Erfordernissen. Hierfür stehen die Primitive aus Tabelle 7-4 zur Verfügung.

AddParamProvisionService (CFS, DFS, node, pars)

E: Für die durch *pars* festgelegten Ein- und Ausgabeparameter des Knotens *node* wird ein Nachforderungsdienst dem Knoten vor- bzw. nachgeschaltet (d.h. es werden die Parameterschnittstellen von S_{pre}^{node} bzw. S_{post}^{node} adaptiert und entsprechende Schreibkanten dem Datenflußschema *DFS* hinzugefügt).

V: node $\in$ N, pars $\subseteq$ $InParams^{node}$ $\vee$ pars $\subseteq$ $OutParams^{node}$, $\forall$ p $\in$ pars: pRm^p = USER_REQUEST

S: Fall 1: Vorschalten eines Nachforderungsdienstes (d.h. es gilt: pars $\subseteq$ $InParams^{node}$)

1. Füge eine zu *pars* korrespondierende Parametermenge *pars** (gleiche Bezeichner und Wertebereiche, inverse Richtung) zu den bereits vorhandenen Ausgabeparametern des Nachversorgungsdienstes S_{pre}^{node} hinzu: $OutParams^*(S_{pre}^{node}) := OutParams(S_{pre}^{node}) \cup pars^*$
2. Verknüpfe die hinzugenommenen Ausgabeparameter von S_{pre}^{node} über Schreibkanten mit den Datenelementen, die von korrespondierenden Eingabeparametern aus *pars* gelesen werden:

   dataEdges:= { (d, S_{pre}^{node}, p^*, write) | $p^* \in pars^* \wedge \exists$ dfc $\in$ DFS mit dfc = (d, node, p, read) $\wedge$ $p^* \equiv$ p}
   AddDataEdges(CFS, DFS, dataEdges)

Fall 2: Nachschalten eines Nachforderungsdienstes (d.h. es gilt: pars $\subseteq$ $OutParams^{node}$) (dto.)

DeleteParamProvisionService(CFS, DFS, node, pars)

E: Entfernt Nachforderungsdienste (inkl. ihrer DF-Schreibkanten) der durch die Menge *pars* festgelegten Ein-/Ausgabeparameter des Knotens *node*

V: node $\in$ N, pars $\subseteq$ $Params^{node}$

S: Im wesentlichen „inverse“ Durchführung der bei *AddParamProvisionService* angewandten Schritte.

Tabelle 7-4: *(Logisches) Hinzufügen und Entfernen vor- bzw. nachgeschalteter Nachforderungsdienste*

Für die Änderung der Markierung von Knoten und Kanten in Ausführungsgraphen schließlich stehen die folgenden beiden Primitive zur Verfügung:

SetNodeState(CFS$_{instance}$, nodeLabel, newState)

E: Setzt für den Knoten mit Bezeichner *nodeLabel* den Knotenstatus auf den Wert *newState*.

V: nodeLabel $\in$ N, newState $\in$ NodeStates

S: $$NS^*(n) := \begin{cases} newState & \text{für } n = nodeLabel \\ NS(n) & \text{sonst} \end{cases}$$

SetEdgeState(CFS$_{instance}$, edge, newState)

E: Setzt für die Kante *edge* den Kantenstatus auf den Wert *newState*.

V: edge $\in$ E, newState $\in$ EdgeStates

S: $$ES^*(e) := \begin{cases} newState & \text{für } e = edge \\ ES(n) & \text{sonst} \end{cases}$$

Tabelle 7-5: *Primitive zur Änderung von Knoten- und Kantenmarkierungen in Ausführungsgraphen*

Darüber hinaus gibt es Primitive, mit denen sich Attribute und Markierungen von Knoten und Kanten abfragen lassen, die aber das Schema und den Zustand der WF-Instanz nicht verändern (z.B. *GetNodeAttr*(*CFS bzw. CFS$_{instance}$, node, attr*) $\rightarrow$ *attrVal*). Aufgrund ihrer intuitiven Semantik verzichten wir an dieser Stelle auf eine explizite Darstellung.

7.2.3 Nullaktivitäten und Graphvereinfachungsregeln

Wie in den vorangegangenen Kapiteln erwähnt, können zur Bildung von KF-Graphen auch Aktivitätenknoten ohne assoziierte Aktion bzw. Vorlage – sog. *Nullaktivitäten* – verwendet werden. Hierfür sieht das ADEPT-Basismodell den speziellen Knotentyp NULL vor. Wird auf Instanzebene ein Knoten dieses Typs aktiviert – dies erfolgt nach denselben Regeln wie bei normalen Aktivitätenknoten – und besitzt er keine vor- bzw. nachgeschalteten Hilfsdienste, führt dies unmittelbar zu seiner Beendigung und damit zur Neubewertung des Status seiner direkten Nachfolgerknoten (vgl. Kapitel 4). Von der Ausführung einer Nullaktivität gehen dem entsprechend keine Effekte auf Daten des WF-Datenkontexts und auf externe Anwendungsdaten aus.

Die Verwendung von Nullaktivitäten kann aus verschiedenen Gründen sinnvoll sein. Wie wir in Kapitel 6 im Zusammenhang mit der Vormodellierung von Vorwärts- und Rückwärtssprüngen gezeigt haben, läßt sich dadurch ein komplexes Ausführungsverhalten im WF-Modell abbilden, ohne daß die regelmäßige Blockstrukturierung der KF-Graphen aufgegeben werden muß. Eine weitere wichtige Anwendung von Nullaktivitäten werden wir in der Folge aufzeigen: Die *kontextunabhängige* Beschreibung der zur Verwirklichung einzelner Basisänderungsoperationen notwendigen Graphtransformationsregeln. Beispielsweise unterstützt ADEPT$_{flex}$ das Einfügen einer neuen Aktivität parallel zu einem beliebigen Kontrollblock des KF-Graphen. Dies wird – vereinfacht ausgedrückt – dadurch realisiert, indem ein AND-Split-Knoten n_1 als direkter Vorgänger und ein AND-Join-Knoten n_2 als direkter Nachfolger des Blocks eingefügt werden, wobei beide Knoten Nullaktivitäten repräsentieren. Der neu einzufügende Schritt wird dann als neuer Zweig der durch (n_1, n_2) gebildeten Parallelverzweigung „eingehängt“ (siehe Abschnitt 7.3.1.2 b). Diese Transformationsschritte bleiben auf beliebige Kontrollblöcke anwendbar, unabhängig davon in welchem genauen Kontext (vgl. Definition 3-10) sie auftreten. Mit anderen Worten: Die Verwendung von Nullaktivitäten erspart uns bei der Formulierung der entsprechenden Graphtransformationsregeln die aufwendige Behandlung von Sonderfällen, die ansonsten notwendig werden würde. Diese kontextunabhängigen Transformationen schließen nicht aus, daß anschließend im KF-Graphen Nullaktivitäten auftreten, die zur Abbildung des gewünschten Ausführungsverhaltens eigentlich nicht benötigt werden (die dieses Verhalten aber auch nicht beeinflussen). In solchen Fällen kann – falls gewünscht – eine Vereinfachung des Graphen vorgenommen werden, die auf einer wohldefinierten Menge von Transformations- bzw. Vereinfachungsregeln basiert. Beispiele für solche Regeln zeigt Abbildung 7-2. Aufgrund ihrer intuitiven Semantik verzichten wir an dieser Stelle auf eine formale Darstellung dieser und weiterer Vereinfachungsregeln. Die in Abbildung 7-2 dargestellten Transformationen können auch in „umgekehrter“ Richtung angewendet werden.

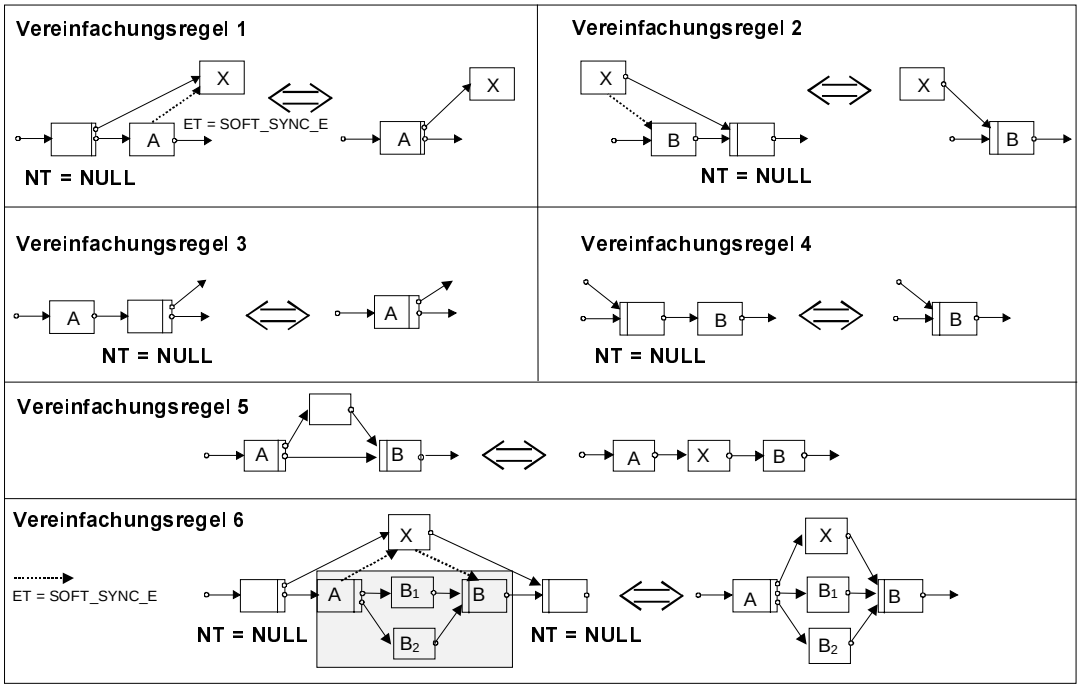


Abbildung 7-2: *Nullaktivitäten und Graphvereinfachungsregeln*

Auf Grundlage der in den Abschnitten 7.2.1 bis 7.2.3 angestellten Überlegungen können wir nun die Basisänderungsoperationen von ADEPT$_{flex}$ und ihre Semantik präzise angeben. Konkret legt ADEPT$_{flex}$ für jede Änderungsoperation formale Vorbedingungen, Graphtransformationsregeln, Mechanismen für das Aufdecken möglicher Probleme sowie unterschiedliche Strategien für ihre Behandlung fest. In den folgenden beiden Abschnitten beschreiben wir grundlegende Konzepte und Problemstellungen am Beispiel des (dynamischen) Einfügens und Löschens von Aktivitäten bzw. Kontrollblöcken. Der darauffolgende Abschnitt faßt die anderen Basisänderungsoperationen zusammen.

7.3 Die Einfüge-Operation

Das Einfügen neuer Aktivitäten in den Ausführungsgraphen einer WF-Instanz kann aus unterschiedlichen Gründen notwendig werden, etwa wenn zur Modellierungszeit nicht eingeplante Arbeitsschritte spontan ausgeführt oder fehlende bzw. fehlerhafte Daten nachträglich festgelegt bzw. korrigiert werden sollen. Auch die Unterstützung evolutionärer Prozesse, bei denen sich die genaue Struktur des WF erst im Verlauf seiner Ausführung ergibt, erfordert die Bereitstellung dieser Funktion. Neben benutzerinitiierten Einfügeoperationen müssen Aktivitäten auch automatisch, d.h. bei Eintreten bestimmter Ereignisse, in den WF eingefügt werden können [MüRa99]. Wie in Kapitel 2 gezeigt unterstützen heutige WfMS die dynamische Hinzunahme von Aktivitäten – wenn überhaupt – nur in rudimentärer Form. So können bei einigen dokumentenorientierten WfMS Aktivitäten direkt vor oder nach der Ausführung einer Aktivität eingefügt werden. Die Anknüpfung ihrer Eingabeparameter an den WF-Datenkontext ist dagegen nicht oder nur eingeschränkt möglich. Der Erhalt der Korrektheit und Konsistenz des WF fällt somit überwiegend in den Verantwortungsbereich des Anwenders.

In gewisser Weise ist das dynamische Einfügen von Aktivitäten (bzw. ihrer assoziierten Anwendungskomponenten) in den WF-Graphen einer WF-Instanz vergleichbar mit dem Hinzufügen eines Unterprogrammaufrufs zu einem laufenden Anwendungsprogramm, und das inmitten seiner Ausführung! In unserem Kontext bedeutet dynamisches Einfügen, daß eine Vorlage für den neu einzufügenden Arbeitsschritt aus dem Repository ausgewählt, die Ausführung der WF-Instanz (teilweise) unterbrochen, die gewünschten Modifikationen – sie bestehen im wesentlichen aus der Hinzunahme neuer Knoten und Kanten (inkl. Datenflußkanten) – auf dem Ausführungsgraphen der WF-Instanz durchgeführt und anschließend der Zustand des WF so angepaßt werden, daß mit der WF-Kontrolle auf Basis des modifizierten Ausführungsgraphen fortgefahren werden kann. Oberstes Ziel ist dabei der Erhalt der Korrektheit und Konsistenz des WF. Grundlegend hierfür sind die Regeln zur Strukturierung der KF-Graphen und DF-Modelle sowie die in Kapitel 4 geforderten Dynamikeigenschaften. Sie werden in der Folge zur Überprüfung der Korrektheit von Graph- und Zustandstransformationen herangezogen.

Aus der Unterstützung dynamischer Einfügeoperationen ergeben sich verschiedene Anforderungen:

- Es müssen vormodellierte Schablonen (*Templates*) für elementare Arbeitsschritte oder Sub-WFs sowie vorgefertigte Standardbausteine (z.B. für die Bildung von Schleifen- oder Verzweigungsstrukturen) in den Ausführungsgraphen einer WF-Instanz eingefügt werden können.
- Jede bei der Modellbildung bzw. -abänderung zulässige Einfüge-Operation muß auch auf Instanzebene anwendbar sein, vorausgesetzt der aktuelle Status des WF erlaubt dies. D.h. auch auf Instanzebene müssen neue Aktivitäten direkt vor oder nach einem Schritt, parallel zu einzelnen Aktivitätenknoten oder Kontrollblöcken oder auch als neuer Teilzweig einer beliebigen Verzweigung eingefügt werden können. Auch komplexere Operationen, wie das Einfügen eines neuen Schrittes zwischen zwei Mengen von Aktivitätenknoten, müssen möglich sein.
- Aktivitäten müssen in Bereiche des Ausführungsgraphen eingefügt werden können, die noch nicht betreten wurden bzw. deren Ausführung noch nicht feststeht. Prinzipiell sollte es auch möglich sein, neue Aktivitäten vor bereits aktivierte, aber noch nicht gestartete Schritte einzufügen. In einem solchen Fall muß die Aktivierung dieser Schritte wieder aufgehoben werden, um mit der Kontrolle in einem konsistenten Zustand fortfahren zu können.
- Die Ein- und Ausgabeparameter neu eingefügter Aktivitätenknoten müssen (dynamisch) an das Schema des WF-Datenkontexts angeknüpft werden. Ggf. muß dieses Schema erweitert werden, um wieder zu einem korrekten Datenflußschema zu gelangen.

7.3.1 Einfügen von Aktivitäten

In diesem Abschnitt behandeln wir das Einfügen elementarer und komplexer Aktivitäten in den Ablaufgraphen. Zunächst befassen wir uns mit den hierzu notwendigen Schematransformationen. Um die sich dabei ergebenden Fragestellungen angemessen diskutieren zu können, nehmen wir eine Trennung in Funktions-, Kontrollfluß- und Datenflußsicht vor. Für jeden dieser Aspekte zeigen wir, welche Vorbedingungen erfüllt bzw. welche Festlegungen getroffen werden müssen, um die gewünschte Einfüge-Operation verwirklichen zu können. Darüber hinaus gehen wir darauf ein, welche zusätzliche Bedingungen für das Einfügen von Aktivitäten in den Ausführungsgraphen laufender WF-Instanzen erfüllt sein müssen, um anschließend wieder einen konsistenten Zustand zu erhalten.

7.3.1.1 Funktionale Sicht

Zunächst ist festzulegen, *was* eingefügt werden soll. Zu diesem Zweck muß aus dem Repository eine Vorlage $templ \in Templates$ ausgewählt werden, die den neu hinzukommenden Schritt bzw. die mit ihm verknüpfte Aufgabe genau spezifiziert. Wie in Kapitel 3 beschrieben, stellt eine solche Vorlage an ihrer Schnittstelle verschiedene Informationen bereit, etwa zur Aufgabenkategorie, zu benötigten Eingabe- und produzierten Ausgabedaten, zu Fehlercodes, zum Ausführungsmodus (interaktiv, automatisch) und zu Bearbeiterrollen. Darüber hinaus legt sie fest, ob ein elementarer Arbeitsschritt oder ein (Sub-) WF vorliegt. In ersterem Fall ist die Vorlage mit einer Anwendungskomponente oder einer einfachen Systemfunktion verknüpft, in letzterem Fall ist ihr ein separater WF-Graph zugeordnet. Die Eigenschaften der ausgewählten Vorlage bilden die Vorgabe für den einzufügenden Arbeitsschritt. Sie werden benötigt, um die Aktivität korrekt in das Kontroll- und Datenflußschema des WF einzubetten . Beispielsweise müssen für den einzufügenden Schritt Parameterbeschreibungen vorliegen, um seine Ein- und Ausgabedaten an den WF-Datenkontext anknüpfen zu können.

Beim Einfügen komplexer Aktivitäten (= Knoten mit assoziierter WF-Vorlage ω_{add}) muß bei hierarchischen WFs zusätzlich gewährleistet werden, daß durch die Hinzunahme des Sub- WF, der seinerseits hierarchisch strukturiert sein kann, keine rekursiven Aufrufstrukturen entstehen (siehe Abschnitt 3.8). Dies erreichen wir durch Anwendung von Regel KF-8, die Zyklen in *Vorlagenabhängigkeitsgraphen*[1] (VAG) ausschließt. Der nachfolgende Satz sagt aus, unter welchen Voraussetzungen diese Regel auch nach dem Einfügen komplexer Aktivitäten zugesichert werden kann.

Satz 7-1 (Erhalt der Zyklenfreiheit in VAGs)

Bezeichne $\omega_{top} \in WT$ die Vorlage des Toplevel-WF und $G_{\omega_{top}} = [V_{\omega_{top}}, A_{\omega_{top}}]$ den zugehörigen VAG, von dem wir annehmen, daß er zyklenfrei ist. Ferner entspreche $\omega_{sub} \in V_{\omega_{top}}$ der Vorlage eines Sub-Workflows beliebiger Hierarchiestufe und $Pred_{\omega_{sub}} \subset V_{\omega_{top}}$ der Menge aller Knoten von $G_{\omega_{top}}$ die Vorgänger von ω_{sub} (inklusive) im VAG sind.

Die Vorlage des neu einzufügenden Aktivitätenknotens n_{add} sei durch $Template(n_{add}) = \omega_{add} \in WT$ und ihr zugehöriger VAG – bei dem einzufügenden Sub-WF kann es sich selbst wieder um einen hierarchischen WF handeln – durch $G_{\omega_{add}} = [V_{\omega_{add}}, A_{\omega_{add}}]$ gegeben.

<u>Dann gilt</u>:

$G_{\omega_{add}}$ ist zyklenfrei <u>und</u> $V_{\omega_{add}} \cap Pred_{\omega_{sub}} = \emptyset$ (d.h. ω_{add} basiert auf keiner Vorlage, die Vorgänger von ω_{sub} im Graphen $G_{\omega_{top}}$ ist) (*)

$\Leftrightarrow$

Das Einfügen des Knotens n_{add} in den KF-Graphen $CFS_{\omega_{sub}}$ der Vorlage ω_{sub} führt wieder zu einem zyklenfreien VAG $G_{\omega_{top}}{}^{*}$ (d.h. Regel KF-8 gilt weiterhin)

[1] Der VAG $G_\omega = [V_\omega, A_\omega]$ einer WF-Vorlage ω ist wie folgt definiert (vgl. Definition 3-13): Seine Knotenmenge V_ω umfaßt alle WF- und Aktivitätenvorlagen (inkl. ω), die für die Definition von Knoten des WF-Graphen von ω oder hierarchisch untergeordneter WF-Graphen verwendet werden; Pfeile $\omega_1 \rightarrow \omega_2$ aus A_ω beschreiben Abhängigkeiten der Form „Die Vorlage ω_1 verwendet zur Definition eines Aktivitätenknoten die Vorlage ω_2“.

Der Beweis kann Anhang B entnommen werden. Das Kriterium (*) gibt eine notwendige und hinreichende Bedingung für die Verwendung von WF-Vorlagen beim Einfügen komplexer Aktivitäten an.

Beispiel 7-4 (Verwendung von WF-Vorlagen beim Einfügen neuer Aktivitäten)

Im Beispiel aus Abbildung 7-3 a) sind der VAG der WF-Vorlagen ω_{top} und ω_{add} dargestellt. Es soll nun ein neuer Aktivitätenknoten mit ω_{add} als Vorlage in den KF-Graphen $CFS_{\omega_{sub}}$ eingefügt werden. Wir erhalten dann $Pred_{\omega_{sub}} = \{\omega_{top}, \omega_{sub}\}$ und damit $V_{\omega_{add}} \cap Pred_{\omega_{sub}} \neq \emptyset$. Hieraus folgt mit Satz 7-1 sofort, daß das Einfügen der komplexen Aktivität zu einem Zyklus im VAG führen würde (→ Abbildung 7-3 b). Die Einfügeoperation ist wegen Regel KF-8 deshalb nicht zulässig.

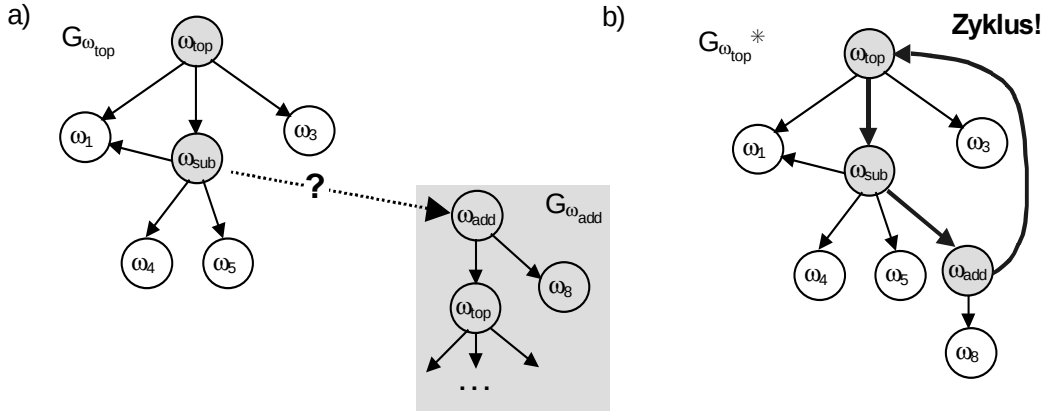


Abbildung 7-3: *Nichtzulässige Verwendung von WF-Vorlagen beim Einfügen komplexer Aktivitäten*

Prinzipiell ist es auch möglich, Aktivitätenknoten ohne Vorlage in den Ablaufgraphen einzufügen. Die entsprechenden Knoten repräsentieren dann Nullaktivitäten (vgl. Abschnitt 7.2.3), besitzen also keine assoziierte Aktion. Zu erwähnen bleibt noch, daß neu hinzukommende Schritte innerhalb des KF-Schemas, in das sie eingefügt werden, einen eindeutigen Bezeichner besitzen müssen.

7.3.1.2 Einfügen neuer Aktivitäten in KF-Graphen (Schemaebene)

Im folgenden bezeichne $CFS = (N, E, D, ...)$ das KF-Schema des WF vor Anwendung der Einfügeoperation. Um eine neue Aktivität einzufügen, muß der genaue Kontext angegeben werden, in den sie im KF-Graphen eingebettet werden soll. ADEPT$_{flex}$ unterscheidet drei grundlegende Fälle:

- **Serielles Einfügen** (*serialInsert*-Operation): Die Aktivität wird zwischen zwei direkt aufeinanderfolgenden Knoten $n_{src} \in N$ und $n_{dest} \in N$ im KF-Graphen eingefügt.
- **Paralleles Einfügen** (*parallelInsert*-Operation): Die Aktivität wird parallel zu einem Kontrollblock (n_{begin}, n_{end}) des KF-Graphen eingefügt; n_{begin}, $n_{end} \in N$ entsprechen dem Anfangs- bzw. dem Endknoten dieses Blocks.
- **Einfügen neuer Teilzweige** (*branchInsert*-Operation): Die Aktivität wird als neuer Teilzweig einer Verzweigung eingefügt. Einen Spezialfall stellt dabei das Einfügen eines neuen Zweiges in eine bedingte Verzweigung (→ *conditionalInsert*) dar.

Diese drei Insert-Operationen werden benötigt, um die Vollständigkeit von ADEPT$_{flex}$ bzgl. des Einfügens von Aktivitäten sicherstellen zu können. Komplexe Einfügeoperationen, wie die Hinzunahme von Aktivitäten zwischen zwei Knotenmengen, können auf diese Insert-Operationen und andere Basisänderungsoperationen zurückgeführt werden (vgl. Abschnitt 7.6.1).

a) Die serialInsert-Operation

Der einfachste Fall liegt vor, wenn die neue Aktivität zwischen zwei direkt aufeinanderfolgenden Knoten $n_{src} \in N$ und $n_{dest} \in N$ des KF-Graphen CFS eingefügt werden soll. Die hierzu notwendigen KF-Schemaerweiterungen können mit Hilfe der *serialInsert*-Operation bewerkstelligt werden. Einige Beispiele für ihre Anwendung zeigt Abbildung 7-4.

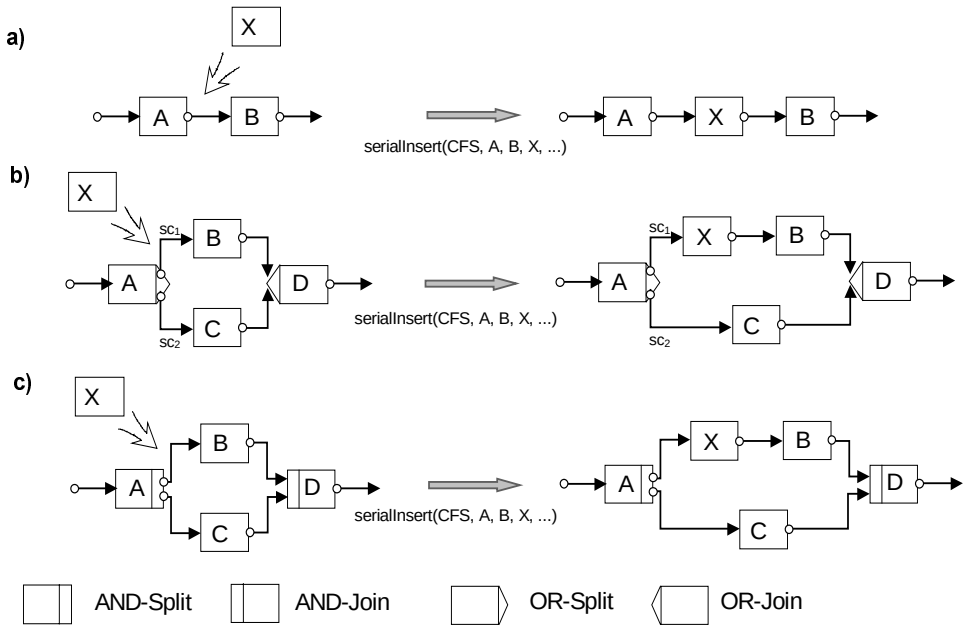


Abbildung 7-4: *Beispiele für die Anwendung der* ***serialInsert****-Operation (Einfügen von X zwischen den direkt aufeinanderfolgenden Knoten A und B)*

Um die *serialInsert*-Operation korrekt anwenden zu können, müssen die folgenden Vorbedingungen erfüllt sein (*nodeLabel* sei der Bezeichner der neu einzufügenden Aktivität):

- $nodeLabel \notin N$, d.h. es darf im KF-Graphen keinen Knoten mit gleichem Bezeichner geben
- $n_{src}, n_{dest} \in N$ und es gilt $n_{dest} \in c_succ(n_{src})$, d.h. der Knoten n_{dest} ist direkter Nachfolger von n_{src} bzgl. normaler Kontrollkanten des Typs `CONTROL_E`

Treffen dieses Bedingungen zu, so können die aus der Anwendung der *serialInsert*-Operationen resultierenden Graphersetzungen unmittelbar auf die in Abschnitt 7.2.2 eingeführten Änderungsprimitive zurückgeführt werden. Dadurch liegt auch die Semantik dieser Operation eindeutig fest.

serialInsert(CFS, n_{src}, n_{dest}, nodeLabel, template)
input
  CFS: KF-Graph, zu dem die neue Aktivität hinzugefügt werden soll
  n_{src}, n_{dest}: Anfangs- und Endaktivität, zwischen denen eingefügt wird
  nodeLabel: Bezeichner für den neu eingefügten Knoten
  template: Vorlage für neu einzufügende Aktivität (`NULL` → Nullaktivität)
begin
  AddNodes(CFS, {nodeLabel});
  if template = NULL **then**
    SetNodeAttr(CFS, {nodeLabel}, "NT", `NULL`)
  else
    SetNodeAttr(CFS, {nodeLabel}, "NT", `ACTIVITY`)
  endif
  SetNodeTemplate(CFS, nodeLabel, template);
  edgeCode:= GetEdgeAttr(CFS, (n_{src}, n_{dest}, `CONTROL_E`), "EC");[2]
  DeleteEdges(CFS, {(n_{src}, n_{dest}, `CONTROL_E`)});
  AddEdges(CFS, {(n_{src}, nodeLabel, `CONTROL_E`), (nodeLabel, n_{dest}, `CONTROL_E`)});
  SetEdgeAttr(CFS, (n_{src}, nodeLabel, `CONTROL_E`), "EC", edgeCode);
end

[2] Beim Einfügen auf Instanzebene wird zusätzlich der aktuelle Status der Kontrollkante (*src*, *dest*, `CONTROL_E`) festgestellt und anschließend der neuen Kante (*src*, *nodeLabel*, `CONTROL_E`) mittels *SetEdgeState* zugewiesen.

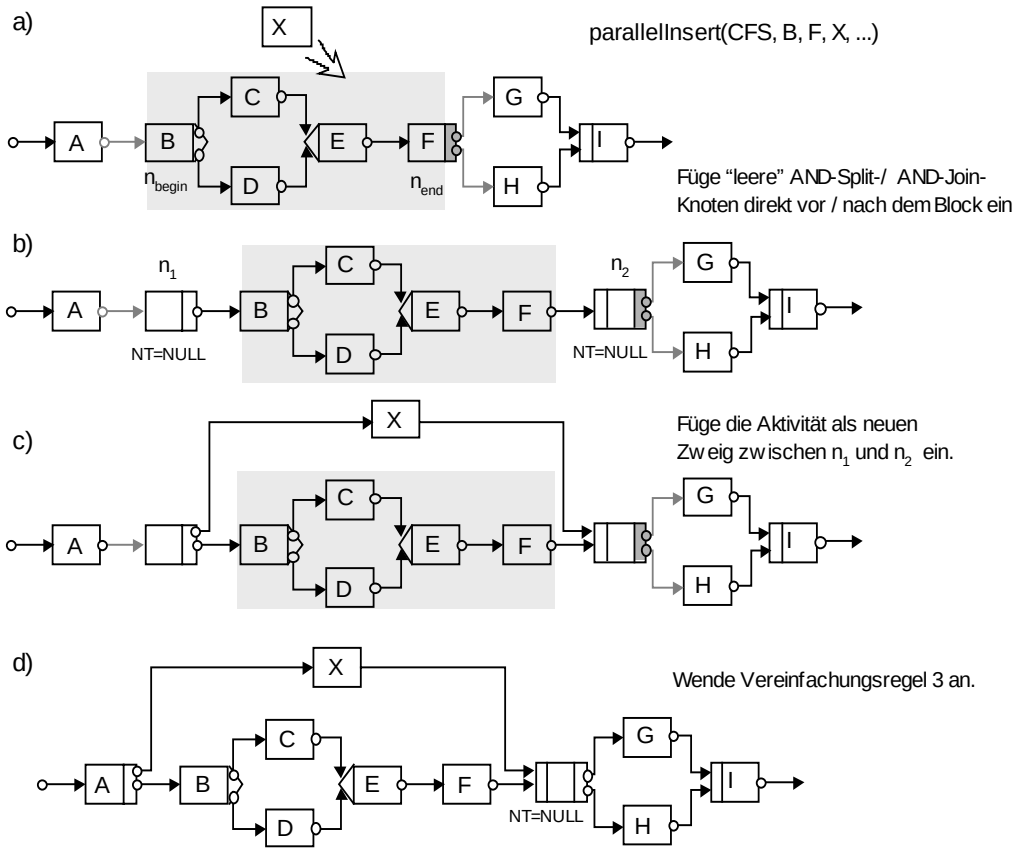


Abbildung 7-5: *Beispiel für die Anwendung der* ***parallelInsert****-Operation (Einfügen von X parallel zu dem durch (B, F) gebildeten Kontrollblock)*

Die *serialInsert*-Operation gestattet es auch, einen neuen Knoten zwischen einem OR-Split-Knoten n_{src} und dem Startknoten n_{dest} eines davon ausgehenden Teilzweiges einzufügen (vgl. Abbildung 7-4 b). Hier muß der Auswahlwert der Kontrollkante (n_{src}, n_{dest}, `CONTROL_E`) übernommen werden.

b) Die parallelInsert-Operation

Die *serialInsert*-Operation allein reicht nicht aus, um alle denkbaren bzw. praktisch relevanten Einfügeoperationen verwirklichen zu können. Im allgemeinen muß es auch möglich sein, neue Aktivitäten parallel zu einem bestimmten Bereich des KF-Graphen einzufügen. Hierzu wird von ADEPT$_{flex}$ die *parallelInsert*-Operation bereitgestellt. Sie erlaubt es, eine neue Aktivität (mit Bezeichner *nodeLabel* und Vorlage *template*) parallel zu einem bestehenden Kontrollblock (n_{begin}, n_{end}) des KF-Graphen einzufügen (n_{begin} und n_{end} beschreiben den Anfangs- bzw. Endknoten dieses Blocks). Dabei kommen die folgenden Graphtransformationen zur Anwendung (eine formale Darstellung erfolgt unten)

1. Füge einen AND-Split n_1 als direkten Vorgänger des Knotens n_{begin} sowie einen korrespondierenden AND-Join n_2 als direkten Nachfolger des Knotens n_{end} in den KF-Graphen ein. Beide Knoten repräsentieren Nullaktivitäten, besitzen also den Typ NT = `NULL`. Bei der Einbettung von n_1 in den KF-Graphen übernimmt dieser Knoten das Eingangsverhalten und die Eingangskontrollkanten (des Typs `CONTROL_E`) von n_{begin}. Entsprechend übernimmt n_2 das Ausgangsverhalten und die ausgehenden Kontrollkanten von n_{end}.
2. Füge einen neuen Knoten mit Bezeichner *nodeLabel* und Vorlage *template* als neuen Zweig in die durch n_1 und n_2 gebildete parallele Verzweigung ein.
3. Wende Graphvereinfachungsregeln (vgl. Abschnitt 7.2.3) auf das resultierende KF-Schema an.

Abbildung 7-5 zeigt ein Beispiel für die Anwendung der *parallelInsert*-Operation. Hier wird ein neuer Knoten mit Bezeichner *X* parallel zu dem grau dargestellten Kontrollblock (mit n_{begin} = `B` und n_{end} = `F`) eingefügt. Wie leicht erkennbar ist, übernimmt die neu eingefügte Nullaktivität n_2 die Ausgangssemantik (`ALL_Of_ALL`) und die Ausgangskontrollkanten des Blockendknotens `F`. Entsprechend wer-

den der Nullaktivität n_1 die Eingangssemantik (ONE_Of_ONE) und die Eingangskontrollkanten des Blockstartknotens B zugewiesen. Dies wird erforderlich, um die regelmäßige Blockstrukturierung des KF-Graphen aufrechterhalten zu können. Nach Durchführung der Transformationsschritte 1 und 2 erhalten wir den in Abbildung 7-5 c) dargestellten KF-Graphen. Er kann durch die Anwendung von Vereinfachungsregel 3 (vgl. Abschnitt 7.2.3) weiter reduziert werden, woraus sich der in Abbildung 7-5 d) dargestellte KF-Graph ergibt. Anhand dieses Beispiels läßt sich erkennen, wie bei der Anwendung der *parallelInsert*-Operation die regelmäßige Blockstrukturierung des KF-Graphen aufrechterhalten wird. Auch die anderen von ADEPT geforderten KF-Strukturierungsregeln bleiben erhalten. Um dies im allgemeinen Fall zusichern zu können, müssen die folgenden Vorbedingungen erfüllt sein:

- *nodeLabel* $\notin N$, d.h. es darf im KF-Graphen noch keinen Knoten mit gleichem Bezeichner geben
- Die beiden Knoten $n_{begin}, n_{end} \in$ N – {*Start*, *End*} legen den Start- bzw. Endknoten eines beliebigen Kontrollblocks (vgl. Definition 3-9) des KF-Graphen fest.

Die *parallelInsert*-Operation kann unabhängig vom konkreten Kontext, in den der Kontrollblock in den KF-Graphen eingebettet ist (vgl. Definition 3-10), angewendet werden. Für die Durchführung der Transformationsschritte 1 und 2 ist es unwesentlich, welche Eingangssemantik der Startknoten dieses Kontrollblocks und welche Ausgangssemantik sein Endnoten besitzen. Ebenso spielt es keine Rolle, ob ein offener oder ein geschlossener Kontrollblock (vgl. Definition 3-10) vorliegt. Die notwendigen Graphtransformationen können wieder auf die in Abschnitt 7.2.2 beschriebenen Änderungsprimitive zurückgeführt werden, wodurch die Semantik der *parallelInsert*-Operation festliegt:

parallelInsert(CFS, n_{begin}, n_{end}, nodeLabel, template)

input
...
n_{begin}, n_{end}: Start- / Endknoten des Kontrollblocks zu dem die neue Aktivität parallel eingefügt wird
...
begin
// *Erzeugen zweier Nullaktivitäten n_1 und n_2 - Vor der Anwendung des entsprechenden Primitivs*
// *AddNodes müssen für die beiden Knoten eindeutige (interne) Bezeichner generiert werden. Hier-*
// *zu stehe die Funktion GenerateUniqueNodeLabel zur Verfügung (hier nicht weiter beschrieben).*
n_1:= GenerateUniqueNodeLabel(CFS); // *Erzeuge eindeutigen Knotenbezeichner (intern)*
n_2:= GenerateUniqueNodeLabel(CFS); // *Erzeuge eindeutigen Knotenbezeichner (intern)*
AddNodes(CFS, {n_1, n_2});

// *Einfügen eines leeren AND-Split-Knotens vor dem Block. Er übernimmt das Eingangs-*
// *verhalten und die Eingangskontrollkanten des Startknotens n_{begin} des Blocks und erhält als*
// *direkten Nachfolger diesen Knoten zugewiesen.*

SetNodeAttr(CFS, n_1, "V_{in}", $V_{in}(n_{begin})$); SetNodeAttr(CFS, n_1, "V_{out}", ALL_Of_ALL);
SetNodeAttr(CFS, n_{begin}, "V_{in}", ONE_Of_ONE);

forall e $\in$ E **with** $Id_{dest}^{e} = n_{begin}$ **and** ET^{e} = CONTROL_E **do**
edgeCode:= GetEdgeAttr(CFS, e, "EC");
(Bei Anwendung auf Ausführungsgraph zusätzlich: edgeState:= GetEdgeState(CFS, e))
DeleteEdges(CFS, {e});

e_{new}:= (Id_{source}^{e}, n_1, CONTROL_E);
AddEdges(CFS, {e_{new}});
SetEdgeAttr(CFS, e_{new}, "EC", edgeCode);
(Bei Anwendung auf KF-Ausführungsgraph zusätzlich: SetEdgeState(CFS, e_{new}, edgeState))
done
AddEdges(CFS, {(n_1, n_{begin}, CONTROL_E)});

// *Einfügen eines leeren AND-Join-Knotens nach dem Block. Er übernimmt das Ausgangsver-*
// *halten und die Ausgangskontrollkanten des Blockendknotens n_{end} und erhält als direkten Vor-*
// *gänger den Knoten n_{end} zugewiesen.*

SetNodeAttr(CFS, n_2, "V_{out}", $V_{out}(n_{end})$); SetNodeAttr(CFS, n_2, "V_{in}", ALL_Of_ALL);

```
SetNodeAttr(CFS, n_end, "V_out", ONE_Of_ONE);

forall e ∈ E with (Id_source^e = n_end and ET^e = CONTROL_E) do
  edgeCode:= GetEdgeAttr(CFS, e, "EC"); edgeState:= GetEdgeState(CFS, e);
  DeleteEdges(CFS, {e});

  e_new:= ( n2, Id_dest^e, CONTROL_E);
  AddEdges(CFS, {e_new});
  SetEdgeAttr(CFS, e_new, "EC", edgeCode);
done
AddEdges(CFS, {(n_end, n2, CONTROL_E)});
// Knoten für neue Aktivität erzeugen und Vorlage zuordnen
AddNodes(CFS, {nodeLabel});
if template = NULL then
      SetNodeAttr(CFS, {nodeLabel}, "NT", NULL);
else
      SetNodeAttr(CFS, {nodeLabel}, "NT", ACTIVITY);
endif;
SetNodeTemplate(CFS, nodeLabel, template);
AddEdges(CFS, {(n1, nodeLabel, CONTROL_E)});
AddEdges(CFS, {(nodeLabel, n2, CONTROL_E)});
end
```

In Abschnitt 7.6.1 werden wir eine komplexe Einfügeoperation (→ Einfügen zwischen zwei Knotenmengen) vorstellen, die auf der *parallelInsert*-Operation basiert und bei der die Bestimmung des Kontrollblocks (n_{begin}, n_{end}) automatisch erfolgt.

c) Die branchInsert-Operation

Zur Vervollständigung der beiden vorangehend eingeführten Einfügeoperationen benötigen wir eine dritte Operation, die es uns gestattet zu einer beliebigen Verzweigung des KF-Graphen einen neuen Teilzweig hinzuzufügen. Entsprechende Erweiterungen sind im allgemeinen weder mit der *serialInsert*- noch mit der *parallelInsert*-Operation möglich. Für das Einfügen neuer Teilzweige stellt ADEPT$_{flex}$ die *branchInsert*-Operation bereit. Beispiele für ihre Anwendung zeigt Abbildung 7-6.

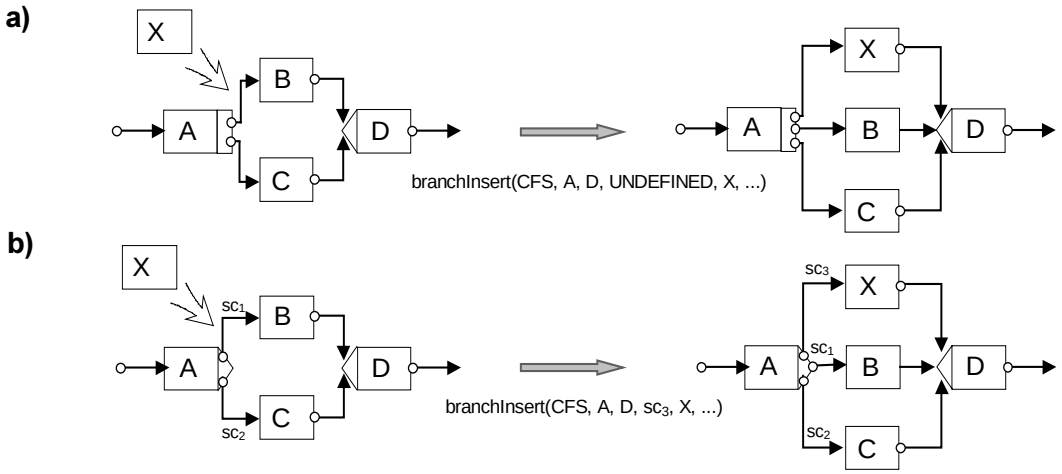


Abbildung 7-6: *Beispiele für die Anwendung der* ***branchInsert****-Operation* (*Einfügen von X als neuer Teilzweig der durch (A, D) gebildeten (a) Parallelverzweigung mit finaler Auswahl (b) bedingten Verzweigung*)

Im allgemeinen fügt diese Operation eine Aktivität (mit Bezeichner *nodeLabel* und Vorlage *template*) als neuen Teilzweig in eine Verzweigung des KF-Graphen ein. Diese Verzweigung wird durch Angabe ihres Split-Knotens *split* und des zugehörigen Join-Knotens *join* festgelegt. Um die *branchInsert*-Operation anwenden zu können, müssen die folgenden Vorbedingungen erfüllt sein:

- *nodeLabel* $\notin N$, d.h. es darf im KF-Graphen keinen Knoten mit gleichem Bezeichner geben

- $split \in SplitNodes^{CFS} \wedge join = join^{CFS}(split) \in JoinNodes^{CFS}$
- Gilt $V_{out}^{split} = V_{in}^{join}$ = `ONE_Of_ALL`, so muß für den neuen Zweig zusätzlich ein Auswahlwert SC^{new} angegeben werden. SC^{new} muß im Wertebereich des dem Split-Knoten zugeordneten Entscheidungsparameters $DP(split)$ liegen und für alle vom OR-Split-Knoten *split* ausgehenden Kontrollkanten *e* muß gelten: $SC^{e} \neq SC^{new}$.

Die notwendigen Graphtransformationen können wieder mit den in Abschnitt 7.2.2 eingeführten Änderungsprimitiven festgelegt werden:

```
branchInsert(CFS, split, join, selcode, nodeLabel, template)
input
  ...
  split, join:      Split-/Join-Knoten der Verzweigung
  selcode:          Auswahlwert bei Einfügung in bedingte Verzweigung (ansonsten UNDEFINED)
  ...
begin
  AddNodes(CFS, {nodeLabel});
  if template = NULL then
        SetNodeAttr(CFS, {nodeLabel}, "NT", NULL);
  else
        SetNodeAttr(CFS, {nodeLabel}, "NT", ACTIVITY);
  endif;
  SetNodeTemplate(CFS, {nodeLabel}, template);
  AddEdges(CFS, {(split, nodeLabel, CONTROL_E), (nodeLabel, join, CONTROL_E)});
  SetEdgeAttr(CFS, (split, nodeLabel, CONTROL_E), "EC", selcode);
end
```

Wie in Kapitel 3 erörtert, ist einer Vorlage eine Menge von Fehler- bzw. Ausnahmecodes zugeordnet. Sie faßt semantische Fehler, die zum *Scheitern* der Aktivitätenbearbeitung führen können, zusammen. In Kapitel 6 haben wir verschiedene Möglichkeiten zur Behandlung solcher semantischen Fehler diskutiert, insbesondere in Bezug auf das kontrollierte Zurücksetzen der WF-Instanz. Grundlage bildet die Verwendung von Fehlerkanten. Sie ist prinzipiell auch für neu eingefügte Aktivitäten möglich, allerdings müssen die entsprechenden Fehlerrücksprünge explizit durch Anwendung der dafür vorgesehenen Basisänderungsoperation (siehe Abschnitt 7.5.2) vereinbart werden. Erfolgt dies nicht, so ist die konkrete Art der Fehlerbehandlung zur Laufzeit vom Benutzer festzulegen (z.B. Überspringen oder Wiederholen der Aktivität, Festlegung des Rücksprungziels durch Benutzer).

7.3.1.3 Einfügen neuer Aktivitäten auf Workflow-Instanzebene

Die vorgestellten Einfügeoperationen sind sowohl auf Vorlagen- als auch auf Instanzebene anwendbar. Für ihre Anwendung auf WF-Instanzen ergeben sich zwei weitere grundlegende Fragestellungen:

1. In welchen Bearbeitungszuständen der WF-Instanz, d.h. unter welchen Zustandsmarkierungen ihres Ausführungsgraphen, darf eine neue Aktivität an der gewünschten Position eingefügt werden und in welchen Bearbeitungszuständen ist dies nicht erlaubt?
2. Welche Markierungsanpassungen müssen bei der Anwendung einer Einfügeoperation für die Knoten und Kanten des Ausführungsgraphen vorgenommen werden, um anschließend mit der WF-Kontrolle in einem konsistenten Zustand fortfahren zu können?

Der erste Punkt bezieht sich darauf, daß strukturelle Änderungen des WF nicht in jedem Bearbeitungszustand der WF-Instanz zugelassen werden dürfen, d.h. die Anwendbarkeit von Einfügeoperationen unterliegt auf Instanzebene, zusätzlich zu den vorgestellten *strukturellen* Voraussetzungen, auch gewissen *Statuseinschränkungen*. Ihre Überprüfung muß *vor* dem Einfügen der Aktivität erfolgen. Der zweite Punkt betrifft Maßnahmen, die *nach* der Durchführung einer solchen Einfügeoperation vorgenommen werden müssen. Beide Aspekte sind essentiell für das dynamische Hinzufügen von Aktivitäten zu einer sich in Ausführung befindlichen WF-Instanz.

Im folgenden bezeichne $CFS_{instance}$ = (*CFS*, *NS*, *ES*, *It*) den KF-Ausführungsgraphen der WF-Instanz vor dem Einfügen der neuen Aktivität. Wir zeigen zunächst, welche Zustandsbeschränkungen für die drei Basiseinfügeoperationen bestehen. Generell fordern wir, daß eine neue Aktivität bzw. ein neuer Knoten – mit Ausnahme von Nullaktivitäten – niemals vor eine bereits gestartete, beendete oder nicht mehr ausführbare Aktivität (→ Knotenmarkierung `RUNNING`, `COMPLETED` oder `SKIPPED`) eingefügt werden darf. Wie aus Beispiel 7-5 ersichtlich wird, würden sich dadurch im Ausführungsgraphen inkonsistente Markierungen und damit ein undefiniertes Ausführungsverhalten ergeben.

Beispiel 7-5 (Unzulässige Einfügeoperationen auf Instanzebene)

In Abbildung 7-7 soll die *serialInsert*-Operation (→ Einfügen von `X` zwischen `A` und `B`) auf einen KF-Graphen (Vorlage) und auf einen davon abgeleiteten, strukturgleichen KF-Ausführungsgraphen (Instanz) angewendet werden. Auf Vorlagenebene ist die Operation korrekt anwendbar, nicht dagegen auf Instanzebene: Würde man `X` im aktuellen Zustand des Ausführungsgraphen zwischen `A` und `B` einfügen und anschließend den Status dieses Graphen nach den in Kapitel 4 vorgestellten Regeln neu bewerten, so ergäbe sich der unten rechts dargestellte KF-Ausführungsgraph. Es ist offensichtlich, daß die Markierung dieses Graphen inkonsistent ist (vgl. Satz 4-1).

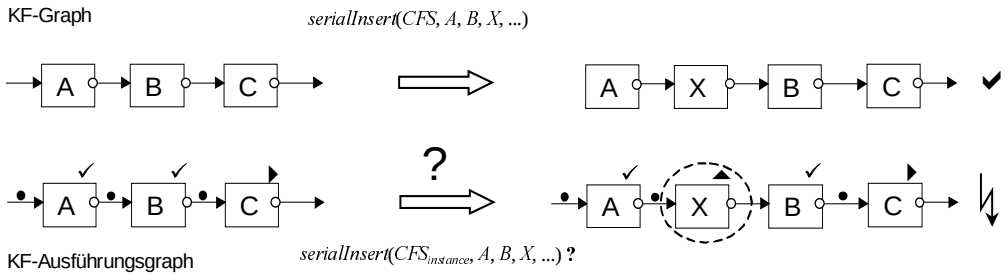


Abbildung 7-7: *Inkonsistente Statusmarkierungen bei unkontrollierter Anwendung einer Einfügeoperation.*

Um Einfügeoperationen auf Instanzebene korrekt anwenden zu können, müssen bestimmte Bedingungen hinsichtlich des aktuellen WF-Status festgelegt werden:

- *serialInsert*: Für das serielle Einfügen einer Aktivität (mit Typ *NT* = `ACTIVITY`) in den KF-Ausführungsgraphen $CFS_{instance}$ muß sich der Knoten n_{dest}, vor den eingefügt werden soll, in einem der beiden Zustände `NOT_ACTIVATED` oder `ACTIVATED` befinden. In letzterem Fall – Einfügen vor einem bereits aktivierten, aber noch nicht gestarteten Knoten – wird die Aktivierung von n_{dest} wieder aufgehoben. Damit verbunden ist das Entfernen entsprechender Einträge aus Arbeitslisten.
- *parallelInsert*: Soll eine Aktivität (mit Typ *NT* = `ACTIVITY`) parallel zu einem Kontrollblock (n_{begin}, n_{end}) des KF-Ausführungsgraphen $CFS_{instance}$ eingefügt werden, müssen sich alle direkten Nachfolger des Knotens n_{end} (bzgl. Kontroll- und Sync-Kanten) ebenfalls in einem der beiden Zustände `NOT_ACTIVATED` oder `ACTIVATED` befinden. Dies ist notwendig, um zwischen dem Knoten n_{end} und seinen Nachfolgern einen AND-Join einfügen zu können (vgl. Abschnitt 7.3.1.2).
- *branchInsert*: Im Hinblick auf das Einfügen neuer Teilzweige müssen Statusbeschränkungen differenzierter festgelegt werden, abhängig von der Art der Verzweigung. Für parallele Verzweigungen bleibt die *branchInsert*-Operation solange anwendbar, wie der AND-Join-Knoten noch nicht gestartet wurde. Abgesehen davon bestehen keine weiteren Restriktionen. Demzufolge können einer parallelen Verzweigung auch dann noch Teilzweige hinzugefügt werden, wenn ihr AND-Split-Knoten oder einzelne Teilzweige bereits beendet worden sind (für ein Beispiel siehe Abbildung 7-10). Dies trifft im wesentlichen auch auf Parallelverzweigungen mit finaler Auswahl zu. Allerdings ist hier das Einfügen neuer Teilzweige nur dann möglich, wenn der OR-Join-Knoten noch nicht aktiviert wurde, d.h. die Auswahl des Siegerzweiges noch nicht stattgefunden hat. Noch etwas anders verhält es sich mit dem Einfügen neuer Teilzweige in eine bedingte Verzweigung. Da hier die Verzweigungsentscheidung durch den OR-Split-Knoten getroffen wird, macht das Einfügen neuer Zweige nur dann Sinn, solange dieser Split-Knoten noch nicht beendet worden ist.

Tabelle 7-6 faßt die Statusvoraussetzungen für die verschiedenen Operationen nochmals zusammen:

Art der Einfügeoperation	Statusvoraussetzungen
serialInsert($CFS_{instance}$, *src*, *dest*, ...)	$NS(dest) \in$ {NOT_ACTIVATED, ACTIVATED}
parallelInsert($CFS_{instance}$, n_{begin}, n_{end}, ...)	$\forall\ n \in succ(n_{end})$: $NS(n) \in$ {NOT_ACTIVATED, ACTIVATED}
branchInsert($CFS_{instance}$, *split*, *join*, [*selcode*],...)	Parallele Verzweigung: $NS(join) \in$ {NOT_ACTIVATED, ACTIVATED} Parallele Verzweigung mit finaler Auswahl: $NS(join) \in$ {NOT_ACTIVATED} Bedingte Verzweigung: $NS(split) \notin$ {COMPLETED, SKIPPED}

Tabelle 7-6: *Statusvoraussetzungen für die Anwendung von Insert-Operationen*

Gestatten die Markierungen eines Ausführungsgraphen die Anwendung einer Einfügeoperation nicht, muß die beabsichtigte Änderung verworfen werden. In Einzelfällen besteht die Möglichkeit, die WF-Bearbeitung in einen früheren Zustand zurückzusetzen (vgl. Kapitel 6), in dem die Änderung eingebracht werden kann. Beim Einfügen innerhalb von Schleifenkörpern kann es sinnvoll sein, Änderungen, die in der aktuellen Schleifeniteration infolge des zu weit fortgeschrittenen WF-Status nicht mehr anwendbar sind, vorzumerken und sie vor Beginn der nächsten Iteration „verspätet“ einzubringen.

Nach dem dynamischen Einfügen einer Aktivität auf Instanzebene muß der Status des zugrundeliegenden KF-Ausführungsgraphen neu bewertet werden. Diese Neubewertung erfolgt auf Basis der in Kapitel 4 vorgestellten Markierungs- und Ausführungsregeln. Abgesehen von der gesonderten Behandlung von Nullaktivitäten sind keine zusätzlichen Regeln erforderlich, um die notwendigen Statusanpassungen (z.B. die Festlegung von Zuständen für neu eingefügte Knoten und Kanten) vorzunehmen. Ob eine neu hinzugefügte Aktivität sofort, später oder möglicherweise überhaupt nicht aktiviert wird, hängt vom konkreten Kontext ab, in dem sie in den Ausführungsgraphen eingebettet wird. Beispielsweise erfolgt für seriell eingefügte Aktivitäten die Aktivierung immer dann sofort, wenn der Quellknoten *src* zum Zeitpunkt des Einfügens bereits den Status COMPLETED besitzt. Andererseits muß eine neu eingefügte Aktivität in der Folge nicht notwendigerweise zur Ausführung kommen. Dies ist der Fall, wenn die Aktivität in einen Teilzweig einer bedingten Verzweigung eingefügt wird und die Verzweigungsentscheidung noch aussteht. Generell gilt, daß Statusanpassungen nicht notwendig werden, wenn die Änderung in einem Bereich des Ausführungsgraphen durchgeführt wird, dessen Knoten und Kanten aktuell als NOT_ACTIVATED bzw. NOT_SIGNALED markiert sind.

Beispiel 7-6 (Durchführung von Markierungsanpassungen nach dynamischem Einfügen von Aktivitäten)

Auf den KF-Ausführungsgraphen aus Abbildung 7-8 a) werden verschiedene Einfügeoperationen angewendet: Abbildung 7-8 b) zeigt den Graphen, nachdem X parallel zu B eingefügt und sein Status neu bewertet wurde. Die Neubewertung des Status basiert auf der Markierungsregel M_1 und der Ausführungsregel A_1. Das Ergebnis einer seriellen Einfügung (*serialInsert*) zeigt Abbildung 7-8 c). Hier wurde X direkt vor den bereits aktivierten Schritt B eingefügt. Dem entsprechend wurde vor dem Einfügen von X die Aktivierung von B wieder aufgehoben. X ist nach der Neubewertung des Graphen sofort aktiviert (Ausführungsregel A_1). Ein zweites Beispiel für das serielle Einfügen zeigt Abbildung 7-8 d). Im Gegensatz zu c) werden hier keine Markierungsanpassungen notwendig. Der Grund dafür ist, daß die Änderung in einem Bereich des Graphen erfolgt, der noch nicht „betreten“ worden ist. Nicht zulässig wäre im aktuellen Bearbeitungsstatus des Graphen aus Abbildung 7-8 a) das Einfügen von X zwischen L_S und A. Unter gewissen Voraussetzungen können solche Änderungen vorgemerkt und zu Beginn der nächsten Schleifeniteration eingebracht werden.

Wie bereits erwähnt, nehmen *Nullaktivitäten* im ADEPT-Modell eine Sonderrolle ein. Wie das Beispiel der *parallelInsert*-Operation zeigt, kann es in bestimmten Fällen sinnvoll sein, eine Nullaktivität – im Gegensatz zu normalen Aktivitäten – auch vor einem bereits gestarteten oder beendeten Knoten einzufügen. Aufgrund der speziellen Semantik von Knoten dieses Typs ist dies unproblematisch. In einem solchen Fall wird die Nullaktivität sofort als COMPLETED markiert. Zusätzlich wird in der Ablaufhistorie ein entsprechender Eintrag an der „richtigen“ Stelle vorgenommen, um den WF im Fehler-

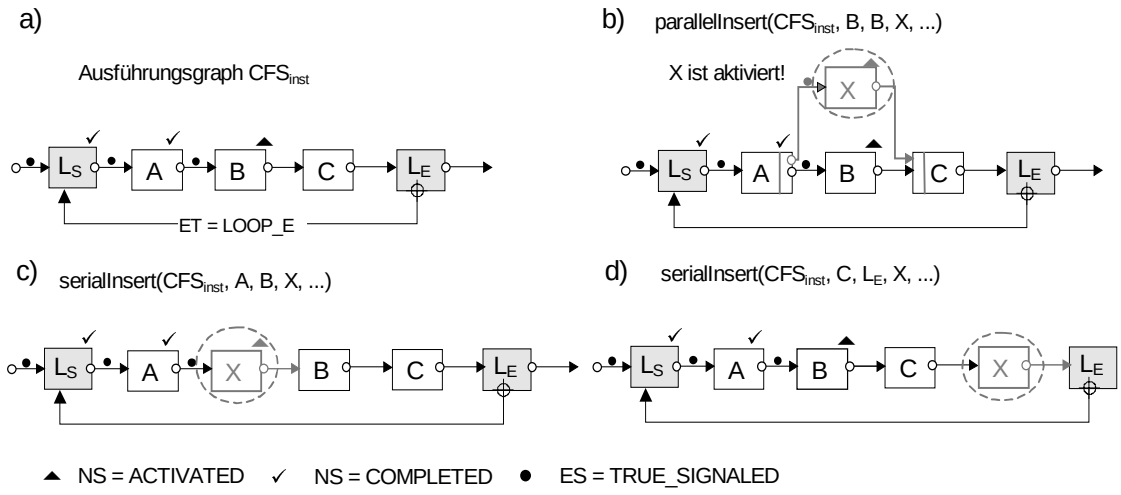


Abbildung 7-8: *Dynamisches Einfügen in Ausführungsgraphen und begleitende Statusanpassungen.*

fall korrekt zurücksetzen zu können (siehe Kapitel 6). Wir verzichten an dieser Stelle auf die Darstellung weiterer Details.

Die in diesem Abschnitt diskutierten Statusvoraussetzungen und -anpassungen stellen sicher, daß beim dynamischen Einfügen von Aktivitäten nicht nur ein korrekt strukturierter KF-Graph, sondern auch wieder konsistente Graphmarkierungen resultieren. Das bedeutet insbesondere, daß wichtige Dynamikeigenschaften, wie Terminierung des WF oder Aktivierbarkeit von Aktivitäten (inkl. des neu hinzugefügten Knotens), weiterhin gelten. Die Grundlage hierfür bilden die vom ADEPT-Basismodell festgelegten Markierungs- und Ausführungsregeln und die daraus ableitbaren Aussagen zum dynamischen Verhalten (vgl. Kap. 4). Die Vorteile des gewählten Markierungsansatzes sind im Zusammenhang mit der Überprüfung von Statusvoraussetzungen ebenfalls nochmals deutlich geworden. Durch den Erhalt der Markierungen bereits abgearbeiteter oder nicht mehr ausführbarer WF-Bereiche können Informationen zum bisherigen Verlauf der WF-Ausführung – im Gegensatz z.B. zu Petri-Netz-basierten WF-Metamodellen [AAH98, Ober96] – direkt aus dem Ausführungsgraphen entnommen werden. Dem entsprechend einfach ist feststellbar, ob eine Änderung im aktuellen Status eingebracht werden kann oder nicht. Dagegen müßten bei Petri-Netz-basierten Ansätzen aufwendige Erreichbarkeitsanalysen [Baum96] durchgeführt werden, um herauszufinden, ob eine neu eingefügte Transition unter einer Folgemarkierung aktivierbar ist oder nicht. Schließlich gestatten die Markierungs- und Ausführungsregeln des ADEPT-Basismodells die automatische Anpassung von Knoten- und Kantenmarkierungen nach dem dynamischen Einfügen von Aktivitäten. Wesentlich dabei ist, daß diese Regeln unabhängig vom Kontext, in den die Aktivität eingefügt wird, anwendbar sind.

7.3.1.4 Begleitende Anpassungen des Datenflusses

Im allgemeinen werden beim Einfügen neuer Aktivitäten in einen KF-Graphen begleitende Anpassungen des Datenflusses erforderlich, um dessen Korrektheit aufrechtzuerhalten. Wie eingangs erwähnt, wird die einzufügende Aktivität durch Angabe eines eindeutigen Bezeichners und durch Zuordnung einer Vorlage funktional spezifiziert. Dadurch liegen auch ihre Ein- und Ausgabeparameter und deren Eigenschaften (u.a. Wertebereich, Nachforderbarkeit, optionaler/obligater Parameter) explizit fest (siehe Kapitel 3). Diese Vorgaben werden für die korrekte Abbildung der Lese- und Schreibzugriffe von Aktivitätenparametern auf den WF-Datenkontext benötigt.

Soll eine Aktivität mit Bezeichner X und Vorlage *templ* neu eingefügt werden, so können – unabhängig von der gewählten Art der Einfügeoperation – die folgenden Festlegungen bzgl. der Anknüpfung von Aktivitätenparametern getroffen werden:

- $IP_{add} \subseteq InParams^X$ ($\equiv InParams^{templ}$): Teilmenge der (nachforderbaren) Eingabeparameter von X, deren Versorgung durch einen vorgeschalteten, interaktiven Nachforderungsdienst erfolgen soll.
- $OP_{add} \subseteq OutParams^X$ ($\equiv OutParams^{templ}$): Teilmenge der Ausgabeparameter von X, die an ein neu zu generierendes Datenelement geknüpft werden sollen

- $IN_{map} \subset InParams^X \times D$: Menge von Bindungen zwischen Aktivitäteneingabeparametern und globalen Datenelementen; $(par, d) \in IN_{map} \Rightarrow$ es soll eine Lesekante vom Eingabeparameter *par* zum Datenelement *d* eingefügt werden. (Das Datenelement *d* kann auch von vorangehenden Änderungsoperationen derselben Änderungstransaktion eingefügt worden sein.)
- $OUT_{map} \subset OutParams^X \times D$: Menge von Bindungen zwischen Aktivitätenausgabeparametern und globalen Datenelementen; $(par, d) \in OUT_{map} \Rightarrow$ es soll eine Schreibkante vom Ausgabeparameter *par* zum Datenelement *d* eingefügt werden.

Diese Mengen werden beim Einfügen neuer Aktivitäten in den WF-Graphen und der damit verbundenen Adaption des zugehörigen Datenflußmodells wie folgt verwendet (wir greifen wieder auf die in Abschnitt 7.2.2 eingeführten Änderungsprimitive zurück):

```
[serial | parallel | branch] Insert(CFS, DFS, ..., X, templ, ..., IP_add, OP_add, IN_map, OUT_map)
input
  ...
  CFS, DFS:        Abzuänderndes KF-/DF-Schema
  X, template:     Bezeichner / Vorlage für die einzufügende Aktivität
  ...
begin
  // Einfügen einer Aktivität mit Bezeichner X und Vorlage template in den KF-Graphen CFS
  // (siehe hierzu Abschnitt 7.3.1.2)
  ....
  // Installieren vorgeschalteter Nachforderungsdienste für Parameter aus IP_add
  forall par ∈ IP_add do
      dataLabel:= generateNewId(CFS, par);   // Bezeichner für neues Datenelememt erzeugen
      AddDataElements(CFS, DFS, {dataLabel}, pDom^par, UNDEFINED);
      AddDataEdges(CFS, DFS, {(dataLabel, X, par, read)});
  done
  AddParamProvisionService(CFS, DFS, X, IP_add);  // vgl. Tabelle 7-4
  // Anlegen neuer Datenelemente und Schreibzugriffe für Parameter aus OP_add
  forall par ∈ OP_add do
      dataLabel:= generateNewId(CFS, par);   // Bezeichner für neues Datenelememt erzeugen
      AddDataElements(CFS, DFS, {dataLabel}, pDom^par, UNDEFINED);
      AddDataEdges(CFS, DFS, {(dataLabel, X, par, write)});
  done
  // Lese- und Schreibzugriffe auf existierende Datenelemente einfügen
  newDataEdges:= ∅;
  forall (par, d) ∈ IN_map do
      newDataEdges:= newDataEdges ∪ {(d, X, par, read)}
  done
  forall (par, d) ∈ OUT_map do
      newDataEdges:= newDataEdges ∪ {(d, X, par, write)}
  done
  AddDataEdges(CFS, DFS, newDataEdges);
end // [serial | parallel | branch] Insert
```

Bevor wir auf Beschränkungen in Bezug auf die Festlegung der Mengen IP_{add}, OP_{add}, IN_{map} und OUT_{map} eingehen – sie werden notwendig, um wieder zu einem korrekten DF-Modell zu gelangen – geben wir ein einfaches Beispiel an:

Beispiel 7-7 (Anpassung von DF- und Datenkontext-Schema beim Einfügen neuer Aktivitäten)

Abbildung 7-9 zeigt zwei Beispiele für das Einfügen einer Aktivität `X` (inkl. ihrer Lese- und Schreibzugriffe auf den WF-Datenkontext) in einen WF-Graphen. In beiden Fällen wird davon ausgegangen, daß `X` je einen Eingabe- und Ausgabeparameter (in_X bzw. out_X) besitzt. In a) wird der Eingabeparameter in_X auf das vorhandene

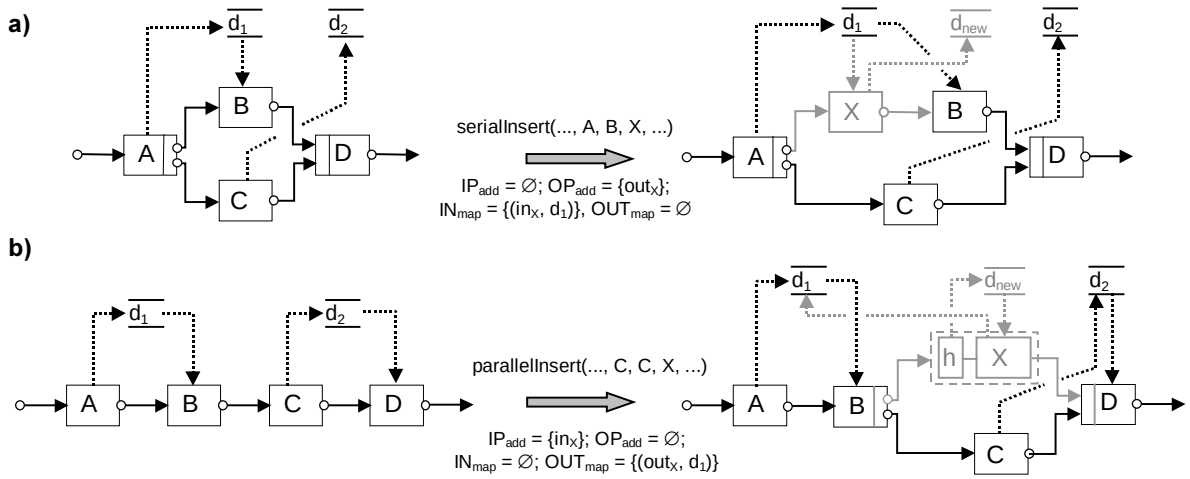


Abbildung 7-9: (*Dynamisches) Einfügen und begleitende Anpassungen des Datenflusses*

Datenelement d_1 abgebildet. Der eingefügte Lesezugriff kann versorgt werden, da für den neuen WF-Graphen (CFS^*, DFS^*) gilt: $WriterExists(CFS^*, DFS^*, \mathtt{X}, \mathtt{d_1}) = True$. Der Ausgabeparameter out_X wird seinerseits an das neu eingefügte Datenelement $\mathtt{d_{new}}$ geknüpft. Auf dieses Datenelement können z.B. Folgeänderungen Bezug nehmen. Nicht zulässig dagegen wäre die Abbildung von out_X auf $\mathtt{d_2}$, da dieses Datenelement von der parallelen Aktivität C geschrieben wird (Regel DF-2). In Abbildung b) wird zur Versorgung von in_X der Aktivität ein (interaktiver) Nachforderungsdienst vorgeschaltet. Dieser Hilfsdienst schreibt seinen Ausgabewert in das neu eingefügte Datenelement $\mathtt{d_{new}}$, das dann von X gelesen wird. Der Ausgabeparameter out_X wird auf das vorhandene Datenelement $\mathtt{d_1}$ abgebildet, wodurch die Korrektheit des Datenflusses erhalten bleibt.

Im folgenden bezeichne $WS = (CFS, DFS)$ den WF-Graphen *vor* und $WS^* = (CFS^*, DFS^*)$ den WF-Graphen *nach* dem Einfügen der neuen Aktivität (inkl. ihrer Lese- und Schreibzugriffe auf den WF-Datenkontext). Um zu einem korrekten DF-Modell DFS^* zu gelangen, muß sichergestellt werden, daß

- die Wertebereiche von Datenelementen und Parametern kompatibel zueinander sind
- alle obligaten Eingabeparameter der hinzugefügten Aktivität sicher versorgt werden können
- die eingefügte Aktivität keine Schreibzugriffe auf Datenelemente erhält, die von parallelen Knoten geschrieben werden.

Die beiden zuletzt genannten Forderungen sind grundlegend für den Erhalt der in Kapitel 5 vorgestellten Regeln DF-1 und DF-2. (Die Regel DF-3, die zwei direkt aufeinanderfolgende Schreibzugriffe auf ein Datenelement ausschließt, klammern wir aus Gründen der besseren Lesbarkeit aus.) Wir geben nun Bedingungen für die Bildung der Mengen IP_{add}, OP_{add}, IN_{map} und OUT_{map} an, die notwendig und hinreichend für die Korrektheit des aus der Einfügeoperation resultierenden DF-Modells DFS^* sind. Dazu benötigen wir noch die folgenden beiden Parametermengen:

$$IP_{map} := \{par \in InParams^X \mid \exists\, d \in D \text{ mit } (par, d) \in IN_{map}\} \text{ und}$$
$$OP_{map} := \{par \in OutParams^X \mid \exists\, d \in D \text{ mit } (par, d) \in OUT_{map}\}$$

Bzgl. der Festlegung der Mengen IP_{add} und IN_{map} treffen wir die folgenden Einschränkungen

1. $IP_{add} \subseteq \{par \in InParams^X \mid pRm^{par} = \mathtt{USER_REQUEST}\}$
2. $IN_{map} \subset \{ (par, d) \in InParams^X \times D \mid pDom^{par} \supseteq dom^d\}$
   Für $(par_1, d_1), (par_2, d_2) \in IN_{map}$ muß gelten: $[\, par_1 = par_2 \Rightarrow d_1 = d_2 \,]$
3. $IP_{add} \cap IP_{map} = \emptyset$ und $IP_{add} \cup IP_{map} \supseteq \{ par \in InParams^X \mid pDm^{par} = \mathtt{MANDATORY}\}$
4. $\forall\, (par, d_r) \in IN_{map}$ mit $pDM^{par} = \mathtt{MANDATORY}$
   $\Rightarrow d_r \in C_X := \{d \in D \mid WriterExists(CFS^*, DFS, X, d) = True\}$

Die 1. Einschränkung wird notwendig um sicherzustellen, daß Nachforderungsdienste ausschließlich für nachforderbare Eingabeparameter (vgl. Kapitel 3) vorgeschaltet werden. Soll ein Eingabeparameter dagegen an ein bereits vorhandenes Datenelement angeknüpft werden, ist darauf zu achten, daß ihre Wertebereiche kompatibel zueinander sind und daß der Eingabeparameter nicht an mehrere Datenelemente gebunden wird. Dies erreichen wir mit der 2. Forderung. Die 3. Bedingung gewährleistet, zusammen mit der 2. Bedingung, daß von jedem Eingabeparameter *maximal* eine und von jedem obligaten Eingabeparameter *genau* eine Lesekante ausgeht. Bedingung 4 schließlich garantiert, daß obligate Lesezugriffe in dem aus der Änderung resultierenden WF-Graphen sicher versorgt sind. Für Parameter der Menge IP_{add} muß dies nicht mehr explizit gefordert werden, während für Parameter der Menge IP_{map} die Versorgung des Lesezugriffs auf Basis des in Kapitel 5 vorgestellten *WriterExists*-Verfahrens erfolgt. (Anmerkung: Die Bedingung 4 kann auch unter Bezugnahme auf den Ausgangsgraphen *CFS* formuliert werden. Für die *parallelInsert*-Operation etwa ergibt sich z.B.: „... $d_r \in \{d \in D \mid WriterExists(CFS, DFS, n_{begin}, d) = True\}$“.)

Ähnliche Einschränkungen müssen auch in Bezug auf die Anknüpfung von Ausgabeparametern, d.h. die Festlegung der Mengen OP_{add} und OUT_{map} getroffen werden:

5. $OUT_{map} \subset \{ (par, d) \in OutParams^X \times D \mid pDom^{par} \subseteq dom^d \}$
   Für $(par_1, d_1), (par_2, d_2) \in OUT_{map}$ muß gelten: $[\, par_1 = par_2 \Leftrightarrow d_1 = d_2 \,]$
6. $OP_{add} \cap OP_{map} = \emptyset$
7. Für $(par, d) \in OUT_{map}$ muß gelten: $\forall\, n \in N$ mit $Writes^{CFS}(n, d) = True$ und
   $(split, join) := BranchNodes^{CFS^*}(n, X) \neq (\texttt{UNDEFINED}, \texttt{UNDEFINED}) \Rightarrow V_{in}^{join} \neq \texttt{ALL_Of_ALL}$

Soll ein Ausgabeparameter an ein bereits vorhandenes Datenelement angeknüpft werden, müssen ihre Wertebereiche kompatibel zueinander sein. Darüber hinaus darf von einem Ausgabeparameter maximal eine Schreibkante ausgehen. Umgekehrt dürfen verschiedene Parameter der eingefügten Aktivität ihre Ausgaben niemals in dasselbe Datenelement schreiben. Diesen Forderungen kommen wir durch die Einschränkungen 5 und 6 nach. Bedingung 7 stellt sicher, daß es durch die Abbildung der Ausgabeparameter auf vorhandene Datenelemente nicht zu parallelen Schreibzugriffen anderer Knoten kommt. Aus diesen Überlegungen heraus folgt unmittelbar der Satz 7-2:

Satz 7-2 (Erhalt der Korrektheit von DF-Schemata beim Einfügen von Aktivitäten)

Sei $WS = (CFS, DFS)$ ein korrekter WF-Graph, in den eine Aktivität (inkl. Datenzugriffen) mit Bezeichner X und Vorlage $template \in Templates$ eingefügt werden soll. Ferner seien durch $(IP_{add}, OP_{add}, IN_{map}, OUT_{map})$ die für X durchzuführenden Parameterbindungen festgelegt. Dann gilt:

Erfüllen IP_{add}, OP_{add}, IN_{map} und OUT_{map} die Bedingungen 1–7, so resultiert aus der Anwendung der beschriebenen Insert-Operationen wieder ein korrekter WF-Graph $WS^* = (CFS^*, DFS^*)$. Insbesondere ergibt sich wieder ein korrektes DF-Modell DFS^*, das den Regeln DF-1 und DF-2 genügt.

Wir kennen nun die generellen Einschränkungen, denen Parameterbindungen neu einzufügender Aktivitäten unterliegen. In der Regel besteht hinsichtlich ihrer konkreten Festlegung Gestaltungsspielraum, abhängig von den durch die Vorlage definierten Parametereigenschaften sowie der Zusammensetzung des Datenkontext-Schemas. Ein spezieller Fall liegt vor, wenn alle obligaten Eingabeparameter der Aktivität X (bzw. ihrer Vorlage) nachforderbar sind und ihre Versorgung durch einen vorgeschalteten Hilfsdienst erfolgen soll. Hier erhalten wir $IP_{add} = InParams^X$ und $IN_{map} = \emptyset$, so daß die Bedingungen 1–4 trivialerweise erfüllt sind. Ähnliches gilt auch für Ausgabeparameter, wenn sie ausschließlich an neu zu generierende Datenelemente angeknüpft werden sollen ($OP_{add} \subseteq OutParams^X$, $OUT_{map} = \emptyset$). In der Praxis führen diese einfachen Ansätze allerdings nicht immer zu befriedigenden Lösungen, da unnötige oder redundante Dateneingaben resultieren können oder nachfolgende Schritte nicht auf die Ausgabewerte neu eingefügter Aktivitäten zugreifen können. Spätestens dann müssen Parameter (dynamisch) auf existierende Datenelemente abbildbar sein. Abhängig vom Anwendungskontext der Einfügeoperation stehen diese Verknüpfungen vorab fest (z.B. wenn eine neue Aktivität eingefügt werden soll, um fehlende oder fehlerhafte Ausgaben vorangehender WF-Aktivitäten festzulegen bzw. zu korrigieren) oder aber sie müssen vom Anwender explizit vorgenommen werden (z.B. beim Ad-hoc-

Einfügen einer Aktivität). Nicht immer ist es möglich, die obligaten Eingabeparameter des neu hinzugefügten Schrittes X vollständig auf vorhandene Datenelemente abzubilden oder ihre Versorgung durch Vorschalten von Nachforderungsdiensten sicherzustellen. In solchen Fällen muß die Einfügeoperation entweder zurückgenommen werden oder es sind begleitende Änderungen durchzuführen. Beispielsweise kann im KF-Graphen vor dem neuen Schritt X eine weitere Aktivität Y eingefügt werden, deren Ausgaben dann zur Versorgung der noch fehlenden Eingabeparameter von X verwendet werden. Um den Anwender bei der Auswahl solcher „Versorgeraktivitäten" zu unterstützen, können einzelne Parameter (bzw. Gruppen von Parametern) bei ihrer Beschreibung (vgl. Kap. 3) als „speziell nachforderbar" gekennzeichnet und mit einer Vorlage für einen Versorgungsschritt verknüpft werden. Diese Vorlage kann durch eine beliebige Anwendungskomponente implementiert werden.

Das skizzierte Beispiel zeigt, daß eventuell mehrere Basisänderungsoperationen zusammenhängend ausgeführt werden müssen, um wieder ein korrektes DF-Modell zu erhalten. Wie wir später sehen werden, trifft ähnliches auch auf das Löschen von Aktivitäten zu. ADEPT erlaubt es deshalb, mehrere Basisänderungsoperationen zu einer *Änderungstransaktion* zusammenzufassen und stellt sicher, daß bei ihrer Ausführung entweder alle Operationen ausgeführt werden oder gar keine. Die Überprüfung der Korrektheit des Datenflusses erfolgt dabei mit Bezug auf die Gesamtheit der angewandten Änderungsoperationen. Ebenso werden Knoten- und Kantenmarkierungen erst im Anschluß an die Durchführung dieser Operationen, d.h. beim Commit der Änderungstransaktion, angepaßt. Jede Änderungstransaktion überführt einen WF-Instanz, ausgehend von einem korrekten Zustand, wieder in einen neuen, ebenfalls korrekten Zustand. Zwischenzustände, die sich nach der Anwendung einzelner Operationen oder Änderungsprimitive ergeben, sind für andere Transaktionen nicht sichtbar.

Wir diskutieren kurz einen wichtigen Aspekt, der sich beim Einfügen einer Aktivität X und ihrer Datenzugriffe auf *WF-Instanzebene* ergibt. Bisher haben wir gefordert, daß bei der Anknüpfung obligater Eingabeparameter an existierende Datenelemente gewährleistet ist, daß diese sicher versorgt sind, unabhängig davon, welche Ausführungszweige vor der Aktivierung von X konkret durchlaufen werden. Dem entsprechend ist die Menge der Datenelemente, auf die obligate Eingabeparameter von X abgebildet werden dürfen, eingegrenzt durch

$$C_X := \{d \in D \mid WriterExists(CFS^*, DFS, X, d) = True\}$$

Beim Einfügen einer neuen Aktivität X auf Instanzebene kann diese Forderung ggf. weniger restriktiv gestaltet werden, indem zusätzlich der aktuelle Status, in dem sich die WF-Instanz zum Zeitpunkt der Änderung befindet, und die bisher stattgefundenen Schreibzugriffe berücksichtigt werden. Durch die Ausnutzung von Informationen zum bisherigen Verlauf der WF-Ausführung kann deshalb die Menge der Datenelemente, auf die (obligate) Eingabeparameter des neu eingefügten Schrittes potentiell abgebildet werden können, ggf. erweitert werden. Es ergibt sich:

$$C_X^{(inst)} := C_X \cup \{d \in D \mid ReadDataElement(CFS_{instance}^*, X, d) \neq \mathtt{NULL}\}$$

Für die Formulierung dieser Menge haben wir die in Kapitel 5 eingeführte Funktion *ReadDataElement* verwendet. Bei der Abbildung von Lesezugriffen auf Datenelemente der Menge $C_X^{(inst)}$ kann es im Zusammenhang mit Schleifen- oder Fehlerrücksprüngen notwendig werden, das Einfügen von X wieder zurückzunehmen. Wir werden in Kapitel 8 zeigen, daß es für die Anwendbarkeit der Einfügeoperation hier wichtig ist, ob X (maximal) einmal zur Ausführung kommen soll (*temporäres Einfügen*) oder bis zur Beendigung der WF-Instanz erhalten bleiben soll (*permanentes Einfügen*). Des weiteren müssen bei der Bildung der Menge $C_X^{(inst)}$ Schreibzugriffe auf d, die innerhalb von Zweigen einer noch nicht abgeschlossenen Parallelverzweigung mit finaler Auswahl erfolgten, gesondert behandelt werden. Wir verzichten an dieser Stelle auf die Darstellung dieses Sonderbehandlung.

Beispiel 7-8 (Ausnutzen des WF-Status beim Einfügen von Lesezugriffen)

Im Ausführungsgraphen aus Abbildung 7-10 erhält die neu eingefügte Aktivität X einen Lesezugriff auf das Datenelement d, obwohl d im KF-Graphen nicht in jedem Ausführungspfad, der zur Aktivierung von X führt, geschrieben wird (d.h. $d \notin C_X$). Hier kann zusätzlich die Information genutzt werden, daß der obere Ausführungszweig zum Zeitpunkt des Einfügens bereits ausgeführt worden ist und B das Datenelement d geschrieben hat (angezeigt durch die signalisierte DF-Schreibkante). Dem entsprechend gilt: $d \in C_X^{(inst)}$. Wird der WF zu einem späteren Zeitpunkt in einen Zustand vor Ausführung von B oder vorangehender Aktivitäten zurückgesetzt,

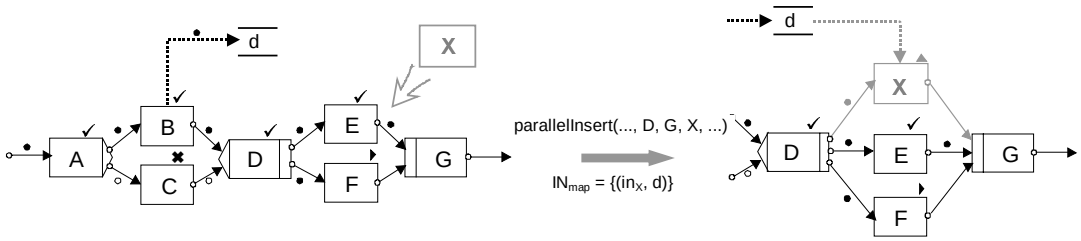


Abbildung 7-10: *Ausnutzen des WF-Status beim Einfügen von Lesezugriffen.*

muß das Einfügen von X allerdings wieder zurückgenommen werden. Ansonsten wäre beim Durchlaufen des unteren Zweiges der Lesezugriff von X auf d nicht mehr versorgt.

Im Zusammenhang mit der Abbildung von Parametern auf Datenelemente ergibt sich eine Reihe weiterer interessanter Fragestellungen, auf die hier aus Platzgründen nur kurz eingegangen werden kann. Auf Grundlage von Satz 7-2 kann die Menge der Datenelemente, auf die Aktivitätenparameter potentiell abbildbar sind, eingegrenzt werden. Im allgemeinen reicht dies allerdings nicht aus, um Anwender bei der Anknüpfung von Aktivitätenparametern an das Datenkontext-Schema hinreichend unterstützen zu können. Für anspruchsvollere Anwendungen sollte es möglich sein, solche Abbildungen semiautomatisch vorzunehmen. Dies erfordert weitergehende Konzepte, die auch Semantikaspekte bei der Beschreibung und Verknüpfung von Parametern und Datenelementen mit einbeziehen. Im einfachsten Fall könnten z.B. Parameterbezeichner einem (domänenabhängigen) kontrollierten Vokabular entnommen werden. Dadurch ließe sich der beschriebene Abbildungsprozeß aus Sicht des Anwenders bereits wesentlich vereinfachen bzw. besser unterstützen. Ein weitergehender Ansatz, der auch von ADEPT verfolgt wird, ist die Assoziation von Parametern mit *semantischen Konzepten*. Dadurch können zwischen Parametern und Datenelementen Beziehungen wie „semantisch äquivalent“, „semantisch ähnlich“ oder „semantisch heterogen“ und zugehörige Ähnlichkeitsmaße abgeleitet werden. Dem entsprechend kann die Menge an Datenelementen, auf die Parameter der eingefügten Aktivität abgebildet werden dürfen, weiter eingegrenzt werden. Für die Beschreibung und Verwaltung semantischer Konzepte und ihrer Beziehungen (z.B. Generalisierung, Spezialisierung, Aggregation, usw.) bieten sich Repräsentationsformalismen wie semantische Netze oder Frames an. Wir verzichten an dieser Stelle auf die Darstellung weiterer Details und verweisen auf [Blas96].

7.3.2 Einfügen vorgefertigter Kontrollblöcke

Auf Basis der bisher vorgestellten Insert-Operationen lassen sich elementare und komplexe Aktivitäten auf Vorlage oder Instanzebene einfügen. Um Vollständigkeit bzgl. der angebotenen Menge an Basiseinfügeoperationen zu erzielen, d.h. um alle möglichen Erweiterungen eines gegebenen WF-Graphen realisieren zu können, sind noch zwei spezielle Einfügeoperationen erforderlich: *InsertLoopBlock* und *InsertBranchingBlock*. Sie werden zur Bildung neuer Schleifen- und Verzweigungsstrukturen benötigt. Um die Korrektheit des WF-Graphen bzw. WF-Ausführungsgraphen zu bewahren, dürfen solche Strukturen – im Gegensatz zu normalen Aktivitätenknoten – immer nur zusammenhängend in den Graphen eingefügt bzw. wieder entfernt werden. Ansonsten ergeben sich keine besonderen Problemstellungen. Wie im Fall einzelner Aktivitätenknoten kann die Einfügung wieder zwischen zwei direkt aufeinanderfolgenden Knoten, parallel zu einem Kontrollblock oder als neuer Teilzweig einer Verzweigung erfolgen. Auch die auf Instanzebene zu berücksichtigenden Statusabhängigkeiten (vgl. Abschnitt 7.3.1.3) gelten unverändert. Wir stellen die beiden Operationen kurz informell vor:

- **[serial | parallel | branch] InsertLoopBlock:** Fügt in das WF-Schema an der angegebenen Position einen Schleifenkontrollblock ein. Dies erfolgt durch Hinzunahme eines Schleifenanfangs- und Schleifenendknotens (mit Knotentypen STARTLOOP bzw. ENDLOOP) sowie einer sie verbindenden Rücksprungkante (des Typs LOOP_E). Dem Schleifenendknoten wird dabei eine spezielle Vorlage zugeordnet, die die Bedingung für die Termination der Schleife beschreibt[3]. Dies erfolgt durch Festlegung eines Prädikats auf den Eingabeparametern der Vorlage. Sie können an bereits vorhandene oder neue Datenelemente geknüpft werden. In letzterem Fall sind wieder be-

[3] Dem Schleifenanfangsknoten kann optional eine Vorlage zugeordnet werden. Sie beschreibt die Aktionen, die bei seiner erstmaligen Aktivierung ausgeführt werden soll (z.B. zur Initialisierung eines Schleifenzählers).

gleitende Änderungen (z.B. Einfügen neuer Aktivitäten im Schleifenkörper) durchzuführen. Das Ergebnis der Bedingungsauswertung wird über einen Bool'schen Ausgabeparameter zurückgeliefert, der an ein neu generiertes Datenelement *LOOP* geknüpft wird. Der Schleifenkörper selbst ist zunächst „leer" und kann durch die Anwendung weiterer Insert-Operationen sukzessive gebildet werden. Üblicherweise erfolgt dies innerhalb derselben Änderungstransaktion, in der auch die Schleifenstruktur als solche angelegt wird (vgl. Beispiel 7-9).

- **[serial | parallel | branch] InsertBranchingBlock**: Fügt in das WF-Schema an der angegebenen Position eine neuen Verzweigungskontrollblock mit einem „leeren" Zweig ein: Dies erfolgt durch die Hinzunahme eines Split- und Join-Knotens sowie einer sie verbindenden Kontrollkante (des Typs `CONTROL_E`). Für die beiden Knoten muß die genaue Ausgangs- bzw. Eingangssemantik festgelegt werden, wobei die Kombination OR-Split /AND-Join ausgeschlossen ist (vgl. Regel KF-3). Darüber hinaus kann jeweils optional eine Aktivitätenvorlage (z.B. zur Festlegung einer Verzweigungsbedingung) zugeordnet werden, deren Parameter dann an das Schema des WF-Datenkontexts anzuknüpfen sind. Beim Einfügen einer bedingten Verzweigung muß zusätzlich ein Datenelement als Entscheidungsparameter vereinbart und für den eingefügten leeren Zweig ein entsprechender Auswahlwert festgelegt werden. Dieser Zweig wird in der Folge als Default-Zweig (vgl. Kapitel 3) behandelt. Seine Definition bzw. die Festlegung anderer Zweige kann durch Anwendung der beschriebenen Insert-Operationen erfolgen.

Beispiel 7-9 (Einfügen von Schleifen)

Ein Beispiel für das Einfügen einer neuer Schleife in einen WF-Graphen zeigt Abbildung 7-11. Hier wird innerhalb derselben Änderungstransaktion zunächst ein neuer Schleifenkontrollblock eingefügt und anschließend die Aktivität *X* dem (leeren) Schleifenkörper hinzugefügt. Diese Aktivität schreibt das Datenelement, das vom Schleifenendknoten L_E zur Auswertung der Schleifenbedingung benötigt wird.

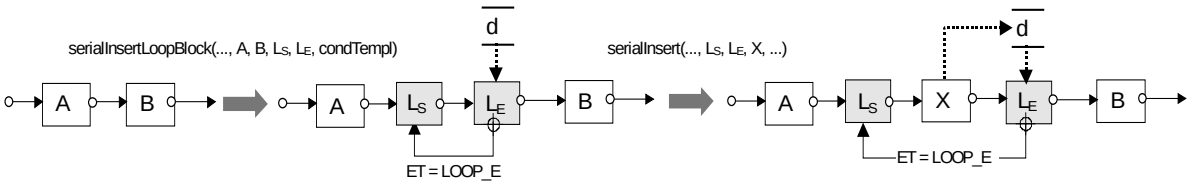


Abbildung 7-11: *Einfügen neuer Schleifenstrukturen*

Operationen für das Einfügen von Kontrollblöcken werden üblicherweise nicht ad hoc auf einzelne WF-Instanzen angewendet, sondern dienen vielmehr der Erstellung und Änderung von WF-Vorlagen. In diesem Kontext müssen sie dann ggf. auch auf laufende Instanzen propagiert werden können. Insgesamt werden die vorgestellten Basiseinfügeoperationen, die sowohl auf Vorlagen- als auch auf Instanzebene anwendbar sind, den eingangs diskutierten Anforderungen gerecht. Ihre Anwendung garantiert, daß auch nach dem Einfügen der neuen Aktivität bzw. des neuen Blocks, die Regeln für die korrekte Strukturierung von WF-Graphen erfüllt sind. Dies schließt den Erhalt der regelmäßigen Blockstrukturierung mit ein. Aus Gründen der besseren Darstellbarkeit haben wir zwischen seriellem, parallelem und bedingtem Einfügen unterschieden. Diese Operationen können bei geeigneter Parametrisierung auch in einer Einfügeoperation zusammengefaßt werden.

7.4 Die Lösch-Operation

Eine ebenso wichtige Operation wie das (dynamische) Einfügen ist das Löschen von Aktivitäten und Kontrollblöcken. Damit verbunden ist nicht nur ihr Entfernen aus dem KF-Graphen, sondern auch der Wegfall sämtlicher von ihnen ausgehender Lese- und Schreibzugriffe auf den WF-Datenkontext. Wichtigstes Ziel ist wieder der Erhalt der Korrektheit und Konsistenz des WF und seiner Daten. Wir behandeln zunächst das Löschen einzelner Aktivitätenknoten. Anschließend gehen wir auf Erweiterungen ein, die im Hinblick auf das Entfernen von Kontrollblöcken (z.B. Schleifen- oder Verzweigungsstrukturen) notwendig werden.

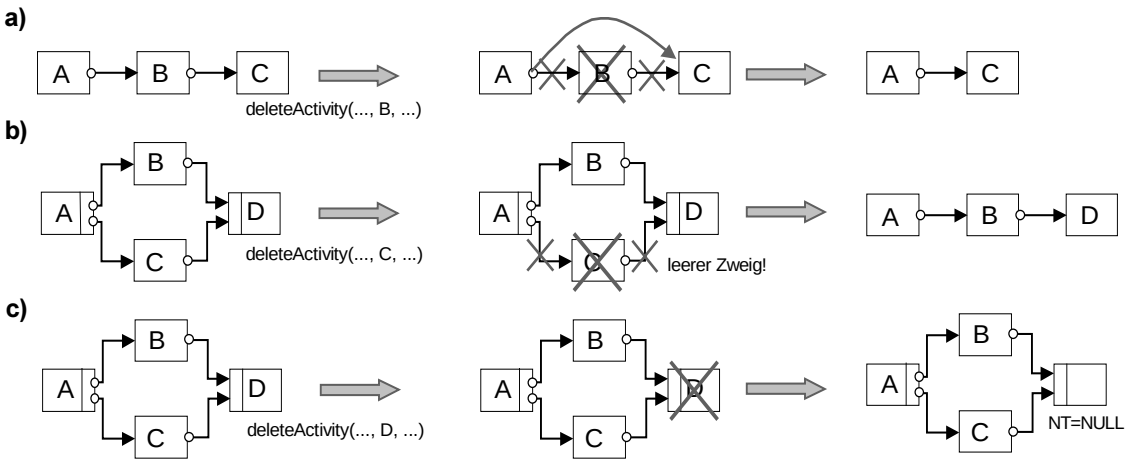


Abbildung 7-12: *Löschen von Aktivitäten aus KF-Graphen (deleteActivity)*

7.4.1 Löschen von Aktivitäten

Das Löschen von Aktivitäten kann aus unterschiedlichen Gründen erforderlich werden: Durch die Optimierung und Neugestaltung betrieblicher Abläufe [NiPi95, HaSt95] können bisherige Arbeitsschritte in der Folge entfallen. Während der Ausführung einer bestimmten WF-Instanz kann es situationsbedingt notwendig werden, die Bearbeitung einzelner oder mehrerer Aktivitäten zu überspringen (vgl. Kapitel 6). Auch das Überspringen (ohne Nachholen) entspricht logisch dem Entfernen der Aktivität (inkl. ihrer Datenzugriffe) aus dem Ausführungsgraphen, da durch das Auslassen der Bearbeitung Daten für nachfolgende Schritte fehlen können. Schließlich kann das Löschen einer Aktivität X auch die Folge vorausgehender Löschoperationen sein, insbesondere wenn dadurch obligate, nicht-nachforderbare Eingaben von X unversorgt bleiben.

7.4.1.1 Löschen von Aktivitätenknoten in KF-Graphen (Schemaebene)

Die von ADEPT$_{flex}$ bereitgestellte Basisoperation *deleteActivity* ermöglicht es, einzelne Aktivitäten aus WF-Graphen bzw. WF-Ausführungsgraphen zu löschen. Sie kann ausschließlich auf normale Aktivitätenknoten (NT = `ACTIVITY`) und Nullaktivitäten (NT = `NULL`) angewendet werden, nicht dagegen auf die Start- bzw. Endaktivität des WF und auch nicht auf einzelne Schleifenanfangs- oder Schleifenendknoten. Erstere sind integraler Bestandteil jedes KF-Graphen und dürfen unter keinen Umständen gelöscht werden (vgl. Regel KF-1). Letztere können nur zusammen mit der gesamten Schleife, nicht aber einzeln aus dem KF-Graphen entfernt werden (vgl. Regel KF-7). Aus diesem Grund unterstützt ADEPT$_{flex}$ eine weitere Basisoperation (→ *deleteBlock*) für das Entfernen von Kontrollblöcken. Sie wird im nachfolgenden Abschnitt vorgestellt. Die genannten Vorbedingungen für die Verwendung von *deleteActivity* können in ADEPT noch um benutzerdefinierte Einschränkungen ergänzt werden. Sie legen z.B. fest, ob ein Aktivitätenknoten während der WF-Ausführung gelöscht bzw. seine Ausführung ausgelassen werden darf oder nicht [Kubi98]. In der Folge gehen wir davon aus, daß entsprechende Überprüfungen vor Anwendung der *deleteActivity*-Operation stattgefunden haben.

Im folgenden bezeichne $WS = (CFS, DFS)$ einen WF-Graphen und $X \in N$ die daraus zu löschende Aktivität mit $NT^X \in$ {`ACTIVITY`, `NULL`}. Entspricht der Knoten X keiner Split- bzw. Join-Aktivität und besitzt er keine einmündenden oder ausgehenden Spezialkanten, so kann X bei Anwendung der *deleteActivity*-Operation physisch aus dem KF-Graphen entfernt werden. Dazu werden der Knoten und seine ein- und ausgehende Kontrollkante gelöscht und durch eine neue Kontrollkante ersetzt, die den bisherigen Vorgängerknoten von X mit seinem bisherigen direkten Nachfolgerknoten verknüpft (vgl. Abbildung 7-12 a). Falls X der einzige Knoten eines Teilzweiges einer Parallelverzweigung ist, kann das Einfügen dieser Kontrollkante entfallen. Der Zweig wird vollständig aus dem KF-Graphen entfernt. Besitzt die Verzweigung anschließend nur noch einen Teilzweig, so kann zur Vereinfachung des Graphen zusätzlich die Aus- und Eingangssemantik ihres Split- bzw. Join-Knotens in `ONE_Of_ONE` abgeändert werden (vgl. Abbildung 7-12 b). Diese Vorgehensweise ist auf Split- und Join-Knoten oder Knoten mit einmündenden oder ausgehenden Spezialkanten nicht anwendbar. Soll

ein solcher Aktivitätenknoten gelöscht werden, wird er nicht wirklich aus dem KF-Graphen entfernt, sondern durch eine Nullaktivität (vgl. Abschnitt 7.2.3) ersetzt. Ein Beispiel zeigt Abbildung 7-12 c). Die Substitution der zu löschenden Aktivität durch eine Nullaktivität kann auf WF-Instanzebene auch in anderem Kontext sinnvoll sein, etwa wenn ein Knoten nur temporär gelöscht und später wieder eingefügt werden soll. Ein Beispiel ist das einmalige Überspringen einer Aktivitätenbearbeitung in einer bestimmten Schleifeniteration (vgl. Kapitel 8).

Wir können nun die *deleteActivity*-Operation und ihre Wirkung formulieren:

```
deleteActivity (CFS, DFS, X, replaceByNull)
input
  CFS, DFS:        WF-Graph aus dem die Aktivität gelöscht werden soll
  X:               Bezeichner für die zu löschende Aktivität
  replaceByNull:   Aktivität soll in jedem Fall durch Nullaktivität ersetzt werden
begin
  // Entfernen aller Lese- und Schreibzugriffe der Aktivität und ihrer vor-/nachgeschalteter
  // Aktionen (ausgenommen: nachgeschalteter Lesezugriff auf Entscheidungsparameter
  // bei OR-Split-Knoten)
  dataEdges:= ∅
  forall (d, n, par, mode) ∈ DFS with (n = X or n ∈ PreS^X or (n ∈ PostS^X and d ≠ DP^X) do
      dataEdges:= dataEdges ∪ {(d, n, par, mode)}
  done
  DeleteDataEdges(CFS, DFS, dataEdges)
  // Darf bzw. soll der Knoten physisch aus dem KF-Graphen gelöscht werden oder ist eine
  // Nullaktivität als Ersatzknoten festzulegen?
  if  (replaceByNull = True or (V_in^X ≠ ONE_Of_ONE or V_out^X ≠ ONE_Of_ONE) or
      (∃ e ∈ E with X ∈ {Id_dest^e, Id_source^e} and ET^e ≠ CONTROL_E)) then
      // Ersetze den zu löschenden Knoten durch Nullaktivität: Abänderung des Knotentyps,
      // Dereferenzierung der bisher zugrundeliegenden Vorlage, Deaktivierung vor-/nachge-
      // schalteter Nachforderungsdienste – Alle anderen Knotenattribute (z.B. V_in^X und V_out^X)
      // bleiben unverändert. Dasselbe gilt für ein- / ausgehende Kanten des Knotens!
      SetNodeAttr(CFS, X, "NT", NULL)
      SetNodeTemplate(CFS, X, NULL)
      DeleteParamProvisionService(CFS, DFS, X, Params^X)
  else
      // Physisches Entfernen des Aktivitätenknotens aus dem KF-Graphen: X besitzt je genau
      // eine einmündende / ausgehende Kontrollkante; predX / succX bezeichne den zugehörigen
      // direkten Vorgänger / Nachfolger von X
      e_in:=  In den Knoten X einmündende Kontrollkante mit e_in = (predX ,X, CONTROL_E)
      e_out:= Vom Knoten X ausgehende Kontrollkante mit e_out = (X, succX, CONTROL_E)
      // Falls zu löschende Aktivität letzter verbleibender Knoten in einem Teilzweig einer Parallel-
      // verzweigung ist, die noch weitere Teilzweige besitzt: Knoten und seine ein-/ausgehende
      // Kontrollkante ersatzlos entfernen; andernfalls durch neue Kontrollkante ersetzen
      branchC:= Cardinality({e ∈ E | ET^e = CONTROL_E ∧ Id_source^e = predX ∧ e ≠ e_in})
      if (V_out^predX = ALL_Of_ALL and V_in^succX ≠ ONE_Of_ONE and branchC ≥ 1) then
          // Falls nur noch ein Teilzweig verbleibt, Eingangs- und Ausgangssemantik von
          // Split- und Join-Knoten anpassen
          if branchC = 1 then
              SetNodeAttr(CFS, predX, "V_out", ONE_Of_ONE)
              SetNodeAttr(CFS, succX, "V_in", ONE_Of_ONE)
          endif
      else
          e_new:= (predX, succX, CONTROL_E)
          AddEdges(CFS, {e_new});
          SetEdgeAttr(CFS, e_new, "EC", GetEdgeAttr(CFS, e_in, "EC"))
```

```
            (Bei Löschen in Ausführungsgraphen CFS_instance zusätzlich:
            SetEdgeState(CFS_instance, e_new, GetEdgeState(CFS_instance, e_in));
        endif
        DeleteEdges(CFS, {e_in, e_out})
        DeleteNodes(CFS, {X})
    endif
end // deleteActivity
```

7.4.1.2 Löschen von Aktivitätenknoten auf Instanzebene

Die Anwendbarkeit von *deleteActivity* wird auf WF-Instanzebene zusätzlich durch den aktuellen WF-Status eingeschränkt. Bisher noch nicht gestartete Knoten mit Markierung *NS* = `NOT_ACTIVATED` oder *NS* = `ACTIVATED` dürfen gelöscht werden. In letzterem Fall werden vor dem eigentlichen Löschen entsprechende Einträge aus Arbeitslisten entfernt. Das Löschen bereits beendeter oder nicht mehr ausführbarer Aktivitätenknoten ist dagegen nicht erlaubt. Ebensowenig können laufende Schritte (*NS* = `RUNNING`) aus dem Ausführungsgraphen entfernt werden, es sei denn, sie werden zuvor abgebrochen und zurückgesetzt (vgl. Kapitel 6). Nach Anwendung einer Löschoperation können die notwendigen Anpassungen von Knoten- und Kantenmarkierungen, wie schon im Fall der Einfügeoperation, auf Grundlage der in Kapitel 4 vorgestellten Markierungs- und Ausführungsregeln erfolgen. Wichtig ist an dieser Stelle der Hinweis, daß auf Instanzebene Knoten nie physisch aus dem WF-Graphen gelöscht, sondern immer durch eine Nullaktivität ersetzt werden (siehe Kapitel 8).

7.4.1.3 Begleitende Anpassungen des Datenflusses

Durch das Löschen von Aktivitäten und ihrer Datenzugriffe aus einem WF-Graphen *WS* = (*CFS*, *DFS*) können nachfolgenden Schritten obligate Eingabedaten fehlen. Würden keine begleitenden Anpassungen des zugrundeliegenden DF-Modells vorgenommen, so könnte es in der Folge auf WF-Instanzebene zu schwerwiegenden Programmfehlern kommen, wenn Aktivitäten bzw. Aktivitätenprogramme mit unversorgten Parameterdaten aufgerufen werden (vgl. Regel DF-1).

Für den Erhalt der Korrektheit des Datenflusses muß zunächst festgestellt werden, welche konkreten Auswirkungen das Löschen einer Aktivität *X* auf die Parameterversorgung nachfolgender Schritte hat. Zu diesem Zweck ist für jeden obligaten Schreibzugriff von *X* (inkl. Schreibzugriffen seiner vor- und nachgeschalteten Hilfsdienste) zu prüfen, ob alle obligaten Lesezugriffe nachfolgender Schritte auf dasselbe Datenelement noch versorgt sind. (Die Überprüfung ist auch in Bezug auf die Versorgung des Entscheidungsparameters bei OR-Split-Knoten erforderlich.) Die Grundlage hierfür bildet das in Kapitel 5 vorgestellte *WriterExists*-Verfahren.

DF-Analysen beim Löschen eines Aktivitätenknotens X:

Bezeichne $D_{written}^{X} := \{d \in D \mid ActivityWrites^{DFS}(X, d, \texttt{MANDATORY})\}$ die Menge aller Datenelemente, die von *X* (inkl. seiner vor- und nachgeschalteten Hilfsdienste) obligat geschrieben werden. Dann müssen beim Löschen des Knotens *X* folgende DF-Analysen durchgeführt werden:

Für jeden obligaten Lesezugriff, der von einem Nachfolgerknoten von *X* auf ein Datenelement der Menge $D_{written}$ ausgeführt wird, muß geprüft werden, ob er weiterhin versorgt ist [(CFS^*, DFS^*) bezeichne den WF-Graphen, das bei Anwendung der *deleteActivity*-Operation auf *X* resultiert].

Gilt $\forall\, d \in D_{written}^{X}$, $\forall\, n \in succ^*(X)$ mit $ActivityReads^{DFS}(n, d, \texttt{MANDATORY})$[4]:
$WriterExists(CFS^*, DFS^*, n, d) = True$?

Falls es sich bei *X* um einen OR-Split handelt, muß zusätzlich für den „leeren" Ersatzknoten X_{null} geprüft werden, ob sein Lesezugriff auf das Datenelement $DP(X_{null})$ – es wird benötigt um die Verzweigungsentscheidung zu treffen – versorgt ist Gilt $WriterExists(CFS^*, DFS^*, X_{null}, d) = True$? (Anm. Beim Löschen einzelner Aktivitäten bleibt die Ausgangssemantik für den leeren Ersatzknoten erhalten.)

[4] vgl. Definition 5-2: Das Prädikat $ActivityReads^{DFS}(n, d, \texttt{MANDATORY})$ evaluiert genau dann zu *True*, wenn der Knoten *n* oder einer seiner vor-/nachgeschalteten Aktionen das Datenelement *d* zwingend liest.

Die DF-Analysen, die beim Löschen einer Aktivität und ihrer Schreibzugriffe notwendig werden, lassen sich in ADEPT somit gezielt eingrenzen. Dadurch wird eine effiziente Anwendung von Löschoperationen überhaupt erst möglich. Im Gegensatz dazu etwa müßten bei Petri-Netz- oder Statechart-basierten WF-Beschreibungen [Gruh91, LöHa94, Ober96, WoWe97] komplexe Erreichbarkeitsanalysen (siehe z.B. [Baum96]) durchgeführt werden, wenn vergleichbare Fragestellungen untersucht werden sollen. Aufgrund der mangelhaften Strukturierbarkeit entsprechender Modelle wäre hier eine Eingrenzung der Analysen nicht oder nur sehr eingeschränkt möglich. Legt man in ADEPT zusätzlich das in Abschnitt 5.4.4 diskutierte Konzept logischer Blockparameter zugrunde, kann der Aufwand für die durchzuführenden DF-Analysen weiter reduziert werden. Angenommen, die zu löschende Aktivität X gehört dem Teilzweig eines (abgeschlossenen) Verzweigungskontrollblocks mit OR-Join an. Wird dann ein von X obligat geschriebenes Datenelement d^* nicht in jedem anderen Teilzweig dieser Verzweigung ebenfalls obligat geschrieben, so steht bereits vor dem Löschen des Knotens X fest, daß d^* durch die *logischen* Ausgaben des Verzweigungsblocks nicht sicher versorgt wird. Dem entsprechend müssen beim Entfernen von X nur die obligaten Lesezugriffe auf d^*, die innerhalb des betreffenden Teilzweiges (inkl. Lesezugriffen des Join-Knotens) stattfinden, überprüft werden (dies folgt unmittelbar aus der Korrektheit des Ausgangsgraphen).

Als Ergebnis der DF-Analyse erhalten wir eine Menge obligater Lesezugriffe $DFS_{missing} \subseteq DFS^*$ die nach dem Entfernen der Aktivität X nicht mehr versorgt sind. Nur wenn $DFS_{missing} = \emptyset$ gilt, sind keine weiteren Maßnahmen erforderlich. Dieser Fall tritt auf, wenn X keine obligaten Ausgaben macht oder wenn alle nachfolgenden Lesezugriffe auf Elemente der Menge $D_{written}$ optional sind bzw. anderweitig versorgt werden können. Wir zeigen nun, wie mit Lesezugriffen der Menge $DFS_{missing}$ zu verfahren ist, um nach dem Löschen des Knotens X wieder ein korrektes DF-Modell zu erhalten. Hierzu benötigen wir die folgende Definition:

Definition 7-1 (Datenabhängigkeit)

Bezeichne $WS = (CFS, DFS)$ einen korrekten WF-Graphen (insbesondere gelte die Regel DF-1) und $WS^* = (CFS^*, DFS^*)$ bezeichne den aus der Anwendung der Operation *deleteActivity*(CFS, DFS, X) hervorgehende WF-Graphen.

Ein Knoten $n_{read} \in N$ mit $n_{read} \in succ^*(X)$ ist *datenabhängig* von der Aktivität $X :\Leftrightarrow \exists\, d \in D_{written}^{X}$ mit $ActivityReads^{DFS}(n_{read}, d, \texttt{MANDATORY}) \wedge WriterExists(CFS^*, DFS^*, n_{read}\ d) = False$, d.h. nach dem Löschen von X sind die obligaten Lesezugriffe von n_{read} auf den Datenkontext nicht mehr vollständig versorgt. Im besonderen heißt n_{read} datenabhängig von X bzgl. $d^* \in D_{written}^{X} :\Leftrightarrow$
$ActivityReads^{DFS}(n_{read}, d^*, \texttt{MANDATORY}) \wedge WriterExists(CFS^*, DFS^*, n_{read}, d^*) = False$.

Wird eine Aktivität X aus dem WF-Graphen gelöscht, so tritt eine Verletzung der Korrektheit des Datenflußes genau dann auf, wenn es von X datenabhängige Aktivitäten gibt. Für diesen Fall muß die Löschoperation entweder abgebrochen werden oder es sind begleitende Maßnahmen zur Wiederherstellung der Korrektheit des Datenflusses durchzuführen. Die Entwicklungs- und Laufzeitkomponenten von ADEPT$_{flex}$ bieten u.a. die folgenden Strategien zur Behandlung fehlender Daten an (X bezeichne den zu löschenden Schritt):

1. **Löschen datenabhängiger Schritte**: Von X datenabhängige Aktivitäten werden ebenfalls aus dem WF-Graphen entfernt.
2. **Einfügen spezieller Versorgeraktivitäten**: Zwischen dem gelöschten Knoten X (bzw. seinen direkten Vorgängerknoten) und den von ihm datenabhängigen Schritten werden ein oder mehrere Ersatzschritte zur Versorgung der fehlenden Daten in den WF-Graphen eingefügt.
3. **Aktivierung von Nachforderungsdiensten**: Aktivitäten des WF-Graphen werden Nachforderungsdienste vor- oder nachgeschaltet, welche die interaktive Festlegung der fehlenden Daten erlauben. ADEPT bietet hierfür Dienste zur dynamischen Generierung von Bildschirmmasken an.
4. **Direkte Abänderung des DF-Modells**: Der Anwender ändert das DF-Modell direkt mit den dafür bereitgestellten Basisoperationen (siehe Abschnitt 7.5.4) ab, um wieder zu einem korrekten Datenfluß zu gelangen.

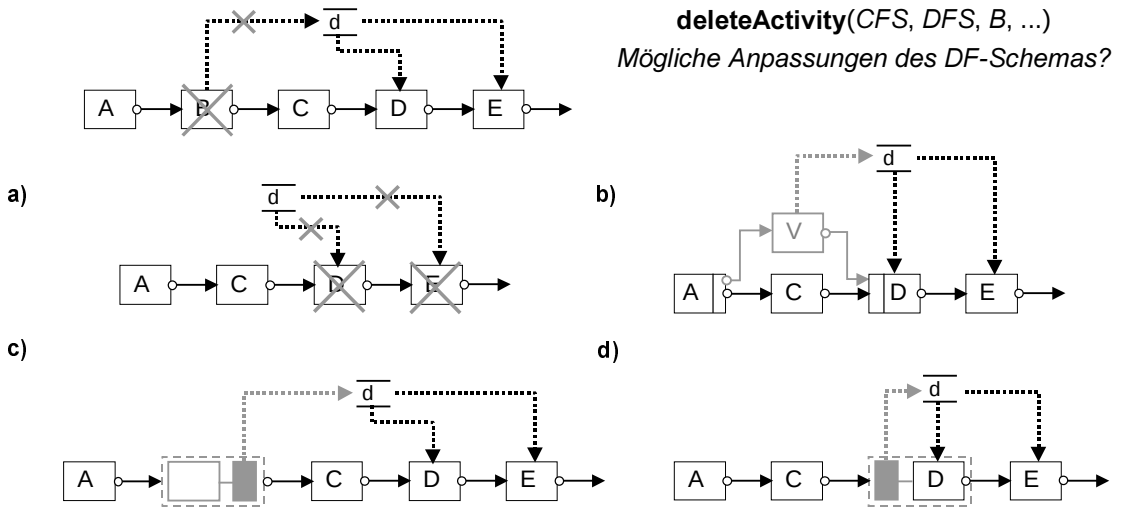


Abbildung 7-13: *Anpassung des DF-Schemas bei Löschen einer Aktivität mit obligaten Schreibkanten* [*a) Löschen datenabhängiger Aktivitäten; b) Einfügen einer Versorgeraktivität V; c) Sofortiges Nachfordern der fehlenden Daten durch Aktivierung eines nachgeschalteten Dienstes; d) Nachfordern der Daten durch Aktivierung eines vorgeschalteten Nachforderungsdienstes.*]

Diese Strategien können auch in Kombination miteinander angewendet werden. Ebenso ist es möglich, für die Festlegung der fehlenden Ausgabeparameter von *X* mehrere Versorgeraktivitäten und/oder Nachforderungsdienste in das Schema einzusetzen. Unabhängig von der gewählten Behandlungsstrategie müssen alle Änderungen (einschl. des Löschens von *X*) innerhalb derselben Änderungstransaktion abgewickelt werden. Aus Gründen der besseren Darstellbarkeit behandeln wir die genannten Fälle und ihre Umsetzung in ADEPT getrennt voneinander:

1. Löschen datenabhängiger Schritte

Die Ermittlung datenabhängiger Schritte erfolgt auf Basis von Definition 7-1. Sie werden bei der vorliegenden Strategie ebenfalls gelöscht (siehe Abbildung 7-13 a), wodurch sich die Notwendigkeit weiterer DF-Analysen und ggf. auch Anpassungen des WF-Graphen ergibt. Infolge transitiver Datenabhängigkeiten zwischen Schritten ist auch das *kaskadierende Löschen* von Aktivitätenknoten möglich. Ist davon ein Knoten betroffen, der nicht gelöscht werden darf (z.B. der Endknoten des WF), muß die Änderungstransaktion entweder abgebrochen und zurückgesetzt werden oder aber es sind alternative Strategien (2. oder 3.) anzuwenden.

2. Einfügen einer speziellen Versorgeraktivität

Auch bei Anwendung dieser Strategie werden zunächst die datenabhängigen Schritte des zu löschenden Knotens *X* ermittelt. Im Gegensatz zu 1. werden sie aber nicht aus dem WF-Graphen entfernt. Vielmehr wird ihre Parameterversorgung dadurch sichergestellt, daß für den gelöschten Knoten ein oder mehrere Ersatzschritte in den WF-Graphen eingesetzt werden. Ihre Schreibzugriffe ersetzen dann die fehlenden Ausgaben von *X* (siehe Abbildung 7-13 b). Ein solcher Versorgungsschritt wird zwischen den direkten Vorgängern des zu löschenden Knotens (oder seinem leeren Ersatzknoten, falls ein solcher vorliegt) und einem oder mehrerer seiner datenabhängigen Schritte eingefügt. Die hierfür benötigte Operation (→ Einfügen einer Aktivität zwischen Knotenmengen) wird in Abschnitt 7.6.1 vorgestellt. Ein so eingefügter Versorgungsschritt erhält Schreibzugriff auf Datenelemente der Menge $D_{written}^{X}$ (siehe oben). Durch diese Vorgehensweise erreichen wir, daß die Festlegung der fehlenden Daten jeweils vor der Aktivierung der infolge des Löschens unterversorgten Schritte abgeschlossen ist (Regel DF-1). Werden mehrere Versorgungsaktivitäten in den WF-Graphen eingefügt muß zusätzlich beachtet werden, daß es nicht zu parallelen Schreibzugriffen auf Datenelementen der Menge $D_{written}^{X}$ kommt (Regel DF-2). Auch das mehrfache, hintereinander ausgeführte redundante Schreiben (Regel DF-3) sollte ausgeschlossen werden.

Die Festlegung der Parameterschnittstelle bzw. Vorlage eines neu einzufügenden Versorgungsschrittes kann unterschiedlich erfolgen, je nach Art der fehlenden Daten. Hilfreich sind in diesem Zusammenhang die zu Aktivitätenparametern vorliegenden Beschreibungsdaten: Sollen für den gelöschten Knoten *speziell nachversorgbare* Ausgabeparameter (z.B. ein Textdokument) festgelegt werden, so kann

die dazu erforderliche Aktivitätenvorlage (z.B. mit einem Texteditor als zugehöriger Anwendungskomponente) direkt aus der Definition der Parameters entnommen werden (vgl. Kapitel 3). Liegen *nachversorgbare* Ausgabeparameter (siehe auch 3.) vor, besteht die Möglichkeit, die Vorlage für den Versorgungsschritt auf Grundlage der Parameterbeschreibungen des gelöschten Knotens automatisch zu generieren. Seine Ausführung besteht dann im wesentlichen aus der dynamischen Erzeugung einer Bildschirmmaske, mit der sich die fehlenden Daten festlegen lassen. Im allgemeinen können Anwender auch beliebige Vorlagen aus dem Aktivitätenrepository verwenden.

3. Installation von Nachforderungsdiensten

Bei dieser Behandlungsform können fehlende Daten entweder direkt nach der gelöschten Aktivität oder vor den Aktivitäten, die diese Daten als Eingaben benötigen, vom Benutzer nachgefordert werden. Im Unterschied zu 2. wird keine neue Aktivität in den Ablaufgraphen eingefügt, sondern es werden existierenden Knoten des WF-Graphen interaktive Versorgungsdienste (→ automatisch generierte Bildschirmmasken) vor- bzw. nachgeschaltet. Notwendige Voraussetzung ist die Nachforderbarkeit der betreffenden Parameter (vgl. Kapitel 3). Wichtig ist in diesem Zusammenhang, daß ein Knoten mit vor- oder nachgeschaltetem Nachforderungsdienst auch nachfolgende Lesezugriffe auf die betreffenden Datenelemente versorgen kann.

Sollen die fehlenden Daten sofort beim Löschen der Aktivität nachgefordert werden, so wird die bisherige Vorlage des Knotens durch eine neue Vorlage mit identischer Parameterschnittstelle, aber ohne assoziierte Anwendungs- bzw. Sub-WF-Komponente ersetzt (diese Möglichkeit haben wir bei der Darstellung der *deleteActivity*-Operation ausgeklammert). Anschließend wird für die Festlegung der Aktivitätenausgaben ein Nachforderungsdienst aktiviert (siehe Abbildung 7-13 c). Das resultierende DF-Modell ist wieder korrekt, da zwischen den ursprünglichen Schreibzugriffen und den nun eingefügten Schreibzugriffen des Nachforderungsdienstes keine Lesezugriffe stattgefunden haben bzw. stattfinden werden. Die zweitgenannte Möglichkeit – die Nachforderung der fehlenden Daten direkt vor den Aktivitäten, die sie als Eingaben benötigen – gestaltet sich aufwendiger. Hier müssen ggf. für mehrere datenabhängige Schritte Nachversorgungsdienste aktiviert werden, ohne daß es dadurch zu mehrfachen, hintereinander ausgeführten Schreibzugriffen auf dieselben Datenelemente kommt (Regel DF-3). Ein Beispiel zeigt Abbildung 7-13 d): Nach dem Löschen der Aktivität `B` wird dem datenabhängigen Schritt `D` ein Dienst zur Nachversorgung des fehlenden Datenwertes vorgeschaltet. Dieser Dienst wird beim Start von `D` ausgeführt. Da die Aktivität `E` Nachfolger des Knotens `D` ist, kann sie den geschriebenen Datenwert ebenfalls lesen, so daß dem Knoten `E` selbst kein Nachforderungsdienst mehr vorgeschaltet werden muß. Neben der Vermeidung redundanter Nachforderungen muß gewährleistet werden, daß es durch das Hinzufügen von Nachforderungsdiensten und ihrer Schreibkanten nicht zu parallelen Schreibzugriffen auf den Datenkontext kommt (Regel DF-2).

Wir skizzieren, wie sich die Aktivierung vorgeschalteter Nachforderungsdienste in ADEPT darstellt. Dazu bezeichne (CFS^*, DFS^*) den WF-Graphen, der aus dem Ursprungsgraphen (CFS, DFS) nach Anwendung der Operation *deleteActivity*(CFS, DFS, X, ...) hervorgegangen ist. Aufgabe ist es dann festzustellen, für welche Aktivitäten welche Nachforderungsdienste aktiviert werden müssen, um anschließend wieder ein korrektes DF-Modell zu erhalten. Zu diesem Zweck ermitteln wir zunächst für jedes von X obligat geschriebene Datenelement $d \in D_{written}^{X}$ die Menge der datenabhängigen Aktivitäten (vgl. Definition 7-1):

$$N_{dep}^{d} := \{n \in N \mid n \text{ ist datenabhängig von } X \text{ bzgl. } d\}$$

Eine naive und nicht immer korrekte Lösung wäre es, für jeden Knoten aus N_{dep}^{d} einen Nachforderungsdienst zur Versorgung des Lesezugriffs auf d vorzuschalten. Diese Vorgehensweise kann – wie oben diskutiert – zu überflüssigen oder parallelen Schreibzugriffen auf das Datenelement führen. Um solche Fälle zu vermeiden, muß deshalb für jedes $d \in D_{written}^{X}$ die Menge der Knoten, für die effektiv Datenwerte nachgefordert werden sollen, bestimmt werden. Dazu gehen wir wie folgt vor:

1. Falls $N_{dep}^{d} = \emptyset$ gilt, gibt es bzgl. d keine datenabhängigen Schritte und wir sind fertig. Ansonsten gehe zu Schritt 2.
2. Sortiere die Knoten der Menge N_{dep}^{d} topologisch, d.h. ordne ihre Elemente in einer Liste $nodeList = n_1 \rightarrow ... \rightarrow n_k$ an, so daß gilt: $n_1 \prec ... \prec n_k$; $n_l \in N_{dep}^{d}$ für $l = 1...k$ ($n \prec m :\Leftrightarrow m \notin pred^*(n)$)

Die topologische Sortierung ist möglich, da im KF-Graphen über Kontroll-/Sync-Kanten keine Zyklen vorliegen können (Regel KF-4*).

3. Bestimme die Menge $N_{add}^{d} \subseteq N_{dep}^{d}$ der Knoten, für die ein Nachforderungsdienst aktiviert werden muß, um alle Lesezugriffe auf *d* sicher versorgen zu können:

```
N_add^d := ∅; DFS** := DFS*          // Kopie des aktuellen DF-Schemas anlegen
repeat
    Entferne das erste Element aus nodeList → Knoten n_p
    // Führt die Aktivierung des Nachforderungsdienstes zu parallelen Schreibzugriffen auf d?
    if ParallelWriterExists(CFS*, DFS**, n_p, d) = True then  // Verletzung von Regel DF-2 !
        Abort (d.h. Vorschalten von Nachforderungsdiensten für Lesezugriffe auf d ist
        nicht möglich)
    endif
    N_add^d := N_add^d ∪ {n_p}; DFS** := DFS** ∪ {(d, pre-n_p, par_d, MANDATORY)}
    // Können durch Aktivierung des Vorschaltdienstes auch Lesezugriffe anderer Knoten
    // versorgt werden?
    for n := n_{p+1} to n_k do
        if WriterExists(CFS*, DFS**, n, d) = True then
            Entferne den Knoten n aus nodeList
        endif
    done
until nodeList = ∅.
```

Wird Schritt 3 nicht abgebrochen, so legt die Menge $N_{add}^{d} \subseteq N_{dep}^{d}$ anschließend genau die Knoten fest, für die ein Versorgungsdienst aktiviert werden muß, um nach dem Löschen von *X* alle bisherigen Lesezugriffe auf *d* weiterhin gestatten zu können. Wird Schritt 3 abgebrochen – dies ist genau dann der Fall, wenn eine Verletzung von Regel DF-2 eintreten würde – so ist das Vorschalten von Versorgungsdiensten für das gegebene Datenelement nicht möglich. Für diesen Fall ist eine andere Variante zu wählen.

Beispiel 7-10 (Ermittlung zu aktivierender Nachforderungsdienste)

Für das Beispiel aus Abbildung 7-13 erhalten wir für die zu löschende Aktivität *B*: N_{dep}^{d} = {D, E} und N_{add}^{d} = {D}. Das bedeutet, daß der Lesezugriff des Knotens E auf d durch den Schreibzugriff des Nachforderungsdienstes von D versorgt ist.

Die Varianten 1-3 sind in Verbindung mit Ad-hoc-Änderungen einer WF-Instanz besonders wichtig, da sie ohne aufwendige Benutzerinteraktionen realisiert werden können. Für Vorlagenänderungen durch den WF-Modellierer stehen dem Anwender zusätzlich Basisoperationen für das Hinzufügen und Entfernen von Datenkanten und -elementen zur Verfügung (siehe Abschnitt 7.5.4).

7.4.2 Löschen von Kontrollblöcken

Um auch Verzweigungs- und Schleifenstrukturen löschen zu können, bietet ADEPT$_{flex}$ die Basisoperation *deleteBlock* an. Sie erlaubt es, abgeschlossene Kontrollblöcke (inkl. aller von ihren Knoten ausgehenden Lese- und Schreibzugriffe auf den WF-Datenkontext) aus einem WF-Graphen zu entfernen. Ein Kontrollblock wird dabei durch Angabe seines Start- und Endknotens (n_{begin}, n_{end}) spezifiziert. Vom Prinzip her wird ähnlich wie beim Entfernen einzelner Aktivitätenknoten verfahren. Besitzt der Startknoten des Blockes eine Eingangssemantik oder sein Endknoten eine Ausgangssemantik, die ungleich ONE_Of_ONE ist, so wird der Block durch einen leeren Ersatzknoten substituiert. Andernfalls kann er vollständig aus dem Graphen entfernt und durch eine Kontrollkante ersetzt werden. Dabei wird wie im Fall der *deleteActivity*-Operation verfahren (vgl. Abschnitt 7.4.1.1). Auf Instanzebene wird das Löschen eines Kontrollblocks dadurch eingeschränkt, daß der Anfangsknoten n_{begin} noch nicht gestartet worden sein darf, d.h. es muß gelten $NS(n_{begin}) \in$ {ACTIVATED, NOT_ACTIVATED}.

Um das DF-Modell des WF korrekt anzupassen, wird wie folgt vorgegangen: Zunächst wird für den zu löschenden Kontrollblock die Menge $D_{written}^{(n_{begin}, n_{end})}$ derjenigen Datenelemente ermittelt, die von

seinen Knoten während der Blockausführung zwingend geschrieben werden. Ihre Bestimmung erfolgt durch die Anwendung des *WriterExists*-Verfahrens auf den durch den Kontrollblock gebildeten KF-Teilgraphen CFS_{sub} und die mit seinen Knoten assoziierten DF-Kanten DFS_{sub}. Ein Datenelement *d* wird genau dann durch den Block zwingend geschrieben, wenn der Endknoten des Blocks *d* obligat lesen darf bzw. lesen dürfte oder wenn er seinerseits *d* obligat schreibt. Formal:

$$WriterExists(CFS_{sub}, DFS_{sub}, n_{end}, d) = True \text{ oder } ActivityWrites^{DFS_{sub}}(n_{end}, d, \texttt{MANDATORY}).$$

Diese Bedingung wird für alle Datenelemente geprüft, die von Knoten des Blocks geschrieben werden. Am Ende erhalten wir die Menge $D_{written}^{(n_{begin}, n_{end})}$ und können dann wie in Abschnitt 7.4.1.3 beschrieben verfahren. Es werden also alle datenabhängigen Schritte des Blocks ermittelt und Maßnahmen zur Sicherstellung ihrer Parameterversorgung ergriffen.

Soll ein offener Kontrollblock, d.h. ein Block mit einmündenden oder ausgehenden Spezialkanten, gelöscht werden, müssen diese Kanten zuerst mit den dafür bereitgestellten Basisoperationen (vgl. Abschnitt 7.5.2) entfernt werden, um *deleteBlock* anwenden zu können. Falls erforderlich können für den leeren Ersatzknoten des Blocks anschließend neue Spezialkanten eingefügt werden.

Die *deleteActivity*-Operation bildet einen Spezialfall von *deleteBlock*. Da es sich bei hierbei jedoch um eine Operation handelt, die wesentlich häufiger als *deleteBlock* angewendet wird, wir sie als separate Basisoperation realisiert, selbst wenn dies nicht ganz konform zu der eingangs erwähnten Forderung nach Minimalität ist.

7.5 Zusammenfassung weiterer Basisänderungsoperationen

In diesem Abschnitt fassen wir die weiteren von $ADEPT_{flex}$ bereitgestellten Basisänderungsoperationen zusammen. Sie werden benötigt, um die eingangs geforderte Vollständigkeit zu erzielen. Wir verzichten in diesem Abschnitt weitgehend auf formale Darstellungen und diskutieren wichtige Aspekte anhand einfacher Beispiele.

7.5.1 Verschieben von Aktivitäten und Kontrollblöcken

Die von $ADEPT_{flex}$ angebotene *moveActivity*-Operation ermöglicht es, Aktivitätenknoten von ihrer aktuellen an eine neue Position im KF-Graphen zu verschieben. Die Festlegung der Zielposition kann dabei ähnlich wie bei den in Abschnitt 7.3 vorgestellten Einfügeoperationen erfolgen, d.h. die zu verschiebende Aktivität kann nach dem Herauslösen aus ihrem bisherigen Kontext zwischen zwei direkt aufeinanderfolgenden Knoten, parallel zu einem Kontrollblock oder als neuer Teilzweig einer Verzweigung wieder in den Graphen eingefügt werden. Unter Verwendung der *moveActivity*-Operation lassen sich komplexe Sprungoperationen in KF-Ausführungsgraphen, wie das Vorziehen von Aktivitäten (vgl. Abschnitt 7.6.2), einfach verwirklichen. Darüber hinaus können komplexe Änderungen, wie das Vertauschen der Bearbeitungsreihenfolge zweier Aktivitäten oder die parallele Anordnung bisher aufeinanderfolgender Aktivitäten, mit dieser Basisoperation effizient realisiert werden. Vereinfacht ausgedrückt, wird bei Anwendung von *moveActivity* die betreffende Aktivität aus ihrer bisherigen Position im KF-Graphen herausgelöst und an der gewünschten Stelle neu eingefügt. Im Gegensatz zur *deleteActivity*-Operation werden der Knoten und seine assoziierten Datenkanten allerdings nicht wirklich aus dem Graphen gelöscht, sondern nur Kontextkanten des Knotens „umgehängt“. Dadurch lassen sich bestimmte, für die Praxis sehr wichtige WF-Modifikationen wesentlich effizienter bewerkstelligen als bei kombinierter Anwendung von Lösch- und Einfügeoperationen. Darüber hinaus können erforderlichenfalls auch bereits gestartete Knoten oder Blöcke (siehe unten) verschoben werden.

Ein einfaches Beispiel für die Anwendung der *moveActivity*-Operation zeigt Abbildung 7-14: Im Ausführungsgraphen aus a) soll die Bearbeitung des Knotens `C` ausnahmebedingt vorgezogen werden, obwohl die Ausführung von `B` noch nicht gestartet wurde. Prinzipiell ist dies möglich, da `C` kein von `B` datenabhängiger Schritt ist. Je nach Ausnahmesituation kann es sinnvoll sein, die Bearbeitung von Aktivität `B` komplett auszulassen, zunächst Aktivität `C` und dann `B` zu bearbeiten oder aber die Aktivität `B` parallel zu `C` fortzuführen. Dem erstgenannten Änderungswunsch kann durch Anwendung der *deleteActivity*-Operation entsprochen werden. Die beiden letztgenannten Fälle lassen sich auf die *moveActivity*-Operation zurückführen. Abbildung 7-14 b) zeigt den Ausführungsgraph nach

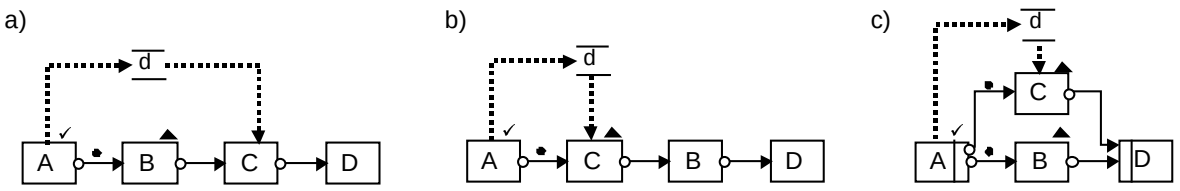


Abbildung 7-14: *Wirkungsweise der moveActivity-Operation*: *b) zeigt das Ergebnis der Anwendung von moveActivity*(..., *C*, `POSITION BETWEEN` *A and B*, ...) *und c) das Ergebnis der Anwendung von moveActivity*(..., *C*, `POSITION PARALLEL TO` (*B*, *B*) *auf den Graphen aus a*)

Verschieben von `C` an die Position zwischen `A` und `B` (serielles Wiedereinfügen). Dabei wurde die Aktivierung von `B` aufgehoben und nach der Neubewertung des Graphen der Knoten `C` aktiviert. In Abbildung 7-14 c) ist der Ausführungsgraph dargestellt, der sich aus a) ergibt, wenn `C` an eine Position parallel zur Aktivität `B` (bzw. zum Kontrollblock (`B`, `B`)) verschoben wird (paralleles Wiedereinfügen). Anhand dieses Beispiels wird bereits ersichtlich, daß für die Anwendung von *moveActivity* gewisse Voraussetzungen getroffen werden müssen, um anschließend wieder einen korrekten und konsistenten WF zu erhalten. Im Beispiel wäre das Verschieben von `C` an eine Position vor dem Knoten `A` aus zweierlei Gründen nicht erlaubt: Zum einen weil die Aktivität `A` (und damit alle ihre Vorgänger im Graphen) bereits abgeschlossen wurden, zum anderen, weil dadurch die Parameterversorgung von `C` nicht mehr gewährleistet wäre.

Bei Anwendung der *moveActivity*-Operation auf Knoten mit ein- oder ausgehenden Spezialkanten werden weitere Überprüfungen erforderlich, um die geforderten Strukturierungsregeln aufrechterhalten zu können. Der Grund hierfür ist, daß diese Kanten beim Verschieben nicht gelöscht, sondern weiterhin erhalten bleiben. Besitzt die zu verschiebende Aktivität X zum Beispiel eine einmündende (ausgehende) Sync-Kante $src \rightarrow X$ ($X \rightarrow src$), so darf ihr Knoten nicht in einen Bereich des Graphen verschoben werden, der vor (nach) dem Knoten *src* liegt. Andernfalls würden unerwünschte Zyklen und damit eine Verletzung von Regel KF-4* resultieren. Auch für die Aufrechterhaltung der anderen für Sync-Kanten definierten Strukturierungsregeln (Regeln KF-5, KF-6 und KF-7) müssen entsprechende Bedingungen definiert werden. Ein Beispiel für das Verschieben einer Aktivität mit ausgehender Sync-Kante zeigt Abbildung 7-15: Um die Bearbeitung von Aktivität `G` vorzuziehen, wird sie an eine Position parallel zu `F` verschoben. Dies ist möglich, da für die Sync-Kante `G` → `D` die geforderten Regeln auch im Anschluß erfüllt sind. Nicht erlaubt wäre dagegen das Verschieben von `G` in einen Bereich, der dem Knoten `D` nachfolgt.

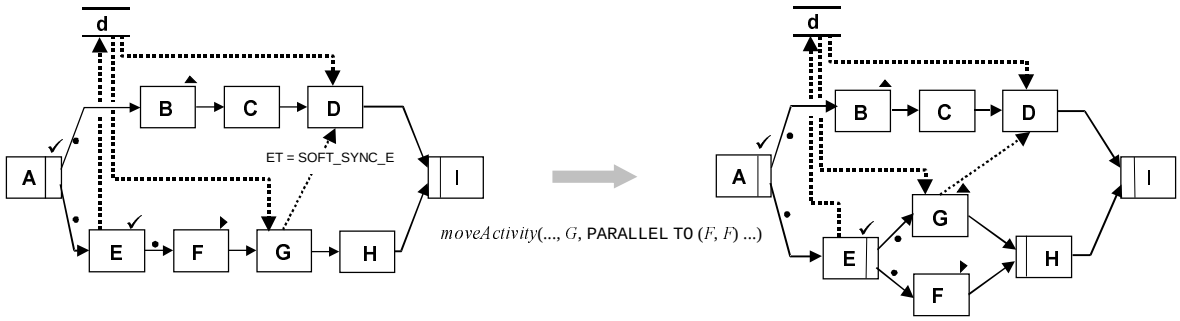


Abbildung 7-15: *Verschieben eines Aktivitätenknotens mit ausgehender Sync-Kante*

Aus Datenflußsicht muß beachtet werden, daß bei Anwendung von *moveActivity* die betreffende Aktivität X früher oder später als ursprünglich vorgesehen durchgeführt wird. Ersteres ist bei Ad-hoc-Vorwärtssprüngen in Ausführungsgraphen der Fall. Dabei muß insbesondere wieder gewährleistet sein, daß alle obligaten Lesezugriffe von X und seiner Nachfolger weiterhin versorgt werden können. Des weiteren muß sichergestellt werden, daß es durch das Verschieben nicht zu parallelen Schreibzugriffen auf dem Datenkontext kommt. Die hierzu notwendigen DF-Analysen und -Anpassungen können, ähnlich wie bei den zuvor vorgestellten Operationen, wieder eingegrenzt werden. Wesentlich dabei ist die Anordnungsbeziehung zwischen der Position der Aktivität vor und nach dem Verschieben.

Die *moveActivity*-Operation kann verallgemeinert werden, indem nicht nur das Verschieben einzelner Aktivitäten, sondern ganzer Kontrollblöcke gestattet wird. Hierfür bietet ADEPT$_{flex}$ die Operation *moveBlock* an. Sie wird auf Modellierungsebene zur Realisierung komplexerer Änderungen benötigt, kann aber auch auf Instanzebene nützlich sein. Wir verzichten an dieser Stelle auf eine Darstellung.

7.5.2 Hinzufügen und Löschen von Spezialkanten

Auch für die Hinzunahme und das Entfernen von Spezialkanten (Sync-, Priorisierungs- und Fehlerkanten) stellt ADEPT$_{flex}$ Basisänderungsoperationen bereit. Wir beschreiben kurz am Beispiel des Hinzufügens und Löschens von Sync-Kanten einige wichtige Aspekte. Sollen bisher parallele Aktivitäten in der Folge sequentiell bearbeitet werden, kann dies durch das Einfügen neuer Sync-Kanten (→ *insertSyncEdge*) erzwungen werden. Umgekehrt führt das Entfernen von Sync-Kanten (→ *deleteSyncEdge*) dazu, daß bisher aufeinanderfolgende Aktivitäten nun parallel bearbeitbar sind. Um wieder einen korrekten und konsistenten Ausführungsgraphen zu erhalten, unterliegt die Anwendung der *insertSyncEdge*- und der *deleteSyncEdge*-Basisänderungsoperation gewissen Einschränkungen. Sie werden in der nachfolgenden Tabelle zusammengefaßt (Kontrollflußsicht):

insertSyncEdge(*CFS*, *src*, *dest*, *etype*) Fügt in den KF-Graphen bzw. KF-Ausführungsgraphen *CFS* eine einfache oder strikte Sync-Kante (mit Kantentyp *etype* ∈ {`SOFT_SYNC_E`, `STRICT_SYNC_E`}) vom Knoten *src* zum Knoten *dest* ein. Die Semantik dieser Operation ist dem entsprechend durch *AddEdges*(*CFS*, {(*src*, *dest*, *etype*)}) festgelegt (vgl. Tabelle 7-2). **Vorbedingungen:** 1. (*src*, *dest*, *etype*) ∉ E, d.h. es gibt noch keine identische Kante im KF-Graphen 2. *src*, *dest* ∈ *N*; *dest* ∉ $pred^*(src)$ 3. $BranchNodes^{CFS}(src, dest) = (split, join) \in N \times N$ mit $V_{out}^{split} = V_{in}^{join}$ = `ALL_Of_ALL` 4. Entspricht *Z* der Knotenmenge eines Teilzweigs einer parallelen Verzweigung mit finaler Auswahl, so muß für *etype* = `STRICT_SYNC_E` gelten: *src* ∈ *Z* ⇒ *dest* ∈ *Z* 5. Für das Einfügen auf Instanzebene muß gelten: *NS*(*dest*) ∈ {`NOT_ACTIVATED`, `ACTIVATED`}. Bei Einfügen vor einem bereits aktivierten Knoten wird dessen Aktivierung wieder aufgehoben.
deleteSyncEdge(*CFS*, *src*, *dest*, *etype*) Entfernt aus dem KF-Graphen bzw. -Ausführungsgraphen *CFS* die Sync-Kante (*src*, *dest*, *etype*). Die Semantik dieser Operation ist festgelegt durch *DeleteEdges*(*CFS*, {(*src*, *dest*, *etype*)}) (vgl. Tabelle 7-2) **Vorbedingungen:** 6. (*src*, *dest*, *etype*) ∈ E; *etype* ∈ {`SOFT_SYNC_E`, `STRICT_SYNC_E`} 7. Für das Löschen einer Sync-Kante auf Instanzebene muß gelten: *ES*(*e*) = `NOT_SIGNALED`

Tabelle 7-7: *Die Basisänderungsoperationen insertSyncEdge und deleteSyncEdge.*

Wie man leicht zeigen kann, führt die Anwendung dieser Operationen unter den genannten Bedingungen wieder zu einem korrekten und konsistenten KF-Graphen bzw. -Ausführungsgraphen. Voraussetzung 2 stellt sicher, daß es durch das Einfügen der Sync-Kante nicht zu Zyklen im KF-Graphen kommt (Regel KF-4*). Bedingung 3 garantiert, daß neue Sync-Kanten ausschließlich zwischen Knoten verschiedener Zweige einer Parallelverzweigung eingefügt werden (Regel KF-5). Durch Bedingung 4 erreichen wir, daß strikte Sync-Kanten nicht aus Teilzweigen einer Parallelverzweigung mit finaler Auswahl hinaus führen (Regel KF-6). Bei der Hinzunahme strikter Sync-Kanten muß zusätzlich zu den genannten Bedingungen 1.- 5. überprüft werden, ob die in Kapitel 4 geforderten Dynamik-Eigenschaften des WF erhalten bleiben. Dabei kann wie in Abschnitt 4.6.4 beschrieben vorgegangen werden.

Die Anwendung der *insertSyncEdge*-Operation hat keine Auswirkungen auf das DF-Modell des Workflow. Für jeden Aktivitätenknoten $n \in N$ bleiben seine Eingabeparameter versorgt, da seine bisherigen Vorgänger auch nach dem Einfügen der Sync-Kante der Menge $pred^*(n)$ angehören (→ Erhalt der Regel DF-1). Ebensowenig kann es durch die Hinzunahme einer Sync-Kante dazu kommen, daß bisher aufeinanderfolgende Knoten in der Folge parallel bearbeitet werden. Damit ist das Entstehen

paralleler Schreibzugriffe auf ein Datenelement ausgeschlossen (→ Erhalt der Regel DF-2). Genau diese beiden Fälle können beim Löschen von Sync-Kanten auftreten. Aus diesem Grund müssen bei Anwendung der *deleteSyncEdge*-Operation zusätzliche Überprüfungen und ggf. auch Anpassungen des Datenflußes vorgenommen werden, um seine Korrektheit aufrechtzuerhalten. Besonders wichtig ist es dabei, den Umfang der erforderlichen DF-Analysen vernünftig einzugrenzen. Wir zeigen am Beispiel der Überprüfung von Regel DF-1 (→ vollständige Versorgung obligater Eingabeparameter von Aktivitäten), wie diese Eingrenzung in ADEPT$_{flex}$ erfolgt.

(*CFS*, *DFS*) sei der korrekte WF-Graph vor dem Löschen der Sync-Kante *src* → *dest*. Aufgabe ist es dann festzustellen, welche obligaten Lesezugriffe auf den WF-Datenkontext durch das Entfernen dieser Kante potentiell nicht mehr versorgt werden können. Zu diesem Zweck ermitteln wir zuerst die Menge all derjenigen Aktivitätenknoten $n \in N$, deren Vorgängermenge $pred^*(n)$ sich infolge des Löschens der Sync-Kante *src* → *dest* verkleinert. Wir erhalten:

$$CriticalSet := \left(succ^*(dest) \cup \{dest\}\right) \neg \bigcup_{\substack{x \in succ(src) \\ x \neq dest}} succ^*(x) \subseteq succ^*(src)$$

Die Bestimmung von *CriticalSet* erfolgt auf der Grundlage des Ausgangsgraphen (*CFS*, *DFS*). Die Aktivitäten dieser Menge (und nur diese) besitzen nach dem Entfernen der Sync-Kante weniger Vorgängerknoten als zuvor und damit auch weniger potentielle Versorger der von ihnen gelesenen Datenelemente (vgl. Kapitel 5). Konkret ergibt sich für einen Knoten $n \in CriticalSet$ nach dem Löschen der Sync-Kante die neue Vorgängermenge $pred^*(n) - LeftOutSet$ mit

$$LeftOutSet := pred^*(dest) \neg \bigcup_{\substack{x \in pred(dest) \\ x \neq src}} \left(pred^*(x) \cup \{x\}\right) \subseteq pred^*(dest)$$

Auch die Bestimmung der Menge *LeftOutSet* kann auf der Grundlage des Ausgangsgraphen (*CFS*, *DFS*) erfolgen. Basierend auf diesen Überlegungen können wir nun angeben, welche DF-Analysen zur Sicherstellung bzw. Überprüfung der Regel DF-1 konkret durchzuführen sind. Beim Löschen der Sync-Kante muß für einen Knoten $n \in CriticalSet$ die Versorgung seines Lesezugriffs auf ein Datenelement *d* genau dann geprüft werden, wenn es in der Menge *LeftOutSet* mindestens eine Aktivität gibt, die dieses Datenelement obligat schreibt. Sind für alle Knoten der Menge *CriticalSet* diese Lesezugriffe auch nach dem Löschen der Sync-Kante versorgt, so ist Regel DF-1 weiterhin erfüllt. Formal:

Satz 7-3 (Parameterversorgung bei Löschen von Sync-Kanten)

Bezeichne (*CFS*, *DFS*) einen korrekten WF-Graphen und (CFS_{new}, *DFS*) den daraus hervorgehenden WF-Graphen nach dem Löschen der Sync-Kante (*src*, *dest*, *etype*) ∈ *E*. Die Knotenmengen *CriticalSet* und *LeftOutSet* seien für (*CFS*, *DFS*) wie oben angegeben definiert. Ferner sei für einen Knoten *n* aus *CriticalSet* die Menge *CriticalDataElements(n)* definiert durch

$$\{d \in D \mid ActivityReads^{DFS}(n, d, \texttt{MANDATORY}) \wedge (\exists\, x \in LeftOutSet: ActivityWrites^{DFS}(x, d, \texttt{MANDATORY}))\}$$

Dann gilt:

$\forall\, n \in CriticalSet, \forall\, d \in CriticalDataElements(n)$ gilt: $WriterExists(CFS_{new}, DFS, n, d) = True$ (*)

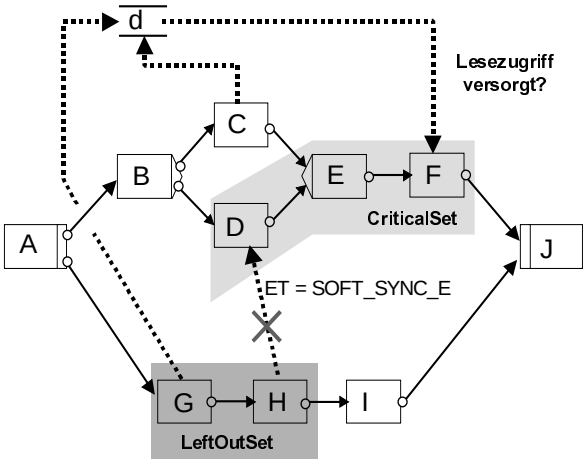
⇔

Der WF-Graph (CFS_{new}, *DFS*) erfüllt Regel DF-1, d.h. alle obligaten Lesezugriffe auf den Datenkontext können versorgt werden.

Mit (*) erhalten wir somit eine notwendige und hinreichende Bedingung für die Sicherstellung der Parameterversorgung beim Löschen einer Sync-Kante. Ist diese Bedingung nach dem Entfernen der Sync-Kante nicht mehr erfüllt, müssen entweder begleitende Maßnahmen ergriffen werden (z.B. Vorschalten von Nachforderungsdiensten), oder aber die Löschoperation ist zu verbieten. Ähnliche Überlegungen müssen auch im Hinblick auf die Sicherstellung von Regel DF-2 angestellt werden. Wir verzichten an dieser Stelle auf eine Darstellung.

Beispiel 7-11 (Löschen einer Sync-Kante)

Für das Löschen der Sync-Kante H → D ergeben sich die Mengen *CriticalSet* = {D, E, F} und *LeftOutSet* = {G, H}. Da der Knoten F ∈ *CriticalSet* das Datenelement d obligat liest und sein bisheriger Vorgängerknoten G ∈ *LeftOutSet* dieses Datenelement obligat schreibt, muß beim Entfernen der Sync-Kante die Versorgung des Lesezugriffs auf d neu überprüft werden (Vor dem Löschen gilt: *WriterExists*(*CFS*, *DFS*, F, d) = *True*; vgl. Beispiel 5-5). Nach dem Entfernen der Kante ergäbe sich: *WriterExists*(CFS_{new}, *DFS*, F, d) = *False*. Soll die Sync-Kante gelöscht werden, müßten deshalb begleitende Änderungen des DF-Modells vorgenommen werden (z.B. Einfügen eines Nachforderungsdienstes vor F).



Auch für das Einfügen und Löschen von Priorisierungs- und Fehlerkanten bietet ADEPT entsprechende Basisoperationen an: *insertPriorityEdge*, *deletePriorityEdge*, *insertFailureEdge* und *deleteFailureEdge*. Für die Anwendung der beiden erstgenannten Operationen bestehen ähnliche Vorbedingungen wie für Sync-Kanten. Für das Einfügen und Löschen von Fehlerkanten sind die Voraussetzungen so definiert, daß anschließend wieder Regel KF-9 (vgl. Kapitel 6) erfüllt ist. Aufgrund der speziellen Semantik von Priorisierung- und Fehlerkanten werden bei ihrem Einfügen bzw. Löschen, im Gegensatz zu Sync-Kanten, keine Anpassungen des DF-Schemas notwendig.

7.5.3 Bildung und Auflösung physischer Blockstrukturen

Aus verschiedenen Gründen kann es wünschenswert sein, in einem WF-Graphen *logische* durch *physische* Blockstrukturen zu ersetzen oder umgekehrt physische Blöcke wieder aufzulösen. Mit logischen Blockstrukturen sind Kontrollblöcke gemeint, während physische Blockstrukturen Sub-Workflows, d.h. Prozeß- oder Blockaktivitäten (vgl. Kapitel 3), beschreiben. Die Bildung von physischen Blöcken dient vor allem der besseren Strukturierung des Gesamtmodells oder der Wiederverwendung von Ausschnitten des WF-Modells in einem anderen Kontext (z.B. zur Definition anderer WF-Modelle). Umgekehrt kann die Auflösung von Sub-WF-Strukturen hilfreich sein, wenn für ihre Ausführung obligate Eingabeparameter fehlen, z.B. infolge des Löschens vorangehender Schritte. Durch die Auflösung der physischen Blockstruktur gewinnt man hier mehr Flexibilität bzgl. der Nachversorgung fehlender Daten. Sie müssen dann nicht mehr notwendigerweise bereits beim Start des Sub-WF vorliegen, sondern können direkt vor den Aktivitäten, die sie effektiv benötigen, nachversorgt werden. Dies ist besonders dann vorteilhaft, wenn die fehlenden Eingabedaten erst im späteren Verlauf der Sub-WF-Ausführung benötigt werden.

Für die Bildung physischer Blockstrukturen stellt ADEPT$_{flex}$ die *nestBlock*-Basisoperation bereit. Sie kann auf einen (abgeschlossenen) Kontrollblock (n_{begin}, n_{end}) eines WF-Graphen (*CFS*, *DFS*) angewendet werden, wenn dieser durch einen strukturgleichen Sub-WF-Graphen ersetzt werden soll. Dazu wird eine neue WF-Vorlage V_{new} mit WF-Graph (CFS_{new}, DFS_{new}) erzeugt und im Vorlagenrepository abgelegt. CFS_{new} entspricht dabei dem durch den Kontrollblock (n_{begin}, n_{end}) gebildeten KF-Teilgraphen von *CFS*, ergänzt um einen WF-Startknoten und einen WF-Endknoten. Das Datenkontext-Schema D_{new} wird durch diejenigen Datenelemente aus *D* gebildet, auf die von Knoten des Kontrollblocks (n_{begin}, n_{end}) lesend oder schreibend zugegriffen wird (inkl. Datenzugriffen dieser beiden Knoten). Dem entsprechend werden die zugehörigen Lese- und Schreibkanten aus *DFS* in das DF-Modell DFS_{new} kopiert. Schließlich werden – ausgehend vom WF-Graphen (CFS_{new}, DFS_{new}) – die Ein- und Ausgabeparameter der Vorlage sowie deren Eigenschaften automatisch bestimmt. Dabei kann wie in Abschnitt 5.4.4 beschrieben vorgegangen werden. Diese Information werden zusammen mit den Aufrufparametern der *nestBlock*-Operation für die Bildung der Vorlagenschnittstelle verwendet. Bei Aufruf von *nestBlock* kann angegeben werden, ob die neu gebildete Vorlage wiederverwendbar sein soll oder nur im Kontext der vorliegenden WF-Definition verwendet wird (vgl. Kapitel 3).

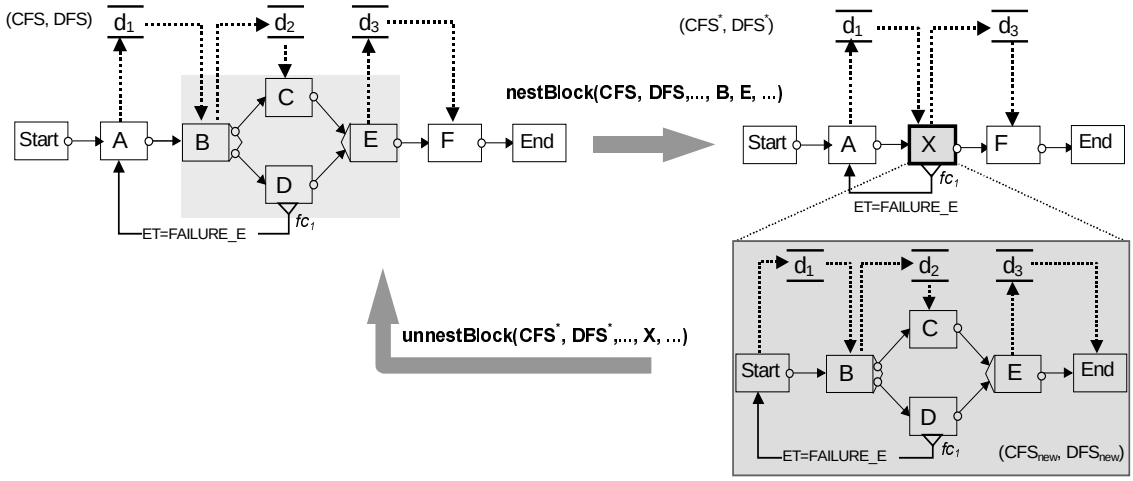


Abbildung 7-16: *Bildung und Auflösung physischer Blockstrukturen in ADEPT$_{flex}$*

Nach dem Erzeugen der neuen (Sub-)WF-Vorlage, wird im Ursprungsgraphen (*CFS*, *DFS*) der durch (n_{begin}, n_{end}) festgelegte Kontrollblock durch einen neuen Knoten ersetzt. Er referenziert die neue Vorlage V_{new} und übernimmt die Eingangssemantik und -kontrollkanten des Knotens n_{begin} sowie die Ausgangssemantik und -kontrollkanten des Knotens n_{end}. Des weiteren werden aus dem bisherigen DF-Modell *DFS* alle Datenkanten der Menge DFS_{new} entfernt. Dasselbe gilt für die zur Menge D_{new} korrespondierenden Datenelemente, wenn sie im Anschluß weder einmündende noch ausgehende Datenkanten besitzen. Schließlich werden die Ein- und Ausgabeparameter des neu gebildeten Blocks nach einfachen Regeln an den Datenkontext angeknüpft. Ein Beispiel für die Anwendung von *nestBlock* zeigt Abbildung 7-16. Hier wird ein Verzweigungskontrollblock durch einen strukturgleichen Sub-WF ersetzt. Anhand des Beispiels ist auch erkennbar, wie mit Fehlerkanten verfahren werden kann, deren Quellknoten innerhalb und deren Zielknoten außerhalb des Kontrollblocks liegen (vgl. Kapitel 6).

Die zu *nestBlock* inverse Operation ist *unnestBlock*. Sie gestattet es, Sub-WF-Graphen aufzulösen und sie in den WF-Graphen der übergeordneten Ebene einzubetten. Dabei müssen ggf. Knoten- und Datenelementbezeichner umbenannt werden, um wieder Eindeutigkeit zu erzielen. Ein Beispiel für die Anwendung dieser Operation zeigt Abbildung 7-16. Die Operationen *nestBlock* und *unnestBlock* können auch auf Instanzebene angewendet werden, wobei keine Statusbeschränkungen bestehen. Beispielsweise kann *unnestBlock* auch auf eine bereits gestartete oder beendete Sub-WF-Instanz angewendet werden. In einem solchen Fall werden Anpassungen im Instanzabhängigkeitsgraphen des Gesamt-WF (vgl. Kapitel 4) vorgenommen und die Ablaufhistorien des Sub-WF und des übergeordneten WF gemischt. Dies erfolgt, indem die Einträge der beiden Historien entsprechend den Zeitpunkten ihres Auftretens in der neuen Ablaufhistorie angeordnet werden. Der Historieneintrag für den Start bzw. das Ende des Sub-WF fällt weg. Das Mischen der beiden Historien wird erforderlich, um auch für den neuen Ausführungsgraphen im Bedarfsfall Rücksetzoperationen ermöglichen zu können.

7.5.4 Änderung von DF-Modellen

Im allgemeinen muß es möglich sein, das DF-Modell eines WF gezielt abzuändern. Beispielsweise kann es infolge einer fehlerhaften Modellierung notwendig werden, eine in einen Aktivitätenknoten einmündende Lesekante zu löschen und durch eine neue Lesekante auf ein anderes Datenelement zu ersetzen. Für die Durchführung solcher Änderungen stellt ADEPT$_{flex}$ die Basisoperationen *insertDataElement*, *deleteDataElement*, *insertDataEdge* und *deleteDataEdge* bereit. Ihre Semantik kann auf Basis der in Tabelle 7-3 vorgestellten Änderungsprimitive beschrieben werden. Hinzu kommt ein Reihe von Restriktionen für ihre korrekte Verwendung. Neue Datenelemente können mittels *insertDataElement* in das DF-Schema einer WF-Vorlage oder -Instanz eingefügt werden, vorausgesetzt sie besitzen eindeutige, im Schema noch nicht verwendete Bezeichner. Die Anwendung der Operation *deleteDataElement* dagegen ist nur erlaubt, wenn das betreffende Datenelement keine assoziierten Datenkanten besitzt. Darüber hinaus können Datenelemente auf WF-Instanzebene nicht mehr gelöscht werden, wenn auf sie bereits Schreibzugriffe stattgefunden haben. Der Grund dafür ist, daß die geschriebenen Werte zur Versorgung der Lesezugriffe nachfolgender WF-Aktivitäten dienen. Des weite-

ren werden sie für ein eventuelles (partielles) Rollback des WF im Fehlerfall benötigt (vgl. Kapitel 6). Dem entsprechend dürfen auch die Datenkanten bereits beendeter Aktivitäten auf Instanzebene nicht mehr entfernt werden. Die Anwendung der *insertDataEdge*-Operation schließlich unterliegt den in Definition 5-1 getroffenen Einschränkungen bzgl. der Verträglichkeit der Wertebereiche von Datenelementen und Parametern. Werden die genannten Operationen innerhalb einer Änderungstransaktion angewendet, muß anschließend wieder ein korrektes DF-Modell resultieren.

7.5.5 Änderung von Knoten- und Kantenattributen

Die bisher vorgestellten Basisoperationen betreffen strukturelle Aspekte. Darüber hinaus bietet ADEPT$_{flex}$ einfache Operationen für die Festlegung und Änderung von Knoten- und Kanteneigenschaften an: *changeActivityAttribute, changeEdgeAttribute* und *changeActivityTemplate*. Die beiden erstgenannten Operationen gestatten es, Attribute von Aktivitätenknoten (z.B. Prioritäten, Entscheidungsparameter) und Kontrollkanten (Auswahlwert, Fehlerwert, Kantenpriorität) abzuändern. Entsprechende Modifikationen können bei Bedarf auch noch auf WF-Instanzebene angewendet werden, zumindest solange sich der Knoten in einem der Zustände `NOT_ACTIVATED` oder `ACTIVATED` befindet bzw. die Kante als `NOT_SIGNALED` markiert ist. Durch Anwendung der Operation *changeActivityTemplate* kann die Vorlage eines (noch nicht gestarteten) Knotens ausgetauscht werden. Daraus kann sich die Notwendigkeit begleitender Änderungen ergeben. Weist z.B. die neue Vorlage gegenüber der bisherigen eine andere Schnittstelle auf, müssen die Bindungen zwischen Aktivitätenparametern und Datenelementen neu erfolgen. Die Anwendung der Operation *changeActivityTemplate* ist nicht notwendig, wenn sich die Implementierung der Vorlage ändert, z.B. durch Austausch einer externen Anwendungskomponente oder durch Neufestlegung eines Verzweigungsprädikates. Solche Änderungen können auf WF-Instanzebene durch Mechanismen wie spätes Binden berücksichtigt werden, ohne daß weitere Anpassungen notwendig werden. Voraussetzung ist allerdings, daß die neue Vorlagenimplementierung plug-in-kompatibel zur bisherigen Vorlagenschnittstelle ist (vgl. Kapitel 3).

7.6 Realisierung komplexer Änderungsoperationen in ADEPT$_{flex}$

Durch kombinierte Anwendung der vorgestellten Basisänderungsoperationen lassen sich anspruchsvolle Änderungsoperationen wie das Einfügen von Aktivitäten zwischen Knotenmengen oder das Vorziehen von Aktivitäten (mit bzw. ohne Nachholen der übersprungenen Schritte) realisieren. ADEPT$_{flex}$ stellt für den Anwendungsentwickler (d.h. den Entwickler von WF-Klientenanwendungen) bzw. den Anwender selbst – zusätzlich zu den beschriebenen Basisänderungsoperationen – eine Reihe solcher vordefinierten, komplexen Änderungsoperationen bereit. Dadurch können sie von Details komplexer WF-Änderungen (z.B. einer komplexen Sprung- oder Einfügeoperation) weitgehend entlastet werden, wodurch sich die Anwendungsentwicklung weiter vereinfacht und die Benutzerfreundlichkeit des System erhöht. Darüber hinaus – und das ist ebenfalls ein interessanter Aspekt – können die Voraussetzungen für die Anwendung komplexer Änderungsoperationen aus den Vorbedingungen der verwendeten Basisoperationen abgeleitet werden. Dem entsprechend ist auch der Korrektheitsnachweis für komplexe Operationen auf die korrekte Anwendung der zugrundeliegenden Basisoperationen zurückführbar. (Wir unterstellen hier, daß die verschiedenen Basisoperationen einer komplexen Änderungsoperation innerhalb derselben Änderungstransaktion abgewickelt werden.)

7.6.1 (Dynamisches) Einfügen einer Aktivität zwischen zwei Knotenmengen

Eine wichtige Änderungsoperation ist das Einfügen einer Aktivität zwischen zwei Knotenmengen. Auf ihrer Basis läßt sich ein breites Spektrum an Einfügeoperationen verwirklichen (vgl. [ReDa98]). Insbesondere kann ein neu eingefügter Knoten mit Aktivitäten aus verschiedenen WF-Ausführungszweigen synchronisiert werden. Ein Anwendungsbeispiel hierzu haben wir in Abschnitt 7.4.1.3 (→ Einfügen von Versorgeraktivitäten) kennengelernt. Im folgenden zeigen wir, wie sich diese komplexe Operation auf die Basisänderungsoperationen abbilden läßt. Dabei konzentrieren wir uns auf Kontrollflußaspekte. Die notwendigen Anpassungen des DF-Schemas erfolgen ähnlich wie in Abschnitt 7.3.1.4 sowie in [ReDa98] beschrieben, so daß wir an dieser Stelle auf eine Darstellung verzichten.

7.6.1.1 Einfügen zwischen Knotenmengen in KF-Graphen

Für die Anwendung der hier vorgestellten Einfügeoperation ist die Angabe zweier nichtleerer Knotenmengen M_{before} und M_{after} erforderlich. Die Menge M_{before} legt dabei diejenigen Knoten fest, die vor der Aktivierung des neu einzufügenden Schrittes X beendet sein sollen bzw. für die feststehen muß, daß sie nicht mehr zur Ausführung kommen. Die Knoten aus M_{before} verzögern also die Aktivierung von X. Dies erlaubt es uns zum Beispiel, die Ausführung von X mit Aktivitäten unterschiedlicher Zweige einer bedingten Verzweigung zu synchronisieren. Umgekehrt umfaßt die Menge M_{after} all diejenigen Knoten, vor deren Start die neu eingefügte Aktivität beendet sein soll. Des weiteren fordern wir, daß der neu einzufügende Schritt in jedem Fall zur Ausführung kommen soll. Bei seiner Einbettung in den WF-Graphen ist deshalb darauf zu achten, daß der betreffende Knoten anschließend keinem Teilzweig einer bedingten Verzweigung oder einer Parallelverzweigung mit finaler Auswahl angehört.

Um nach Anwendung der Einfügeoperation wieder einen korrekten KF-Graphen zu erhalten, treffen wir die folgenden Einschränkungen hinsichtlich der Festlegung dieser erwähnten Knotenmengen:

(1) Für die beiden Knotenmengen M_{before} und M_{after} fordern wir, daß $\forall\, n \in M_{before}$, $\forall\, m \in M_{after}$ gilt: $m \notin pred^*(n)$, d.h. m ist kein Vorgängerknoten von n. (Mit anderen Worten: m ist entweder ein Nachfolgerknoten von n oder die beiden Knoten liegen bisher parallel zueinander.)

(2) Für KF-Graphen mit Schleifen fordern wir zusätzlich, daß der durch die Knotenmenge $B := succ^*(M_{before}) \cap pred^*(M_{after})$ induzierte Teil-KF-Graph von CFS nur vollständige Schleifenstrukturen enthält.

Auf Basis der Knotenmengen M_{before} und M_{after} sowie der Vorlage des einzufügenden Schrittes kann die gewünschte Einfügeoperation verwirklicht werden. Sie basiert auf zwei Basisänderungsoperationen, der *parallelInsert*-Operation (Einfügen einer Aktivität parallel zu einem gegebenen Kontrollblock; vgl. Abschnitt 7.3.1.2) und der in Abschnitt 7.5.2 eingeführten Basisoperation *insertSyncEdge* (Einfügen von Sync-Kanten). Zusätzlich greifen wir auf die in Kapitel 3 eingeführte Operation zur Bestimmung minimaler Kontrollblöcke bzgl. einer gegebenen Knotenmenge zurück.

```
insertBetweenNodeSets(CFS | CFS_instance, M_before, M_after, X, templ)
input
  ...
  X, templ: Bezeichner bzw. Vorlage des neu einzufügenden Knotens
  ...
begin
  // Bestimmung des min. Blockes der die Knoten der beiden Mengen M_before und M_after umschließt
  (n_begin, n_end):= MinBlock^CFS(M_before ∪ M_after)

  // Den ermittelten Block ggf. solange expandieren, bis er nicht mehr in einem Teilzweig einer
  // Verzweigung mit OR-Join enthalten ist. Dies wird erforderlich, um sicherzustellen, daß der
  // neu einzufügende Knoten in jedem Fall aktiviert wird.

  forall join ∈ JoinNodes^CFS with V_in^join = ONE_Of_ALL do
      split:= split^CFS(join);

      // Falls n_begin in einem Teilzweig enthalten ist, muß wegen der regelmäßigen Blockstruktur der
      // Knoten n_end in demselben Teilzweig liegen.

      if n_begin ∈ (c_succ*(split(j)) ∩ c_pred*(j)) then
          (n_begin, n_end):= (split, join)                    // Block expandieren
      endif
  done

  // Die neue Aktivität wird nun parallel zu dem ermittelten Block eingefügt. Dazu verwenden
  // wir die in Abschnitt 7.3.1.2 b) vorgestellte Basisänderungsoperation parallelInsert.
  parallelInsert(CFS | CFS_instance, n_begin, n_end, X, templ)

  // Einfügen einfacher Sync-Kanten zur Erzielung des gewünschten Ausführungsverhaltens.
```

```
  // Dazu verwenden wir die in Abschnitt 7.5.2 vorgestellte Operation insertSyncEdge
  forall n ∈ M_before do
      insertSyncEdge(n, X, SOFT_SYNC_E)
  done
  forall n ∈ M_after do
      insertSyncEdge(X, n, SOFT_SYNC_E)
  done
end // insertBetweenNodeSets
```

Dieser Algorithmus verdeutlicht nochmals, wie wichtig das Konstrukt der einfachen Sync-Kante ist, um die gewünschte Semantik zu erzielen. Des weiteren kann man durch geringfügige Modifikationen des Algorithmus verschiedene Varianten für das Einfügen zwischen Knotenmengen erzielen. Soll die eingefügte Aktivität etwa nur dann ausgeführt werden, wenn der Bereich bzw. der Block, der die Knoten aus M_{before} und M_{after} enthält, auch tatsächlich durchlaufen wird, so kann in obigem Algorithmus auf die Expandierung des minimalen Blocks verzichtet werden. Eine Beschreibung dieser einfachen Variante findet sich in [ReDa98].

Wir illustrieren die Anwendung der *insertBetweenNodeSets*-Einfügeoperation kurz an einem Beispiel:

Beispiel 7-12 (Einfügen zwischen Knotenmengen)

Ein Beispiel zeigt Abbildung 7-17: Es soll die Aktivität `X` in den dargestellten KF-Graphen zwischen den Knotenmengen M_{before} = {`C`, `D`} und M_{after} = {`F`} eingefügt werden. Bei Anwendung der *insertBetweenNodeSets*-Operation wird zunächst der minimale, die Knoten `C`, `D` und `F` umschließende Kontrollblock bestimmt (Schritt 1). Anschließend wird die neue Aktivität parallel zu diesem Block eingefügt, wozu die *parallelInsert*-Operation verwendet wird (Schritt 2). Um die gewünschten Reihenfolgebeziehungen zu erhalten, werden einfache Sync-Kanten von den Knoten `C` und `D` zum Knoten `X` sowie vom Knoten `X` zum Knoten `F` eingefügt (Schritt 3). An diesem Beispiel ist erkennbar, daß die regelmäßige Blockstrukturierung aufrechterhalten wird und daß die Hinzunahme der Sync-Kanten nicht zu Zyklen im KF-Graph führt. Abschließend werden auf den KF-Graphen ADEPT-Vereinfachungsregeln (Vereinfachungsregeln 2 + 3, vgl. Abschnitt 7.2.3) angewendet, wodurch die beiden Nullaktivitäten sowie die Sync-Kante (`X`, `F`, `SOFT_SYNC_E`) eliminiert werden (Schritt 4). Letzteres ist möglich, da der betreffende Zielknoten der Sync-Kante dem Endknoten des zuvor bestimmten Blocks entspricht. Die beschriebene Einfügeoperation wäre auch dann anwendbar, wenn der durch (`B`, `F`) gebildete Kontrollblock eine bedingte Verzweigung darstellen würde.

Aus den eingangs geforderten Bedingungen bzgl. der beiden Mengen M_{before} und M_{after} folgt unmittelbar die Korrektheit des resultierenden KF-Graphen. Dazu formulieren wir den folgenden Satz:

Satz 7-4 (Erhalt der Korrektheit von KF-Schemata beim Einfügen zwischen Knotenmengen)

Sei $CFS = (N, E, D, ...)$ ein korrekter KF-Graph, in den eine Aktivität mit Bezeichner $X \notin N$ und Vorlage $templ \in Templates$ eingefügt werden soll. Ferner seien durch $M_{before}, M_{after} \subset N - \{Start, End\}$ zwei Knotenmengen gegeben, welche die eingangs genannten Bedingungen (1) und (2) erfüllen.

Wird unter diesen Voraussetzungen die *insertBetweenNodeSets*-Operation angewendet, um X zwischen den beiden Mengen M_{before} und M_{after} einzufügen, so resultiert wieder ein korrektes KF-Schema $CFS^* = (N^*, E^*, D^*, ...)$.

Satz 7-4 garantiert u.a. den Erhalt der Blockstrukturierung und der Zyklenfreiheit des KF-Graphen bzgl. Kontroll- und Sync-Kanten. Dadurch sind Verklemmungen während der Ausführung ausgeschlossen und es gelten weiterhin die in Kapitel 4 festgelegten Dynamik-Eigenschaften.

<u>Beweisskizze</u> (zu Satz 7-4):

Es genügt zu zeigen, daß die Voraussetzungen für die korrekte Verwendung der eingesetzten Basisänderungsoperationen zutreffen. Die Bedingungen für die Anwendung der *parallelInsert*-Operation (vgl. Abschnitt 7.3.1.2 b) sind erfüllt: Es gibt im abzuändernden KF-Graphen noch keinen Knoten mit Bezeichner X und durch das Knotenpaar (n_{begin}, n_{end}) ist ein Kontrollblock (mit $n_{begin}, n_{end} \in N - \{Start, End\}$) des KF-Graphen CFS bestimmt. Letzteres folgt unmittelbar aus der Definition der verwendeten

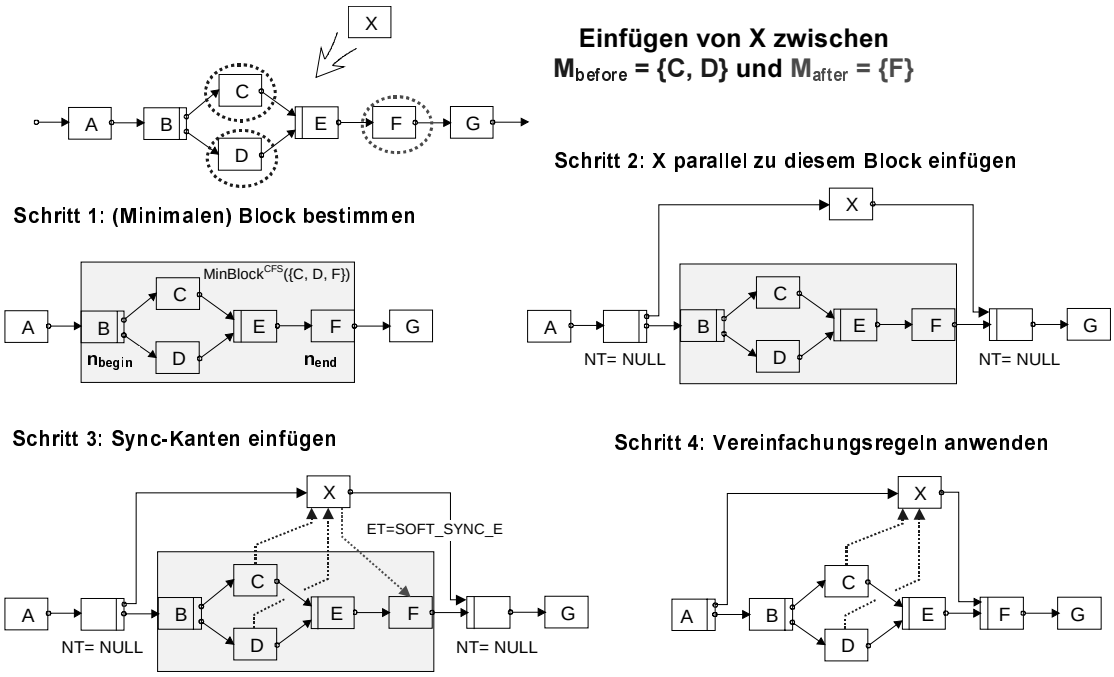


Abbildung 7-17: *Einfügen einer neuen Aktivität zwischen zwei Knotenmengen*

MinBlock-Funktion. Damit ist der Erhalt der regelmäßigen Blockstrukturierung (Regel KF-3) gesichert. Auch die Regeln KF-1, KF-2 und KF-4 bleiben weiterhin gültig.

Auch die Bedingungen für das Einfügen der einfachen Sync-Kanten sind erfüllt: Sie sollen Knoten des durch (n_{begin}, n_{end}) beschriebenen Kontrollblocks mit dem *parallelen* Knoten X verknüpfen. Dies garantiert uns den Erhalt von Regel KF-5.

Auch die Zyklenfreiheit des KF-Schemas CFS^* (und damit Verklemmungsfreiheit; Regel KF-4*) wird durch das Einfügen der Sync-Kanten nicht verletzt. Dies folgt unmittelbar aus Voraussetzung (1).

Schließlich garantiert uns die Bedingung (2), daß die neu einzufügenden Sync-Kanten nicht aus einer Schleife heraus bzw. in eine Schleife hinein führen (Regel KF-7, (6)). Damit sind alle Strukturierungsregeln für die Verwendung einfacher Sync-Kanten weiterhin gültig. Andere Regeln, z.B. die Verwendung von Fehlerkanten betreffend, gelten trivialerweise weiterhin, da sie von der Einfügoperation nicht tangiert werden.

q.e.d.

Auf Basis der bisher getroffenen Einschränkungen für die Knoten der Mengen M_{before} bzw. M_{after} kann nicht ausgeschlossen werden, daß im KF-Graphen Sync-Kanten eingefügt werden, die zur Erzielung des gewünschten Ausführungsverhalten der neuen Aktivität nicht benötigt werden. Dieser Fall tritt insbesondere dann auf, wenn einzelne Knoten der Menge M_{before} (M_{after}) im KF-Graphen aufeinander folgen. Um solche überflüssigen Sync-Kanten zu vermeiden, fordern wir zusätzlich, daß die Knoten der Menge M_{before} (M_{after}) unterschiedlichen Teilzweigen einer Verzweigung angehören müssen. Trifft dies nicht zu, so können die beiden Mengen vor Anwendung der Einfügeoperation nach dem folgendem Prinzip „minimiert“ werden:

- Entferne aus M_{before} alle Knoten der Menge $\{n \in M_{before} \mid \exists\, n^* \in M_{before}$ mit $n^* \in c_succ^*(n)\}$ (Minimierung von M_{before})
- Entferne aus M_{after} alle Knoten der Menge $\{m \in M_{after} \mid \exists\, m^* \in M_{after}$ mit $m^* \in c_pred^*(m)\}$ (Minimierung von M_{after})

Darüber hinaus lassen sich, je nach Kontext in den die neue Aktivität in den Graphen eingesetzt wird, im Anschluß an die Transformationsschritte 1–3 ggf. noch Vereinfachungen des KF-Schemas vornehmen. Grundlage hierfür bilden die in Abschnitt 7.2.3 vorgestellten Vereinfachungsregeln (vgl. Beispiel 7-12). Insgesamt erreichen wir durch die Minimierung der Mengen M_{before} und M_{after} *vor*

sowie der Anwendung von Vereinfachungsregeln *nach* Einfügen der neuen Aktivität, daß die resultierenden KF-Schemata redundanzfrei sind. Dies ist für die Verständlichkeit der Modelle essentiell.

7.6.1.2 Einfügen zwischen Knotenmengen auf Workflow-Instanzebene

Um die *insertBetweenNodeSets*-Operation auf eine WF-Instanz anwenden zu können, muß sie zusätzlich zu den bereits genannten Voraussetzungen bestimmte Statusbedingungen erfüllen. Sie können direkt aus den Statusvoraussetzungen der verwendeten Basisoperationen abgeleitet werden: Das Einfügen neuer Sync-Kanten ist nur möglich, wenn sich ihr Zielknoten – im vorliegenden Fall sind dies alle Knoten der Menge M_{after} – jeweils in einem der beiden Ausführungszustände NS = `NOT_ACTIVATED` oder NS = `ACTIVATED` befindet (vgl. Abschnitt 7.5.2). Ist dies der Fall, so besitzt auch der Endknoten des ermittelten minimalen Blocks (vgl. Abschnitt 7.6.1.1) eine dieser beiden Markierungen. Dem entsprechend weisen alle seine Nachfolgerknoten den Zustand NS = `NOT_ACTIVATED` auf (vgl. Satz 4-1). Die Statusvoraussetzungen für die Anwendung der *parallelInsert*-Operation sind demzufolge ebenfalls erfüllt. Die Anwendung der *insertBetweenNodeSets*-Operation ist somit möglich, wenn die Menge M_{after} nur Knoten umfaßt, die als NS = `NOT_ACTIVATED` oder NS = `ACTIVATED` markiert sind. Dadurch ist sichergestellt, daß der neue Aktivitätenknoten nicht vor laufenden oder bereits beendeten Knoten eingesetzt wird. Wird die neue Aktivität vor einem bereits aktivierten Knoten eingefügt, so wird diese Aktivierung wieder aufgehoben und der betreffende Schritt aus Arbeitslisten entfernt. Für Knoten der Menge M_{before} bestehen dagegen keine Statusbeschränkungen. Insbesondere kann die neue Aktivität auch nach bereits beendeten Knoten eingefügt werden. Die neu eingefügten, von diesen Knoten ausgehenden Kanten werden dann gemäß den ADEPT-Markierungsregeln signalisiert. Befinden sich alle Knoten der Menge M_{before} in einem der Zustände NS = `COMPLETED` oder NS = `SKIPPED`, so ist die neue eingefügte Aktivität sofort ausführbar.

7.6.2 Ad-hoc-Vorwärtssprünge in Ausführungsgraphen

In Kapitel 6 haben wir gezeigt, wie sich mit den Konstrukten des ADEPT-Basismodells Vorwärtssprünge (Shortcuts) unterschiedlicher Semantik vormodellieren lassen. Im allgemeinen ist es nicht möglich, alle während eines Arbeitsablaufs notwendig werdenden Abweichungen a priori im WF-Modell zu berücksichtigen. Zum einen gibt es in der Praxis Ausnahmensituationen, die schlichtweg nicht vorhersehbar sind [RHD98, StMi95, Such87], zum anderen wäre es für viele Anwendungsbereiche zu aufwendig und teuer, alle denkbaren Abweichungen auf Modellebene vorzusehen. Ein adaptives WfMS muß es deshalb berechtigten Akteuren erlauben, Abweichungen vom modellierten Ablauf auch spontan, d.h. während der WF-Ausführung, vorzunehmen. In diesem Abschnitt zeigen wir am Beispiel von Ad-hoc-Vorwärtssprüngen, wie diese Forderung in ADEPT$_{flex}$ umgesetzt wird. Die dabei vorgestellten Sprungoperationen sollen die in Kapitel 6 eingeführten Modellierungskonzepte nicht ersetzen, sondern bilden eine sinnvolle und notwendige Ergänzung für die Behandlung von Vorwärtssprüngen in adaptiven WfMS. Generell stellen Vorwärtssprünge schwerwiegende Eingriffe in die Kontrolle des WF dar, da sie aufwendige Graphrestrukturierungen und Anpassungen des zugrundeliegenden DF-Schemas nach sich ziehen können. Grund hierfür ist, daß durch das Vorziehen von Aktivitäten und dem damit verbundenen Überspringen von Schritten die Versorgung von Aktivitäteneingaben nicht mehr in jedem Fall gesichert ist.

Unabhängig von der konkreten Semantik einer Sprungoperation können innerhalb von Ausführungsgraphen ausschließlich Aktivitäten des Typs NT = `ACTIVITY`, die sich im Status `NOT_ACTIVATED` befinden, das Ziel eines Vorwärtssprungs sein. Ein Vorwärtssprung zu bereits aktivierten, laufenden, beendeten oder nicht mehr ausführbaren Aktivitäten ist dagegen nicht sinnvoll und wird deshalb auch nicht erlaubt. Soll eine Aktivität n_{target} mit Status `NOT_ACTIVATED` vorzeitig ausgeführt werden, so muß angegeben werden, wie mit noch nicht beendeten Vorgängerknoten von n_{target} zu verfahren ist:

$$N_{skip} := \{n \in N \mid n \in pred^{*}(n_{target}) \text{ und } NS(n) \in \{\texttt{NOT_ACTIVATED}, \texttt{ACTIVATED}, \texttt{RUNNING}\}\}$$

Wie in Kapitel 6 unterscheiden wir auch für den dynamischen Fall zwischen Vorwärtssprüngen mit und solchen ohne Nachholen der übersprungenen Schritte. Auch die hybride Anwendung dieser beiden Konzepte ist möglich, wird hier aber nicht weiter behandelt. Zur Vermeidung von Sprüngen in bedingte Verzweigungen oder Parallelverzweigungen mit finaler Auswahl hinein fordern wir für das

Sprungziel n_{target} zusätzlich, daß es deterministisch erreichbar sein muß. Dies ist genau dann der Fall, wenn die Menge N_{skip} keine OR-Split-Knoten enthält, deren zugehöriger OR-Join-Knoten nicht auch in N_{skip} liegt. Auf WF-Instanzebene werden dem Benutzer nach Auswahl der Sprungoperation alle im aktuellen WF-Status anspringbaren Aktivitäten aufgelistet. Er wählt dann einen dieser Schritte als Sprungziel aus, woraufhin ihm das WfMS Vorschläge zur Behandlung von Aktivitäten der Menge N_{skip} (Auslassen, Nachholen) unterbreitet. Die angebotenen Möglichkeiten sind abhängig von Eigenschaften dieser Aktivitäten (Darf die Aktivität gelöscht werden? Besitzt sie nicht nachforderbare obligate Ausgaben) und den Rechten des Benutzers. Nachdem er sich für eine Behandlungsstrategie entschieden hat, wird er vom WfMS über eventuelle Folgen seines Handelns (auszulassende und abzubrechende Aktivitäten) informiert und ggf. aufgefordert, begleitende Änderungen des DF-Modells festzulegen. Anschließend werden die Änderungen auf den Ausführungsgraphen angewendet, woraufhin die Sprungaktivität ausführbar wird.

7.6.2.1 Dynamisches Überspringen von Aktivitäten ohne Nachholen (Auslassen)

Eine Möglichkeit zur Behandlung übersprungener Aktivitäten besteht darin, ihre Bearbeitung auszulassen. Für diesen Fall kann die Sprungoperation auf die in Abschnitt 7.4 vorgestellten Basisoperationen *deleteActivity* und *deleteBlock* zurückgeführt werden. Die Menge der zu löschenden Aktivitäten ist durch N_{skip} (siehe oben) festgelegt, wobei verschiedene Aspekte beachtet werden müssen:

Bereits gestartete Aktivitäten (NS = `RUNNING`) können nicht sofort gelöscht werden, sondern müssen zuvor mit dem Einverständnis ihrer Akteure abgebrochen werden. Für komplexe Aktivitäten bedeutet das, daß die zugehörige Sub-WF-Instanz abgebrochen und zurückgesetzt werden muß. Insgesamt ist die Menge der abzubrechenden Aktivitäteninstanzen festgelegt durch

$$N_{abort} := \{n \in N_{skip} \mid NS(n) = \texttt{RUNNING}\} \subseteq N_{skip}$$

Enthält die Menge N_{skip} der zu überspringenden Aktivitäten einen Schleifenendknoten L_E, dessen zugehöriger Schleifenanfangsknoten nicht in N_{skip} liegt (→ Vorwärtssprung aus einer Schleife heraus), so wird der Knoten L_E nicht aus dem Ausführungsgraphen gelöscht, sondern über ein besonderes Attribut so eingestellt, daß er ohne Auswertung der Schleifenbedingung den Wert *False* in das Datenelement *LOOP* schreibt. Dadurch werden weitere Schleifeniterationen vermieden, das Schleifenkonstrukt bleibt aber erhalten.

Zur Durchführung eines Vorwärtssprungs (mit Auslassen der übersprungenen Schritte) wird zunächst eine Kopie des Ausführungsgraphen angelegt. Darauf basierend, wird der Abbruch laufender Aktivitäten aus der Menge N_{abort} simuliert und es werden die Aktivitäten der Menge N_{skip}, unter Beachtung der oben genannten Restriktionen, aus dem Graphen gelöscht. Dies erfolgt durch einmalige oder mehrfache Anwendung der Basisoperationen *deleteActivity* und/oder *deleteBlock*. Die *deleteBlock*-Operation wird z.B. herangezogen, wenn der zu überspringende Bereich abgeschlossene Kontrollblökke (z.B. Schleifen) enthält. Bei Anwendung der Operationen wird geprüft, ob begleitende Änderungen des DF-Modells vorgenommen werden müssen. Sie werden genau dann erforderlich, wenn es in N_{skip} Aktivitäten gibt, deren datenabhängige Schritte nicht vollständig in dieser Knotenmenge enthalten sind. Für diesen Fall muß der Anwender festlegen, welche Maßnahmen zur Nachversorgung der fehlenden Daten ergriffen werden sollen bzw. ob datenabhängige Schritte – soweit sinnvoll und möglich – kaskadierend gelöscht werden sollen. (Beispielsweise kann es Knoten geben, die als nicht löschbar gekennzeichnet sind. Auch das Sprungziel darf infolge eines solchen kaskadierenden Löschens nicht entfernt werden.) Dabei wird er – wie in Abschnitt 7.4.1.3 beschrieben – durch das System unterstützt, wobei z.B. die Aktivierung benötigter Nachforderungsdienste in bestimmten Fällen auch automatisch, unter Ausnutzung der Parameterbeschreibungen von Aktivitäten, erfolgen kann. Nachdem alle zur Durchführung des Vorwärtssprung notwendigen Änderungen feststehen, werden sie schließlich auf den Original-Ausführungsgraphen angewendet.

Beispiel 7-13 (Vorwärtssprung mit Auslassen)

Ein einfaches Beispiel, bei dem Datenflußaspekte ausgeklammert sind, zeigt Abbildung 7-18. Hier soll in dem links dargestellten Ausführungsgraphen die Bearbeitung von Aktivität `D` vorgezogen werden, wobei die noch nicht beendeten Vorgängerknoten von `D` (N_{skip} = {`B`, `C`}) ausgelassen werden sollen. Dazu wird zunächst `B` abgebrochen. Anschließend werden `B` und `C` mittels *deleteActivity* aus dem Graphen gelöscht, wobei `C` durch eine

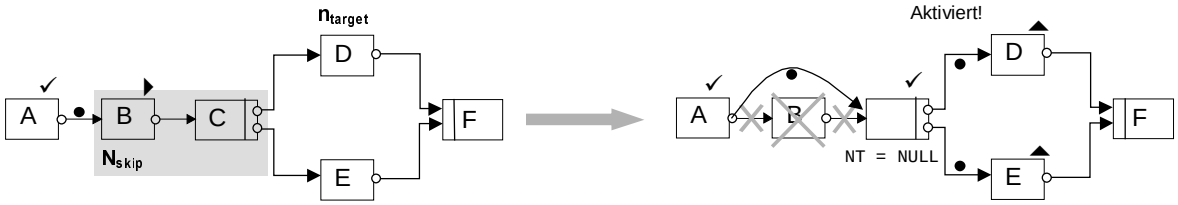


Abbildung 7-18: *Dynamischer Vorwärtssprung mit Auslassen (Kontrollflußsicht)*

Nullaktivität ersetzt wird (vgl. Abschnitt 7.4.1.1). Nach der Neubewertung der Zustandsmarkierungen ergibt sich der rechts dargestellte Ausführungsgraph.

7.6.2.2 Dynamisches Überspringen von Aktivitäten mit Nachholen

Das Auslassen bzw. Löschen übersprungener Aktivitäten stellt nicht immer eine befriedigende Lösung dar. Vielfach besteht der Wunsch, die übersprungenen Schritte parallel zur vorgezogenen Aktivität n_{target} weiter zu bearbeiten (Überspringen mit Nachholen). Je nach Situation kann es erforderlich sein, daß die Knoten aus N_{skip} sofort, d.h. vor den direkten Nachfolgern des Knotens n_{target}, oder aber bis zu einem späteren Punkt $n_{sync} \in succ^*(n_{target})$ im Ablauf nachgeholt werden sollen. In letzterem Fall wird nicht nur die Sprungaktivität n_{target} selbst, sondern eine ganzer Kontrollblock vorzeitig bearbeitbar. Für die Durchführung solcher Vorwärtssprünge stellt ADEPT$_{flex}$ die *jumpForward*-Operation bereit. Bevor wir auf die mit der Anwendung dieser Operation verknüpften Graphtransformationen eingehen, definieren wir zuerst einige Einschränkungen. Sie sollen sicherstellen, daß im Anschluß an den Vorwärtssprung wieder ein konsistenter Ausführungsgraph resultiert (n_{target} bezeichne das Sprungziel):

- Der Knoten n_{target} (NT = `ACTIVITY`) muß die Zustandsmarkierung `NOT_ACTIVATED` besitzen.
- Aufgrund ihrer unklaren Semantik wollen wir Vorwärtssprünge in bedingte Verzweigungen, in Parallelverzweigungen mit finaler Auswahl oder in Schleifen hinein ausschließen. Zu diesem Zweck fordern wir, daß die Menge N_{skip} keine Split-Knoten und keine Schleifenanfangsknoten enthalten darf, deren zugehöriger OR-Join-Knoten bzw. Schleifenendknoten nicht in $N_{skip} \cup \{n_{target}\}$ enthalten ist.
- Der Knoten n_{sync} muß ein Nachfolger von n_{target} bzgl. normaler Kontrollkanten sein: $n_{sync} \in c_succ^*(n_{target})$
- Der durch die Knotenmenge $N_{jump} := \{n_{target}, n_{sync}\} \cup (succ^*(n_{target}) \cap pred^*(n_{sync}))$ induzierte Teilgraph von bildet einen Kontrollblock (vgl. Definition 3-9)

Werden diese Bedingungen erfüllt und bleibt die Korrektheit des zugrundeliegenden DF-Schemas erhalten (siehe unten), so kann der gewünschte Vorwärtssprung gemäß der nun vorgestellten *jumpForward*-Operation erfolgen (Kontrollflußsicht):

jumpForward ($CFS_{instance}$, n_{target}, n_{sync}, ...)
input
...
n_{target}: Ziel des Vorwärtssprungs
n_{sync}: Knoten, vor dessen Aktivierung alle übersprungenen Aktivitäten nachgeholt sein müssen
...
begin
// Bestimmung der letzten im aktuellen WF-Status beendeten Knoten, die vor dem Sprungziel
// liegen.
$N_{completed} := \{n \in pred^*(n_{target}) \mid NS(n) = \texttt{COMPLETED} \wedge (\forall\, n^* \in succ(n): NS(n^*) \neq \texttt{COMPLETED}\ )\}$
// Bestimmung aller im aktuellen WF-Status aktivierten oder laufenden Knoten, die vor dem
// Sprungziel liegen.
$N_{active} := \{n \in c_pred^*(n_{target}) \mid NS(n) \in \{\texttt{ACTIVATED}, \texttt{RUNNING}\}\ \}$
// Bestimmung des minimalen Kontrollblocks, der alle Knoten der Mengen $N_{completed}$ und N_{active}
// sowie den Knoten n_{sync} umschließt.
$(n_{begin}, n_{end}) := MinBlock^{CFS}(N_{completed} \cup N_{active} \cup \{n_{sync}\})$

```
// Füge direkt vor n_target eine Nullaktivität n_1 ein, die die Eingangskontrollkanten und die
// Eingangssemantik dieses Knotens übernimmt und mit ihm über eine ausgehende Kontroll-
// kante verknüpft wird.
Einfügen von n1 (wie im Kommentar beschrieben)

// Füge direkt vor n_sync eine Nullaktivität n_2 ein, die die Eingangskontrollkanten und die
// Eingangssemantik dieses Knotens übernimmt ...
Einfügen von n2 (wie im Kommentar beschrieben)

// Entferne den vorzeitig auszuführenden Kontrollblock (n_target, n_2) – den sog. Nachholbereich –
// aus seinem bisherigen Kontext und füge ihn parallel zum Block (n_begin, n_end) ein. Hierzu kann
// die in Abschnitt 7.5.1 erwähnte Basisoperation moveBlock verwendet werden
moveBlock(CFS_instance, .., (n_target, n2), PARALLEL TO (n_begin, n_end), ...)

// Den Nachholbereich mit seinem ehemaligen Nachfolgerknoten n_sync über eine einfache
// Sync-Kante synchronisieren.
insertSyncEdge(CFS_instance, n2, n_sync, SOFT_SYNC_E)

// Bestmögliche Datenversorgung des vorgezogenen Bereichs über einfache Sync-Kanten sichern.
// Dazu wird von jedem Knoten aus N_completed aus eine einfach Sync-Kante zum Startknoten n_target
// des Nachholbereichs (Zielknoten des Vorwärtssprungs) eingefügt.
forall (n ∈ N_completed) do
    insertSyncEdge(CFS_instance, n, n_target, SOFT_SYNC_E)
done

// Vom Nachholbereich ausgehende Fehlerkanten, die die Regel KF-9 verletzen,
// entfernen (im Fehlerfall wird dann keine automatischer Rücksprung durchgeführt)
forall (n ∈ {n_target} ∪ (succ*(n_target) ∩ pred*(n_sync)) do
    forall (e ∈ E with ET^e = FAILURE_E and Id_source^e = n and Id_dest^e ∉ pred*(n)) do
        deleteFailureEdge(CFS_instance, Id_source^e, Id_dest^e)
    done
done
end // jumpForward
```

Nach Durchführung dieser Schritte werden die Markierungen von Knoten und Kanten entsprechend den in Kapitel 4 vorgestellten Regeln neu bewertet. Der Status bereits aktivierter bzw. gestarteter Aktivitäten wird dadurch nicht geändert. Die Zielaktivität des Sprungs selbst ist anschließend aktiviert, d.h. sie erscheint auf den Arbeitslisten und kann somit sofort durchgeführt werden.

Beispiel 7-14 (Vorwärtssprung mit Nachholen)

Gegeben sei der Ausführungsgraph aus Abbildung 7-19 a). Wir nehmen an, daß ein berechtigter Benutzer infolge einer Ausnahmesituation die Aktivität G vorzeitig bearbeiten und anschließend mit der Kontrolle an diesem Knoten fortfahren möchte. Die noch laufenden bzw. noch nicht bearbeiteten Vorgängeraktivitäten von G – die Schritte D, E und F – sollen nebenläufig weiter bearbeitet werden, spätestens jedoch vor Aktivierung des Knotens J abgeschlossen sein. Diesem Änderungswunsch kann durch Anwendung der *jumpForward*-Operation mit $n_{target} = G$ und $n_{sync} = J$ entsprochen werden. Abbildung 7-19 b) zeigt den Ausführungsgraphen nach Einfügen der beiden Nullaktivitäten n_1 und n_2. Hiervon ausgehend wird der durch (n_{target}, n_2) gebildete Verzweigungskontrollblock, dessen Bearbeitung vorgezogen werden soll, aus seinem bisherigen Kontext gelöst und parallel zu dem durch (A, n_{sync}) gebildeten Block wieder in den Graphen eingefügt. Grundlage hierfür bildet die *moveBlock*-Basisoperation, bei deren Anwendung u.a. leere AND-Split- und AND-Join-Knoten in den Graphen eingefügt werden. Im Anschluß daran werden die einfachen Sync-Kanten C → G und $n_2 \rightarrow n_{sync}$ neu hinzugefügt: Durch Hinzunahme von C → G erreichen wir, daß G weiterhin lesend auf die von den Aktivitäten A, B und C geschriebenen Datenelemente zugreifen darf (falls notwendig). Das Einfügen von $n_2 \rightarrow n_{sync}$ stellt sicher, daß die Knoten des Nachholbereichs wie bisher vor J beendet werden. Nach der Neubewertung der Knoten- und Kantenmarkierungen erhalten wir den in Abbildung 7-19 c) dargestellten Ausführungsgraphen, bei dem die ursprüngliche Sprungaktivität G nun aktiviert ist. Der dargestellte Graph kann durch Anwendung der in Abbildung 7-2 aufgeführten Regeln wieder vereinfacht werden. Wir verzichten an dieser Stelle auf eine Darstellung. Interessant an dem vorliegenden Beispiel ist, daß durch die Anwendung der Sprungoperation Aktivitäten bzw. Kontrollblöcke parallel zueinander ausgeführt werden, die zuvor sequentiell angeordnet waren.

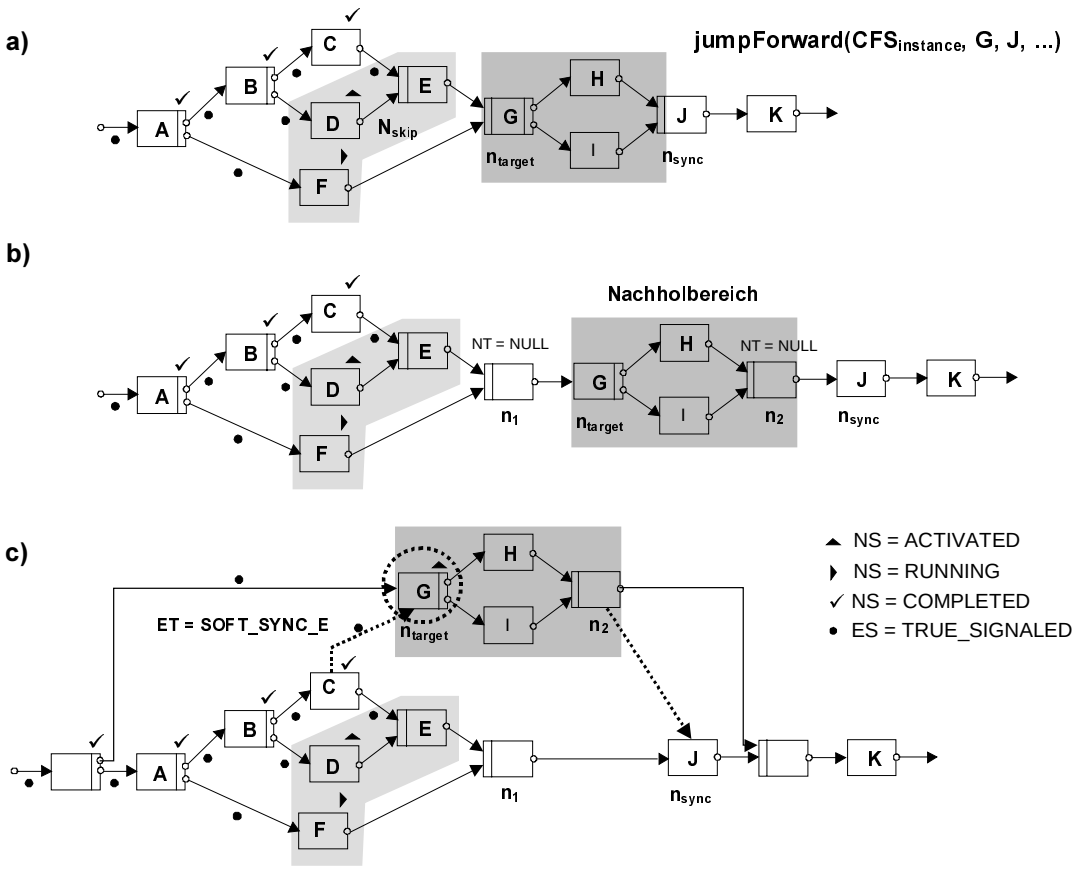


Abbildung 7-19: *Dynamischer Vorwärtssprung mit Nachholen (Kontrollflußsicht)*

Wie das Beispiel 7-14 zeigt, sind nach dem Vorwärtssprung die Aktivitäten des Nachholbereichs (`G`, `H` und `I` im Beispiel) früher als vom Modellierer vorgesehen bearbeitbar. Hinzu kommt, daß sie nun parallel zu bisher vorausgehenden Schritten ausgeführt werden. Dadurch können potentiell Dateninkonsistenzen resultieren, die es zu vermeiden gilt. Zum einen kann es zu parallelen Schreibzugriffen auf Datenelementen durch Knoten des Nachholbereichs und des dazu parallelen Kontrollblocks kommen (Verletzung von Regel DF-2), zum anderen können Knoten des Nachholbereichs durch die vorzeitige Ausführung obligate Eingabedaten fehlen (Verletzung von Regel DF-1). Beide Fälle müssen wieder durch entsprechende DF-Analysen ausgeschlossen werden. Wird dabei z.B. festgestellt, daß einzelnen Aktivitäten infolge des Vorziehens Daten fehlen, so kann die Parameterversorgung wieder durch Vorschalten entsprechender Nachforderungsdienste sichergestellt werden, falls die fehlenden Parameterdaten nachforderbar sind. Bei speziell nachforderbaren Parametern kann die Datenversorgung über zusätzlich einzufügende Schritte erfolgen. Wichtig ist wieder, daß alle Änderungen innerhalb einer Transaktion auf den Ausführungsgraphen angewendet werden, so daß diese entweder vollständig oder gar nicht in die WF-Instanz eingebracht werden. Können z.B. nicht alle obligaten Lesezugriffe versorgt werden, muß die Sprungoperation abgebrochen werden.

7.7 Änderungstransaktionen

Um die von ADEPT geforderten Korrektheits- und Konsistenzeigenschaften gewährleisten zu können, müssen ggf. mehrere Änderungsoperationen logisch zusammenhängend und atomar ausgeführt werden. Ein Beispiel ist das Löschen einer Aktivität, zusammen mit den von ihr datenabhängigen Schritten. Sie kann durch mehrfache Anwendung der Basisänderungsoperation *deleteActivity* realisiert werden. Wir benötigen also ein Konstrukt, ähnlich den Transaktionen in konventionellen Datenbanksystemen. Mit diesem Konstrukt – im folgenden *Änderungstransaktion* (kurz: ÄT) genannt – soll gewährleistet werden, daß entweder alle Operationen einer Änderung ausgeführt werden oder gar keine. Jede ÄT soll eine WF-Instanz, ausgehend von einem korrekten Zustand, wieder in einen neuen, ebenfalls korrekten Zustand überführen. Zwischenzustände, die sich nach Anwendung einzelner Änderungs-

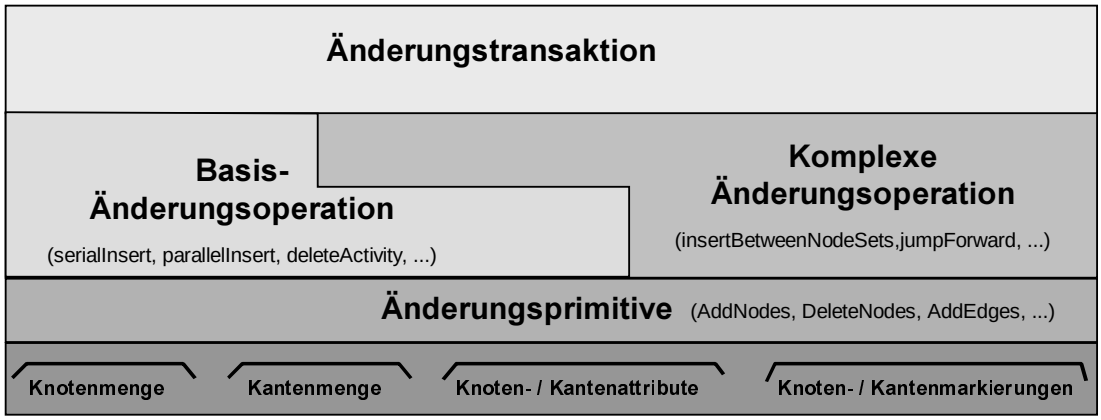


Abbildung 7-20: *Änderungstransaktionen, -operationen und -primitive und WF-Instanzdaten*

operationen der ÄT (bzw. der ihnen zugrundeliegenden Änderungsprimitive) ergeben, sollen für andere Änderungstransaktionen nicht sichtbar sein. Zwei Modelle für Änderungstransaktionen liegen nahe:

- **Einstufiges Transaktionskonzept**: Hier besteht eine ÄT aus einer Menge von Änderungsprimitiven (*AddNodes*, *DeleteNodes*, *AddEdges*, ...). Bei der Festlegung von Änderungen werden die semantisch höheren Änderungsoperationen sofort in entsprechende Änderungsprimitve aufgelöst und so, geklammert in *BeginOfTransaction* und *Commit* bzw. *Abort*, an die Instanzverwaltungskomponente weitergegeben. Änderungsoperationen werden bei diesem Ansatz einfach als mit Anwendungslogik angereicherte Makros von Änderungsprimitiven verstanden. Bei der direkten Übersetzung einer ÄT in Änderungsprimitive geht allerdings die Information, welche Änderungsoperationen konkret ausgeführt wurden, im allgemeinen verloren. Diese Information kann aber dazu dienen, für bestimmte Funktionen eines adaptiven WfMS (Synchronisation konkurrierender Änderungen, Rücknahme von Änderungen), verbesserte Verfahren anbieten zu können.
- **Zweistufiges Transaktionskonzept:** Hier kann eine ÄT aus mehreren, zusammengehörigen Änderungsoperationen bestehen, die ihrerseits auf der Grundlage einer Menge von Änderungsprimitiven realisiert werden. Dieses zweistufige Konzept hat den Vorteil, daß bei der Einbringung der Änderung bekannt ist, welche semantisch höheren Änderungsoperationen durchgeführt werden, d.h. in welchem Kontext Änderungsprimitve ausgeführt werden sollen. Dadurch können z.B. Synchronisationsverfahren effizienter gemacht und Konflikte umgangen werden [Weil97]. Da die Anwendung einzelner Änderungsoperationen keine Datenkonsistenz garantiert, ist es nicht erlaubt, die Wirkungen einzelner Änderungsoperationen im Erfolgsfall nach außen, d.h. für andere Änderungstransaktionen, sichtbar zu machen, bevor die umgebende ÄT beendet wird (vgl. hierzu offen-geschachtelte Transaktionen oder Multi-Level-Transaktionen [Elma92, GrRe93]).

In ADEPT wird das zweistufige Transaktionskonzept angewendet. Wie Transaktionen in traditionellen Datenbanksystemen, sollen auch Änderungstransaktionen dem ACID-Paradigma folgen. Dem entsprechend definieren wir eine Änderungstransaktion als eine Folge zusammengehörender Änderungsoperationen, die den folgenden Kriterien genügen:

- Die Änderungsoperationen der ÄT werden entweder vollständig oder gar nicht in die WF-Instanz eingebracht (Atomarität).
- Die ÄT überführt die WF-Instanz von einem konsistenten in einen neuen konsistenten Zustand (Konsistenz), d.h. nach der Durchführung der ÄT resultiert wieder eine WF-Instanz mit korrektem WF-Graphen und konsistenten Zustandsmarkierungen.
- Die ÄT kapselt die einzelnen Änderungsoperationen gegenüber anderen ÄT (Isolation).
- Wurde die ÄT einmal erfolgreich in die WF-Instanz eingebracht, so ist sie dauerhaft gespeichert (Dauerhaftigkeit). Soll die Änderung später rückgängig gemacht werden (temporäre Änderungen), so muß dies explizit veranlaßt werden (siehe hierzu Kapitel 8).

Wir kommen auf Änderungstransaktionen in Kapitel 8 zurück. Abbildung 7-20 zeigt nochmals die wesentlichen Zusammenhänge von Änderungstransaktionen, -operationen und -primitiven.

7.8 Zusammenfassung

In diesem Kapitel haben wir die Änderungsoperationen des ADEPT$_{flex}$-Modells behandelt. Sie eignen sich gleichermaßen gut für die Anpassung von WF-Vorlagen und die dynamische Modifikation von sich in Ausführung befindlichen WF-Instanzen (z.B. Durchführung von Ad-hoc-Abweichungen). Der vorgestellte Ansatz bildet daher eine solide Grundlage für die Verwirklichung verschiedener Arten von Änderungen.

Am Beispiel des Einfügens und des Löschens von Aktivitäten und Kontrollblöcken haben wir gezeigt, welche Aspekte dabei von besonderer Relevanz sind. Um eine klare und eindeutige Änderungssemantik zu erzielen und um eine saubere Implementierungsgrundlage zu schaffen, bilden wir jede Basisänderungsoperation auf eine Menge von wohldefinierten Änderungsprimitiven ab. Dabei handelt es sich um einfache, mengenbasierte Operationen zur Manipulation der Knoten- und Kantenmengen (inkl. Attributen) eines WF-(Ausführungs-)Graphen. Um nach der Durchführung einer Graphmodifikation die geforderte Korrektheit und Konsistenz zusichern zu können, haben wir für die einzelnen Änderungsoperationen formale Vor- und Nachbedingungen definiert. Sie betreffen strukturelle Modellaspekte und – im Fall von Instanzänderungen – die Zustandsmarkierungen des WF-Ausführungsgraphen. Durch die Vorgabe geeigneter Vorbedingungen können bestimmte Modelleigenschaften, wie regelmäßige Blockstrukturierung oder Verklemmungsfreiheit, bereits per Konstruktion, d.h. ohne wietere Überprüfungen, erzielt werden. Des weiteren wird bei der Anwendung von Änderungsoperationen auf den Ausführungsgraphen einer WF-Instanz dafür gesorgt, daß anschließend wieder konsistente Zustandsmarkierungen resultieren, ohne daß der Anwender entsprechende Anpassungen selbst vornehmen muß. Die Grundlage hierfür bilden die Markierungs- und Ausführungsregeln des ADEPT-Basismodells, die unabhängig vom Kontext der Änderung anwendbar sind. Schließlich haben wir gezeigt, wie zu verfahren ist, wenn nach der Anwendung einer Änderungsoperation die geforderten Nachbedingungen, etwa die Korrektheit des Datenflußmodells, (noch) nicht erfüllt sind. Im allgemeinen macht es hier wenig Sinn, eine feste Behandlungsstrategie vorzugeben. Statt dessen haben wir, abhängig von der Art der Änderung, verschiedene Strategien aufgezeigt, mit denen sich die geforderte Korrektheit und Konsistenz aufrechterhalten lassen. Sie können dazu genutzt werden, Anwendern die Folgen einer beabsichtigten Modifikation aufzuzeigen und sie bei der Durchführung notwendiger Begleitänderungen angemessen zu unterstützen.

Neben der geforderten Korrektheit und Konsistenz werden die ADEPT$_{flex}$-Operationen auch den anderen, eingangs dieses Kapitels genannten Entwurfszielen gerecht. Insbesondere ist die angebotene Menge an Basisoperationen *vollständig*. Für zwei korrekt strukturierte WF-Modelle w_1 und w_2 bedeutet dies, daß w_1 durch Anwendung einer endlichen Folge von Basisoperationen in w_2 überführt werden kann. (Wir verzichten an dieser Stelle auf einen Beweis dieser Eigenschaft.) Wie gezeigt, stellt sich der Begriff der Vollständigkeit auf Instanzebene etwas differenzierter dar, da hier die Anwendbarkeit von Änderungsoperationen zusätzlich durch den aktuellen WF-Status eingeschränkt wird. In Bezug auf die erwähnte Minimalitätseigenschaft haben wir Kompromisse getroffen, um häufig benötigte Kernfunktionen, wie das Löschen von Aktivitäten, effizient implementieren zu können. Schließlich haben wir am Beispiel einer komplexen Einfügeoperation und verschiedener Sprungoperationen gezeigt, wie sich auf der Grundlage der ADEPT$_{flex}$-Basisoperationen semantisch höherwertige Änderungsoperationen verwirklichen lassen. Dadurch können komplexe Änderungen auf Benutzerebene kompakt und komfortabel festgelegt werden.

Insgesamt bildet das ADEPT-Basismodell eine ausgezeichnete Grundlage für die (dynamische) Änderung von Workflows zur Laufzeit. Durch die gewählte Blockstruktur, durch die Unterscheidbarkeit verschiedener Knoten- und Kantentypen, durch die explizite Festlegung von Modellaspekten und ihrer Korrektheitseigenschaften, durch den gewählten Markierungsansatz, und vieles mehr, lassen sich Änderungen auf semantisch hoher Ebene und mit der geforderten Zuverlässigkeit und Flexibilität umsetzen. Darüber hinaus können durch Ausnutzung von Struktureigenschaften, die erforderlichen Analysen (z.B. zur Überprüfung der Korrektheit des Datenflusses) gezielt eingegrenzt werden, was für die effiziente Einbringung von Änderungen ebenfalls sehr wichtig ist.

8 Permanente und temporäre WF-Instanzänderungen

Ein Vorzug des ADEPT-Basismodells besteht darin, daß Rücksprünge unterschiedlicher Semantik unterstützt werden. Konkret wird zwischen Schleifen- und Fehlerrücksprüngen differenziert. Beide können bei der Modellierung berücksichtigt werden. Darüber hinaus sind Fehlerrücksprünge auch spontan auf WF-Instanzen anwendbar. In beiden Fällen wird derselbe Bereich eines Ausführungsgraphen mehrfach durchlaufen, woraus sich einige interessante Fragestellungen ergeben.

8.1 Motivation und Problemstellung

Sowohl für die Beschreibung von Schleifen- als auch von Fehlerrücksprüngen stellt das ADEPT-Basismodell explizite Konstrukte bereit. Dadurch verbessert sich nicht nur die Lesbarkeit der WF-Modelle, sondern es wird unmittelbar erkennbar, ob ein auftretender Zyklus die Folge einer fehlerhaften Modellierung ist oder ob er einen Schleifen- bzw. Fehlerrücksprung beschreibt. Unerwünschte Zyklen, die zur Laufzeit zu Verklemmungen führen können, werden bereits bei der Erstellung bzw. Änderung von WF-Modellen ausgeschlossen. Darüber hinaus ist bei der Neumarkierung einer Schleifen- bzw. Fehlerkante für das WfMS sofort erkennbar, daß im Ausführungsgraph ein Rücksprung erfolgt, d.h. ein zuvor bereits durchlaufener Bereich des WF-Graphen erneut betreten wird. Abhängig von der Semantik des Rücksprungs erfolgen dann verschiedene Maßnahmen (z.B. Neumarkierung von Knoten und Kanten des Rücksprungbereichs), um im Kontrollfluß fortfahren zu können. Sollen Rücksprünge und dynamische Änderungen durch dasselbe System unterstützt werden, ergeben sich interessante Fragestellungen. So muß bei der Änderung des Körpers einer sich in Ausführung befindlichen Schleife entschieden werden, ob die Modifikation nur für die aktuelle Iteration oder auch für Folgeiterationen erhalten bleiben soll. Ersteres trifft man in der Praxis häufig an, etwa wenn situationsbedingt Ad-hoc-Abweichungen vom geplanten Ablauf vorgenommen werden, die später, aufgrund des Wegfalls der Ausnahme, nicht mehr erforderlich sind (sog. *temporäre Änderung*). Dem gegenüber stehen Änderungen, die für die Restdauer der WF-Instanz erhalten bleiben sollen (sog. *permanente Änderung*).

Beispiel 8-1 (Anwendungsbeispiel):

In der Medizin gibt es langandauernde Behandlungsprozesse, die in mehrere Zyklen unterteilt sind [RHD98]. Die einzelnen Behandlungszyklen, die ihrerseits eine komplexe Ablaufstruktur aufweisen können, laufen im Normalfall identisch ab. In Notfallsituationen, z.B. bei Verschlechterung des Zustandes des Patienten, oder sonstigen Ausnahmen kann es erforderlich werden, bei der Durchführung des aktuellen Behandlungszyklus vom geplanten Ablauf abzuweichen. Fällt die betreffende Ausnahme in der Folge wieder weg, können die restlichen Behandlungszyklen wie geplant abgewickelt werden. Es kann aber auch erforderlich werden, für einen bestimmten Patienten bzw. seinen Behandlungsprozeß das ansonsten übliche Standardprocedere dauerhaft abzuändern, etwa wenn er eine Medikamentenallergie besitzt. In einem solchen Fall müssen die festgelegten Änderungen bis zur Beendigung des Behandlungsprozesses, ggf. also auch für mehrere Zyklen, erhalten bleiben. – Ein entsprechendes Beispiel, bei dem beide Änderungsarten gemeinsam auftreten, zeigt Abbildung 8-1.

Die meisten der in der Literatur diskutierten Ansätze für adaptives WF-Management (vgl. Kapitel 10) unterstützen ausschließlich permanente WF-Instanzänderungen. Dadurch wird ihre praktische Anwendbarkeit, insbesondere im Hinblick auf die Behandlung von Ausnahmen, erheblich eingeschränkt. Idealerweise sollte ein adaptives WfMS beide Änderungsarten zulassen, d.h. WF-Änderungen sollten – je nach Situation – permanent oder temporär (z.B. nur für eine Schleifeniteration) auf WF-Instanzen angewendet werden können. Permanente Änderungen sollen nach ihrer Einbringung für die Restdauer der WF-Instanz erhalten bleiben. Bei wiederholtem Durchlaufen des modifizierten Bereichs wird die Änderung dem entsprechend weiter berücksichtigt. Beispielsweise kann ein Schritt, der permanent in einen Schleifenkörper eingefügt wird, mehrmals zur Ausführung kommen[5]. Im Gegensatz dazu werden temporäre Änderungen nur einmalig auf eine bestimmte Region des WF-Ausführungsgraphen angewendet. Wird dieser Bereich später nochmals durchlaufen, etwa infolge eines Schleifenrücksprungs, muß die Änderung zuvor wieder rückgängig gemacht werden, um den gewünschten Effekt zu erzielen.

[5] Permanente Änderungen beziehen sich hier auf die dauerhafte Abänderung einer einzelnen WF-Instanz, sind also nicht zu verwechseln mit WF-Vorlagenänderungen. Allerdings kann das Konzept permanenter Änderungen vom Prinzip her dazu verwendet werden, um Vorlagenänderungen auf bereits laufende Instanzen zu propagieren.

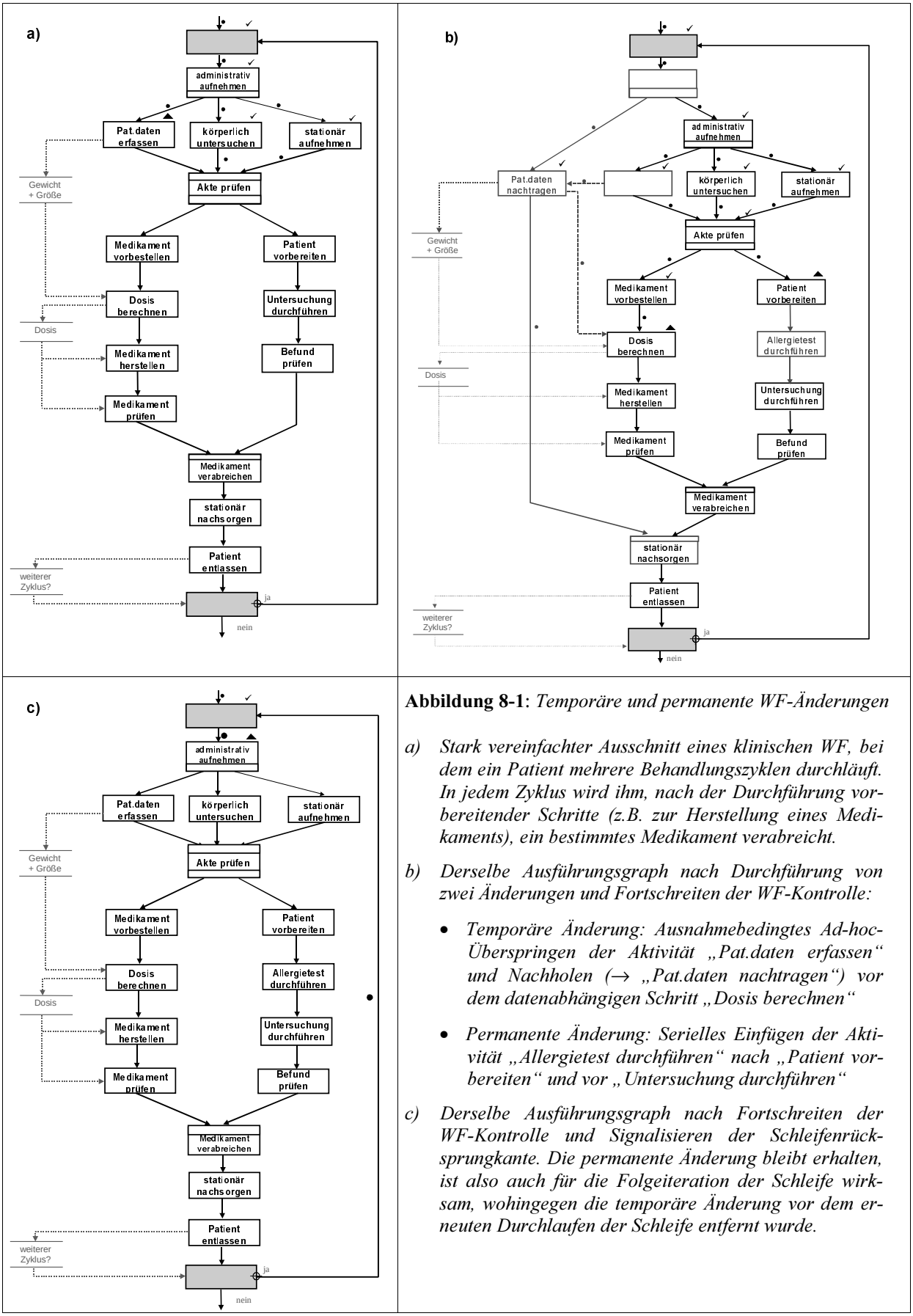


Abbildung 8-1: *Temporäre und permanente WF-Änderungen*

a) Stark vereinfachter Ausschnitt eines klinischen WF, bei dem ein Patient mehrere Behandlungszyklen durchläuft. In jedem Zyklus wird ihm, nach der Durchführung vorbereitender Schritte (z.B. zur Herstellung eines Medikaments), ein bestimmtes Medikament verabreicht.

b) Derselbe Ausführungsgraph nach Durchführung von zwei Änderungen und Fortschreiten der WF-Kontrolle:

- *Temporäre Änderung: Ausnahmebedingtes Ad-hoc-Überspringen der Aktivität „Pat.daten erfassen“ und Nachholen (→ „Pat.daten nachtragen“) vor dem datenabhängigen Schritt „Dosis berechnen“*
- *Permanente Änderung: Serielles Einfügen der Aktivität „Allergietest durchführen“ nach „Patient vorbereiten“ und vor „Untersuchung durchführen“*

c) Derselbe Ausführungsgraph nach Fortschreiten der WF-Kontrolle und Signalisieren der Schleifenrücksprungkante. Die permanente Änderung bleibt erhalten, ist also auch für die Folgeiteration der Schleife wirksam, wohingegen die temporäre Änderung vor dem erneuten Durchlaufen der Schleife entfernt wurde.

Die integrierte Unterstützung dieser Konzepte stellt sich nicht so einfach dar, wie es auf den ersten Blick erscheinen mag, insbesondere wenn auf eine WF-Instanz sowohl temporäre als auch permanente Änderungen angewendet werden sollen. Für diesen Fall benötigen wir Regeln, die festlegen, wann eine Änderung permanent und wann sie nur temporär auf eine gegebene WF-Instanz angewendet werden kann. Die wichtigste Forderung in diesem Zusammenhang ist, daß eine permanente Änderung niemals auf der Grundlage zuvor durchgeführter temporärer Änderungen eingebracht werden darf. Andernfalls können Abhängigkeiten entstehen, die beim späteren Wegfall der temporären Änderung dazu führen, daß Inkonsistenzen bzw. Programmfehler resultieren oder aber die permanente Änderung wieder rückgängig gemacht werden muß. Dies wäre z.B. der Fall, wenn ein permanent eingefügter Prozeßschritt datenabhängig von einer zuvor temporär eingefügten Aktivität ist. Ähnliche Abhängigkeiten können auch zwischen temporären WF-Änderungen entstehen, wenn sie in verschiedenen Bereichen des WF oder innerhalb geschachtelter Schleifen vorgenommen werden. Für alle diese Fälle muß der Anwender davon ausgehen können, daß eine vom WfMS akzeptierte Änderung in der Folge niemals zu Konsistenzproblemen führt. Des weiteren müssen auftretende Probleme ohne aufwendige Benutzerinteraktionen behandelt werden können.

Bevor wir auf Konzepte zur korrekten Durchführung permanenter und temporärer Änderungen eingehen, führen wir in den beiden folgenden Abschnitten zuerst wichtige Grundlagen ein.

8.2 Verwaltung von Änderungen in einer Änderungshistorie

Um (temporäre) Änderungen einer WF-Instanz zu einem späteren Zeitpunkt wieder korrekt zurücknehmen zu können, müssen für sie entsprechende Informationen vorliegen. Zu diesem Zweck verwaltet ADEPT für jede WF-Instanz, zusätzlich zu den bereits erwähnten Kontrolldaten und WF-Daten, eine *Änderungshistorie*. Sie protokolliert alle auf der WF-Instanz durchgeführten Änderungen in der Reihenfolge ihrer Einbringung. Dem entsprechend wird die Historie durch eine geordnete Liste repräsentiert, die initial leer ist. Jeder neu hinzukommende Historieneintrag beschreibt die von einer bestimmten *Änderungstransaktion* (kurz: ÄT) während der WF-Ausführung vorgenommenen Modifikationen. Er besitzt die folgende Struktur (vereinfachte Darstellung):

(*TrTimestamp*, *TrId*, *TrDuration*, *TrInitiator*, *TrStartRegion*, *TrAppliedOps*)

- *TrTimestamp*: Fortlaufende Nummer, im folgenden auch als *Transaktionszeitstempel* bezeichnet, die den Änderungstransaktionen einer Instanz bei ihrer Durchführung zugewiesen wird. Ein solcher Zeitstempel wird bei einer zentralen Stelle reserviert und legt die Reihenfolge der Einbringung von Änderungstransaktionen (bezogen auf die gegebene WF-Instanz) fest. Nebenläufige Änderungstransaktionen werden in eine (logische) Reihenfolge gezwungen (vgl. Kapitel 9).
- *TrId*: Eindeutiger Bezeichner für die Änderungstransaktion.
- *TrDuration* $\in$ {`PERMANENT`, `TEMPORARY`}: Legt fest, ob die Änderung permanent oder temporär auf die WF-Instanz angewendet werden soll.
- *TrInitiator*: Initiator der Änderung.
- *TrStartRegion*: Um entscheiden zu können, ob bei einem Rücksprung die Auswirkungen einer ÄT mit *TrDuration* = `TEMPORARY` unwirksam gemacht werden sollen oder nicht, wird für jede ÄT ein sog. *Startbereich* festgelegt. Er umfaßt eine Teilmenge der Knoten des WF-Ausführungsgraphen mit folgender Semantik: Befinden sich die Knoten des Startbereichs der ÄT im Rücksprungbereich, so müssen alle von dieser temporären ÄT verursachten Änderungen wieder zurückgenommen werden. Der Startbereich einer ÄT wird aus den Startbereichen der zur Anwendung gekommenen Änderungsoperationen abgeleitet. Wie dies genau erfolgt und wie die Startbereiche einzelner Änderungsoperationen definiert sind, werden wir in Abschnitt 8.3 zeigen.
- *TrAppliedOps*: Wie in Kapitel 7 beschrieben, können im Verlauf einer ÄT mehrere Änderungsoperationen (kurz: ÄO) zur Ausführung kommen. Sie werden, entsprechend der Reihenfolge ihrer Einbringung, in einer geordneten Liste *TrAppliedOps* verwaltet. Ein Listeneintrag umfaßt Informationen zur Ausführung einer einzelnen ÄO und besitzt die folgende Struktur:

  (*OpTimestamp*, *OpType*, *OpStartRegion*, *OpAppliedPrimitives*)

- *OpTimestamp*: Alle innerhalb der ÄT durchgeführten Änderungsoperationen werden fortlaufend numeriert, so daß die Reihenfolge ihrer Anwendung auf die WF-Instanz feststeht. Diese eindeutige Positionsnummer bezeichnen wir im folgenden als ÄO-Zeitstempel.
- *OpType*: Legt fest, welche Änderungsoperation (z.B. *insertBetweenNodeSets*) mit welchen Parametern (z.B. M_{before}, M_{after}, ...) angewendet wurde.
- *OpStartRegion*: Der Startbereich einer ÄO ist ähnlich definiert wie für eine ÄT (siehe unten).
- *OpAppliedPrimitives*: Wie in Kapitel 7 beschrieben, wird jede ÄO auf eine Menge von Graphänderungsprimitiven (z.B. Einfügen / Löschen von Knoten und Kanten) abgebildet. Sie werden, zusammen mit ihren Aufrufparametern (z.B. Knotenbezeichnern), ebenfalls protokolliert. Handelt es sich um Primitive, bei denen Graphattribute modifiziert wurden (z.B. *SetNodeAttr*, *SetEdgeAttr*), werden zusätzlich die alten Werte (*Before Images*) gespeichert.

Durch die Verwaltung einer Änderungshistorie kann auch die Benutzerfreundlichkeit des Systems verbessert werden. Auf ihrer Grundlage können Anwender eine durchgeführte Strukturänderung auf Wunsch wieder stornieren. Ein solches *Undo* ist möglich, solange von der Änderung noch kein Auswirkungen auf die WF-Ausführung ausgegangen sind. Dies kann mit einfachen Mitteln, unter Verwendung der Informationen aus der Ablaufhistorie, festgestellt werden. Im folgenden benötigen wir die Änderungshistorie für die automatische Rücknahme temporärer Änderungen.

8.3 Kriterium für die Rücknahme temporärer Änderungen

Ist im Ausführungsgraphen einer WF-Instanz der Bereich, in dem eine temporäre Änderung vorgenommen wurde, einmal abgearbeitet – dies ist der Fall, wenn alle Knoten dieser Region eine der beiden Markierungen `COMPLETED` oder `SKIPPED` besitzen – und soll er anschließend nochmals durchlaufen werden, so müssen zuvor die Auswirkungen der temporären Änderung unwirksam gemacht werden. Bei einem solchen Rücksprung wäre es am einfachsten, alle zuvor eingebrachten temporären Änderungen wieder rückgängig zu machen. Dieser naive Ansatz ist im allgemeinen aber nicht korrekt. Der Grund dafür ist, daß eine WF-Instanz mehrere temporäre Änderungen besitzen kann, die unabhängig voneinander, d.h. durch verschiedene Änderungstransaktionen, in unterschiedlichen Bereichen des Ausführungsgraphen eingebracht worden sind. Erfolgt z.B. in einem Teilzweig einer Parallelverzweigung ein Schleifenrücksprung, so dürfen temporäre Änderungen, die auf andere Teilzweige angewendet wurden, dadurch nicht beeinträchtigt werden. Ebensowenig sollten bei einem Rücksprung temporäre Änderungen entfernt werden, die in einem Bereich des Ausführungsgraphen liegen, der noch nicht betreten worden ist. Das Beispiel aus Abbildung 8-2 verdeutlicht diese beiden Fälle.

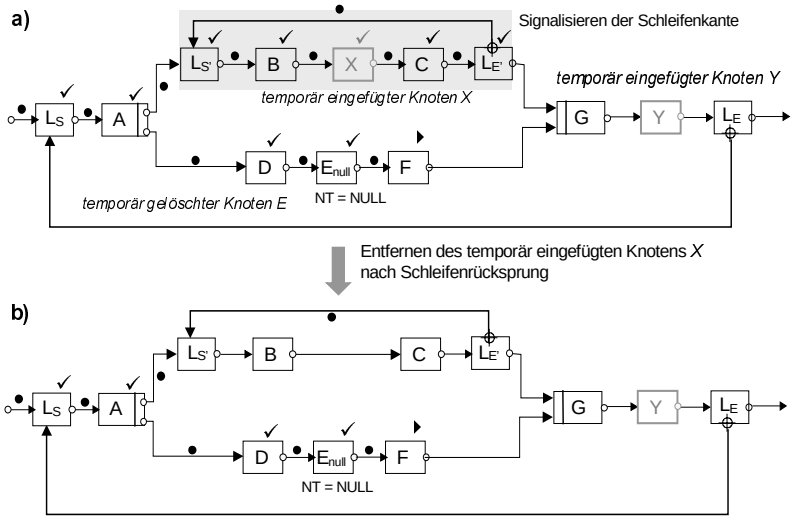


Abbildung 8-2: *Rücknahme temporärer Änderungen bei Parallelverzweigungen. (Beim dem Signalisieren der Schleifenrücksprungkante $L_{E'} \rightarrow L_{S'}$ – vgl. hierzu auch Markierungsregel M_3 – wird die temporäre Einfügung von X zurückgenommen. Die beiden anderen temporären Änderungen mit dem Startbereich $\{E_{null}\}$ (→ Löschen von E) bzw. dem Startbereich {Y} (→ Einfügen von Y) bleiben dagegen erhalten.)*

Für die korrekte Rücknahme temporärer Änderungen benötigen wir ein Kriterium, anhand dessen das WfMS selbständig und ohne teure Benutzerinteraktionen entscheiden kann, welche temporären Änderungen bei der Durchführung eines Rücksprungs zurückgenommen werden sollen und welche nicht. Wie bereits erwähnt, wird zu diesem Zweck für jede ÄT T ein Startbereich $TrStartRegion^T$ verwaltet. Er entspricht einer Teilmenge der Knoten des WF-Ausführungsgraphen (nach Anwendung der betreffenden Änderungen) mit folgender Semantik:

Liegen alle Knoten aus $TrStartRegion^T$ im Rücksprungbereich – das ist der Bereich des WF-Ausführungsgraphen, dessen Knoten infolge des Rücksprungs neu markiert werden (z.B. ein Schleifenkörper) – und ist T eine temporäre ÄT, so müssen die von T durchgeführten (temporären) Graphmodifikationen wieder unwirksam gemacht werden.

Auf das Verfahren zur Rücknahme temporärer Änderungen gehen wir in Abschnitt 8.5 ein. Zuvor beschreiben wir, wie die Startbereiche für die $ADEPT_{flex}$-Operationen definiert sind und wie sich daraus der Startbereich für ÄT ableiten läßt.

8.3.1 Definition von Startbereichen für die $ADEPT_{flex}$ -Operationen

Die Startbereiche für die $ADEPT_{flex}$-Änderungsoperationen sind wie folgt definiert:

- *Einfügen von Aktivitäten und Kontrollblöcken*: Für die beiden Einfügeoperationen *serialInsert* und *parallelInsert* (vgl. Abschnitt 7.3) wird der Startbereich durch den neu eingefügten Knoten X gebildet. Dasselbe gilt für die Operation *branchInsert*, wenn eine Aktivität X als neuer Teilzweig einer Parallelverzweigung (mit AND- oder OR-Join) eingefügt werden soll. Für alle diese Fälle definieren wir: $OpStartRegion = \{X\}$. Soll eine Aktivität X mittels *branchInsert* als neuer Teilzweig einer bedingten Verzweigung (*split*, *join*) eingefügt werden, so wird als Startbereich der OR-Split-Knoten dieser Verzweigung gewählt: $OpStartRegion = \{split\}$. Würde man hier, wie bei den anderen Operationen, den Knoten X als Startbereich festlegen, so könnten sich Probleme ergeben, wenn anschließend weitere Knoten vor X in den betreffenden Zweig eingefügt werden. – Ähnliche Festlegungen gelten für das Einfügen vorgefertigter Schleifen- oder Verzweigungskontrollblöcke (*insertLoopBlock*, *insertBranchingBlock*) in einen WF-Graphen (vgl. Abschnitt 7.3.2). Der Bezugspunkt zur Bildung des Startbereichs ist hier der Startknoten des Blocks.
- *Löschen von Aktivitäten und Kontrollblöcken*: Soll durch Anwendung der *deleteActivity*-Operation (vgl. Abschnitt 7.4) eine Aktivität X aus dem Ausführungsgraphen einer WF-Instanz gelöscht werden, wird der betreffende Knoten nicht physisch entfernt, sondern durch eine Nullaktivität X_{null} ersetzt. Dieser „Dummy"-Knoten bildet den Startbereich der Löschoperation, d.h. es gilt $OpStartRegion = \{X_{null}\}$. Auch für das Löschen eines Kontrollblocks, einer auf Instanzebene eher selten benötigten Operation, nehmen wir vereinfachend an, daß die darin enthaltenen Knoten jeweils durch Nullaktivitäten ersetzt werden. Der Knoten einer gelöschten Aktivität X bleibt auf WF-Instanzebene somit im Ausführungsgraphen erhalten, ist allerdings mit keiner Aktion mehr assoziiert. Zur besseren Unterscheidung verwenden wir dann als Bezeichner X_{null} anstelle von X, intern handelt es sich aber um denselben Knoten. Würde man den Knoten physisch entfernen, wäre er bei einem möglichen Rücksprung nicht mehr im Ausführungsgraphen enthalten. Es müßte daher ein anderer Startbereich für die Löschoperation, wie z. B. $OpStartRegion = pred(X)$, vereinbart werden.
- *Verschieben von Aktivitäten und Kontrollblöcken*: Der Startbereich der in Abschnitt 7.5.1 skizzierten Operation *moveActivity* wird durch den zu verschiebenden Knoten X und seinen bisherigen direkten Vorgängerknoten gebildet: $OpStartRegion = \{X\} \cup pred(X)$. Für die Operation *moveBlock* ist der Startbereich entsprechend durch den Startknoten n_{begin} des zu verschiebenden Blockes und dessen direkte Vorgänger definiert: $OpStartRegion = \{n_{begin}\} \cup pred(n_{begin})$.
- *Hinzufügen und Löschen von Spezialkanten* (vgl. Abschnitt 7.5.2): Der Startbereich der Operationen *insertSyncEdge*, *deleteSyncEdge*, *insertPriorityEdge* und *deletePriorityEdge* wird jeweils durch den Quellknoten der hinzugefügten bzw. gelöschten Kante $src \rightarrow dest$ definiert, d.h. es gilt: $OpStartRegion = \{src\}$. Im Gegensatz dazu vereinbaren wir für die beiden Operationen *insertFailureEdge* und *deleteFailureEdge* als Startbereich den Zielknoten der Fehlerkante.

- *Änderung von Knoten- und Kantenattributen*: Der Startbereich der beiden Basisoperationen *changeActivityAttribute* und *changeActivityTemplate* (vgl. Abschnitt 7.5.5) ist durch den Knoten *X* gegeben, dessen Attribut bzw. Vorlage geändert wird. Dasselbe gilt für Knoten, denen ein Nachforderungsdienst vor- oder nachgeschaltet wird. Für die Operation *changeEdgeAttribute* definieren wir als Startbereich den Quellknoten der Kante, es sei denn die Operation bezieht sich auf eine Fehlerkante. Für diesen Fall wählen wir als Startbereich den Zielknoten der Kante.
- *Bildung und Auflösung von Blockstrukturen*: Startbereich der Operation *nestBlock* (vgl. Abschnitt 7.5.3) ist der neu hinzukommende Aktivitätenknoten *X*, der den angegebenen Kontrollblock substituieren soll. Für die Operation *unnestBlock* – sie ersetzt eine komplexe Aktivität *X* durch einen Kontrollblock (n_{begin}, n_{end}) – definieren wir als Startbereich *OpStartRegion* = {n_{begin}}.
- *Änderung von DF-Modellen*: Für die Operationen *insertDataEdge* und *deleteDataEdge* (vgl. Abschnitt 7.5.4) ist der Startbereich durch den Knoten festgelegt, von dem die betreffende Datenkante ausgeht bzw. zu dem sie hinführt. Für die beiden Operationen *insertDataElement* und *deleteDataElement* bleibt der Startbereich undefiniert.
- *Komplexe Änderungsoperationen*: Auch für komplexe Änderungsoperationen (vgl. Abschnitt 7.6) gibt es vordefinierte Startbereiche. Der Startbereich der *insertBetweenNodeSets*-Operation wird durch die Knotenmenge M_{before} und den neu eingefügten Knoten *X* gebildet. Für das dynamische Überspringen einer Knotenmenge N_{skip} mit Auslassen haben wir angenommen, daß die übersprungenen Schritte, ebenso wie im Fall des Löschens einzelner Knoten, durch Nullaktivitäten ersetzt werden. Dem entsprechend konstituiert sich der Startbereich dieser Operation aus diesen „Dummy"-Knoten. Für das dynamische Überspringen einer Knotenmenge N_{skip} mit Nachholen der übersprungenen Schritte (*jumpForward*-Operation), wird der Startbereich durch die Menge N_{skip} und den vorgezogenen Schritt n_{target} gebildet.

Der Startbereich einer Änderungsoperation kann also Knoten umfassen, die erst mit Anwendung der Operation in die Instanz eingebracht werden. Demzufolge bezieht sich diese Knotenmenge immer auf den aus der Änderung hervorgehenden WF-Ausführungsgraphen.

8.3.2 Bestimmung des Startbereichs einer Änderungstransaktion

Wie bereits erwähnt, können im Verlauf einer ÄT mehrere Änderungsoperationen zur Ausführung kommen. Um für eine temporäre ÄT entscheiden zu können, ob ihre Auswirkungen bei einem Rücksprung aufgehoben werden sollen oder nicht, muß auch für Änderungstransaktionen ein Startbereich vereinbart werden. In jedem Fall werden die Auswirkungen einer ÄT immer nur vollständig oder aber gar nicht zurückgenommen. Im folgenden geben wir an, wie der Startbereich einer ÄT aus den Startbereichen der zugehörigen Änderungsoperationen abgeleitet werden kann. Dabei sind verschiedene Strategien denkbar, von denen die jeweils am besten geeignetste ausgewählt werden kann. In vielen Fällen kann die folgende Definition herangezogen werden:

Definition 8-1 (Kompletter Startbereich einer ÄT)

Sei durch CFS_{inst} der Ausführungsgraph einer WF-Instanz gegeben. *T* bezeichne eine auf der Instanz auszuführende ÄT und $TrAppliedOps^T$ die Menge der dabei zur Anwendung kommenden Änderungsoperationen (inkl. ihrer Aufrufparameter). Für eine einzelne Änderungsoperation $Op \in TrAppliedOps^T$ sei ihr Startbereich gemäß den in a) vereinbarten Festlegungen definiert und durch die Knotenmenge $OpStartRegion^{Op}$ gegeben. Dann kann der Startbereich der gesamten ÄT gebildet werden durch

$$TrStartRegion^T = \bigcup_{Op \in TrAppliedOps^T} OpStartRegion^{Op}$$

Wir bezeichnen $TrStartRegion^T$ als *kompletten Startbereich* der ÄT.

Wie das Beispiel 8-2 zeigt, können für komplexe Änderungen auch andere Festlegungen sinnvoll sein.

Beispiel 8-2 (Kompletter Startbereich einer ÄT):

Eine in einem Schleifenkörper enthaltene Aktivität *X* soll in der aktuellen Iteration der zugehörigen Schleife ausnahmebedingt übersprungen werden. Zu diesem Zweck wird eine ÄT *T*, die den Knoten *X* temporär löscht,

auf die Instanz angewendet. Begleitend wird von derselben ÄT ein von X datenabhängiger Knoten Y, der außerhalb der Schleife liegt, ebenfalls entfernt. In beiden Fällen kommt die *deleteActivity*-Operation zur Anwendung, wobei die zu löschenden Aktivitäten X und Y durch Nullaktivitäten ersetzt werden. Der komplette Startbereich der ÄT ist dann gegeben durch $TrStartRegion^T = \{X_{null}, Y_{null}\}$. Wird die Schleife erneut durchlaufen, so würde die ÄT nicht rückgängig gemacht werden, da lediglich der Knoten X_{null} im Rücksprungbereich liegt.

In solchen Fällen kann, alternativ zu Definition 8-1, der ÄT-Startbereich wie folgt vereinbart werden:

Definition 8-2 (Reduzierter Startbereich einer ÄT)

Sei durch CFS_{inst} der Ausführungsgraph einer WF-Instanz gegeben. T bezeichne eine auf der Instanz auszuführende ÄT mit kompletten Startbereich (vgl. Definition 8-1):

$$TrStartRegion^T = \bigcup_{Op \in TrAppliedOps^T} OpStartRegion^{Op}$$

Dann bezeichnen wir die Knotenmenge

$$TrStartRegionRed^T = \left\{ n \in TrStartRegion^T \,\middle|\, \forall\, m \in TrStartRegion^T \setminus \{n\}: m \notin pred^*(n) \right\}$$

als (*transitiv*) *reduzierten Startbereich* der ÄT.

Beispiel 8-3 (Reduzierter Startbereich einer ÄT):

Für die temporäre ÄT aus Beispiel 8-2 erhalten wir $TrStartRegionRed^T = \{X_{null}\}$. Wird diese Knotenmenge als Startbereich der Änderung vereinbart, wird sie bei dem beschriebenen Schleifenrücksprung zurückgenommen.

Transitiv reduzierte Startbereiche können unvorteilhaft sein, wenn von einer temporären ÄT mehrere Aktivitäten in unterschiedlichen Bereichen des WF-Graphen eingefügt werden. Angenommen, im Ausführungsgraphen aus Abbildung 8-2 a) sind die beiden Knoten X und Y von derselben temporären ÄT eingefügt worden. Wird der transitiv reduzierter Startbereich {X} verwendet, so werden die beiden Knoten bei wiederholter Ausführung der inneren Schleife bereits wieder entfernt, obwohl bisher nur der Knoten X ausgeführt worden ist. Soll dieser Effekt nicht auftreten, muß der komplette Startbereich {X, Y} verwendet werden. In Einzelfällen können auch benutzerdefinierte Startbereiche sinnvoll sein (siehe Abschnitt 8.4.3.2). Aus diesem Grund kann in $ADEPT_{flex}$ auch eine beliebige Knotenmenge als Startbereich vereinbart werden. Wichtig ist an dieser Stelle der Hinweis, daß die im folgenden beschriebenen Verfahren unabhängig davon anwendbar sind, wie die Definition der Startbereiche konkret erfolgt. Es setzt lediglich voraus, daß für ÄT ein solcher Startbereich festgelegt wird. Neben den diskutierten Möglichkeiten sind weitere Kriterien für die Rücknahme von ÄT denkbar, etwa in Abhängigkeit von Zeitaspekten oder von Wertebelegungen für Datenelemente.

8.4 Permanente und temporäre Abänderung einer WF-Instanz

Sollen Instanzänderungen mit unterschiedlicher Gültigkeitsdauer unterstützt werden, muß a priori ausgeschlossen werden, daß permanente von temporären Änderungen abhängen. Dies wäre zum Beispiel der Fall, wenn ein permanent eingefügter Knoten ein Datenelement obligat liest, das ausschließlich von Aktivitäten geschrieben wird, die zuvor temporär eingefügt wurden. Zwar sind solche Abhängigkeiten zum Zeitpunkt der Änderungseinbringung meist unkritisch, können aber im weiteren Verlauf der WF-Ausführung, wenn temporäre Änderungen wieder zurückgenommen werden, zu Inkonsistenzen oder Fehlern führen. Da sich solche Probleme zum Zeitpunkt ihres Auftretens nur auf sehr unbefriedigende Art und Weise behandeln lassen, werden sie in $ADEPT_{flex}$ bereits bei der Einbringung permanenter Änderungen ausgeschlossen. Auch Abhängigkeiten zwischen verschiedenen temporären Änderungen können bei Rücksprüngen zu Problemen führen, wenn nur eine der beiden Änderungen zurückgenommen wird. Im einzelnen ergeben sich die folgenden Fragestellungen:

- Unter welchen Voraussetzungen kann eine Änderung permanent und wann kann sie lediglich temporär auf eine WF-Instanz angewendet werden?
- Wie muß die Ausführung einer permanenten Änderung erfolgen, um bei der späteren Rücknahme temporärer Änderungen Probleme zu vermeiden?

- Auf der Grundlage welcher Informationen (z.B. bzgl. anderer WF-Änderungen) darf eine neue temporäre Änderung eingebracht werden?

8.4.1 Grundlagen

Für jede WF-Instanz werden, zusätzlich zu der in Abschnitt 8.2 eingeführten Änderungshistorie, die folgenden „logischen“ Prozeßgraphen verwaltet:

- ein Prozeßgraph $P_{current}$, der den aktuellen Ausführungsgraphen der WF-Instanz mit allen bisher durchgeführten und aktuell gültigen Änderungen widerspiegelt. $P_{current}$ umfaßt dem entsprechend auch Informationen zum aktuellen Status der WF-Instanz (Graphmarkierungen, Datenkontext).
- ein Prozeßgraph P_{perm}, der sich aus der Vorlage bzw. dem Ursprungs-WF-Graphen der WF-Instanz durch Anwendung aller bisher eingebrachten permanenten Änderungen ergibt. P_{perm} repräsentiert das dauerhaft gültige Schema der Instanz (nur Kontroll- und Datenfluß, kein Status).

Wie wir nachfolgend sehen werden, bilden diese beiden WF-Graphen, zusammen mit der Änderungshistorie der WF-Instanz, die Grundlage für die korrekte Durchführbarkeit temporärer und permanenter WF-Modifikationen. Der Ausführungsgraph $P_{current}$ spiegelt alle aktuell gültigen Änderungen – permanente und temporäre – und den aktuellen Status der WF-Instanz wider. Er bildet deshalb die Grundlage für die Ausführung der WF-Instanz. Die Struktur dieses WF-Ausführungsgraphen ist nur dann identisch mit dem Ursprungsgraphen der WF-Instanz, wenn auf sie bisher noch keine permanenten Änderungen angewendet und alle eventuell zuvor durchgeführten temporären Änderungen wieder zurückgenommen worden sind. Der zusätzliche Prozeßgraph P_{perm}, der ausschließlich permanente Änderungen berücksichtigt, wird durch das hier angewandte Verfahren notwendig. Er dient der Überprüfung der Korrektheit neu einzubringender, permanenter Änderungen. Würde eine solche Überprüfung ausschließlich auf dem Prozeßgraphen $P_{current}$ erfolgen, wären Abhängigkeiten zwischen permanenten und temporären Änderungen unvermeidbar. Der Prozeßgraph P_{perm} kann mehr oder weniger Knoten und Kanten als $P_{current}$ umfassen, abhängig von der Art der durchgeführten Änderungen.

Wir wollen an dieser Stelle keine Annahmen zur physischen Repräsentation dieser beiden Graphen treffen. $P_{current}$ kann prinzipiell aus der WF-Vorlage und den verfügbaren Historiendaten (Ablaufhistorie, Datenelementhistorien und Änderungshistorie) erzeugt werden. Auch P_{perm} läßt sich aus der WF-Vorlage und den vorliegenden Informationen rekonstruieren. Er ergibt sich, wenn alle permanenten Änderungen (in der Reihenfolge ihrer Einbringung) auf den Ursprungsgraphen angewendet werden.

8.4.2 Durchführung permanenter Änderungen einer WF-Instanz

Wie bereits erwähnt, muß bei der Anwendung einer permanenten Änderung beachtet werden, daß sie nicht auf Grapheigenschaften basiert, die von vorangehenden temporären Modifikationen erzeugt wurden. Unter Verwendung der beiden Prozeßgraphen $P_{current}$ und P_{perm} erfolgt die Ausführung einer permanenten ÄT T_{new} logisch in zwei Stufen:

Zuerst wird geprüft, ob aus der Anwendung von T_{new} auf P_{perm} wieder ein Prozeßgraph mit korrektem Kontroll- und Datenflußschema resultiert. Dies erfolgt unabhängig vom aktuellen Status der WF-Instanz und den aktuell für sie gültigen temporären Änderungen. Dadurch stellen wir sicher, daß die permanente Änderung bei späteren Rücksprüngen, wenn bestimmte temporäre Änderungen infolge ihrer Rücknahme nicht mehr wirksam sind, die Korrektheit und Konsistenz der WF-Instanz nicht verletzt. Treten bei der Korrektheitsprüfung bzgl. P_{perm} unauflösbare Konflikte auf, so wird die ÄT abgebrochen und der Benutzer informiert. Andernfalls wird in der 2. Stufe des Verfahrens geprüft, ob die permanente ÄT auch auf den aktuellen Ausführungsgraphen $P_{current}$ korrekt anwendbar ist. Diese Überprüfung wird notwendig, da die permanente Änderung bei einem zu weit fortgeschrittenen Status der WF-Instanz oder bei unauflösbaren Konflikten mit zeitlich vorausgehenden, temporären Änderungen aktuell nicht angewendet werden darf. Letzteres wäre z.B. der Fall, wenn obligate Eingabeparameter eines permanent einzufügenden Schrittes nicht versorgt werden können, weil ein Schreiberknoten aus dem Ausführungsgraphen temporär gelöscht wurde.

Ergeben sich bei der Überprüfung der Anwendbarkeit von T_{new} auf $P_{current}$ keine Probleme, d.h. führt die Anwendung dieser ÄT nicht zu Korrektheits- und Konsistenzverletzungen dieses Graphen, kann

sie dauerhaft in die WF-Instanz eingebracht werden. Die ÄT wird dann – aus logischer Sicht – atomar auf die beiden Graphen $P_{current}$ und P_{perm} angewendet und es wird ein entsprechender Änderungseintrag (mit *TrDuration* = PERMANENT) der Änderungshistorie hinzugefügt. Ergeben die Korrektheitsprüfungen bzgl. $P_{current}$ dagegen unauflösbare Konflikte, so darf die ÄT im aktuellen Zustand der WF-Instanz nicht angewendet werden. Für diesen Fall können zwei alternative Strategien verfolgt werden:

- T_{new} wird abgebrochen und der Benutzer entsprechend informiert.
- T_{new} wird vorgemerkt und – falls möglich – zu einem späteren Zeitpunkt, d.h. mit Verzögerung, auf die WF-Instanz angewendet. Dieser Fall ist für langlaufende, zyklische Prozesse relevant.

Eine wichtige Grundlage für die Überprüfung der korrekten Anwendbarkeit einer permanenten ÄT bilden die vom ADEPT-Basismodell geforderten Strukturierungsregeln und die für ihre Analyse bereitgestellten Verfahren. Des weiteren können bei der Einbringung einer permanenten Änderung potentielle Konflikte zwischen dieser Änderung und vorangehenden, temporären Modifikationen auf der Basis effizienter Konflikttests festgestellt werden. Zuerst wird geprüft, ob aktuell überhaupt temporäre Änderungen eingebracht worden sind. (Dies kann durch einen einfachen Zähler verwirklicht werden, der mit 0 initialisert wird und der beim Einbringen bzw. Wegfall einer temporären Änderung jeweils um 1 erhöht bzw. erniedrigt wird.) Ist dies nicht der Fall, sind zur Anwendung der Änderung auf $P_{current}$ lediglich Statusbedingungen zu prüfen. Falls es aktuell gültige, temporäre Änderungen gibt, kann festgestellt werden, ob die Änderungen in sich überlappenden oder in disjunkten Bereichen des Prozeßgraphen liegen oder ob sie Lese- oder Schreibkanten auf dieselben Datenelemente manipulieren. In den meisten Fällen lassen sich dadurch Konflikte mit einfachen Mitteln ausschließen. Wir verzichten an dieser Stelle auf die Darstellung entsprechender Konfliktrelationen und -tests.

8.4.3 Durchführung temporärer Änderungen einer WF-Instanz

Das Konzept temporärer WF-Änderungen ist für die angemessene Behandlung von Ausnahmesituationen in WfMS unerläßlich, insbesondere wenn beim späteren Wegfall einer Ausnahme, die durch sie hervorgerufenen WF-Änderungen wieder rückgängig gemacht werden sollen. In heutigen WfMS muß die Rücknahme nicht mehr gültiger Änderungen, sofern das WfMS dynamische WF-Modifikationen überhaupt zuläßt, manuell durch den Anwender erfolgen. Dies ist bei komplexeren Prozessen und bei einer großen Zahl von Prozeßinstanzen schlichtweg nicht praktikabel.

8.4.3.1 Anforderungen im Hinblick auf die Einbringung temporärer Änderungen

Die Änderungsoperationen (und ihre Aufrufparameter) einer temporären ÄT T_{new} werden vom Benutzer, in Interaktion mit dem System, festgelegt (vgl. Kap. 10). Aus verschiedenen Gründen sollte es dabei möglich sein, daß T_{new} direkt oder indirekt Bezug auf die (aktuell gültigen) Effekte zuvor eingebrachter Änderungstransaktionen T_1, ..., T_n nehmen kann. Erst dadurch wird es möglich, daß

1. die von T_1, ..., T_n eingefügten Schreibzugriffe auf den WF-Datenkontext zur Versorgung von Eingabeparametern der von T_{new} eingefügten Aktivitäten dienen können.
2. durch die Anwendung von T_{new} die Effekte zuvor eingebrachter Änderungen bei Bedarf korrigiert bzw. aufgehoben werden können (z.B. Löschen einer zuvor temporär eingefügten Aktivität).
3. die Änderungsoperationen von T_{new} die von T_1, ..., T_n eingefügten Knoten referenzieren können (z.B. serielles Einfügen einer neuen Aktivität direkt nach einem temporär eingefügten Knoten).

Abhängigkeiten zwischen temporären Änderungstransaktionen können in Einzelfällen auch Probleme verursachen, insbesondere wenn die Änderungen nicht gemeinsam aus dem Ausführungsgraphen entfernt werden. Dieser Fall kann auftreten, wenn ihre Startbereiche in unterschiedlichen Regionen des WF-Ausführungsgraphen (z.B. unterschiedlichen Parallelzweigen, aufeinanderfolgenden oder geschachtelten Schleifen) liegen und deshalb bei einem Schleifenrücksprung ggf. nur eine Teilmenge der aktuell gültigen temporären Änderungstransaktionen zurückgenommen wird. Hierauf gehen wir im folgenden Abschnitt ein.

8.4.3.2 Probleme bei temporären Änderungen und ihre Behandlung

a) Kritische Abhängigkeiten und Wechselwirkungen zwischen temporären Änderungen

Für eine neu einzubringende, temporäre ÄT T_{new} muß zuerst überprüft werden, ob sie korrekt auf $P_{current}$ anwendbar ist. Ist dies nicht der Fall, etwa wenn der aktuelle Status dieses Ausführungsgraphen die Anwendung einer oder mehrerer Änderungsoperationen von T_{new} nicht zuläßt, wird die ÄT abgebrochen und der Benutzer entsprechend informiert.

Ist die temporäre ÄT T_{new} dagegen korrekt auf $P_{current}$ anwendbar, scheint einer erfolgreichen Einbringung nichts mehr im Wege zu stehen. Wie das Beispiel aus Abbildung 8-2 zeigen soll, ist dem nicht immer so. Angenommen in Abbildung 8-2 a) ist die temporär eingefügte Aktivität Y datenabhängig von dem zuvor von einer anderen ÄT temporär eingefügten Schritt X. Wird die Einfügung von X zurückgenommen bevor Y zur Ausführung kommt (vgl. Abbildung 8-2 b), würde – zumindest bei strikter Auslegung der ADEPT-Strukturierungsregeln – ein nicht korrekter Ausführungsgraph resultieren (→ Verletzung von Regel DF-1). Eine Möglichkeit zur Vermeidung solcher Probleme besteht darin, Abhängigkeiten zwischen temporären Änderungen nur zuzulassen, wenn feststeht, daß sie immer gemeinsam, d.h. bei der Durchführung derselben Rücksprünge, unwirksam gemacht werden. Dies kann für vormodellierte Schleifen- und Fehlerrücksprünge, nicht dagegen für dynamisch festgelegte Ad-hoc-Rücksprünge (vgl. Kapitel 6) sichergestellt werden, da hier der Rücksprungbereich a priori nicht festliegt. Darüber hinaus erfordert dieser *pessimistische Ansatz* aufwendige Überprüfungen bei der Ausführung temporärer Änderungen.[6]

Ein *optimistisches Verfahren* dagegen würde solche Abhängigkeiten zwischen temporären Änderungen, bis auf wenige Ausnahmen (siehe unten), nicht explizit ausschließen. Dadurch können aufwendige Überprüfungen bei der Änderungseinbringung vermieden werden. Dem liegt die Annahme zugrunde, daß unerwünschte Wechselwirkungen zwischen temporären WF-Änderungen, die nicht innerhalb desselben Rücksprungbereichs liegen und die deshalb nicht immer gemeinsam aus dem WF-Ausführungsgraphen entfernt werden, selten sind, was in der Praxis auch mehrheitlich zutrifft [Meye96]. Dem entsprechend beeinträchtigt die Rücknahme einer temporären Änderung andere Änderungstransaktionen, deren Startbereich nicht in den aktuellen Rücksprungbereich fällt, in der Regel nicht. Sollten dennoch potentielle Probleme auftreten, kann immer noch versucht werden, sie automatisch zu behandeln. Bezogen auf das obige Beispiel etwa gilt, daß die von der temporär eingefügten Aktivität X geschriebenen Datenelemente auch nach dem Entfernen dieses Knotens noch im Datenkontext-Schema der WF-Instanz erhalten bleiben[7]. Die Versorgung der Lesezugriffe von Y ist also weiter gewährleistet, auch wenn dies aus der Struktur des dargestellten Prozeßgraphen, die nach dem Entfernen von X resultiert, nicht unmittelbar hervorgeht. Hier nutzt das System intern Informationen zum aktuellen WF-Status und zum aktuellen WF-Datenkontext. Ist eine automatische Behandlung nicht möglich, nehmen wir bei diesem optimistischen Ansatz in seltenen Fällen in Kauf, daß „abhängige" Änderungen ggf. zurückgenommen werden müssen, um wieder zu einem korrekten Ausführungsgraphen zu gelangen. Die Rücknahme solcher abhängiger Änderungen gestaltet sich aus Systemsicht unproblematisch, solange die von ihnen modifizierten Bereiche des WF-Graphen bisher noch nicht durchlaufen worden sind. Das dem nicht immer so ist, zeigt das folgende Beispiel:

---

[6] Angenommen, die Rücknahme temporärer Änderungen würde ausschließlich in Verbindung mit Schleifenrücksprüngen erfolgen. Dann würden für jede temporäre ÄT die Ereignisse bzw. Rücksprünge, die zu ihrer Rücknahme führen können, a priori (d.h. zum Zeitpunkt ihrer Einbringung) feststehen. Umgekehrt ließe sich für jeden Schleifenrücksprung vorab bestimmen, welche temporären Änderungen potentiell zurückzunehmen sind. Unter Ausnutzung dieser Information und dem aktuellen Status von $P_{current}$ kann dann ein (aufwendiges) Verfahren konstruiert werden, das sicherstellt, daß eine neue temporäre ÄT T nur auf der Basis solcher bereits eingebrachter Änderungen angewendet werden darf, die entweder zusammen mit T oder zeitlich nach T wieder aus $P_{current}$ entfernt werden. Wir verzichten in dieser Arbeit auf die Darstellung eines solchen Verfahrens.

[7] Von der Rücknahme temporärer Änderungen sind Datenelemente nicht betroffen, sofern auf sie im Verlauf der bisherigen WF-Ausführung mindestens ein Schreibzugriff stattgefunden hat. Dadurch bleiben Datenelementwerte bis zur Terminierung der WF-Instanz erhalten, es sei denn sie werden explizit in Verbindung mit dem (partiellen) Rollback des WF zurückgesetzt.

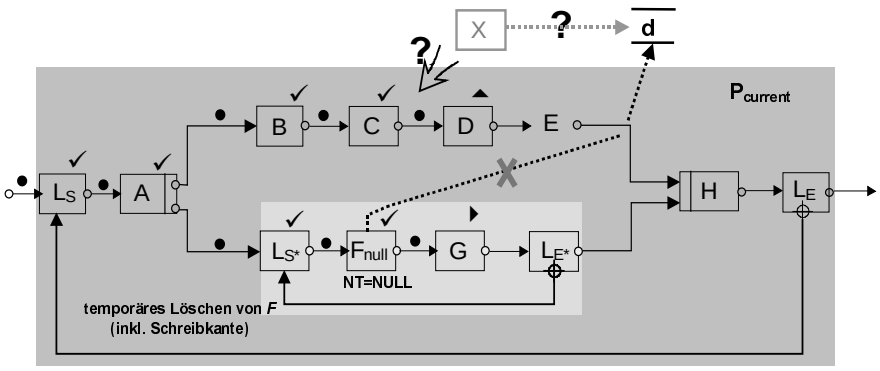


Abbildung 8-3: *Problematische Wechselwirkungen zwischen temporären Änderungen in Parallelzweigen*

Beispiel 8-4 (Problematische Wechselwirkungen zwischen temporären Änderungen)

In der WF-Instanz aus Abbildung 8-3 wurde die Aktivität F und ihr Schreibzugriff auf d temporär gelöscht. Das temporäre Einfügen von X zwischen C und D und die Hinzunahme der Schreibkante von X nach d ist, bezogen auf den dargestellten Ausführungsgraphen $P_{current}$, zulässig. Allerdings können sich bei Anwendung dieser Änderung in der Folge Probleme ergeben. Bei wiederholter Ausführung der inneren Schleife werden der temporär entfernte Knoten F und seine Schreibkante wieder eingefügt, wodurch es zu parallelen Schreibzugriffen auf d und damit zu einer Verletzung von Regel DF-2 kommt. Der Grund für die Entstehung dieser Problematik ist, daß das Einfügen von X auf Grundlage der temporären Löschung von G erfolgt, die zuletzt genannte Änderung aber möglicherweise vor der erstgenannten wieder unwirksam wird. Im allgemeinen treten derartige Probleme auf, wenn die Startbereiche der Änderungen in unterschiedlichen Teilzweigen einer Parallelverzweigung liegen. Eine Möglichkeit besteht darin, die Einfügung von X zuzulassen, sie aber im Fall der Wiedereinbringung von F zurückzunehmen. Dieser Ansatz wäre problematisch, da X und ggf. seine Nachfolgerknoten C und D bereits beendet sein können, noch bevor die innere Schleife zum wiederholten Male ausgeführt wird. Für diesen Fall kann der temporär eingefügte Knoten X nicht einfach aus dem Ausführungsgraphen entfernt werden. Vielmehr muß dann, um in einem konsistenten Zustand fortfahren zu können, die Bearbeitung des oberen Zweiges vor die Ausführung von X zurückgesetzt werden. Eine solche Vorgehensweise ist im allgemeinen nicht akzeptabel.

Auch wenn es sich hier um ein konstruiertes Beispiel handelt, müssen solche Fälle berücksichtigt werden. Sie ergeben sich immer dann, wenn bei der Rücknahme einer temporären Änderung eine „abhängige“ ÄT T_{dep} ebenfalls zurückgenommen werden soll, es aber mindestens einen Knoten $n \in TrStartRegion^{T_{dep}}$ gibt, der außerhalb des Rücksprungbereichs liegt und der eine Markierung $NS(n) \neq$ NOT_ACTIVATED besitzt. In einem solchen Fall darf die abhängige Änderung T_{dep} nicht ohne weiteres aus dem Ausführungsgraphen entfernt werden, da ansonsten inkonsistente Zustände resultieren können. Da in diesem Fall auch das partielle Zurücksetzen des WF keine zufriedenstellende Lösung darstellt, sollte die Entstehung solcher Situationen von vornherein ausgeschlossen werden. Dies erreicht man dadurch, daß eine neu einzubringende, temporäre Änderung T_{new} auf eine vorausgehende, temporären Änderung T nur Bezug nehmen darf, wenn gilt:

$$TrStartRegion^{T} \prec TrStartRegion^{T_{new}} :\Leftrightarrow$$
$$\forall\, m \in TrStartRegion^{T_{new}}:\ \exists\, n \in TrStartRegion^{T}: n \in pred^{*}(m) \cup \{m\} \qquad (☆)$$

Durch dieses einfach überprüfbare Kriterium erreichen wir, daß bei einem Rücksprung, der zur Aufhebung von T führt, nicht nur die Knoten des Startbereichs von T, sondern auch die des Startbereichs von T_{new} als NOT_ACTIVATED markiert werden. Dies folgt unmittelbar aus den Markierungseigenschaften von WF-Graphen und aus der Bedingung (☆), da jeder Knoten aus dem Startbereich von T_{new} Nachfolger von mindestens einem Knoten aus dem Startbereich von T (inkl.) ist. Muß nun T_{new} bei der Rücknahme von T, aufgrund von Abhängigkeiten zu dieser ÄT, ebenfalls zurückgenommen werden, können die oben skizzierten Probleme vermieden werden. Die Bedingung (☆) ist für zwei Änderungstransaktionen T und T_{new} immer erfüllt, wenn sie denselben Startbereich besitzen. Dadurch wird es beispielsweise möglich einen temporär eingefügten Knoten vor seiner Ausführung wieder zu löschen (siehe unten).

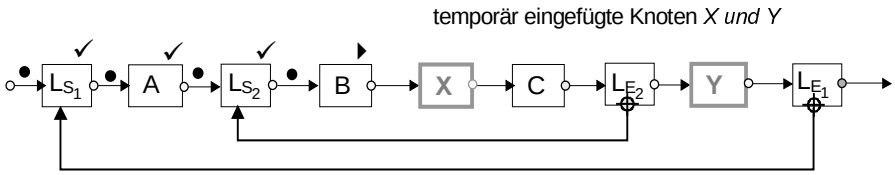
Soll etwa eine Aktivität X temporär in den Ausführungsgraphen einer WF-Instanz eingefügt werden, so kann bei Zugrundelegung des Kriteriums (☆) auf alle aktuell gültigen, temporären Änderungen

Bezug genommen werden, deren Startbereich vor der neu einzufügenden Aktivität $X - \{X\}$ bildet den Startbereich der Einfügeoperation – liegt. Im Gegensatz dazu muß die Einfügung von X unabhängig von temporären Änderungen, die in parallelen Ausführungszweigen (vgl. obiges Beispiel) oder in nachfolgenden WF-Regionen vorgenommen worden sind, und unabhängig von deren späterer Rücknahme möglich sein.

Prinzipiell sollte eine temporäre ÄT die Effekte einer vorausgehenden temporären Änderungen vollständig oder teilweise wieder aufheben können. Angenommen in den Ausführungsgraphen einer WF-Instanz wurde eine Aktivität Z temporär eingefügt, die dann aber doch nicht zur Ausführung kommen soll. Dann muß der betreffende Knoten vor seinem Start wieder aus $P_{current}$ entfernt werden. Da Z in P_{perm} nicht enthalten ist, muß dies in dem hier angewandten Verfahren mittels einer temporären Löschung erfolgen. Der Startbereich der betreffenden ÄT wird dann durch den leeren Ersatzknoten Z_{null} gebildet, der die zu löschende Aktivität Z logisch ersetzt. Intern werden diese beiden Knoten identisch behandelt, d.h. die beiden Änderungstransakionen besitzen denselben Startbereich. Dies führt bei einem eventuellen Rücksprung, in dessen Rücksprungbereich der Knoten Z bzw. Z_{null} liegt, dann dazu, daß zuerst die temporäre Löschung und dann die temporäre Einfügung von Z wieder rückgängig gemacht (vgl. Abschnitt 8.5). Aus Benutzersicht taucht der betreffende Knoten im weiteren Verlauf der Ausführung – wie gewünscht – nicht mehr im Ausführungsgraphen der WF-Instanz auf.

b) Festlegung geeigneter Startbereiche für temporäre Änderungstransaktionen

Der Festlegung „geeigneter“ Startbereiche für temporäre ÄT ist entscheidend für die spätere Rücknahme der von ihnen bewirkten Änderungen. In Abschnitt 8.3 haben wir mit *kompletten* und (transitiv) *reduzierten* Startbereichen zwei mögliche Vereinbarungen für diese Knotenmenge kennengelernt. In der Mehrzahl der Fälle sind komplette Startbereiche ausreichend, um die vom Anwender gewünschte Änderungssemantik zu erzielen. Wie anhand von Beispiel 8-2 gezeigt, kann in Einzelfällen auch die Verwendung (transitiv) reduzierter Startbereiche sinnvoll sein. Anhand des nachfolgenden Beispiels wollen wir noch kurz die Möglichkeit benutzerdefinierter Startbereiche diskutieren. Dazu sei der folgende Ausschnitt eines Ausführungsgraph $P_{current}$ einer WF-Instanz gegeben:



Wir nehmen an, daß die beiden Knoten X und Y von einer temporären ÄT T_1 in $P_{current}$ eingefügt worden sind. Des weiteren stelle sich später heraus, daß die Einfügung von X nicht notwendig gewesen wäre, wohingegen der Knoten Y erhalten bleiben soll. Die Aktivität X soll deshalb durch die Anwendung einer temporären ÄT T_2 – die *permanente* Löschung eines *temporär* eingefügten Knotens ist bei dem hier angewandten Verfahren nicht möglich – wieder aus dem Ausführungsgraphen entfernt werden. Der komplette bzw. reduzierte Startbereich von T_2 ist dann durch den leeren Ersatzknoten X_{null} gegeben. Wurde für die ÄT T_1 als Startbereich die Knotenmenge {X, Y} vereinbart (kompletter Startbereich), kann nun der unerwünschte Effekt auftreten, daß bei einem Rücksprung der inneren Schleife die temporäre Löschung von X wieder aufgehoben wird, nicht dagegen das vorangehende, temporäre Einfügen von X und Y. Das heißt der Knoten X würde plötzlich wieder im Ausführungsgraphen auftauchen. Um solche Effekte auszuschließen, gibt es zwei alternative Möglichkeiten:

- Es werden sowohl X als auch Y von einer ÄT T_2 gelöscht. Anschließend wird Y von einer ÄT T_3 wieder an der entsprechenden Stelle im Ausführungsgraphen temporär eingefügt.
- Man expandiert den Startbereich der oben skizzierten ÄT T_2 auf die Knotenmenge {X, Y} - die Festlegung solcher anwendungsdefinierter Startbereiche ist in $ADEPT_{flex}$ möglich.

Das geschilderte Problem würde im vorliegenden Beispiel nicht auftreten, wenn für die ÄT T_1 der reduzierte Startbereich {X} vereinbart worden wäre. Für die nachfolgenden Beispiele wählen wir der Einfachheithalber jeweils komplette Startbereiche. Die Anwendung der in der Folge vorgestellten Verfahren ist allerdings unabhängig davon, wie die Startbereiche für ÄT konkret festgelegt werden.

8.4.3.3 (Optimistisches) Verfahren

Auf der Grundlage der in Abschnitt 8.4.3.2 angestellten Überlegungen ergibt sich das folgende optimistische Verfahren zur Einbringung einer neuen temporären Änderung T_{new}:

1. Zuerst wird geprüft, ob T_{new} auf den aktuellen Ausführungsgraphen $P_{current}$ der WF-Instanz korrekt anwendbar ist. Ist dies nicht möglich, so wird die ÄT abgebrochen und der Benutzer darüber informiert. Andernfalls wird mit Schritt 2 fortgefahren.
2. T_{new} darf nur auf der Grundlage solcher temporären ÄT T eingebracht werden, für welche die Relation $TrStartRegion^{T} \prec TrStartRegion^{T_{new}}$ (☆) zutrifft. Darauf basierend konstruieren wir, ausgehend vom Ursprungsgraphen $P_{initial}$ der WF-Instanz (d.h. dem WF-Graphen ihrer Vorlage), einen Hilfsgraphen $P_{temporary}$ nach folgendem Grundprinzip:

   Zunächst wird $P_{temporary}$ als Kopie des WF-Graphen $P_{initial}$ angelegt. Dann wird die Änderungshistorie der WF-Instanz in Vorwärtsrichtung, d.h. beginnend mit dem ältesten Eintrag, durchlaufen. Für jeden traversierten Historieneintrag einer ÄT T wird wie folgt verfahren:

   - Beschreibt T eine permanente Änderung, wird sie auf $P_{temporary}$ angewendet.
   - Beschreibt T eine temporäre Änderung, wird sie auf $P_{temporary}$ angewendet, wenn gilt: $TrStartRegion^{T} \prec TrStartRegion^{T_{new}}$

   Nach der Erzeugung dieses Prozeßgraphen wird überprüft, ob T_{new} korrekt auf $P_{temporary}$ anwendbar ist. Trifft dies nicht zu, wird die ÄT abgebrochen und der Benutzer informiert. Andernfalls wird die ÄT atomar auf den Prozeßgraphen $P_{current}$ angewendet und es wird ein entsprechender Eintrag (mit *TrDuration* = TEMPORARY) der Änderungshistorie hinzugefügt. Der Graph $P_{temporary}$, der lediglich der Überprüfung der Anwendbarkeit von T_{new} dient, wird nicht weiter benötigt. Auch der Prozeßgraph P_{perm} spielt bei der Ausführung temporärer Änderungen keine Rolle.

Beispiel 8-5 (Ausführung temporärer Änderungen)

Wir beziehen uns auf Beispiel 8-4. Bezogen auf den Ausführungsgraphen $P_{current}$ (vgl. Abbildung 8-3) ist das Einfügen der Aktivität X zwischen C und D sowie die Hinzunahme der Schreibkante X→ d möglich. Konstruiert man $P_{temporary}$, ergibt sich der in Abbildung 8-4 dargestellte Prozeßgraph. X kann nicht in $P_{temporary}$ eingefügt werden, da ansonsten Regel DF-2 verletzt wäre. Demzufolge ist das temporäre Einfügen der Aktivität X und ihres Schreibzugriffs nicht möglich.

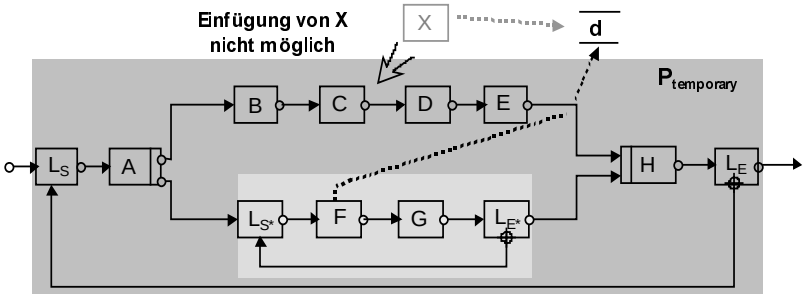


Abbildung 8-4: *Hilfsgraph zur korrekten Einbringung temporärer Änderungen*

Das skizzierte Verfahren zur Einbringung temporärer Änderungen läßt sich noch optimieren. Beispielsweise kann durch einmaliges Durchlaufen der Änderungshistorie festgestellt werden, ob für die WF-Instanz aktuell überhaupt temporäre ÄT vorliegen, für die das Kriterium (☆) zutrifft. Ist dies nicht der Fall, so sind die beiden Prozeßgraphen $P_{current}$ und $P_{temporary}$ strukturgleich, so daß für die Einbringung von T_{new} die Durchführung des 1. Schrittes (des Verfahrens) ausreichend ist. Andernfalls muß auch der 2. Schritt, der mit der Konstruktion des Prozeßgraphen $P_{temporary}$ verbunden ist, angewendet werden. Für diesen Fall lassen sich potentielle Konflikte zwischen temporären Änderungen in vielen Fällen durch einfache Konflikttests (z.B. „Manipulieren die beiden Änderungen dieselben Datenelemente?“) ausschließen. Wir verzichten an dieser Stelle auf eine Darstellung dieser Optimierungen.

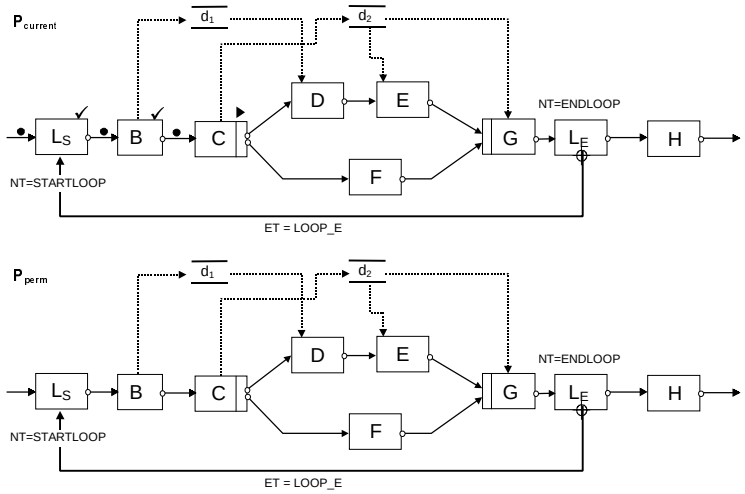
Zusammenfassend halten wir fest, daß eine temporäre Änderung T_{new} sowohl auf der Grundlage zuvor durchgeführter permanenter Änderungen als auch auf der Grundlage bestimmter temporärer Änderungen eingebracht werden darf. Insbesondere können durch die Anwendung einer temporären Änderung auch die Effekte vorausgehender Änderungen vollständig oder teilweise aufgehoben werden. Darüber hinaus macht sich $ADEPT_{flex}$ beim Einbringen temporärer Änderungen den aktuellen WF-Status zunutze. Durch Rückgriff auf Informationen zum bisherigen Verlauf der WF-Ausführung kann z.B. die Menge der Datenelemente, auf die obligate Eingabeparameter eines temporär eingefügten Schrittes potentiell abgebildet werden können, ggf. vergrößert werden. Ein Beispiel hierzu haben wir in Abschnitt 7.3.1.4 vorgestellt.

8.4.4 Beispiel für die Einbringung permanenter und temporärer Änderungen

Wir wollen das Prinzip der Einbringung von Änderungen an einem Beispiel illustrieren:

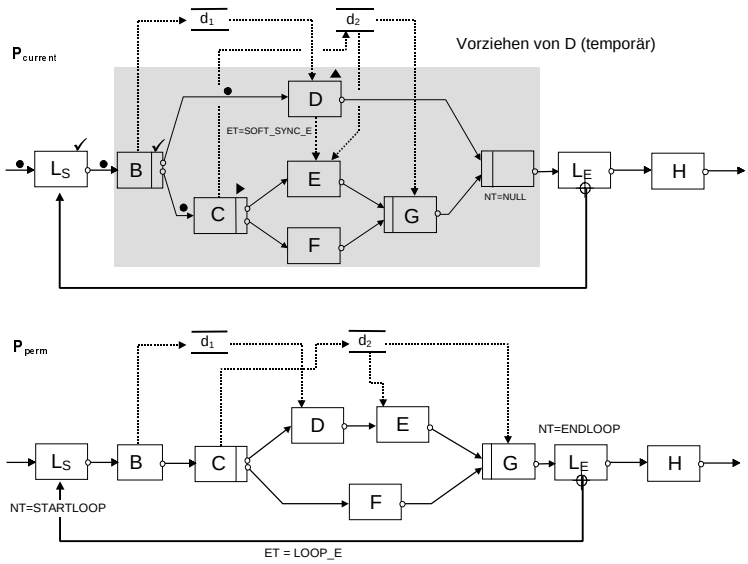
Beispiel 8-6 (Einbringung temporärer und permanenter Änderungen in eine WF-Instanz)

1. Der Teil-Ausführungsgraph einer WF-Instanz sei durch $P_{current}$ gegeben. Wir nehmen an, daß bisher noch keine dynamische Änderung durchgeführt worden ist. Dem entsprechend sind die beiden logischen Graphen $P_{current}$ und P_{perm} strukturgleich, wobei P_{perm} keine Statusmarkierungen besitzt.



2. Sei die WF-Instanz aus 1. gegeben, d.h. die Schritte L_S und B sind abgeschlossen und C wird aktuell ausgeführt. Wir nehmen weiter an, daß eine Ausnahmesituation eintritt, die es notwendig macht, die Aktivität D sofort zu bearbeiten. Alle anderen Reihenfolgenbeziehungen (z.B. E nach D, G nach E und F) sollen aber erhalten bleiben. Da es sich um eine spontane, situationsbedingte Abweichung handelt, soll sie nur temporär auf die WF-Instanz angewendet werden. Dem entsprechend muß geprüft werden, ob die Änderung auf den Prozeßgraphen $P_{current}$ angewendet werden kann. (Der Prozeßgraph $P_{temporary}$ muß im vorliegenden Fall nicht konstruiert werden, da bisher noch keine anderen temporären Änderungen erfolgten.)

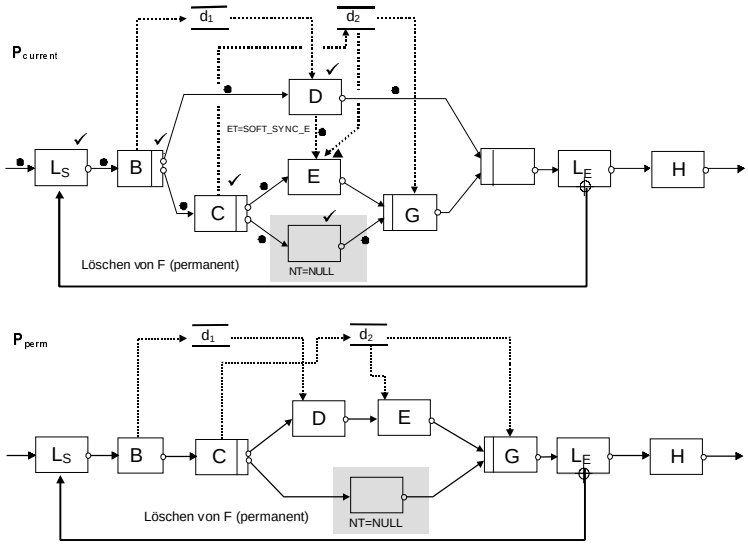
   Die gewünschte Änderung kann mittels der in Kapitel 7 vorgestellten Sprungoperation *jumpForward* (mit n_{target} = D, n_{sync} = E) verwirklicht werden. Dazu wird D aus seiner aktuellen Position entfernt und parallel zu dem durch (B, G) gebildeten Block verschoben. Zusätzlich wird eine einfache Sync-Kante von D nach E eingefügt, um die Reihenfolgenbeziehung zwischen D und E weiter aufrechtzuerhalten. Schließlich wird die Parameterversorgung von D überprüft. Sie kann, auch bei Vorziehen dieses Schrittes, garantiert werden, da er seine Eingabedaten ausschließlich von B erhält und dieser Schritt bereits beendet wurde. Zudem produziert die Aktivität D selbst keine Ausgabedaten, so daß durch ihre vorzeitige Bearbeitung keine parallelen Schreibzugriffe auf Datenelemente entstehen können. Die Änderung wird nun auf den Prozeßgraphen $P_{current}$ angewendet und ein entsprechender ÄT-Eintrag der Änderungshistorie hinzugefügt. Der Graph P_{perm} bleibt gegenüber 1. unverändert. Zusätzlich wird der Status von $P_{current}$ neu bewertet, woraufhin D aktiviert wird.



Die Änderungshistorie besitzt nach der Durchführung dieser temporären Änderung die folgende Gestalt:

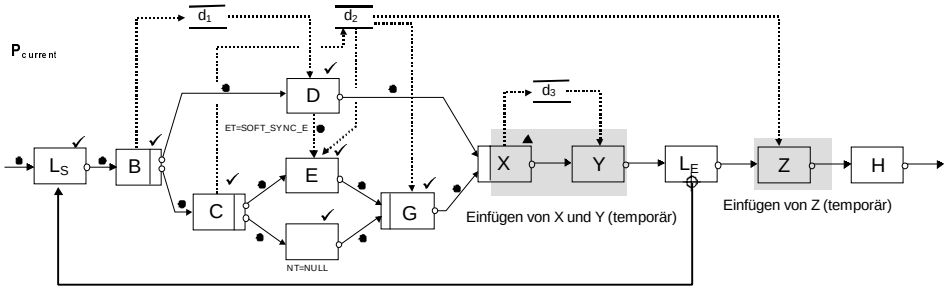
Änderungshistorie (vereinfachte Darstellung)							
TrTimestamp	TrID	TrDuration	TrStartRegion	TrAppliedOps (Liste der angewendeten Änderungsoperationen)			
				OpTimesta	OpType	OpStartReg	OpAppliedPrimitives
1	471	temporary	{C, D}	1	jumpForward n_{target} = D, n_{sync} = E	{C, D}	AddNodes({n_1}); SetNodeAttr(n_1, V_{in}, ALL_Of_ALL) [BeforeImage = ONE_Of_ONE] DeleteEdges({(G, L_E, CONTROL_E); InsertEdges({ (G, n_1, CONTROL_E), (n_1, L_E, CONTROL_E)}); SetNodeAttr(B, V_{out}, ALL_Of_ALL); [BeforeImage = ONE_Of_ONE] DeleteEdges({ (C,D, CONTROL_E), (D, E, CONTROL_E)}); InsertEdges({ (B, D, CONTROL_E), (D, n_1, CONTROL_E), (D, E, SOFT_SYNC_E)});

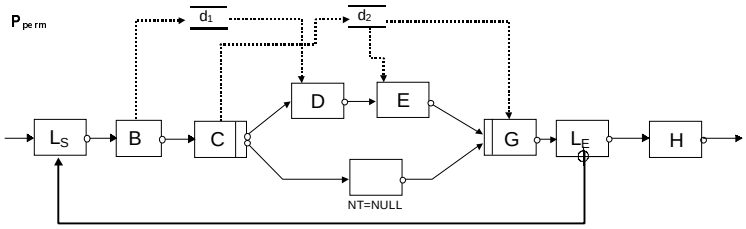
3. Die Aktivitäten C und D seien mittlerweile beendet und die E und F aktiviert. Es soll nun in einer weiteren ÄT der (aktivierte) Knoten F permanent gelöscht werden. Dazu muß geprüft werden, ob das Löschen sowohl für $P_{current}$ als auch für P_{perm} erlaubt ist. Dies trifft im vorliegenden Fall zu, so daß der Koten aus beiden Prozeßgraphen entfernt werden kann (auf Instanzebene durch Substitution mit Nullaktivität realisiert).



Änderungshistorie (vereinfachte Darstellung)							
TrTimestamp	TrID	TrDuration	TrStartRegion	TrAppliedOps (Liste der angewendeten Änderungsoperationen)			
				OpTimesta	OpType	OpStartReg	OpAppliedPrimitives
1	471	temporary	{C, D}	1	jumpForward(n_{target}, n_{sync}) n_{target} = D, n_{sync} = E	{C, D}	...
2	501	permanent	{F_{null}}	1	deleteActivity(F)	{F_{null}}	SetNodeAttr(F, NT, NULL) [BeforeImage = ACTIVITY] SetNodeTempl(F, NULL-Template) [BeforeImage = bisheriges Template]

4. Im weiteren Verlauf werden noch zwei weitere temporäre ÄT auf die WF-Instanz angewendet. Eine davon fügt die Aktivität X und den datenabhängigen Schritt Y ein. Die andere ÄT fügt die Aktivität Z zwischen dem Schleifenendknoten und dem Knoten H temporär ein. Die resultierenden Prozeßgraphen $P_{current}$ und P_{perm} und die Änderungshistorie stellen sich anschließend wie folgt dar:





Änderungshistorie (vereinfachte Darstellung)							
TrTimestamp	TrID	TrDuration	TrStartRegion	TrAppliedOps (Liste der angewendeten Änderungsoperationen)			
				OpTimesta	OpType	OpStartReg	OpAppliedPrimitives
1	471	temporary	{C, D}	1	jumpForward(n_{target}, n_{sync}) n_{target} = D, n_{sync} = E	{C, D}	...
2	501	permanent	{F_{null}}	1	deleteActivity(F)	{F_{null}}	...
3	634	temporary	{Z}	1	serialInsert(Z, L_E, H,...)	{Z}	...
4	723	temporary	{X, Y}	1	serialInsert(X, n_1, L_E, ...}	{X}	...
				2	serialInsert(Y, X, L_E, ...}	{Y}	...

5. Angenommen, es wird im dargestellten Status versucht, direkt nach H eine weitere Aktivität W, die das Datenelement d_3 obligat lesen soll, permanent einzufügen. Da d_3 ausschließlich von der temporär eingefügten Aktivität X geschrieben wird, ist dieses Datenelement im Prozeßgraphen P_{perm} nicht enthalten. Dem entsprechend verweigert das System die Einbringung dieser Änderung.

Im folgenden Abschnitt werden wir zeigen, wie in diesem Beispiel temporäre Änderungen bei der Durchführung von Schleifenrücksprüngen wieder zurückgenommen werden.

8.5 Rücknahme von temporären Änderungen bei Schleifenrücksprüngen

Bisher haben wir beschrieben, wie permanente und temporäre Änderungen korrekt in eine WF-Instanz eingebracht werden können und welche Kontrolldaten dazu erforderlich sind. In diesem Abschnitt geben wir auf ein darauf basierendes Verfahren zur Rücknahme temporärer Änderungen, d.h. ihrer Entfernung aus dem Ausführungsgraphen $P_{current}$ an. Sie kann bei Schleifen- oder Fehlerrücksprüngen not-

wendig werden, da in beiden Fällen ein zuvor bereits abgearbeiteter Bereich des Ausführungsgraphen nochmals durchlaufen wird. Um die nachfolgenden Diskussionen nicht mit unnötigen Details zu überfrachten, schränken wir uns bei der Beschreibung des Verfahrens auf Schleifenrücksprünge ein. Der vorgestellte Ansatz ist vom Prinzip her allerdings auch bei Fehlerrücksprüngen anwendbar [ReDa98].

Im folgenden bezeichne L_S den Startknoten und L_E den Endknoten der Schleife, die infolge der Neumarkierung ihrer Schleifenrücksprungkante erneut durchlaufen werden soll. Der Rücksprungbereich ist in diesem Fall durch die Knotenmenge L_{body} des Schleifenkörpers gegeben:

$$L_{body} := c_succ^*(L_S) \cap c_pred^*(L_E)$$

Das wesentliche Kriterium für die Rücknahme einer temporären ÄT T bei der Durchführung dieses Schleifenrücksprungs ist dann, daß der Startbereich von T vollständig in L_{body} enthalten ist. Formal:

$$\left(TrDuration^T = \texttt{TEMPORARY}\right) \wedge \left(TrStartRegion^T \subseteq L_{body}\right)$$

Am einfachsten wäre es, die Änderungshistorie in umgekehrter Reihenfolge zu durchlaufen und für jeden Historieneintrag, der dieses Kriterium erfüllt, die betreffende Änderung durch die inverse Ausführung ihrer Änderungsoperationen rückgängig zu machen. Dieser Ansatz würde in Verbindung mit dem vorgestellten optimistischen Verfahren zur Einbringung temporärer Änderungen nicht immer zu einem korrekten Ergebnis führen. Der Grund dafür ist, daß wir Abhängigkeiten zwischen temporären Änderungen, die nicht innerhalb desselben Schleifenkörpers liegen, generell nicht ausgeschlossen haben. Dadurch kann es bei der Rücknahme einer temporären ÄT in sehr seltenen Fällen notwendig werden, abhängige temporäre Änderungen (mit größerem Zeitstempel) ebenfalls zurückzunehmen bzw. in modifizierter Form wieder einzubringen, um Inkonsistenzen oder fehlerhafte Prozeßgraphen zu vermeiden. Aufgrund der in Abschnitt 8.4.3 geforderten Bedingung (☆) ist in einem solchen Fall gewährleistet, daß die von der Rücknahme betroffenen Änderungen in jedem Fall in einem Bereich des Prozeßgraphen liegen, der nach der Durchführung des Rücksprungs unmarkiert ist. Dadurch können wir zum Beispiel ausschließen, daß ein temporär eingefügter Knoten, der aktuell als `COMPLETED` oder `RUNNING` markiert ist, entfernt werden muß.

Darüber hinaus muß berücksichtigt werden, daß bei der Anwendung einer temporären oder permanenten ÄT auf den Prozeßgraphen $P_{current}$ auch „unkritische“ Abhängigkeiten zu vorangehend durchgeführten, temporären Änderungen entstehen können. Ein Beispiel hierzu zeigt Abbildung 8-5. Hier wird in den Ausführungsgraphen $P_{current}$ zuerst der Schritt `X` seriell zwischen `A` und `B` (*serialInsert*) von einer temporären ÄT T_1 eingefügt. Anschließend wird der Knoten `Y` parallel zum Knoten `B` (*parallelInsert*) von einer permanenten ÄT T_2 hinzugenommen. Wie man leicht sieht, wurde von T_2 u.a. die Ausgangssemantik des Knotens `X` modifiziert und die Kontrollkante `X` → `Y` eingefügt. Es wird also bei der Einbettung des parallelen Knotens auf einen anderen Knoten Bezug genommen, der später infolge der Rücknahme der temporären Änderung T_1 nicht mehr im Prozeßgraphen $P_{current}$ präsent ist. Um solche und ähnliche Fälle bei Rücksprüngen erkennen und durch das Verfahren korrekt behandeln zu können, müssen bei der Rücknahme einer temporären ÄT T ggf. auch andere, nicht (vollständig) im Rücksprungbereich liegende Änderungstransaktionen (mit größerem Zeitstempel als T) zunächst ebenfalls zurückgenommen und anschließend wieder neu in den Prozeßgraphen eingespielt werden, erforderlichenfalls mit einem modifizierten Satz an Änderungsprimitiven. Für unser Beispiel etwa ergibt sich nach einem Schleifenrücksprung der in Abbildung 8-5 d) dargestellte Graph. Die Rücknahme und das Wiedereinspielen einer ÄT ist im allgemeinen nur erforderlich, wenn sie zeitlich nach der Durchführung der ältesten zurückzunehmenden ÄT auf den Prozeßgraphen $P_{current}$ angewendet wurde und wenn der von ihr modifizierte Bereich im Rücksprungbereich oder ihm nachfolgenden Regionen liegt.

Des weiteren muß berücksichtigt werden, daß sich eine ÄT eventuell aus mehreren Basisänderungsoperationen zusammensetzt, die möglicherweise auf unterschiedlichen, nicht notwendigerweise zusammenhängenden Bereichen des Prozeßgraphen $P_{current}$ angewendet wurden. Beispielsweise können durch eine ÄT zwei Aktivitäten X und Y in $P_{current}$ eingefügt worden sein, die unterschiedlichen parallelen Zweigen angehören oder von denen X ein Vorgänger von L_S und Y ein Nachfolger dieses Knotens ist. In letzterem Fall darf beim Schleifenrücksprung nur die Einfügeoperation für den Knoten Y zurückgenommen und wieder eingespielt werden (sog. partielles *Undo*/*Redo*), während die Einfügung von X, die in einem als abgearbeitet markierten Teil des Ausführungsgraphen liegt, unverändert erhalten bleiben muß. Um die Atomarität der ÄT weiterhin zu gewährleisten, muß bei einem solchen

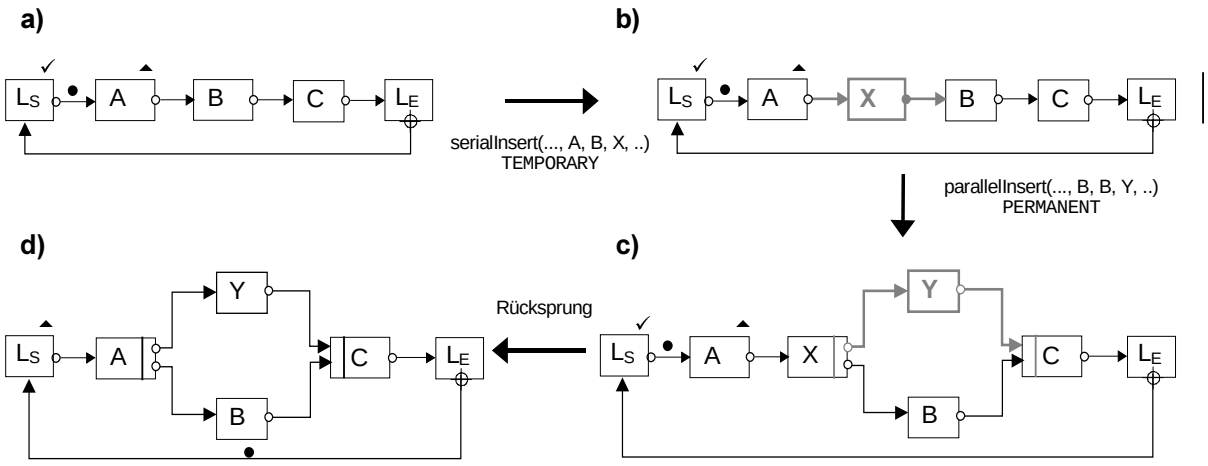


Abbildung 8-5: *Bezugnahme zwischen temporären und permanenten Änderungen*

partiellen *Undo*/*Redo* sichergestellt sein, daß das *Redo* der zurückgenommenen Operationen (im Beispiel das Wiedereinfügen von *Y*) ohne Probleme und automatisch durch das WfMS erfolgen kann.

Wir stellen nun ein Verfahren vor, das diese Fälle berücksichtigt. Grundlegend hierfür sind die Änderungshistorie der WF-Instanz und die von ihr verwalteten Daten (vgl. Abschnitt 8.2). Dabei nehmen wir an, daß die Änderungshistorie in einer nach Transaktionszeitstempeln aufsteigend sortierten Liste organisiert ist, d.h. der aktuellste Historieneintrag befindet sich am Ende der Liste.

8.5.1 Bestimmung der betroffenen Änderungstransaktionen

Im folgenden sei durch (L_S, L_E) eine Schleife und durch L_{body} die Knotenmenge des zugehörigen Schleifenkörpers gegeben. Wir geben zuerst an, welche temporären Änderungstransaktionen beim Signalisieren der Schleifenkante $L_E \rightarrow L_S$ dauerhaft entfernt werden müssen (*Undo*) und welche temporären und permanenten Änderungstransaktionen zunächst ganz oder teilweise zurückgenommen und anschließend wieder neu in den Graphen $P_{current}$ eingespielt werden sollen (*Undo*/*Redo*).

In jedem Fall rückgängig gemacht werden müssen diejenigen temporären Änderungen, deren Startbereich vollständig im Rücksprungbereich liegt. Sie bilden das sog. *UndoSet* des Schleifenrücksprungs:

$$UndoSet := \{T \in ChangeHistory \mid (TrDuration^T = \texttt{TEMPORARY}) \wedge (TrStartRegion^T \subseteq L_{body})\}$$

Falls das *UndoSet* gleich der leeren Menge ist, muß keine ÄT zurückgenommen werden. Andernfalls bezeichne T_{fix} die älteste ÄT aus dieser Menge, d.h. es soll gelten: ($T_{fix} \in UndoSet$ und $\forall\, T \in UndoSet$: $TrTimestamp^{T_{fix}} \leq TrTimestamp^T$).

Für Änderungen, die auf der Grundlage von Änderungstransaktionen der Menge *UndoSet* eingebracht worden sind, die aber selbst nicht dieser Menge angehören, muß geprüft werden, ob sie nach der Rücknahme der Transaktionen aus dem *UndoSet* unverändert anwendbar bleiben. Zu diesem Zweck ermitteln wir die Menge derjenigen temporären und permanenten Änderungstransaktionen, die infolge des Schleifenrücksprungs und der damit verbundenen Rücknahme von temporären Änderungen zunächst ganz oder teilweise zurückgenommen und anschließend – falls möglich – wieder eingespielt werden sollen. Sie bilden das sog. *RedoSet* des Rücksprungs. Gemäß obigen Überlegungen kann das *RedoSet* wie folgt definiert werden (falls $UndoSet \neq \emptyset$ gilt):

$$RedoSet := \{T \in ChangeHistory \mid T \notin UndoSet \wedge TrTimestamp^{T_{fix}} < TrTimestamp^T \wedge (\exists\, Op \in TrAppliedOps^T \text{ mit } \{L_S\} \in pred(n)\ \forall\, n \in OpStartRegion^{Op})\}$$

Das *RedoSet* umfaßt alle Änderungstransaktionen, die zeitlich nach T_{fix} in den Prozeßgraphen $P_{current}$ eingebracht wurden und die mindestens eine Änderungsoperation besitzen, deren Startbereich dem

Knoten L_S nachfolgt[8]. Nicht im *RedoSet* enthalten sind die Änderungstransaktionen der Menge *UndoSet* selbst. Für ein einzelnes $T \in RedoSet$ müssen beim Schleifenrücksprung die folgenden Operationen zurückgenommen und – soweit möglich – anschließend wieder neu eingespielt werden:

$$OpsRedoSet^T := \{ Op \in TrAppliedOps^T \mid \{L_S\} \in pred(n)\ \forall\ n \in OpStartRegion^{Op} \} \subseteq TrAppliedOps^T$$

In Bezug auf Änderungstransaktionen aus der Menge *RedoSet* unterscheiden wir die folgenden Fälle:

Fall 1: *Vollständiges Undo / Redo von T* ($OpsRedoSet^T = TrAppliedOps^T$)

Stimmen die beiden Mengen $OpsRedoSet^T$ und $TrAppliedOps^T$ überein, so wird die ÄT *T* bei dem nachfolgend skizzierten Verfahren zuerst vollständig zurückgenommen und später versucht, sie wieder in den Ausführungsgraphen $P_{current}$ einzuspielen. Grundsätzlich stellt sich die Frage, unter welchen Bedingungen die Wiedereinbringung von *T* möglich ist und wann Schwierigkeiten auftreten können. Permanente Änderungstransaktionen können ohne Probleme wieder in den Prozeßgraphen $P_{current}$ eingebracht werden. Die Wiedereinbringung einer temporären ÄT dagegen kann (muß aber nicht) zu Problemen führen, wenn sie in der folgenden Teilmenge des *RedoSet* enthalten ist (vgl. hierzu Bedingung (☆) aus Abschnitt 8.4.3.2)

$$RedoSet_{critical} := \{T^* \in RedoSet \mid TrDuration^T = \texttt{TEMPORARY} \wedge (\exists\ T \in UndoSet \text{ mit } TrTimestamp^T < TrTimestamp^{T^*} \wedge TrStartRegion^T \prec TrStartRegion^{T^*})\}$$

Bei der Bildung dieser Menge nutzen wir aus, daß eine temporäre ÄT T^* nur dann von einer anderen temporären ÄT *T* abhängig und dem entsprechend durch deren Rücknahme beeinträchtigt sein kann, wenn T^* zeitlich nach *T* in die WF-Instanz eingebracht worden ist und wenn der Startbereich von T^* dem Startbereich von *T* „nachfolgt“ im Sinne von Bedingung (☆) (vgl. Abschnitt 8.4.3). Für alle anderen Fälle ist die Wiedereinbringung temporärer Änderungen unkritisch, auch wenn ggf. ein geänderter Satz an Basisänderungsprimitiven angewendet werden muß.

Fall 2: *Partielles Undo/Redo von T* ($OpsRedoSet^T \subset TrAppliedOps^T$)

In diesem Fall muß lediglich eine Teilmenge der Änderungsoperationen von *T* zurückgenommen und anschließend wieder eingespielt werden. Dabei muß gewährleistet sein, daß die Wiedereinbringung der Operationen in den Prozeßgraphen gelingt:

- *T ist permanente ÄT*: Da Abhängigkeiten permanenter Änderungen von zuvor durchgeführten temporären ÄT (vgl. Abschnitt 8.4.2) durch das hier angewandte Verfahren ausgeschlossen sind und permanente Änderungen in der Reihenfolge ihrer ursprünglichen Durchführung wiederhergestellt werden (siehe unten), ist die Wiedereinbringung der Operationen in $P_{current}$ möglich.
- *T ist temporäre ÄT*: Die Wiedereinbringung der zurückgenommenen Änderungen in $P_{current}$ wäre nach der Konstruktion des Verfahrens nur dann problematisch, wenn beim Schleifenrücksprung eine andere ÄT T^* mit kleinerem Zeitstempel als *T* zurückgenommen werden und die Bedingung $TrStartRegion^{T^*} \prec TrStartRegion^T$ gelten würde. Wegen $OpsRedoSet^T \subset TrAppliedOps^T$ kann diese Situation jedoch nicht eintreten. Es gibt mindestens eine Änderungsoperation aus der Menge $TrAppliedOps^T$, deren Startbereich nicht im Nachfolgebereich des Knotens L_S liegt. Da von der Rücknahme nur temporäre ÄT betroffen sein können, deren Startbereich dem Knoten L_S nachfolgt, kann es ein solches T^* nicht geben.

Auf der Grundlage dieser Überlegungen zerlegen wir das *RedoSet* in die beiden disjunkten Teilmengen $RedoSet_{critical}$ und $RedoSet_{uncritical} := RedoSet - RedoSet_{critical}$.

Beispiel 8-7 (UndoSet und RedoSet)

Wir beziehen uns auf den 4. Schritt von Beispiel 8-6 an. Für die Schleife (L_S, L_E) mit L_{body}:= {B, C, D, E, F_{null}, G, X, Y} erhalten wir im aktuellen Zustand die Mengen *UndoSet* = {471, 723} und *RedoSet* = {501, 634}. Ferner ergibt sich $RedoSet_{critical}$ = {634} und $RedoSet_{uncritical}$ = {501}. Älteste zurückzunehmende ÄT ist T_{fix} = 471.

[8] Hier besteht potentiell eine Abhängigkeit zu einer vorangehend durchgeführten und infolge des Schleifenrücksprungs zurückzusetzenden, temporären ÄT. Für alle anderen Fälle sind solche Abhängigkeiten ausgeschlossen.

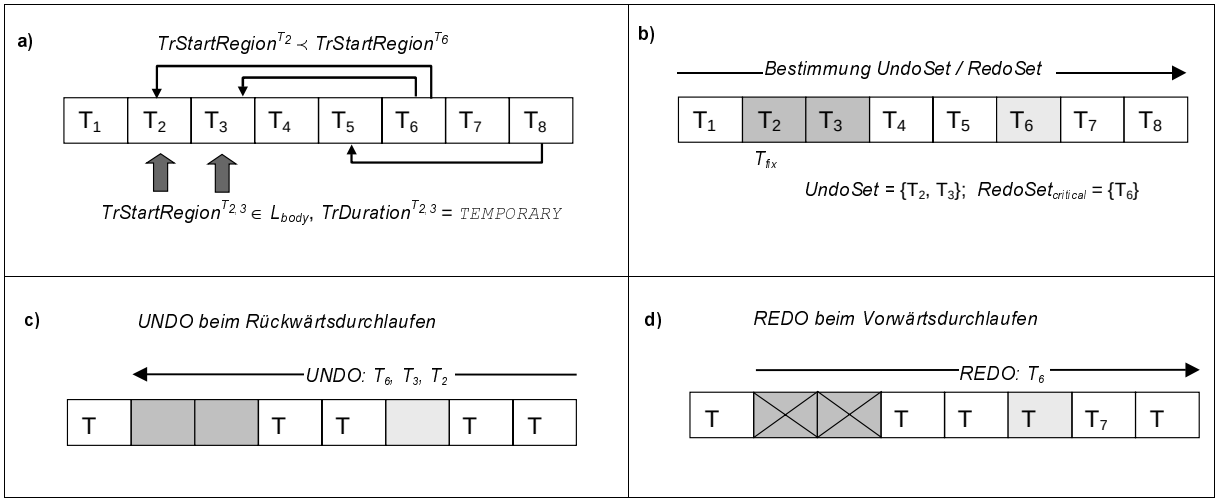


Abbildung 8-6: *Vereinfachte Darstellung des Undo/Redo-Verfahrens zur Rücknahme von Änderungstransaktionen bei einem Schleifenrücksprung: a) Änderungshistorie zum Zeitpunkt des Schleifenrücksprungs ($TrTimestamp^{T_i} < TrTimestamp^{T_j}$ für $i < j$); b) Bestimmung der Mengen UndoSet und RedoSet; c) Undo aller Transaktionen aus $UndoSet \cup RedoSet$; (d) Versuch der Wiedereinbringung von Transaktionen der Menge RedoSet.*

Ein Algorithmus zur Bestimmung von *UndoSet*, $RedoSet_{critical}$ und $RedoSet_{uncritical}$ kann dem Anhang C.6 entnommen werden. Neben diesen Mengen, die in einem Durchlauf durch die Änderungshistorie ermittelt werden, bestimmt dieser Algorithmus den „Fixpunkt" T_{fix} und für jedes $T \in RedoSet$ die Menge der zurückzunehmenden Änderungoperationen $OpsRedoSet^T \subseteq TrAppliedOps^T$.

8.5.2 Verfahren zur korrekten Rücknahme temporärer Änderungen

Die Rücknahme temporärer Änderungen basiert auf einem kombinierten UNDO-REDO-Verfahren. Zuerst werden die Mengen *UndoSet* und *RedoSet* ($RedoSet_{uncritical}$ und $RedoSet_{critical}$), wie beschrieben, ermittelt. Dann wird die Änderungshistorie rückwärts, d.h. beginnend mit dem aktuellsten Eintrag, bis zum Fixpunkt T_{fix} (d.h. der ÄT mit kleinstem Zeitstempel, die zurückgenommen werden muß) durchlaufen. Dabei werden für jeden traversierten Historieneintrag, der eine ÄT aus der Menge $UndoSet \cup RedoSet$ beschreibt, die protokollierten Änderungen rückgängig gemacht[9] (UNDO). Für Änderungstransaktionen aus dem *RedoSet* erfolgt ggf. nur ein partielles UNDO, bei dem nur eine Teilmenge der bei der Ausführung der ÄT zur Anwendung gekommenen Änderungsoperationen zurückgenommen wird (vgl. die Diskussion eingangs von Abschnitt 8.5). Das Verfahren ist so konstruiert, daß die von der Rücknahme der Änderungen betroffenen Bereiche des Ausführungsgraphen nach der Durchführung des Schleifenrücksprungs in jedem Fall unmarkiert sind. Dadurch werden keine weitergehenden Maßnahmen, wie z.B. der Abbruch laufender Prozeßschritte, erforderlich.

Im Anschluß an dieses UNDO wird die Historie, beginnend mit dem Eintrag von T_{fix}, in Vorwärtsrichtung durchlaufen. Für jeden traversierten Eintrag einer ÄT T werden folgende Aktionen ausgeführt:

- Gehört T der Menge *UndoSet* an, so wird der zugehörige Eintrag aus der Historie entfernt und die temporäre ÄT T nicht mehr in $P_{current}$ eingebracht.
- Gehört T der Menge *RedoSet* an, wird versucht, die zuvor zurückgenommenen Änderungsoperationen aus $OpsRedoSet^T$ wieder in den Ausführungsgraphen der Instanz einzuspielen (REDO). Wie bereits erwähnt, ist dies in der Mehrzahl der Fälle möglich, wobei ggf. ein geänderter Satz an Basisänderungsprimitiven angewendet werden muß.
- Gilt $T \notin UndoSet \cup RedoSet$, so werden keine Aktionen durchgeführt.

[9] Eine Ausnahme bildet in diesem Zusammenhang die Änderungsoperation *insertDataElement* (vgl. Abschnitt 7.5.4). Sie wird beim UNDO einer ÄT gesondert behandelt und nur zurückgenommen, wenn auf das betreffende Datenelement noch kein Schreibzugriff stattgefunden hat. Andernfalls bleibt das Datenelement im Datenkontextschema der WF-Instanz erhalten, ebenso kann auf seine Werte weiter (lesend) zugegriffen werden kann.

Das Prinzip dieses Verfahrens illustriert Abbildung 8-6. Wir haben dabei eine vereinfachte Darstellung gewählt, die von den diskutierten Details (z.B. partielles `UNDO` / `REDO`) zum Teil abstrahiert.

Wir fassen nun das Verfahren für die Rücknahme temporärer Änderungen zusammen, verzichten dabei aber auf die Darstellung von Implementierungsaspekten. Des weiteren nehmen wir an, daß das Verfahren zunächst auf eine Kopie von $P_{current}$ angewendet wird, bevor die ermittelten Anpassungen tatsächlich erfolgen.

---

1. Durchlaufe die Änderungshistorie, beginnend mit dem ältesten Eintrag, in Vorwärtsrichtung und ermittle dabei die Mengen *UndoSet*, *RedoSet*, $RedoSet_{uncritical}$ und $RedoSet_{critical}$, die ÄT T_{fix} sowie für jedes $T \in RedoSet$ die Menge $OpsRedoSet^T$ der zurückzunehmenden Änderungsoperationen. (T_{fix} ist die älteste ÄT aus dem *UndoSet*, d.h. die ÄT mit dem kleinsten Zeitstempel).

   Ist *UndoSet* gleich der leeren Menge, sind wir fertig. Für diesen Fall müssen keine ÄT zurückgenommen werden. Andernfalls wird mit Schritt 2 des Verfahrens fortgefahren

2. Durchlaufe die Änderungshistorie – beginnend mit dem aktuellsten Eintrag – in Rückwärtsrichtung bis zum Eintrag von T_{fix}. Führe für jeden Historieneintrag einer ÄT $T \in UndoSet \cup RedoSet$ ein vollständiges bzw. partielles `UNDO` von T bzgl. $P_{current}$ durch:

   - Für $T \in UndoSet$ erfolgt ein vollständiges `UNDO`, indem alle bei der Ausführung von T angewandten Änderungsoperationen $TrAppliedOps^T$ in der umgekehrten Reihenfolge ihrer Einbringung (d.h. absteigend nach Operations-Zeitstempel) zurückgesetzt werden. (Wie oben erwähnt, erfordert die Rücknahme der Operation *insertDataElement* eine gesonderte Behandlung. Sie wird nur vorgenommen, wenn auf das betreffende Datenelement noch keine Schreibzugriffe stattgefunden haben.)

   - Für $T \in RedoSet$ erfolgt ein vollständiges oder partielles `UNDO`, abhängig von der konkreten Festlegung der Menge $OpsRedoSet^T \subseteq TrAppliedOps^T$. Es werden alle Änderungsoperationen aus $OpsRedoSet^T$ in der umgekehrten Reihenfolge ihrer Einbringung zurückgesetzt.

   In beiden Fällen sind die von der Rücknahme betroffenen Regionen des Prozeßgraphen $P_{current}$ nicht markiert (Knotenmarkierungen `NOT_ACTIVATED`).

   Die Rücknahme einer Änderungsoperation $Op \in TrAppliedOps^T$ besteht aus der inversen Anwendung der bei ihrer Einbringung verwendeten Änderungsprimitive $OpAppliedPrimitives^{Op}$. Beispielsweise können die aus der Anwendung des Änderungsprimitivs *DeleteEdges* resultierenden Effekte durch Ausführung des inversen Primitivs *InsertEdges* aufgehoben werden. Die Rücknahme von Primitiven wie *SetNodeAttr* oder *SetEdgeAttr* erfolgt durch Einspielen der *Before-Images*.

   Es werden somit zunächst also alle Änderungstransaktionen, deren Zeitstempel größer oder gleich dem von T_{fix} ist und die der Menge $UndoSet \cup RedoSet$ angehören, vollständig bzw. partiell zurückgenommen. Im Schritt 3 des Verfahrens wird dann versucht, die für die Änderungstransaktionen der Menge *RedoSet* zurückgenommenen Änderungen wieder einzuspielen. Für Änderungstransaktionen der Menge $RedoSet_{uncritical}$ ist dies aufgrund der Konstruktion des Verfahrens ohne Schwierigkeiten möglich. In der Mehrzahl der Fälle gelingt das Wiedereinbringen auch für Änderungstransaktionen der Menge $RedoSet_{critical}$. Lediglich in Einzelfällen, insbesondere bei existentiellen Abhängigkeiten zwischen temporären Änderungen (z.B. Datenabhängigkeiten), ist die Wiedereinbringung möglicherweise kritisch.

3. Durchlaufe die Änderungshistorie – ausgehend von dem Eintrag für T_{fix} – nun in *Vorwärtsrichtung* und verfahre mit jedem traversierten Eintrag einer ÄT T wie folgt:

   - $T \in UndoSet$: Beschreibt der aktuelle Historieneintrag eine temporäre ÄT, die im 2. Schritt des Verfahrens vollständig zurückgenommen wurde und die der Menge *UndoSet* angehört, so wird die Änderung nicht mehr in den Ausführungsgraphen $P_{current}$ eingebracht und ihr Historieneintrag wird entfernt (Anmerkung: $ADEPT_{flex}$ entfernt diesen Eintrag nicht physisch aus der Historie, sondern markiert ihn lediglich als gelöscht).

   - $T \in RedoSet$: Beschreibt der aktuelle Historieneintrag eine ÄT, die im 2. Schritt des Verfahrens vollständig oder teilweise zurückgenommen wurde und die der Menge *RedoSet* angehört,

so wird versucht, die Änderung wieder in den Prozeßgraphen $P_{current}$ einzuspielen. Dabei unterscheiden wir die folgenden Fälle:

- $T \in RedoSet_{uncritical}$: Beschreibt der aktuelle Historieneintrag eine permanente oder temporäre ÄT, die in keiner kritischen Abhängigkeitsbeziehung zu einer zurückgenommenen temporären ÄT steht, so kann T ohne Schwierigkeiten wieder in $P_{current}$ eingebracht werden. Dazu werden die zuvor zurückgenommenen Änderungsoperationen der Menge $OpsRedoSet^T$ in der ursprünglichen Reihenfolge ihrer Einbringung, d.h. aufsteigend nach ihren Zeitstempeln, wieder auf $P_{current}$ angewendet. Aufwendige Überprüfungen, wie sie bei der erstmaligen Einbringung von T notwendig waren, können entfallen. Insbesondere ist keine Überprüfung der Korrektheit des Datenflußes mehr notwendig.

  Zur Wiedereinbringung einer einzelnen Änderungsoperation $Op \in OpsRedoSet^T$ werden nicht immer die in der Änderungshistorie protokollierten Änderungsprimitive $OpAppliedPrimitives^{Op}$ verwendet, sondern die Änderungsoperation wird erneut auf den Prozeßgraphen $P_{current}$ mit dem dafür vorgesehenen Modifikationsalgorithmus angewendet. Dabei kann sich für gewisse Operationen (z.B. Einfügen einer Aktivität zwischen zwei Knotenmengen) gegenüber $OpAppliedPrimitives^{Op}$ ein veränderter Satz an Primitiven ergeben, abhängig davon, welche temporären Änderungen zuvor aus $P_{current}$ entfernt wurden.

  Nach der erfolgreichen Wiedereinbringung von T bleibt der bisherige Historieneintrag dieser Transaktion mit unverändertem Transaktionszeitstempel in der Änderungshistorie erhalten. Die Menge der zur Umsetzung einzelner Änderungsoperationen angewandten Änderungsprimitive wird ggf. angepaßt.

- $T \in RedoSet_{critical}$: Beschreibt der aktuelle Historieneintrag eine temporäre ÄT, die potentiell in einer kritischen Abhängigkeitsbeziehung zu einer vorangehend zurückgenommenen, temporären ÄT steht (vgl. obige Definition der Menge $RedoSet_{critical}$), so muß geprüft werden, ob T wieder korrekt in den Prozeßgraphen $P_{current}$ eingespielt werden kann. Im Gegensatz zu dem gerade beschriebenen Fall sind ähnliche Überprüfungen durchzuführen, wie bei der erstmaligen Einbringung von T. Dies schließt DF-Analysen mit ein.

  Zur Wiedereinbringung der Operationen aus $TrAppliedOps^T$ wird das in Abschnitt 8.4.3 beschriebene Verfahren in optimierter Form (siehe Abschnitt 8.5.3) angewendet. Ist die Wiedereinbringung möglich, so wird die Änderung – ggf. in angepaßter Form und auf der Grundlage eines anderen Satzes and Änderungsprimitiven – auf $P_{current}$ durchgeführt. Für diesen Fall bleibt der bisherige Historieneintrag von T mit unverändertem Transaktionszeitstempel in der Änderungshistorie erhalten. Die Mengen der Änderungsprimitive zur Umsetzung einzelner Änderungsoperationen werden ggf. neu festgelegt.

  In seltenen Fällen kann der Versuch scheitern, die temporäre Änderung T erneut einzubringen, etwa wenn auftretende Probleme nicht automatisch durch das WfMS behoben werden können. Die Wiedereinbringung wird dann abgebrochen und der entsprechende Eintrag von T aus der Änderungshistorie entfernt. Zusätzlich werden der Initiator der ÄT ($TrInitiator^T$) und der Prozeßverantwortliche informiert. Dies erfolgt asynchron, so daß es zu keinen Blockierungen kommt. Dem Benutzer steht es dann frei, geeignet zu reagieren und die temporäre Änderung z.B. in anderer Form, d.h. durch Anwendung einer anderen ÄT, vorzunehmen. Wie bereits erwähnt, sind solche Fälle eher theoretischer Natur, müssen aber einkalkuliert werden, um die Korrektheit und Konsistenz des Ausführungsgraphen jederzeit gewährleisten zu können. Aufgrund der Definition der Menge $RedoSet_{critical}$ ist gewährleistet, daß die Nicht-Wiedereinbringung von T für alle Änderungsoperationen dieser ÄT gilt und, daß der von der Rücknahme betroffene Bereich des Prozeßgraphen unmarkiert ist.

- $T \notin UndoSet \cup RedoSet$: Der aktuelle traversierte Historieneintrag beschreibt eine ÄT, die vom Schleifenrücksprung nicht betroffen ist (z.B. eine ÄT deren Startbereich in einem parallelen Ausführungszweig liegt). Dem entsprechend muß nichts unternommen werden.

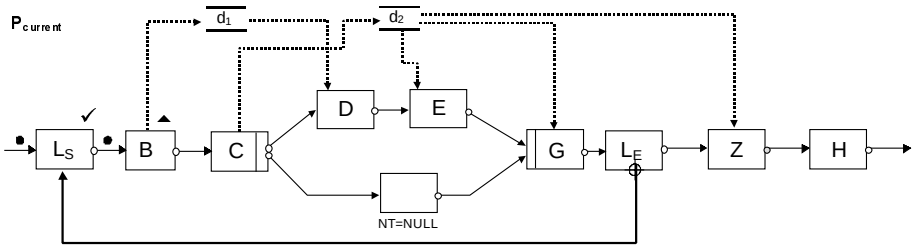
Wir illustrieren dieses Verfahren anhand zweier Beispiele, klammern dabei aber den Fall der partiellen Rücknahme einer ÄT der Einfachheithalber aus:

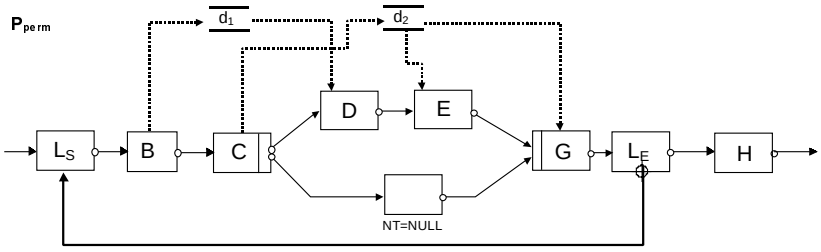
Beispiel 8-8 (Rücknahme temporärer Änderungen)

Wir wenden dieses Verfahren auf das Beispiel 8-6, im Anschluß an die Durchführung von Schritt 4, an. Die aktuelle Iteration der Schleife sei beendet und die Schleifenrücksprungkante mit TRUE signalisiert worden. Bevor der Schleifenkörper erneut durchlaufen wird, müssen ggf. temporäre Änderungen entfernt werden.

- Es gilt: L_{body}:= {B, C, D, E, F_{null}, G, X, Y}. Im 1. Schritt des Verfahrens wird die Historie in Vorwärtsrichtung durchlaufen. Wir erhalten: *UndoSet* = {471, 723}, *RedoSet* = {501, 634} mit $RedoSet_{critical}$ = {634} und $RedoSet_{uncritical}$ = {501}. Ferner ergibt sich: T_{fix} = 471 (hier 1. Eintrag der Historie).
- Im 2. Schritt wird die Historie, beginnend mit dem aktuellsten Eintrag (*TrId* = 723), rückwärts bis T_{fix} = 471 durchlaufen. Jede traversierte ÄT, die in der Knotenmenge *UndoSet* ∪ *RedoSet* = {471, 723, 634, 501} enthalten ist, wird zurückgesetzt. Dies erfolgt, indem ihre Änderungsoperationen in umgekehrter Reihenfolge ihrer Einbringung zurückgenommen werden. Die Rücknahme einer einzelnen Operation basiert dabei auf der inversen Anwendung der zugrundeliegenden Änderungsprimitive.
- Nun wird die Historie, beginnend mit dem Eintrag der ÄT T_{fix} = 471, vorwärts durchlaufen:
  - Die ÄT mit *TrId* = 471 gehört der Menge *UndoSet* an. Sie wird deshalb nicht mehr eingebracht und ihr Eintrag aus der Historie gelöscht bzw. als gelöscht markiert.
  - Die ÄT mit *TrId* = 501 gehört der Menge $RedoSet_{uncritical}$ an, da keine Abhängigkeiten zur ÄT 471 bestehen. Dem entsprechend kann diese ÄT ohne Schwierigkeiten wieder in $P_{current}$ eingebracht werden (hier mit demselben Satz an Änderungsprimitiven).
  - Die ÄT mit *TrId* = 634 ist in der Menge $RedoSet_{critical}$ enthalten, da eine potentielle Abhängigkeit zur ÄT 471 vorliegt. Dem entsprechend wird geprüft, ob die ÄT wieder in den Prozeßgraphen $P_{current}$ eingebracht werden kann. Dabei wird auch die Versorgung des Lesezugriffs von Z auf das Datenelement d_2 überprüft. Nach Abschluß dieser Überprüfungen steht fest, daß die Wiedereinbringung möglich ist.
  - Die ÄT mit *TrId* = 723 ist in der Menge *UndoSet* enthalten. Die ÄT wird deshalb nicht mehr eingebracht und ihr Eintrag aus der Historie gelöscht.

Es erbeben sich die beiden folgenden Prozeßgraphen:





Änderungshistorie (vereinfachte Darstellung)							
TrTimestamp	TrID	TrDuration	TrStartRegion	TrAppliedOps (Liste der angewendeten Änderungsoperationen)			
				OpTimesta	OpType	OpStartReg	OpAppliedPrimitives
2	501	permanent	{F_{null}}	1	deleteActivity(F)	{F_{null}}	...
3	634	temporary	{Z}	1	serialInsert(Z, L_E, H,...)	{Z}	...

Das Beispiel zeigt schön, wie temporäre Änderungen des Ausführungsgraphen einer WF-Instanz im Verlauf der Ausführung wieder „verschwinden“. Das folgende Beispiel zeigt, daß im Schritt 2 des Verfahrens Änderungstransaktionen nicht zurückgenommen werden, wenn ihre Änderungen Bereiche des Ausführungsgraphen betreffen, die parallel zum aktuellen Rücksprungbereich liegen.

Beispiel 8-9 (Rücknahme von Änderungen bei parallelen Ausführungszweigen)

Wir wenden das Rücknahmeverfahren auf den Ausführungsgraphen aus Abbildung 8-2 a) an. Hier soll die innere Schleife nochmals durchlaufen werden. Das *UndoSet* dieses Rücksprung wird durch die ÄT gebildet, die den Knoten X temporär eingefügt hat. Das *RedoSet* bzw. $RedoSet_{critical}$ entspricht der ÄT, die den Knoten Y temporär eingefügt hat. Abbildung 8-2 b) zeigt den Ausführungsgraphen nach Durchführung des Rücksprung. Die Einfügung von X wurde zurückgenommen, während die von Y wieder erfolgreich eingespielt werden konnte. Erwähnenswert ist, daß die in einem Parallelzweig erfolgte temporäre Löschung des Knotens E weiterhin bestehen bleibt, unabhängig davon an welcher Position in der Änderungshistorie sich der betreffende ÄT-Eintrag befindet. Diese ÄT ist weder im *UndoSet* noch im *RedoSet* des beschriebenen Schleifenrücksprungs enthalten.

8.5.3 Optimierungsmöglichkeiten

In der Regel sind für die Wiedereinbringung von Änderungstransaktionen der Menge $RedoSet_{critical}$ im Schritt 3 des Verfahrens – im Gegensatz zu ihrer erstmaligen Anwendung – keine vollständigen KF- und DF-Analysen erforderlich. Statt dessen muß geprüft werden, ob es zwischen einer wiedereinzubringenden ÄT T_{retry} und den zuvor nicht mehr eingebrachten (temporären) Änderungstransaktionen (mit kleinerem Zeitstempel) potentielle Abhängigkeiten gibt. Dies setzt voraus, daß geeignete Konfliktrelationen und -tests definiert sind. Bezogen auf den KF-Graphen der Instanz liegt z.B. ein potentieller Konflikt zwischen der ÄT T_{retry} und einer nicht mehr eingebrachten ÄT T mit kleinerem Zeitstempel ($TrTimestamp^{T} < TrTimestamp^{T_{retry}}$) vor, wenn bei der Rücknahme von T ein Knoten entfernt wird, der von T_{retry} referenziert wird (z.B. von einer von dieser ÄT eingefügten Kante). Was das DF-Modell der Instanz angelangt, liegt zum Beispiel ein Konflikt zwischen T_{retry} und einer nicht mehr eingebrachten ÄT T ($TrTimestamp^{T} < TrTimestamp^{T_{retry}}$) vor, wenn bei der Rücknahme von T eine Schreibkante auf ein Datenelement entfernt wird, auf das die ÄT T_{retry} eine obligate Lesekante einfügen will. Tritt kein solcher DF-Konflikt auf, sind zur Überprüfung der Versorgung obligater Eingabeparameter (Regel DF-1) keine weiteren Analysen notwendig. Andernfalls muß geprüft werden, ob das betreffende Datenelement bereits geschrieben worden ist bzw. anderweitig versorgt werden kann. Ähnlich können Konfliktrelationen für andere Modellaspekte definiert und zur effizienten Wiedereinbringung von Änderungstransaktionen verwendet werden.

8.6 Anwendung des Verfahrens bei hierarchischen Workflows

In diesem Abschnitt diskutieren wir informell einige wichtige Aspekte, die sich bei Anwendung des Verfahrens auf hierarchische Workflows ergeben. Wie in den Kapiteln 4 und 5 beschrieben, besitzt jede WF-Instanz, unabhängig davon, ob sie einen Toplevel- oder einen Sub-WF darstellt, ihren eigenen Ausführungsgraphen und Datenkontext. Zusätzlich werden alle relevanten Ausführungsereignisse in einer instanzspezifischen Ablaufhistorie protokolliert (vgl. Kapitel 6). Die hierarchischen Beziehungen zwischen (Sub-) WF-Instanzen, die im Kontext desselben Gesamt-WF zur Ausführung kommen, werden über den Instanzabhängigkeitsgraph IAG (siehe Kapitel 4) verwaltet. Zusätzlich unterhält $ADEPT_{flex}$ für jede (Sub-)WF-Instanz eine logisch separate Änderungshistorie, unabhängig davon, auf welcher Hierarchiestufe sie im IAG angesiedelt ist. Die Änderungshistorie protokolliert alle für die betreffende (Sub-)WF-Instanz durchgeführten dynamischen Änderungen. Durch die Verwendung separater Änderungshistorien für (Sub-)WF-Instanzen erhöht sich der Aufwand für die Realisierung der in Kapitel 7 vorgestellten Operationen zur (dynamischen) Bildung und Auflösung von Sub-WF-Strukturen. Ähnlich wie die Ablaufhistorie, muß bei Anwendung der *nestBlock*-Operation die bisherige Änderungshistorie zerlegt werden (*history split*). Umgekehrt kommt es bei Durchführung der *unnestBlock*-Operation zur Zusammenführung von Änderungshistorien (*history merge*). Wir verzichten an dieser Stelle auf die Darstellung dieser Erweiterungen.

Im Zusammenhang mit der Unterstützung hierarchischer Workflows und dynamischer WF-Änderungen ergibt sich ein weiterer wichtiger Aspekt. Angenommen, innerhalb einer Schleife befindet sich eine komplexe Aktivität $X_{complex}$. Zur Laufzeit kann es dann bei wiederholter Ausführung der Schleife zur mehrfachen Instanziierung dieser komplexen Aktivität kommen. Für jede Sub-WF-Instanz können im Prinzip unterschiedliche dynamische Änderungen – temporäre und permanente – vorgenommen werden. Permanent bedeutet zunächst nur, daß die betreffende Änderung nach ihrer Einbringung für die Restdauer der Sub-WF-Instanz erhalten bleibt. In diesem Zusammenhang stellt

sich die Frage, wie mit permanenten Abänderungen einer Sub-WF-Instanz *subInst*, die in einer bestimmten Schleifeniteration des übergeordneten WF *parent(subInst)* ausgeführt wird, verfahren werden soll. Sollen diese Änderungen nur für die Sub-WF-Instanz der aktuellen Schleifeniteration gültig sein (pseudo-permanent) oder sollen sie auch für zukünftige Instanzen der komplexen Aktivität, d.h. ihrer Ausführung in Folgeiterationen, wirksam werden.

Beide Fälle sind für die Praxis relevant und werden von $\text{ADEPT}_{\text{flex}}$ unterstützt. Standardmäßig wird die 1. Variante zugrundegelegt: Eine komplexe Aktivität $X_{complex}$ wird bei der WF-Definition mit einer Sub-WF-Vorlage $\omega_{initial} \in$ *Templates* verknüpft. Diese Vorlage wird bei jeder Instanziierung von $X_{complex}$ verwendet, u.U. also mehrfach innerhalb derselben WF-Ausführung. Dabei wird jedes Mal eine neue Kopie des zugrundeliegenden Schemas angelegt, die dann instanzspezifisch abgeändert werden kann. Wird $X_{complex}$ im Verlauf der WF-Ausführung zum wiederholten Mal instanziiert, bleiben alle im Verlauf früherer Instanzen angewandten dynamischen Änderungen unberücksichtigt. Das bedeutet nicht notwendigerweise, daß für alle Instanzen von $X_{complex}$ die Ausgangs-WF-Graphen identisch sind. Werden vom WF-Modellierer Änderungen am Schema der Vorlage $\omega_{initial}$ vorgenommen, so können diese nach ihrer Freigabe bei Neuinstanziierungen von $X_{complex}$ berücksichtigt werden. Grundlage hierfür bildet das von ADEPT unterstützte späte Binden von Sub-Workflows.

Alternativ dazu können permanente Änderungen einer Instanz von $X_{complex}$ auch auf zukünftige Ausprägungen dieser komplexen Aktivität übertragen werden, vorausgesetzt, sie kommen im Kontext derselben übergeordneten WF-Instanz zur Ausführung. Zu diesem Zweck muß der betreffende Aktivitätenknoten mit dem Attribut `APPLY_CHANGES_UPWARD` gekennzeichnet werden, was sowohl statisch bei der WF-Modellierung als auch dynamisch während der Ausführung erfolgen kann. Bei der Beendigung einer Instanz *subInst* von $X_{complex}$ wird dann wie folgt verfahren: Wurden im Verlauf der Ausführung von *subInst* keine permanenten Änderungen durchgeführt, kann wie oben beschrieben vorgegangen werden. Andernfalls wird zuerst eine neue Vorlage $\omega_{derived}$ erzeugt, die als Schema den zuletzt gültigen Prozeßgraphen P_{perm} der Sub-WF-Instanz (mit allen auf ihr durchgeführten permanenten Änderungen) zugewiesen erhält. Anschließend wird vom WfMS auf der Ebene der übergeordneten WF-Instanz eine Änderung vorgemerkt, welche die bisherige Vorlage $\omega_{initial}$ von $X_{complex}$ durch die neue Vorlage $\omega_{derived}$ ersetzen soll. Diese WF-Änderung, die auf Grundlage der Basisoperation *changeActivityTemplate* verwirklicht werden kann, wird dann zum nächstmöglichen Zeitpunkt eingebracht, d.h. nach einem Rücksprung, der zur wiederholten Ausführung von $X_{complex}$ führt.

Der letztgenannte Ansatz ließe sich dahingehend erweitern, daß lediglich ausgewählte permanente Änderungen für zukünftigen Instanziierungen berücksichtigt werden. Allerdings würde sich dadurch der Aufwand für die Einbringung permanenter Änderungen erhöhen, da wir zwischen pseudo-permanenten, d.h. nur für eine konkrete Instanz gültigen Änderungen und echt permanenten Änderungen unterscheiden müßten. Zur Verwirklichung dieses Ansatzes müßte ähnlich wie im Fall der Einbringung temporärer und permanenter WF-Änderungen verfahren werden, indem unterschiedliche Prozeßgraphen für pseudo-permanente und permanente Änderungen verwaltet werden.

8.7 Zusammenfassung

In diesem Kapitel haben wir fortschrittliche Konzepte und Verfahren für die Einbringung und Rücknahme von WF-Instanzänderungen vorgestellt. Die Unterstützung von Änderungen unterschiedlicher Gültigkeitsdauer – temporär bzw. permanent – stellt keine ADEPT-spezifische Notwendigkeit dar, sondern ergibt sich aus den Erfordernissen realer WF-Anwendungen [RHD98]. Sie ist deshalb für die Realisierung flexibler WF-basierter Anwendungssysteme unverzichtbar.

Die diskutierten Lösungskonzepte bilden einen wichtigen Beitrag zur Verwirklichung fortschrittlicher, adaptiver WfMS. Die Konzepte können für unterschiedliche Einsatzzwecke verwendet und nach den Erfordernissen der jeweiligen Anwendung adaptiert werden. So kann es durchaus wünschenswert sein, eine temporäre Änderung bei Fehlerrücksprüngen zu erhalten und sie nur im Fall von Schleifenrücksprüngen zu entfernen. Die vorgestellten Verfahren sind vom Prinzip her an solche Erfordernisse anpaßbar. Darüber hinaus schaffen sie die logische Grundlage für die Unterstützung unterschiedlicher Änderungsarten durch dasselbe System. Änderungen einer WF-Vorlage können, bei Zugrundelegung des Konzepts permanenter Änderungen, ggf. auch noch auf bereits laufende WF-Instanzen propagiert werden, selbst wenn für diese instanzspezifische Ad-hoc-Änderungen vorgenommen worden sind.

Umgekehrt können ausnahmebedingte, einmalige Abweichungen vom geplanten Ablauf optimal durch das Konzept temporärer Änderungen unterstützt werden. Insgesamt erzielen wir auf Grundlage der vorgestellten Konzepte somit eine ganz neue Qualität im Hinblick auf die Realisierung flexibler Workflows.

Die vorgestellten Verfahren sind theoretisch fundiert und – was für die Erweiterbarkeit des Ansatzes sehr wichtig ist – unabhängig von den vom WfMS konkret angebotenen Änderungsoperationen. Bei der Entwicklung der Konzepte und Verfahren konnten wieder die positiven Eigenschaften des ADEPT-Basismodells genutzt werden. Zu nennen sind hier die explizite Modellierung und damit die explizite Repräsentation von Schleifen- und Fehlerrücksprüngen sowie der von ADEPT gewählte Markierungsansatz. Durch diese Möglichkeiten kann mit einfachen Mitteln festgestellt werden, ob es nach der Beendigung eines Prozeßschrittes zu einem Rücksprung im Ausführungsgraphen kommt oder ob eine Änderung im aktuellen WF-Status anwendbar ist. Erst dadurch können Konzepte wie die Einbringung und Rücknahme temporärer Änderungen sinnvoll und effektiv unterstützt werden. Insgesamt haben wir uns in diesem Abschnitt auf logische Aspekte und weniger auf Implementierungsfragestellungen (z.B. zur internen Repräsentation der WF-Instanz) konzentriert. Das vorgestellte Rücknahmeverfahren liefert Informationen darüber, welche Änderungen bei einem bestimmten Rücksprung entfernt, welche in modifizierter Form wieder eingebracht und welche unverändert beibehalten werden können. Seine Anwendung erfolgt im ADEPT-WfMS zunächst auf einer Kopie der WF-Instanz. Die dabei ermittelten Änderungen werden dann, sofern keine neuen Konflikte auftreten, atomar eingebracht. Dies erfolgt in einem kritischen Abschnitt, in dem keine anderen Änderungen zulässig sind.

Die vorgestellten Konzepte für dynamische WF-Änderungen verdeutlichen, daß zwischen den verschiedenen Funktionen eines adaptiven WfMS zahlreiche nicht-triviale Abhängigkeiten bestehen. Die Liste der beschriebenen Wechselwirkungen könnte bei Einbeziehung anderer WF-Aspekte wie Zeit oder Datenschutz [Weil97, Kubi98] noch ergänzt werden. Diese Abhängigkeiten müssen sorgfältig analysiert und verstanden werden, um zu praxistauglichen Gesamtlösungen zu gelangen. Ähnlich wie im Bereich der *Concurrency Control* in Datenbanksystemen [Dada96] kann ein solches System nicht dadurch realisiert werden, indem man einen „Balkon“ neben den anderen baut, um speziellen Erfordernissen Rechnung zu tragen. Statt dessen benötigen wir ein formales Grundgerüst mit klarer Semantik, das es uns erlaubt, Aussagen zur Korrektheit und Konsistenz von Workflows und von WF-Änderungen zu machen. $ADEPT_{flex}$ stellt ein solches Grundgerüst bereit.

9 Der ADEPT$_{workflow}$ -Prototyp

Die vorgestellten Konzepte zur Modellierung, Steuerung und dynamischen Änderung von Workflows wurden größtenteils im ADEPT$_{workflow}$-System implementiert. ADEPT$_{workflow}$ ist der Prototyp eines adaptiven WfMS, das auf dem ADEPT-Basismodell fußt und das seit 1997 in der Abteilung Datenbanken und Informationssysteme der Universität Ulm entwickelt wird. Das Ziel ist die Bereitstellung eines flexiblen Prozeßunterstützungssystems zur Realisierung zuverlässiger und robuster WF-Anwendungen. Zusätzlich zu den in dieser Arbeit vorgestellten Aspekten unterstützt ADEPT$_{workflow}$ eine Reihe weiterer fortschrittlicher Funktionen, etwa zur Behandlung temporaler Aspekte (z.B. Definition und Überwachung von Zeitabständen zwischen Aktivitäten) oder zur Abbildung organisatorischer Strukturen (z.B. Vertreterregelungen). Die Grundlagen dafür wurden in verschiedenen Diplomarbeiten geschaffen [Grim97, Hens97, Kirs96, Kony96, Kubi98, Weil97]. Interessant ist vor allem das Zusammenwirken dieser Erweiterungen mit den in dieser Arbeit behandelten Konzepten. Beispielsweise müssen bei dynamischen WF-Modifikationen ggf. auch Zeitdaten mit angepaßt werden.

ADEPT$_{workflow}$ zielt auf die flexible Unterstützung unternehmensweiter WF-Anwendungen. Um dies effizient bewerkstelligen zu können, besteht die Möglichkeit der verteilten Ausführung von WF-Instanzen. Dazu können ein WF-Schema zur Modellierungszeit partitioniert und einzelne WF-Instanzen zur Laufzeit verteilt von verschiedenen WF-Servern kontrolliert werden. Letzteres erfordert Konzepte zur Migration der Kontrolle von WF-Instanzen zwischen WF-Servern [BaDa98, BaDa99]. Die Herausforderung bei der Implementierung des ADEPT-Systems bestand in der integrierten Unterstützung dieser Anforderungen und in der Bereitstellung angemessener Werkzeuge und Schnittstellen für die Anwendungsentwicklung. Eine erste Version des Prototypen wurde auf der CeBit'98 und verschiedenen anderen Messen präsentiert, eine fortschrittlichere Version war auf der CeBit'2000, der EDBT'2000 [Hens00] und der BIS‘2000 [DRK00] zu sehen. In diesem Kapitel wird lediglich ein Grobüberblick zu ADEPT$_{workflow}$ und seinen Komponenten gegeben. Aus Platzgründen wird auf viele interessante Implementierungsaspekte (z.B. Synchronisation nebenläufiger Änderungen, Schnittstellen für die Anwendungsprogrammierung) nicht im Detail eingegangen.

9.1 Architektur und Implementierung von ADEPT$_{workflow}$

In diesem Abschnitt wird ein Überblick zur Architektur des ADEPT$_{workflow}$-Prototypen gegeben.

9.1.1 Überblick

ADEPT$_{workflow}$ basiert auf einer Client-Server-Architektur (vgl. Abbildung 9-1). Ein *WF-Server* verwaltet Organisations- und Prozeßmodelle, und übernimmt die Steuerung und Überwachung der darauf basierenden WF-Instanzen. Mit jedem WF-Server können mehrere *WF-Klienten* verbunden sein, wobei zwischen Modellierungs- und Laufzeit-Klienten unterschieden wird. Erstere dienen der Erstellung und Verwaltung von Organisations- und Prozeßmodellen, letztere unterstützen die Prozeßteilnehmer bei der Präsentation von Arbeitslisten, der Kontrolle und Überwachung von Prozeß- und Aktivitäteninstanzen sowie der Ad-hoc-Definition von WF-Instanz-Änderungen. Darüber hinaus benachrichtigen Laufzeit-Klienten die Benutzer über das Eintreten bestimmter Ereignisse, wie z.B. Zeitverletzungen. Für die Programmierung von Klientenprogrammen wird ein API angeboten, das gegenüber den von der WfMC (vgl. Kapitel 2) angebotenen Standard-Programmierschnittstellen [WFMC98a, WFMC98c] verschiedene Erweiterungen besitzt. Sie sind notwendig, um fortschrittliche Systemfunktionen für die Anwendungsprogrammierung zugänglich zu machen.

Die Kommunikation zwischen Klient und Server erfolgt nach dem Auftragsprinzip (in der Regel synchron). Um eine anstehende Aktivität zu starten oder eine dynamische Änderung durchzuführen, erteilt ein WF-Klient dem WF-Server einen entsprechenden Auftrag. Eingehende Klientenaufträge werden serverseitig zunächst in einer persistenten Warteschlange zwischengespeichert, bevor sie durch einen freien Aktivitätsträger (Prozeß, Thread) ausgeführt werden. Um bestimmte Aufträge vorrangig behandeln zu können, ist eine priorisierte Bearbeitung möglich. Im Anschluß an die Auftragsbearbeitung übermittelt der Server die Ergebnisse, die in einer persistenten Ausgangsschlange zwischengespeichert werden, zurück an den Klient. Zu erwähnen bleibt, daß der Systemkern eine Zeitsteuerungskomponente besitzt, die beim Eintreten bestimmter temporaler Ereignisse registrierte Aktionen (z.B.

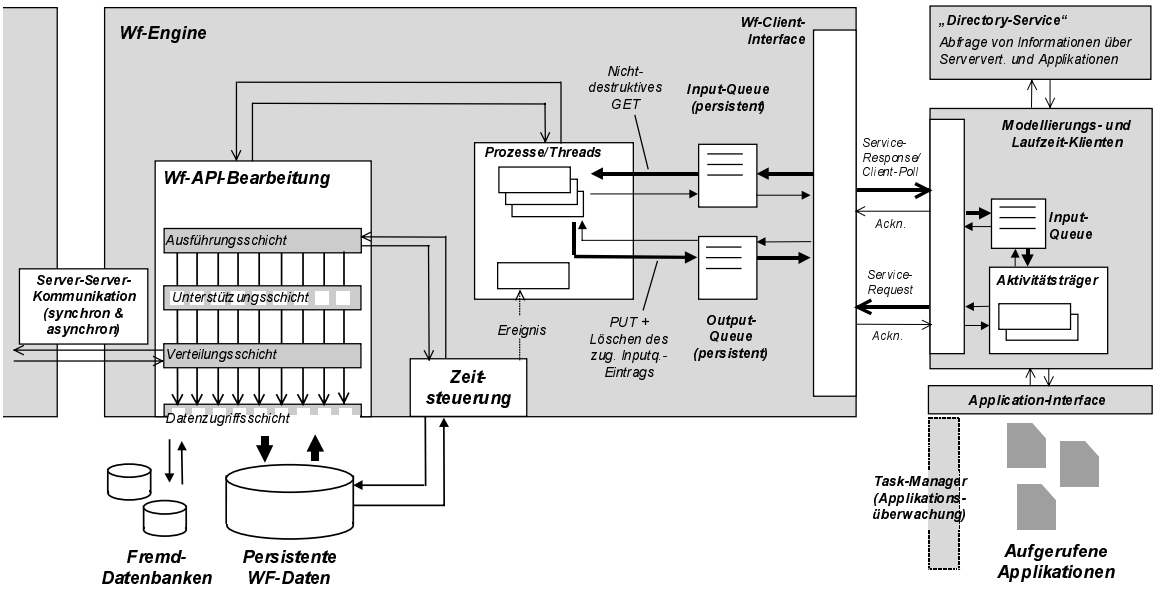


Abbildung 9-1: *Grobarchitektur des ADEPT$_{workflow}$-Prototypen*

Notifikation von Benutzern) auslösen kann. Dadurch können zum Beispiel Arbeitslisten periodisch aktualisiert und zeitliche Beschränkungen für einzelne Aktivitäten serverseitig kontrolliert werden.

9.1.2 Schichten des Systemkerns

Um eine hohe Anpaßbarkeit und Erweiterbarkeit des Systems zu erzielen, wurde der Systemkernel als Mehrschichtenarchitektur realisiert (vgl. Abbildung 9-2).

Die oberste Ebene bildet die *Ausführungsschicht*. Sie liest Klienten-API-Aufrufe (z.B. Starten einer Aktivität) aus der Eingangswarteschlange und koordiniert anschließend ihre Bearbeitung.

Dazu wird jeder Klienten-Aufruf in Funktionsaufrufe der untergeordneten *Unterstützungsschicht* umgesetzt. Diese Schicht stellt Dienste für die Verwaltung von Prozeßvorlagen, Prozeßinstanzen, Arbeitslisten, Organisationsmodellen und Aktivitätenprogrammen bereit. Jeder Dienst setzt sich dabei aus einer Menge von Funktionen zusammen. Die *Prozeßvorlagenverwaltung* etwa umfaßt Funktionen für das Laden, Speichern, Freigeben, Sperren und Versionieren von Prozeßvorlagen sowie für die Erzeugung neuer Prozeßinstanzen. Zur *Prozeßinstanzverwaltung* gehören Methoden für das Lesen, Ändern und Erzeugen von WF-Instanzdaten (z.B. Ausführungs- und Datenhistorien), für die WF-Steuerung (z.B. Berechnung von Folgemarkierungen bei Beendigung von Aktivitäten) und für die Durchführung von Laufzeitabweichungen (z.B. Einfügen / Löschen von Aktivitäten). Zur Bearbeitung von Klienten-API-Aufrufen ist eine enge Integration der Dienste der Unterstützungsschicht notwendig. Bei der Beendigung einer Aktivität müssen zum Beispiel Anpassungen des WF-Status vorgenommen, Bearbeiter für nachfolgende Arbeitsschritte bestimmt und Arbeitslisten aktualisiert werden. Auch dynamische WF-Änderungen erfordern die Ausführung mehrerer Dienste der Unterstützungsschicht, etwa zur Anpassung des WF-Graphen, des WF-Status und der Arbeitslisten.

Zur Ausführung einzelner Dienste der Unterstützungsschicht werden wiederum Operationen der untergeordneten *Datenzugriffsschicht* aufgerufen, die ihrerseits auf eine Menge von Datenbankoperationen abgebildet werden. Solche Basisoperationen bilden die Grundlage für die Erzeugung, das Lesen und die Modifikation von WF-Objekten (z.B. Aktivitätenknoten, Kontrollabhängigkeiten, Graphmarkierungen, Arbeitslisteneinträge). Zu erwähnen bleibt, dass die Bearbeitung eines einzelnen API-Aufrufs und alle damit verbundene Funktionsausführungen durch eine DB-Transaktion geschützt werden, so daß die Konsistenz der WF-Daten auch im Fehlerfall (z.B. nach Ausfall eines Servers) gewahrt bleibt. Aufgrund der persistenten Speicherung der Auftragsdaten kann eine Aufgabenbearbeitung nach Fehlschlägen wiederholt werden.

Wie eingangs erwähnt, können einzelne WF-Instanzen verteilt durch mehrere WF-Server ausgeführt werden. Das Ziel ist, das Gesamtkommunikationsaufkommen zu minimieren und einen hohen Durch-

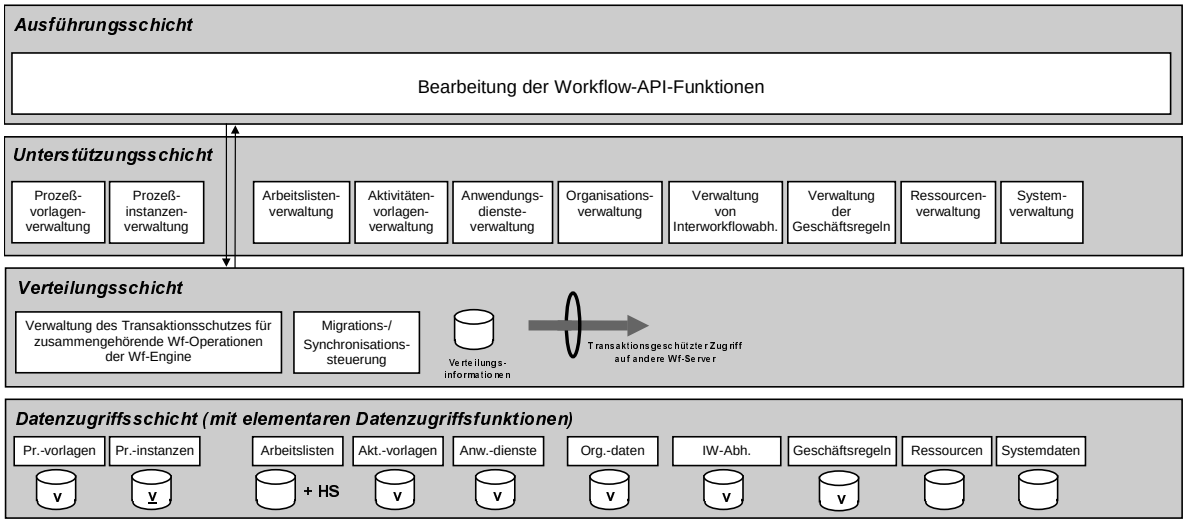


Abbildung 9-2: *Mehrschichtenarchitektur des ADEPT$_{workflow}$-Kerns (v steht für „verteilte Speicherung“)*

satz zu erzielen. Zu diesem Zweck können WF-Graphen zur Entwurfszeit, unter Ausnutzung wohldefinierter Kostenformeln und Optimierungsverfahren, in Teilgraphen bzw. Partitionen unterteilt werden. Jeder Partitition wird entweder statisch ein fester WF-Server zugeordnet, der zur Laufzeit die Kontrolle ihrer Aktivitäten übernimmt oder es erfolgt eine *variable Serverzuordnung* [BaDa00], bei der ein WF-Server erst zur Laufzeit bestimmt wird (z.B. in Abhängigkeit von der Serverwahl für Vorgängeraktivitäten). An der Ausführung einer WF-Instanz können dem entsprechend mehrere WF-Server (evtl. gleichzeitig) beteiligt sein. Beim Übergang zwischen zwei Partitionen wird deshalb eine *Migration* der Kontrolle notwendig, damit auf dem Server der Zielpartition mit der Ausführung der WF-Instanz fortgefahren werden kann. Dabei werden Instanzdaten (z.B. Ausschnitte der Ausführungs-, Datenelement- und Änderungshistorien) vom Quell- zum Zielserver transferiert. Zu erwähnen bleibt, daß ein Server nicht notwendigerweise den globalen Status der Instanz kennen muß, um die Aktivitäten einer Partition zu steuern. Das bedeutet, daß bei parallelen Ausführungszweigen keine strikte Synchronisation erforderlich wird. Die verteilte WF-Steuerung wird in ADEPT in der *Verteilungsschicht* realisiert. Dabei bleiben Verteilungsaspekte weitgehend vor Anwendungsprogrammierern und Anwendern verborgen. Weitere Details findet man in [BaDa97, BaDa98, BaDa99, BaDa00]. In [RBD99] werden Aspekte dynamischer Änderungen bei verteilter WF-Steuerung diskutiert.

9.1.3 Ausgewählte Implementierungsaspekte

In diesem Abschnitt werden ausgewählte Implementierungsaspekte, die für den Entwurf eines adaptiven WfMS essentiell sind, exemplarisch behandelt. Sie betreffen die interne Repräsentation von WF-Instanzen und deren Systemeinbettung. Aus Platzgründen wird auf die Darstellung anderer Implementierungsaspekte, wie Rollenauflösung, Datentransport und -bereitstellung, Protokollierung, Monitoring (Selbstüberwachung des Systems, Prozeßüberwachung), Caching, Robustheits- und Verfügbarkeitsaspekte sowie Performanz und Fehlerbehandlung, verzichtet.

9.1.3.1 Interne Repräsentation von WF-Instanzen

Die Art und Weise in der WF-Vorlagen und WF-Ausführungsgraphen systemintern repräsentiert werden, hat wesentlichen Einfluß auf die Unterstützung von Ad-hoc-Abweichungen. In gängigen WfMS üblich ist die Repräsentation der WF-Graphen als

- (interpretierte) Graphstruktur (Variante 1)
- kompilierte Prozeßpartikel (Variante 2)
- Regelmenge (Variante 3)
- (migrierendes) (Daten-)Objekt (Variante 4)

Bei der *Variante 1* wird die „externe“ Repräsentation des WF-Graphen in eine äquivalente interne Darstellung übertragen. Auf WF-Instanzebene müssen die Graphen dann noch um zusätzliche Informationen (z.B. Zustandsmarkierungen, Versionierungsinformationen zu Prozeßvariablen) „angereichert“ werden, um für die Ausführungskomponente des WfMS interpretierbar zu sein.

Bei der *Variante 2* wird – vereinfacht ausgedrückt – der WF-Graph in eine Menge miteinander kommunizierender endlicher Automaten übersetzt, die dann auf die betroffenen Rechnerknoten verteilt werden. Im Prinzip wird für jede WF-Aktivität ein solcher Automat erzeugt, ggf. aber können Schritte, die am selben Rechnerknoten auszuführen sind, durch denselben Automaten beschrieben werden. Die WF-Steuerung erfolgt bei diesem dezentralen Ansatz nachrichtenbasiert, wobei ein hoher Grad an Autonomie erzielt wird.

Bei der *Variante 3* werden die Kontrollflußbedingungen des WF-Graphen auf eine Menge von Regeln abgebildet. Als Implementierungsplattform dient dabei oftmals ein erweitertes (aktives) DBMS. Der Kontrollfluß wird durch Triggermechanismen und der Datenfluß auf der Grundlage von Datenbankobjekten gesteuert. Durch die Verwendung einer gemeinsamen DB für die Anwendungen und das WfMS ist bei diesem datenbankgestützten Konzept eine relativ einfache Synchronisation von WF-Daten und Anwendungsdaten möglich.

Bei der *4. Variante* schließlich werden WF-Instanzen als Objektinstanzen betrachtet, für die bestimmte Operationen bzw. Methoden angeboten werden. Je nach Implementierungsvariante „wandert“ ein solches WF-Objekt von Rechnerknoten zu Rechnerknoten oder es werden die Objektmethoden entfernt bearbeitet (z.B. mittels CORBA). – Variante 2 und Variante 3 wurden beim Entwurf von ADEPT$_{workflow}$ frühzeitig ausgeschlossen, da bei ihnen die Unterstützung von Ad-hoc-Abweichungen nicht oder nur mit erheblichem Zusatzaufwand realisierbar wäre. Abgesehen davon bestehen einige weitere Nachteile, etwa in Bezug auf die Herstellung des globalen Prozeßstatus (Variante 2) oder die Systemperformanz (Variante 3). Variante 4 wurde nicht gewählt, da hier die Unterstützung paralleler Ausführungszweige erhebliche Schwierigkeiten bereitet.

Für die korrekte Durchführung von Ad-hoc-Abweichungen ist eine interpretierende Ausführung des WF-Graphen am günstigsten. Um etwa feststellen zu können, welche Aktivitäteneingabedaten infolge einer dynamischen Änderung nicht mehr sicher versorgt sind, muß der gesamte WF-Graph vorliegen oder mit vertretbarem Aufwand rekonstruierbar sein. In ADEPT$_{workflow}$ wird deshalb die erste Variante – Repräsentation von WF-Ausführungsgraphen als interpretierbare Graphstruktur – verwendet. Durch den direkten Bezug zwischen externem und internem Ausführungsmodell gehen, im Gegensatz zu den anderen Varianten, keine semantisch wichtigen Informationen bei der Übersetzung „verloren“. Des weiteren können die Struktureigenschaften der WF-Graphen auch auf Ausführungsebene (z.B. zur Überprüfung der Korrektheit dynamischer Änderungen) genutzt werden. Viele der im ADEPT-Kernel implementierten Funktionen machen hiervon Gebrauch.

Um bestimmte Analysen (z.B. bzgl. Datenfluß) zu beschleunigen, werden zusätzliche Metadaten für die WF-Modelle verwaltet. Beispielsweise werden zu den Kontrollblöcken von WF-Graphen Informationen verwaltet, wie Start- und Endknoten oder logische Ein- und Ausgabeparameter (vgl. Kapitel 5).

In der derzeitigen Systemversion wird bei der Erzeugung einer neuen WF-Instanz eine Kopie der Ursprungsvorlage erzeugt, die dann „lokal“ abgeändert werden kann. Bei vielen Ansätzen referenzieren die Instanzen lediglich die globale Vorlage, wodurch Ad-hoc-Änderungen einzelner Instanzen nicht ohne weiteres möglich sind (vgl. Kapitel 2). Der Bezug zur globalen Vorlage bleibt in ADEPT allerdings erhalten, um bei späteren Systemerweiterungen auch die Evolution von WF-Schemata und die damit verbundene Anpassung laufender Workflows unterstützen zu können.

9.1.3.2 Systemeinbettung von WF-Instanzen

Für die Steuerung und Verwaltung von WF-Instanzen gibt es ebenfalls mehrere Alternativen: WF-Instanzen können als Betriebsystemprozeß, als Thread oder als (interpretierbare) Datenstruktur repräsentiert werden. Die erstgenannte Variante, bei der jede WF-Instanz durch einen eigenen Betriebssystemprozeß gesteuert bzw. verwaltet wird, ist praktisch kaum einsetzbar. Der Grund dafür ist, daß bei einer großen Zahl von WF-Instanzen viele aktive Betriebssystemprozesse resultieren, so daß sich ein hoher Verwaltungsaufwand einstellt. Wesentlich effizienter ist die Realisierung mittels Threads, da

hier die WF-Instanzen denselben Adreßraum verwenden. Das Problem der vielen aktiven WF-Instanzen (bzw. Threads) bleibt aber bestehen. ADEPT$_{workflow}$ wählt die dritte Variante, bei der WF-Instanzen (= WF-Graph + WF-Zustand + WF-Daten) intern durch interpretierbare Datenstrukturen repräsentiert werden, die zusätzlich persistent in der Datenbank des WfMS gespeichert werden. Das WfMS selbst verfügt über einen Pool von „Executor"-Threads. Immer wenn an einer WF-Instanz etwas zu tun ist, wird sie einem Executor-Thread zur Bearbeitung zugewiesen. Der Vorteil dieser (aufwendiger zu implementierenden) Variante ist, daß immer nur so viele Threads aktiv sind, wie auch tatsächlich benötigt werden. Durch diesen Ansatz werden potentiell sehr viele WF-Instanzen (performant) handhabbar, insbesondere wenn – wie im Fall von ADEPT$_{workflow}$ – zusätzlich das Caching von WF-Instanzen unterstützt wird.

9.1.3.3 Weitere Bemerkungen

Die Realisierung der verschiedenen Schichten des ADEPT$_{workflow}$-Kernels gestaltete sich wesentlich komplexer als es die Kurzbeschreibung aus Abschnitt 9.1.2 vermuten läßt. Allein die Implementierung der Funktionen zur Verwaltung von WF-Instanzen – sie sind in der Unterstützungsschicht angesiedelt – erforderte intensive Vorarbeiten. So wurden bereits zwei Jahre vor Beginn der Implementierungsarbeiten zu ADEPT$_{workflow}$ verschiedene Komponenten zur Steuerung und zur (persistenten) Verwaltung von WF-Instanzen prototypisch implementiert [Blas95]. Die dabei gewonnenen Erkenntnisse [Blas95, Blas96], etwa im Hinblick auf das partielle Zurücksetzen der WF-Ausführung oder das Datenkontext-Management (z.B. Bereitstellung und Übernahme von Aktivitätenparametern, kontextabhängiges Lesen der richtigen Werte von Datenelementen, Zurücksetzen von Datenelementen), konnten nutzbringend bei der Implementierung von Komponenten des ADEPT-WfMS verwendet werden.

Um Aktivitätenfehlschläge behandeln und eine evtl. Kompensation von Aktivitätenausführungen durchführen zu können, werden in ADEPT$_{workflow}$ für Datenelemente ggf. mehrere Versionen verwaltet. Das ADEPT-System erkennt, welche Version für einen bestimmten Lesezugriff herangezogen werden muß, welche Versionen für (evtl.) Kompensationen noch benötigt werden und wann Versionen nicht mehr gebraucht werden. Alle Kontrolldaten und alle WF-relevanten Daten werden persistent verwaltet, um im Fehlerfall die korrekte Wiederaufnahme der WF-Ausführung gewährleisten zu können. Zu erwähnen bleibt, daß im Mehrbenutzerbetrieb konkurrierende Zugriffe auf WF-Daten ebenfalls korrekt verarbeitet werden.

9.1.4 Realisierungsplattform

Die Implementierung des ADEPT-Prototypen erfolgte vollständig in Java. Für die Kommunikation zwischen Klienten und Server wird Java-RMI (*Remote Method Invocation*) eingesetzt. Als Datenbank dient Oracle V8. Die Anbindung von WF-Servern an die Datenbank basiert auf JDBC Typ 4, d.h. auf Seite des WF-Servers wird ein komplett in JAVA implementierter Datenbanktreiber verwendet, der dem *JDBC-Standard* genügt. Derzeit (Stand: Mai 2000) umfaßt der Quellcode ca. 330 Klassenimplementierungen und ca. 120.000 Zeilen Programmcode. Das DB-Schema besitzt mehr als 50 Tabellen.

9.2 Durchführung einer Ad-hoc-Abweichung in ADEPT$_{workflow}$

Zur korrekten Durchführung einer Laufzeitabweichung muß ein Anwender ggf. mehrere Interaktionen mit dem System durchführen. Soll zum Beispiel eine Aktivität gelöscht werden, muß der Initiator dieser Änderung begleitende Anpassungen vornehmen, wenn dadurch die Parameterversorgung nachfolgender Schritte nicht mehr gewährleistet werden kann (siehe Kapitel 7). Wichtig ist, daß alle Modifikationen innerhalb derselben Änderungstransaktion auf die Instanz angewendet werden. Um dabei lange Blockierungen zu vermeiden, wird in ADEPT ein optimistisches Verfahren zur Synchronisation nebenläufiger Änderungstransaktionen verwendet. Hierbei wird eine Änderung in die Phasen Definition, Analyse und Durchführung unterteilt, die wir kurz skizzieren wollen (für Details siehe [Weil97]).

9.2.1 Definitionsphase

In der *Definitionsphase* legt der Anwender, unterstützt durch ein WF-Klientenprogramm, eine Ad-hoc-Abweichung für die WF-Instanz fest. Dazu wählt er eine Änderungsoperation (z.B. Einfügen ei-

ner Aktivität) aus und erfragt dann vom System relevante Informationen zum aktuellen Prozeßkontext (siehe unten). Diese Daten benötigt er, um Details der Änderung (z.B. Position des neu einzufügenden Schrittes) exakt festlegen zu können. Abhängig von der Art der durchzuführenden Änderungsoperation und den bereits festgelegten Informationen wird der Prozeßkontext sinnvoll eingeschränkt. Zur Änderung von Aktivitätenattributen genügt es zum Beispiel, Informationen zu den jeweiligen Knoten bereitzustellen (z.B. aktueller Knotenstatus und aktuelle Attributfestlegungen). Für das Einfügen neuer Schritte werden dagegen weitergehende Kontextinformationen erforderlich, etwa zum aktuellen Status von Aktivitäten, zu ihren Knoten und deren Anordnungsbeziehungen sowie zu den verfügbaren Aktivitätenvorlagen, die zur Ad-hoc-Definition neuer Schritte verwendet werden dürfen. Mit Hilfe dieser Informationen und seiner ungefähren Kenntnis des Gesamtprozesses kann der Benutzer nun festlegen, unter welchen Bedingungen der neu eingefügte Schritt gestartet und bis wann er beendet werden soll.

Soll etwa für die WF-Instanz eine neue Aktivität X zwischen zwei Knotenmengen M_{before} und M_{after} eingefügt werden (vgl. Kapitel 7), wählt der Anwender zunächst die entsprechende Änderungsoperation aus. Der Klient erfragt dann vom WF-Server relevante Informationen zum Prozeßkontext, die er dazu verwendet, den Benutzer bei der Durchführung der Änderung zu assistieren. In einem ersten Schritt bietet er dem Benutzer eine Liste von Aktivitäten an, nach denen er die neue Aktivität X einfügen kann. Der Benutzer wählt hiervon eine Teilmenge M_{before} (mindestens aber einen Knoten) aus. Die selektierte Knotenmenge wird ggf. noch transitiv reduziert, bevor der WF-Klient vom Benutzer die Knotenmenge M_{after} erfragt, vor die X eingefügt werden soll. Dabei nutzt er die bereits vorhandene Information zum Prozeß- und Änderungskontext aus, indem er dem Anwender nur noch diejenigen Knoten zur Auswahl anbietet, die Nachfolger mindestens eines Knotens aus der Menge M_{before} sind und die noch nicht gestartet wurden. Die formalen Vorbedingungen der Änderungsoperation werden hier also gezielt genutzt, um den Benutzer bei der Durchführung der Modifikation zu unterstützen. Dabei bleiben Realisierungsdetails, wie die durchzuführenden Graphtransformationen und Markierungsanpassungen, weitgehend vor dem Anwender verborgen.

Der bereitgestellte WF-Demo-Klient (vgl. Abschnitt 9.3.3) unterstützt den Benutzer auch bei der Anknüpfung der Parameter des neu eingefügten Schrittes an den WF-Datenkontext. Für die betreffende Aktivität werden dem Anwender, abhängig von der gewünschten Einfügeposition, alle Datenelemente angezeigt, die bei der Aktivierung des neuen Knotens versorgt wären, d.h. auf die die Aktivität lesend zugreifen könnte. Der Anwender kann die Eingabeparameter des neuen Schrittes entweder an diese Datenelemente anknüpfen, vorausgesetzt sie sind typverträglich, oder alternativ dazu interaktive Versorgungsschritte „vorschalten“, wenn die betreffenden Parameter nachforderbar sind. Letzteres führt automatisch zur Erzeugung neuer Datenelemente sowie zum Aufruf automatisch generierter Eingabemasken beim Start des eingefügten Schrittes. Auch die Ausgabeparameter können auf existierende oder aber neu einzufügende Datenelemente abgebildet werden. Auch im Zusammenhang mit dem Löschen von Aktivitäten bietet der WF-Demo-Klient eine angemessene Unterstützung an. Einem Anwender, der eine Aktivität löschen oder überspringen will, werden alle Folgeschritte angezeigt, deren Eingabeparameter dadurch nicht mehr versorgt werden können. Ferner werden ihm verschiedene Möglichkeiten angeboten, um solche Probleme angemessen zu behandeln (z.B. durch automatische Generierung von Datennachforderungsdiensten oder Einfügen spezieller Versorgungsschritte).

9.2.2 Analysephase

Nachdem die gewünschte Änderung vom Anwender definiert worden ist, wird sie an den WF-Server übermittelt und dort einer *Analyse* unterzogen. Zuerst überprüft der Server, ob der Benutzer zur Durchführung der Änderung berechtigt ist (für Details hierzu siehe [Kubi98]). Ist dies nicht der Fall, wird die Änderungstransaktion abgebrochen und an den WF-Klienten eine entsprechende Mitteilung geschickt. Andernfalls folgt die eigentliche Analysephase, in der serverseitig nochmals geprüft wird, ob die Korrektheit und Konsistenz der WF-Instanz durch die Anwendung der Änderung gewahrt bleibt. Der Server lädt dazu alle benötigten WF-Instanzdaten in den Hauptspeicher. Um etwa die korrekte Anwendbarkeit der im vorangegangenen Abschnitt diskutierten Einfügeoperation zu prüfen, müssen Informationen zum Status der Knoten aus der Menge M_{after} – der Schritt darf nicht vor bereits laufende oder beendete Knoten eingefügt werden – sowie zu den Anordnungsbeziehungen der Aktivitäten aus den beiden Mengen M_{after} und M_{before} vorliegen. Darüber hinaus muß geprüft werden, ob die vom Benutzer definierte Anknüpfung der Aktivitätenparameter an den WF-Datenkontext korrekt sind.

Je nach Benutzerfestlegung können dazu mehr oder weniger aufwendige Überprüfungen anfallen und unterschiedliche Informationen zum aktuellen Datenflußschema der Instanz verwendet werden.

Nach der Durchführung der Analysen wird das Ergebnis zurück an den WF-Klienten gemeldet. Wurden im WF-Server bei der Überprüfung der Änderung Probleme festgestellt, wird eine Beschreibung dieser Fehler (z.B. Liste unversorgter Aufrufparameter), zusammen mit Vorschlägen zu ihrer Behandlung (z.B. Hinzufügen von Datennachversorgungsdiensten), an den WF-Klienten übermittelt. Diese Informationen werden dem Benutzer gegenüber angemessen präsentiert, so daß er ggf. geeignete Korrekturen bzw. Anpassungen vornehmen kann. Die Definitions- und Analysephase können sich dem entsprechend wiederholen, bis entweder eine korrekte Änderungsdefinition resultiert oder der Benutzer die Änderungstransaktion abbricht. Werden keine Fehler mehr festgestellt und stimmt der Anwender der Änderung zu, wird sie in der Durchführungsphase in die Instanz eingebracht.

9.2.3 Durchführungsphase

Um lange Sperren auf der WF-Instanz und damit verbundene Blockierungen zu vermeiden, werden die in der Definitionsphase inkrementell festgelegten Änderungen (inkl. der erforderlichen Überprüfungen) lediglich auf einer Kopie des Instanzgraphen ausgeführt. Erst im Anschluß daran erfolgt die eigentliche *Durchführung* der Änderungstransaktion im WF-Server, in der die festgelegten Operationen tatsächlich auf die WF-Instanz angewendet werden. Dies erfolgt in einem kritischen Abschnitt, d.h. während der Änderungseinbringung dürfen sich keine weiteren Änderungstransaktionen zu der betreffenden WF-Instanz in ihrer Durchführungsphase befinden bzw. diese Phase betreten. Darüber hinaus werden für Aktivitätenknoten, die in dem von der Änderung betroffenen Bereich des Ausführungsgraphen liegen, Statusänderungen nur noch eingeschränkt zugelassen (im wesentlichen Wechsel vom Zustand `RUNNING` in den Zustand `COMPLETED`, ohne sofortige Neubewertung des Graphen).

Zu Beginn der Durchführungsphase muß nochmals validiert werden, ob die Korrektheit und Konsistenz der Instanz gewahrt bleiben. Der Grund dafür ist, daß zwischen der erfolgreichen Überprüfung der Änderung in der Analysephase und dem Eintritt in die Durchführungsphase evtl. konkurrierende Änderungstransaktionen (inkl. Statusänderungen) durchgeführt wurden. In den meisten Fällen kann dies mit einfachen Mittel (z.B. durch Anwendung einfacher Zeitstempelverfahren) ausgeschlossen werden. Wurde zwischenzeitlich keine konkurrierende Änderung durchgeführt, kann die gewünschte Modifikation sofort „eingespielt“ werden. Dazu werden die in der Definitionsphase für die einzelnen Änderungsoperationen abgeleiteten Graphänderungsprimitve (vgl. Kapitel 7) auf die WF-Instanz angewendet. Es können sich sowohl Anpassungen der Kontroll- und Datenflußstruktur der WF-Instanz als auch Änderungen ihres Status ergeben. Für den letztgenannten Fall müssen auch Arbeitslisten angepaßt werden. Betroffene WF-Klienten werden über die stattgefundene Änderung und die notwendige Aktualisierung der von ihnen verwalteten Arbeitslisten benachrichtigt.

Wurden dagegen vor dem Beginn der Durchführungsphase eine konkurrierende Änderungstransaktionen auf die WF-Instanz angewendet (inkl. Statusänderungen), müssen zusätzliche Tests durchgeführt werden, um potentielle Konflikte (z.B. bzgl. Modifikationen der Kontroll- und Datenflußstruktur sowie des WF-Status) zwischen den Änderungstranskationen ausschließen zu können. In den meisten Fällen genügen hierfür einfache Konflikttests, bei denen die Mengen der von zwei Änderungstransaktionen manipulierten Knoten, Attribute oder Datenelemente miteinander verglichen werden (für Details siehe [Weil97]). Werden keine Konflikte festgestellt, wird wie oben beschrieben verfahren. Andernfalls wird versucht die Änderungstransaktion durch einen modifizierten (neu zu berechnenden) Satz an Basisänderungsprimitven in die Instanz einzubringen. Nur wenn dies nicht gelingt, wird die Änderungstransaktion abgebrochen und eine entsprechende Mitteilung an den Initiator der Änderung verschickt.

Alle für eine WF-Instanz durchgeführten Änderungen werden in ihrer *Änderungshistorie* persistent gespeichert. Die Änderungshistorie bildet die Grundlage für die effiziente Durchführung von Konflikttests, für die Rücknahme temporärer Änderungen bei Schleifen- oder Fehlerrücksprüngen und für die vollständige Dokumentation aller durchgeführten Abweichungen. Darüber hinaus wird in Fehlerfällen ein (partielles) Zurücksetzen der WF-Instanz (inkl. der zuvor durchgeführten Strukturänderungen) möglich.

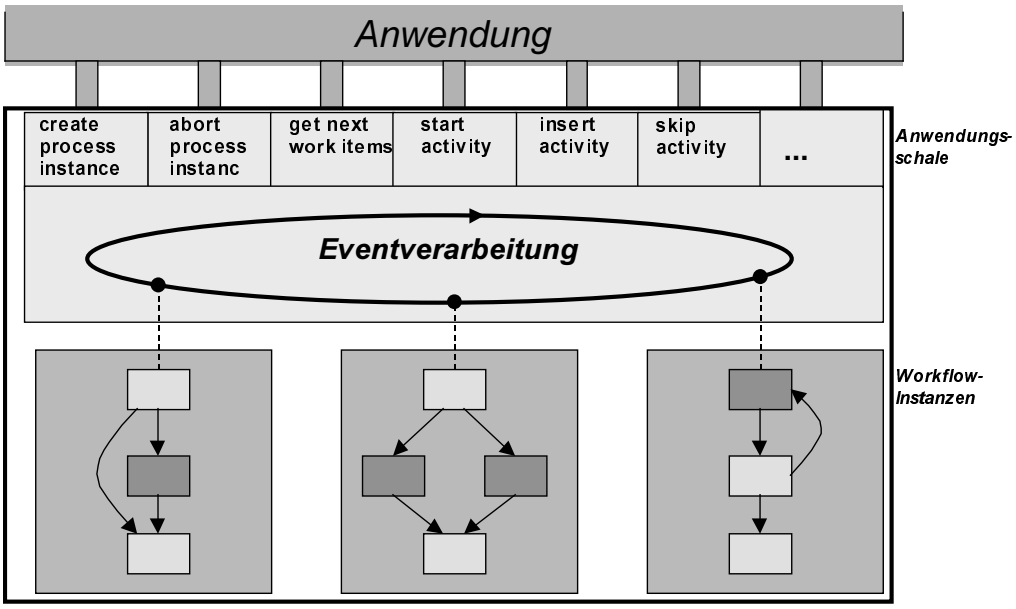


Abbildung 9-3: *Anwendungsprogrammierung in ADEPT$_{workflow}$ (siehe [Dada95])*

9.3 Anwendungsentwicklung in ADEPT$_{workflow}$

9.3.1 Grundlegende Vorgehensweise zur Entwicklung von WF-Anwendungen

Vorgangsorientierte Anwendungen werden in ADEPT$_{workflow}$ im wesentlichen durch die graphische Beschreibung ihrer Ablaufstruktur, d.h. durch die Definition einer geeigneten Prozeßvorlage, und durch das „Einstecken“ vorgefertigter Programmbausteine in diese Vorlage entwickelt. Ersteres wird durch die Bereitstellung eines graphischen, syntaxgesteuerten WF-Editors (siehe Abschnitt 9.3.2) unterstützt. Letzteres erfolgt indirekt durch die Verknüpfung wiederverwendbarer Aktivitätenvorlagen mit Anwendungsprogrammen und durch die Zuordnung dieser Vorlagen zu Knoten des WF-Graphen. Sind für alle Knoten geeignete Vorlagen eingesetzt und alle dadurch festgelegten Aktivitätenparameter geeignet miteinander verknüpft, ist die WF-Implementierung – von außen gesehen – eine ablauffähige Anwendung. Ihre Startparameter können beim Aufruf (d.h. bei der Erzeugung einer WF-Instanz) mittels einer Standardmethode versorgt und ihre Ergebnisse über eine Standardmethode wieder an die Aufrufumgebung zurückgeliefert werden.

Um WF-Instanzen in einheitlicher Form ansprechen und in ein darüber liegendes Anwendungsprogramm, das dann in einem gewissen Sinne nur eine Anwendungsschale (*Application shell*) darstellt [Dada95], einbinden zu können, werden alle WF-Instanzen und ihre Arbeitsschritte – unabhängig von ihrem Typ bzw. ihrer Vorlage – mit denselben generischen Methoden angesprochen (siehe Abbildung 9-3). Sie dienen der Erzeugung von WF-Instanzen, der Abfrage ihres Status, dem Erfragen der für einen Benutzer wählbaren Schritte, dem Anstoßen oder Abbrechen aktivierter bzw. laufender Schritte, der dynamischen Festlegung von WF-Attributen (z.B. Ausführungszeiten) sowie der Ad-hoc-Definition von Laufzeitabweichungen. Die Anwendungsschale übernimmt die Verwaltung der vom angemeldeten Benutzer zu bearbeitenden Arbeitsschritte und sie reagiert auf auftretende Systemereignisse (z.B. „Zeitüberschreitung“, „dynamische WF-Änderung“).

Bereits in einer frühen Phase des ADEPT-Projektes hat sich gezeigt, daß für die Kontrolle und Änderung von WF-Instanzen eine semantisch „hohe“ Schnittstelle angeboten werden muß [Dada95]. Nur dadurch lassen sich angepaßte Bedieneroberflächen, wie sie in zahlreichen Einsatzgebieten erforderlich sind, realisieren. Aus diesem Grund bietet ADEPT ein umfassendes API (= Application Programming Interface) an, mit dem Prozeß- und Aktivitäteninstanzen kontrolliert, überwacht und bei Bedarf dynamisch abgeändert werden können. Die Funktionalität dieses API geht über die Grundfunktionalität der Klienten-Programmierschnittstellen heutiger WfMS hinaus. Insbesondere werden Me-

thoden zur Definition und Durchführung von Ad-hoc-Abweichungen (z.B. Einfügen / Löschen von Aktivitäten) angeboten. Mittels der Funktion *dynamicInsert* etwa kann eine neue Aktivität in einen WF-Instanzgraphen eingefügt werden. Die Aufrufparameter dieser Methode legen für den neu einzufügenden Schritt fest, welche Aktivitätenvorlage verwendet, an welcher Position im Graphen er eingefügt und wie seine Aufrufparameter versorgt werden sollen (z.B. durch automatisch generierte Eingabeformulare). In diesem Zusammenhang werden weitere Methoden angeboten, mit denen ein WF-Klient relevante Informationen zum Kontext einer Änderung (z.B. Nachfolger einer Aktivität) und zu ihrer Durchführung (z.B. verfügbare Aktivitätenvorlagen) erfragen kann. Ein WF-Demo-Klient, der unter Verwendung des ADEPT-API realisiert wurde, wird in Abschnitt 9.3.3 vorgestellt.

Auch ein WF-Server kann Nachrichten bzw. Aufträge an registrierte Klienten übermitteln (sog. „Push-Pull-Modell“). Dadurch können WF-Klienten periodisch oder beim Eintreten bestimmter Ereignisse über Aktualisierungen ihrer Arbeitslisten informiert werden. Für adaptive WfMS kann eine solche aktive Benachrichtigung von Klienten sinnvoll sein, etwa wenn zur Durchführung einer dynamischen Änderung die Zustimmung von Benutzern eingeholt werden muß oder wenn sie über eine stattgefundene Änderung und ihre Auswirkungen (z.B. Arbeitslistenanpassungen) umgehend informiert werden sollen. Um auf solche Servermitteilungen bzw. -ereignisse angemessen reagieren zu können, mußte das Klienten-API erweitert werden.

Wie wir Kapitel 10 noch sehen werden, bietet die Mehrzahl der in der Literatur diskutierten Ansätze für adaptive WfMS keine solchen Programmierschnittstellen an. Statt dessen müssen dynamische Änderungen, sofern sie überhaupt unterstützt werden, vom Benutzer mit Hilfe eines (graphischen) Editors festgelegt werden. Währenddessen sind meist weder das Fortschreiten der WF-Instanz noch irgendwelche anderen Änderungen möglich. Die Mehrzahl dieser Ansätze bietet dem Benutzer auch keine semantisch höherwertigen Editier-Operationen und keine ausreichende Unterstützung bei der Durchführung von Konsistenzprüfungen an.

9.3.2 Klienten zur Erstellung von Workflow- und Organisationsmodellen

Die Modellierungsumgebung umfaßt (Buildtime-) Klienten für die Spezifikation von Prozeß- und Organisationsmodellen. Die grundlegende Philosophie dieser, auf der Basis von Java implementierten Werkzeuge ist es, potentielle Fehler oder Inkonsistenzen in den Modellen bereits bei ihrer Erstellung bzw. Änderung auszuschließen. Dies soll einerseits durch die Bereitstellung syntaxgesteuerter Editoren, andererseits durch angemessene Korrektheits- und Konsistenprüfungen gewährleistet werden.

Für die WF-Modellierung stellt ADEPT$_{workflow}$ den grafischen, syntaxgesteuerten Editor *ADEPT$_{editor}$* zur Verfügung [Mich99]. Dieses Werkzeug basiert auf den in dieser Arbeit vorgestellten Konzepten und Verfahren. ADEPT$_{editor}$ gestattet es nicht nur, neue WF-Vorlagen schnell und fehlerfrei zu erstellen, sondern ermöglicht auch die rasche und korrekte Anpassung existierender Vorlagen an Änderungen in den Geschäftsprozessen. Für den Modellierer bietet ADEPT$_{editor}$ komfortable Editier- und Änderungsoperationen an, die auf die ADEPT$_{flex}$-Operationen zurückgeführt werden. Um den Entwerfer auch bei der Durchführung komplexer Vorlagenänderungen angemessen unterstützen zu können, wurden höherwertige Operationen (z.B. Schleife um existierenden Kontrollblock einfügen) implementiert.

Die Blockstrukturierung der WF-Graphen und die Mehrzahl der in Kapitel 3 geforderten Struktureigenschaften (z.B. Zyklenfreiheit in Verbindung mit Sync-Kanten, Regeln zur Verwendung von Fehlerrücksprungkanten) werden bereits während der Erstellung der Modelle, durch die Vorgabe geeigneter Regeln für die Anwendung der Editoroperationen, sichergestellt. Darüber hinaus können verschiedene Analysen durchgeführt werden, etwa zur Prüfung der Korrektheit von Datenflüssen. Datenflüsse werden durch die Verknüpfung der Ein- und Ausgabeparameter von Aktivitäten mit globalen Datenelementen modelliert. Dabei kommen die in Kapitel 5 vorgestellten Analyseverfahren zur Anwendung. Schließlich bestehen für die Beschreibung von Aktivitäten dieselben Freiheitsgrade, wie sie eingangs von Kapitel 3 diskutiert worden sind (z.B. Differenzierung zwischen zwischen nachforderbaren und nicht nachforderbaren Parametern). Mit Hilfe von ADEPT$_{editor}$ können somit die für prozeßorientierte Anwendungen typischen Probleme (Verklemmungen, unerwünschte Zyklen, nicht aktivierbare Ablaufzweige, Aufruf von Aktivitätenprogrammen mit fehlenden Eingabedaten, usw.) bereits frühzeitig, beim Entwurf des Anwendungssystems, ausgeschlossen werden.

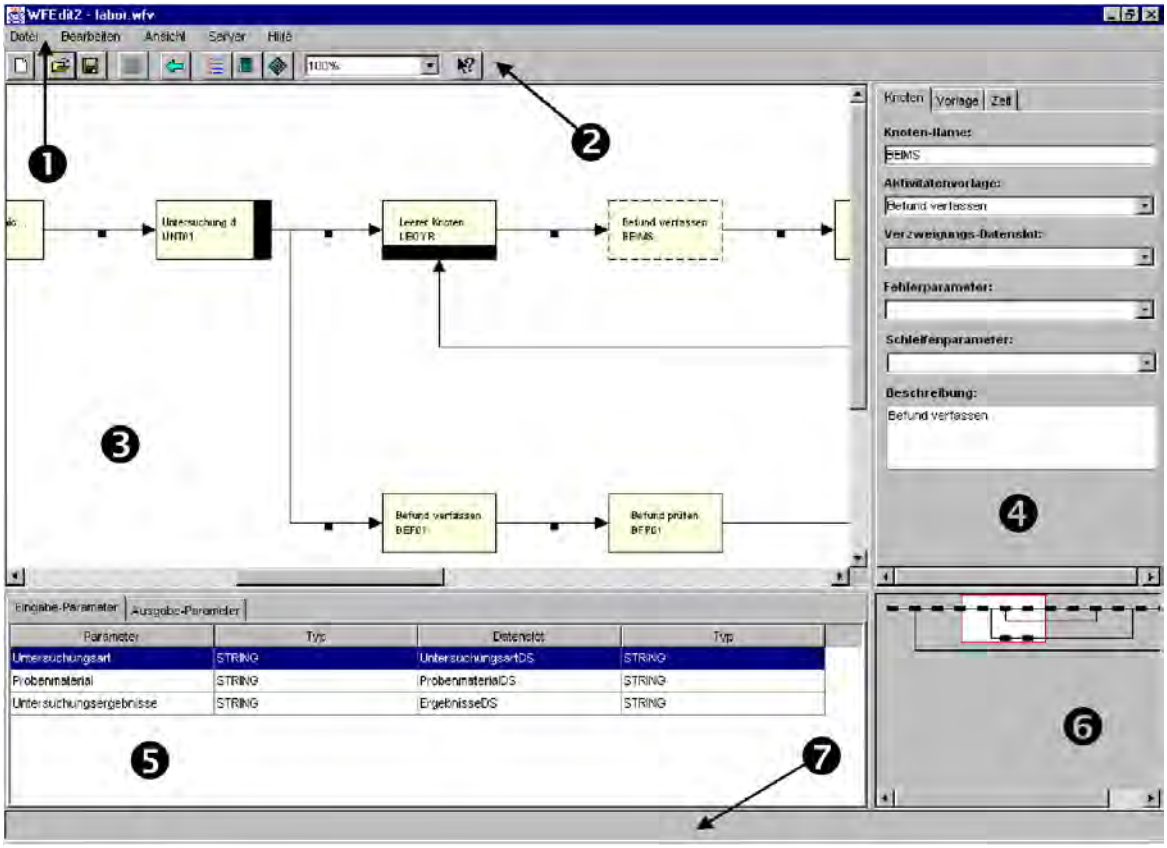


Abbildung 9-4: *Hauptfenster des WF-Editors WFEdit2 (vgl. [Mich99])*

Die Bedieneroberfläche von ADEPT$_{editor}$, die unter ergonomischen Gesichtspunkten entworfen wurde [Mich99], zeigt Abbildung 9-4:

Der Zugriff auf die Editorfunktionen ist über die Hauptmenüleiste (❶) möglich, die wichtigsten Befehle werden zusätzlich über eine Symbolleiste (❷) angeboten. Der WF-Graph der aktuell bearbeiteten Vorlage wird in einem Subfenster (❸) dargestellt, Detailinformationen zu seinen Knoten und Kanten (z.B. Bearbeiterzuordnungen, Zeitattribute) können in den rechts davon angezeigten Registern festgelegt werden (❹). WF-Graphen können mit unterschiedlichen Zoom-Faktoren dargestellt werden. Darüber hinaus wird eine Sichtenbildung durch das Ein- und Ausblenden von Kanten (eines bestimmten Typs) unterstützt. In der Placemarkerbox (❻) wird der gesamte WF-Graph in einer Übersicht angezeigt, wobei jeweils der aktuell dargestellte Bereich hervorgehoben ist. Die Placemarkerbox kann auch zur Navigation verwendet werden. Die Definition des Datenflusses erfolgt, im Gegensatz zu der in dieser Arbeit gewählten Darstellungsform, tabellarisch in dem links unten abgebildeten Fenster (❺). Es zeigt für einen selektierten Knoten seine Ein- und Ausgabeparamter an und ermöglicht ihre Verknüpfung mit globalen Datenelementen. Dabei können Detailinformationen (z.B. alle Schreiber und Leser eines bestimmten Datenelements oder alle für einen gegebenen Aktivitätenknoten „verfügbaren" Datenelemente) explizit angezeigt werden. Die wichtigsten Programmmeldungen werden in einer Statusleiste (❼) angezeigt.

Für ein erstelltes Modell können zu jedem Zeitpunkt die genannten Korrektheits- und Konsistenzüberprüfungen durchgeführt werden. Ein Beispiel für das Prüfprotokoll eines solchen Tests zeigt Abbildung 9-5.

Für die interne Repräsentation von WF-Graphen und für ihre visuelle Darstellung wurden fortschrittliche Konzepte und Algorithmen implementiert. Die Graphrepräsentation basiert auf erweiterten Adjazenzlisten [CLR89], wobei Knoten und Kanten (mit ihren Attributen) in separaten Hashtabellen verwaltet werden. Dies gestattet einen schnellen Zugriff auf Graphelemente und ermöglicht die einfache Bestimmung der von einem gegebenen Knoten ausgehenden Kanten. Für die Graphvisualisierung wurden gängige Algorithmen [Paul93, VMJ97] angepaßt, um Darstellungen mit möglichst wenigen

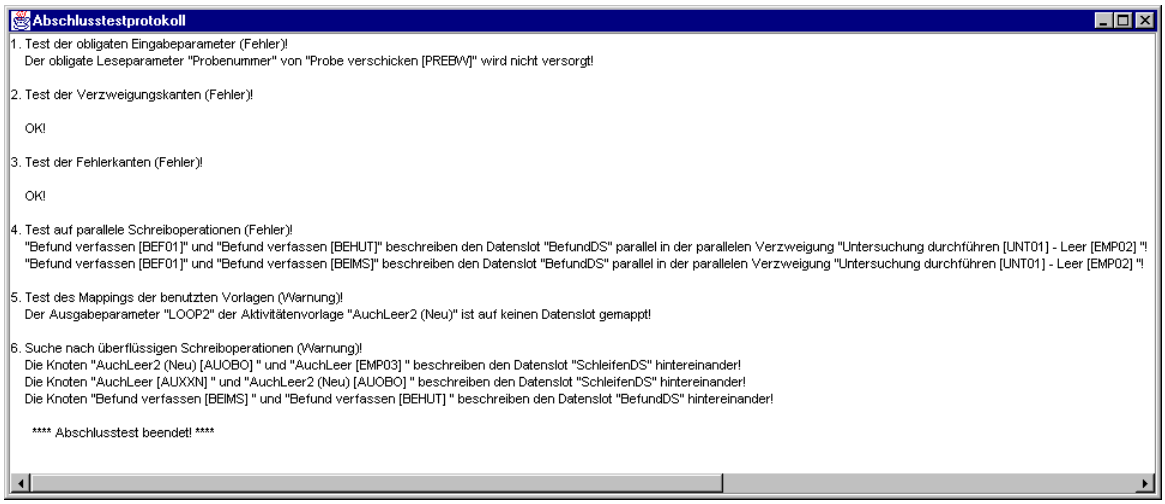


Abbildung 9-5: *Ergebnis eines Abschlußtests*

Kantenkreuzungen zu erhalten. Hilfreich ist in diesem Zusammenhang die Unterscheidbarkeit zwischen verschiedenen Kantentypen gewesen. Beispielsweise werden Kreuzungen zwischen normalen Kontrollkanten und Schleifenkanten vollständig verhindert. Um Kreuzungen zwischen Kontroll- und Datenflußkanten zu vermeiden, wird für die Festlegung von Datenflüssen eine tabellarische Darstellungsform gewählt. Weitere Details werden in [Mich99] beschrieben.

Die mit ADEPT$_{editor}$ erstellten Prozeß- und Aktivitätenvorlagen können in einem Repository, das auf einem RDBMS basiert, abgelegt werden. Ihre Freigabe ist erst möglich, nachdem alle erforderlichen Korrektheitsüberprüfungen erfolgreich verlaufen sind.

Zu erwähnen bleibt, daß WF-Modelle nicht nur mit Hilfe des graphischen Editors ADEPT$_{editor}$, sondern auch unter Verwendung einer skriptbasierten Beschreibungssprache (ADL = ADEPT Definition Language; siehe [Hens97]) spezifiziert werden können. Eine ADL-Beschreibung kann problemlos in den graphischen Editor geladen werden. Umgekehrt kann eine mit ADEPT$_{editor}$ erstellte Vorlage – alternativ oder ergänzend zu ihrer Speicherung im Repository – auch als ADL-Skript in einer Datei abgelegt werden. Dadurch können Vorlagen, unterstützt durch einen einfachen Check-Out /Check-In-Mechanismus, mit ADEPT$_{editor}$ auch dann bearbeitet werden, wenn keine Verbindung zum WF-Server besteht (z.B. bei mobilen Klienten). Darüber hinaus können andere Werkzeuge (z.B. Geschäftsprozeßmodellierungswerkzeuge) ADL-Beschreibungen für den Export/Import von ADEPT-Modellen verwenden.

Um interaktive Arbeitsschritte zur Laufzeit den richtigen Arbeitslisten zuordnen zu können, muß bei der Definition der Aktivität (bzw. ihrer Vorlage) eine geeignete *Bearbeiterformel* angegeben werden, die die Menge ihrer potentiellen Bearbeiter festlegt. Bei der Definition solcher Formeln kann der WF-Modellierer Bezug auf die Entitäten eines Organisationsmodells nehmen (z.B. Mitarbeiter, Stellen, Rollen, Fähigkeiten oder Organisationseinheiten), wobei auch komplexe Bearbeiterformeln über konjunktive oder disjunktive Verknüpfungen festlegbar sind (für weitere Details siehe [Kubi98]). Für die Definition von Organisationsmodellen (inkl. Vertreterregelungen) wird das Werkzeug ADEPT$_{orgmanager}$ angeboten (siehe Abbildung 9-6), das im Rahmen eines studentischen Praktikums entwickelt wurde und das ebenfalls wieder verschiedene Konsistenzprüfungen unterstützt.

9.3.3 Klienten zur Kontrolle, Überwachung und Ad-hoc-Änderung von Workflows

Um die Möglichkeiten des ADEPT$_{workflow}$-Systems demonstrieren zu können, wurde ein spezieller Laufzeit-Klient implementiert (siehe Abbildung 9-7). Der realisierte „Standard-Klient“ unterstützt u.a. die Kontrolle und Überwachung von Workflow- und Aktivitäteninstanzen, die Präsentation von Arbeitslisten sowie die Ad-hoc-Definition von WF-Instanzänderungen. Er wurde auf der Grundlage der in Abschnitt 9.3.1 erwähnten Programmierschnittstellen implementiert, kann also bei Bedarf durch andere Bedieneroberflächen ersetzt werden. Eine Besonderheit des Demo-Klienten ist die Möglichkeit

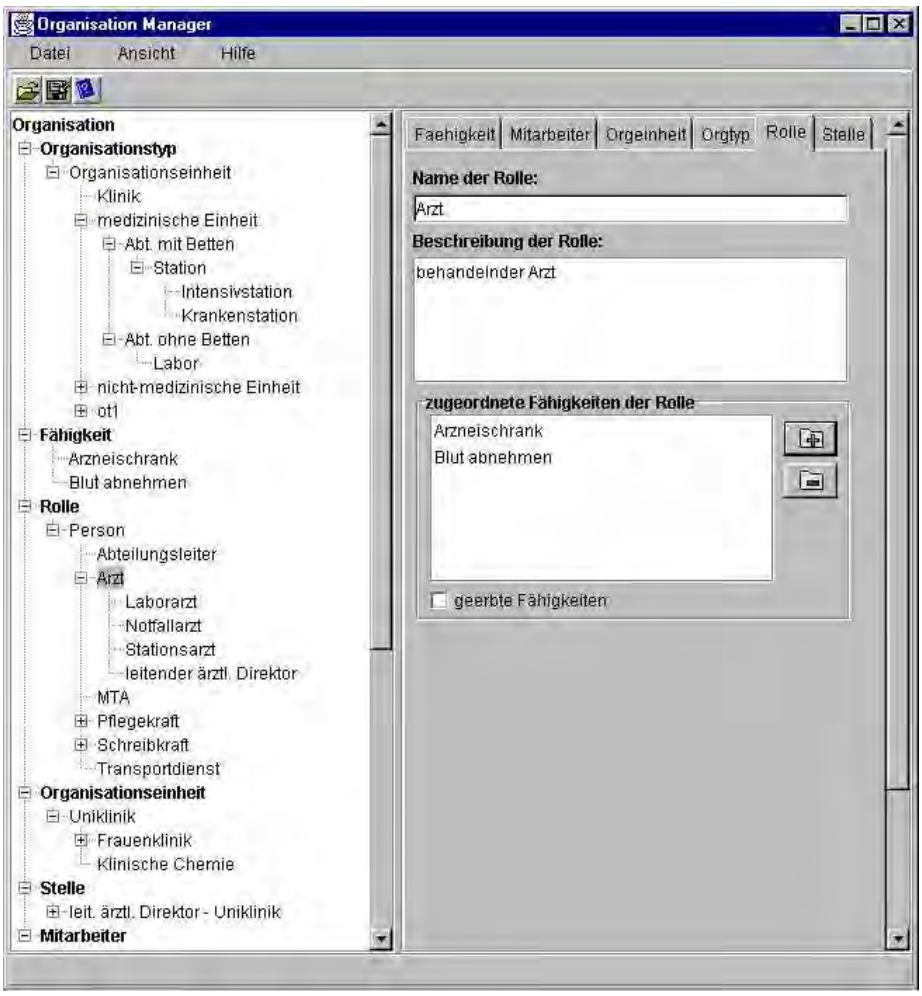


Abbildung 9-6: *Das Werkzeug ADEPT$_{orgmanager}$*

zur Visualisierung von WF-Ausführungsgraphen und ihrer Bestandteile (Kontroll- und Datenflußstruktur, Zeitattribute, Statusmarkierungen von Knoten und Kanten, Historieninformationen). Dadurch wird für den Anwender nicht nur der aktuelle Status einer WF-Instanz ersichtlich, sondern er kann auch die Auswirkungen durchgeführter WF-Änderungen und WF-Migrationen besser nachvollziehen

Um eine Ad-hoc-Änderung festzulegen, muß der betreffende WF-Ausführungsgraph nicht editiert, sondern die Änderung kann auf einer semantisch höheren Ebene definiert werden. Hilfreich sind hier, neben den ADEPT$_{flex}$-Basisoperationen, höherwertige Änderungsoperationen, wie das Einfügen einer neuen Aktivität zwischen zwei Knotenmengen oder das Vorziehen von Aktivitäten (vgl. Kapitel 7). Der Demo-Klient assistiert berechtigte Akteure bei der Anwendung dieser Operationen, indem er Informationen zum Kontext der Änderung (z.B. zu verfügbaren Arbeitsschritten, einzuhaltenden Reihenfolgen oder bestehenden Datenabhängigkeiten) bereitstellt, dem Benutzer eine Schnittstelle zur Festlegung der Abweichungen anbietet und ihn über (mögliche) Konsequenzen der beabsichtigen Änderung informiert (z.B. unversorgte Aufrufparameter, zeitliche Inkonsistenzen). Nach Durchführung der Änderung werden ihm die Ergebnisse (in visualisierter Form) angezeigt und andere betroffene Benutzer benachrichtigt (ggf. verbunden mit der Aktualisierung ihrer Arbeitslisten).

Der Demo-Klient übernimmt noch eine Reihe weiterer Aufgaben (z.B. An-/Abmelden eines Benutzers beim System, Anzeige von Terminüberschreitungen) auf deren Darstellung wir an dieser Stelle verzichten. Abschließend sei bemerkt, daß der derzeitige Demo-Klient nicht als Änderungsschnittstelle für „normale“ Anwender gedacht ist, sondern lediglich Demonstrationszwecken dient. Eine Diskussion zu Anforderungen an Endbenutzerschnittstellen adaptiver WfMS findet sich in [Saue96, RSD97].

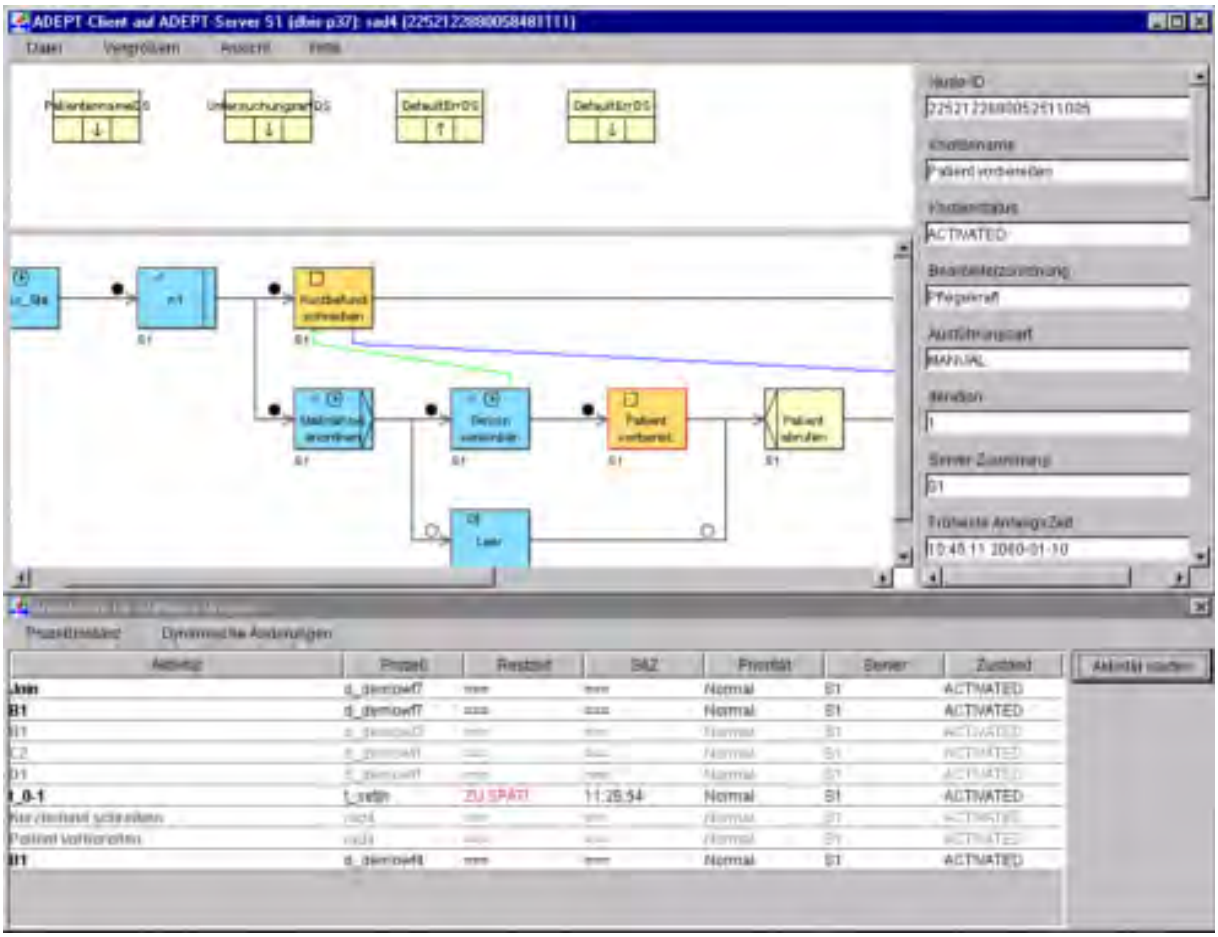


Abbildung 9-7: *Bedieneroberfläche des Demo-Klienten*

Teil IV

Diskussion und Zusammenfassung

10 Diskussion verwandter Ansätze

Die Mehrzahl der in der Literatur diskutierten Konzepte für adaptive WfMS (z.B. [KDB98, SiWe98]) ist zeitlich erst nach den ersten Veröffentlichungen zu ADEPT [Dada95, ReDa97] erschienen. Diese verwandten Arbeiten lassen sich verschiedenen Bereichen der Informatik zuordnen. Dem entsprechend vielfältig (und teilweise auch exotisch) sind die für die Verwirklichung adaptiver Workflows vorgeschlagenen Lösungskonzepte. Sie werden in diesem Abschnitt behandelt und gegenüber ADEPT bzw. $ADEPT_{flex}$ abgegrenzt. Als primäres Unterscheidungsmerkmal wählen wir dabei den von den Ansätzen jeweils verwendeten Basisformalismus zur Definition und Ausführung von Workflows. Er hat wesentlichen Einfluß darauf, auf welche Art und Weise entsprechende Workflows zur Laufzeit angepaßt werden können und welche Problemstellungen dabei zu lösen sind.

Generell kann zwischen applikativen Ansätzen, die auf existierenden Beschreibungsformalismen aufsetzen, und proprietären Ansätzen, die eigene Metamodelle verwenden, unterschieden werden. Konkret teilen wir die Ansätze für adaptive WfMS entlang der folgenden Beschreibungstechniken ein:

- Petri-Netze
- Erweiterte Zustandsautomaten (Statecharts)
- Graphen, Aktivitätennetze, Graphgrammatiken
- ECA-Regeln
- Deklarative WF-Beschreibungen, KI-Planer
- Transaktionale WF-Ausführungsmodelle
- Temporale Logiken, Prozeßalgebren und Prozeßgrammatiken
- Deskriptive WF-Beschreibungen
- Sonstige Beschreibungstechniken

Für einzelne Ansätze ist es nicht immer möglich, sie exakt einer dieser Kategorien zuzuordnen. Beispielsweise gibt es Vorschläge, bei denen der Standardablauf eines Workflows mittels einer graphischen Sprache festgelegt wird, wohingegen Ausnahmen und ihre Behandlung durch eine Menge von Regeln beschrieben werden [CFN99, MüRa99].

Die Praxistauglichkeit eines Workflow-Metamodells für (adaptive) WfMS muß sich an mehreren Kriterien messen lassen, wie Ausdrucksmächtigkeit, Strukturierbarkeit, Formalisierbarkeit, Ausführbarkeit, Analysier- und Validierbarkeit, Verständlichkeit, Wartbarkeit oder Adaptivität (vgl. Kapitel 3). Viele der in der Literatur diskutierten Vorschläge konzentrieren sich lediglich auf einige dieser Aspekte. Es ergeben sich zwar theoretisch interessante Resultate, ihre Anwendung in der Praxis scheitert dagegen meist. So basieren viele Vorschläge für adaptives Workflow-Management auf ausdrucksschwachen Formalismen, die sich nicht für die Unterstützung eines breiteren Spektrums an Prozessen eignen. Umgekehrt gibt es einige Ansätze, die ausdrucksstarke Metamodelle verwenden, die aber Anforderungen wie Korrektheit, Robustheit, Wartbarkeit oder Bedienbarkeit vernachlässigen.

Im folgenden behandeln wir Ansätze für adaptive WfMS, deren Metamodelle auf den obigen Beschreibungsformalismen basieren. Neben einer Diskussion ihrer generellen Stärken und Schwächen wird jeweils darauf eingegangen, welche Arten von Änderungen sich verwirklichen lassen, welche Ebene sie betreffen (WF-Schema, WF-Instanz), wie sie zu definieren sind und wie dabei die Korrektheit des Workflows gewährleistet wird. Dazu werden die wesentlichen Charakteristika der betrachteten Beschreibungsformalismen kurz vorgestellt und ihre grundlegende Eignung für die Verwirklichung adaptiver Workflows diskutiert. Alternativ dazu könnte als Kriterium für die Kategorisierung adaptiver Prozeßmodelle das jeweilige Einsatzgebiet herangezogen werden. Der Grund dafür ist, daß viele der in der Literatur diskutierten Ansätze auf die Unterstützung spezieller Prozesse zielen, etwa aus dem Entwicklungsbereich [SaBa93] oder der Büroautomatisierung [ElNu80]. Bei Entwicklungsprozessen („Engineering Workflow“) ist der genaue Ablauf abhängig von der Struktur des zu entwickelnden Produkts. Bei Büroprozessen stehen dagegen Formularflüsse im Vordergrund. Da wir uns primär für die Unterstützung von „Production Workflows“ interessieren (vgl. Kapitel 2), verzichten wir an dieser Stelle weitgehend auf solche anwendungsbezogenen Betrachtungen.

10.1 Vergleich mit Petri-Netz-basierten Ansätzen

Petri-Netze sind ein graphischer Formalismus zur Beschreibung und Analyse nebenläufiger Systeme [Baum96, Reis90]. Ihr Ursprung geht auf Arbeiten von C.A. Petri aus den 60er Jahren zurück [Petr62]. Die Idee, Petri-Netze auch zur Beschreibung von Arbeitsabläufen zu verwenden, entstand Mitte der 70er Jahre im Bereich der Büroautomatisierung [Zism77]. Weitere einschlägige Projekte, in denen Petri-Netze als Formalismus zur Modellierung und Ausführung von Prozessen verwendet wurden, sind Officetalk-D [ElBe82], DOMINO-W [KSW87], SPADE [BFG93] und INCOME [OSS94]. Ein Grundproblem der hier entwickelten Netzvarianten ist, daß lediglich starre Abläufe abbildbar sind [FaWi96]. Erst seit kurzem gibt es Untersuchungen zu dynamischen Netzänderungen, auf die im folgenden näher eingegangen wird. Zuvor fassen wir wichtige Grundlagen zu Petri-Netzen zusammen.

10.1.1 Grundlagen

Ein Petri-Netz ist ein gerichteter, bipartiter Graph mit zwei Knotenarten: *Stellen* und *Transitionen*. Erstere repräsentieren passive Zustandselemente (Bedingungen) und werden graphisch durch Kreise dargestellt, letztere beschreiben Aktionen, deren Darstellung durch Balken oder Rechtecke erfolgt. Stellen können mit Transitionen durch Pfeile verknüpft werden, wodurch die *Flußrelation* des Netzes festliegt. Die Menge der Stellen, von denen aus ein Pfeil zu einer Transition t führt, heißt *Vorbereich* von t. Umgekehrt ordnen wir Stellen, die Ziel eines von t ausgehenden Pfeils sind, dem *Nachbereich* von t zu. Knoten desselben Typs sind untereinander nicht verknüpfbar. Zur Abbildung des dynamischen Verhaltens werden Marken (engl. *Tokens*) verwendet, die Stellen zugeordnet werden können. Die aktuelle Markenbelegung eines Netzes definiert seinen Zustand, der als *Markierung* bezeichnet wird. Eine Markierung kann durch die Ausführung von Transitionen (*Schalten*) verändert werden. Im einfachsten Fall ist eine Transition *schaltbereit*, wenn alle Stellen aus ihrem Vorbereich mindestens eine Marke besitzen. Beim Schalten wird aus diesen Stellen jeweils eine Marke abgezogen und den Stellen des Nachbereichs jeweils eine Marke zugeführt. Ein Beispiel zeigt Abbildung 10-1a). Markierte bzw. sich in Ausführung befindliche Netze werden auch als *Systeme* bezeichnet. Insgesamt ist ein Netz durch einen bipartiten Graphen, eine Anfangsmarkierung M_0 sowie eine Menge von Schaltregeln charakterisiert. Abhängig von der für Stellen, Transitionen und Marken festgelegten Semantik sowie den verwendeten Schaltregeln ergeben sich unterschiedliche Netzvarianten. Differenzen bestehen hinsichtlich der Anzahl der Marken, die eine Stelle tragen darf oder die beim Schalten einer Transition von Stellen des Vorbereichs abgezogen bzw. in Stellen des Nachbereichs zugeführt werden. Des weiteren wird zwischen anonymen und individuellen Marken differenziert.

Die einfachste Petri-Netz-Variante bilden *Bedingungs/Ereignis-Netze* (B/E-Netze). Hier repräsentieren Stellen einfache Bedingungen, die entweder wahr (Stelle trägt eine Marke) oder falsch (Stelle trägt keine Marke) sind. Marken besitzen dementsprechend keine Struktur, sondern dienen lediglich der Darstellung von Wahrheitswerten. Transitionen beschreiben Ereignisse, die genau dann eintreten, wenn alle Bedingungen ihres Vorbereichs gültig und alle Bedingungen ihres Nachbereichs ungültig sind. Beim Eintreten des Ereignisses werden umgekehrt die Bedingungen des Vorbereichs ungültig

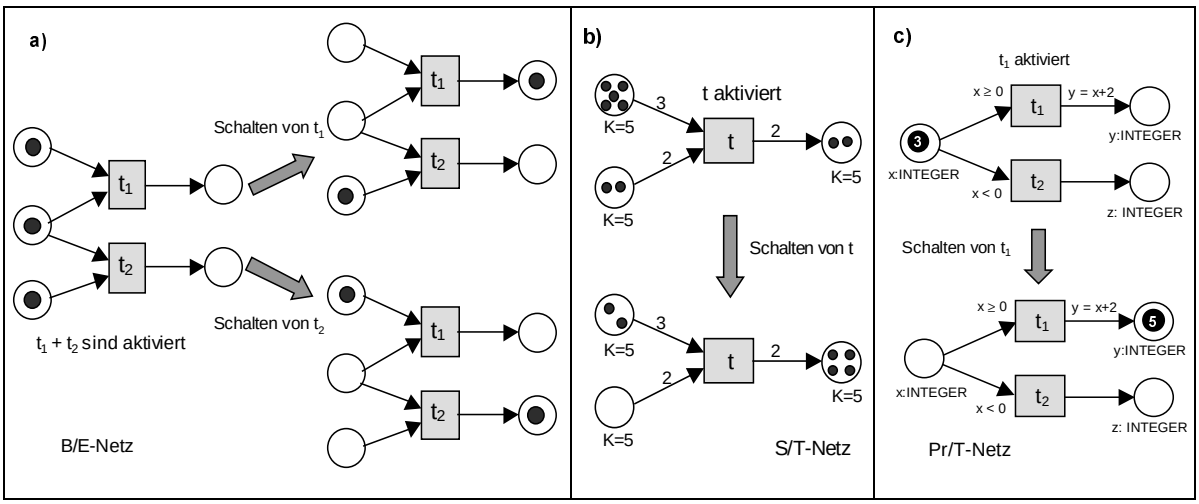


Abbildung 10-1: *Schaltverhalten in B/E-, S/T- und Pr/T-Netzen*

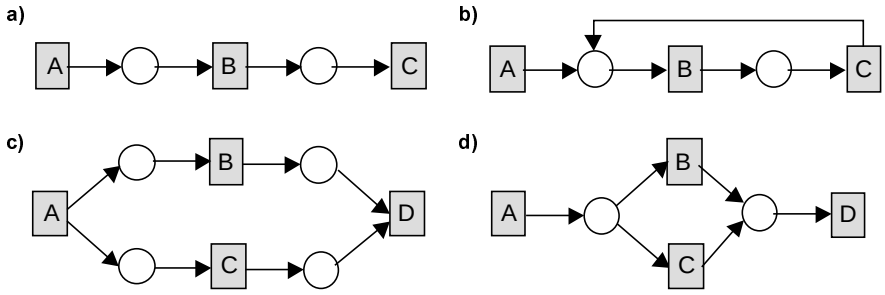


Abbildung 10-2: *Modellierung von Kontrollflußstrukturen mit B/E-Netzen - (a) Sequenz (b) Iteration (c) Parallele Verzweigung (d) Bedingte Auswahl*

und die des Nachbereichs gültig. Mit B/E-Netzen lassen sich sequentielle, nebenläufige und sich wechselseitig ausschließende Aktivitäten modellieren (siehe Abbildung 10-2).

Im Gegensatz zu B/E-Netzen kann in *Stellen/Transitions-Netzen* (S/T-Netze) eine Stelle mehr als eine Marke tragen, wobei die maximale Anzahl durch die Vorgabe einer Kapazität beschränkt werden kann. Darüber hinaus besteht die Möglichkeit, Kanten mit Gewichten zu versehen. Sie legen fest, wieviel Marken beim Schalten der zugehörigen Transition aus der assoziierten Stelle abgezogen bzw. in diese zugeführt werden sollen. Eine Transition *t* eines S/T-Netzes kann genau dann schalten, wenn für jede Stelle *s* ihres Vorbereichs nicht mehr Marken abgezogen werden sollen als in *s* liegen. Umgekehrt muß für jede Stelle des Nachbereichs genügend freier Platz vorhanden sein, um die zufließenden Marken aufnehmen zu können (vgl. Abbildung 10-1 b).

Aufgrund des geringen Strukturierungsgrades von B/E- und S/T-Netzen können unübersichtliche und kaum mehr verständliche Modelle resultieren [ElNu93]. Größtes Manko ist, daß anonyme Marken keine Werte annehmen können. Dadurch sind Datenflüsse nicht modellierbar. Ebensowenig ist es möglich, das Schalten von Transitionen abhängig von prädikativen Bedingungen vorzunehmen [Part98]. B/E- und S/T-Netze sind deshalb in ihrer Grundform für die Modellierung von Workflows nicht geeignet. Um diesen Mängeln zu begegnen, wurden erweiterte Petri-Netze entwickelt. Bei *gefärbten Petri-Netzen* [Jens90] sind Marken unterscheidbar. In *Prädikat/Transitions-Netzen* (Pr/T-Netze) besitzen sie zusätzlich eine Struktur [Genr91], so daß Datenflüsse modellierbar sind. Darüber hinaus können prädikative Schaltbedingungen für Transitionen definiert werden. Dabei tragen Marken die Werte für freie Variablen, die zur Bildung dieser Prädikate verwendet werden. In der Regel wird zwischen Eingangs- und Ausgangsbedingungen einer Transition unterschieden. Die Transition kann schalten, wenn die festgelegte Eingangsbedingung für alle Marken aus Stellen ihres Vorbereichs gültig ist. Zusätzlich müssen neue Werte für Marken aus dem Nachbereich, entsprechend der festgelegten Ausgangsbedingungen, berechnet werden. Ein Beispiel zeigt Abbildung 10-1 c). Durch die Verwendung von Variablen lassen sich komplexe Sachverhalte übersichtlicher modellieren als bei B/E- oder S/T-Netzen. Auch Pr/T-Netze haben sich in ihrer Grundform als ungeeignet für die WF-Modellierung erwiesen [ElNu93]. Die Netze sind für Anwender immer noch schwer verständlich. Darüber hinaus gestaltet sich der Einsatz herkömmlicher Pr/T-Netze problematisch, wenn sie in Kombination mit Datenmodellen eingesetzt werden, die auch komplexe Objektstrukturen zulassen [Ober96, OSS94].

Ein Formalismus, der speziell für die Prozeßmodellierung entwickelt wurde, sind *FunSoft-Netze* [Gruh91, DeGr94, DLS96]. In der Terminologie dieser Netze werden Stellen als *Kanäle* und Transitionen als *Instanzen* bezeichnet. Kanäle repräsentieren typisierte Objektspeicher. Instanzen entsprechen Aktivitäten, die mit Anwendungskomponenten verknüpft oder durch Sub-Netze verfeinert werden. Der Datenfluß wird definiert, indem Kanäle und Instanzen durch *Pfeile* verbunden werden. Die in eine Instanz einmündenden bzw. von ihr ausgehenden Pfeile legen fest, aus welchen Kanälen Eingabeobjekte gelesen und in welche Kanäle Ausgabeobjekte abgelegt werden sollen. Zusätzlich kann für einmündende Pfeile angegeben werden, ob für das Schalten der Instanz lediglich eine Kopie des betreffenden Objektes (Pfeilbeschriftung `CO`) oder das Objekt exklusiv (Pfeilbeschriftung `IN`) benötigt wird. Im erstgenannten Fall wird die Objektmarke nicht aus dem Eingangskanal abgezogen. Besitzt ein Kanal mehrere Marken, so kann die Reihenfolge des Zugriffs (FIFO, LIFO, wahlfrei) explizit angegeben werden. Schließlich können für Instanzen unterschiedliche Schaltmodi vereinbart werden, die festlegen, ob sie beim Schalten ein oder mehrere Objekte in einem Ausgangskanal ablegen oder ob sie

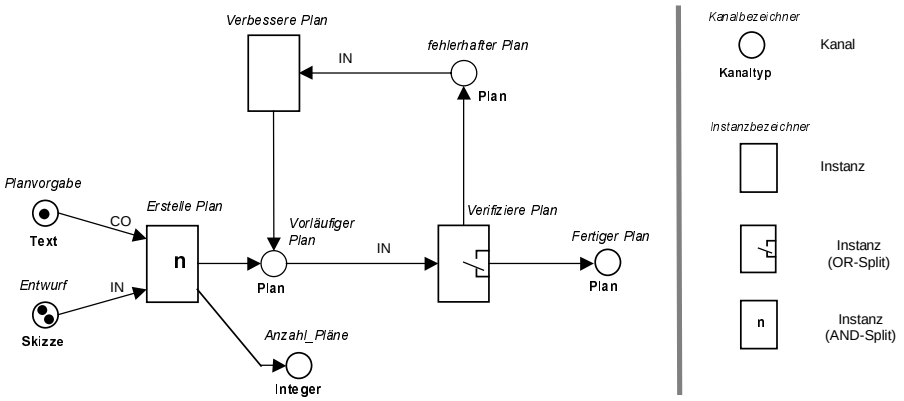


Abbildung 10-3: *Beispiel eines einfachen FunSoft-Netzes*

alternative Ausgaben produzieren dürfen. Im letztgenannten Fall füllt die Instanz die Kanäle ihres Nachbereichs nur selektiv. Ein Beispiel zeigt Abbildung 10-3.

Petri-Netze sind mathematisch fundiert und besitzen ein formales Ausführungsmodell, so daß sie sich gut für eine Rechnerunterstützung eignen. Für erweiterte Netze sind in eingeschränktem Umfang auch formale Überprüfungen auf Vollständigkeit und Widerspruchsfreiheit möglich. Wichtige Dynamik-Eigenschaften, wie Verklemmungsfreiheit und Lebendigkeit, denen im WF-Bereich eine hohe Bedeutung zukommt, sind theoretisch ebenfalls ausreichend untersucht. Allerdings können Analysen des dynamischen Verhaltens sehr komplex und für große Netze ineffizient werden. Um etwa festzustellen, ob ein WF terminiert oder jeder seiner Schritte aktivierbar ist, müssen aufwendige Erreichbarkeitsanalysen durchgeführt werden. Häufig müssen dazu alle von der Anfangsmarkierung des Netzes aus erreichbaren Folgemarkierungen berechnet werden, was sich aufgrund der kombinatorischen Vielfalt möglicher Markierungen mit zunehmender Netzgröße immer aufwendiger gestaltet. Da bei S/T-Netzen die Menge erreichbarer Markierungen nicht immer endlich ist, reicht eine schlichte Erreichbarkeitsanalyse, in deren Verlauf jede erreichbare Markierung irgendwann produziert wird, nicht aus.

Solange Petri-Netze ausschließlich zu Modellierungszwecken eingesetzt werden, fällt der für die Analysen zu treibende Aufwand nicht sonderlich ins Gewicht [AAH98, HVA97]. Sobald sie als Grundlage für die Steuerung und ggf. dynamische Änderung von WFs dienen sollen, gilt dies nicht mehr, insbesondere wenn eine große Anzahl von Instanzen und Benutzern unterstützt werden muß. Im ADEPT-Basismodell werden die für WFs geforderten Dynamik-Eigenschaften in der Mehrzahl der Fälle bereits durch die Art der Konstruktion der WF-Graphen sichergestellt. Erreichbarkeitsanalysen sind lediglich bei Verwendung strikter Sync-Kanten notwendig, ihre Komplexität kann aber durch die Ausnutzung der Struktureigenschaften der WF-Graphen noch reduziert werden. Petri-Netze weisen eine Reihe weiterer Nachteile auf, wenn sie zur Modellierung und Steuerung von WFs verwendet werden:

- *Mangelhafte Trennbarkeit unterschiedlicher Modellaspekte*: Semantisch unterschiedliche Modellaspekte wie Kontroll- und Datenfluß sind nicht klar voneinander trennbar, sondern müssen auf der Grundlage derselben Modellelemente (Stellen, Transitionen, Marken) beschrieben werden. Dem entsprechend basieren alle Korrektheitsprüfungen auf denselben aufwendigen Verfahren. Des weiteren können bereits für einfache Prozesse unübersichtliche und kaum mehr verständliche Modelle resultieren, für die auch keine Sichtenbildung möglich ist, etwa durch Ausblenden des modellierten Datenflusses. Dies hat dazu geführt, daß sich bei Petri-Netzen die Modellbildung sehr stark an Informationsflüssen orientiert, während sich Kontrollflüsse mehr oder weniger implizit aus den festgelegten Datenflußbeziehungen ergeben.

- *Nachteile Petri-Netz-basierter Ausführungsmodelle*: Aus der aktuellen Markierung eines Netzes ist nicht direkt erkennbar, wie sich der bisherige Verlauf des zugehörigen WF darstellt. Um Fragestellungen beantworten zu können wie „Welche Abschnitte des Netzes wurden bereits abgearbeitet?“ oder „Welche Bereiche können in der Folge noch durchlaufen werden?“ müssen zusätzliche Informationen verwaltet oder aufwendige Erreichbarkeitsanalysen durchgeführt werden. Darüber hinaus gibt es keine allgemeinen Kriterien, die Aussagen zur erfolgreichen Terminierung eines Netzes erlauben. Statt dessen muß der Modellierer zulässige Endmarkierungen explizit vorgeben, so daß das System in der Lage ist, sie von Markierungen zu unterscheiden, die Folge

eines *Deadlocks* sind. Der von ADEPT verfolgte Markierungsansatz, bei dem auch nicht zur Ausführung kommende Zweige markiert werden (*True*-/*False*-Semantik), bietet hier Vorteile. Darüber hinaus kann in ADEPT aus den Markierungen eines Ausführungsgraphen direkt auf die internen Bearbeitungszustände von Aktivitäten geschlossen werden, was für Petri-Netz-basierte Modelle üblicherweise nicht zutrifft. Wie wir in Kapitel 7 gesehen haben, sind solche Statusinformationen für die Prüfung der Anwendbarkeit von Änderungsoperationen essentiell.

- *Implizite Modellierung von Kontrollflußstrukturen*: Mit Petri-Netzen lassen sich zyklische Prozeßstrukturen beschreiben. Ihre Festlegung erfolgt „implizit", so daß für den Netzinterpreter beim Schalten einer Transition nicht direkt feststellbar ist, ob ein Rücksprung im Kontrollfluß erfolgt oder nicht. Solange es nur um die Interpretation der Netze geht, fällt dieser Aspekt nicht ins Gewicht. Sollen jedoch Netze während ihrer Ausführung abgeändert werden, können Probleme resultieren. So würde sich allein die Behandlung der in Kapitel 8 diskutierten Konzepte aufwendig gestalten. Die fehlende Möglichkeit zur expliziten Abbildung von Schleifen ist auch nachteilig für die Lesbarkeit der WF-Modelle. So können „korrekte" Zyklen nicht von Zyklen, die Folge einer fehlerhaften Modellierung sind, unterschieden werden.

Als wichtigste Argumente für die Verwendung von Petri-Netzen werden die sehr gut entwickelte Theorie und die Verfügbarkeit von Analyseverfahren genannt [Aals98]. Dem gegenüber stehen die beschriebenen Limitationen. Noch nicht diskutiert haben wir bisher Aspekte, die Anpassungsfähigkeit von Petri-Netzen betreffend. Für adaptive WfMS wichtige, in den genannten Formalismen kaum oder gar nicht betrachtete Fragestellungen betreffen die Vormodellierung planbarer Abweichungen, die dynamische Änderung von Workflows (unter Erhalt ihrer Konsistenz), die Behandlung semantischer Fehler (z.B. durch kontrolliertes Zurücksetzen), die Integration operativer Sicherheitsanforderungen (z.B. Parameterversorgung gerufener Aktivitäten) sowie die Einbeziehung von Endbenutzeraspekten.

10.1.2 Adaption in Petri-Netz-basierten WF-Modellen

Erweiterte Petri-Netze wurden im Bereich der Büroautomatisierung bereits Anfang der 80er Jahre für die Ablaufmodellierung und -steuerung eingesetzt [ElNu80]. Schon damals hat sich gezeigt, daß eine starre Ausführung vormodellierter Netze nicht immer akzeptabel ist [ElBe82]. Die vorgeschlagenen Lösungen hatten allerdings eher exemplarischen Charakter und waren meist auf bestimmte Arten von Anwendungen (z.B. Unterstützung von Formularflüssen) beschränkt. In diesem Abschnitt gehen wir auf die in der aktuellen Literatur diskutierten Ansätze zur Adaption von Petri-Netzen ein. Diese lassen sich grob in zwei Gruppen einteilen, die entweder die *Anpassung von Netzmarkierungen* oder die *Änderung der Netzstruktur* zum Schwerpunkt haben. Diese beiden grundlegenden Ansätze werden ergänzt um Konzepte zur *Vormodellierung* von Ausnahmebehandlungen und um einfache Mechanismen zur Prozeßadaption, wie das *dynamische Binden* und das *späte Modellieren* von Sub-Netzen.

10.1.2.1 Adaption durch Vormodellierung von Ausnahmebehandlungen

Eines der ersten Petri-Netz-basierten Vorgangssteuerungssysteme war DOMINO-W [KSW87, KrWo87]. Bei diesem WfMS können Formularflüsse mit Hilfe eines interaktiven Planungswerkzeugs und einer Wissensbasis graphisch entworfen werden [ViSo91]. Die resultierenden WF-Modelle werden in eine Darstellung der Sprache CoPlan-S transformiert, deren formale Ausführungssemantik durch Pr/T-Netze festgelegt ist. Dadurch lassen sich die bekannten Analyseverfahren anwenden. Nach erfolgreicher Überprüfung wird das Netz in ein ausführbares Programm übersetzt, das zur Laufzeit den Formularfluß (entsprechend der definierten Reihenfolge) steuert. Zur Ausnahmebehandlung werden einfache Mechanismen angeboten, etwa zur Weiterleitung eines Formulars oder zum Zurücksetzen der Bearbeitung. Komplexere Ausnahmebehandlungen müssen dagegen bereits beim WF-Entwurf vorgesehen werden, da strukturelle Netzmodifikationen zur Laufzeit nicht unterstützt werden. Die Inflexibilität dieser Konzepte wurde von den Autoren später selbst eingeräumt [Krei91, Krei91a].

Ebenso wie DOMINO-W geht die meisten Petri-Netz-basierten WfMS davon aus, daß alle für die WF-Steuerung relevanten Ausnahmen und die zu ihrer Behandlung notwendigen Aktionen bereits beim WF-Entwurf berücksichtigt werden können [KaDe96, DGSZ94, Ober96]. Im Unterschied zum ADEPT-Modell bieten diese Systeme keine speziellen Konzepte zur Vormodellierung von Abweichungen, etwa um die unterschiedliche Bedeutung einer Aktivität zu verschiedenen Ausführungszeit-

punkten (Normal- vs. Ausnahmeschritt) auszudrücken. Statt dessen müssen Abweichungen mit denselben Sprachmitteln beschrieben werden wie Standardabläufe, was zu schwer verständlichen Netzen und einem explosionsartigen Anwachsen der Komplexität von Analyseverfahren führt [EKR95]. Der Vorteil der formalen Überprüfbarkeit der Modelle besitzt dann nur noch theoretischen Charakter. In [Ober90] werden Petri-Netze als Formalismus zur Beschreibung von Ausnahmebehandlungen vorgeschlagen. Dies wird damit begründet, daß entsprechende Prozesse sehr umfangreich und komplex sein können, so daß ihre Unterstützung dieselben Steuerungsmechanismen erfordert, die auch für normale Abläufe notwendig sind. In [ObSt91] werden semi-formale Netztypen (Kanal-/Instanzen-Netze) zur Beschreibung schwach strukturierter Ausnahmeprozesse vorgeschlagen. Für die strukturierbaren Prozeßanteile können schrittweise formale Beschreibungen in Form von Pr/T-Netzen festgelegt werden.

10.1.2.2 Adaption durch spätes Binden oder spätes Modellieren von Sub-Netzen

Einige Petri-Netz-basierte WF-Metamodelle begegnen dem Vorwurf der Starrheit durch die Bereitstellung von Mechanismen für das dynamische Binden [BFG93, Han97] und das späte Modellieren [HHJS97] von Sub-Workflows. Die Grundannahme dieser Ansätze ist, daß sich Prozesse auf abstrakter Ebene, d.h. auf höheren Hierarchiestufen, gut strukturieren lassen, während dies mit zunehmender Schachtelungstiefe eines Sub-Prozesses nicht immer a priori möglich ist.

a) Spätes Binden von Sub-Workflow-Modellen in HOON

HOON [HHSW96, Han97] sieht den Hauptgrund für die Inflexibilität Petri-Netz-basierter WF-Modelle darin, daß die zur Bearbeitung von Aktivitäten benötigten Ressourcen – hierzu zählen auch Aktivitätenprogramme und Sub-Workflow-Modelle – bereits bei der Netzmodellierung festgelegt werden müssen. Als Folge dieser statischen Zuordnung können, selbst bei einfachen Änderungen wie dem Austausch eines Aktivitätenprogramms oder der Modifikation eines Sub-Workflows, Anpassungen des Gesamtmodells und ggf. dessen Neuübersetzung erforderlich werden. Um dieses Manko zu beheben, wurde in HOON ein erweiterter Netz-Formalismus entwickelt [LöHa94], bei dem die zur Ausführung von Aktivitäten benötigten Ressourcen erst zur Laufzeit, beim Schalten der Transitionen, gebunden werden. Dadurch können z.B. noch nicht instanziierte Sub-Netze strukturell modifiziert werden, ohne daß das übergeordnete Netz angepaßt werden muß. Darüber hinaus besteht die Möglichkeit, beim Binden zwischen verschiedenen Ressourcen (z.B. Sub-Netzen) auszuwählen, wobei die Selektion entweder automatisch durch das WfMS oder manuell durch einen Benutzer erfolgen kann. Für den erstgenannten Fall muß der betreffenden Transition ein *Ressourcenkontrollnetz* zugewiesen werden, das den Auswahlprozeß steuert. Insgesamt läßt sich mit diesem Ansatz die hierarchische Struktur eines Netzes dynamisch konfigurieren, was für verschiedene Prozeßinstanzen zu unterschiedlichen Schachtelungstiefen führen kann. Das Konzept des späten Bindens ist aber nicht ausreichend, um adaptive Anwendungssysteme realisieren zu können. Da instanziierte Netze strukturell nicht modifiziert werden können, lassen sich bereits einfache Anpassungen nicht abbilden.

b) Spätes Modellieren von Sub-Workflows in MOVE

In MOVE [HHJS97, HeJu96] werden für die WF-Modellierung *FunSoft-Netze* (vgl. Abschnitt 10.1.1) verwendet. MOVE erweitert diesen Formalismus um die Möglichkeit, Sub-WFs mit nicht oder nur teilweise planbarer Ablaufstruktur erst zur Ausführungszeit einer WF-Instanz zu definieren (*Late Modeling*). Allerdings muß zur Entwurfszeit des Netzes feststehen, für welche Transitionen zur Laufzeit neue Sub-Netze angelegt bzw. teilweise definierte Sub-Netze geändert werden dürfen. Zu diesem Zweck kennzeichnet der Modellierer die Transitionen explizit und legt für jede von ihnen fest, wer Änderungen ihres Sub-Netzes vornehmen darf und unter welchen Bedingungen sie möglich sein sollen. Letzteres erfolgt durch die Angabe einer Menge von Stellen aus dem Vorbereich der Transition, deren Belegung notwendige Voraussetzung für das späte Modellierern ist. Während eines Editiervorgangs können andere Teile des übergeordneten Netzes parallel weiterbearbeitet werden, vorausgesetzt ihre Ausführung ist unabhängig vom Ergebnis des modellierten Sub-Netzes. Erst wenn der Editiervorgang abgeschlossen und alle weiteren Vorbedingungen für die Ausführung der Transition erfüllt sind, kann das neu modellierte Netz als Sub-WF instanziiert werden. Zu erwähnen bleibt, daß die Festlegung eines Sub-Netzes zur Laufzeit immer im Kontext einer konkreten WF-Instanz erfolgt und sich dem entsprechend nur auf deren Ausführung auswirkt. Ein einfaches Beispiel zeigt Abbildung 10-4.

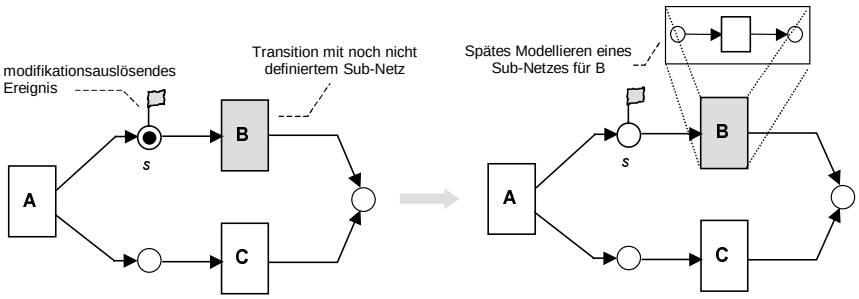


Abbildung 10-4: *Spätes Modellieren von Sub-WFs* (*Das Sub-Netz B kann editiert werden, wenn die gekennzeichnete Stelle s mit einer Marke belegt wird. Die Transition C kann parallel dazu ausgeführt werden*)

Das späte Modellieren von Sub-WFs stellt eine wichtige Funktion eines adaptiven WfMS dar. Die Implementierung dieses Konzeptes ist trivial, da das Editieren eines Sub-Netzes stets vor seiner Instanziierung abgeschlossen sein muß. Strukturelle Änderungen eines sich in Ausführung befindlichen Netzes sind damit aber nicht realisierbar. Vielmehr wird die Modifizierbarkeit von WF-Instanzen auf vorab definierte Bereiche eingeschränkt, was voraussetzt, daß die nicht planbaren Anteile zur Modellierungszeit bekannt sind. Komplexe Ausnahmebehandlungen und Ad-hoc-Abweichungen sind dagegen nicht möglich. Andere wichtige Aspekte, wie das späte Modellieren von Sub-Netzen bei zyklischen Prozessen, werden ebenfalls nicht behandelt. Schließlich setzt der vorgestellte Ansatz voraus, daß Endanwender in der Lage sind, einen Workflow mit Hilfe eines Petri-Netz-Editors abzuändern.

10.1.2.3 Adaption durch dynamische Änderung von Netzmarkierungen

In diesem Abschnitt behandeln wir Petri-Netz-basierte WF-Metamodelle, bei denen das Verhalten von Workflows durch die Anpassung von Netzmarkierungen beeinflußt werden kann.

a) Dynamische Änderungen von Netzmarkierungen in Chautauqua

Chautauqua [ElMa97] ist ein Petri-Netz-basiertes WfMS, bei dem durch die Änderung der Lokation von Marken einfache Ad-hoc-Eingriffe in die WF-Kontrolle verwirklichbar sind. Für die Modellierung werden BICN-Netze (*Basic Information Control Nets*) verwendet. Ein solches Netz besitzt je einen ausgezeichneten Start- und Endknoten. Neben normalen Aktivitätenknoten gibt es noch spezielle Knoten für Aufsplittungen und -Zusammenführungen des Kontrollflusses. Für jedes Netz wird ein Vorgangsformular definiert, das während der Ausführung von WF-Instanzen bearbeitet werden kann.

Zur Laufzeit referenzieren alle Instanzen eines bestimmten WF-Typs dasselbe Netz. Dazu wird bei ihrer Erzeugung jeweils eine individuelle Marke angelegt, die Träger von Kontrolldaten (z.B. Prozeß-Id) und Anwendungsdaten (= Vorgangsformularinstanz) ist. Initial wird diese Marke dem Startknoten des Netzes zugeordnet und durchläuft dann die verschiedenen Aktivitäten. In jedem Arbeitsschritt können die Felder des zugeordneten Vorgangsformulars selektiv editiert werden. Jede umlaufende Marke besitzt einen internen Zustand, der Auskunft über den Status des Arbeitsschrittes gibt, dem sie aktuell zugeordnet ist. Zustandsübergänge können durch einfache Operationen, etwa dem Editieren des Vorgangsformulars oder der Beendigung der Formularbearbeitung, herbeigeführt werden. Beim Übergang in den Zustand *Completed* wird die Marke, entsprechend der definierten Ablauflogik, an die nächste Lokation im Netz weitergeleitet. Mit Ausnahme paralleler Ausführungszweige befindet sich die Marke einer WF-Instanz immer genau an einer Lokation des Netzes. Bei OR-Aufsplittungen muß der zu wählende Zweig vom Bearbeiter der unmittelbar vorangehenden Aktivität festgelegt werden. Bei AND-Aufsplittungen werden Kopien der Marke erzeugt, die dann in den verschiedenen Teilzweigen unabhängig voneinander bearbeitbar sind. Diese Kopien müssen bei UND-Zusammenführungen des Kontrollflusses wieder „verschmolzen“ werden, wobei Schreibkonflikte auf Formularfeldern erst zu diesem Zeitpunkt erkannt und dem Anwender in einer Konfliktliste angezeigt werden. Wenn eine Marke den Endknoten erreicht, terminiert die Ausführung der zugehörigen WF-Instanz.

In [ElMa97, ElNu93] wird der *SendTo*-Operator vorgestellt, mit dem sich die Marken (Formularinstanzen) eines BICN-Netzes „verschieben“ lassen. Hierzu muß der Anwender durch Angabe eines 3-Tupels (*Akteur*, *Rolle*, *Aktivität*) festlegen, an welche Lokation im Netz und an welchen Akteur die Marke weitergeleitet werden soll. Die einzelnen Komponenten sind optional, wodurch verschiedene Freiheitsgrade resultieren. Ein Beispiel zeigt Abbildung 10-5. Das links dargestellte Netz besitzt zwei

Abbildung 10-5: *Überspringen von Aktivitäten durch Verschieben von Marken*

Marken, die von unterschiedlichen WF-Instanzen stammen ($inst_1$ und $inst_2$). Angenommen für die Instanz $inst_1$ - sie kontrolliert aktuell die Aktivität *A* - soll die Ausführung von *B* bzw. *C* ausgelassen und mit der Kontrolle direkt bei *D* (Bearbeiter: „Müller“) fortgefahren werden. Um diesen Vorwärtssprung zu realisieren, muß die Bearbeitung von *A* beendet und die Instanzmarke mittels *SendTo* an die Adresse *(Müller, -, D)* weitergeleitet werden. Hieraus ergibt sich das rechts abgebildete Netz.

Auf ähnliche Art und Weise lassen sich einfache Rücksprünge und eine Reihe weiterer Ausnahmebehandlungen (z.B. Weiterleitung einer Aufgabe an anderen Akteur) verwirklichen [ElNu93]. Unklar ist, ob der SendTo-Operator bei parallelen Ausführungszweigen korrekt anwendbar bleibt. Hier können mehrere Kopien einer Marke vorliegen, so daß durch das Verschieben einzelner Marken inkonsistente Markierungen resultieren können, die schlimmstenfalls zu Verklemmungen führen.

b) Dynamische Änderung von Netzmarkierungen in MWMS

MWMS bietet ebenfalls einen Petri-Netz-basierten Ansatz, um spontane Vorwärts- und Rückwärtssprünge in adaptiven WfMS zu ermöglichen [AgMi96, AMP94]. Im Gegensatz zu Chautauqua werden inkonsistente Markierungen ausgeschlossen. MWMS betrachtet ausschließlich Kontrollflußaspekte, was gegenüber ADEPT eine signifikante Einschränkung bedeutet. Für die Modellierung werden *elementare Petri-Netze* (*EN-Netze*) verwendet, eine spezielle Variante von Ein-Marken-Netzen. Sie weisen gegenüber den in Abschnitt 10.1.1 erwähnten B/E-Netzen strukturelle Einschränkungen auf, verhalten sich ansonsten aber ähnlich. Ausgeschlossen werden Zyklen, darüber hinaus muß jede Transition einen nicht-leeren Vor- und Nachbereich besitzen, wobei eine dieser beiden Mengen einelementig sein soll. Schließlich dürfen verschiedene Transitionen nicht denselben Vor- und Nachbereich besitzen. Für *EN-Systeme* wird zusätzlich gefordert, daß bei der Markierung der Stellen aus dem Vorbereich einer Transition, die Stellen ihres Nachbereichs keine Marken tragen.

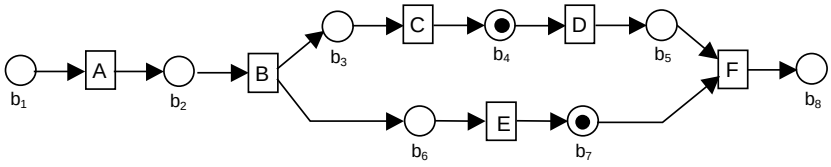


Abbildung 10-6: *Beispiel eines EN-Systems (Aktivität D kann schalten)*

In Ausnahmesituationen kann es erforderlich werden, die Bearbeitung von Aktivitäten zu wiederholen oder auszulassen. Soll in dem obigen WF die Ausführung der Aktivität *D* übersprungen werden, muß die Markierung des EN-Systems angepaßt werden (Entfernen der Marke aus b_4 und Hinzufügen einer Marke zu b_5), um mit der Interpretation des Netzes fortfahren zu können. Markierungsänderungen führen dabei nicht immer zu einem konsistenten Folgezustand. Beispielsweise würde das Verschieben der Marke aus b_4 in die Stelle b_8 zu einer Verklemmung führen. Um solche Fälle auszuschließen, berechnet MWMS für jeden erreichbaren WF-Zustand alle Markierungen, die als Sprungziele in Frage kommen, automatisch und bietet sie Benutzern zur Auswahl an. Zu diesem Zweck wird für jedes EN-Netz sein *Erreichbarkeitsgraph* bestimmt. Seine Knoten repräsentieren globale Zustände, die von der Anfangsmarkierung des Netzes durch Schalten einer Folge von Transitionen erreichbar sind [Baum96]. Eine Kante $M_1 \rightarrow M_2$ mit Beschriftung *t* drückt aus, daß die Transition *t* unter M_1 aktiviert ist und beim Schalten von *t* die Markierung M_2 resultiert. Aufgrund der für EN-Netze getroffenen Einschränkungen ist ihr Erreichbarkeitsgraph stets endlich und azyklisch. Er wird für die Erzeugung eines endlichen Automaten (*elementares Transitionssystem*; *ET-System*) verwendet, dessen aktueller Zustand jeweils die aktuelle Markierung des zugehörigen EN-Systems widerspiegelt. Abbildung 10-7

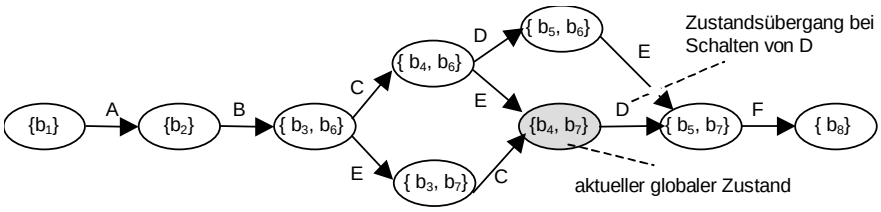


Abbildung 10-7: *Beispiel eines ET-Systems*

zeigt das zum EN-System aus Abbildung 10-6 gehörende ET-System. Es bildet die Basis für die automatische Berechnung aller im Zustand $\{b_4, b_7\}$ zulässigen Sprungziele.

Grundsätzlich unterscheidet MWMS zwischen Vorwärts- und Rückwärtssprüngen [AgMi98]. Durch einen Vorwärtssprung (*Roll Forward*) können Markierungen erzeugt werden, die normalerweise erst über eine Schaltfolge von Transitionen vom aktuellen Zustand aus erreichbar sind. Im Beispiel ergeben sich $\{b_8\}$ und $\{b_5, b_7\}$ als mögliche Sprungziele. Ein Vorwärtssprung nach $\{b_5, b_7\}$ würde bewirken, daß *D* übersprungen und *F* ausführbar wird. Die möglichen Ziele eines Rückwärtssprungs (*Roll Back*) sind im Beispiel durch die Vorgängerknoten von $\{b_4, b_7\}$ festgelegt: $\{b_1\}$, $\{b_2\}$, $\{b_3, b_6\}$, $\{b_4, b_6\}$ und $\{b_3, b_7\}$. Soll etwa in den Zustand vor Ausführung von *B* zurückgesprungen werden, muß als Sprungziel $\{b_2\}$ gewählt werden. MWMS unterscheidet weiter zwischen strikt und schwach linearen Sprüngen. Bei ersteren (*strongly linear jumps*) wird genau eine Marke verschoben (z.B. Vorwärtssprung nach $\{b_5, b_7\}$), während bei letzteren (*weakly linear jumps*) mindestens 2 Marken entfernt und eine hinzugefügt werden (z.B. Vorwärtssprung nach $\{b_8\}$).

Aspekte *dynamischer Strukturänderungen* sind in MWMS bisher nur oberflächlich untersucht worden [AgMi98]. Vorgesehen sind drei Operatoren, deren Anwendung eine der folgenden Änderungen bewirken soll: Parallelisierung bisher aufeinanderfolgender Transitionen, Sequenzialisierung bisher paralleler Transitionen und Vertauschen zweier Transitionen. Abbildung 10-8 zeigt die Anwendung dieser Operatoren auf ET-Systeme. Hervorgehoben sind die Zustände, in denen die betreffenden Modifikationen in keinem Fall zulässig sind. So kann die Ausführungsreihenfolge zweier Transitionen *A* und *B* nach dem Schalten von *A* nicht vertauscht werden. Ein Einfügeoperator ist nicht vorgesehen.

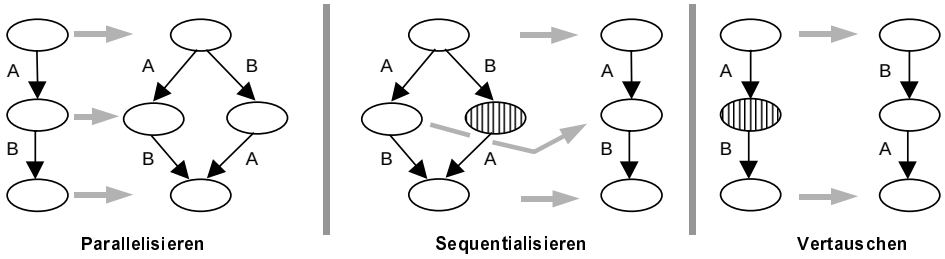


Abbildung 10-8: *Vorgesehene Änderungsoperationen in MWMS (ET-System-Sicht)*

Der von MWMS verfolgte Ansatz ist eher theoretischer Natur. Zu limitierend sind die für die WF-Modellierung getroffenen Annahmen, die weder die Abbildung komplexerer Kontrollflußstrukturen (z.B. Schleifen, prädikative Verzweigungen) noch die Berücksichtigung von Datenflußaspekten erlauben.

c) Zusammenfassende Bewertung

Auch in ADEPT werden Zustandsmarkierungen bei der Durchführung von Sprungoperationen korrekt angepaßt. Für die Überprüfung der Zulässigkeit solcher Operationen dienen einfache Graphbetrachtungen sowie Informationen zur bisherigen Ausführung, d.h. es müssen keine komplexen Erreichbarkeitsanalysen durchgeführt werden. Durch die Verwendung einer Ausführungshistorie kann - im Gegensatz zu Chautauqua und MWMS - vermieden werden, daß die WF-Kontrolle in Bearbeitungszweige zurückgesetzt wird, die zuvor gar nicht durchlaufen wurden. Der Erhalt der Konsistenz von Zustandsmarkierungen ist aber nur einer von mehreren Aspekten, der sich im Zusammenhang mit der Unterstützung von Vorwärts- und Rückwärtssprüngen stellt. Weitere in ADEPT behandelte, in den genannten Ansätzen aber nicht aufgegriffene Fragestellungen, sind: Konzepte zur Vormodellierung planbarer Vorwärts-/Rückwärtssprünge, Zurücksetzen der WF-Kontrolle unter Wahrung von Integritätsaspekten (z.B. Wiederherstellen alter Werte von Datenelementen), Rücksprünge in Zustände frühe-

rer Iterationen einer Schleife, Erhalt der Datenkonsistenz bei der Durchführung von Vorwärtssprüngen, korrekte Behandlung von Sprüngen bei Auftreten bedingter Verzweigungen sowie Unterstützung von Sprüngen unterschiedlicher Semantik (z.B. Vorwärtssprünge mit / ohne Nachholen der übersprungenen Aktivitäten, Schleifen- vs. Fehlerrücksprünge, automatische vs. spontane Rücksprünge).

10.1.2.4 Adaption durch (dynamische) Änderung der Struktur von Netzen

a) Officetalk-D

Die Notwendigkeit dynamischer Netzänderungen wurde erstmals in Officetalk-D [ElBe82] erkannt. Das hier entwickelte Büroinformationssystem unterstützt die Definition und Steuerung einfacher Formularflüsse, wobei als Prozeßbeschreibungssprache ICN-Netze zum Einsatz kommen (für Details siehe [Elli79]). Zur Laufzeit werden die Prozeßinstanzen eines bestimmten Typs durch dasselbe Netz koordiniert, was aufgrund der Unterscheidbarkeit von Marken kein Problem darstellt. Soll ein sich in Ausführung befindliches Netz geändert werden, muß zuerst der zu modifizierende Bereich festgelegt werden. Der Netzinterpreter stellt sicher, daß keine neue Marken mehr in Stellen des Änderungsbereichs zufließen. Umgekehrt darf die Änderung erst dann erfolgen, wenn alle noch aktiven Marken aus dem betreffenden Bereich abgeflossen sind. Dieser Ansatz erweist sich in der Praxis als untauglich. Korrektheits- und Konsistenzaspekte bleiben weitgehend unberücksichtigt. Erschwerend kommt hinzu, daß zur Durchführung einer Änderung möglicherweise eine große Anzahl von Prozeßinstanzen unnötig lange blockiert wird. Ebenfalls nicht möglich sind strukturelle Änderungen einzelner Instanzen.

b) SPADE und reflektive Ansätze

SPADE [BFG93] ist eine Umgebung zur Unterstützung von Entwicklungsprozessen. Zur Prozeßmodellierung dient die Sprache SLANG, die auf *ERP-Netzen* basiert, einer Erweiterung von Pr/T-Netzen und gefärbten Petri-Netzen. ERP-Netze können hierarchisch strukturiert und ihre Sub-Netze zur Ausführungszeit dynamisch gebunden werden. SLANG ist eine *reflektive*[1] Sprache, mit der nicht nur „normale“ Prozesse definiert werden, sondern auch die sie manipulierenden *Meta-Prozesse*. Ein Beispiel für einen Meta-Prozeß ist der Ablauf, der die Schritte einer Prozeßmodifikation festlegt. Für jede Prozeßinstanz wird eine Datenstruktur verwaltet, die Informationen zur Netzstruktur und zu Ausführungsdaten enthält. Die Ausführung erfolgt durch Interpretation dieser Datenstruktur. Soll die Prozeßinstanz abgeändert werden, muß zuerst ihre Ausführung suspendiert werden. Anschließend wird ein Änderungsprozeß (Meta-Prozeß) gestartet, der ebenfalls durch ein ERP-Netz beschrieben ist. Dieses Netz erhält als Eingabe eine Marke, die die relevanten Daten der abzuändernden Prozeßinstanz trägt. Die eigentliche Netzänderung erfolgt mit Hilfe eines graphischen Editor, spezielle Änderungsoperationen gibt es nicht. Wie mit den Markierungen des Netzes bei seiner Änderung verfahren wird, geht aus den Veröffentlichungen zu SPADE nicht hervor. Vergleichbare Ansätze werden in [MKVC98], [BoMu98] und [EdHo98] behandelt. Das in [MKVC98] vorgestellte Baan-WfMS verwendet als Beschreibungssprache zeitorientierte, gefärbte Petri-Netze [Jens90]. Der in [BoMu98] beschriebene Ansatz ist allgemeiner und kann auf verschiedene Netzformalismen angewendet werden. Auch in [EdHo98] wird ein reflektiver Ansatz zur WF-Adaption verfolgt. Allerdings werden keine Petri-Netze verwendet, sondern ein ausdrucksstarker Formalismus (*Task Structures* [HOR98]), dessen formale Semantik durch eine Prozeßalgebra definiert ist.

c) Flußnetze (Flow Nets)

Ein mathematischer Formalismus zur strukturellen Änderung von Petri-Netzen wird in [EKR95, EKW98] beschrieben. Für die WF-Modellierung und -Steuerung werden Flußnetze (*Flow Nets*) verwendet, die eine ähnliche Audrucksmächtigkeit wie die in Abschnitt 10.1.2.3 beschriebenen BICN-Netze besitzen. Zur Laufzeit kann ein solches Netz mehrere WF-Instanzen kontrollieren, was durch die Unterscheidbarkeit von Marken möglich wird. Ein Beispiel zeigt Abbildung 10-9 a). Strukturelle Änderungen werden mit einem graphischen Editor vorgenommen [ElMa97], wobei es keine vordefi-

---

[1] Ein System heißt *reflektiv*, wenn es über ein Verhaltensmodell von sich selbst verfügt und auf einer Meta-Ebene darüber schließen kann. Vollständige Reflektivität liegt vor, wenn System- und Metaprozesse in derselben Weise beschrieben werden, etwa durch Verwendung derselben Modellierungssprache.

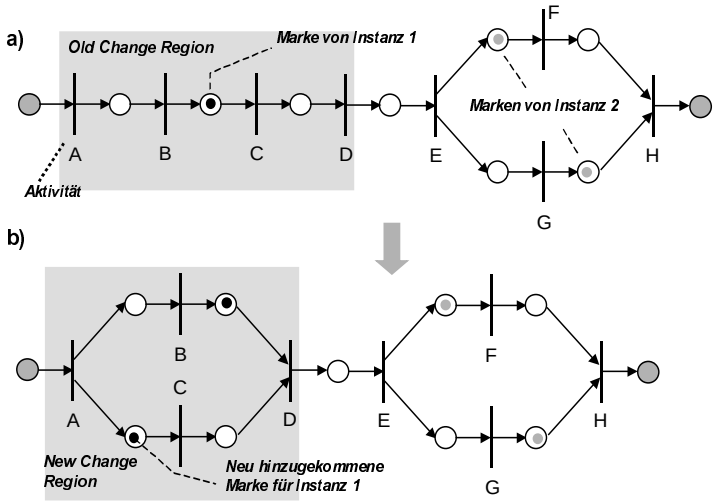


Abbildung 10-9: *Dynamische Änderung durch Austausch von Subnetzen und Anpassung von Markierungen.*

nierten Änderungsoperationen oder höhere Änderungsschnittstellen gibt. Alle Modifikationen erfolgen zunächst auf einer Kopie des Netzes. Währenddessen ist das Netz für andere Strukturänderungen gesperrt, wohingegen Modifikationen seines Zustandes bzw. seiner Markierung weiterhin möglich sind. Formal wird bei einer dynamischen Änderung ein gegebenes Netz in ein neues Netz transformiert. Letzteres besitzt gegenüber dem alten Netz nicht nur eine modifizierte Struktur, sondern ggf. auch eine angepaßte Markierung. Die Semantik einer solchen Transformation wird durch eine Graphsubstitution beschrieben, bei der ein markiertes Subnetz (*Old Change Region*) durch ein anderes markiertes Subnetz (*New Change Region*) ersetzt wird, wobei beide Subnetze in denselben Kontext des Gesamtnetzes eingebettet werden. Strukturänderungen können sich, je nach aktueller Lokation der Marken, auf mehrere Instanzen auswirken. Gezielte Abänderungen einzelner WF-Instanzen sind nicht möglich.

Interessant ist die Fragestellung wie mit den von einem Netz kontrollierten WF-Instanzen verfahren werden soll, wenn das Netz strukturell abgeändert wird. Unkritisch sind die Instanzen, deren Marken zum Zeitpunkt der Änderungseinbringung vollständig außerhalb der *Old Change Region* liegen. Für alle anderen Instanzen muß eine gesonderte Behandlung erfolgen, bei der die Marken aus der *Old Change Region* auf Stellen der *New Change Region* übertragen oder aus dem Netz entfernt werden (sog. *Fall Out Tokens*). Auch das Hinzufügen neuer Marken kann für einzelne Instanzen notwendig werden, um mit ihrer Kontrolle fortfahren zu können. Ein Beispiel zeigt Abbildung 10-9. Das in a) dargestellte Netz kontrolliert zwei WF-Instanzen, was durch die unterschiedliche Färbung der Marken zum Ausdruck gebracht wird. Das Netz soll nun so abgeändert werden, daß die bisher sequentiell auszuführenden Aktivitäten *B* und *C* in der Folge parallel bearbeitbar sind. Diese Änderung ist für die *Instanz 1* kritisch, da sie eine Marke in der *Old Change Region* besitzt. Abbildung 10-9 b) zeigt das aus dieser Änderung hervorgegangene Netz. Dabei wurde für die *Instanz 1* in der *New Change Region* eine neue Marke hinzugefügt, um eine Verklemmungssituation zu vermeiden. In [ElMa97] wird ein Netz-Editor beschrieben, der dem Modellierer beim Editieren eines Netzes anzeigt, welche WF-Instanzen von diesem Netz aktuell kontrolliert werden und in welchen Lokationen sich ihre Marken befinden. Der Modellierer kann für die WF-Instanzen, deren Marken in der *Old Change Region* liegen, manuelle Änderungen vornehmen und ihre Korrektheit mittels Erreichbarkeitsanalysen sicherstellen. Eine solche Vorgehensweise ist bei einer großen Zahl von Instanzen jedoch kaum praktikabel.

In [EKR95] wird eine spezielle Klasse von Netzänderungen (sog. *Synthetic Cutover Change*) vorgestellt, bei denen die *New Change Region* die *Old Change Region* vollständig als Teilnetz enthält. WF-Instanzen deren Marken sich zum Zeitpunkt der Änderung in der *Old Change Region* befanden, werden anschließend weiter so abgearbeitet, als hätte der Wechsel nicht stattgefunden. Für Instanzen, deren Marken neu in die *New Change Region* zufließen, können die vorgenommenen Modifikationen dagegen berücksichtigt werden. Ein Beispiel zeigt Abbildung 10-10. Hier soll das in a) dargestellte Netz so abgeändert werden, daß die bisher parallel ausführbaren Aktivitäten *F* und *G* in der Folge sequentiell bearbeitet werden. Das in b) abgebildete Netz resultiert, wenn diese Änderung als *Synthetic Cutover Change* durchgeführt wird. Hier werden alle Instanzen entweder gemäß der durch das alte

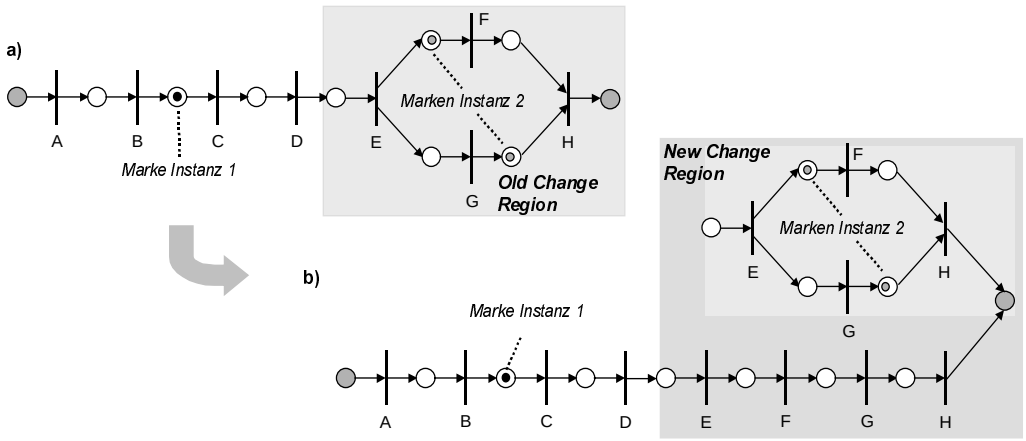


Abbildung 10-10: *Synthetic Cutover Change (Sequentialisieren von F und G)*

Netz definierten Struktur ausgeführt oder aber sie berücksichtigen die vorgenommene Änderung vollständig. Die Frage nach der korrekten Anpassung von Markierungen stellt sich bei dieser Vorgehensweise trivialerweise nicht. Nachteilig ist, daß nun zwei Versionen eines Schemas in einem Netz „verschmolzen“ werden. Hierunter leidet nicht nur die Lesbarkeit der Modelle, sondern es bleibt auch unklar, wie der Modellierer bei möglichen Folgeänderungen die verschiedenen Versionen voneinander trennen soll. Versionierungskonzepte, wie sie von MOKASSIN [JoHe98] oder TRAM [KrGe99] vorgeschlagen werden, sind hier unerläßlich, wenn man zu wartbaren Modellen gelangen will.

Die Autoren beschränken ihre Untersuchungen auf Kontrollflußänderungen und klammern Fragestellungen zur Anpassung von Datenflüssen aus. Ihre Berücksichtigung wäre durch die Verwendung getypter Marken zwar prinzipiell möglich, würde aber zu den in Abschnitt 10.1.1 diskutierten Problemen führen. Darüber hinaus könnten große Änderungsbereiche resultieren, die im ungünstigsten Fall das gesamte Netz betreffen. Ein großes Manko besteht darin, daß Ad-hoc-Änderungen *einzelner* Instanzen nicht unterstützt werden und alle Änderungen deshalb immer permanent erhalten bleiben.

d) WF-Netze

Ein erweiterter Petri-Netz-Formalismus, der speziell für die WF-Modellierung entwickelt wurde, sind WF-Netze [Aals97, Aals98a, HVA97]. Im Unterschied zu den oben diskutierten Ansätzen kann hier die strukturelle Modifikation von Netzen, ähnlich wie in $\text{ADEPT}_{\text{flex}}$, durch die Anwendung einer Menge von *Strukturtransformationsregeln* erfolgen [VoAa97]. Ein WF-Netz [Aalst98b] ist ein Pr/T-Netz mit zwei ausgezeichneten Stellen i und o, die seine Quelle und Senke festlegen. Bei der Erzeugung einer WF-Instanz wird eine Netzkopie mit einer Anfangsmarkierung I angelegt, unter der die Quelle mit einer Marke belegt ist. Die Instanz terminiert, wenn eine Endmarkierung O erreicht wird, unter der die Senke als einzige Stelle markiert ist. Durch Erreichbarkeitsanalysen wird sichergestellt, daß jede von I aus erreichbare Markierung M entweder gleich O oder aber daß O von M aus erreichbar ist. Des weiteren wird gefordert, daß eine Stelle unter jeder erreichbaren Markierung maximal eine Marke trägt (*sicheres* Netz). In [Aals98a] werden Bedingungen für die Bildung von WF-Netzen genannt, die diese Eigenschaften per Konstruktion sicherstellen. Strukturelle Modifikation können mit einem graphischen Editor auf der Grundlage von 8 Transformationsregeln festgelegt werden [VoAa97]. Beispiele für solche Regeln sind:

- *Verfeinerung*: Ersetzen einer Transition mit je einer Ein-/Ausgangsstelle durch ein Sub-Netz
- *AND-Split* (*OR-Split*): Aufsplittung einer Stelle (Transition) in zwei Stellen (Transitionen) - ein- und ausgehende Kanten werden dabei dupliziert
- *Synchronisation*: Sequenzialisierung (Parallelisierung) zweier bisher nebenläufiger (direkt aufeinanderfolgender) Transitionen durch Einfügen (Entfernen) einer Stelle zwischen ihnen
- *Reduktion*/*Erweiterung*: Ist k_1 der einzige Vorgänger von k_2 und k_2 der einzige Nachfolger von k_1, so können k_1 und k_2 sowie die Kante $k_1 \rightarrow k_2$ entfernt werden, wobei jeder Knoten des Vorbereichs von k_1 mit jedem Knoten des Nachbereichs von k_2 verknüpft wird (*Reduktion*). Die hierzu inverse Transformation wird als *Erweiterung* bezeichnet (siehe Abbildung 10-11).

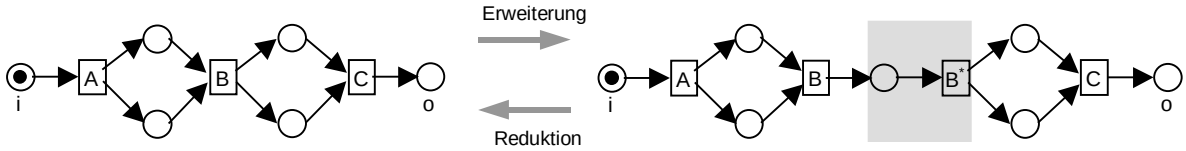


Abbildung 10-11: *Beispiele für WF-Netz-Transformationen*

In [Aals97] wird gezeigt, daß bei Anwendung der Transformationsregeln auf ein WF-Netz die gewünschten Dynamikeigenschaften erhalten bleiben. Eine Ausnahme bilden Synchronisationsregeln. Hier können Eigenschaften wie Sicherheit oder Lebendigkeit nur durch zusätzliche Erreichbarkeitsanalysen sichergestellt werden. Im Gegensatz zu $\text{ADEPT}_{\text{flex}}$ lassen sich nur bestimmte Änderungen realisieren, d.h. die angebotene Operatormenge ist unvollständig. Hinzu kommt, daß viele der für die Anwendung der Regeln getroffenen Einschränkungen [VoAa97] zu restriktiv sind. Ein weiteres Manko ist die Beschränkung auf Kontrollfluß- und die Ausklammerung von Datenflußaspekten. Unklar ist auch, ob und - falls ja - wie die Markierungen eines Netzes bei strukturellen Änderungen angepaßt werden. Auf die Problematik temporärer Änderungen wird ebenfalls nicht eingegangen.

e) Rekonfigurierbare Netze und generische WF-Modelle

Rekonfigurierbare Netze verfolgen einen anderen Ansatz zur Unterstützung dynamischer WF-Änderungen [BaOl98]. Ein solches Netz besteht in Wirklichkeit aus einer Menge vordefinierter Petri-Netze, die unterschiedliche *Konfigurationen* des modellierten Systems festlegen. Eine Einzelkonfiguration beschreibt eine mögliche Ausführungsvariante des abgebildeten Prozesses. Für einen WF mit Aktivitäten *A* und *B* etwa könnte eine Konfiguration festlegen, daß diese beiden Aktivitäten parallel bearbeitet werden können, während eine andere vorschreibt, daß sie sequentiell auszuführen sind. Zur Laufzeit wird für eine WF-Instanz zu jedem Zeitpunkt immer genau eine Konfiguration für die Steuerung zugrundegelegt. Eine dynamische „Änderung“ erfolgt nicht durch die Modifikation der aktuellen Konfiguration, sondern durch den Wechsel zu einer anderen, vordefinierten Konfiguration. Die Annahme dabei ist, daß alle denkbaren Ausführungsvarianten eines Prozesses bereits zur Modellierungszeit bekannt und durch geeignete Konfigurationen beschreibbar sind. Für das skizzierte Beispiel müßten zwei Netze modelliert werden, um die unterschiedlichen Ausführungsvarianten abzubilden. Sollen für eine Instanz die bisher sequentiell auszuführenden Aktivitäten *A* und *B* parallel bearbeitbar sein, muß dem entsprechend ein Wechsel zwischen den beiden Konfigurationen vollzogen werden. Das wesentliche Problem besteht darin, die Marken des alten Netzes auf Marken des neuen Netzes zu übertragen. Um dies automatisch bewerkstelligen zu können, muß der WF-Entwerfer für je zwei Konfigurationen eine entsprechende Abbildungsvorschrift festlegen, die eine Konversion in beide Richtungen erlaubt. Anspruchsvollere Änderungen sind nicht verwirklichbar, da beim Wechsel zwischen Konfigurationen lediglich Marken verschoben werden können, wohingegen die Hinzunahme oder das Entfernen von Marken nicht möglich ist. Die Annahme, daß stets alle Ausführungsvarianten eines Prozesses zur Modellierungszeit bekannt und durch separate Netze beschreibbar sind, trifft in der Praxis häufig nicht zu. Aber selbst wenn dem so wäre, würde die skizzierte Vorgehensweise allein aus kombinatorischen Gründen zu kaum mehr wartbaren Modellen führen. Bereits für einfache Abläufe müßten Dutzende bis Hunderte von Netzkonfigurationen erstellt und geeignete Abbildungsvorschriften zwischen ihnen definiert werden - bei n Konfiguration sind $n\,(n\text{-}1)\,/\,2$ Abbildungen festzulegen.

Einen ähnlichen Ansatz verfolgen *generische Prozeßmodelle* [Aals99]. Sie bestehen jeweils aus einer Menge von WF-Vorlagen, von denen jede eine mögliche Ausführungsvariante eines bestimmten Prozesses beschreibt. Einzelne Vorlagen werden durch ein Petri-Netz definiert und - zusammen mit den anderen Vorlagen des generischen Prozeßmodells - in einem Vererbungsdiagramm verwaltet. Im Unterschied zu rekonfigurierbaren Netzen lassen sich hierarchische WFs abbilden, indem ein Sub-WF beliebiger Hierarchiestufe wieder durch ein genetisches Modell repräsentiert wird. Die Auswahl einer konkreten Vorlage für einen Sub-WF erfolgt erst bei seiner Instanziierung, abhängig von aktuellen Wertebelegungen der WF-Variablen. Dynamische WF-Änderungen basieren, wie bei rekonfigurierbaren Netzen, auf der Idee, für WF-Instanzen ihre bisherige Vorlage durch eine andere Vorlage des generischen Modells zu ersetzen. Prinzipiell ist es auch möglich, eine neue Vorlage dynamisch zu einem Vererbungsdiagramm hinzuzufügen. Wie bei rekonfigurierbaren Netzen müssen geeignete Abbildungsvorschriften für den Wechsel zwischen zwei Vorlagen definiert werden. Um nicht für jedes Paar

von Vorlagen eine solche Abbildung definieren zu müssen, wird eine minimale Vorlage festgelegt. Sie besitzt exakt die Eigenschaften, die allen Vorlagen des Prozeßmodells gemeinsam ist (z.B. Grundmenge an Aktivitäten). Der dynamische Wechsel zwischen zwei Vorlagen erfolgt dann indirekt durch Abbildungsfunktionen zwischen ihnen und der minimalen Vorlage. Dieser Ansatz ist im allgemeinen nicht ohne Informationsverlust anwendbar und stellt daher keine Verbesserung gegenüber rekonfigurierbaren Netzen dar. Ähnliche Formalismen werden in [AsBu96, Valk81] diskutiert.

10.1.3 Zusammenfassende Bewertung

Die Hauptvorteile von Petri-Netzen sind ihre formale Grundlage und ihre sehr gut entwickelte Theorie. Dadurch sind die Voraussetzungen für die Prüfung wichtiger Dynamikeigenschaften gegeben. Darüber hinaus sind verschiedene andere formale Prüfungen möglich [Baum96]. Trotz dieser Vorteile sind Petri-Netze bisher den Nachweis schuldig geblieben, daß sie sich gut für die Modellierung und Ausführung von WFs eignen. Bei Anwendung auf den WF-Bereich treten Mängel vor allem dann zutage, wenn weitergehende Anforderungen berücksichtigt werden müssen, etwa in Bezug auf die Vormodellierung von Ausnahmen, die dynamische Abänderung von WFs (unter Erhalt ihrer Konsistenz), die Berücksichtigung operativer Sicherheitsanforderungen (z.B. WF-Terminierung) oder die Einbeziehung von Benutzeraspekten. In der Mehrzahl der Fälle wird implizit davon ausgegangen, daß der Endanwender - z.B. ein Sachbearbeiter - Petri-Netz-Formalismen leicht erlernen kann und in der Lage ist, ein Netz in einer Ausnahmesituation mit Hilfe eines graphischen Editors rasch und fehlerfrei abzuändern [ElMa97, VoAa97]. Tabelle 10-1 faßt die verschiedenen Ansätze nochmals zusammen.

	HOON	*MOVE*	*Chautauqua*	*MWMS*	*SPADE*	*WF-Netze*	*Rekonf. Netze*
Ausdrucksmächtigkeit	++	++	+	––	+	++	+
Adaption durch spätes Binden	j	j	n	n	j	n	n
Adaption durch spätes Modellieren	n	j	n	n	n	n	n
Adaption durch verhaltensorientierte Änderungen (Kontrollflußeingriffe)	n	n	j	j	n	n	j/n
Unterstützung von Ad-hoc-Sprüngen	n	n	j	j	n	n	n
Gewährleistung konsistenter Zustände	n	n	n	j	n	n	n
Zurücksetzen der Kontrolle im Fehlerfall	n	n	n	n	n	n	n
Adaption durch strukturelle Änderungen	n	j/n	j	j/n	j	j	j/n
Änderungsschnittstelle	n	Editor	Editor	n	Editor	Editor	-
Bereitstellung von Änderungsoperationen	-	n	n	n	n	j	n
Vollständigkeit	-	-	-	-	-	n	-
Formale Semantik	-	-	-	-	-	j	-
Formale Vor-/Nachbedingungen	-	-	-	-	-	j/n	-
Eingrenzung notwendiger Analysen für Korrektheitsprüfungen	-	-	-	-	-	n	-
Begleitende Anpassung von Datenflüssen	-	j/n	n	n	n	n	j/n
Dynamische Änderungen wirken sich aus	-	j/n	j	j	j	?	j
... auf einzelne WF-Instanz	-	j	n	j	j	?	j
... auf alle kontrollierten Instanzen	-	n	j	n	n	?	n
... auf zukünftige Instanzen	-	n	j	n	n	?	n
Änderungszeitpunkt: Offline / On-the-fly	-	Off	On	On	Off	Off	Off
Anpassung von Netzmarkierungen	-	-	manuell	autom.	manuell	?	automatisch
Unterstützung temporärer Änderungen	-	-	n	n	n	n	n

Tabelle 10-1: *Petri-Netz-basierte Formalismen für adaptives Workflow-Management*

10.2 Vergleich mit Statechart-basierten Ansätzen

Als Alternative zu Petri-Netz-basierten Prozeßbeschreibungssprachen werden in der WF-Literatur erweiterte (hierarchische) Zustandsautomaten (*Statecharts*) vorgeschlagen [Weik97, Teeg96].

10.2.1 State- und Activitycharts

State- und Activitycharts sind ein Formalismus, der ursprünglich für den Entwurf und die Analyse reaktiver Systeme entwickelt wurde [Hare87]. Ein *Activitychart* definiert die Funktionen (*Activities*) des Systems und legt die Informationsflüsse zwischen ihnen fest. Seine graphische Darstellung erfolgt durch ein hierarchisches Datenflußdiagramm, dessen Knoten die Aktivitäten und dessen Kanten die Datenflußbeziehungen repräsentieren. Die auszutauschenden Datenelemente werden durch Kantenbeschriftungen festgelegt. Ein einfaches Beispiel zeigt Abbildung 10-12 a). Aus diesem Activitychart geht hervor, welche Funktionen das System umfaßt (*A*, *B*, und *C*) und welche Datenelemente zwischen ihnen ausgetauscht werden ($DATA_1$, $DATA_2$), nicht jedoch wann dies geschieht und wie häufig. Zu diesem Zweck muß ein separater Statechart spezifiziert werden.

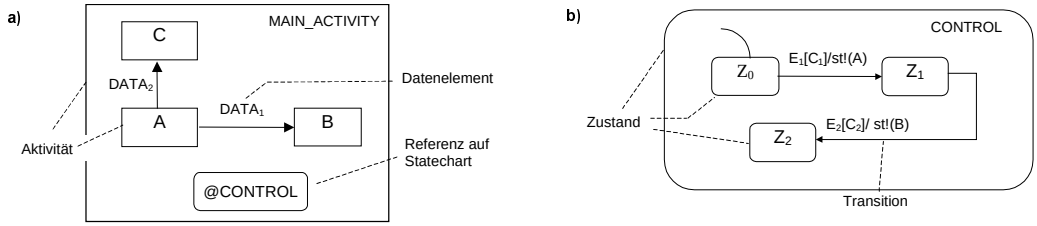


Abbildung 10-12: *(a) Activitychart und (b) zugehöriger Statechart*

Ein *Statechart* ist ein erweiterter Zustandsautomat, bei dem Aktionen zur Ausführung kommen, wenn ein Übergang zwischen zwei Zuständen stattfindet oder wenn ein neuer Zustand betreten wird. Statecharts erweitern herkömmliche Zustandsübergangsdiagramme [Part98] in mehrerlei Hinsicht. Die Bedingungen für Zustandsübergänge und die dabei auszuführenden Aktionen können mittels *ECA-Regeln* formuliert werden. Vereinfacht ausgedrückt findet ein Übergang statt, wenn das spezifizierte Ereignis *E* (z.B. Betreten eines Zustandes) eintritt und dabei die Bedingung *C* (z.B. Belegung einer Variablen mit bestimmtem Wert) erfüllt ist. Für diesen Fall wird die Aktion *A* (z.B. Starten einer Aktivität) ausgeführt und der Wechsel in den neuen Zustand vollzogen. Die Angabe der einzelnen Regelkomponenten ist optional, wodurch Freiheitsgrade resultieren. Ein Beispiel zeigt Abbildung 10-12 b). Weitere Erweiterungen gegenüber herkömmlichen Automaten betreffen die hierarchische Strukturierung von Automaten und die Abbildung nebenläufiger Zustände. Im letztgenannten Fall können die zugehörigen Statecharts über Ereignisse miteinander kommunizieren. Gegenüber Petri-Netzen ermöglichen State- und Activitycharts eine bessere Trennung von Kontroll- und Datenfluß. Ähnlich wie bei Petri-Netzen läßt sich die Struktur und das dynamische Verhalten von Statecharts formal analysieren.

State- und Activitycharts wurden erstmals in MENTOR als Formalismus für die Spezifikation und Verifikation von Workflows verwendet [Wodt95, WWWK96]. Einen Schwerpunkt dieses Projektes bildete die Untersuchung von Konzepten zur Partitionierung und verteilten Ausführung von WFs [Muth98]. Es wird formal bewiesen, daß die verteilte Ausführung eines Statecharts äquivalent zur zentralen Ausführung ist, wenn seine Partitionierung nach gewissen Regeln erfolgt [WoWe97]. Dynamische WF-Änderungen wurden nicht untersucht. Aspekte der verteilten Ausführung von WFs sind auch im ADEPT-Projekt behandelt worden [BaDa97, BaDa98]. Insbesondere wurden Konzepte, die dynamische Änderungen bei verteilter Steuerung unterstützen sollen, entwickelt [RBD99].

10.2.2 Dynamische Änderungen von Statecharts in HieraStates

Im HieraStates-Modell [Teeg96, Teeg98] wird ein WF-Schema durch einen erweiterten Zustandsautomaten beschrieben, der Ähnlichkeiten zu Statecharts aufweist. Im Unterschied zu herkömmlichen Statecharts bietet HieraStates auch Konzepte für dynamische Änderungen an.

WF-Zustände werden durch Knoten und WF-Aktivitäten durch beschriftete Kanten (Transitionen) modelliert. Jedes HieraStates-Diagramm besitzt einen eindeutigen Anfangszustand, der mit einem ein-

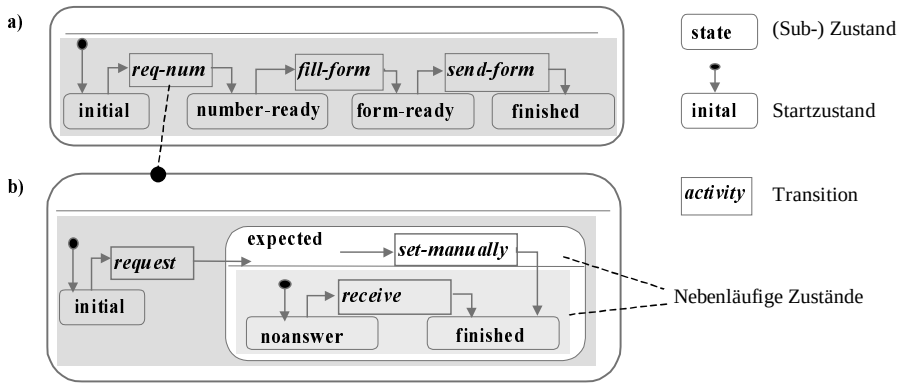


Abbildung 10-13: *WF-Modellierung mit HieraStates-Automaten*

mündenden Pfeil markiert ist, sowie einen oder mehrere Endzustände. In jedem Nichtendzustand ist mindestens eine Transition aktiviert. Ihre Bearbeitung erfolgt durch den Aufruf eines Anwendungsprogramms oder durch die Ausführung eines Sub-WF, was einen Wechsel zwischen zwei Zuständen bewirkt. Besitzt ein Zustand mehrere ausgehende Transitionen, kann der Benutzer zur Laufzeit eine davon auswählen. Parallele Aktionen werden über nebenläufige Zustände abgebildet. Darüber hinaus besteht die Möglichkeit zur hierarchischen Strukturierung von Zuständen. Inwieweit sich prädikative Verzweigungen und Datenflüsse abbilden lassen, geht aus den Veröffentlichungen zu HieraStates nicht hervor. Ebensowenig wird dargestellt, ob weitere Beschreibungselemente des Statechart-Formalismus (z.B. Verknüpfung von Transitionen mit ECA-Regeln, Synchronisation nebenläufiger Zustände durch Ereignisse, usw.) übernommen worden sind. Ein Beispiel zeigt Abbildung 10-13. Das obere Zustandsdiagramm beschreibt den sequentiellen Ablauf einer Warenbestellung, der aus den drei Aktivitäten „Registriernummer ermitteln (*req-num*)“, „Bestellformular ausfüllen“ (*fill-form*) und „Bestellformular versenden“ (*send-form*) besteht. Die Transition *req-num* wird durch einen Sub-Workflow verfeinert, dessen Zustandsdiagramm ebenfalls abgebildet ist. Dieser Automat besitzt die Zustände *initial* und *expected*, wobei der Zustand *expected* durch einen Sub-Automaten weiter verfeinert wird. Ausgehend vom Zustand *initial* kann bei einer Vergabestelle eine neue Registriernummer angefordert werden (*request*), was einen Wechsel in den Unterzustand *noanswer* des Zustands *expected* bewirkt. Sobald eine Nummer zugeordnet worden ist – die Bestimmung der Registriernummer durch die Vergabestelle wurde hier nicht modelliert – kann sie mit der Transition *receive* entgegengenommen werden.

Planbare Abweichungen vom „normalen“ Ablauf können mit Hilfe spezieller Transitionen, die in verschiedenen Zuständen des Automaten aktivierbar sind, vormodelliert werden. Im vorliegenden Beispiel etwa kann die Situation eintreten, daß die Vergabestelle auf eine Anfrage nicht antwortet und sich der Bearbeiter die Registriernummer anderweitig beschafft. Um mit der Kontrolle fortfahren zu können, muß diese Nummer dem System auf anderem Wege bekannt gemacht und in dem dafür vorgesehenen WF-Attribut gespeichert werden. Im dargestellten Beispiel ist dies mittels der Transition *set-manually* möglich. Sie wird beim Eintritt in den Zustand *expected* aktiviert und kann in jedem Unterzustand des abgebildeten Sub-Automaten ausgeführt werden. Im allgemeinen kann von einem beliebigen Zustand des Automaten in beliebige andere Zustände gewechselt werden. Solche planbaren Abweichungen lassen sich auch in ADEPT auf einer semantisch höheren Ebene beschreiben. Das partielle Zurücksetzen eines WF wird vom HieraSates-Modell nicht unterstützt.

Benutzer können die Struktur eines sich in Ausführung befindlichen Statecharts mit Hilfe eines graphischen Editors abändern [Teeg98]. Eine Änderung bezieht sich immer auf exakt eine WF-Ausprägung, da für jede Instanz eine Kopie des WF-Schemas verwaltet wird. Unterstützt werden lediglich additive Änderungen, wie die Hinzunahme neuer Zustände und Transitionen oder die Verfeinerung existierender Zustände (vgl. Abbildung 10-14). Nicht möglich ist dagegen das Entfernen von Transitionen und Zuständen, da nach Meinung der Autoren „jede Art des Löschens von Plankomponenten problematisch ist, da die Komponente gerade von der Instanz benutzt werden kann“ [Teeg98]. Dies ist zweifellos richtig, rechtfertigt aber nicht, auf eine derart wichtige Kernfunktion eines adaptiven WfMS zu verzichten. Die rein additive Änderungssemantik erschwert nicht nur die Umsetzung einfacher Operationen, sondern läßt auch zahlreiche wichtige Änderungen nicht zu. Beispielsweise kann in den in Abbildung 10-14 a) dargestellten Automaten keine Aktivität eingefügt werden, die nach *A* und vor

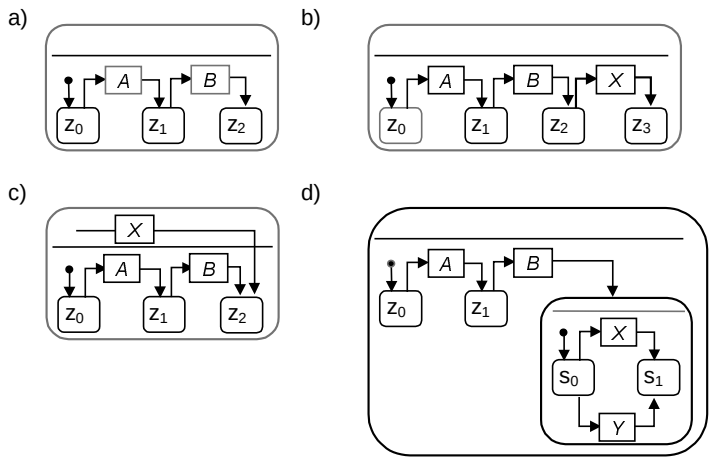


Abbildung 10-14: *Strukturelle Modifikationen von HieraSates-Diagrammen*

Die Diagramme b) - d) ergeben sich aus dem Diagramm a) wie folgt:

(b) Hinzunahme des (End-)Zustands Z_3 und Einfügen der Transition X zwischen Z_2 und Z_3

(c) Hinzunahme der speziellen Transition X (die in jedem Zustand des in a) dargestellten Automaten anwendbar sein soll)

(d) Verfeinerung des Zustandes Z_2 durch einen Subautomaten mit alternativen Transitionen X und Y

B ausgeführt wird. Weitere Schwächen des HieraStates-Formalismus betreffen die Nichtberücksichtung des Datenflusses und die fehlende Behandlung von Korrektheitsaspekten. Für die Anwendung der erwähnten Änderungsprimitve bestehen keine Einschränkungen, was auf Instanzebene zu Problemen (z.B. Verklemmungen) führen kann. Obwohl sich Schleifen abbilden lassen, werden essentielle Fragestellungen, etwa in Bezug auf die Unterstützung temporärer WF-Änderungen, nicht behandelt.

In HieraStates können WF-Modifikationen vordefiniert werden. Im Mittelpunkt steht das Konzept des *Features* [Teeg98]. Ein Feature bezieht sich immer auf ein bestimmtes WF-Schema und beschreibt eine Menge von Zuständen und Transitionen, die während der Ausführung entsprechender Instanzen hinzugefügt werden können. Genau genommen beschreibt ein Feature die Menge von Basisoperationen, deren Anwendung die gewünschten Erweiterungen herbeiführt. Features können aufeinander aufbauen, was durch Vererbungshierarchien zum Ausdruck gebracht wird. Um ein bestimmtes Feature auf einen gegebenen Plan anwenden zu können, müssen deshalb ggf. zuvor andere Features eingebracht werden. Würde HieraStates nichtadditive Änderungen (z.B. Löschen von Zuständen) erlauben, wäre dieses Konzept nicht mehr anwendbar.

10.3 Vergleich mit graph- und netzbasierten Ansätzen

In diesem Abschnitt behandeln wir weitere graphische Beschreibungsformalismen. Wir konzentrieren uns dabei auf die angeboten Möglichkeiten zur dynamischen Adaption von Workflows.

10.3.1 Adaptives Workflow-Management in PoliFlow

Eines der ersten Projekte, in dem Aspekte adaptiver WfMS näher beleuchtet wurden, war PoliFlow [Sieb96, Sieb97, Sieb98]. Im Unterschied zu ADEPT steht hier weniger die theoretische Untersuchung dynamischer WF-Änderungen und ihre technische Realisierung im Mittelpunkt, sondern vielmehr die Konzeption eines Gesamtmodells für adaptive WfMS. Der entwickelte Ansatz zielt einerseits auf die Vermeidung bzw. Automatisierung manueller Anpassungen, was durch die Verwendung einer flexiblen WF-Engine und eines mächtigen WF-Metamodells erreicht werden soll, andererseits sollen berechtigte Akteure auch während der Ausführung spontane Anpassungen vornehmen können. Solche Ad-hoc-Änderungen lassen sich durch verschiedene Kontrollmechanismen einschränken. Neben einfachen Konsistenzprüfungen zum Erhalt der syntaktischen Korrektheit von WF-Modellen, werden Anpassungsrechte (Wer darf welche Änderungen durchführen?) und benutzerdefinierte Integritätsbedingungen kontrolliert, um die semantische Korrektheit der WFs zu sichern.

Das in [Schr97] skizzierte WF-Metamodell verfolgt - ähnlich wie das ADEPT-Basismodell - einen blockbasierten Beschreibungsansatz. Aktivitäten können durch die geschachtelte Anwendung von Sequenzen, parallelen und bedingten Verzweigungen verknüpft werden. Dabei bestehen syntaktische Einschränkungen, etwa um sicherzustellen, daß eine Verzweigung mindestens zwei Teilzweige besitzt. Insgesamt erreicht dieses Metamodell nicht die Ausdrucksmächtigkeit des ADEPT-Modells, da wichtige Beschreibungselemente, wie Schleifen, Fehlerkanten, Sync-Kanten und AND-/XOR-Ver-

zweigungen, fehlen [Schr97]. Ob und – falls ja – wie sich Datenflüsse in PoliFlow abbilden lassen, geht aus den Veröffentlichungen nicht hervor.

Basierend auf dem skizzierten Modell werden Dienste angeboten, die sowohl die Anpassung struktureller als auch verhaltensbezogener WF-Eigenschaften erlauben sollen. Die Kontrollflußstruktur einer WF-Instanz kann mit einem graphischen Editor modifiziert werden. Er umfaßt Operationen für das Einfügen und Löschen von Aktivitäten. Dabei wird zwischen sequentiellem, parallelem und alternativem Einfügen unterschieden, je nachdem in welchen Kontext der neue Arbeitsschritt eingebettet wird. Komplexere Einfügeoperationen, wie die Hinzunahme einer Aktivität zwischen zwei Knotenmengen, sind nicht verwirklichbar. In [Schr97] wird auch nicht darauf eingegangen, wie ein eingefügter Schritt zu seinen Aufrufdaten kommt und wie seine Ausgabedaten in den WF-Datenkontext übernommen werden. Ähnliche Feststellungen gelten auch in Bezug auf die angebotene Löschoperation. Insgesamt ist die Realisierung von Strukturänderungsoperationen in PoliFlow, aufgrund des ausdrucksschwachen WF-Metamodells und der Ausklammerung von Datenflußaspekten, relativ einfach. Durch das Fehlen von Schleifen- / Fehlerrücksprungkanten fallen viele der in ADEPT behandelten Fragestellungen weg.

PoliFlow bietet zusätzlich Operationen an, mit denen sich das Verhalten einer WF-Instanz ändern läßt, ohne ihren Ausführungsgraphen strukturell anpassen zu müssen. So kann die Bearbeitung von Aktivitäten wiederholt, ausgelassen oder vorgezogen werden. Soll etwa eine noch nicht aktivierte Aktivität X spontan bearbeitet werden, wird sie nebenläufig zur regulären Ausführung gestartet und ein entsprechender Vermerk in einem Logfile vorgenommen. Trifft der normale Kontrollfluß auf die vorzeitig bearbeitete Aktivität X und ist X noch nicht beendet, wird das Fortschreiten des WF verzögert. Andernfalls wird die Ausführung mit den direkten Nachfolgern von X fortgesetzt. Auch in $\text{ADEPT}_{\text{flex}}$ sind solche Vorwärtssprünge möglich, im Unterschied zu PoliFlow werden sie allerdings durch die dynamische Restrukturierung des Ausführungsgraphen verwirklicht. Dadurch kann erreicht werden, daß der Kern der WF-Engine nicht erweitert werden muß, um diese spezielle Operation zu unterstützen. Darüber hinaus bleiben mögliche Folgeänderungen für den Anwender noch beherrschbar. Diese Abgeschlossenheit bietet PoliFlow nicht, was auch zu Schwierigkeiten in Verbindung mit anderen Systemfunktionen, etwa dem partiellen Zurücksetzen der WF-Kontrolle, führen würde.

10.3.2 Dynamische Änderung von Aktivitätennetzen in WASA

WASA verwendet ein ähnliches WF-Metamodell wie MQSeries Workflow (vgl. Kapitel 2), erweitert dieses aber um Konzepte zur Verwirklichung dynamischer WF-Änderungen [Wesk98, WHKS98]. Ein WF wird durch einen gerichteten, azyklischen Graphen (V, E) repräsentiert, dessen Knotenmenge V die Aktivitäten und dessen Kantenmenge E ihre Kontrollabhängigkeiten festlegen. Zur Modellierung alternativer Verzweigungen können die von einem Knoten ausgehenden Kontrollkanten zusätzlich mit prädikativen *Transitionsbedingungen* verknüpft werden, die bei erfolgreicher Beendigung der Aktivität, auf der Grundlage der von ihr produzierten Ausgaben, bewertet werden. Kontrollkanten deren Transitionsbedingung erfüllt ist werden mit *TRUE* signalisiert, andernfalls erhalten sie den Status *FALSE* zugewiesen. Bei der Erzeugung einer WF-Instanz werden genau die Aktivitäten aktiviert, die keine einmündenden Kontrollkanten besitzen. Im Verlauf der Ausführung wird eine neue Aktivitäteninstanz immer dann erzeugt, wenn alle einmündenden Kontrollkanten signalisiert worden sind und mindestens eine von ihnen den Status *TRUE* besitzt. Weisen dagegen alle einmündenden Kanten den Status *FALSE* auf, so ist die Aktivität nicht mehr ausführbar. Für diesen Fall werden ihre ausgehenden Kontrollkanten ebenfalls mit *FALSE* signalisiert, was zur Bewertung nachfolgender Knoten führt. Der WF terminiert, wenn es keine bearbeitbaren Aktivitäten mehr gibt. Schleifen sind nicht abbildbar, da das Auftreten von Zyklen zu Verklemmungen führen würde. Datenflüsse sind nur mit Einschränkungen modellierbar. So darf ein Eingabeparameter einer Aktivität $y \in V$ nur dann mit Ausgabeparametern einer Vorgängeraktivität $x \in V$ verknüpft werden, wenn y ein *direkter* Nachfolger von x ist. Dadurch sind der Kontroll- und Datenfluß nicht sauber voneinander trennbar, was sich negativ auf die Lesbarkeit der Modelle auswirkt. Des weiteren wird nicht garantiert, daß die Eingabeparameter einer Aktivität bei ihrer Aktivierung versorgt sind.

WASA unterscheidet zwischen Ad-hoc-Eingriffen in die WF-Kontrolle und strukturellen WF-Änderungen.

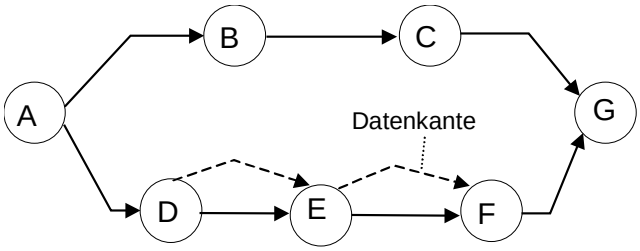


Abbildung 10-15: *WASA-Workflow-Graph*

a) Ad-hoc-Eingriffe in die Kontrollflußsteuerung (ohne strukturelle Modifikationen)

Ad-hoc-Eingriffe in die WF-Kontrolle erfolgen mittels einfacher Operationen. Sie beeinflussen das Verhalten einer WF-Instanz, lassen ihre Ablaufstruktur aber unverändert [WHKS98, Wesk97].

Mit der Operation *StopInstance* kann eine Aktivitäteninstanz abgebrochen werden. Ihr Knoten wird als nicht ausgeführt markiert und seine ausgehenden Kanten werden mit *FALSE* signalisiert. Demzufolge werden die Zielknoten dieser Kanten neu bewertet und evtl. ebenfalls übersprungen, usw. (sog. *Dead Path Elimination*). Angenommen im Ausführungsgraphen aus Abbildung 10-15 sind die Aktivitäten *B* und *D* aktiv. Dann bewirkt *StopInstance(D)*, daß *D* abgebrochen und die Bearbeitung von *D*, *E* und *F* übersprungen wird. Der Knoten *G* kann ausgeführt werden, sobald *B* und *C* beendet sind. Die Bearbeitung einer noch nicht gestarteten Aktivität *X* kann mittels *SkipInstance* übersprungen werden. Im Unterschied zu *StopInstance* werden hier alle ausgehenden Kontrollkanten von *X* mit *TRUE* signalisiert. Bezogen auf das obige Beispiel bewirkt die Ausführung von *SkipInstance(D)*, daß *D* übersprungen und die Kante $D \rightarrow E$ mit *TRUE* signalisiert wird, was zur Aktivierung von *E* führt. In vielen Fällen ist die gewählte Semantik, bei der immer *alle* ausgehenden Kontrollkanten auf *TRUE* gesetzt werden, aber unangemessen. Angenommen von einem Knoten *X* gehen zwei Kontrollkanten aus, deren Transitionsbedingungen so festgelegt worden sind, daß immer genau einer der beiden Nachfolgerknoten von *X* zur Ausführung kommt. Bei Anwendung von *SkipInstance(X)* würden beide von *X* ausgehenden Kanten mit *TRUE* bewertet werden, was zur Ausführung von Aktivitäten führt, die alternativen Ausführungspfaden angehören. Auch die Behandlung fehlender Daten ist nicht zufriedenstellend gelöst. Durch die sehr restriktiven Vorgaben für die Datenflußmodellierung können beim Auslassen einer Aktivität ohnehin nur ihren direkten Nachfolgerknoten Eingaben fehlen. Ihre Nachversorgung muß manuell durch den Anwender oder auf der Grundlage von Default-Werten erfolgen.

Mit der Operation *StartInstance* kann eine noch nicht aktivierte Aktivität vorzeitig bearbeitet werden. Angenommen in obigem Beispiel soll nach der Beendigung von *A* die Ausführung von *F* vorgezogen werden. *StartInstance*(*F*) bewirkt, daß *F* aktiviert und die Ausführungen von *B*, *C*, *D* und *E* ausgelassen werden. Warum die beiden Aktivitäten des oberen Zweiges in diesem Fall nicht mehr bearbeitbar sein sollen, wird nicht begründet. ADEPT bietet flexiblere Möglichkeiten zur Verwirklichung von Vorwärtssprüngen an, etwa für das Vorziehen einer Aktivität (mit/ohne Nachholen der überprungenen Schritte) und die Vormodellierung planbarer Vorwärtssprünge. Darüber hinaus gibt es in ADEPT klar definierte Kriterien für die Zulässigkeit von Sprungzielen (auch in Verbindung mit Verzweigungen und Schleifen) sowie Mechanismen für die Anpassung von Zustandsmarkierungen und Datenflüssen.

Mittels *RepeatInstance* kann die Kontrolle einer WF-Instanz vor die Bearbeitung einer bereits abgeschlossenen Aktivität zurückgesetzt werden. In obigem Beispiel könnte nach der Beendigung von *F* die Bearbeitung des unteren Zweiges vor *D* zurückgesetzt werden, ohne daß davon die Ausführung des oberen Zweiges beeinträchtigt wird. Für die betroffenen Aktivitäten *D*, *E* und *F* werden Kompensationsprogramme ausgeführt. Allerdings bleiben die von den zurückgesetzten Aktivitäten produzierten Ausgaben im Datenkontext der WF-Instanz erhalten und können als Eingabedaten für wiederholt auszuführende Aktivitäten dienen. Ob mit solchen „*dirty reads*“ eine saubere Ausführungssemantik erzielbar ist, muß bezweifelt werden. Abgesehen davon ist unklar, wie eine wiederholt auszuführende Aktivität ihre Eingabedaten von im WF-Graphen nachfolgenden Aktivitäten erhalten soll. Unverständlich ist, warum Schleifen nicht modellierbar sind, spontane Rücksprünge aber unterstützt werden. Rücksprünge sind in WASA nur zulässig, wenn bestimmte Regeln erfüllt sind. Eine davon legt fest, daß bei nebenläufig ausgeführten Teilzweigen eine Rücksetzoperation nur innerhalb einzelner Teilzweige erfolgen kann, wohingegen das Zurücksetzen der WF-Bearbeitung vor den gemeinsamen AND-

Split-Knoten nicht möglich ist. Des weiteren darf eine Rücksetzoperation nur dann erfolgen, wenn alle im Zuge früherer Rücksetzoperation wiederholt zur Ausführung kommenden Aktivitäten abgeschlossen sind. Diese Einschränkungen sind zu restriktiv, da sie viele praktisch relevante Fälle ausklammern. Des weiteren kann in Zweige zurückgesprungen werden, die zuvor gar nicht durchlaufen wurden.

b) Dynamische Änderungen der Ablaufstruktur von WF-Instanzen

Für die Verwirklichung struktureller Änderungen werden vier Operationen angeboten, auf deren Grundlage sich Aktivitäten und Kontrollkanten zu einem WF-Graphen hinzufügen bzw. aus ihm entfernen lassen. Diese Operatormenge ist unvollständig. Beispielsweise lassen sich damit weder Anpassungen des Datenflusses noch der mit Kontrollkanten verknüpften Transitionsbedingungen realisieren. Unklar ist deshalb, wie in WASA die Eingabeparameter einer neu eingefügten Aktivität versorgt werden. Darüber hinaus werden keine formalen Voraussetzungen genannt, die zur Prüfung der Zulässigkeit von Änderungsoperationen dienen. Die Durchführung einer Änderung erfolgt mit Hilfe eines Editors, wobei es für Anwender keine methodische Unterstützung gibt, etwa durch syntaxgesteuerte Änderungen oder änderungsspezifische (Datenfluß-)Analysen. Änderungen sind für die Restdauer einer WF-Instanz wirksam, d.h. sie bleiben auch nach Rücksprüngen im WF-Graphen erhalten [RVW97]. Dies kann im Widerspruch zum temporären Charakter von Ausnahmebehandlungen stehen.

In [VWW96] geben die Autoren an, daß die Modifikation eines Schemas immer alle seine Instanzen betrifft, wohingegen Ad-hoc-Änderungen einzelner Instanzen nicht möglich sind. Diese Einschränkung wird in [Wesk99] augehoben. Hier ist eine Schemaänderung nur dann auf eine laufende WF-Instanz anwendbar, wenn ihre bisherige Ausführungshistorie auf der Grundlage des neuen Schemas erzeugbar gewesen wäre. Dieses Kriterium ist identisch mit dem in WIDE eingeführten *Compliance*-Kriterium (siehe unten). Insgesamt sind die angebotenen Konzepte zur Verwirklichung dynamischer Änderungen unvollständig. Es werden weder technische Details noch formale Grundlagen präsentiert. Statt dessen erfolgt die Auseinandersetzung mit dynamischen Änderungen auf exemplarischer Ebene. Hinzu kommt, daß aufgrund des einfachen Metamodells viele der in dieser Arbeit diskutierten Problemstellungen (z.B. Korrektheit von Datenflüssen, temporäre Änderungen) nicht behandelt werden.

10.3.3 Dynamische Evolution von WF-Schemata in WIDE und TRAM

Ein weitergehender Ansatz wird von WIDE [Casa96] verfolgt. Großen Raum nimmt hier die Untersuchung von Fragestellungen zur Evolution von WF-Schemata ein, wohingegen Ad-hoc-Änderungen einzelner WF-Instanzen nicht betrachtet werden. Konkret wird zwischen Änderungen von WF-Typen (*static WF evolution*) und den dadurch evtl. notwendig werdenden Anpassungen von WF-Instanzen (*dynamic WF evolution*) unterschieden.

Die Grundlage für alles sind ein graphbasiertes WF-Metamodell [CCPP95] und eine Menge von Änderungsoperationen [CCPP98]. WIDE differenziert zwischen verschiedenen Arten von Aufsplittungs- und Synchronisationsknoten, mit denen sich sowohl bedingte und parallele Verzweigungen als auch Iterationen abbilden lassen (vgl. Abbildung 10-16). Ein Aufsplittungsknoten (*Fork*) besitzt genau einen Vorgänger und mehrere Nachfolger. Nach seiner Beendigung werden entweder alle (*total fork*) oder nur eine Teilmenge (*conditional fork*) der Nachfolgerknoten aktiviert. Umgekehrt besitzt ein Synchronisationsknoten (*Join*) mehrere Vorgänger und genau einen Nachfolger, der aktiviert wird, nachdem entweder alle (*total join*) oder mindestens einer dieser Vorgängerknoten (*iterative join*) beendet worden sind. Jeder WF-Graph besitzt genau einen Start- und mindestens einen Stopknoten. Jeder normale Knoten muß vom Startknoten aus über eine Pfeilfolge erreichbar sein. Einschränkungen in Bezug auf die Strukturierung der WF-Modelle oder ihr dynamisches Verhalten gibt es nicht. Ein

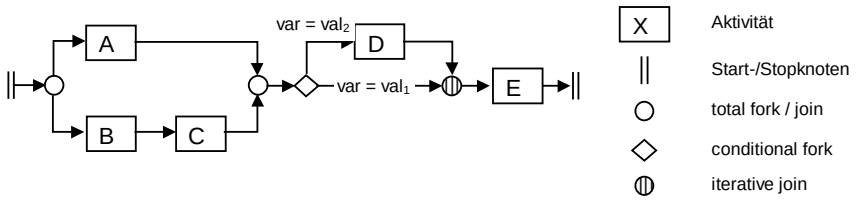


Abbildung 10-16: *Workflow-Modellierung in WIDE*

weiteres Manko ist, daß Datenflüsse nicht explizit modellierbar sind, sondern sich implizit aus den Lese- und Schreibzugriffen von Aktivitäten auf WF-Variablen ergeben.

Für den Modellierer wird eine vollständige und minimale Menge von Änderungsoperationen angeboten, mit der er die Kontrollflußstruktur eines WF-Graphen, unter Erhalt seiner syntaktischen Korrektheit, modifizieren kann. Andere WF-Aspekte werden weitgehend ausgeklammert. Beispiele für Änderungsoperationen sind das Einfügen von Aktivitäten, die Hinzunahme von Fork-/Join-Konstrukten (vgl. Abbildung 10-17 a) oder die Modifikation von Verzweigungsprädikaten. Für jede Operation werden ihre Vorbedingungen und ihre Semantik formal definiert [CCPP98]. Positiv ist die Vollständigkeit der angebotenen Operatormenge (bzgl. Änderungen der Kontrollflußstruktur). Die Forderung nach Minimalität führt dagegen zu teilweise schwer nachvollziehbaren Operationen. Beispielsweise wird das Löschen einer Aktivität nicht, wie man es erwarten würde, durch das Entfernen dieses Knotens und das Umhängen assoziierter Kontrollkanten verwirklicht, sondern durch die kombinierte Anwendung mehrerer Basisoperationen. Ein Knoten *X* wird „gelöscht", indem eine bedingte Verzweigung in den WF-Graphen eingebettet wird, von der ein Teilzweig leer ist und der andere aus dem Knoten *X* besteht. Durch geeignete Festlegung der Verzweigungsprädikate wird sichergestellt, daß der von *X* gebildete Teilzweig nicht zur Ausführung kommt (vgl. Abbildung 10-17 b).

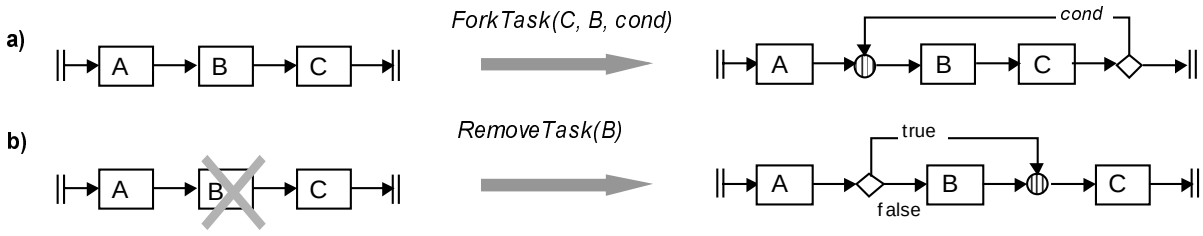


Abbildung 10-17: *Beispiel einer Basisänderungsoperation (a) und einer komplexen Operation (b) in WIDE*

An einem WF-Schema vorgenommene Änderungen können in WIDE ggf. auch auf bereits laufende Instanzen angewendet werden. In [CCPP98] werden hierfür verschiedene Strategien diskutiert. Im Mittelpunkt steht dabei das am Ende von Abschnitt 10.3.2 erwähnte *Compliance*-Kriterium, wobei unklar ist, wie es in Verbindung mit Schleifen funktioniert. Aus den Veröffentlichungen zu WIDE geht nicht hervor, in welchen WF-Zuständen die Anwendung einer Änderungsoperation erlaubt ist und welche Zustandsänderungen sich im Anschluß ergeben.

Der von WIDE verfolgte Ansatz zur Evolution von WF-Schemata hat zahlreiche Nachfolgearbeiten [BPS97, JoHe98, KrGe99, LOL98, Wesk99] inspiriert. In [LOL98] wird eine Sprache vorgestellt, mit der sich Strategien zur Evolution von WF-Schemata beschreiben lassen. Eine solche Strategie definiert für die Instanzen eines geänderten Schemas, abhängig von ihrem aktuellen Zustand und ihrer Ausführungshistorie, welche von ihnen auf das neue Schema migriert, welche nach dem alten Schema fortgeführt oder welche abgebrochen werden sollen. Spezielle Operationen zur Abänderung von WF-Schemata gibt es nicht. Auch dynamische Änderungen einzelner WF-Instanzen werden nicht unterstützt. Ein weitergehender Ansatz zur Evolution von WF-Schemata wird von TRAM verfolgt [KrGe99]. Für die WF-Modellierung wird ein ähnlicher Graphformalismus verwendet wie in FlowMark. Darüber hinaus kann ein WF-Schema durch die Anwendung von Änderungsoperationen modifiziert werden, sofern dadurch bestimmte Invarianten nicht verletzt werden. Hierbei handelt es sich um einfache Korrektheitsbedingungen (z.B. Ausschluß von Namenskonflikten, syntaktisch korrekte Verknüpfung von Parametern), die in ähnlicher Form auch für das ADEPT-Basismodell vorliegen, und die für das Schema sowohl vor als auch nach der Anwendung der Änderungsoperationen erfüllt sein müssen. Weitergehende Korrektheitskriterien, etwa im Hinblick auf die Versorgung von Aktivitätenparametern, werden nicht diskutiert. Da keine Details zu den angebotenen Änderungsoperationen vorliegen, ist ein Vergleich mit den $ADEPT_{flex}$-Operationen nicht möglich. Den Schwerpunkt von TRAM bildet die Entwicklung von Konzepten zur Versionierung von WF-Typen. Bei der Änderung eines WF-Typs entsteht eine neue Version, die zusammen mit den alten Versionen des Typs im selben WF-Schema gespeichert werden. TRAM versucht bei einer Typänderung, die zugehörigen Instanzen automatisch auf die neue Version zu migrieren. Dabei wird ein ähnlicher Ansatz wie in WIDE verfolgt.

In dieser Arbeit sind Betrachtungen zur Evolution und Versionierung von WF-Schemata weitgehend ausgeklammert worden. Nichtsdestotrotz wird in ADEPT an diesen Fragestellungen gearbeitet, wobei im Fall von WF-Typänderungen die Behandlung bereits gestarteter WF-Instanzen das größte Problem darstellt [RHD98]. Dies gilt besonders dann, wenn auf Instanzebene auch Ad-hoc-Abweichungen möglich sein sollen. Die Propagierung von Typ-Änderungen auf lokal modifizierte Instanzen wird von keinem der diskutierten Ansätze behandelt. Statt dessen wird bei Änderungen einzelner Instanzen stets eine neue Version des WF-Schemas erzeugt. Ebenfalls nicht hinreichend untersucht ist die Fragestellung, wie die Propagierung von Typ-Änderungen bei einer großen Anzahl von Instanzen effizient bewerkstelligt werden soll. Das ADEPT-Modell bietet eine ausgezeichnete Basis für die Behandlung solcher Aspekte. Darüber hinaus besitzt ADEPT gegenüber den genannten Ansätzen zahlreiche Erweiterungen, etwa für das partielle Zurücksetzen der WF-Kontrolle, die Verwaltung temporärer Änderungen oder die dynamische Änderung von Workflows bei verteilter Steuerung [RBD99].

10.3.4 Zusammenfassung weiterer Ansätze

Weitere graphbasierte WF-Metamodelle, die in gewissem Umfang dynamische Änderungen zulassen, haben wir in Kapitel 2 kennen gelernt. Nennenswert sind die von ProMInanD, InConcert und Regatta verfolgten Ansätze. Generelle Limitationen sind hier, neben der geringen Ausdrucksmächtigkeit und der zu starken Dokumentenorientierung, vor allem die geringe theoretische Fundierung und die mangelhafte Robustheit der Systeme bei dynamischen Änderungen. Einen interessanten Ansatz bietet auch ObjectFlow [HsKl96]. Dieses WfMS verwendet ein graphbasiertes WF-Modell, dessen Ausführungssemantik durch ein Petri-Netz festgelegt ist. Interessant ist die Auseinandersetzung mit dynamischen WF-Eigenschaften und der Hinweis, daß ihre Zusicherung in Verbindung mit Rücksprüngen im Kontrollfluß nicht möglich ist. Ein interessantes Konzept bildet der dynamische Sub-WF, der das mehrmalige parallele oder serielle Ausführen eines Aktivitätenknotens ermöglicht.

10.4 Vergleich mit Graphgrammatik-basierten Ansätzen

Als Formalismus zur Darstellung statischer Ablaufstrukturen und zur Beschreibung des dynamischen Ablaufverhaltens haben Graphen mittlerweile ihren festen Platz im WF-Bereich gefunden. In diesem Abschnitt behandeln wir eine spezielle Klasse graphbasierter Prozeßmodelle, die als formale Grundlage Graphgrammatiken oder Graphersetzungssysteme [Gött88, Roze97] verwenden.

10.4.1 Graphgrammatiken und Graphersetzungssysteme

Während sich die klassische Graphentheorie mit der Analyse statischer Grapheigenschaften befaßt [Neum75], führen typische Fragestellungen adaptiver WfMS dazu, sich mit der Charakterisierung von Graphklassen und dynamischen Graphmodifikationen auseinanderzusetzen. Hier bieten die Ende der 70er Jahre entwickelten Graphgrammatiken (GG), die eine Verallgemeinerung von Chomsky-Grammatiken darstellen, einen interessanten Ansatz [Nagl79]. Eine GG ist, ähnlich wie eine „normale“ Grammatik, ein System von Regeln (Produktionen), die aus einem Startgraphen alle Graphen einer bestimmten Klasse erzeugen [Gött88, Roze97]. Eine Produktionsregel legt die Ersetzung einer linken durch eine rechte Seite fest, wobei sowohl die linke als auch die rechte Seite Graphen sind, also nicht wie bei konventionellen Grammatiken durch lineare Zeichenfolgen beschrieben werden. Eine Produktion ist auf einen Graphen anwendbar, wenn er die linke Seite als Teilgraph enthält. Eng verwandt damit sind Graphersetzungssysteme (GES) [EhTa96, Roze97]. Im Gegensatz zu GG, bei denen die Angabe eines Systems zur Erzeugung bzw. Erkennung von Graphen einer bestimmten Klasse im Vordergrund steht, interessiert hier vordergründig die Beschreibung von Graphtransformationsprozessen. Dynamische Aspekte werden über Graphersetzungsregeln dargestellt. Eine solche Regel identifiziert einen Subgraphen in einem Ausgangsgraphen und substituiert diesen durch einen neuen Subgraphen. GG und GES werden in zahlreichen Gebieten der Informatik angewendet [Roze97], etwa zur Beschreibung der Semantik von (Programmier-)Sprachen oder zur Spezifikation strukturbezogener Editoren. (Die Wirkungsweise von Kommandos wie Erzeugen, Löschen oder Abändern von Teilstrukturen ist auf Graphersetzungen zurückführbar.) Die Idee, GG und GES zur Spezifikation und Modifikation von Workflows zu nutzen, ist in den Projekten DYNAMITE [HJKW95] und Obligations [BoKa95] verfolgt worden, hat ansonsten im WF-Bereich bisher aber wenig Resonanz gefunden.

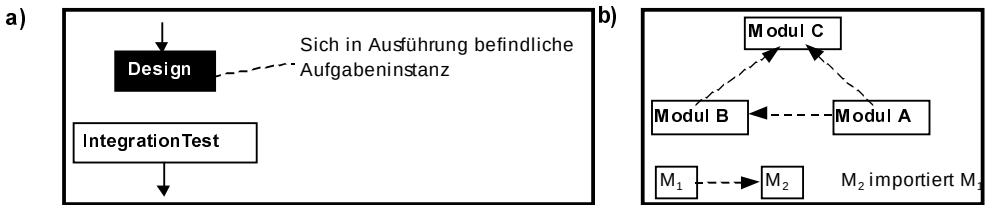


Abbildung 10-18: *Initiales Aufgabennetz (links) und festgelegte Modulstruktur (rechts)*

10.4.2 Unterstützung evolutionärer Softwareentwicklungsprozesse in DYNAMITE

Ein auf der Theorie der GG basierender Ansatz, der die rechnergestützte Koordination von Softwareentwicklungsprozessen (SEP) zum Ziel hat, wird von DYNAMITE verfolgt [HJKW95, HJKW96, HKWJ97]. Ein Charakteristikum von SEP ist ihre Dynamik, d.h. die konkrete Aufgaben- und Ablaufstruktur ergibt sich erst zur Laufzeit. Zwar stehen grundlegende Aufgabentypen, wie Entwurf, Implementierung, Test oder Integration, a priori fest, nicht dagegen ihre konkreten Ausprägungen und deren Anordnungsbeziehungen. Sie ergeben sich aus der Struktur der zu entwickelnden Software, die in der Regel erst im Verlauf der Ausführung festgelegt wird. Die Folge ist eine schrittweise Evolution der Prozeßstruktur. Darüber hinaus können während der Ausführung Rückgriffe erfolgen, die ebenfalls zu Prozeßveränderungen führen, etwa wenn beim Testen eines Moduls ein Fehler festgestellt wird und dadurch Änderungen des Entwurfs notwendig werden.

Eine Prozeßinstanz wird durch ein *Aufgabennetz* repräsentiert, dessen Definition und Ausführung fein granular verschränkt sind. Ein Beispiel für ein Netz, wie es sich zu Beginn der Ausführung darstellen könnte, zeigt Abbildung 10-18a). Angenommen bei der Bearbeitung der Aufgabe *Design* wird die rechts abgebildete Modulstruktur festgelegt, dann wird das Netz wie in Abbildung 10-19 (links) dargestellt expandiert. Wichtig ist, daß der Entwerfer diese Erweiterungen nicht manuell vornehmen muß, sondern die erforderlichen Implementierungs- und Testschritte automatisch, auf der Grundlage der spezifizierten Modulstruktur, vom System erzeugt und in das Netz eingefügt werden. Die Ausgabeports eines Aufgabenknotens können mit den Eingabeports nachfolgender Knoten verknüpft werden, wodurch der Datenfluß zwischen Aufgaben festgelegt wird. Die Netzausführung erfolgt gemäß der aktuellen Netzstruktur, wobei ein Prozeßschritt nicht notwendigerweise abgeschlossen sein muß, bevor mit der Bearbeitung abhängiger Folgeschritte begonnen werden kann. Beispielsweise kann ein Modul implementiert werden, sobald seine Schnittstelle festliegt und unabhängig davon, ob die Schnittstellen anderer Module ebenfalls bekannt sind oder nicht. Die Entwurfsaufgabe bleibt solange aktiv, bis alle ihre Ausgaben feststehen. Umgekehrt kann mit einer Aufgabenbearbeitung begonnen werden, auch wenn ihre Eingaben noch nicht vollständig vorliegen. Im allgemeinen kann eine Aufgabeninstanz während ihrer kompletten Lebensdauer Eingaben empfangen und Ausgaben produzieren.

Tritt beim Testen eines Moduls ein Fehler auf, kann der Anwender einen *Rückgriff* im Ablauf initiieren. Dadurch wird die wiederholte Ausführung bestimmter Aufgaben oder die Durchführung von Korrekturen möglich. Insbesondere können Daten der „gescheiterten“ Aufgabe (z.B. Fehlerbericht eines Tests) an Vorgängerknoten kommuniziert werden. Die Behandlung eines Rücksprungs ist vom aktuellen Status des Zielknotens abhängig. Ist die durch diesen Knoten repräsentierte Aufgabe noch aktiv, kann sie den Rückgriff direkt verarbeiten, etwa durch die Erzeugung einer neuen Version eines Ausgabedokuments oder durch das Einfügen neuer Aufgaben in das Netz. Andernfalls muß die Aufgabe reaktiviert werden. In einem solchen Fall wird eine neue Aufgabenversion erzeugt, was ggf. dazu führt, daß für abhängige Aufgaben ebenfalls neue Versionen generiert und ausgeführt werden. Eine aktive Aufgabe kann verschiedene Versionen ihrer Ausgaben erzeugen und mehrere Versionen ihrer Eingaben lesen. Ein Beispiel für einen Rückgriff zeigt Abbildung 10-19 (rechts). Hier wurde ein beim Testen des Moduls *B* festgestellter Fehler zunächst an die Aufgabe *Implement_B* übermittelt, woraufhin diese reaktiviert wurde. Nach der Änderung der Modulimplementierung erfolgte ein weiterer Rückgriff zur Aufgabe *Design*, um eine modifizierte Modulstruktur (Implementierung des Moduls *D*, das vom Modul *B* importiert werden soll) festzulegen. Letzteres führte schließlich zur Aufnahme entsprechender Implementierungs- und Testschritte in das Aufgabennetz.

Das zentrale Anliegen von DYNAMITE ist es, die dynamische Evolution von Netzen zu unterstützen. Zu diesem Zweck legt der Entwerfer ein Prozeßschema fest, das für Netzinstanzen die Grobstruktur

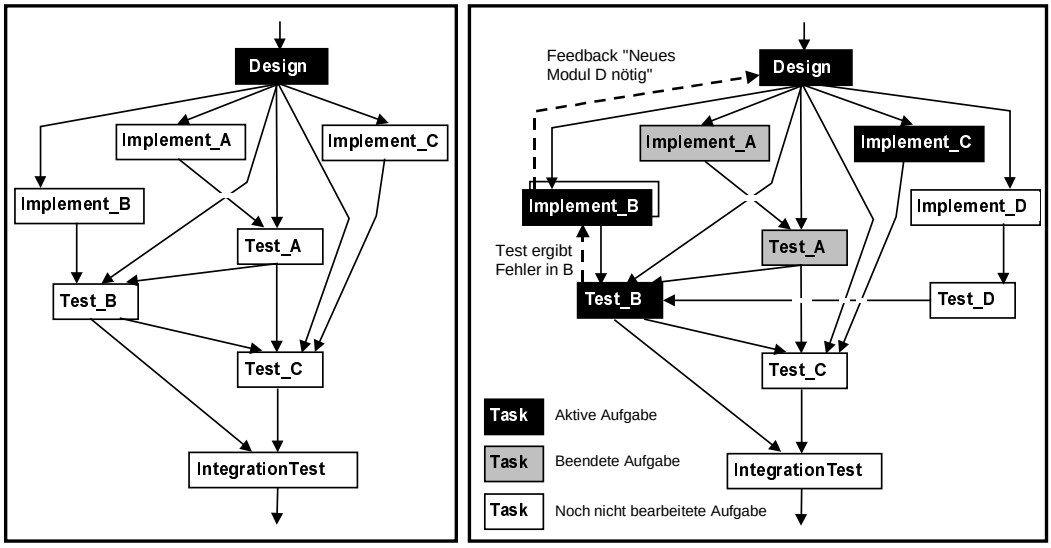


Abbildung 10-19: *Expandiertes Aufgabennetz (links) und infolge eines Rückgriffs dynamisch modifiziertes Netz (rechts) - Aktive Aufgabeninstanzen sind invertiert dargestellt.*

vorgibt und das die Menge der zur Ausführungszeit konstruierbaren Aufgabennetze einschränkt, indem es zulässige Aufgabentypen (z.B. Entwurf, Implementierung) und deren Eigenschaften vorgibt. Konkret können zu einem sich in Ausführung befindlichen Netz nur Instanzen der Aufgabentypen hinzugefügt werden, die durch das Schema erlaubt werden. Für jeden Aufgabentyp beschreibt das Schema, wieviele Ausprägungen in einem Netz erzeugt werden dürfen (Min-/Max-Angabe), was die Ein- und Ausgabeports der Aufgabe sind, welche Dokumenttypen über diese Ports ausgetauscht werden und wie die Ein-/Ausgabeports von Aufgabeninstanzen zu verknüpfen sind. Darüber hinaus werden für Instanzen eines Aufgabentyps zulässige Kontrollabhängigkeiten zu Instanzen anderer Aufgabentypen (mittels *Forward-Kanten*) und zulässige Rücksprünge festgelegt (mittels *Feedback-Kanten*).

Die Semantik des DYNAMITE-Metamodells wird mit der Sprache PROGRES formal definiert [HJKW95]. Sie basiert auf der Theorie der Graphersetzungssysteme und ermöglicht die Festlegung komplexer Graphstrukturen und ihrer Semantik mittels Graphersetzungsregeln. Damit werden die zulässigen Strukturen und das Verhalten von Aufgabennetzen festgelegt. Eine Ersetzungsregel beschreibt den Austausch eines Sub-Netzes mit bestimmten Muster durch einen anderes Subnetz. Ihre Anwendung kann an bestimmte Bedingungen geknüpft werden, wodurch es möglich wird, daß eine Änderung nur in gewissen Zuständen anwendbar ist. So kann eine Aufgabenschnittstelle nicht modifiziert werden, falls die Aufgabe bereits beendet wurde. Die Erzeugung einer neuen Aufgabenversion ist nur zulässig, wenn die aktuelle Version einen Endzustand aufweist. Zu erwähnen bleibt, daß Ersetzungsregeln nicht nur zur Spezifikation zulässiger Editier- bzw. Änderungsoperationen verwendet werden, sondern auch zur Beschreibung von Ausführungs- und Versionierungsoperationen. Dadurch sind alle Operationen auf Aufgabennetzen einheitlich darstellbar [HJKW95, HKWJ97].

DYNAMITE bietet einen theoretisch fundierten Ansatz zur dynamischen Evolution von Prozeßinstanzen. Die Anwendung eines Graphersetzungssystems bietet vielversprechende Perspektiven, um diese Evolution kontrolliert durchführen zu können, etwa abhängig von der Struktur des zu entwickelnden Produkts. Darüber hinaus werden interessante Konzepte zur Versionierung von Aufgabeninstanzen und -dokumenten angeboten. Solange es um die Unterstützung von SEP geht, mögen diese Konzepte optimal sein, für die Verwirklichung adaptiver WfMS reichen sie jedoch nicht aus. Ein wesentlicher Unterschied zu $ADEPT_{flex}$ besteht darin, daß Ad-hoc-Abweichungen von dem durch das Netz festgelegten Ablauf (z.B. Löschen eines Knotens) nicht unterstützt werden. Korrektheitsaspekte, etwa im Hinblick auf die Verknüpfung von Aufgabenports oder dynamische WF-Eigenschaften, werden ebenfalls nicht behandelt. Ein großes Manko ist schließlich die geringe Ausdrucksmächtigkeit des verwendeten Metamodells. Die Knoten eines Netzes können lediglich sequentiell oder parallel angeordnet werden, bedingte Verzweigungen (XOR-Split bzw. -Join), normale Schleifen oder Fehlerrücksprünge mit partiellem Zurücksetzen lassen sich dagegen nicht abbilden. Eine weitere Einschränkung ist, daß ein Prozeßschema während der Ausführung seiner Instanzen nicht modifiziert werden kann.

10.4.3 Komposition von Prozeßschablonen in Obligations

In *Obligations* [Bogi95, BoKa95] ergibt sich ein Instanz-Ausführungsgraph durch die Überlagerung mehrerer Schablonen, die unterschiedliche Sichten des Ablaufs widerspiegeln. Eine WF-Änderung erfolgt durch die Hinzunahme bzw. das Entfernen von Schablonen oder durch deren Abänderung.

WF-Vorlagen werden durch Aktivitätennetze repräsentiert, deren Struktur und Verhalten durch eine Graphgrammatik [Goer90] formal definiert sind. Knoten eines Netzes entsprechen elementaren Aktivitäten oder Sub-Netzen, die durch eine Menge von Ein-/Ausgabeports charakterisiert sind. Ausgabeports können die von der Aktivität produzierten Marken aufnehmen und mit Eingabeports nachfolgender Knoten verknüpft werden. Dadurch liegen der Kontroll- und Datenfluß fest, wobei eine klare Trennung dieser beiden Aspekte nicht möglich ist. Für Knoten kann ein differenziertes Schaltverhalten festgelegt werden, wodurch sich AND-/OR-Aufsplittungen und -Zusammenführungen sowie Schleifen abbilden lassen. Da für die Modellierung nur ein Knoten- und Kantentyp bereitstehen, sind die Möglichkeiten zur Strukturierung der Netze gering. Beispielsweise kann nicht zwischen normalen Kontrollkanten und Schleifenkanten unterschieden werden. Im Gegensatz zu Standard-Petri-Netzen bleiben Marken, die in einem Ausgabeport abgelegt werden, bis zur Terminierung der Instanz erhalten.

Für jeden WF-Typ wird eine Basisschablone verwaltet, die den konkreten Ablauf mehr oder weniger detailliert vorgibt. Der Ausführungsgraph einer WF-Instanz kann zur Laufzeit modifiziert werden. Hierzu dient ein graphischer Editor, der Operationen für das Erzeugen, Löschen und Ändern von Knoten und Kanten anbietet. Voraussetzungen für die Anwendung dieser Primitive werden nicht genannt. Ebensowenig gibt es semantisch höherwertige Operationen. Um den allgemeingültigen Teil eines Ausführungsgraphen, d.h. seine Basisschablone, von seinem lokalen Teil trennen zu können, werden instanzbezogene Änderungen in einer separaten Schablone verwaltet. Der Ausführungsgraph ergibt sich dann durch die Überlagerung der beiden Schablonen, unter Einbeziehung von WF-Kontrolldaten. In Wirklichkeit wird eine Änderungsschablone nicht durch einen Graphen repräsentiert, sondern durch eine Menge von Graphmanipulationsobjekten (*Create, Delete* und *Modify*), die sich auf Knoten und Kanten der Basisschablone beziehen und deren Anwendung die gewünschten Änderungen bewirken. Ihre Semantik und die aus ihrer Anwendung resultierenden Netzmodifikationen werden durch Delta-Graphgrammatiken beschrieben [Bogi95]. Zu erwähnen bleibt, daß Instanzen ihre Basisschablone lediglich referenzieren, so daß sich Änderungen dieser Schablone auf mehrere Instanzen auswirken können. Obligations verwaltet für jede Basisschablone einen Versionsbaum, auf dessen Elemente sich die Instanzen beziehen. Für jede Instanz muß bei ihrer Erzeugung angegeben werden, welches Element des Versionsbaums zur Konstruktion ihres Ausführungsgraphen verwendet werden soll.

Im gerade skizzierten Szenario sehen die Autoren lediglich einen Spezialfall ihres Ansatzes. Im allgemeinen kann der Ausführungsgraph einer WF-Instanz durch die Überlagerung ihrer Basisschablone mit einer oder mehreren anderen Schablonen (Schablone mit lokalen Änderungen, Schablone mit benutzerdefinierter Konfigurationen, usw.) konstruiert werden. Für die Verwaltung dieser Schablonen werden Techniken der multiplen Vererbung eingesetzt, wobei die Graphmanipulationsobjekte einer Schablone jeweils Bezug auf Graphobjekte weniger spezieller Schablonen nehmen. Kernstück bildet ein Kompositionsalgorithmus, der den Ausführungsgraphen einer WF-Instanz aus ihrer Basisschablone, einer Menge darauf basierender, hierarchisch organisierter Schablonen und den WF-Kontrolldaten erzeugt [Bogi95]. Bei dieser Komposition können Fehler auftreten, die von dem Algorithmus zwar erkannt, die aber nicht von vornherein ausgeschlossen werden. Probleme können sich ergeben, wenn Graphobjekte einer Schablone, auf die speziellere Schablonen Bezug nehmen, entfernt werden. Wird bei einer lokalen Instanzänderung eine neue Kante eingefügt, die von einem bestimmten Ausgabeport ausgeht und wird dieser Ausgabeport später infolge einer globalen Änderung aus der Basisschablone gelöscht, resultiert eine Inkompatibilität zwischen der globalen und der lokalen Änderung. Eine weitere Schwäche des Kompositionsalgorithmus ist, daß für hergeleitete Instanzausführungsgraphen potentielle Verklemmungssituationen nicht ausgeschlossen werden. Interaktionen zwischen Kontroll- und Datenfluß bleiben ebenfalls unberücksichtigt. Beispielsweise wird nicht sichergestellt, daß Daten beim Aufruf einer Aktivität in der benötigten Form vorliegen.

10.5 Vergleich mit regelbasierten Ansätzen

Ein deklarativer Formalismus für die WF-Modellierung sind Regeln [EKP98, HKBB92]. Regelba-

sierten Beschreibungssprachen liegt die Annahme zugrunde, daß rein präskriptive Kontrollflußkonstrukte nicht audrucksmächtig genug sind, um die externen und internen Ereignisse, die Einfluß auf das Verhalten eines WF haben, angemessen abzubilden. Regelbasierte Formalismen werden in zunehmenden Maße auch für die Spezifikation von Ausnahmebehandlungen in WfMS [CFM99, CLK99, LSMK98, MüRa99] und die Verwirklichung erweiterbarer WF-Metamodelle [Joer99] verwendet.

10.5.1 Spezifikation von Workflows mittels ECA-Regeln

Eine *regelbasierter* WF besteht aus einer endlichen Menge von Regeln, deren Gesamtheit die WF-Aktivitäten sowie ihre Kontroll- und Datenflußbeziehungen festlegen. Diese Regelmenge wird von der WF-Engine zur Laufzeit interpretiert, wobei unterschiedliche Ausführungsmodelle denkbar sind. Am verbreitesten ist das *ECA-Modell*, bei dem sich eine Regel aus den drei Komponenten Ereignis (*Event*), Bedingung (*Condition*) und Aktion (*Action*) zusammensetzt. Tritt während der WF-Ausführung das Ereignis *E* ein, wird die Bedingung *C* (z.B. Belegung einer WF-Variablen mit einem bestimmten Wert) überprüft und - falls *C* zutrifft - die Aktion *A* angestoßen (z.B. Starten einer Aktivität). Als Folge können neue Ereignisse (z.B. Beendigung einer Aktivität, Veränderung einer WF-Variablen) entstehen, was die wiederholte Auswertung der Regelmenge nach sich zieht. Durch die geeignete Festlegung der Regeln lassen sich sequentielle, parallele, alternative und zyklische Kontrollstrukturen abbilden [EKP98]. Vom Prinzip her sind der Ausdrucksmächtigkeit regelbasierter Formalismen kaum Grenzen gesetzt. So können, neben aktivitätenbezogenen Ereignissen, auch daten-, zeit- oder benutzerbezogene Ereignisse und Bedingungen spezifiziert werden [CFM99, EPK98].

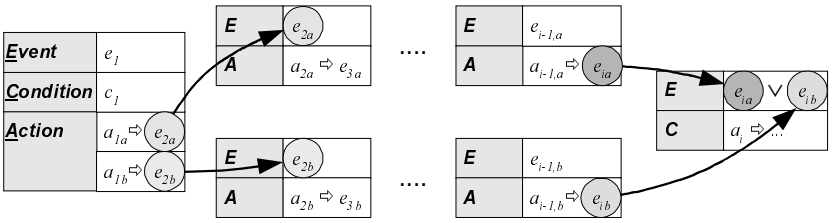


Abbildung 10-20: *Implizite Festlegung von Kontrollabhängigkeiten mittels Regeln. (Die Visualisierung dient hier lediglich zu Veranschaulichungszwecken. Tatsächlich erfolgt die Beschreibung mittels Regeln deklarativ.)*

In [DHL90] wird das Modell der „*Triggers and Transactions*“ vorgestellt, bei dem (transaktionale) Workflows durch Regeln spezifiziert werden. Als wichtigen Vorteil nennen die Autoren die einfache Erweiter- und Abänderbarkeit von Regelmengen, ohne daß eine Neuübersetzung erforderlich wird. Weitere Ansätze, bei denen Kontroll- und Datenabhängigkeiten mittels Regeln beschrieben werden, sind ACTA [Elma92] und INCAS [BMR96]. Um in die Steuerung eines Ablaufs einzugreifen, können in INCAS Regelmengen während ihrer Ausführung geändert werden.

Ein Kernproblem regelbasierter WF-Beschreibungen ist die *implizite* Darstellung der Kontroll- und Datenflußbeziehungen zwischen WF-Aktivitäten. Dadurch ist eine Strukturierung und eine graphische Darstellung der WF-Modelle nur bedingt möglich. Darüber hinaus bestehen eingeschränkte Möglichkeiten zur formalen Analyse des dynamischen WF-Verhaltens. Selbst wenn die Regelmenge eines WF auf einen anderen Formalismus, etwa auf ein Petri-Netz oder in eine Darstellung in temporaler Logik, abbildbar wäre, würden entsprechende Analysen immer noch einen enormen Berechnungsaufwand mit sich bringen [HOR98]. Das vielleicht größte Problem regelbasierter WF-Spezifikationen betrifft die Wartbarkeit und Pflege der WF-Modelle. Bereits für einfache Prozesse können sich umfangreiche, für den Anwender kaum mehr überschaubare Regelmengen ergeben [DHL91]. Aufgrund der unzureichenden Visualisierungs- und Validierungsmöglichkeiten können Regelinteraktionen, die zur Laufzeit zu unerwünschten Effekten (z.B. Verklemmungen) führen, bei der Modellierung nicht immer erkannt werden. Diese Probleme verschärfen sich, wenn neben Kontroll- und Datenabhängigkeiten weitere Aspekte durch das WF-Modell abgebildet werden sollen.

10.5.2 Regelbasierte Modellierung von automatischen Ausnahmebehandlungen

In der Literatur werden verschiedene Ansätze diskutiert, bei denen das normale Verhalten eines WF mittels graphischen Sprachen definiert wird, während Ausnahmen und ihre Behandlung davon ge-

trennt, durch eine Menge von ECA-Regeln, festgelegt werden können [CFM99, CLK99, HaAl98, MüRa99]. Die Verwendung von Regeln wird damit begründet, daß viele Ausnahmen asynchron zum normalen Ablauf auftreten, sowohl was ihre Entstehung als auch ihre Behandlung anbetrifft [CFM99].

In TREX [Stip98] kann der WF-Entwerfer einfache Regeln zur automatischen Behandlung aktivitätenbezogener Ausnahmen (z.B. Scheitern der Bearbeitung) definieren. Eine Regel legt für eine WF-Aktivität fest, welche Behandlung (z.B. Wiederholen der Aktivität, Ausführung einer alternativen Aktivität) beim Auftreten von Ausnahmen eines bestimmten Typs angewendet werden soll. Komplexere Ausnahmebehandlungen (z.B. Sprungoperationen) sind nicht modellierbar. Der von OPERA [HaAl98] verfolgte Ansatz ist in dieser Hinsicht flexibler. Hier kann der Entwerfer, zusätzlich zum normalen Ablauf, Ausnahmebehandlungsprozesse festlegen, die zur Laufzeit durch externe Applikationen oder beim Eintreten bestimmter Ereignisse und Bedingungen (festgelegt durch Prädikate auf den WF-Daten) automatisch durch das WfMS angestoßen werden. Für ihre Modellierung stehen dem Entwerfer dieselben Beschreibungsmittel zur Verfügung, wie für die Definition normaler Prozesse.

Ein weitergehender Ansatz wird von WIDE [Casa96] verfolgt. Für die Modellierung des Normalablaufs wird eine graphische Beschreibungssprache verwendet. Ausnahmebedingte Abweichungen werden mittels ECA-Regeln festgelegt, die bei der Definition mehrerer WF-Schemata wiederverwendbar sind [CFM99]. Zu diesem Zweck werden für häufige Ausnahmen, die während der WF-Ausführung auftreten können, Regelschablonen angelegt und in einem globalen Katalog verwaltet. Im Gegensatz zu OPERA können sich die Regeln nicht nur auf WF-Variablen beziehen, sondern auch auf prozeß- und zeitbezogene Ereignisse. Dem entsprechend vielfältig sind die Möglichkeiten zur Formulierung der Regeln. Mögliche Aktionen einer Regel sind das Versenden einer Nachricht oder die Manipulation von WF-Objekten und -Instanzen (z.B. Starten einer Aktivität). Darüber hinaus kann der Aktionsteil einer Regel strukturelle Anpassungen des WF-Ausführungsgraphen vornehmen. Allerdings wird nicht gezeigt, wie entsprechende Aktionen spezifiziert werden und wie die damit zusammenhängenden Anpassungen zur Laufzeit erfolgen. Ebensowenig werden Probleme behandelt, die sich im Zusammenhang mit der Änderung von WF-Schemata und ECA-Regeln ergeben. Beispielsweise können infolge der Änderung eines WF-Schemas einzelne Regeln nicht mehr auswertbar sein, etwa wenn sie sich auf WF-Objekte beziehen, die bei der Schemaänderung gelöscht wurden.

ADOME [CLK99, CKL97] unterscheidet zwischen *prozeduralen* und *deklarativen* Ausnahmebehandlungsroutinen. Erstere erlauben einfache Sprünge im Kontrollfluß, während letztere die automatische Behandlung asynchroner Ausnahmen ermöglichen. Dazu wird, ebenso wie in WIDE, ein regelbasierter Mechanismus verwendet. Die Zuordnung einer ECA-Regel zu einem WF-Schema kann sowohl zur Modellierungs- als auch zur Laufzeit erfolgen. In beiden Fällen ist die Regel auch für Sub-WFs gültig, es sei denn für diese Sub-Prozesse werden speziellere Behandlungsroutinen vereinbart. Die Ausdrucksmächtigkeit des ADOME-Regelformalismus ist geringer als in WIDE. Unterstützt wird lediglich die Behandlung aktivitätenbezogener Ausnahmen, wobei Aktionen wie das Wiederholen oder Überspringen von Aktivitäten, die Ausführung alternativer Aktivitäten oder der Abbruch des WFs möglich sind. Konzepte zur Verwirklichung dynamischer Änderungen werden nicht behandelt.

Ein interessanter Ansatz wird von AgentWork verfolgt [MüRa99]. Das hier verwendete WF-Metamodell orientiert sich am ADEPT-Basismodell und hat mehrere Elemente davon übernommen (z.B. symmetrische Strukturierung, einfache Sync-Kanten). Einen Schwerpunkt bildet die Untersuchung der Fragestellung, ob und – falls ja – wie sich laufende WF-Instanzen in Ausnahmesituationen *automatisch* anpassen lassen, um mit der WF-Kontrolle korrekt fortfahren zu können. Ähnlich wie in WIDE werden Standardabläufe und globale Ausnahmeregeln getrennt voneinander beschrieben. Im Unterschied zu WIDE erfolgt die Verknüpfung dieser beiden Ebenen nicht bei der WF-Modellierung, sondern dynamisch zur Laufzeit. Dies ist möglich, da Regeln auf einer semantisch hohen Ebene beschrieben werden und sich nicht auf konkrete Objekte (z.B. Aktivitätenknoten) eines bestimmten WF-Schemas beziehen. Eine globale Regel könnte etwa festlegen, daß einem Patienten das Medikament X verabreicht werden soll, wenn sein Blutwert Y unter einen vorgegebenen Schwellenwert fällt. Eine solche Regel kann unabhängig davon an welchen konkreten Prozessen der Patient aktuell teilnimmt zur Anwendung kommen. Im allgemeinen referenzieren Regeln Anwendungsobjekte, die von mehreren WF-Instanzen manipuliert werden. Stellt AgentWork eine Werteveränderung globaler Objekte fest, wertet es die Regelmenge neu aus und ermittelt die anzupassenden Instanzen. Für jede von ihnen wird der minimale Bereich ihres WF-Graphen ermittelt, der von der beabsichtigten Modifikation be-

troffen ist. Da Regeln auf WF-Instanzen mit beliebigem Schema und Status anwendbar sein sollen, ist die Abbildung der durch sie festgelegten „abstrakten“ Änderungen (z.B. Hinzunahme einer Aktivität) auf konkrete Modifikationen der Instanzausführungsgraphen (z.B. Hinzufügen eines Aktivitätenknotens und seiner assoziierten Kontrollkanten) nicht trivial. AgentWork bietet hierfür generische Graphalgorithmen an, mit denen sich z.B. für neu einzufügende Aktivitäten geeignete Zielpositionen automatisch bestimmen lassen [MüRa99]. Möglich sind die automatische Hinzunahme und das Entfernen von Aktivitäten, wobei ausschließlich Kontrollflußaspekte betrachtet werden. Obwohl Schleifen modellierbar sind, werden temporäre Änderungen nicht unterstützt.

In METEOR wird ein WF auf eine Menge ausführbarer (J)ECA-Regeln abgebildet [LSMK98]. Ausnahmebehandlungen basieren hier auf Techniken des fallbasierten Schließens. Tritt während der Ausführung einer WF-Instanz eine Ausnahme auf, versucht das System eine geeignete Behandlungsroutine zu ermitteln. Dazu werden zuerst alle relevanten Informationen zu ähnlichen Ausnahmefällen aus einer Fallbasis bestimmt. Der in der aktuellen Situation am besten passende Fall wird dann zur Behandlung der vorliegenden Situation herangezogen, wobei ggf. noch Adaptionen vorzunehmen sind. Dem entsprechend wird ein neuer Eintrag in der Fallbasis gespeichert. Die Auswahl passender Ausnahmebehandlungen aus der Fallbasis erfolgt auf der Grundlage von Ähnlichkeitsmaßen, die sowohl strukturelle als auch semantische Ähnlichkeiten zwischen den Ausnahmefällen (Ausnahmeart, Behandlungsform, Auswirkungen) berücksichtigen. In TEAMWORK [BEMM98] werden als Erweiterung *Fuzzy Rules* verwendet, um Inferenzen zu Auswirkungen einer Prozeßänderung zu treffen.

10.5.3 Regelbasierte Erweiterung von WF-Metamodellen in MOKASSIN

MOKASSIN bietet verschiedene Konzepte für die Flexibilisierung von WfMS an [Joer98, JoHe98]. Im Gegensatz zu ADEPT interessiert weniger die Fragestellung, auf der Grundlage welcher Operationen dynamische Änderungen durchgeführt und welche formalen Korrektheitskriterien dabei beachtet werden sollen, sondern vielmehr wie das Ausführungsverhalten eines WFs möglichst flexibel und für die Benutzer konfigurierbar gestaltet werden kann. Ein wesentliches Ziel ist deshalb die Bereitstellung eines erweiterbaren WF-Metamodells. Darüber hinaus soll die Anwendbarkeit dynamischer Änderungen nicht an allgemeingültige Korrektheitsbedingungen geknüpft werden, sondern abhängig vom Kontext der Änderung erfolgen können. In der angemessenen Definition des WF-Verhaltens sehen die Autoren die Grundlage sowohl für die Modellierung weniger starrer WFs als auch die Unterstützung dynamischer Änderungen. Grundlegend für beides ist die Integration regelbasierter Techniken mit Standardkontrollkonstrukten einer graphischen Beschreibungssprache [Joer99].

Wichtigstes Beschreibungselement in MOKASSIN ist die *Aufgabe*. Ihre Definition besteht aus einer Schnittstelle sowie ggf. mehreren Implementierungen, von denen eine zur Laufzeit dynamisch gebunden wird. Die *Aufgabenschnittstelle* legt die Attribute und Parameter der Aufgabe fest und beschreibt ihr kontextfreies Verhalten. Eine *Aufgabenimplementierung* ist entweder ein Anwendungsprogramm oder ein Prozeßmodell. Letztere werden graphisch durch ein *Aufgabennetz* dargestellt, dessen Knoten die Aktivitäten und dessen Kanten die Kontroll- und Datenflußbeziehungen zwischen ihnen festlegen. Aktivitätenknoten können mit einer Split-/Join-Semantik (AND/OR) und Kontrollkanten mit einer Transitionsbedingung verknüpft werden. Iterationen sind über einen speziellen Kantentyp modellierbar. Datenflüsse werden modelliert, indem Aktivitätenparameter durch Datenkanten verknüpft werden.

Der Entwerfer kann das Ausführungsverhalten eines WF durch zweierlei Maßnahmen beeinflussen. Einerseits kann er das kontextfreie Verhalten einzelner Aktivitäten anpassen, andererseits besteht die Möglichkeit, das WF-Metamodell um benutzerdefinierte Konstrukte zu erweitern. Das *kontextfreie* Verhalten einer Aktivität wird durch einen Statechart spezifiziert, der die möglichen Bearbeitungszustände und die zulässigen Übergänge zwischen ihnen, die mit einer ECA-Regel verknüpft sein können, festlegt. Standardmäßig wird jeder Aktivität derselbe Automat zugeordnet, er kann allerdings aktivitätenbezogen adaptiert werden, etwa durch die Hinzunahme neuer Zustände bzw. Transitionen oder durch die Änderung der mit Transitionen verknüpften ECA-Regeln. Dadurch läßt sich ein angepaßtes kontextfreies Ausführungsverhalten erzielen. Das *kontextabhängige* Verhalten einer Aktivität ergibt sich aus ihren Kontrollabhängigkeiten zu anderen Knoten des Netzes. Wie bereits erwähnt, werden solche Netze graphisch modelliert. Eine Besonderheit von MOKASSIN ist, daß die Menge der vordefinierten Kontrollabhängigkeiten um benutzerdefinierte Konstrukte erweitert werden kann. Zu diesem

Zweck muß eine Regelmenge angegeben werden, die die Ausführungssemantik des neuen Konstrukts festlegt. Um ein Netz ausführen zu können, wird es zuerst in eine Regelmenge übersetzt, die sowohl das kontextfreie als auch das kontextabhängige Verhalten der WF-Aktivitäten widerspiegelt. Zur WF-Steuerung wird diese Regelmenge von den Laufzeitkomponenten des WfMS interpretiert.

Aus der Integration von Standardkonstrukten mit regelbasierten Beschreibungstechniken resultiert ein ausdrucksstarker Formalismus. Einerseits wird dadurch die Grundlage für die Erweiterbarkeit des Metamodells geschaffen, andererseits ergeben sich aber auch Nachteile. An erster Stelle zu nennen ist die mangelhafte Analysierbarkeit der strukturellen und dynamischen Eigenschaften der Modelle. In [Joer99] wird gezeigt, wie sich durch geeignete Festlegung von ECA-Regeln die Ausführungssemantik des in ADEPT eingeführten Konstrukts der Sync-Kante simulieren läßt. Wie wir gesehen haben, kann die Verwendung von Sync-Kanten zur Laufzeit zu Verklemmungen führen. Während ADEPT solche Fehlersituationen bei der Erstellung bzw. Änderung von Modellen, auf der Grundlage effizienter Analysen (z.B. auf Zyklenfreiheit) ausschließt, bietet MOKASSIN in dieser Hinsicht keine Unterstützung. Selbst wenn die Regelmenge eines WF in eine andere Darstellungsform, etwa auf ein Petri-Netz, abbildbar wäre, müßten immer noch aufwendige Erreichbarkeitsanalysen mit nicht mehr polynomialer Komplexität durchgeführt werden. Unklar ist, ob und - falls ja - wie die Spezifikation anderer WF-Aspekte auf Korrektheit geprüft werden kann. MOKASSIN macht es sich einfach, indem es formale Korrektheitsbedingungen als zu inflexibel einstuft und deshalb komplett auf sie verzichtet.

MOKASSIN bietet Konzepte für die Versionierung von WF-Schemata und für die Propagierung von Schemaänderungen auf laufende WF-Instanzen an [JoHe98]. Die Repräsentation von WF-Schemata und -Instanzen ist eng integriert. Auch die Ad-hoc-Änderung einzelner WF-Instanzen ist möglich. Hierzu muß eine Kopie des zugehörigen Schemas erzeugt werden, die dann vom Benutzer mit Hilfe eines graphischen Editors abgeändert werden kann. Dieser Editor bietet einfache Änderungsprimitive an, etwa für das Hinzufügen von Aktivitätenknoten und für die Festlegung von Kontrollabhängigkeiten. Diese Primitive sind auf einem weitaus niedrigeren Abstraktionsniveau angesiedelt als die $ADEPT_{flex}$-Änderungsoperationen. Aufgrund fehlender Korrektheitskriterien ist MOKASSIN nur eingeschränkt in der Lage, den Benutzer auf potentielle Probleme einer Änderung hinzuweisen oder ihm alternative Strategien für deren Behandlung anzubieten. $ADEPT_{flex}$ räumt beide Möglichkeiten ein, wobei der Umfang und die Komplexität der notwendigen Graphanalysen durch die Ausnutzung von Struktureigenschaften der Modelle erheblich reduziert werden kann.

Für die Anwendung von Schemaänderungen auf laufende Instanzen werden keine (allgemeinen) Statusbeschränkungen getroffen. Beispielsweise kann ein neuer Knoten auch dann als direkter Vorgänger einer Aktivität *X* in einen Ausführungsgraphen eingefügt werden, wenn *X* bereits gestartet oder beendet worden ist. Die Autoren begründen dies damit, daß beim Einfügen in einer Schleife, die betreffende Aktivität in möglichen Folgeiteration noch zur Ausführung kommen kann [Joer99]. Ein gravierender Nachteil dabei ist, daß sich nun globale Zustände ergeben können, die bei normaler Ausführung des neuen Schemas nicht erzeugbar wären. Hieraus resultieren Schwierigkeiten, wenn Systemfunktionen, wie das Zurücksetzen der WF-Kontrolle im Fehlerfall oder die Unterstützung temporärer Änderungen, realisiert werden sollen. Abgesehen davon kann der Verzicht auf Statusbeschränkungen schwerwiegende Fehler im Verlauf der weiteren Ausführung mit sich bringen. Sollen zum Beispiel zwei Aktivitäten *A* und *B* gemeinsam in den Ausführungsgraphen eingefügt werden, von denen *A* Daten für *B* produziert, muß sichergestellt werden, daß *A* in jedem Fall vor *B* zur Ausführung kommt. Dies ist bei dem von MOKASSIN gewählten Ansatz im allgemeinen nicht möglich. Zumindest besitzt der WF-Modellierer aber die Möglichkeit, das kontextfreie Verhalten von Aktivitäten geeignet zu konfigurieren. Beim Einfügen eines neuen Schrittes X etwa generiert das WfMS für jede Aktivität vor die *X* eingefügt werden soll, das Ereignis *add_predecessor[X]*. Hierauf kann auf Instanzebene reagiert werden, etwa durch den Abbruch oder die Kompensation von *X*.

10.6 Vergleich mit deklarativen, planbasierten Ansätzen

Planungssysteme stellen Verfahren für die zielbasierte Erstellung und Ausführung von *Handlungsplänen* bereit [Hert89], vereinzelt auch für deren Anpassung während der Ausführung [KaHe92, Hamm90, WMLW95]. Es bestehen gewisse Querbezüge zu den in dieser Arbeit behandelten Problemstellungen, die im folgenden diskutiert werden. Eine Auseinandersetzung mit Planungssystemen ist auch deshalb lohnenswert, weil es vermehrt Bestrebungen gibt, Techniken des Planens und des auto-

matischen Beweisens auf den WF-Bereich anzuwenden [BeMy98, BeKl98, LiCo93, Lutz88]. Bevor wir hierauf näher eingehen, fassen wir grundlegende Konzepte *klassischer Planungsverfahren zusammen*. Ihre Kenntnis ist für das Verständnis der nachfolgenden Diskussionen notwendig. Detaillierte Beschreibungen zu Planungssystemen finden sich in [Hert89, Mart92, Wilk88].

10.6.1 Klassische Planungsverfahren in der Künstlichen Intelligenz

Ein *Plan* ist eine geordnete Menge von Aktionen, durch deren Ausführung ein gegebenes *Ziel* erreicht werden soll. Ein Beispiel ist die Folge von Handlungen, die ein Roboter bewerkstelligen muß, um von seinem aktuellen Standort zu einem bestimmten Zielort zu gelangen. Einzelne Planaktionen werden als *Operatorinstanzen* (kurz: *Operatoren*) bezeichnet, ihre formale Beschreibung, die Eigenschaften wie Vor- und Nachbedingungen festlegt, entsprechend als *Operatorschemata*. Die Zustände, in denen Planoperatoren anwendbar sind bzw. die aus ihrer Ausführung resultieren, nennen wir *Situationen*. Sie umfassen jeweils eine Menge von logischen Aussagen über die Welt.

Das Hauptanliegen von Planungssystemen ist die automatische Herleitung von Plänen bei gegebener Start- und Zielsituation. Hierzu müssen die zu verwirklichenden Ziele operationalisiert sowie Pläne, Operatoren und Situationen formal repräsentiert werden. Die Grundproblematik besteht darin, einen *korrekten Plan* zu ermitteln. Korrekt bedeutet, daß der Plan in einer gegebenen Startsituation anwendbar ist, seine Operatoren während der Planausführung (entsprechend der definierten Ordnung) anwendbar bleiben und nach der Planausführung die gewünschten Zielmerkmale gelten. Für die Erstellung korrekter Pläne gibt es mehrere Verfahren, die unterschiedliche Repräsentationsformen für Pläne, Operatoren und Situationen wählen. Gemeinsam ist ihnen eine Menge von Annahmen, die für den zu planenden Bereich gelten. So wird davon ausgegangen, daß vollständige Bereichsinformationen vorliegen, die sich während der Erstellung und Ausführung des Plans nicht ändert. Davon ausgenommen sind Veränderungen, die direkt aus der Anwendung der Operatoren resultieren. Des weiteren legen Operatorinstanzen kontextfreie, deterministische Aktionen fest, die weder an Zeitpunkte gebunden sind noch eine Dauer besitzen. Schließlich wird angenommen, daß es genau einen Planer gibt und kein Fallwissen für die Planung vorliegt. Je nachdem, welche Algorithmen und Repräsentationsformen verwendet werden, ergeben sich unterschiedliche Planungsverfahren.

10.6.1.1 Lineares und nichtlineares Planen

Beim *linearen Planen* (LP) wird ein Plan durch ein Tupel $(O, \prec)$ repräsentiert, das aus einer Menge von Operatoren O und einer *strikten* Ordnung $\prec$ auf O besteht. Strikt bedeutet, daß Operatoren sequentiell angeordnet sind. Das wohl bekannteste lineare Planungssystem ist STRIPS [FiNi71]. Hier werden Situationen durch eine Menge von Merkmalen (sog. *fluents*) beschrieben. Ein Operator $o \in O$ entspricht einem Tripel (Pre_o, Add_o, Del_o) mit folgender Semantik: Pre_o beschreibt die Menge der Merkmale, die erfüllt sein müssen, um o anwenden zu können. Add_o definiert die Merkmale die nach der Ausführung von o der neuen Situation hinzugefügt werden (sog. ADD-List) und Del_o legt die Merkmale fest, die bei der Anwendung von o gegenüber der Ausgangssituation wegfallen (DELETE-List). Jedes Merkmal, das nicht in der ADD- oder DELETE-Liste von o enthalten ist, überdauert die Anwendung dieses Operators. Ist der Operator o in der Situation S anwendbar (d.h. $Pre_o \subseteq S$) und kommt er deshalb zur Ausführung, ergibt sich die Merkmalsmenge $S_{new} := (S \neg Del_o) \cup Add_o$. Ausgehend von einer Startsituation S_o und einer Zielsituation G ist es nun die Aufgabe des linearen Planers, aus einer Grundmenge vordefinierter Operatorschemata eine endliche Folge von Operatoren $o_1, o_2, ..., o_n$ abzuleiten, deren sequentielle Ausführung zu den gewünschten Zielmerkmalen führt. Das bedeutet, daß der Planoperator o_1 in S_o ausführbar sein muß ($Pre_{o_1} \subseteq S_0$) und daß hieraus eine Situation S_1 resultieren soll, in der o_2 anwendbar wird ($Pre_{o_2} \subseteq S_1$), usw. Am Ende der Planausführung, d.h. nach Beendigung des Operators o_n, soll sich eine Situation S_n ergeben, in der alle Zielmerkmale erfüllt sind ($G \subseteq S_n$). Algorithmisch wird das LP auf eine Suche in einem Zustandsraum zurückgeführt, der durch die Anwendung aller in einer gegebenen Situation jeweils ausführbaren Operatoren expandiert wird. Die Suchverfahren arbeiten entweder von den Zielen ausgehend rückwärts (*Zielregression*) oder mit der Startsituation beginnend vorwärts (*progressive Planung*). Ein Grundproblem dabei ist, daß der Zustandsraum bei einer größeren Anzahl von Merkmalen und Operatorschemata sehr schnell „explodiert“. Abhängig davon, welche der zu erreichenden Teilziele der Planer zuerst bearbeitet, können sich

zudem mehr oder wenige optimale Pläne ergeben. Das bedeutet, daß auch Pläne resultieren können, deren Ausführung zwar zum gewünschten Ziel führt, dieses aber nicht auf dem kürzesten bzw. kostengünstigsten Weg erreicht. Der Grund hierfür sind negative Interaktionen von Teilzielen, infolge derer die Bearbeitung eines Ziels bestimmte Teile der Arbeit „zerstören“ kann, die man geleistet hat, um ein anderes Ziel zu erfüllen. Diese Problematik versucht man beim nichtlinearen Planen zu vermeiden.

Während die LP davon ausgeht, daß ein Plan eine Folge von Aktionen mit totaler Ordnung darstellt, werden beim *nichtlinearen Planen* (NLP) die Planoperatoren lediglich partiell geordnet. Dadurch wird bei der Planerstellung vermieden, daß zu früh eine bestimmte Operatorreihenfolge festgelegt wird. Die partielle Ordnung ist nicht gleichbedeutend mit nebenläufiger Ausführung. Vielmehr ist sie so zu verstehen, daß die zugehörigen Operatoren in jeder vollständigen Ordnung ausführbar sind, die mit der partiellen Ordnung verträglich ist. Dem entsprechend wird ein nichtlinearer Plan genau dann als *korrekt* bezeichnet, wenn jede Serialisierung des Plans die gewünschte Zielsituation herstellt, d.h. korrekt im Sinne eines linearen Plans ist. Aufgabe der NLP ist es, bei gegebener Start- und Zielsituation einen korrekten Plan zu ermitteln.

Hierzu definieren wir einige wichtige Begriffe. Wir nennen einen Operator $o = (Pre_o, Add_o, Del_o)$ einen *Erzeuger* (*Zerstörer*) des Merkmals m, wenn m in der ADD-List (DELETE-List) von o enthalten ist. Des weiteren heißt o ein *Nutzer* von m, falls das Merkmal m eine notwendige Vorbedingung für die Ausführung von o ist ($m \in Pre_o$). Sei nun $(O, \prec)$ ein nichtlinearer Plan und seien $e, n \in O$ zwei Operatoren und m ein Merkmal, das von e erzeugt und von n genutzt wird. Dann heißt n *kausal abhängig* von e bzgl. m (Notation: $e \rightarrow_m n$), falls $e \prec n$ gilt und es keinen Erzeuger $o \in O$ von m gibt mit $e \prec o \prec n$. Eine kausale Abhängigkeit drückt aus, welcher Operator welches Merkmal für welchen anderen Operator erzeugt, d.h. wer der letzte Produzent des Merkmals ist. Für einen korrekten, nichtlinearen Plan $(O, \prec)$ muß gelten, daß es zu jedem Operator $n \in O$ der ein Merkmal m nutzt, mindestens einen Operator $e \in O$ gibt, der dieses Merkmal erzeugt und für den $e \prec n$ gilt (oder aber m muß durch die Startsituation festliegen; sog. *Wohlgeformtheit*). Diese Bedingung reicht noch nicht aus, um die Korrektheit gewährleisten zu können. Grund ist, daß ein von einem Operator erzeugtes Merkmal durch nachfolgende Operatoren wieder zerstört werden kann. Es ist also nicht ausgeschlossen, daß eine kausale Abhängigkeit $e \rightarrow_m n$ einen m-Zerstörer $z \in O$ umschließt ($e \prec z \prec n$) oder der Plan nach seiner Vervollständigung einen solchen m-Zerstörer umschließen könnte (sog. *Konflikt*). Aufgabe des Planers ist es, solche Konflikte auszuschließen. Ein Plan ist *fertig*, wenn er wohlgeformt und konfliktfrei ist.

Auch das NLP wird algorithmisch wieder auf ein Suchproblem abgebildet. Im Gegensatz zum LP wird die Suche nicht im Zustandsraum durchgeführt, sondern im Raum der nichtlinearen Pläne. Ein Vorteil dabei ist, daß die Anzahl möglicher Pläne beim NLP infolge der partiellen Ordnung von Operatoren deutlich geringer ist als beim LP. Die Idee des NLP ist vom Prinzip her einfach [Hert89]: Wenn an einem bestimmten Punkt der Planung mehrere Teilziele vorliegen - dies ist zu Beginn der Planung oder beim Herstellen der Vorbedingungen eines Operators der Fall -, bearbeitet sie der Planer zunächst unabhängig voneinander. D.h. er ermittelt die Teilpläne, ohne vorzeitig die Reihenfolge festzulegen, in der ihre Operatoren im Gesamtplan vorkommen sollen. Bei Vorliegen eines wohlgeformten Plans muß der Planer für kausale Abhängigkeiten noch untersuchen, ob für eine Serialisierung der Teilpläne der Fall eintreten kann, daß sich zwischen den Erzeuger und den Nutzer des zugehörigen Merkmals ein Zerstörer schieben kann. Falls nein, wird weiter nichts über die Reihenfolge ausgesagt, in der die Teilpläne ausgeführt werden sollen. Wird ein solcher Konflikt festgestellt, muß eine Bereinigung durchgeführt werden, indem der Planer z.B. die bisherige partielle Ordnung durch Hinzunahme von Reihenfolgenbeziehungen verfeinert oder neue Operatoren hinzufügt [Hert89].

Insgesamt stellt das Erkennen und Auflösen von Konflikten eine der Hauptschwierigkeiten beim NLP dar. Es ist deshalb vor allem dann sinnvoll anwendbar ist, wenn es zwischen verschiedenen Teilzielen wenig Interaktionen gibt. Darüber hinaus kann es beim klassischen NLP vorkommen, daß der Planer ein- und denselben Plan mehrfach und auf unterschiedliche Art und Weise erzeugt. Dies ergibt praktisch jedoch wenig Sinn, insbesondere dann nicht, wenn ein solcher Plan vom Benutzer bereits als nicht akzeptabel abgelehnt wurde. Eine wichtige Erweiterung ist hier das *systematische NLP*, das für ein Problem alle möglichen Pläne ohne Wiederholung generiert. Der bekannteste Vertreter nichtlinearer Planer ist SIPE [Wilk88].

10.6.1.2 Einstufiges und mehrstufiges Planen

In [Hert89] wird weiter zwischen *einstufigem* und *mehrstufigem* Planen unterschieden Beim einstufigen Planen wird ein Plan in einem Zug erstellt, beim mehrstufigen Planen wird in verschiedenen Abstraktionsstufen vorgegangen. In beiden Fällen kann LP oder NLP angewendet werden.

Beim mehrstufigen Planen wird eine Reduktion der Komplexität der Planungsaufgabe erzielt, indem zuerst ein grober Plan erstellt wird, der dann schrittweise verfeinert wird. Diese Verfeinerung kann sich sowohl auf Zustände als auch auf die verwendeten Operatoren beziehen. Bei der *Situationsabstraktion* werden die einzelnen Merkmale einer Zielsituation gewichtet. Bei der Erzeugung von abstrakten Plänen werden nur die Merkmale herangezogen, die ein bestimmtes Mindestgewicht besitzen. Später wird der Plan verfeinert, indem sukzessive auch Merkmale mit geringerer Gewichtung berücksichtigt werden. Beim Verfeinern kann sich herausstellen, daß der abstrakte Plan in der bisherigen Form nicht realisierbar ist. Für diesen Fall muß dann zuerst ein alternativer abstrakter Plan gefunden werden (z.B. durch Backtracking). Bei der *Operatorabstraktion* werden neue Operatoren „konstruiert“, die in der realen Welt nicht direkt ausführbar sind (und für die es dem entsprechend kein Operatorschema gibt). Es handelt sich dabei um die logische Zusammenfassung einer Folge von elementaren Operatoren. Abstraktere Operatoren werden beim Planen genauso verwendet, wie andere Operatoren auch und unterscheiden sich von diesen nur dadurch, daß sie zusätzlich zu Vorbedingungen, `ADD`-List und `DELETE`-List auch noch einen sog. *Plot* besitzen. Er gibt an, durch welche Teilpläne oder ausführbare Operatoren der abstrakte Operator bei seiner Verfeinerung ersetzt werden soll.

Zu diesen Verfahren gibt es verschiedene Erweiterungen, etwa das *konditionale* Planen, das *probabilistische* Planen, das Planen mit *Constraints*, das *Meta-Planen* und das *verteilte* Planen [Hert89].

10.6.1.3 Anpassung und Reparatur von Plänen zur Laufzeit

Ursprünglich basierten KI-Planungsverfahren auf der Annahme, daß sich das Wissen über mögliche Planausführungsschritte bzw. die Menge der dem Planer bekannten Operatorschemata, nach Planungsbeginn nicht mehr ändert. Des weiteren wurde unterstellt, daß ein fertiger Plan während seiner Ausführung nicht mehr modifiziert werden muß. Beide Annahmen treffen in der Praxis nicht immer zu [BeMy98]. Vielmehr können während der Ausführung eines Plans Situationen eintreten,

- in denen Planoperatoren ausgeführt werden sollen, obwohl ihre Vorbedingungen nicht (vollständig) erfüllt sind (*precondition failure*).
- in denen Planoperatoren nicht erfolgreich ausgeführt werden können, so daß die definierten Nachbedingungen entweder gar nicht oder nur unvollständig erfüllt sind (*action failure*).

Zur Behandlung dieser Probleme werden in der Literatur verschiedene Vorschläge gemacht [BeMy98, KaHe92, WMLW95]. Sie zielen darauf, Pläne zur Laufzeit automatisch zu „reparieren“, wenn Fehlersituationen eintreten. Für das dynamische Umplanen werden sowohl fallbasierte als auch generische Ansätze vorgeschlagen [Hamm90, Wilk85]. Um etwa festzustellen, welche Auswirkungen ein semantischer Fehler (*action failure*) auf die weitere Planausführung hat und welche Anpassungen zur Wiederherstellung der Plankorrektheit notwendig sind, müssen kausale Abhängigkeiten zwischen Operatoren erfaßt und explizit verwaltet werden [KaHe92, Wilk85]. Darüber hinaus muß die Planreparatur so erfolgen, daß möglichst viel vom bisherigen Plan erhalten bleibt [Myer98]. Bei der Mehrzahl gängiger Systeme muß die Ausführung eines Plans während seiner Änderung vollständig unterbrochen werden (*synchrones Umplanen*), wodurch die Bearbeitung evtl. lange blockiert wird. Ein alternativer Ansatz wird in [WMLW95] diskutiert. Hier können die Ausschnitte des Plans, die vom Umplanungsprozeß nicht direkt betroffen sind, teilweise nebenläufig ausgeführt werden (*asynchrones Umplanen*).

10.6.2 Anwendung von Planungstechniken auf den Workflow-Bereich

In zunehmendem Maße gibt es Bestrebungen, Techniken des Planens auf den WF-Bereich anzuwenden und sie als Grundlage für die Realisierung adaptiver WfMS zu verwenden [BeKl99, DMP97, Myer98, LiCo93]. Bevor wir einige dieser Ansätze näher vorstellen und gegenüber ADEPT abgrenzen, gehen wir zunächst auf generelle Querbezüge zwischen Planungssystemen und WfMS ein.

10.6.2.1 Grundlegende Konzepte für ein planbasiertes Workflow-Management

Um den WF-Entwerfer unterstützen zu können, muß dem Planer eine deklarative Beschreibung des Prozesses vorliegen. Einzelne Aktivitäten können als Planoperatoren und das WF-Modell als Plan repräsentiert werden. Der Entwurf wird dem entsprechend zur Planungsaufgabe, bei der ein Planer, ausgehend von den *WF-Zielen*, das Modell sukzessive expandiert, bis ein wohlgeformter und konfliktfreier Plan resultiert. Voraussetzung dafür bildet die formale Beschreibung von Aktivitätenvorlagen durch Operatorschemata. In [Lutz88] wird ein solcher Ansatz für den Bereich der Büroautomatisierung vorgestellt. Ausgehend von einer formalen Repräsentation der Prozeßziele wird der WF durch Rückwärtsplanung bestimmt. Ein ähnlicher Ansatz wird von POLYMER verfolgt [BhCr90, LeCr89]. In beiden Fällen besteht ein WF-Modell aus einer Menge von Zielen und partiell geordneten Aktivitäten.

Lineare Pläne sind nicht ausdrucksstark genug, um reale Prozesse angemessen abbilden zu können. Der Grund ist, daß sie lediglich die Abbildung sequentieller WFs gestatten. WF-ähnliche Strukturen findet man vor allem beim mehrstufigen NLP vor. Hier entspricht ein Plan einem gerichteten Graphen, dessen Knoten die Aktivitäten und dessen Kanten die Anordnungsbeziehungen zwischen ihnen widerspiegeln. Mit den in Abschnitt 10.6.1 beschriebenen Konzepten sind aber nur sequentielle und parallele WFs abbildbar. Die Berücksichtigung bedingter Pfade macht bereits aufwendige Erweiterungen notwendig (sog. *konditionales Planen*). Ein inhärentes Problem planbasierter Modelle ist auch die fehlende Möglichkeit zur Abbildung von Schleifen. Dadurch bleibt ihre Anwendbarkeit auf bestimmte WF-Klassen beschränkt. Hinzu kommt, daß in Planungssystemen oftmals kein expliziter Datenfluß spezifiziert wird, da Planoperatoren auf einem globalen Modell der Welt operieren. D.h. es wird davon ausgegangen, daß Operatoren weder die Historie der vorliegenden Situation kennen, noch daß irgendwelche Abhängigkeiten von früher getroffenen Planentscheidungen festgehalten werden können. Die *WF-Steuerung* entspricht der Ausführung eines Plans, wobei zwischen den Ausführungskomponenten herkömmlicher Planungssysteme und prozeßorientierter WfMS große Unterschiede bestehen. So ist die Integration fremder Anwendungskomponenten kein primäres Anliegen von Planungssystemen. Dies würde sich aufgrund der impliziten bzw. fehlenden Festlegung von Datenflüssen schwierig gestalten, da die korrekte Parameterversorgung und -übernahme nicht gewährleistet wäre. Darüber hinaus fehlen Konzepte zur persistenten Verwaltung und Versionierung der WF-Daten. Sie sind notwendig, um das kontrollierte Zurücksetzen im Fehlerfall zu ermöglichen.

Im Kontext dieser Arbeit von besonderem Interesse sind die in Abschnitt 10.6.1.3 erwähnten Möglichkeiten zur Reparatur von sich in Ausführung befindlichen Plänen. Hier bestehen Querbezüge zu *dynamischen WF-Änderungen*. Allerdings gestalten sich Planänderungen oftmals weniger schwierig als Modifikationen komplexer WFs. Pläne sind einfach strukturierte Graphen, die aufgrund ihrer azyklischen Struktur die wiederholte Ausführung von Prozeßfragmenten (Schleifen) nicht zulassen. Dadurch vereinfachen sich die Verfahren zur dynamischen Planadaption erheblich. Beispielsweise muß nicht zwischen temporären und permanenten Plananpassungen differenziert werden. Statt dessen bleiben Änderungen eines Plans bis zum Ende seiner Ausführung erhalten, es sei denn sie fallen späteren Umplanungen zum Opfer. Ein grundlegendes Problem, das sich bei herkömmlichen Planungssystemen ergibt, ist die implizite bzw. fehlende Festlegung von Datenflüssen zwischen Planoperatoren. Sie macht dynamische Planadaptionen schwierig und fehleranfällig.

10.6.2.2 Anwendung von Planungstechniken zur Behandlung von Ausnahmen in ACT

Das im Projekt „Intelligentes Workflow-Management“ entwickelte ACT-System wendet Techniken des Planens auf den WF-Bereich an [BeKl98, BeKl99]. Kernstück bildet ein nichtlinearer Planer, der den Entwerfer bei der Erstellung und Validierung von WF-Modellen unterstützen soll. Zur Laufzeit steuert und überwacht ACT die Ausführung von Planinstanzen und führt beim Auftreten semantischer Fehler automatische Plankorrekturen durch.

a) WF-Modellierung und -Adaption in ACT

ACT liegt eine deklarative Beschreibung von WFs zugrunde. Ein WF-Modell besteht aus einer formalen Spezifikation der WF-Ziele, einer Menge von WF-Variablen, einer Menge von Aktivitäten (inkl. Vor-/Nachbedingungen) sowie verschiedenen Metadaten. Letztere beschreiben *kausale* Abhängigkeiten zwischen Aktivitäten ($A \rightarrow_c B$: A etabliert Bedingung c für B) sowie (semantische) Beziehungen zwischen WF-Variablen. Ein WF-Modell heißt *vollständig*, wenn alle Ziele erreichbar sind

und es für jede Vorbedingung *pre* einer beliebigen Aktivität Y eine kausale Abhängigkeit $X \rightarrow_{pre} Y$ gibt, die sicherstellt, daß diese Bedingung zur Laufzeit erfüllbar ist. Sind darüber hinaus alle Integritätsbedingungen für WF-Variablen erfüllt, so heißt das Modell *korrekt*.

ACT repräsentiert Aktivitäten durch Operatoren, deren Vor- und Nachbedingungen feststehen. Ein Beispiel für eine Vorbedingung ist das Vorliegen eines Eingabedokuments. Umgekehrt führt eine Operatorausführung zur Erzeugung von Ausgaben und zum Eintreten von Nachbedingungen. Produziert (Benötigt) eine Aktivität eine Ausgabe (Eingabe) vom Typ *docTyp* mit Bezeichner *docId*, wird ihr die Nachbedingung (Vorbedingung) *exists_doc* (*docTyp*, *docId*) zugewiesen. Darauf basierend erfolgt die Herleitung eines vollständigen und korrekten WF-Modells semi-automatisch durch einen NLP. Ausgangspunkt bildet die formale Beschreibung der Startsituation und der zu erreichenden WF-Ziele. Sie legt fest, welche Dokumente zu Beginn bzw. nach Beendigung der Ausführung vorliegen sollen. Bei der Planerstellung verwaltet der Planer die aktuelle Menge O feststehender WF-Aktivitäten, die Menge C der zwischen ihnen bestehenden kausalen Abhängigkeiten sowie die Menge P aller noch nicht gesicherten Vorbedingungen von Aktivitäten aus O. Zu Beginn der Planung umfaßt O genau zwei Elemente, die Start- und die Endaktivität des WF. Die Nachbedingungen der Startaktivität definieren die Ausgangssituation und die Vorbedingungen der Endaktivität die zu erreichenden WF-Ziele. P umfaßt initial die für die Endaktivität definierten Vorbedingungen und C ist leer. Hiervon ausgehend verfährt der Planer wie folgt: Solange $P \neq \emptyset$ gilt, wählt er eine beliebige Vorbedingung *pre* (einer Aktivität *n*) aus P aus. Dann versucht er, eine bereits vorhandene Aktivität $e \in O$ zu bestimmen, welche *pre* als Nachbedingung erzeugt. Falls dies gelingt, wird die kausale Abhängigkeit $e \rightarrow_{pre} n$ der Menge C hinzugefügt und *pre* wird aus P entfernt. Andernfalls wird zuerst eine neue Aktivität *e'*, die *pre* als Nachbedingung besitzt, aus der Grundmenge definierter Aktivitäten bestimmt und in O eingefügt. Anschließend wird wie oben beschrieben verfahren. Für den letztgenannten Fall müssen noch die Vorbedingungen der Aktivität *e'* in die Menge P aufgenommen werden. Das Verfahren terminiert, sobald P leer ist.

Hergeleitete WF-Modelle können vom Entwerfer noch abgeändert werden. Umgekehrt kann er halbfertige Pläne vorgeben, die dann als Eingabe für den Planer dienen und von diesem vervollständigt werden. In beiden Fällen muß ein korrektes und vollständiges WF-Modell resultieren. Benutzerdefinierte Festlegungen werden notwendig, wenn Kontroll- bzw. Datenabhängigkeiten zwischen Aktivitäten explizit modelliert werden sollen. Beispielsweise kann zwischen einer Aktivität X mit Nachbedingung *exists_doc* (*docTyp*, $docId_1$) und einer Aktivität Y mit Vorbedingung *exists_doc* (*docTyp*, $docId_2$) ein Datenfluß modelliert werden, indem in das WF-Modell die semantische Beziehung „$docId_1 = docId_2$“ aufgenommen wird. Aus der Gesamtheit der festgelegten Datenflußbeziehungen werden dann kausale Abhängigkeiten abgeleitet („Wer ist der letzte Erzeuger des von einer Aktivität konsumierten Dokuments?“) und mit Hilfe eines Reason-Maintenance-Systems in einem Abhängigkeitsgraphen verwaltet [BeKl99]. Darauf basierend sind Schlußfolgerungen zur Korrektheit und Vollständigkeit von Modellen möglich. Zu erwähnen bleibt, daß eine Aktivität alternative Nachbedingungen besitzen kann. Für diesen Fall wird in unterschiedlichem Kontext geplant, woraus ein WF-Modell mit bedingten Ausführungszweigen resultiert. Die Abbildung zyklischer Prozesse ist nicht möglich. Ein einfaches Beispiel („Buchen einer Reise“) zeigt Abbildung 10-21 a). Die Ausgangssituation (*Vorliegen des Zielortes*) wird als Nachbedingung des Startoperators und die zu erreichenden WF-Ziele (*Fahrt_gebucht* $\wedge$ *Hotel_gebucht*) werden als Vorbedingung des Endoperators abgebildet. Vorbedingung für die Ausführung der beiden Operatoren „*Flug buchen*“ und „*Hotel buchen*“ ist das Vorliegen des Reiseortes. Als Nachbedingung ergeben sich die Zustände *Fahrt_gebucht(Flugzeug)* bzw. *Hotel_gebucht*. Der dargestellte Plan ist vollständig und korrekt. Kausale Abhängigkeiten bestehen genau zwischen den Operatoren, die durch einen Pfeil miteinander verknüpft sind. Auf der Grundlage dieser kausalen Abhängigkeiten sind Fragen beantwortbar wie „Welche WF-Ziele sind nicht mehr erreichbar, wenn eine Aktivität eine zur Modellierungszeit spezifizierte Nachbedingung nicht erfüllt (*action failure*)?“ oder „Warum ist eine Aktivität nicht ausführbar (*precondition failure*)?“.

Die Notwendigkeit von Planadaptionen wird an die Erfüllbarkeit von Vorbedingungen, über die auch der Datenfluß spezifiziert wird, geknüpft. Wird eine Aktivität infolge eines semantischen Fehlers abgebrochen und ist dadurch ein Existenzprädikat in der Vorbedingung eines nachfolgenden Operators nicht mehr erfüllbar, wird eine Umplanung gestartet. Wie sich dieser Meta-Prozeß darstellt und welche Strategien für Plananpassungen existieren, wird in den ACT-Veröffentlichungen nicht behandelt.

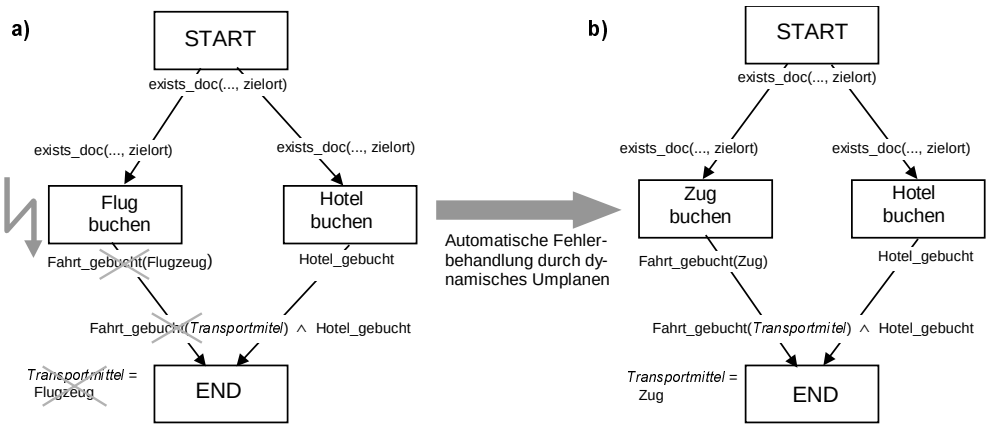


Abbildung 10-21: *Automatische Fehlerbehandlung in ACT durch dynamisches Umplanen*

Für das Beispiel aus Abbildung 10-21 a) nehmen wir an, daß während der Ausführung einer Instanz die Situation eintritt, daß kein Flug gebucht werden kann. Die Nachbedingung des Operators „Flug buchen“ ist demzufolge nicht erfüllbar. Aufgrund der zum Endoperator bestehenden kausalen Abhängigkeit schließt der ACT-Planer dann automatisch, daß die WF-Ziele nicht mehr verwirklichbar sind. In einem solchen Fall wird versucht, eine automatische Reparatur des Plans vorzunehmen. Eine mögliche Lösung zeigt Abbildung 10-21 b). In dem hier abgebildeten Plan wurde gegenüber a) der Operator „Flug buchen“ durch den Operator „Zug buchen“ ersetzt und eine neue kausale Abhängigkeit zwischen „Zug buchen“ und dem Endoperator abgeleitet. Es resultiert wieder ein korrekter Plan.

b) Bewertung von ACT und Vergleich mit ADEPT

Im Gegensatz zu vielen anderen Ansätzen behandelt ACT Korrektheitsfragestellungen und bezieht darüber hinaus semantische Aspekte in den WF-Entwurf mit ein. Die Annahme der Autoren, daß ihr deklarativer Beschreibungsformalismus ausdrucksstark genug ist, um jede beliebige WF-Sprache abbilden zu können [BeKl99], ist allerdings falsch. Die für Planer inhärenten Probleme, wie die hohe Komplexität der Verfahren und die fehlende Möglichkeit zur Abbildung zyklischer Prozeßstrukturen, werden nicht diskutiert. Dem entsprechend sind Schleifen- und Fehlerrücksprünge, wie sie von ADEPT unterstützt werden, nicht abbildbar, so daß die Anwendbarkeit von ACT auf bestimmte Teilklassen von WFs beschränkt bleibt. Darüber hinaus sieht der deklarative Formalismus keine Möglichkeiten zur Vormodellierung planbarer Abweichungen vor. Der zur Unterstützung bedingter Ausführungen verfolgte Ansatz ist ebenfalls problematisch. Liegen Aktivitäten mit alternativen Nachbedingungen vor, muß in unterschiedlichem Kontext geplant werden. Das bedeutet, daß für jede mögliche Aktivitätenmenge einer WF-Ausführung logisch ein separater Plan erstellt wird. Demzufolge muß für jeden Plankontext ein eigener Graph mit kausalen Abhängigkeiten abgeleitet und verwaltet werden. Bei n bedingten Verzweigungen sind dies 2^n Abhängigkeitsgraphen. Angesichts der hohen Komplexität planbasierter Verfahren - viele Basisprozeduren zum NLP sind *NP*-hart - ist dieser Ansatz ineffizient und nicht zur Unterstützung einer großen Anzahl von WF-Typen und -Instanzen geeignet. Ein weiterer Mangel ist, daß kein formales Ausführungsmodell vorliegt. Es ist unklar, wie die Zustände von WF-Instanzen formal repräsentiert werden und welche Zustandsübergänge sich nach der Beendigung einer Aktivität ergeben können. Ebensowenig geht aus den ACT-Veröffentlichungen hervor, ob und - falls ja - welche Aussagen zum dynamischen Verhalten von WFs gewonnen werden können.

Der verfolgte Ansatz zur Abbildung von Datenflüssen und zur Verwaltung der sich daraus ergebenden kausalen Abhängigkeiten bieten gegenüber ADEPT keine Vorteile. Mit dem in Kapitel 5 skizzierten *WriterExists*-Algorithmus läßt sich die Vollständigkeit von Datenflußschemata effizienter und für eine größere Klasse von WFs feststellen, als dies in ACT möglich ist. Dies gilt in besonderem Maße, wenn im WF-Modell bedingte Pfade auftreten. Fraglich ist, ob sich der Aufwand für die explizite Verwaltung kausaler Abhängigkeiten rechnet. Für jeden Lesezugriff einer Aktivität wird der letzte Schreiber des betreffenden Dokuments explizit gespeichert, wobei im Fall bedingter Verzweigungen ggf. mehrerer solcher Versorgeraktivitäten vorliegen. Dieselben Informationen lassen sich in ADEPT durch einfache Graphanalysen ableiten. Bei Ausfall eines Schreibers führt ACT automatisch eine Planreparatur durch, unabhängig davon, ob das betreffende Dokument schon von einer anderen Vorgängeraktivität

geschrieben wurde oder nicht. Für den erstgenannten Fall kann auch ein „veralteter“ Wert zur Versorgung des Lesezugriffs dienen. ADEPT bietet diese Option an. Darüber hinaus räumt ADEPT bei der Beschreibung von Aktivitätenparametern mehrere Freiheitsgrade (optional vs. obligat, nachforderbar vs. nicht nachforderbar) ein, die im Hinblick auf die Adaption von Datenflußschemata nützlich sind.

Wie stellen sich die von ACT angebotenen Konzepte zur Durchführung dynamischer Änderungen im Vergleich zu $ADEPT_{flex}$ dar? ACT unterstützt einfache Änderungen einzelner WF-Instanzen. Sie werden automatisch durchgeführt, wenn aufgrund eines semantischen Fehlers die Nachbedingungen einer Aktivität und demzufolge die Vorbedingungen von Folgeaktivitäten nicht mehr erfüllbar sind. Für diesen Fall wird die Ausführung unterbrochen und es wird versucht, den WF automatisch anzupassen. Benutzerdefinierte Ad-hoc-Abweichungen, etwa das dynamische Hinzufügen von Aktivitäten zu einer WF-Instanz oder das Zurücksetzen der WF-Ausführung, werden von ACT dagegen nicht unterstützt. Ihre Berücksichtigung ist für ein adaptives WfMS unerläßlich, wenn es breit einsetzbar sein soll. Die Unterstützung benutzerdefinierter Änderungen würde sich in ACT ohnehin schwierig gestalten. Um eine WF-Instanz ändern zu können, muß ein Benutzer die zugrundeliegende Ablauflogik verstehen. Hierzu eignen sich deklarative Beschreibungen schlecht, was von den Autoren auch indirekt eingeräumt wird. Sie gehen davon aus, daß der Modellierer nicht direkt mit ihrem Formalismus konfrontiert wird, sondern daß ein von ihm in einer höheren Beschreibungssprache erstelltes WF-Modell in eine deklarative Darstellung übersetzt wird [BeKl99]. Problem ist dann, daß auf Instanzebene durchzuführende Änderungen nicht mehr ohne weiteres auf dieser semantisch höheren Ebene stattfinden können.

10.6.2.3 Planung und Ausführung von Entwicklungsprozessen in EPOS und ComoKit

EPOS [NgCo96, JaCo93] ist eine Umgebung zur Unterstützung von Softwareentwicklungsprozessen (SEP). Ein solches Modell umfaßt mehrere Teilmodelle, von denen das Produkt- und Aufgabenmodell für die nachfolgende Diskussion relevant sind. Das Produktmodell definiert die Modulstruktur der zu entwickelnden Software, legt also die zu erstellenden Programmdateien und deren Beziehungen (z.B. „main.c includes m.h“) fest. Das Aufgabenmodell definiert den eigentlichen WF in Form eines Netzplans. Es besteht aus einer Menge von Aufgaben (z.B. „Codiere main.c“, „Übersetze m.c“, „Binde main.o und m.o“) sowie den zwischen ihnen bestehenden Kontroll- und Datenabhängigkeiten. Aufgabennetze müssen nicht explizit modelliert, sondern können mit Hilfe eines nichtlinearen, hierarchischen Planers automatisch hergeleitet werden [CLH95, LiCo93]. Zu diesem Zweck muß der Entwerfer das zu realisierende Produktmodell und die für die Netzkonstruktion zulässigen Aufgabentypen vorgeben. Hierzu bedient er sich einer deklarativen Beschreibungssprache [LiCo93], mit der sich für Aufgabentypen (z.B. „Übersetze C-Programm“) ihre Vor-/Nachbedingungen (z.B. „C-Quelldatei codiert“; „Objekt-Code generiert“) sowie ihre Ein-/Ausgabedokumente definieren lassen. Um den Planer anwenden zu können, wird diese Informationen in eine für ihn verständliche Darstellungsform (Ausgangs-/Zielsituation, Menge zulässiger Operatorschemata) transformiert. Die Zielsituation ergibt sich direkt aus dem Produktmodell. Aufgabentypen werden auf Operatorschemata abgebildet, wobei sich die Vorbedingungen eines Operators aus den Vorbedingungen und Eingabeparametern des korrespondierenden Aufgabentyps ergeben. Ähnliches gilt in Bezug auf die Festlegung von Operatornachbedingungen. Im Anschluß an diese Schritte erfolgt der eigentliche Planungsprozeß, bei dem durch Zielregression und Dekomposition ein hierarchisches Aufgabennetz abgeleitet wird. Die Struktur der Aufgabennetze wird dabei durch die Typen und Anzahl der Objekte des Produktmodells determiniert.

Während der Netzausführung arbeitet die Steuerungskomponente eng mit dem Planer zusammen [CLH95, LiCo93]. Wird ein Platzhalter für eine komplexe Aufgabe erreicht, wird der Planer darüber informiert, so daß er die Aufgabe dynamisch durch ein Subnetz verfeinern kann, etwa abhängig von der vorliegenden Produktstruktur. Möglich sind auch benutzergesteuerte Modifikationen des Produktmodells. In einem solchen Fall wird ein Umplanungsprozeß initiiert, der für den SEP ein angepaßtes Aufgabennetz herleitet. Gegenüber dem alten Aufgabennetz können in diesem Netz neue Aufgaben hinzukommen oder bisherige Aufgaben wegfallen bzw. durch andere ersetzt werden. Eine solche Änderung kann auch zur Laufzeit erfolgen, wobei Inkonsistenzen nicht ausgeschlossen werden. Sind die von der Umplanung betroffenen Aufgabenknoten aktiv, wird die Anpassung des Netzes solange verzögert, bis sie abgeschlossen sind. Ein Ummodellierungsprozeß wird in EPOS als *Meta-Prozeß* bezeichnet. Er ist mit denselben Mitteln beschreibbar wie ein SEP.

Für EPOS gelten ähnliche Beschränkungen wie für ACT. An erster Stelle zu nennen ist die geringe Ausdrucksmächtigkeit des verwendeten Formalismus - es können lediglich sequentielle und parallele Aufgabenbearbeitungen abgebildet werden. Durch die starke Anwendungsbezogenheit kann in EPOS Domänenwissen gezielt zur (dynamischen) Evolution der Prozeßstruktur genutzt werden. Eine von der Produktstruktur unabhängige Definition und Abänderung von WFs (Aufgabennetzen), wie sie für adaptive WfMS benötigt wird, ist dagegen nicht möglich. Beispielsweise können Aufgabennetze vom Benutzer nicht direkt abgeändert werden.

Ein ähnlicher Ansatz wird von ComoKit verfolgt [DeMa96, DMP97]. Auch hier steht die Planung und Ausführung komplexer Entwicklungsprozesse im Vordergrund. Das ComoKit-System umfaßt zwei Komponenten, den ComoKit-Modeler [MDP97], der die Definition von Produkt- und Prozeßmodellen (keine Schleifen, keine bedingten Verzweigungen) unterstützt, und den ComoKit-Scheduler, der diese Modelle als Eingabe erhält und darauf basierend die Planung und Ausführung von Entwicklungsprozessen steuert [DMP97]. Dazu werden die zwischen den Modellkomponenten entstehenden Abhängigkeiten mit Hilfe einer Planungskomponente verwaltet. Sie gestattet die Abbildung voneinander abhängiger Entscheidungen mit Hilfe eines *Truth Maintenance Systems*. Ähnlich wie in EPOS können Prozeßmodelle dynamisch verfeinert bzw. erweitert werden.

10.6.2.4 Weitere Ansätze

In CPREF werden zur Erzeugung und Ausführung von Plänen ebenfalls Techniken des hierarchischen NLP angewendet [BeMy98, Myer98]. Dabei kann der Planer auf eine Bibliothek vordefinierter Operatoren und Pläne zurückgreifen. Einen großen Raum nimmt die Untersuchung der Fragestellung ein, wie sich Laufzeitanpassungen eines Plans mit möglichst geringen Störungen erzielen lassen. Dabei geht CPREF ähnlich wie ACT vor. Allerdings sollen auch Benutzereingriffe in die WF-Kontrolle möglich sein, um auf unvorhersehbare Situationen angemessen reagieren zu können. Im Projekt IWCM [BeMy98] werden diese Konzepte aufgegriffen, um ein adaptives WfMS zu entwickeln. Veröffentlichungen hierzu gibt es bisher nicht. Zu erwähnen sind noch Ansätze, bei denen mit Hilfe von Planern neue Prozesse durch Synthese vordefinierter Prozeßfragmente erzeugt werden [WMLW95].

Eine Diskussion weiterer KI-Techniken und ihren Perspektiven für ein adaptives Prozeßmanagement findet man in [BEMM98, KlDe98, LSMK98, TLA91, ViSo91].

10.6.3 Zusammenfassende Bewertung

Die Anwendung von Planungsverfahren zur Modellierung und Adaption von WFs bietet interessante Perspektiven. Der Entwurf wird zur Planungsaufgabe und die Behandlung von Ausnahmen zum Umplanungsproblem. Vielversprechend ist die Idee, WF-Modelle mit mehr semantischen Informationen anzureichern. Dadurch lassen sich Erklärungskomponenten realisieren, die Anwendern die Auswirkungen semantischer Fehler und die zu ihrer Behandlung notwendigen Aktionen adäquat vermitteln.

Trotz dieser Vorteile darf nicht übersehen werden, daß KI-Planer ursprünglich für Anwendungsbereiche entwickelt wurden, in denen die Steuerung von Geräten (z.B. Robotern) und die Kontrolle vollautomatisierter Prozesse im Mittelpunkt stehen. Im Gegensatz zu WF-basierten Anwendungen wird hier ein Agent, der wenig oder gar kein Wissen über seine Umwelt besitzt, kontrolliert. Für solche Anwendungen ist es relativ einfach, die Startsituation (z.B. Standort eines Roboters) und die zu erreichenden Ziele (z.B. Bewegen des Roboters an einen Zielort) formal zu beschreiben und – basierend auf einer überschaubaren Grundmenge an Operatoren (z.B. Aktionen des Roboters) – einen korrekten und vollständigen Plan herzuleiten.

Wie in diesem Abschnitt diskutiert, ist die Übertragung dieser Konzepte auf den WF-Bereich nur teilweise möglich, da nicht immer alle relevante Prozeßziele formalisierbar sind. Aber selbst wenn dem so wäre, ist die automatische Herleitung von WF-Modellen, mit Ausnahme produktorientierter Workflows (vgl. Abschnitt 10.6.2.3), kein realistisches Ziel. In großen Anwendungsumgebungen gibt es oftmals Dutzende bis Hunderte von Aktivitätenvorlagen, auf deren Grundlage die Bildung und Abänderung von WF-Modellen erfolgen muß. Angesichts des kombinatorisch großen Suchraums beim Planen und der hohen Komplexität der Verfahren stößt man hier schnell an Grenzen. Abgesehen davon spiegelt eine Bottom-Up-Vorgehensweise, wie sie von planbasierten Ansätzen verfolgt wird, die betriebliche Realität nicht immer wider. Üblicherweise geht der Implementierung WF-basierter An-

wendungen eine umfassende Analyse und Optimierung der zu unterstützenden Prozesse voraus. Hieraus resultieren semantische Prozeßmodelle, die dann als Grundlage für den WF-Entwurf dienen. Der ADEPT-Ansatz unterstützt diese Top-Down-Vorgehensweise und bietet darüber hinaus ein ausdrucksstarkes Metamodell. Ob sich Planungssysteme als Steuerungskomponente für Workflows eignen ist ebenfalls fraglich. Ein WfMS ist ein hochgradig komplexes System, das im Produktionsbetrieb die verschiedenartigsten Aufgaben, von denen Planungssysteme nur einen sehr kleinen Teil abdecken, bewerkstelligen muß. Zu bemängeln sind auch die unzureichenden Möglichkeiten, den Planungsprozeß durch Vorgaben und Änderungen der Benutzer interaktiv zu steuern.

10.7 Vergleich mit transaktionalen WF-Ausführungsmodellen

Die Robustheit heutiger (funktionsorientierter) Informationssysteme basiert in erster Linie darauf, daß sie ein DBMS und dessen Transaktionsunterstützung nutzen. Dabei muß sich der Anwendungsentwickler nicht um systemnahe Details kümmern, sondern lediglich um die geeignete Strukturierung der DB-Transaktionen. Deren Atomaritätseigenschaft stellt dann sicher, daß auch im Fall von Programm- oder Systemfehlern ein konsistenter Systemzustand gewahrt bleibt. Diese Alles-oder-Nichts-Semantik von DB-Transaktionen ist für prozeßorientierte Anwendungssysteme nicht immer sinnvoll, da hier lange Laufzeiten, sowohl der Prozesse als auch der einzelnen Arbeitsschritte, vorliegen können [Wäch96, WoSh96]. Statt dessen muß für den Fall von Aktivitätenfehlschlägen die Möglichkeit zur Korrektur von Arbeit und zur erneuten Durchführung von Teilen des Prozesses gegeben sein [Leym97]. Hierzu sind Mechanismen für das (semantische) Zurücksetzen der WF-Kontrolle und die Wiederherstellung früherer WF-Zustände erforderlich. Ein WfMS, das als Implementierungsplattform für prozeßorientierte Anwendungen dienen soll, muß deshalb fortschrittlichere Transaktionskonzepte anbieten, als dies heutige DBMS tun. In den letzten Jahren wurde hierzu ein Reihe *erweiterter Transaktionsmodelle* (ETM) entwickelt [Alon96, Elma92, Hsu93, JaKe97].

10.7.1 Erweiterte Transaktionsmodelle und transaktionale Workflows

Ein bekanntes Nicht-Standard-Transaktionsmodell sind *Sagas* [GaSa87]. Vereinfacht ausgedrückt wird bei diesem Modell eine langlaufende Transaktion in mehrere Teiltransaktionen mit ACID-Eigenschaften aufgespaltet, deren Änderungen für andere Transaktionen vorzeitig, d.h. vor dem Ende der Gesamttransaktion, sichtbar gemacht werden. Aus diesem Grund ist ein gewöhnliches Transaktions-Rollback, wie man es in DBMS vorfindet, nicht anwendbar. Statt dessen muß für jede Teiltransaktion T_i (z.B. „Termin buchen“) eine geeignete Kompensationstransaktion C_i (z.B. „Termin stornieren“) definiert werden, durch deren Ausführung die von T_i bewirkten Änderungen im Fehlerfall semantisch rückgängig gemacht werden können. Eine Saga ist dann eine Sequenz solcher Paare (T_1, C_1), ..., (T_n, C_n). Tritt während der Ausführung kein Fehler auf, so werden die Teiltransaktionen T_1, ..., T_n in der angegebenen Reihenfolge ausgeführt. Schlägt dagegen die Bearbeitung einer Teiltransaktion T_x fehl, werden die Saga abgebrochen und die bereits beendeten Transaktionen T_1, T_2, ..., T_{x-1} durch die inverse Ausführung ihrer Kompensationstransaktionen C_{x-1}, ..., C_1 automatisch kompensiert. Da die Transaktion T_x zum Abbruchzeitpunkt noch nicht beendet war, erfolgt für sie ein gewöhnliches Rollback.

Weitere prominente ETM-Vertreter sind geschachtelte Transaktionen [Elma92], Contracts [ReSc95, WäRe92] und flexible Transaktionen [ANRS92]. Eine Übersicht zu diesen Modellen, bei denen - vereinfacht ausgedrückt - die Wiederherstellbarkeit operationaler Daten im Vordergrund steht, findet man in [Leym97, WoSh96]. Die Anwendungsprogrammierung gestaltet sich bei ETM relativ aufwendig, da für jeden Schritt einer langdauernden Transaktion ein Kompensationsschritt implementiert werden muß. Dies stellt im allgemeinen eine nicht-triviale Aufgabe dar, da zwischen dem Ende einer Teiltransaktion T_i und der evtl. Ausführung ihrer Kompensationstransaktion C_i ein längerer Zeitraum verstreichen kann, in dem die von T_i bewirkten Änderungen auch von anderen Transaktion gelesen werden können. Hinzu kommt, daß die Art der Kompensation eines Schrittes von verschiedenen Faktoren (z.B. Zeit, WF-Status) abhängig ist, die sich während der Transaktionsausführung ändern können.

Das Konzept der *transaktionalen WFs* [RuSh94, WoSh97] fußt darauf, Kontrollflüsse in deklarativer Form, durch die Angabe von Commit-/Abort-Beziehungen zwischen Aktivitäten, zu beschreiben. Diese systemtechnische Sicht hat verschiedene Weiterentwicklungen in Form prozeßtechnischer Transaktionskonzepte erfahren. Durch das *partielle Zurücksetzen* der WF-Bearbeitung soll beim Fehlschlag

einer Aktivität nicht die Arbeit der gesamten WF-Instanz ungeschehen gemacht werden, sondern nur Teile davon. Ziel ist es, den WF in einen konsistenten Zustand zu versetzen, von dem aus mit der Bearbeitung fortgefahren werden kann. Dabei müssen nicht nur planbare Aktivitätenfehlschläge behandelbar sein, sondern Benutzer sollten ggf. auch spontan in die Kontrolle einer WF-Instanz eingreifen können [TaHw98]. Das ADEPT-Modell unterstützt sowohl planbare als auch spontane Rücksetzoperationen. Wichtige Aspekte sind die korrekte Anpassung der Markierungen des Ausführungsgraphen und die Verwirklichung komplexer Rücksprünge (z.B. in frühere Schleifeniterationen).

WAMO [EdLi95, EdLi96, EdLi98] basiert auf dem Modell der flexiblen Transaktionen [ANRS92]. Für WF-Aktivitäten kann der Modellierer Attribute festlegen, auf deren Grundlage das WfMS automatisch geeignete Fehlerbehandlungsstrategien auswählt. Für den Umgang mit Aktivitätenfehlschlägen werden einfache Mechanismen bereitgestellt, etwa für das Überspringen, Wiederholen oder Zurücksetzen der Aktivitätenbearbeitung oder für die Durchführung alternativer Schritte. In [Leym95, LeRo97] wird ein Ansatz beschrieben, bei dem Aktivitäten in Kompensationssphären gruppiert werden. Schlägt die Bearbeitung einer Sphärenaktivität fehl, werden alle Aktivitäten der Sphäre einzeln kompensiert oder durch eine gemeinsame Kompensationstransaktion zurückgesetzt. Die Definition der Sphären stellt eine nicht-triviale Aufgabe dar, da das Kompensationsverhalten abhängig vom Zustand des zurückzusetzenden WF, von zeitlichen Faktoren oder anderen Aspekten sein kann.

Die skizzierten ETM lassen Änderungen in der Definition einer sich in Ausführung befindlichen Transaktion nicht zu. Darüber hinaus verwenden sie eine sehr enge Definition des Begriffs „Ausnahme“, bei der parametrisierbare Möglichkeiten zur Behandlung von Aktivitätenfehlschlägen im Vordergrund stehen. Ad-hoc-Eingriffe in die Kontrolle von Workflows spielen dagegen keine oder nur eine untergeordnete Rolle [TaHw98]. Um wirklich auf breiter Basis einsetzbar zu sein, müssen für adaptive WfMS geeignete Konzepte für die Beschreibung von Kompensationssphären und für die Implementierung von Kompensationsprogrammen gefunden werden. Darüber hinaus werden ETM benötigt, die nicht nur für den statischen, sondern auch für den dynamischen Fall funktionieren. Dies schließt die Unterstützung temporärer Änderungen und ihre Rücknahme im Fehlerfall ein [ReDa98].

10.7.2 Restrukturierung von Workflows in TAM

TAM ist ein transaktionales Ausführungsmodell, bei dem laufende Workflows dynamisch restrukturiert werden können [LiPu98]. Das TAM-Ausführungsmodell weist Ähnlichkeiten zum Modell der offen geschachtelten Transaktionen [Elma92] auf. Ein WF besteht aus einer Menge von Aktivitäten, die hierarchisch in einem Baum angeordnet werden. Die Blätter dieses Baums entsprechen gewöhnlichen ACID-Transaktionen, während die Nichtblattknoten komplexe Transaktionen beschreiben.

In TAM können sowohl für Aktivitäten derselben als auch für Aktivitäten verschiedener Hierarchiestufen Kontrollabhängigkeiten deklarativ beschrieben werden. Hierzu stehen dem Entwerfer die Relationen *precede*, *compatible*, *enable* und *disable* zur Verfügung: *A precede B* legt fest, daß die Aktivität *B* erst nach dem *Commit* von Aktivität *A* starten kann. Sollen die beiden Aktivitäten *A* und *B* dagegen parallel bearbeitet werden, kann dies mittels der Relation *A compatible B* zum Ausdruck gebracht werden. Komplexere Commit-/Abort-Beziehungen sind mit den Relationen *enable* und *disable* festlegbar. Ähnlich wie bei Sagas [GaSa87] erfolgt der *Commit* einer Kind-Aktivität unabhängig vom Status der übergeordneten Vater-Aktivität. Falls eine Vater-Aktivität abgebrochen wird, werden alle zugehörigen Kind-Aktivitäten, die sich aktuell in Bearbeitung befinden, ebenfalls abgebrochen. Für bereits beendete Kind-Aktivitäten (d.h. Kind-Aktivitäten im Status *commited*) wird eine Kompensationstransaktion ausgeführt (sog. „*termination-sensitive dependency*“; kurz: TSD).

Im Zentrum von TAM steht eine Menge von Split- und Join-Operationen, durch deren Anwendung WF-Instanzen dynamisch modifiziert werden können. In Bezug auf Split-Operationen bietet TAM den Serial-, Parallel- und Unnesting-Split an. Mittels einem *Serial-Split* kann eine (u.U. bereits laufende) Aktivität A in zwei aufeinanderfolgende Aktivitäten A_1 und A_2 aufgesplittet werden, wobei unterschiedliche Kontrollabhängigkeiten für A_1 und A_2 möglich sind. Beispielsweise kann der Anwender angeben, daß A_2 erst nach dem Commit von A_1 aktiviert (d.h. A_1 *precede* A_2) werden darf oder daß A_2 nur ausgeführt werden soll, wenn A_1 abgebrochen wird (A_1 *precede* $A_2 \wedge$ *abort*(A_1) *enable* A_2). Ein serieller Split ist nützlich, wenn die von einer laufenden Aktivität beanspruchten Ressourcen, die für die weitere Bearbeitung nicht mehr benötigt werden, vorzeitig für andere Aktivitäten bzw. „Transaktio-

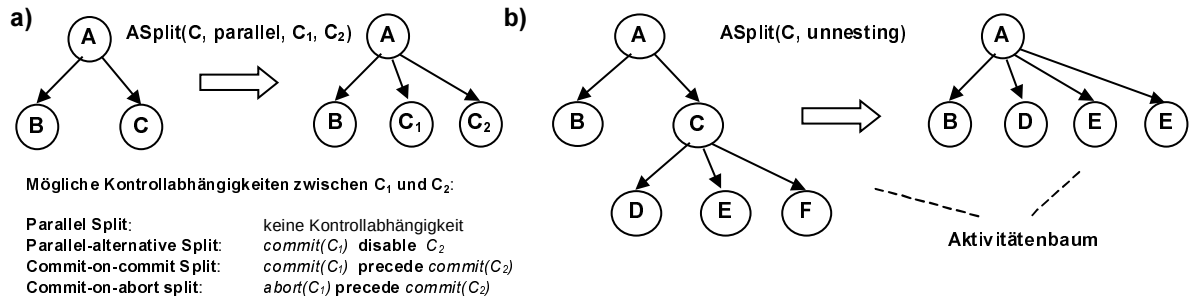


Abbildung 10-22: *(a) Parallel Split und (b) Unnesting Split*

nen“ freigegeben werden sollen. Ein paralleler Split (vgl. Abbildung 10-22 a) ermöglicht die Zerlegung einer Aktivität *B* in zwei parallele Aktivitäten B_1 und B_2 (B_1 *compatible* B_2). Dadurch kann z.B. die Bearbeitung einer Aufgabe dynamisch auf mehrere Bearbeiter aufgeteilt werden. Während diese beiden Operationen auf elementare Aktivitäten anwendbar sind, läßt sich mit Hilfe der *Unnesting-Split-Operation* (vgl. Abbildung 10-22 b) ein Nichtblattknoten des Aktivitätenbaums auflösen, so daß sich seine Kind-Aktivitäten in der Hierarchie des Aktivitätenbaums eine Ebene nach oben bewegen. Die Anwendung einer Split-Operation auf eine WF-Instanz führt zur Restrukturierung ihres Aktivitätenbaums und zur Anpassung der darauf definierten Menge von Kontrollabhängigkeiten. TAM gestattet nur solche Split-Operationen, die das oben erwähnte TSD-Kriterium aufrechterhalten. Darüber hinaus müssen bestehende Kontrollabhängigkeiten zwischen Aktivitäten semantisch erhalten bleiben und es dürfen keinerlei Konflikte (z.B. syntaktische Fehler) entstehen. Statusvoraussetzungen für die Anwendung von Split-Operationen werden nicht genannt. Ähnliches gilt in Bezug auf die angebotenen Join-Operatoren, mit denen sich Aktivitäten auf unterschiedliche Art zusammenfassen lassen.

Die deklarative Beschreibung von Kontrollabhängigkeiten, wie sie in TAM erfolgt, mag für die Spezifikation transaktionaler Ausführungsmodelle ausreichend sein. Sie ist es nicht, wenn es um die Beschreibung von WF-Modellen geht, die zur Laufzeit von Anwendern spontan abgeändert werden können sollen. Hier bedarf es strukturierter und verständlicher Darstellungsformen. Fraglich ist, ob sich in Änderungen, wie das Einfügen, Löschen oder Verschieben von Aktivitäten, mit den TAM-Operationen verwirklichen lassen. Auch in Bezug auf die Ausdrucksmächtigkeit des verwendeten Formalismus bleiben Fragen offen. Unklar ist, ob bedingte oder iterative Ausführungen modellierbar sind. Ferner geht aus [LiPu98] nicht hervor, ob in TAM auch Datenabhängigkeiten zwischen Aktivitäten oder andere WF-Aspekte beschreibbar sind. Dem entsprechend oberflächlich bleiben die im Zusammenhang mit der Anwendung der Split- und Join-Operationen diskutierten Korrektheitsaspekte.

10.8 Vergleich mit temporalen Logiken, Prozeßalgebren und -grammatiken

In diesem Abschnitt fassen wir einige weitere Formalismen zusammen, die in der Literatur für die Spezifikation und Adaption von Prozessen vorgeschlagen werden.

10.8.1 Temporale Logiken und Prozeßalgebren

Ein ausdrucksstarker Formalismus für die WF-Modellierung oder allgemeiner für die Spezifikation des dynamischen Verhaltens von Informationssystemen sind *temporale Logiken* [MaPn92]. Sie erweitern die Aussagen- bzw. Prädikatenlogik um temporale Operatoren, mit denen Bedingungen über zukünftige und vergangene Ereignisse formuliert werden können. Einfache Formen temporaler Logik wurden in der Vergangenheit bereits für die Spezifikation von Ausführungsabhängigkeiten in erweiterten Transaktionsmodellen verwendet [ASSR93, Elma92]. Vom Prinzip her eignen sie sich deshalb auch für die WF-Modellierung, insbesondere wenn es um die formale Verifikation von WF-Eigenschaften (z.B. mittels Theorembeweisern oder Modellprüfern) geht. Allerdings werden temporale Logiken den eingangs von Kapitel 3 diskutierten Anforderungen an WF-Beschreibungssprachen nicht gerecht. Ein Problem, das auftreten kann wenn die zugrundeliegende Logik zu ausdrucksmächtig ist, ist die unzureichende Ausführbarkeit logikbasierter Spezifikationen [Weik97]. Darüber hinaus leidet die Verständlichkeit und Wartbarkeit logikbasierter WF-Beschreibungen unter den fehlenden Mög-

lichkeiten zur Visualisierung bzw. Konvertierbarkeit in andere (graphische) Beschreibungsformalismen. Aus diesen Gründen eignen sich temporale Logiken nicht als Grundlage für adaptive WfMS.

Ein anderer Formalismus für die Spezifikation von WF-Modellen sind *Prozeßalgebren* [Henn89, Part98]. Neben Elementen zur Modellierung sequentieller und paralleler Prozesse bieten sie einen Regelapparat zur Verifikation von Modellen an. Implementierungsaspekte spielen, wenn überhaupt, nur ein untergeordnete Rolle. Trotz ihrer theoretischen Fundierung haben Prozeßalgebren im WF-Bereich bisher keine Beachtung gefunden. Im Kern bestehen ähnliche Probleme wie bei logikbasierten Methoden, insbesondere muß wieder in Frage gestellt werden, ob Anwender einen Zugang zu algebraischen Prozeßdarstellungen finden. Logikbasierte Ansätze sind den Prozeßalgebren zumindest in puncto automatische Ausführbarkeit und Verifizierbarkeit der Prozeßmodelle überlegen [Weik97].

Die verstärkte Integration streng formaler Techniken in WfMS ermöglicht präzise Aussagen über das Systemverhalten und einen WF-Entwurfsprozeß auf einer semantisch hohen Ebene. Dennoch muß bezweifelt werden, daß sich diese Formalismen auf den WF-Bereich anwenden lassen. Schwierigkeiten ergeben sich spätestens dann, wenn Endbenutzeraspekte oder operative Aspekte (z.B. Verklemmungfreiheit, kontrolliertes Zurücksetzen der WF-Kontrolle) berücksichtigt werden müssen.

10.8.2 Prozeßgrammatiken

Eine Alternative zu herkömmlichen Prozeßbeschreibungssprachen bilden *Prozeßgrammatiken* [Pent94]. Ein deklarativer, grammatikalischer Ansatz für die Prozeßspezifikation wird in [Pent94] vorgestellt. Das zugrundeliegende grammatikalische Modell umfaßt ein Lexikon elementarer Aktivitäten und eine Menge von Regeln für deren Verknüpfung zu komplexen Workflows. Durch ein solches Modell wird nicht ein einzelner WF-Typ spezifiziert, sondern eine ganze Klasse von Workflows. Ihre Instanzen sind aus dem Lexikon und den spezifizierten Regeln ableitbar. Durch die dynamische Veränderung der Regelmenge während der WF-Ausführung können laufende WF-Instanzen an unvorhersehbare Situationen angepaßt werden.

Auch im System *Zippin* [GPP96] werden Prozesse durch Grammatiken beschrieben. Während der WF-Ausführung wird inkrementell ein Ableitungsbaum zur Grammatik konstruiert, der die aktuelle Struktur und den Ablauf der WF-Instanz widerspiegelt. Um strukturelle Änderungen vorzunehmen, kann die Grammatik dynamisch abgeändert werden, was zur Folge hat, daß der Ableitungsbaum mittels Backtracking an die neue Situation angepaßt wird. Zu erwähnen bleibt, daß sich mit ihnen nicht nur bekannte Prozesse abbilden, sondern auch neue Prozeßmodelle gewinnen bzw. entwerfen lassen. Der zuletzt genannte Aspekt wird zum Beispiel in [Malo99] genutzt, um ein Handbuch für Geschäftsprozesse zu beschreiben. Dieses Handbuch soll die Grundlage für die Analyse der Organisation, für die Restrukturierung betrieblicher Prozesse, für die Ableitung neuer Prozeßmodelle, und für die automatische Generierung von Aktivitätenprogrammen dienen.

10.9 Vergleich mit sonstigen Ansätzen

10.9.1 Deskriptive Workflow-Modellierung in MOBILE

In MOBILE [BuJa96, JBS97] können WFs mittels einer *skriptbasierten* Sprache definiert werden. Als Hauptgrund für die Inflexibilität von WfMS wird die *präskriptive* WF-Modellierung gesehen. Präskriptiv bedeutet, daß der WF zur Entwicklungszeit in allen Details festgelegt sein muß und daß zur Laufzeit von dieser Spezifikation nicht abgewichen werden darf [HSS96]. Als sinnvolle Ergänzung hierzu wird das Konzept des *deskriptiven WF* vorgeschlagen, bei dem nicht alle Details a priori feststehen müssen, sondern lediglich die Eigenschaften, die für alle WF-Instanzen gelten sollen. Beispielsweise kann die Menge zulässiger Aktivitätentypen vorgegeben werden, ohne dem Benutzer vorzuschreiben, in welcher Reihenfolge, unter welchen Bedingungen oder wie häufig Aktivitäten dieses Typs zur Ausführung kommen sollen. Dies setzt voraus, daß die Aktivitätenprogramme so gekapselt und autonom sind, daß sie in beliebiger Reihenfolge aufrufbar sind – eine Annahme die für vorgangsorientierte Anwendungen jedoch unrealistisch ist.

MOBILE bietet zumindest die Möglichkeit, die Kontroll- und Datenabhängigkeiten zwischen WF-Aktivitäten partiell vorzumodellieren und sie zur Laufzeit durch den Benutzer vervollständigen zu lassen. Des weiteren können deskriptive WF-Modelle an definierten Stellen eines präskriptiven WF-

Modells als Sub-WF eingebunden werden. Technische Details hierzu wurden bisher nicht veröffentlicht. Auch Hinweise auf Implementierungen dieses Konzepts gibt es nicht. Interessant wäre, wie sich das zugrundeliegende Ausführungsmodell darstellt und wie zur Laufzeit mit möglichen Fehlersituationen (Verklemmungen, unversorgte Aktivitätenparameter, usw.) umgegangen wird. In späteren Veröffentlichungen [JST97, Hein98b, HoJa98] wird die Notwendigkeit dynamischer WF-Änderungen erkannt, allgemeingültige Konzepte zu ihrer Verwirklichung werden aber nicht behandelt. In [HoJa98] wird am Beispiel des Einfügens einer Aktivität exemplarisch gezeigt, in welchen Zuständen einer WF-Instanz eine solche Änderung zulässig ist und in welchen nicht - vergleichbare Erkenntnisse des ADEPT-Projekts wurden in sehr viel allgemeinerer Form bereits ein Jahr zuvor veröffentlicht [ReDa97]. In [Hein98b] wird zwischen „Flexibilität durch Selektion“ und „Flexibilität durch Adaption“ unterschieden, womit die Vormodellierung von Ausnahmen bzw. die dynamische Adaption von WFs gemeint sind.

10.9.2 Unterstützung von Design-Workflows in CONCORD

CONCORD [RiMi97, MHR96, Ritt94] realisiert ein Modell zur Unterstützung von Entwurfsprozessen. Neben der transaktionsgeschützten Verarbeitung von Entwurfsdaten werden verschiedene Mechanismen für die Steuerung von Entwicklungsprozessen und für die Kooperationskontrolle zwischen Entwicklungsaufgaben angeboten. Dabei wird zwischen der Kooperations-, der Entwurfskontroll- und der Werkzeugausführungsebene unterschieden.

Auf der *Kooperationsebene* wird eine Entwurfsaufgabe (*Design Activity*; kurz: DA) hierarchisch in Subaufgaben (Sub-DAs) zerlegt, die an verschiedene Akteure delegiert werden. Ein Akteur kann eine ihm zugewiesene Sub-DA entweder selbst bearbeiten oder weiter zerlegen, indem er seinerseits Sub-DAs an andere Akteure delegiert. Dadurch wird zur Ausführungszeit dynamisch ein Aktivitätenbaum aufgebaut, ähnlich wie in WAM [FaWi96], OOActS [Teeg96a], DWM [KwBa97] oder ComoKit [DeMa96]. Auf der *Entwurfskontrollebene* kann für jede komplexe DA ein Designflow definiert werden, der die Kontroll- und Datenflußbeziehungen ihrer Sub-DAs festlegt [RiMi97]. Modellierbar sind Sequenzen, parallele und alternative Verzweigungen. Darüber hinaus kann eine Aktivitätenausführung zur Laufzeit solange iteriert werden, bis ihre Nachbedingungen erfüllt sind. Allgemeine Schleifenstrukturen sind nicht abbildbar. Zur Modellierung von Datenflüssen wird ein ähnlicher Ansatz wie in FlowMark (vgl. Kapitel 2) verfolgt [RiMi97]. Elementare DAs, die nicht weiter verfeinert werden sollen, können mit einer Design-Operation verknüpft werden. Für ihre Bearbeitung wird auf der *Werkzeugausführungsebene* ein Programm gerufen.

Der Designflow einer DA kann zur Modellierungs- oder zur Laufzeit definiert werden. Im ersten Fall ist der Ablauf vorgegeben und kann während der Ausführung nicht abgeändert werden. Im zweiten Fall wird er zur Laufzeit schrittweise, durch Aneinanderreihen von Sub-DAs, festgelegt. Durch eine solche Ad-hoc-Definition lassen sich sequentielle WFs abbilden. Komplexere Änderungen werden aber nicht unterstützt. Es besteht aber die Möglichkeit, bei nicht erfolgreicher Bearbeitung einer Aktivität, den Kontrollfluß bis zu einem bestimmten Punkt zurückzusetzen und anschließend einen alternativen Weg oder den gleichen Weg mit einer anderen Parameterbelegung erneut zu durchlaufen. Beim Zurücksetzen werden frühere Versionen von Entwurfsobjekten wiederhergestellt. Eine Stärke von CONCORD sind die angebotenen Mechanismen zur Kontrolle der Kooperation zwischen den DAs eines Aktivitätenbaums, die auf der gemeinsamem Bearbeitung von Entwurfsdaten basieren. Es werden Synchronisationsverfahren für die transaktionsgeschützte Verarbeitung von Entwurfsdaten bereitgestellt, wobei ein speziell auf die Verwaltung versionierter, komplex strukturierter Entwurfsdaten zugeschnittenes und Isolation gewährleistendes Sperrprotokoll dafür sorgt, daß diese Daten bei nebenläufigen Werkzeuganwendungen korrekt verarbeitet werden. Darüber hinaus besteht die Möglichkeit, Zwischenergebnisse einer DA vor ihrer Beendigung an andere DAs weiterzugeben [RiMi97].

10.9.3 Tolerieren von Prozeßabweichungen in SENTINEL und PROSYT

Wie in dieser Arbeit gezeigt, lassen sich viele Ausnahmen, die während der Bearbeitung einer WF-Instanz auftreten können, durch die dynamische Restrukturierung ihres Ausführungsgraphen behandeln.

Das von SENTINEL [CNGM95] unterstützte Konzept der „Abweichungen“ fußt auf einem anderen Konzept, das einen vordefinierten WF nicht erzwingt, sondern Inkonsistenzen zwischen der Modell-

und Prozeßausführung zuläßt und verwaltet, um später eine Rückführung auf einen definierten Ausführungspfad zu ermöglichen.

Eine Weiterentwicklung dieses Ansatzes erfolgte in PROSYT [Cugo98]. Auch hier werden Abweichungen vom vordefinierten WF toleriert und der Benutzer darin unterstützt, die Modellausführung wieder in Einklang mit der realen Prozeßausführung zu bringen. Im Gegensatz zu ADEPT orientiert sich bei PROSYT die Modellbildung nicht an den Aktivitäten, sondern an den Objekten (z.B. Dokumente), die im Verlauf der WF-Ausführung erzeugt und bearbeitet werden [Cugo98]. Jedes Objekt wird charakterisiert durch eine Menge von Attributen und Operationen, die logisch WF-Variablen und -Aktivitäten entsprechen. Durch die Festlegung geeigneter Vorbedingungen für ihre Ausführung (i.w. Bool'sche Ausdrücke auf Attributen) lassen sich Kontrollflüsse modellieren. Darüber hinaus können Invarianten vereinbart werden, die gültige Objektzustände festlegen. Während der WF-Ausführung können Objekte erzeugt und die mit ihnen verknüpften Operationen aufgerufen werden. Dabei können Akteure von den vordefinierten Aufrufreihenfolgen abweichen. Um solche Abweichungen kontrolliert durchführen zu können, werden verschiedene Strategien angeboten, die sowohl benutzer- als auch statusabhängig zur Anwendung kommen. Soll eine Operation, deren Vorbedingungen (noch) nicht erfüllt sind, vorzeitig ausgeführt werden, kann das System diese Abweichung entweder tolerieren oder aber es weist sie zurück. Falls die Änderung akzeptiert wird, muß überprüft werden, ob dadurch für das Modell festgelegte Invarianten verletzt werden. Für diesen Fall werden unterschiedliche Behandlungsformen unterstützt. Besitzt der Erhalt der Konsistenz oberste Priorität, kann die Prozeßausführung (partiell) unterbrochen werden, um notwendige Anpassungen vornehmen zu können. Alternativ besteht die Möglichkeit, die Prozeßausführung trotz der aufgetretenen Inkonsistenz fortzusetzen, was in der Folge allerdings zu Fehlersituationen führen kann.

Kontroll- und Datenflüsse können in PROSYT nur implizit und deklarativ beschrieben werden, ohne daß irgendwelche Korrektheitsbedingungen bestehen. Dadurch ergeben sich für die Wartung und Pflege der Modelle de facto ähnliche Probleme wie bei regelbasierten Beschreibungen.

10.10 Zusammenfassung der Diskussion

Die meisten der in diesem Kapitel diskutierten Ansätze unterstützen spezielle Formen dynamischer WF-Änderungen. Sie sind aus mehreren Gründen nicht als Grundlage für adaptive WfMS geeignet. Eine WF-Änderung erfolgt in der Mehrzahl der Fälle dadurch, daß ein Akteur die Ausführung einer laufenden WF-Instanz unterbricht und ihren Ausführungsgraphen mit Hilfe eines (graphischen) Editors modifiziert. Andere Benutzerschnittstellen oder Schnittstellen für die Entwicklung eigener Klientenanwendungen (mit entsprechenden Manipulationsmöglichkeiten), wie sie vom ADEPT-WfMS angeboten werden, gibt es nicht. Während der Modifikation eines WF-Graphen dürfen keine anderen Änderungen stattfinden, oftmals ist auch das Fortschreiten der WF-Kontrolle nicht möglich. Für die Festlegung der gewünschten Modifikationen werden für den Anwender, bis auf wenige Ausnahmen, nur einfache Änderungsprimitve angeboten, etwa zur Manipulation der Knoten und Kanten eines WF-Graphen. Semantisch höhere Änderungsoperationen, wie sie $ADEPT_{flex}$ bereitstellt, sucht man dagegen vergeblich. Dasselbe gilt im Hinblick auf fortschrittliche Funktionen, wie die Unterstützung temporärer Änderungen. Abgesehen davon sind fast ausschließlich Modifikationen der Kontrollflußstruktur und des Status möglich, wohingegen Anpassungen anderer WF-Aspekte ausgeklammert bleiben.

Nur wenige der diskutierten Ansätze verfügen über ein formales Modell, das die Semantik von Änderungsoperationen sowie ihre Vor- und Nachbedingungen (z.B. hinsichtlich WF-Status) präzise festlegt. Ein Grund hierfür ist oftmals die geringe theoretische Fundierung der verwendeten WF-Metamodelle. So werden bei vielen Systemen keine ausreichenden Bedingungen für die korrekte Modellierung von Kontroll- und Datenflußschemata genannt. Des weiteren gibt es kein formales Ausführungsmodell und dem entsprechend auch keine Kriterien für das korrekte Verhalten von WFs, etwa zur Unterscheidung korrekter WF-Endzustände von Verklemmungssituationen. Da diese Aspekte bereits bei der WF-Modellierung unberücksichtigt bleiben, können sie bei dynamischen WF-Änderungen systemseitig nicht behandelt werden. Es bleibt somit die Aufgabe des Anwenders, zu erkennen, ob eine beabsichtigte Modifikation zu Problemen führt und – falls ja – wie diese zu behandeln sind. Leider wird dabei nicht bedacht, daß Ad-hoc-Änderungen in der Mehrzahl der Fälle nicht von einem EDV-Experten, sondern von Endanwendern vorgenommen werden können müssen. Der zuletzt genannte Aspekt trifft vor allem auch auf Beschreibungsansätze wie Petri-Netze oder Statecharts zu.

Einerseits sind diese Formalismen sehr ausdrucksstark, d.h. sie gestatten die Abbildung eines breiten Spektrums von Prozessen, andererseits bieten sie nur unzureichende Strukturierungsmöglichkeiten an (z.B. keine speziellen Konstrukte zur Modellierung von Schleifen- oder Fehlerrücksprüngen). Dem entsprechend unübersichtlich und schwer verständlich präsentieren sich die resultierenden WF-Modelle dem Anwender gegenüber. Die angebotenen Möglichkeiten zur Analyse des dynamischen Verhaltens sind aufgrund der umfangreichen Modelle dann oftmals nur von theoretischer Natur.

Schließlich gibt es in der Literatur einige wenige formale Ansätze für adaptive, vorgangsorientierte Anwendungen. Sie basieren größtenteils auf speziellen Methoden, wie Graphgrammatiken oder KI-Planern, die aus verschiedenen Gründen als allgemeine Grundlage für adaptive WfMS ungeeignet erscheinen. Auffällig bei Systemen wie ACT, EPOS oder DYNAMITE ist vor allem die Ausdrucksschwäche der verwendeten Formalismen (z.B. Fehlen von Schleifen und Fehlerrücksprüngen, keine bedingten Verzweigungen) und die Beschränkung auf bestimmte Arten von Anwendungen (z.B. evolutionäre Prozesse) oder Änderungen. Abschließend sei bemerkt, daß es für viele der in der Literatur diskutierten Ansätze keine Hinweise auf (prototypische) Implementierungen gibt.

11 Zusammenfassung

Durch das rasante Wachstum des elektronischen Handels im Internet, durch die zunehmende Globalisierung, durch neue Formen der Kooperation zwischen Unternehmen und vieles mehr, wird die Unterstützung unternehmensweiter und -übergreifender Geschäftsprozesse in Zukunft immer mehr an Bedeutung gewinnen [DaRe99]. Dabei müssen die vor Ort eingesetzten Informationssysteme prozeßorientiert integriert und die Anwender verschiedener Bereiche oder Unternehmen bei der Durchführung einer großen Zahl von Prozeßinstanzen (*Production Workflows*) unterstützt werden. Wie eingangs dieser Arbeit gezeigt, würde die Realisierung derartiger Anwendungssysteme mit Hilfe konventioneller Implementierungstechniken schnell zu komplexen und nicht mehr wartbaren Anwendungssystemen führen. Darüber hinaus wären sehr systemnahe Kenntnisse notwendig, über die normale Anwendungsprogrammierer üblicherweise nicht verfügen. Aus diesen Gründen sind verteilte, vorgangsorientierte Anwendungen nur dann mit der erforderlichen Zuverlässigkeit und Robustheit implementierbar, wenn es eine „intelligente“ Systemschicht gibt, die die zu unterstützenden Prozesse kennt und die deshalb wichtige Aufgaben der Prozeßsteuerung und -überwachung übernimmt.

WfMS besitzen das Potential, die Entwicklung vorgangsorientierter Anwendungen nachhaltig zu verändern. Faktisch wird durch ihren Einsatz die Realisierung und der Betrieb prozeßorientierter Anwendungssysteme im größeren Stil überhaupt erst möglich. Dabei sollten die einzelnen Anwendungsfunktionen eines WF idealerweise als isolierte, wiederverwendbare Komponenten implementiert werden können, deren Eingabeparameter beim Aufruf von der Laufzeitumgebung des WfMS versorgt werden und die lediglich dafür sorgen müssen, daß nach ihrer Beendigung korrekte Werte für Ausgabeparameter erzeugt werden. Alle anderen Aufgaben der Ablaufsteuerung und -überwachung sollen durch das WfMS übernommen werden. WF-Technologie bietet deshalb die Chance, zu einer neuen Art der Entwicklung verteilter Informationssysteme zu gelangen, bei der Anwendungen durch die (graphische) Beschreibung von Prozeßvorlagen und durch das Einstecken vorgefertigter Programmbausteine in diese Vorlagen entwickelt werden.

Dieses Potential wird von heutigen WfMS bisher nur sehr rudimentär umgesetzt. Zwar findet man die beschriebene Trennung von Ablauflogik und Anwendungscode in einigen prozeßorientierten WfMS vor, diese Systeme weisen aber derart gravierende Mängel auf, daß sie für einen breiteren Einsatz nicht in Frage kommen. Diese Mängel sind oftmals Folge der unzureichenden theoretische Fundierung, der zu geringen Ausdrucksmächtigkeit, der mangelhaften Strukturierung oder der schlechten Verständlichkeit der verwendeten WF-Metamodelle. Insbesondere fehlt dadurch eine formale Basis für die Verwirklichung fortschrittlicher Systemfunktionen. Dies gilt auch in Bezug auf die Aspekte Flexibilität und Adaptivität. Heutige WfMS erzwingen entweder die starre Ausführung des vormodellierten WF oder sie lassen Ad-hoc-Abweichungen nur unter erheblichen Einschränkungen (z.B. keine Schleifen, keine Anpassung des Datenflusses) zu. Da potentielle Laufzeitprobleme, wie Verklemmungssituationen oder der Aufruf von Aktivitätenprogrammen mit nicht versorgten Eingabeparametern, bei den meisten Systemen noch nicht einmal zur Modellierungszeit ausgeschlossen werden, überrascht es nicht, daß dynamische WF-Änderungen zur Laufzeit nicht zugelassen werden.

Das Anliegen dieser Arbeit ist es gewesen, ein durchgängiges formales WF-Metamodell für die Modellierung, die Ausführung und die (dynamische) Änderung von Workflows zu entwickeln und prototypisch zu implementieren. Dabei sollte ein zuverlässiges und robustes Ausführungsverhalten mit der Möglichkeit, zur Laufzeit ad hoc vom geplanten Ablauf abweichen zu können, verknüpft werden. Diese Ziele konnten erfolgreich verwirklicht werden. Die wichtigsten Ergebnisse der Arbeit, die das Gerüst des ADEPT-Ansatzes und des darauf basierenden ADEPT-WfMS bilden, lassen sich wie folgt zusammenfassen:

ADEPT-Basismodell

Das in dieser Arbeit entwickelte ADEPT-Basismodell bildet die formale Grundlage für die Modellierung, Steuerung und Änderung von Workflows. Es handelt sich hierbei um ein ausdrucksstarkes WF-Metamodell, das die integrierte Beschreibung verschiedener WF-Aspekte erlaubt. In der vorliegenden Arbeit haben wir uns auf funktionale Aspekte sowie auf die Modellierung von Kontroll- und Datenflüssen konzentriert. Darüber hinaus gibt es verschiedene Erweiterungen des ADEPT-Basismodells, mit denen sich auch andere WF-Aspekte, wie zeitliche Beschränkungen (z.B. minimale und maximale

Zeitabstände zwischen Aktivitäten) und organisatorische Regelungen (z.B. Bearbeiterzuordnungen für Aktivitäten, Vertretungen) angemessen abbilden lassen. Für sie gibt es, ebenso wie für die vorgestellten Modellaspekte, überprüfbare Regeln zur Sicherung der Korrektheit und Konsistenz der Workflow-Modelle.

Im Zentrum des für die *Kontrollflußmodellierung* gewählten Ansatzes steht das Konzept der regelmäßigen Blockstrukturierung, bei dem Kontrollstrukturen wie Sequenzen, Verzweigungen oder Schleifen auf logische, ineinander geschachtelte Kontrollblöcke mit eindeutigen Start- und Endknoten abgebildet werden. Dieser Strukturierungsansatz bietet zahlreiche Vorteile hinsichtlich der Modellierung, Analyse und Änderung von WF-Graphen. Aufgrund der Eindeutigkeit von Verzweigungs- und Schleifenstrukturen gestaltet sich nicht nur die WF-Modellierung übersichtlicher und fehlerfreier, sondern es wird dadurch erst möglich, semantisch höherwertige WF-Änderungen (z.B. Rücksetzen einzelner Ausführungszweige, Verschieben von Blöcken, Vorziehen von Aktivitäten) und fortschrittliche Änderungskonzepte (z.B. temporäre Änderungen) sinnvoll unterstützen zu können. Darüber hinaus erlaubt die gewählte Blockstruktur die effiziente Analyse von Kontroll- und Datenflüssen, was insbesondere im Kontext dynamischer Änderungen wichtig ist. Um eine ausreichende Ausdrucksmächtigkeit zu erzielen, bietet das Basismodell eine Reihe weiterer Beschreibungskonzepte an, für die es ebenfalls klar definierte Strukturierungsregeln gibt. Zu nennen sind Nullaktivitäten, verschiedene Kantentypen zur Synchronisation nebenläufiger Aktivitäten, zur Festlegung präferierter Ausführungsreihenfolgen oder zur Vormodellierung von Fehlerrücksprüngen sowie Konzepte zur Abbildung hierarchischer WFs. Auf Modellebene wird intern zwischen verschiedenen Knoten- und Kantentypen differenziert, wodurch sich weitere Vorteile ergeben. Beispielsweise können dem Modellierer unterschiedliche Sichten auf einen WF-Graphen eingeräumt werden. Des weiteren ist zur Ausführungszeit für das WfMS immer sofort erkennbar, wann es im Kontrollfluß zu einem Schleifen- oder Fehlerrücksprung kommt und wann nicht.

Grundlegend für die *Kontrollflußsteuerung* sind der gewählte Markierungsansatz sowie die darauf basierenden Ausführungs- und Markierungsregeln. Im Gegensatz zu gängigen WF-Ausführungsmodellen werden auch nicht mehr ausführbare bzw. wählbare Zweige markiert. Des weiteren bleiben die Markierungen beendeter Knoten und signalisierter Kanten im weiteren Verlauf der WF-Ausführung, mit Ausnahme von Rücksprüngen, erhalten. Beide Aspekte sind essentiell, um effizient überprüfen zu können, ob eine dynamische WF-Änderung im aktuellen Status einer WF-Instanz anwendbar ist oder nicht. Basierend auf diesem Ausführungsmodell haben wir formale Bedingungen für ein korrektes dynamisches Verhalten von Workflows (z.B. Terminierung, Verklemmungsfreiheit) formuliert. Wir haben gezeigt, wann die gewünschten Dynamik-Eigenschaften durch die Art der Konstruktion der Modelle sichergestellt werden können und wann zusätzliche Analysen durchzuführen sind. Hilfreich ist in diesem Zusammenhang, neben der gewählten Blockstruktur, die Unterscheidbarkeit zwischen verschiedenen Knoten- und Kantentypen.

Um einen fehlerfreien Ablauf zur Laufzeit zu erzielen und dynamische Änderungen und ihre Seiteneffekte gezielt behandeln zu können, unterstützt das ADEPT-Basismodell die explizite *Beschreibung von Datenflüssen*. In dieser Arbeit haben wir grundlegende Aspekte der Modellierung, Analyse und Validation von Datenflüssen in erweiterten, blockbasierten WF-Graphen vorgestellt (inkl. Analyseverfahren). Darauf basierend haben wir gezeigt, wie ihre Steuerung und das Datenkontext-Management zur Laufzeit erfolgen. Im Gegensatz zu existierenden Ansätzen, gibt es für die Datenflußmodellierung gewisse Regeln, die unter anderem garantieren sollen, daß zur Laufzeit keine Aktivitätenprogramme mit fehlenden Werten für obligate Eingabeparameter gerufen werden. Dies ist z.B. essentiell, um bei dynamischen Änderungen der Prozeßstruktur systemseitig überhaupt erkennen zu können, welche begleitenden Anpassungen des Datenflusses vorgenommen werden müssen, um anschließend fehlerfrei mit der WF-Ausführung fortfahren zu können. Wie erwähnt, werden entsprechende Fragestellungen in existierenden Ansätzen nicht behandelt.

Im Unterschied zu den meisten WF-Metamodellen bietet das ADEPT-Basismodell auch Beschreibungskonzepte für die Vormodellierung planbarer Abweichungen an, etwa um die unterschiedliche Bedeutung einer Aktivität (Normal- vs. Ausnahmeschritt) zu verschiedenen Ausführungszeitpunkten des WF auszudrücken oder um beim Auftreten eines semantischen Fehlers einen Rücksprung im WF-Graphen automatisch durchführen zu können. Wir haben in dieser Arbeit gezeigt, wie und unter welchen Voraussetzungen sich Vorwärts- und Rückwärtssprünge mit Hilfe des ADEPT-Basismodells

abbilden lassen. Für solche vormodellierten Abweichungen kann das zukünftige Systemverhalten bereits zur Entwurfszeit analysiert, animiert und validiert werden. Die vorgestellten Konzepte ermöglichen einerseits eine bessere Unterscheidbarkeit von Normal- und Ausnahmeschritten zur Laufzeit, andererseits tragen sie zu einer besseren Strukturierung der WF-Modelle bei.

Insgesamt wird das ADEPT-Basismodell den eingangs von Kapitel 3 diskutierten Anforderungen an ein WF-Metamodell gerecht: Ausdrucksmächtigkeit, Strukturierbarkeit, Ausführbarkeit, Formalisierbarkeit, Analysier- und Validierbarkeit, Verständlichkeit, Visualisierbarkeit und Änderbarkeit.

Dynamische Workflow-Änderungen mit $\text{ADEPT}_{\text{flex}}$

Auf der Grundlage des ADEPT-Basismodells wurde der $\text{ADEPT}_{\text{flex}}$-Ansatz entworfen. Seinen Kern bildet eine vollständige Menge von Änderungsoperationen, die sowohl auf Vorlagen- als auch auf Instanzebene anwendbar sind. Um eine klare und eindeutige Änderungssemantik zu erzielen, wird jede Basisänderungsoperation auf einfache, mengenbasierte Operationen zur Manipulation der Knoten- und Kantenmengen des WF-Graphen zurückgeführt. Des weiteren gibt es zu jeder Änderungsoperation formale Vor- und Nachbedingungen, die sowohl strukturelle Modellaspekte als auch Zustandsmarkierungen betreffen. Dadurch können bestimmte Modelleigenschaften, wie regelmäßige Blockstrukturierung oder Verklemmungsfreiheit, auch nach der Anwendung der Operationen gewährleistet werden. Des weiteren wird auf Instanzebene automatisch dafür gesorgt, daß anschließend wieder WF-Graphen mit konsistenten Zustandsmarkierungen resultieren. Die Grundlage hierfür bilden die Markierungs- und Ausführungsregeln des ADEPT-Basismodells, die unabhängig vom Kontext der Änderung anwendbar sind. Schließlich haben wir gezeigt, wie zu verfahren ist, wenn nach der Anwendung einer Änderungsoperation die geforderten Nachbedingungen, etwa die Korrektheit des Datenflußmodells, nicht erfüllt sind. Dabei kann, abhängig von der Art der Änderung und ihrem aktuellen Kontext, zwischen unterschiedlichen Behandlungsstrategien gewählt werden.

Im $\text{ADEPT}_{\text{flex}}$-Ansatz besteht weiter die Möglichkeit, daß WF-Änderungen nur temporär auf den Ausführungsgraphen einer WF-Instanz angewendet und nach Schleifen- oder Fehlerrücksprüngen wieder entfernt werden. Entsprechende Möglichkeiten sind für die angemessene Behandlung von Ausnahmen unerläßlich, werden aber von keinem der in der Literatur diskutierten Ansätze angeboten (vgl. Kapitel 10). In dieser Arbeit haben wir fortschrittliche Konzepte und Verfahren für die Einbringung und Rücknahme temporärer Änderungen vorgestellt. Wir haben gezeigt, welche Informationen zusätzlich vorliegen müssen und welche Maßnahmen konkret erforderlich sind, um bei einem Rücksprung im Kontrollfluß entscheiden zu können, welche Änderungen aus dem Ausführungsgraphen entfernt, welche in modifizierter Form wieder eingebracht und welche unverändert beibehalten werden können. Die Unterstützung temporärer Änderungen ist aber nur einer von mehreren Aspekten, die sich im Zusammenhang mit der Unterstützung von Vorwärts- und Rückwärtssprüngen stellen. Weitere in ADEPT behandelte, in den in der Literatur diskutierten Ansätzen aber nicht aufgegriffene oder nur unbefriedigend gelöste Fragestellungen, sind: Erhalt der Konsistenz von Zustandsmarkierungen, Konzepte zur Vormodellierung planbarer Vorwärts-/Rückwärtssprünge, Zurücksetzen der WF-Kontrolle unter Wahrung von Integritätsaspekten (z.B. Wiederherstellen alter Werte von Datenelementen), Rücksprünge in Zustände früherer Iterationen einer Schleife, Erhalt der Datenkonsistenz bei der Durchführung von Vorwärtssprüngen, korrekte Behandlung von Sprüngen bei Auftreten bedingter Verzweigungen sowie Unterstützung von Sprüngen unterschiedlicher Semantik (z.B. Vorwärtssprünge mit/ohne Nachholen der übersprungenen Aktivitäten, Schleifen- vs. Fehlerrücksprünge, automatische vs. spontane Rücksprünge).

Insgesamt bildet das ADEPT-Basismodell nicht nur für die WF-Modellierung eine ausgezeichnete Grundlage, sondern auch für die dynamische Änderung von WF-Instanzen. Durch die gewählte Blockstruktur, durch die Unterscheidbarkeit verschiedener Knoten- und Kantentypen, durch die explizite Festlegung von Modellaspekten und ihrer Korrektheitseigenschaften, durch den gewählten Markierungsansatz, und vieles mehr, lassen sich Änderungen auf semantisch hoher Ebene und mit der geforderten Zuverlässigkeit und Flexibilität umsetzen. Viele der im Zusammenhang mit (dynamischen) WF-Änderungen auftretenden Bedingungen sind deshalb bereits mit einfachen Mitteln überprüfbar, erfordern also im Gegensatz zu Formalismen wie Petri-Netzen oder Statecharts keine aufwendigen Analysen. Durch die Ausnutzung der Struktureigenschaften der WF-Graphen können darüber hinaus die erforderlichen Kontroll- und Datenflußanalysen gezielt eingegrenzt werden.

Die Mehrzahl der in dieser Arbeit vorgestellten Konzepte und Verfahren wurden prototypisch implementiert und in das ADEPT-WfMS integriert.

Ausblick

Wie so oft, wirft die Bearbeitung eines Problems neue bzw. weiterführende Fragestellungen auf, die Gegenstand von Folgearbeiten sein können. Aufbauend auf der vorliegenden Arbeit bieten sich unter anderem die folgenden Themen für weitergehende Untersuchungen an:

- *Temporale Aspekte*: Welche zusätzlichen Konzepte sind für die Beschreibung temporaler Aspekte (z.B. zeitliche Mindest- und Maximalabstände, Dauern) erforderlich? Welche Semantik muß ihnen zugrundegelegt werden? Welche Korrektheits- und Konsistenzforderungen sind zu fordern und wie lassen sie sich möglichst effizient überprüfen? Welche Probleme können zur Laufzeit auftreten, wenn Aktivitäten „zu spät“ oder „zu früh“ bearbeitet werden oder dynamische WF-Änderungen durchgeführt werden müssen? Wie lassen sie sich angemessen behandeln?
- *Modellerweiterungen*: Welche Erweiterungen sind in Bezug auf die Datenflußmodellierung erforderlich, um einen höheren Grad an Parallelität zu erzielen? Macht es Sinn zwischen logischen und physischen Datenflüssen zu unterscheiden? Wie läßt sich das Mapping von Parametern möglichst komfortabel und automatisch durchführen?
- *Autorisierung*: Wer darf welche Änderungen an welchen Prozeßvorlagen bzw. -instanzen vornehmen? Welche Protokollierungsmechanismen sind erforderlich? Wer muß welchen Änderungen zustimmen? Wer muß über stattgefundene Änderungen informiert werden?
- *Schemaevolution*: Wann dürfen Änderungen einer WF-Vorlage auch noch auf bereits laufende Instanzen „propagiert“ werden und wann ist dies nicht mehr möglich? Wie ist dabei mit WF-Instanzen zu verfahren, für die zuvor Ad-hoc-Änderungen vorgenommen wurden? Wie kann die Propagierung bei einer großen Anzahl von Instanzen effizient bewerkstelligt werden? Wie geht man zur Laufzeit mit Änderungen des Organisationsmodells um, wenn dieses von laufenden WF-Instanzen referenziert wird?
- *Benutzerschnittstellen*: Wie legt ein Benutzer eine Ad-hoc-Änderung für eine WF-Instanz fest? Welche Erklärungskomponenten benötigt er dazu? Sollen den Benutzern unterschiedliche Sichten bzw. Änderungsschnittstellen (z.B. graphischer Editor, graphische Bedieneroberfläche) eingeräumt werden?
- *Verteilte WF-Ausführung und dynamische Änderungen*: Wie lassen sich dynamische Änderungen auf WF-Instanzen anwenden, deren Ausführung von mehreren WF-Servern (siehe z.B. [RBD99, BaDa99a, BRD00]) kontrolliert wird?
- *Implementierungsaspekte*: Wie lassen sich nebenläufige Änderungstransaktionen synchronisieren? Welche Commit-Protokolle müssen zur Synchronisation des WfMS und aufgerufener Aktivitätenprogramme unterstützt werden? Wie müssen WF-Graphen und WF-Instanzen systemintern repräsentiert werden?
- *Process Mining*: Welche Informationen zu Laufzeitänderungen müssen protokolliert werden? Welche Rückschlüsse können hieraus für Prozeßoptimierungen bzw. Vorlagenänderungen gezogen werden?

Einige dieser Themen sind, begleitend zu dieser Arbeit, im ADEPT-Projekt behandelt worden. Eine ausführliche Diskussion hierzu findet man in [RBD99]. Bei allen genannten Punkten handelt sich nicht nur um ein Problem der technischen Realisierung in einem WfMS, sondern es besteht noch grundsätzlicher Forschungsbedarf. Insbesondere muß wieder ein geeignetes formales Modell zugrundegelegt werden. Aufgrund ihres hohen Potentials bleibt zu hoffen, daß prozeßorientierte WfMS in einigen Jahren so selbstverständlich für die Anwendungsentwicklung eingesetzt werden, wie heute Datenbanksysteme. Bis dahin ist allerdings – auch technologisch – noch einiges zu tun.

Literatur

[AAH98] Adam, N.; Atluri, V.; Huang, W.: *Modeling and Analysis of Workflows Using Petri Nets.* Journal of Intelligent Inf. Systems, Vol. 10, No. 2, März/April 1998, S. 131–158

[Aals97] van der Aalst, W.: *Verification of Workflow Nets.* Proc. Application and Theory of Petri Nets, 1997, (LNCS 1248, Springer-Verlag), S. 407–426.

[Aals98] van der Aalst, W.: *Proc. of the Workshop on Workflow-Management: Net-based Concepts, Models, Techniques, and Tools.* Lissabon, Juni, 1998.

[Aals98a] van der Aalst, W.: *Finding Errors in the Design of a Workflow Process: A Petri-net-based Approach.* In [Aals98], S. 60–81.

[Aals98b] van der Aalst, W.: *The Application of Petri-Nets to Workflow Management.* The Journal of Circuits, Systems and Computers, Vol. 8, No. 1, 1998, S. 21–66.

[Aals99] van der Aalst, W.: *Generic Workflow Models*: *How to Handle Dynamic Change and Capture Management Information?* Proc. 4rd IFCIS Int'l Conf. on Cooperative Information Systems (CoopIS'99), Edinburgh, September 1999, S. 115–126

[AbSa94] Abbott, K.; Sarin, S.: *Experiences with Workflow Management: Issues for the Next Generation.* Proc. ACM Conf. on Computer-Supported Cooperative Work (CSCW'94), Chapel Hill, North Carolina, Oktober 1994, S. 113–120

[Adam96] Adametz, H; Barthel, B.; Faustmann, G.; Fleischer, J.; Messer, B.; Wikarski, D.: *Konzepte flexibler Workflow-Management-Systeme.* ISST-Berichte Nr. 31/96, Fraunhofer-Institut für Software- und Systemtechnik (ISST), Berlin, Februar 1996, S. 7–31

[Adle95] Adler, R. M.: *Emerging Standards for Component Software*. IEEE Computer, Vol. 28, No. 3, März 1995, S. 68–77

[AgMi96] Agostini, A.; De Michelis, G.: *Modeling the Document Flow within a Cooperative Process as a Resource for Action.* Technischer Bericht, Cooperation Technologies Laboratory, DSI, Universität Mailand, 1996

[AgMi98] Agostini, A.; De Michelis, G.: *Simple Workflow Models*. In [Aals98], S. 146–163.

[AGSC95] Antunes, P; Guimarães, N.; Segovia, J.; Cardenosa J.: *Beyond Formal Processes: Augmenting Workflow with Group Interaction Techniques.* In [COOCS95], S. 1–9

[AHL92] Auramäki, E.; Hirschheim, R.; Lyytinen, K.: *Modelling Offices Through Discourse Analysis*: *The SAMPO Approach.* The Computer Journal, Vol. 35, No. 4, 1992

[Alon96] Alonso, G.; Agrawal, D.; El Abbadi, A.; Kamath, M.; Günthör, R.; Mohan, C.: *Advanced Transaction Models in Workflow Contexts.* Proc. 12th Int'l Conf. on Data Engineering. New Orleans, Louisiana, Februar 1996

[AMP94] Agostini, A.; De Michelis, G.; Petruni, K.: *Keeping Workflow Models as Simple as Possible.* Proc. Workshop on Computer-Supported Cooperative Work, Petri Nets and Related Formalisms, Saragossa, Spanien, Juni 1994, S. 11-23

[ANRS92] Ansari, M.; Ness, L.; Rusinkiewicz, M.; Sheth, A.: *Using Flexible Transactions to Support Multi-System Telecommunications Applications.* Proc. Int'l Conf. on Very Large Databases (VLDB'92), Vancouver, August 1992, S. 65–76

[AsBu96] Asperti, A.; Busi, N.: *Mobile Petri Nets*. Technischer Bericht, Nr. UBLCS-96-10, Universität Bologna, 1996.

[ASSR93] Attie, P.C.; Singh, M.P.; Sheth, A.; Rusinkiewicz, M.: *Specifying and Enforcing Intertask Dependencies*. Proc. 19th Int'l Conf. on Very Large Databases (VLDB'93), Dublin, August 1993, S. 134–145

[AuLe88] Auramäki, E.; Leppänen, M.: *Exceptions and Office Information Systems.* Proc. IFIP WG 8.4 Working Conf. on Office Inf. Sys., Elsevier Science Publ., 1988, S. 167–182

[BaDa97] Bauer, T.; Dadam, P.: *A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration*. Proc. 2nd IFCIS Conf. on Coop. Inf. Sys. (CoopIS '97), Kiawah Island, South Carolina, Juni 1997, S. 99–108

[BaDa98] Bauer, T.; Dadam, P.: *Architekturen für skalierbare Workflow-Management-Systeme – Klassifikation und Analyse.* Ulmer Informatik-Berichte, Nr. 98-02, Januar 1998

[BaDa99] Bauer, T.; Dadam, P.: *Efficient Distributed Control of Enterprise-Wide and Cross-Enterprise Workflows*. In [DaRe99], Oktober 1999, S. 56–64

[BaDa99a] Bauer, T.; Dadam, P.: *Verteilungsmodelle für Workflow-Management-Systeme – Klassifikation und Simulation.* Informatik Forschung und Entwicklung, Vol. 14, No. 4, Dezember 1999, S. 203–217

[BaDa00] Bauer, T.; Dadam, P.: *Efficient Distributed Workflow Management Based on Variable Server Assignments.* Proc. 12th Conf. on Advanced Information Systems Engineering (CaISE'00), Stockholm, Juni 2000, S. 94–109

[Band95] Bandinelli, S.; Fuggetta, A.; Lavazza, L.; Loi, M.; Picco, G.: *Modeling and Improving an Industrial Process*. IEEE ToSE, Vol. 21, No. 5, Mai 1995 S. 440–453

[BaOl98] Badouel. E.; Oliver, J.: *Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes.* In [Aals98], Juni 1998, S. 129–145.

[Baum96] Baumgarten, B.: *Petri-Netze – Grundlagen und Anwendungen*. 2. Auflage, Spektrum Akademischer Verlag, 1996

[BaWa95] Barthelmess, P.; Wainer, J.: *Workflow Systems: A few Definitions and a few Suggestions*. In [COOCS95], S. 138–147

[BaWa95a] Barthelmess, P.; Wainer, J.: *Workflow Modeling*. Proc. of CYTED-RITOS Int'l Workshop on Groupware, Lissabon, September 1995

[BeDa96] Beuter, T.; Dadam, P.: *Anwendungsspezifische Anforderungen an Workflow-Management-Systeme am Beispiel der Domäne Concurrent Engineering*. Ulmer Informatik-Berichte, Nr. 96-04, Universität Ulm, Juni 1996

[BeKl98] Beckstein, C.; Klausner, J.: *A Meta Level Architecture for Workflow Management.* Proc. 3rd Conf. Integrated Design and Process Techn. (IADT'98), Berlin, Juli 1998.

[BeKl99] Beckstein, C.; Klausner, J.: *A Planning Framework for Workflow Management.* Proc. Workshop on Intelligent Workflow and Process Management, Stockholm, Aug. 1999.

[BEMM98] Benjamin, P.; Erraguntla, M.; Mayer, R.; Marshall, C.: *Toolkit for Enabling Analysis and Modeling of Adaptive Workflows (TEAMWORK).* In [KDB98]

[BeMy98] Berry, P.; Myers, K.: *Adaptive Process Management: An AI Perspective.* In [KDB98]

[BES98] Bukhres, O.; Eder, J.; Salza, S. (Hrsg.): *Proc. EDBT-Workshop on Workflow Management Systems* (held in conjunction with the 6th Int'l Conf. On Extending Database Technology (EDBT'98)), Valencia, Spanien, März 1998

[BFG93] Bandinelli, S.; Fuggetta, A.; Ghezzi, C.: *Software Process Model Evolution in the SPADE Environment*. IEEE ToSE, Vol. 19, No. 12, 1993, S. 1128–1144

[BhCr90] Bhandaru, N.; Croft, W.: *An Architecture for Supporting Goal-based Cooperative Work.* Proc. IFIP WG 8.4 Conf. on Multi-User Interfaces and Applications, Heraklion, Griechenland, September 1990, S. 337–354

[Biec96] Biechele, B.; Ernst, D.; Schmid, J.; Schulte, W.; Houdek, F.: *Erfahrungen bei der Modellierung eingebetteter Systeme mit verschiedenen SA/RT-Ansätzen*. Ulmer Informatik-Berichte, Nr. 96-09, Dezember 1996

[Blas95] Blaser, R.: *Implementierung von Schnittstellen des ADEPT-Datenmanagers und einer Test- und Demonstrationsumgebung.* Software-Praktikum, Universität Ulm, 1997.

[Blas96] Blaser, R.: *Konfiguration verteilter Anwendungen aus vorgefertigten Programmbausteinen.* Diplomarbeit, Universität Ulm, 1996

[BlNu95] Blumenthal, R.; Nutt, G.J.: *Supporting Unstructured Workflow Activities in the Bramble ICN System.* In [COOCS95], S. 130–137

[BMR96] Barbará, D.; Mehrotra, S.; Rusinkiewicz, M.: *INCAS: Managing Dynamic Workflows in Distributed Environments.* Journal of Database Mgmt, Vol. 7, No. 1, 1996, S. 5–15.

[Bogi95] Bogia, D.: *Supporting Flexible, Extensible Task Descriptions In and Among Tasks.* Dissertation, University of Urbana, Illinois, 1995

[BoKa95] Bogia, D.; Kaplan, S.: *Flexibility and Control for Dynamic Workflows in the wOrlds Environment.* In [COOCS95], S. 148–161

[BoMu98] Borgida, A.; Murata, T.: *Workflows as Persistent Objects with Persistent Exceptions – A Framework for Flexibility.* In [KDB98]

[Borg85] Borgida, A.: *Language Features for Flexible Handling of Exceptions in Information Systems.* ACM Trans. on Database Systems, Vol. 10, No. 4, Dez. 1985, S. 565–603

[BPS97] Bichler, P.; Preuner, G.; Schrefl, M.: *Workflow Transparency*. Proc. 9th Int'l Conf. Advanced Information Systems Engineering (CaiSE'97), Barcelona, Juni 1997 (LNCS 1250, Springer-Verlag), S. 423–436

[BRD00] Bauer, T.; Reichert, M.; Dadam, P.: *Effiziente Durchführung von Prozessmigrationen in verteilten Workflow-Management*. Ulmer Informatik-Berichte Nr. 2000-08, Juni 2000

[Brei93] Breitbart, Y.; Deacon, A.; Schek, H.-J.; Sheth, A.; Weikum, G.: *Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows.* SIGMOD Record, Vol. 22, No. 3, September 1993, S. 23–30

[BuJa96] Bußler, C.; Jablonski, S.: *Die Architektur des modularen Workflow-Management-Systems MOBILE*. In [VoBe96], S. 369–388

[Casa96] Casati, F.; Grefen, P.; Pernici, B.; Pozzi, G.; Sánchez, G.: *WIDE Workflow Model and Architecture.* Technical Report No. 96–19, CTIT, Universität Twente, 1996

[CCPP95] Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G.: *Conceptual Modeling of WorkFlows*. Proc. 14th Int'l Conf. Object-Orientation and Entity-Relationship Modeling (OOER '95), Gold Coast, Australien, Dezember 1995, S. 341–354

[CCPP98] Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G.: *Workflow Evolution*. Data & Knowledge Engineering, Vol. 24, No. 3, Januar 1998, S. 211-238

[CFM99] Casati, F.; Fugini, M.; Mirbel, I.: *An Environment for Designing Exceptions in Workflows.* Information Systems, Vol. 24, No. 3, 1999, S. 255–273.

[CKL97] Chiu, D.; Karlapalem, K.; Li, Q.: *Exception Handling in ADOME Workflow Management System.* Technical Report No. HKUST–CS97–02, Dept. of Computer Science, University of Science and Technology, Hong Kong, 1997

[CKO92] Curtis, B.; Kellner, M.I.; Over, J.: *Process Modeling*. Communications of the ACM, Vol. 35, No. 9, September 1992, S. 75–90

[CLH95] Conradi, R.; Liu, Ch.; Hagaseth, M.: *Planning Support for Cooperating Transactions in EPOS.* Information Systems, Vol. 20, No. 4, 1995, S. 317–336

[CLK99] Chiu, D.; Li, Q.; Karplapalem, K.: *A Meta Modeling Approach To Workflow Management Systems Supporting Exception Handling.* Information Systems, Vol. 24, No. 2, 1999, S. 159–184.

[CLR89] Cormen, T.; Leiserson, C.; Rivest, R.: Introduction to Algorithms. McGraw-Hill, 1989

[CNGM95] Cugola, G.; Di Nitto, C.; Ghezzi, C.; Mantione, M.: *How to Deal with Deviations During Process Model Enactment.* Proc. 17th Int'l Conf. on Software Engineering, Seattle, April 1995, S. 265–273.

[COOCS95] *Proc. ACM Conf. on Organizational Computing Systems*, Milpitas, California, August 1995, ACM Press

[CrMa95] Craven, N.; Mahling, D.: *Goals and Processes*: *A Task Basis for Projects and Workflows*. In [COOCS95], S. 237–249

[Cugo98] Cugola, G.: *Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models.* IEEE ToSE, Vol. 24, No. 11, November 1998, S. 982–1001.

[Dada95] Dadam, P.; Kuhn, K.; Reichert, M.; Beuter, T.; Nathe, M.: *ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen.* Proc. 25. GI–Jahrestagung, Zürich, September 1995 (Informatik Aktuell, Springer-Verlag), S. 677–686

[Dada96] Dadam, P.: *Verteilte Datenbanken und Client/Server-Systeme - Grundlagen, Konzepte und Realisierungsformen*. Springer-Verlag, 1996

[DaRe99] Dadam, P.; Reichert, M. (eds.): *Proc. Workshop on Enterprise-Wide and Cross-Enterprise Workflow-Management: Concepts, Systems, Applications*, 29. Jahrestagung der GI (Informatik'99), Paderborn, Oktober 1999.

[DeGr94] Deiters, W.; Gruhn, V.: *The FUNSOFT Net Appoach to Software Process Management.* Int'l Journal of Software Engineering and Knowledge Engineering, Vol. 4, No. 2, 1994, S. 229–256

[DeMa96] Dellen, B.; Maurer, F.: *Integrating Planning and Execution in Software Development Processes.* Proc. of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'96), Stanford, 1996

[DEXA97] Wagner, R.: *Proc. 8th Int'l Workshop on Database and Expert Systems Applications*, Toulouse, Frankreich, September 1997, IEEE Computer Society Press

[DGSZ94] Dinkhoff, G.; Gruhn, V.; Saalmann, A.; Zielanko, M.: *Business Process Modelling in the Workflow Management Environment LEU.* Proc. 13th Int'l Conf. on the Entity-Relationship Approach (ER'94), Manchester, 1994 (LNCS 881, Springer), S. 46–63

[DHL90] Dayal, U.; Hsu, M.; Ladin, R.: *Organizing Long-Running Activities with Triggers and Transactions*. Proc. ACM SIGMOD Int'l Conf. on the Management of Data, Atlantic City, NJ, 1990, S. 204–214 (Nachgedruckt in [Ston94], S. 324–334)

[DHL91] Dayal, U.; Hsu, M.; Ladin, R.: *A Transactional Model for Long-Running Activities.* Proc. 17th Int'l Conf. on Very Large Databases (VLDB'91), Barcelona, September 1991, S. 113–122

[DHLS96] Deiters, W.; Herrmann, T.; Lüffler, T.; Striemer, R.: *Identifikation und Unterstützung semi-strukturierter Prozesse in prozeßorientierten Telekooperationssystemen.* Proc. DCSCW'96, Stuttgart-Hohenheim, Sept. /Okt. 1996, Springer-Verlag, S. 261–274

[Dijk68] Dijkstra, E.: *Cooperating Sequential Processes*. In: Genuys, F. (ed.): Programming Languages, Academic Press, London, 1968

[DiSa97] Dierker, M.; Sander, M.: *Lotus Notes 4.5 und Domino – Integration von Groupware und Internet.* Addison Wesley, 1997

[DLS96] Deiters, W.; Lindert, F.; Schiprowski, R.: *A Transaction Concept for FUNSOFT Nets.* Proc. Workshop Geschäftsprozeßmodellierung und Workflowsysteme, Informatik'96, Band 2, Klagenfurt, 1996, S. 89–107

[DMP97] Dellen, B.; Maurer, F.; Pews, G.: *Knowledge Based Techniques to Increase the Flexibility of Workflow Management.* Data & Knowledge Engineering, Vol. 23, No. 3, September 1997, S. 269–296

[Dona89] Donald, B.R.: *Error Detection and Recovery in Robotics*. Springer, LNCS 336, 1989.

[DRK97] Dadam, P.; Reichert, M.; Kuhn, K.: *Clinical Workflows - The Killer Application for Process-oriented Information Systems?* Ulmer Informatik-Berichte, Nr. 97–16, November 1997.

[DRK00] Dadam, P.; Reichert, M.; Kuhn, K.: *Clinical Workflows - The Killer Application for Process-oriented Information Systems?* Proc. 4th Int'l Conf. on Business Information Systems (BIS'2000), Posen, Polen, April 2000, S. 36–59

[EdHo98] Edmont, D.; ter Hofstede, A.: *Achieving Workflow Adaptability by Means of Reflection.* In [KDB98]

[EdLi95] Eder, J.; Liebhart, W.: *The Workflow Activity Model WAMO*. Proc. 3rd Int'l Conf. on Cooperative Information Systems, Wien, Mai 1995, S. 87–98

[EdLi96] Eder, J.; Liebhart, W.: *Workflow Recovery.* Proc. 1st IFCIS Int'l Conf. on Cooperative Information Systems (CoopIS'96), Brüssel, Juni 1996, S. 124–134

[EdLi98] Eder, J.; Liebhart, W.: *Contributions to Exception Handling in Workflow-Management.* In [BES98], S. 3-10

[EhTa96] Ehrig, H.; Taentzer, H.: *Computing by Graph Transformation – A Survey and Annotated Bibliography.* Technischer Bericht Nr. 96-21, TU Berlin, März 1996.

[EKP98] Endl, R.; Knolmayer, G.; Pfahrer, M.: *Modeling Processes and Workflows by Business Rules.* SWORDIES Report, Universität Bern, Juli 1998.

[EKR95] Ellis, C.A.; Keddara, K.; Rozenberg, G.: *Dynamic Change within Workflow Systems*. In [COOCS95], S. 10–21

[EKW98] Ellis, C.A.; Keddara, K.; Wainer, W.: *Modeling Workflow Dynamic Changes Using Timed Hybrid Flow Nets.* In [Aals98], Juni 1998, S. 109–128.

[ElBe82] Ellis, C.A.; Bernal, M.: *Office-Talk-D*: *An Experimental Office Information System.* Proc. 1st ACM-SIGOIS Conf. on Office Information Systems, 1982, S.131–140

[Elli79] Ellis, C.A.: *Information Control Nets: A Mathematical Model of Office Information Flow.* Proc. of the ACM Conf. on Simulation, Modeling, and Measurement of Computer Systems, Boulder, Colorado, USA, 1979, S. 225–240

[Elli83] Ellis, C.A.: *Formal and Informal Models of Office Activity.* In Mason, R. (Ed.): Information Processing, Elsevier Science Publ. (North-Holland), 1983, S. 11-22

[Elma92] Elmargarmid, A.K. (Hrsg.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publ., 1992

[ElMa97] Ellis, C.A.; Maltzahn, C.: *The Chautauqua Workflow System*. In Proc. 30th Hawaii Int'l Conf. on System Sciences, Maui, Hawaii, Januar 1997

[ElNu80] Ellis, C.A.; Nutt, G.J.: *Office Information Systems and Computer Science*. ACM Computing Surveys, Vol. 12, No. 1, März 1980, S. 27–60.

[ElNu93] Ellis, C.A.; Nutt, G.J.: *Modeling and Enactment of Workflow Systems*, Proc. 14th Int'l Conf. on Application and Theory of Petri Nets, Chicago, Juni 1993 (LNCS 691, Springer-Verlag), S. 1–16

[ElWa94] Ellis, C.A.; Wainer, J.: *Goal-based Models of Collaboration.* Collaborative Computing, Vol. 1, 1994, S. 61–86

[FaWi96] Faustmann, G.; Wikarski, D.: *Exception Handling in Petri-Net-based Workflow Management.* In [PAKM96].

[FGHW88] Flores, F.; Graves, M.; Hartfield, B.; Winograd, T.: *Computer Systems and the Design of Organizational Interaction.* ACM Transactions on Office Systems, Vol. 6, No. 2, April 1988, S. 153–172

[FiNi71] Fikes, R.; Nilsson, N.: *A New Approach to the Application of Theorem Proving.* Artificial Intelligence, Vol. 2, 1971, S. 189–208.

[GaSa87] Garcia-Molina, H.; Salem, K.: *SAGAS*. Proc. SIGMOD'87 Conf., San Francisco, Mai 1987, S. 249–260 (Nachdruck in [Ston94], S. 290–300)

[Geha82] Gehani, N.H.: *The Potential of Forms in Office Automation.* IEEE Transactions on Communication, Vol. 30, No. 1, Januar 1982, S. 120–125

[Genr91] Genrich, H.J.: *Predicate Transition Nets.* In Jensen, K.; Rozenberg, G. (Hrsg.): High Level Petri Nets: Theory and Application. Springer-Verlag, 1991

[GHS95] Georgakopoulos, D.; Hornick, M.; Sheth, A.: *An Overview of Workflow Management.* Distributed and Parallel Databases, Vol. 3, No. 2, April 1995, S. 119–153

[Goer90] Goering, S.: *A Graph Grammar Approach to Concurrent Programming.* Dissertation, University of Illinois, Urbana, 1990 (Report No. UIUCDCS–R–90–1576)

[Goes97] Goesmann, T.; Hagemeyer, J.; Löffeler, T.; Rolles, R.; Striemer, R.: *Einführung von Workflow-Management in die Praxis aus Sicht verschiedener Perspektiven – das Projekt MOVE.* In Becker, J.; Rosemann, M. (Hrsg.): Usability Engineering, Arbeitsbericht, Institut für Wirtschaftsinformatik, Universität Münster, 1997, S. 51-68.

[Gött88] Göttler, H.: *Graphgrammatiken in der Softwaretechnik – Theorie und Anwendungen.*Informatik-Fachberichte Nr. 178, Springer-Verlag, 1988

[GPP96] Glance, N.; Pagani, D.; Pareschi, R.: *Generalized Process Structure Grammars (GPSG) for Flexible Representations of Work.* Proc. 7th Int'l Conf. on Computer Supported Cooperative Work (CSCW'96), Boston, MA, 1996, S. 180–189.

[Grei88] Greif, I. (Hrsg.): *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publ., 1988.

[Grim97] Grimm, M.: ADEPT–TIME: *Temporale Aspekte in flexiblen Workflow-Management-Systemen.* Diplomarbeit, Universität Ulm, 1997

[Gros97] Gross, T.: *Towards Flexible Support for Cooperation: Group Awareness in Shared Workspaces.* In [DEXA97], S. 406–411

[GrRe93] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publ., 1993

[Grud94] Grudin, J.: *Computer-Supported Cooperative Work: History and Focus*. IEEE COMPUTER, Vol. 27, No. 5, Mai 194, S. 19–26

[Gruh91] Gruhn, V.: *Validation and Verification of Software Process Models*. Dissertation, Universität Dortmund, Fachbereich Informatik, 1991

[Gruh93] Gruhn, V.: *Entwicklung von Informationssystemen in der LION-Entwicklungsumgebung*. In Scheschonk, G.; Reisig, W. (Hrsg.): Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen, Springer-Verlag, 1993, S. 31–45

[GSS99] Gulbins, J.; Seyfried, M.; Strack-Zimmermann, H.: *Dokumenten-Management – Vom Imaging zum Business-Dokument*. Springer, 1999.

[HaAl98] Hagen, C.; Alonso, G.: *Flexible Exception Handling in the OPERA Process Support System.* Proc. 18th Int'l Conf. on Distributed Computing Systems (ICDCS'98), Amsterdam, 1998, S. 526–533.

[Hamm90] Hammond, K.: *Explaining and Repairing Plans that Fail.* Artificial Intelligence, Vol. 45, 1990, S. 173–228.

[Han97] Han, Y.: *Software Infrastructure for Configurable Workflow Systems.* Dissertation, Fachbereich Informatik, TU Berlin, April 1997

[Han97a] Han, Y.: *HOON – A Formalism Supporting Adaptive Workflows*. Tech. Report No. #UGA-CS-TR-97-005, Dep. of Comp. Science, University of Georgia, Nov. 1997

[Hare87] Harel, D.: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, Vol. 8, 1987, S. 231–74

[HaSh98] Han, Y.; Sheth, A.: *On Adaptive Workflow Modeling*. Proc. 4th Int'l Conf. on Information Systems Analysis and Synthesisis, Orlando, Florida, Juli 1998

[HaSt95] Hammer, M.; Stanton, S.A.: *The Reengineering Revolution – The Handbook.* Harper Collins Publ., 1995

[Hast99] Hastedt-Marckwardt, C.: *Workflow-Management-Systeme – Grundlagen, Standards und Trends.* Informatik Spektrum, Band 22, Heft 2, April 1999, S. 99–109.

[Härd97] Härder, Th (Hrsg.).: *Themenheft Workflow-Management*. Informatik Forsch. Entw., Band 12, Heft 2, 1997

[HEA90] Hämmäinen, H.; Eloranta, E.; Alasuvanto, J.: *Distributed Form Management.* ACM Transactions on Information Systems, Vol. 8, No. 1, Januar 1990, S. 50–76

[Hein98a] Heinl, P.: *Exceptions During Workflow Execution.* In [BES98], S. 11-20.

[Hein98b] Heinl, P.; Horn, S.; Jablonski, S.; Neeb, J.; Stein, K.; Teschke, M.: *A Comprehensive Approach to Flexibility in Workflow Management Systems.* Technischer Bericht TR-I6-1998-6, Fakultät für Informatik, Universität Erlangen-Nürnberg, 1998.

[HeJu96] Herrmann, T.; Just-Hahn, K.: *Organizational Learning with Flexible Workflow Management Systems.* Proc. of the Workshop on Organizational Learning, CSCW'96, SIGOIS Bulletin, Vol. 17, No. 3, Dezember 1996, S. 54–57

[Henn89] Hennessy, M.: *Algebraic Theory of Processes*. Cambridge: The MIT Press, 1989

[Hens97] Hensinger, C.: *ADEPTflex – Dynamische Modifikation von Workflows und Ausnahmebehandlung in WfMS*. Diplomarbeit, Universität Ulm, 1997

[Hens00] Hensinger, C.; Reichert, M.; Bauer, T.; Strzeletz, T.; Dadam, P.: *$ADEPT_{workflow}$ – Advanced Workflow Technology for the Efficient Support of Adaptive, Enterprise-wide Processes*, 7th Int'l. Conf. on Extending Database Technology, Software Demonstration Track, Konstanz, März 2000, S. 29–30

[Hens00a] Hensinger, C.: *Dokumentenorientiertes Workflow-Management mit Lotus Notes.* Begleitunterlagen zum Praxis-Seminar „Geschäftsprozeßmodellierung und Workflow-Management", Deutsche Informatik Akademie, Ulm, Oktober 2000

[Hert89] Hertzberg, J.: *Planen – Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz.* Reihe Informatik, Band 65, BI Wissenschaftsverlag, 1989

[HHJS97] Hagemeyer, J.; Hermann, T.; Just-Hahn, K.; Striemer, R.: *Flexibilität bei Workflow-Management-Systemen.* In: Likowski, R. et al. (Hrsg.): Proc. Software-Ergonomie'97, Dresden, März 1997, S. 179–190

[HHSW96] Han, Y.; Himminghofer, J.; Schaaf, T.; Wikarski, D.: *Management of Workflow Resources to Support Runtime Adaptability and System Evolution*. In [PAKM96].

[HJKW95] Heimann, P.; Joeris, G.; Krapp, C.-H.; Westfechtel, B.: *A Programmed Graph Rewriting System for Software Process Management*. Proc. of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA `95), Volterra, Italien, August 1995, S. 123–132

[HJKW96] Heimann, P.; Joeris, G.; Krapp, C.-H.; Westfechtel, B.: *DYNAMITE*: *Dynamic Task Nets for Software Process Management.* Proc. 18th Int'l Conf. Software Engineering (ICSE'96), Berlin, März 1996, S. 331–341

[HKBB92] Heinemann, G.; Kaiser, G.; Barghouti, N.; Ben-Shaul, L.: *Rule Chaining in MARVEL: Dynamic Binding of Parameters*. IEEE Expert, Vol. 7, No. 6, 1992, S. 26–32

[HKS94] Hasenkamp, U.; Kirn, S.; Syring, M. (Hrsg.): *CSCW – Computer Supported Cooperative Work*: *Informationssysteme für dezentralisierte Unternehmensstrukturen*. Addison-Wesley, 1994

[HKWJ97] Heimann, P.; Krapp, C.-H.; Westfechtel, B; Joeris, G.: *Graph-based Software Process Management*. Int'l Journal of Software Engin. and Knowledge Engin., Vol. 7, 1997

[HNOR95] Hilpert, W.; Nastansky, L.; Ott, M.; Riempp, G.: *Geschäftsprozeßorientierte Modellierung /Einführung von Groupware-Anwendungen.* Technischer Bericht, Workgroup Computing Competence Center, Universität Paderborn, 1995

[HoJa98] Horn, S.; Jablonski, S.: *An Approach to Dynamic Instance Adaptation in Workflow Management Applications.* In [KDB98]

[HOR98] ter Hofstede, A.; Orlowska, M.; Rajapakse, J.: *Verification Problems in Conceptual Workflow Specifications.* Data & Knowledge Engin., Vol. 24, No. 3, 1998, S. 239-256

[HsKl96] Hsu, M.; Kleissner, C.: *ObjectFlow: Towards a Process Management Infrastructure.* Distributed and Parallel Databases, Vol. 4, 1996, S. 169–194

[HSS96] Heinl, P.; Schuster, H.; Stein, K.: *Behandlung von Ad-hoc-Workflows im MOBILE Workflow-Modell.* Proc. Softwaretechnik in Automation und Kommunikation – Rechnergestützte Teamarbeit (STAK'96), München, März 1996, S. 229–242

[Hsu93] Hsu, M. (Hrsg.): *Special Issue on Workflow and Extended Transaction Systems*. IEEE Bulletin of the Technical Commitee on Data Engineering, Vol. 16, No. 2, 1993

[Hsu95] Hsu, M. (Hrsg.): *Special Issue on Workflow Systems*, IEEE Bulletin of the Technical Commitee on Data Engineering, Vol. 18, No. 1, 1995

[HVA97] Hauschildt, D.; Verbeek, E.; van der Aalst, W.: *WOFLAN: A Petri-net-based Workflow Analyzer.* Computing Science Report No. 97-12, Universität Eindhoven, August 1997.

[Jabl95] Jablonski, S.: *Workflow-Management-Systeme: Motivation, Modellierung, Architektur.* Informatik-Spektrum, Band 18, Heft 1, Februar 1995, S. 13–24

[Jabl97] Jablonski, S.: *Architektur von Workflow-Management-Systemen.* In [Härd97], S. 72–81

[JaBu96] Jablonski, S.; Bussler, C.: *Workflow Management – Modeling Concepts, Architecture, and Implementation.* International Thompson Computer Press, 1996

[JaCo93] Jaccheri, M.; Conradi, R.: *Techniques for Process Model Evolution in EPOS.* IEEE Transactions on Software Engineering, Vol. 19, No. 12, 1993, S. 1145–1156

[JaKe97] Jajodia, S.; Kerschberg, L. (Hrsg.): *Advanced Transaction Models and Architectures*, Kluwer Academic Publishers, 1997

[JBS97] Jablonski, S.; Böhm, M.; Schulze, W. (Hrsg.): *Workflow-Management: Entwicklung von Anwendungen und Systemen.* Dpunkt-Verlag, 1997

[Jens90] Jensen, K.: *Coloured Petri Nets*. In Rozenberg, G. (Hrsg.): Advances in Petri Nets, LNCS 483, Springer-Verlag,1990, S. 342–416

[JKMW97] Joos, B.; Katzsch, R.M.; Meier, A.; Wernet, C.: *Drei Workflow-Management-Systeme im praktischen Vergleich: Workflow, Staffware und InConcert.* Theorie und Praxis der Wirtschaftsinformatik, Heft 193, 34. Jahrgang, Januar 1997, S. 81–103

[Joer98] Joeris, G.: *Aspekte und Konzepte der Flexibilität in Workflow-Managementsystemen.* In [SiWe98], 1998, S. 3–12.

[Joer99] Joeris, G.: *Defining Flexible Workflow Execution Behaviors.* In [DaRe99], S. 49–55.

[JoHe98] Joeris, G.; Herzog, O.: *Managing Evolving Workflow Specifications*. Proc. 3rd IFCIS Conf. on Cooperative Information Systems (CoopIS'98), New York, August 1998.

[JST97] Jablonski, S.; Stein, K.; Teschke, M.: *Experiences in Workflow Management for Scientific Computing*. In [DEXA97], S. 56–61

[KaDe96] Karl, R.; Deiters, W.: *Staffware Workflow.* Studie zu Workflow-Management – Groupware Computing, Teil 2, dsk Beratungs-GmbH für Bürokommunikation und Computertechnik, Januar 1996, S. 413–474.

[KaHe92] Kambhampati, S.; Hendler, J.: *A Validation-structure-based Theory of Plan Modification and Reuse.* Artificial Intelligence, Vol. 55, No. 2, 1992, S. 192–258.

[KaRa90] Karbe, B.; Ramsperger, N.: *Influence of Exception Handling on the Support of Cooperative Office Work.* Proc. IFIP WG 8.4 Conf. on Multi-User Interfaces and Applications, Heraklion, Griechenland, September 1990

[KaRa91] Karbe, B.; Ramsperger, N.: *Concepts and Implementation of Migrating Office Processes*. In Bianer, W.; Hernandez, D. (Hrsg.): Verteilte Künstliche Intelligenz und kooperatives Arbeiten, Springer-Verlag, 1991, S. 136–147

[KaRa96] Kamath, M.; Ramamritham, K.: *Correctness Issues in Workflow Management*. Distributed Systems Engineering Journal, Vol. 3, No. 4, 1996, S. 213–221.

[Karb94] Karbe, B.: *Flexible Vorgangssteuerung mit ProMInanD*. In [HKS94], S. 117–133

[KDB98] Klein, M.; Dellaroca, C.; Bernstein, A.: *CSCW-98 Workshop: Towards Adaptive Workflow Systems.* Workshop im Rahmen der Conf. on Computer-Supported Cooperative Work, Seattle, WA, November 1998.

[Kirs96] Kirsch, M.: *Realisierung einer Entwicklungsumgebung für die Modellierung und Animation flexibler Workflows*. Diplomarbeit, Universität Ulm, 1996

[KiUn94] Kirn, S.; Unland, R.: *Workflow Management mit kooperativen Softwaresystemen: State of the Art und Problemabriß.* Arbeitsbericht Nr. 29, Institut für Wirtschafsinformatik, Universität Münster, März 1994

[KKS97] Knoop, J.; Koschützki, D.; Steffen, B.: *Basic-block Graphs: Living Dinosaurs?* Technischer Bericht MIP-9715, Universität Passau, Juli 1997.

[KlDe98] Klein, M.; Dellaroca, C.: *A Knowledge-Based Approach to Handling Exceptions in Workflow Systems.* In [KDB98], 1998

[Kony96] Konyen, I.: *Organisations- und Ablaufstrukturen im Krankenhaus – Anforderungen, Werkzeuge und deren Anwendung.* Diplomarbeit, Universität Ulm, 1996

[KRD95] Kuhn, K.; Reichert, M.; Dadam, P.: *Unterstützung der klinischen Kooperation durch Workflow-Management-Systeme - Anforderungen, Probleme, Perspektiven.* In: Proc. 40. Jahrestagung der GMDS (GMDS'95), Bochum, September 1995, S. 437–441

[Krei91] Kreifelts, T.; Hinrichs, E.; Klein, K.-H.; Seuffert, P.; Woetzel, G.: *Experiences with the DOMINO Office Procedure System*. Proc. 2nd European Conf. on CSCW (ECSCW'91), Amsterdam, 1991, S. 117–130

[Krei91a] Kreifelts, T.: *Coordination of Distributed Work: From Office Procedures to Customizable Activities.* Proc. 4. Internationaler GI-Kongreß Wissensbasierte Systeme, München, Oktober 1991, S. 148-159

[KrGe99] Kradolfer, M.; Geppert, A.: *Dynamic Workflow Schema Evolution Based on Workflow Type Versioning and Workflow Migration.* Proc. 4rd IFCIS Int'l Conf. on Cooperative Information Systems (COOPIS'99), Edinburgh, September 1999, S. 104–114

[Krue92] Krueger, C.: *Software Reuse.* ACM Comp. Surveys, Vol. 24, No. 2, 1992, S. 131–184

[KRW90] Karbe, B.; Ramsperger, N.; Weiss, P.: *Support of Cooperative Work by Electronic Circulation Folders*. Conf. on Office Information Systems, Cambridge, MA., USA, 1990 (Abgedruckt in ACM SIGOIS Bulletin, Vol. 11, No. 2+3, S. 109–117)

[KrWo87] Kreifelts, T.; Woetzel, G.: *Distribution and Exception Handling in an Office Procedure System.* In Bracchi, G.; Tsichritzis, D. (Hrsg.): Office Systems: Methods and Tools, Elsevier Science Publ., 1987, S.187–208

[KrZe96] Krcmar, H.; Zerbe, S.: *Negotiation Enabled Workflow (NEW): Workflow-Systeme zur Unterstützung flexibler Geschäftsprozesse*. Arbeitspapier Nr. 94, Lehrstuhl für Wirtschaftsinformatik, Universität Hohenheim, Januar 1996

[KSW87] Kreifelts, T.; Seuffert, P.; Woetzel, G.: *Ausnahmebehandlung, Verteilung und Adressierung im Vorgangssystem DOMINO.* Proc. 17. GI–Jahrestagung, München, 1987 (Informatik–Fachberichte 156, Springer-Verlag), S. 682–696

[Kubi98] Kubicek, M.: *Organisatorische Aspekte in flexiblen Workflow-Management-Systemen*. Diplomarbeit, Universität Ulm, Abt. DBIS, 1998

[Kuhn94a] Kuhn, K.; Reichert, M.; Nathe, M.; Beuter, T.; Heinlein, C.; Dadam, P.: *A Conceptual Approach to an Open Hospital Information System*, Proc. 12th Int'l Congress of the European Federation for Medical Inform. (MIE'94), Lissabon, Mai 1994, S. 374–378

[Kuhn94] Kuhn, K.; Reichert, M.; Nathe, M.; Beuter, T.; Dadam, P.: *An Infrastructure for Cooperation and Communication in an Advanced Clinical Information System.* Proc. 18th Ann. Symp. on Computer Applications in Medical Care (SCAMC'94), Washington, November 1994, S. 519–523

[KwBa97] Kwan, M.; Balasubramanian, P.: *Dynamic Workflow Management*: *A Framework for Modeling Workflows.* Proc. 30th Annual Hawaiian Int'l Conf. On System Sciences (HICSS-30), Maui, Hawaii, Januar 1997, Vol. IV, S. 367-376

[LeAl94] Leymann, F.; Altenhuber, W.: *Managing Business Processes as an Information Resource*. IBM Systems Journal, Vol. 33, No. 2, 1994, S. 326–348

[LeCr89] Lefkowitz, L.; Croft, W.: *Planning and Execution of Tasks in Cooperative Work Environments.* Proc. 5th IEEE Conf. on Artificial Intelligence Applications, Miami, Florida, 1989, S. 255–261

[LeRo97] Leymann, F.; Roller, D.: *Workflow-based Applications*. IBM Systems Journal, Vol. 36, No.1, 1997, S. 102–123

[LeRo00] Leymann, F.; Roller, D.: *Production Workflow*. Prentice Hall, 2000.

[Leym95] Leymann, F.: *Supporting Business Transactions via Partial Recovery in Workflow Management Systems*, Proc. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW '95), Dresden, März 1995 (Informatik Aktuell, Springer-Verlag), S. 51–70

[Leym97] Leymann, F.: *Transaktionsunterstützung für Workflows.* In [Härd97], S. 82–90

[LiCo93] Liu, C.; Conradi, R.: *Automatic Replanning of Task Networks for Process Model Evolution in EPOS.* Proc. European Software Engineering Conference'93 (ESEC'93), Garmisch-Partenkirchen, Sept. 1993, S. 434–450.

[LiPu97] Liu, L.; Pu, C.: *ActivityFlow: Towards Incremental Specification and Flexible Coordination of Workflow Activities.* Proc. 16th Int'l Conf. on Conceptual Modeling, Los Angeles, November 1997 (LNCS 1331, Springer-Verlag), S. 169–182.

[LiPu98] Liu, L.; Pu, C.: *Methodical Restructuring of Complex Workflow Activities.* Proc. 14th Int'l Conf. On Data Engineering (ICDE'98), Orlando, Florida, Feb. 1998, S. 342-350.

[LöHa94] Löwe, M.; Han, Y.: *Process Modeling and Control with Higher-order-Nets*. Forschungsbericht Nr. 94–34, Fachbereich Informatik, TU Berlin, 1994

[LOL98] Liu, C.; Orlowska, M.; Li, H.: *Automatic Handover in Dynamic Workflow Environments.* Proc. 10th Conf. on Advanced Information Systems Engineering (CAiSE'98), Pisa, Italien, Juni 1998, S. 159–171.

[LSMK98] Luo, Z.; Sheth, A.; Miller, J.; Kochut, K.: *Defeasibile Workflow, its Computation and Exception Handling.* In [KDB98]

[Lutz88] Lutze, R.: *Customizing Cooperative Office Procedures by Planning.* Proc. Conf. on Office Inf Sys, Palo Alto, CA, SIGOIS Bulletin, Vol. 9, No. 2/3, 1988, S. 63–67

[MAGK95] Mohan, C.; Alonso, G.; Günthör, R.; Kamath, M.: *Exotica: A Research Perspective on Workflow Management Systems.* In [Hsu95], S. 19–26

[Malo99] Malone, T. et al: *Tools for Inventing Organizations: Toward a Handbook of Organizational Processes*. Mgmt Science, Vol. 45, No. 3, März 1999, S. 425–443.

[MaPn92] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems-Specification*. Springer-Verlag, 1992

[Mart92] Von Martial, F.: *Coordinating Plans of Autonomous Agents*. Lecture Notes in Artificial Intelligence 610, Springer Verlag, 1992

[McSa93] McCarthy, D.; Sarin, S.: *Workflow and Transactions in InConcert*. In [Hsu93], S.53–56

[MDP97] Maurer, F.; Dellen, B.; Pews, G.: *CoMo-Kit – Modellierung und Operationalisierung von wissensintensiven Arbeitsabläufen.* EMISA Forum, Heft 2, 1997, S. 21–27

[Meye96] Meyer, J.: *Anforderungen an zukünftige Workflow-Management-Systeme: Flexibilisierung, Ausnahmebehandlung und Dynamisierung – Erörterung am Beispiel medizinisch-organisatorischer Abläufe.* Diplomarbeit, Universität Ulm, 1996

[MHR96] Mitschang, B.; Härder, T.; Ritter, N.: *Design Management in CONCORD: Combining Transaction Management, Workflow Management, and Cooperation Control.* Proc. 6th Int'l Workshop on Res. Iss. in Data Engin. (RIDE'96), New Orleans, 1996, S. 160–168

[MKVC98] Meijler, T.; Kessels, H.; Vuijst, C.; le Comte, R.: *Realising Run-time Adaptable Workflows by Means of Reflection in the Baan Workflow Engine.* In [KDB98]

[Mich99] Michel, R.: *Die Realisierung eines graphischen Workflow-Editors in Java.* Diplomarbeit, Universität Ulm, 1999.

[MüRa99] Müller, R.; Rahm, E.: *Rule-Based Dynamic Modification of Workflows in a Medical Domain.* Proc. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW '99), Freiburg, März 1999, S. 429-448.

[Muth98] Muth, P.; Wodtke, D.; Weißenfels, J.; Kotz-Dittrich, A.; Weikum, G.: *From Centralized Workflow Specification to Distributed Workflow Execution.* Journal of Intelligent Information Systems, Vol. 10, No. 2, März/April 1998, S. 159–184

[MWFF92] Medina-Mora, R.; Winograd, T.; Flores, R.; Flores, F.: *The Action Workflow Approach to Workflow Management Technology*. Proc. ACM Conf. on Comp. Supported Cooperative Work (CSCW '92), Toronto, Kanada, Oktober/ November 1992, S. 281–288

[Myer98] Myers, K.: *Towards a Framework for Continuous Planning and Execution.* Proc. of the AAAI Symposium on Distributed Continual Planning, Menlo Park, CA, 1998.

[Nagl79] Nagl, M.: *Graph-Grammatiken.*Vieweg-Verlag, 1979

[NaOt96] Nastansky, L.; Ott, M.: *Teambasiertes Workflowmanagement und Analyse prozeßorientierter Teamarbeit im Bereich zwischen kooperativer und strukturierter Vorgangsbearbeitung.* Technischer Bericht, Universität Paderborn, 1996

[NeMo93] Neumann, K.; Morlock, M.: *Operations Research*. Carl Hanser Verlag, 1993

[Neum75] Neumann, K.: *Operations Research Verfahren, Band III – Graphentheorie, Netzplantechnik.* Carl Hanser, 1975

[NgCo96] Nguyen, M. Conradi, R.: *Towards a Rigorous Approach for Managing Process Evolution.* Proc. 15th Int'l Conf. on Conceptual Modeling (ER'96), Cottbus, Oktober 1996 (LNCS 1157, Springer-Verlag), S. 18–35

[NGT92] Nierstrasz, O.; Gibbs, S.; Tsichritzis, D.: *Component-Oriented Software Development*, Communications of the ACM, Vol. 35, No. 9, September 1992, S. 160–164

[NiPi95] Nippa, M.; Picot, A. (Hrsg.): *Prozeßmanagement und Reengineering – Die Praxis im deutschsprachigen Raum.* Campus-Verlag, 1995

[NTMS91] Nierstrasz, O.; Tsichritzis, D.; de Mey, V.; Stadelmann, M.: *Object + Scripts = Applications.* Proc. Esprit'91 Conf., Dordrecht, Niederlande, 1991, S. 534–552

[Nutt96] Nutt, G.J.: *The Evolution Towards Flexible Workflow Systems*. Distributed Systems Engineering Journal, Vol. 3, No. 4, 1996, S. 276–294

[Ober90] Oberweis, A.: *Petri-Netz-Beschreibungstechniken für Exception-Handling-Mechanismen in der Automatisierungstechnik.* Institut für Angewandte Informatik und formale Beschreibungsverfahren, Universität Karlsruhe, Bericht 207, November 1990

[Ober96] Oberweis, A.: *Modellierung und Ausführung von Workflows mit Petri-Netzen*, Teubner-Verlag, 1996

[ObSt91] Oberweis, A.; Stucky, W.: *Die Behandlung von Ausnahmen in Software-Systemen.* Wirtschaftsinformatik, 33. Jahrgang, Heft 6, 1991, S. 492–502

[OSS94] Oberweis, A.; Scherrer, G.; Stucky, W.: *INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems.* Information Systems, Vol. 19, No. 8, Dezember 1994, S. 641–658

[OtNa97] Ott, M.; Nastansky, L.: *Groupware-Based Organization Design for Dynamic Workflow Management and Office Systems*. Proc. Business Information Systems '97, 1997, Posen, Polen, April 1997, S. 545-564

[PAKM96] *Proc. 1st Int'l Conf. on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflows*, Basel, Oktober 1996

[Part98] Partsch, H.: *Requirements-Engineering sytematisch – Modellbildung für softwaregestützte Systeme*. Springer, 1998

[Paul93] Paulisch, F.N.: *The Design of an Extentible Graph Editor*. Springer, LNCS 704, 1993.

[Pent94] Pentland, B.: *A Generative Approach to Process Redesign.* Technical Report, CCS Working Paper No. 178, MIT Sloan School.

[Petr62] Petri, C.A.: *Kommunikation mit Automaten*. Dissertation, TH Darmstadt, 1962.

[PiRo95] Picot, A.; Rohrbach, P.: *Organisatorische Aspekte von Workflow-Management-Systemen*. Information Management, Heft 1/95, 1995, S. 28–35

[RBD99] Reichert, M.; Bauer, T.; Dadam, P.: *Enterprise-Wide and Cross-Enterprise Workflow-Management – Challenges and Research Issues for Adaptive Workflows.* In [DaRe99], S. 56–64

[RDMK00] Reichert, M.; Dadam, P.; Mangold, R.; Kreienberg, R.: *Computerbasierte Unterstützung von Arbeitsabläufen im Krankenhaus - Konzepte, Technologien und deren Anwendung.* Zentralbl Gynakol, Vol 122, No. 1, Januar 2000, S. 53–67.

[ReDa97] Reichert, M.; Dadam, P.: *A Framework for Dynamic Changes in Workflow-Management Systems.* In [DEXA97], S. 42–48

[ReDa98] Reichert, M.; Dadam, P.: *$ADEPT_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control.* Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems, Vol. 10, No. 2, März 1998, S. 93–129

[ReDa98a] Reichert, M.; Dadam, P.: *Towards Process-oriented Hospital Information Systems: Some Insights into Requirements, Technical Challenges and Possible Solutions.* Proc. 43. Jahrestagung der GMDS (GMDS'98), Bremen, September 1998, S. 175–180.

[ReDa00] Reichert, M.; Dadam, P.: *Geschäftsprozessmodellierung und Workflow-Management - Konzepte, Systeme und deren Anwendung.* Industrie Management, (Themenheft: Modellierung und Simulation), Vol. 16, No. 3, Juni 2000, S. 23-27

[Rein93] Reinwald, B.: *Workflow-Management in verteilten Systemen*, Teubner-Texte zur Informatik, Band 7, Teubner-Verlag, 1993.

[Reis90] Reisig, W.: *Petri-Netze*. Springer-Verlag, 1990

[ReMo96] Reinwald, B.; Mohan, C.: *Structured Workflow Management with Lotus Notes Release 4.* Proc. 41th IEEE Computer Society Int'l Conf. (CompCon'96), Santa Clara, CA, Februar 1996, S. 451–457

[ReSc95] Reuter, A.; Schwenkreis, F.: (1995). *ConTracts – A Low-Level Mechanism for Building General-Purpose Workflow Management Systems.* In [Hsu95], S. 4–10

[RHD98] Reichert, M.; Hensinger, C.; Dadam, P.: *Supporting Adaptive Workflows in Advanced Application Environments.* Proc. EDBT-Workshop on Workflow Management Systems, Valencia, Spanien, März 1998, S. 100-109

[RiMi97] Ritter, N.; Mitschang, B.: *Die Assistenzfunktion kooperativer Designflows – verdeutlicht am Beispiel von CONCORD.* In [Härd97], S. 91–100

[Ritt94] Ritter, N.; Mitschang, B.; Härder, T.; Gesmann, M.; Schoenning, H.: *Capturing Design Dynamics – The CONCORD Approach.* Proc. of the 10th Int'l Conf. Data Engineering (ICDE'96), Houston, Texas, Februar 1994, S. 440–451

[RKD96] Reichert, M.; Kuhn, K.; Dadam, P.: *Prozeßreengineering und -automatisierung in klinischen Anwendungsumgebungen.* Proc. GMDS'96, Bonn, Sept. 1996, S. 219–223

[RKD97] Reichert, M.; Kuhn, K.; Dadam, P.: *Optimierung und Unterstützung von Leistungsprozessen im Krankenhaus – Perspektiven, Erfahrungen und Grenzen.* Proc. 20. Dt. Krankenhaustag und INTERHOSPITAL/INTERFAB 97, Hannover, Juni 1997, S. 668 –682

[Roll96] Roller, D.: *Verifikation von Workflows in IBM FlowMark.* In [VoBe96], S. 353–368

[Roze97] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. *Volume 1: Foundations*, 1997

[RSD97] Reichert, M.; Schultheiß, B.; Dadam, P.: *Erfahrungen bei der Entwicklung vorgangsorientierter, klinischer Anwendungssysteme auf Basis prozeßorientierter Workflow-Technologie.* Proc. GMDS'97, Ulm, September 1997, S. 181–187

[Rupi97] Rupietta, W.: *Business Processes and Workflow Management*. In: Bernus, P.; Mertins, K; Schmidt, G. (Hrsg.): Handbook on Architectures of Information Systems, Springer-Verlag, 1997

[RuSh94] Rusinkiewicz, M.; Sheth, A.: *Specification and Execution of Transactional Workflows.* In Kim, W. (Hrsg.): Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press and Addison-Wesley, 1994, S. 592–620

[RuWe94] Rupietta, W.; Wernke, G.: *Umsetzung organisatorischer Regelungen in der Vorgangsbearbeitung mit WorkParty und ORM.* In [HKS94], S. 135–154

[RVW97] Reuß, T.; Vossen, G.; Weske, M.: *Modeling Samples Processing in Laboratory Environments as Scientific Workflows*. In [DEXA97], S. 49–55.

[Saas95] Saastamoinen, H.: *On the Handling of Exceptions in Information Systems.* Dissertation, Universität Jyvaskyla, Finnland, 1995

[SaBa93] Saracelli, K.D.; Bandat, K.F.: *Process Automation in Software Application Development.* IBM Systems Journal, Vol. 32, No. 3, 1993, S. 376–396

[SAM91] Sarin, S.; Abbott, K.; McCarthy, D.: *A Process Model and System for Supporting Collaborative Work.* Proc. ACM SIGOIS Conf. on Organizational Computing Systems, Atlanta, November 1991 (SIGOIS Bulletin, Vol. 12, No. 2+3), S. 213-224

[Saue96] Sauer, U.: *Entwicklung einer Endbenutzerschnittstelle für die flexible Bearbeitung medizinisch-organisatorischer Prozesse*. Diplomarbeit, Universität Ulm, 1996

[SaWh95] Saastamoinen, H.; White, G.: *On Handling Exceptions.* In [COOCS95], S. 302–310

[ScBi96] Schreyjak, S.; Bildstein, H.: *Fehlertolerante Abwicklung von Geschäftsprozessen in Workflow-Management Systemen.* Fakultätsbericht 1996/17, Fakultät für Informatik, Universität Stuttgart, Dezember 1996

[Sche96] Scheer, A.-W.: *ARIS-Toolset: Vom Forschungsprototypen zum Produkt*. Informatik-Spektrum, Vol. 19, No. 2, April 1996, S. 71–78

[Sche98a] Scheer, A.-W.: *ARIS – Modellierungsmethoden, Metamodelle, Anwendungen.* 3. Auflage, Springer, 1998

[Sche98b] Scheer, A.-W.: *ARIS – Vom Geschäftsprozeß zum Anwendungssystem.* 3. Auflage, Springer, 1998

[Schn96] Schneider, G.; Maus, H.; Dietel, C.; Scheller-Houy, A.; Schweitzer, J.: *Concepts for a Fexibilisation of Workflow Management Systems with Respect to Task Adaptable Solutions.* Proc. AAAI–Workshop: AI in Business, Electronic Commerce and Reengineering, Portland, Oregon, August 1996

[Schr97] Schröder, R.: *Realisierung von Diensten zur Anpassung von Workflows während der Laufzeit.* Diplomarbeit Nr. 1460, Fakultät für Informatik, Universität Stuttgart, 1997

[Schu96] Schultheiß, B.: *Prozeßreengineering in klinischen Anwendungsumgebungen – Beispiele, Vorgehensmodelle, Werkzeuge*. Diplomarbeit, Universität Ulm, 1996

[Schw93] Schwab, K.: *Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen.* Dissertation, Universität Bamberg, April 1993

[Schw95] Schwab, K.: *Workflow-Management-Systeme: aktuelle Trends und Perspektiven*. Bamberger Beiträge zur Wirtschaftsinformatik, Nr. 32/1995, Universität Bamberg, 1995

[Schw96] Schwab, K.: *Koordinationsmodelle und Software-Architekturen als Basis für die Auswahl und Spezialisierung von WfMS*. In [VoBe96], S. 295–317

[Shet96] Sheth, A.; Kochut, K.; Miller, J.; Worah, D.; Das, S.; Lin, C.; Palaniswami, D.; Lynch, J.; Shevchenko, I.: *Supporting State-Wide Immunization Tracking Using Multi-Paradigm Workflow Technology.* Proc. 22nd Int'l Conf. on Very Large Databases (VLDB'96), Bombay, Indien, September 1996, S. 263–273

[ShKo97] Sheth, A.; Kochut, K.: *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems*. Proc. of the NATO Advanced Study Institute on Workflow Management Systems and Interoperability. Istanbul, August 1997, S. 12-21

[ShRo93] Shedletsky, J.; Rofrano, J.: *Application Reference Designs for Distributed Systems.* IBM Systems Journal, Vol. 32, 1993, S. 625–646

[Sieb96] Siebert, R.: *Adaptive Workflow for the German Public Administration*. In [PAKM96].

[Sieb97] Siebert, R.: *Adaptive Workflows im Verbundprojekt PoliFlow*. Proc. "Workflow-Management in Geschäftsprozessen im Trend 2000". Schmalkalden, Oktober 1997.

[Sieb98] Siebert, R.: *An Open Architecture for Adaptive Workflow Management Systems*. Proc. of the 3rd Biennial World Conference on Integrated Design and Process Technology, Vol 2 - Issues and Applications of Database Techology, Juli 1998, S. 79-85.

[SiWe98] Siebert, R.; Weske, M. (Hrsg.): *Flexibilität und Kooperation in Workflow-Management-Systemen.* Workshop im Rahmen der D-CSCW'98, September 1998, Erschienen als Bericht Nr. 18/98-I, Fakultät für Informatik, Universität Münster, 1998.

[Stip98] Van Stiphout, R.; Meijler, T.; Aerts, A.; Hammer, D.; le Comte, R.: *TREX*: *Workflow Transactions by Means of Exceptions*. In [BES98], S. 21-26

[StMi95] Strong, D.M.; Miller, S.M.: *Exceptions and Exception Handling in Computerized Information Processes*, ACM Trans. Inf. Sys., Vol. 13, No. 2, April 1995, S. 206–233

[Ston94] Stonebraker, M. (Hrsg.): *Readings in Database Systems*. Second Edition, Morgan Kaufman Publ., 1994

[Stro97] Strobel-Vogt, U.: *SAP Business Workflow in der Logistik.* Vieweg-Verlag, 1997

[Such87] Suchman, L.: *Plans and Situated Actions*. Cambridge University Press, 1987

[Swen94] Swenson, K.: *The Future of Workflow Technology*: *Collaborative Planning.* Proc. Conf. on Groupware (Groupware'94), San Jose, Kalifornien, August 1994

[Swen94a] Swenson, K.; Irwin, K.; Maxwell, R.; Matsumoto, T.; Saghari, B.: *A Business Process Environment to Support Collaborative Planning*. Journal of Collab. Comp., 1994.

[Swen93] Swenson, K.: *Visual Support for Reengineering Work Processes.* Proc. Conf. of Organizational Computing Systems (COOCS'93), Milpitas, Kalifornien, Nov. 1993

[TaHw98] Tang, J.; Hwang, S.-Y.: *A Scheme to Specify and Implement Ad-Hoc Recovery in Workflow Systems.* Proc. 6th Int'l Conf. on Extending Database Technology (EDBT'98), Valencia, März 1998, S. 484–498

[Teeg96] Teege, G.: *HieraStates*: *Supporting Workflows Which Include Schematic and Ad-hoc Aspects.* In [PAKM96].

[Teeg96a] Teege, G.: *Object-Oriented Activity Support: A Model for Integrated CSCW Systems.* Computer Supported Cooperative Work, Vol. 5, No. 1, 1996, S. 93–124

[Teeg98] Teege, G.: *Flexible Workflows: Mitgestaltung durch die Ausführenden.* In [SiWe98], S. 13–21.

[TLA91] Tueni, M.; Li, J.; Ang, J.: *Knowledge-Based Office Automation and CSCW.* In Bowers, J.; Benford, S. (Eds.): Studies in Computer-Supported Cooperative Work, Elsevier Science Publ. (North-Holland), 1991, S. 183–194.

[Tsich82] Tsichritzis, D.: *Form Management*, Comm. of the ACM, Vol. 25, 1982, S. 453–478

[Tsich85] Tsichritzis, D. (Hrsg.): *Office Automation*. Springer-Verlag, 1985

[Valk81] Valk, R.: *Generalizations of Petri Nets.* LNCS, Vol. 118, Springer, 1981, S. 140–155.

[ViSo91] Victor, F.; Sommer, E.: *Supporting the Design of Office Procedures in the DOMINO System.* In Bowers, J. et al. (Eds.): Studies in Computer-Supported Cooperative Work, Elsevier Science Publ. B.V. (North-Holland), 1991, S. 119-130

[VLP95] Veijalainen, J.; Lehtola, A.; Pihlajamaa, O.: *Research Issues in Workflow Systems*. In 8th ERCIM Database Research Group Workshop on Database Issues and Infrastructure in Cooperative Systems (EDRG–8), Trondheim, Norwegen, August 1995

[VMJ97] Vilela, P.; Maldonado, J.; Jino, M.: *Program Graph Visualization.* Software-Practice and Experience, Vol. 27, No. 1, November 1997, S. 1245-1267

[VoAa97] Voorhoeve, M.; van der Aalst, W.: *Ad-hoc Workflow: Problems and Solutions*. In [DEXA97], S. 36–41

[VoBe96] Vossen, G.; Becker, J.: *Geschäftsprozeßmodellierung und Workflow-Management*. Thomson Publ., 1996

[VoEr92] Vogel, P.; Erfle, R.: *Backtracking Office Procedures*. Proc. 15th Int'l Conf. on Database and Expert Systems (DEXA '92), Valencia, 1992, S. 506–511

[VWW96] Vossen, G.; Weske, M.; Wittkowski, G.: *Dynamic Workflow Management on the Web*. Fachbericht Nr. 24/96–I, Fakultät für Informatik, Universität Münster, 1996

[Wäch95] Wächter, H. et al.: *Modellierung und Ausführung flexibler Geschäftsprozesse mit SAP Business Workflow 3.0.* Proc. 25. GI–Jahrestagung und 13. Schweizer Informatikertag (GISI'95), Zürich, September 1995 (Informatik Aktuell, Springer-Verlag), S. 197–204

[Wäch96] Wächter, H.: *Eine Architektur für die zuverlässige Abwicklung verteilter Anwendungen auf gemeinsamen Ressourcen.* Dissertation, Universität Stuttgart, 1996

[WäRe92] Wächter, H.; Reuter, A.: *The ConTract Model.* In [Elma92], S. 219–263

[Webe97] Weber, M.; Partsch, G.; Scheller-Huoy, A.; Schweitzer, J.; Schneider, G.: *Flexible Real-time Meeting Support for Workflow Management Systems*, Proc. 30th Hawaii Int'l Conf. on System Sciences HICSS, Maui, Hawaii, 1997

[Weik97] Weikum, G.; Wodtke, D.; Kotz-Dittrich A.; Muth, P.; Weißenfels, J.: *Spezifikation, Verifikation und verteilte Ausführung von Workflows in MENTOR.* In [Härd97], S. 61–71

[Weil97] Weilbach, P.: *Implementierungsaspekte zur Verwaltung und Synchronisation dynamischer Änderungen in prozeßorientierten Workflow-Management-Systemen.* Diplomarbeit, Universität Ulm, 1997

[Wesk97] Weske, M.: *State-based Modeling of Flexible Workflow Executions in Distributed Environments*. Fachbericht Angewandte Mathematik und Informatik Nr. 13/97-I, Universität Münster, 1997

[Wesk98] Weske, M.: *Flexible Modeling and Execution of Workflow Activities*. Proc. 31st Hawaii Int'l Conf. on System Sciences (HICSS–31), Software Technology Track (Vol VII), 1998, S. 713–722

[Wesk99] Weske, M.: *Adaptive Workflows based on Flexible Assignment of Workflow Schemas and Workflow Instances.* In [DaRe99], S. 42–48

[WFMC95] Workflow Management Coalition: *The Workflow Reference Model.* WFMC-TC00-1003, Version 1.1, Januar 1995

[WFMC96] Workflow Management Coalition: *Terminology & Glossary*. WFMC–TC–1011, Version 2.0, Juni 1996

[WFMC96a] Workflow Management Coalition: *Interface 2: Workflow Client Application Programming Interface (WAPI) Specification*. WFMC–TC–1009, Version 1.2, Okt. 1996

[WFMC96b] Workflow Management Coalition: *Interface 4: – Interoperability Abstract Specification.* WFMC–TC–1012, Version 1.0, Oktober 1996

[WFMC96c] Workflow Management Coalition: *Interface 5: Audit Data Specification.* WFMC–TC–1015, Version 1.0, November 1996

[WFMC98] Workflow Management Coalition: *Interface 1 - Process Definition Interchange*. WFMC-TC-1016, Version 7.04, November 1998

[WFMC98a] Workflow Management Coalition: *Workflow Management Application Programming Interface (Interface 2 & 3) Specification.* WFMC-TC-1009, Version 2.0, Juli 1998

[WFMC98b] Workflow Management Coalition: *Interface 4*: *Interoperability - Internet e-mail MIME Binding.* WFMC-TC-1018, Version 1.1, September 1998

[WFMC98c] Workflow Management Coalition: *Interface 5 - Audit Data Specification.* WFMC-TC-1015, Version 1.1, September 1998

[WHKS98] Weske, M.; Hündling, J.; Kuropka, D.; Schuschel, H.: *Objektorientierter Entwurf eines flexiblen Workflow-Management-Systems.* Inf. Forsch Entw, Vol. 13, 1998, S. 179-195.

[WiFl86] Winograd, T.; Flores, F.: *Understanding Computers and Cognition: A New Foundation For Design*. Ablex Publishing Corporation, Norwood, NJ, 1986

[Wilk85] Wilkins, D.: *Recovering From Execution Errors in SIPE.* Computational Intelligence, Vol. 1, No. 1, 1985, S. 33–45.

[Wilk88] Wilkins, D.: *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufman Publ., San Mateo, CA, 1988

[WMLW95] Wilkins, D.; Myers, K.; Lowrance, J.; Wesley, L.: *Planning and Reacting in Uncertain and Dynamic Environments.* Journal of Experimental and Theoretical AI, Vol. 7, No. 1, 1995, S. 197–227.

[Wodt95] Wodtke, D.; Kotz-Dittrich, A.; Muth, P.; Sinnwell, M.; Weikum, G.: *Mentor: Entwurf einer Workflow-Management-Umgebung basierend auf State- und Activitycharts*. Proc. BTW '95, Dresden, März 1995 (Informatik Aktuell, Springer-Verlag), S. 71–90

[Wolp97] Wolpert, A.: *Ein ereignisorientiertes Programmiermodell auf der Basis von Workflow-Management-Technologie zur Realisierung integrierter medizinischer Arbeitsplätze.* Diplomarbeit, Universität Ulm, 1997

[WoSh96] Worah, D.; Sheth, A.: *What do Advanced Transaction Models Have to Offer for Workflows?* Proc. Int'l Workshop Advanced Transaction Models and Architect., 1996

[WoSh97] Worah, D.; Sheth, A.: *Transactions in Transactional Workflows.* In [JaKe97], S. 3-34

[WoWe97] Wodtke, D.; Weikum, G.: *A Formal Foundation for Distributed Workflow Execution Based on State Charts.* Proc. 6th Int'l Conf. on Database Theory (ICDT'97). Delphi, Griechenland, Januar 1997 (LNCS 1186, Springer-Verlag),

[Wulf97] Wulf, V.: *Konfliktmanagement bei Groupware*. Vieweg-Verlag, 1997

[WWWK96] Wodtke, D., Weissenfels, J., Weikum, G., Kotz-Dittrich, A.: *The Mentor Project: Steps Towards Enterprise-Wide Workflow Management.* Proc. 12th Int'l Conf. on Data Engineering, New Orleans, März 1996, S. 556–565

[Zism77] Zisman, M.: *Representation, Specification and Automation of Office Procedures.* Dissertation, Wharton School, University of Pennsylvania, Philadelphia, 1977

Teil V

Anhang

Anhang A: Übersicht zu wichtigen Funktionen

$c_succ(n)$ ($c_pred(n)$)	Menge aller *direkten* Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten des Typs `CONTROL_E`.
$c_succ^*(n)$ ($c_pred^*(n)$)	Menge aller *direkten* oder *indirekten* Nachfolgerknoten von n bzgl. normaler Kontrollkanten des Typs `CONTROL_E`.
$succ(n)$ ($pred(n)$)	Menge aller *direkten* Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten und Sync-Kanten
$succ^*(n)$ ($pred^*(n)$)	Menge aller *direkten* oder *indirekten* Nachfolgerknoten von n bzgl. normaler Kontrollkanten und Sync-Kanten
$join^{CFS}(s)$ ($split^{CFS}(j)$)	Im KF-Graphen *CFS* zugehöriger Join-Knoten (Split-Knoten) zum Split-Knoten s (Join-Knoten j).
$endloop^{CFS}(L_S)$ ($startloop^{CFS}(L_E)$)	Im KF-Graphen *CFS* zugehöriger Schleifenendknoten (Schleifenanfangsknoten) des Schleifenanfangsknotens (Schleifenendknotens) L_S (L_E).
Id_{source}^{e} (Id_{dest}^{e})	Quellknoten (Zielknoten) der Kante e
$BranchNodes^{CFS}(n_1, n_2)$	Bestimmt den Split- und Join-Knoten (s, j) des kleinsten, die beiden Knoten n_1 und n_2 umschließenden Verzweigungsblocks
$MinBlock^{CFS}(N^*)$	Minimaler Kontrollblock im KF-Graphen *CFS*, der alle Knoten der Menge N^* umschließt
$WriterExists^{CFS, DFS}(n, d)$	Kann ein obligater Lesezugriff des Knotens n auf das Datenelement d bei dem gegebenen Schema (*CFS*, *DFS*) sicher versorgt werden?
$ParallelWriterExists^{CFS, DFS}(n, d)$	Gibt es einen zu n parallelen Knoten von dem eine Schreibkante auf das Datenelement d ausgeht?
$ReadDataElement(...,n, d)$	Kontextabhängiger Wert für den Lesezugriff des Knotens n auf d

Anhang B: Beweise

B.1 Beweis zu Satz 3-1

Satz 3-1 (Transitive Reduktion von KF-Graphen)

Sei $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter wohlstrukturierter, azyklischer KF-Graph. Seien ferner $n, n^* \in N$ zwei Aktivitätenknoten für die gelte:

(V1) Es gibt eine Kantenfolge $(n, n_1, \texttt{CONTROL_E}), (n_1, n_2, \texttt{CONTROL_E}),.., (n_{k-1}, n^*, \texttt{CONTROL_E})$ von n nach n^* mit $n_1 \neq n^*$ (d.h. die Kantenfolge besitzt mindestens die Länge 2)

(V2) Falls $n \in SplitNodes^{CFS}$, so gelte $n^* \neq join^{CFS}(n)$.

Dann folgt: $n^* \notin c_succ(n)$, d.h. es gibt keine Kontrollkante aus E die direkt von n nach n^* führt. Wir nennen CFS transitiv reduziert bzgl. Kanten des Typs ET = CONTROL_E.

Beweis (durch Widerspruch):

Annahme: $n^* \in c_succ(n)$, d.h. $\exists\, e = (n, n^*, \texttt{CONTROL_E}) \in E$ ①

Wegen $e' = (n, n_1, \texttt{CONTROL_E}) \in E$ und $n_1 \neq n^*$ (Voraussetzung V1) besitzt der Knoten n mindestens zwei ausgehende Kanten des Typs ET = CONTROL_E (e und e'). Es gilt somit: $n \in SplitNodes^{CFS}$ (vgl. Regel KF-2). Entsprechend läßt sich zeigen, daß n^* mindestens zwei einmündende Kanten des Typs ET = CONTROL_E besitzt, d.h. es gilt: $n^* \in JoinNodes^{CFS}$. Sei nun $n_{join} := join^{CFS}(n)$ der zu n zugehörige Join-Knoten und $n_{split}^* := (join^{CFS})^{-1}(n^*)$ der zu n^* zugehörige Split-Knoten.

Dann liefert die Anwendung der Eigenschaft (1) von Regel KF-3:
$n_{join} \in c_succ^*(z_i) \vee n_{join} = z_i \;\forall z_i \in c_succ(n)$
Wegen $n^* \in c_succ(n)$ (vgl. Annahme ①) gilt insbesondere: $n_{join} \in c_succ^*(n^*) \vee n_{join} = n^*$. Da gemäß Voraussetzung V2 gilt $n^* \neq join^{CFS}(n)$ (= n_{join}), folgt somit: $n_{join} \in c_succ^*(n^*)$ ②

Aus der „inversen" Anwendung der Eigenschaft (1) von Regel KF-3 erhält man entsprechend: $n_{split}^* \in c_pred^*(z_i) \vee n_{split}^* = z_i \;\forall z_i \in c_pred(n^*)$. Wegen $n \in c_pred(n^*)$ (was gleichbedeutend ist mit $n^* \in c_succ(n)$, vgl. Annahme ①) gilt insbesondere: $n_{split}^* \in c_pred^*(n) \vee n_{split}^* = n$. Mit Voraussetzung V2 folgt: $n_{split}^* \in c_pred^*(n)$ und dementsprechend $n \in c_succ^*(n_{split}^*)$ ③

Aus ③ ergibt sich somit: $n \in c_succ^*(n_{split}^*)$. Da trivialerweise auch $n \in c_pred^*(n^*)$ gilt (wegen Voraussetzung V1) folgt sofort: $n \in c_succ^*(n_{split}^*) \cap c_pred^*(n^*)$.

Da $n^* = join(n_{split}^*)$ gilt, ergibt sich durch Anwendung der Eigenschaft (3) von Regel KF-3: $n_{join} \in c_succ^*(n_{split}^*) \cap c_pred^*(n^*)$ und damit auch: $n_{join} \in c_pred^*(n^*)$ ④

Aus ② + ④ ergibt sich: $n_{join} \in c_succ^*(n^*) \wedge n_{join} \in c_pred^*(n^*)$. Daß n_{join} sowohl Nachfolger als auch Vorgänger von n^* ist, wäre nur möglich, wenn Zyklen im KF-Graphen erlaubt wären. Sie sind aber nach Regel KF-4 ausgeschlossen, womit sich ein Widerspruch zur Annahme ① ergibt.

q.e.d.

B.2 Beweis zu Satz 3-2

Satz 3-2 (Verbundenheit von Aktivitätenknoten mit Start- und Endknoten)

Sei $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter wohlstrukturierter, azyklischer KF-Graph.

a) Für $n \in N, n \neq Start$ gilt stets: $n \in c_succ^*(Start)$, d.h. der Knoten n ist vom Startknoten $Start$ aus über eine Kantenfolge erreichbar.

b) Für $n \in N, n \neq End$ gilt stets: $End \in c_succ^*(n)$; d.h. der Endknoten End ist von jedem anderen Knoten n über ein Kantenfolge erreichbar. Insbesondere gilt: $End \in c_succ^*(Start)$.

Beweis:

Wir beweisen Aussage a) durch Widerspruch: Dazu nehmen wir an, daß es einen Knoten $n^* \in N$, $n^* \neq Start$ gibt, für den gilt: $Start \notin c_pred^*(n^*)$ (*).

Generell besitzt jeder Knoten $k \in N$, $k \neq Start$ mindestens einen direkten Vorgänger im KF-Graphen bzgl. Kanten des Typs ET = `CONTROL_E` (Regeln KF-1 und KF-2). Dem entsprechend hat auch der Knoten n^* mindestens einen direkten Vorgänger n_1. Wegen Annahme (*) muß gelten $n_1 \neq Start$, was bedeutet, daß auch n_1 mindestens einen direkten Vorgänger $n_2 \neq Start$ besitzt. Nach demselben Prinzip ließe sich eine unendliche Folge $n^*, n_1, n_2, n_3, \ldots$ ($n^*, n_\upsilon \neq Start$ für $\upsilon = 1, 2, \ldots$) miteinander verbundener Knoten konstruieren. Da die Knotenmenge N endlich ist (vgl. Definition 3-1), würde dies bedeuten, daß ein Zyklus vorliegt, was jedoch nach Regel KF-4 für normale Kontrollkanten ausgeschlossen ist. Demnach ist die Annahme (*) falsch. Der Beweis von b) kann nach demselben Prinzip erfolgen.

q.e.d.

B.3 Beweis zu Satz 3-3

Satz 3-3 (Unabhängigkeit von Teilzweigen in wohlstrukturierten, azyklischen KF-Graphen)

Sei $CFS = (N, E, \ldots) \in CFS_{simple}$ ein korrekter KF-Graph. Sei ferner $s \in SplitNodes^{CFS}$ ein Split-Knoten aus N mit zugehörigem Join-Knoten $j := join^{CFS}(s)$. Dann gilt für zwei beliebige Knoten $n_1, n_2 \in N$, die unterschiedlichen Zweigen der durch s und j gebildeten Verzweigung angehören, stets: ($n_1, n_2,$ `CONTROL_E`) $\notin E$, d.h. n_1 und n_2 können nicht über eine normale Kontrollkante miteinander verbunden sein.

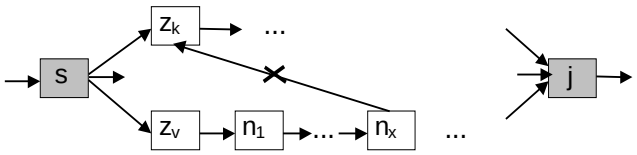
O.B.d.A. besitze die durch die Knoten s und j gebildete Verzweigung keine leeren Zweige. Das bedeutet, daß $\forall\, z_i \in c_succ(s)$ gilt: $z_i \neq j$ (d.h. $z_i \in c_pred^*(j)$). Die Aussage von Satz 3-3 läßt sich formal dann wie folgt fassen:

Für jedes $m \in (c_succ^*(z_i) \cup \{ z_i \}) \cap c_pred^*(j)$ gilt stets:

$$A(m):\ \forall\, k \neq i, \forall\, n \in (c_succ^*(z_k) \cup \{ z_k \}) \cap c_pred^*(j): n \notin c_pred(m) \wedge n \notin c_succ(m)$$

Zur besseren Strukturierung des Beweises von Satz 3-3 benötigen wir die folgenden beiden Lemmata: Lemma A sagt aus, daß es zwischen den Startknoten z_k und z_l zweier verschiedener von s ausgehenden „Zweige“ keine Verbindung über eine Kantenfolge $z_k \rightarrow \ldots \rightarrow z_l$ geben kann. Lemma B besagt, daß es vom Startknoten z_k eines von s ausgehenden Zweiges keine Kontrollkante zu einem Knoten n gibt, der seinerseits über eine Kantenfolge $z_l \rightarrow \ldots \rightarrow n$ erreichbar ist.

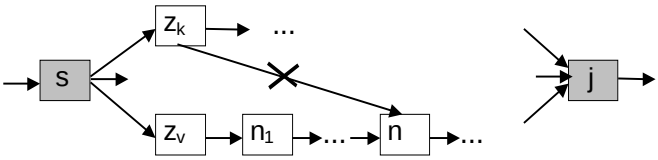
Lemma A: Sei $s \in SplitNodes^{CFS}$. Für $z_v, z_k \in c_succ(s)$, $v \neq k$; und $z_v, z_k \neq join^{CFS}(s)$ gilt stets: $z_k \notin c_succ^*(z_v)$



Beweis von Lemma A (durch Widerspruch):

Wir nehmen an, daß $z_k \in c_succ^*(z_v)$ gilt, d.h. es gibt eine Kantenfolge $z_v \rightarrow n_1 \rightarrow \ldots\ n_x \rightarrow z_k$ die von z_v nach z_k führt. Demzufolge gibt es eine Kantenfolge $s \rightarrow z_v \rightarrow n_1 \rightarrow \ldots\ n_x \rightarrow z_k$ die von s über z_v nach z_k führt. Wegen $z_k \neq join^{CFS}(s)$ und Satz 3-1 (→ transitive Reduktion von Kanten des Typs ET = `CONTROL_E`) folgt dann sofort: $z_k \notin c_succ(s)$, was jedoch im Widerspruch zur Voraussetzung $z_k \in c_succ(s)$ steht. Folglich ist die Annahme $z_k \in c_succ(z_v)$ falsch. q.e.d.

Lemma B: Sei $s \in SplitNodes^{CFS}$ und seien $z_k, z_v \in c_succ(s)$, $k \neq v$ und $z_k, z_v \neq j := join^{CFS}(s)$. Dann gilt für jeden Knoten $n \in (c_succ^*(z_v) \cup \{z_v\}) \cap c_pred^*(join^{CFS}(s))$: $n \notin c_succ(z_k)$



<u>Beweis von Lemma B</u> (durch Widerspruch)

Wir nehmen an, daß die Aussage des Lemmas unter den genannten Voraussetzungen nicht richtig ist:

$\exists\, z_k, z_v \in c_succ(s), k \neq v$ und $z_k, z_v \neq join^{CFS}(s)$ mit:
$\exists\, n^* \in (c_succ^*(z_v) \cup \{z_v\}) \cap c_pred^*(join^{CFS}(s)) : n^* \in c_succ(z_k)$ (*)

Gemäß Annahme (*) gibt es somit eine gerichtete Kante des Typs `CONTROL_E` von z_k nach n^* und es gilt: $n^* \in c_succ^*(z_v) \cup \{z_v\}$. Mit Lemma A folgt sofort $n^* \neq z_v$ und damit: $n^* \in c_succ^*(z_v)$.

Das bedeutet, daß es eine Kantenfolge $\pi = z_v =: n_0 \rightarrow n_1 \rightarrow ... \rightarrow n_{l-1} \rightarrow n_l := n^*$ von z_v nach n^* gibt, für deren Endknoten n^* gilt: $n^* \in c_succ(z_k)$ (im einfachsten Fall erhalten wir $\pi = z_v \rightarrow n^*$). Des weiteren kann nach Lemma A der Knoten z_k nicht in dieser Kantenfolge auftreten, d.h. es gilt: $n_x \neq z_k$ für $x = 0, 1,..., l$. Der Knoten n^* besitzt somit mindestens zwei einmündende Kontrollkanten: Die Kante $e' = (n_{l-1}, n^*,$ `CONTROL_E`$) \in E$, $n_{l-1} \neq z_k$ und gemäß Annahme (*) die Kante $e = (z_k, n^*,$ `CONTROL_E`$)$. Das bedeutet, daß n^* ein Join-Knoten ist, d.h. es gilt $n^* \in JoinNodes^{CFS}$. ①

Für den zu n^* gehörigen (eindeutigen) Split-Knoten $n_{split}^* := split(n^*)$ gilt in jedem Fall, daß n_{split}^* ein Nachfolger des Split-Knotens s und ein Vorgänger des Join-Knotens j ist. Dies folgt unmittelbar aus den Voraussetzungen zu (*) und aus der Anwendung von Regel KF-3, (3). Wir halten somit fest: $n_{split}^* \in c_succ^*(s) \cap c_pred^*(j)$ ②

Gemäß Regel KF-3, (1) gilt $\forall\, j_i \in c_pred(n^*)$: $n_{split}^* \in c_pred^*(j_i) \vee n_{split}^* = j_i$. Da insbesondere $z_k, n_{l-1} \in c_pred(n^{**})$ gilt, ergibt sich aus der Anwendung dieser Regel für den Knoten n_{split}^*:

$$\left(n_{split}^* \in c_pred^*(z_k) \vee n_{split}^* = z_k\right) \wedge \left(n_{split}^* \in c_pred^*(n_{l-1}) \vee n_{split}^* = n_{l-1}\right) \quad (**)$$

Der Knoten n_{l-1} scheidet dabei von vornherein als Split-Knoten von n^* aus, d.h. es gilt $n_{split}^* \neq n_{l-1}$. Grund ist, daß in (**) weder $n_{l-1} \in c_pred^*(z_k)$ noch $n_{l-1} = z_k$ gelten können: Erstere Möglichkeit scheidet aus, da wegen $z_v \in c_pred^*(n_{l-1})$ dann sofort $z_v \in c_pred^*(z_k)$ folgen würde, was jedoch wegen Lemma A nicht der Fall sein kann. Die zweite Möglichkeit $n_{l-1} = z_k$ ist bereits durch ① ausgeschlossen worden. Der Ausdruck aus (**) läßt sich somit reduzieren auf:

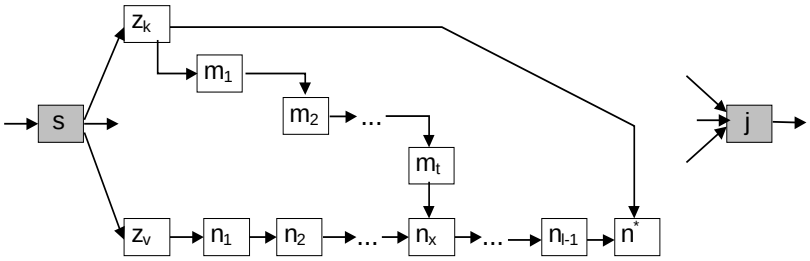
$$\left(n_{split}^* \in c_pred^*(z_k) \vee n_{split}^* = z_k\right) \wedge n_{split}^* \in c_pred^*(n_{l-1})$$

$$\equiv \left(n_{split}^* \in c_pred^*(z_k) \cap c_pred^*(n_{l-1})\right) \vee \left(n_{split}^* = z_k \wedge z_k \in c_pred^*(n_{l-1})\right) \quad (***)$$

Wir zeigen nun durch Widerspruch, daß der Ausdruck $n_{split}^* = z_k \wedge z_k \in c_pred^*(n_{l-1})$ nicht wahr ist.

<u>Einschub</u>: Wir nehmen an, daß $n_{split}^* = z_k \wedge z_k \in c_pred^*(n_{l-1})$ gilt. ③

Aus $z_k \in c_pred^*(n_{l-1})$ folgt, daß es (mindestens) eine Kantenfolge von z_k nach n_{l-1} gibt. Eine dieser Kantenfolgen sei wie folgt konstruiert (vgl. nachfolgende Abbildung): $z_k =: m_0 \rightarrow ... \rightarrow m_t \rightarrow n_x \rightarrow n_{x+1} \rightarrow ... \rightarrow n_{l-1}$, wobei $x \in \{1,..., l\text{-}1\}$ so gewählt sei, daß gilt: $z_k \in c_pred^*(n_r)$ für $r = x, x+1, ..., l\text{-}1$ <u>und</u> $z_k \notin c_pred^*(n_r)$ für $r = 0, ..., x\text{-}1$ ④
(Wegen $z_k \in c_pred^*(n_{l-1})$ und $z_k \notin c_pred^*(n_0)$, $n_0 := z_v$ läßt sich stets eine solche Folge konstruieren).



Aus ④ folgt, daß n_x mindestens zwei einmündende Kontrollkanten besitzt, nämlich (m_l, n_x, CONTROL_E) und (n_{x-1}, n_x, CONTROL_E), $n_{x-1} \neq m_l$; demzufolge ist n_x ein Join-Knoten. Wegen $n_x \in c_succ^*(z_k) \cap n_x \in c_pred^*(n^*)$ und $n^* = join(z_k)$ ($\Leftrightarrow n_{split}^* = z_k$, vgl. Annahme ③) ergibt sich bei Anwendung von Regel KF-3, (3) für den zu n_x gehörigen Split-Knoten:

$$split_{nx} = (join^{CFS})^{-1}(n_x) \in c_succ^*(z_k) \cap c_pred^*(n^*) \qquad ⑤$$

Da $n_{x-1} \in c_pred(n_x)$ gilt, liefert die Anwendung von Regel KF-3, (1):

$$split_{nx} \in c_pred^*(n_{x-1}) \cup \{n_{x-1}\} \quad \Leftrightarrow n_{x-1} \in c_succ^*(split_{nx}) \cup \{split_{nx}\} \quad ⑥$$

Aus ⑤ + ⑥ folgt: $n_{x-1} \in c_succ^*(z_k)$ (bzw. $z_k \in c_pred^*(n_{x-1})$). Dies stellt jedoch einen Widerspruch zu der in ④ getroffenen Festlegung dar. Demensprechend muß die eingangs gemachte Annahme ③ ($n_{split}^* = z_k \wedge z_k \in c_pred^*(n_{l-1})$ falsch sein).

Damit läßt sich (***) schließlich reduzieren zu:

$$n_{split}^* \in c_pred^*(z_k) \cap c_pred^*(n_{l-1}) =^{\blacklozenge} c_pred^*(s) \cup \{s\}$$

Dies steht jedoch im Widerspruch zu ②, wonach gilt: $n_{split}^* \in c_succ^*(s)$ ($\cap c_pred^*(j)$). Somit muß die eingangs dieses Beweises gemachte Annahme (*) falsch sein.

($^{\blacklozenge}$Wie sich leicht zeigen läßt, besitzt z_k neben der Kante (s, z_k, CONTROL_E) keine weiteren einmündenden Kontrollkanten des Typs CONTROL_E)

q.e.d.

<u>Beweis zu Satz 3-3</u> (durch Widerspruch):

Wir führen nun den Beweis von Satz 3-3 durch Widerspruch. Dazu nehmen wir an, daß die Aussage dieses Satzes für (mindestens) einen von s ausgehenden Zweig nicht zutrifft (o.B.d.A. sei dies der Zweig mit dem Startknoten z_1). D.h. wir nehmen an:

$\exists\, n^* \in (c_succ^*(z_1) \cup \{z_1\}) \cap c_pred^*(j)$ mit: $A(n^*)$ trifft <u>nicht</u> zu

:⇔

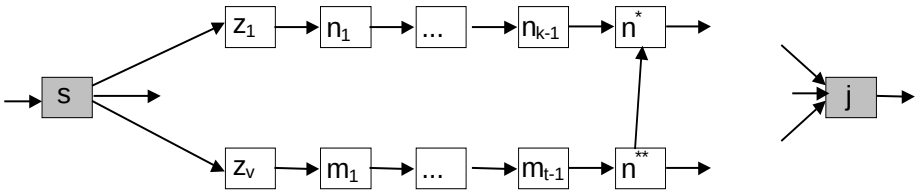
$\exists\, n^* \in (c_succ^*(z_1) \cup \{z_1\}) \cap c_pred^*(j)$ mit:
$\exists\, v \neq 1, \exists\, n^{**} \in (c_succ^*(z_v) \cup \{z_v\}) \cap c_pred^*(j)$ mit $n^{**} \in c_pred(n^*) \vee n^{**} \in c_succ(n^*)$ (♥)

Mit Lemma B folgt sofort $n^* \neq z_1$, d.h. es gilt: „$A(z_1)$ ist wahr"! Entsprechend Annahme (♥) muß es dann (mindestens) einen Nachfolger n^* von z_1 geben, für den gilt: „$A(n^*)$ ist falsch". (Für den Fall, daß es mehrere Nachfolger n von z_1 gibt mit „$A(n)$ ist falsch", wähle man für n^* das n, das auf dem kürzesten Wege (d.h. der kürzesten Kantenfolge) von z_1 aus erreichbar ist). Damit ist sichergestellt, daß es eine Kantenfolge $s \rightarrow z_1 =: n_0 \rightarrow n_1 \rightarrow ... \rightarrow n_k := n^*$, $k \geq 1$(mit Kanten des Typs CONTROL_E) von s über z_1 nach n^* gibt, für deren Knoten n_x, $x = 0, ..., k$-1 gilt: „$A(n_x)$ trifft zu" und für deren Endknoten $n_k = n^*$ gilt: „$A(n_k)$ ist falsch". Mit anderen Worten: Die Knoten $n_0, n_1, ..., n_{k-1}$ besitzen jeweils

- keine einmündende Kontrollkante (n, n_x, CONTROL_E), deren Quellknoten n vom Startknoten z_v, $v \neq 1$ eines anderen Zweiges aus erreichbar bzw. identisch mit diesem Startknoten ist und
- keine ausgehende Kontrollkante (n_x, n, CONTROL_E), deren Zielknoten n vom Startknoten z_v, $v \neq 1$ eines anderen Zweiges aus erreichbar (bzw. identisch mit diesem Startknoten) ist (♠)

Dies gilt jedoch nicht mehr für den Knoten n^*. O.B.d.A. besitze dieser Knoten eine einmündende Kante $e' = (n^{**}, n^*, \text{CONTROL_E}) \in E$ für die gilt, daß ihr Quellknoten n^{**} vom Startknoten z_v, $v \neq 1$ eines anderen Zweiges aus erreichbar ist (vgl. Abbildung).

(Anmerkung: Die Beweisführung für ausgehende Kontrollkanten erfolgt ähnlich, so daß wir an dieser Stelle darauf verzichten).



Demzufolge besitzt der Knoten n^* (mindestens) zwei verschiedene Eingangskontrollkanten des Typs ET = CONTROL_E: $e' = (n^{**}, n^*, \texttt{CONTROL_E}) \in E$ und $e'' = (n_{k-1}, n^*, \texttt{CONTROL_E}) \in E$. Dabei gilt: $n^{**} \neq n_{k-1}$! Wäre dies nicht der Fall, würde $n^{**} = n_{k-1} \in c_succ^*(z_v) \cup \{z_v\}$, $v \neq 1$ gelten (vgl. ♥), was jedoch ein Widerspruch zu ♠) darstellen würde.

Der Knoten n^* ist somit per Definition ein Join-Knoten ($n^* \in JoinNodes^{CFS}$), der – gemäß Annahme (♥) – ein Nachfolger des Split-Knotens s und ein Vorgänger des Join-Knotens j ist. Demzufolge muß für den zu n^* zugehörigen Split-Knoten $n_{split}^* := (join^{CFS})^{-1}(n^*)$ nach Regel KF-3, (3) gelten:

$$n_{split}^* \in c_succ^*(s) \cap c_pred^*(j) \qquad (♦)$$

Mit Regel KF-3, (1) ergibt sich weiter: $\forall j_i \in c_pred(n^{**})$: $n_{split}^* \in c_pred^*(j_i) \vee n_{split}^* = j_i$. Da insbesondere $n^{**}, n_{k-1} \in c_pred(n^*)$ gilt, folgt für den Knoten n_{split}^*:

$$\left(n_{split}^* \in c_pred^*(n^{**}) \vee n_{split}^* = n^{**}\right) \wedge \left(n_{split}^* \in c_pred^*(n_{k-1}) \vee n_{split}^* = n_{k-1}\right) \qquad (♣)$$

Der Knoten n^{**} scheidet dabei von vornherein als Split-Knoten von n^* aus, d.h. es gilt $n_{split}^* \neq n^{**}$. Grund dafür ist, daß weder $n^{**} \in c_pred^*(n_{k-1})$ noch $n^{**} = n_{k-1}$ (vgl. oben) gelten können. In beiden Fällen ergäbe sich ein Widerspruch zu (♠). Ebensowenig kann in (♣) $n_{split}^* = n_{k-1}$ gelten, da weder $n_{k-1} \in c_pred^*(n^{**})$ noch $n_{k-1} = n^{**}$ – auch hier würde in beiden Fällen ein Widerspruch zu (♠) vorliegen – erfüllbar sind. Dementsprechend reduziert sich (♣) zu:

$$n_{split}^* \in c_pred^*(n^{**}) \cap c_pred^*(n_{k-1}) \qquad (♣')$$

Aus (♦) + (♣') ergibt sich dann:

$$n_{split}^* \in M := c_pred^*(n^{**}) \cap c_pred^*(n_{k-1}) \cap c_succ^*(s) \cap c_pred^*(j)$$

Wie man leicht zeigen kann, gilt $M = \emptyset$ (siehe unten). Die eingangs gemachte Annahme (♥) kann somit nicht richtig sein! q.e.d.

(Bleibt noch zu zeigen $M = \emptyset$: Es gilt $M = M_1 \cap M_2$ mit

$M_1 = c_pred^*(n_{k-1}) \cap c_succ^*(s) \cap c_pred^*(j)$ und $M_2 = c_pred^*(n^{**}) \cap c_succ^*(s) \cap c_pred^*(j)$

Unter Ausnutzung von (♠) ergibt sich:

$M_1 = (c_succ^*(z_1) \cup \{z_1\}) \cap c_pred^*(n_{k-1}) \cap c_pred^*(j)$ und

$$M_2 \subseteq \bigcup_{v \neq 1} \left(c_succ^*(z_v) \cup \{z_v\}\right) \cap \left(c_succ^*(n_{k-1}) \cup \{n_{k-1}\}\right) \cap c_pred^*(j)$$

Hieraus folgt mit (♠) sofort $M_1 \cap M_2 = \emptyset$)

q.e.d.

B.4 Beweis zu Satz 3-4

Satz 3-4 (Nachfolger von Aktivitätenknoten verschiedener Teilzweige)

Seien $CFS = (N, E, ...) \in CFS_{simple}$ ein korrekter KF-Graph und $s \in SplitNodes^{CFS}$ ein Split-Knoten mit zugehörigem Join-Knoten $j := join^{CFS}(s)$. Für $z_1, z_2 \in c_succ(s)$, $z_1 \neq z_2$ und $z_1, z_2 \neq j$ gilt stets:

$$n \in c_succ^*(z_1) \cap c_succ^*(z_2) \Rightarrow n \in c_succ^*(j) \cup \{j\}$$

Beweis:

Annahme: $\exists\, n^* \in c_succ^*(z_1) \cap c_succ^*(z_2)$ mit $n^* \in c_pred^*(j)$ (*)

Es gilt insbesondere $n^* \in c_succ^*(z_1) \cap c_pred^*(j)$. Gemäß Satz 3-3 muß dann die Aussage $A(n^*)$ gültig sein und somit auch:

$$\forall\, n \in (c_succ^*(z_2) \cup \{ z_2 \}) \cap c_pred^*(j): n \notin c_pred(n^*) \wedge n \notin c_succ(n^*) \qquad (**)$$

Des weiteren ergibt sich mit (*): $n^* \in c_succ^*(z_2) \cap c_pred^*(j)$. Dementsprechend gibt es eine Kantenfolge $z_2 =: n_0 \rightarrow n_1 \rightarrow n_2 \ldots \rightarrow n_{k-1} \rightarrow n_k := n^*$, $k \geq 1$ von z_2 nach n^* (im einfachsten Fall ist dies $z_2 \rightarrow n^*$) mit Kanten des Typs `CONTROL_E` und für deren Knoten n_x, $x \geq 0$ gilt: $n_x \in (c_succ^*(z_2) \cup \{ z_2 \}) \cap c_pred^*(j)$.

Für $n = n_{k-1}$ ergibt sich dann mit (**): $n_{k-1} \notin c_pred(n^*)$ ($\wedge\, n_{k-1} \notin c_succ(n^*)$). Dies steht im Widerspruch zu $n_{k-1} \in c_pred(n^*)$.

q.e.d.

B.5 Beweis zu Satz 7-1

Satz 7-1 (Erhalt der Zyklenfreiheit in VAGs)

Bezeichne $\omega_{top} \in WT$ die Vorlage des Toplevel-Workflows und $G_{\omega_{top}} = [V_{\omega_{top}}, A_{\omega_{top}}]$ den zugehörigen VAG, der zyklenfrei sei. Ferner entspreche $\omega_{sub} \in V_{\omega_{top}}$ der Vorlage eines Sub-Workflows beliebiger Hierachiestufe und $Pred_{\omega_{sub}} \subset V_{\omega_{top}}$ der Menge aller Knoten von $G_{\omega_{top}}$ die Vorgänger von ω_{sub} (inkl.) im VAG sind. – Die Vorlage des neu einzufügenden Aktivitätenknotens n_{add} sei durch $Template(n_{add}) = \omega_{add} \in WT$ und ihr zugehöriger VAG durch $G_{\omega_{add}} = [V_{\omega_{add}}, A_{\omega_{add}}]$ gegeben.

<u>Dann gilt</u>:

$G_{\omega_{add}}$ ist zyklenfrei <u>und</u> $V_{\omega_{add}} \cap Pred_{\omega_{sub}} = \emptyset$ (d.h. ω_{add} basiert auf keiner Vorlage, die Vorgänger von ω_{sub} im Graphen $G_{\omega_{top}}$ ist)

$\Leftrightarrow$

Das Einfügen des Knotens n_{add} in den KF-Graphen $CFS_{\omega_{sub}}$ der Vorlage ω_{sub} führt wieder zu einem zyklenfreien VAG $G_{\omega_{top}}{}^*$ (d.h. Strukturierungsregel KF-8 gilt weiterhin)

<u>Beweis:</u>

"$\Leftarrow$" (d.h. es gelte: $G_{\omega_{top}}{}^*$ ist zyklenfrei) – Beweis durch Widerspruch:

<u>Annahme</u>: (1) $G_{\omega_{add}}$ enthält einen Zyklus <u>oder</u> (2) $V_{\omega_{add}} \cap Pred_{\omega_{sub}} \neq \emptyset$

(1) kann nicht zutreffen, da nach der Einfügeoperation der Graph $G_{\omega_{add}}$ ein Teilgraph des zyklenfreien Graphen $G_{\omega_{top}}{}^*$ ist. Folglich muß gelten: $V_{\omega_{add}} \cap Pred_{\omega_{sub}} \neq \emptyset$. Das bedeutet, daß es eine Vorlage ω^* gibt, die beiden Mengen angehört. Mit $\omega^* \in V_{\omega_{add}}$ folgt: $\omega_{add} \in Pred_{\omega^*}$ (= Menge aller Knoten aus $V_{\omega_{add}}$, die Vorgänger von ω^* (inkl.) im VAG $V_{\omega_{add}}$ sind). Für den VAG $G_{\omega_{top}}{}^*$, der sich nach dem Einfügen ergibt, muß somit gelten: $\omega^* \in Pred_{\omega_{sub}} \wedge \omega_{add} \in Pred_{\omega^*}$. Hieraus folgt unmittelbar: $\omega_{add} \in Pred_{\omega_{sub}}$. Da ω_{add} eine Aktivität des KF-Graphen $CFS_{\omega_{sub}}$ definiert gilt auch: $\omega_{sub} \in Pred_{\omega_{add}}$ Hieraus folgt unmittelbar, daß der VAG $G_{\omega_{top}}{}^*$ Zyklen enthalten muß, was jedoch einen Widerspruch zu den Voraussetzungen darstellt. Somit ist unsere Annahme falsch.

"$\Rightarrow$" (d.h. es gelte: $G_{\omega_{add}}$ ist zyklenfrei <u>und</u> $V_{\omega_{add}} \cap Pred_{\omega_{sub}} = \emptyset$) – Beweis durch Widerspruch:

<u>Annahme</u>: $G_{\omega_{top}}{}^*$ enthält einen Zyklus

Da sowohl $G_{\omega_{top}}$ als auch $G_{\omega_{add}}$ zyklenfrei sind, muß der Zyklus sowohl Knoten aus $V_{\omega_{add}}$ als auch aus $V_{\omega_{top}}$ umfassen. D.h. es gibt ein $\omega^* \in V_{\omega_{add}} \cap V_{\omega_{top}}$ das in dem Zyklus enthalten ist. Wegen $V_{\omega_{add}} \cap Pred_{\omega_{sub}} = \emptyset$ muß gelten: $\omega^* \in V_{\omega_{top}} - Pred_{\omega_{sub}}$. Das ist jedoch nach der Konstruktion von $G_{\omega_{top}}{}^*$ nicht möglich. Somit ist die Annahme falsch.

q.e.d.

Anhang C: Algorithmen

Anhang C faßt grundlegende Algorithmen zusammen. Die Mehrzahl der Verfahren wurde bereits in den Kapiteln 3 bis 5 ausführlich beschrieben.

C.1 BranchNodes: Bestimmung des minimalen Verzweigungskontrollblocks, der zwei Knoten aus unterschiedlichen Ausführungszweigen enthält

Der Algorithmus nutzt aus, daß zur Bestimmung des Split-Knotens s^* unter den Vorgängermengen der Knoten n_1 und n_2 ein Split-Knoten mit $V_{out}^{s^*} \neq$ `ONE_Of_ONE` gesucht wird. Daher genügt es bei der Ermittlung von s^*, lediglich Split-Knoten zu berücksichtigen.

```
// Bestimmung von Split- und Join-Knoten der „innersten“, zwei Knoten n1 und n2 umschließenden Verzweigung
function BranchNodes(CFS, n1, n2) → (s, j)
input
    CFS      : KF-Graph (N, E, ...)
    n1, n2   : Zwei beliebige Knoten aus N mit n1 ≠ n2
output
    s        : Split-Knoten aus SplitNodes^CFS oder Wert UNDEFINED
    j        : Join-Knoten aus JoinNodes^CFS (mit j = join^CFS(s)) oder Wert UNDEFINED
begin
    if (n1 ∈ c_pred*(n2) or n2 ∈ c_pred*(n1)) then
       return (s, j):= (UNDEFINED, UNDEFINED)
    endif

    Pred_n1:= geordnete Liste aller Vorgänger von n1 mit Ausgangssemantik V_out ≠ ONE_Of_ONE
    Pred_n2:= geordnete Liste aller Vorgänger von n2 mit Ausgangssemantik V_out ≠ ONE_Of_ONE

    // Pred_ni entspricht einer geordneten Liste von Knoten, deren Elemente durch die Knotenmenge Vi mit
    // Vi:= c_pred*(ni) ∩ SplitNodes^CFS definiert sind. Je „ näher“ ein Knoten n aus dieser Liste zum Knoten ni
    // liegt, d.h. je kürzer der Weg von n nach ni ist, desto weiter vorne steht er in der Liste.

    // Durchlaufen der Listen Pred_n1 (äußere Schleife) und Pred_n2 (innere Schleife) jeweils beginnend
    // am Listenanfang
    forall (n ∈ Pred_n1) do
       forall (n‘ ∈ Pred_n2) do
            if (n = n‘) then
                return (s, j):= (n, join^CFS(n));
            endif
       done
    done
end
```

Der Algorithmus prüft zuerst, ob die beiden Knoten n_1 und n_2 in einer gemeinsamen Verzweigung enthalten sind. Dies ist genau dann der Fall, wenn sie im Kontrollfluß nicht aufeinander folgen. Die zwei maximal $|N|/2$-elementigen, geordneten Listen aus Split-Knoten (Pred_n_1 bzw. Pred_n_2) werden dann nach dem Knoten durchsucht, der zuerst in beiden Listen vorkommt. Der vorgestellte Algorithmus besitzt die Komplexität $O(|N|^2)$, seine Laufzeit reduziert sich in der praktischen Anwendung jedoch stark, da die Anzahl der Split-Knoten in einem KF-Graphen im Vergleich zur Gesamtanzahl an Knoten in der Regel relativ gering ausfällt.

C.2 MinBlock: Bestimmung minimaler Kontrollblöcke

Anhang C.2 gibt einen Algorithmus für die Bestimmung minimaler Kontrollblöcke an. Minimale Blöcke können sowohl bzgl. einer gegebenen Knotenmenge als auch in Bezug auf eine Menge von Kontrollblöcken ermittelt werden. Da der minimale Block durch seinen Start- und Endknoten eindeutig bestimmt ist, genügt es jeweils diese beiden Knoten zu ermitteln.

Wir betrachten einen Spezialfall, bei dem die Knotenmenge, für die der minimale (umschließende) Block ermittelt werden soll, aus zwei Aktivitätenknoten n_1 und n_2 ($n_1 \neq n_2$) besteht.

```
// Bestimmung des minimalen Kontrollblocks, der zwei verschiedene Knoten n1 und n2 umschließt
function MinBlock (CFS, n1, n2) → (nstart, nend)
input
    CFS       : Korrekter KF-Graph (N, E, ...) aus der KF-Graphklasse CSFall
    n1, n2    : Zwei beliebige Knoten aus N – {Start, End} mit n1 ≠ n2
output
    nstart, nend : Start- / Endknoten des gesuchten Blockes
begin
    // Bestimmung der Vorgänger- und Nachfolgerknoten von n2 bzgl. Kanten des Typs CONTROL_E
    Succ_n2:= c_succ*(n2);  Pred_n2 := c_pred*(n2);

    // Fall 1: n1 und n2 gehören unterschiedlichen Teilzweigen einer Verzweigung an
    if (n1 ∉ Succ_n2) and (n1 ∉ Pred_n2 ) then
       return  (nstart, nend) :=  BranchNodes(CFS, n1, n2);      // vgl. Anhang C.1
    endif

    // Fall 2: n1 und n2 folgen im Kontrollfluß aufeinander (d.h. es gibt eine Kantenfolge von n1
    // nach n2 bzw. umgekehrt
    if ( n1 ∈ Pred_n2) then
       // Bestimme Kantenfolge Z vom Startknoten über n1 und n2 zu seinem Endknoten. Dazu werden
       // zunächst eine Kantenfolge X vom Startknoten über n1 zum Endknoten und ggf. eine Kanten-
       // folge Y vom Startknoten über n2 zum Endknoten gebildet. Wegen n1 ∈ Pred_n2 müssen diese
       // beiden Kantenfolgen X und Y mindestens einen gemeinsamen Knoten besitzen, der zwischen
       //  n1 und n2 liegt. Hieraus läßt sich Z, ausgehend von den Kantenfolgen X und Y, konstruieren.

       Bestimme Kantenfolge X vom Startknoten Start über n1 zum Endknoten End[1]
             X:= Start=: k0  → k1 → k2 → ...→ n1 → kr  → ...→ kn := End
       if  (n2 in X) then          // n2 bereits in Kantenfolge enthalten
           Z:= X
       else                        // n2 nicht in Kantenfolge enthalten
           Bestimme Kantenfolge Y vom Startknoten Start über n2 zum Endknoten End;
               Y:= Start=: l0  → l1 → l2 →  ... → ls → n2 → ...→ lm := End
           Bestimme (einen) Knoten x ∈ { n1, kr, ..., kn } ∩ { l0,  l1, l2, ..., ls, n2 };
           Konstruiere die Kantenfolge Z wie folgt:
               Z:= Start=: k0  → k1 → k2 → ...→ n1 → kr  ... → x → ... → ls → n2 → ...→ lm := End
       endif
       //  Zerlegung von Z in Z = Z1 → Z2 → Z3
       Z1:= Start=: k0  → k1 → k2 → ...→ n1;  Z2:= n1 → ... → n2; Z3:= n2 → ...→ End;

       // Für die von n1  nach n2 führende Kantenfolge Z2 werden „offene“ Verzweigungs- und Schleifenknoten
       // gezählt:  SplitCounter (JoinCounter) gibt die Anzahl der Split- (Join-)Knoten aus Z2 an, deren zuge-
       // höriger Join- (Split-)Knoten nicht in Z2 liegt. Die „Ränder“ von Verzweigungen dürfen dabei nicht
       // mit gezählt werden (d.h. falls n1 (n2) ein Join- (Split-) Knoten ist, wird der Knoten beim Zählen nicht
       // berücksichtigt. – Entsprechend sind die Zähler LoopStartCounter und LoopEndCounter definiert.

       SplitCounter:= 0; JoinCounter:= 0; LoopStartCounter:= 0; LoopEndCounter:= 0;
       forall n in Z2 do
           if n ∈ SplitNodesCFS and n ≠ n2 and (joinCFS(n) ∉ Z2) then
               SplitCounter := SplitCounter + 1
           endif
```

[1] Die Bestimmung einer solchen Kantenfolge ist trivial. Sie erfolgt, indem man ausgehend von n_1 rückwärts (vorwärts) über Kanten des Typs `CONTROL_E` läuft, bis man beim Startknoten (Endknoten) angelangt. Grundlegend dabei ist, daß Zyklen bzgl. Kanten dieses Typs ausgeschlossen sind (Regel KF-4).

```
            if n ∈ JoinNodes^CFS and n ≠ n1 and (split^CFS(n) ∉ Z2) then
                JoinCounter := JoinCounter + 1
            endif
            if n ∈ LoopStartNodes^CFS and (endloop^CFS(n) ∉ Z2) then
                LoopStartCounter := LoopStartCounter + 1
            elseif n ∈ LoopEndNodes^CFS and (startloop^CFS(n) ∉ Z2) then
                LoopEndCounter := LoopEndCounter + 1
            endif
        done
        // Bestimmung von Start- und Endknoten des minimalen Blockes
          n_start^(1) :=  derjenige Knoten n_start^(1) aus Z1, für den gilt: Auf dem Weg K:= n_start^(1) →... → n1 gibt es
                          JoinCounter Split-Knoten, deren zugehöriger Join-Knoten nicht in K enthalten ist[2]
                          (Setze n_start^(1):= n1, falls JoinCounter = 0 gilt)
          n_start^(2) :=  derjenige Knoten n_start^(2) aus Z1, für den gilt: Auf dem Weg K:= n_start^(2) →... → n1 gibt es
                          LoopEndCounter Schleifenanfangsknoten, deren zugehöriger Endknoten nicht in K
                          enthalten ist (Setze n_start^(2):= n1, falls LoopEndCounter = 0 gilt)
          n_start :=      derjenige der beiden Knoten n_start^(1) und n_start^(2), der „näher“ zum Startknoten liegt

          n_end^(1) :=    derjenige Knoten n_end^(1) aus Z3, für den gilt: Auf dem Weg K:= n2 →... → n_end^(1) gibt es genau
                          SplitCounter Join-Knoten, deren zugehöriger Split-Knoten nicht in K enthalten ist
                          (Setze n_end^(1):= n2, falls SplitCounter = 0 gilt)
          n_end^(2) :=    derjenige Knoten n_end^(2) aus Z3, für den gilt: Auf dem Weg K:= n2 →... → n_end^(2) gibt
                          es LoopStartCounter Schleifenendknoten, deren zugehöriger Anfangsknoten nicht
                          in K enthalten ist (Setze n_end^(2):= n2, falls LoopStartCounter = 0 gilt)
          n_end :=        derjenige der beiden Knoten n_end^(1) und n_end^(2), der näher zum Endknoten liegt
        return (n_start, n_end)
    else  // d.h. es gilt: n1 ∈ Succ_n2;
        // analoge Vorgehensweise wie bei "n1 ∈ Pred_n2" mit Vertauschung von n1 und n2
        ...
    endif
end
```

Unter Ausnutzung dieses speziellen MinBlock-Algorithmus kann ein verallgemeinertes Verfahren angegeben werden, mit dem sich der minimale, zwei Kontrollblöcke B_1 und B_2 umschließende Block bestimmen läßt. (Bei der Bestimmung des minimalen, zwei Kontrollblöcke B_1 und B_2 umschließenden Blockes werden Informationen über Anordnungsbeziehungen der Start- und Endknoten von B_1 und B_2 genutzt.) Hiervon ausgehend kann wiederum ein Algorithmus angegeben werden, der auf eine beliebige Zahl von Kontrollblöcken anwendbar ist. Da einzelne Aktivitätenknoten ebenfalls Kontrollblöcke darstellen, läßt sich damit auch der minimale Block bzgl. einer Knotenmenge N^* bestimmen.

C.3 NextMarking: Neubestimmung von Knoten- und Kantenmarkierungen

```
// Neubestimmung der Knoten- und Kantenmarkierungen eines KF-Ausführungsgraphen nach der Beendigung
// eines Aktivitätenknotens node, d.h. nach seinem Wechsel in den Zustand COMPLETED – Aus Gründen der
// besseren Lesbarkeit beschränken wir uns bei der Darstellung des NextMarking-Verfahrens auf die in den
// Kapitel 3 - 5 eingeführten Konstrukte und Regeln, wohingegen wir Priorisierungs- und Fehlerkanten
// ausklammern. Die bei der Verwendung dieser beiden Kantentypen durchzuführenden Markierungsanpas-
// sungen wurden ausführlich in Kapitel 6 beschrieben.
```

[2] Zur Bestimmung dieses Knotens durchläuft man die Kantenfolge Z_1:= *Start* → k_1 → k_2 → ...→ n_1 rückwärts d.h. beginnend mit n_1. Trifft man auf einen Split-Knoten *s* und ist $join^{CFS}(s)$ nicht in der Kantenfolge s → ...→ n_1 enthalten, so dekrementiert man *JoinCounter* um 1. Besitzt dieser Zähler anschließend den Wert 0 ist man fertig und es gilt $n_{start}^{(1)}$= s. Andernfalls läuft man weiter rückwärts bis zum nächsten offenen Split-Knoten und dekrementiert *JoinCounter* wieder um 1, usw. Auf ähnliche Art können $n_{start}^{(2)}$, $n_{end}^{(1)}$ und $n_{end}^{(2)}$ bestimmt werden.

```
function NextMarking(CFS_instance, node) → (NS_new, ES_new)
input
    CFS_instance :  KF-Ausführungsgraph CFS_instance = (CFS, NS_old, ES_old, It_old)
    node         :  Aktuell beendeter Aktivitätenknoten
output
    NS_new, ES_new  :  Knoten- und Kantenmarkierungen nach Neubewertung des Graphen
begin
    forall n ∈ N do          // Initialisierung der Neumarkierungen von Knoten
      NS_new(n):= NS_old(n)
    done
    forall e ∈ N do          // Initialisierung der Neumarkierungen von Kanten
      ES_new(e):= ES_old(e)
    done
    NS_new(node):= COMPLETED;     // Neumarkierung des beendeten Knotens
    SignaledEdges:= ∅;            // Menge aller neu signalisierten Kanten
    // Fall 1: Beendigung des Endknotens führt zu WF-Termination
    if NT^node = ENDFLOW then     // Beendeter Knoten ist Endknoten des WF
      return (NS_new, ES_new)
    endif
    // Fall 2: Anwendung von Markierungsregel M_3 bei Schleifenrücksprung
    if NT^node = ENDLOOP then     // Beendeter Knoten ist Endknoten einer Schleife
      if LOOP^node = TRUE then    // Schleifenbedingung ist erfüllt
            // Schleifenkörper (inkl. Start- und Endknoten) neu markieren
            L:= Menge aller Knoten des Schleifenkörpers (inkl. Start- und Endknoten);
            forall x ∈ L do       // Knoten / Kanten aus L neu markieren
                NS_new(x):= NOT_ACTIVATED;
                forall e ∈ E with Id_source^e = x do
                    ES_new(e):= NOT_SIGNALED
                done
            done
            // Schleifenrücksprungkante mit TRUE signalisieren
            Für die Schleifenrücksprungkante e ∈ E mit Id_source^e = node und ET^e = LOOP_E:
                ES_new(e):= TRUE_SIGNALED
                SignaledEdges:= SignaledEdges ∪ {e};
            goto EvaluateExecutionRules
      else                        // Schleifenbedingung ist nicht erfüllt
            Für die Schleifenrücksprungkante e ∈ E mit Id_source^e = node und ET^e = LOOP_E:
                ES_new(e):= FALSE_SIGNALED
            // Weiter bei Fall 3
      endif
    endif
    // Fall 3: Anwendung von Markierungsregel M_1 bei Beendigung von normalen Knoten
    // und von AND-Split-Knoten → Signalisieren ausgehender Kontroll- und Sync-Kanten
    if V_out^node ∈ {ONE_Of_ONE, ALL_Of_ALL} then
      forall e ∈ E with Id_source^e = node do
            if ET^e ∈ {CONTROL_E, SOFT_SYNC_E, STRICT_SYNC_E} then
                ES_new(e):= TRUE_SIGNALED;
                SignaledEdges:= SignaledEdges ∪ {e}
            endif
      done
```

```
        goto EvaluateExecutionRules
    endif

    // Fall 4: Anwendung von Markierungsregel M2 bei Beendigung eines OR-Split-Knotens:
    // Aktivierung genau eines Zweiges; Markierung der anderen Zweige als nicht ausführbar
    if (V_out^node = ONE_Of_ALL) then
        forall e ∈ E with Id_source^e = node and ET^e = CONTROL_E do
            if DP^node = SC^e then    // Eingangskante des ausgewählten Zweiges
                ES_new(e):= TRUE_SIGNALED;
                SignaledEdges:= SignaledEdges ∪ {e}
            else                      // Eingangskante eines nicht gewählten Zweiges
                ES_new(e):= FALSE_SIGNALED;
                // Bestimme alle Knoten des von e ausgehenden Teilzweiges
                Z:= (c_succ*( Id_dest^e) ∪ {Id_dest^e}) ∩ c_pred*(join^CFS(node));

                forall n ∈ Z do
                    NS_new(n):= SKIPPED;   // Knoten als nicht mehr ausführbar markieren
                    forall e ∈ E with Id_source^e = n do     // Markieren ausgehender Kanten
                        ES_new(e):= FALSE_SIGNALED;
                        if ET^e = SOFT_SYNC_E then   // Zielknoten muß evaluiert werden
                            SignaledEdges:= SignaledEdges ∪ {e}
                        endif
                    done
                done
            endif
        done
        // Markieren aller vom OR-Split-Knoten ausgehenden Sync-Kanten
        forall e ∈ E with Id_source^e = node and ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E}do
            ES_new(e):= TRUE_SIGNALED;
            SignaledEdges:= SignaledEdges ∪ {e}
        done
        goto EvaluateExecutionRules
    endif

// Für Kanten der Menge SignaledEdges muß geprüft werden, ob durch das Signalisieren der Kante(n) ihr
// Zielknoten ausführbar wird.
EvaluateExecutionRules:

    while SignaledEdges ≠ ∅ do

        Entnehme ein Element e aus SignaledEdges  (d.h. SignaledEdges:= SignaledEdges – {e})

        n:= Id_dest^e;          // Zielknoten der Kante, deren Status neu bewertet werden muß

        // Fall 1: Anwendung von Ausführungsregel A3 nach Schleifenrücksprung
        if   ET^e = LOOP_E then  // Signalisierte Kante ist Schleifenrücksprungkante
            NS_new(n):= ACTIVATED;  // Startknoten der Schleife neu aktivieren
            goto NextIteration
        endif

        // Fall 2: Anwendung von Ausführungsregel A1 bei normalen Aktivitätenknoten und
        // AND-Join-Knoten
        if  V_in^n ∈ {ONE_Of_ONE, ALL_Of_ALL} then
            activate:= TRUE;
            forall  e ∈  E with Id_dest^e = n  do
                if  ET^e ∈ {CONTROL_E, STRICT_SYNC_E} and ES_new(e) ≠ TRUE_SIGNALED then
                    activate:= FALSE; break;  // For-Schleife verlassen
```

```
            elseif (ET^e = SOFT_SYNC_E and ES_new(e) = NOT_SIGNALED) then
                activate:= FALSE; break;  //  For-Schleife verlassen
            endif
        done
        if activate = TRUE then NS_new(n):= ACTIVATED  endif
    else if  V_in^n = ONE_Of_ALL then
        //  Fall 3: Anwendung von Ausführungsregel A2 bei OR -Join-Knoten
        activate:= FALSE;
        //  Eine eingehende Kontrollkante markiert?
        forall  e ∈  E with Id_dest^e = n  and ET^e = CONTROL_E do
            if ES_new(e) = TRUE_SIGNALED then
                    activate:= TRUE;  e*:= e; break;  //  eingehende Kante des Siegerzweiges
            endif;
        done
        //  Sync-Kanten markiert?
        if activate = TRUE  then   // es gibt eine mit TRUE signalisierte Eingangs-Kontrollkante
            forall e ∈ E with Id_dest^e = n  and ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E} do
                if (ES_new(e) = NOT_SIGNALED) then
                    activate:= FALSE; break
                endif;
                if (ET^e = STRICT_SYNC_E) and (ES_new(e) ≠ TRUE_SIGNALED) then
                    activate:= FALSE; break
                endif
            done
        endif
        if activate = TRUE then          // Voraussetzungen für Aktivierung erfüllt
            NS_new(n):= ACTIVATED
        endif
        //  Zurücksetzen von Verliererzweigen im Falle einer parallelen Verzweigung
        //  mit finaler Auswahl
        s:= split^CFS(n);
        if (V_out^s = ALL_Of_ALL) then//  AND-Split / OR-Join – Kombination
            E_in^n:= {e ∈ E | e ≠ e* ∧  Id_dest^e = n   und ET^e = CONTROL_E};
            forall e ∈ E_in^n do    // einmündende Kontrollkanten von Verliererzweigen
                ES_new(e):= FALSE_SIGNALED;
                Für x = Id_source^e setze:
                    Z_x:= ({x} ∪ c_pred*(x)) ∩ c_succ*(s)    //  Knoten des Verliererzweiges
                forall n* ∈ Z_x do
                    //  Knotenmarkierung auf SKIPPED setzen
                    NS_new(n*):= SKIPPED;
                    //  alle einmündenden Kontrollkanten auf FALSE_SIGNALED setzen
                    InEdges:= {e ∈ E | Id_dest^e = n* ∧ ET^e = CONTROL_E};
                    forall  e‘ ∈ InEdges do
                        ES_new(e‘):= FALSE_SIGNALED
                    done
                    //  alle ausgehenden Sync-Kanten auf FALSE_SIGNALED setzen
                     SyncEdges:= Alle vom Knoten n* ausgehenden Sync-Kanten
                     forall  e‘ ∈ SyncEdges do
                         ES_new(e‘):= FALSE_SIGNALED
                         //  Da die Sync-Kante bisher u.U. noch nicht signalisiert worden ist, muß
```

```
                                    // der Zielknoten neu bewertet werden (Anm. Aufgrund von Regel KF-6
                                    // sind in diesem Zusammenhang keine strikten Sync-Kanten betroffen.
                                    if Id_dest^e' ∉ Z_x then // Zielknoten muß ggf. neu bewertet werden
                                       SignaledEdges:= SignaledEdges ∪ { e' }
                                    endif
                                done
                            done
                        done
                    endif
            endif
            NextIteration: // Sprungmarke
        done
        return (NS_new, ES_new)
end
```

C.4 CheckSyncRelation: Verwendung von Sync-Kanten

```
// Die Funktion CheckSyncRelation prüft für eine Sync-Kante e, ob die folgende Bedingung zutrifft: Die Quell-
// aktivität Id_source^e von e gehört keiner Verzweigung mit OR-Join-Knoten an (bedingte Verzweigung oder paral-
// lele Verzweigung mit finaler Auswahl), die nicht auch den Zielknoten Id_dest^e von e enthält.

function CheckSyncRelation (CFS, e) → (result)

input
    CFS      : Korrekter KF-Graph (N, E, D, ...) aus der KF-Graphklasse CFS_all
    e        : Sync-Kante aus E
output
    result   : Bool'scher Wert, der anzeigt, ob die Bedingung erfüllt ist oder nicht

begin

    (split, join):= BranchNodes(CFS, Id_source^e, Id_dest^e);

    Y:= Bestimme eine Kantenfolge Id_source^e =:y_0 → y_1 → ... → y_k:= join über normale Kontrollkanten
          vom Knoten Id_source^e zum Knoten join

    openBranchCount:= 0;

    // Gibt es auf dem Weg Y von Id_source^e =:y_0 nach join einen OR-Join-Knoten, dessen
    // korrespondierender AND-/OR-Split-Knoten nicht auf Y liegt

    for n:= y_0 to y_k do    // Die Kantenfolge Y durchlaufen

        if (V_in^n ≠ ONE_Of_ONE and n ≠ y_0) then
            openBranchCount:= openBranchCount – 1;
            if (openBranchCount = -1) then
                if V_in^n = ONE_Of_ALL then   // OR-Join-Knoten
                    result:= FALSE; return result;
                else
                    openBranchCount:= 0;
                endif
            endif
        endif

        if (V_out^n ≠ ONE_Of_ONE  and n ≠ y_k) then
            openBranchCount:= openBranchCount + 1;
        endif
    done
    return TRUE;
end
```

C.5 WriterExists: Sicherstellung der Versorgung obligater Lesezugriffe

```
// Der Algorithmus WriterExists prüft, ob ein obligater Lesezugriff des Knotens n_read auf das Datenelement d
// sicher versorgt werden kann. Das hier vorgestellte Verfahren berücksichtigt Schreibzugriffe auf d von
// Vorgängern bzgl. normaler Kontrollkanten und Sync-Kanten.

Algorithmus (WriterExists)

function WriterExists(CFS, DFS, n_read, d) → (exists)

input
    CFS      : Korrekter KF-Graph (N, E, D, ...) aus der KF-Graphklasse CFS_all
    DFS      : Zugehöriges DF-Schema.
    n_read   : Beliebiger Aktivitätenknoten aus N – {Start}
    d        : Datenelement aus D

output
    exists   : Bool'scher Wert, der anzeigt, ob sichere Versorgung des Lesezugriffs möglich ist

begin

    // Teil I: Datenversorgung durch Aktivitätenknoten überprüfen, die Vorgänger von n_read bzgl. normaler
    // Kontrollkanten (des Typs CONTROL_E) sind.

    // Schritt 1: Bestimme alle Vorgänger von n_read bzgl. normaler Kontrollkanten und initialisiere für jeden
    // Knoten die Attribute Supplied und Counter mit FALSE bzw. 0.

    Pred:= c_pred*(n_read);

    forall n ∈ N do
        Supplied(n):= FALSE; Counter(n):= 0
    done

    // Schritt 2: Für direkte Nachfolger n* von Schreiberknoten aus Pred (bzgl. normaler Kontrollkanten):
    // Erhöhe Counter(n*) um 1 und füge n* der Menge NodeList der zu betrachtenden Knoten hinzu.
    // (Anm.: Es werden nur solche Schreibzugriffe berücksichtigt, die vor n_read im Kontrollfluß erfolgen.)

    NodeList:= ∅;
    forall n ∈ Pred do
        // Schreibt Aktivität n (bzw. eine ihr vor-/nachgeschaltete Aktion) das Datenelement d obligat
        if ActivityWrites^DFS(n, d, MANDATORY) then // (vgl. Definition 5-2)
            forall n* ∈ c_succ(n) do   // Direkte Nachfolger des Schreiberknotens n
                if n* ∈ Pred ∪ {n_read} then
                    Counter(n*):= Counter(n*) + 1; NodeList:= NodeList ∪ {n*}
                endif
            done
        endif
    done

    // Schritte 3+4: Menge der als „versorgt" markierten Knoten solange erweitern, bis sie entweder n_read
    // enthält oder aber kein weiterer Knoten mehr markiert werden kann.
    // Überprüfe für Elemente aus der Menge der aktuell betrachteten Knoten (NodeList), ob sie als versorgt
    // markiert werden dürfen. Falls ja, füge für den jeweiligen Knoten seine direkten Nachfolger der Menge
    // NewNodeList (= Menge der in der Folgeiteration zu betrachtenden Knoten) hinzu

    while NodeList ≠ ∅ and Supplied(n_read) = FALSE do
        NewNodeList:= ∅;        // Menge der in der nächsten Iteration zu betrachtenden Knoten
        forall n ∈ NodeList do
            // Nach Konstruktion des Verfahrens schreibt mind. 1 direkter Vorgänger von n das
            // Element d oder er darf d obligat lesen → Ist d bei Aktivierung von n sicher versorgt?
            if V_in^n ∈ {ONE_Of_ONE, ALL_Of_ALL} or (Counter(n) = # Eingangskontrollkanten von n) then
                // Markiere den Knoten n als versorgt
                Supplied(n):= TRUE;
```

```
                    // Erhöhe den Zähler aller direkten Nachfolger n* von n um 1 und füge die Knoten
                    // n* der Menge NewNodeList hinzu (allerdings nur, falls gilt: n* ∈ Pred und
                    // falls n* noch nicht als versorgt markiert wurde)
                    forall n* ∈ c_succ(n) do
                        if (n* ∈ Pred ∪ {n_read} and Supplied(n*) = FALSE) then
                            Counter(n*):= Counter(n*) + 1; NewNodeList:= NewNodeList ∪ {n*}
                        endif
                    done
                endif
            done
            NodeList:= NewNodeList;
done

// Schritt 5: Ist sichere Datenversorgung über normale Kontrollkanten (→ Teil I) möglich?
if (Supplied(n_read) = TRUE) then
    return exists:= TRUE;
endif

// Teil II: Datenversorgung über Sync-Kanten prüfen – Diese Überprüfung wird nur durchgeführt, falls
//  keine sichere Versorgung durch Vorgänger bzgl. normaler Kontrollkanten möglich ist (d.h. unter der
//  aktuellen Markierung gilt: Supplied(n_read) = FALSE)

// Schritt 6: Bestimme alle Vorgänger von n_read bzgl. normaler Kontrollkanten und bzgl. Sync-Kanten:
// - Prüfen, ob es Vorgängerknoten aus parallelen Zweigen gibt, die d obligat schreiben
// - Falls ja, initialisiere Knoten- und Kantenmarkierungen für nachfolgendes Verfahren

SyncPred: = pred*(n_read);                  // Vorgänger bzgl. Kontroll- und Sync-Kanten
ParallelNodes:= SyncPred – Pred;  // Vorgänger aus parallelen Zweigen

Writers:= {n ∈ ParallelNodes | ActivityWrites^DFS(n, d, MANDATORY) = TRUE};

if (Writers = ∅) then                // Keine Schreiberknoten in der Menge ParallelNodes enthalten
    return exists: = FALSE;          // Schreiberknoten die Vorgänger bzgl. normaler Kontrollkanten
endif;                               // sind, wurden bereits in Schritt 2 berücksichtigt!

forall n ∈ SyncPred ∪ {n_read} do        // Setze für alle Knoten das Attribut SyncSupplied auf FALSE
    SyncSupplied(n):= FALSE;             // Dieses Attribut zeigt an, ob Lesezugriffe von n auf d durch
done                                     // Vorgänger bzgl. Sync-Kanten versorgbar sind

// Markieren der für die folgenden Betrachtungen relevanten Kontroll- und Sync-Kanten. Es werden nur
// solche Kanten berücksichtigt, deren Zielknoten ein Vorgänger von n_read (inkl.) ist und der noch nicht als
// versorgt markiert worden ist. Diese Kantenmenge wird im weiteren Verlauf – jeweils nach der Neumar-
// kierung von Knoten – sukzessive eingeschränkt, so daß unnötige (redundante) Überprüfungen wegfallen.

forall e ∈ E do
    if (Id_dest^e ∈ SyncPred ∪ {n_read}) and Supplied(Id_dest^e) = FALSE then
        Consider(e):= TRUE;
    else
        Consider(e):= FALSE;
    endif
done

// Schritt 7: Schreiberknoten von d, die Vorgänger von n_read bzgl. Sync-Kanten (und Kontrollkanten) sind,
// auswählen (→ in Schritt 2 noch nicht berücksichtigt) und Datenversorgung (d.h. Versorgung von Lese-
// zugriffen auf das Datenelement d) ihrer direkten Nachfolger überprüfen.

forall n ∈ Writers do

    // Überprüfe, ob direkte Nachfolger des Schreiberknotens n das Datenelement d obligat lesen dürfen
    // – Diese Überprüfung ist für einen Nachfolger x ∈ succ(n) nur erforderlich, wenn x noch nicht als
    // versorgt markiert worden ist und wenn x ein Vorgänger von n_read (inkl.) ist. Diese beiden Be-
    // dingungen treffen zu, wenn Consider(n → x) = TRUE gilt (vgl. oben).

    forall e ∈ E with (Id_source^e = n and Consider(e) = TRUE) do
```

```
            if ET^e = CONTROL_E then
                // Direkte Nachfolger bzgl. normaler Kontrollkanten behandeln wie in Schritt 2
                Counter(Id_dest^e):= Counter(Id_dest^e) + 1; NodeList:= NodeList ∪ { Id_dest^e };
            elsif ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E } then
                //  Direkte Nachfolger bzgl. Sync-Kanten gesondert behandeln. Die Funktion
                //  SyncDestSupplied wird nachfolgend erörtert.
                if SyncDestSupplied(e) = TRUE then
                    SyncSupplied(Id_dest^e):= TRUE;  NodeList:= NodeList ∪ { Id_dest^e }
                endif
            endif
        done
    done

    //  Sonderfall: Es konnte kein Nachfolgerknoten eines Schreibers in die Menge NodeList aufgenommen
    //  werden. Hier muß für noch nicht bewertete Zielknoten von Sync-Kanten geprüft werden, ob sie nicht
    //  nicht doch – infolge transitiver Abhängigkeiten zu Schreiberknoten (bzgl. Sync-Kanten) paralleler
    //  Zweige – d obligat lesen dürfen (die Routine SyncDestSupplied wird nachfolgend beschrieben)
    if NodeList = ∅ then
        forall e ∈ E with ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E} and Id_source^e ∉ Writers do
            if  (Consider(e) = TRUE and SyncDestSupplied(e) = TRUE) then
                    SyncSupplied(Id_dest^e):= TRUE; NodeList:= NodeList ∪ { Id_dest^e };
            endif
        done
    endif
    //  Schritte 8+9: Menge der als „versorgt“ markierten Knoten solange erweitern, bis sie entweder
    //  n_read  enthält oder aber kein weiterer Knoten mehr als versorgt markiert werden kann. Dazu:
    //  Überprüfe für Elemente aus der Menge der aktuell betrachteten Knoten (NodeList), ob sie als
    //  versorgt markiert werden dürfen. Falls ja, füge jeweils die direkten Nachfolgerknoten der Menge
    //  NewNodeList der in der Folgeiteration zu betrachtenden Knoten hinzu.

    while (NodeList ≠ ∅ and Supplied(n_read) = FALSE) do
        NewNodeList:= ∅;        // Menge der in der nächsten Iteration zu betrachtenden Knoten
        forall n ∈ NodeList do
            //  Darf der Knoten n das Datenelement d obligat lesen?
            if  (V_in^n ∈ {ONE_Of_ONE, ALL_Of_ALL} and  Counter(n) ≥ 1) or
                (V_in^n =  ONE_Of_ALL and Counter(n) =  # der in n einmündenden Kontrollkanten) or
                (SyncSupplied(n) = TRUE) then
                    Supplied(n):= TRUE;                 //  Markiere den Knoten n als versorgt
                    for all e ∈ E with Id_dest^e = n do     // Einmündende Kanten eines versorgten Knotens
                        Consider(e):= FALSE             //  in der Folge nicht mehr berücksichtigen (da
                    done                                //  sonst unnötige bzw. redundante Checks)
            endif
            if (Supplied(n) = TRUE) then                //  Konnte der Knoten n neu markiert werden?
                    //  Überprüfe Versorgung direkter Nachfolger (wie in Schritt 7)
                    forall e ∈ E with (Id_source^e = n and Consider(e) = TRUE) do
                        if ET^e = CONTROL_E then
                            Counter(Id_dest^e):= Counter(Id_dest^e) + 1; NewNodeList:= NewNodeList ∪ { Id_dest^e};
                        elsif ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E } then
                            if SyncDestSupplied(e) = TRUE then
                                SyncSupplied(Id_dest^e):= TRUE; NewNodeList:= NewNodeList ∪ { Id_dest^e }
                            endif
                        endif
                    done
            endif
        done
```

```
        // Sonderfall: Es konnte kein Knoten mehr in die Menge NewNodeList aufgenommen werden.
        // Hier muß für relevante Sync-Kanten, deren Zielknoten nicht mit Supplied = TRUE markiert
        // ist, (nochmals) explizit geprüft werden, ob sie als versorgt markiert werden dürfen.

        if NewNodeList = ∅ then
            forall e ∈ E with ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E} and Consider(e) = TRUE do
                if SyncDestSupplied(e) = TRUE then
                    SyncSupplied(Id_dest^e):= TRUE; Consider(e):= FALSE; NodeList:= NodeList ∪ { Id_dest^e };
                endif
            done
        endif
        NodeList:= NewNodeList;
    done
    // Schritt 10: Ist sichere Datenversorgung möglich?
    if (Supplied(n_read) = TRUE) then
        return exists:= TRUE
    endif

    return exists:= FALSE;
end // WriterExists
```

```
// Darf der Zielknoten einer Sync-Kante e das Datenelement d obligat lesen, d.h. kann er als versorgt markiert
// werden?
// Die Funktion SyncDestSupplied wird lokal vom WriterExists-Algorithmus in unterschiedlichem Kontext
// aufgerufen. Dabei kommt es zur Überprüfung des in Kapitel 5 vorgestellten OR-Split-Kriteriums, insbe-
// sondere werden auch transitiven Abhängigkeiten zu Vorgängern (bzgl. Sync-Kanten) verschiedener
// Parallelzweige berücksichtigt. Ist das definierte OR-Split-Kriteriums erfüllt, kann der Zielknoten der
// Sync-Kante, als versorgt markiert werden.
// Die Funktion SyncDestSupplied macht Gebrauch von den im Verlauf des WriterExists-Verfahrens bisher
// ermittelten Knoten- (→ Attribut Supplied) und Kantenmarkierungen (→ Attribut Consider). Insbesondere
// versucht sie, durch Ausnutzung dieser Informationen, den Umfang der notwendig werdenden Überprüfungen
// zu reduzieren.

function SyncDestSupplied(e) → (result)
input
    e:          Sync-Kante aus E (mit Quellknoten Id_source^e und Zielknoten Id_dest^e)
output
    result:     Bool'scher Wert der anzeigt, ob d vom Zielknoten obligat gelesen werden darf
begin
    // Zielknoten strikter Sync-Kanten dürfen das Datenelement d in jedem Fall obligat lesen, wenn ihr
    // Quellknoten d obligat lesen darf oder wenn er d obligat schreibt.

    if  (ET^e=STRICT_SYNC_E and
         (Supplied(Id_source^e) = TRUE or ActivityWrites^DFS(Id_source^e, d, MANDATORY)) then
        return (result:= TRUE)
    endif
    // Bestimme alle Vorgängerknoten von Id_dest^e aus Parallelzweigen (bzgl. Sync-Kanten), die potentiell
    // zur Versorgung eines obligaten Lesezugriffs von Id_dest^e auf d beitragen können. Zur Minimierung
    // der nachfolgenden Überprüfungen werden bei der Bestimmung dieser Vorgängermenge nur noch
    // die Sync-Kanten e* berücksichtigt, für die Consider(e*) = TRUE gilt, d.h. deren Zielknoten noch
    // nicht als versorgt markiert worden ist.

    providerNodes:= Alle Knoten des KF-Graphen, die in einem parallelen Zweig zu Id_dest^e liegen
                    und die Vorgänger dieses Knotens (bzgl. Sync- und Kontroll-Kanten) sind
                    (dabei werden nur Sync-Kanten e* mit Consider(e*) = TRUE berücksichtigt!)
    // Bestimme den kleinsten Kontrollblock, der die Knotenmenge providerNodes und den Knoten Id_dest^e
    // umschließt (→ paralleler Verzweigungskontrollblock)

    (split, join):= MinBlock (CFS, providerNodes ∪ { Id_dest^e });
    // Initialisiere Prüfmarkierungen für Knoten dieser Menge
```

```
    forall (n ∈ providerNodes ∪ {split}) do
        BackCounter(n):= 0; Checked(n):= FALSE;
    done

    // Für diejenigen Knoten aus der Menge providerNodes, deren Lesezugriffe auf das Datenelement d
    // sicher versorgt werden können oder die d obligat schreiben: Erhöhe für jeden Vorgängerknoten n
    // (bzgl. normaler Kontrollkanten) den Zähler BackCounter(n) um 1 und füge den Knoten der Menge
    // BackNodeList hinzu

    BackNodeList:= ∅;
    forall n ∈ providerNodes with (ActivityWrites^DFS(n, d, MANDATORY) or Supplied(n) = TRUE) do
        Checked(n):= TRUE;
        forall n* ∈ c_pred(n) with Checked(n*) = FALSE do
            BackCounter(n*):= BackCounter(n*) + 1; BackNodeList:= BackNodeList ∪ {n*}
        done
    done

    // Markierungen von Knoten mit Checked = TRUE solange „rückwärts" propagieren, bis entweder der
    // Split- bzw. Startknoten des gegebenen Kontrollblocks mit Checked = TRUE markiert werden kann oder
    // dies für keinen Knoten mehr möglich ist. Dazu: Überprüfe jeweils für Elemente aus der Menge
    // BackNodeList, ob sie mit Checked = TRUE markiert werden dürfen. Falls ja, füge ihre direkten Vor-
    // gänger der Menge NewNodeList der in der Folgeiteration zu betrachtenden Knoten hinzu.

    while BackNodeList ≠ ∅ and Checked(split) = FALSE do

        NewNodeList:= ∅;        // Menge der in der nächsten Iteration zu betrachtenden Knoten
        forall n ∈ BackNodeList do
            // Nach Konstruktion ist mind. 1 direkter Nachfolger von n mit Checked = TRUE markiert.
            if ( (V_out^n = ONE_Of_ONE) or (V_out^n = ALL_Of_ALL and V_in^join^CFS(n) = ALL_Of_ALL)
              or (V_out^n = ONE_Of_ALL and BackCounter(n)= # von n ausgehender Zweige)) then
                Checked(n):= TRUE       // Markiere den Knoten n als geprüft
            else  // Sonderfall: Parallelverzweigung mit finaler Auswahl
                if (V_out^n = ALL_Of_ALL and V_in^join^CFS(n) = ONE_Of_ALL) then
                    // Split-Knoten n darf nur dann markiert werden, wenn der zugehörige Join-
                    // Knoten join^CFS(n) mit Checked = TRUE markiert ist!
                    if (join^CFS(n) ∈ providerNodes and Checked(join^CFS(n)) = TRUE)
                        Checked(n):= TRUE;
                    endif
                endif
            endif
            // Falls der Knoten n neu markiert wird: Erhöhe den Zähler aller direkten Vorgänger n* von n um
            // den Wert 1 und füge die Vorgängerknoten n* der Menge NewNodeList hinzu (allerdings nur,
            // falls gilt: n* ∈ ProviderNodes ∪{split} und falls n* noch nicht mit Checked(n*) = TRUE
            // markiert worden ist)
            if Checked(n) = TRUE then       // Wurde der Knoten n neu markiert?
                forall n* ∈ c_pred(n) do
                    if (n* ∈ (providerNodes ∪ {split}) and Checked(n*) = FALSE) then
                        BackCounter(n*):= BackCounter(n*) + 1; NewNodeList:= NewNodeList ∪ {n*}
                    endif
                done
            endif
        done
        BackNodeList:= NewNodeList;
    done

    // Nach der Konstruktion des Verfahrens darf Id_dest^e das Datenelement genau dann obligat lesen, wenn der
    // Startknoten des eingangs ermittelten Kontrollblocks als Checked(split) = TRUE markiert worden ist.

    return (result:= Checked(split));
end  // SyncDestSupplied
```

C.6 Ermittlung von UndoSet, $RedoSet_{critical}$, $RedoSet_{uncritical}$ und T_{fix}

Der folgende Algorithmus bezieht sich auf Abschnitt 8.5 aus Kapitel 8. Er ermittet die Mengen *UndoSet*, $RedoSet_{critical}$ und $RedoSet_{uncritical}$ in einem Durchlauf durch die Änderungshistorie nach folgendem Prinzip:

Algorithmus zur Bestimmung von UndoSet, RedoSet, $RedoSet_{critical}$, $RedoSet_{uncritical}$ und T_{fix}

```
UndoSet:= ∅; RedoSet:= ∅;  RedoSet_critical:= ∅; RedoSet_uncritical = ∅; T_fix:= UNDEFINED;

//  Durchlaufe die Änderungshistorie – beginnend mit dem ältesten Eintrag – in Vorwärtsrichtung.
//  Stelle für jeden Eintrag fest, ob er einer dieser Mengen  zuzuordnen ist.

forall T ∈ ChangeHistory do     // Änderungshistorie in Vorwärtsrichtung durchlaufen
   if (TrDuration^T = TEMPORARY and TrStartRegion^T ⊆ L_body) then
         UndoSet:= UndoSet ∪ {T}
         if (T_fix = UNDEFINED) then
               T_fix:= T          // Fixpunkt: Älteste ÄT, die zurückgenommen werden muß
         endif
   else
         if (UndoSet ≠ ∅) then
               OpsRedoSet^T:= ∅;
               forall Op ∈ TrAppliedOps^T with ({L_S} ∈ pred(n) ∀ n ∈ OpStartRegion^Op) do
                     OpsRedoSet^T:= OpsRedoSet^T ∪ {Op}
               done
               if OpsRedoSet^T ≠ ∅ then  //  Transaktion dem RedoSet hinzufügen
                     RedoSet:= RedoSet ∪ {T}
                     //  Müssen bei der Wiedereinbringung Checks, z.B. bzgl. Datenfluß, erfolgen?
                     forall T' ∈ UndoSet do
                        if (TrStartRegion^T' ≺ TrStartRegion^T) then
                              RedoSet_critical:= RedoSet_critical ∪ {T}; break  // Innere for-Schleife verlassen
                        endif
                     done
               endif
         endif
   endif
done
RedoSet_uncritical := RedoSet – RedoSet_critical
```

Der Algorithmus bestimmt für jedes $T \in RedoSet$ auch die Menge der zurückzunehmenden Änderungoperationen $OpsRedoSet^T \subseteq TrAppliedOps^T$. Darüber hinaus wird die älteste temporäre ÄT T_{fix} aus dem *UndoSet* ermittelt.