Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme
Leiter: Prof. Dr. Peter Dadam

Konzepte, Architektur und Implementierung adaptiver Prozessmanagementsysteme

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik der Universität Ulm

Vorgelegt von:
Ulrich Walter Kreher geboren in Heilbronn-Neckargartach

2014

Amtierende Dekanin:	Prof. Dr. Tina Seufert
Gutachter:	Prof. Dr. Manfred Reichert
	Prof. Dr. Peter Dadam
Tag der Promotion:	24.7.2014

Kurzfassung

Prozessmanagementsysteme (PrMS) besitzen erhebliches Potential: Betriebliche Abläufe werden in Form von Prozessmodellen explizit repräsentiert und durch eine mächtige Laufzeitumgebung ausgeführt. Die aktive Steuerung und Überwachung durch das PrMS macht die Abläufe transparent und jederzeit nachvollziehbar. Auf diese Weise kann das PrMS beispielsweise die Einhaltung vorgegebener Richtlinien gewährleisten. Wissenschaftlich fundierte Konzepte, wie Korrektheitskriterien für Prozesse und deren Anwendung zur Modellier- und Ausführungszeit, vereinfachen die Umsetzung der Abläufe, erhöhen die Robustheit zur Ausführungszeit und verringern den Testaufwand im Vergleich zu herkömmlicher Softwareentwicklung.

Der verstärkte Einsatz von PrMS bei der Realisierung von Informationssystemen sowie die damit einhergehende Trennung von Prozess- und Anwendungslogik hat weitreichende Folgen, ähnlich wie die in der Anwendungsentwicklung seit langem vollzogene Trennung von Anwendungslogik und Daten auf Grundlage von Datenbankmanagementsystemen (DBMS). Das Potential von PrMS wird im Vergleich zu DMBS bisher jedoch kaum genutzt. Dies liegt nicht zuletzt daran, dass wenig fundierte Systemarchitekturen die Praxistauglichkeit von PrMS stark einschränken. Dennoch ist die Architektur von PrMS im Gegensatz zu DBMS bisher kaum Forschungsgegenstand. Bei DBMS wurden wissenschaftliche Konzepte, wie das Serialisierbarkeitsprinzip, deren technische Umsetzung sowie die Systemarchitektur als Bindeglied detailliert untersucht. Obwohl es für PrMS vergleichbare wissenschaftliche Konzepte gibt, etwa Prozessmetamodelle mit klar definierter Ausführungssemantik, fehlen Untersuchungen der technischen Umsetzung, realitätsnahe Implementierungen der Konzepte sowie Diskussionen zu deren Integration in reale Systemarchitekturen. Bei den in der Praxis eingesetzten PrMS ist die Integration fortschrittlicher Konzepte sehr komplex. Insbesondere kann sie zu massiven Einschränkungen des PrMS führen, etwa bezüglich der Leistungsfähigkeit.

Die vorliegende Arbeit leistet einen wichtigen Beitrag, diese Lücke zwischen wissenschaftlichen Konzepten und umfassender Implementierung zu schließen. Ausgangspunkt bildet die ausführliche Diskussion funktionaler und nicht-funktionaler Anforderungen an ein fortschrittliches adaptives PrMS sowie an entsprechende wissenschaftliche Konzepte. Aufbauend auf diesen Anforderungen werden innovative technische Konzepte für die Implementierung eines PrMS entwickelt und im Detail untersucht. Hierzu gehören beispielsweise Datenstrukturen zur Prozessrepräsentation in Primär- und Sekundärspeicher. Bei den entsprechenden Untersuchungen werden insbesondere auch Abhängigkeiten zwischen verschiedenen Anforderungen und technischen Konzepten berücksichtigt. Diese ganzheitliche Betrachtung trägt dazu bei, dass Optimierungen einzelner Aspekte im Zusammenspiel nicht zu einer Verschlechterung führen.

Simulationen ermöglichen eine ausführliche Evaluation und Validation der technischen Konzepte. Eine im Rahmen dieser Arbeit entwickelte und im Detail beschriebene Systemarchitektur eines adaptiven PrMS integriert die technischen Konzepte und erfüllt sowohl alle funktionalen als auch nicht-funktionalen Anforderungen an ein fortschrittliches adaptives PrMS. Eine umfassende Machbarkeitsimplementierung validiert die Architektur und die entwickelten Konzepte im praktischen Einsatz.

Inhaltsverzeichnis

I Grundlagen und Anforderungen **1**

1 Einleitung **3**
1.1 Problemstellung 4
1.2 Beitrag der Arbeit 6
1.3 Gliederung und Aufbau der Arbeit 7

2 Grundlagen **9**
2.1 Prozessmanagement und Adaptivität 9
2.2 ADEPT-Metamodell 12
2.2.1 Konzeptionelles Objektmodell 12
2.2.2 Modellelemente 15
2.2.3 Ausführungsmodell 23
2.3 ADEPT-Änderungsmodell 30
2.3.1 Änderungsoperationen 31
2.3.2 Änderungsszenarien 34
2.4 Anwendungsintegration (*Plug and Play*) 38
2.4.1 Modellierzeit 40
2.4.2 Laufzeit 42

3 Anforderungen an Architektur und Implementierung **45**
3.1 Ermittlung der Anforderungen 45
3.2 Funktionale Anforderungen 46
3.3 Nicht-funktionale Anforderungen 49

II Konzepte, Architektur, technische Umsetzung **55**

4 Speicherung und Verwaltung unveränderter Prozessinstanzen **57**
4.1 Speicherrepräsentation 58
4.1.1 Partitionierung von Prozessmodellen 58
4.1.2 Prozessvorlagen 59
4.1.3 Prozessinstanzen 63
4.1.4 Quantitativer Vergleich der Prozessinstanzrepräsentationen 69
4.1.5 Weitere Aspekte 72
4.2 Physische Speicherstrukturen 76
4.2.1 Primärspeicher 76
4.2.2 Sekundärspeicher 78

4.3 Zusammenfassung 80

5 Speicherung und Verwaltung veränderter Prozessinstanzen 81
5.1 Realisierungskonzepte 81
5.1.1 Variante ÄR-1: Individuelle Instanzgraphen 83
5.1.2 Variante ÄR-2: Operationale Instanzdeltas 85
5.1.3 Variante ÄR-3: Strukturelle Instanzdeltas 86
5.2 Qualitativer Vergleich der Realisierungskonzepte 88
5.3 Quantitativer Vergleich der Realisierungskonzepte 89
5.3.1 Variante ÄR-1: Individuelle Instanzgraphen 90
5.3.2 Variante ÄR-2: Operationale Instanzdeltas 91
5.3.3 Variante ÄR-3: Strukturelle Instanzdeltas 91
5.3.4 Fazit 91
5.4 Implementierungsvarianten struktureller Instanzdeltas 92
5.4.1 Variante ÄR-3-1: Substitutionsvorlagen 93
5.4.2 Variante ÄR-3-2: Gekapselte Vorlagen 96
5.4.3 Qualitativer Vergleich der Implementierungsvarianten 98
5.5 Quantitativer Vergleich der Implementierungsvarianten 99
5.5.1 Variante ÄR-3-1: Substitutionsvorlagen 99
5.5.2 Variante ÄR-3-2: Gekapselte Vorlagen 100
5.5.3 Fazit 100
5.6 Physische Speicherstrukturen im Sekundärspeicher 101
5.7 Verwandte Arbeiten 103
5.8 Zusammenfassung 103

6 Clusterung von Prozessinstanzen aufgrund ihrer Zustände 105
6.1 Konzept zur Clusterung unveränderter Prozessinstanzen 105
6.2 Anwendungsfälle 107
6.2.1 Verträglichkeitsprüfung bei Schemaevolution 108
6.2.2 Änderungspropagation bei Schemaevolution 111
6.2.3 Weiterschalten 112
6.3 Quantitative Betrachtungen 113
6.3.1 Sequentielle Prozesse 113
6.3.2 Alternative Verzweigungen 113
6.3.3 Parallele Verzweigungen 114
6.3.4 Gesamter Speicherbedarf 115
6.4 Varianten der Clusterung unveränderter Prozessinstanzen 116
6.4.1 Variante IC-1: Zustandsaggregation 117
6.4.2 Knotenaggregation 118
6.4.3 Variante IC-3: Späte Clustererzeugung 123
6.4.4 Fazit 124
6.5 Clusterung veränderter Prozessinstanzen 124
6.6 Zusammenfassung 125

7 Konzeption und Architektur eines adaptiven Prozessmanagementsystems 127
7.1 Gesamtarchitektur 127

7.2 Datenmodelle . . . 130
7.2.1 Allgemeine Entitäten der Datenmodelle . . . 131
7.2.2 Aktivitätenmodell . . . 135
7.2.3 Prozessmodell . . . 144
7.2.4 Ausführungsmodell (*Execution Model*) . . . 152
7.2.5 Laufzeitumgebungsmodell (*Runtime Environment*) . . . 159
7.2.6 Arbeitslistenmodell . . . 162
7.2.7 Organisationsmodell . . . 168
7.3 Dienste . . . 173
7.3.1 Verwaltung von Aktivitätenvorlagen (*ActivityRepository*) . . . 174
7.3.2 Verwaltung von Prozessvorlagen (*ProcessRepository*) . . . 176
7.3.3 Verwaltung des Organisationsmodells (*OrgModelManager*) . . . 178
7.3.4 Verwaltung laufender Prozesse (*ProcessManager*) . . . 180
7.3.5 Verwaltung von Prozessdaten (*DataManager*) . . . 183
7.3.6 Prozessinterpretation und -ausführung (*ExecutionManager*) . . . 185
7.3.7 Verwaltung von Arbeitslisten (*WorklistManager*) . . . 188
7.3.8 Ausführung von Anwendungsbausteinen (*RuntimeManager*) . . . 192
7.3.9 Prozessänderungen (*ChangeOperations*) . . . 200
7.3.10 Weitere Dienste . . . 202
7.4 Programmierschnittstelle (API) . . . 210
7.4.1 Verwendung als Bibliothek . . . 211
7.4.2 Programmiersprachenunabhängige Verwendung (*Web Services*) . . . 213
7.5 Umsetzung der funktionalen Anforderungen . . . 215
7.6 Umsetzung der nicht-funktionalen Anforderungen . . . 216
7.6.1 Skalierbarkeit/Leistungsfähigkeit . . . 217
7.6.2 Erweiterbarkeit . . . 219
7.6.3 Unabhängigkeit/Modularität und Wiederverwendung/Redundanzvermeidung . . . 221
7.7 Zusammenfassung . . . 222

III Validation **225**

8 Simulation **227**
8.1 Unveränderte Prozessinstanzen . . . 227
8.1.1 Testumgebung . . . 228
8.1.2 Ablauf der Messung . . . 229
8.1.3 Speicherbedarf . . . 231
8.1.4 Laufzeit von Zugriffen . . . 232
8.1.5 Fazit . . . 245
8.2 Veränderte Prozessinstanzen . . . 247
8.2.1 Testumgebung . . . 248
8.2.2 Ablauf der Messung . . . 248
8.2.3 Messergebnisse für den Speicherbedarf . . . 249
8.2.4 Messergebnisse für die Laufzeit von Zugriffen . . . 251
8.2.5 Fazit . . . 252
8.3 Zusammenfassung . . . 253

9 Machbarkeitsimplementierung **255**
9.1 Prozessausführung . . . 255
9.2 Protokollierung . . . 256
9.3 Organisationsmodell . . . 259
9.4 Arbeitslistenverwaltung . . . 259
9.5 Prozessüberwachung . . . 262
9.6 Anwendungsintegration und -aufruf . . . 263
9.7 Prozessmodellierung . . . 264
9.8 Instanzspezifische Änderungen . . . 271
9.9 *Plug and Play* . . . 272
9.10 Prozessverwaltung und -versionierung . . . 277
9.11 Programmierschnittstelle . . . 278
9.12 Zusammenfassung . . . 284

IV Diskussion und Zusammenfassung **285**

10 Stand der Technik **287**
10.1 Vergleichskriterien . . . 287
10.2 Akademische Ansätze und Forschungsprototypen . . . 288
10.2.1 Chautauqua . . . 288
10.2.2 Milano . . . 289
10.2.3 YAWL . . . 290
10.2.4 MOBILE . . . 291
10.2.5 DYNAMITE . . . 294
10.2.6 MOKASSIN . . . 295
10.2.7 WASA . . . 296
10.2.8 PHILharmonicFlows . . . 298
10.3 Systeme aus der Praxis . . . 299
10.3.1 Activiti . . . 299
10.3.2 *IBM WebSphere* . . . 300
10.4 Zusammenfassung . . . 302

11 Zusammenfassung und Ausblick **305**

Literaturverzeichnis **311**

V Anhang **333**

A Definitionen **335**

B UML-Referenz **339**
B.1 Zustandsdiagramm . . . 339
B.2 Klassendiagramm . . . 340
B.3 Sequenzdiagramm . . . 341

C Algorithmen **343**
C.1 Zustandsbestimmung bei impliziten Zuständen 343
C.2 Instanzspezifische Änderungen 344
C.2.1 Substitutionsvorlagen 344
C.2.2 Gekapselte Vorlagen 350

D Verwendete Abkürzungen für qualitativen Vergleich **355**

Teil I

Grundlagen und Anforderungen

1 Einleitung

Die digitale Revolution hat in den letzten 20 Jahren massive Effizienz- und Qualitätssteigerungen ermöglicht, insbesondere durch Informationssysteme zur digitalen Verwaltung von Kunden-, Auftrags- und Produktdaten (s. Abb. 1.1). Die zusätzliche Integration von Metainformationen, wie Aufbau- und Ablaufstrukturen eines Unternehmens, ermöglicht weitere Optimierungen bis hin zur umfassenden Integration ganzer Geschäftsprozesse in die betrieblichen Informationssysteme. Dies dient Dokumentationszwecken sowie der Verbesserung der Prozessqualität [LR13], und es vereinfacht die Kommunikation zwischen Organisationseinheiten. Durch entsprechende Ablaufautomatisierung lässt sich die Kommunikation sogar vollständig ersetzen. Auf diese Weise lässt sich die Effizienz ebenso verbessern wie die Qualität der Abläufe und damit auch die der Produkte oder Dienstleistungen des Unternehmens.

Die Erweiterung von Informationssystemen um den Aspekt der Prozessunterstützung erhöht die Komplexität und damit den Aufwand für ihre Entwicklung und Wartung. So wirft die Umsetzung betrieblicher Abläufe zahlreiche Anforderungen auf, beispielsweise die automatische Zuordnung von Aufgaben zu Benutzern, die Überwachung von Abläufen, die Eskalation bei Zeitüberschreitungen sowie die Nachvollziehbarkeit der Abläufe, etwa zwecks Überprüfbarkeit der Einhaltung gesetzlicher Vorschriften. Da diese Anforderungen unabhängig von der Anwendungsdomäne eines Informationssystems bestehen, werden sie idealerweise auch unabhängig und separat vom Informationssystem implementiert. Dadurch vereinfacht sich einerseits das Informationssystem entsprechend, andererseits wird das System zur Ablaufsteuerung und Kontrolle komplexer. Allerdings wird ein solches Prozeßmanagementsystem (PrMS) nur einmal implementiert und kann dann in verschiedene Informationssysteme integriert werden. Idealerweise ist das PrMS möglichst allgemein gestaltet, um eine breite Nutzbarkeit zu erzielen. Eine hohe Allgemeingültigkeit erzielt man durch Abstraktion, in diesem Fall durch die Abstraktion der betrieblichen Abläufe/Geschäftsprozesse.

Dies lässt sich mit der Datenabstraktion in Informationssystemen mit Hilfe von Datenbankmanagementsystemen (DBMS) vergleichen. Hier hat sich das relationale Datenmodell [Cod70] durchgesetzt. Dieses logische Modell abstrahiert von der physischen Repräsentation der Daten und etabliert so die Unabhängigkeit vom Informationssystem (s. Abb. 1.2). So können dieselben Daten über das DBMS auch von anderen (Informations-)Systemen genutzt werden. Die Abstraktion erlaubt es, Daten in gewissem Rahmen zu ändern, ohne dass die

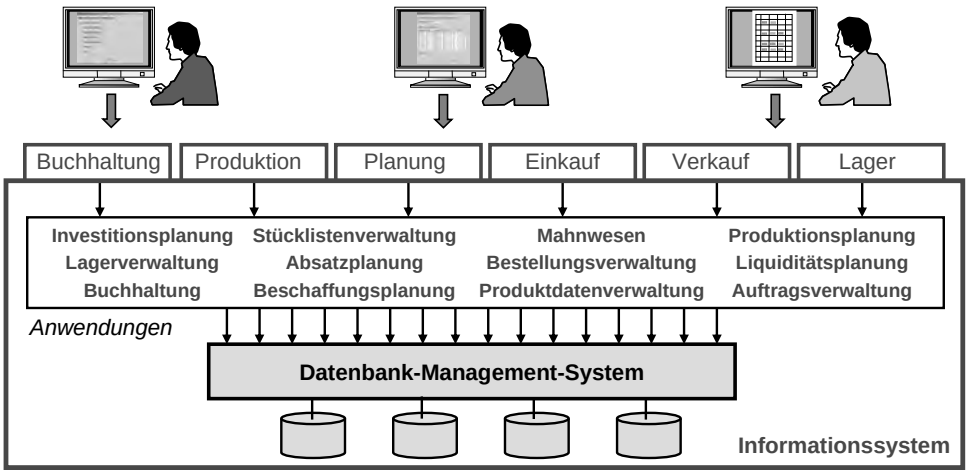


Abbildung 1.1: Beispiel für ein betriebliches Informationssystem (aus [Dad13] entnommen)

verwendenden Systeme angepasst werden müssen. Zudem kann das DBMS selbst optimiert werden, wovon alle verwendenden Systeme profitieren. Wie bei der Separation betrieblicher Abläufe in ein PrMS ergeben sich auch an DBMS zusätzliche Anforderungen. Hierzu gehören neben einer hohen Leistungsfähigkeit verschiedene Eigenschaften wie Mehrbenutzerbetrieb, Fehlertoleranz und damit einhergehend die Synchronisation von lesenden und schreibenden Zugriffen auf das DBMS.

Die Leistungsfähigkeit ist auch die wichtigste Anforderung an ein PrMS zur Verwaltung und Kontrolle betrieblicher Abläufe. Darüber hinaus gibt es viele weitere Anforderungen. Ein großer Teil ergibt sich einerseits daraus, dass an einem Prozess viele verschiedene Personen beteiligt sind, deren Zusammenarbeit koordiniert werden muss. Andererseits müssen Prozesse (teil-) automatisiert ausgeführt werden. Die Ablaufautomatisierung sollte sich auf die Unterstützung der Benutzer beschränken, d. h. Automatisierung ist kein Selbstzweck und darf die Benutzer des PrMS nicht unnötig einschränken. Ohne Unterstützung durch ein Informationssystem werden die betrieblichen Abläufe durch die beteiligten Personen „gelebt“. Die Personen können so flexibel auf Probleme oder Änderungen reagieren und die Abläufe entsprechend adaptieren. Diese Flexibilität ist unabdingbar, damit ein PrMS als Hilfe und nicht als Hindernis bei der Arbeit angesehen wird [RRMD09, RW12].

Ziel dieser Arbeit ist die Entwicklung und Diskussion der Konzepte, Architektur und Implementierung eines PrMS, mit besonderem Augenmerk auf Leistungsfähigkeit, Adaptivität sowie allen für den Produktiveinsatz wichtigen Anforderungen. Zudem kann das PrMS sowohl als Grundlage für ein Informationssystem, als auch eigenständig verwendet werden.

1.1 Problemstellung

Die Erweiterung von Informationssystemen um die Steuerung und Kontrolle betrieblicher Abläufen ist der nächste logische Schritt nach der Datenabstraktion durch DBMS. PrMS sind aber längst noch nicht so verbreitet wie DBMS. Woran liegt das?

Die Umsetzung des relationalen Datenmodells [Cod70] in einem produktiv einsetzbaren DBMS ist schon seit mehreren Jahrzehnten Gegenstand umfangreicher Forschung. Neben vielen speziellen Teilaspekten von DBMS (z. B. [BM72, Lit80, BHG87]) gibt es zahlreiche

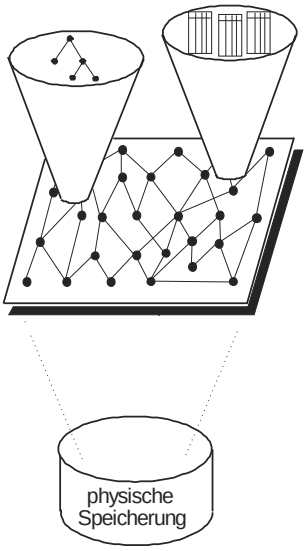


Abbildung 1.2: Abstraktion der Daten bei DBMS (aus [Dad13] entnommen)

Veröffentlichungen, die das System als Ganzes betrachten, also die DBMS-Architektur (z. B. [Sto81, GMB+81, GMUW02]). Obwohl sich PrMS semantisch oberhalb von DBMS befinden und zahlreiche Metamodelle existieren [vdAtHKB03, Rei00, KGD99, OAS07, OMG09], gibt es nur wenige Arbeiten zur Architektur und Implementierungskonzepten von PrMS (z. B. [LR99, Jab97, Wes99]). Dies erklärt auch, warum heutige PrMS noch nicht den Reifegrad vonDBMS erreichen und daher nicht in der Breite eingesetzt werden. Für ein leistungsfähiges System sind Architekturbetrachtungen ebenso unabdingbar wie Untersuchungen zu Implementierungskonzepten; sie haben in hohem Maße zum Erfolg von DBMS beigetragen.

Ein großes Problem ist dabei der funktionale Umfang eines allgemeingültigen PrMS. Die Hauptaufgabe eines PrMS liegt in der Steuerung und Kontrolle betrieblicher Abläufe. Dazu müssen diese maschinenlesbar repräsentiert werden, was durch Prozessmodellierung erfolgt. Für die Steuerung von Abläufen müssen zudem einzelne Prozessschritte den jeweiligen Benutzern (sog. Bearbeitern) zugewiesen werden. Hierbei müssen fachliche und rechtliche Aspekte berücksichtigt werden, etwa welcher Bearbeiter für die Durchführung einer Aufgabe befähigt ist oder welcher Bearbeiter welche Daten lesen darf. Dies erfordert, dass das PrMS die Bearbeiter kennt und benötigt dementsprechend eine Benutzerverwaltung ähnlich wie in einem DBMS. Bei einem PrMS ist diese jedoch umfangreicher, da sie weitergehende fachliche Aspekte und größere Teile der Organisationsstruktur abbildet (z. B. Abteilungen und Projekte).

Eine Effizienzsteigerung durch das PrMS lässt sich durch die Automatisierung von Prozessen (oder Teilen davon) erreichen. Hierzu müssen Aufgaben im Prozess (Prozessschritte) so realisiert und implementiert werden, dass sie das System direkt ausführen kann. Auch diese Aufgabenimplementierungen können durch Abstraktion allgemeingültig und damit wiederverwendbar gestaltet werden. Auf diese Weise lässt sich dieselbe Aufgabenimplementierung, etwa die Ermittlung von Kundendaten, in unterschiedlichen Prozessimplementierungen verwenden. Die Wiederverwendung beschränkt sich nicht nur auf automatische Prozessschritte sondern ist auch für Prozessschritte mit Benutzerinteraktion möglich.

Möglichst benutzerfreundlich, einfach und fehlerfrei funktioniert die Wiederverwendung, wenn die Aufgabenimplementierungen zentral hinterlegt und mit Zusatzinformationen angerei-

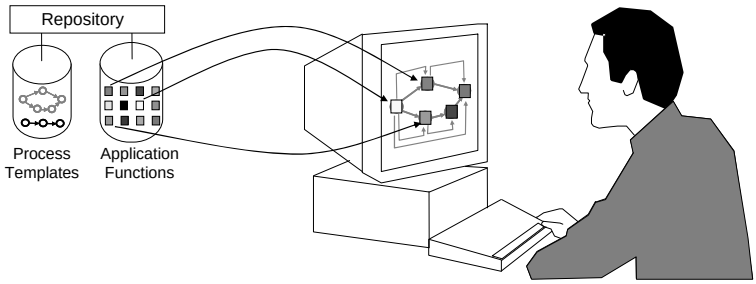


Abbildung 1.3: *Plug and Play* mit Anwendungsbausteinen(aus [Dad97] entnommen)

chert werden. So lässt sich die Prozessmodellierung mittels *Plug and Play* [Dad97] realisieren (s. Abb. 1.3). Bei dieser werden die einzelnen Prozessschritte auf einer graphischen Oberfläche durch Ziehen mit der Maus mit den entsprechenden Aufgabenimplementierungen verknüpft. Durch Zusatzinformationen, etwa den von der Aufgabenimplementierung benötigten und erzeugten Daten, kann das System bereits bei der Modellierung überprüfen, ob die Daten bei der Ausführung auch tatsächlich vorhanden sind und keine notwendigen Eingabedaten fehlen. Dies verringert den Testaufwand für Prozessimplementierungen signifikant.

Die Kontrolle der Abläufe ist in einem PrMS einfach und effizient durchzuführen. Dazu gehört z. B. die Eskalation bei Zeitüberschreitungen [LWR14] oder eine Reaktion auf fehlgeschlagene Prozessschritte [LRD10b]. Da zum Zeitpunkt der Prozessmodellierung jedoch meist nicht alle möglichen Situationen und die entsprechenden Reaktionen bekannt sind, können die Reaktionen nicht in jedem Fall automatisch erfolgen. Dies muss bei Bedarf von Benutzern entschieden werden, die vom System informiert werden. Ein solcher Benutzereingriff kann auch im normalen Prozessablauf notwendig werden, etwa wenn ein unvorhergesehener Sonderfall auftritt und der betriebliche Ablauf angepasst werden muss [RHD98]. Die Anpassung kann einerseits spezifisch für einen Prozess sein, andererseits können gesetzliche Änderungen alle zukünftigen oder sogar aktuell noch ausgeführte Prozesse treffen [RRD04c, RWRW05]. Da solche Änderungen oftmals durch fachliche und nicht durch technische Gründe bedingt sind, werden fachliche Bearbeiter bei der Problembeseitigung meist mit eingebunden. Flexibler ist es, wenn die fachlichen Benutzer das Problem selbst beseitigen können und sie keine technische Unterstützung durch IT-Spezialisten benötigen. Beim Ändern ist eine Unterstützung des Benutzers durch das System hilfreich, da so Fehler vermieden werden können.

Aufgrund der häufigen Benutzerinteraktion, etwa bei der Ausführung von Prozessschritten oder der Durchführung von Änderungen, muss das PrMS trotz der umfangreichen Funktionen leistungsfähig sein. Das heißt, das System muss auf Benutzeranfragen in angemessener Zeit reagieren. Da die Reaktionsfähigkeit auch von den technischen Ressourcen abhängt, muss das System gut skalieren, d. h., eine Erhöhung der Ressourcen steigert die Reaktionsfähigkeit entsprechend. Aufgrund des Umfangs eines PrMS und infolge der Abhängigkeiten der Funktionen untereinander, ist die Etablierung von Leistungsfähigkeit und Skalierbarkeit [Bau01, BRD03] anspruchsvoll und erfordert umfassende Architekturbetrachtungen.

1.2 Beitrag der Arbeit

Die vorliegende Arbeit entstand zum großen Teil im Rahmen des vom Land Baden-Württemberg geförderten Projekts *AristaFlow* [DRR+07]. Ein Schwerpunkt dieses Projekts bildete der

Entwurf der Architektur und die Implementierung eines leistungsfähigen PrMS. Des weiteren wurde die Integration von Softwareartefakten (*Anwendungsbausteine*) für Prozessschritte untersucht, um eine möglichst benutzerfreundliche Prozessmodellierung im *Plug and Play*-Stil zu ermöglichen. Beide Schwerpunkte sind Gegenstand dieser Arbeit.

Funktionale und nicht-funktionale Anforderungen sind grundlegend für den Entwurf eines Systems. Dementsprechend bilden sie die Grundlage für die vorliegende Arbeit. Auf ihnen baut einerseits die Architektur auf, andererseits werden damit existierende PrMS verglichen. Eine große Rolle spielen Adaptivität als funktionale und Leistungsfähigkeit als nicht-funktionale Anforderung. Da momentan nur das Prozessmetamodell ADEPT (*Application development based on encapsulated process templates* [DKR+95, Rei00]) eine umfassende und theoretisch fundierte Adaptivität bereitstellt [WRMR07, MRH08], nämlich sowohl für Änderungen an Prozessinstanzen als auch Prozessvorlagen, orientiert sich diese Arbeit an ADEPT. Die Diskussionen lassen sich aber problemlos auf andere Prozessmetamodelle übertragen.

Die beiden Aspekte Adaptivität und Leistungsfähigkeit wurden bisher noch nicht gemeinsam untersucht, dementsprechend widmet sich diese Arbeit ausführlich verschiedenen Konzepten zur Repräsentation von Prozessvorlagen und -instanzen innerhalb eines PrMS. Dies umfasst verschiedene Varianten zur Datenrepräsentation für Prozessvorlagen sowie unveränderte und veränderte Prozessinstanzen in Primär- und Sekundärspeicher. Die Repräsentationsvarianten werden anhand vorgegebener Kriterien evaluiert und mittels einer Implementierung simuliert.

Des weiteren stellt diese Arbeit eine umfassende Architektur eines adaptiven PrMS vor, die alle funktionalen und nicht-funktionalen Anforderungen erfüllt. Zur Architektur gehören die Datenmodelle sowie die Dienste, die die Funktionen des Systems realisieren. Die Datenmodelle bilden zusammen mit den Schnittstellen der Dienste die Programmierschnittstelle für die Verwendung des PrMS als Entwicklungsplattform für prozessorientierte (Informations-)Systeme. Die gesamte Architektur ist in der *AristaFlow BPM Suite* [DRRM+09] umgesetzt.

1.3 Gliederung und Aufbau der Arbeit

Die vorliegende Arbeit besteht aus vier Teilen (s. Abb. 1.4): Im einleitenden Teil I wird die Problemstellung erläutert. Anschließend werden die Grundlagen vorgestellt, die für das Verständnis der Arbeit hilfreich sind (Kapitel 2). Hierzu gehören das ADEPT-Metamodell, die ADEPT-Änderungsoperationen und die ADEPT-Anwendungsintegration (*Plug and Play*). Am Ende des einleitenden Teils werden in Kapitel 3 die funktionalen und nicht-funktionalen Anforderungen an ein adaptives PrMS beschrieben und diskutiert.

In Teil II werden die Konzepte zur Speicherrepräsentation unveränderter (Kapitel 4) und veränderter (Kapitel 5) Prozessinstanzen vorgestellt und technisch umgesetzt. Anschließend stellt Kapitel 6 eine Variante zur Repräsentation von Prozessinstanzdaten vor, die für Änderungen an Prozessvorlagen optimiert ist. In Kapitel 7 beschreiben wir ausführlich die Architektur eines adaptiven PrMS.

Eine Validation sowohl der entwickelten Implementierungskonzepte als auch der Architektur ist Gegenstand von Teil III. Dort werden zum einen die Varianten zur Speicherrepräsentation simuliert (Kapitel 8), zum anderen die Implementierung der Architektur durch die *AristaFlow BPM Suite* sowie Beispiele für die Verwendung als Plattform vorgestellt (Kapitel 9).

Teil IV schließt die Arbeit mit der Diskussion verwandter Arbeiten (Kapitel 10) sowie einer Zusammenfassung und einem Ausblick (Kapitel 11) ab.

I	Grundlagen und Anforderungen	1	Einleitung
		2	Grundlagen
		3	Anforderungen an Architektur und Implementierung
II	Konzepte, Architektur, technische Umsetzung	4	Speicherung und Verwaltung unveränderter Prozessinstanzen
		5	Speicherung und Verwaltung veränderter Prozessinstanzen
		6	Clusterung von Prozessinstanzen aufgrund ihrer Zustände
		7	Konzeption und Architektur eines adaptiven PrMS
III	Validation	8	Simulation
		9	Machbarkeitsimplementierung
IV	Diskussion und Zusammenfassung	10	Stand der Technik
		11	Zusammenfassung und Ausblick

Abbildung 1.4: Aufbau der Arbeit

2 Grundlagen

In diesem Kapitel stellen wir Grundlagen der Arbeit vor, die für das weitere Verständnis hilfreich sind. Dies umfasst Prozessmanagement allgemein, aber besonders auch adaptives Prozessmanagement, d. h. Prozessmanagement bei dem Prozesse während ihrer Ausführung in ihrer Struktur und damit in ihrem Verhalten flexibel geändert werden können. Mögliche Änderungen sind beispielsweise das Einfügen, Löschen oder Verschieben von Prozessschritten.

Damit ein Prozess überhaupt elektronisch unterstützt ausgeführt werden kann, muss er als ausführbares Prozessmodell vorliegen [RW12]. Dieses Prozessmodell muss einem Metamodell genügen, das wiederum eine klar definierte formale und operationale Semantik besitzt. Dies schließt auch Adaptivität mit ein, die daher im Metamodell verankert sein sollte. Für adaptives Prozessmanagement hat sich das ADEPT-Metamodell [Rei00] etabliert, auf das wir in den folgenden Abschnitten detailliert eingehen. Nach adaptivem Prozessmanagement im allgemeinen stellen wir das Metamodell vor, das für die Modellierung und Ausführung von Prozessen dient. Danach gehen wir auf das ADEPT-Änderungsmodell ein, das Adaptivität auf verschiedenen Ebenen ermöglicht. Im letzten Abschnitt betrachten wir, wie in ADEPT die Integration von Anwendungen erfolgt. Die Anwendungsintegration ist wichtig, um Prozessmanagement breit in der Praxis und in bestehenden IT-Infrastrukturen einsetzen zu können.

2.1 Prozessmanagement und Adaptivität

Prozessmanagement und damit auch PrMS haben zum Ziel, die betrieblichen Abläufe in Informationssysteme zu integrieren und durch das System unterstützt auszuführen [DRRM11, RD00]. Dadurch sollen die Abläufe explizit und nachvollziehbar gemacht werden. Bisher sind die betrieblichen Abläufe meist nur in den Köpfen der Mitarbeiter oder graphisch modelliert; sie werden jedoch nicht von einem PrMS ausgeführt (s. Abb. 2.1). Sind Abläufe in ein Informationssystem integriert, geschieht dies oftmals direkt auf Implementierungsebene. Da monolithische Informationssysteme schlecht wiederverwendbar und wartbar sind, sind sie heutzutage Gegenstand von serviceorientierten Architekturen (SOA) [Erl05, RS04]. Ältere Systeme werden entsprechend umgebaut, neue Systeme werden dienstorientiert implementiert. Häufig gibt es in einem Betrieb auch mehrere Informationssysteme für verschiedene Zwecke,

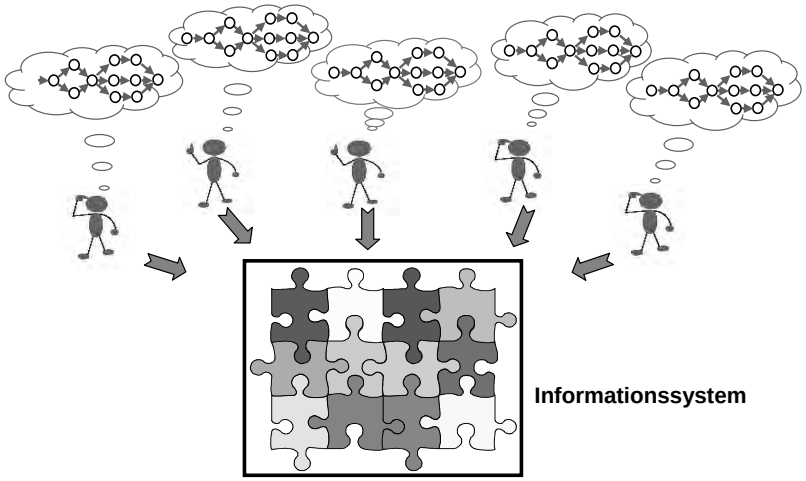


Abbildung 2.1: Monolithisches Informationssystem (aus [Dad13] entnommen)

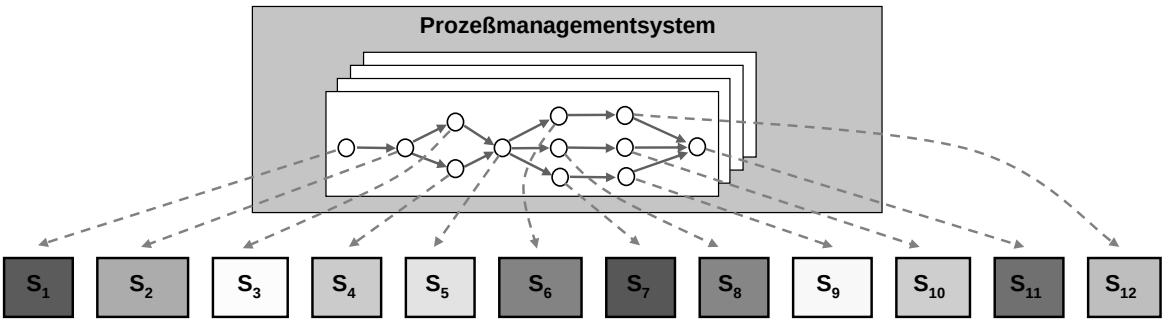


Abbildung 2.2: Verknüpfung von Aufgaben/Prozessschritten mit Implementierungen (aus [Dad13] entnommen)

die jedoch innerhalb eines betrieblichen Ablaufs zusammen genutzt werden. Die notwendige Interaktion zwischen den Systemen und ihre Verwendung ist meist kompliziert. Hier kann EAI (*Enterprise Application Integration*) [Pry05, LHHR05] helfen. EAI koppelt verschiedene Systeme an eine gemeinsame Plattform und integriert sie so. Hierdurch ergeben sich Vorteile für die Benutzer und Implementierer.

Beide Situationen können signifikant von Prozessmanagement profitieren: Sowohl SOA als auch EAI haben die Zerlegung eines oder mehrerer Systeme zum Gegenstand. Ziel der Zerlegung ist es, die einzelnen Teile anschließend einfach von außen aufzurufen zu können und deren Kommunikation mit der Außenwelt zu verbessern. Dabei findet meist auch eine Reduktion der Schnittstelle nach außen statt (hauptsächlich bei EAI) oder eine Gruppierung nach funktionalen Aspekten (hauptsächlich bei SOA). Betrachtet man diese reduzierten oder gruppierten Teile als eigenständige Artefakte, lassen sich diese relativ einfach an Prozessschritten hinterlegen (s. Abb. 2.2). Die von den Artefakten benötigten Eingabedaten und erzeugten Ausgabedaten werden über den Prozess und damit durch das PrMS geschleust. Auf diese Weise werden über einen Prozess verschiedene Teile eines oder mehrerer Informationssysteme verknüpft und die Systeme somit in und über den Prozess integriert. Damit ergänzt Prozessmanagement SOA ideal [Man]. Gleichzeitig verhält sich das PrMS wie eine EAI-Plattform.

Werden die Artefakte geeignet gekapselt, können sie in verschiedenen Prozessen (wieder-)verwendet werden. Hierzu werden die Artefakte (oder Stellvertreterobjekte) an einer zentralen Stelle hinterlegt. Durch zusätzliche Metainformationen kann Verwendung der Artefakte signifikant verbessert und vereinfacht werden [Buc12]. Bei der Prozessmodellierung wählt der Benutzer ein geeignetes Artefakt von der zentralen Stelle aus und hinterlegt es an einem Prozessschritt. Stehen dabei Informationen zu den Ein-/Ausgabedaten zur Verfügung, kann das PrMS die Daten des Artefakts geeignet in den Prozess integrieren. Dies kann soweit gehen, dass das PrMS die Ein-/Ausgabedaten aufeinanderfolgender Artefakte automatisch verbindet und so einen Datenfluss zwischen den Artefakten herstellt. Im Idealfall erfolgt dieses *Plug and Play* [Dad97] vollautomatisch, so dass der Prozessmodellierer nur die Artefakte auswählt und ihre Reihenfolge festlegt; der Datenfluss wird komplett vom PrMS erzeugt. Dabei kann das PrMS überprüfen, ob der Datenfluss korrekt ist. Beispielsweise kann es so verhindern, dass ein Artefakt Daten zwingend benötigt, die von vorherigen Artefakten aber nur optional erzeugt werden. Zur Laufzeit würde dies zu einem Fehler führen, der ggf. innerhalb des Artefakts auftritt und dementsprechend oft wenig aussagekräftig und schlecht nachzuvollziehen ist.

Bei der Prozessausführung wird allen Prozessschritten ein definierter Zustand zugeordnet. Der erste Prozessschritt erhält initial einen Zustand, der ihn als ausführbar kennzeichnet; alle nachfolgenden Prozessschritte sind als inaktiv gekennzeichnet. Benutzer des PrMS, die Prozessschritte ausführen sollen, bekommen diese Prozessschritte in Form einer *Arbeitsliste* angezeigt (s. Abb. 2.3). Sobald ein Prozessschritt als ausführbar gekennzeichnet wird, wird er in die Arbeitslisten der bei der Prozessmodellierung spezifizierten Bearbeiter hinzugefügt. Die eigentliche Ausführung des Prozessschritts wird durch einen dieser Bearbeiter gestartet. Dabei wird das am Prozessschritt hinterlegte Artefakt aufgerufen. Seine Eingabedaten werden, falls vorhanden, vom PrMS beim Start bereitgestellt. Beim Beenden werden die Ausgabedaten des Artefakts vom PrMS entgegengenommen. Anschließend versieht das PrMS den entsprechenden Prozessschritt als abgeschlossen und führt eine Neubewertung der Prozessschritte durch. Bei diesem *Weiterschalten* werden der oder die nachfolgenden Prozessschritte als ausführbar gekennzeichnet und wiederum den Arbeitslisten der entsprechenden Bearbeiter hinzugefügt. Die Zustandsänderungen und die ausführenden Bearbeiter werden protokolliert. So lässt sich die komplette Ausführung eines Prozesses nachvollziehen.

Auf die bisher beschriebene Weise können Prozesse einfach modelliert und ausgeführt werden. Sie sind jedoch unflexibel. So können die Bearbeiter nicht vom modellierten Prozess abweichen. Dies ist wie bei Abläufen, die direkt auf Implementierungsebene in Informationssysteme integriert sind [SM95]. Treten bei der Ausführung Probleme auf, findet der weitere Ablauf meist außerhalb des PrMS statt. Hierdurch gehen die beschriebenen Vorteile von Prozessmanagement verloren. Adaptives Prozessmanagement umgeht dies, indem ein Prozess auch noch während seiner Ausführung strukturell geändert werden kann. So lassen sich Prozessschritte entfernen oder neue Prozessschritte hinzufügen. Diese (Nach-)Modellierung kann sowohl für einzelne Prozessinstanzen erforderlich sein, etwa bei Ausnahmefällen, als auch für mehrere Prozessinstanzen. Mehrere laufende Prozessinstanzen müssen z. B. aufgrund gesetzlicher Änderungen, die bei der Modellierung noch nicht bekannt waren, geändert werden.

Um den Unterschied zwischen normaler Prozessmodellierung und (Nach-)Modellierung aufgrund von Problemen im Rahmen der Adaptivität möglichst gering zu halten, sollte das PrMS den Benutzer in beiden Fällen ähnlich gut unterstützen. Dies betrifft z. B. die zahlreichen Prüfmöglichkeiten der normalen Prozessmodellierung, etwa zur Feststellung von Inkonsistenzen im modellierten Datenfluss. Darüber sollte das PrMS stets Änderungen verbieten, die zu ungültigen und damit unerwünschten Zuständen führen. Dies ist beispielsweise beim Einfügen

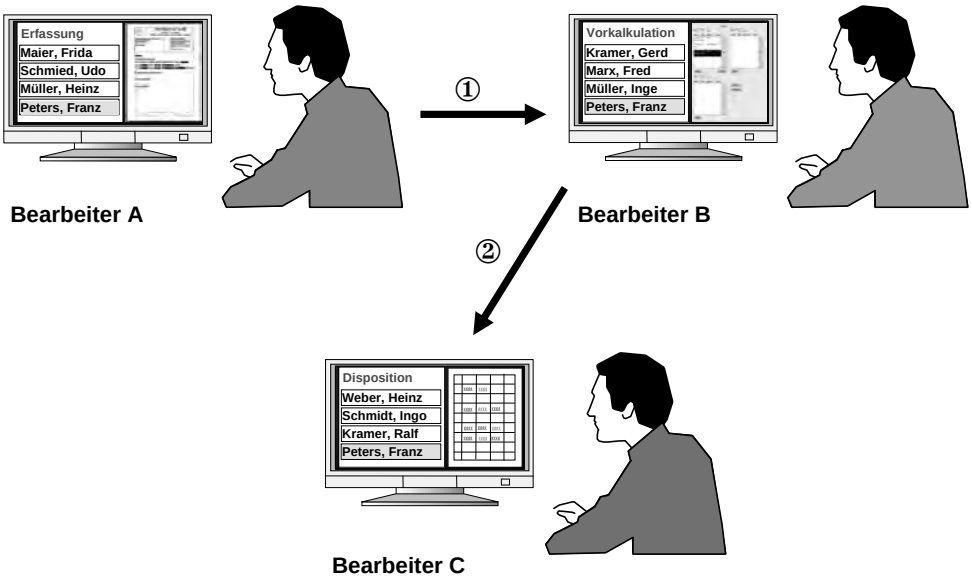


Abbildung 2.3: Aufgaben-/Arbeitslisten bei den entsprechenden Bearbeitern (aus [Dad13] entnommen)

eines neuen Prozessschritts zwischen bereits abgeschlossenen Prozessschritten der Fall, da der neu eingefügte Prozessschritt nie konsistent ausgeführt werden kann. Es sollen auch keine Änderungen erfolgen können, die globale Integritätsbedingungen (sog. *Compliance*) verletzen [LKRM+10].

2.2 ADEPT-Metamodell

In diesem Abschnitt stellen wir den Standardteil des ADEPT-Metamodells vor, der Adaptivitätsaspekte zunächst ausklammert. Dieser beruht im Kern auf den in [Rei00, RD97, RHD98] vorgestellten Ansatz, wir berücksichtigen aber auch Erweiterungen aus nachgelagerten Forschungsarbeiten [Wol08, For09] und den langjährigen praktischen Einsatz [DR09]. Zuerst erläutern wir das konzeptionelle Objektmodell, d. h. wir definieren und etablieren die Begriffe, die für Prozessmanagement und PrMS im allgemeinen und für das ADEPT-Metamodell im speziellen von Bedeutung sind. Anschließend betrachten wir die Syntax und Semantik der statischen und dynamischen Elemente des ADEPT-Metamodells, also Prozessmodelle und deren Ausführung.

2.2.1 Konzeptionelles Objektmodell

Das konzeptionelle Objektmodell ist nicht auf ADEPT beschränkt, sondern gilt generell für Prozessmanagement. Aufgrund des technischen Vorsprungs von ADEPT gegenüber anderen PrMS, etwa in Bezug auf Adaptivität und Anwendungsintegration, ist an einigen Stellen ein umfassenderes Objektmodell notwendig. Dieses erfordert teilweise eine Bedeutungserweiterung oder -einschränkung von im PrMS-Bereich verwendeten Begriffe.

2.2.1.1 Grundlegende Objekte des Prozessmodells

Das zentrale Objekt bei PrMS ist der *Prozess*. Dieser definiert die Ausführung von Prozessschritten in einer definierten Abfolge sowie unter Beachtung wohldefinierter Ausführungsbedingungen. Ein Prozessschritt wiederum ist eine Arbeitseinheit, die eine oder mehrere Tätigkeiten umfasst. Die Tätigkeiten können sowohl durch das System unterstützte als auch manuelle oder automatisch ausgeführte Tätigkeiten sein. Wichtig ist, dass sich die Arbeitseinheiten klar von anderen Arbeitseinheiten abgrenzen. Das heißt, eine Menge von Tätigkeiten lässt sich eindeutig einer Arbeitseinheit zuordnen. Beispielsweise kann die Abgrenzung dadurch definiert werden, dass bestimmte Tätigkeiten von einer anderen Person durchgeführt werden oder ein bestimmter zeitlicher Abstand zwischen zwei Tätigkeiten vorhanden sein muss.

Sinnvollerweise besitzt ein Prozess ein konkretes Ziel und umfasst dementsprechend nur Tätigkeiten, die diesem Ziel dienen [LR12]. Dies ist zwar aus technischer Sicht nicht notwendig, es vereinfacht aber das Verständnis eines Prozesses signifikant. Wir gehen deshalb im folgenden davon aus, dass ein Prozess ein oder mehrere Ziele hat. Damit lassen sich Prozesse auch nach Zielen klassifizieren. Prozesse mit demselben Ziel gehören zum selben *Prozesstyp*, z. B. Bestellprozesse oder Prozesse zur Verarbeitung eingehender Rechnungen. Naheliegenderweise variiert die Klassifikation und damit der Prozesstyp mit dem Abstraktionsgrad der Ziele, etwa eine allgemeine Bestellung oder die Unterscheidung zwischen der Bestellung einer Ware und einer Dienstleistung. Man spricht hierbei auch von Prozessvariabilität [Hal10, HBR10, ATW+13]. Da die Klassifikation aus technischer Sicht aber irrelevant ist, gehen wir in der weiteren Diskussion von beliebig abstrakten Zielen aus.

Jeder Prozess besitzt einen statischen und einen dynamischen Aspekt. Der statische Aspekt ist die Definition der Abfolge von Prozessschritten und Informationen zu den Prozessschritten. Dies umfasst etwa die Reihenfolge der Prozessschritte, die Tätigkeit(en) eines Prozessschritts und deren Bearbeiter. Man spricht auch von der *Struktur* eines Prozesses. Der dynamische Aspekt definiert den Zustand eines Prozesses, der wiederum über den Zustand der einzelnen Prozessschritte definiert ist, etwa welche Prozessschritte bereits ausgeführt wurden, von wem welche Prozessschritte aktuell ausgeführt werden und welche noch zur Ausführung anstehen. Beide Aspekte werden im konzeptionellen Objektmodell unterschieden: der statische Aspekt wird durch eine *Prozessvorlage*, d. h. ein Prozessmodell, repräsentiert, der dynamische Aspekt durch *Prozessinstanzen* (s. Abb. 2.4). Eine Prozessinstanz existiert ab dem Zeitpunkt, ab dem ein Prozess erzeugt und gestartet wird. Beim Start wird zudem festgelegt, wie der gestartete Prozess aussehen soll, d. h. auf welcher Prozessvorlage er ablaufen soll. Damit gehört jede Prozessinstanz eindeutig zu einer Prozessvorlage (s. Abb. 2.4).

2.2.1.2 Prozessänderungen

Speziell beim ADEPT-Metamodell kommt der Aspekt der Adaptivität hinzu, also die strukturelle Änderung von Prozessen. Wir werden diese im folgenden kurz als *Prozessänderungen* oder nur als *Änderungen* bezeichnen, wenn der Bezug klar ist. Prozessänderungen beschreiben hier nicht die normalen Zustandsänderungen bei der Ausführung einer Prozessinstanz, etwa das Starten oder Beenden von Prozessschritten. Vielmehr beziehen sich Prozessänderungen auf die Prozessstruktur.

Prozessänderungen haben verschiedene Geltungsbereiche: Prozessvorlagen oder Prozessinstanzen. Änderungen an Prozessvorlagen betreffen alle Prozesse eines Prozesstyps. Beispielhaft sei das Einfügen eines zusätzlichen Prozessschritts zur Überprüfung der Kreditwürdigkeit bei

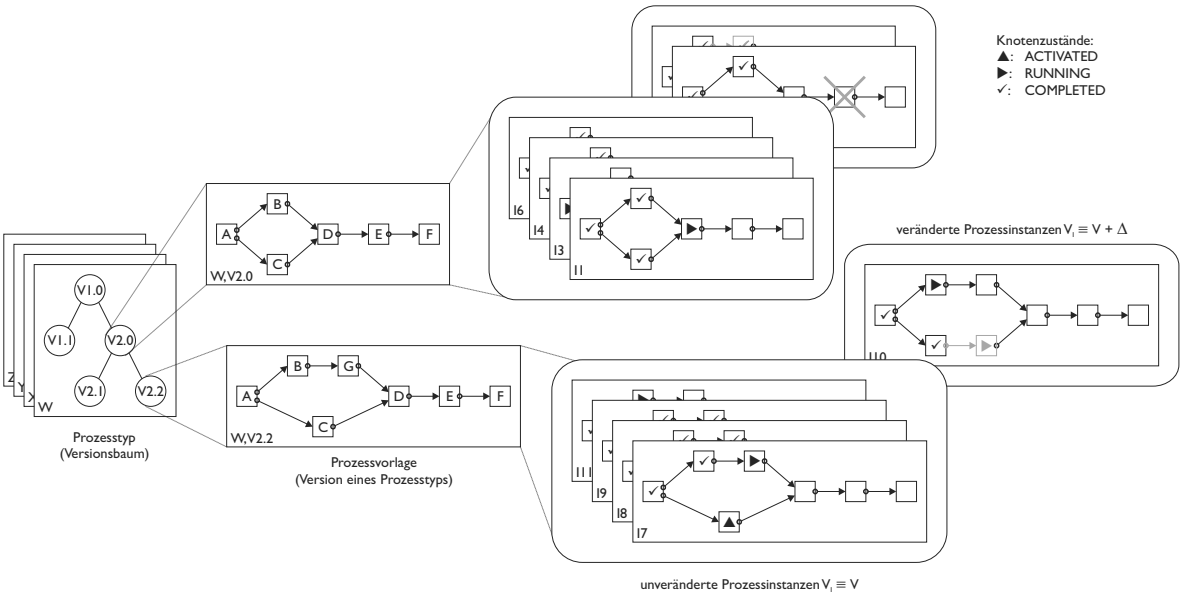


Abbildung 2.4: Konzeptionelles Objektmodell

einem Bestellprozess genannt. Solche Änderungen werden auf Prozessvorlagen angewendet und führen zu einer neuen Prozessvorlage, die sich von der alten Prozessvorlage um eben diese strukturellen Änderungen unterscheidet. Da beide Prozessvorlagen zum selben Prozesstyp gehören, stellt die neue Prozessvorlage eine neue *Version* des Prozesstyps dar. Die Änderungen von Prozessvorlagen müssen nicht linear erfolgen, so dass im Laufe der Zeit ein Versionsbaum für einen Prozesstyp entstehen kann (links in Abb. 2.4). Da Änderungen an Prozessvorlagen zu neuen Prozessvorlagen führen, müssen diese Änderungen nicht sofort wirksam werden. So können die Änderungen nur für die Prozessinstanzen gelten, die ab einem bestimmten Datum (z. B. 1.1.2014) gestartet werden. Für vor diesem Datum gestartete Prozessinstanzen gilt noch die alte Prozessvorlage. Daneben können die Änderungen, speziell bei ADEPT, auch für bereits laufende Prozesse gelten und auf diese – soweit möglich – angewendet werden. Dies erfolgt im Rahmen einer *Schemaevolution* [Rin04, RRD04c, RRD03b, RRD04d, RRD04b]. Naheliegenderweise werden Prozessvorlagen auch nicht direkt geändert, wenn sie zur Instanzierung freigegeben sind. In so einem Fall finden Änderungen immer auf einer Kopie der Prozessvorlage statt [RW12].

Daneben können Prozessänderungen auch nur für bestimmte Prozessinstanzen gelten. Diese maschinenspezifischen Änderungen [Rei00] gelten dann nur für eine konkrete Prozessausführung, etwa den Prozess für eine konkrete Bestellung eines bestimmten Kunden. Andere Kunden und andere Bestellungen desselben Kunden sind von der Änderung nicht betroffen. Man unterscheidet demzufolge unveränderte und veränderte Prozessinstanzen. Veränderte Prozessinstanzen beruhen zwar logisch immer noch auf derselben Prozessvorlage – der *ursprünglichen Prozessvorlage* –, um die Änderungen logisch einfach darstellen zu können, erhalten sie jedoch eine *instanzspezifische Prozessvorlage* (rechts in Abb. 2.4). Diese ist logisch dieselbe Version des Prozesstyps wie vor den Änderungen. Die Durchführung von Prozessänderungen beider Geltungsbereiche sowie die Wechselwirkung zwischen ihnen wird im folgenden Abschnitt 2.3 im Detail dargestellt.

2.2.1.3 Weitere Objekte

Damit die Ausführung eines Prozesses auch nach dessen Ende nachvollzogen werden kann, werden alle Zustandsänderungen in der sog. *Ausführungshistorie* protokolliert (s. Prozessinstanz `I7` in Abb. 2.4). Eine Ausführungshistorie umfasst mindestens die Art der Zustandsänderung, die betroffenen Prozessschritte, das Datum und die Uhrzeit der Zustandsänderung sowie den auslösenden Bearbeiter, z. B. „Abschluss des Versands der 3. Bestellung von Kunde Müller durch Mitarbeiter Schmidt am 3.11.2013 um 9:20 Uhr“ [RW12]. Logisch werden Ausführungshistorien den jeweiligen Prozessinstanzen zugeordnet, d. h. jede Prozessinstanz besitzt ihre eigene Ausführungshistorie. Oftmals wird eine Ausführungshistorie auch *Ausführungsspur* (*Trace*) genannt.

Durch instanzspezifische Änderungen kann der Prozessablauf vom ursprünglich geplanten Ablauf abweichen, d. h. dem durch die entsprechende Prozessvorlage vorgegebenen Ablauf. So können in der Ausführungshistorie Einträge zu zusätzlichen Prozessschritten existieren, die sich nicht durch die Prozessvorlage ergeben. Deshalb besitzen veränderte Prozessinstanzen zusätzlich eine *Änderungshistorie*, in der sämtliche Änderungen gegenüber der ursprünglichen Vorlage protokolliert werden [GRRvdA06, RKL$^+$06, RRJK06].

Es gibt Prozessmetamodelle, bei denen der Prozess die von Prozessschritten gelesenen oder erzeugten Daten kennt (*Prozessdaten*). Beispielsweise erlaubt dies, im Prozess alternative Pfade entsprechend den Daten auszuwählen, etwa zusätzliche Prozessschritte für eine Genehmigung, wenn die Ausgaben einen bestimmten Wert überschreiten. So kann auch der Prozess(instanz)-name beeinflusst werden, z. B. „Bestellung von Kunde Müller“, wobei der Name „Müller“ erst im ersten Prozessschritt bekannt wird. Auch bei diesen Daten werden Zugriffe in einer *Datenhistorie* protokolliert, so dass nachvollzogen werden kann, welcher Prozessschritt welche Daten gelesen oder geschrieben hat. Im Extremfall werden alle gelesenen und geschriebenen Daten vom Prozess bereitgestellt bzw. entgegengenommen. Dies ermöglicht die Ausführung von Prozessschritten in einem *Historienmodus*. Dabei werden bereits abgeschlossene Prozessschritte nochmals „ausgeführt“, wobei nur die gelesenen und geschriebenen Daten angezeigt werden. Es findet keine wirkliche Ausführung statt; die eigentliche Ausführung kann aber einfach nachvollzogen werden. Nützlich ist dies etwa, um eine in einem Prozessschritt getroffene kritische Entscheidung im Kontext der bei der eigentlichen Ausführung zur Verfügung stehenden Daten zu bewerten.

2.2.2 Modellelemente

Die Prozessstrukturen im ADEPT-Metamodell setzen sich aus *Kontroll-* und *Datenfluss* zusammen. Der Kontrollfluss definiert die Reihenfolge, in der Prozessschritte ausgeführt werden. Der Datenfluss gibt an, welche Daten der Prozessschritte im Prozess selbst bekannt sind sowie welche Prozessschritte diese Daten erzeugen und lesen. Wir gehen im folgenden zuerst auf den Kontrollfluss in ADEPT ein. Anschließend betrachten wir den Datenfluss, bevor wir weitere Aspekte beschreiben, die sich durch das Zusammenwirken von Kontroll- und Datenfluss ergeben.

2.2.2.1 Kontrollfluss

Der Kontrollfluss in ADEPT beruht auf gerichteten azyklischen Graphen, die einer Blockstruktur genügen [RD98, Rei00]. Knoten repräsentieren Prozessschritte, Kontrollflusskanten geben die Reihenfolge der Ausführung zwischen Prozessschritten an. Eine Kontrollflusskante von

einem Knoten A zu einem anderen Knoten B bedeutet, dass der von B repräsentierte Prozessschritt erst ausgeführt werden darf, wenn A beendet ist, oder feststeht, dass A gar nicht mehr zur Ausführung kommt (s. Abschnitt 2.2.3). Besitzt ein Knoten mehrere Nachfolger (Vorgänger), ist er ein Verzweigungsknoten (Vereinigungsknoten). Um die Blockstruktur gewährleisten zu können, besitzt jeder Verzweigungsknoten (Vereinigungsknoten) einen eindeutigen Vereinigungsknoten (Verzweigungsknoten). Beide spannen einen *Kontrollblock* auf. Jede von einem Verzweigungsknoten ausgehende Kante definiert einen *Zweig* dieses Kontrollblocks. Alle Knoten auf einem solchen Zweig bilden selbst wieder ein Kontrollblock. Dieser kann auch nur aus einer Knotensequenz bestehen. Für die Einhaltung der Blockstruktur darf jeder Kontrollblock nur eine eingehende und eine ausgehende Kontrollkante besitzen. Die eingehende Kontrollkante geht vom entsprechenden Verzweigungsknoten aus, die ausgehende Kontrollkante mündet in den entsprechenden Vereinigungsknoten.

Beispiel 2.1 (Kontrollfluss und Parallelität)
Abb. 2.5 zeigt beispielhaft Kontrollfluss und -blöcke. Knoten D und H etwa spannen einen Kontrollblock auf. Die von Knoten D ausgehenden Zweige sind jeweils Knotensequenzen mit Knoten E sowie den Knoten F und G. Der von D und H gebildete Kontrollblock wiederum stellt einen Zweig des durch die Knoten C und Knoten K gebildeten Kontrollblocks. Knoten C und K spannen einen parallelen Kontrollblock auf, d. h. die Knoten auf den entsprechenden Zweigen werden gleichzeitig ausgeführt. Dabei muss die Reihenfolge innerhalb des Kontrollblocks eines Zweigs berücksichtigt werden. So können die Knoten D bis H parallel zu den nebenläufigen Knoten I bis J ausgeführt werden Dabei wird z. B. Knoten I immer vor Knoten J ausgeführt. Knoten K wiederum kann erst ausgeführt werden, wenn alle seine direkten Vorgänger (Knoten H und J) beendet sind.

Mit einer strikten Blockstruktur können parallele Zweige nicht synchronisiert werden, d. h. es ist nicht möglich, dass ein Prozessschritt eines parallelen Zweigs erst nach Beendigung eines Prozessschritts eines anderen Zweigs derselben parallelen Verzweigung ausgeführt wird. Es sind lediglich Abhängigkeiten zu direkten Vorgängern und Nachfolgern möglich. Aus diesem Grund gibt es im ADEPT-Metamodell (gerichtete) Synchronisationskanten [RD97, RD98]. Der Zielknoten einer solchen Kante darf erst dann ausgeführt werden, wenn der Quellknoten der Kante beendet ist oder feststeht, dass er nicht mehr ausgeführt werden kann, etwa weil er auf einem abgewählten Zweig einer alternativen Verzweigung liegt. In Abb. 2.5 gilt dies für Knoten J, der über eine Synchronisationskante mit Knoten F verbunden ist. Da für die Synchronisation ein spezieller Kantentyp verwendet wird, wird die Blockstruktur zumindest für normale Kontrollkanten nicht verletzt. Außerdem gelten für Synchronisationskanten zusätzliche Einschränkungen bezüglich ihrer Verwendung. Sie dürfen nur zwischen verschiedenen Zweigen einer parallelen Verzweigung existieren. Die Zweige dürfen beliebig verschachtelt sein und weitere innere Verzweigungen besitzen.

Beispiel 2.2 (Alternative Verzweigungen)
Knoten D und H spannen einen alternativen Kontrollblock auf. Dabei wird genau ein Zweig der alternativen Verzweigung ausgeführt. In Abb. 2.5 etwa sind dies entweder Knoten E oder die von Knoten F und G gebildete Sequenz. Knoten H wiederum kann erst ausgeführt werden, wenn der ausgewählte Zweig beendet ist. Die Entscheidung, welcher Zweig ausgeführt wird, wird von Knoten D getroffen.

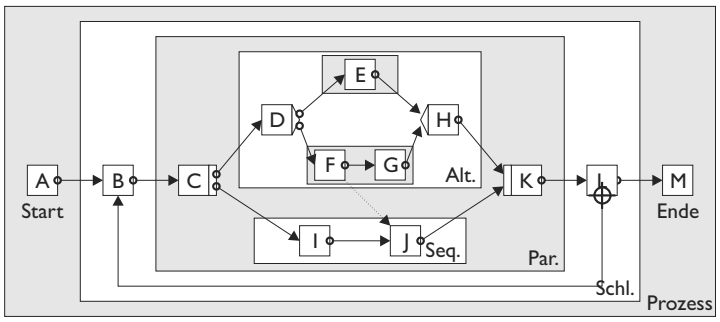


Abbildung 2.5: Kontrollfluss mit Blockstruktur in ADEPT (in Anlehnung an [Rei00])

Neben Alternativen und Parallelität erlaubt der Kontrollfluss von ADEPT Schleifen. So spannen Knoten B und L in Abb. 2.5 einen entsprechenden Kontrollblock auf. Die Schleifenrücksprungkante zwischen B und L ist keine normale Kontrollkante, wodurch die Schleife explizit wird. Ihre Semantik ist, dass die beiden Knoten der Schleifenrücksprungkante sowie die Knoten dazwischen (der *Schleifenrumpf*) mehrmals ausgeführt werden können. Ob dies passiert, wird – ähnlich wie bei alternativen Verzweigungen – von dem Knoten bestimmt, von dem die Schleifenrücksprungkante ausgeht. Damit wird die Schleifenbedingung erst am Ende der Schleife ausgewertet und die Schleife mindestens einmal durchlaufen. Bezüglich des normalen Kontrollflusses wird Knoten B immer vor Knoten L ausgeführt.

Damit der gesamte Prozess selbst einen Kontrollblock bildet, gibt es die speziellen Start- und Endknoten. Diese repräsentieren keine Prozessschritte sondern bilden nur den Rahmen für den Prozess. Dementsprechend könnte man auf beide Knoten verzichten. Allerdings sind sie sowohl aus konzeptioneller als auch technischer Sicht nützlich. Da sie immer existieren, bilden sie auch den Ausgangspunkt jeder ADEPT-Prozessvorlage: die *Nullvorlage*. Diese besteht aus je einem Start- und Endknoten, die durch eine Kontrollkante verbunden sind, und steht für einen leeren Prozess.

Knoten- und Kantenattribute Knoten repräsentieren Prozessschritte und besitzen verschiedene Attribute. Dazu gehört eine bezüglich der Prozessvorlage eindeutige Kennung (ID). In Abb. 2.5 und Abb. 2.6 sind dies Buchstaben. Daneben besitzen Knoten einen Namen und eine Beschreibung. Beide müssen nicht eindeutig sein, ein eindeutiger Name ist aber für die Modellierung und Ausführung nützlich. Außerdem ist an den Knoten hinterlegt, wer den repräsentierten Prozessschritt ausführen soll. Dies können beispielsweise konkrete Personen („Mitarbeiter Schmidt“), Rollen/Stellen („Sekretariat“) oder Organisationseinheiten („Buchhaltung“) sein. Diese *Bearbeiterzuordnungen* hängen davon ab, mit was das entsprechende PrMS umgehen kann. Normalerweise gibt es für diese Zwecke ein separates Organisationsmetamodell, das die Organisationsstrukturen abbildet [RMR09, RMR08, RR05, RR07]. Dieses kann auch durch ein separates System wie LDAP (*Lightweight Directory Access Protocol* [YHK95]) bereitgestellt werden. Es muss lediglich gewährleistet sein, dass das Organisationsmodell bei der Ausführung eines Prozessmodells zur Verfügung steht und es die verwendeten Bearbeiterzuordnungen auf konkrete Bearbeiter auflösen kann.

Kontrollflusskanten repräsentieren Beziehungen/Relationen zwischen Prozessschritten und haben nur wenige Attribute. Hierzu gehört die Richtung, da die Relationen asymmetrisch

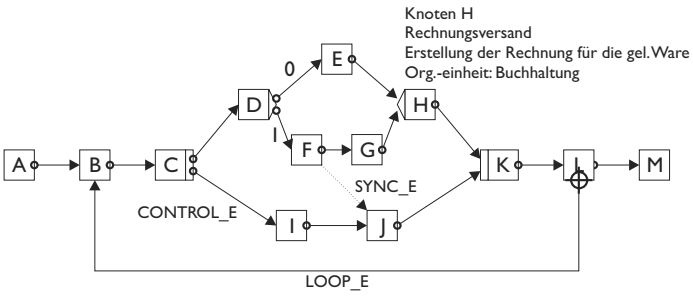


Abbildung 2.6: Knoten- und Kantenattribute

sind. Außerdem besitzen sie im ADEPT-Metamodell einen Typ. Der explizite Kantentyp vereinfacht Analysen und Überprüfungen, da die transitive Hülle je nach Analyse auf bestimmte Kantentypen eingeschränkt werden kann. Beispielsweise können die Prozessgraphen so als azyklisch betrachtet werden, wenn Schleifenrücksprungkanten ignoriert werden können.

Kanten, die von einem Verzweigungsknoten einer alternativen Verzweigung ausgehen (Kante D–E und D–F in Abb. 2.6), besitzen ein spezielles Attribut zu ihrer Identifikation. Dieses ist eindeutig bezüglich aller von einem Verzweigungsknoten ausgehenden Kanten. Über diese Kennung wird der entsprechende Pfad ausgewählt, der sich zur Laufzeit bei der Entscheidung am Verzweigungsknoten ergibt. Für diese Zwecke kann auch die Kennung des jeweiligen Zielknotens der Kante verwendet werden. Dies hat jedoch den Nachteil, dass sich bei strukturellen Prozessänderungen die Kennung ändern kann. Dies tritt beispielsweise auf, wenn zwischen dem Verzweigungsknoten und einem seiner Nachfolger ein weiterer Knoten eingefügt wird oder ein Nachfolger gelöscht wird.

Die Verwendung eines expliziten Kantentyps eröffnet noch weitergehende Möglichkeiten. So lassen sich durch die Einführung neuer Kantentypen weitere Beziehungen innerhalb der Prozessmodelle und damit beliebige Relationen zwischen Prozessschritten berücksichtigen. Kantenattribute erlauben entsprechend die Abbildung von Attributen von Relationen. Ein Beispiel hierfür sind zeitliche Beziehungen wie Mindest- und Maximalabstände zwischen der Ausführung zweier Prozessschritte [Gri97, Lan08, LWR14].

Metainformationen Aufgrund der Zyklenfreiheit und der Blockstruktur lassen sich zahlreiche Metainformationen ableiten, die für eine leistungsfähige technische Realisierung nützlich sind. Hierzu gehört eine topologische Sortierung der Knoten. In Abb. 2.7 sind die Knoten mit einer Kennung versehen, die die Kriterien einer topologischen Sortierung erfüllt. Bei Knoten in parallelen und alternativen Verzweigungen gibt es keine eindeutige topologische Sortierung. So ist ein Tausch von Knoten 7 und 8 in Abb. 2.7 ebenfalls eine gültige topologische Sortierung.

Sinnvollerweise orientiert sich die Sortierung aber an den Blockstrukturen und Zweigen, so dass alle Knoten eines Zweigs entweder vor oder nach einem parallelen (oder alternativen) Zweig sortiert werden. Dies führt dazu, dass der erste und letzte Knoten eines Zweigs ein diskretes Intervall bilden, in dem sämtliche Knoten des Zweigs liegen. Weiterhin sind diese Intervalle vollständig und damit minimal, d. h. es gibt für jede Zahl des Intervalls einen Knoten im entsprechenden Kontrollblock. Gibt es Synchronisationskanten zwischen den Zweigen einer parallelen Verzweigung, so wird jeweils die erste Synchronisationskante bei der topologischen Sortierung berücksichtigt. Da es zwischen zwei Zweigen mehrere Synchronisationskanten in

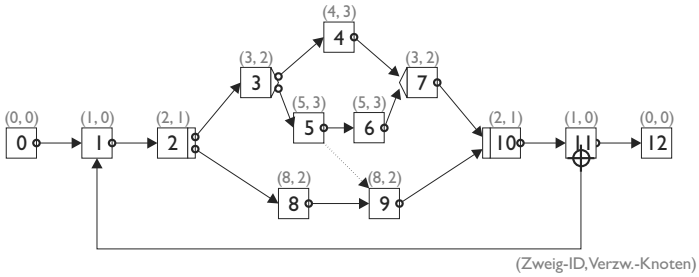


Abbildung 2.7: Knotenmetainformationen

beide Richtungen geben kann, sofern dadurch kein Zyklus entsteht, können bei der topologischen Sortierung nicht alle berücksichtigt werden.

Weitere Informationen für Knoten sind jeweils die Kennungen des Zweigs und des umgebenden Verzweigungsknotens. Die Zweigkennung identifiziert den innersten Zweig, auf dem ein Knoten liegt. Die Kennung selbst ist identisch mit der topologischen Kennung des ersten Knotens des Zweigs. Die Start- und Endknoten selbst besitzen nur der Vollständigkeit halber die beiden Kennungen. Für sie sind die Kennungen undefiniert, aber auch irrelevant. Die topologische Sortierung, zusammen mit diesen Kennungen, ermöglicht auf einfachem Weg Aussagen zur Graphstruktur, etwa ob zwei beliebige Knoten in einer (transitiven) Vorgänger-/Nachfolgerbeziehung stehen, ohne dass dazu der komplette Graph iteriert werden muss.

Beispiel 2.3 (Knotenmetainformationen)
In Abb. 2.7 liegen Knoten 8 und 9 beide auf demselben Zweig, dessen erster Knoten 8 ist. Der Zweig von Knoten 3 und 7 beginnt bei 3. Knoten 4 dagegen liegt auf einem inneren Zweig und deshalb nicht auf Zweig 3, sondern auf dem Zweig 4. Dementsprechend ist der Verzweigungsknoten der Vorgänger des ersten Knotens eines Zweigs. Für Knoten 8 und 9 ist dies Knoten 2, ebenso für Knoten 3 und 7. Für die inneren Zweige zwischen 3 und 7 ist dies Knoten 3. Da auch Start- und Endknoten einen Kontrollblock mit einem Zweig aufspannen, liegen die Knoten 1 und 11 auf dem Zweig 1 und referenzieren den Startknoten (Knoten 0) als Verzweigungsknoten.

2.2.2.2 Datenfluss

Die Reihenfolge der Ausführung von Prozessschritten wird maßgeblich dadurch bestimmt, welche Daten die einzelnen Prozessschritte bei ihrer Ausführung produzieren und benötigen. So wird sich ein Prozessschritt B bezüglich des Kontrollflusses nach Prozessschritt A befinden, wenn Prozessschritt B Daten benötigt, die von Prozessschritt A erzeugt werden. Die zwischen Prozessschritten ausgetauschten Daten werden im ADEPT-Metamodell über Datenelemente modelliert [Rei00]. Ein Datenelement besitzt einen bezüglich des entsprechenden Prozesses eindeutigen Namen und einen Datentyp. Produziert ein Prozessschritt bei seiner Ausführung Daten, wird dieser mit einer gerichteten Kante mit dem entsprechenden Datenelement verbunden. Die Richtung einer solchen *Schreibkante* geht vom Knoten zum Datenelement. Umgekehrt gehen *Lesekanten* von einem Datenelement zu einem Knoten. Sie geben an, dass der entsprechende Prozessschritt die Daten aus dem Datenelement für seine Ausführung benötigt. Es kann maximal eine Lese- und eine Schreibkante zwischen einem Knoten und einem Datenelement geben.

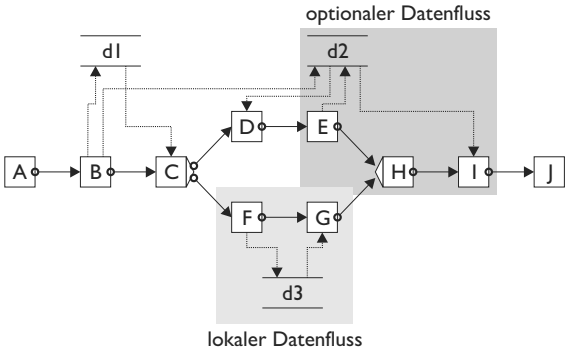


Abbildung 2.8: Datenfluss in ADEPT

Die Datenelemente im ADEPT-Metamodell sind global bezüglich einer Prozessvorlage. Das heißt, alle Knoten der Prozessvorlage können unter gewissen Einschränkungen beliebig lesend oder schreibend auf die Datenelemente zugreifen. Knoten von Subprozessen können nur auf die Datenelemente der eigenen Prozessvorlage zugreifen und nicht auf die des Elternprozesses. Das ADEPT-Metamodell lässt es zu, die Sichtbarkeit von Datenelementen von der Blockstruktur des Kontrollflusses (s. Abschnitt 2.2.2.1) abhängig zu machen. Dies ist vergleichbar mit strukturierten Programmiersprachen, bei denen Variablen nur innerhalb des Blocks sichtbar sind, in dem sie deklariert wurden (s. Abb. 2.8, Knoten F und G, Datenelement d3). Allgemeine Betrachtungen zum Thema Datenfluss findet man in [RtHEvdA05].

Bei lesenden und schreibenden Zugriffen auf Datenelemente unterschiedet man zwischen *obligaten* und *optionalen* Datenzugriffen. Schreibt ein Knoten ein Datenelement obligat, besitzt das Datenelement nach Ausführung des entsprechenden Prozessschritts auf jeden Fall einen Wert. Beim optionalen Schreiben dagegen kann das Datenelement auch keinen Wert (sog. *NULL-Wert*) haben. Ein obligater Lesezugriff bedeutet, dass der entsprechende Prozessschritt bei seiner Ausführung auf jeden Fall einen gültigen Wert aus dem Datenelement liest; dies darf nicht der NULL-Wert sein. Entsprechend kann ein optional lesender Knoten auch mit dem NULL-Wert umgehen.

Korrektheit des Datenflusses Durch die Unterscheidung zwischen obligaten und optionalen Datenzugriffen, lässt sich ein naheliegendes Korrektheitskriterium des ADEPT-Metamodells ableiten: Ein obligater Lesezugriff darf im Verlauf einer Prozessausführung nur erfolgen, wenn davor (bezüglich des Kontrollflusses) stets ein obligater Schreibzugriff erfolgt ist. Andernfalls kann nicht gewährleistet werden, dass der entsprechende Prozessschritt bei seiner Ausführung alle benötigten Daten zur Verfügung hat und somit problemlos ausgeführt werden kann. Dabei müssen auch alternative Verzweigungen berücksichtigt werden: Wird ein Datenelement nur in einem Zweig einer alternativen Verzweigung (obligat) geschrieben, gilt dies bezüglich aller Knoten nach dem alternativen Vereinigungsschritt nur als optionales Schreiben (s. Abb. 2.8, Knoten E und I, Datenelement d2). Dieses Schreiben ist nur dann obligat für die Knoten nach dem Vereinigungsknoten, wenn alle Zweige das Datenelement obligat schreiben. Diese Kriterien sind vergleichbar mit Übersetzern höherer Programmiersprachen, z. B. Java [jav] und .NET [net]. So müssen Variablen initialisiert oder geschrieben worden sein, bevor das erste Mal lesend auf sie zugegriffen wird. Die Einschränkungen gelten analog bei bedingten Verzweigungen.

Eine weitere Einschränkung des Datenflusses beim ADEPT-Metamodell gibt es im Zusammenhang mit Parallelität. So ist es nicht erlaubt, dass ein Datenelement aus zwei verschiedenen Zweigen einer parallelen Verzweigung geschrieben wird [Rei00]. In solch einem Fall ist die Ausführungsreihenfolge der schreibenden Knoten nicht festgelegt und das Datenelement würde zur Laufzeit den Wert haben, den der zuletzt ausgeführte Knoten geschrieben hat. Damit wäre der Wert nicht-deterministisch. Ein weiteres damit einhergehendes Problem ist vergleichbar mit verlorengegangenen Änderungen (*Lost Updates*) bei DBMS [HR01]. Dabei finden mehrere Schreibzugriffe direkt hintereinander statt, d. h. ohne zwischenzeitliches Lesen des Datenelements. Dadurch sind alle bis auf den letzten Schreibzugriff unnötig. Ist die Reihenfolge zwischen den schreibenden Knoten vorgegeben, etwa durch Synchronisationskanten, gilt das Verbot der Schreibzugriffe aus parallelen Zweigen nicht.

Datenelementattribute Wie erwähnt besitzen Datenelemente einen Datentyp. Für den Rahmen dieser Arbeit beschränken wir uns auf die folgenden (elementaren) Datentypen: Zeichenkette (`STRING`), Ganzzahl (`INTEGER`), Gleitkommazahl (`FLOAT`), Boolescher Wert (`BOOLEAN`), Zeit (`DATE`), Ressourcenbezeichner (`URI` (*Uniform Resource Identifier*)) und benutzerdefinierte Datentypen (`UDT` (*User-Defined Data Type*)). Erweiterungen der Datentypen hinsichtlich komplexer Datentypen und Listen werden in [For09] ausführlich diskutiert. Sie sind für diese Arbeit aber von untergeordneter Bedeutung.

Ein weiteres Attribut kennzeichnet Datenelemente als öffentlich oder nicht. Bei nicht-öffentlichen Datenelementen ist gewährleistet, dass deren Werte nur über Lesekanten von den entsprechenden Prozessschritten gelesen werden können. Von anderen Stellen kann auf die Werte nicht zugegriffen werden. Die Werte öffentlicher Datenelemente dagegen können jederzeit gelesen werden.

Beispiel 2.4 (Öffentliches Datenelement)
In einem Bestellprozess wird der Name des entsprechenden Kunden in einem öffentlichen Datenelement gespeichert. Dies erlaubt es, den Kundennamen bei den Arbeitslisteneinträgen dieses Prozesses anzuzeigen. Dadurch können die Bearbeiter dieser Prozessschritte leichter zwischen den Prozessschritten verschiedener Bestellprozesse und damit verschiedener Kunden unterscheiden.

Parameter Neben den Datenelementen und -kanten gehören *Parameter* zum Datenfluss im ADEPT-Metamodell. Diese sind logisch einem Knoten zugeordnet und entsprechen den aus der strukturierten Programmierung bekannten Ein- und Ausgabeparametern von Prozeduren und Funktionen. Ebenso wie bei Prozeduren bilden Parameter im ADEPT-Metamodell die Schnittstelle zwischen der inneren Logik eines Knotens (*Aktivität*) und seiner Umgebung (s. Abb. 2.9). Dadurch erhöhen sie die Unabhängigkeit der inneren Logik (der Prozeduren) und somit auch deren Wiederverwendbarkeit. Entsprechend sind Parameter im ADEPT-Metamodell ebenso wie Parameter von Prozeduren nur innerhalb der Aktivität (der Prozedur) sichtbar.

Im ADEPT-Metamodell werden Parameter mit Datenkanten verbunden und zwar Lesekanten mit Eingabeparametern und Schreibkanten mit Ausgabeparametern. Dabei muss es für jede Datenkante einen entsprechenden Parameter geben. Allerdings müssen nicht alle Parameter mit einer Datenkante verbunden werden. So ist es beispielsweise möglich, Ausgabeparameter eines Knotens zu ignorieren, die in einem bestimmten Prozess nicht benötigt werden. Im Gegensatz zu Prozeduren gibt es im ADEPT-Metamodell nur Wertparameter (*call-by-value*). Dies ist

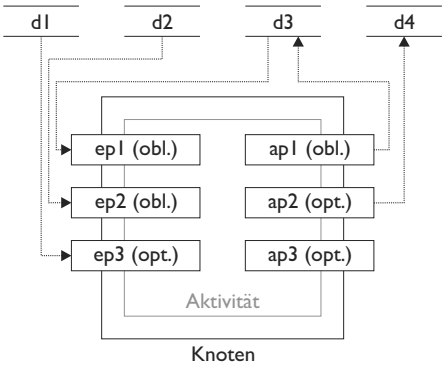


Abbildung 2.9: Knotenparameter

jedoch keine Einschränkung, da die Werte vom Prozessschritt als Referenzen interpretiert werden können. Insbesondere der Datentyp `URI` eignet sich somit für die Realisierung von Referenzparametern (*call-by-reference*).

2.2.2.3 Weitere Aspekte

Während die bisher vorgestellten Elemente des ADEPT-Metamodells eher struktureller Natur sind, gibt es noch semantische Elemente für die Prozessausführung. In erster Linie sind dies *Aktivitäten*, also Elemente, die die an einem Prozessschritt realisierte Logik/Funktion repräsentieren [TRI09]. Eng verwandt mit Aktivitäten sind Subprozesse, also Prozesse, die nicht eigenständig sondern im Kontext eines anderen Prozesses, dem Elternprozess, ausgeführt werden. Wir gehen in diesem Abschnitt auf diese weiteren Aspekte ein.

Aktivität Aktivitäten werden an Knoten hinterlegt und ergänzen diese um Informationen, die notwendig sind, um tatsächlich etwas ausführen und so Daten verarbeiten zu können. Dies können beispielsweise Aufrufe von Prozeduren, Webdiensten oder Anwendungsprogrammen (z. B. Textverarbeitung) sein (s. Knoten D in Vorlage A, V2.2. in Abb. 2.4). Die von diesen *Anwendungsbausteinen* verarbeiteten Daten definieren die Ein-/Ausgabeparameter der Aktivität und damit auch des Knotens (s. Abschnitt 2.2.2.2). Neben den Ein-/Ausgabeparametern besitzen Aktivitäten eine eindeutige Kennung, einen Namen und einen Typ. Aktivitäten sind unabhängig von einem bestimmten Prozess, d. h., sie können in beliebigen Prozessen verwendet werden, sofern der dazugehörige Knoten entsprechend an den Datenfluss angebunden ist.

Neben normalen Aktivitäten gibt es im ADEPT-Metamodell auch *Entscheidungsaktivitäten*. Dies sind besondere Aktivitäten, die für die Verzweigungsknoten alternativer Verzweigungen und Schleifenendknoten verwendet werden. Sie besitzen gegenüber normalen Aktivitäten einen speziellen ganzzahligen obligaten Ausgabeparameter. Dessen Wert bestimmt zur Laufzeit, welcher Pfad der nachfolgenden alternativen Verzweigung gewählt wird bzw. ob die Schleife nochmals durchlaufen wird [Rei00]. Ein weiterer Aktivitätstyp sind Subprozesse, die wir im folgenden Abschnitt diskutieren.

Subprozesse Die Ausführung eines Prozesses ist auf abstrakter Ebene mit der Ausführung eines Programms oder einer Prozedur vergleichbar. Damit liegt es nahe, auch ganze Prozesse durch Aktivitäten zu kapseln und diese dann analog zu Programm- oder Prozeduraufrufen in

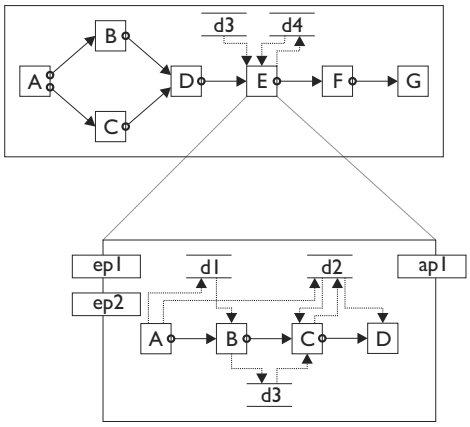


Abbildung 2.10: Datenübergabe an Subprozess

anderen Prozessen zu verwenden. Solche *Subprozesse* (auch *Kindprozesse* genannt) unterscheiden sich nicht von normalen Prozessen; auch sie beruhen auf denselben Prozessvorlagen. Um eine Wiederverwendbarkeit zu ermöglichen, sollten sie jedoch nicht fest an den entsprechenden Elternprozess gekoppelt werden. Dies wird wie bei Aktivitäten über Ein-/Ausgabeparameter erreicht. Im ADEPT-Metamodell werden Eingabeparameter eines Prozesses dargestellt, indem der Startknoten des Prozesses bestimmte Datenelemente obligat oder optional schreibt. Die Datentypen der Parameter entsprechen den Datentypen der entsprechenden Datenelemente. Umgekehrt manifestieren sich obligate oder optionale Ausgabeparameter eines Prozesses als obligate oder optionale Lesekanten seines Endknotens (s. Abb. 2.10). So bleibt die Korrektheit des Datenflusses eines Prozesses mit Parametern gewährleistet, und es können die bekannten Algorithmen zur Überprüfung des Datenflusses angewendet werden [Rei00]. Besitzt ein Subprozess beispielsweise einen obligaten Ausgabeparameter, wird das entsprechende Datenelement obligat vom Endknoten gelesen. Da obligate Lesezugriffe nicht erlaubt sind, wenn davor kein obligater Schreibzugriff stattgefunden hat, ist die Versorgung des Ausgabeparameters immer gewährleistet.

2.2.3 Ausführungsmodell

Neben den Prozessstrukturen und damit den statischen Aspekten gehören zum ADEPT-Metamodell auch die dynamischen Aspekte, d. h. die Prozessausführung. Wir erläutern in diesem Abschnitt zunächst die Ausführung von Prozessgraphen. Der Aufruf von Aktivitäten ist Gegenstand von Abschnitt 2.4.2. Auf Ebene von Prozessgraphen bildet das Zustandsmodell von Prozessschritten (s. Abschnitt 2.2.3.1) die Grundlage der dynamischen Aspekte. Dieses spezifiziert die Zustände von Prozessschritten und die entsprechenden Zustandsübergänge. Auf diesem Zustandsmodell sind Ausführungsregeln definiert, die unter anderem das Weiterschalten regeln (s. Abschnitt 2.2.3.2). Über das Weiterschalten sorgt das PrMS wiederum dafür, dass die einzelnen Prozessschritte in der durch die Prozessvorlage vorgegebenen Reihenfolge abgearbeitet werden. Da für die Ausführung von Prozessschritten üblicherweise auch Daten aus dem Prozess benötigt werden, ist auch der Datenzugriff im Ausführungsmodell geregelt (s. Abschnitt 2.2.3.3). Daneben umfasst das Ausführungsmodell weitere Aspekte wie *abhängige Bearbeiterzuordnungen* (s. Abschnitt 2.2.3.5) und *variable Parallelität* (s. Abschnitt 2.2.3.5).

2.2.3.1 Zustandsmodell von Prozessschritten

Das zentrale Element im ADEPT-Ausführungsmodell ist die *Prozessinstanz*. Eine solche Prozessinstanz repräsentiert die Ausführung eines bestimmten Prozesses und wird durch ihren Zustand charakterisiert. Der Zustand einer Prozessinstanz wiederum ist über die Zustände ihrer Prozessschritte definiert. Abb. 2.11 zeigt das Zustandsmodell für Prozessschritte. Die Zustände und Zustandsübergänge des Zustandsmodells besitzen eine klar definierte Semantik, die eine Auswertung und damit maschinelle Prozessausführung überhaupt erst ermöglicht. Die Ausführung beruht auf *Ausführungsregeln*, mit dem das PrMS Neubewertungen der Zustände vornimmt, etwa das Weiterschalten direkt nachdem ein Prozessschritt beendet wurde.

Initial befindet sich ein Prozessschritt im Zustand `NOT_ACTIVATED` (s. Abb. 2.11). Diesen Zustand kann er nur im Rahmen des Weiterschaltens verlassen, d. h. sobald alle seine Vorgänger im Prozessgraphen abgeschlossen sind oder nicht mehr ausgeführt werden können. Ein aktivierter (`ACTIVATED`) Prozessschritt steht autorisierten Bearbeitern [RMR09, RMR08] in deren Arbeitsliste zur Ausführung zur Verfügung. Durch Auswählen des entsprechenden Arbeitslisteneintrags kann ein Bearbeiter den Prozessschritt für sich reservieren (`SELECTED`). Dies ist nützlich, falls es mehrere mögliche Bearbeiter gibt. Durch die Reservierung wird der Eintrag aus allen Arbeitslisten entfernt, die den Prozessschritt nicht ausgewählt haben.

Beim Starten eines Prozessschritts werden die hinterlegte Aktivität und damit der ihr zugeordnete Anwendungsbaustein gestartet und ausgeführt (`RUNNING`). Sofern die Aktivität dies erlaubt, kann sie während ihre Ausführung unterbrochen werden, was sich im Zustand des Prozessschritts widerspiegelt (`SUSPENDED`). Ist die Ausführung beendet, wechselt der Zustand in `COMPLETED`, falls die Ausführung erfolgreich war, oder in `FAILED`, falls sie gescheitert ist. Sowohl im Fehlerfall als auch während seiner Ausführung kann ein Prozessschritt zurückgesetzt werden (`reset`). Dies versetzt ihn in den Zustand vor seiner Ausführung zurück und verwirft die bereits geschriebenen Daten. Der Zustand `SKIPPED` kennzeichnet schließlich Prozessschritte, die sich in abgewählten Zweigen einer alternativen Verzweigung befinden und deshalb nicht zur Ausführung kommen.

Eine Erweiterung des Zustandsmodells gegenüber [Rei00] bildet die Möglichkeit für Rückfragen und dem entsprechenden Zustand `ENQUIRED`. Benötigt ein Bearbeiter während der Ausführung eines Prozessschritts die Hilfe eines anderen Bearbeiters, unterbricht er zunächst die Ausführung dieses Prozessschritts. Im unterbrochenen Zustand kann er dann den Prozessschritt mit einer Frage versehen und an einen anderen Bearbeiter weiterleiten. Dieser kann die Ausführung fortsetzen, um eine Antwort auf die Frage zu finden. Anschließend kann er den Prozessschritt unterbrechen und mitsamt der Antwort an den Fragesteller zurückleiten, der die Ausführung fortsetzt und beendet. War die Antwort nicht ausreichend, kann eine neue Rückfrage an denselben oder einen anderen Bearbeiter gestellt werden. Rückfragen können zudem geschachtelt sein, d. h. der Gefragte kann selbst eine Rückfrage stellen und den Prozessschritt weiterleiten. Im Zustandsmodell aus Abb. 2.11 gibt es hierfür den Zähler `currentEnquiries`. Statt einer Antwort kann ein Bearbeiter einen Prozessschritt mit Rückfrage auch direkt beenden. Da der Fragesteller aber noch auf eine Antwort wartet, wechselt der Prozessschritt nicht in den Zustand `COMPLETED` sondern in den Zustand `ENQUIRED`. Dort verbleibt er in der Arbeitsliste des angefragten Bearbeiters, bis dieser eine Antwort gibt. Nach Beantwortung erhält der Fragesteller den Prozessschritt nochmals, um die Antwort abzurufen. Erst danach wechselt der Prozessschritt in den Zustand `COMPLETED`.

Für die Zustände des Zustandsmodells gibt es graphische Repräsentationen, mit denen die Graphstruktur angereichert werden kann (s. Abb. 2.4 rechts). Mit diesen Graphmarkierungen

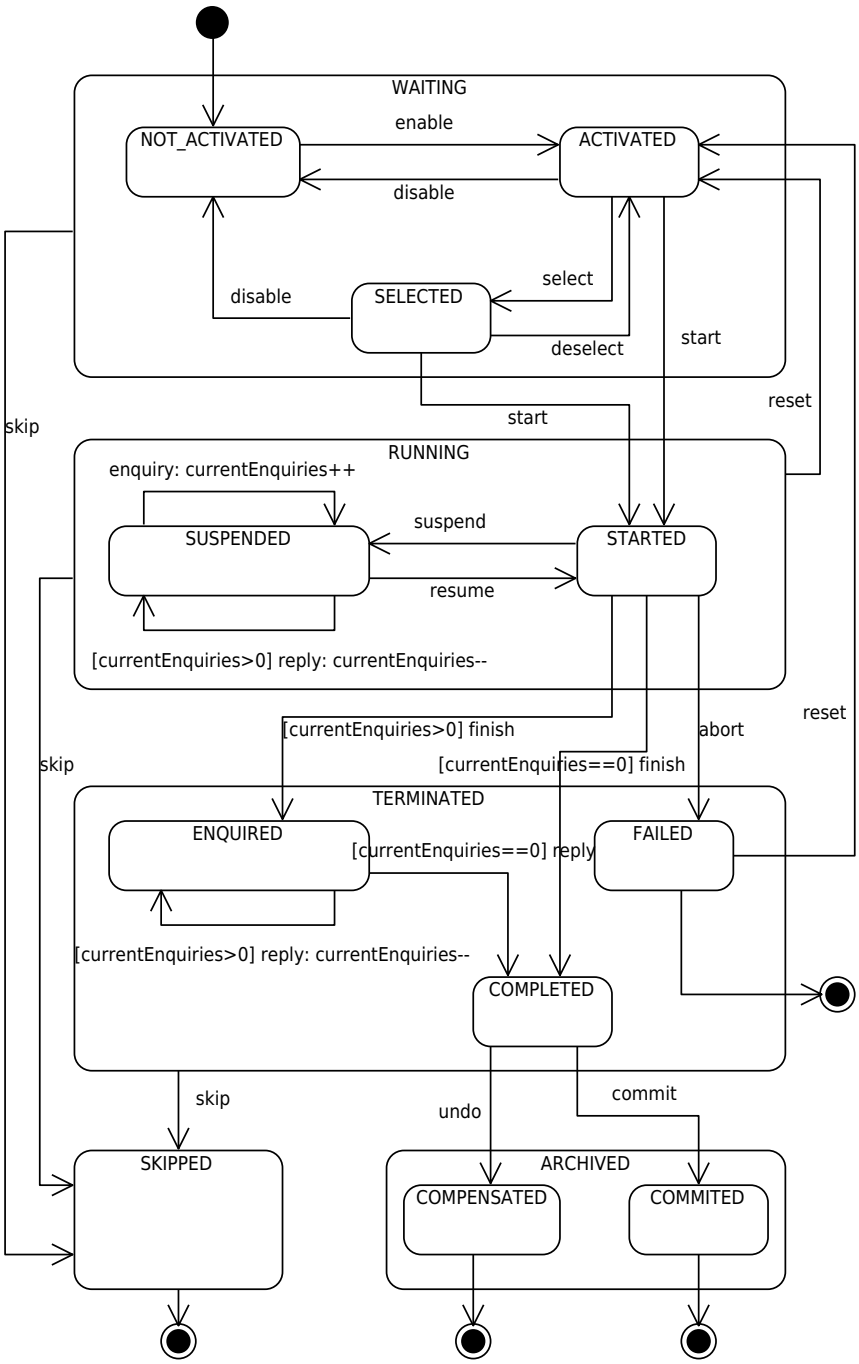


Abbildung 2.11: Zustandsmodell von Prozessschritten als UML-Zustandsdiagramm (in Anlehnung an [Rei00])

lässt sich der Zustand einer Prozessinstanz zusammen mit dem gesamten Prozessablauf jederzeit benutzerfreundlich darstellen. Dadurch lässt sich etwa der Ausführungsfortschritt kontrollieren [RD97, RD98]. Die Graphmarkierungen stellen eine konsolidierte Sicht auf die Ausführungshistorie dar.

Da Subprozesse weitgehend wie Aktivitäten behandelt werden, gilt das Zustandsmodell auch für die Knoten eines Prozessgraphen, die Subprozesse repräsentieren. Der Zustand eines solchen Knotens spiegelt den Ausführungszustand der entsprechenden Subprozessinstanz wider, was wiederum eine Aggregation der Zustände von deren Prozessschritte ist. Da der Zustand des Knotens der Subprozessinstanz nur indirekt über ihre Prozessschritte beeinflusst werden kann, sind nicht alle Zustandsübergänge für ihn relevant. So sind beispielsweise sind Rückfragen an diesem Knoten nicht möglich. Dies ist aber auch gar nicht sinnvoll, da Subprozesse üblicherweise nicht nur einen Bearbeiter haben.

2.2.3.2 Weiterschalten

Das Zustandsmodell wird durch sog. *Ausführungsregeln* erweitert. Diese Regeln spezifizieren, wann welcher Zustandsübergang bei Prozessschritten stattfindet [Rei00]. Auslöser für die Anwendung der Ausführungsregeln von Prozessschritten ist eine (semantische) Zustandsänderung. Häufig ist dies eine Benutzerinteraktion wie das Starten einer neuen Prozessinstanz oder das Starten oder Beenden eines Prozessschritts (über einen Funktionsaufruf). Es kann aber auch das Eintreten eines Ereignisses sein, etwa der Ablauf einer Zeitspanne nach der eine Eskalation stattfindet. Abhängig vom Auslöser wendet das PrMS die vorgegebenen Ausführungsregeln an, die zu entsprechenden Zustandsänderungen führen. Diese wiederum verursachen weitere Aktionen, etwa die Erzeugung eines Arbeitslisteneintrags für einen zur Bearbeitung anstehenden Prozessschritt.

Die wichtigste und häufigste Anwendung von Ausführungsregeln ist das Weiterschalten. Dabei werden die Zustände der Prozessschritte einer Prozessinstanz neu bewertet, nachdem sich der Zustand eines einzelnen Prozessschritts geändert hat oder eine Prozessinstanz gestartet wurde. Abb. 2.12 zeigt das Weiterschalten anhand eines einfachen Beispiels:

Beispiel 2.5 (Weiterschalten einer Prozessinstanz)
Der zur Ausführung anstehende Knoten C der Prozessinstanz I5 (unterer Zweig der parallelen Verzweigung) wird von einem Bearbeiter gestartet. Dies führt zur Zustandsänderung des entsprechenden Knotens (`RUNNING`). Auch das anschließende Beenden wirkt sich erst einmal nur auf Knoten C aus (`COMPLETED`). Hier findet zwar ein Weiterschalten statt, die Ausführungsregeln verhindern aber, dass der Vereinigungsknoten D ausgeführt werden kann (Zustand `ACTIVATED`), da noch nicht alle seine direkten Vorgängerknoten abgeschlossen sind. Dies ist erst der Fall, wenn Knoten B beendet wird (s. Abb. 2.12 links unten).

Die Ausführungsregeln berücksichtigen alle Elemente des Kontrollflusses, also neben Knotensequenzen auch parallele Verzweigungen, Synchronisationskanten zwischen parallelen Zweigen, alternative Verzweigungen und Schleifen. Diese Elemente besitzen jeweils eine unterschiedliche Ausführungssemantik. So werden beispielsweise bei einer alternativen Verzweigung ein Knoten aktiviert und die Knoten abgewählter Zweige übersprungen (`SKIPPED`) gesetzt (sog. *Deadpath-Elimination*, [Rei00]), während bei parallelen Verzweigungen alle Nachfolger des Verzweigungsknotens aktiviert werden.

Prozessschritte innerhalb von Schleifen werden abhängig von der Schleifenbedingung ggf. mehrmals durchlaufen. Beim Weiterschalten werden dazu bei positivem Ergebnis der Schlei-

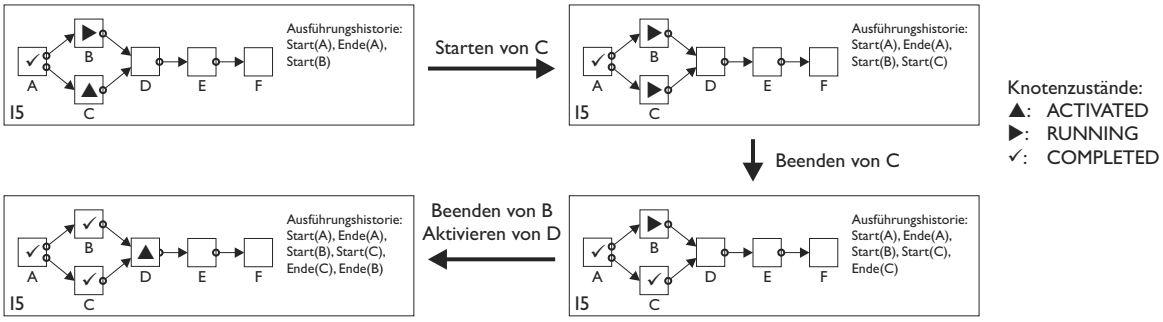


Abbildung 2.12: Weiterschalten einer Prozessinstanz

fenbedingung die Zustände aller Knoten innerhalb der Schleife sowie des Schleifenendknotens zurückgesetzt (`NOT_ACTIVATED`) und der Schleifenstartknoten aktiviert (`ACTIVATED`). Damit können Prozessschritte innerhalb einer Schleife mehrmals durchlaufen werden. Zur Unterscheidung der einzelnen Durchläufe besitzt jeder Knoten neben seinem Zustand einen Iterationszähler. Diese werden beim Beenden eines Knotens erhöht und geben somit an, wie oft ein Knoten während gesamten Prozessausführung durchlaufen wurde.

Aus Gründen der Nachvollziehbarkeit, etwa für Analysen von Prozessausführungen, aber auch für einige Algorithmen werden alle Zustandsübergänge in einer Ausführungshistorie protokolliert (s. Abb. 2.12). Neben den betroffenen Prozessschritten und deren Zustandsübergänge gehören dazu auch der Zeitpunkt des Zustandsübergangs sowie der Bearbeiter, der den Zustandsübergang veranlasst hat.

Die Ausführungsregeln und die Korrektheitskriterien für Kontrollfluss gewährleisten, dass unter Einhaltung gewisser Annahmen bezüglich Fairness keine unerwünschten Zustände bei der Ausführung einer Prozessinstanz eintreten [Rei00]. Unerwünschte Zustände sind beispielsweise Prozessschritte, die aufgrund der Prozessstruktur nie aktiviert werden können und damit überflüssig sind. Ebenso unerwünscht sind Prozesse, die in endlicher Zeit keinen definierten Endzustand erreichen.

2.2.3.3 Datenzugriff

Neben dem Kontrollfluss betrifft die Ausführung einer Prozessinstanz und deren Prozessschritte auch den Datenfluss und damit lesende und schreibende Datenzugriffe. Beim Starten oder Fortsetzen eines Prozessschritts wiederum werden die Werte seiner Eingabeparameter aus den entsprechenden Datenelementen gelesen. Beim Beenden eines Prozessschritts werden die Werte seiner Ausgabeparameter übernommen und den entsprechenden Datenelementen zugewiesen. Dabei wird überprüft, ob alle obligaten Ausgabeparameter tatsächlich mit Werten versorgt werden. Ist dies nicht der Fall, muss die Ausführung des Prozesses sofort mit einem Fehler abgebrochen werden. In diesem Fall erfüllt der Prozessschritt nicht die Zusicherungen, die bei der Korrektheitsprüfung zur Modellierungszeit verwendet wurden. Dies kann zu schwerwiegenden Konsequenzen führen, wenn ein obligates Lesen dieses Werts durch einen nachfolgenden Prozessschritt entgegen der Zusicherung nicht stattfinden kann.

Ähnlich wie beim Beenden eines Prozessschritts werden auch beim Unterbrechen seiner Ausführung die bereits vorhandenen Werte von Ausgabeparametern den entsprechenden Datenelementen zugewiesen. Da der Prozessschritt jedoch noch nicht abgeschlossen sondern nur unterbrochen ist, müssen einerseits noch nicht alle obligaten Ausgabeparameter mit einem

Wert versorgt worden sein, andererseits sind die Werte selbst noch nicht endgültig. Das heißt, die Werte können beim Fortsetzen des Prozessschritts nochmals geändert werden. Insbesondere werden sie von anderen Prozessschritten noch nicht gelesen.

Ähnlich wie beim Kontrollfluss verhindern die Ausführungsregeln zusammen mit den Korrektheitskriterien auch beim Datenfluss unerwünschte Zugriffe. So sind keine Prozessausführungen möglich, die zu nicht-deterministischem Lesen führen. Somit lassen sich die Werte von Eingabeparametern immer eindeutig bestimmen. Unter bestimmten Umständen erfordert das deterministische Lesen allerdings den aufwendigen *ReadDataElement*-Algorithmus (s. [Rei00] für Details). Der Algorithmus erfordert jedoch eine Ausführungshistorie sowie eine Datenhistorie. In der Datenhistorie wird protokolliert, welcher Wert einem Datenelement von welchem Knoten, zu welchem Zeitpunkt und von welchem Bearbeiter zugewiesen wurde. Aus Gründen des Datenschutzes können auch lesende Zugriffe in der Datenhistorie protokolliert werden. Dies ist seitens des ADEPT-Metamodells nicht erforderlich. Zudem können die Lesezugriffe aufgrund der Ausführungshistorie rekonstruiert werden.

2.2.3.4 Modellelemente

Sämtliche bei der Ausführung eines Prozesses anfallenden Daten sind Bestandteil der entsprechenden *Prozessinstanz* (s. Abschnitt 2.2.1.3), d. h. sie charakterisieren diese Prozessinstanz. Eine Prozessinstanz umfasst daher neben dem ihr zugeordneten Prozessgraphen und seinen Ausführungsregeln immer auch eine Menge von Zustandsdaten, etwa Knotenzustände, Iterationszähler, Werte von Datenelementen sowie Ausführungs- und Datenhistorie. Diese Laufzeitdaten repräsentieren im ADEPT-Metamodell eine konkrete Prozessausführung. Bei instanzspezifischen Änderungen gehören neben diesen Zustandsdaten auch noch eine Änderungshistorie [RRJK06] und strukturelle Informationen zu einer Prozessinstanz.

2.2.3.5 Weitere Aspekte

Weitere Aspekte des Ausführungsmodells umfassen abhängige Bearbeiterzuordnungen [Kub98] und variable Parallelität [Wol08]. Beide verknüpfen ähnlich wie Verzweigungsentscheidungen den Kontrollfluss mit Werten aus dem Datenfluss.

Abhängige Bearbeiterzuordnungen Die bisher informell diskutierten Bearbeiterzuordnungen sind statisch, d. h. die Menge der Bearbeiter, die einen Prozessschritt zur Ausführung angeboten bekommen, wird bei der Prozessmodellierung festgelegt. Die Bearbeitermenge wird zwar erst zur Laufzeit mittels des Organisationsmodells aufgelöst, es können dabei jedoch keine Informationen berücksichtigt werden, die sich erst zur Prozesslaufzeit ergeben. Ein Beispiel hierfür ist eine abhängige Bearbeiterzuordnung, die einen Prozessschritt demjenigen Bearbeiter zuweist, der auch den vorherigen Prozessschritt ausgeführt hat. Dabei kann nicht einfach dieselbe Bearbeiterzuordnung verwendet werden, da diese normalerweise mehrere Bearbeiter für die Ausführung qualifiziert und somit auch andere Bearbeiter in Frage kämen.

Lösen lässt sich dies nur, indem die abhängige Bearbeiterzuordnung relativ zur Bearbeiterzuordnung des vorherigen Prozessschritts definiert wird, etwa „gleicher Bearbeiter wie Knoten X“. Dies hat mehrere Konsequenzen: Zum einen muss gewährleistet sein, dass Knoten X existiert und stets vor dem Knoten mit der abhängigen Bearbeiterzuordnung ausgeführt wird. Dies lässt sich bereits bei der Modellierung analog zur Überprüfung obligater Eingabeparameter gewährleisten. Zum anderen muss die Bearbeiterzuordnung zur Laufzeit entsprechend

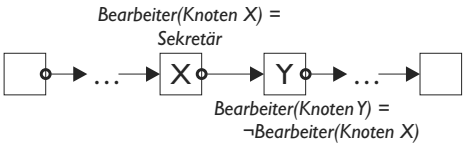


Abbildung 2.13: Vier-Augen-Prinzip durch abhängige Bearbeiterzuordnung

aufgelöst werden können, wozu Laufzeitdaten (etwa die Ausführungshistorie zur Ermittlung des Bearbeiters von Knoten X) benötigt werden.

Beides lässt sich konzeptionell einfach über den Datenfluss lösen: Knoten X bekommt einen zusätzlichen obligaten Ausgabeparameter für den Bearbeiter. Dieser wird in ein Datenelement geschrieben, das der Knoten mit der abhängigen Bearbeiterzuordnung obligat liest. Durch die Korrektheit des Datenflusses wird somit gewährleistet, dass die Abhängigkeit zur Laufzeit über den Datenfluss aufgelöst werden kann. Dies erspart den Zugriff auf die Ausführungshistorie.

Abhängige Bearbeiterzuordnungen sind nicht darauf beschränkt, einen Bearbeiter eines anderen Knotens zu spezifizieren. Die über das Datenelement bereitgestellte Information kann beliebig in der Bearbeiterzuordnung verwendet werden. So kann etwa auch der Vorgesetzte eines Bearbeiters eines Vorgängerknotens spezifiziert werden, sofern das zugrundeliegende Organisationsmodell diese Information bereitstellt. Auch das Vier-Augen-Prinzip lässt sich mittels abhängiger Bearbeiterzuordnungen leicht realisieren, indem „gleicher Bearbeiter wie Knoten X“ negiert wird (s. Abb. 2.13).

Wie wir noch detailliert vorstellen werden, lässt sich das Konzept auch auf andere Knotenattribute übertragen. So können Knotennamen beispielsweise Werte aus Datenelementen referenzieren und diese dann zur Laufzeit integrieren. Dies erlaubt es, dass ein bestimmter Knoten einer Prozessvorlage in allen Prozessinstanzen dieser Prozessvorlage einen unterschiedlichen Namen hat. Wird der Knotenname in den Arbeitslisten verwendet, können die Benutzer die entsprechenden Arbeitslisteneinträge leicht unterscheiden.

Variable Parallelität Listenwertige Datenstrukturen, deren Größe erst zur Laufzeit feststeht, können mit den bisher vorgestellten Kontrollflusselementen nur mittels einer Schleife elementweise bearbeitet werden. Dies liegt daran, dass nur mittels Verzweigungsentscheidungen bei der Ausführung eine Beziehung zwischen Kontroll- und Datenfluss hergestellt werden kann. Es kann aber sinnvoll sein, listenwertige Datenstrukturen nicht nur sequentiell, sondern bei Bedarf auch parallel elementweise zu bearbeiten. Dies wird durch *variable Parallelität* ermöglicht [RR06, Wol08]. Variable Parallelität realisiert das Kontrollflussmuster der Mehrfachinstanzierung (Muster 13 und 14, [vdAtHKB03]).

Da bei variabler Parallelität, im Gegensatz zu normaler Parallelität und Schleifen, der Kontrollfluss erst dynamisch zur Laufzeit bestimmt wird – die Anzahl der parallelen Zweige steht erst im Verlauf der Ausführung der Prozessinstanz fest – kann dies nicht statisch vormodelliert werden. Eine einfache Möglichkeit der dynamischen Allokation besteht in der Verwendung von Subprozessen (s. Abschnitt 2.2.2.3). Subprozesse werden weitgehend wie normale Prozesse erzeugt, gestartet und ausgeführt. Damit bezieht sich variable Parallelität immer auf einen Knoten, an dem ein Subprozess hinterlegt ist. Für die dynamische Allokation muss dieser Knoten mindestens einen listenwertigen Eingabeparameter besitzen, für dessen Listenelemente jeweils eine Subprozessinstanz gestartet wird. Weitere listenwertige Eingabeparameter können auf die gleiche Art und Weise auf die einzelnen Subprozessinstanzen verteilt werden (s. Abb. 2.14).

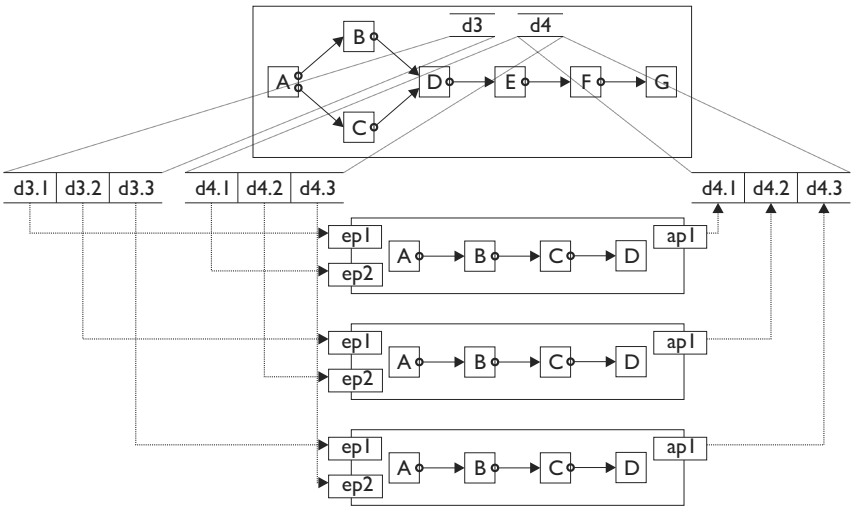


Abbildung 2.14: Variable Parallelität

Bei nicht-listenwertigen Eingabeparametern erhalten alle Subprozessinstanzen denselben Wert. Dies gilt auch für listenwertige Eingabeparameter, die nicht aufgeteilt werden sollen. Die Ausgabeparameter der Subprozesse sind auf Ebene des Elternprozesses alle listenwertig; jede Subprozessinstanz liefert den Wert eines Listenelements.

2.3 ADEPT-Änderungsmodell

In diesem Abschnitt stellen wir das ADEPT-Änderungsmodell vor. Das Änderungsmodell stammt aus [Rei00] und [Rin04]; ebenso berücksichtigen wir Erweiterungen aus neueren Forschungsarbeiten [Jur06, RRJK06, RKL+06, RJR07] und den praktischen Einsatz [RD08, RDRM+09b, LKRD10]. Änderungen im ADEPT-Metamodell beziehen sich, wie bereits erwähnt, entweder auf Prozessvorlagen oder Prozessinstanzen (s. Abschnitt 2.2.1.2). In beiden Fällen hat Korrektheit wieder eine große Bedeutung. Deshalb beruht das Änderungsmodell auf einer umfassenden theoretischen Grundlage [RRMD09, RD09]. Hierzu gehört eine Menge an wohldefinierten Änderungsoperationen und -mustern [WRRM08, RMRW08a]. Diese werden sowohl auf Prozessvorlagen als auch auf Prozessinstanzen angewendet. Im Fall von Prozessvorlagen entstehen durch Änderungen neue oder geänderte Prozessvorlagen. Diese können dann wie die unveränderten Prozessvorlagen verwendet und instanziert werden. Änderungsoperationen werden somit auch dazu verwendet, Prozessvorlagen neu zu erstellen und anzupassen. Im Fall von Prozessinstanzen gelten die Änderungen nur für die betroffene Prozessinstanz, andere Prozessinstanzen derselben Prozessvorlage sind von den Änderungen nicht betroffen. Die Prozessinstanz erhält durch die Änderungen aus logischer Sicht eine *instanzspezifische Prozessvorlage*.

In Abschnitt 2.3.1 stellen wir Änderungsoperationen auf abstrakter Ebene vor und erläutern die Gemeinsamkeiten und Unterschiede bei ihrer Anwendung auf Prozessvorlagen und Prozessinstanzen. In Abschnitt 2.3.2 behandeln wir diese in einen größeren Kontext und diskutieren die Szenarien der Anwendung von Änderungsoperationen bei Prozessinstanzen, Prozessvorlagen sowie die Kombination beider Anwendungen.

2.3.1 Änderungsoperationen

Das ADEPT-Änderungsmodell definiert eine Menge an *Änderungsoperationen*. Hierzu gehören beispielsweise das Einfügen, Verschieben und Löschen einzelner Knoten (s. Abb. 2.16), aber auch das Einfügen, Verschieben und Löschen eines ganzen Kontrollblocks, etwa eines Verzweigungs- oder Schleifenblocks. Die theoretischen Grundlagen definieren die Änderungsoperationen als Transformationen auf Prozessgraphen. Durch geeignete Definition der Änderungsoperationen kann so gewährleistet werden, dass bestimmte Korrektheitskriterien (z. B. die Einhaltung der Blockstruktur, keine isolierten Knoten) bei jeder Transformation erhalten bleiben. Dazu besitzen die Änderungsoperationen klar definierte Vorbedingungen, die bei Anwendung auf einen Prozessgraphen erfüllt sein müssen. Erfüllt eine Änderungsoperation alle Vorbedingungen, ist sie mit dem entsprechenden Prozessgraphen *verträglich* (*compliant*).

Änderungsoperationen setzen sich aus einfachen *Änderungsprimitiven* zusammen [RW12]. Änderungsprimitive beschränken sich auf elementare Transformationen wie das Erzeugen, Ändern und Löschen einzelner Knoten, Kanten, Datenelemente oder Datenkanten des jeweiligen Prozessgraphen. Einzelne Änderungsprimitive können strukturelle Eigenschaften verletzen, weshalb sie im Regelfall nicht isoliert, sondern nur im Kontext einer Änderungsoperation angewendet werden dürfen. Beispielsweise kann ein Änderungsprimitiv beim Einfügen eines Knotens zwischen zwei anderen Knoten kurzzeitig zu einem isolierten Knoten führen; durch Anwendung weiterer Änderungsprimitive im Rahmen der Änderungsoperation wird die Isolation aber wieder aufgehoben.

Da Änderungsoperationen aus theoretischer Sicht Transformationen sind, können sie problemlos kombiniert werden, sofern die einzelnen Änderungsoperationen immer verträglich zum Ergebnis der jeweils vorherigen Transformation sind. Die kombinierte Anwendung mehrerer Änderungsoperationen wird als *Änderungstransaktion* bezeichnet. Durch Änderungstransaktionen werden komplexe Änderungen, die mehrere Änderungsoperationen erfordern, atomar durchgeführt. Ein Beispiel ist das Ersetzen eines Knotens durch einen anderen Knoten, was durch die beiden Änderungsoperationen Einfügen und Löschen realisiert werden kann. Bei Änderungstransaktionen steht weniger die strukturelle Korrektheit im Vordergrund, die ja durch die einzelnen Änderungsoperationen weitgehend gewährleistet wird, sondern vor allem die semantische Korrektheit eines Prozesses.

2.3.1.1 Änderungen von Prozessvorlagen

Bei der Änderung oder Erstellung von Prozessvorlagen betreffen die Änderungsoperationen und ihre jeweiligen Vorbedingungen nur die Prozessstruktur, also Knoten, Kanten, Datenelemente und Datenkanten (s. Abschnitt 2.2.2). Deshalb wird vor Anwendung einer Änderungsoperation ihre strukturelle Verträglichkeit überprüft. Erst wenn diese gewährleistet ist, kann die Änderungsoperation angewendet werden. Sie führt dann zu einer Änderung der Prozessstruktur und damit der Prozessvorlage.

Die Korrektheit einer Prozessstruktur bezüglich des ADEPT-Metamodells wird aus praktischen Gründen jedoch nicht ausschließlich über Vorbedingungen und die Definition von Änderungsoperationen realisiert. So sind einige Korrektheitskriterien relativ aufwendig zu überprüfen, etwa die Verklemmungsfreiheit bei Synchronisationskanten zwischen verschiedenen Zweigen einer parallelen Verzweigung. Außerdem entsprechen die Änderungsoperationen für den Datenfluss weitgehend ihren Änderungsprimitiven. Dadurch ist der Datenfluss zwar nicht immer korrekt, aber die Erstellung und Änderung von Prozessvorlagen ist wesentlich einfacher.

Andernfalls ließe sich beispielsweise ein Teil des Datenflusses nur umständlich von optionalen auf obligate Datenzugriffe umstellen. Allerdings erfordert dies eine Korrektheitsprüfung nach Durchführung aller Änderungen und vor Freigabe der entsprechenden Prozessvorlage zur Instanzierung [DR09, DRRM+09].

2.3.1.2 Änderungen von Prozessinstanzen

Die Anwendung von Änderungsoperationen auf Prozessinstanzen (*instanzspezifische Änderungen*) umfasst neben der Prozessstruktur auch Informationen des Ausführungsmodells, etwa Knotenzustände [RW12]. Die Zustandsinformationen sind einerseits für Vorbedingungen relevant, andererseits müssen sie ggf. im Rahmen einer Änderungsoperation angepasst werden. Die strukturellen Änderungen bei Prozessinstanzen entsprechen den Änderungen bei Prozessvorlagen, allerdings gibt es bezüglich der strukturellen Vorbedingungen einige zusätzliche Einschränkungen. So sind nicht alle Änderungen, die strukturell bei einer Prozessinstanz gültig wären, auch sinnvoll. Beispielsweise ist es strukturell zulässig, eine Synchronisationskante von einem abgeschlossenen Knoten zu einem noch nicht aktivierten Knoten einzufügen. Diese Synchronisationskante hat jedoch keinerlei Auswirkung auf die Ausführung der Prozessinstanz und wird deshalb durch entsprechende Vorbedingungen ausgeschlossen.

Neben den Korrektheitskriterien für die Prozessstruktur müssen bei Änderungen an Prozessinstanzen auch die Zustandsinformationen berücksichtigt werden. Wichtigstes Kriterium hierbei ist, dass die Vergangenheit nicht geändert werden darf, d. h. bereits abgeschlossene Prozessschritte dürfen nicht geändert werden [RRD04c, RRD03b, CCPP98]. Dies gilt auch für Prozessschritte, die nicht mehr zur Ausführung kommen, etwa bei alternativen Verzweigungen. Theoretische Grundlage dieses Kriteriums ist, dass sich der bisherige Prozessablauf zum Zeitpunkt der Änderung (wie in der Ausführungshistorie protokolliert) auch mit der geänderten, instanzspezifischen Prozessvorlage erzeugen lässt. Das Kriterium selbst wird über die zusätzlichen Vorbedingungen der Änderungsoperationen realisiert.

Die Durchführung einer Änderungsoperation auf einer Prozessinstanz erfordert dieselben Änderungen wie bei einer Prozessvorlage. Zusätzlich können Anpassungen der Zustandsinformationen notwendig werden. Dies hängt von der Änderungsoperation und vom aktuellen Zustand der Prozessinstanz ab:

Beispiel 2.6 (Zustandsanpassungen bei instanzspezifischen Änderungen)
Das Löschen eines aktivierten Knotens ist zulässig, allerdings muss anschließend der Zustand seines Nachfolgerknotens entsprechend dem Zustandsmodell (s. Abschnitt 2.2.3.1) angepasst werden. Andernfalls könnte diese Änderungsoperation zum Abbruch der Prozessinstanz führen, wenn kein Knoten mehr aktiviert ist und somit kein Knoten mehr ausgeführt werden kann.

Ähnlich verhält es sich beim Einfügen eines Knotens vor einem bereits aktivierten Knoten (s. Abb. 2.15). Direkt nach dem Einfügen ist ein ein Knoten aktiviert, dessen Vorgänger noch nicht ausgeführt worden ist und damit weder beendet noch übersprungen ist. Die so entstandene Lücke ist gemäß dem Ausführungsmodell nicht zulässig (s. Abschnitt 2.2.3).

2.3.1.3 Weitere Aspekte

Neben den Vorbedingungen ist für jede Änderungsoperation der *Änderungskontext* definiert. Dieser umfasst die durch die Änderungsoperation hinzugefügten, geänderten oder gelöschten

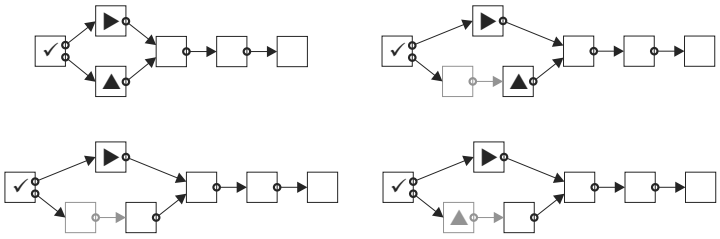


Abbildung 2.15: Zustandsanpassung bei instanzspezifischer Änderung

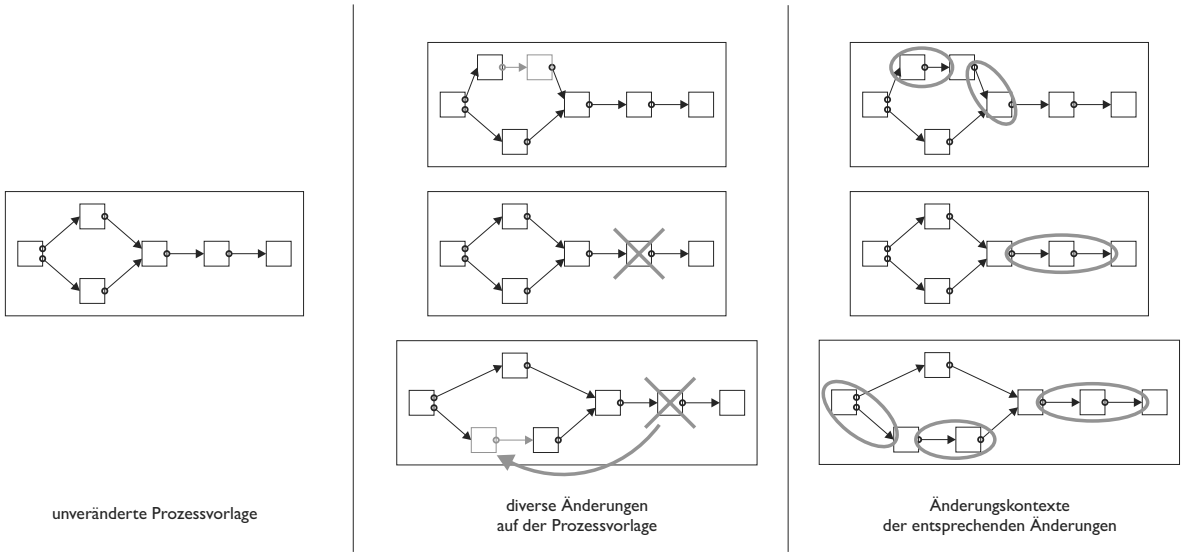


Abbildung 2.16: Änderungskontexte beim Einfügen, Löschen und Verschieben von Knoten

Objekte (z. B. Knoten, Kanten und Datenelemente) und ggf. zusätzlich weitere Objekte aus dem Randbereich der Änderung:

Beispiel 2.7 (Änderungskontext)
Der Änderungskontext beim (seriellen) Einfügen eines Knotens umfasst die zwei Knoten zwischen denen der neue Knoten eingefügt wird; beim Verschieben von Knoten umfasst der Änderungskontext die Kante, die die ursprüngliche Position des verschobenen Knotens „überspringt“, die zwei Knoten, zwischen denen der verschobene Knoten eingefügt wird, sowie den verschobenen Knoten selbst (s. Abb. 2.16).

Der Änderungskontext einer Änderungstransaktion ergibt sich aus der Aggregation der Änderungskontexte der einzelnen Änderungsoperationen. Der Änderungskontext ist einerseits für Korrektheitsprüfungen notwendig – soweit sich Korrektheit noch nicht aus den Änderungsoperationen selbst ergibt – andererseits spielen sie eine wichtige Rolle bei der internen Repräsentation von Änderungen.

Analog zu Ausführungs- und Datenhistorien werden Änderungsoperationen in einer Änderungshistorie protokolliert. Diese umfasst neben den Änderungsoperationen auch die Änderungsprimitive, die Änderungstransaktion sowie den Zeitpunkt der Durchführung und den auslösenden Benutzer [RRJK06, Rei00, GRMR+08]. Bei instanzspezifischen Änderungen wird darüber hinaus ein spezieller Eintrag in der entsprechenden Ausführungshistorie vorgenommen.

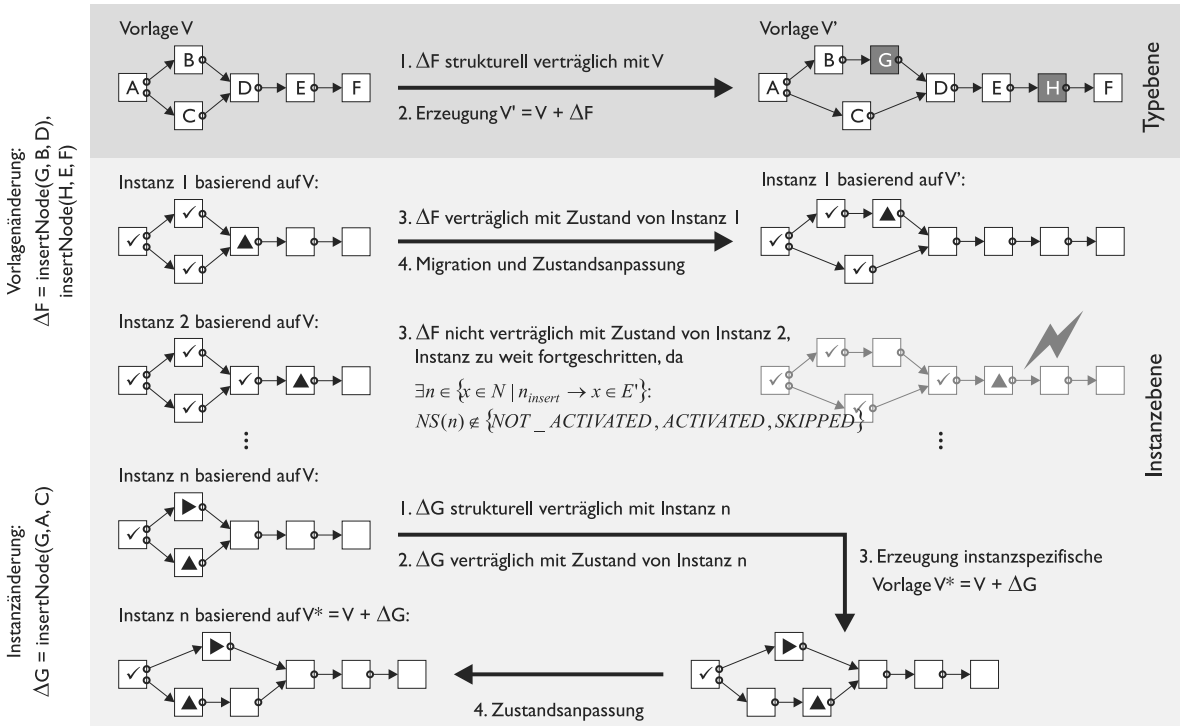


Abbildung 2.17: Prozessvorlagen- und Prozessinstanzänderung (logische Sicht)

Dieser verdeutlicht, dass die Ausführungshistorie ab diesem Eintrag nicht mehr den Ablauf auf der ursprünglichen Prozessvorlage widerspiegelt und zusätzliche Informationen notwendig sind (beispielsweise die Änderungshistorie), um den Prozessablauf nachzuvollziehen.

2.3.2 Änderungsszenarien

Wie in Abschnitt 2.3.1 vorgestellt, werden Änderungsoperationen sowohl für das Anlegen und Ändern von Prozessvorlagen als auch für instanzspezifische Änderungen verwendet. In diesem Abschnitt stellen wir drei verschiedenen Szenarien vor, in denen Änderungsoperationen angewendet werden können. Das erste Szenario betrifft instanzspezifische Änderungen, wie sie bereits im vorherigen Abschnitt skizziert wurden. Das zweite Änderungsszenario ist die *Schemaevolution*. Diese erweitert das Ändern einer existierenden Prozessvorlage V (und ihrer Transformation zu V') um die Propagation der vorgenommenen Änderungen auf bereits laufende Prozessinstanzen [RRD04a, RRD04c]. Das heißt, die laufenden Prozessinstanzen von V sollen auf die geänderte neue Prozessvorlage V' migriert werden und auf dieser weiterlaufen. Eine solche Instanzmigration ist vor allem im Kontext langlaufender Prozessinstanzen unabdingbar. Da von einer Schemaevolution auch geänderte Prozessinstanzen betroffen sein können [RRD04d, RRD04b], also Prozessinstanzen mit einer instanzspezifischen Prozessvorlage V^*abstammend von einer Prozessvorlage V, stellen wir im dritten Szenario die Kombination von instanzspezifischen Änderungen und einer Schemaevolution vor.

2.3.2.1 Instanzspezifische Änderungen

Wie bereits in Abschnitt 2.3.1.2 erläutert, sind bei instanzspezifischen Änderungen sowohl strukturelle als auch zustandsbezogene Vorbedingungen der Änderungsoperationen relevant. Außerdem werden sowohl die Prozessstrukturen als auch die Zustandsinformationen entsprechend den Änderungsoperationen angepasst.

Eine instanzspezifische Änderung durch eine Änderungstransaktion ΔG (s. Instanz n in Abb. 2.17) läuft folgendermaßen ab:

Zuerst wird die strukturelle Verträglichkeit der Änderungstransaktion mit der (ursprünglichen oder bereits instanzspezifischen) Prozessvorlage überprüft. Ist die Änderungstransaktion strukturell verträglich, wird ihre Verträglichkeit bezüglich des Zustands geprüft. Ist auch diese gegeben, werden die strukturellen Änderungen durchgeführt. Diese führen zu einer ausschließlich für die veränderte Prozessinstanz geltenden Prozessvorlage $V^* = V + \Delta G$. Abschließend wird noch eine Zustandsanpassung der geänderten Prozessinstanz durchgeführt, falls erforderlich.

Bei instanzspezifischen Änderungen können die Verträglichkeitsprüfungen auch direkt nacheinander für jede Änderungsoperation einer Änderungstransaktion erfolgen. Das heißt, die strukturelle Verträglichkeit wird nicht auf einmal für alle Änderungsoperationen der Änderungstransaktion überprüft, sondern strukturelle Verträglichkeit und die Verträglichkeit bezüglich des Zustands werden abwechselnd und für jede Änderungsoperation einzeln überprüft. Nur wenn eine Änderungsoperation hinsichtlich beider Aspekte verträglich ist, wird die nächste Änderungsoperation betrachtet.

2.3.2.2 Schemaevolution

Der erste Schritt einer Schemaevolution besteht aus der Änderung einer existierenden Prozessvorlage V durch eine Menge von Änderungen ΔF und damit die Erzeugung einer neuen Prozessvorlage $V' = V + \Delta F$ abstammend von V. Wie in Abschnitt 2.3.1.1 erläutert, wird vor Anwendung einer Änderungsoperation ihre strukturelle Verträglichkeit mit der resultierenden (Zwischen-)Vorlage geprüft. Anschließend wird die Änderungsoperation angewendet, was zu entsprechenden strukturellen Anpassungen der Prozessvorlage führt. Sind die Änderungen abgeschlossen und erfüllt die neue Prozessvorlage alle Korrektheitskriterien, kann sie für die Erzeugung neuer Prozessinstanzen herangezogen werden.

Bei wichtigen Umgebungsänderungen, beispielsweise der Änderung gesetzlicher Vorgaben, die auch für laufende Prozessinstanzen relevant sind, können die auf der Prozessvorlage durchgeführten Änderungen auch auf die entsprechenden Prozessinstanzen propagiert werden [RRD04d, RRD04b]. Dabei wird bei den betroffenen Prozessinstanzen die zugrundeliegende Prozessvorlage V durch V' ausgetauscht. Dies ist jedoch im allgemeinen nicht für alle Prozessinstanzen möglich, da wieder die Korrektheitskriterien des ADEPT-Metamodells eingehalten werden müssen. So kann eine Prozessinstanz nicht auf die neue Prozessvorlage V' migriert werden, wenn die Änderungen ΔF einen Bereich betreffen, den die Prozessinstanz bereits durchlaufen hat (vgl. Instanz 2 in Abb. 2.17). Eine Ausnahme bilden Schleifen, bei denen ein Bereich mehrfach durchlaufen werden. Vor der Propagation muss bezogen auf die Änderungstransaktion ΔF für jede Prozessinstanz, die migriert werden soll, die Verträglichkeit bezüglich des Zustands überprüft werden (sog. *Trace Equivalence*) [RW12]. Die Verträglichkeit entspricht der Verträglichkeit bezüglich des Zustands bei instanzspezifischen Änderungen [RD98, Rei00]. Ist sie gewährleistet, wird die Prozessinstanz migriert.

Im Gegensatz zu instanzspezifischen Änderungen sind bei der Migration keine Änderungen an der Struktur notwendig, da diese bereits in der Prozessvorlage V' realisiert sind. Die Migration selbst besteht lediglich aus dem Umhängen der Prozessinstanz von V auf V'. Falls erforderlich müssen anschließend jedoch noch Zustandsanpassungen der Prozessschritte wie bei instanzspezifischen Änderungen vorgenommen werden. Eine solche Zustandsanpassung ist beispielsweise notwendig, wenn durch ΔF ein neuer Knoten vor einem aktuell ausführbaren, aber noch nicht ausgeführten Knoten eingefügt wird (vgl. Instanz 1 in Abb. 2.17). Im Rahmen der Zustandsanpassung wird der neue Knoten als ausführbar markiert und dessen Nachfolger auf den Initialzustand zurückgesetzt [RD98, Rei00, RD09].

Sind die Änderungen ΔF nicht verträglich mit der instanzspezifischen Prozessvorlage $V^* = V + \Delta G$, kann diese Prozessinstanz nicht auf die Prozessvorlage $V' = V + \Delta F$ migriert werden. Sie kann jedoch, wie jede andere Prozessinstanz auch, Gegenstand instanzspezifischer Änderungen sein. Dadurch können beispielsweise nur die verträglichen Änderungsoperationen von ΔF auf sie angewendet werden. Diese stellt dann allerdings keine Schemaevolution mehr dar, sondern eine normale instanzspezifische Änderung, die im Rahmen einer Schemaevolution durchgeführt wird [RMRW08b, RMR10].

Ein wichtiger Unterschied zwischen instanzspezifischen Änderungen und Schemaevolution ist die Reihenfolge der Verträglichkeitsprüfung und der Anwendung von Änderungen. Bei instanzspezifischen Änderungen werden alle Überprüfungen für jede Änderungsoperation sofort durchlaufen. Dies erlaubt, bei jeder Änderungsoperation sofort sowohl auf Struktur- als auch auf Zustandsverträglichkeit zu überprüfen und ggf. die Änderungsoperation sofort zurückzuweisen. Bei einer Schemaevolution dagegen wird jede Überprüfungsphase jeweils für eine Menge von Änderungsoperationen (also die entsprechende Änderungstransaktion) ausgeführt, bevor die nächste Überprüfungsphase stattfindet. Dies liegt daran, dass die Änderungen auf einer Prozessvorlage eingebracht werden und dort nur strukturelle Verträglichkeit geprüft werden kann; Zustandsverträglichkeit lässt sich erst bei der Propagation der Änderungen auf Prozessinstanzen testen.

Weitere Unterschiede gibt es bei der Verträglichkeit bezüglich des Zustands. So ist es bei Schemaevolution erlaubt, eine Prozessinstanz zu migrieren, wenn Änderungen einen abgewählten Zweig einer alternativen Verzweigung betreffen. Dies ist kein Problem, da die Ausführungshistorie der Prozessinstanz sich auch mit diesen Änderungen erzeugen lässt. Die Prozessinstanz hätte somit von Anfang an auf der neuen Prozessvorlage mit den Änderungen laufen können und hätte trotzdem dieselbe Ausführungshistorie erzeugt. Die Änderung eines abgewählten Zweigs einer alternativen Verzweigung wäre auch instanzspezifisch zulässig; da sie aber keinerlei Auswirkungen auf die Prozessinstanz hat, ist eine solche Änderung nicht erwünscht und wird deshalb zurückgewiesen. Dagegen ist sie im Rahmen einer Schemaevolution durchaus sinnvoll, da hier möglichst viele Prozessinstanzen auf die neue Prozessvorlage migriert werden [Rin04]. Dies hat unter anderem den Vorteil, dass die Prozessinstanzen auch bei einer zukünftigen Schemaevolution aufgrund von Änderungen an der zuvor aktualisierten Prozessvorlage berücksichtigt werden können.

2.3.2.3 Schemaevolution bei instanzspezifischen Änderungen

Da das ADEPT-Änderungsmodell sowohl instanzspezifische Änderungen als Änderungen im Kontext einer Schemaevolution ermöglicht, kann es sich ergeben, dass die Prozessvorlage V einer bereits individuell geänderten Prozessinstanz durch Anwendung der Änderungsoperationen ΔF zu $V' = V + \Delta F$ ändert und die Prozessinstanz somit migriert werden soll [RRD04b,

RRD03b, RRD04d]. Hier kann nicht direkt das Verfahren für unveränderte Prozessinstanzen angewendet werden, da die Prozessinstanz aufgrund der individuellen Änderung nicht auf der Prozessvorlage V, sondern auf einer durch Änderungsoperationen ΔG individuell geänderten Prozessvorlage $V^* = V + \Delta G$ beruht. Damit können die Verträglichkeitsprüfungen von ΔF bezüglich des Zustands unter Umständen gar nicht mehr angewendet werden. Dies ist beispielsweise der Fall, wenn ΔF einen Knoten betrifft, der durch ΔG gelöscht wurde und damit in V^* gar nicht mehr vorhanden ist.

Aus diesem Grund müssen bei der Migration von Prozessinstanzen mit instanzspezifischen Änderungen im Kontext einer Schemaevolution vor den Zustandsüberprüfungen und -anpassungen erst die Prozessstrukturen untersucht werden. Hierzu reicht es nicht aus, einfach die strukturelle Verträglichkeit durch Anwendung der Änderungen ΔF der Prozessvorlage $V' = V + \Delta F$ auf die instanzspezifische Vorlage $V^* = V + \Delta G$ zu überprüfen. Vielmehr muss die Kombination der Änderungsoperationen ΔF und ΔG auf Verträglichkeit überprüft werden. Wichtig ist dabei, in welcher Beziehung die Mengen von Änderungsoperationen ΔF und ΔG zueinander stehen [RRD04b]. Diese können gleich, Teilmengen voneinander, überschneidend oder disjunkt sein (s. Abb. 2.18). Die jeweilige Beziehung bestimmt den Ablauf der Migration der Prozessinstanz.

Für die strukturelle Verträglichkeit sind die Änderungsoperationen relevant, die nur in einer der Mengen vorhanden sind, also die symmetrische Differenz

$$H = (\Delta F \cup \Delta G) \setminus (\Delta F \cap \Delta G)$$

Sind die Änderungsoperationen $\Delta F \setminus \Delta G$ und $\Delta G \setminus \Delta F$ strukturell verträglich mit der Prozessvorlage V, wird die Verträglichkeit bezüglich des Zustands von $\Delta F \setminus \Delta G$ gegenüber dem momentanen Zustand der Prozessinstanz überprüft. Ist auch hier Verträglichkeit gegeben, kann die geänderte Prozessinstanz auf die neue Prozessvorlage V' migriert werden. Hierzu müssen, wie bei der Durchführung instanzspezifischer Änderungen, zuerst strukturelle Anpassungen vorgenommen werden. Diese Anpassungen führen dazu, dass die Prozessinstanz eine instanzspezifische Vorlage $V'^* = V' + (\Delta G \setminus \Delta F)$ erhält. Anschließend wird der Zustand der Prozessinstanz noch bezüglich $\Delta F \setminus \Delta G$ angepasst.

Die notwendigen Schritte einer Schemaevolution bei instanzspezifischen Änderungen bilden damit eine Mischung aus Schemaevolution und instanzspezifischen Änderungen. Nach den strukturellen Überprüfungen und Anpassungen der Prozessvorlage, wie im Fall einer Schemaevolution, erfolgen nacheinander die Überprüfung der strukturellen Verträglichkeit und der Verträglichkeit bezüglich des Zustands, wie bei instanzspezifischen Änderungen. Allerdings umfassen diese Überprüfungen nur die Änderungsoperationen der symmetrischen Differenz der beiden Mengen an Änderungsoperationen (s. o.). Anschließend werden ggf., ebenso wie bei instanzspezifischen Änderungen, die Struktur und abschließend der Zustand angepasst. Die Zustandsanpassung wird ebenfalls nur für die Änderungsoperationen der symmetrischen Differenz notwendig.

Ein spezieller Fall ist die Gleichheit der Änderungsoperationen ΔF und ΔG. Das bedeutet, dass die instanzspezifischen Änderungen die mit der Schemaevolution einhergehenden Änderungen vorweggenommen haben. Die betroffene Prozessinstanz kann einfach auf die neue Prozessvorlage V' umgehängt und alle instanzspezifische Änderungen ΔG logisch entfernt werden. Dadurch wird aus der geänderten Prozessinstanz auf $V^* = V + \Delta G$ wieder eine unveränderte Prozessinstanz auf $V' = V + \Delta F$ [RRD04b].

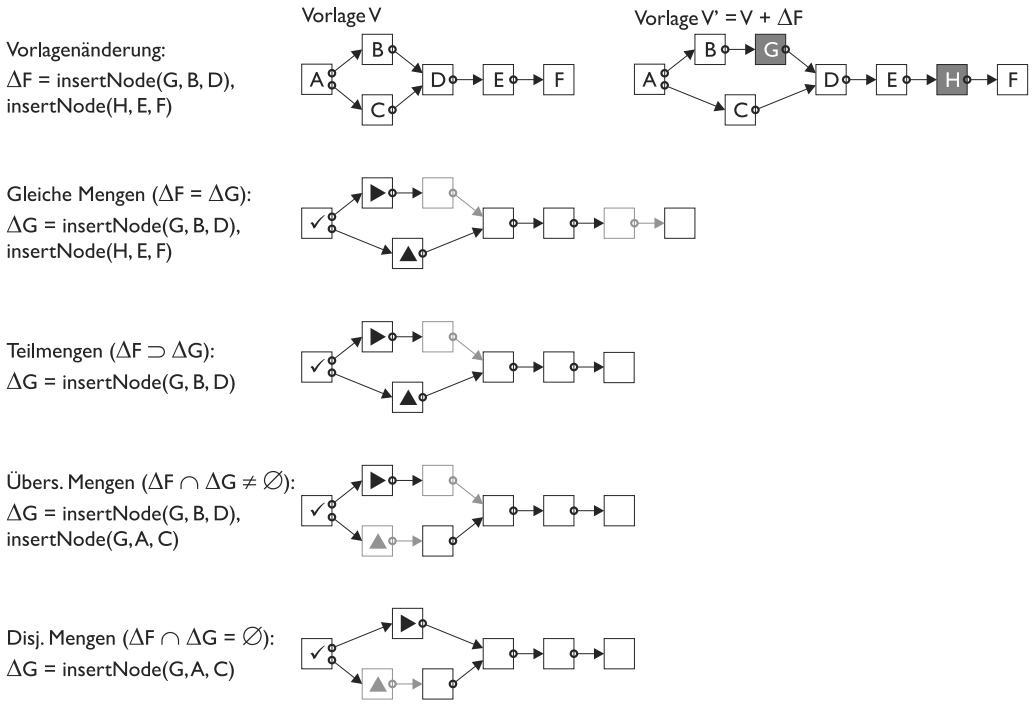


Abbildung 2.18: Beziehungen zwischen Vorlagenänderung und instanzspezifischen Änderungen

Beispiel 2.8 (Beziehung von Änderungsoperationen)
Sind die Änderungsmengen ΔF und ΔG gleich ($\Delta F = \Delta G$, $H = \emptyset$) oder ist ΔG eine Teilmenge von ΔF ($\Delta G \subset \Delta F$, $H = \Delta F \setminus \Delta G$), wurden die Änderungen der Schemaevolution (teilweise) durch die instanzspezifischen Änderungen vorweggenommen. In diesem Fall sind keine Überprüfungen notwendig und die Prozessinstanz kann direkt auf die neue Prozessvorlage V' migriert werden. Dabei verliert sie die instanzspezifischen Änderungen und wird zu einer normalen Prozessinstanz auf V'. Des weiteren sind keine Zustandsanpassungen notwendig. Es gilt:

$$V'^{*} = V' + (\Delta G \setminus \Delta F) = V' + \emptyset = V'$$

2.4 Anwendungsintegration (*Plug and Play*)

Ein weiterer Schwerpunkt des ADEPT-Metamodells ist die *Anwendungsintegration* [Bla96, AAD+04, RD09]. Diese bezieht ich auf die Softwareartefakte (u. a. Anwendungsprogramme, Anwendungsfunktionen, Eingabemasken, Prozeduren oder Webdienste), die die eigentliche Arbeit eines Prozessschritts erledigen oder zumindest den Bearbeiter bei der Erledigung unterstützen. Die Anwendungsintegration hat zum Ziel, die Heterogenität dieser Artefakte vor dem Benutzer zu verbergen und die Artefakte möglichst einfach und nahtlos in das PrMS zu integrieren. Einerseits bezieht sich dies auf die Prozessmodellierung, bei der die Artefakte mittels Ziehen und Ablegen mit der Maus (*Drag and Drop*) mit Prozessschritten verknüpft werden; dabei fällt kein oder nur wenig Konfigurationsaufwand an. Andererseits bezieht sich die Anwendungsintegration auf die Prozessausführung, bei der die verschiedenen Artefakte

vom PrMS, ähnlich wie Zusatzmodule (Plugins), integriert werden, was beispielsweise auch ihre graphische Oberfläche umfasst, die innerhalb und nicht separat von der prozessorientierten Anwendung angezeigt wird. Insgesamt soll durch die Anwendungsintegration die Verwendung der Artefakte gegenüber dem Benutzer ähnlich funktionieren wie *Plug and Play* bei Hardware, etwa der Anschluss von externen Geräten über USB (*Universal Serial Bus*).

Die Artefakte der Anwendungsintegration werden auch *Anwendungsbausteine* genannt, da sie die Bausteine der durch den Prozess realisierten Anwendung sind. Im ADEPT-Metamodell werden sie in Prozessen durch Aktivitäten (s. Abb. 2.2.2.3) repräsentiert. Dagegen repräsentieren *Aktivitätenvorlagen* die Anwendungsbausteine, bevor sie in einem bestimmten Prozess hinterlegt werden. Aktivitäten und Aktivitätenvorlagen stellen alle Informationen zur Verfügung, die für *Plug and Play* benötigt werden (s. Abb. 2.19). Hierzu gehören unter anderem Aufrufinformationen des Anwendungsbausteins, aber auch dessen Abhängigkeiten zu seiner Umgebung. Hierbei unterscheidet man zwischen logischen und technischen Abhängigkeiten. Logische Abhängigkeiten umfassen beispielsweise Ein- und Ausgabedaten eines Anwendungsbausteins oder Reihenfolgeabhängigkeiten zwischen verschiedenen Anwendungsbausteinen. Technische Abhängigkeiten bestehen z. B. zu einem bestimmten Betriebssystem, etwa durch die Verwendung einer speziellen Programmiersprache für einen Anwendungsbaustein, oder zu externen Bibliotheken, die zur Ausführung benötigt werden.

Die Kenntnis dieser Abhängigkeiten ist für die Anwendungsintegration wichtig, da die Abhängigkeiten die Verwendung eines Anwendungsbausteins einschränken und damit dem Ziel von *Plug and Play* widersprechen. Logische Abhängigkeiten schränken die Verwendung innerhalb von Prozessen ein und beeinflussen die Prozessmodellierung. Technische Abhängigkeiten wiederum betreffen die Ausführung (Laufzeit) und schränken damit die Ausführung dieser Anwendungsbausteine und der entsprechenden Prozessschritte ein. Um die Einschränkungen zu minimieren und weitgehend freies *Plug and Play* zu ermöglichen, werden diese Abhängigkeiten gekapselt; logische Abhängigkeiten durch den *Prozesskontext* einer Aktivität, technische Abhängigkeiten durch die *Ausführungsumgebung* eines Anwendungsbausteins (s. Abb. 2.20). Der Prozesskontext etabliert beispielsweise die Anbindung an den Datenfluss, d. h. er definiert, welcher Eingabeparameter von welchem Datenelement des Prozesses stammt und welcher Ausgabeparameter in welches Datenelement geschrieben wird. Die Ausführungsumgebung wiederum abstrahiert von technischen Details der Softwareartefakte, etwa davon, in welcher Programmiersprache ein Artefakt vorliegt und welche zusätzliche Software für seine Ausführung benötigt werden (z. B. JVM (*Java Virtual Machine*)).

Die Anwendungsintegration ist zwar weitgehend unabhängig vom verwendeten Prozessmetamodell, das ADEPT-Metamodell unterstützt sie allerdings in besonderem Maße. Hierzu gehört unter anderem die direkte Repräsentation von Anwendungsbausteinen durch die Aktivitäten eines Prozesses. Parameter (s. Abschnitt 2.2.2.2) trennen die Aktivitäten von Datenelementen und anderen Aktivitäten, was Teil der Kapselung durch den Prozesskontext ist. Auch die weitreichenden Korrektheitskriterien und -überprüfungen fördern *Plug and Play*, da sie mögliche Ausführungsprobleme, wie die Nicht-Versorgung notwendiger Eingabeparameter, bereits bei der Modellierung aufdecken. Dies reduziert Testaufwände und erleichtert damit die Verwendung von Anwendungsbausteinen in einem Prozess.

Im folgenden gehen wir im Detail auf die verschiedenen Aspekte der Anwendungsintegration zur Modellier- und Laufzeit ein.

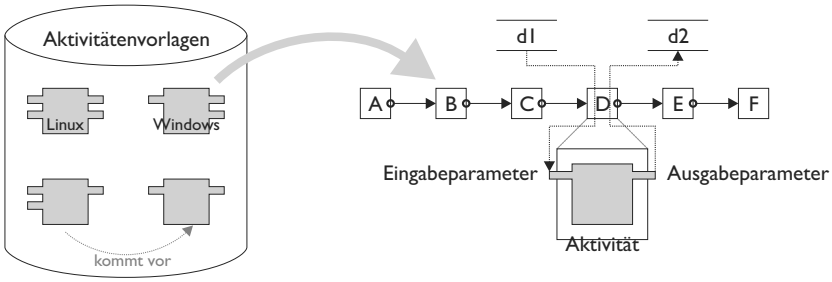


Abbildung 2.19: Einbindung von Aktivitätenvorlagen mittels Ziehen in eine Prozessvorlage

2.4.1 Modellierzeit

Plug and Play bei der Prozessmodellierung zielt darauf, die Hinterlegung von Aktivitäten an Prozessschritten einfach mittels Ziehen und Ablegen mit der Maus zu ermöglichen. Dabei wird der Prozesskontext weitgehend automatisch erstellt. Dazu gehören etwa die Verknüpfung von Ein-/Ausgabeparametern mit vorhandenen Datenelementen oder die Erstellung neuer Datenelemente sowie die Auflösung der Abhängigkeiten des Anwendungsbausteins. Idealerweise muss der Prozessmodellierer keine zusätzlichen Angaben machen und kann nach Hinterlegung aller Aktivitäten eines Prozesses diesen sofort ausführen. Des weiteren sollte der Prozessmodellierer einen Anwendungsbaustein in beliebigen Prozessen verwenden können. Damit dies sinnvoll möglich ist, müssen die Anwendungsbausteine sowohl möglichst unabhängig voneinander als auch von der Ausführungsumgebung sein. Je eingeschränkter die Anwendungsbausteine in ihrer Verwendung sind, etwa bezüglich Abhängigkeiten zu anderen Anwendungsbausteinen, desto eingeschränkter ist die Prozessmodellierung.
Die Aktivitätenvorlagen werden in einem zentralen Speicher hinterlegt, dem *Aktivitätenrepository* (s. Abb. 2.19 links). Das Aktivitätenrepository ist vergleichbar mit UDDI-Verzeichnis (*Universal Description, Discovery and Integration*) im Zusammenhang mit Webdiensten [RS04, Sto04]. Durch Wiederverwendung und zentrale Bereitstellung von Aktivitätenvorlagen wird die Zeitspanne zwischen Prozesserstellung und -ausführung erheblich verkürzt. So kann ein Anwendungsbaustein für das manuelle Schreiben einer Rechnung oder deren automatische Erstellung (z. B. bei einer Bestellung) in allen Prozessen verwendet werden, die eine Rechnungserstellung beinhalten.

Zur Modellierzeit sind die Ein- und Ausgabedaten eines Anwendungsbausteins wichtig. Dazu gehören Ein-/Ausgabeparameter, vergleichbar denen einer Prozedur, aber auch Daten, die aus einer externen Datenbank eingelesen oder in eine solche geschrieben werden. Während Ein-/Ausgabeparameter eine direkte Entsprechung im ADEPT-Metamodell haben, werden externe Daten über *virtuelle Parameter* repräsentiert [AAD+04]. Virtuelle Parameter werden zur Modellierzeit wie normale Parameter behandelt. Sie helfen dementsprechend bei Datenflussüberprüfungen. Dadurch wird gewährleistet, dass zur Laufzeit auch tatsächlich externe Daten vorhanden sind. Bei der Prozessausführung werden die Werte virtueller Parameter seitens des PrMS nicht berücksichtigt.

Neben den Ein-/Ausgabedaten eines Anwendungsbausteins spielt dessen Konfiguration eine große Rolle bei der Anwendungsintegration. Durch Konfigurationen lassen sich Anwendungsbausteine spezialisieren und an den gewünschten Anwendungskontext anpassen. Dies ist umso wichtiger, je generischer ein Anwendungsbaustein ist. Gerade für *Plug and Play* ist eine hohe Generizität von Anwendungsbausteinen wünschenswert, da sich dadurch ihre Unabhängigkeit

und somit ihre Wiederverwendbarkeit erhöht (s. Beispiel 2.9). Die Konfigurationswerte eines Anwendungsbausteins sind prozessspezifisch und gehören damit zum Prozesskontext (s. Abb. 2.20). Sie werden im Gegensatz zu den Werten von Ein-/Ausgabeparametern zur Modellierzeit festgelegt. Aufgrund der Gleichheit von Aktivitäten und Subprozessen gibt es sinnvollerweise neben den Konfigurationen für Anwendungsbausteine auch Konfigurationen für einen ganzen (Sub-)Prozess (s. Abb. 2.20). Werden diese mit Konfiguration von Anwendungsbausteinen kombiniert, lassen sich aktivitätsübergreifende Konfigurationen realisieren:

Beispiel 2.9 (Generischer Anwendungsbaustein und Prozesskonfiguration)
In einem Prozess finden mehrere Zugriffe auf eine SQL-Datenbank statt. Hierzu wird ein generischer Anwendungsbaustein verwendet, der in beliebige SQL-Datenbanken schreiben und aus ihnen lesen kann. Seine Konfiguration erfordert die Daten des Servers, eine Benutzerkennung sowie den Namen der Datenbanktabelle. Damit die Konfigurationswerte nicht bei jeder solchen Aktivität innerhalb des Prozesses gespeichert werden müssen, werden sie nur einmal in der Prozesskonfiguration hinterlegt. Die Prozesskonfiguration wird zur Laufzeit mit der Konfiguration der jeweiligen Aktivität vereinigt.

Generische Anwendungsbausteine besitzen nicht nur umfangreiche Konfigurationen. Durch die Möglichkeit der Spezialisierung können sich zudem ihre Abhängigkeiten sowie ihre Ein-/Ausgabeparameter ändern. Beispielsweise hängen bei der Ausführung von SQL-Abfragen mit dem erwähnten Anwendungsbaustein die Eingabeparameter davon ab, welche Bedingungen in der jeweiligen SQL-Abfrage vorhanden sind, etwa für die Filterung der Ergebnismenge nach mehreren Kriterien. Diese Variabilität bezüglich der Ein-/Ausgabeparameter muss beim *Plug and Play* berücksichtigt werden. Das heißt, dass der Prozessmodellierer die Parameter entsprechend anlegen muss oder sie abhängig vom Prozesskontext automatisch erzeugt werden. Durch dynamisches Hinzufügen von Eingabeparametern zur Modellierzeit lässt sich ein weiteres Problem lösen, das nicht nur generische Anwendungsbausteine haben:

Beispiel 2.10 (Zur Modellierzeit nicht feststehender Konfigurationswert)
In einem Prozess wird eine vom Benutzer spezifizierte Entität aus einer SQL-Datenbank ausgelesen. Dazu gibt ein Bearbeiter in einem Prozessschritt die Kennung (ID) der Entität ein. Im nächsten Prozessschritt wird die Entität gelesen und im darauffolgenden Prozessschritt wird sie dem Bearbeiter angezeigt. Der lesende Prozessschritt verwendet einen Anwendungsbaustein, der die Daten des Servers, eine Benutzerkennung sowie den Namen der Datenbanktabelle aus der Konfiguration erhält. Um unerwünschte Zugriffe zu vermeiden, darf aus Sicherheitsgründen immer nur die Benutzerkennung des suchenden Bearbeiters verwendet werden. Da Konfigurationswerte bereits zur Modellierzeit vorliegen, der Bearbeiter aber erst zur Laufzeit feststeht, kann die Benutzerkennung für den Datenbankzugriff nicht aus der Konfiguration kommen.

Eine einfache Lösung für dieses Problem wäre es, seitens des Anwendungsbausteins für alle vermeintlichen Konfigurationswerte entsprechende Eingabeparameter zu haben. Dies führt jedoch zu einer großen Menge an Eingabeparametern, die zudem allesamt zur Laufzeit zur Verfügung gestellt werden müssen. Oftmals jedoch stehen zahlreiche dieser Werte, wie normale Konfigurationswerte, bereits bei der Modellierung fest, etwa der Datenbankserver. Zudem müsste bereits bei der Implementierung des Anwendungsbausteins berücksichtigt werden, dass alle Konfigurationswerte als Eingabeparameter entgegengenommen werden. Solch erhebliche Auswirkungen auf die Implementierung sollten jedoch vermieden werden. Deshalb ist es sinnvoll,

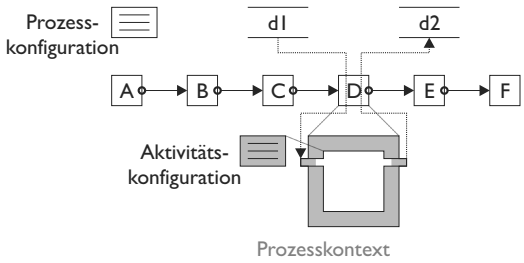


Abbildung 2.20: Prozesskontext einer Aktivität

die Trennung zwischen Konfigurationswerten und Eingabedaten beizubehalten. Stattdessen bekommt der Prozessmodellierer bei dem entsprechenden Anwendungsbaustein die Möglichkeit, Eingabeparameter für Konfigurationswerte zu erzeugen und feste Werte für Eingabeparameter festzulegen. Er ist daher nicht an die vom Implementierer vorgegebene Trennung gebunden. Diese Flexibilität erhöht die Wiederverwendbarkeit von Anwendungsbausteinen und fördert deshalb auch das *Plug and Play*.

Beispiel 2.11 (Festlegung eines Konfigurationswerts zur Laufzeit)
Da der Konfigurationswert für die Benutzerkennung nicht zur Modellierzeit feststeht (s. Beispiel 2.10) muss er zur Laufzeit bereitgestellt werden. Hierfür bietet sich der Datenfluss an. Dazu legt der Prozessmodellierer einen Eingabeparameter für die Benutzerkennung an und schreibt diese über einen Ausgabeparameter eines Vorgängerschritts. Zusätzlich wird bei der Modellierung festgelegt, dass dieser Eingabeparameter kein echter Eingabeparameter ist, sondern dem Anwendungsbaustein über seine Konfiguration angeboten wird. Der Anwendungsbaustein kann den Wert damit aus seiner Konfiguration auslesen, der Wert wird jedoch nicht wie die anderen Konfigurationswerte bei der Modellierung festgelegt sondern erst zur Laufzeit über den Datenfluss.

Beispiel 2.12 (Festlegung eines Eingabeparameterwerts zur Modellierzeit)
Ist der Anwendungsbaustein aus Beispiel 2.11 so implementiert, dass er die Benutzerkennung für den Datenbankzugriff nicht als Konfigurationswert erwartet, sondern bereits einen Eingabeparameter hierfür besitzt, stellt sich das Problem des vorherigen Beispiels nicht. Allerdings ist dann die umgekehrte Situation ein Problem: Es gibt aus technischen Gründen nur eine Benutzerkennung für den Zugriff auf die Datenbank, d. h., es können keine verschiedenen Benutzerkennungen verwendet werden. In diesem Fall ist es nicht sinnvoll, die Benutzerkennung über den Datenfluss zur Verfügung zu stellen, da er in jeder Prozessinstanz gleich ist. Stattdessen wird sein Wert wie der eines Konfigurationswerts durch den Prozessmodellierer festgelegt. Der Anwendungsbaustein erhält ihn zur Laufzeit aber über den entsprechenden Eingabeparameter zugewiesen.

2.4.2 Laufzeit

Da *Plug and Play* bei der Prozessmodellierung von den konkreten Implementierungen der Anwendungsbausteine abstrahiert, kann sich zur Laufzeit für die Bearbeiter von Prozessschritten das Problem ergeben, dass diese mit vielen unterschiedlichen Anwendungen arbeiten müssen. Abhilfe schafft hier die Integration der Anwendungsbausteine in das PrMS und insbesondere die Integration der graphischen Oberfläche der unterschiedlichen Anwendungsbausteine in ein

und dieselbe prozessorientierte Anwendung. Die Bearbeiter können damit alle Prozessschritte innerhalb derselben Anwendung ausführen. Die Integration zur Laufzeit umfasst bei *Plug and Play* aber nicht nur die graphische Oberfläche, sondern alle technischen Details von Anwendungsbausteinen; beispielsweise auch die einheitliche Installation von Anwendungsbausteinen. Im Idealfall werden Anwendungsbausteine so integriert, wie dies bei Zusatzmodulen (Plugins) von Anwendungsprogrammen der Fall ist; Anwendungsbausteine sind damit Zusatzmodule für das PrMS.

Für neu zu entwickelnde Anwendungsbausteine können diese Anforderungen meist gut umgesetzt werden. Die Integration existierender Artefakte (z. B. Anwendungen und Module) ist jedoch aus technischer Sicht oftmals schwierig. Hierzu gehören die unterschiedlichen Aufrufe verschiedener Artefakte, etwa über Kommandozeile, RPC (*Remote Procedure Call*) oder Webdienstaufruf, sowie die technischen Abhängigkeiten zu externen Bibliotheken und zur Systemumgebung, inklusive der graphische Benutzeroberfläche. Ein großer Teil dieser Schwierigkeiten lässt sich durch Kapselung der Anwendungsbausteine durch Ausführungsumgebungen lösen. Ausführungsumgebungen verbergen die Unterschiede der Anwendungsbausteine gegenüber dem PrMS und bieten diesem eine einheitliche Schnittstelle. Umgekehrt entkoppeln Ausführungsumgebungen die Anwendungsbausteine von der Laufzeitumgebung und stellen die benötigten Ressourcen in der richtigen Form bereit. Beispielsweise können graphische Benutzeroberflächen über externe Bibliotheken simuliert und in andere Benutzeroberflächen integriert werden.

Die Verwendung einer Ausführungsumgebung zur Laufzeit erhöht nicht nur die Unabhängigkeit zwischen PrMS und Anwendungsbaustein, sie ermöglicht auch die Einflussnahme auf zahlreiche Ausführungsaspekte. Beispielsweise lässt sich so die Ausführung ähnlich kontrollieren, wie einzelne Prozesse eines Betriebssystems, etwa das Beenden eines Betriebssystemprozesses von außen durch den *Windows Taskmanager*. Auch die entfernte Ausführung von Anwendungsbausteinen ist möglich, indem die Ausführungsumgebung alle Details der Verteilung übernimmt. Außerdem kann problemlos zwischen einer normalen und testweisen Ausführung gewechselt werden. Eine testweise Ausführung kann durch Verwendung einer anderen Konfiguration (einer *Testkonfiguration*) für den Anwendungsbaustein gekennzeichnet sein, z. B. wird bei einem Anwendungsbaustein für Datenbankzugriffe eine spezielle Testdatenbank verwendet. Daneben kann zu Testzwecken auch der Aufruf eines einfachen Formulars zur Anzeige der Eingabeparameterwerte und Entgegennahme der Ausgabeparameterwerte vom Benutzer den Aufruf des eigentlichen Anwendungsbausteins ersetzen.

Ein weiterer Vorteil der Entkopplung von Anwendungsbausteinen und PrMS ist die unabhängige und damit einfachere Entwicklung von Anwendungsbausteinen auf der einen und verwendenden prozessorientierten Programm auf der anderen Seite. Steht eine entsprechende Ausführungsumgebung zur Verfügung, können Anwendungsbausteine von beliebigen prozessorientierten Programmen ausgeführt werden; umgekehrt kann ein prozessorientiertes Programm beliebige Anwendungsbausteine und damit beliebige Prozessinstanzen ausführen. Eine Ausführungsumgebung selbst hängt zwar sowohl vom Anwendungsbaustein als auch vom verwendenden Programm ab, sie kann aber meist auf allgemeine Weise implementiert werden und so zahlreiche Anwendungsbausteine kapseln, etwa eine Ausführungsumgebung für den Aufruf beliebiger Webdienste.

3 Anforderungen an Architektur und Implementierung

In diesem Kapitel diskutieren wir sowohl die funktionalen als auch nicht-funktionalen Anforderungen an die Architektur und Implementierung adaptiver PrMS. Diese bilden den Ausgangspunkt für die Konzeption und Umsetzung des Systems sowie die Realisierung fortschrittlicher Implementierungskonzepte in den nachfolgenden Kapiteln. Zuerst gehen wir darauf ein, wie sich diese Anforderungen im Vorfeld der Arbeit ergeben haben.

3.1 Ermittlung der Anforderungen

Die im Rahmen dieser Arbeit relevanten Anforderungen stammen aus zahlreichen Quellen. Die grundlegenden funktionalen Anforderungen gehen auf die vorgestellten ADEPT-Grundlagen zurück, d. h. sie beziehen sich auf das ADEPT-Metamodell und das ADEPT-Änderungsmodell [Rei00, RD98, RD97]. Des weiteren orientieren sich die Anforderungen an denen vergleichbarer Softwaresysteme, etwa DBMS [HR01] und Informationssystemen. Weitergehende Anforderungen haben sich aus der Anwendung von ADEPT in verschiedenen Fachgebieten ergeben, etwa Transportwesen und Logistik [BKK04], elektronischem Handel [BBKK02], Gesundheitswesen [RHD98, LR05, LR07, DRK00, Rei11], Softwaretechnik [GOR10a, GOR10b, GOR11b], Releasemanagement [MHHR06, MRH07, MRH08] und Katastrophenschutz [WR09]. Außerdem wurden bei den Anforderungen auch Ergebnisse genereller Studien zu prozessorientierten Informationssystemen [MRB08] berücksichtigt.

Zu diesen Anforderungen kommen eigene Erfahrungen mit der ersten prototypischen Implementierung von ADEPT [Rei00, RRD03a] hinzu. Erkenntnisse, die zusammen mit Partnern aus Industrie und Forschung im Rahmen des AristaFlow-Projekts [RDRM+09a, RRKD05] gewonnen wurden, runden die Anforderungen ab. Dabei zeigte sich unter anderem, dass adaptives Prozessmanagement als EAI-Plattform und damit als Grundlage für speziellere Informationssysteme dienen kann. Gerade für den Einsatz als EAI-Plattform haben nicht-funktionale Anforderungen aber eine große Bedeutung. So sind Leistungsfähigkeit und Wartbarkeit, die sich beide auf verschiedene Aspekte auswirken, unabdingbar dafür, dass das PrMS über den Status eines Demonstrators hinauskommt.

3.2 Funktionale Anforderungen

Eine wichtige funktionale Anforderung eines Systems für adaptives Prozessmanagement ist die **Prozessausführung** (FA 1). Dabei werden die im entsprechenden Metamodell definierten Ausführungsregeln auf Prozesse angewendet. Die Prozesse selbst liegen in Datenstrukturen vor, die dem zugrundeliegenden Metamodell entsprechen. Durch Anwendung der Ausführungsregeln ändern sich die Zustände der Prozesse. Um diese Zustandsänderungen und damit Prozessausführungen nachvollziehen zu können, müssen alle **Zustandsänderungen** im Prozess **protokolliert** werden (FA 2). Die Nachvollziehbarkeit kann nicht nur für den Benutzer von Interesse sein, sondern auch für bestimmte funktionale Aspekte des Metamodells, etwa zwecks Gewährleistung von Korrektheitseigenschaften im Kontext einer Schemaevolution (s. Abschnitt 2.3.2.2).

Zu den protokollierten Informationen gehören einerseits die Zustandsänderungen von Prozessschritten und damit verknüpfte Informationen, z. B. Start eines Prozessschritts mit Angabe des Zeitpunkts und des auslösenden Bearbeiters (FA 2.1 **Ausführungshistorie**). Andererseits gehören dazu Zustandsänderungen der Daten, die einem Prozess bekannt sind (FA 2.2 **Datenhistorie**). Dies können beispielsweise Daten sein, die Grundlage für Entscheidungen bei alternativen Verzweigungen sind. Neben den Änderungen von Daten durch schreibende Zugriffe können zusätzlich lesende Zugriffe protokolliert werden. Dies ist jedoch nicht unbedingt notwendig, da sich die lesenden Zugriffe durch die Strukturen des jeweiligen Prozessgraphen ergeben und somit auch ohne Protokoll eindeutig nachvollzogen werden können.

An einem Prozess sind gewöhnlich mehrere Bearbeiter beteiligt, die verschiedene Prozessschritte ausführen. Dazu muss das System geeigneten Bearbeitern den entsprechenden Prozessschritt jeweils zuordnen können. Dies führt zu zwei weiteren funktionalen Anforderungen:

Bei der Prozessmodellierung muss für Prozessschritte eine Bearbeiterzuordnung hinterlegt werden. Diese spezifiziert den oder die für die Ausführung des Prozessschritts qualifizierten Bearbeiter. Sinnvollerweise beruhen diese Bearbeiterzuordnungen nicht direkt auf konkreten Bearbeitern. Eine solch enge Kopplung von Prozessschritten an Bearbeiter würde dazu führen, dass Stellenneubesetzungen oder die Umverteilung von Aufgaben in einer Organisation immer auch Anpassungen der Bearbeiterzuordnungen und damit der Prozessvorlage erfordern [RMR09, RMR08, RCWR04]. Durch den Bezug auf abstraktere Einheiten wie Stellen, Rollen oder Abteilungen vermeidet man dies. Hierzu wird ein **Organisationsmodell** benötigt (FA 3), das die abstrakten Einheiten mit den konkreten Bearbeitern in Bezug setzt. Welches Metamodell dem Organisationsmodell zugrunde liegt, ist aus Sicht der Anforderung weniger wichtig, solange sich damit möglichst viele existierende Organisationsmodelle einfach abbilden lassen und die notwendigen abstrakten Einheiten existieren. Für die Erstellung und Pflege des Organisationsmodells wird eine entsprechende **Verwaltungskomponente** benötigt (FA 3.1).

Bei der Prozessausführung werden die abstrakten Bearbeiterzuordnungen mit Hilfe des Organisationsmodells auf eine Menge von konkreten **Bearbeitern aufgelöst** (FA 3.2). Hierbei müssen vorgenommene Änderungen am Organisationsmodell berücksichtigt werden. Das heißt, die Bearbeiterzuordnungen sollten möglichst spät auf eine Bearbeitermenge abgebildet werden. Andernfalls stellt sich das eingangs beschriebene Problem bei Stellenneubesetzungen oder Umverteilungen von Aufgaben erneut. Die späte Auflösung kann allerdings dazu führen, dass sich eine leere Menge an Bearbeitern ergibt. Dies kann beispielsweise passieren, wenn eine bestimme Rolle einem Prozessschritt zugeordnet ist, für diese Rolle im Organisationsmodell

aber kein konkreter Bearbeiter mehr existiert. Auf diese Ausnahmesituation muss das PrMS entsprechend sinnvoll reagieren; in keinem Fall darf der Prozessschritt dadurch liegenbleiben.

Sind die Prozessschritte konkreten Bearbeitern zugewiesen, müssen sie den Bearbeitern entsprechend aufbereitet angeboten werden. Dies kann beispielsweise mittels Einträgen in Arbeitslisten erfolgen, wobei jeder Bearbeiter eine eigene Arbeitsliste besitzt. Die Aufbereitung und Darstellung der Prozessschritte ist Aufgabe der **Arbeitslistenverwaltung** (FA 4). Da die Arbeitslisten die Zuordnung von Bearbeiten zu Prozessschritten realisieren, sind sie auch Gegenstand von Funktionen, die diese Zuordnung anpassen. Hierzu gehört beispielsweise die lastbezogene **Verteilung** der Arbeitslisteneinträge. Dabei erhalten nicht alle durch die Bearbeiterzuordnung qualifizierten Bearbeiter einen Prozessschritt in ihrer Arbeitsliste, sondern nur diejenigen, die aktuell wenig oder keine Einträge in ihrer Arbeitsliste haben. So lässt sich die Auslastung der einzelnen Bearbeiter ausgleichen. Die Verteilung kann man beliebig erweitern, beispielsweise indem der Aufwand eines Prozessschritts mit berücksichtigt wird und nicht nur deren Anzahl. Auch die Berücksichtigung der Abwesenheit von Bearbeitern, etwa bei Urlaub, und die Anwendung entsprechender Vertreterregelungen ist Teil der Verteilung von Arbeitslisteneinträgen. Daneben ist in bestimmten Fällen auch eine manuelle Verteilung sinnvoll, was einer **Delegation** entspricht [RvdAtHE05]. Auch beliebige andere Verteilungsverfahren sind denkbar, da diese ggf. sehr speziell sind, ist das PrMS diesbezüglich erweiterbar. Ähnlich wie bei der Bearbeiterzuordnung muss die Arbeitslistenverwaltung sinnvoll mit Sonderfällen umgehen können. Ein Sonderfall sind beispielsweise Zyklen in Vertreterregeln, d. h. zwei Bearbeiter sind gleichzeitig abwesend und vertreten sich gegenseitig (ggf. auch transitiv). Bei naiver Umsetzung kann dies leicht zu einer Endlosschleife in der Arbeitslistenverwaltung führen.

Werden Prozessschritte von Menschen durchgeführt, ist eine rechtzeitige Bearbeitung nicht zwingend gewährleistet. So kann ein Prozessschritt länger unausgeführt bleiben als eigentlich sinnvoll, beispielsweise wenn die Bestellung eines Kunden liegenbleibt. Bei einer automatischen Prozessausführung durch ein PrMS werden Prozesse deshalb sinnvollerweise auch **überwacht** (FA 5). Besitzt ein Prozess zeitliche Einschränkungen kann das System bei drohenden Verletzungen aktiv werden. Durch **Eskalation** (FA 5.1) kann das PrMS die Benutzer gezielt auf solche Verletzungen hinweisen, etwa per E-Mail oder durch Hervorhebung des entsprechenden Prozessschritts in der Arbeitsliste. Auch die Delegation an einen wenig ausgelasteten Bearbeiter oder einen Vorgesetzten sind sinnvolle Eskalationsmaßnahmen. Wie bei der Verteilung von Arbeitslisteneinträgen sind hier Erweiterungen denkbar, die das System vorsehen sollte.

Überwachung ist aber nicht nur bei explizit hinterlegten Einschränkungen sinnvoll, sondern auch für die Ermittlung von Leistungskennzahlen (KPI (*Key Performance Indicator*)). Leistungskennzahlen lassen sich anhand von Prozessen in der Regel leicht ermitteln; Beispiele sind die durchschnittliche Bearbeitungszeit bestimmter Prozessschritte oder die mit bestimmten Prozessen realisierte Umsätze eines Unternehmens. Diese Leistungskennzahlen beruhen auf **Analysen** laufender und abgeschlossener Prozesse (FA 5.2). Das PrMS muss die dafür benötigten Daten zur Verfügung stellen und Anfragen sowie Datenaggregation ermöglichen. Auf diese Weise lassen sich auch Fehler und Engpässe in den Prozessen finden und beheben sowie Prozesse entsprechend optimieren. Das System muss die Aktualität der analysierten Daten sicherstellen und die Daten idealerweise in Echtzeit zur Verfügung stellen.

Da in Prozessen viele Daten anfallen, die Gegenstand von Analysen sein können, ist der Datenumfang entsprechend groß. Gleichzeitig soll die normale Prozessausführung hiervon nicht beeinflusst werden, etwa indem die Anzahl der analysierten Daten die Anzahl der gleichzeitig ausgeführten Prozesse beeinträchtigt. Aus diesen Gründen ist eine **persistente Speicherung**

aller Daten unerlässlich (FA 6). Hier ist darauf zu achten, dass die für Analysen benötigten Daten in geeigneter Form vorliegen. Dies ist kritisch für die nicht-funktionale Anforderung der **Leistungsfähigkeit** (NA 1): Einfach analysierbare Daten benötigen meist viel Speicherplatz und dementsprechend mehr Zeit für das Schreiben; umgekehrt erfordert schnelleres Schreiben oftmals Datenrepräsentationen, die wiederum für Analysen ungünstig sind.

Die Ausführung von Prozessschritten entspricht gewöhnlich dem Aufruf und der Ausführung eines Anwendungsbausteins. Die entsprechende Anwendungslogik kann neu implementiert sein, oftmals wird auch auf existierende Anwendungslogik zurückgegriffen. Damit muss das PrMS unterschiedlichste, auch existierende **Anwendungen integrieren und aufrufen** können (FA 7). So wird aus einem PrMS eine Integrationsplattform, die existierende Anwendungen über Prozesse miteinander verbindet (EAI). Für die Anwendungsintegration ist es nützlich, wenn die Architektur des PrMS die weitgehende Unabhängigkeit der Anwendungsbausteine gewährleistet. Dies bezieht sich einerseits auf andere Anwendungsbausteine, andererseits auf das System oder die Integrationsplattform selbst. Wie in Abschnitt 2.4.2 beschrieben, erhöht die Unabhängigkeit der Anwendungsbausteine (sog. lose Kopplung) deren Wiederverwendbarkeit in verschiedenen Prozessen [Erl05]. Die Unabhängigkeit der Anwendungsbausteine vom ausführenden PrMS wiederum erhöht die Flexibilität bezüglich Ausführungsort und -zeit von Anwendungsbausteinen.

Damit ein Prozess ausgeführt werden kann, muss er dem entsprechenden Metamodell genügen und in dessen definierten Datenstrukturen vorliegen. Dies erfolgt bei der **Prozessmodellierung** (FA 8). Die Prozessmodellierung muss nicht unbedingt durch vom PrMS bereitgestellte Modellierungswerkzeuge erfolgen, sofern das PrMS ein entsprechendes Datenaustauschformat definiert. Für **Adaptivität** (FA 9) ist es jedoch sinnvoller, die Prozessmodellierung direkt im System zu fordern.

Adaptivität ermöglicht **Änderungen an laufenden Prozessen** (FA 9.1). Dabei werden bestimmte Prozessinstanzen während ihrer Laufzeit strukturell geändert. Wichtig dabei ist, dass die Änderungen dem Metamodell entsprechen. So müssen alle strukturellen Einschränkungen, etwa Blockstrukturierung, auch bei geänderten Prozessinstanzen eingehalten werden Außerdem dürfen keine inkonsistenten Ausführungszustände entstehen, beispielsweise durch das Einfügen von Prozessschritten zwischen zwei bereits ausgeführten Prozessschritten oder das Löschen eines Prozessschritts, der für spätere Prozessschritte wichtige Daten liefert. Schließlich muss das System gewährleisten, dass eine instanzspezifische Änderung auch wirklich nur für eine bestimmte Prozessinstanz gilt und die anderen Prozessinstanzen hiervon unberührt bleiben.

Für die Definition von instanzspezifischen Änderungen sollten weitgehend dieselben Möglichkeiten bestehen wie bei der Prozessmodellierung. Das heißt, die Durchführung von Änderungen auf einer laufenden Prozessinstanz unterscheidet sich vom Prinzip her nur wenig von der Erstellung und Änderung von Prozessvorlagen. Es sind lediglich einige zusätzliche Einschränkungen zu beachten, damit kein inkonsistenter Ausführungszustand entsteht. Diese Einschränkungen müssen bereits durch das Metamodell vorgegeben werden, so dass das PrMS diese nur korrekt umsetzen muss.

Neben laufenden Änderungen einzelner Prozessinstanzen ermöglicht Adaptivität die Änderung von Prozessvorlagen zur Laufzeit und die Propagation dieser Änderungen auf laufende Prozessinstanzen (**Schemaevolution**, FA 9.2). Auch dabei kommt es zu Laufzeitanpassungen. Diese erfolgen jedoch nicht für einzelne Prozessinstanzen, sondern für alle Prozessinstanzen der betroffenen Prozessvorlage. Hier steht deshalb weniger die Modellierung im Vordergrund als vielmehr die korrekte Durchführung der Schemaevolution. Insbesondere darf die Sche-

maevolution keine inkonsistenten Zustände in den betroffenen Prozessinstanzen erzeugen und die Propagation der Änderungen auf bereits laufende Prozessinstanzen muss auch bei individuell geänderten Prozessen funktionieren. Zur Verhinderung inkonsistenter Zustände gelten dieselben Einschränkungen wie bei der Anwendung instanzspezifischer Änderungen. Bei der Schemaevolution mit individuell geänderten Prozessinstanzen entstehen zusätzliche Schwierigkeiten, da die individuellen Änderungen und die durch die Schemaevolution propagierten Änderungen widersprüchlich sein können oder (teilweise) gleich sind (s. Abschnitt 2.3.2.3).

Die Modellierung von Prozessschritten und deren Anbindung an den Prozesskontext soll möglichst automatisch im ***Plug and Play***-Stil erfolgen (FA 10). Das heißt, dass bei der Hinterlegung eines Anwendungsbaustein an einem Prozessschritt möglichst wenig durch den Modellierer gemacht werden muss. Besitzt der Anwendungsbaustein zugehörige Parameter, die er aus dem Prozess erhält oder an diesen übergibt, sollte diese Anbindung weitgehend automatisch erfolgen. Idealerweise verknüpft das System die Ein-/Ausgabeparameter mit semantisch gleichwertigen Ausgabe-/Eingabeparametern von Anwendungsbausteinen von Vorgänger-/Nachfolgerschritten [Bla96]. Dadurch erstellt das System automatisch einen korrekten und mit hoher Wahrscheinlichkeit auch den vom Prozessmodellierer gewünschten Datenfluss.

Durch die Möglichkeit der Schemaevolution in adaptiven PrMS sind kontinuierliche Prozessverbesserungen möglich [LRW08, LRW09a, LRW10, LRW11]. Hierdurch entsteht eine potentiell große Menge an Prozessvorlagen. Diese müssen benutzerfreundlich **verwaltet und versioniert** werden (FA 11). So lassen sich Änderungen an Prozessvorlagen jederzeit nachvollziehen, was beispielsweise wichtig wird, wenn eine durchgeführte Änderung nicht die erwartete Verbesserung mit sich bringt. Außerdem werden die Unterschiede zwischen Prozessvorlagen für Schemaevolution benötigt, um zu überprüfen, dass Propagation der Änderungen auf die betroffenen Prozessinstanzen keine Inkonsistenzen verursacht.

Schließlich sollen die von einem adaptiven PrMS bereitgestellten Funktionen nicht nur über vom System bereitgestellte Werkzeuge und Anwendungsprogramme, sondern auch über eine breit nutzbare **Programmierschnittstelle** (*Application Programming Interface* (API)) verfügbar sein (FA 12). Dies ermöglicht es, das System als **Plattform** für die Entwicklung unterschiedlichster Anwendungssysteme zu verwenden. Außerdem wird das PrMS selbst zu einer Menge von Diensten, die Gegenstand unternehmensweiter Anwendungsintegration (EAI) sein können. Somit können Funktionen des PrMS selbst als Anwendungsbausteine in den vom PrMS ausgeführten Prozessen verwendet werden.

3.3 Nicht-funktionale Anforderungen

Grundlage der nicht-funktionalen Anforderungen an ein adaptives PrMS ist der Standard ISO/IEC9126-Standard [ISO01]. Dieser hat Softwarequalität zum Ziel und berücksichtigt neben funktionalen auch nicht-funktionale Anforderungen. Allerdings fehlen im ISO/IEC9126-Standard nicht-funktionale Anforderungen, die kennzeichnend für Architekturen sind, etwa Einfachheit, Unabhängigkeit und Wiederverwendung. Dementsprechend gehen die in diesem Abschnitt vorgestellten nicht-funktionalen Anforderungen über die des ISO/IEC9126-Standards hinaus.

Die wichtigsten nicht-funktionalen Anforderungen für Softwaresysteme betreffen ihre **Leistungsfähigkeit** (NA 1) und **Robustheit** (NA 2). Insbesondere bei interaktiven Softwaresystemen, also bei von Menschen benutzten Softwaresystemen, ist beides unabdingbar. Ein robustes System erfüllt seine funktionalen Anforderungen zuverlässig und sollte möglichst

Tabelle 3.1: Funktionale Anforderungen an ein adaptives PrMS

FA 1	Prozessausführung
FA 2	Protokollierung
FA 2.1	Ausführungshistorie
FA 2.2	Datenhistorie
FA 3	Organisationsmodell
FA 3.1	Organisationsmodellverwaltung
FA 3.2	Auswertung von Bearbeiterzuordnungen
FA 4	Arbeitslistenverwaltung (Verteilung von Arbeitslisteneinträgen)
FA 5	Prozessüberwachung
FA 5.1	Eskalation
FA 5.2	Prozessanalysen
FA 6	Persistente Speicherung
FA 7	Anwendungsintegration und -aufruf
FA 8	Prozessmodellierung
FA 9	Adaptivität
FA 9.1	Instanzspezifische Änderungen
FA 9.2	Schemaevolution
FA 10	*Plug and Play* (Anwendungsintegration bei der Modellierung)
FA 11	Prozessverwaltung und -versionierung
FA 12	Programmierschnittstelle (API), Entwicklungsplattform

keine kritischen Fehler erzeugen. Tritt ein derartiger Fehler auf, bleibt die Funktionsfähigkeit eines robusten Systems weitgehend erhalten. Andernfalls drohen Systemausfälle, die die Verwendbarkeit aus Sicht der Nutzer signifikant verringert.

Eine grundlegende Leistungsfähigkeit ist notwendig, damit ein System von Nutzern akzeptiert und verwendet wird. Dazu gehört, dass Interaktionen und Befehle in einer angemessenen Zeit abgeschlossen sind oder das System zumindest innerhalb einer kurzen Zeitspanne reagiert. Bei Plattformen ist dies wichtig, da hier vorab nicht klar ist, ob eine Funktion interaktiv verwendet wird oder nicht. Dies hängt vom darauf aufbauenden Anwendungssystem ab. Im Fall eines PrMS kommt zusätzlich der Aspekt der Zeit in Prozessen hinzu [LWR10, LWR14, LPCR13]. Gibt es innerhalb von Prozessen strikte Zeitvorgaben, wird die Leistungsfähigkeit des PrMS gerade bei automatisch ausgeführten Prozessen und Prozessschritten wichtig. Ein effizientes Arbeiten und damit ein geringer Anteil des PrMS an der gesamten Bearbeitungszeit ist unabdingbar, wobei die Analysen zur Prozessüberwachung FA 5.2 große Auswirkungen haben können. Bei adaptiven PrMS kommen zusätzlich instanzspezifische Änderungen (FA 9.1) hinzu. Deren Realisierung sollte sich möglichst wenig auf die Leistungsfähigkeit auswirken.

Gerade die Umsetzung individueller Änderungen an Prozessinstanzen hat starke Auswirkungen auf die **Skalierbarkeit** (NA 1) des Systems. Skalierbarkeit bezeichnet das Verhältnis von gesteigerter Leistungsfähigkeit eines Systems zu den eingesetzten Ressourcen (z. B. Anzahl der Prozessoren). Eine gute Skalierbarkeit besteht bei einem proportionalen Verhältnis, d. h. bei einer Ressourcenverdopplung verdoppelt sich auch die Leistungsfähigkeit. Eine Verdopplung der Leistungsfähigkeit führt im Idealfall zur Halbierung von Ausführungszeiten und damit zur Verdopplung der innerhalb eines Zeitraums durch das System ausgeführten Aufgaben. Eine schlechte Skalierbarkeit ist dadurch gekennzeichnet, dass eine Leistungssteigerung einen

überproportionalen Einsatz von Ressourcen erfordert. Das Verhältnis kann aufgrund der theoretischen Komplexität von Algorithmen und Datenstrukturen polynomiell oder gar exponentiell werden.

Eine gute Skalierbarkeit zu erzielen wird umso schwieriger, je komplexer ein System ist. So stellt sich bei einem PrMS das Problem, dass Prozesse häufig interaktiv sind. Zudem interagieren die Benutzer gleichzeitig mit verschiedenen Prozessinstanzen, eine gute Interaktivität muss also nebenläufig gewährleistet sein. Gleichzeitig hängt die Leistungsfähigkeit eines PrMS auch von der Komplexität der ausgeführten Prozesse ab. Da die Benutzer die Prozesse nicht kennen müssen, können komplexe Prozesse leicht zu einer Verschlechterung der wahrgenommenen Leistungsfähigkeit führen. Dementsprechend wichtig ist es, hier durch zusätzliche Ressourcen sowie durch eine gute Skalierbarkeit geeignete Abhilfe schaffen zu können.

Bei adaptiven PrMS hängt die Leistungsfähigkeit zusätzlich von instanzspezifischen Änderungen ab. Deren Auswirkungen auf die Leistungsfähigkeit ähneln der Komplexität von Prozessen: die Benutzer wissen normalerweise nichts von der Existenz instanzspezifischer Änderungen, die Änderungen selbst haben aber Auswirkungen auf die Datenstrukturen und Funktionen und damit auch auf das Laufzeitverhalten des PrMS. Beim Entwurf und der Realisierung instanzspezifischer Änderungen spielt Skalierbarkeit eine entsprechend große Rolle.

Wie bereits im Kontext funktionaler Anforderungen (s. Abschnitt 3.2) diskutiert, muss ein PrMS erweiterbar sein. Dies ist umso wichtiger, je allgemeiner ein Softwaresystem ist; ein allgemein einsetzbares Softwaresystem ist gerade durch die Abwesenheit spezieller Funktionen gekennzeichnet. Auf eben diese speziellen Funktionen kann jedoch nicht immer verzichtet werden. Gerade für den Einsatz als Entwicklungsplattform (FA 12) sind **Erweiterungen** über Zusatzmodule (Plugins) an entscheidenden Stellen unverzichtbar (NA 3). Hierzu zählen beispielsweise spezielle Verteilungsalgorithmen für die Arbeitslistenverwaltung (FA 4) oder Eskalation (FA 5.1).

Spezielle Erweiterungen benötigen oftmals zusätzliche Daten, die in den allgemeinen Datenmodellen nicht existieren. So ist neben der **Erweiterbarkeit auf Dienstebene** (NA 3.1) auch eine **Erweiterbarkeit auf Modellebene** (NA 3.2) wichtig. Dabei lassen sich die verwendeten Datenmodelle einfach um zusätzliche Attribute ergänzen und somit zusätzliche Funktionen (etwa als Erweiterung eines Dienstes) realisieren. Beispielsweise können zusätzliche Kennzahlen an Prozessschritten (oder anderen Modellelementen) für die Verteilung von Arbeitsschritten oder für die Eskalation herangezogen werden. Die Erweiterbarkeit auf Modellebene erlaubt zudem, zusätzliche Informationen bei allgemeinen Funktionen zu nutzen, etwa für Entscheidungen innerhalb eines Prozesses oder für Prozessänderungen zur Laufzeit.

Die Erweiterbarkeit eines Softwaresystems wird durch **Einfachheit** (NA 4) bedeutend verbessert. Im Fall der Erweiterbarkeit betrifft dies in erster Linie die Einfachheit der Schnittstellen (NA 4.1), insbesondere der Schnittstellen für die Erweiterung des PrMS. Für eine Nutzung als Entwicklungsplattform (FA 12) sollten alle Schnittstellen möglichst einfach sein. Es hilft allerdings nicht, Anwendungsentwicklung durch eine umfangreiche Programmierschnittstelle zu vereinfachen, wenn dadurch die Komplexität auf die Anwendungsentwicklung verlagert wird. Die Einfachheit der Schnittstellen kann durch Einfachheit der Implementierung (NA 4.2) erreicht werden, dies ist aber nicht notwendig. Allerdings fördert eine möglichst einfach gehaltene Implementierung die Wartbarkeit (NA 7) und somit die Robustheit (NA 2) des Softwaresystems. Beispielsweise ist es in einem umfangreichen Softwaresystem zu vermeiden, dass Änderungen an einer Stelle unvorhergesehene Konsequenzen an anderen Stellen haben.

Im Fall eines adaptiven PrMS etwa sollten sich Änderungen an der Implementierung des Weiterschaltens eines Prozesses nicht auf Änderungsoperationen auswirken. Dementsprechend fördert die Einfachheit der Schnittstellen die Wartbarkeit und Robustheit sowohl des PrMS selbst als auch der Anwendungssysteme, die das PrMS als Plattform verwenden.

Gegen unvorhergesehene Konsequenzen bei Implementierungsänderungen hilft die strikte Modularität des Systems (**Separation of Concerns**), also die weitgehende **Unabhängigkeit** der einzelnen Teile eines Softwaresystems voneinander (NA 5). Wo Abhängigkeiten existieren, sollen diese über klar definierte (einfache) Schnittstellen etabliert werden. Ebenso wie Einfachheit verbessert dies wieder die Wartbarkeit (NA 7) und somit die Robustheit (NA 2). Module können komplett ausgetauscht und durch andere Implementierungen ersetzt werden. Ebenso wie bei der Einfachheit gibt es neben dem erläuterten internen Aspekt (NA 5.1) den externen Aspekt der Unabhängigkeit (NA 5.2). So sollte das Softwaresystem möglichst wenig Abhängigkeiten zu anderen Softwareplattformen besitzen, insbesondere wenn es sich um umfangreiche Plattformen handelt. Beispielsweise beschränkt es das Einsatzgebiet eines Softwaresystems erheblich, wenn es das DBMS eines bestimmten Herstellers und womöglich noch in einer bestimmten Version erfordert. Gerade im Fall von DBMS gibt es für diese Zwecke den SQL-Standard [ISO] als Schnittstelle.

Etwas widersprüchlich zur Unabhängigkeit ist die nicht-funktionale Anforderung der **Wiederverwendung und Redundanzvermeidung** (NA 6). Dazu gehört, dass Funktionen oder Variationen von ähnlichen Funktionen nicht mehrmals an verschiedenen Stellen im Softwaresystem existieren. Dies verbessert einerseits die Wartbarkeit (NA 7), da Fehler in solchen Funktionen nicht mehrfach behoben werden müssen. Andererseits verringert es die Unabhängigkeit der Module, wenn verschiedene Module eine Funktion gemeinsam nutzen. Sind relativ kleine Teile verschiedener Module sehr ähnlich, zahlt sich die Wiederverwendung gegenüber der verminderten Unabhängigkeit meistens aus. Dazu werden die wiederverwendeten Funktionen oft geeignet parametrisiert, so dass sie trotz Wiederverwendung modulspezifische Eigenheiten beibehalten können. Sollen größere Teile wiederverwendet werden, ist es ratsam, aus diesen ein neues Modul zu bilden. Dieses Modul kann dann von den ursprünglichen Modulen gemeinsam genutzt werden. Meistens sind in diesem Fall die wiederverwendeten Teile und damit das neue Modul auch funktional so abgeschlossen, dass sich dadurch die funktionale Modularisierung des Systems sogar verbessert (NA 5).

Die **Wartbarkeit** eines Systems (NA 7) ist eng mit anderen nicht-funktionalen Anforderungen verbunden. Wartbarkeit steht dafür, wie einfach es in einem System ist, Fehler zu finden und zu beheben. Insofern führt eine gute Wartbarkeit normalerweise zu einer guten Robustheit (NA 2). Fehlersuche und -behebung werden naheliegenderweise durch die Einfachheit (NA 4.2) und Abgeschlossenheit der Module (NA 5.1) sowie Redundanzvermeidung (NA 6) verbessert. Während unübersichtliche und große Module die Fehlersuche erschweren, müssen bei Fehlern in redundanten Funktionen die Fehler mehrfach behoben werden.

Was auf die Fehlersuche und -behebung zutrifft, gilt naheliegenderweise auch für die Änderung existierender Funktionen sowie das Hinzufügen neuer Funktionen. Dementsprechend hängt die **Flexibilität** eines Systems (im Sinne von Veränderbarkeit, NA 8) von denselben nicht-funktionalen Anforderungen ab: Einfachheit der Module (NA 4.2), Abgeschlossenheit der Module (NA 5.1) und Redundanzvermeidung (NA 6). Zusätzlich erhöht natürlich auch die Erweiterbarkeit (NA 3) die Flexibilität eines Systems. Die nicht-funktionale Anforderung der Flexibilität (NA 8) entspricht der funktionalen Anforderung nach Adaptivität (FA 9). Während die Adaptivität sich auf die vom System verwalteten Prozesse bezieht, ist bei der Flexibilität das System selbst Gegenstand der Anforderung.

Tabelle 3.2: Nicht-funktionale Anforderungen an ein adaptives PrMS

NA 1	Skalierbarkeit/Leistungsfähigkeit
NA 2	Robustheit
NA 3	Erweiterbarkeit
NA 3.1	Dienstebene
NA 3.2	Modellebene
NA 4	Einfachheit
NA 4.1	Einfachheit der Schnittstellen/der Verwendung
NA 4.2	Einfachheit der Implementierung
	→ Wartbarkeit (NA 7)
	→ Robustheit (NA 2)
NA 5	Unabhängigkeit/Modularität
	→ Wartbarkeit (NA 7)
	→ Robustheit (NA 2)
NA 5.1	Interne Unabhängigkeit
NA 5.2	Externe Unabhängigkeit
NA 6	Wiederverwendung/Redundanzvermeidung
	→ Wartbarkeit (NA 7)
	→ Unabhängigkeit/Modularität (NA 5)
NA 7	Wartbarkeit
NA 8	Veränderbarkeit/Flexibilität

Teil II

Konzepte, Architektur, technische Umsetzung

4

Speicherung und Verwaltung unveränderter Prozessinstanzen

Die Skalierbarkeit und Leistungsfähigkeit (NA 1) eines PrMS wird maßgeblich von der Speicherrepräsentation und damit den Datenstrukturen von Prozessvorlagen und -instanzen bestimmt. Diese Datenstrukturen untersuchen wir in den folgenden beiden Kapiteln. Wir betrachten in diesem Kapitel zunächst Prozessvorlagen und unveränderte Prozessinstanzen, bevor wir in Kapitel 5 detailliert auf die Speicherrepräsentation veränderter Prozessinstanzen eingehen. Beide Kapitel behandeln zuerst die Speicherrepräsentation auf logischer Ebene und gehen anschließend jeweils auf die physische Repräsentation der benötigten Datenstrukturen ein. Anhand häufig benötigter Operationen in einem PrMS diskutieren wir die Funktionsweise der Datenstrukturen: Weiterschalten (s. Abschnitt 2.2.3.2), Schemaevolution (s. Abschnitt 2.3.2.2) und Anfragen an Kollektionen von Prozessinstanzen.

Anfragen an Kollektionen von Prozessinstanzen dienen beispielsweise dazu, festzustellen, bei welchen Prozessinstanzen aktuell ein bestimmter Schritt ausgeführt wird. Solche Anfragen sind insofern kritisch, als Informationen von potentiell vielen Prozessinstanzen benötigt werden. Dies kann zum Einlagern von Prozessinstanzen in den Primärspeicher führen, was sowohl Rechenzeit als auch Speicherplatz kostet und zudem Auswirkungen auf andere Laufzeitfunktionen (z. B. Weiterschalten) hat. Weitere Beispiele für solche Anfragen betreffen Aktivitäten, die ein bestimmter Benutzer ausgeführt hat oder ausführen darf, sowie gezielte Abfragen auf Historiendaten.

Die theoretischen Betrachtungen der folgenden Kapitel werden in Kapitel 8 durch eine Simulation komplettiert. Dabei werden anhand prototypischer Implementierungen der Speicherbedarf der Datenstrukturen sowie die Laufzeit der Operationen gemessen.

Quantitative Vergleiche der Datenstrukturen sind für ein PrMS unabdingbar. Wir betrachten dabei Aufwände und den (Primär-)Speicherbedarf abhängig von elementaren Größen wie der Anzahl der Prozessinstanzen im System. Beim Speicherbedarf von Prozessinstanzen beschränken wir uns auf Struktur- und Zustandsdaten (M_{Total}). Im Prozess erzeugte Daten sind nicht Gegenstand der Betrachtung. Diese sind unabhängig von der Repräsentation der Kontrollflussstrukturen und Zustandsinformationen, d. h. sie stellen bezüglich des Speicherbedarfs lediglich einen konstanten Wert dar, von dem an dieser Stelle abstrahiert werden darf.

Des weiteren abstrahieren wir in den folgenden beiden Kapiteln von der Komplexität der Prozessvorlagen. Insgesamt machen wir folgende vereinfachende Annahmen:

Alle Prozessvorlagen besitzen dieselbe Anzahl an Knoten und sind vergleichbar aufgebaut. Das heißt, die Prozessvorlagen besitzen dieselbe Anzahl an parallelen (und alternativen) Verzweigungen. Die Verzweigungen wiederum besitzen gleich viele und gleich lange Zweige. Des weiteren nehmen wir an, dass pro Knoten dieselben Einträge in der Ausführungshistorie existieren, d. h. jeder Knoten durchläuft dieselben Zustände, und es werden dieselben Zustandsübergänge protokolliert.

Falls eine Speicherrepräsentation stärker als andere auf Änderungen eines Aspekt reagiert, z. B. wenn eine Verdopplung der parallelen Verzweigung den Speicherbedarf exponentiell statt linear erhöht, gehen wir darauf explizit ein. Die in den quantitativen Vergleichen verwendeten Bezeichnungen für Größen sind in Anhang D zusammengefasst.

4.1 Speicherrepräsentation

Startpunkt unserer Betrachtungen für die logische Repräsentation von Datenstrukturen unveränderter Prozessinstanzen bildet die Partitionierung von Prozessgraphen. Erfolgt diese Zerlegung sinnvoll, kann sie den Aufwand für das Ein-/Auslagern minimieren. Sie ist weitgehend unabhängig von der Repräsentation auf physischer Ebene, die wir später betrachten. Im Anschluss an die Partitionierung untersuchen wir weitere Aspekte zur Optimierung von Zugriffen durch Metadaten. Die angestellten Überlegungen zur Repräsentation der Speicherstrukturen sind für die Leistungsfähigkeit des PrMS wichtig, vergleichbar zu entsprechenden Bestrebungen in Datenbanksystemen. Dort hat beispielsweise die Wahl einer Implementierungsvariante von Indexen große Auswirkungen auf den Speicherbedarf und damit den effizienten Datenzugriff [SH99]. Die gewählte Realisierung hängt jedoch auch von funktionalen Anforderungen ab, etwa der Art der Zugriffe.

4.1.1 Partitionierung von Prozessmodellen

Neben einer kompakten Speicherrepräsentation der Daten schränkt die Partitionierung von Prozessgraphen den (Primär-)Speicherbedarf ein. Es wird jedoch nicht der absolute Speicherbedarf minimiert, sondern lediglich die Datenmenge, die zu einem bestimmten Zeitpunkt und für eine bestimmte Funktion (z. B. Weiterschalten) benötigt wird. Dies hat weitreichende Auswirkungen: Zum einen wird das Ein-/Auslagern der Daten der Prozessgraphen beschleunigt, zum anderen können zu einem Zeitpunkt die Daten einer größeren Anzahl von Prozessen im Primärspeicher sein, was wiederum die Häufigkeit des Ein-/Auslagerns von Daten verringert.

Wir unterscheiden zwischen *horizontaler* und *vertikaler Partitionierung* (s. Abb. 4.1). Die Unterteilung der Knotenmenge (*horizontale Partitionierung*) ermöglicht es, nur die Knoten einzulagern, die für die Ausführung einer PrMS-Funktion aktuell benötigt werden. Dies zahlt sich bei großen Prozessgraphen aus. *Vertikale Partitionierung* dagegen unterstützt das Einlagern ausgewählter Attribute (z. B. Knotenzustände). So reicht es im Kontext einer Schemaevolution beispielsweise aus, wenn das Zustandsattribut der Knoten im Primärspeicher bereitgestellt wird. Informationen zum Bearbeiter werden für diesen Anwendungsfall dagegen nicht benötigt, sofern die Bearbeiterzuordnung nicht geändert worden ist. Neben dem Primärspeicherbedarf verbessert eine angemessene Partitionierung auch das Laufzeitverhalten des PrMS, da insgesamt weniger Daten aus dem Sekundärspeicher ausgelesen und ggf. transformiert werden

müssen (s. Abb. 4.11). Allerdings wird die Implementierung komplizierter, da bei jedem Sekundärspeicherzugriff die jeweils benötigten Daten anzugeben sind. Nur so können für eine Anfrage die geeignete Partitionierung ausgewählt und die Datenmenge minimiert werden. Zusätzlich sollten Daten nachgefordert werden können, die beim erstmaligen Einlagern noch nicht angefordert worden sind. Dabei sollten bereits eingelagerte Daten nicht nochmals geladen werden, so dass das Einlagern auch bei Nachforderung nur die minimale Menge an Daten umfasst.

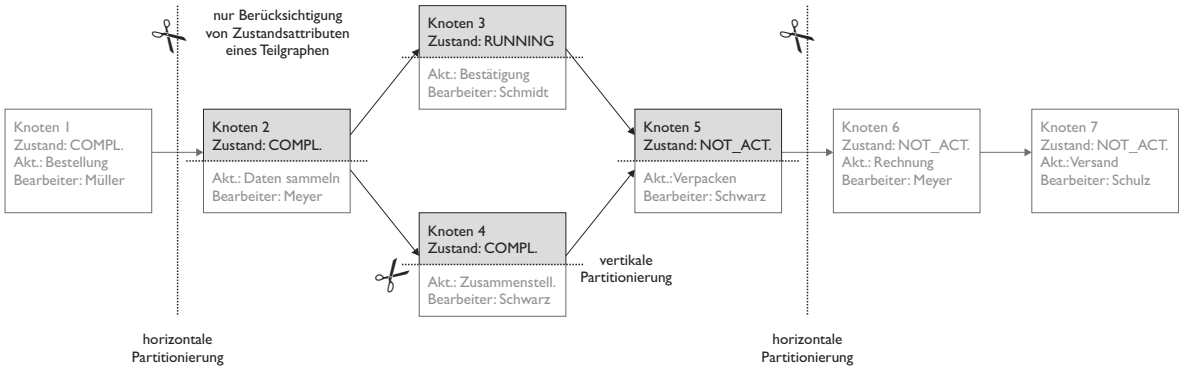


Abbildung 4.1: Partitionierung von Prozessgraphen

4.1.2 Prozessvorlagen

Prozessvorlagen stehen mit anderen Entitäten (z. B. Prozesstyp, Eltern-/Kindprozessvorlagen, Prozessinstanzen) in Beziehung. Da diese Beziehungen relativ selten benötigt werden, reicht es üblicherweise aus, sie ausschließlich im Sekundärspeicher zu hinterlegen. Kritisch dagegen ist die Speicherrepräsentation der Knoten und Kanten einer Prozessvorlage. Aufgrund der zahlreichen Attribute eines Knotens (z. B. auszuführendes Programm, Bearbeiterzuordnung Ein-/Ausgabedaten) ist es empfehlenswert, Knoten als eigenständige Entitäten (d. h. Objekte) zu repräsentieren. Dies können sowohl Objekte in objektorientierten Programmiersprachen als auch Datenstrukturen strukturierter Programmiersprachen (`STRUCT`/`RECORD`) sein. Die vertikale Partitionierung kann durch leichtgewichtige Stellvertreterobjekte (*Proxy*, [GHJV95]) realisiert werden oder indem entsprechende Attributwerte durch Platzhalter (z. B. `NULL`-Werte) ersetzt werden.

Zeitliche Abhängigkeiten, Reihenfolgen, und weitere Beziehungen werden über typisierte Kanten (z. B. Kontroll-, Schleifen-, Zeitkanten) realisiert. Abhängig von ihrem Typ kann eine Kante unterschiedliche Attribute besitzen, etwa Typinformation der Quell- und Zielknoten, Auswahlprädikate bei alternativen Verzweigungen (sog. Transitionsbedingungen) oder Minimal-/Maximalabstände bei Zeitkanten. Generell besitzen Kanten nur wenige (ca. 5) Attribute, weshalb sich mehrere konzeptionelle Realisierungsmöglichkeiten anbieten (s. Abb. 4.2):

- Explizite Kanten (Kanten als vollwertige Entitäten)
- Implizite Kanten (Knoten besitzen Attribute, deren Werte Referenzen auf Nachfolger-/Vorgängerknoten darstellen)
- Repräsentation als Adjazenzmatrix
- Repräsentation als Adjazenzliste.

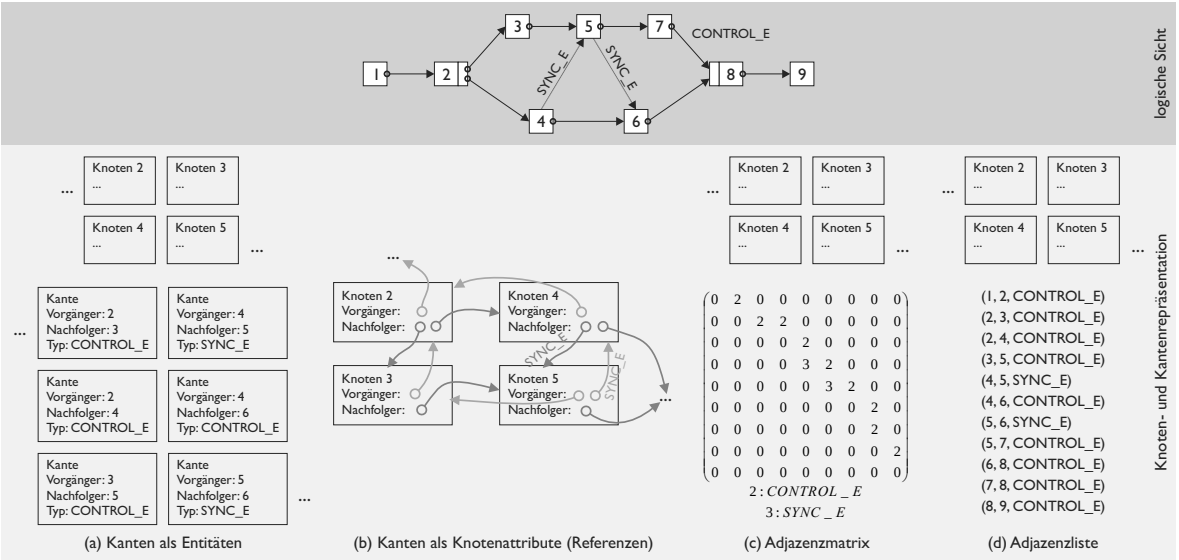


Abbildung 4.2: Prozessvorlage und entsprechende Kantenrepräsentationen

4.1.2.1 Variante KR-1: Kanten als Entitäten

Trotz weniger Attribute können Kanten als eigene Objekte oder Strukturen realisiert werden (s. Abb. 4.2 a). Dies ermöglicht eine flexible Handhabung, etwa indem alle Kantenobjekte in einer Menge – vergleichbar einer Adjazenzliste (Variante KR-4) – verwaltet werden. Sie können auch direkt als Objektreferenzen bei den jeweiligen Knoten (vgl. Variante KR-2) abgelegt sein. Insgesamt erfordert Variante KR-1 einen hohen Speicherbedarf für die physische Repräsentation im Primärspeicher. So beträgt in Java [jav] der initiale Speicherbedarf pro Objekt (der *Objektkopf* oder *Objektheader*) 8 Byte. Damit lohnt sich eine Repräsentation als eigenständige Entität nur, wenn die Kante viele Attribute oder Attribute „großer“ Datentypen besitzt. Beispielsweise belegt der Objektheader bei 32 Attributen eines kleinen Datentyps (32 * 1 Byte) oder bei 4 Attributen des größten Datentyps (4 * 8 Byte) immer noch 20 % des gesamten Speicherbedarfs des Objekts (40 Byte).

4.1.2.2 Variante KR-2: Kanten als Referenzen

In Programmiersprachen können Kanten direkt als Referenzen realisiert werden, die dann z. B. bei den Quellknoten hinterlegt werden (s. Abb. 4.2 b). Besser sind doppelte Referenzen, da Kanten sowohl vom Quell- zum Zielknoten als auch umgekehrt verfolgt werden können. Problematisch dabei ist, dass Kantentypen schwer zu realisieren sind, da Programmiersprachenreferenzen keine Attribute haben. Eine Möglichkeit besteht darin, die von einem Knoten ausgehenden Kanten entsprechend ihrer Richtung und ihres Typs jeweils in einer eigenen Menge zu hinterlegen. Dies erfordert aber viel Speicher, da dann bei jedem Knoten für jeden Kantentyp und für jede Richtung (Vorgänger und Nachfolger) eine Kantenmenge benötigt wird. Ein noch größeres Problem ist die Realisierung von Kantenattributen, da jeder Kantentyp verschiedene Attribute besitzen kann.

Bei zahlreichen Funktionen ist es nötig, die Vorgänger/Nachfolger eines Knotens (transitiv) zu bestimmen. Hierzu wird so lange über die Kanten iteriert, bis alle Kanten vom entsprechenden Knoten bis zum Start-/Endknoten der Vorlage festgestellt worden sind. Sind Kanten als

Knotenattribute realisiert, müssen dazu die entsprechenden Knotenobjekte eingelagert werden. Liegen Knoten und Kanten getrennt vor, kann hierauf verzichtet werden. Durch Kopplung von Knoten und Kanten vereinfacht sich die horizontale Partitionierung, die direkt durch Aufteilung der Knoten gegeben ist.

4.1.2.3 Variante KR-3: Adjazenzmatrix

Hier existiert für jeden Knoten einer Vorlage jeweils eine Zeile und eine Spalte, d. h. jeder Matrixeintrag steht für eine Relation zwischen zwei Knoten [CLRS09]. Über eine geeignete Codierung dieser Einträge können verschiedene Relations- und damit verschiedene Kantentypen hinterlegt werden (s. Abb. 4.2 c). Ein Vorteil liegt darin, dass alle Kanten ohne Aufwand sowohl vorwärts (ausgehend von Zeilen) als auch rückwärts (ausgehend von Spalten) verfolgt werden können, Quell- und Zielknoten einer Kante sind implizit durch die jeweilige Zeile und Spalte gegeben. Im Gegensatz zu Variante KR-2 kann dadurch ohne Knotenzugriffe über Kanten iteriert werden. Jedoch stellt sich wieder das Problem der Realisierung von Kantenattributen. Nachteilig ist zudem der Speicherbedarf, der quadratisch von der Anzahl der Knoten in der Prozessvorlage abhängt (s. Abb. 4.3).

4.1.2.4 Variante KR-4: Adjazenzliste

Jeder Eintrag einer Adjazenzliste steht für eine Kante zwischen zwei Knoten. Im Gegensatz zur Adjazenzmatrix sind die Quell-/Zielknoten explizit (s. Abb. 4.2 d), und es werden nur tatsächlich vorhandene Kanten berücksichtigt [CLRS09]. Dies hat insbesondere positive Auswirkungen auf den benötigten Speicherplatz (s. Abb. 4.3). Zudem können beliebige Attribute hinterlegt und sogar nachträglich ergänzt werden. Auch bei Iterationen sind keine Zugriffe auf Knoten nötig. Allerdings erfordert der Zugriff bei mehreren ausgehenden Kanten eines Knotens und insbesondere bei gerichteten Kanten in der Rückrichtung ein ineffizientes Durchsuchen der Adjazenzliste. Aufgrund des fehlenden wahlfreien Direktzugriffs ist die horizontale Partitionierung problematisch.

4.1.2.5 Fazit

Wichtigstes Kriterium für den Vergleich dieser vier Varianten ist der Speicherbedarf. Die Anzahl der Kanten in einer Vorlage entspricht ungefähr der Anzahl der Knoten. Realisiert man Kanten als eigene Entitäten (KR-1), ergibt sich für jede Vorlage nahezu eine Verdopplung des Speicherbedarfs. Den Speicherbedarf für die anderen Varianten (KR-2, KR-3 und KR-4) in Abhängigkeit von der Knotenanzahl zeigt Abb. 4.3. Auffällig ist der quadratische Speicherbedarf für die Realisierung mittels Adjazenzmatrix. Ab 10 Knoten übersteigt dieser die anderen beiden Varianten. Diese besitzen einen linearen Speicherbedarf, wobei die Adjazenzliste initial mehr Speicher benötigt als Variante KR-2. Dies liegt an der verwendeten Datenstruktur für die Realisierung der Liste. Dieser lohnt sich bereits ab 3 Knoten. Bei einer üblichen Vorlagengröße von 15-30 Knoten ist die Adjazenzliste hinsichtlich Speicherbedarf somit am besten.

Für eine endgültige Bewertung reicht der Vergleich des Primärspeicherbedarfs alleine aber nicht aus. Ebenfalls wichtig sind der wahlfreie Zugriff auf Kanten und die Möglichkeit der horizontalen Partitionierung. Die vertikale Partitionierung ist aufgrund der geringen Anzahl an Attributen bei Kanten unbedeutend. Dagegen ist die Realisierung der Attribute wichtig. So sollten sich Attribute später leicht ändern und ergänzen lassen, z. B. zwecks Einführung neuer Kantentypen bei Erweiterungen des Metamodells. Ein weiteres Kriterium ist die Unterstützung

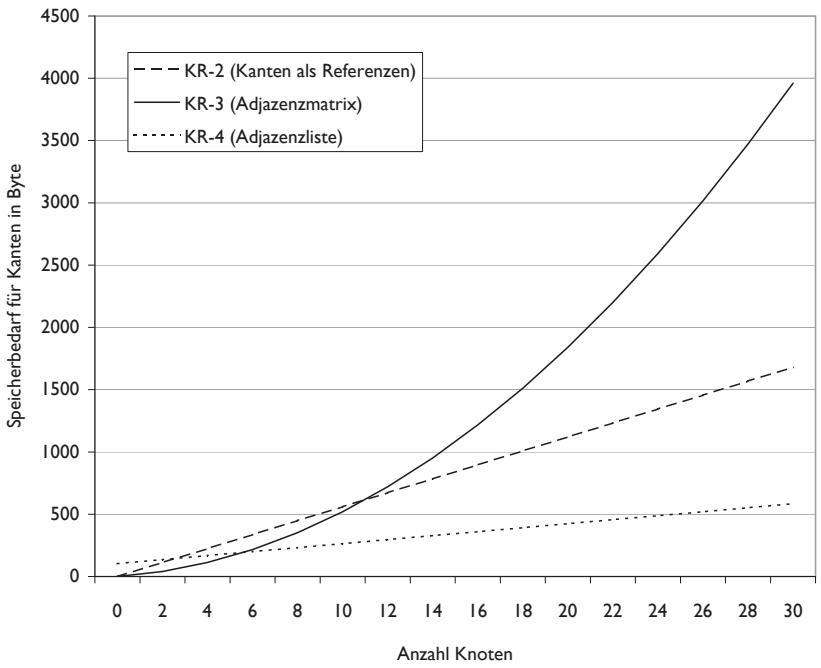


Abbildung 4.3: Primärspeicherbedarf für verschiedene Kantenrealisierungen

für Iterationen direkt über Kanten, ohne dass hierfür Knotenobjekte gelesen werden müssen. Dies ist beispielsweise für die Bestimmung von Vorgänger- und Nachfolgerbeziehungen zwischen Knoten nützlich, was häufig benötigt wird. Tabelle 4.1 bewertet die Varianten KR-2, KR-3 und KR-4 hinsichtlich dieser Aspekte.

Tabelle 4.1: Vergleich verschiedener Kantenrepräsentationen

	KR-2 Kanten als Referenzen	KR-3 Adjazenz-matrix	KR-4 Adjazenzliste
Speicherbedarf	+	--	++
Realisierung von Attributen	--	-	++
wahlfreier Zugriff	+	++	-
Partitionierung	++	+	-

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

Aufgrund der guten Realisierung von Attributen sind Adjazenzlisten (KR-4) die bevorzugte Realisierungsvariante – zumindest für Prozessmodelle mit expliziten Schleifenkanten. In diesem Fall ist eine topologische Sortierung der Knoten und damit auch der Kanten möglich. Diese erlaubt es, die Nachteile bezüglich des wahlfreien Zugriffs und der Partitionierung abzumildern. So kann bei jedem Knoten der Index der ersten ausgehende Kante des Knotens in der Adjazenzliste hinterlegt werden. Weitere von diesem Knoten ausgehende Kanten befinden sich direkt hinter dieser Indexposition in der Liste. Für einen effizienten Zugriff auf die Vorgängerbeziehung, d. h. die Rückrichtung, werden allerdings eine zusätzliche Adjazenzliste sowie die entsprechenden Indizes bei den Knoten benötigt. Neben dem effizienten Zugriff auf Kanten ermöglichen diese Indizes darüber hinaus die horizontale Partitionierung einer

Adjazenzliste. Hierbei wird ausgenutzt, dass die Indizes direkt bei den Knoten abgelegt sind. So geben die Indizes des ersten und des letzten Knotens der Partition jeweils den Teil der Adjazenzliste an, der Bestandteil der Partition ist. Die Teile der Adjazenzlisten vor und nach diesen Indizes werden für diese Partition nicht benötigt.

4.1.3 Prozessinstanzen

Bisher wurden nur statische Prozessdaten betrachtet. Ein größeres Datenvolumen fällt infolge der großen Anzahl von Prozessinstanzen an (bis zu mehrere zehntausend). Deshalb ist deren Speicherrepräsentation wesentlich kritischer hinsichtlich der Leistungsfähigkeit eines PrMS als diejenige von Prozessvorlagen. Eine Prozessinstanz besteht (logisch) aus der Struktur der zugrundeliegenden Vorlage (sog. *Instanzvorlage*), angereichert um Zustandsinformation (s. Abb. 2.4). Beides sind voneinander unabhängige Aspekte.

4.1.3.1 Instanzvorlagen

Die Repräsentation von Prozessvorlagen wurde in Abschnitt 4.1.2 betrachtet. Diese Überlegungen sind auf Instanzvorlagen übertragbar. Zusätzlich stellt sich die Frage, ob für jede Prozessinstanz ein kompletter Graph gespeichert werden soll (*Vorlagenkopie*) oder nicht (*Vorlagenreferenz*).

Variante VG-1: Vorlagenkopie Bei Vorlagenkopien wird bei der Erzeugung einer Prozessinstanz eine komplette Kopie der entsprechenden Prozessvorlage erstellt und um Attribute zur Aufnahme von Zustandsdaten ergänzt (z. B. Knotenzustände, s. Abb. 4.4 a). Damit entspricht die Speicherrepräsentation einer Prozessinstanz unmittelbar der konzeptionellen Sicht. Beim Weiterschalten können die Nachfolger eines beendeten Knotens direkt im Instanzobjekt bestimmt werden. Vorlagenkopien sind problematisch bei Schemaevolution, da jede Instanzvorlage zunächst einmal unabhängig von der ursprünglichen Prozessvorlage ist. Wird die ursprüngliche Prozessvorlage geändert, müssen zusätzlich zu den ohnehin aufwendigen Verträglichkeitsprüfungen und Zustandsanpassungen für jede Prozessinstanz auch strukturelle Änderungen an der Instanzvorlage vorgenommen werden. Diese sind bei jeder von der Änderung betroffenen (unveränderten) Prozessinstanz gleich. Die Gesamtkosten ($C_{SchemaEvo}$) für eine Schemaevolution (s. Abschnitt 2.3.2.2) betragen mit Vorlagenkopien:

Formel 4.1 (Gesamtkosten für eine Schemaevolution mit Vorlagenkopien)

$$\begin{aligned} C_{SchemaEvo} &= (C_{StructCompl} + C_{StateCompl}) * N_{AllInsts} \\ &+ C_{StructAdapt} * N_{ComplInsts} + C_{StateAdapt} * N_{ComplInsts} \end{aligned}$$

Diese Formel verwendet folgende Größen:

$C_{SchemaEvo}$: Gesamtkosten für Schemaevolution einer Prozessvorlage

$C_{StructCompl}$: Kosten für die Überprüfung der Verträglichkeit bezüglich der Struktur einer Prozessinstanz; diese hängen von der Art der Vorlagenänderung ab.

$C_{StateCompl}$: Kosten für die Überprüfung der Verträglichkeit bezüglich des Zustands einer Prozessinstanz; sie hängen von der Art der Vorlagenänderung und dem momentanen Zustand der Prozessinstanz ab.

$C_{StructAdapt}$: Kosten für die strukturellen Anpassungen einer Instanzvorlage; diese hängen von der Art der Vorlagenänderung ab, sie sind aber für jede Vorlagenkopie gleich.

$C_{StateAdapt}$: Kosten für die Anpassungen des Zustands einer Prozessinstanz; diese hängen von der Art der Vorlagenänderung und dem momentanen Zustand der Prozessinstanz ab.

$N_{AllInsts}$: Anzahl aller Prozessinstanzen der geänderten Prozessvorlage

$N_{ComplInsts}$: Anzahl der Prozessinstanzen, zu deren Zustand die Vorlagenänderungen verträglich sind und die dementsprechend migriert werden können.

Variante VG-2: Vorlagenreferenz Beim Instanzieren mit Vorlagenreferenzen werden lediglich die Datenstrukturen für Zustandsdaten (z. B. eine Liste aller Knotenzustände) sowie eine Referenz auf die instanzierte Prozessvorlage (s. Abb. 4.4 b) angelegt.

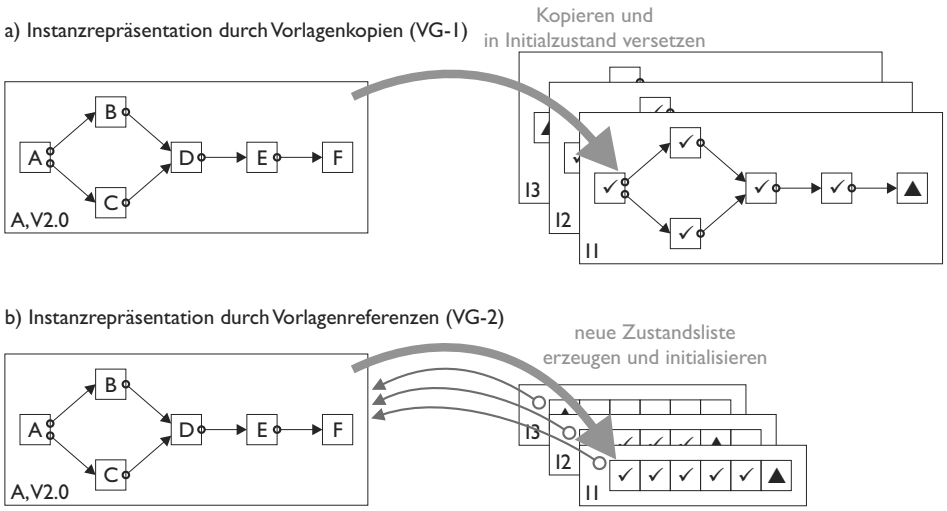


Abbildung 4.4: Erzeugung neuer Instanzobjekte

Bei Variante VG-2 existiert keine Kopie der Vorlagendaten. Die Bestimmung von Nachfolger-/Vorgängerknoten (z. B. beim Weiterschalten im Prozess) kann in diesem Fall nicht direkt

im Instanzobjekt erfolgen, sondern nur indirekt über die zugrundeliegende Prozessvorlage, was einige wenige zusätzliche Speicherzugriffe erfordert. Mit Vorlagenreferenzen ist die Propagation von Änderungen bei Schemaevolution wesentlich effizienter durchführbar. So reicht es aus, die strukturellen Änderungen lediglich einmal auf der Prozessvorlage durchzuführen und anschließend die Verträglichkeit der Änderungen mit den entsprechenden Prozessinstanzen zu überprüfen [RRD02, RRD04c, RRD04a]. Bei Prozessinstanzen, deren Zustand zu den Änderungen verträglich ist, müssen dann nur die Vorlagenreferenz geändert und die Knotenzustände angepasst werden. Im Gegensatz zu Variante VG-1 sind keine strukturellen Änderungen für jede Prozessinstanz nötig. Die Gesamtkosten ($C_{SchemaEvo}$) für eine Schemaevolution (s. Abschnitt 2.3.2.2) sind dementsprechend um den Summand $C_{StructAdapt} * N_{ComplInsts}$ geringer gegenüber Variante VG-1.

Tabelle 4.2: Vergleich verschiedener Vorlagengraphen

	VG-1 Vorlagen-kopien	VG-2 Vorlagen-referenz
Speicherbedarf	-	++
Schemaevolution	--	++
Zugriff	+	O
Partitionierung	+	++

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

Fazit Vergleicht man die beiden Varianten (s. Tabelle 4.2), ist VG-2 vorzuziehen. Vorlagenkopien (VG-1) sind aufwendiger zu erzeugen, und die Strukturinformation ist redundant bei jeder Prozessinstanz abgelegt, was zu großem Speicherbedarf führt. Variante VG-2 dagegen benötigt weniger Speicher bei minimaler Verschlechterung ($O(1)$) des Zugriffsverhaltens. Daneben lassen sich Änderungen bei einer Schemaevolution bei Variante VG-2 viel effizienter durchführen als bei Variante VG-1. Auch der Aufwand für das Ein-/Auslagern der Prozessinstanzen ist deutlich geringer, da Instanzobjekte wesentlich kleiner sind. Vorlagenkopien werden z. B. in den PrMS Chameleon [MS02] und ProMInanD [KRW90, Kar94] verwendet, Vorlagenreferenzen bei Staffware [tib]. In der Regel sind nur wenige Prozessvorlagen im System gleichzeitig instanziert (s. Tabelle 6.1). Somit können diese dauerhaft im Primärspeicher gehalten werden und müssen nicht ein-/ausgelagert werden, was sich bei Variante VG-2 vorteilhaft auswirkt.

4.1.3.2 Zustände von Prozessinstanzen

Neben struktureller Information besitzt eine Prozessinstanz einen definierten Zustand (s. Abschnitt 2.2.3.1). Abb. 4.5 zeigt verschiedene Realisierungsmöglichkeiten für die Zustandsdaten. Obwohl das Zustandsmodell explizite Kontenzustände definiert (s. Abschnitt 2.2.3.1) werden hierfür keine separaten Datenstrukturen notwendig; die Ausführungshistorie reicht für deren Rekonstruktion aus (s. Abb. 4.5 links).

Der aktuelle Zustand einer Prozessinstanz spiegelt sich in den Zuständen ihrer einzelnen Aktivitäten wider. Ihre Repräsentation kann mittels expliziter Markierungen erfolgen, wobei jeder Knoten eine eigene Markierung zugewiesen bekommt [RRD04c]. Bei impliziten Markierungen dagegen werden nur die aktiven Knoten verwaltet, also die zur Bearbeitung anstehenden oder in Bearbeitung befindlichen Aktivitäten. Letzteres entspricht der Tokensemantik von

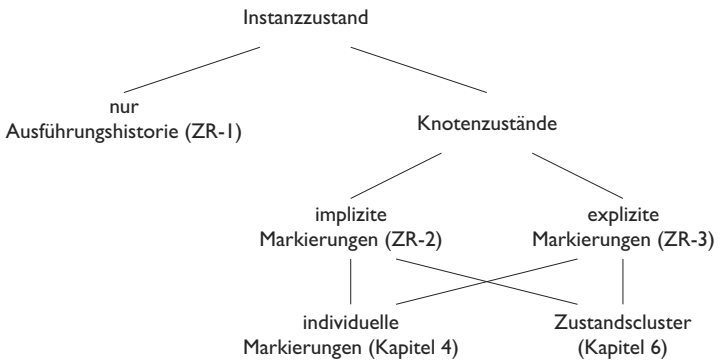


Abbildung 4.5: Klassifikation der Realisierungsmöglichkeiten von Zustandsdaten

Petrinetzen [Bau96, vdAVH04]. Die Zustände nicht-aktiver Knoten können hieraus jedoch nur teilweise rekonstruiert werden.

Für die Repräsentation der Markierungen bei Knoten gibt es zwei Möglichkeiten: Sie können entweder für jede Prozessinstanz individuell verwaltet werden, oder es werden Zustandscluster gebildet. Bei Zustandsclustern werden Prozessinstanzen im selben Zustand zusammengefasst und der entsprechende Zustand nur einmal für alle Prozessinstanzen realisiert. Im folgenden werden zunächst drei alternative Repräsentationsmöglichkeiten für individuelle Instanzmarkierungen untersucht und verglichen. Auf Zustandscluster wird in Kapitel 6 eingegangen.

Variante ZR-1: Repräsentation von Zuständen durch Ausführungshistorien Um die Ausführung eines Prozesses jederzeit nachvollziehen zu können, werden sämtliche Zustandsübergänge inklusive weiterer Informationen (z. B. Zeitpunkt des Zustandsübergangs) in einer Ausführungshistorie protokolliert (s. Abschnitt 2.2.1.3). Aus dieser lassen sich aktuelle sowie frühere Zustände einer Prozessinstanz (und damit auch deren Knotenzustände) rekonstruieren. Dieser Ansatz ist für das Weiterschalten im Kontrollfluss ausreichend. Beim Beenden einer Aktivität müssen, außer dem ohnehin notwendigen Eintrag in der Historie keine weiteren Daten manipuliert werden. Die Neubewertung des Zustands nachfolgender Aktivitäten ist jedoch aufwendig. Hierzu werden zuerst alle Nachfolgerknoten des beendeten Knotens ermittelt. Anschließend werden von diesen Knoten sämtliche Vorgängerknoten ermittelt. Abhängig von deren Zuständen werden die Zustände der Nachfolgerknoten gesetzt. Der Zustand eines Knotens ergibt sich durch den letzten Eintrag des Knotens in der Ausführungshistorie. Da die Zustände von potentiell vielen Knoten ermittelt werden müssen und die Ausführungshistorie umfangreich ist, ist das Weiterschalten mit Ausführungshistorien aufwendig. Dies gilt analog für die Zustandsrekonstruktion für Verträglichkeitsprüfungen bei Schemaevolution. Der Zustand muss dabei für jeden zu überprüfenden Knoten einer betroffenen Prozessinstanz ermittelt werden. Sind anschließend Zustandsanpassungen nötig, müssen diese aus Gründen der Konsistenz auch in der Historie vermerkt werden. Insgesamt ist Schemaevolution mit Variante ZR-1 damit sehr aufwendig.

Ausführungshistorien benötigen potentiell viel Speicherplatz. Einerseits ist jeder Historieneintrag relativ groß, da er neben dem Zustandsübergang und dem betroffenen Knoten unter anderem einen Zeitstempel sowie den für den Zustandsübergang verantwortlichen Bearbeiter enthält. Andererseits gibt es für jeden Knoten mehrere Einträge in der Historie (schlimmstenfalls für jeden Zustandsübergang einen), wobei die Anzahl von den durch das Prozessmetamodell definierten Zustände und Zustandsübergängen abhängt. Beim Auftreten

von Schleifen im Prozess kommen weitere Einträge für die Knoten innerhalb der Schleife hinzu (s. Abb. 4.6 a).

Aufgrund der großen Datenmengen sind Historien nur in Verbindung mit horizontaler und vertikaler Partitionierung (s. Abschnitt 4.1.1) sinnvoll. Bei horizontaler Partitionierung kommt allerdings erschwerend hinzu, dass Ausführungshistorien chronologisch sortiert sind. Dadurch kann es bei Schritten in parallelen Zweigen zu Verschränkungen der Historieneinträge kommen, d. h. nicht alle Einträge eines Knotens stehen hintereinander. Eine horizontale Partitionierung erfordert aber, dass die Daten verschiedener Knoten getrennt werden können.

Die Verschränkung von Historieneinträgen verhindert auch den effizienten Zugriff auf Historiendaten. Zwar lassen sich die Zustände der wichtigsten Knoten relativ schnell rekonstruieren, da diese am Ende einer chronologisch sortieren Historie stehen. Allerdings lassen sich andere Knotenzustände nur durch zusätzliche Indexstrukturen, vergleichbar denen bei Adjazenzlisten (s. Abschnitt 4.1.2.4), effizient bestimmen. Sind die Historieneinträge zusätzlich verschränkt, kann der Knotenzustand nur durch sequentielle Iteration der Historieneinträge nach dem referenzierten Historieneintrag erfolgen.

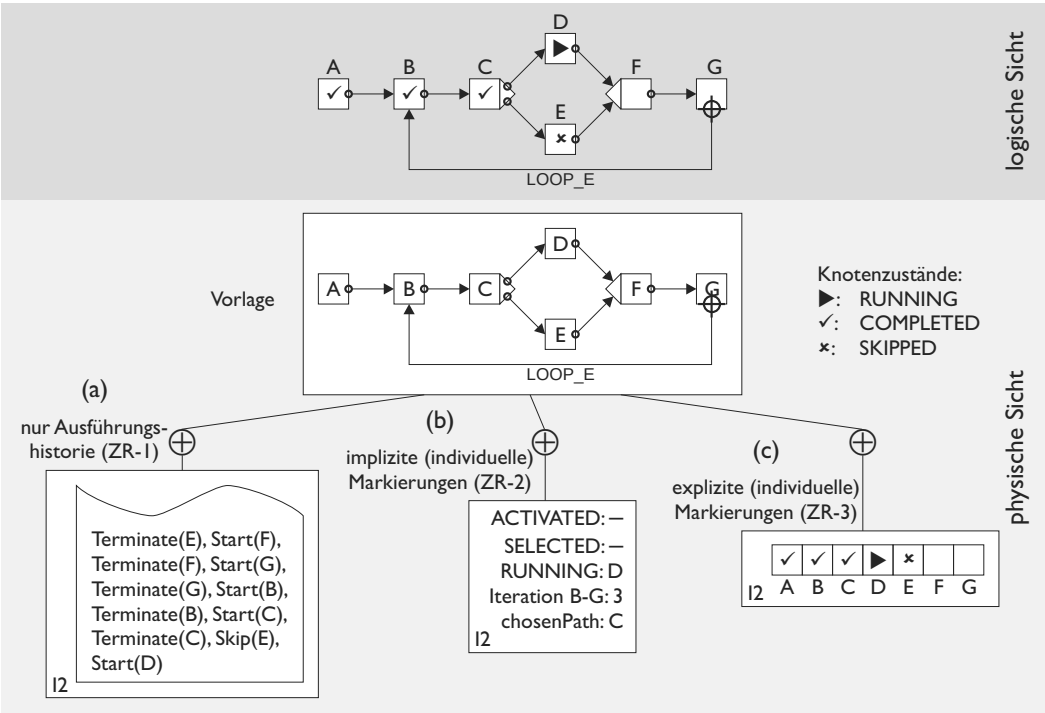


Abbildung 4.6: Realisierungsmöglichkeiten für Knotenzustände

Variante ZR-2: Implizite Markierungen Bei impliziten Markierungen besitzen nur aktive Knoten eine Markierung, also alle aktuell ausgeführten oder zur Ausführung anstehenden Aktivitäten. Die Rekonstruktion der Zustände anderer Knoten erfolgt über die Vorgänger-/Nachfolgerbeziehung zu einem aktiven Knoten. Da es in einer Prozessvorlage keine isolierten Knoten gibt, existiert diese Beziehung stets. Vorgängerknoten sind abgeschlossen oder wurden abgewählt, Nachfolgerknoten stehen zukünftig noch zur Ausführung an. Nicht möglich ist die Zustandsrekonstruktion bei Metamodellen mit impliziten Schleifen (z. B. Petrinetze), bei denen es keinen speziellen Kantentyp für Schleifen gibt. Stattdessen werden Schleifen einfach

durch das entsprechende Setzen von normalen (Vorwärts-)Kanten realisiert. Hierdurch lassen sich jedoch keine eindeutigen Vorgänger-/Nachfolgerbeziehungen bestimmen: Ein Knoten innerhalb eines Kantenzyklus ist sowohl Nachfolger als auch Vorgänger von sich selbst sowie von allen anderen Knoten des Zyklus. Eine eindeutige Vorgänger-/Nachfolgerbeziehung ist für fortschrittliche Funktionalität (Schemaevolution, instanzspezifische Änderungen) aber unabdingbar. Bei expliziten Schleifen wird zur Zustandsrekonstruktion darüber hinaus ein Iterationszähler mit verwaltet (s. Abschnitt 2.2.3.2). Zudem werden die bei alternativen Verzweigungen gewählten Pfade gespeichert (s. Abb. 4.6 b). Eine detaillierte Beschreibung des Rekonstruktionsalgorithmus findet sich in Anhang C.1.

Der Laufzeitaufwand der Zustandsrekonstruktion bei impliziten Markierungen ist linear zur Anzahl der Knoten einer Prozessinstanz, sofern keine Historienzugriffe erfolgen. Für einfache Funktionen ist dies ausreichend, jedoch nicht für die Verträglichkeitsprüfung bei Schemaevolution. Hier ist der Algorithmus für jeden von der Änderung betroffenen Knoten jeder Prozessinstanz einmal auszuführen, was die Laufzeit $O(a * n * m)$ ergibt. Diese hängt von der Anzahl der aktiven Knoten (a), der Anzahl der von der Änderung betroffenen Knoten (n) sowie der Anzahl der zu untersuchenden Prozessinstanzen (m) ab. Bei einer Schemaevolution können viele Prozessinstanzen betroffen sein, was dementsprechend zu einem großen Faktor m führt.

Variante ZR-2 benötigt insgesamt wenig Speicher. Dies beschränkt sich auf die Zustände aktiver Knoten (insgesamt ca. 3-4 Byte pro Prozessinstanz), gewählte Pfade bei alternativen Verzweigungen (ca. 3 Byte pro Prozessinstanz) sowie pro Schleife einen Zähler (ca. 2 Byte pro Prozessinstanz). Implizite Zustände bieten trotzdem nur eine geringfügig schlechtere Unterstützung für das Weiterschalten im Vergleich zu Variante ZR-1. Beim Beenden von Aktivitäten müssen diese lediglich aus der Liste der aktiven Knoten der Prozessinstanz entfernt und ggf. die nach dem Weiterschalten neu aktivierten Knoten hinzugefügt werden. Beim Weiterschalten ist, ähnlich wie bei ZR-1, die Rekonstruktion von Zuständen nötig, was allerdings weniger Speicher benötigt. Für Schemaevolution ist ZR-2 schlecht geeignet. Der Laufzeitaufwand der Zustandsrekonstruktion wird aufgrund der Abhängigkeit sowohl von der Knotenanzahl als auch der Anzahl der Prozessinstanzen sehr groß.

Variante ZR-3: Explizite Markierungen Jeder Aktivität – auch wenn sie bereits abgeschlossen, abgewählt oder noch nicht ausgeführt worden ist – wird eine explizite Markierung zugewiesen, wodurch der Rekonstruktionsaufwand für Knotenzustände wegfällt. Auch die Entscheidungen bei alternativen Verzweigungen und Schleifen brauchen nicht verwaltet zu werden, da sich diese aus den expliziten Markierungen ergeben. Für explizite Schleifen werden die Knotenzustände des Schleifenkörpers vor einem erneuten Schleifendurchlauf zurückgesetzt. Die Markierungen spiegeln damit nur den jeweils aktuellen bzw. letzten Schleifendurchlauf wider. Vorherige Iterationen können aus der Ausführungshistorie rekonstruiert werden, was aber nur für wenige Anwendungsfälle nötig ist.

Da die Menge der möglichen Knotenzustände bei allen Prozessmetamodellen begrenzt ist (z. B. 8 mögliche Zustände pro Knoten beim ADEPT-Metamodell, s. Abschnitt 2.2.3.1), sind wenige Bit pro Knoten für die Speicherung des Zustands ausreichend (insgesamt ca. 20–40 Byte pro Prozessinstanz, abhängig von der Knotenanzahl). Dies ist mehr als bei Variante ZR-2, jedoch bedeutend weniger als eine vollständige Ausführungshistorie benötigt. Variante ZR-3 ist für das Weiterschalten gut geeignet. Im Gegensatz zu den beiden anderen Varianten müssen zwar mehr Markierungen geändert werden, jedoch ist die Bestimmung von Knotenzuständen

einfach, da sie nicht rekonstruiert werden müssen. Dasselbe gilt in Bezug auf Schemaevolution. Verträglichkeitsprüfungen sind effizient ausführbar, die Zustandsanpassungen erfolgen direkt bei den entsprechenden Knoten. Variante ZR-3 benötigt zudem wesentlich weniger Primärspeicher als Variante ZR-1, was ein schnelles Ein-/Auslagern aller Zustandsdaten ermöglicht.

Auch für instanzübergreifende Anfragen bezüglich Zustandsinformationen sind explizite Markierungen gut geeignet. Da die Zustände nicht rekonstruiert werden müssen, können sie direkt aus den einzelnen Prozessinstanzen ausgelesen werden, was meistens direkt im Sekundärspeicher erfolgen kann und somit keine Einlagerung notwendig wird.

Fazit Tabelle 4.3 bewertet die Zustandsrepräsentationen. Explizite Markierungen haben einige Vorteile, allerdings benötigen implizite Markierungen wesentlich weniger Speicher und stellen somit eine sinnvolle Alternative dar.

Tabelle 4.3: Vergleich verschiedener Zustandsrepräsentationen

	ZR-1 Ausführungs-historie	ZR-2 Implizite Markierungen	ZR-3 Explizite Markierungen
Speicherbedarf	-	++	+
Zugriff/Rekonstruktion	-	O	++
Partitionierung	O	O	++

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

4.1.4 Quantitativer Vergleich der Prozessinstanzrepräsentationen

In diesem Abschnitt vergleichen wir die verschiedenen Repräsentationen von Prozessinstanzen qualitativ bezüglich ihres Speicherbedarfs. Der Speicherbedarf hängt von verschiedenen Größen ab (s. Tabelle 4.4), wobei für jede Repräsentation unterschiedliche Größen relevant sind.

Für Variante VG-1 (Vorlagenkopie) beschränken wir uns auf die Kombination mit expliziten Markierungen (ZR-3) für die Zustandsrepräsentation. Ausführungshistorien (ZR-1) und implizite Markierungen (ZR-2) sind mit Vorlagenkopien nicht sinnvoll, da beide Varianten für die Zustandsrepräsentation die explizit vorhandene Graphstruktur nicht nutzen. Im Gegensatz dazu werden explizite Markierungen (ZR-3) direkt in der Vorlagenkopie hinterlegt, wodurch sich vollständige und in sich abgeschlossene Datenstrukturen für Prozessinstanzen ergeben.

4.1.4.1 Vorlagenkopien mit expliziten Markierungen

Der Speicherbedarf von Vorlagenkopien mit expliziten Markierungen (VG-1 und ZR-3) ergibt sich aus der Anzahl der Prozessinstanzen im System $N_{Instances}$ multipliziert mit dem Speicherbedarf einer Prozessinstanz. Der Speicherbedarf einer Prozessinstanz wiederum berechnet sich aus ihrer Knotenanzahl N_{Nodes}, die der Knotenanzahl der zugehörigen Prozessvorlage entspricht, multipliziert mit der Summe aus Speicherbedarf für die statischen Informationen eines Knotens M_{Node}, seinen Zustandsinformationen $M_{NodeState}$ sowie seinen weiteren dynamischen Informationen $M_{InstanceNode}$. Die statischen Informationen gehören zum Knoten der zugehörigen Prozessvorlage und umfassen beispielsweise die eindeutige Kennung eines Knotens oder eine Bearbeiterzuordnungsregel. Die Zustandsinformationen sind durch die expliziten Markierungen gegeben, während die weiteren dynamischen Informationen, z. B. der

Tabelle 4.4: Relevante Größen für Speicherbedarf unveränderter Prozessinstanzen

$M_{ActiveNodes}$: Speicherbedarf (pro Prozessinstanz) für Informationen zur Verwaltung aller aktiven Knoten
Bei Variante ZR-2 (s. Abschnitt 4.1.3.2) werden nur die aktiven Knoten einer Prozessinstanz sowie einige weitere Informationen zur vollständigen Rekonstruktion des Zustands einer Prozessinstanz gespeichert.

M_{Edge}: Speicherbedarf für eine Kante einer Prozessvorlage, unabhängig von ihrer Repräsentation

$M_{ExecEntry}$: Speicherbedarf für einen Eintrag in der Ausführungshistorie

$M_{InstanceNode}$: Speicherbedarf für die Repräsentation eines Instanzknotens

$M_{IterationCount}$: Speicherbedarf für den Iterationszähler für einen Instanzknoten

M_{Node}: Speicherbedarf für einen Knoten einer Prozessvorlage

$M_{NodeState}$: Speicherbedarf für die Repräsentation eines Knotenzustands

M_{Total}: Gesamter Speicherbedarf für alle Prozessvorlagen- und Prozessinstanzdaten

N_{Edges}: Anzahl der Kanten pro Prozessvorlage

$N_{ExecEntriesPerNode}$: Anzahl der Einträge in der Ausführungshistorie pro Knoten
Da ein Knoten mehrere Zustände durchläuft werden mehrere Einträge in der Ausführungshistorie protokolliert.

$N_{Instances}$: Anzahl aller Prozessinstanzen im System unabhängig von der jeweiligen Prozessvorlage

N_{Nodes}: Anzahl der Knoten pro Prozessvorlage

$N_{Templates}$: Anzahl aller Prozessvorlagen im System

Bearbeiter eines Knotens, Zeitinformationen oder Iterationszähler, zum Knoten der Prozessinstanz gehören. Aufgrund der Vorlagenkopie kommt noch der Speicherplatz für die Kanten der Prozessvorlage hinzu ($N_{Edges} * M_{Edge}$).

Formel 4.2 (Speicherbedarf für Vorlagenkopien mit expliziten Markierungen)

$$\begin{aligned} M_{Total} &= N_{Instances} * (N_{Nodes} * (M_{Node} + M_{NodeState} + M_{InstanceNode}) \\ &+ N_{Edges} * M_{Edge}) \end{aligned}$$

4.1.4.2 Vorlagenreferenzen mit Ausführungshistorie

Der bei der Verwendung von Vorlagenreferenzen (VG-2) benötigte Speicher ergibt sich aus der Anzahl der Prozessvorlagen im System $N_{Templates}$ multipliziert mit dem Speicherbedarf für eine Prozessvorlage. Dieser wiederum entspricht der Knotenanzahl N_{Nodes} multipliziert mit der Speichergröße der statischen Knoteninformationen M_{Node}.

Die Zustandsdaten und weitere dynamische Information (z. B Bearbeiter eines Prozessschritts) werden durch die Ausführungshistorien (ZR-1) zur Verfügung gestellt. Der hierfür

benötigte Speicherplatz berechnet sich aus der Anzahl der verwalteten Prozessinstanzen $N_{Instances}$ multipliziert mit der Größe einer Ausführungshistorie. Wir nehmen der Einfachheit halber an, dass alle Historieneinträge gleich viel Speicherplatz M_{Entry} benötigen. Die Anzahl der Historieneinträge entspricht der Anzahl der Knoten N_{Nodes} multipliziert mit der Anzahl der Historieneinträge pro Knoten $N_{EntriesPerNode}$. Diese Anzahl ergibt sich aus den Zustandsübergängen, die in der Ausführungshistorie protokolliert werden. Dies unterscheidet sich von der Gesamtzahl der Zustände im Metamodell N_{States}, da ein Knoten im Normalfall nicht alle im Modell definierten Zustände durchläuft. So nimmt ein Knoten nur einen der Endzustände `COMPLETED`, `FAILED` oder `SKIPPED` ein und erfährt dementsprechend auch nur einen entsprechenden Zustandsübergang.

Formel 4.3 (Speicherbedarf für Vorlagenreferenzen mit Ausführungshistorie)

$$\begin{aligned} M_{Total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Instances} * N_{Nodes} * N_{ExecEntriesPerNode} * M_{ExecEntry} \end{aligned}$$

4.1.4.3 Vorlagenreferenzen mit impliziten Markierungen

Wie bei Vorlagenreferenzen mit Ausführungshistorie müssen bei Vorlagenreferenzen mit impliziten Markierungen (VG-2 und ZR-2) die Vorlagenobjekte nur einmal im Speicher bereitgestellt werden. Der Speicherbedarf berechnet sich wieder aus der Anzahl der Prozessvorlagen multipliziert mit der Anzahl der Knoten multipliziert mit dem benötigten Knotenspeicherplatz ($N_{Templates} * N_{Nodes} * M_{Node}$). Zusätzlich wird je Prozessinstanz entsprechender Speicherplatz für die impliziten Markierungen benötigt. Dieser hängt von der Struktur der zugrundeliegenden Vorlage ab (z. B. Anzahl und Komplexität paralleler und alternativer Verzweigungen). Bei Variante ZR-2 hängt der Speicherbedarf, anders als bei den anderen Repräsentationsvarianten, nicht von der Knotenanzahl ab.

Vereinfachend wird in der Berechnung ein konstanter Speicherbedarf für diese Informationen angenommen ($M_{ActiveNodes}$). Zusätzlich werden auch hier für die sog. *Instanzknoten* – im Gegensatz zu Vorlagenknoten – dynamische Informationen benötigt ($M_{InstanceNode}$). Bei impliziten Markierungen werden Iterationszähler im Vergleich mit anderen Varianten nur einmal pro Prozessinstanz und Schleife hinterlegt, sie sind somit bei $M_{ActiveNodes}$ berücksichtigt und müssen vom Speicherbedarf eines Instanzknotens abgezogen werden ($M_{InstanceNode} - M_{IterationCount}$). Multipliziert man dies mit der Anzahl der Knoten einer Prozessinstanz (N_{Nodes}), ergibt sich der zusätzliche Speicherbedarf pro Prozessinstanz, der für zusätzliche dynamische Daten (z. B. Zeitinformationen) benötigt wird.

Formel 4.4 (Speicherbedarf für Vorlagenreferenzen mit impl. Markierungen)

$$\begin{aligned} M_{Total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Instances} * (M_{ActiveNodes} + N_{Nodes} * (M_{InstanceNode} - M_{IterationCount})) \end{aligned}$$

4.1.4.4 Vorlagenreferenzen mit expliziten Markierungen

Vorlagenreferenzen mit expliziten Markierungen (VG-2 und ZR-3) benötigen neben dem Speicherplatz für die Prozessvorlagen ($N_{Templates} * N_{Nodes} * M_{Node}$) auch Speicher für die Zustandsrepräsentation und die dynamischen Knoteninformationen einer Prozessinstanz (z. B.

Zustände oder Bearbeiter eines abgeschlossenen Schritts). Dieser entspricht der Anzahl der Knoten (N_{Nodes}) multipliziert mit dem Speicherbedarf für die Zustands- und Instanzknotenrepräsentation ($M_{NodeState} + M_{InstanceNode}$).

Formel 4.5 (Speicherbedarf für Vorlagenreferenzen mit expl. Markierungen)

$$\begin{aligned} M_{Total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Instances} * N_{Nodes} * (M_{NodeState} + M_{InstanceNode}) \end{aligned}$$

4.1.4.5 Beispiele für den Speicherbedarf der Realisierungvarianten für Knotenzustände

Nimmt man beispielhaft bestimmte Größen für die Variablen der präsentierten Formeln an, so lassen sich absolute Zahlen für den Speicherbedarf bestimmen. Tabelle 4.5 zeigt dies für zwei Klassen von Prozessen. Die beiden Klassen unterscheiden sich in der Anzahl an Prozessvorlagen, Knoten pro Prozessvorlage, Prozessinstanzen pro Prozessvorlage sowie der Anzahl an parallelen und alternativen Zweige pro Prozessvorlage. So haben einfache Prozesse zwei parallele Verzweigungen pro Prozessvorlage, zwei parallele Zweige pro Verzweigung und drei Knoten pro Zweig und parallele Prozesse vier parallele Verzweigungen pro Prozessvorlage, drei parallele Zweige pro Verzweigung, und sechs Knoten pro Zweig.

Tabelle 4.5: Speicherbedarf der verschiedenen Realisierungsvarianten für Knotenzustände

		einfache Prozesse 10 Vorl. à 20 Knoten, 100 Instanzen	parallele Prozesse 10 Vorl. à 100 Knoten, 100 Instanzen	einfache Prozesse 200 Vorl. à 20 Knoten, 100.000 Instanzen	parallele Prozesse 200 Vorl. à 100 Knoten, 100.000 Instanzen
Vorlagenkopien (expl. Markierungen) (VG-1)		512.000 Byte	2.560.000 Byte	472.800.000 Byte	2.364.000.000 Byte
Vorlagen-refer-enzen	Ausführungshistorie (ZR-1)	360.000 Byte	1.800.000 Byte	320.800.000 Byte	1.604.000.000 Byte
	impl. Markierungen (ZR-2)	108.000 Byte	532.000 Byte	68.800.000 Byte	336.000.000 Byte
	expl. Markierungen (ZR-3)	112.000 Byte	560.000 Byte	72.800.000 Byte	364.000.000 Byte

Auffallend ist der hohe Speicherbedarf bei Vorlagenkopien mit expliziten Markierungen in allen Fällen. Trotz der im Vergleich zu Vorlagenreferenzen mit Ausführungshistorie kompakten Speicherung von Zuständen, benötigen Vorlagenkopien mit expliziten Markierungen fast den doppelten Speicherbedarf. Dies liegt an der hohen Redundanz der strukturellen Informationen. Je mehr Prozessinstanzen pro Vorlage existieren, umso größer wird die Differenz. Auch die Vorlagenreferenzen mit Ausführungshistorie haben im Vergleich zu den anderen Varianten mit Vorlagenreferenzen einen relativ hohen Speicherbedarf. Somit sind diese beiden Ansätze keine ernsthaften Realisierungsvarianten – zumindest in der hier vorgestellten Form.

Den niedrigste Speicherbedarf haben Vorlagenreferenzen mit impliziten und expliziten Markierungen. Aufgrund des geringen Unterschieds können beide Varianten bezüglich des Speicherbedarfs als gleichwertig angesehen werden. Daher gibt die benötigte Laufzeit den Ausschlag für die ein oder andere Variante.

4.1.5 Weitere Aspekte

Bei der Repräsentation der Daten von Prozessvorlagen und -instanzen können noch weitere Aspekte berücksichtigt werden, die maßgeblichen Einfluss auf die Leistungsfähigkeit eines PrMS besitzen. Eine starke Verbesserung der Zugriffszeiten etwa lässt sich durch Optimierung

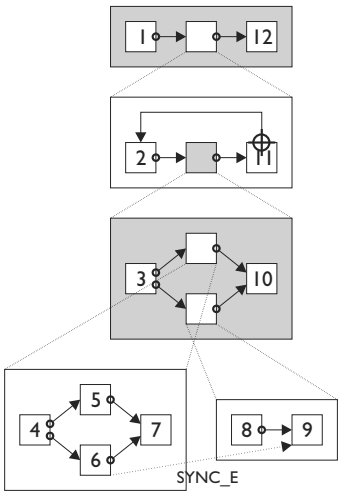


Abbildung 4.7: Realisierung von Blockstrukturen als Objekte

häufig benötigter Funktionen erzielen, etwa zur Bestimmung, ob zwei Knoten in Vorgänger-/Nachfolgerbeziehung zueinander stehen. Im Normalfall erfordert die Bestimmung der Vorgänger-/Nachfolgerbeziehung ein Durchlaufen des Graphen mit einem Aufwand von $O(n)$. Durch eine topologische Sortierung der Knoten genügt ein einfacher Vergleich mit Aufwand $O(1)$. Wie bereits erwähnt (s. Abschnitt 4.1.2) kann dies bei Prozessmodellen mit expliziten Schleifenkanten erzeugt werden. Hierzu werden Schleifenkanten bei der Bestimmung der Sortierung einfach ignoriert. Da die topologische Sortierung nur pro Vorlage und nicht pro Prozessinstanz zu speichern ist, wird nur wenig zusätzlicher Speicherplatz benötigt. Bei den meisten Prozessmetamodellen ist eine topologische Sortierung auch möglich, da sie auf Schleifen ganz verzichten oder explizite Schleifenkonstrukte besitzen.

Im Gegensatz dazu sind Blockstrukturen nur in wenigen Prozessmetamodellen vorhanden (z. B. BPEL [OAS07], ADEPT [RD97, RD98]). Durch Bereitstellung der Blockstrukturen als zusätzliche Metainformation (s. Abschnitt 2.2.2.1) sind weitere Optimierungen möglich. Dadurch können nicht benötigte Blöcke (also ganze Knotenmengen) als abstrakte Knoten aggregiert werden, was beispielsweise das Durchlaufen eines Graphen beschleunigt. Auch für die Realisierung der Blockstrukturierung gibt es mehrere Möglichkeiten:

4.1.5.1 Variante BS-1 (Blockstrukturen als Objekte)

Intern kann eine Prozessvorlage eines blockbasierten Metamodells durch eine Hierarchie von Blöcken realisiert werden (s. Abb. 4.7). Dabei sind Blöcke mit Subprozessen vergleichbar. Diese Speicherrepräsentation unterscheidet sich wesentlich von Metamodellen ohne Blockstrukturen. In dieser reinen Form sind Blöcke aber nicht realisierbar, da selbst blockbasierte Metamodelle oft noch Strukturen außerhalb der regelmäßigen Blockstruktur erlauben, z. B. Links in BPEL [OAS07] oder Synchronisationskanten im ADEPT-Metamodell (s. Abschnitt 2.2.2.1). Zudem benötigt die Strukturinformation für die Blöcke viel Speicher gegenüber einer flachen Graphstruktur.

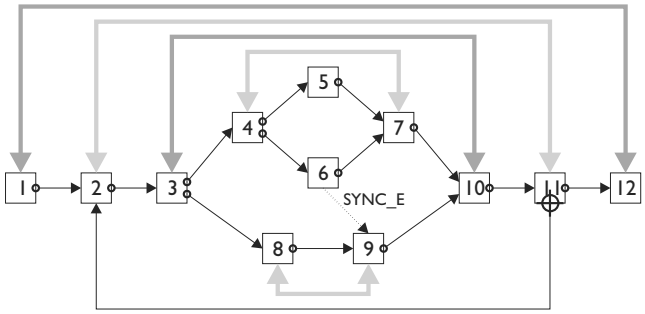


Abbildung 4.8: Realisierung von Blockstrukturen mittels Kanten (vgl. Abb. 2.7)

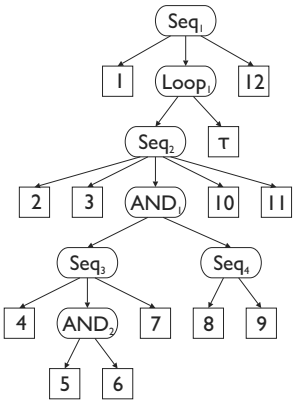


Abbildung 4.9: Realisierung von Blockstrukturen mittels Baum

4.1.5.2 Variante BS-2 (Blockstrukturen als Relation)

Eine einfache Möglichkeit, Blöcke mit einer flacher Graphrepräsentation zu realisieren, ist die Verwendung eines speziellen Kantentyps. Kanten dieses Typs bestehen zwischen Blockanfangs-/ Blockendknoten (s. Abb. 4.8). Hierdurch sind ähnlich weitreichende Optimierungen wie bei Variante BS-1 möglich, beispielsweise das Überspringen ganzer Blöcke. Trotzdem bleibt die interne Datenrepräsentation weiterhin einfach und einheitlich.

4.1.5.3 Variante BS-3 (Blockstrukturen als Baum)

Eng verwandt mit Variante BS-1 ist die Repräsentation von Blockstrukturen als Baum, etwa *Refined Process Structure Tree* [VVK08, VVK09] (s. Abb. 4.9). Die Blockstrukturen werden dabei nicht direkt in den Datenstrukturen der Prozessvorlage, sondern separat davon abgelegt. Dadurch sind Strukturen möglich, die nicht der Blockstruktur genügen. Allerdings können Blöcke beim Durchlaufen der Prozessstruktur nicht so einfach übersprungen werden wie bei den vorherigen beiden Varianten, da die Blockstrukturen nicht direkt in der Prozessvorlage repräsentiert sind. Zudem benötigt diese Variante mehr Speicher als Variante BS-1.

4.1.5.4 Variante BS-4 (Blockstrukturen als Matrix)

Einen Schritt weiter geht die Repräsentation von Blockstrukturen als (Ordnungs-)Matrix (*Order Matrix*) [LRW11, Li10] (s. Abb. 4.10). Diese repräsentiert einen Prozess als Baum (Variante BS-3) und hat dieselben Vorteile wie Variante BS-3. Zusätzlich können Blockstrukturen

	1	2	3	4	5	6	7	8	9	10	11	12	T
1		1	1	1	1	1	1	1	1	1	1	1	1
2	0		1	1	1	1	1	1	1	1	1	1	L
3	0	0		1	1	1	1	1	1	1	1	1	L
4	0	0	0		1	1	1	+	+	1	1	1	L
5	0	0	0	0		+	1	+	+	1	1	1	L
6	0	0	0	0	+		1	+	+	1	1	1	L
7	0	0	0	0	0	0		+	+	1	1	1	L
8	0	0	0	+	+	+	+		1	1	1	1	L
9	0	0	0	+	+	+	+	0		1	1	1	L
10	0	0	0	0	0	0	0	0	0		1	1	L
11	0	0	0	0	0	0	0	0	0	0		1	L
12	0	0	0	0	0	0	0	0	0	0	0		0
T	0	L	L	L	L	L	L	L	L	L	L	1	

Abbildung 4.10: Realisierung von Blockstrukturen mittels Matrix

in dieser Repräsentation leicht geändert werden, und Graphanalysen sind durch die Nutzung der Blockstrukturen sehr effizient [LRW11, Li10]. Dafür wird bei dieser Variante am meisten Speicher benötigt; die Matrix enthält wenig Information und zudem Redundanzen (die alle Knotenbeziehungen existieren zweimal). Außerdem lassen sich nur die direkt umgebenden Blöcke zweier Knoten leicht bestimmen. Verschachtelte Blöcke, also Blöcke, die andere Blöcke umgeben, lassen sich bei dieser Repräsentation nicht so leicht bestimmen wie bei den anderen Varianten.

4.1.5.5 Fazit

Tabelle 4.6: Vergleich verschiedener Repräsentationen für Blockstrukturen

	BS-1 Blöcke als Objekte	BS-2 Blöcke als Relationen	BS-3 Blöcke als Baum	BS-4 Blöcke als Matrix
Speicherbedarf	O	+	-	--
Realisierung von Attributen	O	+	+	O
Zugriff	-	+	O	-
Partitionierung	O	+	+	++

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

Aufgrund der relationalen Natur von Blockstrukturen, kann man ihre Repräsentation bezüglich derselben Aspekte vergleichen wie die Repräsentation von Kanten (s. Tabelle 4.6). Analog zu Kanten benötigt auch die Realisierung von Blöcken mittels expliziter Objekte viel Speicherbedarf gegenüber einer relationalen Speicherung. Dies liegt an den zusätzlichen, weitgehend leeren Knoten, die in den entsprechenden Elternblöcken benötigt werden, ähnlich wie im Fall von Subprozessknoten. Diese zusätzlichen Knoten sind auch der Grund dafür, dass der Zugriff bei Variante BS-1 nicht so effizient möglich ist wie bei Variante BS-2: Kindblöcke referenzieren den entsprechenden Knoten im Elternblock und dieser den entsprechenden Kindblock. Beim Durchlaufen des Graphen über mehrere Blöcke hinweg muss deshalb bei jeder Blockgrenze eine Referenz aufgelöst werden. Dies ist bei Variante BS-2 nicht notwendig.

Hier können die Blockrelationen auch einfach ignoriert werden, wenn sie bei einer Anfrage keine Effizienzsteigerung bringen.

Die Realisierung von Blockstrukturen in separaten Datenstrukturen bei Varianten BS-3 und BS-4 benötigt noch mehr Speicherplatz als bei Variante BS-1. Die Trennung von den Datenstrukturen für den Prozessgraphen erschwert den Zugriff und damit die Verwendung der Blockstrukturen beim Durchlaufen des Graphen. Die Separation hat allerdings Vorteile bezüglich der Partitionierung. Während die Blockstrukturen bei Varianten BS-3 und BS-4 separat von den Graphstrukturen partitioniert werden, ist die Partitionierung bei Variante BS-2 implizit gegeben. Am schwierigsten ist sie bei Variante BS-1. Diese stellt bereits eine horizontale Partitionierung des Graphen dar, was bei der Partitionierung explizit berücksichtigt werden muss.

Sollen Blockstrukturen mit zusätzlichen Attributen versehen werden, etwa dem Blocktyp oder Zeitinformationen (z. B. die maximale Dauer des Blocks), ist dies bei allen Varianten in vergleichbarer Weise möglich: Blockobjekte, Blockkanten und Baumelemente können mit zusätzlichen Attributen versehen werden. Bei Variante BS-4 können einfache Attribute direkt in der Matrix repräsentiert werden. Letztlich sind die Einträge selbst bereits Attribute, im Normalfall der Blocktyp. Zusätzliche oder komplexe Attribute sind hier allerdings schwieriger zu repräsentieren als bei den anderen Varianten.

4.2 Physische Speicherstrukturen

Bisher haben wir die logische Repräsentation von Graphstrukturen, Zustandsdaten und weiteren Metadaten diskutiert. In diesem Abschnitt betrachten wir die physische Repräsentation der Daten von Prozessvorlagen und -instanzen. Die physische Repräsentation kann in Primär- und Sekundärspeicher unterschiedlich sein (z. B. objektorientiert und relational). In einem solchen Fall ist beim Ein-/Auslagern eine Transformation der Daten nötig (s. Abb. 4.11). Um keine negativen Auswirkungen auf die Leitungsfähigkeit des PrMS zu haben, muss eine solche Transformation möglichst einfach sein.

Ein weiterer Aspekt ist die Verwendung gespeicherter Prozeduren (*Stored Procedure*) oder vergleichbarer Spracherweiterungen bei DBMS [HR01, SH99]. Dadurch können Transformationen eingespart werden, da keine Einlagerung in den Primärspeicher notwendig wird. Gespeicherte Prozeduren sind jedoch funktional nicht so einfach in der Handhabung wie Programmiersprachen und sind zudem wesentlich schwieriger zu warten (NA 7).

Ob auf Daten im Primär- oder Sekundärspeicher zugegriffen wird, hängt von der Zugriffsart ab. Werden Daten häufig benötigt (z. B. Propagation im Rahmen einer Schemaevolution), erfolgt dies am besten im Primärspeicher. Dagegen erfolgt die Abfrage oder Änderung einzelner Knotenattribute idealerweise direkt im Sekundärspeicher. Dies ist wesentlich effizienter, da die zum Ein-/Auslagern notwendige Transformation bzw. Rücktransformation entfällt.

Um die vom PrMS bereitgestellten Funktionen sinnvoll nutzen zu können, ist es unabdingbar, dass für den Aufrufer einer Funktion weitestgehend transparent ist, in welchem Speicher die Daten manipuliert werden.

4.2.1 Primärspeicher

Im Primärspeicher gibt es drei Arten von Speicherstrukturen für die physische Realisierung von Entitäten (z. B. Prozessinstanzen und Knoten): unstrukturierte Daten (z. B. untypisierte Container), strukturierte Daten (z. B. Objekte) und semi-strukturierte Daten (z. B. Arrays).

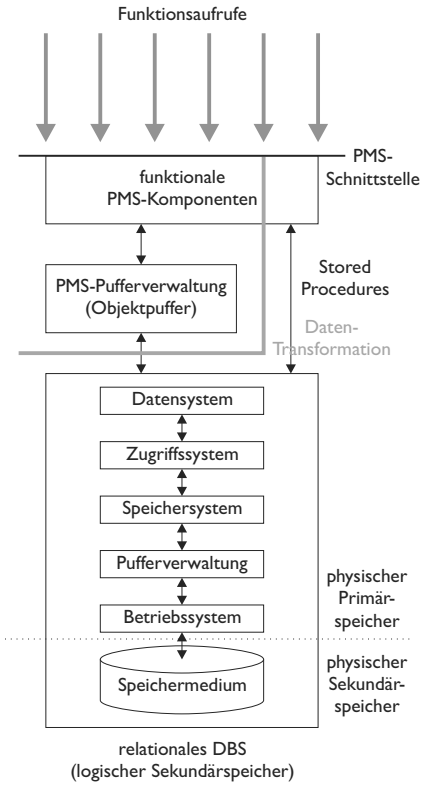


Abbildung 4.11: Datenzugriffe in PrMS

4.2.1.1 Variante PS-1 (Unstrukturierte Daten)

Unstrukturierte Daten sind für die Speicherverwaltung des Betriebssystems oder der Laufzeitumgebung (z. B. JVM) rein binäre Objekte, d. h. sie werden ausschließlich in den zugreifenden Funktionen interpretiert. Dort ist auch die Objektstruktur hinterlegt, etwa die Speicherstelle des Datums und der Versatz (*Offset*) des Attributs. Problematisch sind variabel große Attribute, etwa die als Zeichenkette bei einer Aktivität hinterlegte Bearbeiterzuordnung. Sie verhindern den direkten Zugriff auf Attribute, da die Position im Bitstrom variiert. Deshalb muss immer die gesamte Entität eingelesen werden, was viel Zeit kostet. Zudem wird hierdurch die Partitionierung erschwert.

Die Verwendung unstrukturierter Daten ist vergleichbar mit manueller Speicherverwaltung. Diese ist nur in Hochleistungsumgebungen sinnvoll, in denen Leistungsfähigkeit oberste Priorität hat. Insbesondere müssen massive Probleme bei der Wartung (NA 7) in Kauf genommen werden. In allen anderen Fällen empfiehlt es sich, die Arbeit dem Compiler oder der Laufzeitumgebung zu überlassen und strukturierte Daten zu verwenden. Die automatische Speicherverwaltung besitzt heutzutage aufgrund des hohen Entwicklungsstands nur geringe Auswirkungen auf Speicherbedarf und Laufzeit [Wil92, Jon96, App87].

4.2.1.2 Variante PS-2 (Programmiersprachenobjekte)

Aufgrund der Verbreitung objektorientierter Programmiersprachen bieten sich Objekte für die Realisierung von Entitäten an. Entitätsattribute werden als Objektvariablen direkt mittels Programmiersprachenkonstrukten realisiert. Dadurch ist ein schneller Zugriff ohne zusätzlichen

algorithmischen Aufwand möglich. Es fällt jedoch zusätzlicher Speicherbedarf für Verwaltungsinformationen an. So benötigt ein Objekt in Java initial 8 Byte für den Objektkopf (*Objektheader*). Dies ist bei vielen kleinen Objekten (z. B. ein Objekt pro Kante) problematisch; je kleiner ein Objekt ist, desto höher ist sein initialer Speicherbedarf. Programmiersprachenobjekte lassen sich zudem schlecht vertikal partitionieren. Dies ist zwar mittels statischer Festlegung einzelner Objektpartitionen möglich, etwa mittels Stellvertreterobjekten. Diese führen aber zu einer größeren Anzahl von Objekten und damit zu einem schlechteren Verhältnis von Kopf- zu Nutzdaten.

4.2.1.3 Variante PS-3 (Semi-strukturierte Daten)

Semi-strukturierte Daten bilden einen Kompromiss zwischen unstrukturierten Daten und Programmiersprachenobjekten. Dabei werden gleichartige Daten (z. B. ein bestimmtes Attribut mehrerer Entitäten desselben Typs) zusammen abgelegt, etwa durch Verwendung eines Array. Während bei Variante PS-2 jeweils eine Entität in einem zusammenhängenden Speicherbereich zu finden ist, ist bei Variante PS-3 in einem zusammenhängenden Speicherbereich (Array) genau ein Attribut aller Entitäten abgelegt. Der Zugriff auf entsprechende Attributwerte erfolgt ausschließlich über den Index (die Kennung) der Entität. Dies reicht in den meisten Fällen aus. Findet ein Zugriff bezüglich eines bestimmten Attributwerts statt, etwa die Suche nach allen Knoten in bestimmtem Zustand, kann dies durch die Bereitstellung von Indexstrukturen und -verfahren wie Streuwertverfahren (*Hashing*) sinnvoll unterstützt werden.

4.2.1.4 Fazit

Für die Repräsentation der in einem PrMS anfallenden Daten sind semi-strukturierte Daten, also ein- oder mehrdimensionale Arrays aufgrund der guten Partitionierbarkeit und des einfachen Ein-/Auslagerns am besten geeignet. Gewöhnlich gibt es wenige dynamische Änderungen, wodurch die Datenstrukturen stabil bleiben. Die Eignung für die Repräsentation im Primärspeicher hängt jedoch auch von den Speicherstrukturen im Sekundärspeicher ab. Da aufwendige Datentransformationen schnell zu einem Flaschenhals werden können, sind beispielsweise semi-strukturierte Daten im Primärspeicher bei gleichzeitiger Verwendung einer objektorientierten Datenbank nicht sinnvoll.

Vorteile semi-strukturierter Daten gegenüber Programmiersprachenobjekten sind einerseits der geringere Speicherbedarf, andererseits die strikte Trennung zwischen Daten und Funktionen/Anwendungslogik. Unsere Erfahrungen mit verschiedenen Implementierungen des ADEPT-Metamodells haben gezeigt, dass sich dadurch sowie durch die gute Partitionierbarkeit das Ein-/Auslagern vereinfacht. Ein Nachteil an semi-strukturierten Daten ist jedoch die mangelnde Dynamik der Strukturen. So müssen die Entitätskennungen jederzeit stabil bleiben, was dazu führt, dass beim Löschen leere Einträge in den Speicherbereichen entstehen.

Tabelle 4.7 fasst die qualitative Bewertung zusammen.

4.2.2 Sekundärspeicher

Auch im Sekundärspeicher kann zwischen unstrukturierten (z. B. BLOB (*Binary Large Object*)) und strukturierten Daten (z. B. relationale, normalisierte Daten) unterschieden werden. Strukturierte Daten im Sekundärspeicher sind nur dann sinnvoll, wenn sie auch im Primärspeicher verwendet werden. Für die Umkehrung gilt dies nicht. So kann es günstig sein, strukturier-

Tabelle 4.7: Vergleich verschiedener Repräsentationen für Primärspeicher

	PS-1 Unstrukturierte Daten	PS-2 Progr.sprachen-objekte	PS-3 Semi-strukt. Daten
Speicherbedarf	+	-	O
Zugriff auf Attribute	-	++	+
Partitionierung	-	O	+

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

te Daten im Primärspeicher entitätsweise zu serialisieren und unstrukturiert als BLOB im Sekundärspeicher abzulegen.

Unstrukturierte Daten im Sekundärspeicher können schnell gespeichert und ausgelesen werden. Jedoch sind keine externen Zugriffe und damit auch keine direkten Anfragen an den Sekundärspeicher (z. B. mittels SQL) möglich. Alle Zugriffe müssen über die vom PrMS bereitgestellte Schnittstelle erfolgen (z. B. die Ermittlung der Nachfolger eines bestimmten Knotens). Eine Partitionierung unstrukturierter Daten im Sekundärspeicher ist nicht möglich. Diese müssen komplett ausgelesen und ggf. im Primärspeicher partitioniert werden. Relationale Datenbanken bieten eine ausgereifte Technik für die Speicherung strukturierter Daten. Sie ermöglichen eine einfache Transformation der Datenstrukturen zwischen Primär- und Sekundärspeicher. In vielen (relationalen) DBMS gibt es Unterstützung für gespeicherte Prozeduren, so dass Funktionen direkt im Sekundärspeicher ausgeführt werden können. Zudem sind mittels SQL effiziente Anfragen gegen alle Daten und damit auch gegen Kollektionen von Prozessinstanzen möglich.

Mit relationalen Datenstrukturen im Sekundär- und Programmiersprachenobjekte im Primärspeicher stellt sich das Problem des *Impedance Mismatch* [GR92]. Eine große Schwierigkeit besteht dabei in der unterschiedlichen Realisierung von Referenzen im Primär- und Sekundärspeicher. Dies lässt sich nur mittels einer zentralen Komponente im PrMS lösen, die alle zu einem Zeitpunkt existierenden Primärspeicherreferenzen kennt und die Sekundärspeicherreferenzen entsprechend transformieren kann.

Eine einfache Möglichkeit aufwendiges Ein-/Auslagern zu optimieren und die Transformation zwischen der Datenrepräsentation im Primär- und Sekundärspeicher zu vermeiden, ist die Pufferung der Daten zwischen Primär- und Sekundärspeicher (s. Abb. 4.11). Dabei ist zu beachten, dass ein DBMS intern schon Daten puffert. Eine Pufferverwaltung mit Seiten und Segmenten wie im DBMS ist daher für das PrMS nicht sinnvoll. Statt dessen sollten mit dem PrMS-Puffer geeignete Objekte (z. B. Knoten, Kanten, Zustandsdaten, Historiendaten) verwaltet werden. Aufgrund unterschiedlicher Größen dieser Objekte, ist der Puffer flexibel zu gestalten, was eine aufwendige Pufferverwaltung erfordert. Das Konzept ist mit objektrelationalen DBMS vergleichbar, in denen auch Objektpuffer verwendet werden. Diese werden entweder zusätzlich zu Seiten- und Segmentpuffer verwendet oder sie ersetzen diese [SH99]. Zudem ist eine Kooperation von DBMS- und PrMS-Puffer nötig, damit Daten nicht redundant in beiden Puffern vorliegen. Die Redundanz kann beispielsweise dadurch vermieden werden, dass alle Prozessvorlagen im Primärspeicher oder im PrMS-Puffer gehalten werden und nur die Daten der Prozessinstanzen bei Bedarf vom Sekundärspeicher eingelesen werden.

4.3 Zusammenfassung

Die Speicherung und Verwaltung von Prozessinstanzen hat großen Einfluss auf die Skalierbarkeit und Leistungsfähigkeit (NA 1) eines PrMS. Bei der Diskussion der Speicherrepräsentation für Prozessvorlagen und unveränderte Prozessinstanzen haben wir dafür, neben der Partitionierung der Datenstrukturen zwecks Verbesserung des Speicherbedarfs, die Erweiterbarkeit (NA 3) berücksichtigt.

Prozessvorlagen bestehen hauptsächlich aus Knoten, die Prozessschritte repräsentieren, sowie Beziehungen zwischen den Knoten. Hierzu gehören die Kanten, die die Reihenfolge der Prozessschritte definieren, ebenso wie Blockstrukturen, die eine spezielle Beziehung zwischen bestimmten Knoten (den Blockanfangs- und -endknoten) darstellen. Sowohl Kanten als auch Blockstrukturen werden am besten als Relationen repräsentiert, etwa Adjazenzlisten. Diese Darstellung hat den Vorteil, dass sie problemlos um zusätzliche Attribute erweitert werden kann. Also lassen sich sowohl Kanten als auch Blockstrukturen um zusätzliche Attribute erweitern.

Um Redundanzen zu vermeiden, ist es empfehlenswert, dass Prozessinstanzen selbst keine vollständigen Graphstrukturen besitzen, sondern lediglich die entsprechenden Prozessvorlagen referenzieren. Dies gilt jedoch nur für unveränderte Prozessinstanzen, für veränderte Prozessinstanzen funktioniert dies nicht, wie wir im nächsten Kapitel zeigen werden. Die Zustände von Prozessinstanzen und damit die Zustände der einzelnen Knoten einer Prozessinstanz werden am besten mittels expliziter oder impliziter Markierungen repräsentiert. Während explizite Markierungen etwas mehr Speicher benötigen, fällt bei ihnen, anders als bei impliziten Markierungen, kein Rekonstruktionsaufwand für bestimmte Knotenzustände an. Aufgrund der einfacheren Darstellung werden wir im folgenden explizite Markierungen verwenden.

Für die physische Repräsentation bieten sich semi-strukturierte Daten an. Diese sind der beste Kompromiss zwischen Speicherbedarf und Aufwand für die Speicherverwaltung. Außerdem lassen sie sich sowohl strukturiert als auch unstrukturiert im Sekundärspeicher ablegen.

5 Speicherung und Verwaltung veränderter Prozessinstanzen

Nachdem wir in Kapitel 4 die Speicherrepräsentation von Prozessvorlagen und unveränderten Prozessinstanzen untersucht haben, betrachten wir in diesem Kapitel die Speicherrepräsentation veränderter Prozessinstanzen im Hinblick auf Skalierbarkeit und Leistungsfähigkeit (NA 1). Auch Kapitel 5 unterscheidet zwischen logischer Ebene und physischer Speicherrepräsentation sowie zwischen Primär- und Sekundärspeicher.

5.1 Realisierungskonzepte

Die Umsetzung instanzspezifischer Änderungen hängt davon ab, wie Instanzvorlagen bei unveränderten Prozessinstanzen realisiert werden (s. Abschnitt 4.1.3.1). Bei Variante VG-1 (Vorlagenkopie) werden intern für jede Prozessinstanz eigene Datenstrukturen für die jeweilige Prozessstruktur angelegt, bei Variante VG-2 (Vorlagenreferenzen) existieren diese nur einmal pro Prozessvorlage und werden von den entsprechenden Prozessinstanzen lediglich referenziert. Da Vorlagenreferenzen wesentlich weniger Speicher benötigen, sind sie für die Realisierung der Datenstrukturen für Prozessvorlagen und unveränderte Prozessinstanzen vorzuziehen. Allerdings lassen sich mit dieser Repräsentationsform instanzspezifische Änderungen nicht direkt realisieren. Diese Art von Änderungen muss auf einer Prozessvorlage durchgeführt werden, die nur für die jeweils abzuändernde Prozessinstanz gilt (*instanzspezifische Prozessvorlage*). In den folgenden Abschnitten stellen wir Ansätze vor, die auf Grundlage von Vorlagenreferenzen mit sinnvollen Erweiterungen auch instanzspezifische Änderungen realisieren.

Die einfachste Realisierungsvariante für die Repräsentation instanzspezifischer Änderungen ist die Verwendung individueller Instanzgraphen (Variante ÄR-1). Diese entsprechen einer normalen Vorlagenkopie (Variante VG-1), allerdings besitzt eine Prozessinstanz nur dann einen individuellen, von einer Prozessvorlage abgeleiteten Instanzgraph (und keine einfache Referenz wie bei Variante VG-2), wenn die Prozessinstanz zuvor Änderungen unterworfen worden ist. Der individuelle Instanzgraph selbst repräsentiert eine vollständige Prozessvorlage wie bei Variante VG-1.

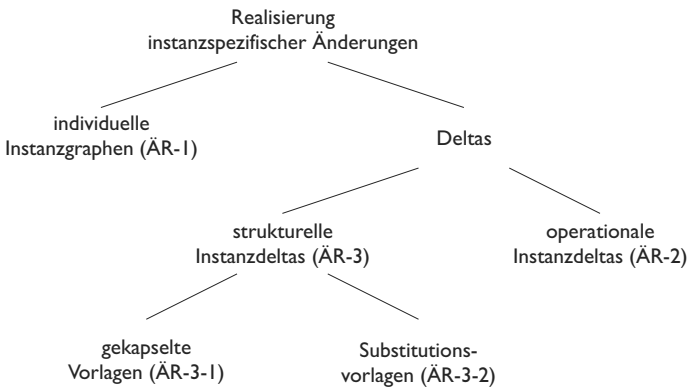


Abbildung 5.1: Klassifikation der Realisierungskonzepte für instanzspezifische Änderungen

Alternativ lassen sich instanzspezifische Änderungen auch mit einer Vorlagenreferenz (Variante VG-1) realisieren. Neben der Referenz werden bei dieser Variante auch die Unterschiede (Deltas) zwischen der durch die Änderungen resultierenden instanzspezifischen Vorlage und der ursprünglichen Prozessvorlage gespeichert. Dadurch ist es nicht nötig, eine vollständige (geänderte) Prozessvorlage und somit einen individuellen Instanzgraphen explizit zu speichern. Das heißt, werden die zusätzlich zur Vorlagenreferenz existierenden Änderungen geeignet repräsentiert, kann die resultierende instanzspezifische Vorlage bei Bedarf dynamisch erzeugt werden.

Die Realisierung der Deltas im Kontext von Variante VG-1 wiederum kann auf zwei Arten erfolgen: Die instanzspezifisch angewandten Änderungsoperationen können direkt als Operationen (z. B. als Änderungshistorie) repräsentiert werden (Variante ÄR-2, sog. *operationale Instanzdeltas*). Alternativ können die aus den Änderungsoperationen resultierenden Graphstrukturen gespeichert werden (Variante ÄR-3, sog. *strukturelle Instanzdeltas* [LRR04, Jur06]). Die geänderten Graphstrukturen werden intern durch Mengen hinzugefügter, geänderter, verschobener und gelöschter Knoten und Kanten repräsentiert. Auch hier kann man wieder zwei Realisierungsvarianten unterscheiden: *Gekapselte Vorlagen* (Variante ÄR-3-1), die implizit und transparent zwischen geänderten Prozessstrukturen und der ursprünglichen Prozessvorlage unterscheiden, und *Substitutionsvorlagen* (Variante ÄR-3-2), bei denen explizit zwischen geänderten Graphstrukturen und ursprünglicher Prozessvorlage unterschieden wird.

Abb. 5.1 gibt einen Überblick über die verschiedenen im Rahmen dieser Arbeit entwickelten und untersuchten Varianten zur Realisierung instanzspezifischer Änderungen.

In diesem Abschnitt werden die vielversprechendsten Varianten ÄR-1, ÄR-2 und ÄR-3 im Detail vorgestellt und miteinander verglichen. Die Anforderungen an diese Varianten entsprechen im Kern wieder denen an die Repräsentation unveränderter Prozessinstanzen (s. Kapitel 4): niedriger Speicherbedarf, horizontale und vertikale Partitionierbarkeit, weitgehende Transparenz für Algorithmen, instanzübergreifende Anfragen (Sekundärspeicher) und Unterstützung von Schemaevolution. Die Varianten ÄR-3-1 und ÄR-3-2 für die Realisierung struktureller Instanzdeltas werden in Abschnitt 5.4 ausführlich diskutiert. Insgesamt ist eine eingehende Betrachtung dieser verschiedenen Implementierungsvarianten unabdingbar, um instanzspezifische Änderungen effizient und in realen Umgebungen einsetzen zu können.

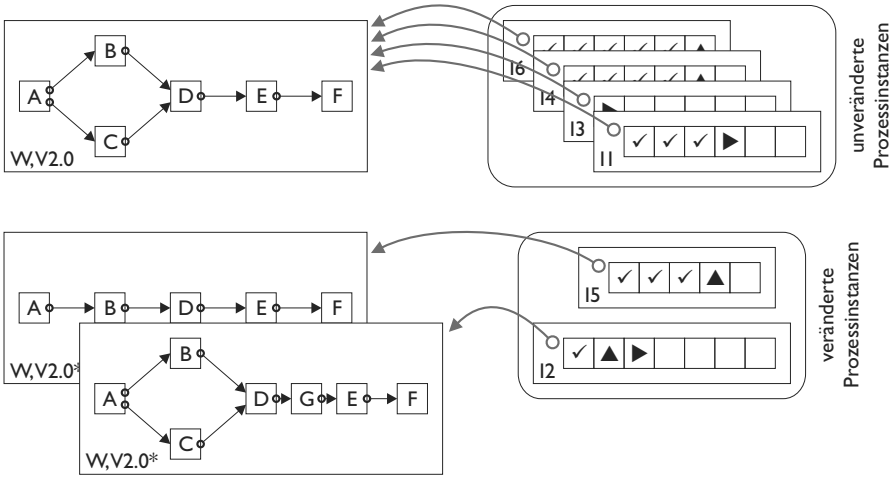


Abbildung 5.2: Änderungen repräsentiert durch individuelle Instanzgraphen (ÄR-1)

5.1.1 Variante ÄR-1: Individuelle Instanzgraphen

Eine naheliegende Lösung zur Realisierung von Instanzänderungen bieten individuelle Instanzgraphen. Dabei wird für die Durchführung von Änderungen eine vollständige Kopie der zugrundeliegenden Vorlage angelegt, auf der dann die entsprechenden Änderungen auch durchgeführt werden. Die Prozessinstanz referenziert anschließend das neu erstellte, instanzspezifische Vorlagenobjekt (s. Abb. 5.2). Im Vergleich zu Vorlagenkopien bei unveränderten Prozessinstanzen (s. Abschnitt 4.1.3.1) wird hier jedoch nur im Fall instanzspezifischer Änderungen eine Vorlagenkopie angelegt, nicht dagegen für unveränderte Prozessinstanzen. Unveränderte Prozessinstanzen referenzieren jeweils die ursprüngliche Prozessvorlage, wie von Variante VG-2 in Abschnitt 4.1.3.1 vorgeschlagen. Realisiert wunde diese Variante z. B. in den PrMS WASA [Wes99, Wes01a] und ProMInanD [KRW90, Kar94].

Zugriffe auf die internen Datenstrukturen erfolgen bei unveränderten und veränderten Prozessinstanzen auf dieselbe Art. In beiden Fällen wird bei Zugriffen auf strukturelle Informationen die Referenz auf die für die Prozessinstanz gültige Prozessvorlage aufgelöst und die Anfrage entsprechend auf der Prozessvorlage verarbeitet. Hierdurch ist eine vollständige Transparenz von Instanzänderungen gegeben. Auf diese Weise benötigen viele Algorithmen keine speziellen Anpassungen, um mit instanzspezifischen Änderungen umgehen zu können. Dies unterstützt die Einfachheit (NA 4) und die Wartbarkeit (NA 7).

Individuelle Instanzgraphen werden intern wie vollwertige Prozessvorlagen verwaltet. Das Anlegen eines individuellen Instanzgraphen entspricht dem Ändern von Prozessvorlagen (s. Abschnitt 2.3.1.1), d. h. die Vorlage wird geklont, und der Klon wird anschließend strukturell geändert. Im Gegensatz zu einer Prozessvorlage kann ein individueller Instanzgraph jedoch nicht (neu) instanziert werden; es gibt exakt eine darauf laufende Prozessinstanz.

Problematisch ist dieses Konzept beim Zusammentreffen beider Änderungsarten, also die Änderung der ursprünglichen Prozessvorlage einer mittlerweile individuell geänderten Prozessinstanz (s. Abschnitt 2.3.2.3). Da für instanzspezifische Änderungen dieselben Konstrukte wie für die Änderungen an der Prozessvorlage verwendet werden, können beide Änderungsarten nicht direkt sondern nur mittels der Änderungshistorie unterschieden werden. Bei diesem Änderungsszenario ist aber genau diese Unterscheidung essentiell. So werden geänderte Prozessinstanzen bei der Migration nicht direkt berücksichtigt, da sie nicht mehr die ursprüngliche

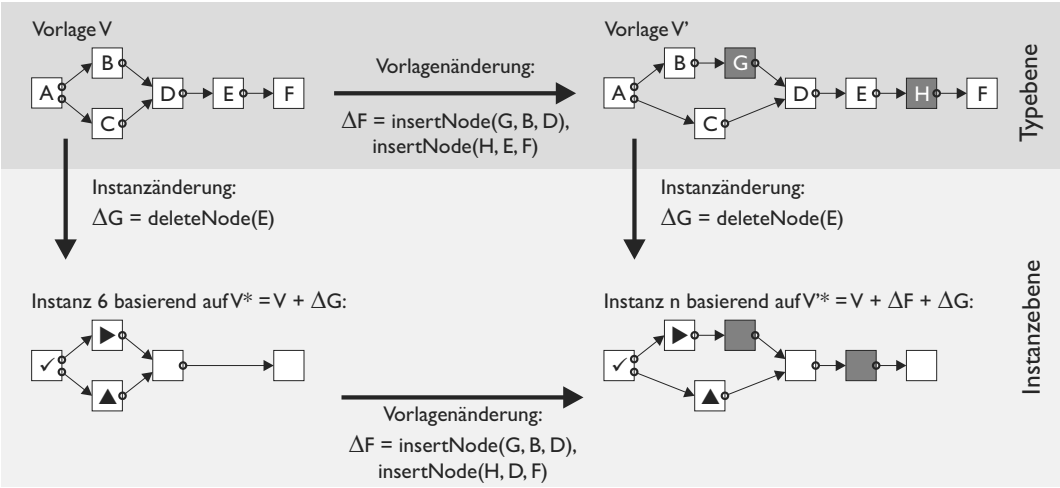


Abbildung 5.3: Schemaevolution mit individuellem Instanzgraphen und angepasstem Änderungskontext

Prozessvorlage referenzieren sondern einen individuellen Instanzgraphen. Diese starke Restriktion wird beispielsweise im PrMS WASA [Wes99, Wes01a] angewendet. Die entsprechenden Prozessinstanzen könnten nur migriert werden, indem die Vorlagenänderungen der Schemaevolution auch auf dem individuellen Instanzgraphen durchgeführt werden, oder indem dieser komplett verworfen und, auf einer Kopie der neuen Prozessvorlage, entsprechend neu aufgebaut wird.

Beide Varianten zur Durchführung der Schemaevolution bei vorhandenen individuellen Instanzgraphen sind aufwendig, da in jedem Fall Änderungen wiederholt werden müssen: dies betrifft entweder die instanzspezifischen Änderungen oder die Änderungen der Prozessvorlage (s. Abb. 5.3). Dies ist besonders kritisch, da für die nächste Schemaevolution wieder eine korrekte Änderungshistorie benötigt wird. Diese muss durch die wiederholten Änderungsoperationen aufwendig erzeugt werden:

Beispiel 5.1 (Änderungskontext bei Schemaevo. mit individuellem Instanzgr.)
Bei Instanz 6 in Abb. 5.3 wurde Knoten E gelöscht. Bei der neuen Prozessvorlage wurde genau hinter diesem Knoten ein neuer Knoten H eingefügt. Bei Instanz 6 existiert der Kontext der entsprechenden Änderungsoperation allerdings nicht mehr und muss aufwendig neu ermittelt werden. Die Änderungsoperation muss deshalb logisch umgewandelt werden, damit sie den neuen Knoten H nach Knoten D einfügt.

Insgesamt sind individuelle Instanzgraphen einfach in der Realisierung und bieten trotzdem eine vollständige Transparenz von Instanzänderungen. Es wird jedoch relativ viel Speicherplatz benötigt, da die Erstellung vollständiger Vorlagenkopien auch Redundanzen bezüglich der Vorlageninformationen mit sich bringt. Noch schwerer wiegt der große Laufzeitaufwand für die Durchführung einer Schemaevolution für geänderte Prozessinstanzen, da beide Änderungsarten intern gleich realisiert werden und kein direkter Vergleich der Änderungen möglich ist.

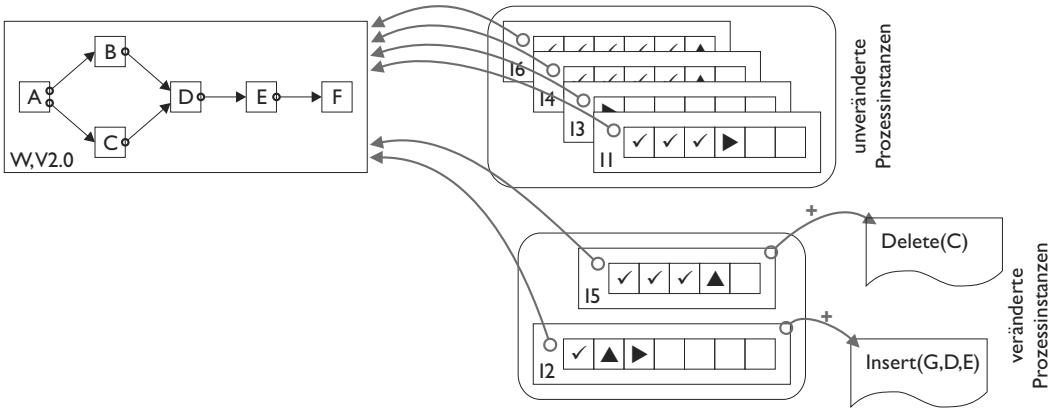


Abbildung 5.4: Änderungen mittels operationaler Instanzdeltas (ÄR-2)

5.1.2 Variante ÄR-2: Operationale Instanzdeltas

Operationale Instanzdeltas unterscheiden sich in zwei Aspekten von individuellen Instanzgraphen: Einerseits werden nur Unterschiede (Deltas) gegenüber der ursprünglichen Prozessvorlage verwaltet. Dadurch kann auf die ursprüngliche Prozessvorlage nicht verzichtet werden, es bleibt aber der Bezug zu ihr bestehen. Andererseits werden Instanzänderungen nicht als Graphstrukturen, sondern in Form von Änderungsoperationen gespeichert. Dies entspricht (einem Teil) der Änderungshistorie (s. Abb. 5.4).

Gegenüber individuellen Instanzgraphen ist die Durchführung von Instanzänderungen mit operationalen Instanzdeltas effizient: Neben der für zahlreiche Funktionen benötigten Änderungshistorie müssen keine zusätzlichen Datenstrukturen erzeugt und verwaltet werden. Dafür müssen vor Zugriffen auf geänderte Prozessinstanzen erst die entsprechenden Änderungen durchgeführt werden. Dies kann beispielsweise auf einer temporären Kopie der ursprünglichen Prozessvorlage erfolgen, um Auswirkungen auf andere, unveränderte Prozessinstanzen zu verhindern. Dies ist mit Variante ÄR-1 vergleichbar, wobei der individuelle Instanzgraph nur bei Bedarf erzeugt wird, was insgesamt jedoch aufwendig sein kann (s. Abb. 5.5).

Große Vorteile besitzt die Realisierung operationaler Instanzdeltas im Zusammenhang mit der Realisierung von Schemaevolution. Da die Instanzänderungen direkt in Form von Operationen vorliegen, kann der Vergleich mit den entsprechenden Vorlagenänderungen effizient erfolgen. Dabei können bei Bedarf relativ einfach die Kontexte von Änderungsoperationen in den operationalen Instanzdeltas geändert oder ganze Änderungsoperationen entfernt werden. Durch diese Anpassungen können die geänderten Prozessinstanzen anschließend ohne zusätzliche strukturellen Anpassungen auf die neue Vorlage umgehängt werden.

Während operationale Instanzdeltas effizient Änderungen an Prozessinstanzen und -vorlagen ermöglichen, gestalten sich Zugriffe auf die zugrundeliegenden Prozessstrukturen problematisch. Dabei muss jedesmal die resultierende instanzspezifische Vorlage erzeugt werden, um benötigte Informationen (z. B. Vorgänger/Nachfolger eines Knotens) ermitteln zu können. Diese Information kann nur mit einem hohen Aufwand aus der Änderungshistorie ermittelt werden (s. Abb. 5.5), sie wird jedoch relativ häufig benötigt. Ohne diesen gravierenden Nachteil wären operationale Instanzdeltas gut für die Realisierung instanzspezifischer Änderungen geeignet. Sie benötigen zudem keinerlei zusätzlichen Speicherplatz, da eine Änderungshistorie zu Dokumentationszwecken in jedem Fall erforderlich ist.

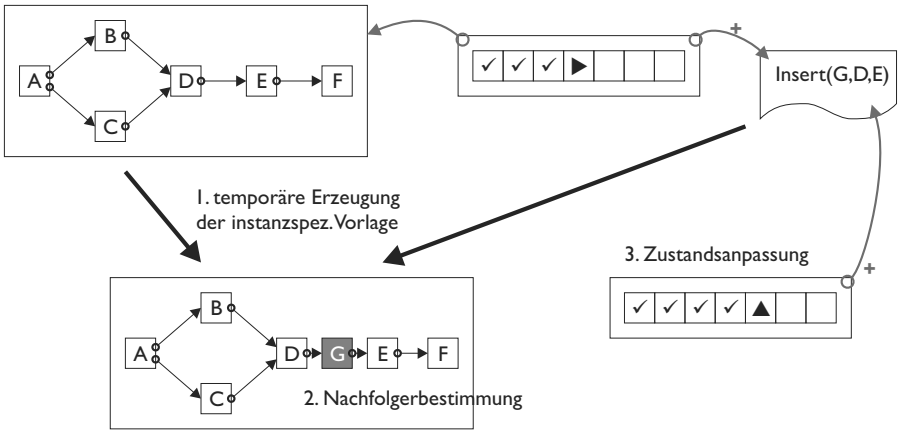


Abbildung 5.5: Weiterschalten mit Rekonstruktion der instanzspezifischen Vorlage

5.1.3 Variante ÄR-3: Strukturelle Instanzdeltas

Strukturelle Instanzdeltas bieten eine vielversprechende Alternative. Sie bewegen sich zwischen individuellen Instanzgraphen und operationalen Instanzdeltas. Dabei verbinden sie deren Vorteile und vermeiden die Nachteile. Wie bei operationalen Instanzdeltas werden nur Instanzänderungen und keine vollständigen Prozessvorlagen verwaltet. Änderungen liegen jedoch nicht operational, sondern wie bei individuellen Instanzgraphen als realisierte Graphstrukturen vor. Dazu wird bei Instanzänderungen der von der Änderung betroffene Bereich (d. h. der Änderungskontext, s. Abschnitt 2.3.1) der ursprünglichen Prozessvorlage kopiert und anschließend geändert. Bei Zugriffen substituiert diese Kopie (das *Substitutionsobjekt*) den entsprechenden Bereich der ursprünglichen Prozessvorlage.

Beispiel 5.2 (Strukturelle Instanzdeltas mittels Änderungskontexten)
In Abb. 5.6 ist bei Instanz 5 der Knoten C gelöscht worden. Da hierbei eine Verzweigung aufgelöst wurde, ändert sich durch das Löschen auch der Knotentyp der Knoten A und D. Deshalb umfasst der Änderungskontext die Knoten A und C sowie, infolge des Wegfalls der Parallelität, auch den Knoten B. Dieser Änderungskontext wurde entsprechend der Änderungsoperation manipuliert, d. h. Knoten C wurde gelöscht, so dass der Kontext nun das Substitutionsobjekt bildet. Durch die Überlagerung des Substitutionsobjekts auf die entsprechenden Knoten der ursprünglichen Prozessvorlage entsteht die für Instanz 5 gültige instanzspezifische Prozessvorlage.

Damit vereinen strukturelle Instanzdeltas die Vorteile individueller Instanzgraphen und operationaler Instanzdeltas. Einerseits kann die instanzspezifische Prozessvorlage bei Bedarf schnell erzeugt werden und ermöglicht anschließend direkte Zugriffe auf ihre Graphstrukturen. Andererseits ist der Speicherbedarf gering, da gegenüber der ursprünglichen Prozessvorlage keine redundanten Informationen verwaltet werden.

Konzeptionell sind zwei Arten der Substitution möglich: *einfache* und *hierarchische* (d. h. mehrfache) *Substitution* (s. Abb. 5.7). Bei der einfachen Substitution existiert für eine geänderte Prozessinstanz jeweils nur ein Substitutionsobjekt, auch wenn die entsprechende Prozessinstanz mehrmals geändert wurde. Dagegen führt bei hierarchischer Substitution jede Änderungstransaktion zu einem eigenen Substitutionsobjekt, das die vorherigen überlagert.

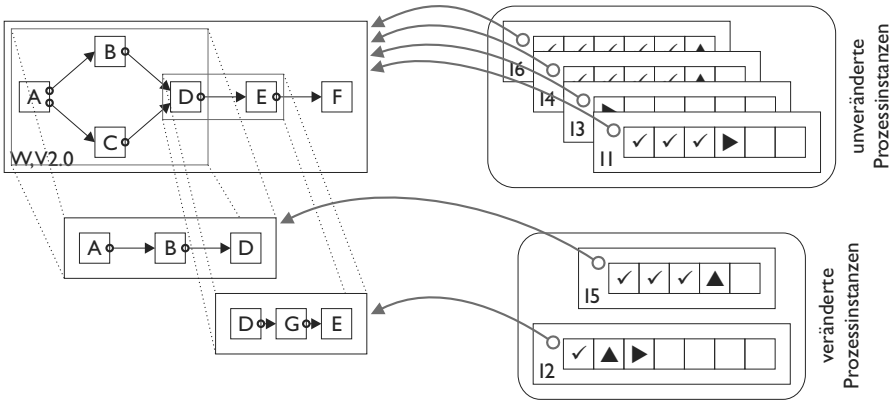
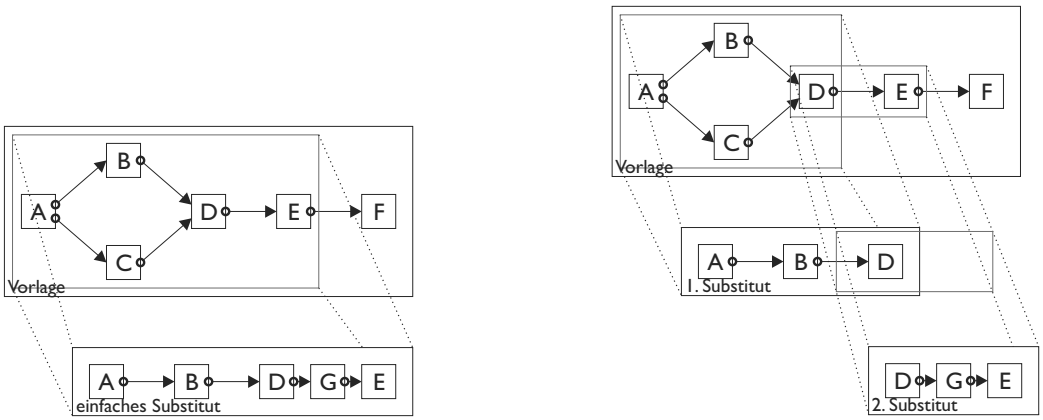


Abbildung 5.6: Änderungen mittels struktureller Instanzdeltas (ÄR-3)

Hierarchische Substitution hat den Vorteil, dass die Substitutionsobjekte zum Zeitpunkt der Änderung schnell erzeugt werden können. Dabei ist es irrelevant, ob bereits Substitutionsobjekte existieren oder nicht. Eine Änderung beruht immer auf der zum Änderungszeitpunkt gültigen (logischen) Prozessvorlage einer Prozessinstanz. Dies führt jedoch potentiell zu einem höheren Speicherbedarf, da ggf. redundante Informationen verwaltet werden. Dies ist beispielsweise der Fall, wenn durch komplementäre Änderungen vorherige Änderungen aufgehoben werden oder ein Änderungskontext für mehrere Substitutionsobjekte gilt (beispielsweise Knoten D in Abb. 5.7). In so einem Fall ist ein konsolidiertes Substitutionsobjekt effizienter bezüglich Speicherbedarf.



(a) Einfache Substitution

(b) Hierarchische Substitution

Abbildung 5.7: Einfache und hierarchische Substitution bei strukturellen Instanzdeltas

Beispiel 5.3 (Einfache und hier. Substitution bei strukturellen Instanzdeltas)
In Abb. 5.7 wurde Knoten C gelöscht und ein Knoten G zwischen Knoten D und E hinzugefügt. Links sieht man die resultierende einfache Substitution, rechts die hierarchische Substitution. Die resultierende instanzspezifische Vorlage ergibt sich bei der hierarchischen Substitution aus der Überlagerung aller Substitutionsobjekte in der Reihenfolge ihrer Erzeugung.

Genau dies entspricht der Funktionsweise der einfachen Substitution. Hierbei wird bei einer Änderung unterschieden, ob sie sich auf eine bereits vorhandene Änderung bezieht und dann zu einer Modifikation des Substitutionsobjekts führt, oder ob eine Änderung einen unveränderten Bereich der ursprünglichen Vorlage betrifft und das Substitutionsobjekt entsprechend erweitert wird. Damit ist die Durchführung von Änderungen aufwendiger als bei hierarchischer Substitution, es wird jedoch weniger Speicher benötigt und der Zugriff auf die instanzspezifische Prozessvorlage ist wesentlich effizienter, da nicht mehrere Substitutionsobjekte unterschieden und überlagert werden müssen. Insgesamt ist die einfache Substitution deshalb vorzuziehen.

5.2 Qualitativer Vergleich der Realisierungskonzepte

Tabelle 5.1 zeigt einen qualitativen Vergleich der vorgestellten Realisierungskonzepte bezüglich verschiedener Anforderungen. Am kritischsten sind der Primärspeicherbedarf sowie die Laufzeit der verschiedenen Realisierungen. Den Primärspeicherbedarf werden wir anschließend noch im Detail qualitativ untersuchen. Beide Aspekte sind Gegenstand umfangreicher Messungen in Kapitel 8.

Tabelle 5.1: Bewertung der Realisierungskonzepte instanzspezifischer Änderungen

	ÄR-1 Individuelle Instanzgraphen	ÄR-2 Operationale Instanzdeltas	ÄR-3 Strukturelle Instanzdeltas
Primärspeicherbedarf	-	++	+
Laufzeit	++	--	+
Transparenz	++	-	O
Partitionierung	+	-	O
Schemaevolution	-	++	O
Anfragen an Kollektionen von Instanzen	++	-	+

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

Der Primärspeicherbedarf bei Variante ÄR-1 ist insgesamt hoch. Insbesondere bei geringfügigen Änderungen, etwa beim Löschen einzelner Knoten, wird viel Information redundant im Vergleich zur ursprünglichen Prozessvorlage gespeichert. Diesen Nachteil vermeiden operationale und strukturelle Instanzdeltas; hier wird deutlich weniger Speicherplatz benötigt. Weitgehend umgekehrt verhält es sich bezüglich der Laufzeit. Variante ÄR-2 ist ineffizient, da die Änderungen nicht strukturell vorliegen und erst rekonstruiert werden müssen. Der Aufwand hängt dabei von der Art und der Komplexität der Änderungen ab. Einfache Änderungen, wie das Löschen eines Knotens, sind noch relativ effizient handhabbar. Jedoch wird das Laufzeitverhalten bei mehreren komplexen Änderungen schnell problematisch. Wesentlich besser verhält sich dagegen Variante ÄR-3. Hier liegen die Änderungen schon in Graphstrukturen vor. Es muss bei Zugriffen lediglich noch eine Substitution vorgenommen werden. Dagegen unterscheidet sich der Zugriff auf instanzspezifische Prozessvorlagen bei Variante ÄR-1 nicht vom Zugriff auf Prozessvorlagen unveränderter Prozessinstanzen. Dies ist bezüglich der Laufzeit am effizientesten.

Auch die weiteren identifizierten Anforderungen werden von den Realisierungskonzepten unterschiedlich unterstützt. Bei Variante ÄR-1 sind Änderungen vollständig transparent und deshalb mit der Realisierung unveränderter Instanzvorlagen vergleichbar. Sie bietet daher bezüglich Transparenz, Partitionierung und Anfragen an Kollektionen von Prozessinstanzen gute bis sehr gute Unterstützung. Dagegen ist eine Partitionierung von Variante ÄR-2

schwierig, da aus der Änderungsoperation selbst nicht sofort hervorgeht, welche Bereiche von ihrer Anwendung betroffen sind. Bei Variante ÄR-3 ist der Änderungskontext dagegen bekannt, die Substitutionsobjekte bilden jedoch nicht unbedingt einen zusammenhängenden Prozessgraphen, weshalb sich die Partitionierung etwas aufwendiger als bei unveränderten Prozessvorlagen gestaltet. Ähnlich verhält es sich bei Anfragen an Kollektionen von Prozessinstanzen. Diese sind bei Variante ÄR-2 schwierig in der Umsetzung, da die Auswirkungen einzelner Änderungsoperationen nicht absehbar sind. Bei Variante ÄR-3 muss nur für diejenigen Anfragen die Substitution vorgenommen werden, die sich auf einen geänderten Bereich der zugrundeliegenden Vorlage beziehen.

Große Auswirkungen hat die Wahl des Realisierungskonzepts für instanzspezifische Änderungen auf die Durchführung einer Schemaevolution. Hierfür ist Variante ÄR-2 am besten geeignet, da der Vergleich der Änderungen von Prozessvorlage und -instanz direkt erfolgen kann. Wird anschließend die Migration durchgeführt, müssen die instanzspezifischen Änderungen, ebenso wie die entsprechende Änderungshistorie, logisch angepasst werden, da sie dann relativ zur neuen Prozessvorlage benötigt werden. Da Variante ÄR-2 direkt über die Änderungshistorie realisiert ist, müssen neben der Änderungshistorie keine weiteren Datenstrukturen für die Repräsentation der instanzspezifischen Änderungen angepasst werden. Dagegen sind sowohl bei Variante ÄR-1 als auch bei Variante ÄR-3, zusätzlich zur Manipulation der Änderungshistorie, strukturelle Anpassungen nötig. Diese gestalten sich bei strukturellen Deltas (ÄR-3) jedoch einfacher, da die entsprechenden Datenstrukturen direkt die Unterschiede zwischen der ursprünglichen und der instanzspezifischen Prozessvorlage repräsentieren, während bei Variante ÄR-1 eine geänderte Prozessinstanz unabhängig von der ursprünglichen Prozessvorlage ist. Durch diese Unabhängigkeit können die Unterschiede nur durch einen Vergleich der vollständigen Objekte abgeleitet werden.

5.3 Quantitativer Vergleich der Realisierungskonzepte

Dem qualitativen Vergleich der Realisierungskonzepte für veränderte Prozessinstanzen folgt in diesem Abschnitt ein quantitativer Vergleich bezüglich des Speicherbedarfs. Der Speicherbedarf hängt von verschiedenen Größen ab (s. Tabelle 5.2). Dazu gehören beispielsweise Anzahl und Art der Änderungen sowie die Beziehungen der Änderungen zueinander. So ist es hinsichtlich des Speicherbedarfs ein großer Unterschied, ob zwei Knoten sequentiell oder aber je ein Verzweigungs- und Vereinigungsknoten eingefügt werden. Während zwei sequentielle Knoten nur Speicher für die Knoten selbst und die entsprechenden Kanten benötigen, entstehen bei neuen Verzweigungs-/Vereinigungsknoten neue Blockstrukturen, die zusätzlich Speicherbedarf benötigen. Je nach Realisierungskonzept hängt der Speicherbedarf auch von der Beziehung der Änderungen ab. So kann es einen Unterschied machen, ob ein Knoten in einem bisher unveränderten Bereich der Prozessstruktur eingefügt wird oder direkt neben einem bereits geänderten Bereich. Abhängig vom Realisierungskonzept sind dann mehr oder weniger neue Kanten notwendig, um die Änderung zu repräsentieren.

Um die Komplexität des quantitativen Vergleichs zu verringern, abstrahieren wir nachfolgend von solchen Details. Wir betrachten nur den Speicherbedarf für neu hinzugefügte, verschobene und gelöschte Knoten in Sequenzen, wobei die Änderungen als unabhängig von anderen Änderungen gelten. Ebenso abstrahieren wir von komplexen Änderungen etwa der Änderung von Blockstrukturen. Andernfalls würden sich komplexe Formeln ergeben, die sehr spezifisch

Tabelle 5.2: Relevante Größen für Speicherbedarf von Realisierungkonzepten für veränderte Prozessinstanzen

$M_{ChangeEntry}$: Speicherbedarf für einen Eintrag in der Änderungshistorie

M_{Edge}: Speicherbedarf für eine Kante einer Prozessvorlage, unabhängig von ihrer Repräsentation

$M_{InstanceChanges}$: Speicherbedarf für die Repräsentation der geänderten Strukturen einer Prozessinstanz gegenüber der ursprünglichen Prozessvorlage

M_{Node}: Speicherbedarf für einen Knoten einer Prozessvorlage

$N_{AddedNodes}$: Anzahl der hinzugefügten Knoten einer Prozessinstanz

N_{Edges}: Anzahl der Kanten pro Prozessvorlage

$N_{MovedNodes}$: Anzahl der verschobenen Knoten einer Prozessinstanz

N_{Nodes}: Anzahl der Knoten pro Prozessvorlage

$N_{RemNodes}$: Anzahl der gelöschten Knoten einer Prozessinstanz

für die jeweilige Änderungsrepräsentation ist. Solche Formeln lassen sich schlecht miteinander vergleichen.

Die Änderung der Knotenanzahl durch instanzspezifische Änderungen wirkt sich auch auf den Speicherbedarf für die Zustandsrepräsentation der entsprechenden Prozessinstanzen aus (s. Abschnitt 4.1.4). Da diese Auswirkungen bei allen Realisierungskonzepten gleich sind, berücksichtigen wir sie nicht weiter und geben nur den Speicherbedarf für die Strukturinformationen der einzelnen Realisierungskonzepte $M_{InstanceChanges}$ an.

5.3.1 Variante ÄR-1: Individuelle Instanzgraphen

Der Speicherbedarf $M_{InstanceChanges}$ für eine Prozessinstanz mit Prozessänderungen repräsentiert durch einen individuellen Instanzgraphen ergibt sich aus dem Speicherbedarf für die entsprechende Prozessvorlage. Dieser setzt umfasst alle Knoten ($N_{Nodes} * M_{Node}$) und Kanten ($N_{Edges} * M_{Edge}$) der Prozessvorlage. Hierzu kommen die hinzugefügten Knoten ($N_{AddedNodes} * M_{Node}$) sowie eine Kante pro hinzugefügtem Knoten ($N_{AddedNodes} * M_{Edge}$). Durch gelöschte Knoten wird Speicher für die Knoten selbst ($N_{RemNodes} * M_{Node}$) und eine Kante pro gelöschtem Knoten eingespart ($N_{RemNodes} * M_{Edge}$). Verschobene Knoten ändern den Speicherbedarf gegenüber einer unveränderten Prozessinstanz nicht.

Formel 5.1 (Speicherbedarf für individuelle Instanzgraphen (ÄR-1))

$$\begin{aligned} M_{InstanceChanges} &= (N_{Nodes} + N_{AddedNodes} - N_{RemNodes}) * M_{Node} \\ &+ (N_{Edges} + N_{AddedNodes} - N_{RemNodes}) * M_{Edge} \end{aligned}$$

5.3.2 Variante ÄR-2: Operationale Instanzdeltas

Bei operationalen Instanzdeltas ergibt sich der Speicherbedarf für die Änderungen an einer Prozessinstanz durch die Größe der entsprechenden Änderungshistorie. Die Änderungshistorie umfasst alle hinzugefügten ($N_{AddedNodes}$), verschobenen ($N_{MovedNodes}$) und gelöschten ($N_{RemNodes}$) Knoten. Dabei macht die Art der Änderung – zumindest unter den geltenden Vereinfachungen – keinen Unterschied. Für jeden geänderten Knoten existiert ein Eintrag in der Änderungshistorie der Größe $M_{ChangeEntry}$.

Formel 5.2 (Speicherbedarf für operationale Instanzdeltas (ÄR-2))

$$M_{InstanceChanges} = (N_{AddedNodes} + N_{MovedNodes} + N_{RemNodes}) * M_{ChangeEntry}$$

5.3.3 Variante ÄR-3: Strukturelle Instanzdeltas

Bei strukturelle Instanzdeltas wird wie bei operationalen Instanzdeltas nur für die Änderungen Speicherplatz benötigt, nicht jedoch für unveränderte Prozessstrukturen. Allerdings kommt bei jedem Knoten der entsprechende Änderungskontext hinzu. Bei hinzugefügten Knoten ($N_{AddedNodes}$) besteht der Änderungskontext aus den zwei umgebenden Knoten sowie zwei Kanten zwischen diesen Knoten und dem neu hinzugefügten Knoten. Da der hinzugefügte Knoten selbst auch Speicher benötigt, beläuft sich der Speicherbedarf auf $3 * M_{Node} + 2 * M_{Edge}$ pro hinzugefügtem Knoten. Bei gelöschten Knoten ($N_{RemNodes}$) besteht der Änderungskontext aus den zwei umgebenden Knoten und der (neuen) Kante, die beide verbindet ($2 * M_{Node} + M_{Edge}$). Das Verschieben eines Knotens benötigt Speicherbedarf wie das Löschen und Hinzufügen des Knotens. Verschobene Knoten zählen damit sowohl zu hinzugefügten Knoten ($N_{AddedNodes} + N_{MovedNodes}$) als auch zu gelöschten Knoten ($N_{RemNodes} + N_{MovedNodes}$).

Formel 5.3 (Speicherbedarf für strukturelle Instanzdeltas (ÄR-3))

$$\begin{aligned} M_{InstanceChanges} &= (N_{AddedNodes} + N_{MovedNodes}) * (3 * M_{Node} + 2 * M_{Edge}) \\ &+ (N_{RemNodes} + N_{MovedNodes}) * (2 * M_{Node} + M_{Edge}) \end{aligned}$$

Berücksichtigt man Abhängigkeiten zwischen Änderungen, so reduziert sich gewöhnlich der Änderungskontext, da sich die einzelnen Änderungskontexte überschneiden. Diese komplexen Auswirkungen sind in Formel 5.3 nicht abgebildet.

5.3.4 Fazit

Tabelle 5.3 zeigt beispielhaft den Speicherbedarf für die Realisierungsvarianten. Dabei wurden feste Werte für den Speicherbedarf eines Knotens, einer Kante sowie eines Eintrags in der Änderungshistorie angenommen. Variiert wurden die Größe der Prozessvorlage (20 oder 100 Knoten) sowie die Anzahl und Art der durchgeführten Änderungen. „Wenig Änderungen“ umfassen 4 hinzugefügte, 2 gelöschte und 1 verschobenen Knoten, „viele Änderungen“ entsprechen 12 hinzugefügter, 10 gelöschter und 6 verschobener Knoten.

Während bei Variante ÄR-2 und ÄR-3 die Größe der Prozessvorlage irrelevant ist und der Speicherbedarf nur von den Änderungen abhängt, spielen die Änderungen bei Variante ÄR-1 eine untergeordnete Rolle. Bei Variante ÄR-1 bestimmt sich der Speicherbedarf nur durch die Anzahl der Knoten nach den Änderungen, also die Größe der Prozessvorlage zuzüglich der

Tabelle 5.3: Speicherbedarf der verschiedenen Realisierungsvarianten für instanzspezifische Änderungen

	kleine Prozeßvorlage 20 Knoten, wenig Änderungen	große Prozeßvorlage 100 Knoten, wenig Änderungen	kleine Prozeßvorlage 20 Knoten, viele Änderungen	große Prozeßvorlage 100 Knoten, viele Änderungen
Individuelle Instanzgraphen (ÄR-1)	4.752 Byte	22.032 Byte	4.752 Byte	22.032 Byte
Operationale Instanzdeltdas (ÄR-2)	280 Byte	280 Byte	1.120 Byte	1.120 Byte
Strukturelle Instanzdeltas (ÄR-3)	4.408 Byte	4.408 Byte	18.032 Byte	18.032 Byte

hinzugefügten und abzüglich der gelöschten Knoten. Variante ÄR-2 benötigt erwartungsgemäß sehr wenig Speicher. Dagegen ist der Speicherbedarf für Variante ÄR-3 sehr groß. Dabei werden allerdings die Abhängigkeiten zwischen Änderungen nicht berücksichtigt. Betreffen zwei Änderungen beispielsweise denselben Änderungskontext, belegt der Änderungskontext nur einmal Speicher. In der einfachen Formel 5.3 ist dies jedoch nicht berücksichtigt. Aufgrund des Prinzips struktureller Instanzdeltas kann deren Speicherbedarf nie größer werden als der von individuellen Instanzgraphen.

Der Speicherbedarf für viele Änderungen bei kleiner Prozessvorlage zeigt sehr deutlich, dass die Formel für Variante ÄR-3 eine zu grobe Näherung ist. In diesem Fall sind 16 Knoten der ursprünglichen Prozessvorlage von Änderungen betroffen. Da diese Prozessvorlage aber nur 20 Knoten besitzt, gibt es zwangsweise Überlappungen bei den Änderungskontexten. Im nächsten Abschnitt gehen wir detailliert auf Variante ÄR-3 ein und diskutieren zwei Implementierungsvarianten. In Kapitel 8 messen wir zudem die Laufzeit der Realisierungsvarianten, da der Speicherbedarf als alleiniges Kriterium nicht ausreicht. Hier wird insbesondere die Laufzeit von ÄR-2 zu beachten sein, da deren Speicherbedarf im Zuge der vorherigen qualitativen Betrachtungen sehr niedrig ausfällt.

5.4 Implementierungsvarianten struktureller Instanzdeltas

Nachdem sich im vorherigen Abschnitt das auf logischer Ebene angesiedelte Realisierungskonzept der strukturellen Instanzdeltas als vielversprechend aber nicht einfach quantifizierbar herausgestellt hat, untersuchen und vergleichen wir in diesem Abschnitt verschiedene Implementierungsvarianten hierzu. Konkret sind drei Möglichkeiten denkbar (s. Abb. 5.1 und Abb. 5.8):

- *Temporäre Realisierung*: Vor einem Zugriff wird die vollständige logische Prozessvorlage physisch realisiert. Dadurch funktionieren die Zugriffe (z. B. beim Weiterschalten im Prozess) wie bei unveränderten Prozessinstanzen.

- *Substitutionsvorlagen*: Hierbei wird vor jedem Zugriff auf einen Knoten oder eine Kante überprüft, ob das entsprechende Element ein Bestandteil der strukturellen Instanzdeltas ist. Dementsprechend findet der Zugriff dann auf den Instanzdeltas (eben den Substitutionsvorlagen) oder der ursprünglichen Prozessvorlage statt.

- *Gekapselte Vorlagen*: Jeder Zugriff auf einen Knoten oder eine Kante findet sowohl auf der ursprünglichen als auch auf den strukturellen Instanzdeltas statt. Beide Zugriffe liefern ein Ergebnis, dessen Vereinigung anschließend das eigentliche Resultat des Zugriffs bildet.

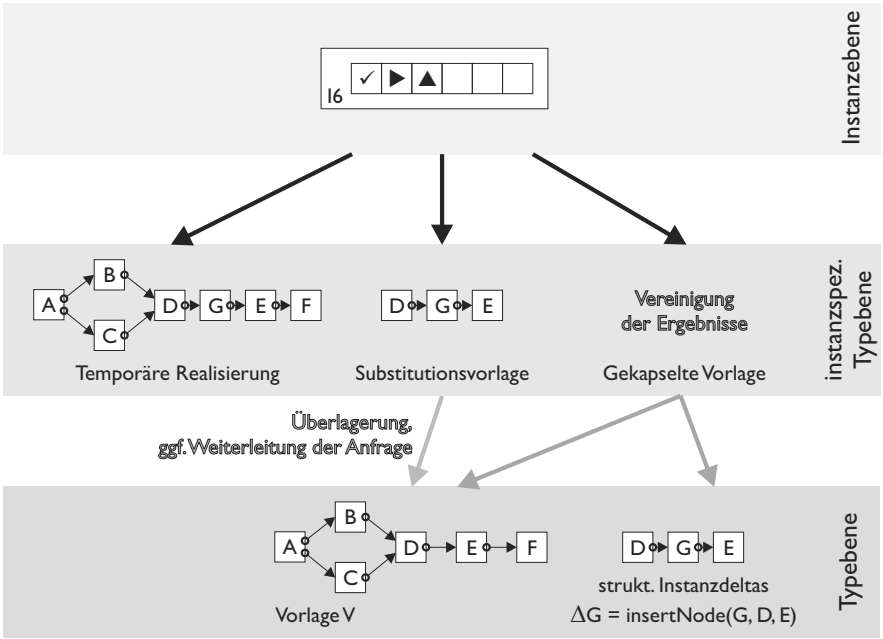


Abbildung 5.8: Ermittlung der Prozessstruktur bei den Implementierungsvarianten

Da bei Zugriffen nicht immer die gesamte Prozessvorlage benötigt wird, sondern oftmals nur einzelne Teile des Prozessgraphen, ist die temporäre Realisierung der vollständigen Prozessvorlage unnötig aufwendig, sowohl bezüglich Speicherbedarf als auch hinsichtlich der für die Rekonstruktion benötigten Laufzeit. Im folgenden werden wir deshalb nur Substitutionsvorlagen und gekapselte Vorlagen weiter betrachten und ihre Funktionsweise anhand repräsentativer Beispiele verdeutlichen. Abschließend folgen ein qualitativer und ein quantitativer Vergleich der Implementierungsvarianten. Eine Simulation der Speicherrepräsentationen findet sich in Kapitel 8.

5.4.1 Variante ÄR-3-1: Substitutionsvorlagen

Implementierungsvariante ÄR-3-1 (Substitutionsvorlagen) ist dadurch gekennzeichnet, dass eine geänderte Prozessinstanz zwei zugrundeliegende Prozessvorlagen besitzt und diese beide referenziert, d. h. die ursprüngliche Prozessvorlage und das Substitutionsobjekt (*Teilvorlage*). Dabei existiert bei einfacher Substitution höchstens ein Substitutionsobjekt, bei hierarchischer Substitution sind es mehrere (s. Abb. 5.7). Zusätzlich ist bei der Prozessinstanz hinterlegt, welche (Teil-) Vorlage für welchen Bereich oder welche Knoten gültig ist (s. Abb. 5.9). Da bei unveränderten Prozessinstanzen nur eine gültige Vorlage existiert, kann hier auf diese Information verzichtet werden.

Um die Funktionsweise von Substitutionsvorlagen zu verdeutlichen, werden wir im folgenden die konkrete Realisierung einer unveränderten Prozessinstanz, einer Prozessinstanz mit gelöschtem Knoten, einer Prozessinstanz mit hinzugefügtem Knoten sowie einer Prozessinstanz mit beiden Änderungen erläutern. Der gelöschte Knoten ist so gewählt, dass daraus eine komplexe Änderung resultiert, da durch das Löschen des Knotens ein leerer Zweig in einer Verzweigung entsteht. Da dies kein gültiges Prozessmodell darstellt, wird der leere Zweig im Rahmen derselben Änderungsoperation einfach mitgelöscht. Dies verdeutlicht die Realisierung von

Substitutionsvorlagen bei komplexen Änderungen. Der zweite Fall stellt eine einfache Änderung dar; dabei wird ein Knoten in einer Sequenz eingefügt. Der dritte Fall bildet die Vereinigung der Änderungen aus Fall 1 und 2 und stellt eine komplexe Änderung dar, die sowohl die vorderen als auch die hinteren Bereiche der Prozessvorlage betrifft. Das Verschieben von Knoten wird nicht explizit betrachtet, da dies durch die Verknüpfung von Löschen und Hinzufügen realisiert wird. Die aus diesen Änderungen resultierenden vollständigen Prozessvorlagen sind in Abb. 5.2 links unten zu sehen.

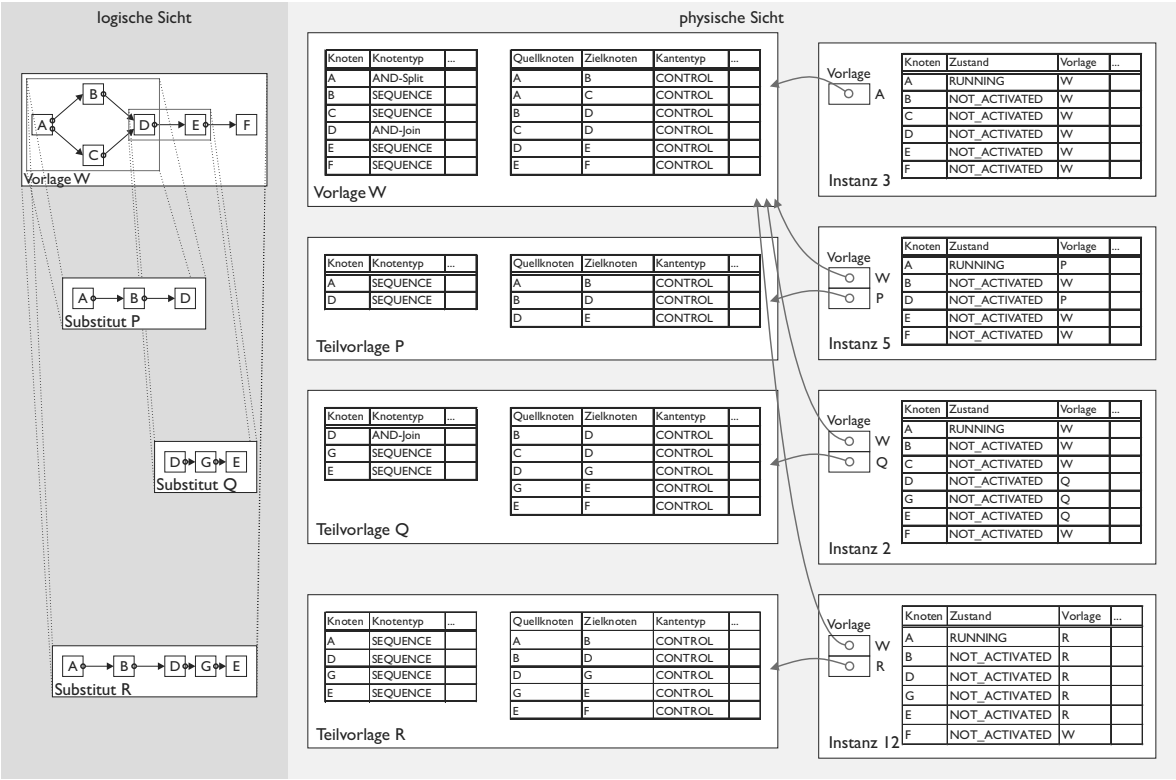


Abbildung 5.9: Realisierung von Substitutionsvorlagen (ÄR-3-1)

Abb. 5.9 zeigt die logische und physische Sicht auf eine unveränderte Prozessinstanz (Instanz 3) und zwei geänderte Prozessinstanzen (Instanzen 5 und 2) sowie die entsprechenden Substitute (logische Sicht) bzw. (Teil-) Vorlagen (physische Sicht). In Instanz 5 wurde Knoten C zusammen mit dem entsprechenden Teilzweig einer Verzweigung gelöscht, was sich in Substitut P widerspiegelt (linke Seite von Abb. 5.9). Für die Realisierung dieser Änderung wurden alle betroffenen Knoten mit sämtlichen Attributen sowie alle ein- und ausgehenden Kanten von der ursprünglichen Prozessvorlage auf eine neue Teilvorlage P kopiert und dort geändert. Ein Knoten ist von einer Änderung betroffen, wenn seine Attribute oder die Menge seiner ein- und ausgehenden Kanten modifiziert wird.

Beim Löschen von Knoten C entsteht ein leerer Zweig, der ebenfalls entfernt wird. Dadurch werden die Knoten A und D modifiziert. Diese ändern ihren Typ von Verzweigungs- bzw. Vereinigungsknoten zu einfachen Sequenzknoten.[1] Die Kanten A–B, B–D und D–E werden nicht modifiziert, sie werden aber bei Teilvorlage P benötigt, um den Übergang vom Substitut P

[1] Dabei ist es unerheblich, ob es sich um parallele oder alternative Verzweigungen handelt.

zur ursprünglichen Vorlage W zu realisieren (sog. *Kontextkanten*). Ohne die Kante D–E würde die für den Knoten D gültige Teilvorlage P keinen Nachfolger für D liefern. Das liegt daran, dass die Information über die jeweils gültige (Teil-) Vorlage in Instanzobjekten (rechte Spalte in Abb. 5.9) nur für Knoten existiert, nicht aber für Kanten.

Nach Erzeugung der Substitutionsvorlage wird noch das Instanzobjekt angepasst (s. Abb. 5.9, rechts in der Mitte). Dazu wird Teilvorlage P als gültige Vorlage bei Knoten A und D hinterlegt. Knoten C wird aus der Zustandsliste gelöscht und ist damit nicht mehr erreichbar. Es existieren zwar noch die Kanten A–C und C–D in Vorlage W, diese werden jedoch nicht mehr berücksichtigt, da Zugriffe auf Knoten A und D ausschließlich über die Teilvorlage P erfolgen, die diese Kanten aber nicht besitzt.

Beim Hinzufügen eines neuen Knotens G bei Instanz 2 werden neben G genau diejenigen Knoten auf Teilvorlage Q kopiert, deren Attribute oder ein- und ausgehende Kanten modifiziert werden. Dies sind im vorliegenden Fall die Knoten D und F. Zusätzlich werden noch die zugehörigen Kanten (d. h. B–D, C–D und E–F) sowie die neu hinzugekommenen Kanten (D–G, G–E) übernommen. Analog zum Löschen von Knoten C bei Instanz 5 muss die Kante D–E der Vorlage W bei Instanz 2 nicht weiter berücksichtigt werden, da sie nach den entsprechenden Anpassungen im Instanzobjekt nicht mehr erreicht werden kann. Diese Anpassungen beschränken sich auf das Einfügen des Knotens G in die Zustandsliste sowie das Hinterlegen von Teilvorlage Q bei den Knoten D, G und E (s. Abb. 5.9, rechts unten).

Bei einem Zugriff auf eine geänderte Prozessinstanz muss vor jedem einzelnen Knotenzugriff auf die (logische) instanzspezifische Prozessvorlage überprüft werden, welche (Teil-) Vorlage für den entsprechenden Knoten gültig ist. Beispielsweise verläuft die Bestimmung der transitiven Nachfolger von Knoten B in Instanz 2 folgendermaßen: Zuerst wird die für Knoten B gültige Vorlage bestimmt. In der Zustandstabelle ergibt sich hierfür Vorlage W (3. Spalte). In Vorlage W wiederum ergibt sich aufgrund der Kontrollkante B–D der Knoten D als Nachfolger von B. Für den Knoten D ergibt sich aus Instanz 2 die Teilvorlage Q. Aus dieser wiederum wird aufgrund der Kontrollkante D–G der Knoten G als Nachfolger von D bestimmt. Auch für Knoten G gilt, wie in Instanz 2 hinterlegt, die Teilvorlage Q. Hieraus lässt sich der Nachfolger E ermitteln. Über die Kontextkante E–F von Teilvorlage Q ergibt sich der Knoten F als Nachfolger von E. Für Knoten F gilt die ursprüngliche Vorlage W. Da dort Knoten F der Endknoten ist, gibt es keine weiteren Nachfolgerknoten mehr.

An diesem Beispiel wird die Funktionsweise von Kontextkanten deutlich. Diese realisieren den Übergang zwischen Vorlage W und Teilvorlage Q. Dementsprechend sind alle Kanten, die in einen geänderten Bereich ein- und ausgehen, sowohl in der ursprünglichen Prozessvorlage als auch im Substitutionsobjekt hinterlegt.

Instanz 12 schließlich vereinigt das Löschen von Knoten C und das Hinzufügen von Knoten G. Wie aus Abb. 5.9 ersichtlich, gilt hier für die meisten Knoten die Teilvorlage R. Dies liegt daran, dass die ursprüngliche Prozessvorlage relativ klein und damit der vordere und hintere Teil der Prozessvorlage dicht beieinander liegen. Bei einer größeren Prozessvorlage gälte für mittlere Knoten durchaus noch Vorlage W. Abb. 5.9 macht auch deutlich, dass die Knoten- und Kantenmengen in Teilvorlage R nicht ganz die Vereinigungen der Knoten- und Kantenmengen der Teilvorlagen P und Q sind. Knoten D ist von beiden Änderungen betroffen, da er Kontextknoten ist. Naheliegenderweise existiert er in Teilvorlage R aber nur einmal. Außerdem gibt es durch das Löschen von Knoten C in Teilvorlage R die Kontextkante C–D nicht. Auch die Kante D–E existiert nicht, da sie durch die Kante D–G ersetzt wird.

5.4.2 Variante ÄR-3-2: Gekapselte Vorlagen

Bei Implementierungsvariante ÄR-3-2 (gekapselte Vorlagen) besitzt das Substitutionsobjekt (*Wrapper*) dieselbe Schnittstelle wie eine vollständige Prozessvorlage. Dies entspricht den bekannten Software-Entwurfsmustern *Decorator* bzw. *Wrapper* [GHJV95]. Dadurch besteht für eine Prozessinstanz kein Unterschied zwischen Zugriffen auf Prozessvorlage und Wrapper, was zu vollständiger Transparenz führt (s. Abb. 5.10). Die in einem Wrapper gekapselten instanzspezifischen Änderungen werden direkt bei Zugriffen berücksichtigt. Hierzu wird eine Anfrage sowohl vom Vorlagenobjekt als auch vom Wrapper bearbeitet, und die beiden Teilergebnisse werden anschließend vereinigt.

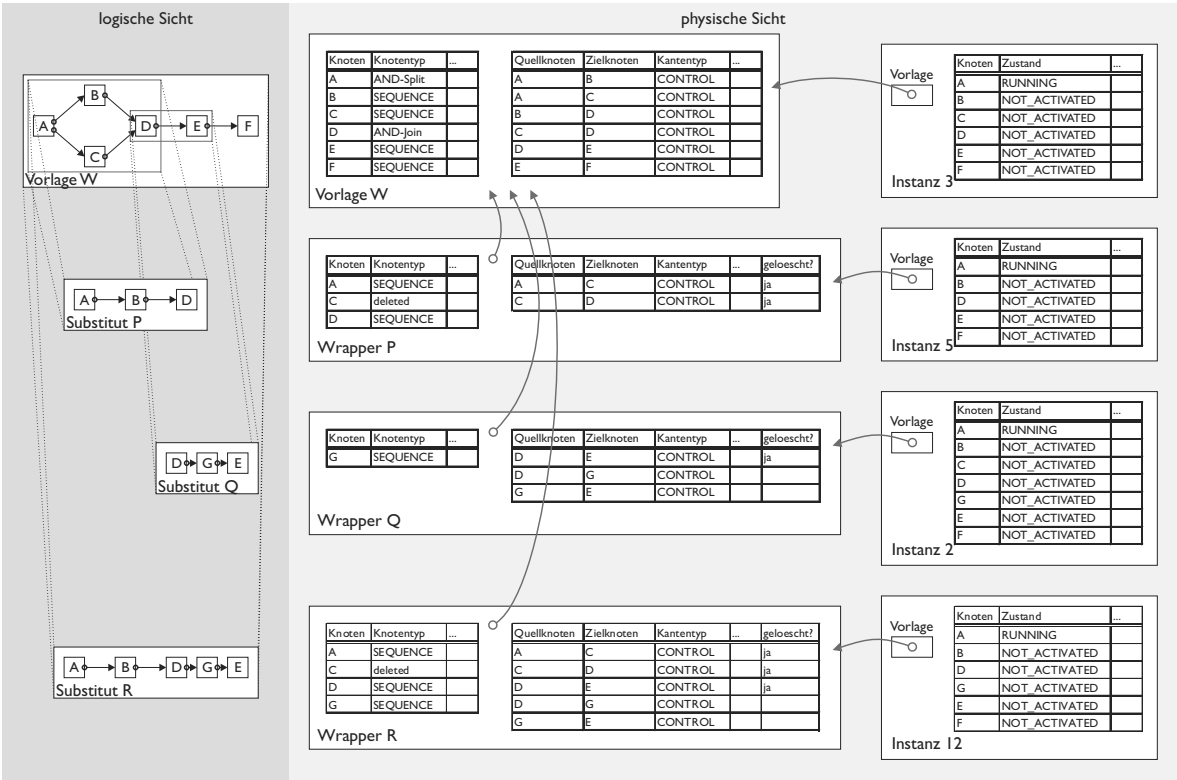


Abbildung 5.10: Realisierung von gekapselten Vorlagen (ÄR-3-2)

Bei der Durchführung instanzspezifischer Änderungen werden wie bei Implementierungsvariante ÄR-3-1 (Substitutionsvorlagen) alle Knoten mit Änderungen in das Substitutionsobjekt (in diesem Fall der Wrapper) übernommen. Bei gekapselten Vorlagen findet jedoch keine Substitution ganzer Graphbereiche statt, sondern es werden einzelne Knoten und Kanten ersetzt.

Beim Löschen von Knoten C in Instanz 5 (s. Abb. 5.10) werden die Knoten A und D in den Wrapper P übernommen und jeweils die Änderung des Knotentyps vermerkt (s. resultierende Datenstrukturen in Abb. 5.10, Wrapper B) . Zusätzlich werden die gelöschten Knoten und Kanten (im Beispiel Knoten C und Kanten A–C und C–D) explizit verwaltet und damit auch in Wrapper B übernommen. Dafür gibt es im Gegensatz zu Variante ÄR-3-1 (Substitutionsvorlagen) jedoch keine Redundanz zwischen ursprünglicher Prozessvorlage und Wrapper (z. B. Kanten A–B und B–D), da keine Kontextkanten benötigt werden. In Instanz 5

wird abschließend noch Knoten C aus der Zustandstabelle entfernt (s. Abb. 5.10, rechts in der Mitte). Auch bei den Instanzobjekten wird bei Variante ÄR-3-2 weniger Speicherplatz benötigt als bei Implementierungsvariante ÄR-3-1, da nur eine Prozessvorlage referenziert und bei Knoten keine Unterscheidung bezüglich der gültigen (Teil-) Vorlage benötigt wird. In Abb. 5.10 spiegelt sich dies im Fehlen der entsprechenden Spalte wider.

Das Hinzufügen eines Knotens ist bei gekapselten Vorlagen einfacher als bei Substitutionsvorlagen (s. Abb. 5.9 und Abb. 5.10). Um den neuen Knoten G zwischen D und E in Instanz 2 einzufügen, wird Knoten G zusammen mit den ein- und ausgehenden Kanten (D–G, G–E) in Wrapper Q hinterlegt. Zusätzlich wird noch die ursprüngliche Kante D–E als gelöscht markiert. Die Änderung wird mit der Aufnahme von G in die Zustandstabelle von Instanz 2 abgeschlossen (s. Abb. 5.10, rechts unten).

Die Bestimmung der transitiven Nachfolger von Knoten B der Instanz 2 werden wie bei Implementierungsvariante ÄR-3-1 iterativ bestimmt. Dazu werden in jedem Schritt die (direkten) Nachfolger des jeweiligen Knotens sowohl mittels der ursprünglichen Prozessvorlage (Vorlage W) als auch mittels Substitutionsobjekt (Wrapper Q) bestimmt, und die einzelnen Ergebnisse werden entsprechend zusammengefasst. Aufgrund von Vorlage W und Kante B-D ergibt sich der Knoten D als Nachfolger von B, während Wrapper Q keine von B ausgehende Kontrollkante besitzt und somit auch keinen Nachfolgerknoten für B liefert. Aus beiden Informationen ergibt sich Knoten D als Nachfolger von B. Als Nachfolger von D wiederum liefert Vorlage W den Knoten E, während Wrapper Q den Knoten G als Nachfolger liefert und zusätzlich die Information, dass die Kante D–E gelöscht ist. Somit ist Knoten E nicht mehr länger Nachfolger von Knoten D, und es ergibt sich Knoten G als alleiniger Nachfolger von D. Dessen Nachfolgerknoten wiederum ist der Vorlage W nicht bekannt, wohingegen Wrapper Q den Knoten E ermittelt, der damit der korrekte Nachfolger von G ist. Der Nachfolger von Knoten E wird wie der Nachfolger von Knoten B ermittelt: Vorlage W liefert Knoten F, Wrapper Q kennt keinen Nachfolgerknoten, womit F als (letzter) Nachfolger bestimmt ist.

Bei Anwendung beider Änderungen (Instanz 12) bilden die Knoten- und Kantenmengen von Wrapper R im Gegensatz zur entsprechenden Teilvorlage R bei ÄR-3-1 die echte Vereinigung der Knoten- und Kantenmengen von Wrapper P und Wrapper Q. Dies liegt daran, dass bei gekapselten Vorlagen auch gelöschte Knoten und Kanten explizit vorgehalten werden, während diese bei Variante ÄR-3-1 (Substitutionsvorlagen) gleich entfernt werden. Substitutionsvorlagen sind damit jedoch etwas aufwendiger zu implementieren. Wie in Abb. 5.10 ersichtlich, haben umfangreichere Änderungen auch bei gekapselten Vorlagen keine Auswirkungen auf die Menge der zu verwalteten Daten für die Repräsentation von Änderungen. Das heißt, obwohl sowohl der vordere Teil als auch der hintere Teil der Prozessvorlage von Änderungen betroffen ist, wird die Repräsentation des (logischen) Substitutionsobjekts (in diesem Fall Substitut R) nicht größer, als wenn die Änderungen einen zusammenhängenden Teil der Prozessvorlage beträfen.

Charakteristisch für Variante ÄR-3-2 ist, dass eine Anfrage nicht nur an die gültige (Teil-) Vorlage geht, sondern dass sie sowohl von der ursprüngliche Prozessvorlage als auch vom Wrapper bearbeitet wird. Die ermittelten Ergebnisse werden anschließend vereinigt, wobei das Ergebnis vom Wrapper höhere Priorität hat. Diese Vereinigung ist potentiell aufwendig, da es sich dabei um Mengenoperationen handelt. Allerdings sind die ermittelten Knotenmengen meistens nicht besonders groß: Bei Prozessvorlagen sind die meisten Knoten sequentiell angeordnet, was zu zweielementigen Mengen bei der Bestimmung direkter Vorgänger-/Nachfolgerknoten führt – ein Knoten von der ursprünglichen Vorlage und ein Knoten vom Wrapper. Lediglich bei Verzweigungs-/Vereinigungsknoten werden die Mengen größer, aber auch hier sind mehr

als 2*4 Elemente, d. h. mehr als vier alternative oder parallele Zweige, in der Menge selten. Dadurch bleiben die notwendigen Mengenoperationen effizient.

Insbesondere für sukzessive Zugriffe bei der Durchführung einer komplexen Funktion ist mit Variante ÄR-3-2 eine Verbesserung leicht möglich, indem temporär die logische instanzspezifische Vorlage realisiert wird. Hierzu wird einfach die Vereinigung der gesamten Knoten- und Kantentabellen von Vorlage und Wrapper erzeugt, und die im Wrapper als gelöscht markierten Objekte werden entfernt. Dadurch ergibt sich direkt auf Ebene der Datenstrukturen von Knoten- und Kantenmengen das resultierende vollständige logische Schema, ohne dass Änderungsoperationen aufwendig nachgespielt werden müssen.

5.4.3 Qualitativer Vergleich der Implementierungsvarianten

Sowohl Implementierungsvariante ÄR-3-1 (Substitutionsvorlagen) als auch Implementierungsvariante ÄR-3-2 (gekapselte Vorlagen) realisieren strukturelle Instanzdeltas. Es gibt jedoch bedeutende Unterschiede bezüglich der verwalteten Information und dem Ablauf von Zugriffen (s. Tabelle 5.4). Dies zeigt sich unter anderem im Speicherbedarf und in der Zugriffszeit. So ist bei Variante ÄR-3-1 vor jedem Zugriff auf Vorlageninformationen jeweils ein zusätzlicher Zugriff auf die geänderte Prozessinstanz notwendig, um die gültige (Teil-) Vorlage zu bestimmen. Anschließend kann die gewünschte Information direkt ermittelt werden. Bei Variante ÄR-3-2 sind mehrere Zugriffe nötig, um die jeweilige Information in der Vorlage und im Wrapper zu bestimmen und um die so bestimmten Ergebnisse anschließend geeignet zusammenzufassen, also zusätzliche Knoten aus dem Wrapper der Ergebnismenge hinzuzufügen und im Wrapper als gelöscht markierte Knoten zu entfernen. Wie erwähnt, ist das Zusammenfassen jedoch nicht allzu zeitaufwendig, da die bei einem Wrapper-/Vorlagenzugriff bestimmten Knoten- und Kantenmengen nicht groß sind. Dies liegt daran, dass Prozesse meistens eine eher sequentielle Struktur besitzen. Zudem kann die Situation durch horizontale Partitionierung weiter verbessert werden.

Tabelle 5.4: Bewertung der Implementierungsvarianten für strukturelle Instanzdeltas

	ÄR-3-1 Substitutions-vorlagen	ÄR-3-2 Gekapselte Vorlagen
Primärspeicherbedarf	O	+
Laufzeit	O	-
Transparenz	-	+
Partitionierung	+	O
Schemaevolution	-	+
Anfragen an Kollektionen von Instanzen	O	+

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

Implementierungsvariante ÄR-3-2 benötigt gegenüber Implementierungsvariante ÄR-3-1 weniger Speicherplatz, da nur die Änderungen der ursprünglichen Prozessvorlage verwaltet werden und damit keine Redundanz existiert, wie etwa die Kontextkanten für den Übergang zwischen Vorlage und Teilvorlage. Es reicht sogar aus, nur die wirklich geänderten Attribute eines Knotens oder einer Kante zu speichern und nicht das komplette Knoten-/Kantenobjekt bereitzustellen. Das Vermerken von gelöschten Objekten bei Variante ÄR-3-2 erfordert zwar zusätzlichen Speicherbedarf, jedoch ist dieser nur wenig höher als bei Variante ÄR-3-1 (vgl. Teilvorlage P in Abb. 5.9 und Wrapper P in Abb. 5.10). Zudem wird die Information über

gelöschte Objekte für den Vergleich von Vorlagenänderungen mit Instanzänderungen bei Schemaevolution benötigt. Bei Variante ÄR-3-1 muss diese erst rekonstruiert werden.

Für Implementierungsvariante ÄR-3-2 spricht auch die vollständige Transparenz und die einfache Erzeugung der logischen instanzspezifische Vorlage. Durch die Kapselung ist es leicht möglich, bei großer Anzahl von Änderungen einen Wrapper durch einen vollständigen individuellen Instanzgraphen zu ersetzen (s. Realisierungskonzept ÄR-1, Abschnitt 5.1.1). Aufgrund der mengenorientierten Zugriffe sind bei Variante ÄR-3-2 Anfragen an Kollektionen von Prozessinstanzen leichter durchführbar als bei Variante ÄR3-1.

5.5 Quantitativer Vergleich der Implementierungsvarianten

Für den quantitativen Vergleich der Implementierungsvarianten für strukturelle Instanzdeltas gelten dieselben Voraussetzungen und Einschränkungen wie beim qualitativen Vergleich der Realisierungskonzepte (s. Abschnitt 5.3). Das heißt, wir betrachten nur einfache Änderungen, die keine Abhängigkeiten untereinander haben. Der Speicherbedarf beider Implementierungsvarianten orientiert sich naheliegenderweise am Speicherbedarf des entsprechenden Realisierungskonzepts ÄR-3 (s. Abschnitt 5.3.3). Im Detail ergeben sich jedoch Unterschiede insbesondere hinsichtlich der Größe des Änderungskontexts. Tabelle 5.5 führt die für den Speicherbedarf der Realisierungsvarianten relevanten Größen auf.

Tabelle 5.5: Relevante Größen für Speicherbedarf von Implementierungsvarianten für veränderte Prozessinstanzen

M_{Edge}: Speicherbedarf für eine Kante einer Prozessvorlage, unabhängig von ihrer Repräsentation

$M_{InstanceChanges}$: Speicherbedarf für die Repräsentation der geänderten Strukturen einer Prozessinstanz gegenüber der ursprünglichen Prozessvorlage

M_{Node}: Speicherbedarf für einen Knoten einer Prozessvorlage

$M_{Reference}$: Speicherbedarf für eine Referenz, etwa von einer Prozessinstanz zu der entsprechenden Prozessvorlage bei instanzspezifischen Änderungen oder dem ihren Zustand repräsentierenden Knoten im entsprechenden Zustandsgraphen

$N_{AddedNodes}$: Anzahl der hinzugefügten Knoten einer Prozessinstanz

$N_{MovedNodes}$: Anzahl der verschobenen Knoten einer Prozessinstanz

N_{Nodes}: Anzahl der Knoten pro Prozessvorlage

$N_{RemNodes}$: Anzahl der gelöschten Knoten einer Prozessinstanz

5.5.1 Variante ÄR-3-1: Substitutionsvorlagen

Substitutionsvorlagen benötigen pro Knoten der Instanzvorlage eine Referenz auf die entsprechende Substitutionsvorlage ($M_{Reference}$). Die Gesamtanzahl der Knoten der Instanzvorlage ergibt sich durch die Anzahl der Knoten der Vorlage (N_{Nodes}) zuzüglich der instanzspezi-

fisch hinzugefügten Knoten ($N_{AddedNodes}$) abzüglich der instanzspezifisch gelöschten Knoten ($N_{RemNodes}$).

Für jeden hinzugefügten Knoten wird bei dieser Variante Speicher für die zwei umgebenden Kontextknoten ($2 * M_{Node}$), die Kanten zu und von den Kontextknoten ($2 * M_{Edge}$) sowie die Kanten zwischen den Kontextknoten und dem neuen Knoten ($2 * M_{Edge}$) benötigt. Auch der neue Knoten selbst belegt Speicher (M_{Node}). Bei gelöschten Knoten sind es wieder die zwei umgebenden Kontextknoten ($2 * M_{Node}$), die Kanten zu und von den Kontextknoten ($2 * M_{Edge}$) sowie die Kante zwischen den Kontextknoten, die den gelöschten Knoten ersetzt (M_{Edge}). Verschobene Knoten zählen wieder sowohl zu den hinzugefügten ($N_{AddedNodes} + N_{MovedNodes}$) als auch zu gelöschten Knoten ($N_{RemNodes} + N_{MovedNodes}$). Sie ändern jedoch nicht Gesamtanzahl an Knoten und sind deshalb für den Speicherbedarf der Referenzen nicht relevant.

Formel 5.4 (Speicherbedarf für Substitutionsvorlagen (ÄR-3-1))

$$\begin{aligned} M_{InstanceChanges} &= (N_{Nodes} + N_{AddedNodes} - N_{RemNodes}) * M_{Reference} \\ &+ (N_{AddedNodes} + N_{MovedNodes}) * (3 * M_{Node} + 4 * M_{Edge}) \\ &+ (N_{RemNodes} + N_{MovedNodes}) * (2 * M_{Node} + 3 * M_{Edge}) \end{aligned}$$

5.5.2 Variante ÄR-3-2: Gekapselte Vorlagen

Gekapselte Vorlagen kommen ohne die Referenz auf die entsprechende Substitutionsvorlage aus. Bei ihnen reicht es aus, nur die Kanten zu und von einem neuen Knoten ($2 * M_{Edge}$), den Knoten selbst (M_{Node}) sowie die gelöschte Kante, die durch den neuen Knoten ersetzt wird (M_{Edge}), zu verwalten. Im Gegensatz zu Substitutionsvorlagen werden die umgebenden Knoten nicht benötigt. Dafür benötigen gelöschte Knoten Speicher. Hierfür reicht eigentlich die Kennung (ID) des Knotens sowie ein Gelöscht-Attribut aus. Um die Formel einfach zu halten, berechnen wir jedoch den Speicherplatz für einen kompletten Knoten M_{Node} auch für gelöschte Knoten. Diese Vereinfachung gilt auch für gelöschte Kanten. Das Löschen eines Knotens benötigt Speicher für zwei gelöschte Kanten ($2 * M_{Edge}$) sowie eine neue Kante, die den gelöschten Knoten ersetzt (M_{Edge}).

Der Speicherbedarf für das Verschieben eines Knotens ergibt sich bei dieser Implementierungsvariante aus den geänderten Kanten. Die ursprünglichen Kanten werden gelöscht ($2 * M_{Edge}$) und zwei neue Kanten hinzugefügt ($2 * M_{Edge}$). Damit benötigen verschobene Knoten wesentlich weniger Speicher als bei Variante ÄR-3-1.

Formel 5.5 (Speicherbedarf für gekapselte Vorlagen (ÄR-3-2))

$$\begin{aligned} M_{InstanceChanges} &= N_{AddedNodes} * (M_{Node} + 3 * M_{Edge}) \\ &+ N_{MovedNodes} * (4 * M_{Edge}) \\ &+ N_{RemNodes} * (M_{Node} + 3 * M_{Edge}) \end{aligned}$$

5.5.3 Fazit

Tabelle 5.6 ergänzt Tabelle 5.3 um den Speicherbedarf für die Implementierungsvarianten Substitutionsvorlagen (ÄR-3-1) und gekapselte Vorlagen (ÄR-3-2). Der Speicherbedarf von Variante ÄR-3-1 ist sehr ähnlich zur entsprechenden Realisierungsvariante ÄR-3. Der Mehrbedarf

ergibt sich durch die Vorlagenreferenzen sowie 2 zusätzlichen Kontextkanten pro Änderung. Dagegen hat Variante ÄR-3-2 einen wesentlich geringeren Speicherbedarf und ist deshalb vorzuziehen. Dies liegt daran, dass die Änderungskontexte kleiner sind, da weniger Knoten und Kanten redundant existieren. Variante ÄR-3-1 benötigt diese für den Übergang zwischen Substitutionsvorlage und ursprünglicher Vorlage (und zurück).

Tabelle 5.6: Speicherbedarf der verschiedenen Varianten für instanzspezifische Änderungen

	kleine Prozeßvorlage 20 Knoten, wenig Änderungen	große Prozeßvorlage 100 Knoten, wenig Änderungen	kleine Prozeßvorlage 20 Knoten, viele Änderungen	große Prozeßvorlage 100 Knoten, viele Änderungen
Individuelle Instanzgraphen (ÄR-1)	4.752 Byte	22.032 Byte	4.752 Byte	22.032 Byte
Operationale Instanzdeltdas (ÄR-2)	280 Byte	280 Byte	1.120 Byte	1.120 Byte
Strukturelle Instanzdeltas (ÄR-3)	4.408 Byte	4.408 Byte	18.032 Byte	18.032 Byte
Substitutionsvorlagen (ÄR-3-1)	4.752 Byte	5.072 Byte	19.208 Byte	19.528 Byte
Gekapselte Vorlagen (ÄR-3-2)	1.552 Byte	1.552 Byte	5.840 Byte	5.840 Byte

Es zeigt sich auch hier wieder, dass die Formeln noch nicht aussagekräftig genug sind. Beide Implementierungsvarianten können nicht wesentlich mehr Speicher benötigen als Variante ÄR-1. Sobald die instanzspezifische Vorlage komplett in der Substitutionsvorlage oder im Wrapper vorliegt, entspricht die jeweilige Implementierungsvariante individuellen Instanzgraphen. Lediglich Variante ÄR-3-1 hat durch die Vorlagenreferenz pro Knoten einen geringfügig höheren Speicherbedarf als Variante ÄR-1. Dementsprechend sind auch in diesem Beispiel die Überlappungen bei den Änderungskontexten, insbesondere beim Beispiel einer kleinen Prozessvorlage mit vielen Änderungen, nicht ausreichend berücksichtigt.

5.6 Physische Speicherstrukturen im Sekundärspeicher

Die Repräsentation von veränderten Prozessinstanzen im Sekundärspeicher besitzt gegenüber der Repräsentation unveränderter Prozessinstanzen einige Besonderheiten, die wir in diesem Abschnitt diskutieren. Insgesamt umfasst die Speicherrepräsentation von Prozessvorlagen und -instanzen damit vier verschiedene Bereiche, die in Abb. 5.11 zu sehen sind: Repräsentation in Primär- und Sekundärspeicher von unveränderten (Bereich I und II) und veränderten Prozessinstanzen (Bereich III und IV).

Bei der Repräsentation veränderter Prozessinstanzen im Sekundärspeicher (Bereich IV) kann wie bei unveränderten Prozessinstanzen (Bereich II, s. Abschnitt 4.2.2) zwischen strukturierter und unstrukturierter Speicherung unterschieden werden. Die Repräsentation im Sekundärspeicher (Bereich IV) hängt allerdings stark von der Repräsentation im Primärspeicher ab (Bereich III). Es ist nicht sinnvoll, hier verschiedene Repräsentationen zu verwenden, etwa individuelle Instanzgraphen (ÄR-1) im Primärspeicher und operationale Instanzdeltas (ÄR-2) im Sekundärspeicher. Wir beschränken die Diskussion deshalb auf gleiche Repräsentationen veränderter Prozessinstanzen in Primär- und Sekundärspeicher.

Im Gegensatz dazu sind die Sekundärspeicherrepräsentationen unveränderter und veränderter Prozessinstanzen (Bereich II und IV) voneinander unabhängig. So können veränderte Prozessinstanzen unstrukturiert (z. B. als BLOB) und unveränderte Prozessinstanzen/-vorlagen strukturiert abgelegt werden. Dies ist sinnvoll, wenn die Repräsentation veränderter Prozessinstanzen für viele Zugriffe auch im Primärspeicher (Bereich III) erst transformiert werden müssen, wie beispielsweise bei operationalen Instanzdeltas (ÄR-2). Unterschiedliche Repräsentationen

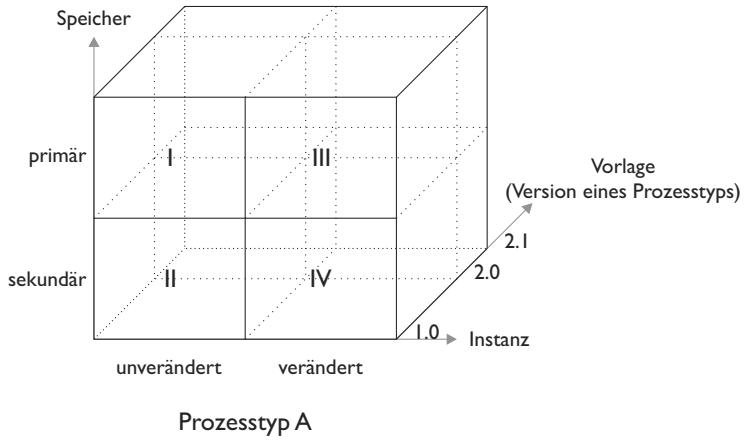


Abbildung 5.11: Dimensionen der Repräsentation von Prozessinstanzen

in den Bereichen II und IV bedeuten allerdings, dass beim Zugriff zwischen unveränderten und veränderten Prozessinstanzen unterschieden werden muss und somit Transparenz verloren geht.

Für die Sekundärspeicherrepräsentation von ÄR-1 (Bereich IV) gelten weitgehend dieselben Überlegungen wie für unveränderte Prozessinstanzen (Bereich II, s. Abschnitt 4.2.2). Eine unstrukturierte Speicherung ist effizient beim kompletten Lesen oder Schreiben, während eine strukturierte Speicherung direkte Anfragen an den Sekundärspeicher erlaubt. Individuelle Instanzgraphen benötigen allerdings viel Speicher. Während man dem Speicherbedarf im Primärspeicher (Bereich III) durch geeignete Partitionierung entgegenwirken kann, ist im Sekundärspeicher jeder individuelle Instanzgraph komplett abgelegt.

Operationale Instanzdeltas (ÄR-2) können aufgrund ihrer Datenstruktur gut strukturiert im Sekundärspeicher repräsentiert werden. Da bei ihnen jedoch keine direkten Zugriffe auf die Prozessstrukturen möglich sind, ist die strukturierte Speicherrepräsentation in der Regel nicht hilfreich. Dies gilt nur für Anfragen, die sich auf Informationen der Änderungshistorie beziehen, etwa nach dem Zeitpunkt einer Änderung oder dem ändernden Benutzer.

Auch für die Sekundärspeicherrepräsentation (Bereich IV) von ÄR-3 gelten wieder ähnliche Überlegungen wie für unveränderte Prozessinstanzen (Bereich II, s. Abschnitt 4.2.2), da vergleichbare Datenstrukturen repräsentiert werden. Im Fall einer strukturierten Repräsentation sind neben den strukturellen Instanzdeltas, die wie unveränderte Prozessvorlagen repräsentiert werden, einige zusätzliche Informationen nötig (s. Abb. 5.9 und Abb. 5.10). Substitutionsvorlagen (ÄR-3-1) benötigen pro Knoten eine Referenz auf die jeweils gültige Vorlage. Da diese Referenz nicht viel zusätzlichen Speicher benötigt, kann sie aus Gründen der Transparenz auch bei unveränderten Prozessinstanzen (Bereich II) existieren. Gekapselte Vorlagen (ÄR-3-2) besitzen bei allen Knoten und Kanten der strukturellen Instanzdeltas ein zusätzliches Attribut, das angibt, ob das jeweilige Objekt gelöscht wurde. Für diese Information wird bei ÄR-3-2 gegenüber der Vorlagenreferenz von ÄR-3-1 weniger Speicherplatz benötigt. Zudem können bei relationaler Speicherung gekapselter Vorlagen zahlreiche Anfragen direkt im Sekundärspeicher ausgeführt werden, da die benötigten Mengenoperationen vom relationalen Datenmodell bereitgestellt werden. So ist es beispielsweise möglich, die strukturellen Instanzdeltas im Sekundärspeicher mit der ursprünglichen Prozessvorlage zu vereinigen und die komplette resultierende Prozessvorlage zurückzugeben.

5.7 Verwandte Arbeiten

Änderungshistorien werden neben der Repräsentationsvariante operationale Instanzdeltas (ÄR-2, s. Abschnitt 5.1.2) auch für *Process Change Mining* [GRMR+08, GRRvdA06] verwendet. Dieses ist ähnlich zu *Process Mining* [vdA11], welches auf Ausführungshistorien beruht. *Process Change Mining* geht jedoch weiter, da es nicht die Prozessstruktur ermittelt sondern aufgrund der Änderungshistorien Prozessverbesserungen und -varianten [Li10, LRW09b] ermittelt. Dies hat keine Auswirkungen auf die Repräsentation veränderter Prozessinstanzen, da die Änderungshistorie aus Gründen der Nachvollziehbarkeit unabhängig von der jeweiligen Repräsentation gespeichert wird (s. Abschnitt 2.2.1.3). Umgekehrt kann das Mining auf Instanzdeltas beruhen, und profitiert dann von der kompakteren Datenrepräsentation gegenüber vollständigen Änderungshistorien.

Eine möglichst kompakte Repräsentation von (Daten-)Änderungen ist bei DBMS von großer Bedeutung. Dies betrifft die Aktualisierung materialisierter Sichten bei Data-Warehouse-Systemen [BG04], die Replikation bei verteilten Datenbanken [Dad96] und Data-Mining [HKP11]. Das Ziel dabei ist eine möglichst redundanzfreie und konsolidierte Datenrepräsentation analog zur Repräsentation veränderter Prozessinstanzen in der vorliegenden Arbeit.

5.8 Zusammenfassung

In diesem Kapitel haben wir die Speicherrepräsentation veränderter Prozessinstanzen unter dem Aspekt der Skalierbarkeit und Leistungsfähigkeit (NA 1) untersucht. Hierzu haben wir verschiedene Realisierungskonzepte diskutiert, um instanzspezifische Änderungen zu repräsentieren. Dabei hat sich die Speicherrepräsentation mit operationalen Instanzdeltas (Variante ÄR-2) als am besten hinsichtlich des Speicherbedarfs herausgestellt. Allerdings hat diese Variante sehr große Auswirkungen auf das Laufzeitverhalten. Den Gegensatz dazu bilden individuelle Instanzgraphen (Variante ÄR-1). Diese Variante hat eine vernachlässigbare Auswirkung auf das Laufzeitverhalten, benötigt aber sehr viel Speicher. Etwas mehr Auswirkungen auf das Laufzeitverhalten haben strukturelle Instanzdeltas (Variante ÄR-3). Dafür benötigt diese Variante etwas weniger Speicher als ÄR-1. Allerdings lässt sich der Speicherbedarf von Variante ÄR-3 durch Formeln nur abschätzen, da das Überlappen von Änderungen sehr komplexe Auswirkungen hat. Entsprechende Formeln wären nur noch wenig aussagekräftig, da sie von zahlreichen Größen und Variablen abhängen. Prinzipbedingt kann der Speicherbedarf von Variante ÄR-3 nicht größer sein als der von Variante ÄR-1.

Während die Varianten ÄR-1 und ÄR-2 direkt implementiert werden können, gibt es für Variante ÄR-3 zwei Implementierungsvarianten: Substitutionsvorlagen (ÄR-3-1) und gekapselte Vorlagen (ÄR-3-2). Sie unterscheiden sich darin, wie die geänderten Graphstrukturen beim Zugriff rekonstruiert werden. Während Variante ÄR-3-1 Änderungen iterativ in die ursprüngliche Prozessvorlage integriert, betrachtet Variante ÄR-3-2 die Änderungen und ursprüngliche Prozessvorlage bei Anfragen getrennt und verknüpft danach die Ergebnismengen der Anfragen geeignet. Der Speicherbedarf von ÄR-3-1 bewegt sich sehr nahe am Speicherbedarf der entsprechenden Realisierungsvariante ÄR-3. Variante ÄR-3-2 ist diesbezüglich wesentlich besser. Dies liegt an der wesentlich geringeren Redundanz. Dafür hat Variante ÄR-3-2 potentiell größere Auswirkungen auf das Laufzeitverhalten aufgrund der mengenorientierte Verarbeitung. Die mengenorientierte Verarbeitung ist jedoch auch ein Vorteil bei relationaler Speicherung von Änderungen im Sekundärspeicher. Das relationale Modell beruht auf Mengen, so dass

die Vereinigung von Änderungen mit den Graphstrukturen der ursprünglichen Prozessvorlage bereits effizient im Sekundärspeicher erfolgen kann.

In Kapitel 8 bewerten wir alle Realisierungs- und Implementierungsvarianten anhand einer Simulation, einerseits bezüglich des Speicherbedarfs, andererseits bezüglich der Laufzeit des Weiterschaltens im Kontrollfluss und der Bestimmung transitiver Vorgänger. Zunächst diskutieren wir in Kapitel 6 einen alternativen Ansatz für die Zustandsrepräsentation und dessen Auswirkungen auf verschiedene Anwendungsfälle sowie instanzspezifischen Änderungen: die Clusterung von Prozessinstanzen aufgrund ihrer Zustände.

6 Clusterung von Prozessinstanzen aufgrund ihrer Zustände

In Kapitel 4 haben wir verschiedene Varianten für die Repräsentation der Zustände von Prozessinstanzen diskutiert. Allen Varianten war gemein, dass die Zustände individuell für jede Prozessinstanz verwaltet werden. Das heißt, jede Prozessinstanz besitzt eine eigene (private) Datenstruktur für ihren Zustand. In diesem Kapitel untersuchen wir die Gruppierung von Prozessinstanzen bezüglich ihres Zustands (*Zustandsclusterung*). Für jede Gruppe (*Zustandscluster*) werden die Zustandsinformationen nur einmal global verwaltet und nicht individuell für jede Prozessinstanz. Dies hat weitgehende Konsequenzen für zahlreiche Funktionen.

In diesem Kapitel stellen wir zuerst das Konzept der Zustandsclusterung vor. Anschließend diskutieren wir drei Anwendungsfälle mit unveränderten Prozessinstanzen, nämlich Verträglichkeitsprüfung und Änderungspropagation bei Schemaevolution sowie das Weiterschalten im Kontrollfluss. Hierauf stellen wir Varianten für die Zustandsclusterung vor, bevor wir auf die Zustandsclusterung von veränderten Prozessinstanzen eingehen.

6.1 Konzept zur Clusterung unveränderter Prozessinstanzen

Jede Prozessvorlage definiert den Ablauf eines Prozesses und damit die möglichen Zustände der Prozessinstanzen. Da sowohl die Knotenmenge einer Prozessvorlage als auch die Menge an Zuständen des Prozessmetamodells endlich ist, ist auch die Menge aller möglichen Zustände einer Prozessinstanz endlich. Jede Prozessvorlage definiert so einen spezifischen, endlichen Raum von Instanzzuständen. Die im Ausführungsmodell des Prozessmetamodells definierten Zustandsübergänge (s. Abschnitt 2.2.3.1) spiegeln sich in Übergängen der Instanzzustände wider. Betrachtet man die durch eine Prozessvorlage definierten Instanzzustände als Knoten und die durch das Prozessmetamodell vorgegebenen Zustandsübergänge als Kanten zwischen den Knoten, erhält man einen *Zustandsgraphen* für jede Prozessvorlage (s. Abb. 6.1 und Abb. 6.2).

Diesen Zustandsgraphen kann man für die Zustandsrepräsentation von Prozessinstanzen der entsprechenden Prozessvorlage verwenden. Dazu referenzieren die Prozessinstanzen den Knoten des Zustandsgraphen, der ihrem momentanen Zustand entspricht (links in

Abb. 6.5). Im Gegenzug entfällt bei jeder Prozessinstanz die individuelle Datenstruktur für die Zustandsrepräsentation. Aus Sicht des Zustandsgraphen bildet jeder Knoten einen Cluster, dem beliebig viele Prozessinstanzen angehören können; die Prozessinstanzen eines Clusters besitzen denselben Zustand.

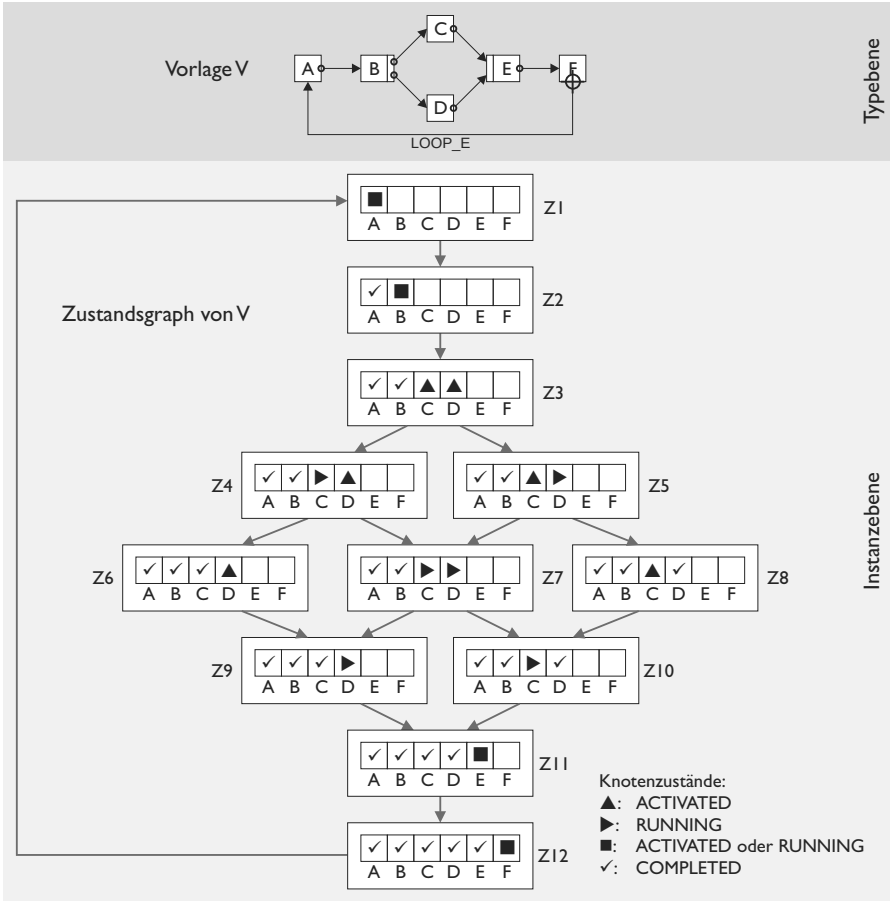


Abbildung 6.1: Zustandsgraph einer Prozessvorlage mit paralleler Verzweigung

Die konkrete Repräsentation der Knoten eines Zustandsgraphen ist unabhängig von der Zustandsclusterung und kann beispielsweise mittels expliziter oder impliziter Zustände erfolgen (s. Abschnitt 4.1.3.2). Im folgenden gehen wir aufgrund der einfacheren Darstellung von expliziten Markierungen aus. Abb. 6.1 zeigt den Zustandsgraphen einer Prozessvorlage mit Parallelverzweigung und Schleife. Da parallele Zweige bezüglich ihrer Zustände unabhängig voneinander sind, ergeben sich im Zustandsgraphen das kartesische Produkt der Zustände der einzelnen Zweige und somit viele Zustandsknoten. Deren Anzahl wächst exponentiell zur Anzahl der Zweige der parallelen Verzweigung.[1] Da die Zweige einer alternativen Verzweigung nicht unabhängig voneinander sind, werden hierfür wesentlich weniger Knoten benötigt (s. Abb. 6.2). Die Pfade durch den Zustandsgraphen spalten sich bei einer alternativen Verzweigung auf und kommen im Gegensatz zur parallelen Verzweigung auch nicht mehr zusammen (Z5 und Z8 in Abb. 6.2 gegenüber Z9, Z10 und Z11 in Abb. 6.1). Stattdessen bleibt die Aufteilung bis zum letzten Instanzzustand erhalten, so dass bei jedem alternativen Verzweigungsknoten

[1] $O(l^b)$ (l: Knoten im längsten Pfad einer Verzweigung, b: Teilzweige der Verzweigung)

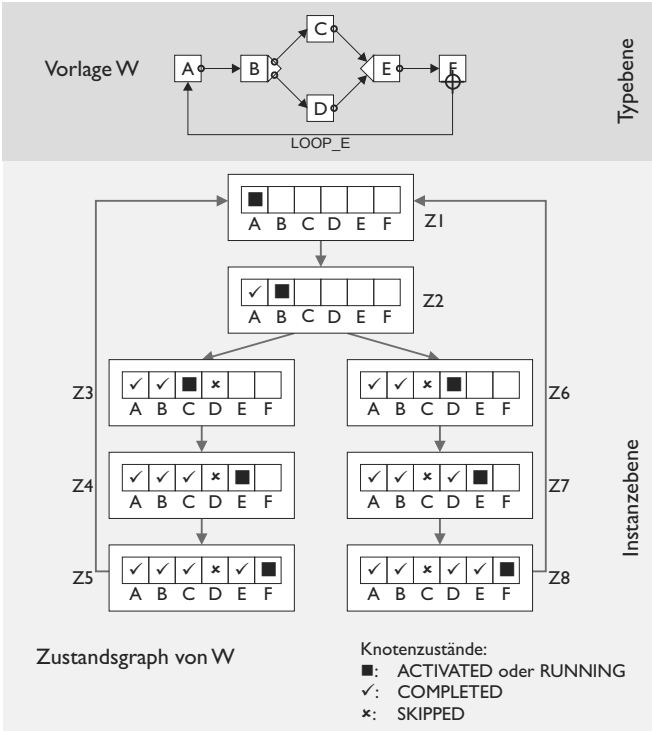


Abbildung 6.2: Zustandsgraph einer Prozessvorlage W mit alternativer Verzweigung

für jeden Zweig voneinander unabhängige Teilzustandsgraphen entstehen. Damit wächst der Zustandsgraph exponentiell zur Anzahl alternativer Verzweigungen[2] in der Prozessvorlage. Schleifen in der Prozessvorlage führen lediglich zu Rücksprungkanten im Zustandsgraphen. Bei alternativen Verzweigungen innerhalb einer Schleife führt dies entsprechend zu einer Rücksprungkante pro Zweig (s. Abb. 6.2). Es entstehen dadurch aber keine neuen Zustandsknoten. Die für Schleifen notwendigen Iterationszähler werden sinnvollerweise nicht im Zustandsgraphen repräsentiert, sondern verbleiben in instanz-individuellen[3] Datenstrukturen. Andernfalls wäre der Zustandsgraph unendlich.

6.2 Anwendungsfälle

Obwohl die Zustandsclusterung von Prozessinstanzen lediglich eine alternative Zustandsrepräsentation ist, wirkt sich die Clusterung erheblich auf Funktionen aus, bei denen der Zustand relevant ist. Hierzu gehören insbesondere die Verträglichkeitsprüfung bei Schemaevolution, die Änderungspropagation bei Schemaevolution sowie das Weiterschalten im Kontrollfluss während einer Prozessausführung. In diesem Abschnitt bewerten wir diese Funktionen im Kontext der Zustandsclusterung. Wir beschränken uns dabei zunächst auf unveränderte Prozessinstanzen.

[2] $O(b^a)$ (b: Teilzweige pro Verzweigung, a alle alternativen Verzweigungen in der Prozessvorlage

[3] Instanz-individuelle Daten gelten wie instanzspezifische Daten nur für einzelne Instanzen, die entsprechenden Datenstrukturen sind jedoch für alle Prozessinstanzen notwendig.

6.2.1 Verträglichkeitsprüfung bei Schemaevolution

Bei Zustandsclusterung vereinfachen sich die Verträglichkeitsprüfungen bezüglich des Zustands bei einer Schemaevolution: Hierzu betrachten wir die entsprechende Definition im Kontext von Zustandsgraphen:

Änderungen im Rahmen einer Schemaevolution sind dann bezüglich des Zustands der betroffenen Prozessinstanzen verträglich, wenn die Ausführungshistorie der Prozessinstanzen auch nach Anwendung der Änderungsoperationen in gleicher Form erzeugt werden kann. Das heißt, die Zustandsübergänge der Prozessschritte können in derselben Reihenfolge auch auf der aus der Schemaevolution resultierenden Prozessvorlage erfolgen. Irrelevant sind dabei abgewählte Zweige alternativer Verzweigungen, da diese keine Einträge in der Ausführungshistorie verursacht haben. Ist ein Prozessschritt aktiviert, kann sein Zustand im Rahmen der Schemaevolution angepasst werden. Daher sind auch die entsprechenden Einträge in der Ausführungshistorie irrelevant (s. Abschnitt 2.3.2.2) – sofern die Aktivierung von Knoten überhaupt in der Ausführungshistorie protokolliert wird.

Bei der Zustandsclusterung repräsentiert jeder Knoten eines Zustandsgraphen eindeutig eine festgelegte Reihenfolge von Zustandsübergängen, also genau die Informationen, die für die Verträglichkeit bezüglich des Zustands relevant sind. Damit müssen die Zustände der Prozessinstanzen nicht individuell geprüft werden, sondern die Überprüfung der Verträglichkeit bezüglich des Zustands kann direkt an den Knoten des Zustandsgraphen ermittelt und darüber auch definiert werden. Deshalb hängt der Aufwand für die Verträglichkeitsprüfung bezüglich des Zustands von der Anzahl der Zustandsknoten ab und nicht von der Anzahl der Prozessinstanzen. Alle Prozessinstanzen im selben Zustand werden bezüglich der Verträglichkeitsprüfung gleich behandelt, wodurch sich der Aufwand für eine Schemaevolution bei sehr vielen betroffenen Prozessinstanzen erheblich reduziert.

Es ist aber noch eine weitere Verbesserung möglich. Untersucht man die Auswirkungen, die Änderungen an Prozessvorlagen auf die entsprechenden Zustandsgraphen haben, lässt sich ein Zusammenhang zwischen der Änderungsoperation, ihrer Verträglichkeit und den Zustandsknoten festlegen. Abb. 6.3 zeigt anhand zweier Zustandsgraphen die Auswirkungen beim Einfügen des Prozessschritts E (ΔF) in eine einfachen Prozessvorlage V (eine Sequenz von Prozessschritten). Die ersten vier Knoten der beiden Zustandsgraphen von V und $V' = V + \Delta F$ sind äquivalent bezüglich der Reihenfolge der Zustandsänderungen, also Z1 und Z1', Z2 und Z2', Z3 und Z3' sowie Z4 und Z4'. Dies wird durch die gefüllten Pfeile dargestellt. Da ΔF keine Auswirkungen auf die Reihenfolge der Zustandsänderungen hat, ist diese Änderungsoperation bezüglich dieser Zustandsknoten verträglich. Z5' und Z6' des Zustandsgraphen von $V' = V + \Delta F$ betreffen Zustandsänderungen des neu eingefügten Prozessschritts E. Diese Änderungen sind im alten Zustandsgraphen nicht vorhanden. Damit ändert sich ab hier die Reihenfolge der Zustandsübergänge. Diese können auf V nicht erzeugt werden, ΔF ist bezüglich dieser und der nachfolgenden Zustandsknoten nicht verträglich.

Ein Sonderfall ist Z5 im Zustandsgraphen von V'. Der aktivierte Zustand ist für Verträglichkeit irrelevant. Hier kann die Aktivierung von Prozessschritt C zurückgenommen und der neu eingefügte Prozessschritt E aktiviert werden. Bezüglich der Definition sind Z5 und Z5' aufgrund dieses Sonderfalls also auch äquivalent. In Abb. 6.3 ist dies durch einen gestrichelten Pfeil repräsentiert. Die Zustandsänderung, die in Z5 stattfindet, nämlich die Aktivierung des Prozessschritts C, entspricht Z7' im Zustandsgraphen von $V' = V + \Delta F$. Diese und vergleichbare Entsprechungen finden sich in der Abbildung in Form gepunkteter Pfeile, also Z5 und Z7', Z6 und Z8', Z7 und Z9' sowie Z8 und Z10'. Sie haben keine Bedeutung für die

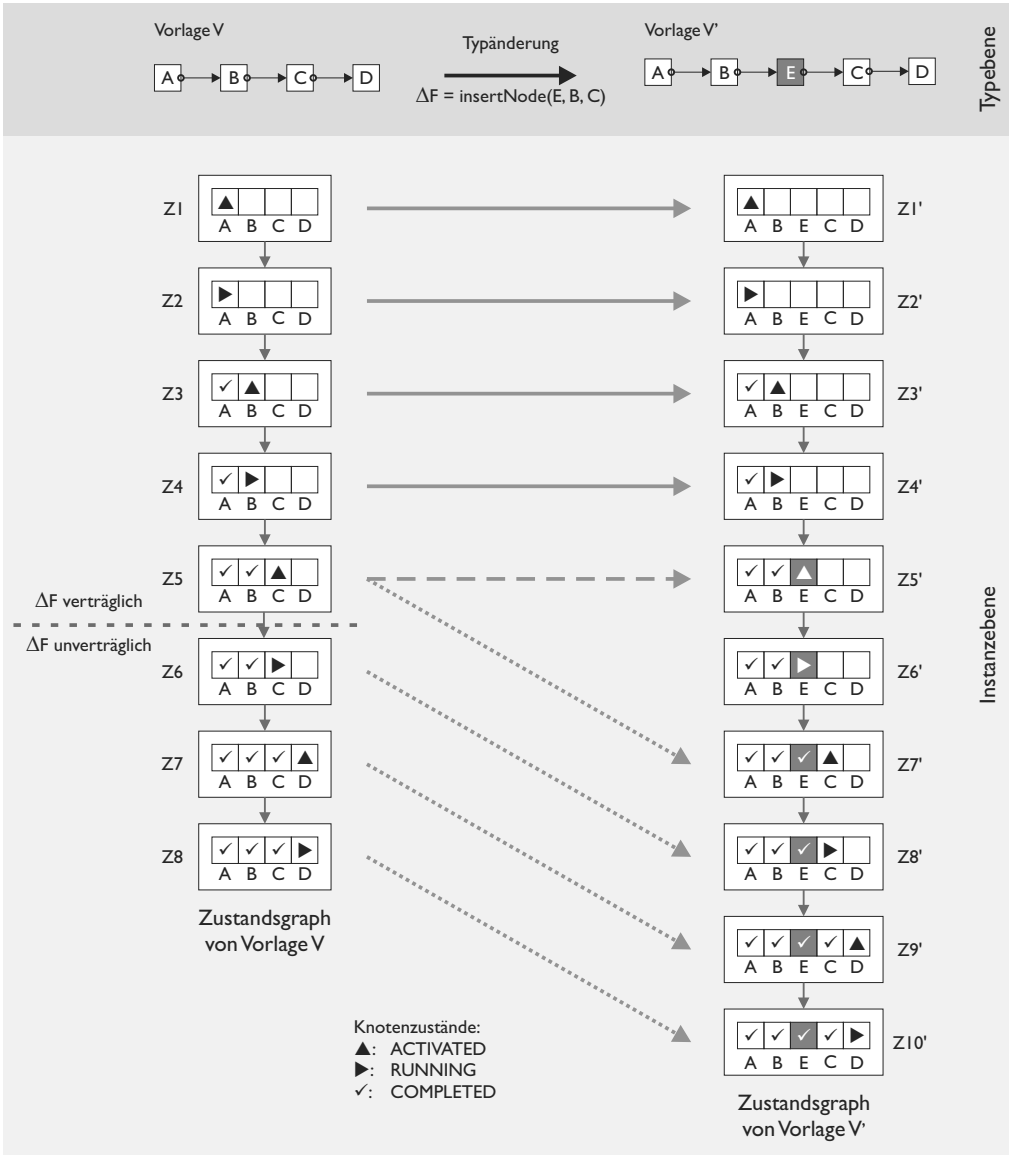


Abbildung 6.3: Änderungen am Zustandsgraphen beim Einfügen eines Prozessschritts

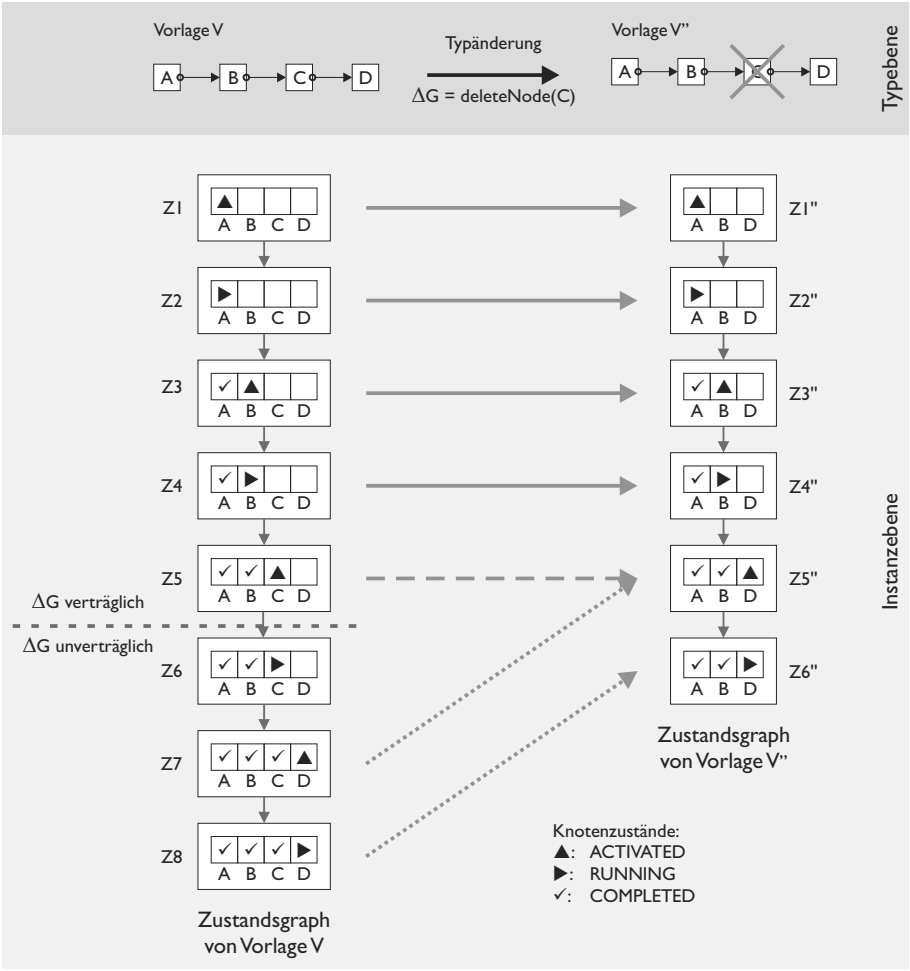


Abbildung 6.4: Änderungen am Zustandsgraphen beim Löschen eines Prozessschritts

Verträglichkeit bezüglich des Zustands. Sie verdeutlichen aber den Zusammenhang zwischen den beiden Zustandsgraphen und die Auswirkungen der Änderung ΔF in Form der zwei neuen Zustandsknoten Z5' und Z6'.

Abb. 6.4 zeigt die Auswirkungen des Löschens von Prozessschritt C (ΔG) auf die Zustandsgraphen der betroffenen Prozessvorlagen V und $V'' = V + \Delta G$. Auch hier sind wieder die ersten vier Zustandsknoten äquivalent bezüglich der Reihenfolge der Zustandsänderungen (gefüllte Pfeile; Z1 und Z1", Z2 und Z2", Z3 und Z3" sowie Z4 und Z4"). Logisch entfallen die beiden Knoten Z5 und Z6 des Zustandsgraphen von V, da sie Zustandsänderungen des Prozessschritts C betreffen. Die beiden letzten Knoten Z7 und Z8 mit den Zustandsänderungen des Prozessschritts D finden sich logisch im Zustandsgraphen von V'' als Z5" und Z6" wieder (gepunktete Pfeile). Die Änderungsoperation ΔG ist nicht verträglich bezüglich dieser beiden Zustandsknoten, da der abgeschlossene Zustand des Prozessschritts C auf der Prozessvorlage $V'' = V + \Delta G$ nicht erzeugt werden kann. Durch die Rücknahme der Aktivierung von Prozessschritten ist Z5 wieder ein Sonderfall. Die Änderungsoperation ist diesbezüglich verträglich und Prozessinstanzen in diesem Zustand können migriert werden (Z5 und Z5").

Auf diese Weise können für alle Änderungsoperationen die jeweiligen Auswirkungen auf den Zustandsgraphen bestimmt werden. Dadurch kann anhand der Änderungsoperationen selbst

festgelegt werden, bezüglich welcher Zustandsknoten diese verträglich sind und bezüglich welcher nicht. Für jede Änderungsoperation können die Zustandsknoten also klassifiziert werden. Umfasst eine Schemaevolution mehrere Änderungsoperationen in einer Änderungstransaktion, ergibt sich die Klassifikation der Zustandsknoten für die gesamte Änderungstransaktion anhand der einzelnen Änderungsoperationen. Ist eine Änderungsoperation bezüglich eines Zustandsknotens unverträglich, ist die gesamte Änderungstransaktion bezüglich dieses Zustandsknotens unverträglich. Eine Änderungstransaktion ist nur dann bezüglich eines Zustandsknotens verträglich, wenn alle ihre Änderungsoperationen bezüglich dieses Zustandsknotens verträglich sind.

Die Verträglichkeitsprüfung bei Schemaevolution kann noch weiter optimiert werden: Die Zustandsknoten einer Klasse sind immer zusammenhängend. Die Grenze zwischen den Klassen existiert auf jedem Weg durch den Zustandsgraphen genau einmal. Das heißt, ab einem bestimmten Zustandsknoten gehören alle direkten und indirekten Nachfolgerknoten zur Klasse der Zustandsknoten bezüglich der die Änderungstransaktion unverträglich ist. Alle Vorgängerknoten gehören zu der Klasse, bezüglich der die Änderungstransaktion verträglich ist. Die Grenze verschiebt sich im Zustandsgraphen für jede Änderungsoperation höchstens nach vorne, aber nie nach hinten. Damit reicht es für die Verträglichkeitsprüfung aus, für jede Änderungsoperation diese Grenze zu ermitteln. Insbesondere muss weder jede Prozessinstanz einzeln überprüft noch muss jeder Zustandsknoten klassifiziert werden. Zustandsclusterung verbessert nicht nur die Verträglichkeitsprüfung bezüglich des Zustands bei Schemaevolution, auch die Änderungspropagation und damit die Migration der laufenden Prozessinstanzen werden erheblich vereinfacht.

6.2.2 Änderungspropagation bei Schemaevolution

Ist die Verträglichkeitsprüfung bezüglich des Zustands bei einer Schemaevolution erfolgt, können die Änderungen propagiert werden, indem die laufenden Prozessinstanzen auf die neue Prozessvorlage migriert und ggf. ihre Knotenzustände angepasst werden. Bei Zustandsclusterung beschränkt sich die Migration auf das Umhängen von Prozessinstanzen vom Zustandsgraphen der ursprünglichen Prozessvorlage in den Zustandsgraphen der geänderten Prozessvorlage. Durch Auswahl des richtigen Zielknotens wird implizit der Zustand der Prozessinstanz entsprechend angepasst, etwa wenn sich der aktivierte Prozessschritt ändert. Der Zielknoten ergibt sich durch die Äquivalenz von Zustandsknoten bezüglich der Zustandsverträglichkeit.

Wie in Abschnitt 6.2.1 beschrieben, gibt es Knoten des ursprünglichen Zustandsgraphen und des Zustandsgraphen der geänderten Prozessvorlagen, die bezüglich der Verträglichkeitsprüfung äquivalent sind. Das sind die Zustandsknoten, die dieselbe Reihenfolge von Zustandsänderungen repräsentieren. Auf diese Weise lassen sich den Knoten des ursprünglichen Zustandsgraphen, Knoten des neuen Zustandsgraphen eindeutig zuordnen. Dies gilt nur für Zustandsknoten, bezüglich derer die Änderungstransaktion verträglich ist. Andere Zustandsknoten benötigen keine Zuordnung, da die entsprechenden Prozessinstanzen aufgrund von Unverträglichkeit gar nicht migriert werden dürfen. In den Abbildungen 6.3 und 6.4 sind diese Zuordnungen durch gefüllte und gestrichelte Pfeile dargestellt. Für die Migration werden die Prozessinstanzen einfach entsprechend diesen Pfeilen umgehängt. Damit ist die Schemaevolution bereits abgeschlossen.

Abb. 6.5 zeigt eine Schemaevolution mit komplexeren Prozessvorlagen und komplexeren Änderungen. Im Zustandsgraphen der Prozessvorlage V wurde die Grenze zwischen den Klassen von

Zustandsknoten ermittelt. Die durchgeführten Änderungen ΔF sind bezüglich der Zustandsknoten vor dieser Grenze verträglich, bezüglich der nachfolgenden Zustandsknoten dagegen unverträglich. Als äquivalenter Knoten zu dem mit aktiviertem Prozessschritt D im Zustandsgraphen von V ergibt sich der Knoten mit aktiviertem Prozessschritt G im Zustandsgraphen von $V' = V + \Delta F$. Dementsprechend erfolgt die Propagation von Änderungen, indem die betroffenen Prozessinstanzen vom linken in den rechten Zustandsgraphen umgehängt werden.

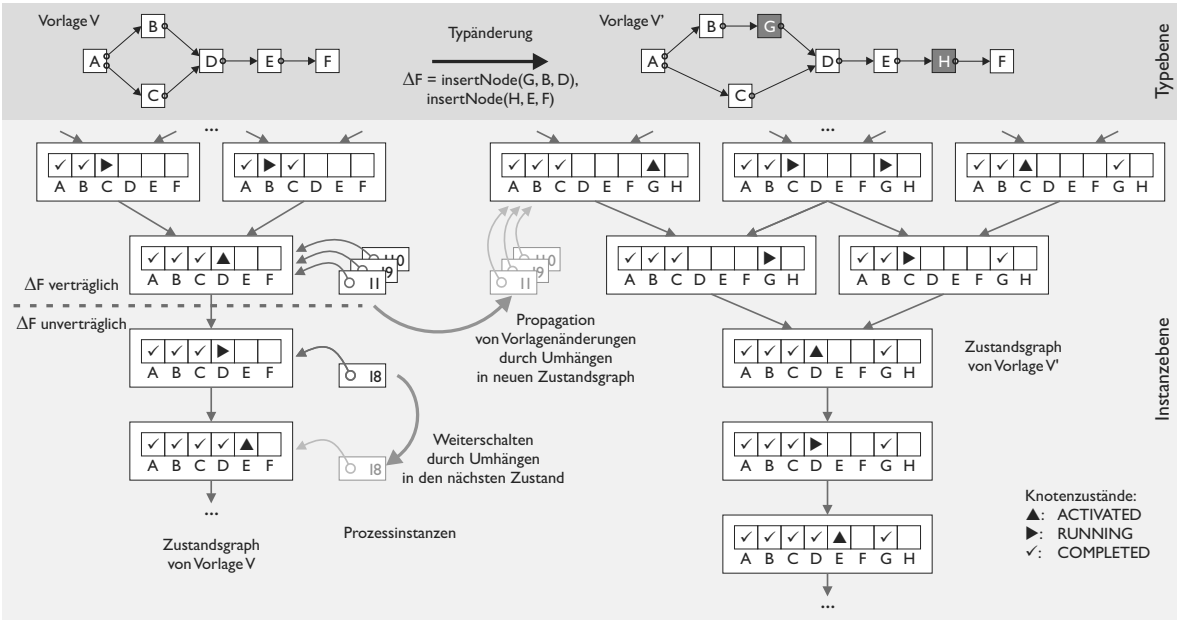


Abbildung 6.5: Zustandsclusterung von Prozessinstanzen

6.2.3 Weiterschalten

Das Weiterschalten einzelner Prozessinstanzen ist bei Zustandsclusterung einfach: Die Kanten im Zustandsgraphen repräsentieren mögliche Zustandsübergänge. Das Weiterschalten ist dann ein solcher Zustandsübergang, so dass eine Prozessinstanz dabei lediglich vom Zustandsknoten in den nachfolgenden Zustandsknoten umgehängt wird (s. Abb. 6.5 links unten). Beim Weiterschalten alternativer Verzweigungsknoten und Schleifenendknoten gibt es mehrere Nachfolger im Zustandsgraphen. Hier wird die Verzweigungsentscheidung berücksichtigt und der entsprechende Nachfolger ausgewählt. Zum Beispiel wird in Abb. 6.2 beim Beenden von Prozessschritt B berücksichtigt, ob der obere oder untere Zweig der alternativen Verzweigung gewählt wurde. Entsprechend wird der jeweils erste Zustandsknoten im linken oder rechten Zweig im Zustandsgraphen Z3 oder Z6 gewählt. Am Ende des Zustandsgraphen bei Z5 und Z8 wird abhängig von der Schleifenentscheidung in Prozessvorlage W entweder Z1 gewählt oder aber die Prozessinstanz beendet.

Beim Weiterschalten von Prozessschritten innerhalb paralleler Verzweigungen gibt es ebenfalls mehrere Nachfolgerknoten im Zustandsgraphen. Hier hängt die Auswahl des neuen Zustandsknotens davon ab, in welchem Zweig der parallelen Verzweigung die Zustandsänderung stattfand. In Abb. 6.1 wird bei Zustandsänderungen im oberen Zweig der parallelen Verzweigung (Prozessschritt C), der linke Nachfolger Z4 im Zustandsgraphen gewählt. Bei

Zustandsänderungen im unteren Zweig der parallelen Verzweigung (Prozessschritt D) erfolgt das Weiterschalten nach rechts unten in Z5.

6.3 Quantitative Betrachtungen

Der Speicherbedarf für die Datenstrukturen bei Zustandsclusterung unterscheidet sich erheblich vom Speicherbedarf der bisher vorgestellten Repräsentations- und Implementierungsvarianten. Er hängt in hohem Maße von der Struktur der Prozessvorlage und den Zuständen des verwendeten Prozessmetamodells ab. Deshalb werden wir bei der quantitativen Betrachtung der Zustandsclusterung, im Gegensatz zu den quantitativen Betrachtungen in den Kapiteln 4 und 5, nicht von der Struktur der Prozessvorlage abstrahieren, sondern diese explizit diskutieren.

Zunächst betrachten wir sequentielle Prozesse, bevor wir anschließend den Speicherbedarf von Zustandsgraphen im Zusammenhang mit alternativen und parallelen Verzweigungen untersuchen. Wir gehen jeweils von einer Zustandsclusterung mit expliziten Markierungen (s. Abschnitt 4.1.3.2) aus.

6.3.1 Sequentielle Prozesse

Bei rein sequentiellen Prozessen ergibt sich der Speicherbedarf für den Zustandsgraphen direkt aus der Anzahl der Zustandsknoten $N_{StateNodes}$ multipliziert mit dem Speicherbedarf eines einzelnen Zustandsknotens $M_{StateNodes}$. Der Speicherbedarf eines Zustandsknoten mit expliziten Markierungen umfasst die Zustandsliste (s. Abb. 6.1). Er ergibt sich aus der Anzahl der Prozessschritte N_{Nodes} multipliziert mit dem Speicherbedarf für die Zustandsinformationen ($M_{NodeState}$). Die Anzahl der Zustandsknoten wiederum entspricht der Anzahl der Prozessschritte N_{Nodes} multipliziert mit der Anzahl der möglichen Zustände im verwendeten Metamodell N_{States} abzüglich des Speicherbedarfs für Initial- und Endzustände. Die Subtraktion ist nötig, da in einem sequentiellen Prozess der Initial- und Endzustand implizit sind. Alle Vorgänger eines aktiven Prozessschritts sind beendet, und alle Nachfolger befinden sich noch im Initialzustand.

Formel 6.1 (Speicherbedarf für Zustandsgraphen bei sequentiellen Prozessen)

$$\begin{aligned} MSeq_{StateNodes} &= NSeq_{StateNodes} * M_{StateNodes} \\ &= N_{Nodes} * (N_{States} - 2) * M_{NodeState} \end{aligned}$$

Formel 6.1 gilt sowohl für sequentielle Prozesse als auch für Prozesse mit Schleifen. Schleifen führen lediglich zu Rücksprungkanten im Zustandsgraphen. Sie haben keine weiteren Auswirkungen auf die Struktur des Zustandsgraphen und damit auch nicht auf den Speicherbedarf (s. Abb. 6.1).

6.3.2 Alternative Verzweigungen

Bei einer Prozessvorlage mit alternativer Verzweigung hängt der Speicherbedarf von weiteren Aspekten ab. Hierzu gehören die der Anzahl an alternativen Verzweigungen einer Prozessvorlage, die Anzahl der Zweige $N_{Branches}(i)$ pro Verzweigung i sowie der Position jeder Verzweigung relativ zum Start- und Endknoten der Prozessvorlage:

Zur Ausführungszeit wird bei jeder alternativen Verzweigung nur ein Zweig ausgewählt und durchlaufen, die Prozessschritte aller anderen Zweige werden in den Zustand „übersprungen“

versetzt. Der ausgewählte Zweig selbst durchläuft die Zustandsübergänge so wie eine Prozessvorlage ohne alternative Verzweigung mit dem entsprechenden Zweig als Sequenz. Die anderen Zweige der alternativen Verzweigung sind irrelevant für die weiteren Zustandsknoten. Da vorab nicht entschieden werden kann, welcher Zweig ausgewählt wird, ergibt sich für jede alternative Verzweigung entsprechend eine Verzweigung im Zustandsgraphen. Diese verzweigt sich im selben Grad $N_{Branches}(i)$ wie die alternative Verzweigung in der Prozessvorlage. Der Vereinigungsknoten ($join_i$) einer Verzweigung i der Prozessvorlage führt jedoch nicht zu einer Vereinigung der Pfade im Zustandsgraphen. Stattdessen bleiben die Pfade im Zustandsgraphen bis zum Ende getrennt. Damit gibt es für jede Zustandsänderung eines Prozessschritts nach einer alternativen Verzweigung pro Zweig ein Zustandsknoten. So spiegelt sich die Zustandsänderung von Prozessschritt F in Abb. 6.1 in je einem Zustandsknoten pro Zweig wider.

Für jede alternative Verzweigung i spaltet sich der Zustandsgraph weiter auf. Die Anzahl der nachfolgenden Zustandsknoten erhöht sich jeweils um den Faktor $N_{Branches}(i)$. Deshalb ist die Formel für die Anzahl der Zustandsknoten einer alternativen Prozessverzweigung rekursiv. Sie gilt immer nur für Bereiche zwischen zwei Prozessschritten; $nodes(x, y)$ ist dabei die Menge der Prozessschritte zwischen Prozessschritt x (einschließlich) und Prozessschritt y (ausschließlich) an. $N_{Nodes}(x, y) = |nodes(x, y)|$ ist die jeweilige Anzahl an Prozessschritten:

Formel 6.2 (Anzahl an Zustandsknoten bei alternativer Verzweigung)

$$NAlt_{StateNodes}(x, y) = \begin{cases} N_{Nodes}(x, join_i) * (N_{States} - 2)) & , join_i \in nodes(x, y) \\ \quad +N_{Branches}(i) * NAlt_{StateNodes}(join_i, y) & \\ (N_{Nodes}(x, y) + N_{Nodes}(y, y)) * (N_{States} - 2) & , sonst \end{cases}$$

Der erste Teil von Formel 6.2 berechnet die Anzahl der Zustandsknoten rekursiv. Sie ist anzuwenden, wenn sich zwischen den Prozessschritten x und y ein Vereinigungsknoten einer alternativen Verzweigung befindet. Existiert keine alternative Verzweigung, gilt der zweite Teil der Formel analog zum sequentiellen Fall.

6.3.3 Parallele Verzweigungen

Im Gegensatz zu alternativen Verzweigungen sind die einzelnen Zweige einer parallelen Verzweigung vollständig voneinander unabhängig, d. h. ihre Zustände können beliebig kombiniert werden. Am Ende der Verzweigung findet eine Vereinigung aller parallelen Zweige statt. Da die Berechnung des Speicherbedarfs für parallele Verzweigungen komplex ist, treffen wir zwecks besserer Lesbarkeit einige vereinfachende Annahmen: Die parallelen Verzweigungen aller Prozesse besitzen gleich viele Zweige ($N_{Branches}$), außerdem besitzt jeder Zweig dieselbe Anzahl an Prozessschritten $N_{NodesPerBranch}$.

Zunächst berechnen wir die Anzahl der Zustandsknoten eines parallelen Verzweigungsblocks. Anschließend erweitern wir die Berechnung so, dass sich die Anzahl der Zustandsknoten für einen ganzen Prozess mit mehreren parallelen Verzweigungen ergibt. Für die Anzahl der Zustandsknoten in einem parallelen Verzweigungsblock gilt unter diesen Voraussetzungen:

Formel 6.3 (Anzahl an Zustandsknoten bei parallelem Verzweigungsblock)

$$NParBlock_{StateNodes} = (N_{NodesPerBranch} * (N_{States} - 2))^{N_{Branches}}$$

Die Anzahl der Zustandsknoten berechnet sich aus der Anzahl der Prozessschritte eines Zweigs, multipliziert mit der Anzahl der durch das Metamodell vorgegebenen Zustände. Da die Zustände der einzelnen Zweige voneinander unabhängig sind und jeder Knotenzustand eines Zweigs mit jedem anderen Knotenzustand eines anderen Zweigs kombiniert werden kann, muss die ermittelte Anzahl ($N_{NodesPerBranch} * (N_{States} - 2)$) mit der Anzahl der Zweige der parallelen Verzweigung potenziert werden.

Um die Gesamtzahl an Zustandsknoten für einen ganzen Prozess mit mehreren parallelen Verzweigungen zu bestimmen, muss die Anzahl an Zustandsknoten für einen parallelen Block mit der Anzahl an parallelen Blöcken $N_{ParallelBlocks}$ multipliziert werden – dies beruht auf der Annahme, dass jeder parallele Block gleich viele Prozessschritte enthält. Zu den Zustandsknoten in parallelen Verzweigungen müssen die Zustandsknoten der umgebenden sequentiellen Prozessschritte addiert werden. Deren Anzahl ergibt sich durch die Anzahl aller Prozessschritte einer Vorlage N_{Nodes}, abzüglich der Prozessschritte in parallelen Verzweigungen ($N_{NodesPerBranch} * N_{Branches} * N_{ParallelBlocks}$).

Formel 6.4 (Anzahl an Zustandsknoten bei paralleler Verzweigung)

$$\begin{aligned} NPar_{StateNodes} &= (N_{Nodes} - (N_{NodesPerBranch} * N_{Branches} * N_{ParallelBlocks})) \\ &\quad *(N_{States} - 2) \\ &+ N_{ParallelBlocks} * (N_{NodesPerBranch} * (N_{States} - 2))^{N_{Branches}} \end{aligned}$$

6.3.4 Gesamter Speicherbedarf

Der gesamte Speicherbedarf der Datenstrukturen bei Zustandsclusterung M_{Total} hängt von der Anzahl an Prozessvorlagen $N_{Templates}$ ab. Für jede Prozessvorlage wird diese zusammen mit einem Zustandsgraphen verwaltet. Der Speicherbedarf für die Prozessvorlagen berechnet sich aus deren Anzahl, multipliziert mit der Anzahl an Prozessschritten pro Vorlage, multipliziert mit dem Speicherbedarf eines Prozessschritts M_{Node}.

Für einen Zustandsknoten ergibt sich der Speicherbedarf aus dem Produkt des Speicherbedarfs eine Knotenzustands $M_{NodeState}$ und der Anzahl an Prozessschritten pro Vorlage N_{Nodes}. Multipliziert man dieses Produkt mit der Anzahl an Zustandsknoten $N^*_{StateNodes}$ ($NSeq_{StateNodes}$, $NAlt_{StateNodes}$ oder $NPar_{StateNodes}$ abhängig von der Prozessstruktur), erhält man den Speicherbedarf für die Zustandsgraphen aller Prozessvorlagen.

Daneben benötigen bei Zustandsclusterung auch die Prozessinstanzen selbst Speicherplatz. Diese beinhalten nicht mehr die eigentlichen Zustandsinformationen, sondern nur Informationen, die bei jeder Prozessinstanz unterschiedlich sind (z. B. Bearbeiter abgeschlossener Schritte). Je Prozessinstanz wird hierfür die Anzahl der Prozessschritte, multipliziert mit dem Speicherbedarf der dynamischen Informationen, benötigt ($N_{Nodes} * M_{InstanceNode}$). Die Zustandsinformation wird mittels einer Referenz auf den entsprechenden Knoten im Zustandsgraphen verwaltet. Diese benötigt ebenfalls etwas Speicher ($M_{Reference}$).

Formel 6.5 (Gesamter Speicherbedarf für Zustandsclusterung)

$$\begin{aligned} M_{total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Templates} * N^*_{StateNodes} * (N_{Nodes} * M_{NodeState}) \\ &+ N_{Instances} * (M_{Reference} + N_{Nodes} * M_{InstanceNode}) \end{aligned}$$

Beispiele für den Speicherbedarf der Realisierungsvarianten für Knotenzustände

Mit Hilfe dieser Formeln ergänzen wir das Beispiel aus Abschnitt 4.1.4.5 für verschiedene Realisierungsvarianten von Knotenzuständen um das Konzept der Zustandsclusterung. Wir nehmen wieder an, dass einfache Prozesse 2 parallele Verzweigungen pro Prozessvorlage, 2 parallele Zweige pro Verzweigung und 3 Prozessschritte pro Zweig haben. Für parallele Prozesse nehmen wir 4 parallele Verzweigungen je Prozessvorlage, 3 parallele Zweige je Verzweigung und 6 Prozessschritte je Zweig an.

Tabelle 6.1: Speicherbedarf der verschiedenen Realisierungsvarianten für Knotenzustände

		einfache Prozesse 10 Vorl. à 20 Knoten, 100 Instanzen	parallele Prozesse 10 Vorl. à 100 Knoten, 100 Instanzen	einfache Prozesse 200 Vorl. à 20 Knoten, 100.000 Instanzen	parallele Prozesse 200 Vorl. à 100 Knoten, 100.000 Instanzen
Vorlagenkopien (expl. Markierungen) (VG-1)		512.000 Byte	2.560.000 Byte	472.800.000 Byte	2.364.000.000 Byte
Vorlagen-refer-enzen	Ausführungshistorie (ZR-1)	360.000 Byte	1.800.000 Byte	320.800.000 Byte	1.604.000.000 Byte
	impl. Markierungen (ZR-2)	108.000 Byte	532.000 Byte	68.800.000 Byte	336.000.000 Byte
	expl. Markierungen (ZR-3)	112.000 Byte	560.000 Byte	72.800.000 Byte	364.000.000 Byte
Instanzcluster mit expl. Markierungen (ZR-3)		174.400 Byte	55.958.400 Byte	72.480.000 Byte	1.462.560.000 Byte

Der Speicherbedarf der Datenstrukturen bei Zustandsclusterung (*Instanzcluster*) hängt erwartungsgemäß in hohem Maße von der Komplexität der Prozesse ab – und zwar stärker, als dies bei den anderen Realisierungsvarianten der Fall ist. Sie verhalten sich in gewisser Hinsicht umgekehrt zu Vorlagenkopien (mit expliziten Markierungen). Das heißt, je mehr Instanzen pro Prozessvorlage existieren, desto geringer wird der Speicherbedarf. Sie eignen sich in der vorgestellten Form deshalb nur für einfache Prozesse oder für Anwendungsfälle mit vielen Prozessinstanzen und einfachen Prozessvorlagen. Aufgrund der erwarteten guten Laufzeit bei Schemaevolution untersuchen wir im nächsten Abschnitt noch Optimierungsmöglichkeiten für Zustandsclusterung. Ein wichtiges Ziel ist die Verringerung der Abhängigkeit zur Komplexität von Prozessvorlagen.

6.4 Varianten der Clusterung unveränderter Prozessinstanzen

Zustandsclusterung lässt sich auf verschiedene Arten umsetzen. In diesem Abschnitt diskutieren wir verschiedene Varianten, die einen reduzierten Speicherbedarf haben. Dies kann durch Aggregation von Knoten des Zustandsgraphen und somit dessen Reduktion erfolgen (s. Abb. 6.6). Alternativ können vom Zustandsgraphen nur die Teile tatsächlich realisiert sein, die für die vorhandenen Prozessinstanzen benötigt werden (späte Clustererzeugung, IC-3). Bei der Reduktion des Zustandsgraphen gehen Informationen verloren, die für das PrMS wichtig sind. Dementsprechend müssen diese Informationen, wie bei der herkömmlichen Zustandsrepräsentation, von instanz-individuellen Datenstrukturen bereitgestellt werden. Dadurch reduziert sich der Speicherbedarf für den Zustandsgraphen, es wird jedoch zusätzlicher Speicherbedarf für Prozessinstanzen benötigt.

Die Aggregation kann einerseits auf Zuständen erfolgen (s. Abb. 6.6), d. h. verschiedene Instanzzustände werden zusammengefasst (Zustandsaggregation, IC-1). Andererseits kann Aggregation auch auf der Prozessstruktur erfolgen. Dabei werden aufeinanderfolgende Prozessschritte logisch zu einem Prozessschritt zusammengefasst und im Zustandsgraphen entsprechend behandelt. Die Zusammenfassung aufeinanderfolgender Prozessschritte kann blockorientiert

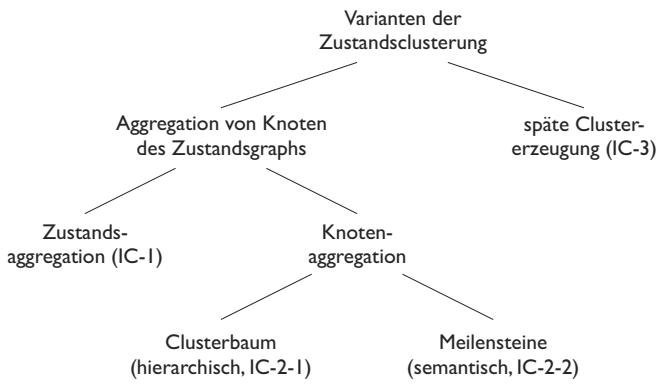


Abbildung 6.6: Varianten der Zustandsclusterung

(Clusterbaum (IC-2-1), s. Abschnitt 6.4.2.2) oder manuell durch den Prozessmodellierer erfolgen (sog. Meilensteine (IC-2-2), s. Abschnitt 6.4.2.1).

6.4.1 Variante IC-1: Zustandsaggregation

Bei Variante IC-1 (Zustandsaggregation) der Zustandsclusterung werden verschiedene, aufeinanderfolgende Zustände eines Prozessschritts zu einem aggregierten Zustand zusammengefasst. Sinnvollerweise bieten sich hierfür Zustände an, deren Unterscheidung bei Zugriffen auf den Zustandsgraphen selten benötigt wird. Dies gilt beispielsweise für die Zustände `ACTIVATED` und `RUNNING` (s. Abschnitt 2.2.3.1). Beide Zustände werden im Rahmen einer Schemaevolution nur dann unterschieden, wenn vor dem betreffenden Prozessschritt ein weiterer Prozessschritt eingefügt wird: Im Zustand `ACTIVATED` ist dies möglich, im Zustand `RUNNING` dagegen nicht. In den Abbildungen 6.1 und 6.2 wurde die Zustandsaggregation für eine einfachere Darstellung bereits angewendet.

Die Zustandsaggregation reduziert den Speicherbedarf vor allem im Zusammenhang mit parallelen Verzweigungen. Dies liegt daran, dass sich durch die Zustandsaggregation die Zustände der einzelnen Zweige einer parallelen Verzweigung reduzieren. Allerdings werden für die Zustandsunterscheidung zusätzliche Datenstrukturen für jede Prozessinstanz benötigt. Diese stellen die expliziten Knotenzustände bereit; es reichen allerdings so viele Knotenzustände, wie es aktive Prozessschritte in der entsprechenden Vorlage gibt. Dies entspricht der Anzahl paralleler Zweige einer Prozessvorlage.

Bei einer großen Anzahl an Prozessinstanzen lohnt sich diese Aggregation nicht mehr, da die Unterscheidung der Zustände in instanz-individuellen Datenstrukturen mehr Speicher benötigt als die Verwendung nicht aggregierter Zustände. Die Grenze dafür, ob die Aggregation mit instanz-individuellen Datenstrukturen sinnvoll ist oder nicht, hängt von der Prozessstruktur ab. Insbesondere die Länge und Anzahl paralleler Zweige sind hierfür entscheidend.

Bei geeigneter Zustandsaggregation gibt es kaum Auswirkungen auf die Verträglichkeitsprüfung bezüglich des Zustands und der Änderungspropagation bei Schemaevolution. Je mehr Zustände aggregiert werden, desto größer sind aber die Auswirkungen. Dabei nähern sich die Abläufe bei der Verträglichkeitsprüfung bezüglich des Zustands und bei der Änderungspropagation mit aggregierten Zuständen den entsprechenden Abläufen bei instanz-individueller Zustandsrepräsentation an. Die Auswirkungen auf das Weiterschalten sind dagegen unabhängig davon, welche Zustände aggregiert werden. Je nach Zustandsübergang werden entweder die

Zuordnung zum Zustandsknoten oder die instanz-individuellen Datenstrukturen geändert. Große Auswirkungen hat die Zustandsaggregation auf Anfragen bezüglich Instanzzuständen. Anfragen nach aggregierten Zuständen entsprechen den Anfragen bei instanz-individueller Zustandsrepräsentation. Sie sind entsprechend aufwendiger als Anfragen nach nicht-aggregierten Zuständen, die über die Zustandsclusterung realisiert werden. Sehr aufwendig sind Anfragen, die sich sowohl auf aggregierte als auch auf nicht-aggregierte Zustände beziehen.

6.4.2 Knotenaggregation

Die Anzahl der Zustandsknoten reduziert sich, wenn eine Vorlage weniger Prozessschritte hat. Das logische Zusammenfassen von Prozessschritten hat somit ähnliche Auswirkungen wie die Zustandsaggregation (IC-1, s. Abschnitt 6.4.1). Bei parallelen Verzweigungen reduziert sich die Anzahl der Zustände aufgrund der geringeren Anzahl an Prozessschritten in den Zweigen. Die Unterscheidung zwischen den aggregierten Prozessschritten muss, wie bei der Zustandsaggregation, in instanz-individuellen Datenstrukturen erfolgen.

Prozessschritte können auf zwei verschiedene Arten aggregiert werden: Bei blockbasierten Prozessmodellen (s. Abschnitt 4.1.5) werden alle Prozessschritte eines Blocks zusammengefasst und im Zustandsgraphen durch einen einzelnen Blockknoten repräsentiert (Clusterbaum, IC-2-1). Daneben können aufeinanderfolgende Prozessschritte durch den Modellierer unabhängig von der Prozessstruktur zusammengefasst werden (Meilensteine, IC-2-2). Beide Varianten haben unterschiedliche Auswirkungen auf die Funktionen. Sie werden in den folgenden beiden Abschnitten diskutiert.

6.4.2.1 Variante IC-2-1: Clusterbaum

Bei blockbasierten Prozessmetamodellen wie ADEPT, haben Prozessvorlagen eine logische Blockstruktur (s. Abb. 6.7 links). Analog zu Subprozessen kann man Blöcke zu einzelnen Blockknoten aggregieren und diese Blockknoten mit entsprechenden Zuständen versehen. Die Blockknoten reflektieren die Zustände der aggregierten Prozessschritte und durchlaufen das Zustandsmodell normaler Prozessschritte (s. Abschnitt 2.2.3.1). Im Zustandsknoten werden sie wie normale Prozessschritte behandelt. Der Name *Clusterbaum* dieser Variante bezieht sich auf die hierarchische Struktur der Blöcke, die verschiedene Zustände in einem Cluster aggregieren.

Abb. 6.7 zeigt links die Blockhierarchie einer einfachen Prozessvorlage, rechts daneben sieht man einen Zustandsgraphen für die 2. Ebene der Blockhierarchie. Dieser fasst mehrere Prozessschritte aufgrund der Blockstruktur zusammen und verwaltet nur noch den Zustand der entsprechenden Blockknoten. Darunter erkennt man die Zustandsknoten des entsprechenden nicht-aggregierten Zustandsgraphen und wie diese dem blockweise aggregierten Zustandsgraphen zugeordnet sind.

Aggregiert man jeden Zweig einer parallelen Verzweigung zu einem Blockknoten, besitzen alle Zweige nur noch die Länge 1, und die parallele Verzweigung lässt sich relativ einfach im Zustandsgraphen darstellen. Zusätzlich dazu kann man die einzelnen Blockknoten über voneinander unabhängige Zustandsgraphen repräsentieren (s. Prozessschritte C und D in Abb. 6.7). In diesem Fall referenzieren Prozessinstanzen, bei denen gerade Prozessschritte der parallelen Verzweigung aktiv sind, für jeden dieser Zweige jeweils einen Zustandsknoten statt insgesamt nur einen Zustandsknoten (s. Abb. 6.8). Dies lässt sich auch auf alternative Verzweigungen anwenden (s. Abb. 6.9). Allerdings muss hier nach Durchlaufen der alternativen Verzweigung in instanz-individuellen Datenstrukturen gespeichert werden, welcher Zweig durchlaufen wor-

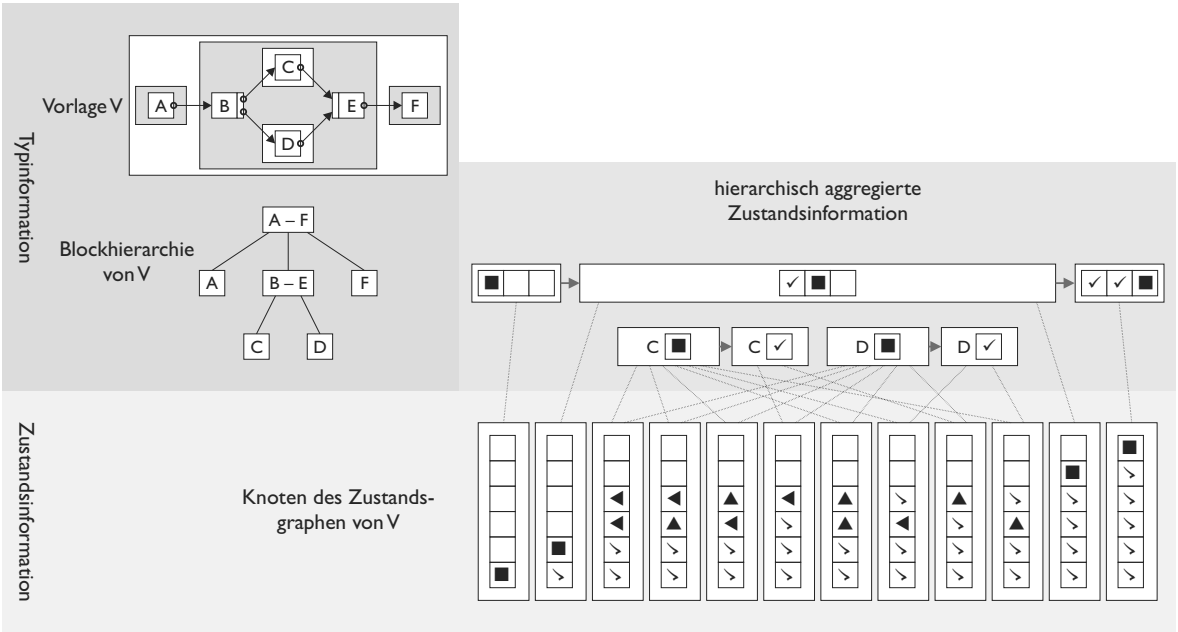


Abbildung 6.7: Clusterbaum (IC 2-1)

den ist. Dies ist für die Verträglichkeitsprüfung bezüglich des Zustands unerlässlich, da eine Prozessinstanz nur migriert werden darf, wenn durch die Schemaevolution übersprungene oder noch nicht durchlaufene alternative Zweige geändert werden; wird der durchlaufene Zweig geändert, sind die Änderungen bezüglich dieser Prozessinstanz nicht verträglich.

Die Auswirkungen von Variante IC-2-1 auf den Speicherbedarf sind mit denen der Zustandsaggregation vergleichbar. Allerdings benötigt die Aggregation von Prozessschritten mehr Speicher für die instanz-individuellen Zustände. Dies ist allerdings nur dann der Fall, wenn ein aggregierter Block gerade durchlaufen wird. Befindet sich die Ausführung vor oder nach diesem Block, werden nur Informationen zu durchlaufenen Zweigen alternativer Verzweigungen benötigt. Zusammen mit dem Zustand des Blockknotens ergeben sich die Zustände der einzelnen Prozessschritte.

Bei Variante IC-2-1 verhalten sich die Funktionen entweder so wie bei normaler Zustandsclusterung oder instanz-individueller Zustandsrepräsentation. Dies hängt davon ab, ob die von der Funktion betroffenen Prozessschritte im Zustandsgraphen aggregiert oder nicht aggregiert vorliegen. Sind etwa die Zustände von aggregierten Prozessschritten für die Verträglichkeitsprüfung oder die Änderungspropagation wichtig, müssen diese instanz-individuell geprüft bzw. angepasst werden. Dasselbe gilt für das Weiterschalten und für Zustandsanfragen. Bei Zustandsanfragen kann der Zustand des Blockknotens für die Entscheidung herangezogen werden, ob die Zustände der Prozessschritte dieses Blocks für die Anfrage relevant sind oder nicht. Sind sie relevant, so muss die Zustandsanfrage auf diesen Prozessschritten wie bei instanz-individueller Zustandsrepräsentation durchgeführt werden. Für solche Anfragen dient der Zustand des Blockknotens dann als Indexstruktur.

Ein Clusterbaum kann auch ausschließlich als eine solche Indexstruktur verwendet werden. Dazu werden alle Knotenzustände mit instanz-individuellen Datenstrukturen repräsentiert. Für jede Prozessvorlage gibt es zusätzlich einen Zustandsgraphen, der jedoch nur die Zustände von Blockknoten enthält. Das Weiterschalten erfolgt wie ohne Zustandsclusterung, d. h. die

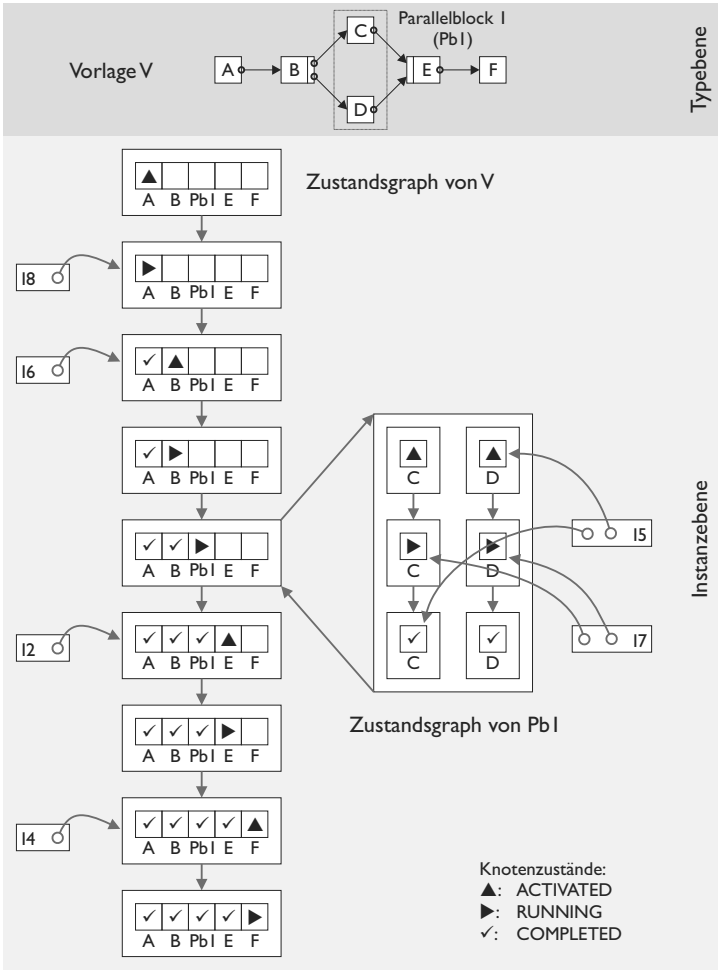


Abbildung 6.8: Getrennte Zustandsgraphen für parallele Zweige

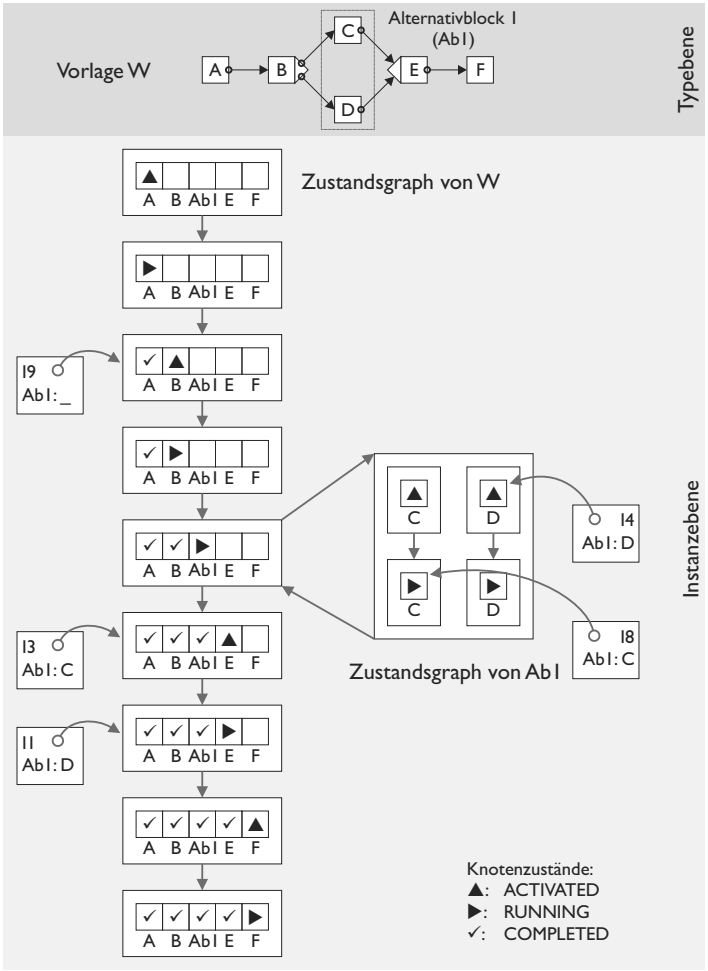


Abbildung 6.9: Getrennte Zustandsgraphen für alternative Zweige

Zustände werden direkt in instanz-individuellen Datenstrukturen geändert. Daneben referenziert jede Prozessinstanz zusätzlich den entsprechenden Zustandsknoten des aggregierten Zustandsgraphen, in dem sie sich befindet. Auf diese Weise können die Verträglichkeitsprüfung bezüglich des Zustands sowie Zustandsanfragen anhand des Zustandsgraphen nicht-betroffene Prozessinstanzen herausfiltern. Anschließend wird die instanz-individuelle Zustandsrepräsentation der verbliebenen Prozessinstanzen genauer überprüft. Eine Variante hiervon, die *grobe Clusterung*, wird in [Glö09] detailliert diskutiert.

Die grobe Clusterung ist auch Grundlage von Laufzeitmessungen mit Clusterung in Kapitel 8.

6.4.2.2 Variante IC-2-2: Meilensteine

Bei Variante IC-2-2 (Meilensteine) werden, wie bei Variante IC-2-1 (Clusterbaum), Prozessschritte aggregiert. Allerdings erfolgt die Aggregation nicht aufgrund der Blockstruktur, sondern sie wird explizit vom Prozessmodellierer vorgegeben. Bei blockorientierten Prozessmetamodellen ist es hilfreich, wenn sich die Aggregation an der Blockstruktur orientiert, dies ist aber nicht zwingend notwendig. Abb. 6.10 zeigt eine Prozessvorlage, die durch 2 Meilensteine in 3 Bereiche unterteilt ist. Jeder dieser Bereiche wird, wie bei Variante IC-2-1, als abstrakter Blockknoten in einem Zustandsgraphen repräsentiert. Die Prozessinstanzen besitzen Datenstrukturen für die Zustände der Prozessschritte des jeweils aktiven Bereichs, die durchlaufenen Zweige alternativer Verzweigungen sowie eine Referenz auf den entsprechenden Zustandsgraphen.

Die Auswirkungen und das Verhalten von Variante IC-2-2 entsprechen den Auswirkungen und dem Verhalten von IC-2-1. Meilensteine haben jedoch den Vorteil, dass die Einteilung durch den Prozessmodellierer explizit erfolgt. Dadurch kann dieser eine für die Verwendung der Prozessvorlage geeignete Aggregation wählen. Grobe Aggregation, also Aggregation vieler Prozessschritte, verringert den Speicherbedarf, während feine Aggregation den Nutzen der Zustandsclusterung für die Funktionen erhöht. Eine Möglichkeit besteht z. B. darin, abhängig von der geschätzten Ausführungszeit von Graphbereichen zu aggregieren. Je länger die Ausführungszeit ist, desto wahrscheinlicher ist, dass in dieser Zeit eine Schemaevolution stattfindet. Dementsprechend können aufeinanderfolgende Prozessschritte, die eine kurze Laufzeit haben und zudem nicht lange im aktivierten Zustand bleiben, grob aggregiert werden, Prozessschritte mit langer Laufzeit oder mit Benutzerinteraktion, d. h. Prozessschritte, die nicht automatisch gestartet werden, werden dagegen eher fein aggregiert. Auf diese Weise kann die Verträglichkeitsprüfung bezüglich des Zustands von der feinen Aggregation im Zustandsgraphen profitieren, während gleichzeitig wenig Speicher für die grobe Aggregation und damit insgesamt für die Zustandsclusterung verwendet wird.

Von der variablen Granularität können auch Zustandsanfragen profitieren. Diese dienen oftmals der Fortschrittskontrolle, die sich normalerweise auch in den Meilensteinen widerspiegelt. Daher rührt auch der Name für Variante IC-2-2 (s. Abb. 6.10). Der Vorteil der expliziten Festlegung der Aggregation ist gleichzeitig der größte Nachteil von Variante IC-2-2. Genauso wie gut gewählte Meilensteine die Leistungsfähigkeit erheblich verbessern, haben schlecht gewählte Meilensteine entsprechend negative Auswirkungen auf den Speicherbedarf und die Laufzeit.

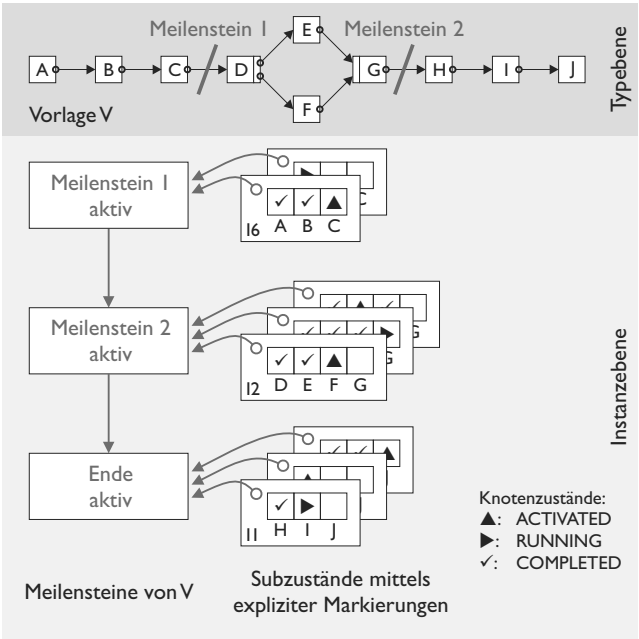


Abbildung 6.10: Meilensteine (IC 2-2)

6.4.3 Variante IC-3: Späte Clustererzeugung

Während die bisher vorgestellten Varianten den Speicherbedarf durch Aggregation und damit Informationsverlust im Zustandsgraphen reduzieren, zielt Variante IC-3 darauf ab, den tatsächlich genutzten Speicher zur Laufzeit zu reduzieren. Dazu wird der Zustandsgraph einer Prozessvorlage nicht vorab erzeugt, sondern nur soweit wie benötigt. Zu einem Zeitpunkt existieren nur diejenigen Zustandsknoten, zu denen gerade mindestens eine Prozessinstanz existiert. Dadurch ist der Speicherbedarf der späten Clustererzeugung nie größer als bei instanz-individueller Zustandsrepräsentation; er ist jedoch maximal so groß wie der vollständige Zustandsraum.

Der geringe Speicherbedarf hat jedoch bedeutende Auswirkungen auf das Weiterschalten. Bei jedem Weiterschalten wird der nächste Zustandsknoten einer Prozessinstanz berechnet. Anschließend wird überprüft, ob dieser Zustandsknoten bereits existiert, so dass die Prozessinstanz dem richtigen Zustandscluster zugeordnet werden kann. Insgesamt ist dieses Vorgehen wesentlich aufwendiger als das Weiterschalten bei instanz-individueller Zustandsrepräsentation. Dies ist kritisch, da das Weiterschalten häufig benötigt wird. Auch die Verträglichkeitsprüfung bezüglich des Zustands gestaltet sich aufwendiger als mit einem vollständigen Zustandsgraphen. Wenn der Zustandsgraph nicht komplett vorliegt, kann die Grenze zwischen verträglichen und unverträglichen Zuständen nicht direkt ermittelt werden.

Auf die Änderungspropagation hat die späte Clustererzeugung kaum Auswirkungen. Während bei vollständigen Zustandsgraphen der neue Zustandsgraph vorab berechnet wird, geschieht dies bei der Änderungspropagation individuell für jeden existierenden Zustandsknoten. Die eigentliche Propagation ist dadurch zwar aufwendiger, allerdings fällt die komplette Erzeugung des neuen Zustandsgraphen weg. Dementsprechend wird auch nur Speicherplatz für die tatsächlich vorhandenen neuen Zustandsknoten belegt. Gut geeignet ist die späte Clustererzeugung für Zustandsanfragen. Da nur benötigte Zustandsknoten existieren, müssen

wenige Daten berücksichtigt werden. Zudem werden die entsprechenden Prozessinstanzen wie bei vollständigen Zustandsgraphen leicht gefunden.

6.4.4 Fazit

Es gibt verschiedene Varianten, den Speicherbedarf bei einer Zustandsclusterung zu verbessern. Dies erfolgt durch Informationsverlust (Aggregation) oder dynamische Erzeugung des Zustandsgraphen. Tabelle 6.2 vergleicht die vorgestellten Varianten bezüglich des Speicherbedarfs und ihrer Auswirkungen auf verschiedene Funktionen. Variante IC-3 (späte Clustererzeugung) hat große Stärken bezüglich des Speicherbedarfs, die negativen Auswirkungen auf das Weiterschalten machen sie jedoch für den praktischen Einsatz untauglich. Alle anderen Varianten erkaufen sich den niedrigeren Speicherbedarf gegenüber vollständigen Zustandsgraphen durch zusätzliche instanz-individuelle Datenstrukturen, die beim Weiterschalten manipuliert und bei Funktionen berücksichtigt werden müssen.

Tabelle 6.2: Vergleich von Instanzclusterungsvarianten

	IC-1 Zustands-aggregation	IC-2-1 Clusterbaum	IC-2-2 Meilensteine	IC-3 Späte Cluster-erzeugung
Speicherbedarf	+	+	+	++
Verträglichkeitsprüfung	O	O	+[1]	-
Änderungspropagation	O	O	O	O
Weiterschalten	+	O	O	--
Zustandsanfragen	-	O	+[1]	++

Bewertung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

[1] bei geeigneten Meilensteinen

Um die Sonderbehandlung zu vermeiden, bietet es sich an, im Normalfall die instanz-individuelle Zustandsrepräsentation zu verwenden. Sind von einer Prozessvorlage viele Prozessinstanzen zu erwarten, erfolgt für diese Prozessvorlage die Zustandsrepräsentation mittels Zustandsclusterung und vollständigen Zustandsgraphen. Daneben können grobe Zustandscluster als Indexstrukturen für solche Prozessvorlagen eingesetzt werden, die häufig im Rahmen einer Schemaevolution geändert werden oder deren Prozessinstanzen häufig Gegenstand von Zustandsanfragen sind.

6.5 Clusterung veränderter Prozessinstanzen

In den vorherigen Abschnitten haben wir die Zustandsclusterung unveränderter Prozessinstanzen behandelt In diesem Abschnitt diskutieren wir kurz veränderte Prozessinstanzen im Kontext einer Zustandsclusterung. Während veränderte Prozessinstanzen durch individuelle Änderungen gekennzeichnet sind, die nur einzelne Prozessinstanzen betreffen, ist das Hauptmerkmal von Zustandsclusterung der Verzicht auf instanz-individuelle Datenstrukturen und die Gruppierung von Prozessinstanzen nach ihrem Zustand. Während instanz-individuelle Datenstrukturen für instanzspezifische Änderungen noch unproblematisch sind (s. Abschnitt 6.4), besteht die Schwierigkeit darin, dass instanzspezifische Änderungen auch Auswirkungen auf den Zustand der betroffenen Prozessinstanzen haben und sich somit auch die Zuordnung zum Zustandsgraphen und sogar der Zustandsgraph selbst ändern.

Beim Weiterschalten halten sich die Auswirkungen noch in Grenzen. So führt das Löschen von Prozessschritten dazu, dass beim Weiterschalten der veränderten Prozessinstanz die entsprechenden Zustandsknoten einfach übersprungen werden. Der Zustand eingefügter Prozessschritte wird wie bei der Aggregation instanz-individuell verwaltet; die Prozessinstanz kann während der Ausführung eingefügter Prozessschritte einen „nahe gelegenen“ oder gar keinen Zustandsknoten referenzieren. Problematisch bei der Zustandsclusterung von veränderten Prozessinstanzen ist die Schemaevolution. Mit Zustandsclusterung werden Änderungen der Schemaevolution bezüglich der Zustände von Prozessinstanzen als unverträglich eingestuft, wenn diese Prozessinstanzen den von Änderungen der Schemaevolution betroffenen Bereich bereits durchlaufen haben – ausgenommen abgewählte alternative Verzweigungen. Darunter fallen auch solche Prozessinstanzen, die die Änderungen der Schemaevolution instanzspezifisch vorweggenommen haben. Bezüglich solcher Prozessinstanzen sind die Änderungen der Schemaevolution aber sowohl bezüglich des Zustands als auch bezüglich der Prozessstruktur verträglich (s. Abschnitt 2.3.2.3). Um die Verträglichkeit zu bestimmen, müssen alle veränderten Prozessinstanzen außerhalb des Zustandsgraphen einer entsprechenden Überprüfung unterzogen werden. Insofern fällt der Vorteil einer Zustandsclusterung dann weg.

Für Zustandsanfragen an veränderte Prozessinstanzen ermöglicht der Zustandsgraph, wie bei den Varianten der Zustandsclusterung, lediglich eine grobe Einteilung. Er ist somit bei veränderten Prozessinstanzen nur eine Indexstruktur (s. Abschnitt 6.4). Insgesamt ist der Nutzen einer Zustandsclusterung bei veränderten Prozessinstanzen so gering, dass sich ihre Integration in Zustandsbäume nicht lohnt. Veränderte Prozessinstanzen werden deshalb am besten herkömmlich repräsentiert, d. h. mit instanz-individueller Datenstrukturen für die Zustände. Dazu wird die Datenstrukturen einer Prozessinstanz bei ihrer Änderung transformiert und die Prozessinstanz aus vom Zustandsgraphen entkoppelt.

6.6 Zusammenfassung

Zustandsclusterung beruht darauf, dass die durch Prozessvorlagen definierten Zustandsräume vorab berechnet und bereitgestellt werden. Dadurch entfallen große Teile der instanz-individuellen Zustandsberechnungen bei den entsprechenden Algorithmen, etwa dem Weiterschalten. Zustandsabhängige Überprüfungen lassen sich so auch für Prozessvorlagen und somit für alle unveränderten Prozessinstanzen auf einmal durchführen. Hier kann, wie bei der Verträglichkeitsprüfung bezüglich des Zustands im Kontext einer Schemaevolution, zudem die Struktur des Zustandsraums ausgenutzt werden. Dies vereinfacht einerseits die Überprüfung selbst, andererseits können Teile der Überprüfung wegfallen. Ohne Zustandsclusterung werden Prozessinstanzen individuell überprüft, wodurch dieselbe Überprüfung eventuell mehrmals durchlaufen wird, etwa wenn mehrere Prozessinstanzen im selben Zustand sind, dieser Zusammenhang aber ohne Zustandsclusterung nicht erkannt wird.

Sinnvollerweise werden die Zustandsräume durch Datenstrukturen bei den jeweiligen Prozessvorlagen realisiert. Dadurch fallen große Teile der individuellen Datenstrukturen von Prozessinstanzen weg, d. h. Prozessinstanzen werden leichtgewichtiger. Dementsprechend sind auch weniger Anpassungen dieser Datenstrukturen nötig, etwa in Zusammenhang mit der Zustandsanpassung beim Weiterschalten. Trotz der großen Zustandsräume, insbesondere bei parallelen Verzweigungen, benötigt deren Repräsentation bei großer Anzahl von Prozessinstanzen weniger Speicher als die instanz-individuellen Zustandsrepräsentation. Dies trifft vor allem

dann zu, wenn es mehr Prozessinstanzen einer Prozessvorlage gibt, als deren Zustandsgraph Knoten hat.

Ein weiterer Vorteil der Zustandsclusterung betrifft die Effizienz von Anfragen bezüglich Instanzzuständen. Jeder Zustandsgraph ist gleichzeitig auch eine Indexstruktur für Instanzzustände, was sich gut für die Überwachung oder für Analysen laufender Prozesse (*Process Performance Management*) [GCC+04] verwenden lässt. Allerdings ist diese Zusammenfassung von Prozessinstanzen auch ein Problem für instanzspezifische Änderungen, da diese Änderungen ja gerade individuell für eine Prozessinstanz sind.

7 Konzeption und Architektur eines adaptiven Prozessmanagementsystems

In diesem Kapitel stellen wir die Konzeption und Architektur eines adaptiven PrMS vor. Ausgangspunkt unserer Betrachtungen bilden die in Kapitel 2 behandelten Grundlagen, die in Kapitel 3 diskutierten funktionalen und nicht-funktionalen Anforderungen an adaptive PrMS sowie die in den Kapiteln 4–6 vorgestellten Konzepte für die Repräsentation von Prozessvorlagen und -instanzen. Zuerst geben wir einen Überblick zur Gesamtarchitektur eines adaptiven PrMS und seiner einzelnen Dienste. Anschließend gehen wir auf die umfassenden Datenmodelle ein, mit denen Informationen zwischen den Diensten ausgetauscht werden, bevor wir diese Dienste dann im Detail vorstellen. Die Schnittstellen der Dienste bilden gleichzeitig die Programmierschnittstelle, die wir im darauffolgenden Abschnitt näher erläutern. Abschließend zeigen wir, wie die vorgestellte Architektur die funktionalen und nicht-funktionalen Anforderungen aus Kapitel 2 umsetzt.

7.1 Gesamtarchitektur

Abb. 7.1 zeigt die in dieser Arbeit entwickelte Gesamtarchitektur als abstraktes Schichtenmodell. Eine Schicht oder Ebene umfasst dabei alle Funktionen derselben logischen Abstraktionsebene. Die Architektur ist im Sinne einer serviceorientierten Architektur [Erl05] strikt dienstorientiert. Das heißt, Funktionen sind in Einheiten (sog. Dienste) gruppiert und zwar so, dass die Dienste weitgehend unabhängig voneinander sind und eine sauber definierte Schnittstelle besitzen (sog. lose Kopplung). Demzufolge haben die Schichten auch mehr einen strukturierenden Charakter. So ist es möglich und auch sinnvoll, dass beispielsweise Benutzerinteraktionsdienste (*User Interaction Layer*) ihre Daten direkt von Diensten unterer Schichten (*Basic Services Layer*) holen, etwa bei der Anzeige einer Prozessinstanz, und die entsprechende Anfrage zwei Schichten (*Abstraction Layer* und *Execution Layer*) überspringt. Die Informationen werden zwischen den Diensten über Datenmodelle ausgetauscht, die über Schnittstellen definiert sind. Dadurch lässt sich eine strikte Trennung zwischen Daten und Anwendungslogik erzielen.

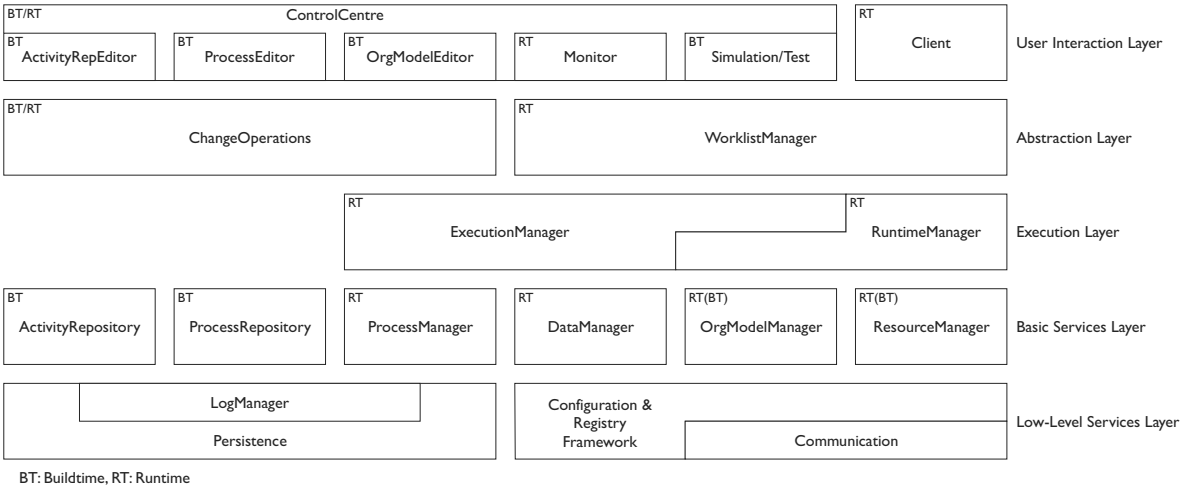


Abbildung 7.1: Gesamtarchitektur

Die Dienste lassen sich in Dienste für die Modellierung (*Buildtime*) und Ausführung (*Runtime*) unterteilen, wobei manche Dienste sowohl für die Modellierung als auch für die Ausführung relevant sind. Im einzelnen gibt es folgende Dienste:

Die Ebene *Low-Level Services Layer* stellt grundlegende, nicht-prozessspezifische Funktionen für die anderen Dienste bereit.

- Der Dienst *Persistence* dient der dauerhaften Speicherung für die von einem beliebigen anderen Dienst verwalteten Daten in einem SQL-DBMS. Dabei wird von der zugrundeliegenden herstellerspezifischen SQL-Umsetzung weitgehend abstrahiert.

- Der Dienst *LogManager* bietet eine allgemeine Schnittstelle für Protokollinformationen. Diese Schnittstelle ähnelt der von relationalen DBMS, ermöglicht aber zusätzlich hierarchische Strukturen, um beispielsweise Protokollinformationen nach Prozessinstanzen strukturieren zu können.

- Der Dienst *Configuration & Registry Framework* bildet die grundlegende Infrastruktur, um andere Dienste zu starten und zu beenden sowie Abhängigkeiten zwischen Diensten (etwa zu *Persistence*) zur Laufzeit aufzulösen.

- Der Dienst *Communication* kapselt die Logik zum entfernten Aufruf eines Dienstes. Er verbirgt somit die für den entfernten Aufruf notwendige Kommunikation.

Die Ebene *Basic Services Layer* stellt grundlegende, prozessspezifische Funktionen für die übergeordneten Dienste bereit.

- Der Dienst *ActivityRepository* verwaltet die Informationen zu Anwendungsbausteinen, die mit Prozessschritten verknüpft sind.

- Der Dienst *ProcessRepository* ist eine Versionsverwaltung für alle erstellten Prozessvorlagen. Werden Prozessvorlagen weiterentwickelt, protokolliert das *ProcessRepository* die jeweiligen Änderungen und die Beziehung sowie die strukturelle Differenz voneinander abgeleiteter Prozessvorlagen.

- Der Dienst *ProcessManager* verwaltet alle laufenden Prozesse, d. h. er stellt Prozessinstanzen und deren Prozessvorlagen zur Verfügung.
- Der Dienst *DataManager* verwaltet die in einem laufenden Prozess anfallenden Datenelementwerte. Diese bilden unter anderem die Grundlage für Verzweigungs- und Rücksprungentscheidungen in Prozessen.
- Der Dienst *OrgModelManager* verwaltet das Organisationsmodell, das die Grundlage für die Zuordnung von Prozessschritten zu Bearbeitern bildet.
- Der Dienst *ResourceManager* verwaltet benötigte Ressourcen (z. B. spezielle Hardware wie Drucker) und ist für Zeitberechnungen in Prozessen zuständig.

Die Ebene *Execution Layer* interpretiert Prozesse und führt diese, oder genauer gesagt deren Prozessschritte, aus.

- Der Dienst *ExecutionManager* nutzt die von *ProcessManager* und *DataManager* bereitgestellten Daten zur Prozessinterpretation, d. h. dem Weiterschalten von Prozessen.
- Der Dienst *RuntimeManager* führt die an Prozessschritten hinterlegten Anwendungsbausteine auf Anforderung durch den *ExecutionManager* aus.

Die Ebene *Abstraction Layer* abstrahiert von Prozessstrukturen (z. B. Knoten und Kanten) in Form von Änderungsoperationen und Arbeitslisten.

- Der Dienst *ChangeOperations* bietet die Änderungsoperationen des ADEPT-Metamodells als Programmierschnittstelle sowie die zugehörige Logik. Hierzu zählt unter anderem die Protokollierung von Änderungen.
- Der Dienst *WorklistManager* bildet die Zustände von Prozessinstanzen in Form von Arbeitslisten und Arbeitslisteneinträgen ab, weist Prozessschritte autorisierten Bearbeitern zu und ermöglicht auf Arbeitslisteneinträgen weitergehende Funktionen (z. B. Eskalation und Delegation).

Die Ebene *User Interaction Layer* umfasst alle Anwendungen für die Interaktion von Benutzern mit dem PrMS.

- Das *ControlCentre* ist eine Sammlung von Anwendungen für Administratoren und Prozessmodellierer:
  - *ActivityRepositoryEditor* ist die graphische Oberfläche für das *ActivityRepository*.
  - *ProcessEditor* dient zur Modellierung von Prozessvorlagen und als graphische Oberfläche für das *ProcessRepository* und den *ProcessManager*.
  - *OrgModelEditor* ist die graphische Oberfläche für den *OrgModelManager*.
  - *Monitor* erlaubt die Überwachung der laufenden Prozessinstanzen aus dem *ProcessManager* sowie der zugehörigen Datenelementwerte aus dem *DataManager*.
  - *Simulation/Test* ermöglicht die testweise Ausführung von Prozessvorlagen, hauptsächlich direkt nach deren Erstellung oder Änderung im *ProcessEditor*.
- *Client* ist die Anwendung für die Ausführung von Prozessen und die Abarbeitung von Prozessschritten.

Der strikten Trennung von Daten und Anwendungslogik folgend, gehen wir im nächsten Abschnitt auf die Datenmodelle ein. Diese repräsentieren die Daten, die von Diensten verwaltet oder zwischen verschiedenen Diensten ausgetauscht werden. Die Datenmodelle bilden die Grundlage der von den Diensten angebotenen Funktionen. Die Dienste und ihre Funktionen diskutieren wir im Detail im Anschluss an die Datenmodelle.

7.2 Datenmodelle

In diesem Abschnitt stellen wir die Datenmodelle vor, welche die von den Diensten verwalteten und verwendeten Daten repräsentieren und mit denen Informationen zwischen den Diensten ausgetauscht werden. Die Datenmodelle setzen die in Kapitel 2 vorgestellten Grundlagen um, also das ADEPT-Metamodell, das ADEPT-Änderungsmodell und die Anwendungsintegration. Neben diesen funktionalen Anforderungen sind insbesondere die nicht-funktionalen Anforderungen der Erweiterbarkeit (NA 3.2), der Unabhängigkeit/Modularität (NA 5.1) sowie der Wiederverwendung/Redundanzvermeidung (NA 6) für die Konzeption der Datenmodelle besonders wichtig.

Die nicht-funktionale Anforderung der Erweiterbarkeit wird über die Datenmodelle selbst realisiert (s. Abschnitt 7.2). Die Modularität der Datenmodelle ergibt sich wie bei den Diensten aufgrund der entsprechenden Funktionen. Daraus ergeben sich z. B. ein Datenmodell für die Prozessmodellierung und eines für die Prozessausführung. Gleichzeitig sind die einzelnen Datenmodelle so umfassend, dass sie von mehreren Diensten verwendet werden können, etwa um Redundanz zu vermeiden. Dementsprechend ist ein Datenmodell für eine Teilmenge der Dienste relevant. Vor einer Detailbeschreibung diskutieren wir, welche Dienste dies für das entsprechende Datenmodell sind.

Die Datenmodelle werden über Schnittstellen, genauer die entsprechenden Methodendeklarationen, definiert. Dadurch wird die Implementierung der Datenmodelle unabhängig von den verwendenden Diensten gestaltet. Sie ist somit austauschbar. Insbesondere legen die Datenmodelle auch keine Implementierungssprache fest. Für eine einfache und einheitlicher Darstellung orientieren wir uns im folgenden an Java [jav]. Dies gilt beispielsweise für Datentypen, Namenskonventionen oder Annotationen für Metainformationen zur Implementierung.

Die Austauschbarkeit der Datenmodellimplementierungen erfordert spezielle (*Factory*-) Objekte. Diese besitzen Fabrikmethoden (*Factory Method*, [GHJV95]) zur Erzeugung von Objekten der entsprechenden Implementierungsklassen. Auch (*Factory*-)Objekte sind über ihre Schnittstelle in Form von Methodendeklarationen charakterisiert; die Implementierung der (*Factory*-)Objekte wird über die zentrale Registrierung (s. Abschnitt 7.3.10.1) zur Laufzeit bereitgestellt.

Darstellung der Datenmodelle Die Datenmodelle werden im folgenden mittels UML-2-Klassendiagrammen (*Unified Modeling Language* [uml]) dargestellt. Eine Übersicht zu den verwendeten UML-Elementen findet sich in Anhang B. Obwohl die Datenmodelle über Methodendeklarationen definiert sind, werden in den Klassendiagrammen meistens Attribute verwendet. Dies ist lediglich der einfacheren und kompakteren Darstellung von Attributen gegenüber Operationen in UML-Klassendiagrammen geschuldet. Ein Attribut `Name:String` im Diagramm entspricht der Methodendeklaration `+getName():String` im Datenmodell.

Lesende Zugriffe reichen für die meisten Attribute aus, da sie nach Erzeugung der entsprechenden Objekte zur Laufzeit nicht mehr geändert werden müssen oder gar dürfen. Falls ein

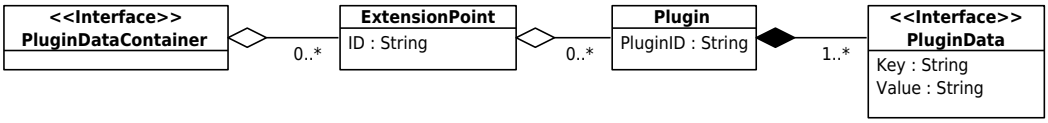


Abbildung 7.2: UML-Klassendiagramm der logischen Struktur von *PluginDataContainer*

Attribut noch geändert werden kann, wird dies explizit gekennzeichnet. Ein änderbares Attribut repräsentiert zusätzlich die Methodendeklaration `+setName(name:String):void`. Bietet ein Element eines Datenmodells komplexere Logik als das Lesen eines einfachen Attributwerts, etwa der Berechnung von Daten, wird hierfür im Klassendiagramm eine Operation verwendet. In diesem Fall ist die Klasse im Diagramm auch explizit mit dem Stereotyp *Interface* gekennzeichnet.

Aufgrund des Umfangs der Datenmodelle werden diese typischerweise auf mehrere Klassendiagramme verteilt, von denen jedes einen bestimmten Aspekt des Datenmodells zeigt. Bezüge zwischen verschiedenen Klassendiagrammen eines Datenmodells werden durch Klassen hergestellt, die in mehreren Diagrammen vorkommen.

7.2.1 Allgemeine Entitäten der Datenmodelle

Mehrere Entitäten werden datenmodellübergreifend verwendet und sind dementsprechend in verschiedenen Klassendiagrammen zu finden. Zum einen sind dies Entitäten für die Erweiterbarkeit (s. Abschnitt 7.2.1.1) der Datenmodelle (*UserAttributeContainer*, *PluginDataContainer*). Diese werden in allen Datenmodellen verwendet. Zum anderen werden Konfigurationen und Konfigurationsdefinitionen in mehreren Datenmodellen verwendet (*ConfigurationEntry*, *Configuration, ActivityConfiguration*).

7.2.1.1 Erweiterbarkeit

Die Schnittstelle *UserAttributeContainer* dient der Erweiterung der Datenmodelle. Sie spezifiziert Methoden, die Zugriff auf eine Menge von Paaren von Zeichenketten bieten (entsprechender Java-Datentyp `Map<String, String>`). Dadurch können entsprechende Klassen beliebige zusätzliche Attribute besitzen, die nicht in der festgelegten Schnittstelle vorgesehen sind. Die Datenstruktur nimmt Attributnamen und -werte auf. Beide müssen als Zeichenketten vorliegen, was für Attributwerte eine leichte Einschränkung darstellt. Durch die Festlegung auf Zeichenketten müssen Zugriffe auf einen Attributwert ggf. den Datentyp konvertieren, beispielsweise von einer Zeichenkette zu einer Fließkommazahl und zurück. Es muss gewährleistet sein, dass alle Zugriffe dieselbe Konversion vornehmen. Es darf nicht passieren, dass eine Fließkommazahl geschrieben und an anderer Stelle als Ganzzahl ausgelesen wird. Dies führt in der Regel zu unnötigen Laufzeitfehlern.

Die Schnittstelle *PluginDataContainer* erfüllt denselben Zweck wie *UserAttributeContainer*, erlaubt aber einen wesentlich einfacheren und zudem besser typisierten Zugriff auf die zusätzlichen Attribute. Die logische Struktur von *PluginDataContainer* (s. Abb. 7.2) beinhaltet auch wieder die Menge von Paaren von Zeichenketten (*PluginData*, s. Abb. 7.2). Jede dieser Menge von Paaren von Zeichenketten besitzt einen Namen (*PluginID*) und bildet die Attributmenge eines *Plugins*. Mehrere *Plugins* werden in einem *ExtensionPoint* zusammengefasst, der einen bezüglich *PluginDataContainer* eindeutigen Namen besitzt (*ExtensionPointID*). *PluginDataContainer* ist die Schnittstelle, die von zahlreichen Elementen in den Datenmodellen

Abbildung 7.3: UML-Klassendiagramm von *PluginDataContainer*

implementiert und damit die beschriebene Datenstruktur zur Verfügung stellt (s. Abb. 7.3). Gegenüber *UserAttributeContainer* besitzt sie also lediglich zwei zusätzliche Hierarchieebenen (*ExtensionPoint* und *Plugin*).

Der im Gegensatz zu *UserAttributeContainer* typsichere Zugriff auf die Daten von *PluginDataContainer* wird über separate Schnittstellen ermöglicht. Diese Schnittstellen besitzen einfache Methoden zum Lesen und Schreiben der Attribute von *PluginData* und werden für ein bestimmtes *Plugin* definiert. Die Methoden der Schnittstellen sind entsprechend annotiert, wobei die Annotationen angeben, welches *PluginData*-Tupel gelesen oder geschrieben wird. Diese Informationen erlauben einer generischen Klasse, eine Implementierung für alle Schnittstellen bereitzustellen. Das heißt, für alle Schnittstellen existiert nur eine Implementierung, die mittels Reflexion die jeweilige Schnittstelle implementiert und ihre Methoden auf die entsprechenden *PluginData*-Tupel abbildet. Naheliegenderweise werden hier, wie bei *UserAttributeContainer,* alle Attributwerte als allgemeine Zeichenketten repräsentiert. Allerdings transformiert die generische Klasse die Typen automatisch entsprechend der Methodenannotationen in Zeichenketten und zurück.

7.2.1.2 Konfigurationen

Wie in den Grundlagen erläutert (s. Abschnitt 2.4.1), besitzen Anwendungsbausteine neben Eingabeparametern häufig noch Konfigurationsinformationen. Für einen möglichst flexiblen Einsatz solcher Anwendungsbausteine in unterschiedlichem Prozesskontext ist es sinnvoll, Eingabeparameter und Konfigurationswerte nicht strikt zu trennen, sondern Eingabeparameter bei der Modellierung mit festen Werten zu versehen (analog zu Konfigurationswerten, s. Beispiel 2.12) sowie Konfigurationswerte aus dem Datenfluss eines Prozesses zu bestimmen (s. Beispiel 2.11).

Eine Besonderheit von Konfigurationen ist die implizite Hierarchiebildung. Bezüglich Vererbung verhalten sich Konfigurationen wie normale Objektattribute. Das heißt, erbt eine Entität B von einer Entität A und hat A eine Konfiguration AK, wie in Abb. 7.4 gezeigt, so hat B diese Konfiguration AK implizit auch. Hat B zusätzlich eine eigene Konfiguration BK, so besitzt diese die Werte von AK und kann diese somit überschreiben. Die Generalisierungsbeziehung zwischen AK und BK in Abb. 7.4 ist damit redundant. Die implizite Hierarchiebildung ist insbesondere für die Umsetzung der Anwendungsintegration durch das Aktivitätenmodell (s. Abschnitt 7.2.2) nützlich.

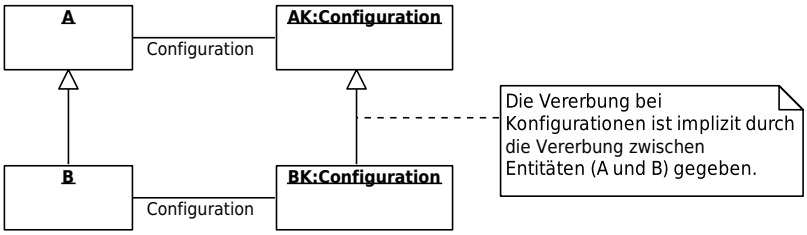


Abbildung 7.4: UML-Objektdiagramm der Konfigurationshierarchie

Beispiel 7.1 (Implizite Konfigurationshierarchie in Datenmodellen)
Ein Datenmodell besitzt die Entitäten A und B, wobei B eine Spezialisierung von A ist. B erbt demnach die Attribute von A, kann deren Werte aber auch überschreiben. Besitzt A die Konfiguration AK mit den Einträgen `AK1=1`, `AK2=2` und `AK3=3`, so umfasst die Vererbung auch die Konfiguration. Das heißt, die Konfiguration BK von B erbt von AK. So besitzt BK neben den eigenen Einträgen `BK1=4`, `BK2=5` und `BK3=6` auch die Einträge `AK1=1`, `AK2=2` und `AK3=3`. Die Konfigurationswerte können auch überschrieben werden, in BK etwa `AK1=7`, `AK2=8` und `AK3=9`.

Damit Konfigurationswerte erst zur Laufzeit eines Prozesses gesetzt werden müssen, trennt das Datenmodell zwischen den eigentlichen Konfigurationswerten (*Configuration*, s. Abb. 7.5) und deren Definition (*ConfigurationDescription*). Die Definition eines Konfigurationswerts (*ConfigurationEntry*) besteht aus seinem innerhalb der Definition und Konfigurationshierarchie eindeutigen Namen (`Name`), einer Beschreibung (`Description`) und einem Datentyp (`DataType`). Ferner umfasst die Definition eines Konfigurationswerts die Modalität (`Optional`) und gibt optional Einschränkungen für den Datentyp (`Restrictions`) an. Ein Beispiel hierfür ist ein regulärer Ausdruck, der mögliche Zeichenketten einschränkt. Da Konfigurationswerte aus dem Datenfluss eines Prozesses stammen können, sind dieselben Datentypen zulässig wie beim Datenfluss (`AdeptDataType`), u, a, Zeichenketten, Ganz- und Gleitkommazahlen, Boolesche Werte und benutzerdefinierte Datentypen.

Beispiel 7.2 (Definition eines Konfigurationswerts für Postleitzahlen)
Die Definition eines Konfigurationswerts für eine deutsche Postleitzahl wäre demnach wie folgt (deutsche Postleitzahlen beginnen nicht mit 00):
`Name`: PLZ, `Description`: Postleitzahl, `DataType`: `AdeptDataType.STRING`, `Optional`: `false`, `Restrictions`: ^([0]{1}[1-9]{1}|[1-9]{1}[0-9]{1})[0-9]{3}$

Der Zugriff auf Konfigurationswerte erfolgt typsicher über die Schnittstelle *Configuration* (s. Abb. 7.5). Deren Methoden `isFixed` und `getConfigurationEntry` bieten weitergehende Möglichkeiten für das Überschreiben von Konfigurationen bei Konfigurationshierarchien. Festgelegte Konfigurationswerte (`isFixed`) verhindern das Überschreiben; `getConfigurationEntry` gibt den Namen der Konfiguration zurück (oder genauer des entsprechenden Modellelements), die den entsprechenden Konfigurationswert liefert.

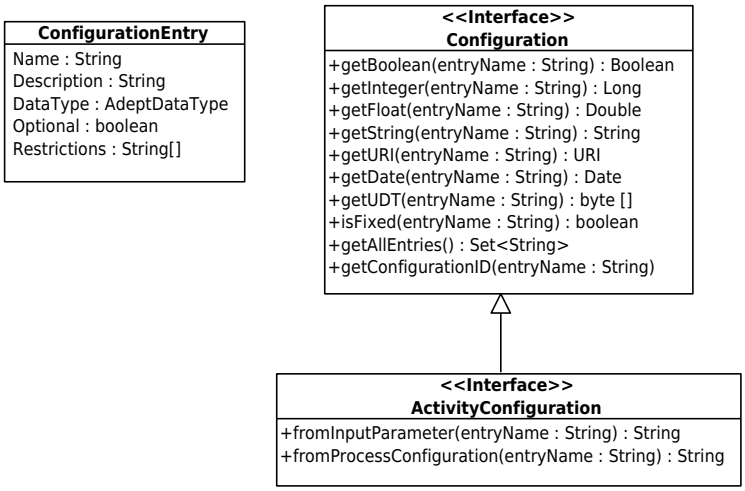


Abbildung 7.5: UML-Klassendiagramm der Konfiguration

Beispiel 7.3 (`getConfigurationEntry beim Überschreiben von Konfigurationen`)
Die Entität B mit Konfiguration BK ist eine Spezialisierung von Entität A und deren Konfiguration AK, wie in Abb. 7.4 dargestellt. BK ist Kindkonfiguration der Konfiguration AK. Bei Zugriffen auf BK werden die Werte von AK bereits berücksichtigt. Damit ein Aufrufer unterscheiden kann, woher ein Konfigurationswert stammt, gibt `getConfigurationEntry` eine Referenz auf B zurück, wenn BK den Wert von AK überschreibt. Überschreibt BK den Wert von AK nicht, gibt `getConfigurationEntry` entsprechend eine Referenz auf A zurück.

ActivityConfiguration Innerhalb eines Prozesses werden bestimmte Konfigurationswerte häufig wiederverwendet. Dies kann durch Wiederverwendung desselben Anwendungsbausteins für mehrere Prozessschritte verursacht werden, oder verschiedene Anwendungsbausteine verwenden innerhalb eines Prozesses dieselbe Konfiguration. Ein Beispiel ist die Verwendung derselben Datenbankkonfiguration innerhalb eines Prozesses. Damit diese gemeinsame Konfiguration nicht an den entsprechenden Aktivitäten redundant gesetzt werden muss (s. Abb. 7.6, links), gibt es Prozesskonfigurationen, die für alle Aktivitäten eines Prozesses sichtbar sind. Entsprechend wird in den Datenmodellen zwischen einer normalen *Configuration* (u. a. für Prozesse) und einer *ActivityConfiguration* (für Aktivitäten/Anwendungsbausteine) unterschieden.

Bei einer *ActivityConfiguration* können die Konfigurationswerte aus der übergeordneten Prozesskonfiguration (*Configuration* an einem *Template*) stammen (s. Abb. 7.6, rechts). Außerdem erlaubt eine *ActivityConfiguration* die Übernahme von Konfigurationswerten aus dem Datenfluss (s. Abb. 7.6, Mitte). Die Anbindung an den Datenfluss erfordert Eingabeparameter und ist dementsprechend nur bei Entitäten sinnvoll, die auch Eingabeparameter haben können, etwa Aktivitäten oder (Sub-)Prozessen. Die Methoden der Schnittstelle *ActivityConfiguration* geben entsprechend den Namen des Konfigurationswerts des entsprechenden Prozesses (`fromProcessConfiguration`) oder des dazugehörigen Eingabeparameters (`fromInputParameter`) zurück.

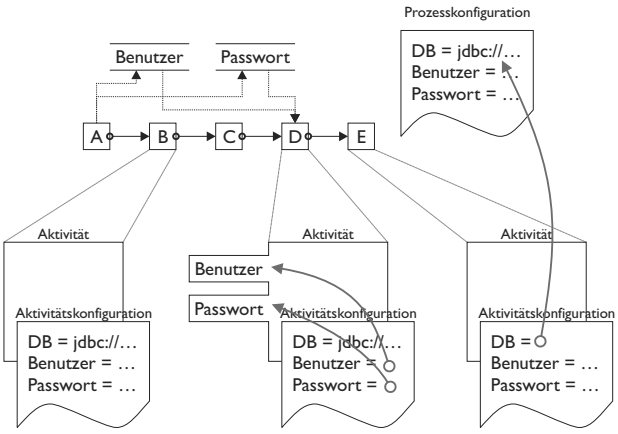


Abbildung 7.6: *ActivityConfiguration*

7.2.2 Aktivitätenmodell

Das Aktivitätenmodell wird zur Verwaltung (und Versionierung) von Aktivitätenvorlagen verwendet, also den Repräsentationen für die an einem Prozessschritt hinterlegbaren Anwendungsbausteine. Somit ist das Aktivitätenmodell für eben diese Verwaltung relevant (*ActivityRepository*), den entsprechenden Editor (*ActivityRepositoryEditor*) sowie dem Editor zur Erstellung von Prozessvorlagen (*ProcessEditor*) (s. Abb. 7.7). Im *ProcessEditor* werden Aktivitätenvorlagen aus dem Aktivitätenmodell mit Prozessschritten verknüpft und entsprechend zu Aktivitäten transformiert, die später im Rahmen der Prozessausführung die Ausführung der entsprechenden Anwendungsbausteine ermöglichen.

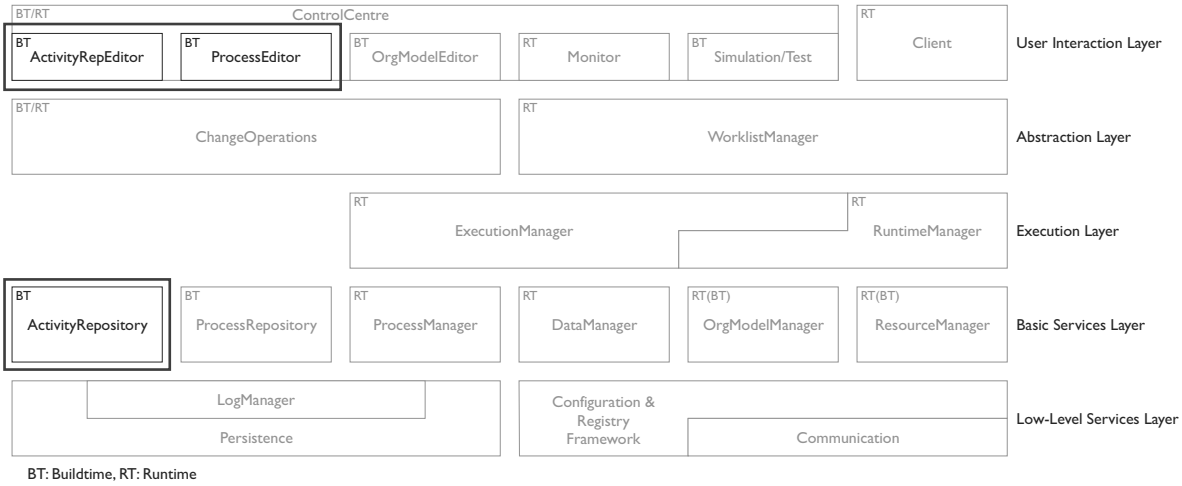


Abbildung 7.7: Verwendung des Aktivitätenmodells in der Gesamtarchitektur

7.2.2.1 Aktivitätenvorlagen

Das Aktivitätenmodell (s. Abb. 7.8, 7.9 und 7.10) stellt sämtliche Informationen zur Verfügung, um einen Anwendungsbaustein bei der Prozessmodellierung an einem Prozessschritt zu hinterlegen und ihn zur Laufzeit auszuführen. Das Aktivitätenmodell und seine Entitäten

selbst sind jedoch an der Ausführung nicht direkt beteiligt. Die wichtigste Klasse des Aktivitätenmodells ist die Aktivitätenvorlage (*ActivityTemplate*). Diese besitzt neben einer eindeutigen Kennung einen aus vier Teilen zusammengesetzten Namen (`ExecutableComponentName`, `OperationName`, `ConfigurationName`, `ActivityName`), verschiedene Metainformationen für die Modellierung (z. B. `Description`, `State`, `IconID`) sowie die für die Ausführung relevanten Attribute, etwa den Namen der den Anwendungsbaustein implementierenden Java-Klasse (`ImplementationClass`) und die zur Laufzeit benötigten Bibliotheken (`ClassPath`).

Eine Aktivitätenvorlage besitzt Ein-/Ausgabeparameter (Entität *ATParameter* als `ReadParameters` und `WrittenParameters`), welche die technischen Ein-/Ausgabeparameter des entsprechenden Anwendungsbausteins repräsentieren. Ein *ATParameter* hat einen Namen, der bezüglich *ActivityTemplate* und der Parameterrichtung (Ein- oder Ausgabe) eindeutig sein muss. Zusätzlich hat *ATParameter* Typinformationen (`DataType`, `UDTName`, `IdentifierID`) für den entsprechenden Parameter und Bitschalter (`Optional`, `Virtual`, `Failure`), die bei der Prozessmodellierung für die Anbindung an den Datenkontext relevant sind.

Generische Aktivitätenvorlagen Eine vorgegebene Menge an Ein-/Ausgabeparametern reicht zwar für viele Anwendungsbausteine aus, für solche mit generischer Schnittstelle gilt dies jedoch nicht. Bei diesen Anwendungsbausteinen stehen die Ein-/Ausgabeparameter erst zur Laufzeit fest und sind bis dahin in bestimmten Grenzen variabel. Die Ein-/Ausgabeparameter können z. B. von der Konfiguration oder Werten bestimmter (fester) Ein-/Ausgabeparameter abhängen. Das folgende Beispiel verdeutlicht dies:

Beispiel 7.4 (Anwendungsbaustein mit variablen Parametertypen)
Ein Anwendungsbaustein zur XML-Transformation (*Extensible Markup Language*) besitzt zwei Eingabeparameter (das zu transformierende XML-Dokument und die Transformationsvorschrift als XSLT (*XSL Transformation*)) sowie einen Ausgabeparameter für das transformierte XML-Dokument. Die Transformation findet auf der einem DOM-Schnittstelle (*Document Object Model*) statt. Da DOM sowohl aus Zeichenketten (`STRING`) als auch aus XML-Dateien (Binärdaten, `USERDEFINED`) erzeugt werden kann, ist beides sowohl für das zu transformierende XML-Dokument als auch für die Transformationsvorschrift in Form von XSLT zulässig. Auch die Ausgabe beruht auf DOM und kann somit entweder Zeichenkette oder XML-Datei sein. Dies gibt dem Prozessmodellierer die Möglichkeit, den Datentyp der umgebenden Prozessschritte zu verwenden und somit die Hinzunahme zusätzlicher Prozessschritte zur Realisierung der gewünschten Transformation zu vermeiden.

Bei generischen Aktivitätenvorlagen dürfen Ein-/Ausgabeparameter also noch nicht fest im Aktivitätenmodell hinterlegt werden. Allerdings sind durchaus Einschränkungen möglich, im Beispiel etwa sind genau zwei Datentypen für die Parameter zulässig: `STRING` oder `USERDEFINED`. Um Fehler bei der Prozessmodellierung so gut es geht zu vermeiden, können die Parameter im Aktivitätenmodell vormodelliert werden. Hierzu dienen die Parametervorlagen (*ATParameterTemplate*). Diese besitzen dieselben Attribute wie *ATParameter*, sind aber selbst keine Parameter, sondern lediglich Schablonen für Parameter, die bei der Prozessmodellierung angelegt werden.

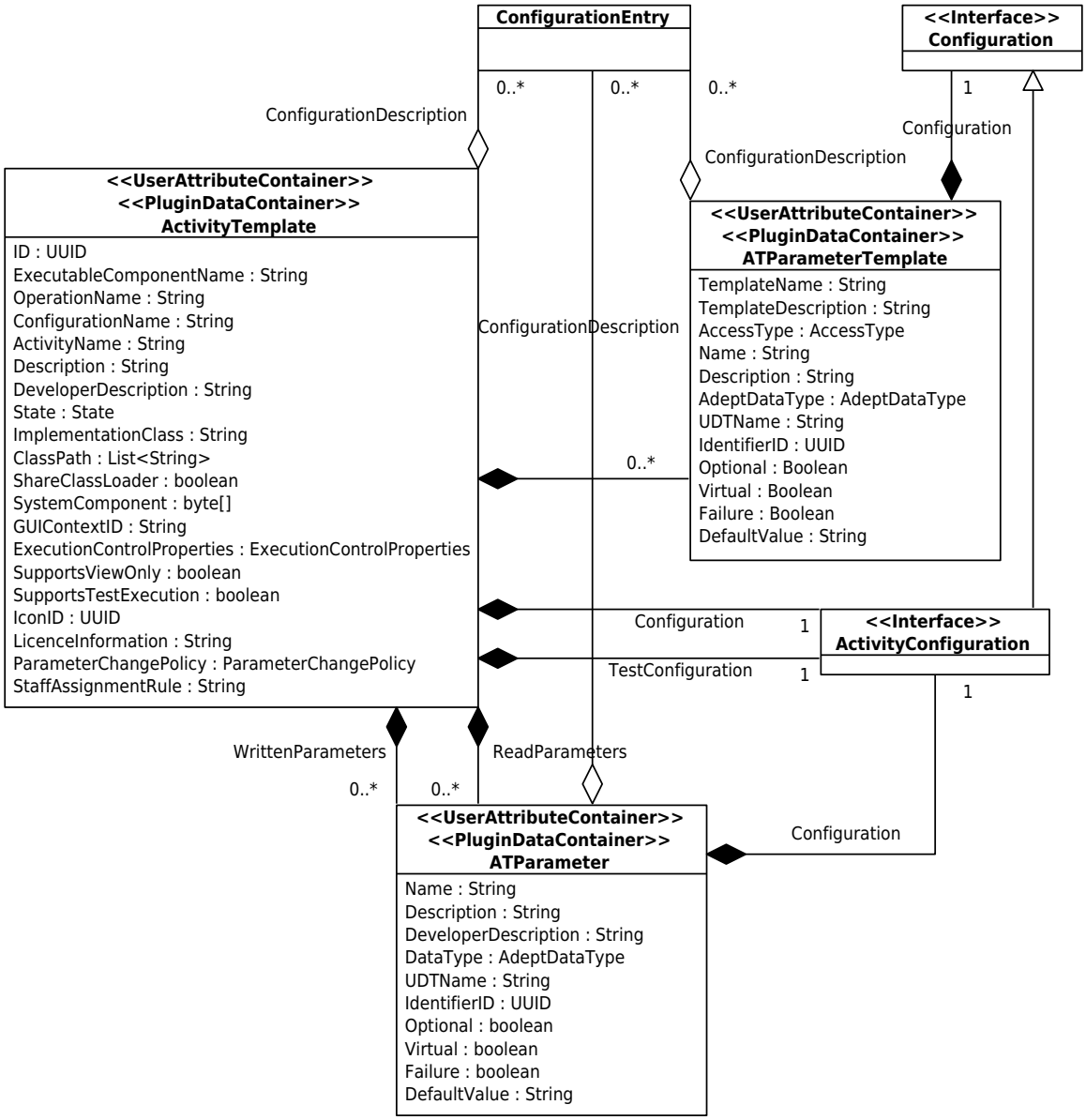


Abbildung 7.8: UML-Klassendiagramm der Aktivitätenvorlagen des Aktivitätenmodells

Beispiel 7.5 (Parametervorlagen für Anwendungsbausteine)
Ein Anwendungsbaustein zur XML-Transformation kann sowohl Zeichenketten (`STRING`) als auch XML-Dateien (Binärdaten, `USERDEFINED`) verarbeiten. Die entsprechende Aktivitätenvorlage besitzt keine echten Ein-/Ausgabeparameter, sondern nur zwei Vorlagen für jeden Ein-/Ausgabeparameter. Die zwei Vorlagen eines Ein-/Ausgabeparameters unterscheiden sich lediglich im Datentyp (`AdeptDataType`); eine Vorlage besitzt den Datentyp `STRING`, die andere `USERDEFINED`.

Die Ein-/Ausgabeparameter von Anwendungsbausteinen können in vielen Aspekten variieren, vom einfachen Austausch eines Parameters wie im vorangehenden Beispiel bis hin zu einer Schnittstelle, die beliebige Ein-/Ausgabeparameter erlaubt. Deshalb stellt bei Parametervorlagen nur der Name (`TemplateName`) ein erforderliches Attribut dar. Alle anderen Attribute können undefiniert bleiben. Diese müssen dann vom Prozessmodellierer bei der Hinterlegung an einem Prozessschritt festgelegt werden. Im Beispiel der XML-Transformation (s. Beispiel 7.5) etwa könnte man für jeden Eingabeparameter nur eine Parametervorlage hinterlegen und bei dieser den Typ (`AdeptDataType`) unspezifiziert lassen. Dann kann der Prozessmodellierer aber einen beliebigen Datentyp verwenden, (u. a. `INTEGER`), was jedoch nicht der Schnittstelle des Anwendungsbausteins entspräche und dementsprechend nicht sinnvoll wäre.

Jede der bisher angeführten Entitäten (*ActivityTemplate*, *ATParameter*, *ATParameterTemplate*) kann Konfigurationswerte besitzen. Dementsprechend referenzieren sie *Configuration*. Aktivitätenvorlagen und Parameter besitzen sogar eine *ActivityConfiguration*, d. h. die Konfigurationswerte können über zusätzliche Eingabeparameter der entsprechenden Aktivität oder von der Konfiguration des entsprechenden Prozesses versorgt werden.

Beispiel 7.6 (Konfigurationen für Ein-/Ausgabeparameter)
Ein generischer Anwendungsbaustein zum Aufruf einer ausführbaren Datei besitzt beliebige Eingabeparameter. Für den Aufruf der ausführbaren Datei kann die Reihenfolge der Eingabeparameter relevant sein. Dies lässt sich durch Konfigurationen an den Eingabeparametern selbst lösen, etwa Konfigurationen der Art `Position=1`. Soll die Reihenfolge der Parameter durch vorherige Prozessschritte festgelegt werden, ergibt sich der Konfigurationswert von `Position` aus einem separaten Eingabeparameter.

Für zwei Eingabeparameter `In1` und `In2` des eigentlichen Aufrufs ergeben sich somit zwei weitere Eingabeparameter `In1Pos` und `In2Pos` vom Typ `INTEGER`. Die Konfiguration von `In1` besteht aus `Position=%In1Pos%`, die von `In2` aus `Position=%In2Pos%`. Zur Laufzeit werden die Konfigurationswerte entsprechend den Werten von `In1Pos` und `In2Pos` gesetzt.

7.2.2.2 Hierarchisches Aktivitätenmodell

Ein wichtiges Entwurfsziel des Aktivitätenmodells ist die Wiederverwendung bereits existierender Objekte des Datenmodells und deren Attribute, wobei Redundanz möglichst zu vermeiden ist. Hierfür sind die bisher vorgestellten Aktivitätenvorlagen nicht geeignet. Betrachten wir hierzu folgendes Beispiel:

Beispiel 7.7 (Generische Aktivitätenvorlage zur Rechnungserstellung)
Für die Erstellung einer Rechnung in einem Textverarbeitungsprogramm gibt es eine Aktivitätenvorlage. Die Inhalte der zu erstellenden Rechnung werden über entsprechende Eingabeparameter (generisch) zur Verfügung gestellt. Die Rechnung wird ohne Benutzerinteraktion automatisch erstellt. Aus Sicht des Textverarbeitungsprogramms ist die Rechnungserstellung nur ein Spezialfall einer Dokumenterstellung. So lassen sich neben Rechnungen beispielsweise auch Auftragsbestätigungen auf vergleichbare Art und Weise erstellen. Daneben erlaubt eine Textverarbeitung aber nicht nur die Erstellung einfacher Dokumente, sondern auch die Erstellung von Serienbriefen oder die Bearbeitung von Dokumenten. Neben einer Textverarbeitung existieren noch weitere Programme, die in Prozessen in ähnlicher Funktion verwendet werden. Hierzu zählt auch eine Tabellenkalkulation, die beispielsweise die Inhalte der in der Textverarbeitung erstellten Rechnung zur Verfügung stellt.

Besäße das Aktivitätenmodell nur Aktivitätenvorlagen, müsste man für jede Erstellung bestimmter Rechnungen, bestimmter Auftragsbestätigungen, bestimmter Dokumente, ... jeweils eine eigene Aktivitätenvorlage haben. Zusätzlich wird für die Bearbeitung der Dokumente jeweils eine eigene Aktivitätenvorlage benötigt, ebenso für sämtliche Aktivitäten, für die die Tabellenkalkulation herangezogen wird. Viele dieser Aktivitätenvorlagen besitzen dieselben Attributwerte, beispielsweise für die Bibliotheken (`ClassPath`) der Textverarbeitung oder der Tabellenkalkulation.

Die bestmögliche Wiederverwendbarkeit erreicht man, indem das Aktivitätenmodell die realen Gegebenheiten möglichst direkt widerspiegelt. Dem Beispiel folgend wird eine Aktivitätenvorlage im Aktivitätenmodell deshalb auf drei Hierarchieebenen aufgeteilt (links in Abb. 7.9). Wird eine Aktivitätenvorlage benötigt, werden diese drei Ebenen zu einem Objekt vom Typ *ActivityTemplate* aus Abb. 7.8 vereinigt. Dabei überschreiben die Attributwerte unterer Hierarchieebenen, die höherer Hierarchieebenen, d. h. untere Hierarchieebenen erben von übergeordneten. Dies bezieht sich sowohl auf Attributwerte der Objekte als auch die referenzierten Objekte wie Ein-/Ausgabeparameter, Parametervorlagen und Konfigurationen sowie deren Attribute.

ExecutableComponentDescription ist die oberste Ebene und repräsentiert z. B. Textverarbeitung und Tabellenkalkulation. *ExecutableComponentDescription* besitzt weitgehend dieselben Attribute wie *ActivityTemplate*. Gegenüber *ActivityTemplate* fehlen hier die Attribute für technische Details. Hierzu zählen Angaben zur testweisen Ausführung (`SupportsTestExecution`), zur Ausführung im Historienmodus (`SupportsViewOnly`) sowie eine Bearbeiterzuordnung. Ferner besitzt *ExecutableComponentDescription* keine Ein-/Ausgabeparameter und keine Konfiguration. Diese Attribute können auf dieser Ebene (z. B. Textverarbeitung und Tabellenkalkulation) nicht sinnvoll festgelegt werden. Es gibt allerdings Parametervorlagen, die Ein-/Ausgabeparameter beschreiben. Diese können auf der nächsten Hierarchiestufe herangezogen werden, um dort konkrete Ein-/Ausgabeparameter festzulegen. dasselbe gilt auch für die Konfiguration. So besitzt *ExecutableComponentDescription* eine *ConfigurationDescription*, also die Definition von Konfigurationswerten. Diese ist aus Gründen der Übersichtlichkeit in Abb. 7.9 nicht eingezeichnet, da jede Entität in diesem Diagramm außer *InheritedConfiguration* und *InheritedActivityConfiguration* eine *ConfigurationDescription* besitzt (s. Abb. 7.8).

Die Entität unterhalb von *ExecutableComponentDescription* ist *Operation*. *Operation* repräsentiert eine Funktion eines Anwendungsprogramms, etwa die Erstellung oder das Bearbeiten eines Dokuments in einer Textverarbeitung. *Operation* hat dieselben Attribute wie *ExecutableComponentDescription* und besitzt zusätzlich eine Bearbeiterzuordnung sowie Ein-/

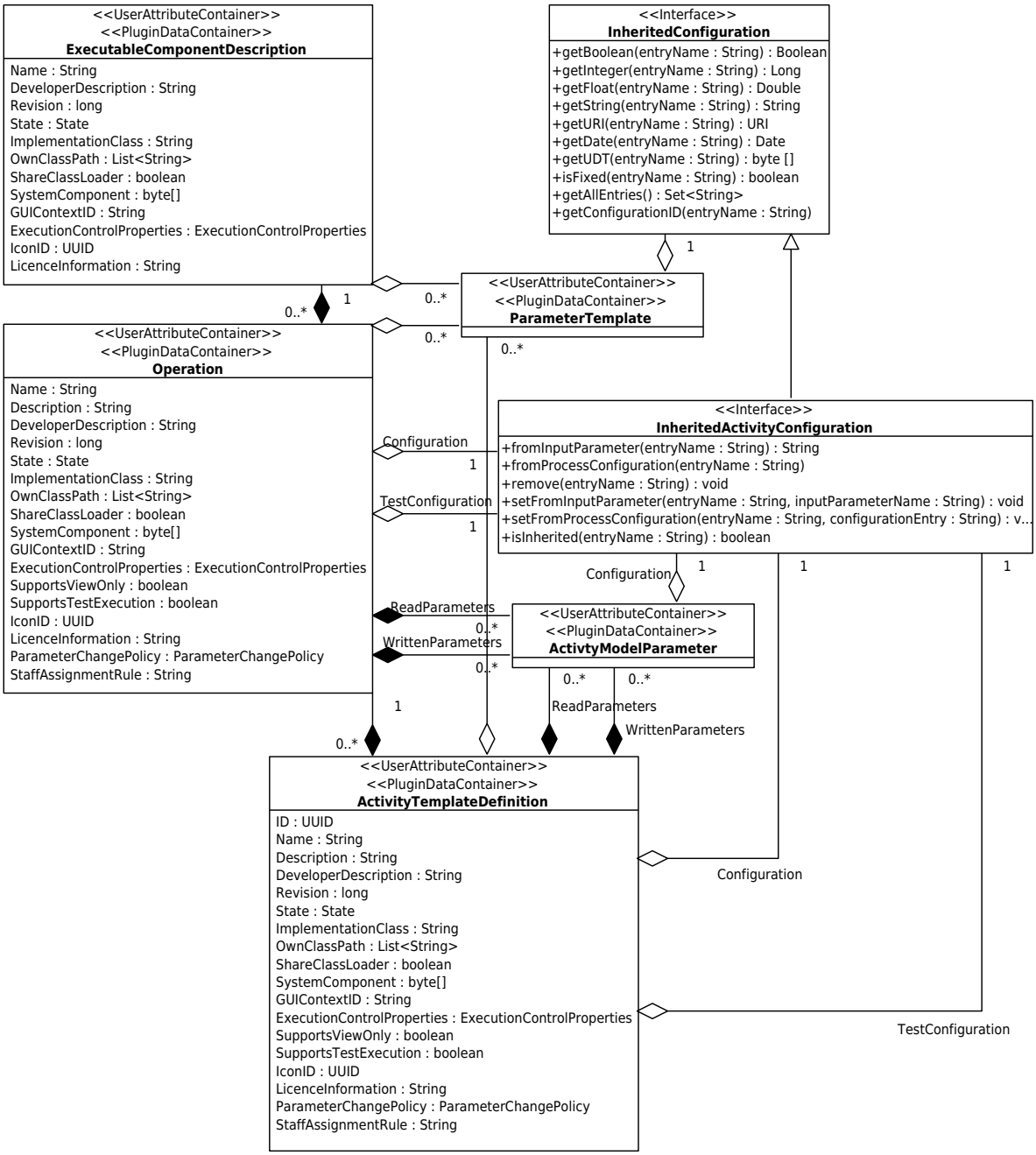


Abbildung 7.9: UML-Klassendiagramm von Modellierungsobjekte im Aktivitätenmodell (ohne *ConfigurationDescription*)

Ausgabeparameter (`ReadParameters` und `WrittenParameters`). Wie im vorherigen Abschnitt festgestellt (s. Abschnitt 7.2.2.1), ist es bei generischen Funktionen nicht sinnvoll, alle Ein-/Ausgabeparameter bereits im Aktivitätenmodell festzulegen. Es sind aber durchaus Einschränkungen möglich, etwa eine Wahlmöglichkeit zwischen zwei Datentypen (s. Beispiel 7.5) oder weitere Einschränkungen wie folgendes Beispiel zeigt:

Beispiel 7.8 (Parametervorlagen für Ein-/Ausgabeparameter von Operationen) Die Operation des Anwendungsbausteins für die XML-Transformation aus Beispiel 7.5 besitzt naheliegenderweise keine Ein-/Ausgabeparameter. Stattdessen hat sie sechs Parametervorlagen, zwei Vorlagen für das zu transformierende XML-Dokument, zwei Vorlagen für die Transformation sowie zwei Vorlagen für das transformierte XML-Dokument. Die Vorlagen unterscheiden sich jeweils im Datentyp.

Ein erweiterter Anwendungsbaustein für die XML-Transformation erlaubt die Transformation mehrerer XML-Dokumente auf einmal. Dazu nimmt er neben der Transformationsvorschrift beliebig viele XML-Dokumente entgegen und verarbeitet diese. Dementsprechend hat der Anwendungsbaustein auch für jedes transformierte XML-Dokument einen eigenen Ausgabeparameter. Die entsprechende Operation des erweiterten Anwendungsbausteins besitzt dieselben Parametervorlagen wie die Operation des einfachen Anwendungsbaustein zur XML-Transformation. Allerdings können bei der Operation des erweiterten Anwendungsbausteins die Parametervorlagen für die XML-Dokument für mehrere Ein-/Ausgabeparameter verwendet werden und nicht nur für einzelne Ein-/Ausgabeparameter.

Um die Möglichkeiten der Erweiterung einer *Operation* um Ein-/Ausgabeparameter widerzuspiegeln, gibt es das Attribut `ParameterChangePolicy`. Dieses gibt den Grad der Generizität einer *Operation* an und spezifiziert, wie Ein-/Ausgabeparameter von unteren Hierarchieebenen geändert werden können. Erlaubt sind Hinzufügen, Ändern und Löschen von Ein-/Ausgabeparametern sowie Hinzufügen virtueller Datenflüsse. Außerdem lässt `ParameterChangePolicy` auch sinnvolle Kombinationen hiervon zu, z. B. Hinzufügen und Ändern aber nicht Löschen. Für den einfachen Anwendungsbaustein aus Beispiel 7.5 etwa ist nur das Ändern zulässig, für den erweiterten Anwendungsbaustein sowohl Hinzufügen als auch Ändern.

Neben Ein-/Ausgabeparametern sowie den entsprechenden Zusatzinformationen für generische Anwendungsbausteine besitzt *Operation* zwei Konfigurationen. Eine Konfiguration dient der normalen Ausführung (*Configuration*), die andere (*TestConfiguration*) wird bei der testweisen Ausführung des entsprechenden Anwendungsbausteins verwendet. Beide Konfigurationen beziehen sich auf die vereinigte Konfigurationsdefinition der (nicht im Diagramm sichtbaren) *ConfigurationDescription* von *ExecutableComponentDescription* und *Operation*.

Auf der untersten Hierarchieebene befindet sich *ActivityTemplateDefinition*. Dies entspricht etwa der Rechnungserstellung als Spezialisierung von Dokumenterstellung. Eine *ActivityTemplateDefinition* besitzt gegenüber einer *Operation* eine zusätzliche Kennung (ID) und hat ggf. eine umfangreichere *Configuration* und *TestConfiguration*. Die Kennung (ID) steht für eine gesamte Vererbungshierarchie, also *ExecutableComponentDescription*, *Operation* und *ActivityTemplateDefinition*, und davon genau einen Stand (`Revision`) eindeutig. Die Kennung ändert sich sobald eine Entität in der Vererbungshierarchie eine neue Revision erhält. Dadurch können die für einen Prozess verwendeten Aktivitätenvorlagen zusammen mit ihrem Stand und ihrer Herkunft (bzgl. *Operation* und *ExecutableComponentDescription*) eindeutig bestimmt werden.

All diese Entitäten bieten sowohl lesenden als auch schreibenden Zugriff auf ihre Attribute. Dabei wird gewährleistet, dass Änderungen durch schreibende Zugriffe zu neuen Revisionen

der betroffenen Objekte führen. Außerdem bieten alle Klassen bei geerbten Attributen sowohl Zugriff auf den eigenen, überschreibenden Wert als auch auf den geerbten Attributwert. Dies gilt nicht für Attribute einfachen Typs sondern auch für Attribute komplexen Typs, etwa Parametermengen und Konfigurationen. Aus diesem Grund ergänzen *InheritedConfiguration* und *InheritedActivityConfiguration* die jeweiligen Oberklassen *Configuration* und *ActivityConfiguration* (s. Abb. 7.5) um Methoden für den Zugriff auf die überschreibenden Konfigurationswerte sowie zur Feststellung, ob ein bestimmter Konfigurationswert überschrieben (oder vererbt) wird. Des weiteren können über die Schnittstelle Konfigurationswerte in der Vererbungshierarchie gelöscht werden, was dem Überschreiben durch den Wert `null` entspricht. Aus Gründen der Übersichtlichkeit sind diese zusätzlichen Methoden nicht im Klassendiagramm eingezeichnet.

7.2.2.3 Wiederverwendung von Konfigurationen im Aktivitätenmodell

Das bisher vorgestellte Aktivitätenmodell ist noch nicht ausreichend. Dazu erweitern wir Beispiel 7.7 zur Rechnungserstellung:

Beispiel 7.9 (Verschiedene Dokumentenvorlagen und Rechnungsarten)
In verschiedenen Abteilungen werden Rechnungen mit einem Textverarbeitungsprogramm erstellt. Dabei verwendet jede Abteilung eine andere Dokumentenvorlage für die Rechnung. Im bisherigen Aktivitätenmodell wird dies über eine *ActivityTemplateDefinition* für jede Dokumentenvorlage abgebildet. Zusätzlich wird in Abteilungen jedoch zwischen verschiedenen Rechnungsarten unterschieden, etwa Rechnungen für Privat- und Geschäftskunden. Jede Rechnungsart erfordert eine separate *ActivityTemplateDefinition*, so dass sich die Gesamtzahl an notwendigen *ActivityTemplateDefinition* als Produkt aus der Anzahl der Dokumentenvorlagen und der Anzahl der Rechnungsarten ergibt.

Die große Zahl an Instanzen von *ActivityTemplateDefinition* entsteht dadurch, dass Dokumentenvorlagen und Rechnungsarten eigentlich voneinander unabhängige Aspekte sind, die aber im Aktivitätenmodell nicht unabhängig voneinander repräsentiert werden. Dies führt zu viel Redundanz und damit zu Problemen bei Änderungen. Aus diesem Grund erweitern wir das Aktivitätenmodell um die Entität *ConfigurationSet* (s. Abb. 7.10).

Ein *ConfigurationSet* ist eine Sammlung von Konfigurationen, Testkonfigurationen, Parametervorlagen und Bearbeiterzuordnungen. Ein *ConfigurationSet* ist beliebigen *Operation*s und damit auch den entsprechenden *ExecutableComponentDescription*s zugeordnet. Die Konfigurationen von *ConfigurationSet* beziehen sich auf die *ConfigurationDescription*s der entsprechenden *Operation*s und *ExecutableComponentDescription*s. *ConfigurationSet* liegt aber außerhalb der von *ExecutableComponentDescription* und *Operation* gebildeten Hierarchie und ist damit unabhängig.

Damit die von einem *ConfigurationSet* definierten Attribute gelten, müssen sie innerhalb der Hierarchie zum Überschreiben herangezogen werden. Dies erfolgt dadurch, dass sich *ActivityTemplateDefinition*s auf ein *ConfigurationSet* beziehen. In diesem Fall wird bei der Vererbung das *ConfigurationSet* zwischen der *Operation* und der *ActivityTemplateDefinition* berücksichtigt, so dass alle Werte der *Operation* von denen des *ConfigurationSet* überschrieben werden. Die Beziehung zwischen *ActivityTemplateDefinition* und *ConfigurationSet* ist optional; bezieht sich eine *ActivityTemplateDefinition* nicht auf ein *ConfigurationSet*, gilt die bisherige Vererbungshierarchie *ExecutableComponentDescription*, *Operation* und *ActivityTemplateDefinition*. Damit die Gültigkeit der Konfigurationen eines *ConfigurationSet* gewährleistet ist, kann es nur

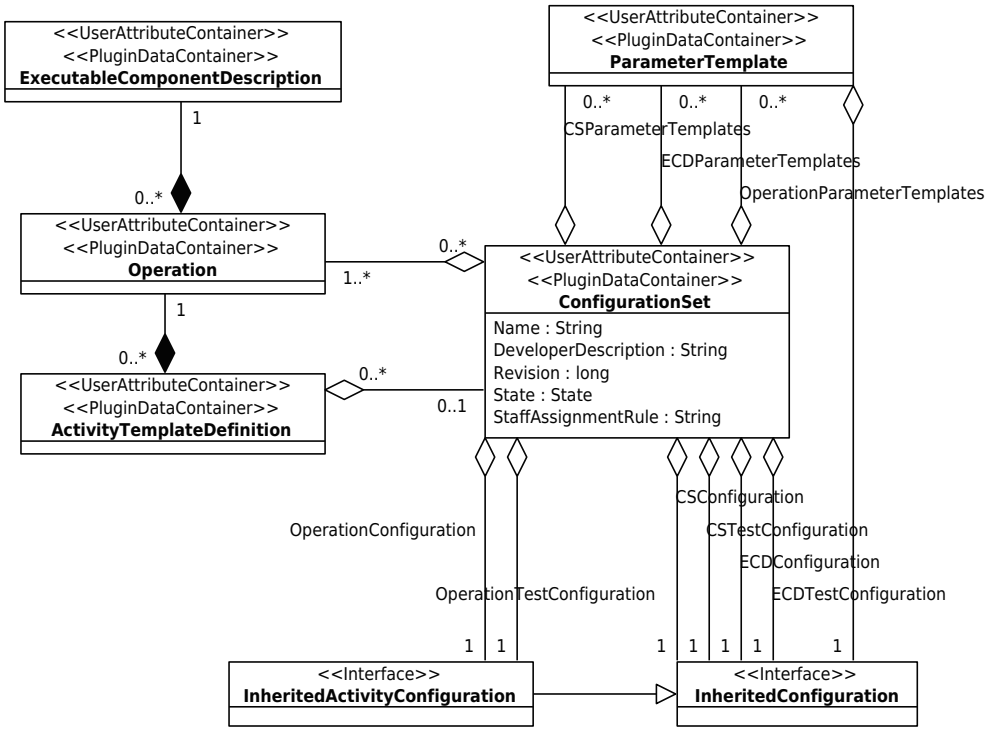


Abbildung 7.10: UML-Klassendiagramm von *ConfigurationSet* im Aktivitätenmodell

dann von einer *ActivityTemplateDefinition*s referenziert werden, wenn das *ConfigurationSet* den entsprechenden *Operation*s zugeordnet ist (s. Abb. 7.11).

Beispiel 7.10 (Verwendung von *ConfigurationSet* im Aktivitätenmodell)
Abb. 7.11 zeigt die Verwendung eines *ConfigurationSet* innerhalb der Hierarchie im Aktivitätenmodell (*ExecutableComponentDescription*, *Operation* und *ActivityTemplateDefinition*). Dabei bezieht sich `ConfigurationSet 1` auf die (*ConfigurationDescription*s der) Operationen `Op1.2` und `Op2.1`. Indirekt bezieht es sich damit auch auf (die *ConfigurationDescription*s von) `ECD1` und `ECD2`. Es darf aber nur von den *ActivityTemplateDefinition*s verwendet werden, die von `Op1.2` und `Op2.1` erben. In Abb. 7.11 wird `ConfigurationSet 1` von `ATD1.2.1`, `ATD2.1.1` und `ATD2.1.2` referenziert. Damit wird bei der Vererbung von deren Konfigurationen nach `Op1.2` und `Op2.1` noch `ConfigurationSet 1` berücksichtigt. Für alle anderen *ActivityTemplateDefinition*s hat `ConfigurationSet 1` keine Bedeutung.

Mit Hilfe von *ConfigurationSet* lässt sich das Beispiel der Rechnungserstellung ohne Redundanz realisieren. Dazu gibt es für jede Abteilung ein *ConfigurationSet*, das die verwendenden Dokumentenvorlagen definiert. Diese werden dann auf die *ActivityTemplateDefinition*s der entsprechenden Rechnungsarten angewendet.

Die vorgestellten vier Hauptentitäten *ExecutableComponentDescription*, *Operation*, *ConfigurationSet* und *ActivityTemplateDefinition* dienen in erster Linie der einfachen und redundanzfreien Verwaltung von Daten. Vor Verwendung in einem Prozessmodell werden diese vier Entitäten zu einer Entität vereinigt: dem *ActivityTemplate* (s. Abb. 7.8). Bei ihrer Vereinigung findet das Überschreiben aller Attributwerte statt, so dass *ActivityTemplate* eine konsolidierte Sicht

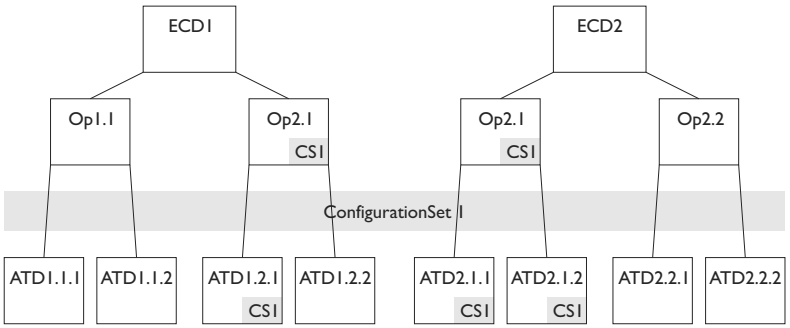


Abbildung 7.11: Hierarchie des Aktivitätenmodells mit *ConfigurationSet*

bietet. Dieses wird durch *Plug and Play* (s. Abschnitt 2.4.1) an einem Prozessschritt hinterlegt, d. h. entsprechend in den Prozesskontext eingebettet, und durch diese Einbettung zu einer Aktivität (*Activity*). Bei der Einbettung werden unter anderem der Datenfluss etabliert und ggf. noch nicht vorhandene, aber notwendige Konfigurationswerte spezifiziert.

7.2.3 Prozessmodell

Das Prozessmodell stellt alle prozessrelevanten Daten zur Verfügung. Es umfasst die Prozessstruktur einer Prozessvorlage in Form von Knoten, Kontrollkanten, Datenelementen, Datenkanten und Parametern. Des weiteren sind ihm die Prozessinstanzen und deren Zustandsinformationen eines laufenden Prozesses (z. B. Knotenzustände) zuzuordnen. Damit realisiert das Prozessmodell das ADEPT-Metamodell [Rei00, RD98]. Dementsprechend betrifft es große Teile der Gesamtarchitektur (s. Abb. 7.12): Es wird zur Modellierung (*ProcessEditor*, *ChangeOperations*, *ProcessRepository*), Ausführung (*ProcessManager*, *DataManager*, *ExecutionManager*, Teile von *WorklistManager*, *ChangeOperations*) und Überwachung (*Monitor*) von Prozessvorlagen und -instanzen verwendet.

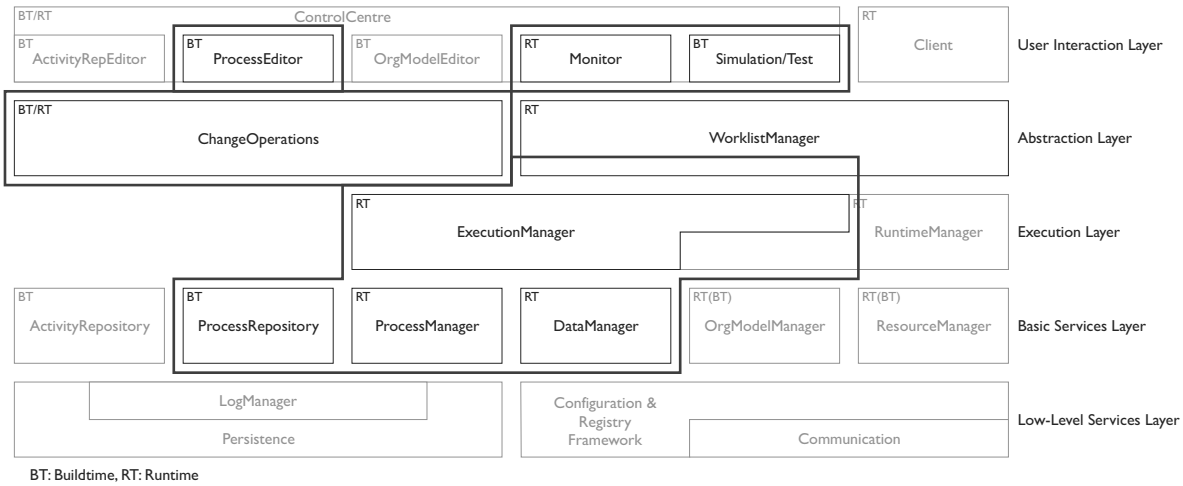


Abbildung 7.12: Verwendung des Prozessmodells in der Gesamtarchitektur

7.2.3.1 Prozessvorlagen und -instanzen

Zentrale Klassen des Prozessmodells sind Prozessvorlagen (*Template*) und -instanzen (*Instance*) (s. Abb. 7.13). Ein *Template* besitzt eine eindeutige Kennung (`ID`), einen Namen (`Name`) und eine Beschreibung (`Description`). Es ist einem Prozesstyp (`ProcessType`) zugeordnet, und es hat einen Verantwortlichen (`SupervisorAgent`). Zusätzlich kann ein *Template* eine Vorlage für den Namen von Prozessinstanzen (`InstanceNameTemplate`) vorgeben. Weiterhin definiert *Template* zahlreiche Methoden für den Zugriff auf die Prozessstruktur. Zur Prozessstruktur gehören unter anderem die Knotentypen, die topologische Ordnung der Knoten, strukturelle Informationen zu einem Knoten, etwa dazu, auf welchem Zweig einer Verzweigung ein Knoten liegt, sowie Vorgänger-/Nachfolgerbeziehungen. Diese Daten werden nicht als Attribute realisiert. Die topologische Ordnung könnte beispielsweise als Ganzzahl in *Node* realisiert werden, allerdings vereinfacht die zentrale Bereitstellung der Prozessstruktur in *Template* gegenüber der Verteilung auf mehrerer *Node*-Objekte die Realisierung von Änderungsoperationen mit Hilfe struktureller Instanzdeltas (s. Abschnitt 5.1.3) erheblich.

Prozessinstanzen besitzen zwei Kennungen, wobei sich die `LogID` einer Prozessinstanz nicht ändert, während `ID` bei allen instanzspezifischen Änderungen modifiziert wird. Dadurch werden vorhandene Referenzen auf eine Prozessinstanz bei jeder instanzspezifischen Änderung ungültig:

Beispiel 7.11 (Änderung der Kennung (ID) einer geänderten Prozessinstanz)
Bei einer laufenden Prozessinstanz werde bei einem aktivierten Prozessschritt die Bearbeiterzuordnung geändert, etwa weil die aktuell zugeordneten Bearbeiter ausgelastet sind. Direkt nach der Änderung will einer der bisher vorgesehenen Bearbeiter den Prozessschritt starten. Da seine Arbeitsliste nicht automatisch aktualisiert wird, besitzt sie noch einen entsprechenden Eintrag. Die Überprüfung der Bearbeiterzuordnung direkt beim Start eines Prozessschritts verhindert zwar die Ausführung, das eigentliche Problem aber ist der veraltete Eintrag. Um diesen erkennen zu können, bekommt die Prozessinstanz bei der Änderung eine neue Kennung, so dass der Arbeitslisteneintrag ungültig wird. Versucht der Bearbeiter den Prozessschritt zu starten, wird die Bearbeiterzuordnung gar nicht erst überprüft, da der referenzierte Prozessschritt/die referenzierte Prozessinstanz nicht mehr existiert.

Durch die Änderung der Kennung werden mit einfachen Mitteln alle Referenzen auf die Prozessinstanz oder ihre Teile (z. B. Knoten) ungültig. Somit wird gewährleistet, dass keine ungültigen Zugriffe erfolgen. Dies müsste sonst durch aufwendige Mechanismen gewährleistet werden. Neben den beiden Kennungen hat jede Prozessinstanz einen Namen, der ggf. auf `InstanceNameTemplate` von *Template* beruht, einen Prozessverantwortlichen (`SupervisorAgent`, im Unterschied zum Prozessvorlagenverantwortlichen), Angaben zur Person, die die Prozessinstanz gestartet hat (`InitiatorAgent`), und die Startzeit (`CreationTime`).

Auch bei Prozessinstanzen werden die Informationen, die von einer Änderung betroffen sein können (z. B. Knotenzustände und Schleifenzähler), nicht bei den Prozessinstanzknoten hinterlegt, sondern zentral bei *Instance*. Somit gibt es keine abhängigen Objekte, wie etwa *NodeInstance*; alle Zustandsinformationen werden direkt von *Instance* bereitgestellt. Eine *Instance* hat ggf. lediglich eine *EBPInstanceReference*. Eine *EBPInstanceReference* ist eine einfache Entität, die an vielen Stellen verwendet wird. Sie identifiziert einen Knoten eindeutig und besitzt entsprechende Attribute. Dazu gehören die Kennung (ID) des Knotens, seine Prozessinstanz sowie Referenzen (vom Typ `URI[]`) zu Diensten, die an der Ausführung des Knotens beteiligt sind (*ExecutionManager* und *RuntimeManager*). Zusätzlich besitzt *EBPInstanceReference* einen Typ (`Type`) mittels dessen eine normale Aktivität von einer

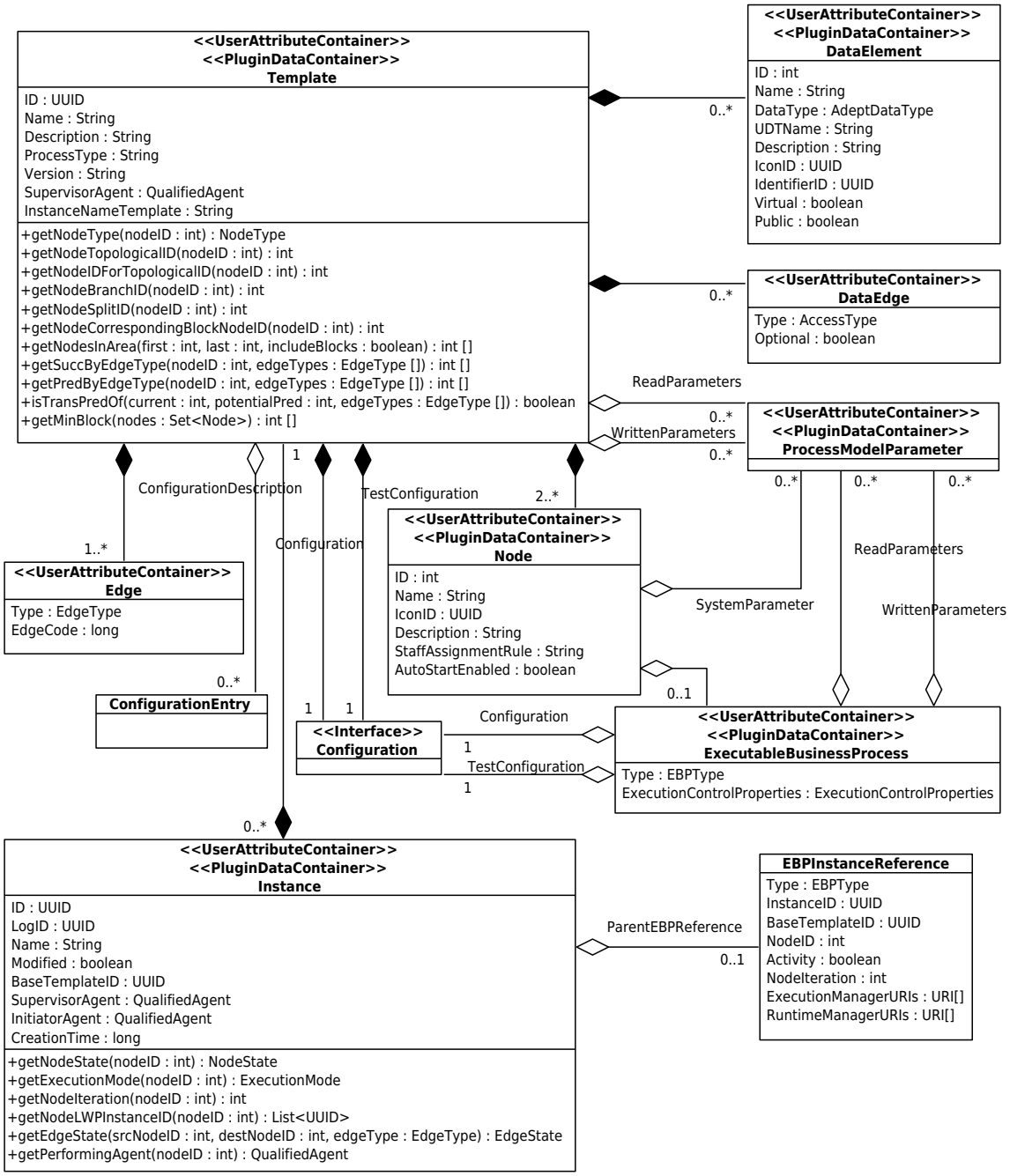


Abbildung 7.13: UML-Klassendiagramm von Prozessvorlagen (abstrakt) des Prozessmodells

Entscheidungsaktivität und einem Subprozess unterschieden wird. Besitzt eine Prozessinstanz eine `ParentEBPReference` (vom Typ *EBPInstanceReference*), handelt es sich um eine Subprozessinstanz, die mittels der `ParentEBPReference` den entsprechenden Knoten der Elterninstanz referenziert. Prozessinstanzen der obersten Ebene haben keine *EBPInstanceReference*.

Im Gegensatz zu *Template* besitzt *Instance* keine Konfiguration (und dementsprechend auch keine Konfigurationsdefinition). Diese wäre nur für instanzspezifische Konfigurationswerte sinnvoll, die beim Starten der Prozessinstanz übergeben oder während der Ausführung geschrieben werden. Die Übergabe von Werten beim Start einer erfolgt aber über Prozesseingabeparameter, für während der Ausführung geschriebene Werte gibt es Datenelemente. Somit ist es im Sinne einer einfachen Schnittstelle besser, instanzspezifische Daten ausschließlich über die bereits existierenden Mechanismen für den Datenfluss bereitzustellen. Auf Ebene von Aktivitäten können Konfigurationswerte ohnehin über Eingabeparameter und damit über den Datenfluss versorgt werden (s. *ActivityConfiguration* in Abb. 7.5 und Abb. 7.6). Damit lassen sich instanzspezifische Konfigurationswerte realisieren.

7.2.3.2 Knoten und Parameter

Da strukturelle Informationen von *Template* strukturelle Informationen bereitgestellt werden, besitzen Knoten im Prozessmodell (*Node*) nur wenige Attribute (s. Abb. 7.13). Hierzu gehören unter anderem eine Kennung (`ID`), ein Name und eine Bearbeiterzuordnung (`StaffAssignmentRule`). Außerdem kann bei Knoten ebenso wie bei Datenelementen (*DataElement*) eine `IconID` gesetzt sein. Diese referenziert ein in einem globalen Ressourcenverzeichnis hinterlegtes Symbol, das für die graphische Anzeige des Knotens bzw. Datenelements verwendet werden soll. Dieses Symbol ersetzt dann die Standarddarstellung von Knoten bzw. Datenelementen in Graphdarstellungen, etwa im *ProcessEditor* und *Monitor*. Beispielsweise lassen sich so Datenelemente durch Abbildungen ihres Inhalts repräsentieren.

Jedes *Template* besitzt, wie im Metamodell definiert, mindestens zwei Knoten: den Start- und Endknoten. Diese beiden Knoten sind, neben den Blockstartknoten (Verzweigungsknoten) paralleler Verzweigungen und Schleifen sowie den Blockendknoten (Vereinigungsknoten) paralleler und alternativer Verzweigungen, die einzigen Knoten, die keinen *ExecutableBusinessProcess* haben. *ExecutableBusinessProcess* ist eine Abstraktion von Aktivitäten und Subprozessen (s. Abb. 7.14). Knoten ohne einen *ExecutableBusinessProcess* werden auch als *Nullknoten* bezeichnet, da sie hauptsächlich strukturellen Zwecken dienen.

Genauso wie Aktivitäten können auch Subprozesse Ein-/Ausgabeparameter besitzen (s. Abb. 7.13); dementsprechend können auch Prozessvorlagen Ein-/Ausgabeparameter haben. Wie in Abschnitt 2.2.2.3 beschrieben sind Eingabeparameter von Prozessvorlagen alle Datenelemente, die der Startknoten schreibt, Ausgabeparameter sind dementsprechend alle vom Endknoten gelesenen Datenelemente. Auf diese Weise stehen die Werte der Eingabeparameter über den normalen Datenfluss allen Prozessschritten der Prozessvorlage zur Verfügung, während die Ausgabeparameter entsprechend geschrieben werden können.

ProcessModelParameter entsprechen Parametern aus dem Aktivitätenmodell (*ActivityModelParameter*), allerdings mit dem Unterschied, dass die Attribute im Prozessmodell nicht geändert werden können. Neben den normalen Ein-/Ausgabeparametern von Aktivitäten und Subprozessen gibt es spezielle Eingabeparameter von Knoten (`SystemParameter`). Diese Parameter dienen nicht der Versorgung des *ExecutableBusinessProcess*, sondern von Knotenattributen, beispielsweise um Abhängigkeiten von Bearbeiterzuordnungen aufzulösen. Dies

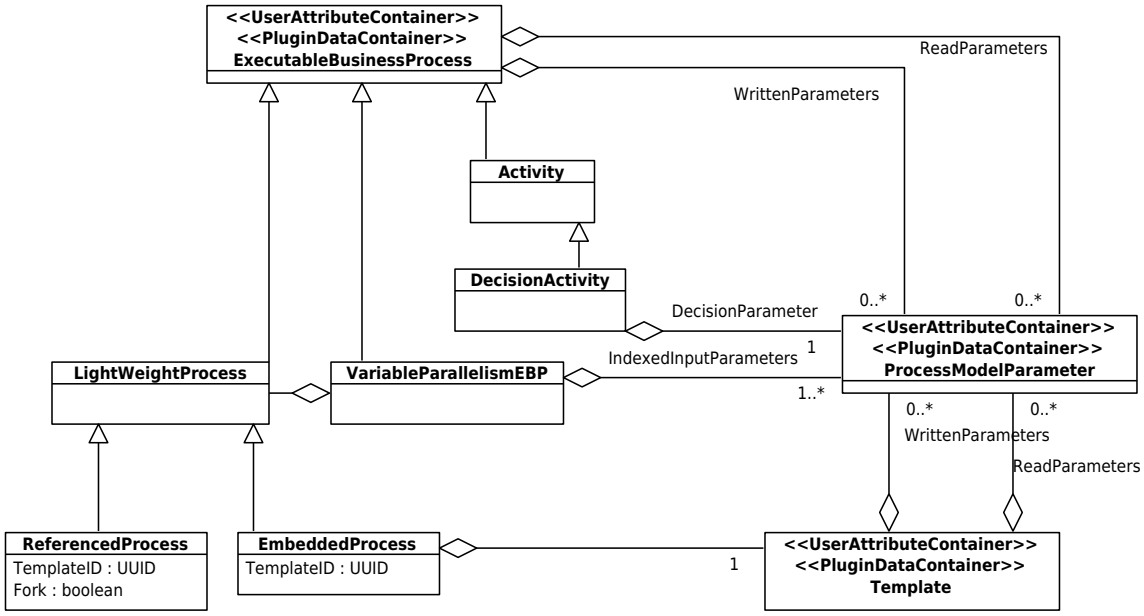


Abbildung 7.14: UML-Klassendiagramm von *ExecutableBusinessProcess* des Prozessmodells

geschieht bei der Ausführung einer Prozessinstanz und wird im folgenden Abschnitt zum Ausführungsmodell (s. Abschnitt 7.2.4) genauer vorgestellt.

7.2.3.3 Aktivitäten

Abb. 7.14 zeigt die Analogie zwischen Subprozess und Aktivität auf Modellebene. *ExecutableBusinessProcess* vereinigt die Gemeinsamkeiten. Dazu gehören Ein-/Ausgabeparameter (`ReadParameters`, `WrittenParameters`) und die Kontrollsignale (z. B. Pausieren, Beenden, ...), die der *ExecutableBusinessProcess* interpretiert, wenn sie während der Ausführung an ihn gesendet werden (`ExecutionControlProperties`). Ausprägungen von *ExecutableBusinessProcess* sind normale Aktivitäten (*Activity*) und Entscheidungsaktivitäten (*DecisionActivity*). Entscheidungsaktivitäten werden für alternative Verzweigungen und Schleifenendknoten benötigt. Sie besitzen zusätzlich zu den normalen Ein-/Ausgabeparameter einen speziellen Ausgabeparameter (`DecisionParameter`), der das Ergebnis der Entscheidung angibt und somit den weiteren Weg durch den Prozess bestimmt.

Auch Subprozesse (*LightWeightProcess*) sind Ausprägungen von *ExecutableBusinessProcess*. Diese können in der entsprechenden Elternprozessvorlage referenziert (*ReferencedProcess*) und eingebettet (*EmbeddedProcess*) sein. Das heißt, sie werden entweder als eigenständige Prozessvorlagen verwaltet und können dementsprechend auch – sofern gewünscht – außerhalb der Elternprozessvorlage verwendet werden, oder sie werden physisch in die Elternprozessvorlage integriert. Dies zeigt sich auch an den Attributen: *ReferencedProcess* besitzt die Kennung (ID) der referenzierten Prozessvorlage, während zu *EmbeddedProcess* ein komplettes *Template* gehört. Bei referenzierten Subprozessen kann die Ausführung der entsprechenden Prozessinstanzen abgespalten werden (`Fork`). Dadurch laufen solche Prozessinstanzen eigenständig und entkoppelt (asynchron) vom Elternprozess ab, so als ob sie separat gestartet worden wären. Dementsprechend wird der entsprechende Knoten im Elternprozess auch sofort nach dem Start der Kindprozessinstanz beendet und nicht erst, wenn die Kindprozessinstanz abgeschlossen

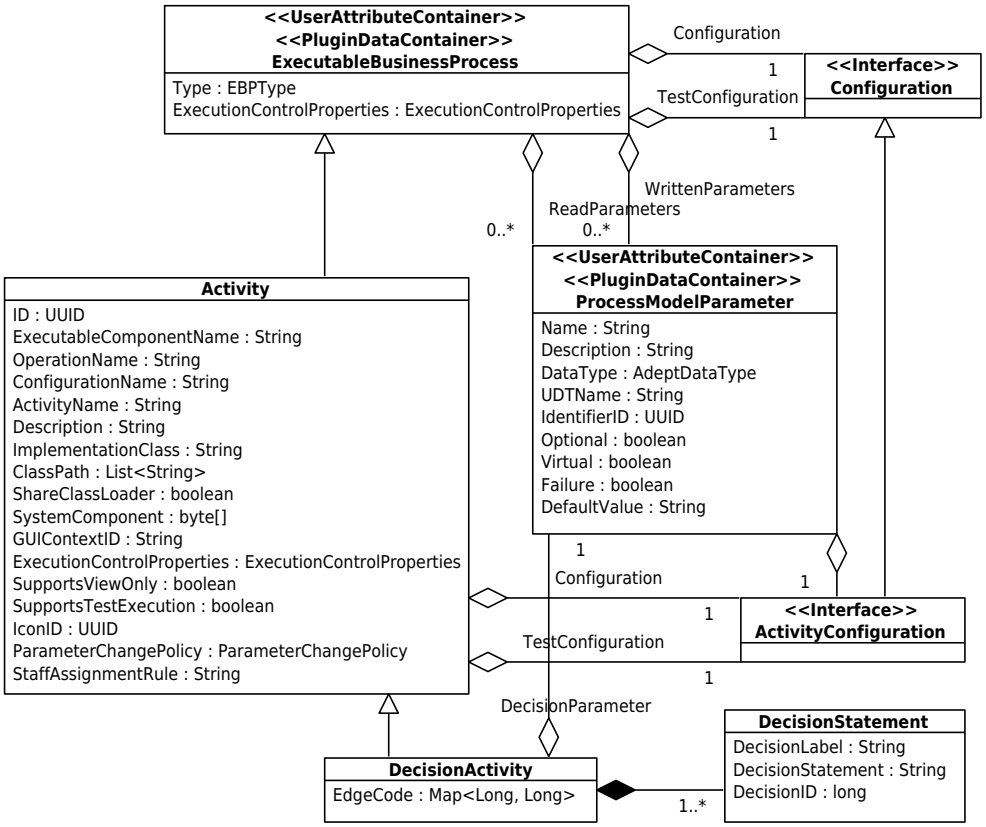


Abbildung 7.15: UML-Klassendiagramm von Aktivitäten des Prozessmodells

ist. Naheliegenderweise können die Ausgabeparameter solcher Kindprozessinstanzen nicht im Elternprozess verwendet werden, da sie erst verzögert und asynchron zur Verfügung stehen.

Einen Sonderfall des *ExecutableBusinessProcess* stellt die variable Parallelität dar (*VariableParallelismEBP*, s. Abschnitt 2.2.3.5). *VariableParallelismEBP* kapselt einen normalen Subprozess (*LightWeightProcess*), der mehrfach gestartet wird, abhängig von der Anzahl der Listenelemente des entsprechenden Eingabeparameters/der entsprechenden Eingabeparameter [vdAtHKB03, RR06]. Die aufgeteilten Eingabeparameter bilden die Menge `IndexedInputParameters`, während die normalen Eingabeparameter (`ReadParameters`) nicht aufgeteilt sondern allen Kindprozessinstanzen gleichermaßen übergeben werden. Die Ausgabeparameter (`WrittenParameters`) unterscheiden sich untereinander nicht, sie sind im Fall von *VariableParallelismEBP* jedoch alle listenwertig.

Abb. 7.15 zeigt *ExecutableBusinessProcess* und *Activity* im Detail. *Activity* entspricht weitgehend der Klasse *ActivityTemplate* des Aktivitätenmodells. Sie besitzt dieselben Attribute außer den nur für die Modellierung relevanten Attributen `DeveloperDescription` und Status im *ActivityRepository* (`State`). Außerdem verwendet/referenziert *Activity* einige Objekte, z. B. die Parameter und die Konfiguration.

Die Entscheidungsaktivität (*DecisionActivity*) erweitert *Activity* um die für Verzweigungen relevanten Informationen. Dazu gehört der bereits erwähnte spezielle Ausgabeparameter (`DecisionParameter`). Zur Laufzeit bestimmt dessen Wert, welcher Pfad der alternativen Verzweigung oder des Schleifenendknotens gewählt wird. Dazu besitzt *DecisionActivity* eine

Zuordnung von Parameterwerten (`DecisionID` vom Typ *long*) zu Kennzahlen der Pfade. Diese Kennzahlen sind an den von der Verzweigung bzw. dem Schleifenendknoten ausgehenden Kanten hinterlegt (`EdgeCode` bei *Edge* in Abb. 7.13). Diese Zuordnung wird durch das Attribut `EdgeCode` in *DecisionActivity* realisiert. Da Kennzahlen für einen menschlichen Benutzer wenig aussagekräftig sind, gibt es zusätzlich die Klasse *DecisionStatement*, die eine Kennzahl (`DecisionID`) mit einer textuellen Beschreibung (`DecisionStatement`) und einem Kurztext für die Anzeige (`DecisionLabel`) verbindet.

7.2.3.4 Veränderte Prozessinstanzen

Wie in Abschnitt 5.1 festgestellt, ist es sinnvoll, dass für die Repräsentation von Prozessänderungen dieselben internen Datenstrukturen verwendet werden wie für unveränderte Prozesse. Das heißt, die durch das Prozessmodell realisierten Datenstrukturen werden sinnvollerweise auch für Prozessänderungen verwendet. Abb. 7.16 zeigt die dafür notwendigen Erweiterungen des Prozessmodells.

Bei Änderungen unterscheidet man, wie bereits in Abschnitt 2.3.2 erläutert, zwischen Änderungen an Prozessvorlagen ($V' = V + \Delta F$; V ist die ursprüngliche Prozessvorlage, ΔF sind die durchgeführten Änderungen und V' ist die resultierende Prozessvorlage) und Änderungen an Prozessinstanzen ($V^* = V + \Delta G$; V ist die ursprüngliche Prozessvorlage der Prozessinstanz, ΔG sind die durchgeführten instanzspezifischen Änderungen und V^* ist die resultierende instanzspezifische Prozessvorlage). In den meisten Fällen sind die aus den Änderungen resultierenden Prozessstrukturen von Interesse, also V' und V^*. Dies gilt beispielsweise für das Weiterschalten einer geänderten Prozessinstanz oder die Anzeige der Prozessinstanz inklusive der instanzspezifischen Änderungen für den Benutzer. Es gibt jedoch auch Funktionen, die die Änderungen selbst benötigen, also ΔF und ΔG. Hierzu gehört beispielsweise die Durchführung einer Schemaevolution bei geänderten Prozessinstanzen oder Prozessverantwortliche, die vorgenommene Änderungen analysieren wollen. Dies spiegelt sich im Prozessmodell entsprechend wider (s. Abb. 7.16):

Änderungen an Prozessvorlagen (*Template*) werden in *TemplateDeltaLayer* verwaltet (ΔF), Änderungen einzelner Prozessinstanzen in *InstanceDeltaLayer* (ΔG). Die Gemeinsamkeiten beider Schnittstellen sind in *DeltaLayer* zusammengefasst. Die entsprechenden Methoden bieten Zugriff auf die Datenstrukturen der Änderungen (Knoten, Kanten und Vorgänger-/Nachfolgerbeziehungen). Dabei kennt ein *DeltaLayer* nur die geänderten Daten, beispielsweise hinzugefügte, verschobene oder gelöschte Knoten, aber keine unveränderten Knoten. *TemplateDeltaLayer* kennt zusätzlich die ursprünglichen Kennungen (IDs) geänderter Knoten. Dies ist notwendig, da bei Änderungen an Prozessvorlagen die Knoten-IDs neu vergeben werden. Geänderte Prozessvorlagen sind in gewisser Hinsicht „neue“ Prozessvorlagen und erhalten damit auch neue Knoten-IDs. Für die Schemaevolution wird aber noch die Zuordnung zu den ursprünglichen Knoten und damit die ursprüngliche Kennung benötigt. Sie wird deshalb vom *TemplateDeltaLayer* bereitgestellt. Bei instanzspezifischen Änderungen gibt es keine derart umfassenden Eingriffe, weshalb *InstanceDeltaLayer* keine zusätzlichen Methoden gegenüber *DeltaLayer* besitzt.

Die gekapselte Sicht auf die geänderten Objekte, also die resultierenden (instanzspezifischen) Prozessvorlagen bieten *ChangeableTemplate* (V') und *ChangeableInstance* (V^*). Ihre Schnittstelle entspricht derjenigen normaler Objekte (*Template* und *Instance*) für unveränderte Prozessvorlagen und -instanzen. Bei Zugriffen über diese Schnittstellen werden die durchgeführten Änderungen implizit berücksichtigt. Daneben implementieren *ChangeableTemplate*

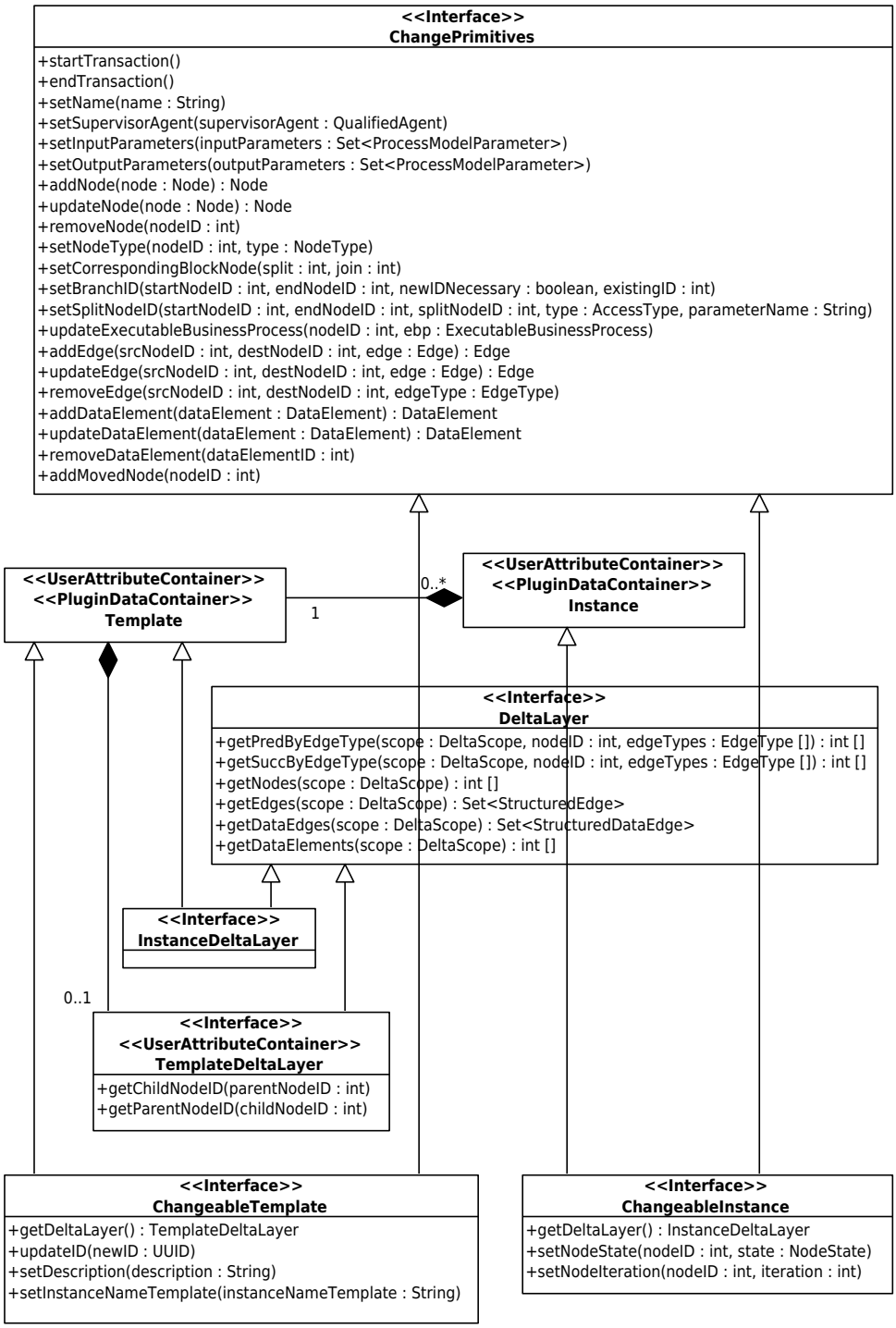


Abbildung 7.16: UML-Klassendiagramm der Klassen des Prozessmodells, die Änderungen an Prozessvorlagen und -instanzen ermöglichen

und *ChangeableInstance* die Änderungsprimitive (*ChangePrimitives*) des Metamodells (s. Abschnitt 2.3.1). Diese legen die entsprechenden Deltas (*TemplateDeltaLayer* und *InstanceDeltaLayer*) in *ChangeableTemplate* und *ChangeableInstance* an und manipulieren diese. Neben den gemeinsamen Änderungsprimitiven besitzen *ChangeableTemplate* und *ChangeableInstance* noch eigene Änderungsprimitive, die nur für Prozessvorlagen bzw. -instanzen relevant sind, sowie Zugriff auf die jeweiligen Deltas.

7.2.4 Ausführungsmodell (*Execution Model*)

Das Ausführungsmodell ist dem Prozessmodell ähnlich. Einerseits reduziert es das Prozessmodell um Modellierungsaspekte, da diese zur Ausführung nicht benötigt werden. Andererseits adressiert es zusätzlich Aspekte, die für die Ausführung von Prozessen und Prozessschritten relevant sind. Hierzu gehört der schreibende Zugriff auf Attribute von Prozessinstanzen und Prozessschritten sowie lesende und schreibende Zugriffe auf Datenelementwerte. Das Ausführungsmodell betrifft dementsprechend ausführende Dienste wie *ExecutionManager*, *ProcessManager*, *DataManager* und *RuntimeManager* und damit auch *Client* sowie *Monitor*. Auch Teile des *WorklistManager* arbeiten mit dem Ausführungsmodell und die Simulation von Prozessvorlagen (*Simulation/Test*) (s. Abb. 7.17).

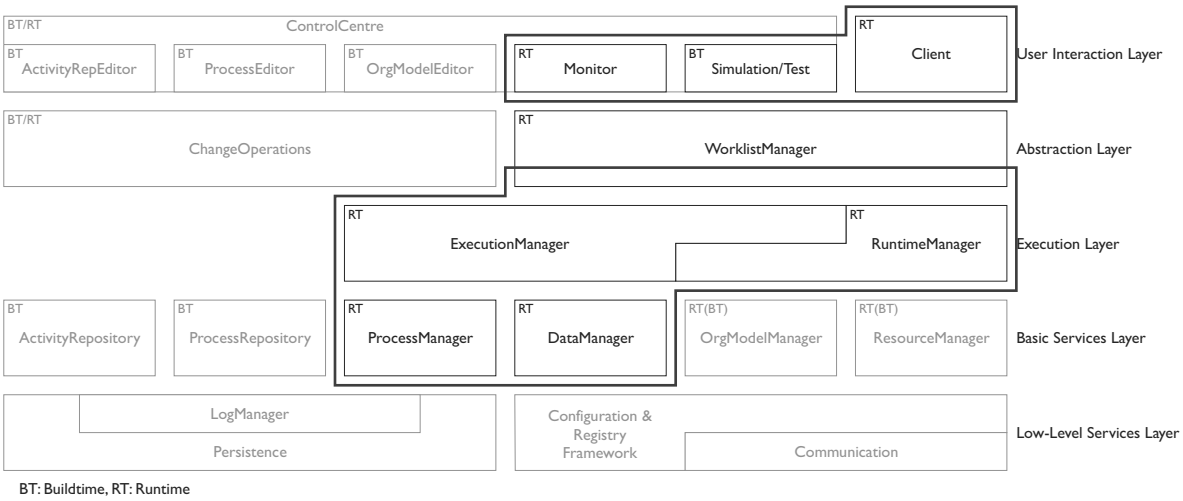


Abbildung 7.17: Verwendung des Ausführungsmodells in der Gesamtarchitektur

7.2.4.1 Prozessinstanzen und Aktivitäten

Wichtigste Entität des Ausführungsmodells ist *ExecutableInstance* (s. Abb. 7.18). Diese Klasse erweitert *Instance* um schreibende Zugriffe auf Zustandsdaten. Hierzu gehören Knotenzustände, Schleifenzähler, Bearbeiter eines Prozessschritts sowie das Setzen der Kennungen von Kindprozessinstanzen – bei variabler Parallelität (*VariableParallelismEBP*) gibt es mehrere Kindprozessinstanzen für einen Prozessschritt.

Das Ausführungsmetamodell (s. Abschnitt 2.2.3) definiert neben normalen Knotenzuständen auch den speziellen Knotenzustand `ENQUIRED` für Rückfragen. Dies ermöglicht einem Bearbeiter (`Originator`), die Ausführung eines Prozessschritts zu unterbrechen, den Prozessschritt mit einer Frage (`Question`) zu versehen und die Frage zusammen mit dem Prozessschritt

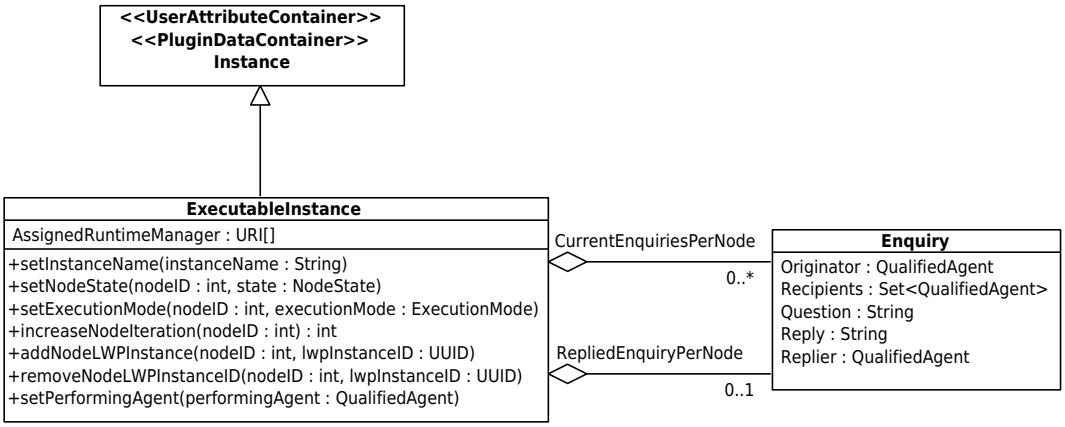


Abbildung 7.18: UML-Klassendiagramm von ausgeführten Prozessinstanzen des Ausführungsmodells

an einen oder mehrere andere Bearbeiter (`Recipients`) weiterzuleiten. Einer der Empfänger beantwortet die Frage (`Replier`) und kann dazu den Prozessschritt auch weiter ausführen. Es ist auch möglich, dass ein Empfänger den Prozessschritt selbst beendet oder eine weitere Rückfrage an andere Bearbeiter stellt und entsprechend auf eine Antwort wartet. Auf jeden Fall muss er die Frage irgendwann beantworten (`Reply`) und an den Fragesteller (`Originator`) zurückleiten – zusammen mit dem unterbrochenen oder beendeten Prozessschritt. Der Fragesteller kann aufgrund der Antwort den Prozessschritt beenden und mit diesem Wissen ähnliche Prozessschritte zukünftig selbständig ausführen. Die entsprechenden Enquiry-Objekte für offene Rückfragen (`CurrentEnquiriesPerNode`) und die letzte Antwort auf eine Rückfrage (`RepliedEnquiryPerNode`) sind für jeden Prozessschritt bei der *ExecutableInstance* hinterlegt.

Neben diesen Erweiterungen des Prozessmodells durch das Ausführungsmodell, gibt es Einschränkungen hinsichtlich der Informationen, die für die Ausführung eines einzelnen Prozessschritts durch den *RuntimeManager* nicht relevant sind. Hierzu zählt etwa die Prozessstruktur. Dementsprechend werden zwischen *ExecutionManager* und *RuntimeManager* auch keine vollständigen Prozessinstanzen ausgetauscht, sondern nur Instanzen von Prozessschritten, genauer EBP-Instanzen (*ExecutableBusinessProcessInstance*, s. Abb. 7.19). Diese besitzen gegenüber *ExecutableBusinessProcess* aus dem Prozessmodell zusätzlich die Namen der entsprechenden Prozessvorlage (`ProcessTemplateName`) und Prozessinstanz (`ProcessInstanceName`). Dafür besitzen sie nur eine *Configuration* und verbergen damit den Unterschied zwischen `Configuration` und `TestConfiguration`. Diese Unterscheidung ist hier nicht mehr nötig, da der *ExecutionManager* bei der Erzeugung einer *ExecutableBusinessProcessInstance* entscheidet, ob im Testmodus oder normalen Modus ausgeführt werden soll. Die *ExecutableBusinessProcessInstance* erhält dann die entsprechende Konfiguration und der *RuntimeManager* muss nicht zwischen normaler Ausführung und der Ausführung im Testmodus unterscheiden.

ActivityInstance und *DecisionActivityInstance* entsprechen in diesem Sinne *Activity* und *DecisionActivity* aus dem Prozessmodell. Etwas anders verhält es sich mit *LightWeightProcessInstance*. Hier werden beide Unterklassen von *LightWeightProcess* (*ReferencedProcess* und *EmbeddedProcess*) in einem Objekt des Ausführungsmodells vereinigt. Die variable Parallelität (*VariableParallelismEBP*) benötigt keine Entsprechung im Ausführungsmodell, da variable Parallelität nicht im normalen *RuntimeManager* interpretiert wird (s. Abschnitt 7.3.8.5) und damit die Informationen auch nicht über das Ausführungsmodell ausgetauscht werden müssen. Nach

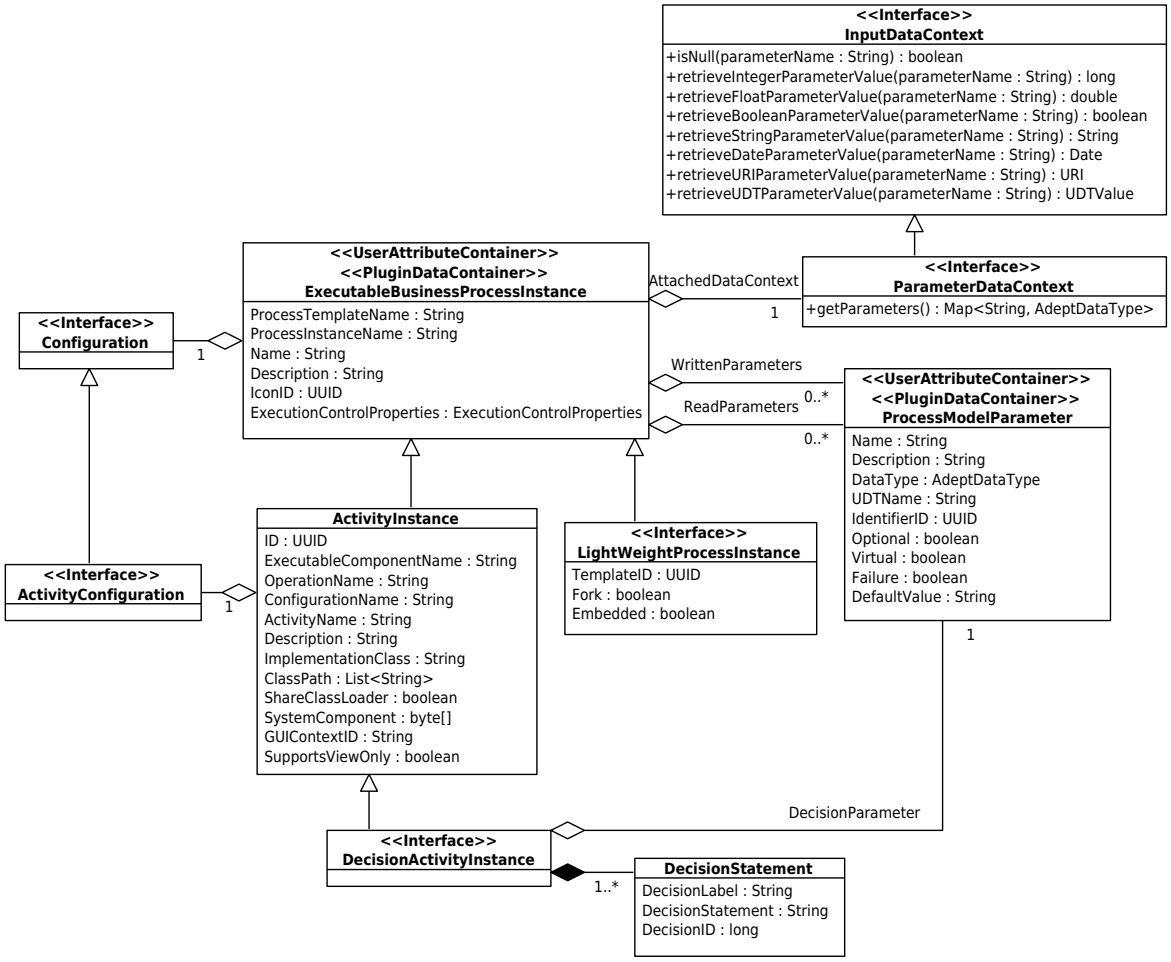


Abbildung 7.19: UML-Klassendiagramm von *Executable Business Process* des Ausführungsmodells

dem Start der einzelnen Kindprozessinstanzen einer variablen Parallelität werden diese durch *LightWeightProcess* im Prozessmodell und entsprechend durch *LightWeightProcessInstance* im Ausführungsmodell repräsentiert.

7.2.4.2 Datenkontext und Parameter

Die Entität *InputDataContext* bietet Zugriff auf die Eingabeparameter eines *ExecutableBusinessProcess* oder einer Prozessinstanz. Diese ist für die Ausführung im *RuntimeManager* aber noch nicht ausreichend, erfüllt an dieser Stelle im Ausführungsmodell (s. Abb. 7.19) aber auch einen anderen Zweck: *ParameterDataContext* erweitert *ExecutableBusinessProcessInstance* um die Werte öffentlicher Datenelemente einer Prozessinstanz, wodurch die Werte dem *WorklistManager* zur Verfügung stehen. So können diese Datenwerte bei Arbeitslisteneinträgen angezeigt werden und bieten somit prozessinstanzspezifische Informationen zu Arbeitsschritten:

Beispiel 7.12 (Aussagekräftige Arbeitslisteneinträge durch Datenelementwerte) Besitzt ein Prozessschritt einer Prozessvorlage für eine Bestellung den Namen „Verpackung der Bestellung“, haben auch alle Arbeitslisteneinträge dieses Prozessschritts diesen Namen. Damit kann der Benutzer die verschiedenen Prozessinstanzen dieser Prozessvorlage jedoch nicht auseinanderhalten. Eine Unterscheidung wäre beispielsweise über die Kennung (ID) der Prozessinstanzen möglich. Aussagekräftiger sind jedoch Informationen aus den Datenelementen der Prozesse. Im Fall eines Bestellprozesses ist dies etwa die Kundennummer und das Datum der Bestellung. Die Werte solcher Datenelemente werden durch *ParameterDataContext* zur Verfügung gestellt, was sprechende Namen für Arbeitslisteneinträge ermöglicht, etwa „Verpackung der Bestellung von Kunde 4711 vom 2.8.2010“.

Da die von *InputDataContext* bereitgestellten Werte naheliegenderweise von der entsprechenden Prozessvorlage abhängen und somit nur für die Arbeitslisteneinträge (der Prozessinstanzen) einer Prozessvorlage existieren, ist kein allgemeiner Zugriff auf diese Werte über *InputDataContext* möglich. Die dafür notwendigen Informationen stellt *ParameterDataContext* zur Verfügung. Dazu gehören die Namen der Datenelemente, für die Werte bereitstehen, sowie deren Datentypen. Damit lassen sich prozessunabhängige Arbeitslisten mit prozessbezogenen Zusatzinformationen realisieren.

Denselben Zweck erfüllen die erwähnten Vorlagen für Prozessinstanznamen (`InstanceNameTemplate`) im Prozessmodell. Diese Namensvorlagen können Variablen aufweisen, die Datenelemente oder Eingabeparameter von Prozessen referenzieren. Beim Start einer Prozessinstanz und bei jeder Wertänderung eines in der Namensvorlage referenzierten Datenelements wird der Name neu bestimmt und über *ExecutableInstance* entsprechend gesetzt (`setInstanceName`). Eine allgemeine Lösung hierfür erlaubt es, Inhalte beliebiger Zeichenketten aus Eingabeparametern zu beziehen. Die Zeichenketten werden somit parametrisiert; die entsprechenden Parameterwerte stammen aus Eingabeparametern von Aktivitäten und Prozessinstanzen. Vergleichbar ist dies mit der Formatierung von Zeichenketten in Java mittels `String.format`. Mit diesem Verfahren werden neben Namen von Prozessinstanzen und Prozessschritten/Arbeitslisteneinträgen auch die Abhängigkeiten von Bearbeiterzuordnungen realisiert.

Notwendig für parametrisierte Zeichenketten ist eine Menge von Eingabeparametern, d. h. die Ersetzung kann nur bei Entitäten vorgenommen werden, die Eingabeparameter besitzen. Im Fall von *ExecutableBusinessProcessInstance* ist dies direkt der Fall; für den Prozessinstanznamen gelten die Eingabeparameter der entsprechenden Prozessvorlage. Für Prozessinstanznamen sind

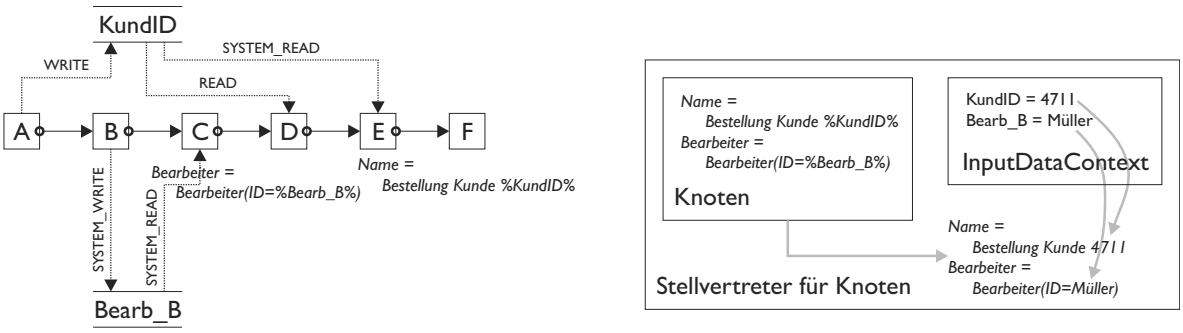


Abbildung 7.20: Parametrisierte Zeichenkettenattribute und Ersetzung durch Stellvertreterobjekt

Eingabeparameter oft nicht ausreichend, da die notwendigen Werte (z. B. die Kundennummer) erst im Laufe des Prozesses ermittelt werden und somit nicht aus Prozesseingabeparametern stammen können. Deshalb werden für Ersetzungen bei Prozessinstanzobjekten öffentliche Datenelemente ebenfalls als Eingabeparameter angesehen.

Parametrisierbare Namen von Knoten/Prozessschritten und abhängige Bearbeiterzuordnungen sind Attribute von Knoten. Knoten besitzen im Prozessmodell jedoch keine Parameter – zumindest keine normalen Ein-/Ausgabeparameter; diese gehören zu dem jeweils hinterlegten *ExecutableBusinessProcess*. Deshalb gibt es für Knoten die speziellen Systemparameter (`SystemParameter`). Dadurch werden diese Parameter sinnvollerweise auch von den normalen Eingabeparametern eines *ExecutableBusinessProcess* getrennt. Einerseits hat ein *ExecutableBusinessProcess* gewöhnlich eine feste Schnittstelle und damit eine vorgegebene Menge an Ein-/Ausgabeparametern, andererseits ist es nicht erwünscht, dass ein *ExecutableBusinessProcess* Zugriff auf die Systemparameter erhält, etwa den in einer abhängigen Bearbeiterzuordnung verwendeten Bearbeiter eines Vorgängerschritts.

Die entsprechenden Systemparameter eines Knotens werden bei der Prozessmodellierung erzeugt, sobald ein Knotenattribut parametrisiert wird, etwa durch Anlegen einer abhängigen Bearbeiterzuordnung oder Anpassen des Knotennamens. Jeder Systemparameter wird nach seiner Erzeugung mit dem dazugehörigen Prozesseingabeparameter oder Datenelement verbunden (s. Knoten E und Datenelement `KundID` in Abb. 7.20). Für abhängige Bearbeiterzuordnungen reicht dies aber noch nicht aus. Diese beziehen sich gewöhnlich auf Bearbeiterinformationen vorausgehender Prozessschritte, diese Informationen stehen aber per se weder in Prozesseingabeparametern noch in Datenelementen.

Wie bereits im Ausführungsmetamodell (s. Abschnitt 2.2.3.5) diskutiert, ist es sinnvoll, Abhängigkeiten von Bearbeiterzuordnungen über den Datenfluss zu realisieren. Deshalb wird zusätzlich zum Systemeingabeparameter des Knotens auch das entsprechende (System-)Datenelement angelegt, das die Bearbeiterinformation aufnimmt. Das Schreiben dieses Datenelements übernimmt der Knoten, zu dem die Abhängigkeit besteht. Bei einer Abhängigkeit des Bearbeiters eines Knotens Y zum Bearbeiter des Vorgängerknotens X, schreibt Knoten X seinen Bearbeiter in das Datenelement. Da der geschriebene Wert nicht explizit von einem *ExecutableBusinessProcess* erzeugt wird, sondern implizit vorhanden ist, gibt es hierfür den speziellen Datenkantentyp `SYSTEM_WRITE` (s. Knoten B und C und Datenelement `Bearb_B` in Abb. 7.20). Der *ExecutionManager* berücksichtigt diese Datenkanten beim Beenden von Prozessschritten und schreibt die entsprechenden Werte in die Datenelemente.

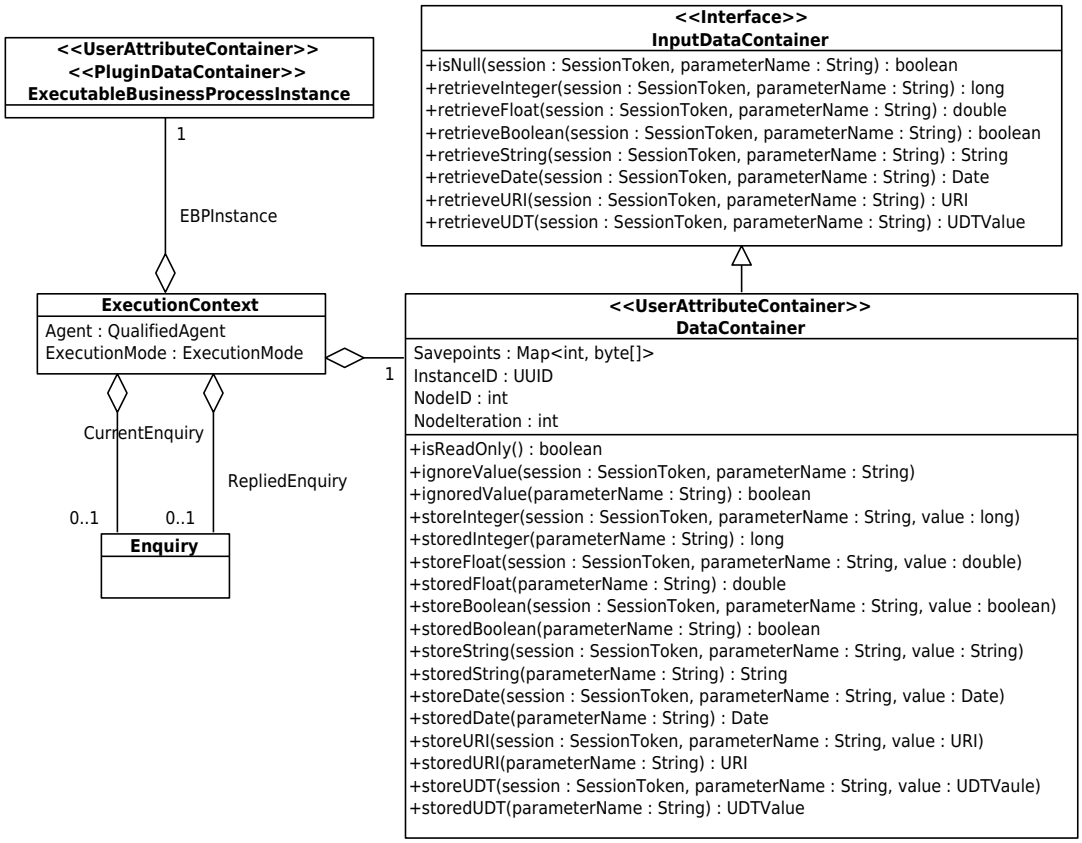


Abbildung 7.21: UML-Klassendiagramm zum Ausführungskontext des Ausführungsmodells

Für das Lesen von Datenelementen als Systemparameter gibt es entsprechend den Datenkantentyp `SYSTEM_READ`. Bei Aktivierung eines Knotens mit solchen Datenkanten erzeugt der *ExecutionManager* einen *InputDataContext*, der nur die entsprechenden Systemparameter enthält. Der *InputDataContext* wird zusammen mit dem Knotenobjekt in einem Stellvertreterobjekt (*Proxy*, [GHJV95]) gekapselt, welches das Knotenobjekt ersetzt. Bei Zugriffen auf eine parametrisierte Zeichenkette setzt das Stellvertreterobjekt die entsprechenden Werte aus dem *InputDataContext* für die Referenzen ein. Somit bleibt die Parameterersetzung gegenüber einem Verwender des Objekts verborgen (s. Abb. 7.20 rechts).

7.2.4.3 Ausführungskontext

Abb. 7.21 zeigt den Teil des Ausführungsmodells, der für die Ausführung von Prozessschritten durch den *RuntimeManager* relevant ist. Dies sind die zwischen *ExecutionManager* und *RuntimeManager* ausgetauschten Modellentitäten. Beim Start einer Aktivität wird dem *RuntimeManager* ein *ExecutionContext* übergeben. Dieser beinhaltet sämtliche Informationen, die zur Ausführung des entsprechenden *ExecutableBusinessProcess* notwendig sind. Dazu gehören neben der entsprechenden *ExecutableBusinessProcessInstance* ein *DataContainer* für die Ein-/Ausgabeparameter sowie ggf. Rückfragen (*Enquiry*). Die Rückfragen umfassen zum einen die aktuelle Rückfrage, falls ein anderer Bearbeiter dem aktuellen Bearbeiter eine Frage

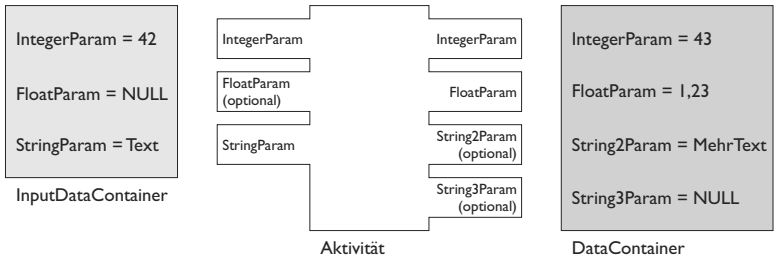


Abbildung 7.22: Zugriff auf *InputDataContainer* und *DataContainer*

gestellt hat (`CurrentEnquiry`), und zum anderen eine Antwort, sofern der aktuelle Bearbeiter zuvor selbst eine Rückfrage zu diesem Prozessschritt gestellt hat (`RepliedEnquiry`).

Ein *DataContainer* bietet Zugriff auf die Werte der Ein- und Ausgabeparameter des ausgeführten *ExecutableBusinessProcess*, wobei auch auf Ebene der Schnittstellen zwischen beiden Parameterarten unterschieden wird (*InputDataContainer* für Eingabeparameter, *DataContainer* für Ausgabeparameter, s. Abb. 7.22). Diese Unterscheidung erlaubt es, jederzeit auf den Wert eines Eingabeparameters zuzugreifen, auch wenn ein Ausgabeparameter desselben Namens existiert und für diesen bereits ein Wert geschrieben worden ist. *DataContainer* ermöglicht das Schreiben von Parameterwerten und das Lesen bereits geschriebener Ausgabeparameterwerte. Diese Werte können auch jederzeit wieder geändert werden, solange die Ausführung noch nicht beendet ist. Für optionale Ausgabeparameter sind auch `NULL`-Werte zulässig, d. h. der entsprechende Parameter wird ignoriert, wofür es die Methoden `ignoreValue` und `ignoredValue` gibt.

Beispiel 7.13 (Lesen und Schreiben von Ein-/Ausgabeparametern)
Abb. 7.22 zeigt eine Aktivität mit den Eingabeparametern `IntegerParam`, `FloatParam` und `StringParam`, wobei `FloatParam` optional ist. Sie hat die Ausgabeparameter `IntegerParam`, `FloatParam`, `String2Param` und `String3Param`. `String2Param` und `String3Param` sind optional. Die Werte der Eingabeparameter werden mittels der `InputDataContainer.retrieve`-Methoden eingelesen. Werte von Ausgabeparameter werden mittels der `DataContainer.store`-Methoden geschrieben. Das Schreiben von Ausgabeparameterwerten kann zu jedem Zeitpunkt erfolgen. Dabei können auch bereits geschriebene Werte neu gesetzt werden. Gelesen werden können die Werte von Ausgabeparametern mittels der `DataContainer.stored`-Methoden. Dadurch kann die Aktivität auch bei gleichnamigen Ein- und Ausgabeparametern zwischen gelesenen und geschriebenen Werten unterscheiden, etwa „42“ als Eingabeparameterwert von `IntValue` und „43“ als Wert des gleichnamigen Ausgabeparameters.

Der Wert des Ausgabeparameters `String3Param` ist undefiniert, was mittels `DataContainer.ignoredValue` ermittelt wird. Ein bereits geschriebener optionaler Ausgabeparameter wie `String2Param` kann mit `DataContainer.ignoreValue` als undefiniert gekennzeichnet werden.

Neben den normalen Ausgabeparametern können auch Zwischenzustände der aktuellen Ausführung im *DataContainer* gespeichert werden (Sicherungspunkte, `Savepoints`). Diese erlauben es, einen fehlgeschlagenen Anwendungsbaustein in einem solchen Zwischenzustand fortzusetzen anstatt ihn nochmals vollständig auszuführen. Sicherungspunkte sind aber nicht nur im Fehlerfall nützlich, über sie wird auch das Pausieren und Fortsetzen von Aktivitäten/Anwendungsbausteinen zu einem beliebigen Zeitpunkt realisiert. Dazu schreibt der Anwendungsbaustein bei einer Pausierungsanforderung seinen aktuellen Zustand, also die Belegungen lokaler Variablen, in einen Sicherungspunkt (vom Typ `byte[]`) und beendet sich. Bei der Fortsetzung der

Ausführung wird dem Anwendungsbaustein der entsprechende Sicherungspunkt mitgeteilt, über den er seinen vorherigen Zustand aus dem *DataContainer* auslesen und wiederherstellen kann. Da die bereits geschriebenen Werte von Ausgabeparametern dabei auch berücksichtigt werden, kann ein Sicherungspunkt leichtgewichtig sein. Er muss lediglich die lokalen Variablen beinhalten, die nicht auch im Datenkontext vorhanden sind.

7.2.5 Laufzeitumgebungsmodell (*Runtime Environment*)

Das Ausführungsmodell wird zwar zum Datenaustausch zwischen *ExecutionManager* und *RuntimeManager* verwendet, es ist aber nicht für eine einfache Anwendungsintegration geeignet. Die Anwendungsintegration bezieht sich auf die Schnittstelle zwischen *RuntimeManager* und Anwendungsbaustein und wird von Entwicklern am häufigsten verwendet. Deshalb sollte die Schnittstelle möglichst wenig Anforderungen an einen Anwendungsbaustein stellen, um so eine möglichst einfache Programmierung zu erlauben (FA 7). Gleichzeitig muss die Schnittstelle aber alle Funktionen zur Verfügung stellen, die für einen Anwendungsbaustein relevant sein können, beispielsweise auch das 2-Phasen-Commit-Protokoll [HR01].

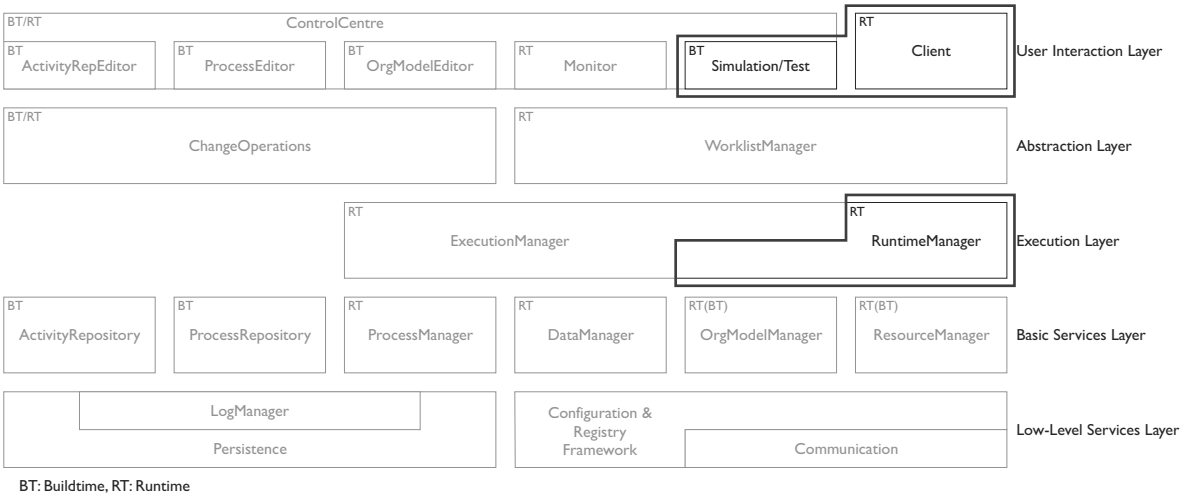


Abbildung 7.23: Verwendung des Laufzeitumgebungsmodells in der Gesamtarchitektur

Zur Anwendungsintegration gehört auch die Integration der graphischen Benutzerschnittstellen von Anwendungsbausteinen in die allgemein gehaltene Ausführungsumgebung (*RuntimeManager*, *Client*, *Simulation/Test*, s. Abb. 7.23). Außerdem muss es die Anwendungsintegration ermöglichen, weitgehend beliebige Programmiersprachen für Anwendungsbausteine zu verwenden. Beides bringt eine Abhängigkeit des Anwendungsbausteins zur Ausführungsumgebung im weiteren Sinne mit, etwa zum *Client* oder zum Betriebssystem. Diese Abhängigkeit sollte sich nur minimal auf das Laufzeitumgebungsmodell auswirken, damit ein einziges, allgemeingültiges Modell für alle Implementierungssprachen und graphische Benutzeroberflächen verwendet wird und keine speziellen Modelle oder Modellerweiterungen notwendig werden.

7.2.5.1 *ExecutableComponent*

Abb. 7.24 zeigt die Schnittstelle zwischen *RuntimeManager* und Anwendungsbaustein. Dabei ist *ExecutableComponent* (links oben) die einzige Schnittstelle, die der Anwendungsbaustein

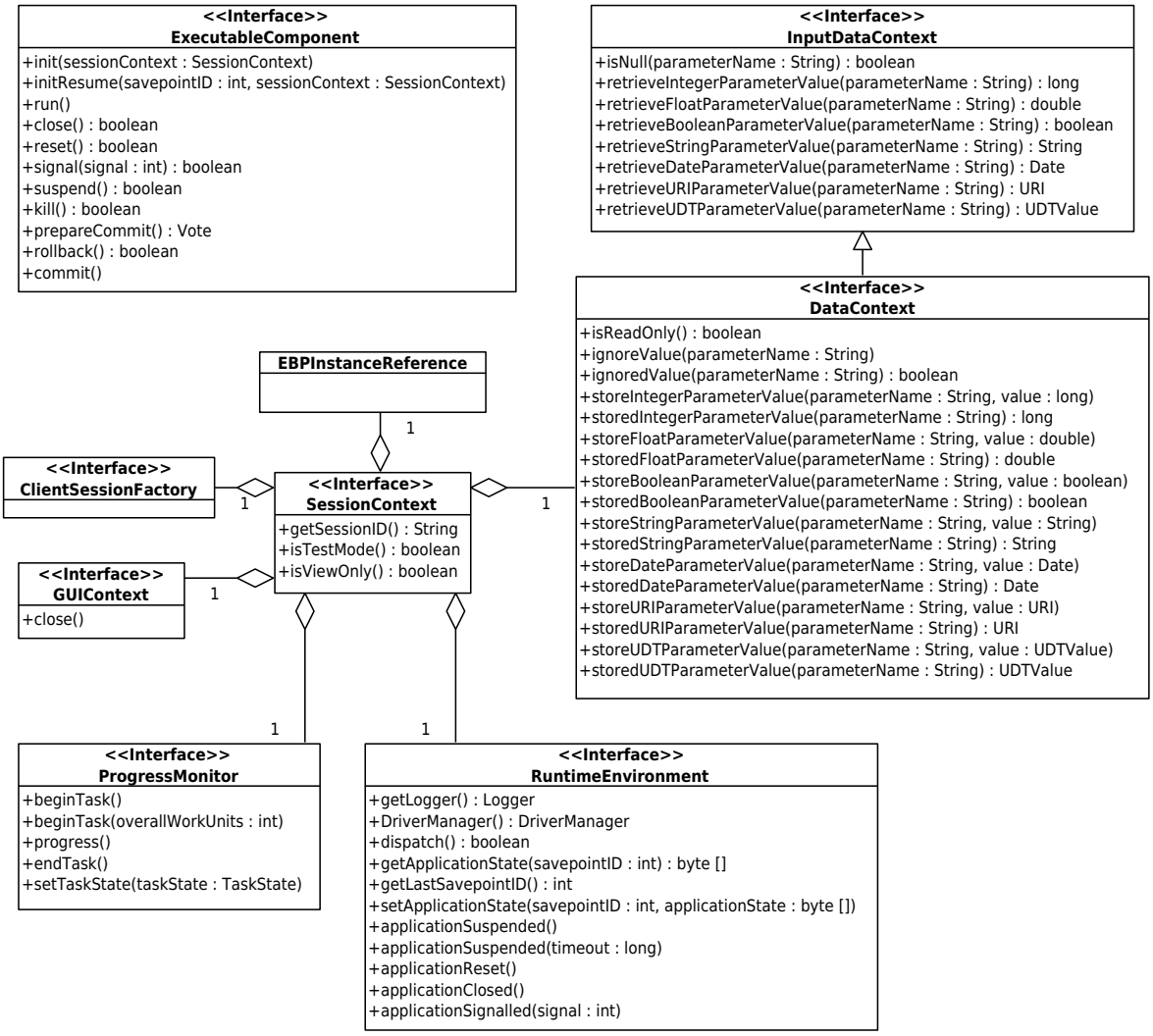


Abbildung 7.24: UML-Klassendiagramm des Laufzeitumgebungsmodells als Schnittstellen von Anwendungsbausteinen und *RuntimeManager*

implementieren muss. Es sind jedoch nicht alle Methoden notwendig, `init` und `run` reichen bereits aus. Über `init` erhält der Anwendungsbaustein Zugriff auf seine Ausführungsumgebung (*SessionContext*), `run` realisiert die eigentliche Ausführung. Unterstützt der Anwendungsbaustein das Pausieren und Fortsetzen der Ausführung, so ist zusätzlich `initResume` relevant, das für die Initialisierung bei der fortgesetzten Ausführung verwendet wird. Dazu wird zusätzlich zum *SessionContext* die Kennung (ID) des Sicherungspunkts mitgeteilt (`savepointID`), von dem aus die Ausführung fortgesetzt werden soll. Die restlichen Methoden von *ExecutableComponent* dienen dazu, dem Anwendungsbaustein von außen Signale mitzuteilen, beispielsweise um eine Ausführung jederzeit von außen zu beenden (`close`) oder abzubrechen (`kill`) (s. Abschnitt 2.4.2). Welche dieser Signale ein Anwendungsbaustein versteht, ist in den `ExecutionControlProperties` im Aktivitäten- (s. Abschnitt 7.2.2) und Prozessmodell (s. Abschnitt 7.2.3) hinterlegt. Zusätzlich gibt die Methoden `prepareCommit`, `rollback` und `commit`, um die Ausführung im Rahmen eines 2-Phasen-Commit-Protokoll durchzuführen.

7.2.5.2 *SessionContext*

Der bei der Initialisierung übergebene *SessionContext* bietet Zugriff auf den Prozesskontext, soweit dieser für einen Anwendungsbaustein relevant ist (*EBPInstanceReference*), die Daten aus dem Prozess (*DataContext*) sowie Schnittstellen zur Kommunikation mit der Umgebung (*RuntimeEnvironment*, *ProgressMonitor*, *GUIContext*, *ClientSessionFactory*). *InputDataContext* und *DataContext* entsprechen *InputDataContainer* und *DataContainer* aus dem Ausführungsmodell (s. Abschnitt 7.2.4), allerdings benötigt der Anwendungsbaustein kein *SessionToken* für Zugriffe. Außerdem sind die Sicherungspunkte nicht im *DataContext* abgelegt, sondern in *RuntimeEnvironment*. Dieser Unterschied zwischen Ausführungs- und Laufzeitumgebungsmodell liegt darin begründet, dass aus Sicht des gesamten PrMS interne Anwendungszustände als Anwendungsdaten betrachtet und entsprechend behandelt werden (im *DataContainer*). Aus Sicht des Anwendungsbausteins ist ein Sicherungspunkt jedoch kein Datenwert im Sinne eines Ein-/Ausgabeparameters, sondern eher mit einer speziellen Datei im Dateisystem vergleichbar, auf die er über die Laufzeitumgebung zugreift.

Neben dem Zugriff auf die in Sicherungspunkten gespeicherten Anwendungszustände, bietet *RuntimeEnvironment* auch Zugriff auf weitere Ressourcen. Hierzu gehört ein Protokolldienst (*Logger*), der Lognachrichten des Anwendungsbausteins mit wichtigen Zusatzinformationen versieht, die der Anwendungsbaustein selbst aber nicht kennt, etwa Details zum entsprechenden Prozessschritt. Weiterhin signalisiert der Anwendungsbaustein über *RuntimeEnvironment* seinen Endzustand, bevor er beendet wird. Dieser Endzustand spiegelt sich anschließend im entsprechenden Prozessschritt wider (beendet, pausiert, fehlgeschlagen, ...). Während *RuntimeEnvironment* die Kommunikation zwischen Anwendungsbaustein und *RuntimeManager* bereitstellt, dienen *GUIContext* und *ProgressMonitor* der Kommunikation mit dem Benutzer. Der *GUIContext* bietet normale graphische Benutzeroberflächen, *ProgressMonitor* erlaubt einfache Verlaufsbalken, die die Dauer der Ausführung widerspiegeln und sich somit gut für langlaufende Anwendungsbausteine ohne Benutzerinteraktion eignen.

7.2.5.3 *GUIContext*

GUIContext abstrahiert von den verschiedenen Möglichkeiten und Bibliotheken, die es für graphische Benutzeroberflächen gibt, z. B. HTML (*Hypertext Markup Language*), AWT (*Abstract Window Toolkit*), SWT (*Standard Window Toolkit*), Qt oder GTK+ (GIMP-Toolkit). *GUIContext* selbst bietet nur die Methode `close` zum Freigeben und Aufräumen von Ressourcen, nachdem die Ausführung des Anwendungsbausteins beendet ist. Abhängig von den Erfordernissen des Anwendungsbausteins und den in der Umgebung vorhandenen Mitteln, erzeugt ein Teil des *RuntimeManager*s vor der Initialisierung eines Anwendungsbausteins einen entsprechenden speziellen *GUIContext*. Dieser bietet die für die jeweilige graphische Oberfläche notwendigen Interaktionsmöglichkeiten (s. Abb. 7.25). So reicht bei AWT und SWT ein Containerobjekt aus, auf dem die graphische Oberfläche des Anwendungsbausteins erzeugt wird. Der *GUIContext* für HTML dagegen stellt eine Abstraktion des HTTP-Protokolls (*Hypertext Transfer Protocol*) zur Verfügung. Dazu gehören URL (*Uniform Resource Locator*), mit denen die erzeugte HTML-Oberfläche mit dem Anwendungsbaustein kommunizieren kann, etwa um Formulardaten zu übergeben. Die Kommunikation erfolgt mit den Methoden `waitForReply` und `setResponse` des *HTMLContext*. Damit wartet der Anwendungsbaustein auf eine Anfrage der erzeugten HTML-Oberfläche mittels *HTTP-Request*. Sobald dieser eintrifft,

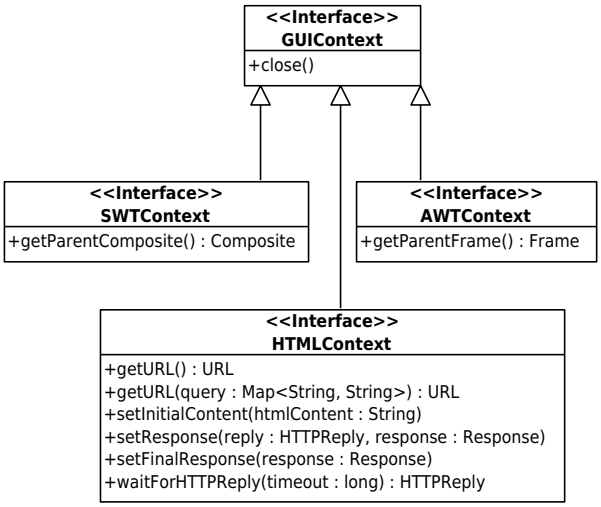


Abbildung 7.25: UML-Klassendiagramm von graphischen Kontexten für Anwendungsbausteine als Teil des Laufzeitumgebungsmodells

kann ihn der Anwendungsbaustein entsprechend verarbeiten und eine Antwort zurücksenden (sog. *HTTP-Reply*).

Während für den *HTMLContext* zur Verarbeitung der HTTP-Kommunikation ein lokaler HTTP-/Webserver benötigt wird, benötigen *SWTContext* und *AWTContext* lokal die entsprechenden betriebssystemabhängigen Bibliotheken. Dies bedeutet jedoch nicht, dass auch die *Client*-Oberfläche auf denselben Bibliotheken beruhen muss. Es reicht, wenn die Bibliotheken vorhanden sind und sie in die *Client*-Oberfläche integriert werden können. So lässt sich mittels einer *SWT/AWT-Bridge* beispielsweise SWT innerhalb von AWT darstellen und umgekehrt. Damit solche verschiedenen Oberflächen zusammen funktionieren, muss lediglich der verwendete *GUIContext* entsprechend implementiert werden; weder der Anwendungsbaustein noch der *Client* sind davon betroffen. Somit schafft der GUIContext eine Trennung zwischen Anwendungsbausteinen und *Client*, um diese unabhängig voneinander zu entwickeln.

7.2.5.4 Besonderheit des Laufzeitumgebungsmodells

Wie man sieht, ist beim Laufzeitumgebungsmodell die Trennung zwischen Daten und Anwendungslogik nicht mehr allzu strikt. Während beispielsweise *DataContext* und *EBPInstanceReference* Daten repräsentieren, sind *ExecutableComponent* und *RuntimeEnvironment* der Anwendungslogik zuzuordnen. Dies ist der Einfachheit der Schnittstelle im Hinblick auf die Entwicklung von Anwendungsbausteinen geschuldet. So kommt das Modell mit wenigen und leicht verständlichen Schnittstellen aus. Die Trennung zwischen Daten und Anwendungslogik ist in diesem Modell auch nicht so wichtig, da die Daten hier nicht gespeichert und nicht mit anderen Diensten ausgetauscht werden; das Laufzeitumgebungsmodell wird nur lokal und innerhalb des *RuntimeManager*s verwendet.

7.2.6 Arbeitslistenmodell

Das Arbeitslistenmodell dient der Interaktion zwischen Bearbeitern von Prozessschritten und dem PrMS – im Gegensatz zur Interaktion zwischen Bearbeitern und Anwendungsbausteinen,

die über den *GUIContext* des Laufzeitumgebungsmodells erfolgt. Das Arbeitslistenmodell ist eine konsolidierte Sicht auf die Zustände aller laufenden Prozessinstanzen. Während die Überwachung (*Monitor*) das Prozessmodell (s. Abschnitt 7.2.3) verwendet, stehen beim Arbeitslistenmodell nicht die Prozessinstanzen als Ganzes im Vordergrund, sondern die auszuführende Arbeit in Form von Prozessschritten. Dementsprechend bezieht sich das Arbeitslistenmodell auf benutzernahe Dienste wie *Client* und *Simulation/Test* sowie der Aufbereitung der Arbeitslisten im *WorklistManager* (s. Abb. 7.26). Durch den Bezug auf den Prozessschritt als maßgebliche Entität, ist das Arbeitslistenmodell auch das Modell, das ressourcen- und zeitbezogene Informationen umfasst, beispielsweise die Priorisierung und die Komplexität von Prozessschritten, Eskalation, wenn ein Prozessschritt zu lange unbearbeitet bleibt, oder Delegation, also die Weitergabe von Arbeit an einen anderen Bearbeiter.

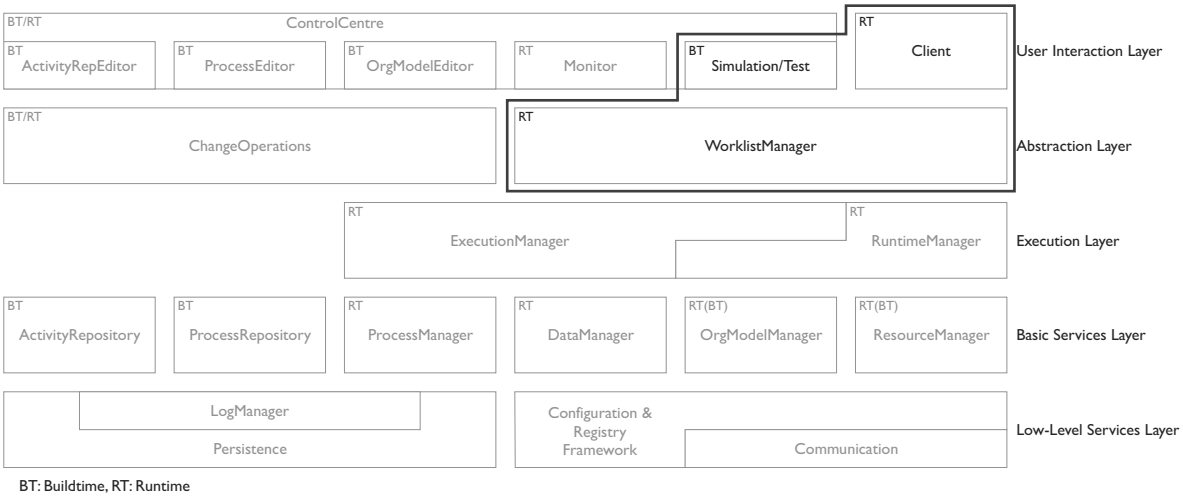


Abbildung 7.26: Verwendung des Arbeitslistenmodells in der Gesamtarchitektur

7.2.6.1 Arbeitslisteneinträge

Zentrale Entität des Arbeitslistenmodells ist der Arbeitslisteneintrag (*WorklistItem*, s. Abb. 7.27 und Abb. 7.30). Dieser repräsentiert einen aktivierten, laufenden oder pausierten Prozessschritt. Die Attribute von *WorklistItem* spiegeln die Repräsentation in Form diverser textueller Bezeichnungen (`Title`, `Description`, `ProcessTemplateName`, `ProcessInstanceName`) wider. Außerdem hat ein Arbeitslisteneintrag eine optionale Referenz auf ein zentral bereitgestelltes Symbol für die graphische Anzeige des Arbeitslisteneintrags (`IconID`) sowie den Zustand des Arbeitslisteneintrags und damit auch des entsprechenden Prozessschritts (`State`). Weitere Attribute dienen der Arbeits- und Zeitplanung, etwa die Zeitpunkte der Aktivierung des entsprechenden Prozessschritts (`ActivationDate`), der Zuweisung (`AssignmentDate`) zu einem Bearbeiter (`AssignedAgent`), der Fälligkeit (`DueDate`) und der Eskalation (`EscalationDate`).

Ein Arbeitslisteneintrag ist unabhängig vom zugrundeliegenden Prozessmodell und ermöglicht es so, beliebige prozessorientierte Informationssysteme durch das Arbeitslistenmodell zusammenzuführen und in einer Arbeitsliste anzuzeigen. Dazu muss lediglich der *WorklistManager* mit den entsprechenden Systemen verbunden werden. Die Unabhängigkeit vom Prozessmodell zeigt sich im Arbeitslistenmodell am Element *ActivityReference*. Dieses repräsentiert einen Prozessschritt oder eine vergleichbare Entität eines prozessorientierten Informationssystems.

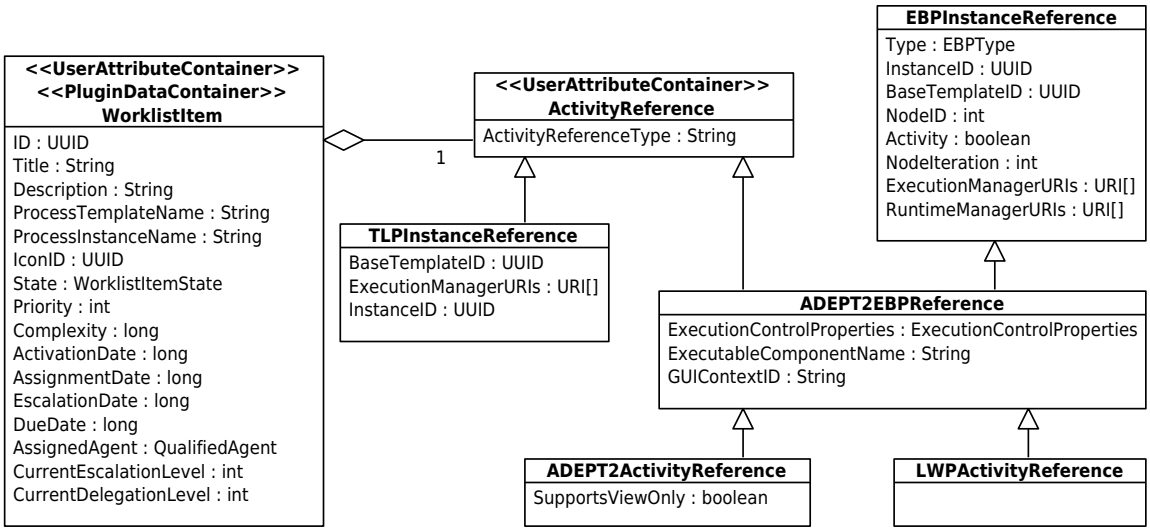


Abbildung 7.27: UML-Klassendiagramm von Arbeitslisteneinträge (abstrakt) im Arbeitslistenmodell

Die in Abb. 7.27 gezeigten abgeleiteten Klassen beziehen sich naheliegenderweise auf das im Rahmen dieser Arbeit geltende Prozessmodell (s. Abschnitt 7.2.3).

TLPInstanceReference ist ein Spezialfall und repräsentiert keinen Prozessschritt, sondern eine ganze Prozessinstanz und zwar eine auf oberster Ebene (*top-level*). Dies ist vergleichbar mit Subprozessinstanzen, die jedoch immer einem Prozessschritt zugeordnet werden können, nämlich dem entsprechenden Prozessschritt der Elterninstanz. Im Fall einer *TLPInstanceReference* gibt es keine Elternprozessinstanz und damit auch keinen entsprechenden Prozessschritt. Deshalb beschränken sich die Attribute auf Informationen bezüglich der Prozessinstanz, etwa `InstanceID` und die Adresse des die Prozessinstanz ausführenden *ExecutionManager* (`ExecutionManagerURIs`). Arbeitslisteneinträge für Prozessinstanzen auf oberster Ebene bilden eine einfache Möglichkeit, Eskalationsmechanismen für ganze Prozessinstanzen anzubieten. Die Eskalation von Prozessinstanzen unterscheidet sich zwar semantisch von der Eskalation eines Prozessschritts, seitens der Implementierung ist aber keine Unterscheidung notwendig. Der semantische Unterschied besteht darin, dass der Verzug bei einer Prozessinstanz in der Regel mehrere Prozessschritte umfasst, die für sich betrachtet nicht in Verzug sein müssen. Dementsprechend erlaubt diese einfache Lösung auch nur grundlegende Eskalationsmechanismen für Prozessinstanzen, etwa die Benachrichtigung des Prozessverantwortlichen. Komplexe prozessschrittspezifischen Maßnahmen sind beispielsweise nicht möglich, da nicht bestimmt werden kann, welcher Prozessschritt verzögert ist.

ADEPT2EBPReference repräsentiert einen Prozessschritt und erweitert deshalb neben *ActivityReference* aus dem Arbeitslistenmodell gleichzeitig auch *EBPInstanceReference* aus dem Prozessmodell (s. Abschnitt 7.2.3). Zusätzlich bietet *ADEPT2EBPReference* die vom entsprechenden *ExecutableBusinessProcess* verarbeiteten Laufzeitsignale (`ExecutionControlProperties`), den Namen des zu rufenden Anwendungsbausteins (`ExecutableComponentName`, s. Abschnitt 7.2.2) und den Namen des verwendeten *GUIContext*s (s. Abschnitt 7.2.5). Mittels des *GUIContext*-Namens kann ein *Client* entscheiden, ob er den benötigten *GUIContext* zur Verfügung stellen und somit den entsprechenden Arbeitsschritt ausführen kann.

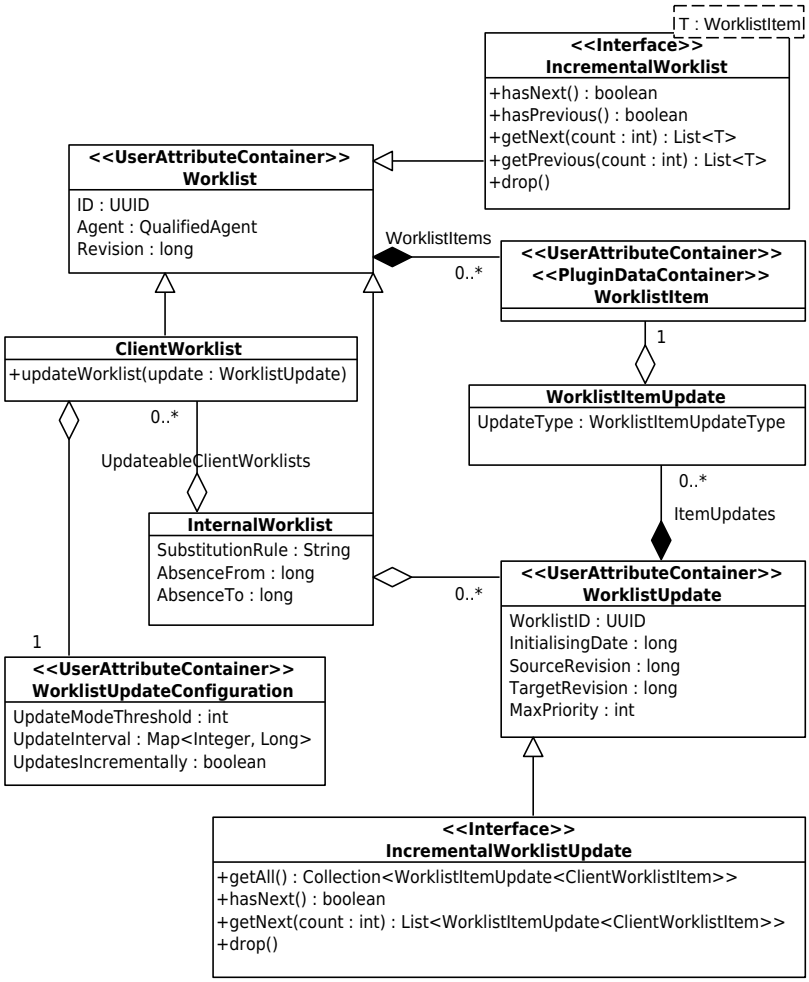


Abbildung 7.28: UML-Klassendiagramm von Arbeitslisten im Arbeitslistenmodell

ADEPT2EBPReference ist entweder eine *ADEPT2ActivityReference* für Aktivitäten (mit dem Schalter für den Historienmodus, `SupportsViewOnly`) oder eine *LWPActivityReference* für Subprozessinstanzen. Wie bei der Vorstellung des Prozessmodells (s. Abschnitt 7.2.3) erwähnt, besitzen Subprozessinstanzen im Gegensatz zu Prozessinstanzen auf höchster Ebene eine *EBPInstanceReference* auf den entsprechenden Prozessschritt der Elterninstanz. Diese *EBPInstanceReference* entspricht der dazugehörigen *LWPActivityReference* bei einem Arbeitslisteneintrag.

7.2.6.2 Arbeitslisten

Eine weitere wichtige Entität des Arbeitslistenmodells ist die Arbeitsliste (*Worklist*, s. Abb. 7.28). Sie verwaltet die Arbeitslisteneinträge eines Benutzers. Eine Arbeitsliste hat neben dem Benutzer (`Agent`) noch eine Kennung (`ID`) und einen Stand (`Revision`), d. h. einen Zähler für die letzte Aktualisierung der Arbeitsliste. Man unterscheidet *ClientWorklist* und *InternalWorklist*. *InternalWorklist* ist die interne Repräsentation des *WorklistManager*, hiervon existiert für jeden Benutzer genau ein Objekt. Ihre Arbeitslisteneinträge werden immer sofort aktualisiert, sobald sich der Zustand der entsprechenden Prozessschritte ändert. Bei ihr wird

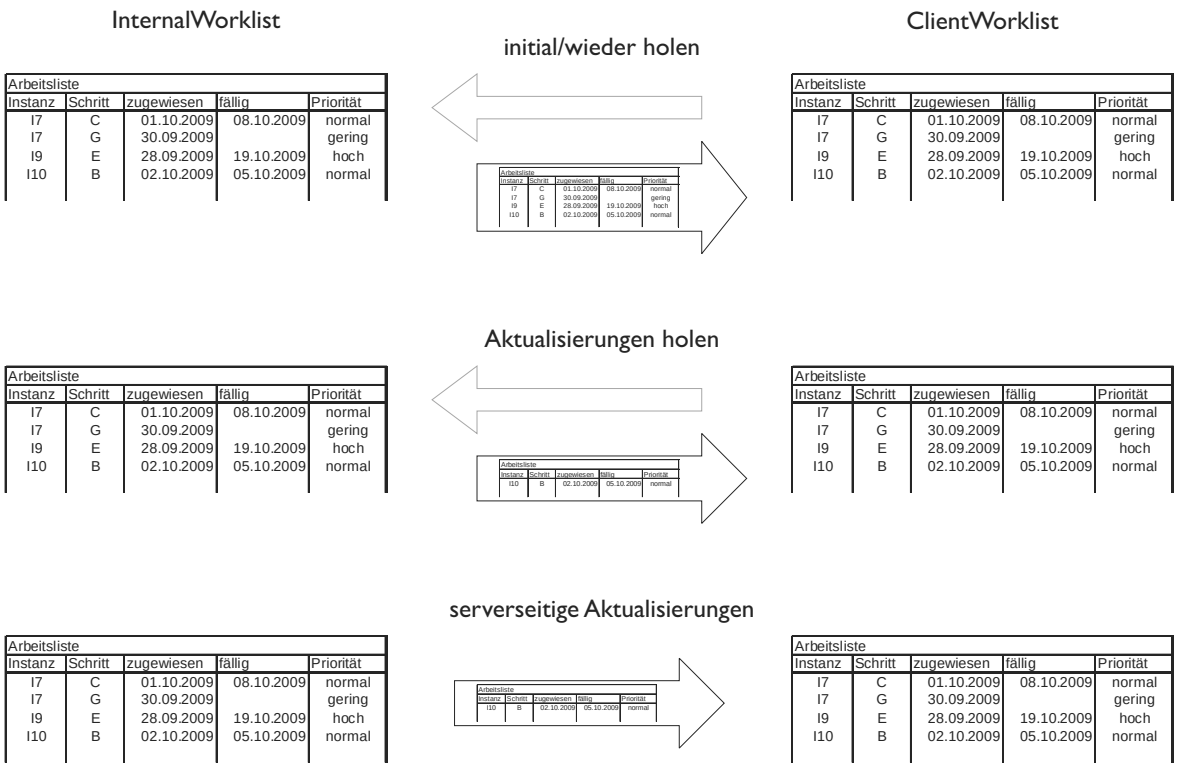


Abbildung 7.29: Aktualisierung einer Arbeitsliste (*ClientWorklist*)

auch hinterlegt, ob und wie lange der entsprechende Benutzer abwesend ist (`AbsenceFrom`, `AbsenceTo`) und welche anderen Benutzer ihn vertreten (`SubstitutionRule`).

Dagegen bildet *ClientWorklist* die Sicht eines Anwendungsprogramms (z. B. *Client*) auf die Arbeitsliste. Jedes dieser Anwendungsprogramme hat seine eigene Sicht, d. h. wenn ein Benutzer mit mehreren Anwendungsprogrammen gleichzeitig arbeitet, hat er auch mehrere *ClientWorklist*-Objekte. Diese spiegeln die *InternalWorklist* wider, sie können sich aber im Stand unterscheiden. Eine *ClientWorklist* muss nicht denselben Aktualitätsgrad wie die entsprechende *InternalWorklist* oder andere *ClientWorklist*-Objekte derselben Arbeitsliste aufweisen. Der Aktualitätsgrad einer *ClientWorklist* hängt von den Bedürfnissen des Anwendungsprogramms und des Benutzers ab. Aktualisierungen der *ClientWorklist* können manuell oder automatisch (durch den Server ausgelöst, sog. *Push-Update*) erfolgen (s. Abb. 7.29). Manuelle Aktualisierungen werden über `updateWorklist` eingebracht. Über *WorklistUpdateConfiguration* kann bei einer *ClientWorklist* angegeben werden, welche Priorität geänderte Arbeitslisteneinträge haben müssen, damit die Änderung eine automatische Aktualisierung auslöst (`UpdateModeThreshold`). Ferner gibt *WorklistUpdateConfiguration* an, welche Zeitspanne zwischen automatischen Aktualisierungen liegt, abhängig von der Priorität der geänderten Arbeitslisteneinträge (`UpdateInterval`). Indem diese Einstellungen für eine *ClientWorklist* angegeben werden, kann beispielsweise ein Arbeitsplatzrechner häufigere Aktualisierungen erhalten als ein mobiles Endgerät.

IncrementalWorklist ist in ihrer Verwendung und ihrem Verhalten zwischen *InternalWorklist* und *ClientWorklist* anzusiedeln. Sie spiegelt einen bestimmten Stand der *InternalWorklist* wider und stellt diesen einem Anwendungsprogramm zur Verfügung. Allerdings werden nicht

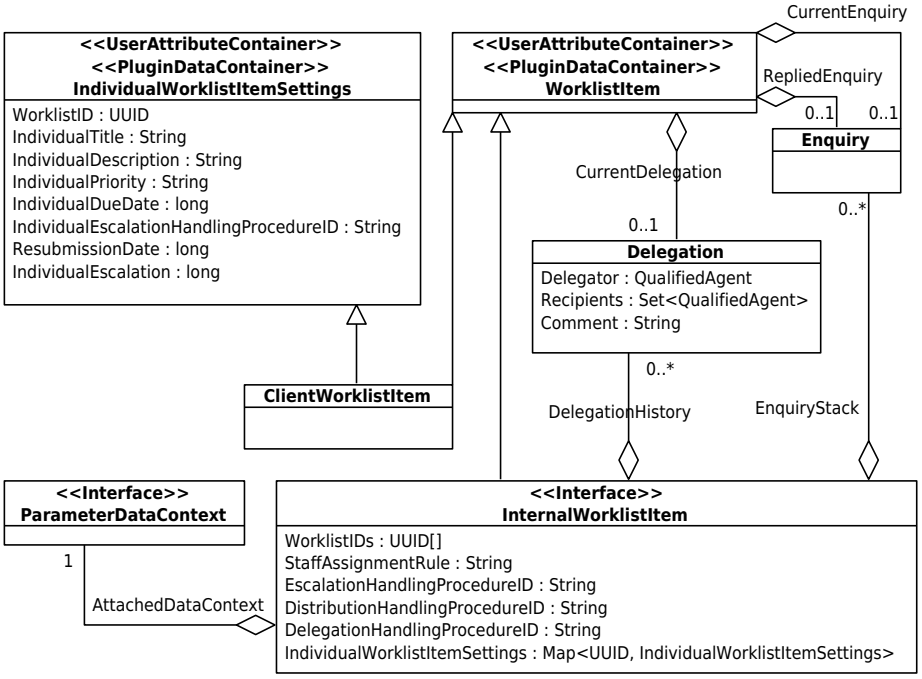


Abbildung 7.30: UML-Klassendiagramm von Arbeitslisteneinträgen im Arbeitslistenmodell

alle Arbeitslisteneinträge auf einmal, sondern gruppenweise übertragen. Dadurch vermindert sich das übertragene Datenvolumen, was insbesondere bei vielen Arbeitslisteneinträgen oder wenig Übertragungskapazität sinnvoll ist. Die Schnittstelle von *IncrementalWorklist* ermöglicht dementsprechend einen positionsbasierten Zugriff auf Arbeitslisteneinträge. Im Gegensatz zu *ClientWorklist* kann eine *IncrementalWorklist* allerdings nicht aktualisiert werden. Stattdessen muss das Anwendungsprogramm eine neue *IncrementalWorklist* vom *WorklistManager* anfordern. Dies liegt daran, dass Aktualisierungen die Reihenfolge der Arbeitslisteneinträge ändern können und die Position nicht mehr stimmt. Somit ist ein positionsbasierter Zugriff nicht ohne weiteres möglich.

Aktualisierungen der *ClientWorklist* erfolgen mittels *WorklistUpdate*. Ein *WorklistUpdate*-Objekt bezieht sich auf eine bestimmte Arbeitsliste (`WorklistID`). Es enthält sämtliche Änderungen der Arbeitsliste, die zwischen einem alten Stand (`SourceRevision`) und einem aktuelleren Stand (`TargetRevision`) erfolgt sind. Die Änderungen betreffen einzelne Arbeitslisteneinträge (`WorklistItemUpdate`) und geben an, ob ein Arbeitslisteneintrag seit dem alten Stand hinzugefügt, geändert oder entfernt (`UpdateType`) worden ist. Zur Verringerung der Datenmenge können auch Aktualisierungen gruppenweise übertragen werden. Dementsprechend ähnelt die entsprechende Schnittstelle *IncrementalWorklistUpdate* der von *IncrementalWorklist*, d. h. sie bietet positionsbasierten Zugriff auf Aktualisierungselemente bietet.

7.2.6.3 Repräsentation von Arbeitslisteneinträgen

Ebenso wie bei Arbeitslisten zwischen internen und externen Sichten unterschieden wird, wird auch bei den entsprechenden Arbeitslisteneinträgen zwischen intern und extern unterschieden. Abb. 7.30 detailliert das abstrakte Klassendiagramm aus Abb. 7.27 entsprechend. *WorklistItem* selbst besitzt, zusätzlich zu den in Abb. 7.27 gezeigten Attributen und Beziehungen, die aktuelle

Rückfrage (`CurrentEnquiry`) und/oder eine Antwort auf eine Rückfrage (`RepliedEnquiry`) entsprechend dem *ExecutionContext* des Ausführungsmodells (s. Abschnitt 7.2.4). Ist der Arbeitslisteneintrag von einem Bearbeiter zur Erledigung weitergeleitet (delegiert) worden, so werden diese Informationen über *Delegation* bereitgestellt. Hierzu gehören der delegierende Bearbeiter (`Delegator`), der oder die Delegationsempfänger (`Recipients`) sowie ein Kommentar zur Delegation (`Comment`) ähnlich einer Rückfrage. Allerdings wird bei einer Delegation im Unterschied zu einer Rückfrage keine Antwort erwartet.

InternalWorklistItem ist die interne Repräsentation eines Arbeitslisteneintrags. Hiervon existiert für jeden Prozessschritt nur ein Objekt, d. h. ein *InternalWorklistItem* wird bei mehreren möglichen Bearbeitern von mehreren *InternalWorklist*s geteilt, was sich in mehreren Kennungen im Attribut `WorklistIDs` widerspiegelt. Am *InternalWorklistItem* sind auch benutzerspezifische Attribute eines Arbeitslisteneintrags hinterlegt (*IndividualWorklistItemSettings*). Dies erlaubt den Benutzern an Arbeitslisteneinträgen einen eigenen Titel (`IndividualTitle`), eine eigene Beschreibung (`IndividualDescription`) sowie eigene Eskalationsinformationen (`IndividualDueDate`, `IndividualPriority`, `IndividualEscalation`, `IndividualEscalationHandlingProcedureID`) zu hinterlegen. Da ein *InternalWorklistItem* von allen möglichen Bearbeitern geteilt wird, bildet das entsprechende Attribut (`IndividualWorklistItemSettings` in *InternalWorklistItem*) von Arbeitslisten-Kennungen auf die individuellen Einstellungen und Attribute ab.

Im Gegensatz zu *WorklistItem* sind beim *InternalWorklistItem* nicht nur die aktuelle Rückfrage und/oder Antwort sowie die aktuelle Delegation hinterlegt, sondern auch alle erfolgten Rückfragen (`EnquiryStack`) bzw. Delegationen (`DelegationHistory`). Außerdem ist die Bearbeiterzuordnung des entsprechenden Prozessschritts sowie der *ParameterDataContext* hinterlegt, mit dem wie beim Ausführungsmodell beschrieben (s. Abschnitt 7.2.4), Datenelementwerte in der Arbeitsliste angezeigt werden können. Da die Verfahren für Eskalation, für die Verteilung neu aktivierter Prozessschritte auf Arbeitslisten (*Distribution*) sowie für die Bestimmung möglicher Delegationsempfänger im *WorklistManager* nicht statisch vorgegeben, sondern dynamisch als Zusatzmodule behandelt werden, sind im *InternalWorklistItem* die Namen der für den Arbeitslisteneintrag zu verwendenden *Plugins* hinterlegt. Bei Anwendung der entsprechenden Funktion (z. B. Eskalation) auf den Arbeitslisteneintrag wird das entsprechende Zusatzmodul gerufen.

ClientWorklistItem ist ein Eintrag in einer *ClientWorklist*. Logisch gehört es zu einem *InternalWorklistItem*, allerdings hat jede *ClientWorklist* eigene *ClientWorklistItem*. Deshalb entsprechen die Attribute von *ClientWorklistItem* weitgehend den Attributen von *InternalWorklistItem*. Allerdings gibt es in *ClientWorklistItem* keine Möglichkeit, diese Attribute zu ändern. Stattdessen können über *IndividualWorklistItemSettings* benutzerspezifische Werte gesetzt werden. Diese gelten nur für einen Benutzer – den Besitzer der jeweiligen *ClientWorklist*. Sie werden aber auf das entsprechende *InternalWorklistItem* propagiert und gelten dann für alle *ClientWorklist*-Objekte des Benutzers.

7.2.7 Organisationsmodell

Im Organisationsmodell werden die für den Einsatz des Systems notwendigen Organisationsstrukturen repräsentiert, etwa Benutzer, Abteilungen und Projektgruppen [Ber05, RMR09, RMR08]. Daneben sind dort auch die technischen Aspekte hinterlegt, die für den Betrieb des Gesamtsystems notwendig sind. Hierzu zählen Passwörter zur Anmeldung am PrMS, die bevorzugte Sprache eines Benutzers, seine Mailadresse sowie Abwesenheitsinformationen.

Die gemeinsame Hinterlegung dieser Aspekte an einer zentralen Stelle vereinfacht die Benutzerverwaltung und vermeidet Redundanz, da nicht immer strikt zwischen organisatorischen und technischen Aspekten unterschieden werden kann, etwa bei der Mailadresse. Außerdem erlaubt es die zentrale Verwaltung, das Organisationsmodell nicht nur für Funktionen des Prozessmanagements zu verwenden (z. B. die Zuweisung von Prozessschritten an die entsprechenden Bearbeiter), sondern als unternehmensweite Quelle für Organisationsdaten. Dazu muss das Organisationsmetamodell umfassend genug sein, um alle in der Praxis relevanten Aspekte abzubilden. Wir stellen dieses Metamodell im folgenden vor und verzichten dabei auf eine Beschreibung der Schnittstelle des Organisationsmodells wie bei den vorangegangenen Modellen. Die Schnittstelle selbst ist allgemeingültig und beruht auf Objekten für Entitäten und Relationen, wobei diese beliebige Attribute besitzen. Verwendet wird das komplette Organisationsmodell nur bei der dazugehörigen Verwaltung (*OrgModelManager*) und dem Editor (*OrgModelEditor*, s. Abb. 7.31). Die zwei wichtigsten Entitäten (*Mitarbeiter* und *Stelle*) werden jedoch von allen Diensten des Systems verwendet, da diese Entitäten den mit einem Dienst interagierenden Benutzer identifizieren.

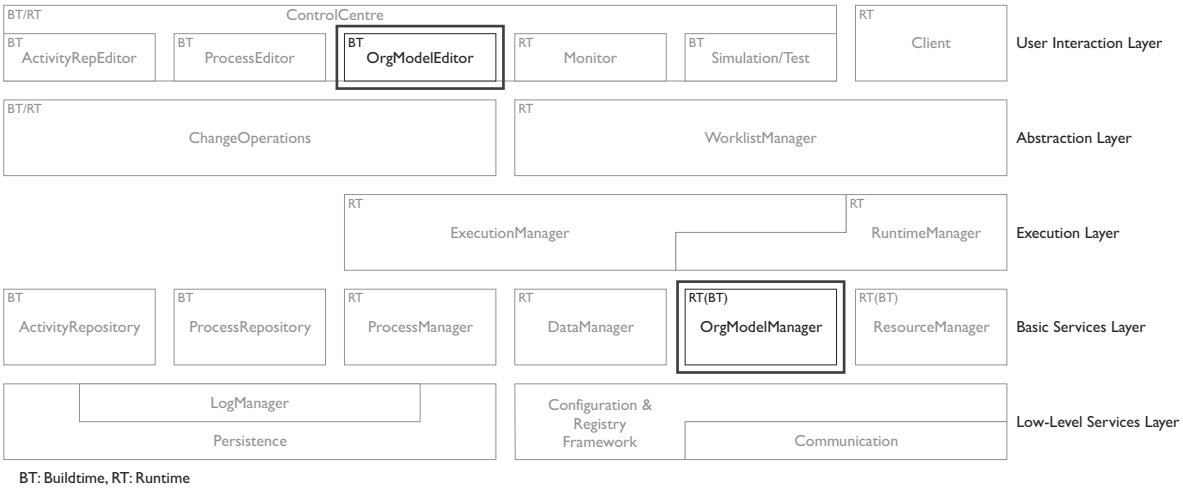


Abbildung 7.31: Verwendung des Organisationsmodells in der Gesamtarchitektur

7.2.7.1 Entitäten des Organisationsmodells

Abb. 7.32 zeigt die Entitäten des Organisationsmetamodells und deren Beziehungen untereinander. Jede Entität hat vorgegebene Attribute, etwa eine eindeutige Kennung (ID) und ein (innerhalb des Entitätstyps) eindeutiger Name. Außerdem kann der Modellierer des Organisationsmodells beliebige andere Attribute für jeden Entitätstyp festlegen.

Benutzer des Systems werden durch die Entität *Mitarbeiter* repräsentiert. Diese Entität hat deshalb neben den bereits aufgeführten weitere Attribute. Dazu gehören das Passwort für die Benutzeranmeldung am System sowie die Mailadresse, über die das System entsprechende Nachrichten an den jeweiligen Mitarbeiter schicken kann. Da auch Maschinen oder Rechnerprozesse mit dem System interagieren können, werden auch diese als Mitarbeiter im Organisationsmodell hinterlegt. Das heißt, Mitarbeiter müssen aus Sicht des Organisationsmodells nicht unbedingt natürliche Personen sein.

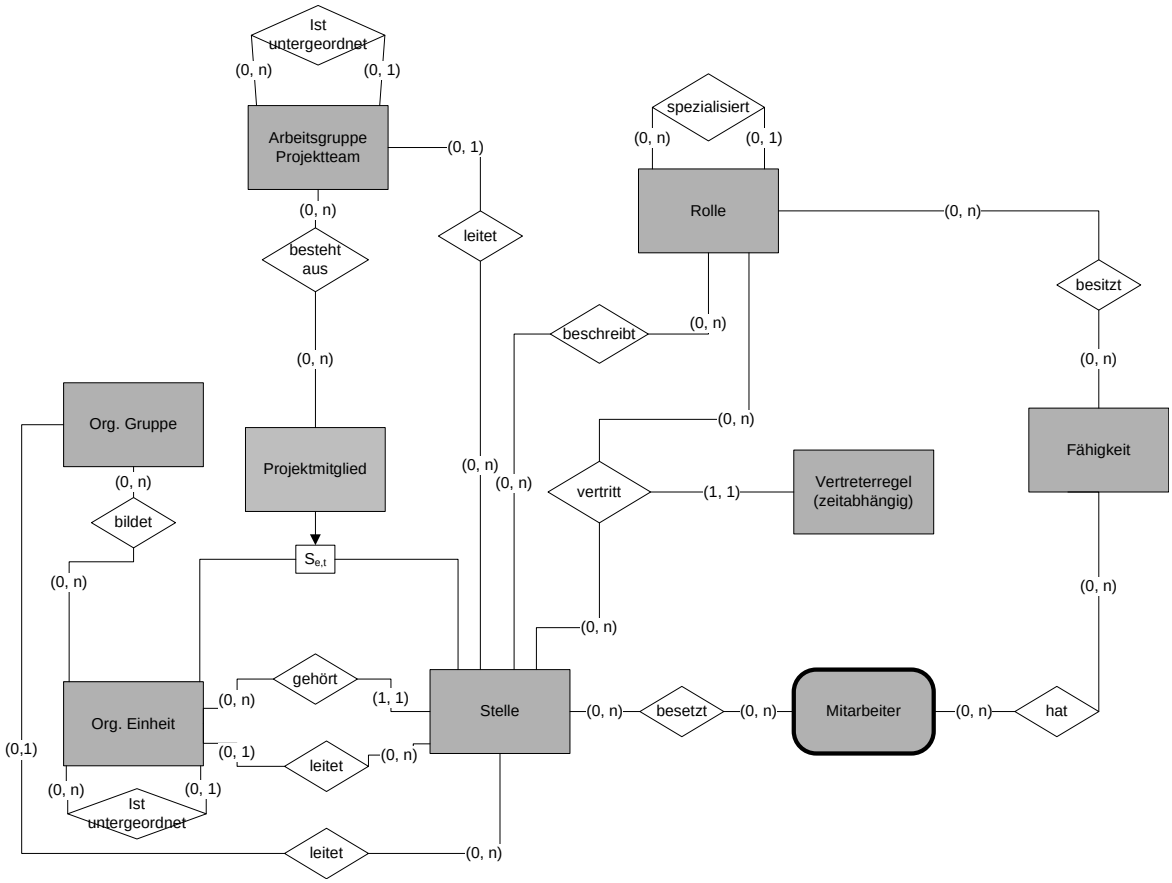


Abbildung 7.32: ER-Diagramm des Organisationsmetamodells [Ber05]

Ein Mitarbeiter besetzt eine oder mehrere *Stellen* (z. B. „Sachbearbeiter“ oder „Sekretär“), wobei eine Stelle auch durch einen oder mehrere Mitarbeiter besetzt sein kann (bei Schichtarbeit oder Arbeitsplatzteilung). Dadurch wird vom konkreten Aufgabenträger (*Mitarbeiter*) abstrahiert, was die Zuordnungen von Prozessschritten zu Bearbeitern erheblich verbessert. So müssen beispielsweise Bearbeiterzuordnungen bei Beförderungen nicht angepasst werden. Über die Stelle wird ein Mitarbeiter in die abstrakten Entitäten der Organisationsstruktur eingebettet. Hierzu gehören hierarchisch strukturierte *Organisationseinheiten*, d. h. eine Organisationseinheit kann mehrere untergeordnete sowie eine übergeordnete Organisationseinheit haben (z. B. Abteilung „Buchhaltung“ und übergeordneter Fachbereich „Rechnungswesen“).

Jeder Organisationseinheit ist eine leitende Stelle zugeordnet (z. B. „Abteilungsleiter“). Umgekehrt ist jede Stelle eindeutig einer Organisationseinheit zugeordnet, wodurch sich implizit zu einer Stelle auch die entsprechende vorgesetzte Stelle ergibt. Dadurch lassen sich bei der Prozessausführung abhängige Bearbeiterzuordnungen wie „Vorgesetzter des Bearbeiters des Vorgängerschritts“ auswerten. Allerdings erfordert eine solche Bearbeiterzuordnung Kenntnis der Stelle des Bearbeiters des Vorgängerschritts – ein Bearbeiter kann ja mehrere Stellen besetzen und damit auch verschiedene Vorgesetzte haben. Aus diesem Grund wird ein Bearbeiter im PrMS immer durch einen Mitarbeiter-Stelle-Paar (*QualifiedAgent*) identifiziert und nicht durch Mitarbeiter alleine. Dies betrifft beispielsweise die Prozessvorlagen- und Prozessinstanzverantwortlichen (`SupervisorAgent` in *Template* und *Instance* im Prozessmodell in Abschnitt 7.2.3, s. Abb. 7.13) sowie die Besitzer von Arbeitslisten (`Agent` in *Worklist* im Arbeitslistenmodell in Abschnitt 7.2.6, s. Abb. 7.28).

Organisationsgruppen sind Organisationseinheiten ähnlich, sie bedürfen jedoch keiner hierarchischen Struktur. Damit lässt sich beispielsweise die dauerhafte interdisziplinäre Zusammenarbeit von Organisationseinheiten repräsentieren. Dies ist vergleichbar mit *Arbeitsgruppen*, die jedoch in der Regel zeitlich beschränkt sind. Sowohl Organisations- als auch Arbeitsgruppen besitzen eine leitende Stelle, wodurch sich weitere Vorgesetztenbeziehungen ergeben, etwa zur Unterscheidung von disziplinarischen und fachlichen Vorgesetzten. Arbeitsgruppen bestehen aus Stellen, Organisationseinheiten und/oder untergeordneten Arbeitsgruppen.

Neben den Entitäten für Organisationsstrukturen gibt es noch Entitäten zur Repräsentation von Fähigkeiten. Jeder Mitarbeiter kann beliebige Fähigkeiten besitzen, beispielsweise die Beherrschung des Zehnfingersystems. Fähigkeiten können beliebig in *Rollen* gruppiert werden, die wie *Organisations-* und *Arbeitsgruppen* hierarchisch strukturiert sein können. Rollen wiederum beschreiben Stellen genauer. An diesen beiden Entitätstypen ist auch die *Vertreterregel* verankert. Damit wird im Modell hinterlegt, welche Stelle eine andere Stelle bezüglich einer oder mehrerer Rollen vertritt. Da die Vertretung in der Regel zeitlich beschränkt ist, besitzt die entsprechende Entität zusätzlich Attribute für die Beschreibung der zeitlichen Gültigkeit.

Beispiel 7.14 (Rolle)
Ein Beispiel für eine Rolle ist ein „EDV-Ansprechpartner“. Diesen gibt es in jeder Abteilung (Organisationseinheit).

Beispiel 7.15 (Vertretung bezüglich einer Rolle)
Der Sekretär (Stelle) der Finanzabteilung (Organisationseinheit) ist für die Verteilung der Abteilungspost zuständig (Rolle). Bei seiner Abwesenheit, übernimmt der Buchhalter (Stelle) der Abteilung die Postverteilung und vertritt damit den Sekretär in dieser Rolle.

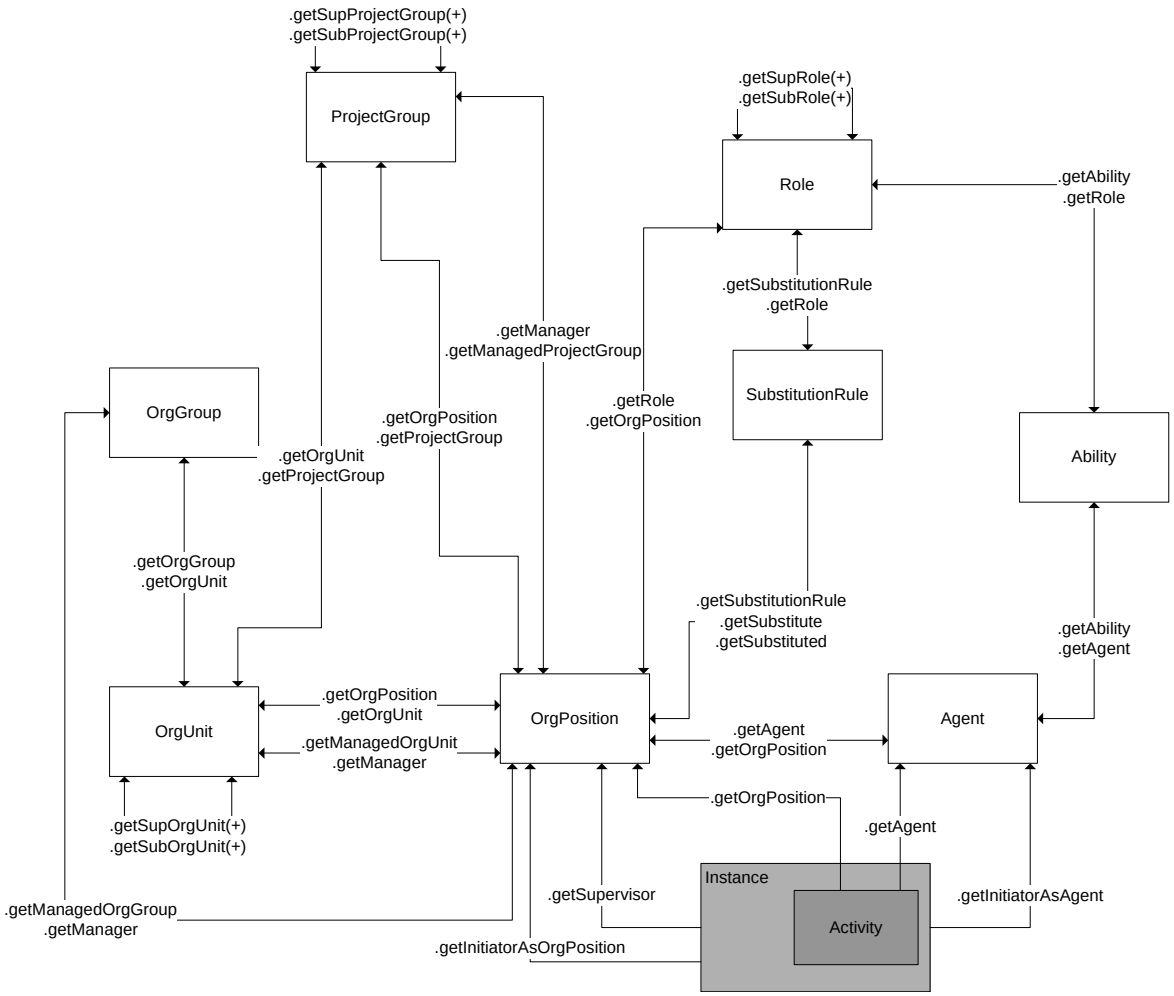


Abbildung 7.33: Navigierender Zugriff auf das Organisationsmodell [Ber05]

7.2.7.2 Anfragen an das Organisationsmodell

Trotz des Umfangs des Organisationsmodells sollen sich zugehörige Anfragen möglichst einfach ausdrücken lassen. Dies ist im relationalen Modell, das dem Organisationsmetamodell zugrunde liegt, jedoch nicht direkt gegeben. Insbesondere die hierarchischen Beziehungen lassen sich nur schwer abbilden. Anfragen an das ADEPT-Organisationsmodell beruhen deshalb auf expliziter Navigation. Durch diese Pfadausdrücke lassen sich insbesondere komplexe Anfragen einfach darstellen [Ber05].

Für die Navigation sind die Beziehungstypen zwischen den einzelnen Entitätstypen mit sprechenden Namen versehen (s. Abb. 7.33). Eine Anfrage beginnt immer mit einem Entitätstyp und ggf. einer oder mehrere Einschränkungen auf Attributwerten, z. B. `OrgUnit(Name='Buchhaltung')`.

Von diesem Entitätstyp ausgehend kann sich die Anfrage dann über Relationen und beliebige Entitätstypen erstrecken. So liefert `OrgUnit(Name='Buchhaltung').getOrgPosition.getAgent` alle Mitarbeiter der Buchhaltungsabteilung. Sämtliche Entitäten auf dem Weg können weiter eingeschränkt werden, etwa `OrgUnit(Name='Buchhaltung').getOrgPosition(Name=`

`'Sekretär').getAgent` um alle Mitarbeiter der Buchhaltungsabteilung zu erhalten, die die Stelle „Sekretär“ innehaben. Dabei können Entitäts- und Relationstypen beliebig oft in einer Anfrage vorkommen. Beispielsweise liefert `Agent(Name='Müller').getOrgPosition.getOrgUnit.getSupOrgUnit.getSupOrgUnit.getManager.getAgent` den Mitarbeiter, der Vorgesetzter des Mitarbeiters „Müller“ über drei Hierarchiestufen ist. Auch hier sind wieder beliebige Einschränkungen auf allen Attributen der betroffenen Entitäten möglich. Allerdings ist eine Einschränkung auf `getSupOrgUnit`, wie in diesem Beispiel, nicht sinnvoll, da eine Organisationseinheit ohnehin nur einer Organisationseinheit untergeordnet ist.

Für die Verwendung im Prozessmanagement, etwa für eine Bearbeiterzuordnung, die einen Prozessschritt dem entsprechenden Bearbeiter zuweist, reichen diese statischen Anfragen jedoch nicht für alle Fälle aus. Hier wird zusätzlich im Modell noch der Prozesskontext benötigt. Dies ist in Abb. 7.33 rechts unten ersichtlich. Es können der Bearbeiter eines Knotens (`Activity.getAgent`), der Mitarbeiter, der eine Prozessinstanz gestartet hat, (`Instance.getAgent`) und der Prozessverantwortliche (`Instance.getAgent`) sowie die jeweilige Stelle (`getOrgPosition`) innerhalb einer Bearbeiterzuordnung referenziert werden. Dies bildet dann eine abhängige Bearbeiterzuordnung, deren Abhängigkeiten zur Laufzeit aufgelöst werden, wie im Ausführungsmodell (s. Abschnitt 7.2.4) beschrieben. Nach Ersetzung der Abhängigkeiten ist die Bearbeiterzuordnung unabhängig und kann dementsprechend vom *OrgModelManager* rein über das Organisationsmodell aufgelöst werden.

7.3 Dienste

Aufbauend auf der Gesamtarchitektur (s. Abschnitt 7.1) und den Datenmodellen (s. Abschnitt 7.2) gehen wir in den folgenden Abschnitten detailliert auf die einzelnen Dienste eines adaptiven PrMS ein. Während die Datenmodelle die zwischen den Diensten ausgetauschten Informationen repräsentieren, stellen die Dienste selbst die Anwendungslogik des adaptiven PrMS bereit. Diese ist nach funktionalen Gesichtspunkten unterteilt. Dementsprechend lassen sich auch die funktionalen Anforderungen auf die verschiedenen Dienste verteilen. Die Verteilung erfolgt mit besonderer Beachtung der nicht-funktionalen Anforderung NA 5 (Unabhängigkeit/Modularität). So sind die Funktionen so auf die Dienste verteilt, dass es möglichst wenig Abhängigkeiten zwischen den Diensten gibt (Kohäsion).

Um die Unabhängigkeit weiter zu erhöhen, stellen alle Dienste ihre Funktionen über Schnittstellen (Java-Interfaces) bereit. Auf diese Weise kann die Implementierung der Dienste einfach ausgetauscht werden. Die Funktionen eines Dienstes unterteilen sich gewöhnlich aufgrund ihrer großen Anzahl auf mehrere Unterdienste. Dies führt dazu, dass die einzelnen Unterdienste nicht mehr als 10 Methoden besitzen und jeder Dienst ungefähr 5 Unterdienste hat. Außerdem werden die Abhängigkeiten nicht von den Diensten selbst, sondern vom *Configuration & Registry Framework* (s. Abschnitt 7.3.10.1) aufgelöst. Abhängig sind zudem Diensttypen und nicht konkrete Dienstinstanzen oder spezielle Dienstimplementierungen. Dadurch wird die weitestgehende (interne) Unabhängigkeit der Dienste erreicht.

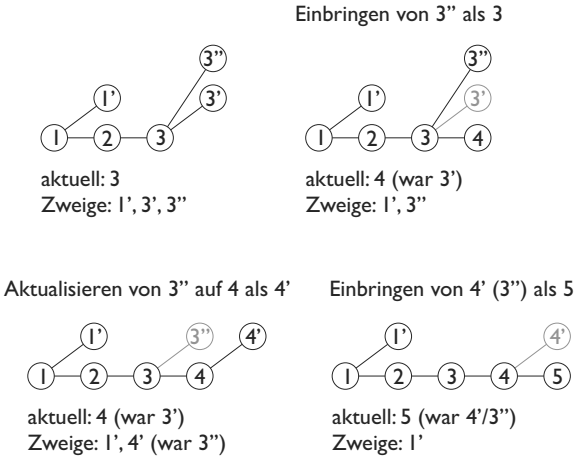


Abbildung 7.34: Versionierung im Aktivitätenrepository

Beispiel 7.16 (Unabhängigkeit der Dienste)
Die Implementierung eines *ExecutionManager* benötigt Informationen über ausgeführte Prozessinstanzen. Diese stellt ein *ProcessManager* über eine definierte Schnittstelle bereit. Der *ExecutionManager* greift nur über diese Schnittstelle auf den *ProcessManager* zu. Dadurch kann die Implementierung des *ProcessManager* ohne Änderungen am *ExecutionManager* ausgetauscht werden. So können verschiedene *ProcessManager*-Implementierungen ihre Daten (Prozessvorlagen und -instanzen) auf verschiedene Arten (z. B. mittels SQL oder XML) speichern.

Die Implementierung eines *ExecutionManager* kann über das *Configuration & Registry Framework* auf eine *ProcessManager*-Implementierung zugreifen. Dazu fordert der *ExecutionManager* aber keine konkrete *ProcessManager*-Implementierung an (Dienstinstanz) sondern den Diensttyp *ProcessManager*. Welche Implementierung bereitgestellt wird, entscheidet das *Configuration & Registry Framework*.

7.3.1 Verwaltung von Aktivitätenvorlagen (*ActivityRepository*)

Das *ActivityRepository* verwaltet das Aktivitätenmodell (s. Abschnitt 7.2.2) und setzt somit einen Teil der funktionalen Anforderung FA 10 (*Plug and Play*) um. Wie beim Aktivitätenmodell beschrieben, wären Aktivitätenvorlagen für *Plug and Play* bereits ausreichend, allerdings würde dies der nicht-funktionalen Anforderung nach Wiederverwendung und Redundanzvermeidung widersprechen.

Das Anlegen, Ändern und Löschen von Aktivitätenmodell-Entitäten im *ActivityRepository* erfolgt innerhalb eines einfachen Lebenszyklus: die Entitäten können in Entwicklung, produktiv oder veraltet sein – Löschen findet nur logisch statt und wird durch veraltet repräsentiert. Jede Änderung von Daten des Aktivitätenmodells findet im Rahmen einer Änderungstransaktion statt und führt zu einem neuen Stand (`Revision`) der betroffenen Entität und ggf. auch ihrer untergeordneten Entitäten. Die Stände der Daten spiegeln sich in globalen Ständen des *ActivityRepository* wider. Jede Änderungstransaktion führt zu einem neuen Stand (*Revision*) des *ActivityRepository*, vergleichbar mit der Versionsverwaltung Subversion [svn].

Im Gegensatz zu Subversion sind die Stände im *ActivityRepository* linear, d. h. es gibt nur einen aktuellen Stand (*Head*, s. Abb. 7.34). Mehrere gültige Stände entstehen in Subversion

dadurch, dass parallel Änderungen durchgeführt werden. Werden diese Änderungen unabhängig voneinander eingebracht und die jeweiligen Stände unabhängig voneinander weiterentwickelt, also weiteren Änderungen unterworfen, so entsteht jeweils ein neuer Zweig (*Branch*). Diese Zweige sind unabhängig voneinander, und sie sind alle gültig. Im *ActivityRepository* ist das Einbringen unabhängiger Änderungen nicht zulässig, so dass es keine Aufspaltung in unabhängige Zweige geben kann.

Beispiel 7.17 (Zweige durch parallele Änderungen)
Auf dem aktuellen Stand 3 (*Revision*) im *ActivityRepository* werden zwei Änderungstransaktionen gestartet. Diese führen zu den Zweigen $3'$ und $3''$ (s. Abb. 7.34 links oben). Beide Zweige gelten zu diesem Zeitpunkt noch nicht, da die Änderungstransaktionen noch nicht abgeschlossen sind. Wird die Änderungstransaktion von Zweig $3'$ eingebracht, führt dies zur Stand 4 im *ActivityRepository* (s. Abb. 7.34 rechts oben). Danach ist Stand 4 der aktuelle Stand, alle vorherigen Stände sind veraltet. Gibt es in der Änderungstransaktion von Zweig $3''$ konkurrierende Änderungen zu der Änderungstransaktion, die zu Stand 4 geführt hat, kann die Änderungstransaktion von Zweig $3''$ nicht ins *ActivityRepository* übernommen werden.

Die Linearität der Stände im *ActivityRepository* führt zu eindeutigen aktuellen Ständen und vereinfacht so die Prozessmodellierung erheblich. Trotzdem sind parallele Änderungen am Aktivitätenmodell möglich, so dass es keine wesentlichen Einschränkungen gegenüber nicht-eindeutigen aktuellen Ständen wie in Subversion gibt. Es kann jederzeit beliebig viele offene Änderungstransaktionen geben. Beim Start einer Änderungstransaktion wird jeweils ein neuer Zweig angelegt, der eine logische Kopie des aktuellen Stands ist. Alle Änderungen einer Änderungstransaktion finden auf dieser Kopie statt. Die Zweige im *ActivityRepository* entsprechen damit den Arbeitskopien von Subversion.

Durch Anlegen neuer Zweige beeinflussen sich parallele Änderungstransaktionen nicht. Beim Beenden einer Änderungstransaktion werden alle ihre Änderungen auf dem zu diesem Zeitpunkt gültigen aktuellen Stand eingebracht. Ist dieser Stand beispielsweise derselbe, auf der der Zweig der Änderungstransaktion beruht, so können die Änderungen direkt übernommen werden. Wurden seit Erzeugung des Zweigs allerdings andere Änderungstransaktionen beendet, muss der Zweig erst mit dem aktuellen Stand und damit mit den zwischenzeitlich eingebrachten Änderungen abgeglichen werden. Weisen diese Änderungen Konflikte mit der Änderungstransaktion auf, so müssen diese behoben werden. Solche Konflikte entstehen etwa dadurch, dass die eingebrachten Änderungen dieselben Objekte geändert haben wie die Änderungstransaktion, aber auf eine andere Weise.

Beispiel 7.18 (Einbringen paralleler Änderungen im `ActivityRepository`)
Vor Beginn einer Änderungstransaktion im *ActivityRepository* wird ein neuer Zweig (*Branch*) angelegt (s. Zweig $3''$ in 7.34, links oben). Dieser entspricht logisch einer Kopie des aktuellen Stands 3 (*Revision*). Bevor die Änderungstransaktion beendet und die Änderungen von Zweig $3''$ eingebracht werden, wird überprüft, ob sich der aktuelle Stand durch parallele Änderungen, etwa Zweig $3'$, zwischenzeitlich geändert hat. Ist dies der Fall, muss der Zweig $3''$ um die Änderungen von Stand 4 (ehemals Zweig $3'$) aktualisiert werden. Dies umfasst auch die Auflösung der ggf. auftretenden Konflikte zwischen den Zweigen $3'$ und $3''$. Dadurch wird der Zweig $3''$ zu Zweig $4'$, d. h. diesem Zweig liegt nicht mehr Stand 3 zugrunde sondern Stand 4. Nun können die Änderungstransaktion beendet und die Änderungen von Zweig $4'$ eingebracht werden. Damit wird Zweig $4'$ (ehemals Zweig $3''$) zum neuen aktuellen Stand 5.

ActivityRepository

FA 10 *Plug and Play* (Anwendungsintegration bei der Modellierung)

Angebotene Funktionen

- Versionierte Verwaltung aller Aktivitätenvorlagen und verwandten Entitäten (Anlegen, Ändern, als veraltet markieren)
- Einhaltung des Lebenszyklus von Aktivitätenvorlagen und verwandten Entitäten
- Bereitstellung und Verwaltung von Metainformationen zu Aktivitätenvorlagen für deren Strukturierung nach beliebigen Aspekten
- Suche nach Aktivitätenvorlagen beruhend auf Metainformationen

Neben den Daten des Aktivitätenmodells verwaltet das *ActivityRepository* zusätzliche Metainformationen zu den einzelnen Entitäten. Hierzu gehören etwa Stichworte für die fachliche Verwendung der Entitäten. Einerseits kann das *ActivityRepository* nach diesen Metainformationen durchsucht werden, ebenso wie nach Attributwerten von Entitäten. Andererseits erlauben die Metainformationen, die Entitäten des Aktivitätenmodells im *ActivityRepository* nach verschiedenen Aspekten zu strukturieren, z. B. alle Aktivitätenvorlagen zur Rechnungserstellung.

7.3.2 Verwaltung von Prozessvorlagen (*ProcessRepository*)

Das *ProcessRepository* erfüllt die funktionale Anforderung FA 11 (Prozessverwaltung und -versionierung). Es verwaltet alle erstellten Prozessvorlagen und deren Beziehungen untereinander, d. h. Informationen dazu, welche Prozessvorlage aus welcher anderen Prozessvorlage hervorgegangen ist (s. Abb. 7.35). Miteinander verwandte Prozessvorlagen bilden einen Versionsbaum, der einen Prozesstyp darstellt. Jeder Prozessbaum hat implizit die Nullvorlage als Wurzel, also eine Prozessvorlage mit nur einem Start- und einem Endknoten.

Das *ProcessRepository* kennt die Unterschiede zwischen verwandten Prozessvorlagen (beispielsweise in Form von Änderungshistorien) und kann hieraus die für Schemaevolution (FA 9.2) notwendigen Deltas für Prozessvorlagen (s. Abschnitt 5.1 und 7.2.3.4) erzeugen. Dabei reicht es aus, nur die Änderungen oder die Änderungshistorie relativ zur jeweiligen Elternprozessvorlage zu speichern. Mit dieser Information lassen sich durch Vereinigung der entsprechenden Änderungen die Unterschiede zweier beliebiger Prozessvorlagen rekonstruieren, sofern beide Vorlagen transitiv in einer Eltern-Kind-Beziehung stehen. Trivialerweise geht dies auch bis zur Nullvorlage, womit sich eine vollständige Änderungshistorie für eine Prozessvorlage ergibt.

7.3.2.1 Lebenszyklus von Prozessvorlagen

Das *ProcessRepository* erlaubt das Anlegen neuer Prozesstypen und neuer Prozessvorlagen, die von existierenden Prozessvorlagen abstammen (einschließlich der Nullvorlage). Es ermöglicht daher den vollständigen Lebenszyklus von Prozessvorlagen (s. Abb. 7.36). Der Lebenszyklus einer Prozessvorlage beginnt mit dem Ableiten als neues Kind einer bereits existierenden Prozessvorlage. Zu diesem Zeitpunkt entspricht die neue Prozessvorlage noch der Elternvorlage;

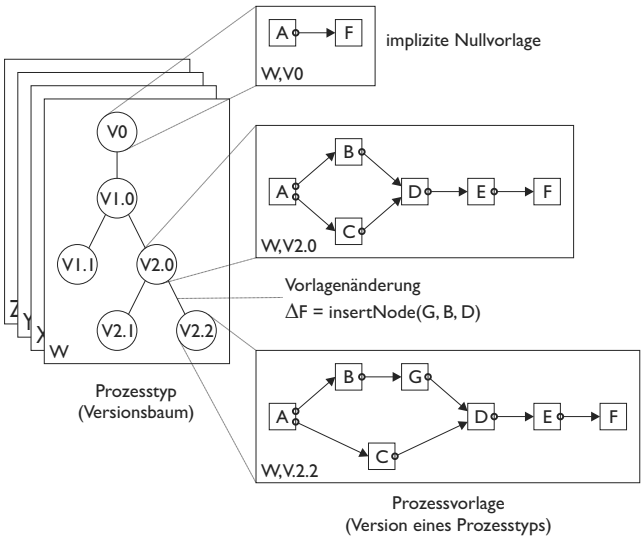


Abbildung 7.35: Daten der Prozessvorlagenverwaltung

sie befindet sich im Zustand `IN_CONSTRUCTION`. Während der Änderungen verbleibt sie in diesem Zustand. Sie wechselt in den Zustand `VALID_AND_TESTING`, wenn sie ausführbar ist, d. h. sie ist konsistent und korrekt bezüglich des Metamodells; beispielsweise gibt es keine Prozessschritte mit unversorgten Eingabeparametern (s. Abschnitt 2.2). Von diesem Zustand aus kann sie zur Instanzierung freigegeben werden (`ACTIVE`). Eine zur Instanzierung freigegebene Prozessvorlage hat im *ProcessRepository* gleichzeitig noch weitere Zustände (s. Abb. 7.36). So kann angegeben werden, ob sie als Basis für eine weitere neue Prozessvorlage verwendet werden kann (`DERIVABLE`/`UNDERIVABLE`), ob sie instanzierbar ist (`UNINSTANCEABLE`/`INSTANCEABLE`), ob laufende Prozessinstanzen einer Elternvorlage auf diese Vorlage migriert werden können (`INSTANCES_NOT_MIGRATABLE_TO`/`INSTANCES_MIGRATABLE_TO`) und ob Prozessinstanzen dieser Vorlage instanzspezifisch geändert werden können (`INSTANCES_UNCHANGEABLE`/`INSTANCES_CHANGEABLE`).

7.3.2.2 Verwendung von Prozessvorlagen als Subprozesse

Bei einer Prozessvorlage kann angegeben werden, ob sie als Subprozess in einer anderen Prozessvorlage verwendet werden darf oder nicht. Hier gilt die Einschränkung, dass Subprozesse nur in einer strikten Hierarchie erlaubt sind, d. h. es dürfen keine Zyklen innerhalb der Hierarchie auftreten [Rei00]. Die Unterscheidung zwischen eingebetteten und referenzierten Subprozessen spiegelt sich in den Zuständen `SU_COPY_REFERENCE` (beide Subprozessvarianten erlaubt), `SU_COPY` (Prozessvorlage darf nur als eingebetteter Subprozess verwendet werden, d. h. die Subprozessvorlage wird in die erstellte Prozessvorlage kopiert), `SU_REFERENCE` (nur als referenzierter Subprozess) und `NO_SU` (darf nicht als Subprozess verwendet werden) wider. Bei Verwendung als referenzierter Subprozess gelten auch wieder die für die Laufzeit relevanten Zustände. Da diese für die Verwendung als Subprozess unterschiedlich sein können, gibt es für Subprozesse separate Zustände (`SU_UNINSTANCEABLE`/`SU_INSTANCEABLE`, `SU_INSTANCES_NOT_MIGRATABLE_TO`/`SU_INSTANCES_MIGRATABLE_TO` und `SU_INSTANCES_UNCHANGEABLE`/`SU_INSTANCES_CHANGEABLE`).

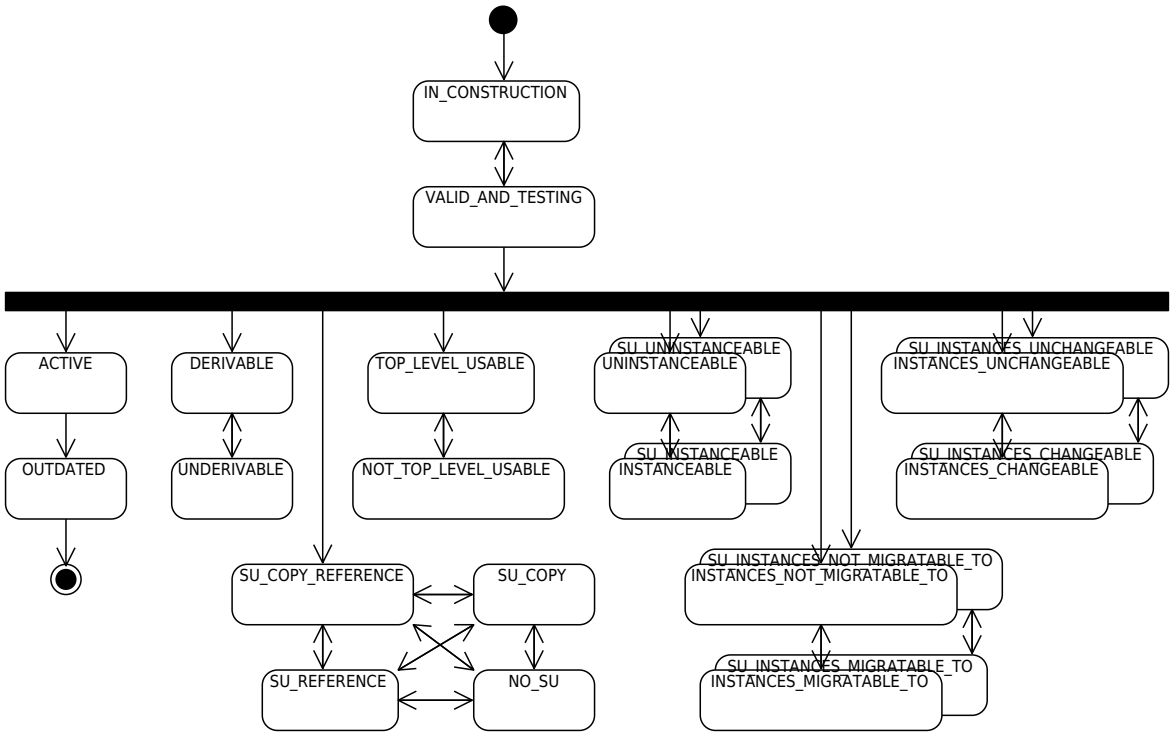


Abbildung 7.36: UML-Zustandsdiagramm des Lebenszyklus' einer Prozessvorlage

Soll eine Prozessvorlage nicht mehr verwendet werden, wird sie in den Zustand `OUTDATED` versetzt. Dies entspricht logisch ihrem Löschen. Allerdings darf die Prozessvorlage aus Gründen der Nachvollziehbarkeit nicht wirklich gelöscht werden. Dadurch entstünden Lücken im Versionsbaum. Die Nicht-Löschbarkeit von Prozessvorlagen ist auch der Grund für die Trennung zwischen *ProcessRepository* und *ProcessManager* (s. Abschnitt 7.3.4): Während das *ProcessRepository* alle jemals erstellten Prozessvorlagen kennt, sind im *ProcessManager* nur die aktuell verwendeten Prozessvorlagen vorhanden.

Für die einfachere Verwaltung von Prozessvorlagen erlaubt das *ProcessRepository* das gezielte Suchen nach Prozessvorlagen. Zur leichteren Auffindbarkeit können diese mit Schlagwörtern und weiteren Metainformationen versehen werden. Daneben kann auch der gesamte Datenbestand nach verwendeten Aktivitätenvorlagen durchsucht werden. Dies erleichtert es, im Falle von Änderungen an einer Aktivitätenvorlage alle betroffenen Prozessvorlagen zu finden und entsprechend anzupassen. dasselbe gilt auch für die Verwendung von Prozessvorlagen als Subprozesse.

7.3.3 Verwaltung des Organisationsmodells (*OrgModelManager*)

Der *OrgModelManager* setzt die funktionale Anforderung FA 3 (Organisationsmodell, Verwaltung und Auswertung von Bearbeiterzuordnungen) um. Dies erfolgt mittels der 5 Schnittstellen *PolicyResolution*, *ModelExplorer*, *ModelChangeOperations*, *ClientAdministration* und *SecurityManager*.

ProcessRepository

FA 9.2 Schemaevolution
FA 11 Prozessverwaltung und -versionierung

Angebotene Funktionen

- Versionierte Verwaltung aller Prozessvorlagen (Anlegen, Ändern, als veraltet markieren) und deren Beziehungen (Änderungen zwischen voneinander abgeleitete Prozessvorlagen)
- Einhaltung des Lebenszyklus von Prozessvorlagen
- Bereitstellung und Verwaltung von Metainformationen zu Prozessvorlagen für deren Strukturierung nach beliebigen Aspekten
- Suche nach Prozessvorlagen beruhend auf Metainformationen oder bestimmten Aktivitätenvorlagen sowie (Sub-)Prozessvorlagen

7.3.3.1 Auswertung von Bearbeiterzuordnungen

PolicyResolution dient in erster Linie zur Auflösung unabhängiger Bearbeiterzuordnungen auf Mengen von Mitarbeitern (FA 3.2); nur Mitarbeiter können Prozessschritte ausführen, aber keine abstrakten Entitäten des Organisationsmodells. Die Pfadausdrücke von Bearbeiterzuordnungen müssen dementsprechend beim Entitätstyp *Mitarbeiter* enden. Um dies zu vereinfachen, gibt es *Pfadvervollständigungen*. Diese werden über *PolicyResolution* verwaltet und spezifizieren für jeden Entitätstyp einen eindeutigen Pfad zum Entitätstyp *Mitarbeiter*. Dieser Pfad ergänzt implizit alle Bearbeiterzuordnung, deren Pfad nicht beim Entitätstyp *Mitarbeiter* enden.

Beispiel 7.19 (Pfadvervollständigung)
Die Pfadvervollständigung für Projektgruppen ist `getOrgPosition.getAgent`. Damit liefert die Anfrage `ProjectGroup(Name='P')` alle Mitarbeiter des Projekts „P“ zurück, da die Anfrage intern zu `ProjectGroup(Name='P').getOrgPosition.getAgent` vervollständigt wird.

Daneben überprüft *PolicyResolution* die Syntax von Bearbeiterzuordnungen, was bei Erstellung der Zuordnungen hilfreich ist. Abhängige Bearbeiterzuordnungen können nicht aufgelöst werden. Die Auflösung der Abhängigkeiten muss durch den Dienst erfolgen, der *PolicyResolution* verwendet, beispielsweise *ExecutionManager* (s. Abschnitt 7.3.6). Dies vereinfacht die Implementierung von *PolicyResolution* und verhindert Abhängigkeiten zu anderen Diensten und deren Datenmodelle.

7.3.3.2 Zugriff und Verwaltung des Organisationsmodells

Einen direkten navigierenden Zugriff auf das Organisationsmodell bietet die Schnittstelle *ModelExplorer*. Sie erfordert keine vorab definierten Pfadausdrücke, sondern bietet zur Laufzeit sämtliche im Modell gespeicherten Informationen, beispielsweise vorhandene Entitäten und deren Attribute. Eng verwandt damit ist die Schnittstelle *ModelChangeOperations* mittels der Entitäten, Beziehungen und Attribute angelegt, geändert und gelöscht werden können. Aufgrund der Erweiterbarkeit des Modells und zwecks Vereinfachung der Schnittstelle beruhen

sowohl *ModelExplorer* als auch *ModelChangeOperations* auf dem relationalen Modell [Cod70]. Das heißt, sie verwenden Entitätstypen, Relationstypen und Attributwerte und nicht die Entitäten aus dem Organisationsmetamodell (s. Abschnitt 7.2.7). Vom Endbenutzer häufig benötigte Funktionen dieser beiden Schnittstellen, wie das Lesen und Schreiben der Mailadresse und der bevorzugten Sprache sowie das Ändern des Passworts, kapselt die Schnittstelle *ClientAdministration*.

7.3.3.3 Authentisierung, Authentifizierung und Autorisierung

Dem *OrgModelManager* zugehörig, aber aufgrund seiner Bedeutung weitgehend eigenständig, ist der *SecurityManager*. Über diesen melden sich Benutzer am System an (Authentisierung). Die Schnittstelle ermöglicht verschiedene Authentisierungsarten, z. B. Passworteingabe oder die zertifikatsbasierte Authentisierung. Intern wird die Authentisierung entweder direkt mit den im Organisationsmodell hinterlegten Informationen vorgenommen, oder sie kann an JAAS (*Java Authentication and Authorization Service* [jaa]) delegiert werden, wodurch auch die üblichen Authentisierungsprotokolle (z. B. LDAP, Kerberos [NYHR05] oder PAM (*Pluggable Authentication Modules* [Smø03])) auf externen Daten verwendet werden können. Bei erfolgreicher Authentisierung gibt der *SecurityManager* ein benutzerspezifisches *ClientSessionFactory*- oder *SessionFactory*-Objekt zurück und authentifiziert so Benutzer. Authentifizierung bedeutet, dass der *SecurityManager* gegenüber den anderen Diensten des PrMS die Identität des Benutzers nachweisen kann. Dies erfolgt über die zurückgegebenen Objekte *ClientSessionFactory* und *SessionFactory*. Mit den zurückgegebenen *ClientSessionFactory*- und *SessionFactory*-Objekten werden leichtgewichtige *SessionToken* erzeugt. Solche *SessionToken* müssen bei allen Aufrufen von Funktionen des PrMS mit übergeben werden. Sie identifizieren den Benutzer und erlauben die Überprüfung, ob der Benutzer berechtigt ist, die jeweilige Funktion aufzurufen (Autorisierung).

Beispiel 7.20 (Anmeldung am *SecurityManager* und Berechtigungsüberprüfung) Ein Benutzer meldet sich mit seinem Benutzernamen und seinem Passwort am *SecurityManager* an. Die Anmeldung (Authentisierung) ist erfolgreich, und der *SecurityManager* gibt ein *ClientSessionFactory*-Objekt zurück (Authentifizierung). Anschließend erzeugt sich der Benutzer über das *ClientSessionFactory*-Objekt ein *SessionToken*. Dieses übergibt er beim Start eines Prozesse. Mit dem *SessionToken* wird überprüft, ob der Benutzer berechtigt ist, den gewünschten Prozess zu starten (Autorisierung).

Neben der Verwendung von LDAP zur Authentisierung können die durch LDAP bereitgestellten Informationen auch direkt in den Datenbestand des *OrgModelManager* integriert werden. Dazu werden die Daten auf Anforderung oder regelmäßig automatisch mit einem existierenden LDAP-Verzeichnis abgeglichen. Für jeden Entitätstyp des Organisationsmodells können dazu beliebige LDAP-Anfragen konfiguriert werden. Die Entitäten und deren Attribute aus den Ergebnismengen der LDAP-Anfragen werden dann mit den entsprechenden Entitäten im *OrgModelManager* synchronisiert.

7.3.4 Verwaltung laufender Prozesse (*ProcessManager*)

Der *ProcessManager* verwaltet die Daten laufender Prozesse. Dazu gehören die sich in Verwendung befindliche Prozessvorlagen, laufende und abgeschlossene Prozessinstanzen sowie deren Zustände und, wo vorhanden, Informationen zu instanzspezifischen Änderungen (s. Abb. 7.37).

OrgModelManager

FA 3	Organisationsmodell
FA 3.1	Organisationsmodellverwaltung
FA 3.2	Auswertung von Bearbeiterzuordnungen

Angebotene Funktionen

- Versionierte Verwaltung des Organisationsmodells mit allen Entitäten und deren Attribute (*ModelExplorer*, *ModelChangeOperations*)
- Auflösung von (unabhängigen) Bearbeiterzuordnungsregeln (*PolicyResolution*)
- Anmeldung von Benutzern am System (*SecurityManager*)
- Verwaltung benutzerspezifischer Daten wie Passwörter oder Abwesenheitsinformationen (*ClientAdministration*)

Die in einem Prozess erzeugten Daten, d. h. die Werte und Historien von Datenelementen, werden nicht im *ProcessManager,* sondern separat im *DataManager* (s. Abschnitt 7.3.5) abgelegt. Somit erfüllt der *ProcessManager* die funktionale Anforderung FA 6 (persistente Speicherung) sowie jeweils Teile der funktionalen Anforderungen FA 1 (Prozessausführung), FA 9.1 (Änderungen an laufenden Prozessen) und FA 9.2 (Schemaevolution).

7.3.4.1 Prozessvorlagen

Im Gegensatz zum *ProcessRepository* (s. Abschnitt 7.3.2) verwaltet der *ProcessManager* nur die Prozessvorlagen, die zu einem Zeitpunkt verwendet/instanziert werden können, sowie die Prozessvorlagen zu denen es noch (abgeschlossene) Prozessinstanzen im *ProcessManager* gibt. Ältere und sich in Entwicklung befindliche Prozessvorlagen werden vom *ProcessRepository* verwaltet. Da dort auch die Beziehungen zwischen den Prozessvorlagen, genauer den einzelnen Versionen eines Prozesstyps, bekannt sind, können die Prozessvorlagen im *ProcessManager* als voneinander unabhängig betrachtet werden. Dies vereinfacht ihre Verwaltung erheblich. Allerdings werden ausführungsrelevante Zustände einer Prozessvorlage sowohl im *ProcessRepository* als auch im *ProcessManager* benötigt. Dazu gehören Verwendungsinformationen, etwa ob eine Prozessvorlage instanzierbar ist, ob ihre Prozessinstanzen zu einer anderen Prozessvorlage migriert werden können oder ob ihre Prozessinstanzen geändert werden dürfen (s. Abb. 7.36). Auch im *ProcessManager* unterscheidet man zwischen normaler Verwendung und der Verwendung als Subprozess. Diese Informationen können entweder bei Bedarf aus dem *ProcessRepository* geholt werden, oder sie werden redundant im *ProcessManager* abgelegt, dann muss allerdings die Konsistenz explizit gewährleistet werden.

7.3.4.2 Prozessinstanzen

Die Verwaltung von Prozessinstanzen im *ProcessManager* umfasst das Anlegen von Prozessinstanzen, d. h. entsprechende Prozessvorlagen werden instanziert. Prozessinstanzen können manipuliert werden, was entweder durch das Weiterschalten (s. Abb. 7.37 Mitte) oder durch instanzspezifische Änderungen erfolgt (s. Abb. 7.37 rechts). Wie Prozessvorlagen besitzen auch Prozessinstanzen einen Lebenszyklus, in dem sie verschiedene Zustände durchlaufen (s.

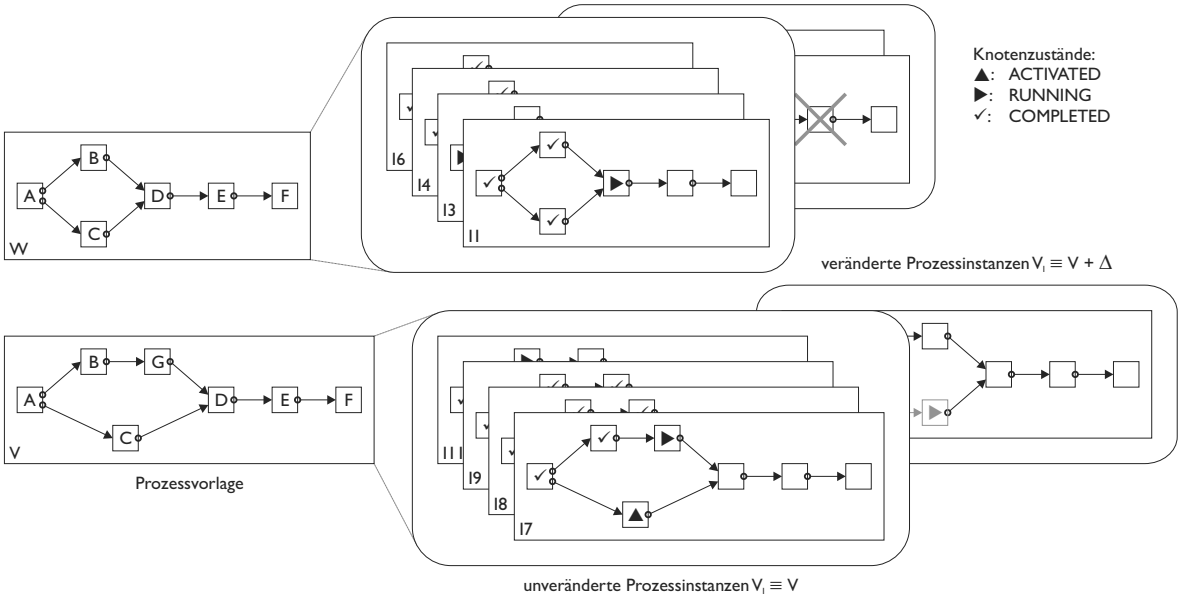


Abbildung 7.37: Daten der Prozessverwaltung

Abb. 7.38). Dies umfasst einerseits Ausführungszustände, die auch Einfluss auf die Knotenzustände haben, andererseits Zustände für die Migration einer Prozessinstanz im Rahmen einer Schemaevolution. Die Ausführungszustände umfassen normale Ausführung (`RUNNING`), beendete Prozessinstanzen (`FINISHED`), Prozessinstanzen, deren Ausführung temporär unterbrochen/pausiert ist (`SUSPENDED`), und Prozessinstanzen, deren Ausführung abgebrochen ist (`ABORTED`). Im Normalfall werden die Ausführungszustände `SUSPENDED` und `ABORTED` auf laufende Prozessschritte propagiert, d. h. laufende Prozessschritte werden pausiert bzw. abgebrochen. Daneben gibt es auch „weiche“ Varianten dieser Ausführungszustände, nämlich `SOFTLY_SUSPENDED` und `SOFTLY_ABORTED`. In diesen beiden Zuständen wird der Zustand der Prozessinstanz nicht auf laufende Prozessschritte propagiert, so dass die Prozessschritte noch zu Ende laufen können.

Damit instanzspezifische Änderungen durchgeführt werden können, muss die Ausführung einer Prozessinstanz kurzzeitig unterbrochen (`SUSPENDED`) werden. Während der Durchführung von instanzspezifischen Änderungen ist eine Prozessinstanz gegen anderweitige Manipulationen wie Weiterschalten oder Zustandsänderungen gesperrt (`CHANGING`). Der *ProcessManager* berücksichtigt dies entsprechend und verhindert so konkurrierende Manipulationen einer Prozessinstanz.

Um den Datenbestand des *ProcessManager* klein zu halten, können abgeschlossene Prozessinstanzen und nicht mehr benötigte Prozessvorlagen archiviert und anschließend entfernt werden. Hierbei ist die Konsistenz der Daten wichtig. Beispielsweise darf eine Prozessvorlage nicht aus dem *ProcessManager* entfernt werden, solange noch Prozessinstanzen existieren, selbst wenn diese alle beendet sind. Auch dürfen keine Prozessvorlagen entfernt werden, die in anderen Prozessvorlagen im *ProcessManager* als Subprozesse verwendet werden. Beim Archivieren von Prozessinstanzen müssen auch erfolgte instanzspezifische Änderungen mit archiviert werden.

Für die Überwachung laufender Prozessinstanzen erlaubt der ProcessManager auch Anfragen und Filter auf dem Datenbestand, also laufende, beendete und abgebrochene Prozessinstanzen

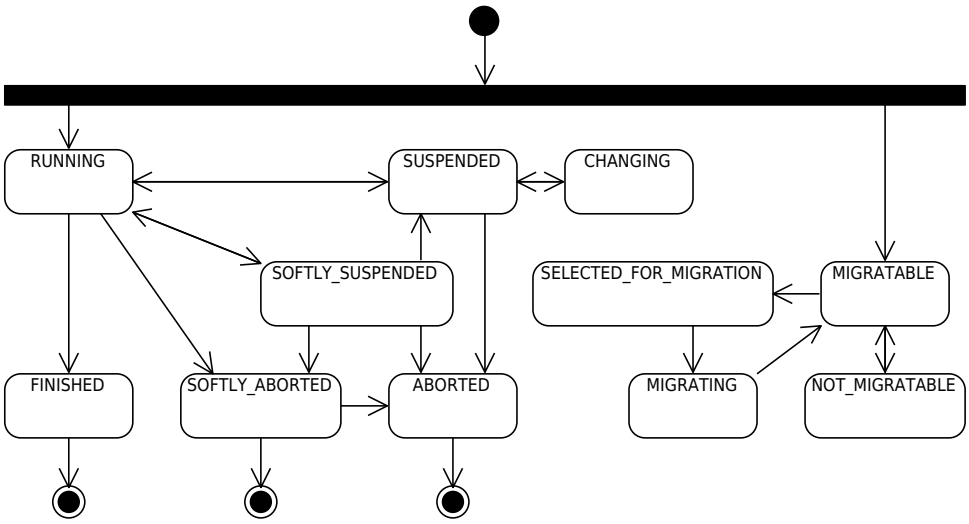


Abbildung 7.38: UML-Zustandsdiagramm des Lebenszyklus' einer Prozessinstanz

und Prozessvorlagen, die noch nicht archiviert worden sind, sowie Aggregation auf diesen Daten. Dadurch kann beispielsweise leicht ermittelt werden, welche Prozessinstanzen gerade in einem bestimmten Bereich aktiv sind oder bei welchen Prozessinstanzen gerade ein bestimmter Prozessschritt aktiv ist. Diese Art der Anfragen kann auch für die Verträglichkeitsprüfung bei einer Schemaevolution [RRD02, RRD04c]nützlich sein.

Auch die Schemaevolution läuft im *ProcessManager* ab. Dazu müssen sich die alte und die neue Version des Prozesstyps (das Ziel der Schemaevolution) im *ProcessManager* befinden. Für die Durchführung der Schemaevolution werden noch die Unterschiede zwischen den beiden Prozessvorlagen benötigt, d. h. die durchgeführten Änderungen. Diese können mit Hilfe des *ProcessRepository* (s. Abschnitt 7.3.2) ermittelt werden. Damit kann der *ProcessManager* die Verträglichkeit der Änderungen mit allen Prozessinstanzen der alten Prozessvorlage überprüfen und die Prozessinstanzen – falls möglich und gewünscht – auf die neue Prozessvorlage migrieren. Damit eine Prozessinstanz überhaupt migriert werden darf, muss sie sich im Zustand `MIGRATABLE` befinden. Ist sie migrierbar und die Migration vom Benutzer gewünscht, ist sie im Zustand `SELECTED_FOR_MIGRATION`. Dieser Zustand erlaubt es, ggf. ein Weiterschalten zu unterbinden, sofern die Prozessinstanz dadurch einen Zustand erreicht, bezüglich dem die Änderungen unverträglich sind. Während der eigentlichen Migration befindet sich die Prozessinstanz im Zustand `MIGRATING` und ist für andere Zugriffe gesperrt.

7.3.5 Verwaltung von Prozessdaten (*DataManager*)

Der *DataManager* verwaltet und speichert die in einem Prozess erzeugten Daten, also die Werte der Datenelemente von Prozessinstanzen. Somit realisiert er die funktionalen Anforderungen FA 2.2 (Datenhistorie) und FA 6 (persistente Speicherung) sowie einen Teil der funktionalen Anforderung FA 1 (Prozessausführung). Beim Starten von Aktivitäten stellt der *DataManager* die Werte der Eingabeparameter aus dem Datenfluss der entsprechenden Prozessinstanz bereit; beim Beenden nimmt er die Werte der Ausgabeparameter entgegen und schreibt sie in die entsprechenden Datenelemente. Der *DataManager* protokolliert alle in ein Datenelement geschriebenen Werte in einer Datenhistorie. Dies wird einerseits dafür benötigt, die in [Rei00]

ProcessManager

FA 6	Persistente Speicherung
FA 1	Prozessausführung
FA 9.1	Instanzspezifische Änderungen
FA 9.2	Schemaevolution

Angebotene Funktionen

- Verwaltung und persistente Speicherung laufender Prozesse (entsprechende Prozessvorlagen und -instanzen)
- Verwaltung und persistente Speicherung von instanzspezifischen Änderungen laufender Prozesse
- Einhaltung des Lebenszyklus von Prozessvorlagen und -instanzen, inklusive dem Sperren von Prozessinstanzen gegenüber parallelen Änderungen
- Archivierung abgeschlossener und nicht mehr benötigter Prozessinstanzen

definierten Einschränkungen der Sichtbarkeit von Datenwerten zu gewährleisten. Beispielsweise gelten die von einer Aktivität geschriebenen Werte nur für ihre direkten Nachfolger, aber nicht für zu ihr parallel ablaufende Aktivitäten (s. Abschnitt 2.2.2.2). Andererseits lässt sich mit der Datenhistorie die Ausführung einer Aktivität zu einem späteren Zeitpunkt lückenlos nachvollziehen.

7.3.5.1 DataContainer

Beim Start einer Aktivität werden die Werte ihrer Eingabeparameter typsicher durch ein Behälterobjekt (*DataContainer*) aus dem Ausführungskontext (*ExecutionContext*) zur Verfügung gestellt (s. Abb. 7.21). Der *DataContainer* wird vom *DataManager* erzeugt und mit den entsprechenden Werten versorgt. Dabei kann der *DataContainer* die Datenwerte entweder direkt zur Verfügung stellen und intern zwischenspeichern, oder er kann diese erst bei Bedarf vom *DataManager* nachfordern. Diese Kapselung ermöglicht interne Optimierungen hinsichtlich der übertragenen Datenmenge oder einem schnellen Datenzugriff.

Da Subprozesse weitgehend wie Aktivitäten behandelt werden und dementsprechend Ein- und Ausgabeparameter besitzen können, erhalten auch sie beim Start einen *DataContainer*. Dieser stellt wie bei einer Aktivität die Werte der Eingabeparameter zur Verfügung und nimmt die Ausgabeparameter entgegen. Auch diese *DataContainer* werden von einem *DataManager* erzeugt, genauer dem *DataManager* der Elternprozessinstanz. Da Subprozesse gewöhnlich länger laufen als Aktivitäten, werden die entsprechenden *DataContainer* zwischengespeichert, bis die jeweilige Subprozessinstanz beendet ist. Dies erledigt der *DataManager* der Subprozessinstanz. Das Zwischenspeichern erleichtert die Unterbrechung der Subprozessausführung, um beispielsweise einen Neustart des Systems durchzuführen.

Auch beim Unterbrechen von Aktivitäten ist der *DataManager* beteiligt. Sichert eine Aktivität ihren Zwischenzustand (Sicherungspunkt), um die Ausführung dort später fortzusetzen, werden diese Daten im *DataManager* gespeichert (s. Abschnitt 7.2.4.3). Im Gegensatz zu anderen Daten werden die Zwischenzustände von Aktivitäten aber nur solange vom *DataManager*

vorgehalten, bis die entsprechende Aktivität beendet ist. Allerdings werden bis zum Beenden der Aktivität alle ihre Sicherungspunkte gespeichert, um bei kritischen Ausführungsfehlern mit einem älteren Zwischenstand fortsetzen zu können.

7.3.5.2 Historienmodus und UDF

Um die Ausführung von Aktivitäten nachzuvollziehen, gibt es den Historienmodus (s. Abschnitt 2.2.1). Dieser wird im *DataManager* nahezu wie eine normale Ausführung behandelt. Die Rekonstruktion der Eingabeparameterwerte erfolgt mittels der Datenhistorie, die direkt die als Ausgabeparameter geschriebenen Werte bereitstellt. Damit erzeugt der *DataManager* auch für den Historienmodus einen *DataContainer*. Dieser unterscheidet sich lediglich durch die vorhandenen Ausgabeparameterwerte von einer normalen Ausführung.

Neben der Bereitstellung geeigneter DataContainer und dem Speichern von Zwischenzuständen, verwaltet der *DataManager* benutzerdefinierte Datentypen (UDT) und benutzerdefinierte Funktionen (UDF (*User-Defined Function*)). Dies ist vergleichbar mit BLOB in relationalen Datenbanken und (meist herstellerspezifischen) Prozeduren, die Anwendungslogik in einer relationalen Datenbank erlauben (s. Abschnitt 2.2.2.2). Der *DataManager* stellt die benutzerdefinierte Funktionen zur Verfügung und führt sie aus.

7.3.5.3 Trennung zwischen *ProcessManager* und *DataManager*

Die Aufteilung der Prozessdaten, also die Attribute von Prozessvorlagen und -instanzen, und der in einem Prozess erzeugten Daten, also die Werte von Datenelementen, in *DataManager* und *ProcessManager* (s. Abschnitt 7.3.4) erhöht die Skalierbarkeit des Gesamtsystems. Ausgenommen bei Zustandsänderungen in Prozessinstanzen (z. B. beim Starten oder Beenden von Aktivitäten), erhalten beide Dienste ihre Anfragen getrennt. So belastet das Nachfordern großer Datenmengen durch einen *DataContainer* oder die Ausführung benutzerdefinierter Funktionen die Verwaltung laufender Prozesse nicht, also den *ProcessManager*. Umgekehrt wirken sich potentiell aufwendige Anfragen und Aggregation auf Prozessinstanzen nicht auf die Verwaltung der Prozessdaten aus.

7.3.6 Prozessinterpretation und -ausführung (*ExecutionManager*)

Der *ExecutionManager* ist für die Prozessausführung zuständig und setzt den größten Teil der Anforderung FA 1 um. Dazu interpretiert er die Zustände von Prozessinstanzen sowie deren Prozessschritte und veranlasst die Zustandsübergänge gemäß dem ADEPT-Ausführungsmodell [Rei00]. Hierfür verwendet er die von *ProcessManager* (s. Abschnitt 7.3.4) und *DataManager* (s. Abschnitt 7.3.5) bereitgestellten Funktionen und Daten. Alle Zustandsübergänge protokolliert der *ExecutionManager* in der Ausführungshistorie (FA 2.1).

7.3.6.1 Zustandsänderung von Prozessschritten

Die häufigsten Zustandsänderungen sind das Starten und das Beenden Aktivitäten. Beide werden gewöhnlich außerhalb des *ExecutionManager* ausgelöst. Der Start von Aktivitäten durch Benutzer erfolgt über deren Arbeitsliste unter Verwendung der entsprechenden *EBPInstanceReference* (s. Abschnitt 7.2.3.1). Aufgrund dieser *EBPInstanceReference* sammelt der *ExecutionManager* die weiteren für die Ausführung benötigten Informationen. Dazu gehören unter anderem die für die Ausführung der Aktivität zu rufende Anwendung, sowie die Daten

DataManager

FA 1	Prozessausführung
FA 2.2	Datenhistorie
FA 6	Persistente Speicherung

Angebotene Funktionen

- Bereitstellung von Datenelementwerten speziell für eine Aktivität
- Schreiben der Werte von Ausgabeparametern einer Aktivität in die entsprechenden Datenelemente
- Erstellen und Verwalten der vollständigen Datenhistorie für alle Datenelemente aller Prozesse
- Zwischenspeicherung von *DataContainer* für Subprozesse
- Verwaltung der Anwendungszustände unterbrochener Aktivitäten
- Bereitstellen von benutzerdefinierten Datentypen (UDT) und Ausführung von benutzerdefinierten Funktionen (UDF) hierauf

(genauer die entsprechenden Werte) der von der Aktivität gelesenen Datenelemente. Diese werden an den für die Ausführung zuständigen *RuntimeManager* (s. Abschnitt 7.3.8) übergeben, der in der *EBPInstanceReference* hinterlegt ist. Anschließend wird der Zustand des entsprechenden Prozessschritts angepasst und damit auch der Zustand der Prozessinstanz.

Ist die Ausführung der Aktivität beendet, meldet dies der zuständige *RuntimeManager* und übergibt die Werte der Ausgabeparameter an den *ExecutionManager*. Der *ExecutionManager* leitet diese Werte zusammen mit Informationen zum schreibenden Prozessschritt an den *DataManager* weiter. Anschließend wird der entsprechende Prozessschritt als beendet markiert, und das Weiterschalten findet statt. Dabei bewertet der *ExecutionManager* die Zustände der Schritte einer Prozessinstanz und entscheidet, ob und wenn ja welche Prozessschritte aktiviert werden können. Handelt es sich beim beendeten Prozessschritt um eine alternative Verzweigung oder ein Schleifenende, so wird aufgrund der Ausgabedaten der Aktivität entschieden, welcher Zweig ausgewählt wird (bei Schleifen entsprechend die Rücksprungkante oder die Kante aus der Schleife heraus). Anschließend werden den Ausführungsregeln des ADEPT-Metamodells folgend der oder die zu aktivierenden Prozessschritte ermittelt und deren Zustand angepasst.

Bei jeder Zustandsänderung von Prozessschritten versendet der *ExecutionManager* Benachrichtigungen an registrierte Dienste. Hierzu zählt beispielsweise die Arbeitslistenverwaltung (*WorklistManager*, s. Abschnitt 7.3.7), die mit diesen Informationen die Arbeitslisten der Benutzer aktualisiert. Bei der Aktivierung eines Prozessschritts wird zusätzlich die hinterlegte Bearbeiterzuordnung bereitgestellt. Handelt es sich dabei um eine abhängige Bearbeiterzuordnung, löst der *ExecutionManager* diese Abhängigkeit auf, wie im Ausführungsmodell beschrieben (s. Abschnitt 7.2.4.2). dasselbe gilt auch für die anderen parametrisierbaren Attribute von Prozessinstanzen und -schritten. Dadurch bleibt die Semantik des ADEPT-Ausführungsmodells auf den *ExecutionManager* beschränkt, muss also nicht bei der Implementierung anderer Dienste, z. B. dem *WorklistManager*, berücksichtigt werden.

7.3.6.2 Zustände von Prozessinstanzen

Bevor eine Prozessinstanz ausgeführt werden kann, muss sie entsprechend ihrer Vorlage angelegt werden. Der *ExecutionManager* sorgt dafür, dass auch die für die Prozessinstanz notwendigen Datenstrukturen in *ProcessManager* und *DataManager* erzeugt werden. Direkt danach wird der Startknoten der neu erzeugten Prozessinstanz ausgeführt. Dabei werden alle Eingabeparameter der Prozessinstanz aus dem bei ihrem Start übergebenen Ausführungskontext (s. Abschnitt 7.2.4.3) gelesen, genauer dessen *DataContainer*, und in die entsprechenden Datenelemente der Prozessinstanz geschrieben. Dies ist insbesondere bei Subprozessen relevant, die wie Aktivitäten behandelt werden und dementsprechend Ein-/Ausgabeparameter besitzen. Wie erwähnt, werden die Eingabeparameter eines Prozesses durch die vom Startknoten geschriebenen Datenelemente bestimmt (s. Abschnitt 7.2.3). Nach dem Schreiben der Eingabeparameter eines Prozesses wird der Startknoten beendet und der normale Algorithmus zum Weiterschalten angewendet.

Analog dazu umfasst die Ausführung des Endknotens bei Subprozessen das Schreiben der Ausgabeparameter in den Elternprozess. Ist der Endknoten einer Prozessinstanz ausgeführt, setzt der *ExecutionManager* den globalen Instanzzustand auf beendet (`FINISHED`, s. Abb. 7.38) und markiert die entsprechenden Daten im *DataManager* als endgültig. Die Markierung ermöglicht die Archivierung der Prozessinstanz mitsamt der durch sie erzeugten Daten. Im Fall einer Subprozessinstanz wird der Zustand auf den entsprechenden Prozessschritt der Elterninstanz propagiert und diese weitergeschaltet.

7.3.6.3 Weitere Zustände von Aktivitäten

Neben dem Starten und Beenden von Aktivitäten erlaubt der *ExecutionManager* auch alle anderen Zustandsänderungen von Aktivitäten (s. Abschnitt 2.2.3). Dazu gehören das Unterbrechen (`SUSPEND`) und das Fortsetzen der Ausführung (`RESUME`). Beim Unterbrechen werden die bisherigen Werte von Ausgabeparametern zusammen mit dem Anwendungszustand (*SessionState*) zwischengespeichert. Beim Fortsetzen werden diese Daten der Aktivität wieder zur Verfügung gestellt. Beim Fehlschlag einer Aktivität (`FAILED`) übernimmt der *ExecutionManager* die bisher erzeugten Werte der Ausgabeparameter und Details zum Fehlschlag. Mit diesen Informationen startet er dann einen Fehlerbehandlungsprozess. Dies kann ein aktivitätsspezifischer, ein prozessspezifischer oder ein allgemeiner Fehlerbehandlungsprozess sein.

Für Aktivitäten wird zwischen extern Zustandsänderungen (z. B. Starten oder Fortsetzen) und intern verursachten Zustandsänderungen unterschieden. Extern verursachte Zustandsänderungen sind beispielsweise die, die über Arbeitslisten erfolgten. Intern verursachte Zustandsänderungen werden durch die Aktivität, genauer durch den mit ihr verknüpften Anwendungsbaustein, verursacht (z. B. Beenden oder Unterbrechen). Der Auslöser für die Zustandsänderung spiegelt sich in den Schnittstellen des *ExecutionManager* wider: Extern verursachte Zustandsänderungen erfolgen über die Unterschnittstelle *ActivityStarting*, intern verursachte Zustandsänderungen erfolgen über die Unterschnittstelle *ActivityTermination*.

Alle Zustandsänderungen von Prozessschritten und Prozessinstanzen protokolliert der *ExecutionManager* in einer Ausführungshistorie. Zusammen mit der Datenhistorie des *DataManager* lassen sich so die Ausführungen aller Prozessinstanzen und deren Prozessschritte jederzeit nachvollziehen und rekonstruieren.

ExecutionManager

FA 1 Prozessausführung
FA 2.1 Ausführungshistorie

Angebotene Funktionen

- Anlegen und Starten von Prozessinstanzen
- Weiterschalten von Prozessinstanzen, Anpassen der Prozessschritte entsprechend den Ausführungsregeln des ADEPT-Metamodells
- Starten und Beenden von Aktivitäten mit Übergabe der Eingabedaten an Aktivitäten bzw. Übernahme der Ausgabedaten
- Auflösung von Abhängigkeiten in Bearbeiterzuordnungen und weiteren Attributen von Prozessinstanzen und -schritten (s. Abschnitt 7.2.4.2)
- Treffen von Verzweigungsentscheidungen bei alternativen Verzweigungen und Schleifen
- Verwalten der globalen Zustände von Prozessinstanzen
- Verarbeitung der Endzustände von Aktivitäten. Neben dem normalen Beenden sind dies z. B. Unterbrechung oder Fehlschlag der Ausführung
- Protokollierung der Ausführung von Prozessinstanzen in der Ausführungshistorie

7.3.7 Verwaltung von Arbeitslisten (*WorklistManager*)

Die funktionale Anforderung FA 4 (Arbeitslistenverwaltung) wird vom *WorklistManager* realisiert. Dieser bereitet die vom *ExecutionManager* (s. Abschnitt 7.3.6) mitgeteilten Informationen (s. Abschnitt 7.2.4) auf und bietet sie in Form von Objekten des Arbeitslistenmodells an. Das Arbeitslistenmodell bildet die Grundlage für die Benutzeroberfläche.

Da die Arbeitslistenverwaltung in der Regel lediglich die Zustände von Prozessen und Prozessschritten aufbereitet, könnten viele ihrer Funktionen auch direkt vom *ExecutionManager* übernommen werden. Die Trennung zwischen Arbeitslistenverwaltung und Prozessausführung verbessert die Skalierbarkeit des Gesamtsystems. So können *ExecutionManager* und *WorklistManager* physisch getrennt voneinander laufen. Außerdem können beliebig viele *ExecutionManager*-Dienstinstanzen mit beliebig vielen *WorklistManager*-Dienstinstanzen zusammenarbeiten. Des weiteren erlaubt die Trennung zwischen Arbeitslistenverwaltung und Prozessausführung, die Arbeitslistenverwaltung mit beliebigen prozessorientierten Informationssystemen zu koppeln, unabhängig von deren Prozessmodell. Da ein Benutzer nur das Arbeitslistenmodell sieht, hat er eine einheitliche Sicht auf alle Schritte, die ihm zugeordnet sind. Technische Heterogenitäten werden durch die Arbeitslistenverwaltung verborgen.

Wie erwähnt, unterscheidet man zwischen einem internen und einem externen Arbeitslistenmodell (s. Abschnitt 7.2.6). Da das Hauptaugenmerk dieses Abschnitts auf der internen Funktionsweise des *WorklistManager*s liegt, bezieht er sich auf den internen Teil des Arbeitslistenmodells, sofern nicht anders angegeben.

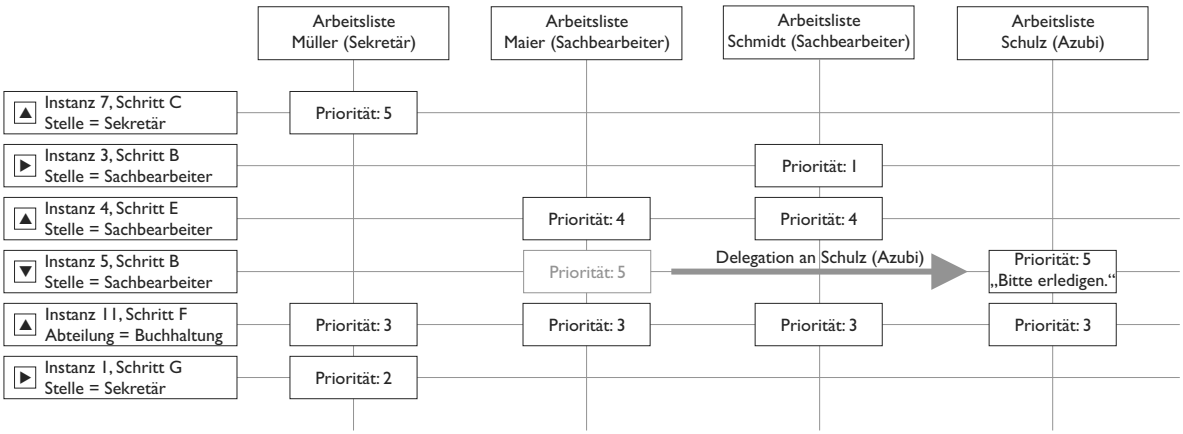


Abbildung 7.39: Logische Matrix der Arbeitslistenverwaltungsdaten

7.3.7.1 Arbeitslisten und Arbeitslisteneinträge

Die vom *WorklistManager* verwaltete Datenstruktur ist logisch eine Matrix (s. Abb. 7.39). Eine Dimension bilden die Repräsentationen der Prozessschritte (*InternalWorklistItem*, links in Abb. 7.30), die zweite Dimension bilden die Arbeitslisten (*InternalWorklist*, oben in Abb. 7.30), von denen es für jeden der Arbeitslistenverwaltung bekannten Benutzer eine gibt, genauer für jeden Stelle-Benutzer-Paar (*QualifiedAgent*, s. Organisationsmodell Abschnitt 7.2.7). Die Zuordnung zwischen Prozessschritten und Arbeitslisten erfolgt durch wechselseitige Referenzierung von *InternalWorklistItem* und *InternalWorklist*. Das heißt, ein *InternalWorklistItem* referenziert eine Menge von *InternalWorklist*s, und eine *InternalWorklist* referenziert eine Menge von *InternalWorklistItem*s.

Änderungen an dieser Datenstruktur ergeben sich unter anderem durch Zustandsänderungen in Prozessinstanzen. Wird beispielsweise ein Prozessschritt aktiviert, werden im *WorklistManager* logisch die entsprechenden Arbeitslisteneinträge erzeugt. Tatsächlich wird dabei, wie oben beschrieben, nur ein *InternalWorklistItem* erzeugt, welches eine oder mehrere *InternalWorklist*s referenziert. Beim Hinzufügen eines neu aktivierten Prozessschritts wird seine (unabhängige) Bearbeiterzuordnung durch den *OrgModelManager* (s. Abschnitt 7.3.3) auf eine Menge von Bearbeitern aufgelöst. Anschließend wird beim einfachen Verteilungsverfahren (s. Abschnitt 7.3.7.2) der Arbeitslisteneintrag den Arbeitslisten aller dieser Bearbeiter hinzugefügt. Ändert sich der Zustand eines Prozessschritts, z. B. durch Auswahl (`SELECTED`) oder Start der Ausführung (`STARTED`), wird das *InternalWorklistItem* aus allen Arbeitslisten entfernt, ausgenommen der Arbeitsliste des entsprechenden Bearbeiters entfernt. Ist eine Aktivität pausiert, verbleibt der entsprechende Arbeitslisteneintrag im Normalfall beim Bearbeiter. Wird eine Auswahl (`SELECTED`) zurückgenommen, wird der Arbeitslisteneintrag wieder allen durch die Bearbeiterzuordnung qualifizierten Bearbeitern in die Arbeitsliste gestellt.

7.3.7.2 Verteilung (*Distribution*)

Die Zusammenstellung von Arbeitslisten hat großen Einfluss auf das Arbeitsverhalten der Benutzer. Während bei einer leeren Arbeitsliste keine Ausführung von Prozessschritten erfolgt, kann ein Benutzer mit einer viele Arbeitslisteneinträgen umfassenden Arbeitsliste leicht den Überblick verlieren. Dies kann dazu führen, dass er Prioritäten falsch setzt und aus

Unternehmenssicht die falschen Prozessschritte zuerst ausführt. So können sich unternehmenskritische Prozesse leicht unnötig verzögern. Aus diesem Grund ermöglicht der realisierte *WorklistManager* verschiedene Verteilungsverfahren [Opi07]: Beim einfachen Verfahren werden Arbeitslisteneinträge allen potentiellen Bearbeiter in deren Arbeitsliste gestellt, die sich aufgrund der Bearbeiterzuordnung ergeben. Andere Verfahren unterwerfen die ermittelte Bearbeitermenge bestimmten Auswahlkriterien, um so die jeweils bestmögliche Bearbeitung zu erzielen. Beispiele hierfür sind das Umlaufverfahren (*Round Robin*) oder Verfahren, die die momentane Last, also die Menge und den geschätzten Aufwand der anderen Einträge der Arbeitslisten berücksichtigen. Dies erfordert ggf. eine gute Schätzung des Aufwands (`Complexity` in *WorklistItem* im Arbeitslistenmodell in Abschnitt 7.2.6).

Für jeden Prozessschritt können unterschiedliche Verteilungsverfahren angewendet werden, was eine flexible *Distribution* erlaubt. Dies erfolgt, wie beim Arbeitslistenmodell (s. Abschnitt 7.2.6) beschrieben, über eine Schnittstelle für Zusatzmodule (Plugins). Im Fall der Verteilung handelt es sich um die Schnittstelle *DistributionHandling*. Diese ermittelt eine Menge von Bearbeitern (*QualifiedAgent*, d. h. Mitarbeiter-Stelle-Paare), denen der entsprechende Arbeitslisteneintrag zugeordnet wird. Das Verteilungsverfahren kann dazu auf die Menge der durch die Bearbeiterzuordnung qualifizierten Bearbeiter, den Arbeitslisteneintrag (*WorklistItem*) sowie die momentanen Arbeitslisten (*Worklist*) der qualifizierten Bearbeiter zugreifen. Die einfache Verteilung lässt sich realisieren, indem die Implementierung von *DistributionHandling* die übergebene Bearbeitermenge unverändert zurückgibt.

7.3.7.3 Eskalation

Eine intelligente Verteilung von Arbeitslisteneinträgen kann zwar vorbeugen, dass Bearbeiter durch zu viele Arbeitslisteneinträge überlastet sind, sie kann jedoch nicht verhindern, dass Arbeitslisteneinträge trotzdem unnötig lange unbearbeitet bleiben. Abhilfe schafft hier Eskalation. Diese findet im Anschluss an die Verteilung statt. Dabei wird überwacht, wie lange sich ein Eintrag bereits in einer Arbeitsliste befindet. Wird eine vordefinierte maximale Dauer überschritten, werden Eskalationsmechanismen angestoßen. Diese sollen helfen, dass es zu keinen kritischen Verzögerungen in der Prozessausführung kommt. Damit leistet der WorklistManager die funktionale Anforderung FA 5.1 (Eskalation), die Teil der funktionalen Anforderung FA 5 (Prozessüberwachung) ist.

In einer Prozessvorlage wird dazu bei den einzelnen Prozessschritten hinterlegt, wann eine Eskalation stattfinden und wie diese aussehen soll. Eine mögliche Reaktion kann beispielsweise die Delegation des Prozessschritts an den Vorgesetzten eines Bearbeiters sein. Auch die Erhöhung der Priorität und damit einhergehend eine optische Hervorhebung in der Arbeitsliste sind übliche Reaktionen.

Die Integration von Eskalationsmechanismen erfolgt über die *Plugin*-Schnittstelle *EscalationHandling*. Diese besteht ausschließlich aus der Methode `deadlineReached`, welche bei Erreichen des Eskalationszeitpunkts mit dem entsprechenden Arbeitslisteneintrag (*WorklistItem*) aufgerufen wird. Außerdem erhält die Methode über *PolicyResolution* auch Zugriff auf das Organisationsmodell, etwa um den Vorgesetzten eines Benutzers zu bestimmen. Des weiteren kann sie bestimmte Attribute des Arbeitslisteneintrags ändern, etwa die Priorität, das Fälligkeitsdatum (`DueDate`) und die für die Eskalation relevanten Attribute (`EscalationDate`, `EscalationHandlingProcedureID`). Durch den Zugriff auf die für die Eskalation relevanten Attribute lassen sich mehrstufige Eskalationsverfahren implementieren: Während einer Eskalation wird festgelegt, wann die nächste Eskalation stattfindet und wie eskaliert werden soll. Ein

Beispiel für eine mehrstufige Eskalation ist die Weiterleitung eines Arbeitslisteneintrags an den Vorgesetzten eine bestimmte Zeitspanne nachdem eine Prioritätserhöhung stattgefunden hat.

7.3.7.4 Delegation und Vertretung

Im Gegensatz zu Verteilung und Eskalation wird Delegation üblicherweise vom Benutzer ausgelöst – obwohl sie auch im Rahmen einer Eskalation durchaus sinnvoll verwendet werden kann. Vor der Delegation werden, ähnlich wie bei der Verteilung, über eine *Plugin*-Schnittstelle (*DelegationHandling*) die Bearbeiter ermittelt, die als Delegationsempfänger in Frage kommen. Die Delegationsempfänger können vom Arbeitslisteneintrag, dem delegierenden Bearbeiter, dem Bearbeiter, der den Prozess gestartet hat, sowie dem Prozessverantwortlichen abhängen. Ermittelt werden sie über das Organisationsmodell (*PolicyResolution*).

Intern werden bei der Delegation die Zuordnung eines Arbeitslisteneintrags zur Arbeitsliste geändert (s. Abb. 7.39), die entsprechenden Datenstrukturen (*Delegation*) für den Arbeitslisteneintrag angelegt und ggf. Attribute angepasst (`DelegationHistory` in *InternalWorklistItem*, s. Abb. 7.30). Mit demselben Prinzip kann auch im Fall der Abwesenheit eines Bearbeiters reagiert werden: Ist ein Bearbeiter aufgrund von Krankheit oder Urlaub längere Zeit abwesend, werden alle ihm zugewiesenen Prozessschritte wie bei der Delegation an die Bearbeiter weitergeleitet, die als seine Vertreter definiert sind (`SubstitutionRule` in *InternalWorklist*, s. Abb. 7.28). So verzögert sich die Ausführung dieser Schritte aufgrund der Abwesenheit nicht.

7.3.7.5 Arbeitslisten und Arbeitslisteneinträge für Anwendungen

Damit der *WorklistManager* alle benötigten Funktionen möglichst einfach anbieten kann, ist das interne Arbeitslistenmodell entsprechend optimiert. Allerdings ist es aufgrund seiner Komplexität weniger für die Anzeige von Arbeitslisten in einem Anwendungsprogramm geeignet. Aus diesem Grund gibt es noch das externe Arbeitslistenmodell (s. Abschnitt 7.2.6). Dieses abstrahiert vom internen Arbeitslistenmodell und konzentriert sich auf die Attribute, die für die Anzeige von Arbeitslisteneinträgen relevant sind. Dies zeigt sich am Verzicht auf die Attribute für die Verteilung-, Eskalation- und Delegation-*Plugins* sowie dem rein lesenden Zugriff auf die Attribute der Arbeitslisteneinträge. Durch schreibenden Zugriff könnten Benutzer das *InternalWorklistItem* ändern, das von mehreren Arbeitslisten referenziert und damit von mehreren Benutzern geteilt wird. Damit ließe sich beispielsweise der Eskalationsmechanismus umgehen.

Trotz dieser Einschränkung können Benutzer individuelle Einstellungen an ihren Arbeitslisten und Arbeitslisteneinträgen vornehmen, die zudem dauerhaft gespeichert werden. So können sie beispielsweise Arbeitslisteneinträgen einen eigenen Namen geben, Zusatzinformationen hinterlegen oder eine individuelle Eskalation definieren, die zusätzlich zu der in der Prozessvorlage definierten Eskalation gilt. Diese individuellen Einstellungen werden allerdings nicht im externen Arbeitslistenmodell sondern beim *InternalWorklistItem* hinterlegt. So wird sichergestellt, dass diese Einstellungen für jedes Anwendungsprogramm mit eigener Arbeitsliste gelten, das ein Benutzer verwendet.

Zusätzlich zur graphischen Anzeige von Arbeitslisten in Klientenprogrammen ist häufig auch eine Benachrichtigung über ausstehende Prozessschritte per Mail erwünscht. Hierfür bietet sich ein eigener Dienst an, der aus Sicht des *WorklistManager*-Diensts ein normales Klientenprogramm ist. Dieser Dienst zeigt aber Arbeitslisten nicht graphisch an, sondern

WorklistManager

FA 4 Arbeitslistenverwaltung (Verteilung von Arbeitslisteneinträgen)
FA 5.1 Eskalation

Angebotene Funktionen

- Konsolidierte Sicht auf aktuell ausführbare Prozessschritte in Form von benutzerspezifischen Arbeitslisten
- Zuordnung von Arbeitslisteneinträgen zu Benutzern und damit auch Zuordnung von zwischen Benutzern und Prozessschritten
- Optimierte Verteilung von Arbeitslisteneinträgen (*Distribution*) auf Arbeitslisten
- Delegation
- Überwachung von Arbeitslisteneinträgen und automatisches Einleiten definierter Maßnahmen (Eskalation)
- Bereitstellung benutzerspezifischer Zusatzattribute und Einstellungen für Arbeitslisteneinträge
- Pflegen von Arbeitslisten; Hinzufügen, Entfernen, Ändern von Arbeitslisteneinträgen entsprechend den dazugehörigen Zuständen der Prozessschritte

erzeugt daraus eine textuelle Darstellung, die dann – je nach gewünschter Aktualisierungshäufigkeit – per Mail an den entsprechenden Benutzer versendet wird. Hierfür kann das externe Arbeitslistenmodell ohne zusätzliche Erweiterung verwendet werden.

7.3.8 Ausführung von Anwendungsbausteinen (*RuntimeManager*)

Der *RuntimeManager* führt Anwendungsbausteine aus, die durch Prozessschritte referenziert werden. Somit realisiert er einen Teil der funktionalen Anforderung FA 1 (Prozessausführung). Zudem erfüllt er zusammen mit dem Laufzeitumgebungsmodell (s. Abschnitt 7.2.5) die Anforderung FA 7 (Anwendungsintegration und -aufruf).

7.3.8.1 Besonderheiten der Architektur

Die Anwendungsintegration ist besonders hinsichtlich der Integration graphischer Oberflächen anspruchsvoll und wirkt sich entsprechend auf die Architektur des *RuntimeManager*s aus. So lässt sich die Integration graphischer Oberflächen oftmals nur durch eine enge Bindung an den *Client* umsetzen. Dies ist der Fall, wenn die graphischen Oberflächen des Anwendungsbausteins direkt auf dem Betriebssystem beruhen, z. B. Windows Forms, AWT oder SWT. Damit diese direkt in die umgebende *Client*-Oberfläche integriert werden können (s. Abb. 7.41), müssen sie lokal beim *Client* ausgeführt werden. Insofern unterscheidet sich der *RuntimeManager* von den anderen Diensten, die zentral (d. h. auf einem Server) ausgeführt werden.

Eine Ausnahme hiervon bilden Anwendungsbausteine mit Web- (z. B. HTML) und vergleichbaren Oberflächen [wpf, jaf] oder Anwendungsbausteine ohne Oberflächen, etwa langlaufende Berechnungen. Für diese bietet sich ein zentraler *RuntimeManager* an. Die Systemumgebung

(Hardware und Software) ist bei zentralen Diensten in der Regel leistungsfähiger und stabiler als bei Arbeitsplatzrechnern.

Bei der zentralen Ausführung werden naheliegenderweise Anwendungsbausteine von Prozessschritten unterschiedlicher Prozesse und durch verschiedene Benutzer gleichzeitig ausgeführt. Die in FA 7 geforderte Unabhängigkeit von Anwendungsbausteinen untereinander erhöht bei der zentralen Ausführung die Stabilität, da sich Fehler bei der Ausführung eines Anwendungsbausteins nicht auf andere Anwendungsbausteine auswirken. Bei lokalen *RuntimeManager*n ist dies auch sinnvoll, allerdings steht hier die Unabhängigkeit der Anwendungsbausteine von der Ausführungsumgebung im Vordergrund. Dies erlaubt es, die Ausführungsumgebung, etwa *Client*s, unabhängig von Anwendungsbausteinen zu entwickeln.

Während ein zentraler *RuntimeManager* verschiedene Benutzer gleichzeitig bedienen muss, ergibt sich bei der Verwendung lokaler *RuntimeManager* für jeden Benutzer das Problem, dass die Aktivitätsausführung vom *ExecutionManager* jeweils dem richtigen der zahlreichen *RuntimeManager* zugewiesen werden muss. Dies wird dadurch realisiert, dass die Startanforderung an den *ExecutionManager* eine *EBPInstanceReference* mit der Adresse des gewünschten *RuntimeManager*s (`RuntimeManagerURIs`) enthält. So ist gewährleistet, dass im Fall eines Anwendungsbausteins mit graphischer Oberfläche diese beim richtigen Benutzer erzeugt wird.

Obwohl lokale und zentrale *RuntimeManager* unterschiedliche Anforderungen erfüllen müssen, werden wir sie im folgenden gemeinsam betrachten. Aus Architektursicht sind einerseits die Übergänge zwischen lokalem und zentralem *RuntimeManager* fließend, andererseits profitieren beide Einsatzszenarien auch von der Erfüllung der Anforderungen des jeweils anderen Szenarios, etwa wenn es um die Unabhängigkeit der Anwendungsbausteine geht. Dies schließt natürlich nicht aus, dass es unterschiedliche *RuntimeManager*-Implementierungen für beide Szenarien geben kann. Da ein *RuntimeManager* wie jeder andere Dienst über seine Schnittstelle definiert ist, lässt sich dies problemlos realisieren.

7.3.8.2 Unabhängigkeit graphischer Oberfläche und von Anwendungsbausteinen

Ein Teil der Anwendungsintegration (FA 7) wird mittels des Laufzeitumgebungsmodells (s. Abschnitt 7.2.5) umgesetzt, zu dem unter anderem die Schnittstellen *ExecutableComponent*, *SessionContext*, *DataContext* und *RuntimeEnvironment* (s. Abb. 7.40) gehören. Diese Schnittstellen bilden auch einen Teil der Trennung zwischen Anwendungsbausteinen und Laufzeitumgebung. Ein weiterer Teil dieser Trennung wird durch den *GUIManager* realisiert. Dieser ist Bestandteil des *RuntimeManager*; er greift aber auf die zugrundeliegenden Systemressourcen zu, die die lokale Systemumgebung oder der *Client* zur Verfügung stellt.

Beispiel 7.21 (Oberflächenintegration im *RuntimeManager*)
Abb. 7.40 verdeutlicht den Zugriff des *RuntimeManager*s auf die zugrundeliegenden Systemressourcen: Beim Start eines Anwendungsbausteins wird über *GUIContext* die für den Anwendungsbaustein passende Schnittstelle für die graphische Oberfläche zur Verfügung gestellt (*GUI-Typ 2*). Dieser Typ muss nicht mit dem Typ des *Client*s übereinstimmen, sondern kann durch eine geeignete *GUIContext*-Implementierung (*GUIContext 1-2* in Abb. 7.40) in den des *Client*s eingebettet werden. Ein Beispiel für verschiedene graphische Oberflächen (AWT und SWT) zeigt Abb. 7.41. Die Erzeugung von *GUIContext*en wird durch den *GUIManager* realisiert. Damit wird die Transformation der verschiedenen GUI-Typen sowohl vor dem Anwendungsbaustein als auch vor dem *RuntimeManager* verborgen. Zudem sind Anwendungsbaustein und *RuntimeManager* unabhängig von der bei verschiedenen GUI-Typen notwendigen Transformation.

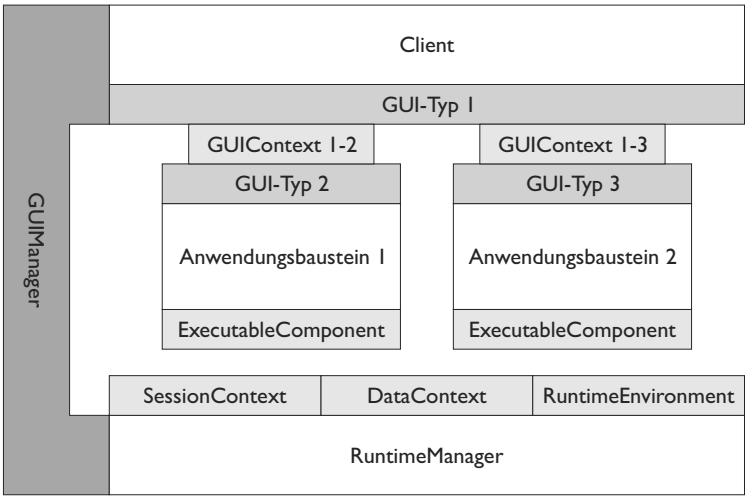


Abbildung 7.40: Ausführung von Anwendungsbausteinen und Oberflächenintegration

Die Trennung der Anwendungsbausteine voneinander wird auf technischer Ebene durch die Verwendung separater Threads, Klassenlader und Sicherheitskontexte für jede Ausführung erreicht. Dies hat jedoch keinerlei Auswirkungen auf die Implementierung eines Anwendungsbausteins. Da die Ausführungskontrolle beim *RuntimeManager* liegt und nicht beim Anwendungsbaustein selbst, ist dieser auch für die Threads, Klassenlader und Sicherheitskontexte verantwortlich. Außerdem ermöglicht dies, einem nicht mehr reagierenden oder in anderer Form auffälligen Anwendungsbaustein die Betriebsmittel zu entziehen und seine Ausführung abzubrechen, ohne dass hiervon andere Anwendungsbausteine oder die Laufzeitumgebung betroffen sind.

7.3.8.3 Interaktion mit dem Anwendungsbaustein über die Laufzeitumgebung

Die Ausführung eines Anwendungsbausteins lässt sich nicht nur durch den Anwendungsbaustein selbst kontrollieren, etwa durch Schaltflächen zum Schließen der Anwendung, sondern auch durch die Laufzeitumgebung und damit durch den *RuntimeManager* kontrollieren. So kann man beispielsweise aus der Arbeitsliste oder über den Monitor einem Anwendungsbaustein auf einem entfernten Rechner bestimmte Signale senden, um diesen etwa zum Beenden, Unterbrechen oder Abbrechen aufzufordern. Diese Ausführungskontrolle durch die Laufzeitumgebung ist mit der von Betriebssystemprozessen etwa durch den *Windows Taskmanager* vergleichbar. Da diese Signale von außen an den Anwendungsbaustein geschickt werden, sind sie asynchron gegenüber der Ausführung des Anwendungsbausteins. Die korrekte Verarbeitung und Reaktion auf die Signale hat großen Einfluss auf die Implementierung eines Anwendungsbausteins. Dies widerspricht der Forderung nach einfachen Implementierungen (NA 4.2), weshalb die Verarbeitung der Signale nicht verpflichtend ist. Welche Signale ein bestimmter Anwendungsbaustein verarbeiten kann, ist im Aktivitätenmodell in den `ExecutionControlProperties` hinterlegt.

Ein Sonderfall ist das Abbrechen-Signal. Wird dieses vom Anwendungsbaustein eine gewisse Zeit ignoriert, gilt der Anwendungsbaustein als nicht reagierend. Dann können ihm die Betriebsmittel entzogen und seine Ausführung abgebrochen werden. Dies ist ein notwendiger Ausnahmefall, um den *RuntimeManager* auch bei laufenden Anwendungsbausteinen in Notfällen (z. B. Stromausfall) beenden zu können.

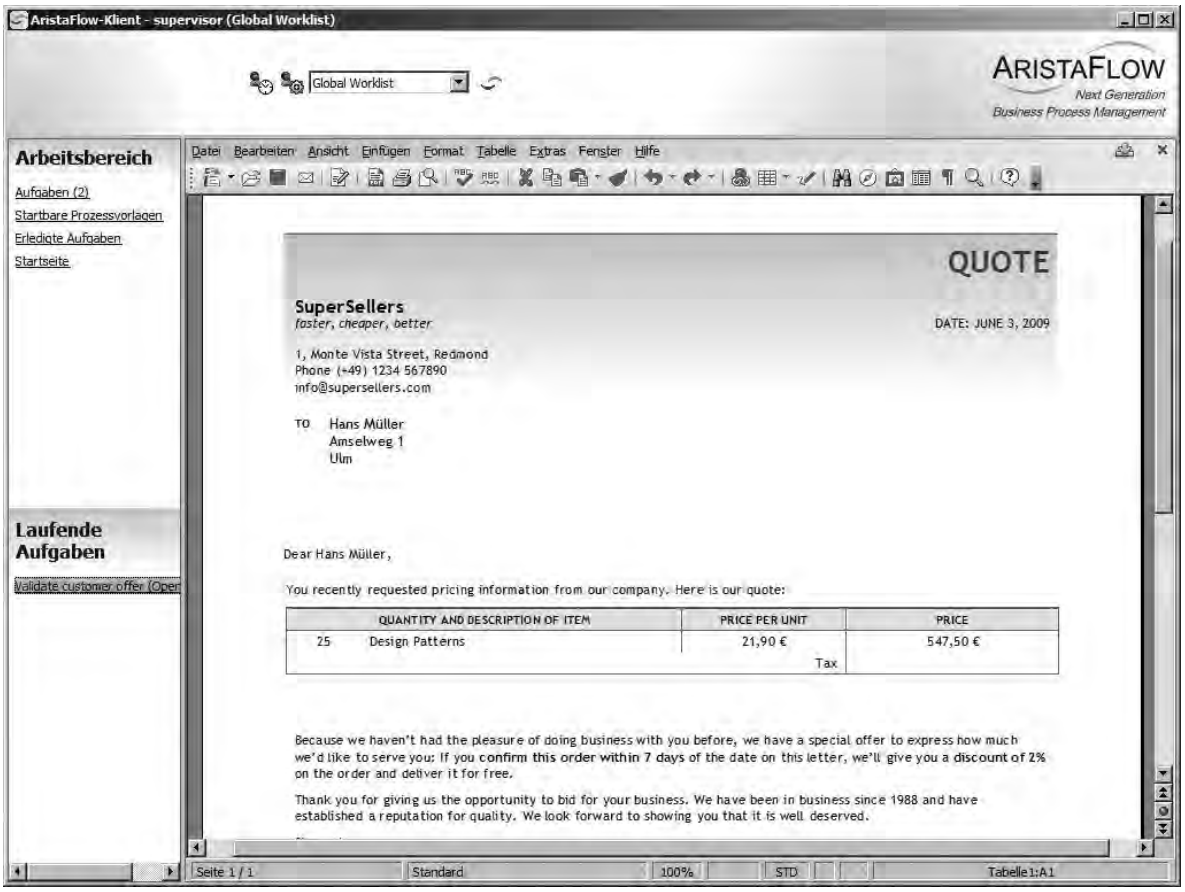


Abbildung 7.41: Beispiel für die Integration von AWT (*OpenOffice.org*) in einen SWT-*Client*

Bei Anwendungsbausteinen mit Benutzerinteraktion werden die Signale zur Ausführungskontrolle von außen oftmals nicht benötigt, da sich über Interaktionen innerhalb des Anwendungsbausteins dasselbe erreichen lässt, z. B. *OK* und *Abbrechen* in Dialogen. Die Signale von außen sind aber einerseits nützlich, wenn diese Interaktionen nicht mehr möglich sind, etwa weil Teile des Anwendungsbausteins nicht mehr reagieren. Andererseits lassen sich über die Signale auch Anwendungsbausteine ohne Benutzerinteraktion kontrollieren, etwa wenn eine langlaufende Berechnung unterbrochen werden soll. Dies gilt unabhängig davon, ob ein Anwendungsbaustein lokal oder zentral ausgeführt wird.

7.3.8.4 Unterbrechen, Fortsetzen und Historienmodus

Neben der normalen Ausführung von Anwendungsbausteinen ermöglicht der *RuntimeManager* auch die Fortsetzung unterbrochener Ausführungen. Die Unterbrechungen müssen durch die Anwendungsbausteine selbst durchgeführt werden, d. h. diese übergeben ihre internen Zustandsinformationen sowie ihren Ausführungszustand (`SUSPENDED`) an die Laufzeitumgebung und beenden dann ihre Ausführung. Dies gilt auch für den Fall, dass die Unterbrechungsanforderung von außen gekommen ist. Beim Fortsetzen initialisiert die Laufzeitumgebung den Anwendungsbaustein und stellt dabei zusätzlich die Zustandsinformationen und den Ausführungszustand zur Verfügung. Es obliegt dem Anwendungsbaustein, seinen vorherigen Ausführungszustand wiederherzustellen (s. Abschnitt 7.2.5). Dadurch kann die Ausführung an der vorher unterbrochenen Stelle fortgesetzt werden.

Es ist zwar technisch möglich, die Ausführung durch die Laufzeitumgebung so zu unterbrechen, dass dies vom Anwendungsbaustein nicht bemerkt wird, wir fordern dies jedoch nicht generell. Realisieren lässt sich solch eine anwendungsunabhängige Unterbrechung, indem der komplette interne Zustand des Anforderungsbausteins (sein *Heap*) sowie alle seine Threads mit ihren jeweiligen Zuständen (ihre *Stack*s) von der Laufzeitumgebung gespeichert werden. Beim Fortsetzen der Ausführung werden diese Informationen wiederhergestellt und die Threads gestartet. Offensichtlich stellt dies jedoch so weitgehende Anforderungen an die Laufzeitumgebung und die zugrundeliegende Systemumgebung, dass wir darauf verzichten.

Eine weitere Ausführungsart ist der Historienmodus. Bei diesem wird der Anwendungsbaustein weitgehend normal ausgeführt, allerdings dient die Ausführung nur dazu, einen bereits abgeschlossenen Prozessschritt nachzuvollziehen. Dazu setzt die Ausführungsumgebung im *SessionContext* den Schalter `isViewOnly`. Dieser zeigt dem Anwendungsbaustein an, dass er die Werte der Ausgabeparameter nicht bearbeiten und schreiben, sondern nur anzeigen darf. Dementsprechend muss der Anwendungsbaustein auch alle Ausgabeparameter wie Eingabeparameter zu behandeln. Zudem ignoriert die Ausführungsumgebung den Endzustand eines Anwendungsbausteins bei seiner Beendigung. Das heißt, es gibt kein unterbrochen oder fehlgeschlagen; dies ist bei einem abgeschlossenen Prozessschritt auch nicht sinnvoll. Naheliegenderweise erfordert der Historienmodus zusätzliche Anwendungslogik beim Anwendungsbaustein. Deshalb kann ein Anwendungsbaustein auf den Historienmodus verzichten, was dann wie `ExecutionControlProperties` im Aktivitätenmodell hinterlegt ist (s. Abschnitt 7.2.2).

7.3.8.5 Ausführung von Kindprozessen (*SubprocessManager*) und anwendungskontrollierte Ausführung von Aktivitäten (*RuntimeService*)

Neben dem *RuntimeManager*, der Anwendungsbausteine lokal oder zentral ausführt, gibt es zwei spezielle *RuntimeManager*: den *SubprocessManager* und den *RuntimeService*. Der

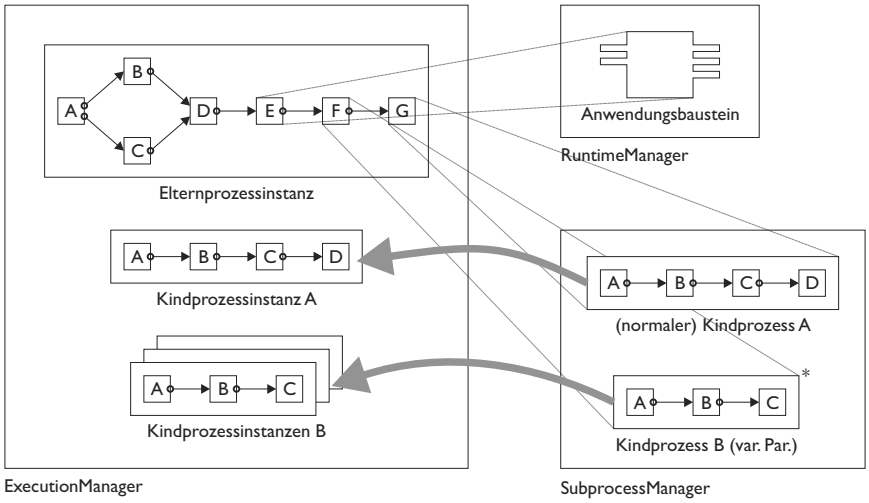


Abbildung 7.42: *RuntimeManager* und *SubprocessManager*

SubprocessManager kapselt die Ausführung von Kindprozessinstanzen durch den *ExecutionManager* so, dass sie sich aus Sicht der Elterninstanz wie Anwendungsbausteine verhalten. Der RuntimeService stellt nur die Informationen für einen Anwendungsbaustein zur Verfügung, führt aber nichts aus und kontrolliert dementsprechend auch nicht die Ausführung.

SubprocessManager

Aus Sicht eines Elternprozesses werden Kindprozesse und Aktivitäten weitgehend gleich behandelt. Idealerweise setzt sich diese Gleichbehandlung auch bei der Ausführung fort. Ein Anwendungsbaustein besitzt allerdings interne Logik, die über das Laufzeitumgebungsmodell von einem *RuntimeManager* ausgeführt wird. Prozesse allgemein und damit auch Kindprozesse haben keine interne Logik, sondern genügen dem ADEPT-Prozessmodell, das vom *ExecutionManager* interpretiert wird. Diese Unterscheidung übernimmt der *SubprocessManager*.

Der *SubprocessManager* implementiert die *RuntimeManager*-Schnittstelle. Damit kann er wie ein normaler *RuntimeManager* verwendet werden. Intern führt er jedoch keine Anwendungsbausteine aus, sondern startet und kontrolliert die Ausführung einer (Kind-)Prozessinstanz im *ExecutionManager*. Die Informationen über den entsprechenden Kindprozess erhält er über das Ausführungsmodell (s. Abschnitt 7.2.4), genauer der Unterklasse *LightWeightProcessInstance* von *ExecutableBusinessProcessInstance* (s. Abb. 7.19). Mit diesen Informationen kann der *SubprocessManager* Prozessinstanzen ausführen lassen und deren Ausführungskontrolle (z. B. Unterbrechen oder Abbrechen) über die *RuntimeManager*-Schnittstelle bereitstellen. Somit wird die Unterscheidung zwischen der Ausführung eines Kindprozesses und eines Anwendungsbausteins vollständig im *SubprocessManager* gekapselt. Der *ExecutionManager* des Elternprozesses behandelt Aktivitäten und Kindprozesse gleich (s. Abb. 7.42).

Wie ein *RuntimeManager* ermöglicht auch der *SubprocessManager* das Senden von Signalen an ausgeführte Kindprozessinstanzen. Hierbei wird das einfachere Zustandsmodell von Knoten auf das umfangreichere Zustandsmodell von Prozessinstanzen (s. Abschnitt 7.3.4 und Abb. 7.38) abgebildet. So ist über die Signale der Laufzeitumgebung keine Unterscheidung zwischen `SOFTLY_SUSPENDED` und `SUSPENDED` möglich. Zudem führt das Beenden eines *SubprocessManager*s nicht zum Beenden der laufenden Kindprozessinstanzen. Dagegen werden alle ausgeführten

Anwendungsbausteine zusammen mit dem entsprechenden *RuntimeManager* beendet. Da ein *SubprocessManager* alle Anfragen direkt an den *ExecutionManager* der Kind- oder der Elterninstanz weiterleitet und so auch die Unabhängigkeit der Ausführung gewährleistet ist, besitzt er weniger Anwendungslogik als ein *RuntimeManager*.

Neben der Ausführung normaler Kindprozesse übernimmt der *SubprocessManager* auch die Ausführung von Kindprozessen einer variablen Parallelität (s. Abschnitt 2.2.3.5). Da variable Parallelität nicht im Ausführungsmodell enthalten ist (s. Abschnitt 7.2.4), greift der *SubprocessManager* hierzu auf den *ProcessManager* zu. Hat er alle notwendigen Informationen, spaltet er die listenwertige Daten der variablen Parallelität entsprechend auf, erzeugt die *DataContainer* für die Kindprozesse und startet diese. Die Ausführungskontrolle erfolgt ähnlich der eines normalen Kindprozesses, nur dass bei variabler Parallelität mehrere Kindprozesse zusammen kontrolliert werden. Entsprechend werden ihre Ausführungszustände aggregiert. So werden die verschiedenen Kindprozesse einer variablen Parallelität nicht alle gleichzeitig beendet. Ihr Elternprozess darf aber erst weiterschalten, wenn alle Kindprozesse beendet sind. Ist dies der Fall, aggregiert der *SubprocessManager* die Ausgabedaten aller Kindprozesse und übergibt diese in Form listenwertiger Ausgabeparameter an den Elternprozess. Damit übernimmt der *SubprocessManager* vollständig die Ausführung variabler Parallelität (s. Abb. 7.42); ein *ExecutionManager* kennt den Unterschied zwischen normalen Kindprozessen und variabler Parallelität ebensowenig wie den Unterschied zwischen einem Anwendungsbaustein und einem Kindprozess.

RuntimeService

Der *RuntimeService* ist ein weiterer spezieller *RuntimeManager*. Dieser dient, ebenso wie der normale *RuntimeManager*, der Ausführung von Aktivitäten. Allerdings liegt die Kontrolle der Ausführung nicht beim *RuntimeService*, sondern ausschließlich bei der aufrufenden Anwendung. Der *RuntimeService* bietet an seiner Schnittstelle alle Informationen zur Verfügung, die auch ein Anwendungsbaustein erhält. Was mit diesen Informationen passiert und wie sie interpretiert werden, ist Sache der aufrufenden Anwendung. Wann und wie die aufrufende Anwendung mit dem *RuntimeService* interagiert, bestimmt sie selbst. Spätestens beim Beenden der Aktivitätsausführung muss die Anwendung den *RuntimeService* entsprechend informieren, etwa über die Werte der Ausgabeparameter. Dies führt dann wie beim Beenden eines Anwendungsbausteins zu den entsprechenden Konsequenzen in der entsprechenden Prozessinstanz.

Beispiel 7.22 (Existierende Anwendung nutzt *RuntimeService*)
Eine existierende Anwendung mit zahlreichen Formularen soll prozessorientiert umgestaltet werden. Das heißt, der Ablauf der Formulare wird nicht mehr in der Anwendung definiert sondern in einem von der Anwendung getrennten Prozess, der in einem PrMS läuft. Die in der Anwendung existierenden Formulare sollen weitgehend unverändert weiter verwendet werden, der Aufruf der Formulare wird wie bei Prozessmanagement üblicherweise über eine in die Anwendung integrierte Arbeitsliste ausgelöst.

Mit dem normalen *RuntimeManager* müssten die Formulare aufwendig aus dem Anwendungsprogramm gelöst oder geeignet gekapselt werden. Da die Kontrolle vom Anwendungsprogramm an den *RuntimeManager* übergeht, sind zusätzliche Anpassungen am Anwendungsprogramm notwendig. Bei Verwendung des *RuntimeService* dagegen bleibt die Kontrolle beim Anwendungsprogramm. Dieses bekommt vom *RuntimeService* die Information, wann welches Formular angezeigt werden soll. Das Formular oder die Anwendung müssen nur dahingehend angepasst werden, dass beim Schließen des Formulars die Ausgabedaten und der Endzustand des Formulars dem *RuntimeService* mitgeteilt werden.

Dieses Vorgehen widerspricht der Anforderung der Anwendungsintegration (FA 7) hinsichtlich der Unabhängigkeit von Anwendungsbausteinen untereinander und von der Ausführungsumgebung. Man kann hier nicht mehr von einer Integration (hinsichtlich Anwendungsbausteinen) sprechen: Durch die Verwendung des *RuntimeService* wird die aufrufende Anwendung eng an den Prozess gekoppelt, was die Wiederverwendung der „Anwendungsbausteine“ erheblich einschränkt. Allerdings bietet der *RuntimeService* eine einfache Möglichkeit, existierende Anwendungen ohne größere Eingriffe in ihre Architektur prozessorientiert umzugestalten. Während sich serviceorientierte Architekturen (SOA) einfach über Anwendungsbausteine integrieren lassen, ist der *RuntimeService* für monolithische Architekturen der einfachere, wenn auch nicht notwendigerweise der bessere Weg. Der *RuntimeService* eignet sich außerdem für Anwendungen, die die Kontrolle nicht abgeben können. Dies betrifft beispielsweise Anwendungen, die das PrMS über seine Webdienst-Schnittstelle ansprechen. Hier können nur Daten ausgetauscht werden, der Webdienst selbst ist rein passiv und kann keine Kontrolle ausüben.

Da der *RuntimeService* von Anwendungen direkt aufgerufen wird und ggf. auch als Webdienst verfügbar ist, trennt er im Gegensatz zum *RuntimeManager* strikt zwischen Daten und Anwendungslogik (s. Abschnitt 7.2.5.4). Er verwendet intern ein abgewandeltes Laufzeitumgebungsmodell, das nur die Daten umfasst, während die Anwendungslogik von der *RuntimeService*-Schnittstelle angeboten wird. Alle Daten einer Aktivität werden über einen speziellen *SessionContext* (*SimpleSessionContext*) bereitgestellt. Große Teile von *RuntimeEnvironment* werden von der Schnittstelle *RemoteRuntimeEnvironment* übernommen, die eine Unterschnittstelle von *RuntimeService* bildet. Hierüber werden beispielsweise wie beim *RuntimeManager* Endzustände mitgeteilt. Allerdings erfordern die entsprechenden Methoden im Gegensatz zu *RuntimeEnvironment* die Werte der Ausgabeparameter, da der *DataContext* nicht als Referenz sondern als Kopie zur Verfügung gestellt wird.

Der *RuntimeService* ist in hohem Maße auf die Kooperation seitens der aufrufenden Anwendung angewiesen, insbesondere weil er trotz der mangelnden Kontrolle sämtliche Funktionen eines *RuntimeManager*s besitzt, also z. B. auch Signale von außen weiterleitet. Allerdings kann er Signale nur weiterleiten, er selbst kann keine aktive Rolle übernehmen. So kann er bei einem Abbrechen-Signal der Anwendung nicht die Betriebsmittel entziehen. Dies ist beim *RuntimeService* jedoch weniger kritisch, da er im Gegensatz zum *RuntimeManager* nichts ausführt und somit eine nicht reagierende Aktivität evtl. parallel laufende Aktivitäten beein-

RuntimeManager

FA 1 Prozessausführung
FA 7 Anwendungsintegration und -aufruf

Angebotene Funktionen

- Lokale und zentrale Ausführung von Anwendungsbausteinen
- Separation der Anwendungsbausteine voneinander und Integration der graphischen Benutzeroberfläche in die Ausführungsumgebung (z. B. *Client*)
- Kontrolle der Ausführung (Threads, Klassenlader und Sicherheitskontexte) und Verarbeitung von Kontrollsignalen von außerhalb des Anwendungsbausteins
- Unterbrechung und Fortsetzung der Ausführung sowie Ausführung im Historienmodus
- Ausführung von Kindprozessen (*SubprocessManager*)
- Ausführung variabler Parallelität (*SubprocessManager*)
- Bereitstellung aller Daten aus dem Prozess zur getrennten Ausführung einer Aktivität (*RuntimeService*)
- Bereitstellung der Signale zur Ausführungskontrolle bei getrennter Ausführung einer Aktivität (*RuntimeService*)
- Übernahme der Ausgabedatendaten einer getrennt ausgeführten Aktivität (*RuntimeService*)

flusst. Ein Ignorieren des Abbrechen-Signals durch die aufrufende Anwendung kann jedoch dazu führen, dass der *RuntimeService* die entsprechende Aktivität selbst als fehlgeschlagen an den *ExecutionManager* meldet, obwohl die Anwendung die Aktivität noch ausführt. Wenn sie das Signal jedoch ignoriert, muss der *RuntimeService* davon ausgehen, dass die Anwendung nicht mehr reagiert und die Aktivität damit fehlgeschlagen ist. Dies kann entsprechend zu einer Inkonsistenz bezogen auf die Zustände der Anwendung und des Prozesses führen. Sobald die Anwendung dem *RuntimeService* das Beenden der Aktivität mitteilt, wird ihr diese Inkonsistenz mitgeteilt. Sie muss sich dann selbst darum kümmern, wieder einen konsistenten Zustand herzustellen.

7.3.9 Prozessänderungen (*ChangeOperations*)

Der Dienst für Prozessänderungen (*ChangeOperations*) setzt die funktionalen Anforderungen FA 8 (Prozessmodellierung) und FA 9 (Adaptivität) um, und zwar sowohl instanzspezifische Änderungen als auch Schemaevolution. Dazu stellt er alle vom Metamodell vorgesehenen Änderungsoperationen (s. Abschnitt 2.3) über seine Schnittstelle zur Verfügung. Die Änderungsoperationen selbst unterscheiden nicht zwischen dem jeweiligen Änderungsszenario (s. Abschnitt 2.3.2), in dem sie verwendet werden. Auch dies erledigt *ChangeOperations*. Darüber hinaus ist *ChangeOperations* für die Transaktionslogik von Änderungstransaktionen verantwortlich (s. Abschnitt 2.3.1), d. h. dass die Kombination von Änderungsoperationen im

ChangeOperations

FA 8 Prozessmodellierung
FA 9 Adaptivität

Angebotene Funktionen

- Bereitstellung aller im Metamodell vorgesehenen Änderungsoperationen
- Bereitstellung von Transaktionslogik für eine Menge zusammengehöriger Änderungsoperationen
- Anlegen der Änderungshistorie, Schreiben der Änderungshistorie durch Änderungsoperationen
- Anwendung der Regeln zur Korrektheitsprüfung, d. h. Überprüfung der Verträglichkeit bezüglich Struktur und Zustand entsprechend des jeweiligen Änderungsszenarios

Rahmen einer Änderungstransaktion atomar durchgeführt werden. Dies erfordert Mechanismen für die Synchronisation von Prozessänderungen, z. B. geeignete Sperrverfahren, und für das Zurücksetzen von Änderungsoperationen im Fehlerfall. Zu Änderungstransaktionen gehört außerdem das Anlegen der zugehörigen Änderungshistorie.

Alle von *ChangeOperations* angebotenen Änderungsoperationen sind eigenständige Einheiten (Java-Klassen), die dieselben Funktionen und damit ähnliche Schnittstellen aufweisen. Jede dieser Einheiten realisiert die Funktionen, die für die jeweilige Änderungsoperation für die verschiedenen Änderungsszenarien notwendig sind. Dazu gehören die Korrektheitsprüfungen bezüglich Struktur und Zustand einer Prozessvorlage/Prozessinstanz, die strukturellen Schematransformationen und die Zustandsanpassungen sowie die Protokollierung der auf dem Datenmodell einer Prozessvorlage/Prozessinstanz durchgeführten Änderungen. Die jeweils durchgeführten Änderungen beruhen auf den Änderungsprimitiven, die im Datenmodell von der Schnittstelle *ChangePrimitives* (s. Abb. 7.16) bereitgestellt werden. Somit kann die Implementierung einer Änderungsoperation in jedem Änderungsszenario eingesetzt werden, also für die Prozessmodellierung, für instanzspezifische Änderungen und für Schemaevolution (s. Abschnitt 2.3.2).

Ein Änderungsszenario wird beim Start einer Änderungstransaktion festgelegt. Dabei legt der *ChangeOperations*-Dienst die jeweils notwendigen Datenstrukturen an, etwa *TemplateDeltaLayer* oder *InstanceDeltaLayer*. Bei der Durchführung der Änderungstransaktion ruft er die Funktionen der durchzuführenden Änderungsoperationen entsprechend des gegebenen Änderungsszenarios auf. Beispielsweise wird bei der Prozessmodellierung für jede Änderungsoperation zuerst die strukturelle Verträglichkeit geprüft, und anschließend werden die strukturellen Änderungen durchgeführt. Dagegen wird bei instanzspezifischen Änderungen die strukturelle Verträglichkeit vor der Zustandsverträglichkeit geprüft und erst bei erfolgreichem Abschluss beider Überprüfungen die erforderlichen Schematransformationen und Zustandsanpassungen vorgenommen. Entsprechend werden bei der Schemaevolution unveränderter Prozessinstanzen nur die Zustandsverträglichkeit geprüft und -anpassungen durchgeführt.

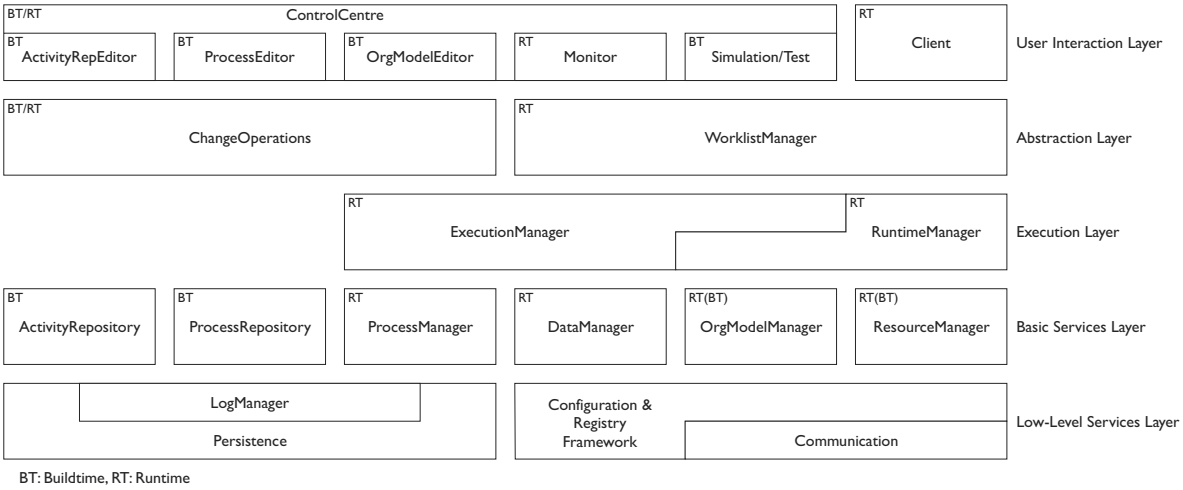


Abbildung 7.43: Gesamtarchitektur

7.3.10 Weitere Dienste

Nachdem wir in den vorangegangenen Abschnitten die wichtigsten Dienste eines adaptiven und flexiblen PrMS vorgestellt haben, gehen wir im folgenden auf weitere Dienste ein. Dazu gehören unter anderem Dienste, die das Rückgrat des Gesamtsystems bilden, aber selbst nichts mit Prozessmanagement zu tun haben.

7.3.10.1 Konfiguration (*Configuration*) und Registrierung (*Registry*)

Um die zahlreichen PrMS-Dienste einfach verwalten zu können, besitzen alle Dienste (außer den Anwendungsprogrammen im *User Interaction Layer* in Abb. 7.43) einen Lebenszyklus, der über die Schnittstelle *ADEPT2Service* etabliert wird. Diese Schnittstelle umfasst Methoden für die Initialisierung, das Starten und das Beenden von Diensten sowie Informationen zu Abhängigkeiten zwischen Diensten, etwa dass ein *ExecutionManager* einen *ProcessManager* benötigt. Durch die expliziten Abhängigkeiten wird gewährleistet, dass ein Dienst ordnungsgemäß gestartet und vollständig verfügbar ist, bevor er von einem anderen Dienst verwendet wird. Der Lebenszyklus der Dienste wird vom *Configuration & Registry Framework* verwaltet. Dieses realisiert somit eine allgemeine Infrastruktur für beliebige Dienste.

Konfiguration von Diensten Zum Lebenszyklus von Diensten gehört ihre Konfiguration. Konfigurationen sind individuell für jeden Dienst und werden ihm (genauer der Dienstinstanz) bei seiner Erzeugung übergeben. Somit sind die Dienste unabhängig davon, in welcher Form ihre Konfigurationen vorliegen, etwa als Dateien oder in Datenbanktabellen, und die Dienste müssen sich nicht selbst um das Laden und Speichern kümmern. Dies übernimmt eine allgemeine Konfigurationsverwaltung. Diese wird vom *Configuration & Registry Framework* übernommen, das zahlreiche Anforderungen hinsichtlich flexibler Konfigurierbarkeit abdeckt. So kann es neben unterschiedlichen Formen, in denen Konfigurationen vorliegen können, auch mit zentralen und dezentralen Konfigurationen umgehen. Auch beliebige Mischformen sind möglich, etwa dass Teile einer Dienstkonfiguration in einer zentralen Datenbank abgelegt sind und andere Teile derselben Dienstkonfiguration von der Konfigurationsverwaltung aus einer lokalen Datei geladen werden.

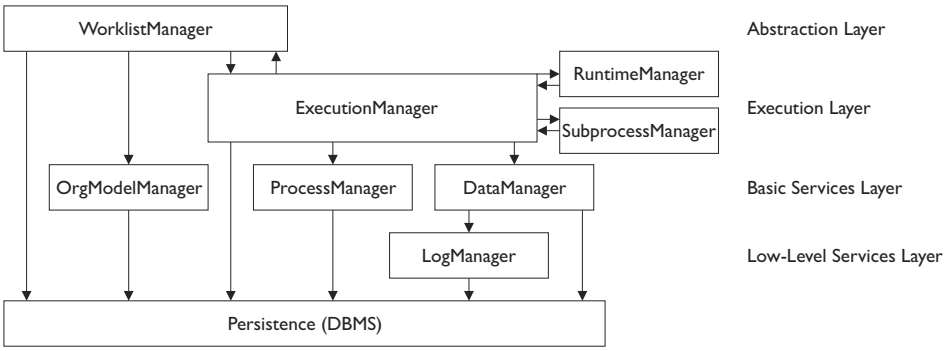


Abbildung 7.44: Verwendungsbeziehungen zwischen Diensten

Verwendungsbeziehungen zwischen Diensten Eine Schwierigkeit der strikten Dienstorientierung (SOA) und der funktionalen Aufteilung des PrMS auf verschiedene Dienste bilden die zahlreichen Verwendungsbeziehungen zwischen Diensten. Abb. 7.44 zeigt beispielhaft ein so entstandenes Dienstgeflecht. Verwaltet man die Abhängigkeiten in einem zentralen Registrierung (*Registry*) und nicht in den Diensten selbst, ergibt sich eine große Flexibilität bei der Bildung solcher Dienstgeflechte. Will ein Dienst einen anderen Dienst verwenden, fordert er diesen von der *Registry* an.

Die *Registry* ist Teil des *Configuration & Registry Framework*, das selbst als Dienst realisiert ist. So wird die *Registry* wie jeder andere Dienst behandelt, das heißt, auch sie unterliegt demselben Lebenszyklus und erhält ihre Konfiguration bei ihrer Erzeugung. Da die *Registry* der erste Dienst des PrMS ist, der gestartet wird, wird sie jedoch individuell erzeugt und gestartet (sog. *Bootstrapping*).

Die Verwendungsbeziehungen zwischen Diensten werden von der *Registry* verwaltet, die entsprechenden Informationen werden in der *Registry*-Konfiguration abgelegt. Dienstgeflechte können somit einfach über die Konfiguration geändert werden. Die *Registry* unterscheidet dabei zwischen Diensttypen und Dienstinstanzen. Ein Diensttyp wird über seine Schnittstelle definiert, eine Dienstinstanz durch eine Implementierung dieser Schnittstelle sowie einen systemweit eindeutigen Namen. Die Schnittstellenimplementierung kann dabei technisch dieselbe Java-Klasse sein wie die Schnittstelle des Diensttyps; es kann mehrere Dienstinstanzen derselben Dienstimplementierung geben, die sich in ihrem Namen unterscheiden. Dadurch können nicht nur die Abhängigkeiten zwischen Diensten, genauer Dienstinstanzen, leicht angepasst werden, auch die Dienstinstanzen selbst sind einfach austauschbar. Dies erlaubt es beispielsweise, spezielle Anforderungen an die Organisationsverwaltung direkt in der Implementierung der Organisationsverwaltung zu berücksichtigen und die ursprüngliche Implementierung durch die angepasste Implementierung in der *Registry*-Konfiguration auszutauschen. Dazu ist es lediglich erforderlich, dass Dienste ihre verwendeten Dienste nicht anhand des Dienstnamens anfordern sondern anhand den Diensttyps.

Entfernte Aufrufe von Diensten über die *Registry* Das *Configuration & Registry Framework* arbeitet eng mit dem *Communication*-Dienst (s. Abschnitt 7.3.10.2) zusammen (s. Abb. 7.43). Dieser kapselt die Datenübertragung zwischen verschiedenen *Dienstknoten* kapselt. Dienstknoten sind beispielsweise verschiedene, durch ein Netzwerk verbundene Rechner oder mehrere JVM auf einem Rechner, die als getrennte Betriebssystemprozesse laufen. Ein Dienstknoten ist dadurch gekennzeichnet, dass auf ihm mindestens eine *Registry* gestartet ist,

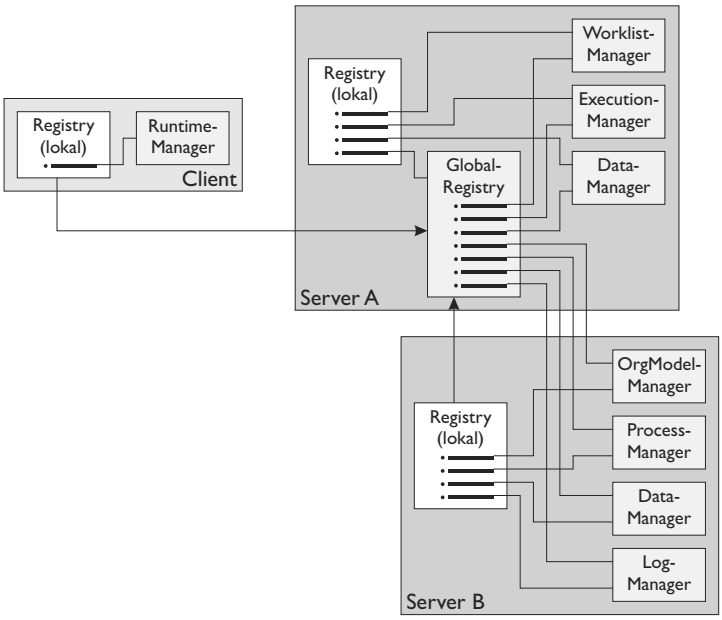


Abbildung 7.45: Verteilung von Diensten (aus [Ben08])

die die Dienste des Knotens verwaltet. Mittels *Communication* bietet diese *Registry* Zugriffe auf Dienste anderer Knoten. *Communication* geht dabei so weit, dass ein entfernter Dienst wie ein lokaler Dienst verwendet werden kann. Die Übertragung der Anfragen und Antworten wird vollständig von *Communication* gekapselt. Dazu überprüft die *Registry* beim Anfordern eines Dienstes, ob die entsprechende Dienstinstanz lokal vorhanden ist. Ist dies der Fall, wird sie zurückgegeben. Ist sie nicht lokal vorhanden, gibt es zwei Möglichkeiten: Entweder sind in der *Registry*-Konfiguration die notwendigen Informationen zum Knoten der Dienstinstanz hinterlegt (die entsprechende Adresse), oder die *GlobalRegistry* kennt die Dienstinstanz. Die *GlobalRegistry* ist ein weiterer Dienst des *Configuration & Registry Framework*, der ähnlich wie DNS (*Domain Name System*) funktioniert und registrierte Dienstinstanznamen auf die entsprechenden Adressen der Dienstinstanzen abbildet (s. Abb. 7.45).

In beiden Fällen bekommt die *Registry* die Information, wo und wie die Dienstinstanz erreichbar ist, und kann damit vom *Communication*-Dienst ein geeignetes Stellvertreterobjekt (*Proxy*, [GHJV95], hier in der Funktion als *Stub*) erhalten. Das Stellvertreterobjekt besitzt dieselbe Schnittstelle wie die entfernte Dienstinstanz und leitet sämtliche Aufrufe an die entfernte Dienstinstanz weiter. Dazu werden die Aufrufe transformiert und übertragen (s. Abschnitt 7.3.10.2).

Configuration & Registry

Angebotene Funktionen

- Verwalten von Diensten und ihres Lebenszyklus (Initialisierung, Starten und Beenden von Diensten)
- Konfiguration von Diensten und Verwaltung dieser Konfigurationen
- Auflösung und Herstellung von Abhängigkeiten zwischen Diensten, bei entfernten Diensten in Zusammenarbeit mit dem *Communication*-Dienst

Beispiel 7.23 (Globale und lokale Dienstregistrierung)
In einem Einsatzszenario eines PrMS gebe es die Server A und B sowie einen *Client* (s. Abb. 7.45). Jeder besitzt eine lokale *Registry*, die verschiedene Dienste verwaltet, sowie mindestens einen *Communication*-Dienst zu jeder *Registry*. Alle Serverdienste wurden nach ihrem Start von der lokalen Registry in der *GlobalRegistry* registriert und sind damit über ihren Dienstinstanznamen global erreichbar, ohne dass ihre genaue Adresse bekannt sein muss. Auf dem *Client* läuft nur ein *RuntimeManager*. Dieser ist nicht in der *GlobalRegistry* hinterlegt, da er nur für die lokale Ausführung zuständig ist. Die *Registry* des *Client* kennt außer der *GlobalRegistry* keine weiteren Serverdienste, sie kann diese jedoch bei Bedarf über ihren Namen erfragen und auf sie zugreifen.

7.3.10.2 Datenaustausch (*Communication*)

Während Verwendungsbeziehungen zwischen Diensten von der *Registry* verwaltet werden, kapselt der *Communication*-Dienst entfernte Dienstaufrufe und stellt Funktionen bereit, damit entfernte Aufrufe sich möglichst wenig von lokalen Aufrufen unterscheiden. Er überbrückt und versteckt damit die Grenzen zwischen verschiedenen Dienstknoten, etwa verschiedenen Rechnersystemen oder JVM. Lediglich die Zeitverzögerung für die Datenübertragung sowie eventuell auftretende Probleme in Zusammenhang mit dem entfernten Aufruf, etwa Verbindungstrennungen, können nicht durch den *Communication*-Dienst verborgen werden.

Der *Communication*-Dienst ist ein eigenständiger Dienst und wird dementsprechend behandelt, d. h. er durchläuft den normalen Lebenszyklus, und er ist über eine Schnittstelle definiert. Dadurch ist hier dieselbe Flexibilität wie bei anderen Diensten möglich, insbesondere können mehrere Instanzen verschiedener *Communication*-Dienste gleichzeitig aktiv sein und verwendet werden. Damit kann beispielsweise der *Client* in Abb. 7.45 mit dem Server A auf andere Weise kommunizieren wie mit Server B. Umgekehrt können Dienste auf einem Server auch gleichzeitig auf verschiedene Arten entfernt aufgerufen werden.

CommunicationStack Im Unterschied zu anderen Diensten wird der *Communication*-Dienst anderen Diensten nicht direkt zur Verfügung gestellt; er wird lediglich von der (lokalen) *Registry* verwendet. Somit wird er sinnvollerweise auch nicht entfernt aufgerufen. Seine Aufgabe ist die Bereitstellung von *Stub*s und *Skeleton*s für die anderen Dienste. *Stub*s sind Objekte, die ein entferntes Objekt lokal repräsentieren, dabei aber lokal keine Anwendungslogik und meistens auch keine Daten vorhalten, sondern immer das entfernte Objekt verwenden. Dazu transformieren sie Methodenaufrufe in eine spezielle Repräsentation, die über eine Netzwerkschnittstelle übertragen wird. *Skeleton*s sind die Gegenstücke hierzu. Sie wandeln die über die

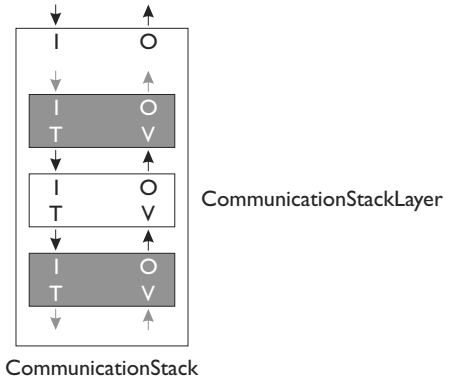


Abbildung 7.46: *CommunicationStack*

Netzwerkschnittstelle empfangenen Daten wieder in Methodenaufrufe um und leiten diese an das eigentliche Objekt weiter. Besitzt ein Methodenaufruf einen Rückgabewert, wird dieser in der umgekehrten Richtung vom *Skeleton*s transformiert, übertragen und vom *Stub* wieder in die ursprüngliche (Objekt-)Repräsentation umgewandelt [TvS06].

Die *Stub*s und *Skeleton*s von *Communication* beruhen auf *CommunicationStack*s (s. Abb. 7.46). Das sind ein oder mehrere Schichten (*CommunicationLayer*) in denen Daten transformiert werden. Jede Schicht nimmt Anfragen vom Typ `I` entgegen, transformiert diese in Daten des Typs `T` und leitet sie an die nächste Schicht weiter. Antworten auf eine Anfrage, etwa der Rückgabewert einer Methode, haben den Typ `V` und werden von der nächsten Schicht nach oben zurückgegeben. Dort werden sie in Daten des Typs `O` transformiert und wiederum zur nächst höheren Schicht weitergeleitet.

Mit *CommunicationStack*s lässt sich die Transformation der Daten leicht anpassen, indem *CommunicationLayer* unterschiedlich kombiniert werden. Es ist lediglich darauf zu achten, dass zusammengehörige *Stub*s und *Skeleton*s zueinander passende (komplementäre) *CommunicationStack*s verwenden. Nur so kann die erzeugte und übertragene Datenrepräsentation wieder korrekt rücktransformiert werden. In der Regel wandelt ein *CommunicationStackLayer* die übertragenen Daten in einen Bytestrom (und zurück) um (*Objektserialisierung*/*Objektdeserialisierung*), ein weiterer *CommunicationStackLayer* bedient die Netzwerkschnittstelle. Falls gewünscht, kann eine zusätzliche Schicht die Daten nach der Serialisierung verschlüsseln, etwa mittels SSL (*Secure Sockets Layer*). Seitens des Skeletons wird dann eine entsprechende Schicht benötigt, die die Daten vor der Deserialisierung wieder entschlüsselt. Auch die (De-)Serialisierung kann einfach ausgetauscht werden, etwa indem man die Daten in XML-Strukturen transformiert, anstatt die von Java bereitgestellte Objektserialisierung zu nutzen.

Kopien und Referenzen Für die Übertragung von Daten ist es meistens ausreichend, diese zu serialisieren und damit auf dem entfernten Dienstknoten eine Kopie der Daten zu erhalten (s. Abb. 7.47 links). Dies ist jedoch dann nicht erwünscht, wenn die Datenmenge groß ist oder die Objektidentität über Dienstknoten hinweg gewährleistet sein muss und sich Objektänderungen, die auf einem Dienstknoten erfolgen, auf allen anderen Dienstknoten widerspiegeln müssen (ein spezieller Fall von *Singleton*, [GHJV95]). Hierfür wird auf dem Dienstknoten, auf dem das *Singleton*-Objekt liegt, ein entsprechendes *Skeleton*-Objekt erzeugt und auf dem entfernten Dienstknoten ein *Stub* (s. Abb. 7.47 rechts). So existiert das *Singleton*-Objekt nur auf einem Dienstknoten und Änderungen sind sofort auf allen referenzierenden Knoten verfügbar.

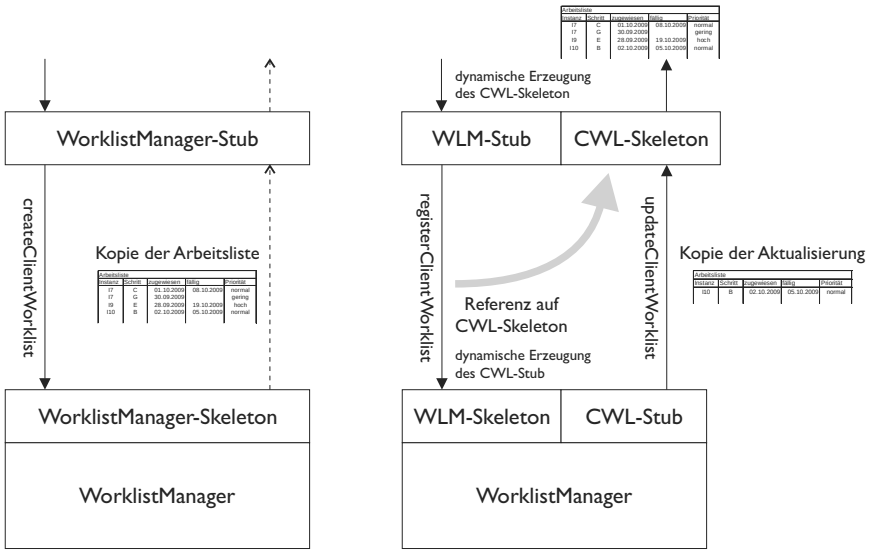


Abbildung 7.47: Übertragung als Kopie und als Referenz

Allerdings ist die Verwendung des Objekts aufwendiger, da sämtliche Anfragen an einen bestimmten Dienstknoten übertragen werden müssen. Dieses Verfahren wird beispielsweise bei *Callback*s angewendet. *Callback*-Objekte werden durch einen Methodenaufruf bei einem anderen (entfernten) Objekt registriert und von diesem über Zustandsänderungen informiert.

Neben der Kapselung von entfernten Aufrufen, wird auch die Unterscheidung zwischen der Übertragung als Kopie und als Referenz vollständig vom *Communication*-Dienst vor den verwendenden Diensten verborgen. Hierfür erweist sich die Trennung zwischen Daten und Anwendungslogik als nützlich; Daten werden generell als Kopien übertragen, Anwendungslogik als Referenzen. Naheliegenderweise gehören *Callback*s zur Anwendungslogik und nicht zu den Daten.

Beispiel 7.24 (Arbeitslistenaktualisierung über Kopien und *Callbacks*)
Arbeitslisten können aktualisiert werden, indem die komplette Arbeitsliste vom *WorklistManager* geholt wird (s. Abb. 7.47 links). Erfolgt dies über Dienstknotengrenzen hinweg, wird die Arbeitsliste vom *Stub* des *WorklistManager* angefordert. Der entsprechende *Skeleton* überträgt eine Kopie der Arbeitsliste. Für eine asynchrone Aktualisierung einer Arbeitsliste wird eine Arbeitsliste beim entsprechenden *WorklistManager* als *Callback*-Objekt registriert. Erfolgt die Registrierung über Dienstknotengrenzen hinweg (s. Abb. 7.47 rechts), wird bei der Registrierung im *WorklistManager-Stub* (*WLM-Stub*) ein entsprechender *Skeleton* (*CWL-Skeleton*) für die Arbeitsliste erzeugt. Der *WLM-Stub* überträgt die Adresse des *CWL-Skeleton* an den *WorklistManager-Skeleton* (*WLM-Skeleton*). Dieser erzeugt einen *Stub* (*CWL-Stub*) für diese Adresse und registriert den *CWL-Stub* im *WorklistManager* für Aktualisierungen. Für den *WorklistManager* verhält sich der *CWL-Stub* wie ein vollwertiges Arbeitslistenobjekt. Arbeitslistenaktualisierungen werden vom (*CWL-Stub*) serialisiert und an den *CWL-Skeleton* übertragen. Dieser deserialisiert die Kopie der Aktualisierungsdaten und reicht diese an das tatsächliche Arbeitslistenobjekt weiter.

Der Austausch von Objekten durch entsprechende Referenzen funktioniert nicht nur zwischen aufrufendem und aufgerufenem Dienst, also bei Methodenparametern, sondern auch bei

Communication

FA 12 Programmierschnittstelle API, Entwicklungsplattform

Angebotene Funktionen

- Kapselung von entfernten Aufrufen
- Übertragung sowohl von Datenkopien als auch von Referenzen
- Unterschiedliche Datentransformationen bei der Datenübertragung

Rückgabewerten von Aufrufen. Dies ist unter anderem nützlich, wenn ein Dienst umfangreiche Daten von einem anderen Dienst anfordert, etwa umfangreiche Arbeitslisten. Hierzu wird nicht die komplette Liste übertragen sondern eine Referenz auf ein *Iterator*-Objekt. Mit dem *Iterator*-Objekt werden dann die Elemente der Liste elementweise oder in Gruppen übertragen. Dies ist vergleichbar mit `java.sql.ResultSet`, das über JDBC (*Java Database Connectivity* [jdb]) den iterativen Zugriff auf Inhalte von Datenbanken ermöglicht.

Webdienste (*Web Services*) Ein Spezialfall eines *Communication*-Dienstes sind Webdienste (*Web Services* [ACKM04]). Mit diesen ist ein standardisierter Datenaustausch zwischen verschiedenen Programmiersprachen und Ausführungsplattformen möglich. Die Programmiersprachen- und Ausführungsplattform-unabhängige Verwendung ist im weiteren Sinne Teil der funktionalen Anforderung FA 12 (Entwicklungsplattform).

Damit der Datenaustausch zwischen verschiedenen Programmiersprachen und Ausführungsplattformen funktioniert, müssen die entsprechenden Dienstschnittstellen und Datenmodelle in Form von WSDL- (*Web Services Description Language*) und XSD-Dokumenten (*XML Schema*) vorliegen. Diese Dokumente können zur Laufzeit beim *Communication*-Dienst angefragt werden. Im Gegensatz zu einem normalen *Communication*-Dienst werden bei Webdiensten *Stub*s und *Skeleton*s nicht automatisch generiert. Dies ist auch nicht notwendig, da Webdienste direkt die unterste Ebene der Kommunikation definieren. Stattdessen gibt es vordefinierte *Skeleton*s, die die Webdienstschnittstelle, etwa die über HTTP übertragenen XML-Daten in einen Dienstaufruf konvertieren. Zusätzlich gibt es vordefinierte *Stub*s, die für *Callback*s verwendet werden. Will ein Dienstknoten *Callback*s nutzen, muss er einen entsprechenden Webdienst bereitstellen, der von den *Stub*s aufgerufen wird

Da sich diese Unterschiede des *Communication*-Dienstes für Webdienste auf Implementierungs- und nicht auf Schnittstellenebene befinden, wird der *Communication*-Dienst für Webdienste wie jeder andere *Communication*-Dienstes verwendet.

7.3.10.3 Verwaltung von Ressourcen (*ResourceManager*)

Die Ressourcenverwaltung setzt Teile der funktionalen Anforderung FA 5 (Prozessüberwachung) um, indem sie Arbeitsschritte überwacht, deren Ausführung plant und vor Ressourcenengpässen warnt. Dabei achtet sie beispielsweise auf die Einhaltung minimaler und maximaler Zeitabstände zwischen Prozessschritten [LWR14]. Unter Berücksichtigung geschätzter Ausführungsdauern analysiert sie vorhandene zeitliche Einschränkungen und warnt ggf. vor Verletzungen. Zusätzlich werden zur Ausführung benötigte Ressourcen berücksichtigt, die nur in begrenzter Anzahl zur

ResourceManager

FA 5 Prozessüberwachung

Angebotene Funktionen

- Überwachung zeitlicher Einschränkungen auf Prozessinstanzen unter Berücksichtigung von Zeit und Ressourcen
- Warnung vor der Verletzung zeitlicher Einschränkungen
- Planung und Einflussnahme auf die Ausführungsreihenfolge aktiver Prozessschritte

Verfügung stehen, etwa bestimmte Räume oder Maschinen. Auch die Nichtverfügbarkeit von Ressourcen kann zur Verletzung zeitlicher Einschränkungen führen.

Um solche Verletzungen weitgehend auszuschließen, kann die Ressourcenverwaltung die Ausführungsreihenfolgen aller aktuellen und zukünftigen Prozessschritte planen und so eine möglichst optimale Ausführungsreihenfolge bestimmen. Damit kann sie den Benutzern Hilfestellung bei der Auswahl des nächsten auszuführenden Prozessschritts geben. Dies kann beispielsweise über die Anpassung von Prioritäten von Arbeitslisteneinträgen erfolgen. Sie arbeitet deshalb eng mit der Arbeitslistenverwaltung zusammen, etwa auch, um mittels Eskalation vor der Verletzung zeitlicher Einschränkungen zu warnen [vdARD07].

Aufgrund der zahlreichen Aspekte, die hier zu berücksichtigen sind, gibt es noch kein vollständiges Metamodell. Auch Teilaspekte sind noch Gegenstand aktueller Forschung ([CP10, LY05]).

7.3.10.4 Historienverwaltung (*LogManager*)

Die Historienverwaltung (*LogManager*) befindet sich in der Architektur auf unterer Ebene. Sie dient dazu, verschiedene Protokolle zu verwalten und zu speichern. Durch die Protokollierung (FA 2) der wichtigsten Funktionsaufrufe und Zustandsänderungen im System realisiert sie Teile der funktionalen Anforderung FA 5 (Prozessüberwachung) und ermöglicht nachträgliche Analysen (FA 5.2). Durch die Speicherung wird auch ein Teil von FA 6 (persistente Speicherung) umgesetzt.

Protokolle und deren Inhalte sind anwendungsspezifisch. Protokolliert werden sinnvollerweise Änderungen an Daten, etwa Zustandsänderungen von Prozessinstanzen und schreibende Zugriffe auf Datenelemente. Bei einem adaptiven PrMS sollten auch strukturelle Änderungen an Prozessvorlagen und -instanzen protokolliert werden. bietet es sich aus Sicherheitsgründen an, Änderungen am Organisationsmodell und lesende Zugriffe auf Datenelemente zu erfassen. Eine umfassende Protokollierung ermöglicht zahlreiche Analysen, etwa statistische Auswertungen von Prozessausführungen wie die Ermittlung durchschnittlicher Kosten für bestimmte Prozesse.

Damit alle Protokolle in ähnlicher Weise verwaltet werden, bietet der *LogManager* eine allgemeine Schnittstelle für Protokolle. Diese bewegt sich etwas oberhalb einer SQL-Datenbankschnittstelle und erweitert die flachen Strukturen relationaler Datenbanken um hierarchische Datenstrukturen. Neben einer solchen allgemeinen Schnittstelle bietet der *LogManager* auch noch die Möglichkeit, anwendungsspezifische Schnittstellen zu hinterlegen, etwa eine spezielle Schnittstelle für die Protokollierung von Prozessausführungen. Intern beruhen

LogManager

FA 2	Protokollierung
FA 5.2	Prozessanalysen
FA 5	Prozessüberwachung
FA 6	Persistente Speicherung

Angebotene Funktionen

- Persistente Speicherung von Historiendaten mit oftmals hierarchischen Datenstrukturen
- Zugriff auf Historiendaten für umfassende Analysen

die anwendungsspezifischen Schnittstellen auf der allgemeinen Schnittstelle. Sie vereinfachen aber die Verwendung des *LogManager*s in den entsprechenden Diensten bedeutend.

Beispiel 7.25 (Hierarchische Daten im Änderungsprotokoll)
Strukturelle Prozessänderungen finden im Rahmen einer von einem Benutzer ausgelösten Änderungstransaktion statt. Diese umfasst gewöhnlich mehrere Änderungsoperationen, welche wiederum aus mehreren Änderungsprimitiven bestehen. Um die Änderungstransaktion vollständig nachvollziehen zu können, müssen alle diese Informationen sowie ihr Zusammenhang protokolliert werden. Die Verwendung hierarchischer Datenstrukturen *LogManager* vereinfacht dies.

7.3.10.5 Datenhaltung (*Persistence*)

Die funktionale Anforderung FA 6 (persistente Speicherung) kann auf verschiedene Arten realisiert werden. So können die entsprechenden Dienste die Speicherung ihrer Daten selbst übernehmen und diese beispielsweise im XML-Format in Dateien ablegen. Dies kann für bestimmte Daten ausreichend und sinnvoll sein. Meistens ist aber die Speicherung in einem DBMS vorzuziehen. Hierfür bietet der *Persistence*-Dienst eine geeignete Abstraktion.

Der *Persistence*-Dienst stellt eine zu JDBC vergleichbare Schnittstelle zur Verfügung. Zusätzlich bietet er die Möglichkeit, Verbindungen zu DBMS zwischenzuspeichern, um diese wiederzuverwenden und somit den Aufwand für deren Erzeugung zu sparen. Der Verbindungsaufbau ist im Vergleich zu Anfragen an ein DBMS über eine Verbindung vergleichsweise teuer. So zahlt sich die Wiederverwendung beispielsweise dann aus, wenn der *ProcessManager* direkt nacheinander zwei verschiedene Prozessinstanzen aus demselben DBMS holt. Außerdem abstrahiert *Persistence* mit einfachen Mitteln von herstellerspezifischen Eigenschaften eines DBMS, etwa von verschiedenen SQL-Dialekten. Dadurch werden die verwendenden Dienste unabhängig vom eingesetzten DBMS. Dies erlaubt es, das PrMS mit verschiedenen DBMS zu betreiben, was eine große Flexibilität hinsichtlich der Anforderungen an die Systemumgebung ermöglicht.

7.4 Programmierschnittstelle (API)

Die Programmierschnittstelle (API, FA 12) umfasst die Schnittstelle der Dienste sowie die Datenmodelle. Sie stellt alle Funktionen eines adaptiven PrMS zur Verfügung und dient somit als

Persistence

FA 6 Persistente Speicherung

Angebotene Funktionen

- Kapselung der persistenten Speicherung in einem DBMS
- Abstraktion von DBMS-spezifischen Eigenschaften

Rahmen zur Erstellung von Klientenprogrammen und prozessbasierten Informationssystemen. Beispiele für solche Implementierungen insbesondere unter Nutzung der Änderungsoperationen sind in [LKRD10] und in Kapitel 9 aufgeführt.

Die Programmierschnittstelle kann entweder als Bibliothek (s. Abschnitt 7.4.1) oder programmiersprachenunabhängig mittels Webdiensten (*Web Services*, s. Abschnitt 7.4.2) verwendet werden. In den folgenden Abschnitten zeigen wir anhand dreier Beispiele die Verwendung der Programmierschnittstelle und erläutern anschließend die Gemeinsamkeiten und Unterschiede beider Verwendungsarten.

7.4.1 Verwendung als Bibliothek

Die Funktionen des PrMS können mittels einer Bibliothek direkt aus beliebigem Programmcode aufgerufen werden, sofern die Bibliothek in einer Form vorliegt, die zu der jeweiligen Programmiersprache kompatibel ist. Dazu stellt die Bibliothek die Datenmodelle sowie die Schnittstellen der Dienste zur Verfügung. Außerdem enthält sie die für den Zugriff auf die Dienstschnittstellen notwendige Infrastruktur. Hierzu gehören die Dienste Konfiguration und Registrierung (s. Abschnitt 7.3.10.1) sowie Datenaustausch (s. Abschnitt 7.3.10.2). Die Infrastruktur muss vor der Verwendung der Dienstschnittstellen einmal initialisiert werden (s. `BootstrapRegistry.load` in Abb. 7.48). Dabei wird die lokale Konfiguration eingelesen und die Kommunikation mit dem PrMS in7itialisiert. In der Konfiguration ist beispielsweise hinterlegt, wo und auf welche Art die Dienste des PrMS aufgerufen werden. Anschließend kapselt die Infrastruktur die Kommunikation mit den Diensten und verbirgt so, dass Dienste ggf. entfernt aufgerufen werden.

Aufgrund der Trennung zwischen Schnittstelle und Implementierung von Diensten ist gewährleistet, dass sich die Implementierung der Dienste jederzeit ändern kann, ohne dass dadurch Änderungen an der Bibliothek oder gar am aufrufenden Programm notwendig werden. Im folgenden zeigen wir die Verwendung der Programmierschnittstelle als Bibliothek anhand der Anmeldung eines Benutzers am System, des Starts einer Prozessinstanz sowie dem Ändern einer Prozessinstanz.

7.4.1.1 Initialisierung und Anmelden am System

Abb. 7.48 zeigt in einem UML 2-Sequenzdiagramm den Ablauf der Initialisierung der Bibliothek und der Anmeldung eines Benutzers am PrMS. Im ersten Schritt wird die Konfiguration eingelesen und die lokal benötigten Dienste initialisiert und gestartet (`1: load` und `1.1: initialise`). Dies ist nur einmal vor dem ersten Zugriff auf das PrMS notwendig. Anschließend wird der *SecurityManager* (s. Abschnitt 7.3.3) angefordert (`2: getSecurityManager`) und

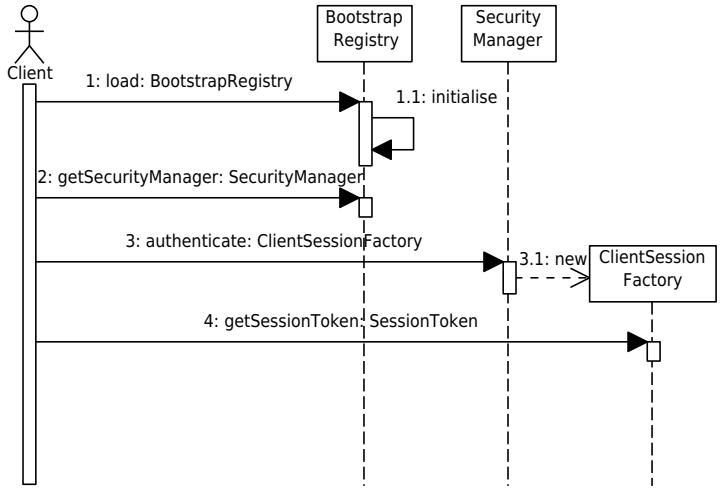


Abbildung 7.48: UML-Sequenzdiagramm des Ladens der Bibliothek und Anmeldens am System

über diesen die Anmeldung mit den Benutzerdaten (Benutzername und Passwort) vorgenommen (`3: authenticate`). Naheliegenderweise ist dies ein entfernter Aufruf, der wie in Abschnitt 7.3.10.2 detailliert beschrieben von der Infrastruktur gekapselt wird.

Sind die Benutzerdaten korrekt, erzeugt der *SecurityManager* seitens des PrMS eine *ClientSessionFactory* (s. Abschnitt 7.3.3.3) und gibt diese zurück. Mit der *ClientSessionFactory* erzeugt der Verwender der Bibliothek *SessionToken* (`4: getSessionToken`), die für meisten Zugriffe auf die Programmierschnittstelle benötigt werden. Anhand der *SessionToken* überprüfen die einzelnen Dienste, ob der entsprechende Benutzer für den jeweiligen Funktionsaufruf berechtigt ist.

7.4.1.2 Starten einer Prozessinstanz

Das Starten einer Prozessinstanz zeigt Abb. 7.49. Dazu wird der *ExecutionManager* (s. Abschnitt 7.3.6) angefordert (`1: getExecutionManager`) und von diesem der Unterdienst *InstanceControl* (`2: getInstanceControl`) . Da Prozessinstanzen normalerweise nur von bestimmten Benutzern gestartet werden können, wird zum Starten ein *SessionToken* benötigt. Dieses wird von der *ClientSessionFactory* geholt (`3: getSessionToken`), die von der Anmeldung am System stammt (s. Abschnitt 7.4.1.1). Anschließend wird eine neue Prozessinstanz gestartet (`4: createAndStartInstance`). Dabei legt der *ExecutionManager* die entsprechenden Datenstrukturen im PrMS an und startet die Prozessinstanz, indem die Zustände der Prozessschritte gesetzt werden.

7.4.1.3 Durchführung instanzspezifischer Änderungen

Um instanzspezifische Änderungen durchzuführen (s. Abb. 7.50), wird zunächst die zu ändernde Prozessinstanz geholt. Dies erfolgt über *InstanceChanging* (`2: getInstanceChanging`) des *ExecutionManager* (`1: getExecutionManager`). Die Berechtigung zum Ändern der Prozessinstanz wird wieder über ein *SessionToken* (`3: getSessionToken`) überprüft. Mit diesem *SessionToken* wird die zu ändernde Prozessinstanz geholt (`4: getInstanceForChanging`). Der Dienst *ChangeOperations* (s. Abschnitt 7.3.9) bietet die Änderungsoperationen, die auf Prozessinstanzen angewendet werden (`5: getChangeOperations`). Abhängig von der Anzahl

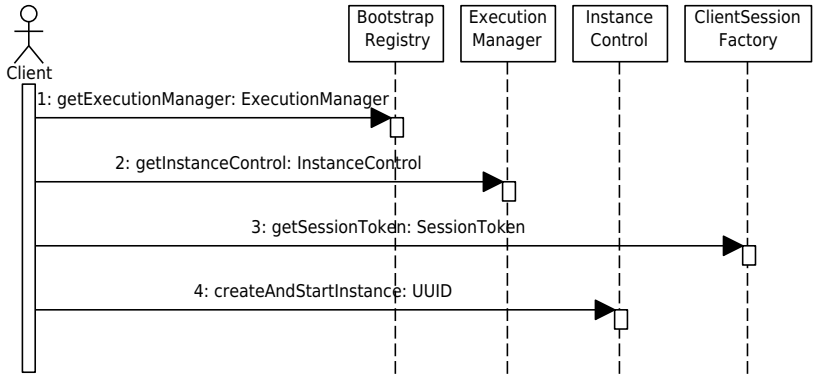


Abbildung 7.49: UML-Sequenzdiagramm des Startens einer Prozessinstanz

der durchzuführenden Änderungsoperationen sind die Schritte 6–8 mehrmals zu wiederholen. Diese umfassen das Holen der gewünschten Änderungsoperation von *ChangeOperations* (`6: getChangeOperation`), die Überprüfung auf die Anwendbarkeit der Änderungsoperation (strukturelle Verträglichkeit und Verträglichkeit bezüglich des Zustands, `7: isPossible`) sowie die eigentliche Änderung der Prozessinstanz (`8: performOperation`). Die Änderung umfasst die Anpassung der Datenstrukturen der Prozessinstanz entsprechend der Änderungsoperation und der verwendeten Änderungsprimitive. Sind alle Änderungen an der Prozessinstanz durchgeführt worden, wird die geänderte Prozessinstanz wieder ins System zurückgespielt (`9: changeInstance`). Nach dem Zurückspielen wird die Ausführung der nun geänderten Prozessinstanz vom *ExecutionManager* wieder aufgenommen.

7.4.2 Programmiersprachenunabhängige Verwendung (*Web Services*)

Die Verwendung der Programmierschnittstelle als Bibliothek erfordert, dass die Bibliothek in einer für den Aufrufer kompatiblen Form vorliegt, etwa in derselben Programmiersprache. Diese Einschränkung umgehen Webdienste (*Web Services*), die programmiersprachen- und ausführungsplattform-unabhängig sind (s. Abschnitt 7.3.10.2). Dabei wird die Programmierschnittstelle entsprechend den Webdienststandards beispielsweise über SOAP auf einem Webserver zentral bereitgestellt und mittels Daten in XML aufgerufen. Die Beschreibungen der Schnittstellen des PrMS werden als WSDL-Dokumente, die Datenmodelle als XSD-Dokumente vom selben Webserver bereitgestellt. Mittels dieser WSDL/XSD-Repräsentation können zahlreiche Programmiersprachen und Ausführungsumgebungen Coderümpfe (*Stub*s) erzeugen, die die Webdienststandards so kapseln, dass direkte Aufrufe innerhalb der Programmiersprache/Ausführungsumgebung möglich sind [Ben08]. Damit wird die Verwendung der Webdienstschnittstelle mit der Verwendung als Bibliothek vergleichbar.

Unterschiede zwischen der Verwendung als Bibliothek und der programmiersprachenunabhängigen Verwendung mittels Webdiensten ergeben sich dadurch, dass die Schnittstellen für Webdienste aufgrund der Webdienststandards strikt zwischen Funktionen und Daten unterscheiden müssen: Funktionen werden auf dem Server aufgerufen, Daten werden zwischen Aufrufer und Server ausgetauscht. Es ist nicht wie innerhalb von Programmiersprachen möglich, Objekte zu verwenden, die sowohl Daten als auch Anwendungslogik enthalten.

Die größten Änderungen der Verwendung als Webdienst ergeben sich damit an den Schnittstellen, die solche Objekte zurückgeben. Dazu gehört die Aufteilung sämtlicher Dienste in

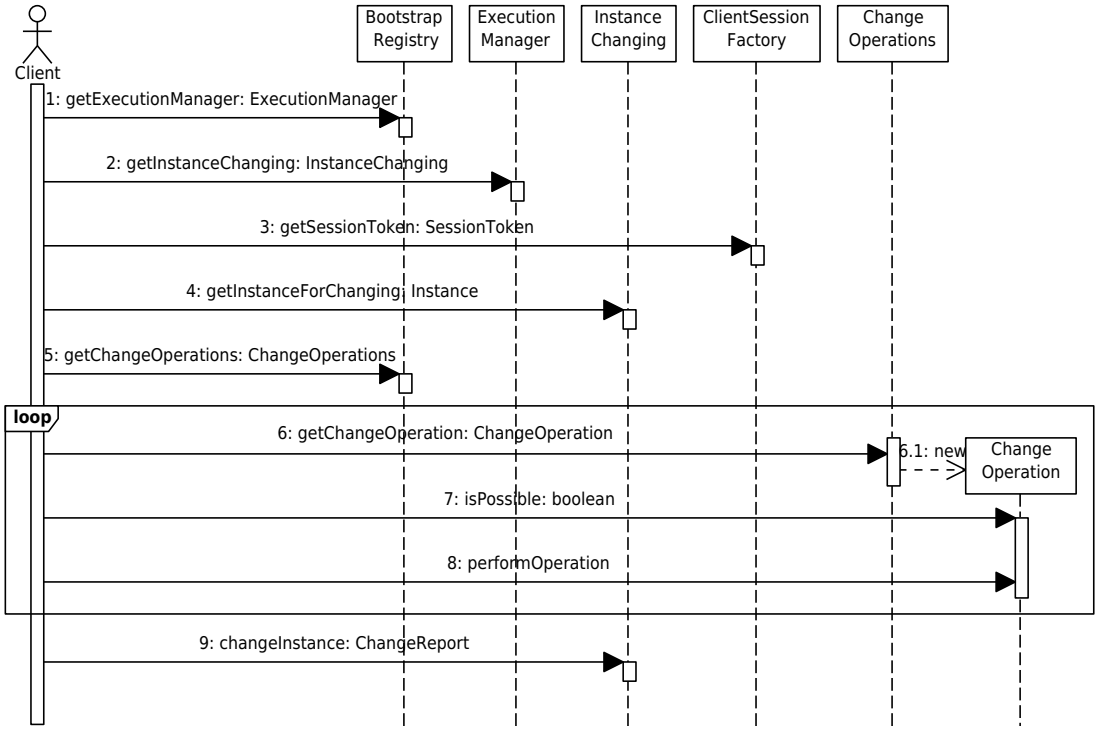


Abbildung 7.50: UML-Sequenzdiagramm des instanzspezifischen Änderns einer Prozessinstanz

Unterdienste, etwa *InstanceControl* als Unterdienst des *ExecutionManager*. Da Unterdienste nur der logischen Unterteilung der Funktionen eines Dienstes dienen, reicht es aus, die Unterteilung über die Namensgebung von Webdienstoperationen zu repräsentieren. Beispielsweise heißt die Operation zum Starten einer Prozessinstanz `instContCreateAndStartInstance`, und ist damit die Kurzform von `ExecutionManager.getInstanceControl.createAndStart-Instance` (s. Abschnitt 7.4.1.2).

Daneben gibt es noch einige wenige Stellen in der Programmierschnittstelle des PrMS, die Objekte mit Anwendungslogik zurückgeben, beispielsweise die *ClientSessionFactory* (s. Abschnitt 7.4.1.1). Die *ClientSessionFactory* wird über Webdienste dahingehend reduziert, dass nur noch ihre Daten *ClientSessionFactory* ausgetauscht werden. Die Logik für die Erzeugung von *SessionToken* muss vom Verwender der Webdienstschnittstelle bereitgestellt werden.

Beispiel 7.26 (XSD für die Webdienstrepräsentation einer *ClientSessionFactory*)

```
<xs:complexType name="ClientSessionDetails">
  <xs:sequence>
    <xs:element name="agent" type="tns:RichAgent" />
    <xs:element name="token" type="xs:base64Binary" />
    <xs:element name="uris" type="bsc:URIList" minOccurs="0" />
    <xs:element name="addAttributes" type="bsc:StringMap"
                minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Von diesen Ausnahmen abgesehen, entsprechen sich die Schnittstellen beider Verwendungsarten. Dies ermöglicht sowohl einen einfachen Wechsel der Verwendungsart bezüglich der Implementierung als auch bezüglich der Implementierer. Das heißt, die Umstellung der Implementierung ist relativ einfach möglich, während gleichzeitig die Wissensbasis für Implementierer weitgehend unabhängig von der Verwendungsart der Programmierschnittstelle ist.

7.5 Umsetzung der funktionalen Anforderungen

In den vorherigen Abschnitten haben wir die Architektur eines adaptiven PrMS im Detail vorgestellt und sind dabei auch darauf eingegangen, wie die Dienste und Datenmodelle die in Abschnitt 3.2 aufgeführten funktionalen Anforderungen umsetzen. In diesem Abschnitt fassen wir dies nochmals zusammen und zeigen, wie sich die funktionalen Anforderungen auf die Dienste und Datenmodelle verteilen.

Die **Prozessausführung** (FA 1) findet hauptsächlich in *ExecutionManager* (s. Abschnitt 7.3.6) und *RuntimeManager* (s. Abschnitt 7.3.8) statt. Ausgeführt werden eigentlich nur Prozessschritte, was im *RuntimeManager* erfolgt. Da Prozessschritte aber innerhalb eines Prozesses ausgeführt werden, müssen auch die Abhängigkeiten zwischen den Prozessschritten berücksichtigt werden. Dies übernimmt der *ExecutionManager*. Dieser interpretiert Prozessinstanzen und ist für deren Zustandsänderungen verantwortlich. Hierzu gehört auch das Bewerten von Entscheidungsergebnissen bei alternativen Verzweigungen und die entsprechende Konsequenzen für die weitere Ausführung von Prozessschritten. Der *ExecutionManager* benötigt dazu die Daten der Prozessinstanzen aus dem *ProcessManager* (s. Abschnitt 7.3.4) und für die Interaktion mit dem *RuntimeManager* und alternative Verzweigungen auch die Daten des *DataManager* (s. Abschnitt 7.3.5). Insofern zählen auch *ProcessManager* und *DataManager* zur FA 1.

Grundlegend für die **Protokollierung** (FA 2) ist der *LogManager* (s. Abschnitt 7.3.10.4). Dieser bietet den verwendenden Diensten eine Schnittstelle für die einfache Protokollierung. Die Protokollierung selbst wird von den Diensten vorgenommen, die die zu protokollierenden Änderungen veranlassen. Hierzu gehören der *ExecutionManager* (s. Abschnitt 7.3.6) für die **Ausführungshistorie** (FA 2.1) und der *DataManager* (s. Abschnitt 7.3.5) für die **Datenhistorie** (FA 2.2).

Das **Organisationsmodell** (FA 3) wird vom *OrgModelManager* (s. Abschnitt 7.3.3) **verwaltet** (FA 3.1). Dieser ist auch für die **Auswertung von Bearbeiterzuordnungen** zuständig. Wurde die Bearbeiterzuordnung eines Prozessschritts ausgewertet, wird der Prozessschritt über die **Arbeitslistenverwaltung** (FA 4) im *WorklistManager* (s. Abschnitt 7.3.7) in die entsprechenden Arbeitslisten aufgenommen. Der *WorklistManager* übernimmt dabei neben der **Verteilung** auch die **Delegation** von Arbeitslisteneinträgen. Auch die **Eskalation** (FA 5.1) als Teil der **Prozessüberwachung** (FA 5) ist Aufgabe des *WorklistManager*-Dienstes. Die **Prozessanalysen** (FA 5.2) erfolgen im *LogManager* (s. Abschnitt 7.3.10.4), der auch zusammen mit dem *ResourceManager* (s. Abschnitt 7.3.10.3) für die allgemeine **Prozessüberwachung** (FA 5) zuständig ist.

Wie für die Protokollierung gibt es auch für die **persistente Speicherung** (FA 6) einen allgemeinen Dienst, die Datenhaltung (*Persistence*, s. Abschnitt 7.3.10.5). Diese abstrahiert vom zugrundeliegenden DBMS, während die eigentliche Speicherung von den Diensten vorgenommen wird, die die entsprechenden Daten verwalten. Hierzu gehören die Daten von Prozessvorlagen und -instanzen im *ProcessManager* (s. Abschnitt 7.3.4), die in Prozessen erzeugten und

konsumierten Daten im *DataManager* (s. Abschnitt 7.3.5) sowie die über den *LogManager* (s. Abschnitt 7.3.10.4) protokollierten Daten.

Die **Integration von Anwendungen und deren Aufruf** (FA 7) ist Bestandteil der Ausführung von Prozessschritten und erfolgt durch den *RuntimeManager* (s. Abschnitt 7.3.8) sowie das dazugehörige *Laufzeitumgebungsmodell* (s. Abschnitt 7.2.5).

Änderungen an Prozessvorlagen und -instanzen erfolgen durch den Dienst *ChangeOperations* (s. Abschnitt 7.3.9). Dieser realisiert sowohl die **Prozessmodellierung** (FA 8) als auch **Adaptivität** (FA 9). Änderungen an laufenden Prozessinstanzen (FA 9.1) werden vom *ProcessManager* (s. Abschnitt 7.3.4) umgesetzt, während bei der Schemaevolution (FA 9.2) neben dem *ProcessManager* (s. Abschnitt 7.3.4) für die Migration laufender Prozessinstanzen auch das *ProcessRepository* (s. Abschnitt 7.3.2) für die Verwaltung der Änderungen an Prozessvorlagen benötigt wird.

Die Anbindung von Anwendungsbausteinen bei der Modellierung an den Prozesskontext im ***Plug and Play***-Stil (FA 10), ergänzt die Integration und den Aufruf von Anwendungsbausteinen zur Laufzeit (FA 7). Die für die Modellierung notwendigen Informationen werden durch das *ActivityRepository* (s. Abschnitt 7.3.1) bereitgestellt. Modellierte Prozessvorlagen und Änderungen an ihnen, also die Prozessverwaltung und -versionierung (FA 11) erfolgen durch das *ProcessRepository* (s. Abschnitt 7.3.2).

Die Verwendung des PrMS als **Entwicklungsplattform** über die **Programmierschnittstelle** (API, FA 12) wird einerseits über die Schnittstelle der einzelnen Dienste erreicht (s. Abschnitt 7.4, Webdienste). Andererseits ermöglicht der spezielle *Communication*-Dienst für Webdienste (s. Abschnitt 7.3.10.2), dass diese Schnittstelle auch programmiersprachenunabhängig verwendet werden kann.

7.6 Umsetzung der nicht-funktionalen Anforderungen

Nachdem in den vorherigen Abschnitten der Schwerpunkt auf funktionalen Aspekten lag, diskutieren wir nun die Umsetzung der nicht-funktionalen Anforderungen durch die vorgestellte Architektur. Einige nicht-funktionale Anforderungen aus Abschnitt 3.3 werden maßgeblich durch die Implementierung bestimmt. Hierzu gehört auch NA 2 (Robustheit). Die Architektur hat auf diese Anforderung nur indirekten Einfluss durch Realisierung von Einfachheit (NA 4) und Modularität (NA 5). Auch NA 7 (Wartbarkeit) verbessert die Robustheit, wobei auch diese in erster Linie durch die Implementierung bestimmt und nur indirekt über NA 4, NA 5 und NA 6 (Wiederverwendung/Redundanzvermeidung) durch die Architektur beeinflusst wird.

NA 4 (Einfachheit) bezieht sich sowohl auf die Implementierung (NA 4.2) als auch auf die Architektur (NA 4.1). Die Einfachheit der Architektur wurde in den vorherigen Abschnitten bei der Vorstellung der Datenmodelle (s. Abschnitt 7.2), der Dienste (s. Abschnitt 7.3) sowie der Programmierschnittstelle (s. Abschnitt 7.4) deutlich. Insbesondere die weitgehende Trennung in Daten und Dienste sowie die funktionale Aufteilung der Dienste gewährleisten die Einfachheit der Architektur.

Die Veränderbarkeit/Flexibilität (NA 8) wird durch NA 4.2, NA 5.1 (interne Unabhängigkeit), NA 6 und NA 3 (Erweiterbarkeit) bestimmt. Ihre Umsetzung durch die Architektur wird deshalb nicht separat sondern zusammen mit diesen bestimmenden nicht-funktionalen Anforderungen diskutiert.

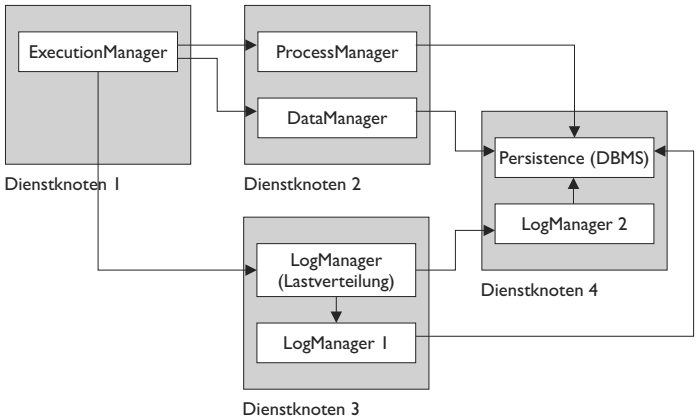


Abbildung 7.51: Verteilte Ausführung und Lastverteilung

7.6.1 Skalierbarkeit/Leistungsfähigkeit

Die Skalierbarkeit/Leistungsfähigkeit (NA 1) wird seitens der Datenmodelle durch eine kompakte und effiziente Datenrepräsentation gewährleistet. Dies haben wir in den Kapiteln 4, 5 und 6 ausführlich diskutiert. Seitens der Dienste wird NA 1 durch verschiedene Aspekte umgesetzt, die unter anderem auf der Modularisierung der Dienste sowie der Trennung zwischen Schnittstelle und Implementierung beruhen.

7.6.1.1 Verteilte Ausführung

Die Modularisierung und Unabhängigkeit der Dienste erlaubt es, diese auf mehrere Dienstknoten zu verteilen. Einerseits können die Dienste dadurch entsprechend ihres Ressourcenbedarfs und der Leistungsfähigkeit der Dienstknoten verteilt werden. Dies ermöglicht eine gute Ausnutzung der vorhandenen Ressourcen. Andererseits können leicht Ressourcen in Form neuer Dienstknoten hinzugefügt werden, die nach einer Umverteilung der Dienste sofort genutzt werden. Dadurch skaliert das System gut. Aufgrund des *Configuration & Registry Framework* (s. Abschnitt 7.3.10.1) und des *Communication*-Dienstes (s. Abschnitt 7.3.10.2) bleibt die Verteilung vor den Verwendern eines Dienstes verborgen. Abb. 7.51 zeigt eine mögliche Verteilung von Diensten auf verschiedene Knoten.

7.6.1.2 Lastverteilung

Neben der Verteilung verschiedener Dienste auf verschiedene Knoten, ermöglicht die Trennung der Dienstschnittstellen von den Dienstimplementierungen auch die Verteilung eines einzelnen Dienstes auf mehrere Knoten (s. *LogManager* in Abb. 7.51). Dabei werden zur Lastverteilung mehrere Instanzen desselben Dienstes auf verschiedenen Dienstknoten ausgeführt. Zusätzlich gibt es eine leichtgewichtige Implementierung der selben Dienstschnittstelle (*Wrapper*), die Anfragen nicht selbst beantwortet, sondern aufgrund definierter Kriterien an die vorhandenen Dienstinstanzen weiterleitet und so die Last verteilt. Dies ist dann sinnvoll, wenn kein Dienstknoten ausreichende Ressourcen für einen last-intensiven Dienst besitzt. Stattdessen fasst die virtuelle Implementierung die Ressourcen mehrerer Dienstknoten virtuell zusammen und erhöht somit die für den Dienst zur Verfügung stehenden Ressourcen. Wie die verteilte Ausführung bleibt auch die Lastverteilung aufgrund des *Configuration & Registry Framework*

(s. Abschnitt 7.3.10.1) und des *Communication*-Dienstes (s. Abschnitt 7.3.10.2) vor Aufrufern verborgen.

7.6.1.3 Große Datenmengen

Ein weiterer Aspekt zur Gewährleistung der Leistungsfähigkeit ist der Umgang mit großen Datenmengen. Diese Mengen können einerseits durch eine serverseitige Filterung vor ihrem Abruf reduziert werden, andererseits ermöglicht es die inkrementelle Übertragung großer Datenmengen, schneller mit der Verarbeitung der Daten beginnen zu können. Beides wird an den Dienstschnittstellen explizit berücksichtigt, wo Anfragen potentiell umfangreiche Ergebnismengen besitzen (s. Abschnitt 7.3.10.2). Hierzu zählt etwa der Abruf von Arbeitslisten.

Im Normalfall erfolgt die Übertragung der Ergebnismenge komplett. Ist die Menge zu groß, um sie in angemessener Zeit übertragen zu können, kann man sie durch Angabe eines Filters verkleinern. Bei einer Arbeitsliste können beispielsweise nur die Einträge mit einer bestimmten Priorität übertragen werden. Ist die Ergebnismenge nach der Filterung immer noch zu groß (oder die Verbindung zu dem Dienst zu langsam), kann die Ergebnismenge auch inkrementell abgerufen werden. Die Anfrage für eine inkrementelle Übertragung großer Datenmengen unterscheidet sich etwas von Anfragen, bei denen die gesamte Ergebnismenge vollständig übertragen wird. Zum einen wird nur ein Stellvertreterobjekt zurückgegeben und im ersten Schritt zunächst einmal kein Element der Ergebnismenge. Zum anderen erfordert die Anfrage ein Sortierkriterium. Dieses definiert eine Reihenfolge der Ergebnismenge, wodurch verhindert wird, dass beim inkrementellen Abruf ein Element der Ergebnismenge mehrfach übertragen wird. Über das zurückgegebene Stellvertreterobjekt können eine frei definierbare Anzahl an Elementen abgerufen werden. Zusätzlich kann die Position angegeben werden, ab der die Elemente in der Ergebnismenge übertragen werden.

Naheliegenderweise beruhen inkrementell übertragene Daten nicht immer auf dem letzten Stand der Daten. So kann sich beispielsweise während dem inkrementellen Abruf einer Arbeitsliste deren Inhalt ändern. Dadurch ändert sich eigentlich auch die Ergebnismenge selbst, was zu Inkonsistenzen bei der Übertragung führen kann. Durch neu hinzugekommene Elemente können sich Elemente in der Ergebnismenge nach hinten verschieben, durch Löschungen nach vorne. Dadurch können manche Elemente mehrfach oder gar nicht übertragen werden. Deshalb spiegelt eine inkrementell übertragene Ergebnismenge immer nur den Zustand der Daten zum Anforderungszeitpunkt wider (sog. *Schnappschuss*).

Beispiel 7.27 (Inkonsistenzen bei inkrementeller Übertragung von Arbeitslisten) Abb. 7.52 zeigt das Problem von Inkonsistenzen bei inkrementeller Übertragung von Arbeitslisten. Es wurden bisher zwei Teile der Arbeitsliste inkrementell übertragen, die Übertragung wird bei Position 11 fortgesetzt. Zu diesem Zeitpunkt kommt ein neuer Arbeitslisteneintrag hinzu (rechts in der Abbildung). Dadurch verschieben sich alle Einträge um eine Position nach hinten. Nun befindet sich an Position 11 ein bereits übertragener Eintrag, während der neue Eintrag an Position 9 nicht übertragen wird. Diese Inkonsistenz lässt sich dadurch vermeiden, dass der Stand der Arbeitsliste zum Zeitpunkt des Abrufs eingefroren wird und spätere Aktualisierungen nicht berücksichtigt werden (s. Arbeitsliste links in Abb. 7.52).

Arbeitsliste

	Instanz	Schritt	zugewiesen	fällig	Priorität
übertragen	I16	L	10.03.2009	19.03.2009	gering
	I22	H	26.03.2009	31.03.2009	normal
	I22	E	24.03.2009	04.04.2009	hoch
	I4	N	16.05.2009	18.05.2009	normal
	I4	F	28.05.2009	08.06.2009	hoch
	I23	C	28.06.2009	11.07.2009	gering
	I15	N	26.07.2009	03.08.2009	gering
	I14	M	06.09.2009	12.09.2009	normal
	I10	B	02.10.2009	05.10.2009	nomal
	I7	C	01.10.2009	08.10.2009	normal
noch zu übertragen	I9	E	28.09.2009	19.10.2009	hoch
	I29	A	10.02.2009		normal
	I1	C	24.02.2009		gering
	I5	G	12.03.2009		hoch
	I29	E	15.04.2009		hoch

Stand bei Anforderung

Arbeitsliste

	Instanz	Schritt	zugewiesen	fällig	Priorität	
übertragen	I16	L	10.03.2009	19.03.2009	gering	
	I22	H	26.03.2009	31.03.2009	normal	
	I22	E	24.03.2009	04.04.2009	hoch	
	I4	N	16.05.2009	18.05.2009	normal	
	I4	F	28.05.2009	08.06.2009	hoch	
	I23	C	28.06.2009	11.07.2009	gering	
	I15	N	26.07.2009	03.08.2009	gering	
	I14	M	06.09.2009	12.09.2009	normal	neuer Eintrag (nicht übertragen)
	I23	D	03.10.2009	04.10.2009	gering	
	I10	B	02.10.2009	05.10.2009	nomal	
	I7	C	01.10.2009	08.10.2009	normal	
noch zu übertragen	I9	E	28.09.2009	19.10.2009	hoch	
	I29	A	10.02.2009		normal	verschobener alter Eintrag (doppelt übertragen)
	I1	C	24.02.2009		gering	
	I5	G	12.03.2009		hoch	
	I29	E	15.04.2009		hoch	

Stand nach Aktualisierung während Abruf

Abbildung 7.52: Inkonsistenz bei inkrementeller Übertragung

7.6.2 Erweiterbarkeit

Erweiterbarkeit (NA 3) ermöglicht es, das allgemeine PrMS leicht mit speziellen Funktionen zu versehen. Dadurch lassen sich die Anforderungen aus unterschiedlichsten Einsatzgebieten realisieren, wodurch das allgemeine PrMS breit einsetzbar wird. Über die Erweiterbarkeit kann das PrMS aber nicht nur speziell angepasst, sondern auch generell um zusätzliche Funktionen erweitert werden. Weder die speziellen noch die allgemeinen Erweiterungen erfordern Schnittstellenänderungen. Dies ist insbesondere beim Einsatz als Entwicklungsplattform (FA 12) wichtig, da Schnittstellenänderungen Inkompatibilitäten zwischen verschiedenen Versionen der Plattform verursachen können.

NA 3 wird auf Modellebene (NA 3.2) und Dienstebene (NA 3.1) unterschiedlich umgesetzt. Beide Aspekte werden im folgenden getrennt voneinander diskutiert. Außerdem betrachten wir in diesem Abschnitt die Erweiterbarkeit der Prozessänderungen (s. Abschnitt 7.3.9). Diese stellt einen Spezialfall der Erweiterbarkeit von Diensten dar.

7.6.2.1 Modellebene

Wie in Abschnitt 7.2.1.1 vorgestellt, erfüllen die meisten Entitäten der Datenmodelle die Schnittstellen *UserAttributeContainer* und *PluginDataContainer*. *UserAttributeContainer* erlaubt beliebige zusätzliche Attribute in Form von Zeichenketten an den Entitäten. Damit können bereits die meisten Erweiterungen realisiert werden. *PluginDataContainer* ermöglicht die Erweiterung der Schnittstellen der Entitäten. Dabei werden die entsprechenden Daten allgemein gespeichert und über einen speziellen Mechanismus typsicher verfügbar gemacht.

Eine Einschränkung von *UserAttributeContainer* und *PluginDataContainer* ist, dass keine komplexen Objekte hinterlegt werden können, die beim entfernten Aufruf als Referenz und nicht als Kopie übertragen werden sollen. Alle Werte müssen serialisierbar sein. Dies gilt aber sowieso auch für alle anderen Attribute der Entitäten der Datenmodelle. Es werden

lediglich Dienstschnittstellen, *Callback*s und Iteratoren für umfangreiche Ergebnismengen beim entfernten Aufruf als Referenz übertragen (s. Abschnitt 7.4, Kopien und Referenzen).

Lokalisierung Die Lokalisierung ist ein Beispiel für die Nutzung von *UserAttributeContainer*. Bei der Lokalisierung werden die Daten an die Gegebenheiten des geographischen Einsatzgebietes des Systems angepasst. Dies umfasst etwa die Übersetzung von Bezeichnungen in verschiedene Sprachen oder die landesübliche Formatierung von numerischen Werten, etwa einem Datum. Da ein PrMS explizit die verteilte Ausführung von Prozessen ermöglicht und Schritte eines Prozesses durchaus in verschiedenen Ländern ausgeführt werden können, reicht eine statische Lokalisierung nicht aus. Vielmehr müssen die Lokalisationsdaten jederzeit für die entsprechenden Attribute der Datenmodelle vorliegen. Dies kann einfach über *UserAttributeContainer* realisiert werden:

Jedes entsprechende Attribut speichert seine Lokalisationsdaten in einer Menge benutzerdefinierter Attribute (`Map<String, String>`) im *UserAttributeContainer* der dazugehörigen Entität. Pro Attribut besteht diese Menge aus allen Lokalisierungen dieses Attributwerts. Da ein *UserAttributeContainer* an eine Entität geknüpft ist und Entitäten wiederum mehrere lokalisierbare Attribute besitzen, muss der Index in den *UserAttributeContainer* geeignet gewählt werden. Dieser setzt sich dem Namen der Lokalisierung und aus dem Attributnamen zusammen: *Instance*

Beispiel 7.28 (Lokalisierung des Prozessinstanznamens)
Der Name einer Prozessinstanz sei „Bestellung“. Dieser ist im Prozessmodell als Wert des Attributs `Name` der Entität *Instance* hinterlegt. Für englische Benutzer heißt die Prozessinstanz „Order“, für französische Benutzer „Commande“. Damit befinden sich die folgenden Zeichenkettenpaare im *UserAttributeContainer* des entsprechenden *Instance*-Objekts: `"de_Name“=“Bestellung“`, `“en_Name“=“Order“`, `“fr_Name“=“Commande“`.

Erweiterungen des Organisationsmodells Das Organisationsmodell ist in ähnlicher Weise erweiterbar (s. Abschnitt 7.2.7). In den meisten Fällen wird es aber nicht in Form von Objektstrukturen benötigt, sondern die Datenbank wird direkt angefragt. Deshalb wird dort auch nicht die Schnittstelle *UserAttributeContainer* verwendet. Stattdessen werden zusätzliche Attribute einfach wie im relationalen Modell üblich in Form weiterer Tabellenspalten bei Substantiellen hinterlegt. Anfragen an das Organisationsmodell (s. Abschnitt 7.2.7.2), die solche Attribute verwenden, werden entsprechend transformiert, so dass auch die zusätzlichen Attribute berücksichtigt werden.

7.6.2.2 Dienstebene

Erweiterungen auf Dienstebene (NA 3.2) können durch zusätzliche Funktionen erfolgen oder durch Änderung existierender Funktionen. Da zusätzliche Funktionen eine Schnittstellenänderung darstellen, die nicht beliebig erfolgen kann, können sie nur durch Anpassung von Dienstimplementierungen realisiert werden. Aufgrund der Trennung zwischen Diensttyp (Schnittstelle) und Dienstinstanz (Implementierung) ist dies jedoch relativ einfach möglich. Die in objektorientierten Programmiersprachen übliche Vererbung reduziert die Anpassung auf die neuen Funktionen. Auf diese Weise können existierende Funktionen einfach geändert werden, wobei auch hier bei geeigneter Implementierung die Anpassungen gering ausfallen.

Einfachere Erweiterungsmöglichkeiten bieten sich durch Zusatzmodule (Plugins). Diese Teildienste unterscheiden wie normale Dienste zwischen Typ (Schnittstelle) und Instanz (Implementierung). Sie beschränken sich in der Regel auf klar umgrenzte Funktionen, etwa Eskalation, Delegation oder Verteilung bei der Arbeitslistenverwaltung (s. Abschnitt 7.3.7). Außerdem müssen die Zusatzmodule von der entsprechenden Dienstimplementierung vorgesehen sein, d. h. die Dienstimplementierung muss Teilfunktionen auslagern und zur Laufzeit dynamisch entscheiden, durch welche Zusatzmodulimplementierung die Teilfunktion ausgeführt wird. Da jedoch Dienstimplementierungen selbst relativ einfach erweitert werden können, können sie hierüber auch um die Möglichkeit für Zusatzmodule erweitert werden.

7.6.2.3 Prozessänderungen

Der Dienst für Prozessänderungen (s. Abschnitt 7.3.9) ist gut Erweiterbarkeit. Bei diesem sind alle Änderungsoperationen Zusatzmodule (Plugins), womit ein Großteil des Dienstes dynamisch ist. Unter Nutzung der vom Prozessmodell bereitgestellten Änderungsprimitiven können auf diese Weise beliebige zusätzliche Änderungsoperationen hinzugefügt werden. Daneben können auch existierende Änderungsoperationen zu einer neuen Änderungsoperation zusammengefasst werden, um so die Verwendung der einzelnen Änderungsoperationen zu vereinfachen. Um die allgemeine Anwendbarkeit von Änderungsoperationen zu erhalten, insbesondere auch neu hinzugefügter Änderungsoperationen, müssen alle Änderungsoperationen Funktionen für Korrektheitsprüfungen bezüglich Struktur und Zustand einer Prozessvorlage/Prozessinstanz, für strukturelle und Zustandsanpassungen sowie für die Protokollierung der Änderungen anbieten (s. Abschnitt 2.3.1).

7.6.3 Unabhängigkeit/Modularität und Wiederverwendung/Redundanzvermeidung

Aufgrund des modularen Aufbaus der vorgestellten Architektur und des großen Einflusses dieser Module auf die nicht-funktionalen Anforderungen Unabhängigkeit/Modularität (NA 5) und Wiederverwendung/Redundanzvermeidung (NA 6), diskutieren wir beide Aspekte in diesem Abschnitt zusammen.

Wie an in der Dienstarchitektur in Abschnitt 7.3 gezeigt, bilden die einzelnen Dienste funktional abgeschlossene Module. Die Dienste sind abgesehen von der erwünschten Verwendung unterer Abstraktionsebenen voneinander unabhängig. Die Verwendung unterer Abstraktionsebenen durch verschiedene Dienste vermeidet funktionale Redundanz. Außerdem sind die Dienste unterer Abstraktionsebenen so allgemein wie möglich, was ihre Wiederverwendbarkeit erhöht. dasselbe gilt auch für die Datenmodelle (s. Abschnitt 7.2). Sie dienen zum Datenaustausch zwischen Diensten und werden dementsprechend inhärent wiederverwendet.

Sowohl bei den Diensten als auch bei den Datenmodellen wird die Wiederverwendung zusätzlich durch den Implementierungsaspekt der Vererbung verbessert. Wie bereits bei der Erweiterbarkeit (s. Abschnitt 7.6.2.2) diskutiert, lassen sich einzelne Dienstmodule leicht mit zusätzlichen Funktionen versehen oder anpassen, wobei die existierende Dienstimplementierung wiederverwendet werden kann. Dies gilt selbstverständlich auch für die Datenmodelle. Da diese wenig bis keine Anwendungslogik besitzen und sie bezüglich zusätzlicher Attribute erweiterbar sind, sind Änderungen an ihren Implementierungen nicht notwendig.

Um die weitgehende Unabhängigkeit trotz Wiederverwendung zu gewährleisten, beruhen Dienste und Datenmodelle auf Schnittstellen. Die Implementierung von Diensten wird durch

die zentrale Registrierung (s. Abschnitt 7.3.10.1) gekapselt. Die Implementierung von Datenmodellen wird durch *Factory*-Objekte erzeugt, wobei diese wie Dienste über ihre Schnittstelle definiert sind und die Implementierung der *Factory*-Objekte von der zentralen Registrierung gekapselt wird. Würden die Implementierungen von Diensten und Datenmodellen nicht außerhalb der Dienste erzeugt werden, verlören die Dienste naheliegenderweise ihre Unabhängigkeit.

Beispiel 7.29 (Transienter *SubprocessManager durch Konfiguration*)
Die zentrale Registrierung erlaubt die einfache Erweiterbarkeit und Austauschbarkeit von Dienstimplementierung. Zudem ist sie für die Verwendungsbeziehungen zwischen Diensten verantwortlich. So lassen sich leicht Dienstgeflechte etablieren, die für spezielle Einsatzgebiete optimiert sind. Ein Beispiel hierfür ist ein Einsatzgebiet, in dem es viele Subprozesse gibt, die schnell ausgeführt werden sollen, die aber gleichzeitig tolerant gegenüber Mehrfachausführungen sind. Das heißt. die Subprozesse und ihre Daten müssen und sollen nicht persistiert werden.

Da es bereits die Möglichkeit gibt, Prozessdaten mittels SQL zu speichern, bietet sich für diesen Fall HSQLDB ([hsq]) an. Dies ist eine leichtgewichtiges und SQL-kompatibles DBMS, wofür mit einfachen Mitteln eine Implementierung für den Datenhaltungsdienst (s. Abschnitt 7.3.10.5) bereitgestellt werden kann. Außerdem werden noch ein *ProcessManager* (s. Abschnitt 7.3.4) und ein *DataManager* (s. Abschnitt 7.3.5) für die Datenverwaltung benötigt, sowie ein *ExecutionManager* (s. Abschnitt 7.3.6) für die Subprozesse, der die beiden Verwaltungsdienste verwendet. Außerdem gibt es den *SubprocessManager* (s. Abschnitt 7.3.8.5), der die Subprozessausführung auf dem entsprechenden *ExecutionManager* anstößt. Außer für den Datenhaltungsdienst können die normalen Dienstimplementierungen verwendet werden. Aufgrund der Unabhängigkeit der Dienste kann so leicht ein für das spezielle Einsatzgebiet passendes Dienstgeflecht bereitgestellt werden.

7.7 Zusammenfassung

In diesem Kapitel haben wir die Architektur eines adaptiven Prozessmanagementsystems im Detail vorgestellt. Grundlage hierfür sind die funktionalen und nicht-funktionalen Anforderungen aus Kapitel 3. Die Architektur ist strikt dienstorientiert und folgt den Prinzipien von SOA [Erl05]. Die einzelnen Dienste nutzen gemeinsame Datenmodelle für den Austausch von Informationen. Sowohl die Dienste als auch die Datenmodelle sind über Schnittstellen definiert, die jeweilige Implementierung ist austauschbar. Alle Schnittstellen zusammen sind als Programmierschnittstelle verfügbar sowohl für die Implementierung der Dienste des PrMS selbst als auch für beliebige prozessorientierte Anwendungen, die das PrMS als Bibliothek nutzen. Die Programmierschnittstelle ist zudem als Webdienstschnittstelle unabhängig von einer Programmiersprache.

Die Dienste und Datenmodelle setzen die funktionalen Anforderungen umfassend und möglichst einfach (NA 4) um. Die Funktionen werden gemäß der nicht-funktionalen Anforderungen Unabhängigkeit/Modularität (NA 5) und Wiederverwendung/Redundanzvermeidung (NA 6) disjunkt auf die Dienste verteilt. Funktionen, die dieselben Datenmodelle manipulieren oder semantisch zusammengehören, finden sich im selben Dienst. Die Trennung zwischen Schnittstelle und Implementierung macht die Datenmodelle und Dienste einfach erweiterbar (NA 3) und flexibel (NA 8). Trotzdem bleibt die Verwendung einfach (NA 4), da Trennung zwischen Schnittstelle und Implementierung ebenso wie die physische Trennung durch die Nutzung mehrerer Dienstknoten durch das *Configuration & Registry Framework* gekapselt wird. Neben

der Erweiterung trägt dies auch erheblich zur Skalierbarkeit/Leistungsfähigkeit (NA 1) des Systems bei, da eine transparente Lastverteilung erfolgen kann.

Die vorgestellte Architektur wurde im Rahmen dieser Arbeit in einer Machbarkeitsimplementierung umgesetzt. Diese stellen wir Kapitel 9 als zweiten Teil der Validation dieser Arbeit vor. Im ersten Teil der Validation in Kapitel 8 stellen wir zunächst die Simulation und die Simulationsergebnisse der Konzepte zur Speicherrepräsentation unveränderter und veränderter Prozessinstanzen vor.

Teil III

Validation

8 Simulation

In den Kapiteln 4–6 haben wir den Speicherbedarf und die Laufzeit von Zugriffen der verschiedenen Konzepte und Varianten qualitativ und auf abstrakter Ebene auch quantitativ untersucht. Wir erweitern und validieren diese Untersuchungen im vorliegenden Kapitel. Dabei unterscheiden wir wieder zwischen unveränderten (s. Abschnitt 8.1) und veränderten Prozessinstanzen (s. Abschnitt 8.2). Der Schwerpunkt bei unveränderten Prozessinstanzen liegt auf der Repräsentation von Blockstrukturen (s. Abschnitt 4.1.5) und der Zustandsrepräsentation (s. Abschnitt 4.1.3.2), inklusive der Zustandsclusterung (s. Kapitel 6). Bei veränderten Prozessinstanzen liegt er auf dem Vergleich der verschiedenen Realisierungskonzepte und Implementierungsvarianten (s. Kapitel 5). Um möglichst aussagekräftige Ergebnisse zu erhalten, verwenden wir für unveränderte und veränderte Prozessinstanzen unterschiedliche Testumgebungen und führen unterschiedliche Messungen durch. Deshalb erfolgen die Messungen auch auf speziellen Implementierungen. Dazu gehören die für die Simulation notwendigen Funktionen und Datenstrukturen. Bevor wir in den folgenden Abschnitten die Messergebnisse vorstellen und interpretieren, gehen wir jeweils auf die Testumgebung und den Ablauf der Messungen für unveränderte und veränderte Prozessinstanzen ein.

8.1 Unveränderte Prozessinstanzen

Bei der Simulation unveränderter Prozessinstanzen werden die Zustandsrepräsentationen ZR-2 (implizite Markierungen, s. Abschnitt 4.1.3.2) und ZR-3 (explizite Markierungen, s. Abschnitt 4.1.3.2) sowie die Repräsentation von Blockstrukturen BS-1 (Blockstrukturen als Objekte, s. Abschnitt 4.1.5.1) und BS-2 (Blockstrukturen als Relation, s. Abschnitt 4.1.5.2) miteinander verglichen. Der Vergleich erfolgt anhand des Speicherbedarfs der Datenrepräsentationen sowie der Laufzeit verschiedener Funktionen, die auf den funktionalen Anforderungen (s. Abschnitt 3.2) beruhen. Hierzu gehören das Ein- und Auslagern von Prozessvorlagen und -instanzen, Anfragen gegen Kollektionen von Prozessinstanzen sowie das Weiterschalten in Prozessinstanzen. Daneben werden Messungen zur *groben Clusterung* durchgeführt [Glö09]. Bei dieser Art der Zustandsclusterung (s. Kapitel 6) ist der Zustandsgraph nicht vollständig, jede Prozessinstanz muss ihren Zustand selbst verwalten. Dementsprechend ist die grobe

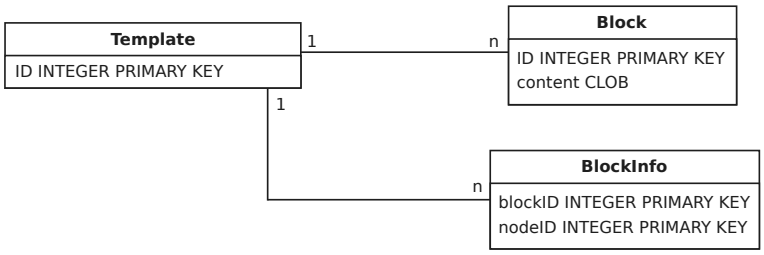


Abbildung 8.1: ER-Diagramm der Repräsentation von BS-1 im Sekundärspeicher als CLOB (aus [Glö09])

Clusterung auch keine alternative Zustandsrepräsentation. Durch die Messungen wird lediglich ihr Mehraufwand ermittelt.

8.1.1 Testumgebung

Die Simulation wird in Java [jav] implementiert und durchgeführt. Eine JVM hat zahlreiche Hintergrundfunktionen, die Einfluss auf Laufzeitmessungen haben. Hierzu gehört JIT (Just-in-time-Kompilierung), also die Kompilierung der in Java üblichen Zwischensprache (*Bytecode*) in die Sprache der tatsächlich verwendeten physischen Maschine. JIT läuft vollständig unter Kontrolle der JVM, der Benutzer hat nur eingeschränkte Mittel der Einflussnahme. Ähnlich ist die Situation bei der automatischen Speicherbereinigung. Diese kann jedoch explizit im Programmcode aufgerufen werden, was auch zwischen den Messungen passiert. Dadurch verringert sich die Wahrscheinlichkeit, dass die Speicherbereinigung während einer Messung läuft. Zudem gibt es pro Testfall mehrere Messungen, von denen das arithmetische Mittel gebildet wird. Der Einfluss von JIT wird dadurch verringert, dass es pro Messung wiederum mehrere Testdurchläufe gibt, von denen die ersten nicht in die Messung eingehen.

Das Ein- und Auslagen der Datenstrukturen erfordert neben der Repräsentation im Primärspeicher auch Datenstrukturen im Sekundärspeicher. Die Repräsentationskonzepte BS-1 und BS-2 (s. Abschnitt 4.1.5) sowie ZR-2 und ZR-3 (s. Abschnitt 4.1.3.2) werden im Primärspeicher direkt als Javaobjekte realisiert. Daneben gibt es eine zusätzliche Java-Datenstruktur für die grobe Clusterung im Primärspeicher. Für die Repräsentation im Sekundärspeicher werden zwei grundlegend unterschiedliche Varianten verwendet und verglichen: Speicherung als XML-Datei und Verwendung des relationalen DBMS PostgreSQL [pos]. Innerhalb dieser Varianten gibt es wieder unterschiedliche Realisierungen für Blockstrukturen und Zustandsrepräsentationen. Bei der Speicherung als XML-Datei spiegeln sich BS-1 und BS-2 sowie ZR-2 und ZR-3 in unterschiedlichen XML-Schemata wider. Im Fall relationaler DBMS werden Teile der Blockinformationen und Zustandsrepräsentationen nicht atomar sondern als ein Tupelattribut gespeichert (s. Abb. 8.1, Abb. 8.2, Abb. 8.3). Dazu werden die Blockinformationen und Zustandsinformationen bei ZR-3 jeweils als XML-Datenstruktur serialisiert und in ein Tupelattribut geschrieben. Da PostgreSQL XML-Daten in einem Tupelattribut interpretieren kann, sind Anfragen des Inhalts mittels XPath-Ausdrücken (*XML Path Language*) direkt über SQL möglich (s. Abb. 8.2). In den Messungen wird dies neben der üblichen Speicherung von XML-Daten als CLOB (*Character Large Object*, s. Abb. 8.1) in relationalen DBMS verwendet. PostgreSQL wird über die entsprechenden JDBC-Treiber angesprochen.

Weitere Details zur Testumgebung bei unveränderten Prozessinstanzen sind in [Glö09] aufgeführt.

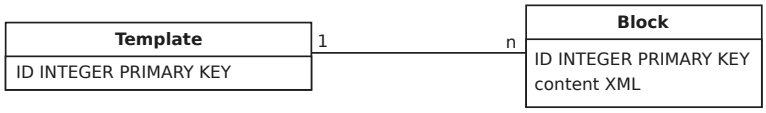


Abbildung 8.2: ER-Diagramm der Repräsentation von BS-1 im Sekundärspeicher XML (aus [Glö09]

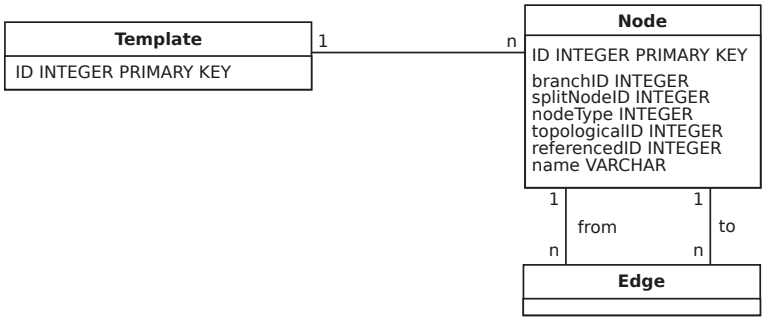


Abbildung 8.3: ER-Diagramm der Repräsentation von BS-2 im Sekundärspeicher (aus [Glö09])

8.1.2 Ablauf der Messung

Der Speicherbedarf im Primärspeicher wird abhängig von der Anzahl der Knoten und Kanten sowie dem Anteil der Blockstrukturen im Verhältnis zur Gesamtzahl an Knoten gemessen (s. Abschnitt 8.1.3). Dazu werden eine einfache Prozessvorlage und eine einfache Prozessinstanz verwendet, deren Knotenanzahl und Anzahl der Blöcke variiert werden. Der unterschiedliche Anteil an Blockstrukturen dient dem Vergleich des Speicherbedarfs von BS-1 und BS-2. Der Speicherbedarf im Sekundärspeicher ist aufgrund der niedrigen Kosten für diese Speicherart vernachlässigbar und wird deshalb nicht gemessen. Zudem lassen sich hier durch zahlreiche Zusatzdaten wie Indexstrukturen und Transaktionsprotokolle kaum belastbare Messungen durchführen.

Um bei den Laufzeitmessungen weitgehend von Hardwareeigenschaften zu abstrahieren, werden deren Messergebnisse mittels eines ermittelten Hardwarefaktors normiert. Die Ergebnisse sind also keine absoluten Zahlen. Bei Laufzeitmessungen, die ausschließlich im Primärspeicher (s. Abschnitt 8.1.4.4) erfolgen, werden die Ergebnisse aus Gründen der Lesbarkeit zusätzlich mit 1.000 multipliziert. Alle Messungen werden jeweils 1.000 Mal durchgeführt, das jeweilige Messergebnis ist das arithmetische Mittel dieser Durchläufe.

Um die Simulation einfach zu halten, werden für Messungen zur Laufzeit (s. Abschnitt 8.1.4), etwa Ein-/Auslagern, Anfragen gegen Kollektionen von Prozessinstanzen, Weiterschalten von Prozessinstanzen, fest definierte Prozessvorlagen verwendet und nicht wie bei den Speicherbedarfsmessungen die Anzahl der Knoten und Blöcke variiert. Um dennoch Aussagen zum Einfluss der Größe und Komplexität von Prozessstrukturen machen zu können, werden stattdessen drei unterschiedliche Prozessvorlagen stellvertretend verwendet: eine einfache, eine komplexe und eine künstliche[1] Prozessvorlage. Während die einfache Prozessvorlage nur wenige Knoten und zwei Blöcke besitzt (s. Abb. 8.4), hat die komplexe Prozessvorlage zahlreiche Knoten und Blöcke (s. Abb. 8.5). Die künstliche Prozessvorlage bewegt sich dazwischen (s. Abb. 8.6): Sie ist komplexer als die einfache Prozessvorlage ist, jedoch noch so einfach, dass sich

[1] Diese Prozessvorlage besitzt eine Struktur, die für die Messungen optimiert ist und keinen Praxisbezug hat.

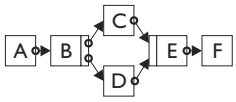


Abbildung 8.4: Einfache Prozessvorlage für Messungen

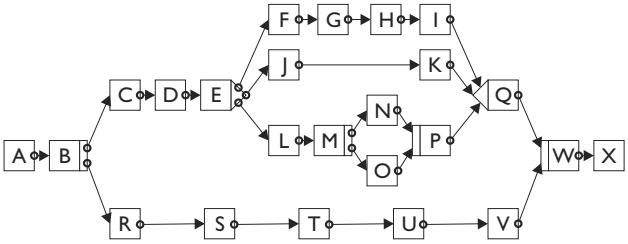


Abbildung 8.5: Komplexe Prozessvorlage für Messungen

leicht Prozessinstanzen erzeugen lassen, deren Zustände sich gleich auf alle Knoten verteilen. Dabei existiert für jeden Knoten der Prozessvorlage genau eine Prozessinstanz, in der der Knoten laufend (`RUNNING`) ist. Diese Prozessvorlage und die Zustandsgleichverteilung der Prozessinstanzen wird für Anfragen an Kollektionen von Prozessinstanzen verwendet.

Diese Prozessvorlagen werden auch für Messungen mit Prozessinstanzen herangezogen. Durch die Verwendung von Vorlagenreferenzen (VG-2, s. Abschnitt 4.1.3.1) hat die Prozessstruktur nur indirekten Einfluss auf die Datenstrukturen von Prozessinstanzen: Bei ZR-2 bestimmt die Anzahl der parallelen Zweige die Anzahl der explizit zu verwaltenden Knotenzustände, bei ZR-3 bestimmt die Knotenanzahl die Menge der Knotenzustände. Somit ist bei ZR-2 der Einfluss der Prozessstruktur vermutlich weitgehend vernachlässigbar, während bei ZR-3 die Größe der Datenstrukturen für Prozessinstanzen proportional zur Größe der Prozessvorlage ist.

Für Kollektionen von Prozessinstanzen (s. Abschnitt 8.1.4.3) gibt es für jede Sekundärspeichervariante (XML-Datei und relationales DBMS) fünf verschieden Anfragen: Die Ermittlung aller Prozessinstanzen einer bestimmten Prozessvorlage (*Instanzen*) sowie vier Anfragen nach Start- und Endknoten sowie den Zuständen `RUNNING` und `COMPLETED`. Dies zeigt, welche Auswirkungen die Zustandsrepräsentation auf die Laufzeit der Anfragen gegen Kollektionen von Prozessinstanzen hat. Die entsprechenden Messungen heißen: *StartRunning*, *StartComplete*, *EndRunning* und *EndComplete*.

Im Fall des relationalen DBMS PostgreSQL werden für die Anfragen an Kollektionen von Prozessinstanzen keine gespeicherten Prozeduren (*Stored Procedure*) verwendet. Dementsprechend müssen beim CLOB-Datentyp alle Prozessinstanzen eingelagert und im Primärspeicher durchsucht werden. Lediglich bei ZR-2 und den Anfragen nach dem Zustand `RUNNING` (ggf. auch beim Zustand `SKIPPED`) können die entsprechende Prozessinstanzen direkt im DBMS ermittelt werden, da diese Information direkt als Tupelattribut vorliegt. Dagegen werden

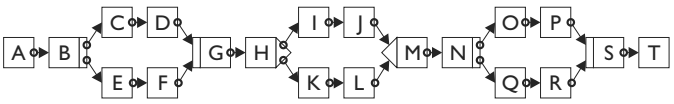


Abbildung 8.6: Künstliche Prozessvorlage für Messungen

beim XML-Datentyp die entsprechenden Möglichkeiten von PostgreSQL genutzt und so die Einlagerung vermieden.

Die Messungen zum Weiterschalten von Prozessinstanzen (s. Abschnitt 8.1.4.4) erfolgen ausschließlich im Primärspeicher. Deshalb ist hier die Größe der Prozessvorlage vernachlässigbar. Die Struktur/Komplexität der Prozessvorlage ist dagegen besonders relevant. So macht es beim Weiterschalten einen großen Unterschied, ob eine parallele Vereinigung 2 oder 10 eingehende Zweige besitzt. Dementsprechend wird für Laufzeitmessungen zum Weiterschalten die komplexe Prozessvorlage verwendet, da diese eine realistischere Prozessstruktur als die anderen beiden Prozessvorlagen aufweist. Die Prozessstruktur wird dadurch berücksichtigt, dass die Laufzeit des Weiterschaltens für Knoten verschiedenen Typs gemessen wird: (Knoten C (normal), Knoten B (parallele Verzweigung), Knoten E (alternative Verzweigung) und Knoten W (parallele Vereinigung).

8.1.3 Speicherbedarf

Der Speicherbedarf von Prozessvorlagen und -instanzen im Primärspeicher lässt Rückschlüsse darauf zu, wie oft die entsprechenden Objekte ein- und ausgelagert werden müssen: Bei einem hohen Speicherbedarf können bei konstanter Speichergröße weniger Objekte im Primärspeicher gehalten werden. In diesem Fall müssen Objekte häufiger ein- und ausgelagert werden. Dies wiederum erfordert Rechenzeit und hat so negative Auswirkungen auf die Leistungsfähigkeit des Gesamtsystems. Trotzdem ist eine ausschließliche Optimierung hinsichtlich des Speicherbedarfs nicht sinnvoll. Durch übermäßige Optimierung können sich die Zugriffszeiten für bestimmte Funktionen etwa das Weiterschalten verschlechtern und damit auch die Leistungsfähigkeit des Gesamtsystems.

Abb. 8.7 zeigt den Speicherbedarf pro Prozessvorlage abhängig von der Knotenanzahl. Für BS-1 (Blockstrukturen als Objekte) und BS-2 (Blockstrukturen als Relation) gibt es jeweils Prozessvorlagen mit 5%, 10% und 20% Anteil an Blockstrukturen.

Deutlich ist der erhöhte Speicherbedarf für BS-1 gegenüber BS-2 zu erkennen. Der Unterschied wird umso größer, je mehr Blockstrukturen in der Prozessvorlage vorhanden sind. Die treppenförmigen Anstiege sind in allen Fällen auf die Verwendung einer Streuwerttabelle zurückzuführen. Diese wird bei Überschreiten einer bestimmten Grenze vergrößert. Gibt es mehr Blockstrukturen (*20% Split*) oder ist deren Repräsentation aufwendig (BS-1), wird die Grenze der Streuwerttabelle schneller erreicht. Deshalb gibt es in diesen Fällen mehr Treppenstufen und kürzere Abstände dazwischen, s. Speicherbedarf von *BS-1 (20% Split)*.

Abb. 8.8 zeigt den Speicherbedarf pro Prozessinstanz abhängig von der Knotenanzahl. Während der Speicherbedarf für ZR-2 (implizite Markierungen) konstant und unabhängig von der Knotenanzahl ist, steigt der Speicherbedarf für ZR-3 (explizite Markierungen) linear an – von den üblichen Vergrößerungen der Streuwerttabelle abgesehen. Die Struktur des Prozesses ist hierbei von untergeordneter Bedeutung und blieb bei der Messung unberücksichtigt. Die Struktur hat lediglich Auswirkungen auf den Speicherbedarf für ZR-2, etwa durch die Anzahl der notwendigen expliziten Markierungen (aufgrund paralleler Zweige) oder der Verzweigungsentscheidungen. Sie hat keine Auswirkungen auf ZR-3, weshalb sich das Verhältnis des Speicherbedarfs zwischen den Repräsentationsvarianten ändert. Bei einer realistischen Größe einer Prozessvorlage von 15–20 Knoten bleibt der Speicherbedarf von ZR-2 jedoch unter dem von ZR-3. Zudem erhöht sich bei einer komplexeren Struktur in der Regel auch die Knotenanzahl der entsprechenden Prozessvorlage.

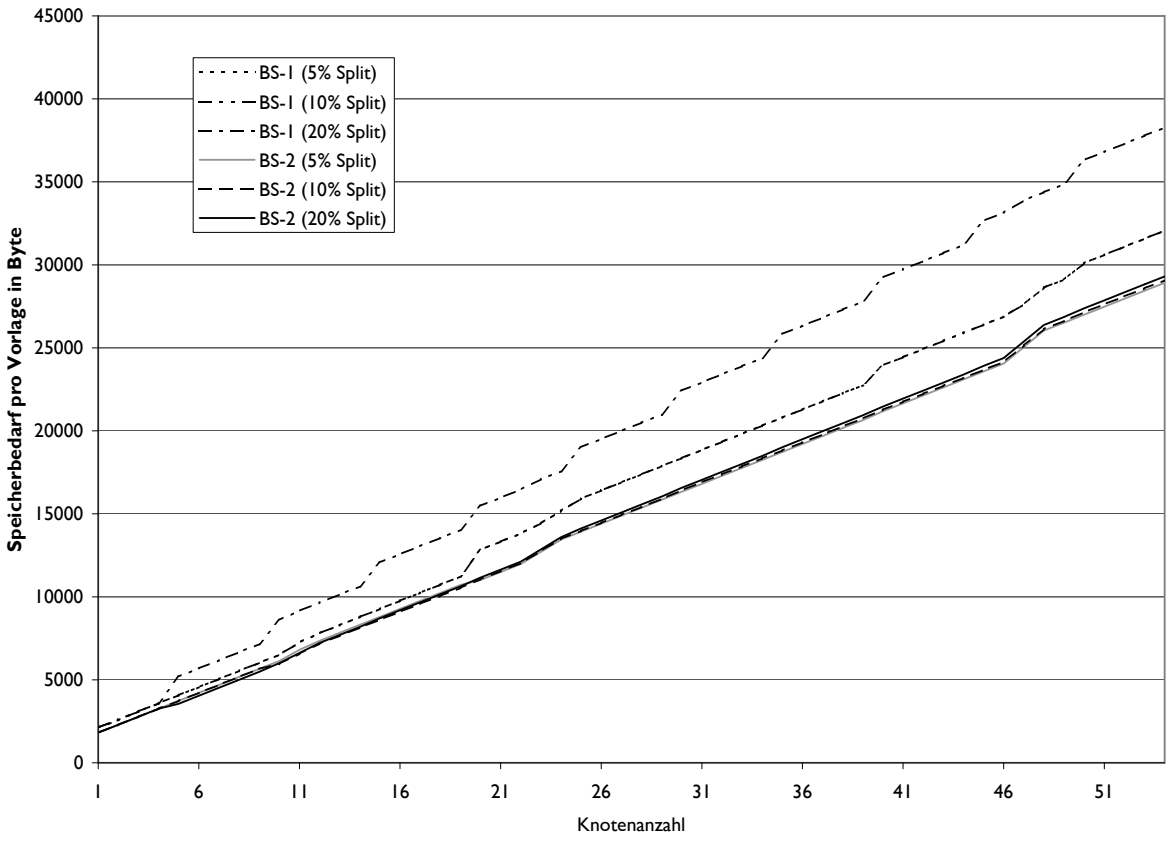


Abbildung 8.7: Speicherbedarf für Prozessvorlagen

8.1.4 Laufzeit von Zugriffen

Da die Leistungsfähigkeit eines PrMS nicht nur aufgrund des Speicherbedarfs beurteilt werden kann, untersuchen wir in diesem Abschnitt die Laufzeit von Zugriffen. Besonders wichtig ist hierbei die Laufzeit der häufig benötigten Funktion des Weiterschaltens von Prozessinstanzen (s. Abschnitt 8.1.4.4). Um im Primärspeicher weiterschalten zu können, müssen die entsprechenden Prozessinstanzen erst vom Sekundär- in den Primärspeicher eingelagert werden. Gegebenenfalls müssen zu diesem Zeitpunkt auch andere Objekte (Prozessinstanzen und/oder Prozessvorlagen) vom Primär- in den Sekundärspeicher ausgelagert werden. Zur benötigten Laufzeit des Weiterschaltens kommen damit noch die Laufzeiten für Ein- und Auslagern hinzu. Diese betrachten wir in Abschnitt 8.1.4.1 und Abschnitt 8.1.4.2.

Aufgrund der Unabhängigkeit von Primär- und Sekundärspeicher können die Daten in beiden Speicherarten unterschiedlich repräsentiert werden. So können Zustände im Sekundärspeicher explizit (ZR-3) und im Primärspeicher implizit (ZR-2) repräsentiert werden. Dazu werden die Datenstrukturen beim Ein- und Auslagern entsprechend transformiert werden. Die Messungen für das Ein- und Auslagern umfassen neben gleichen Datenstrukturen in Primär- und Sekundärspeicher auch Kombinationen mit unterschiedlichen Repräsentationen.

Neben den Auswirkungen auf das Ein-/Auslagern der Daten, hat die Repräsentation im Sekundärspeicher daneben eine erhebliche Bedeutung für Anfragen gegen Kollektionen von Prozessinstanzen. Diesen Anfrage betreffen viele Prozessinstanzen, die für die Ergebnisermittlung alle eingelagert werden müssten. Da dies potentiell eine schlechte Laufzeit hat, ist das Ziel,

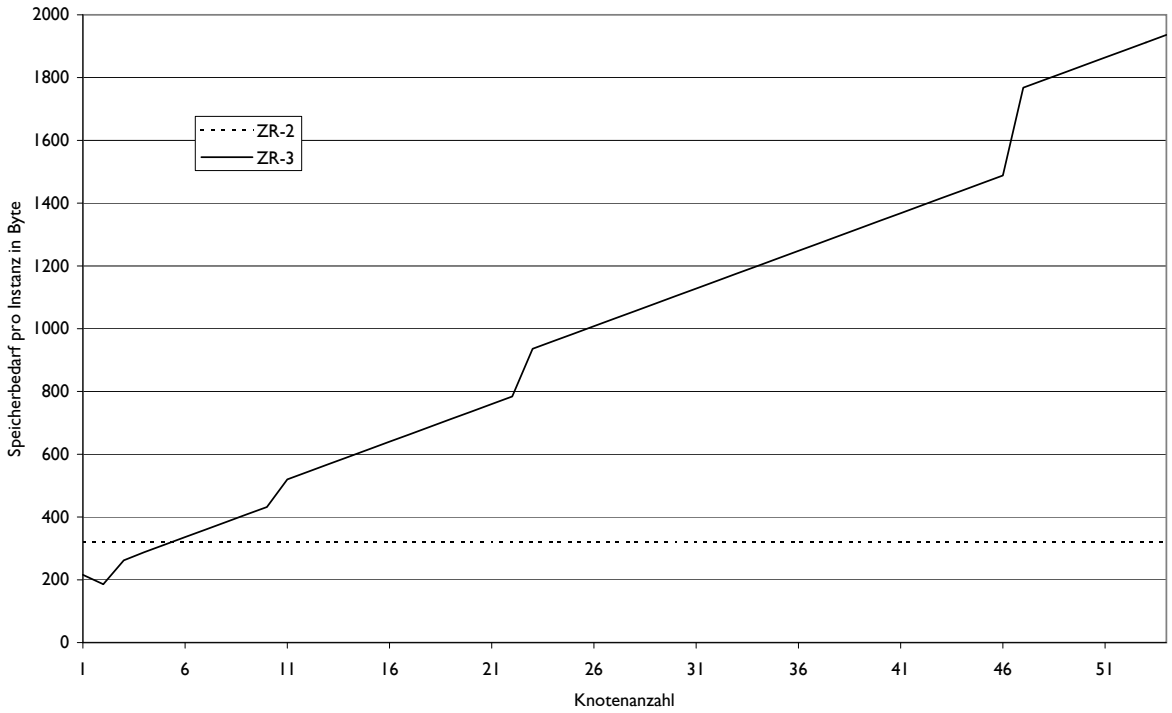


Abbildung 8.8: Speicherbedarf für Prozessinstanzen

möglichst wenige Daten einzulagern oder das Ergebnis ohne Einlagern direkt im Sekundärspeicher zu ermitteln. Die Ergebnisermittlung hängt damit in hohen Maße von der Repräsentation der Daten im Sekundärspeicher ab. Wir untersuchen die Laufzeit einiger häufig benötigter Anfragen in Abschnitt 8.1.4.3 anhand verschiedener Sekundärspeicherrepräsentationen.

8.1.4.1 Ein-/Auslagern von Prozessvorlagen

Das Ein- und Auslagern von Daten ist notwendig, da normalerweise nicht alle Daten im Primärspeicher gehalten werden können. Selbst wenn dies der Fall wäre, müssen die Daten trotzdem persistiert werden (s. FA 6 in Abschnitt 3.2), damit sie auch bei einem Neustart des Systems vorhanden sind. Dementsprechend müssen Änderungen an Daten spätestens beim Beenden des Systems geschrieben und nach dem Neustart oder vor ihrer Verwendung im Primärspeicher eingelesen werden. Dies entspricht weitgehend dem Ein- und Auslagern, weshalb die Laufzeiten für Ein-und Auslagern selbst bei großem Primärspeicher relevant sind.

Prozessvorlagen sind diesbezüglich ein Sonderfall: Sie werden zumindest mit VG-2 (Vorlagenreferenzen, s. Abschnitt 4.1.3.1) wesentlich öfter gelesen (und damit eingelagert) als geschrieben (ausgelagert). Im Fall von unveränderten Prozessinstanzen werden sie sogar nur einmal während ihrer gesamten Nutzungsdauer geschrieben und anschließend nur noch gelesen. Trotzdem messen wir auch die Laufzeit für das Auslagern von Prozessvorlagen. Zum einen rundet es den Vergleich der Repräsentation von Blockstrukturen, zum anderen sind die Messungen auch für die Repräsentation von instanzspezifischen Änderungen mittels ÄR-1 (individuelle Instanzgraphen, s. Abschnitt 5.1.1) und ÄR-3 (strukturelle Instanzdeltas, s. Abschnitt 5.1.3) relevant. Die Datenstrukturen für instanzspezifische Änderungen können durchaus ausgelagert werden.

XML Bei der Speicherung von Prozessvorlagen als XML-Datei ist naheliegenderweise kein direkter Zugriff auf einzelne XML-Elemente möglich. Eine XML-Datei muss immer sequentiell mindestens bis zum Auftreten des gesuchten Elements durchlaufen werden. Damit ist eine blockorientierte Partitionierung nur möglich, indem die einzelnen Blöcke auf verschiedene XML-Dateien verteilt werden. Hierdurch verschlechtert sich jedoch die Laufzeit, da Zugriffe auf verschiedene Dateien aufwendiger sind als fortlaufende Zugriffe auf eine Datei. Die folgenden Messungen werden deshalb nur mit einer XML-Datei pro Prozessvorlage durchgeführt.

Einlagern Abb. 8.9 zeigt die Messergebnisse für das Einlagern von Prozessvorlagen aus dem Sekundärspeicher (XML-Dateien) in den Primärspeicher. Sowohl im Primärspeicher als auch im Sekundärspeicher wird zwischen BS-1 (Blockstrukturen als Objekte) und BS-2 (Blockstrukturen als Relation) unterschieden. Bei verschiedenen Repräsentationen in Primär- und Sekundärspeicher werden die Datenstrukturen beim Einlagern transformiert. Dies ist bei der dunkelgrauen und schwarzen Säule links und bei der weißen und hellgrauen Säule rechts der Fall.

Es fällt auf, dass die Laufzeiten bei BS-2 im Sekundärspeicher bis auf einen Fall immer schneller sind. Das dürfte auf die kompaktere Repräsentation und die somit kleinere XML-Struktur von BS-2 zurückzuführen sein. Überraschenderweise ist das Einlagern der komplexen Prozessvorlage mit Transformation von BS-2 nach BS-1 schneller als dasselbe Einlagern ohne Transformation. Im Fall der einfachen Prozessvorlage ist dagegen das Einlagern mit Transformation von BS-2 nach BS-1 langsamer als ohne Transformation. Dagegen fällt die Transformation von BS-1 im Sekundärspeicher nach BS-2 kaum ins Gewicht verglichen mit dem direkten Einlagern von BS-1. Insgesamt zeigen die Messungen, dass BS-2 möglichst ohne Transformation BS-1 vorzuziehen ist.

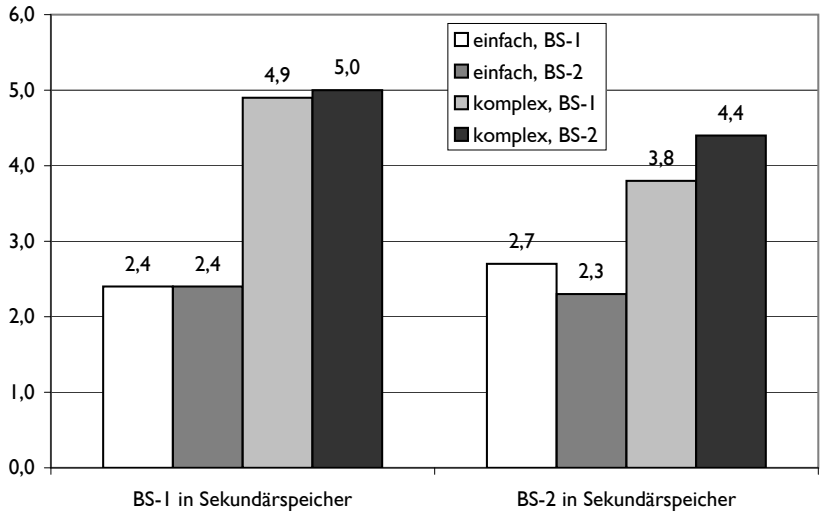


Abbildung 8.9: Einlagern von Prozessvorlagen bei XML-Speicherung

Auslagern Die Messergebnisse für das Auslagern von Prozessvorlagen aus dem Primärspeicher in XML-Dateien zeigt Abb. 8.10. Auch hier sind bei der dunkelgrauen und schwarzen Säule links und bei der weißen und hellgrauen Säule rechts Transformationen der Repräsentation von Blockstrukturen notwendig. Erstaunlicherweise sind die Laufzeiten immer kürzer als beim

Einlagern von Prozessinstanzen (s. Abb. 8.9). Zudem ist bei der komplexen Prozessvorlage das Auslagern mit Transformation immer schneller als das Auslagern ohne Transformation. Beides lässt sich nicht sinnvoll erklären. Die Laufzeitunterschiede bei der komplexen Prozessvorlage bei BS-2 im Sekundärspeicher (rechts) könnte man dadurch erklären, dass bei BS-1 die Datenstrukturen im Primärspeicher schneller ausgelesen werden können als bei BS-2 im Primärspeicher. Dann müsste dies allerdings dazu führen, dass bei BS-1 im Sekundärspeicher (links) die Laufzeit ohne Transformation schneller ist als die mit Transformation (hellgraue und schwarze Säule links). Die geschriebene Datenmenge ist in beiden Fällen gleich, da beide BS-1 im Sekundärspeicher haben. Auch hier ist jedoch bei BS-1 im Sekundärspeicher das Auslagern mit Transformation schneller.

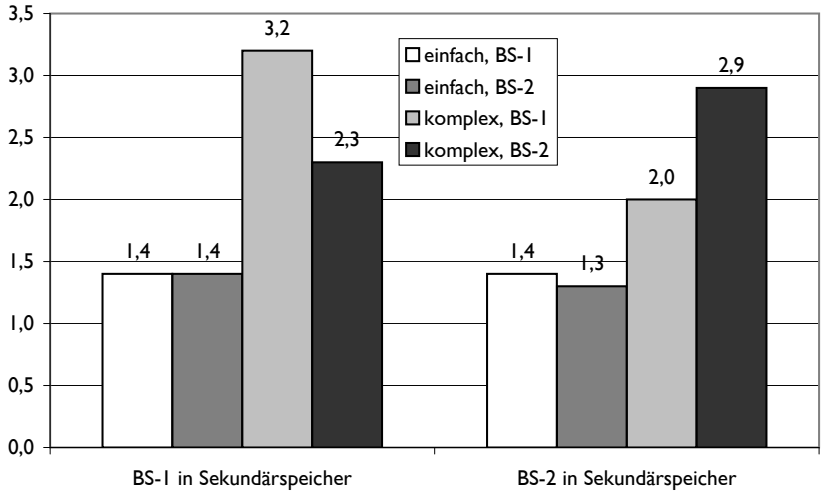


Abbildung 8.10: Auslagern von Prozessvorlagen bei XML-Speicherung

Wie beim Einlagern ist auch beim Auslagern in XML die Variante BS-2 mit oder ohne Transformation im allgemeinen schneller als BS-1.

PostgreSQL Während die Speicherung von Prozessvorlagen als XML-Dateien keine blockorientierte Partitionierung zulässt, ist dies mit der gewählten Sekundärspeicherrepräsentation in einem relationalen DBMS kein Problem. Zwar werden auch hier die Blockinformationen als XML serialisiert und als Tupelattribut abgelegt, die für die Partitionierung notwendigen Informationen sind aber in einem separaten Tupelattribut gespeichert. Über dieses kann direkt auf die einzelnen Blöcke zugegriffen werden.

Die als XML serialisierten Blockinformationen bei BS-1 werden bei PostgreSQL in einem Tupelattribut des Datentyps CLOB oder XML abgelegt. Insbesondere bei Anfragen gegen Kollektionen ist diese Unterscheidung wichtig. Deshalb werden beide Varianten bei allen Laufzeitmessungen separat betrachtet. Gegenüber den Laufzeitmessungen der XML-Speicherung gibt es bei PostgreSQL deshalb vier zusätzliche Säulen. Bei BS-2 gibt es diese Unterscheidung nicht, da hier die Blockstrukturen eine Relation sind und die Daten auch relational abgelegt sind.

Einlagern Abb. 8.11 zeigt die Messergebnisse für das Einlagern von Prozessvorlagen aus PostgreSQL in den Primärspeicher. Die zusätzliche Variante für BS-1 in der Mitte ist vergleichbar

mit BS-1 links. Das heißt, in beiden Fällen ist bei der dunkelgrauen und schwarzen Säule eine Transformation der Blockstrukturen von BS-1 nach BS-2 notwendig. Rechts wird bei der weißen und hellgrauen Säule von BS-2 nach BS-1 transformiert. Dies zeigt sich auch an den Messergebnissen: Mit Transformation ist das Einlagern höchstens genauso schnell oder langsamer als ohne Transformation.

Beim Vergleich der BS-1-Varianten ist das Einlagern mit CLOB-Datentyp schneller als mit XML-Datentyp. Dies kann an einer aufwendigeren Verwaltung des XML-Datentyps in den JDBC-Treibern von PostgreSQL liegen. Etwas überraschend ist die kürzere Laufzeit von BS-1 im Sekundärspeicher bei der komplexen Prozessvorlage sowohl als CLOB- als auch als XML-Datentyp (links und Mitte) und sogar mit Transformation gegenüber BS-2 im Sekundärspeicher (rechts). Bei der einfachen Prozessvorlage ist BS-2 im Sekundärspeicher wie erwartet schneller als BS-1 im Sekundärspeicher. Genauere Untersuchungen zeigen, dass BS-2 im Sekundärspeicher abhängig von der Anteil an Blockstrukturen langsamer wird als BS-1. Zudem spielt auch die Verteilung der Knoten auf die Blöcke und damit die Prozessstruktur insgesamt eine entscheidende Rolle. Ein Grund hierfür können die komplexeren Datenstrukturen sein, die für die relationale Speicherung von BS-2 verwendet werden (s. Abb. 8.3).

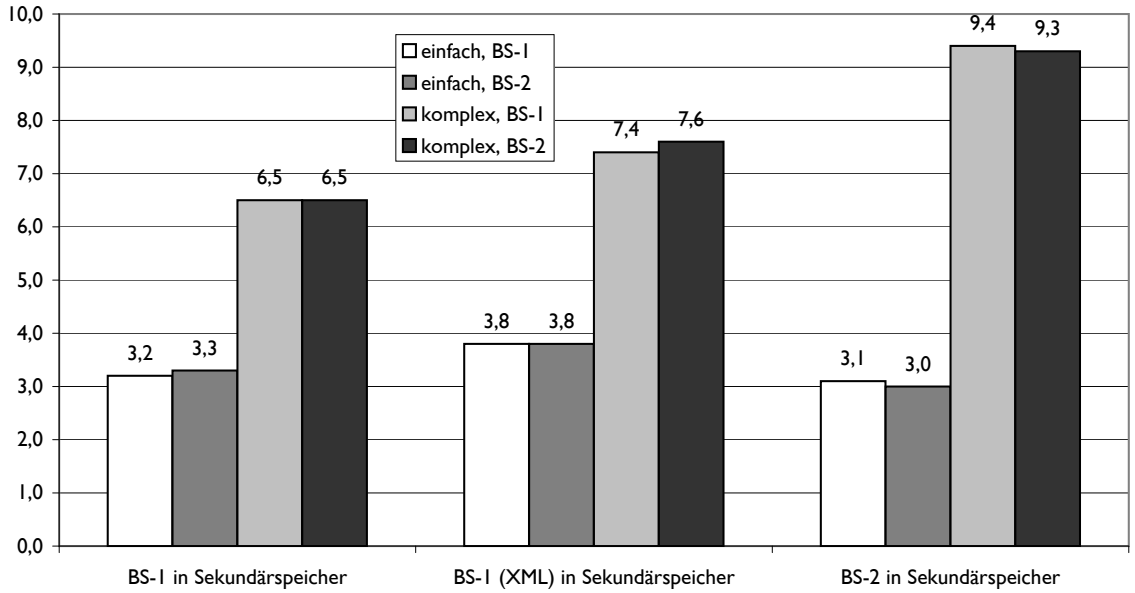


Abbildung 8.11: Einlagern von Prozessvorlagen mit PostgreSQL

Das Einlagern von Prozessvorlagen aus XML-Dateien ist etwas schneller als das Einlagern aus PostgreSQL. Einfache Dateizugriffe in Java sind effizienter als JDBC-Zugriffe, die in der Regel über eine TCP-Verbindung (*Transmission Control Protocol*) erfolgen. Das Einlagern der einfachen Prozessvorlage ist bei XML und bei PostgreSQL ungefähr doppelt so schnell wie das Einlagern der komplexen Prozessvorlage – abgesehen von BS-2 in PostgreSQL.

Auslagern Bei den Laufzeiten für das Auslagern in PostgreSQL (s. Abb. 8.12) zeigt sich bei BS-1 im Sekundärspeicher ein umgekehrtes Bild wie beim Einlagern: Das Auslagern mit XML-Datentyp ist erheblich schneller (Mitte) als das Auslagern mit CLOB-Datentyp. Hier erwartet man eher die gegenteilige Situation, etwa da für den XML-Datentyp zusätzliche

interne Informationen im DBMS gespeichert werden. Dies kann auch an der zusätzlichen Relation liegen, die beim CLOB-Datentyp existiert (s. Abb. 8.1).

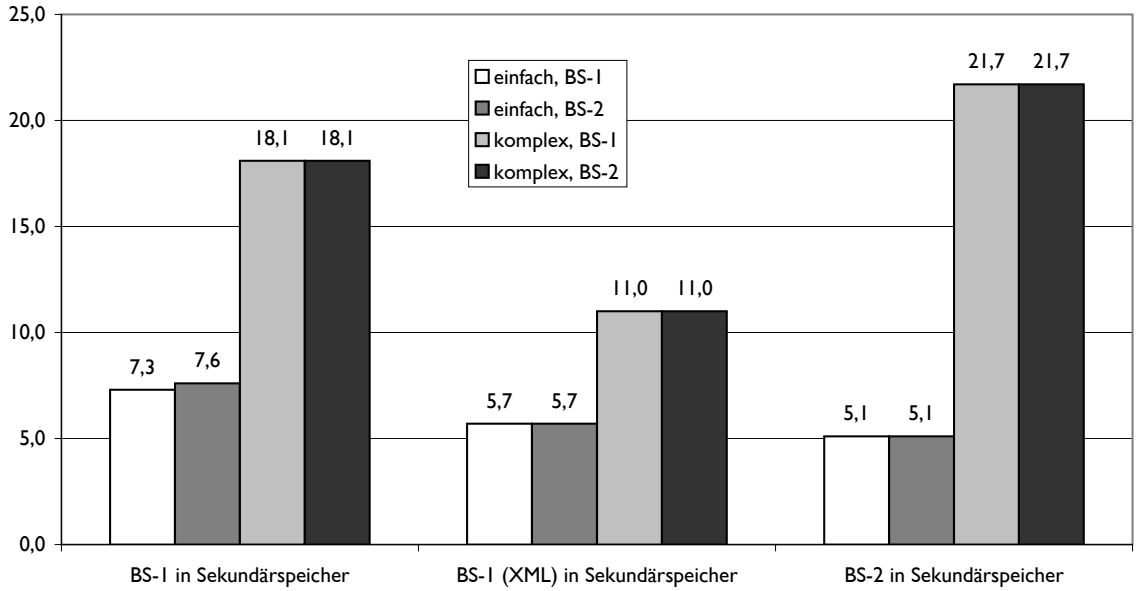


Abbildung 8.12: Auslagern von Prozessvorlagen mit PostgreSQL

Beim Vergleich zwischen BS-1 und BS-2 im Sekundärspeicher zeigt sich eine weitergehende Verschlechterung bei der komplexen Prozessvorlage mit BS-2 im Sekundärspeicher (rechts). Während die einfache Prozessvorlage mit BS-2 im Sekundärspeicher schneller ist als BS-1, ist die komplexe Vorlage um den Faktor 4 langsamer als die einfache Prozessvorlage. Auch beim Auslagern hängt die Verschlechterung mit der Prozessstruktur und dem Anteil an Blockstrukturen zusammen.

8.1.4.2 Ein-/Auslagern von Prozessinstanzen

Prozessinstanzen werden hauptsächlich für das Weiterschalten benötigt. Dementsprechend werden sie oft geändert und werden oft ausgelagert. Für die Anzeige des aktuellen Prozesszustands sowie ggf. für Anfragen gegen Kollektionen von Prozessinstanzen müssen Prozessinstanzen nur eingelagert werden. Insgesamt müssen sie aber wesentlich häufiger ausgelagert werden als Prozessvorlagen.

XML Bei XML-Dateien ist kein direkter Zugriff auf einzelne XML-Elemente möglich, etwa Knotenzustände. Wie Prozessvorlagen müssen deshalb auch Prozessinstanzen bei Speicherung als XML-Dateien immer komplett eingelesen werden. Für ZR-2 ist dies ein kleineres Problem, da hier die Datenmenge meist nicht allzu umfangreich ist. Für ZR-3 kann die Datenmenge aber insbesondere bei großen Prozessinstanzen durchaus zum Problem werden. Wie bei Blockstrukturen wäre eine direkter Zugriff auf Knotenzustände nur möglich, indem Knoten einzeln oder gruppiert in separaten XML-Dateien gespeichert würden. Auch dies ist aber zu aufwendig.

Einlagern Abb. 8.13 zeigt die Messergebnisse für das Einlagern von Prozessinstanzen aus XML-Dateien. Dabei zeigt sich, dass es bei ZR-2 im Sekundärspeicher keine Unterschiede zwischen einfachen (weiße und dunkelgraue Säule links) und komplexen Prozessinstanzen (hellgraue und schwarze Säule links) und zwischen ZR-2 (weiße und hellgraue Säule links) und ZR-3 (dunkelgraue und schwarze Säule links) im Primärspeicher gibt. Selbst einfachen Prozessinstanzen bei ZR-3 im Sekundärspeicher (weiße und dunkelgraue Säule rechts) haben dieselbe Laufzeit. Lediglich bei ZR-3 im Sekundärspeicher dauert das Einlagern der komplexen Prozessinstanz geringfügig länger. Insgesamt ist dieser Laufzeitunterschied aber nicht so groß, um ZR-3 im Sekundärspeicher als Repräsentationsvariante auszuschließen. Die Transformation der Zustandsrepräsentation hat wie die Transformation der Blockstruktur bei (s. Abb. 8.9) keinerlei Auswirkung auf die Laufzeit.

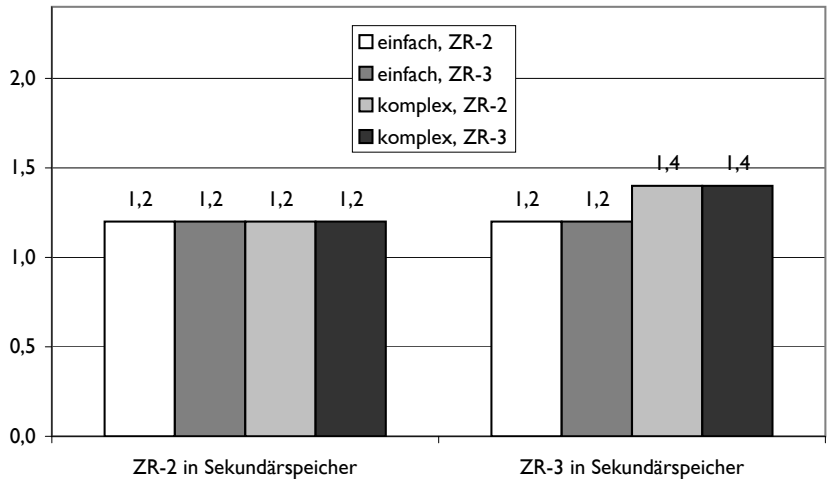


Abbildung 8.13: Einlagern von Prozessinstanzen bei XML-Speicherung

Auslagern Wie beim Einlagern zeigt sich auch beim Auslagern von Prozessinstanzen bei ZR-2 im Sekundärspeicher kein Unterschied zwischen der einfachen und der komplexen Prozessinstanz (links in Abb. 8.14). Bei ZR-3 im Sekundärspeicher fällt sowohl bei der einfachen als auch bei der komplexen Prozessinstanz der Einfluss der Transformation der Zustandsrepräsentation auf. In beiden Fällen ist die Laufzeit mit Transformation zu ZR-2 im Primärspeicher größer (weiße und hellgraue Säule rechts) als wenn ZR-3 im Primärspeicher verwendet wird (dunkelgraue und schwarze Säule rechts). Wie beim Einlagern (s. Abb. 8.13) ist auch beim Auslagern in XML-Dateien ZR-2 besser. Auch hier der Unterschied wieder nicht allzu groß. Wie bei Messungen mit Prozessvorlagen bei XML-Speicherung ist auch bei Prozessinstanzen das Auslagern schneller als das Einlagern.

PostgreSQL Bei Prozessinstanzen werden in PostgreSQL die Zustandsinformationen als Tupelattribut wie Blockstrukturen bei Prozessvorlagen abgelegt. Dazu werden die Daten entsprechend in XML serialisiert. Auch bei den Messungen zum Ein- und Auslagern von Prozessinstanzen werden beide Varianten der Speicherung in PostgreSQL betrachtet: Zustandsinformationen als CLOB- sowie als XML-Datentyp, sowohl für ZR-2 als auch für ZR-3 im Sekundärspeicher.

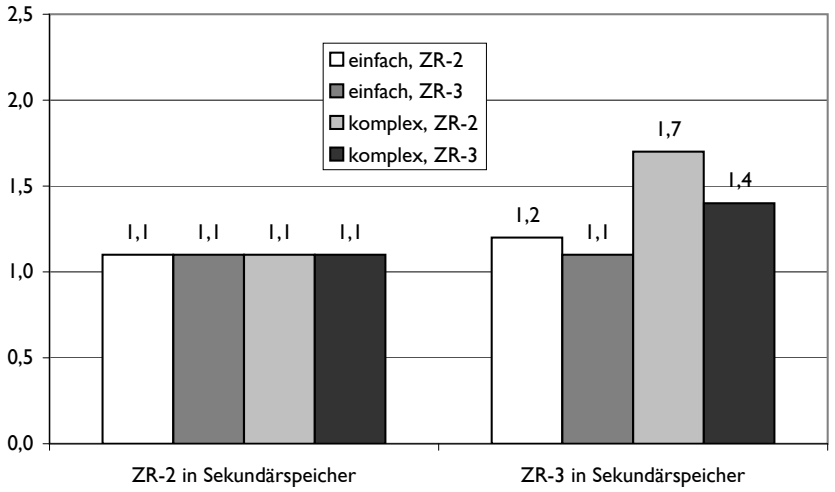


Abbildung 8.14: Auslagern von Prozessinstanzen bei XML-Speicherung

Einlagern Die Messergebnisse beim Einlagern von Prozessinstanzen aus PostgreSQL sind qualitativ vergleichbar mit den Messergebnissen beim Einlagern aus XML-Dateien. Die Transformation der Zustandsrepräsentation hat keine Auswirkungen, und bei ZR-2 gibt es keinen Unterschied zwischen der einfachen und der komplexen Prozessinstanz. Lediglich bei ZR-3 im Sekundärspeicher ist das Einlagern der komplexen Prozessinstanz (hellgraue und schwarze Säule rechts der Mitte) geringfügig langsamer als das Einlagern der einfachen Prozessinstanz (weiße und dunkelgraue Säule rechts der Mitte). Bei der komplexen Prozessinstanz gibt es auch einen geringen Unterschied zwischen dem CLOB- und dem XML-Datentyp. Der XML-Datentyp ist wie bei den Blockstrukturen beim Einlagern von Prozessvorlagen langsamer als der CLOB-Datentyp (Vergleich zwischen rechts der Mitte und rechts).

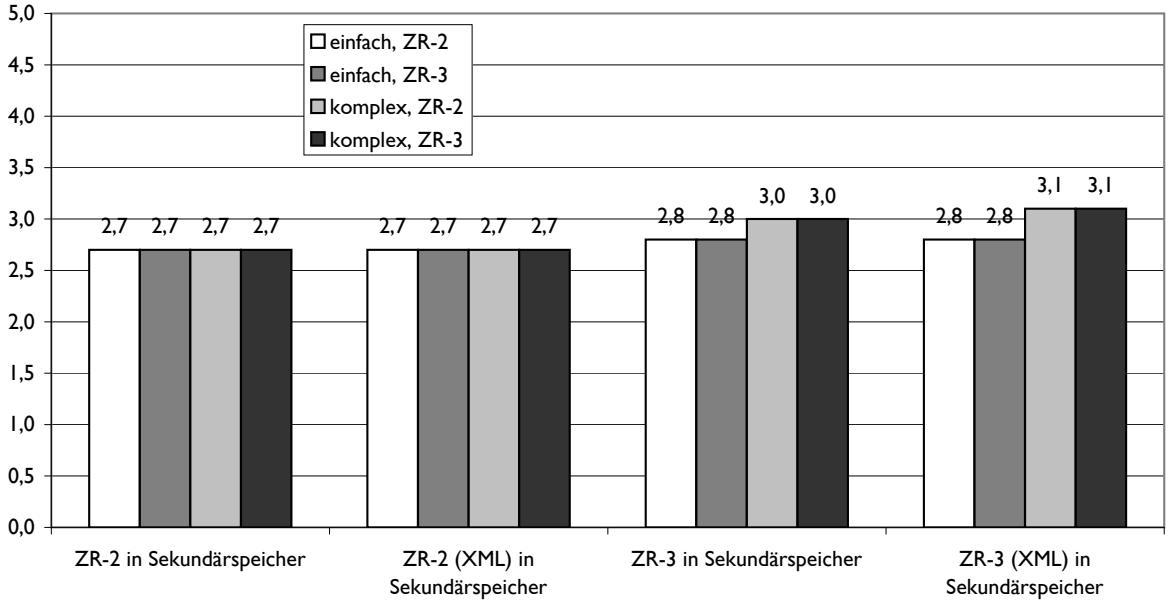


Abbildung 8.15: Einlagern von Prozessinstanzen mit PostgreSQL

Auslagern Beim Auslagern von Prozessinstanzen mit PostgreSQL (s. Abb. 8.16) gibt es bei ZR-2 Sekundärspeicher mit CLOB-Datentyp (links) keine Unterschiede zwischen den Zustandsrepräsentationen im Primärspeicher und zwischen einfacher und komplexer Prozessinstanz. Zudem sind alle Laufzeiten langsamer als ZR-2 mit XML-Datentyp und sogar ZR-3 im Sekundärspeicher. Eigentlich erwartet man hier die kürzesten Laufzeiten. Die entsprechenden Messergebnisse sind nicht erklärbar. Auch die Messungen von ZR-2 mit XML-Datentyp (links der Mitte) zeigen ein unerklärliches Verhalten: bei Verwendung von ZR-3 im Primärspeicher und damit einer zusätzlichen Transformation sind die Laufzeiten kürzer als ohne Transformation. Zudem ist die komplexe Prozessinstanz im Fall von ZR-2 im Primärspeicher (hellgraue Säule) schneller als die einfache Prozessinstanz (weiße Säule); im Fall von ZR-3 ist die komplexe Prozessinstanz genauso schnell wie die einfache Prozessinstanz.

ZR-3 im Sekundärspeicher erfüllt dagegen die Erwartungen: Mit Transformation (weiße und hellgraue Säule rechts der Mitte und rechts) ist das Auslagern langsamer als ohne Transformation (dunkelgraue und schwarze Säule rechts der Mitte und rechts), und das Auslagern der komplexen Prozessinstanz (hellgraue und schwarze Säule rechts der Mitte und rechts) ist langsamer als das Auslagern der einfachen Prozessinstanz (weiße und dunkelgraue Säule rechts der Mitte und rechts). Wie beim Einlagern (s. Abb. 8.15) gibt es nur einen geringen Unterschied zwischen dem CLOB- und dem XML-Datentyp.

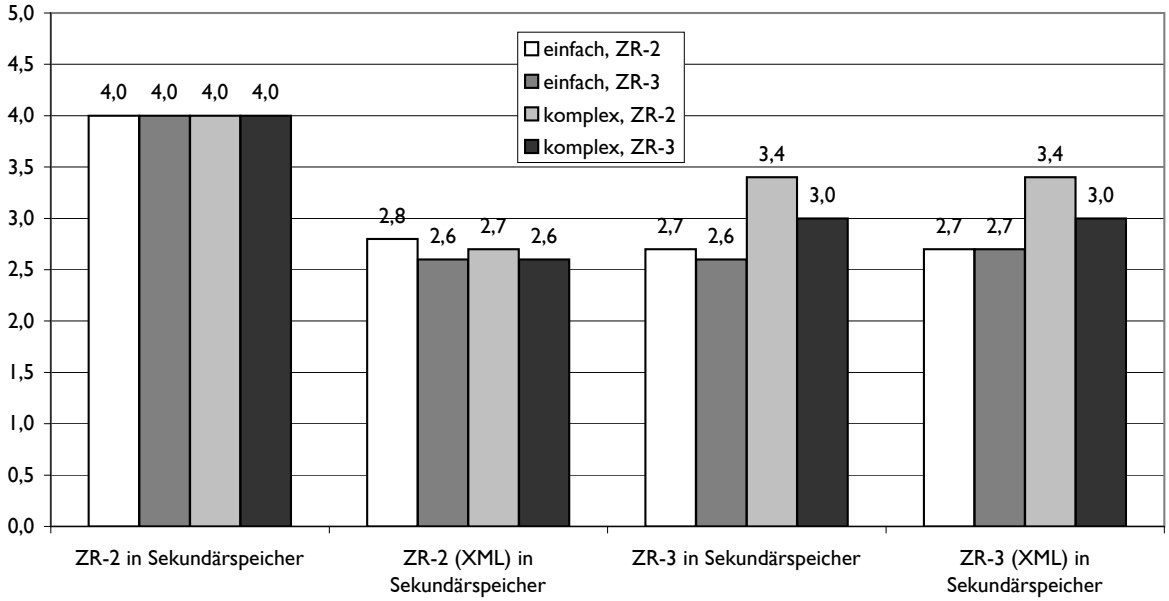


Abbildung 8.16: Auslagern von Prozessinstanzen mit PostgreSQL

8.1.4.3 Anfragen gegen Kollektionen von Prozessinstanzen

Für Anfragen gegen Kollektionen von Prozessinstanzen werden alle Prozessinstanzen einer bestimmten Prozessvorlage (*Instanzen*) sowie alle Prozessinstanzen mit dem Start- und Endknoten in den Zuständen `RUNNING` und `COMPLETED` ermittelt (*StartRunning*, *StartComplete*, *EndRunning* und *EndComplete*). Jede Anfrage wird auf Prozessinstanzen der einfachen Prozessvorlage und der künstlichen Prozessvorlage durchgeführt, wobei die Zustände der Prozessinstanzen jeweils so gewählt sind, dass es für jeden durch die Prozessvorlage definierten Zustand eine

Prozessinstanz gibt. Auch hier wird wieder zwischen der Speicherung als XML-Datei und in PostgreSQL unterschieden.

XML Bei Anfragen gegen Kollektionen von Prozessinstanzen, die als XML-Dateien gespeichert sind, gibt es wie bereits erwähnt keinen direkten Zugriff auf Elemente der XML-Dateien und damit auf die Attribute der Prozessinstanzen. Dementsprechend müssen für die Anfragen alle Prozessinstanzen eingelagert werden, was zu hohen Laufzeiten führt. Beim Einlagern werden die Zustandsrepräsentationen nicht transformiert, d. h. sie sind in Primär- und Sekundärspeicher gleich. Abb. 8.17 zeigt die Messergebnisse der verschiedenen Anfragen.

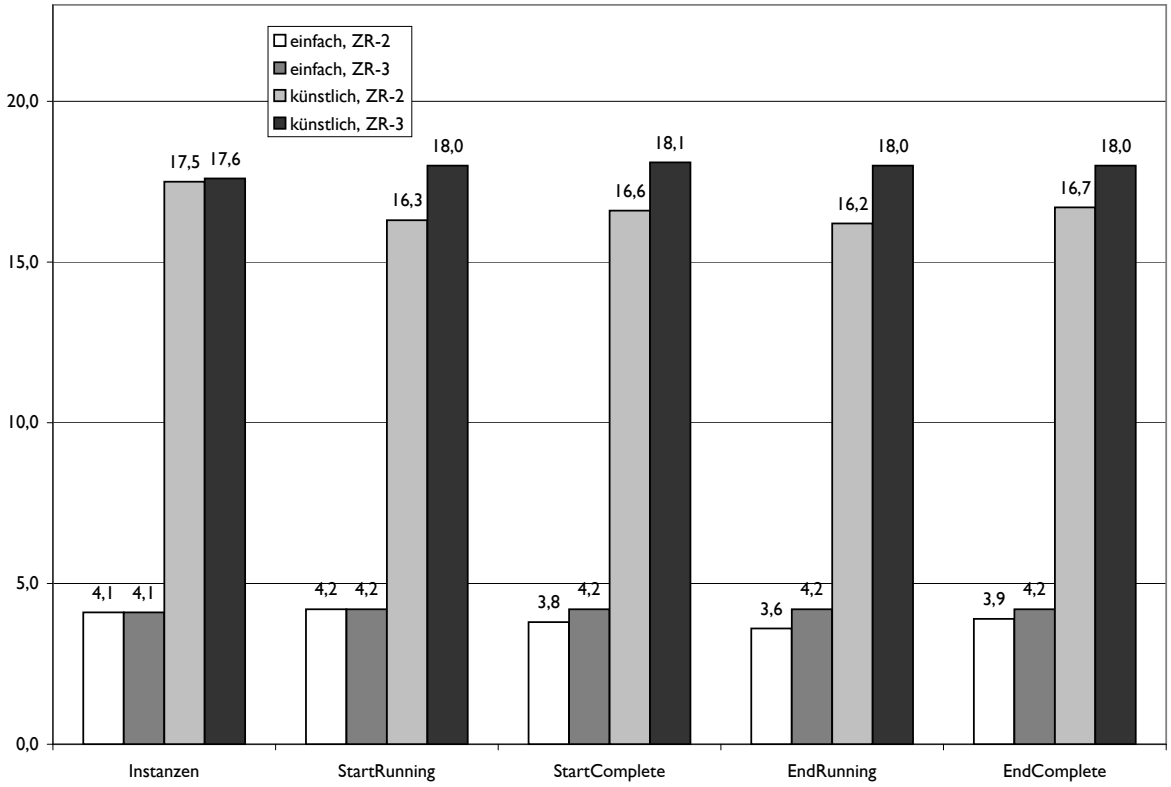


Abbildung 8.17: Anfragen gegen XML-Speicherung

Bei den Prozessinstanzen der einfachen Prozessvorlage gibt es keinen oder nur einen geringen Unterschied zwischen ZR-2 und ZR-3 (weiße und dunkelgraue Säulen). Der Unterschied ist auf die geringere Datenmenge von ZR-2 gegenüber ZR-3 zurückzuführen. Die Anfragebearbeitung selbst ist im Vergleich zum Einlagern der XML-Daten schnell, dementsprechend sind die Unterschiede zwischen den verschiedenen Anfragen gering. Lediglich bei *EndRunning* und *EndComplete* ist bei ZR-2 (weiße Säulen) erwartungsgemäß die Anfrage nach dem explizit verwalteten Zustand `RUNNING` schneller als nach dem Zustand `COMPLETED`, der rekonstruiert werden muss.

Bei den Prozessinstanzen der künstlichen Prozessvorlage (hellgraue und schwarze Säulen) macht sich das aufwendigere Einlagern aus den großen XML-Dateien stark bemerkbar; die Laufzeiten sind um über den Faktor 4 langsamer als die der einfachen Prozessvorlage. Die Unterschiede zwischen ZR-2 (hellgraue Säulen) und ZR-3 (schwarze Säulen) sind größer,

was auch auf die größeren Datenmengen bei ZR-3 zurückzuführen ist. Während der gesuchte Knoten keine Auswirkungen auf die Laufzeit der Anfrage hat, sind bei ZR-2 bei der künstlichen Prozessvorlage die Anfragen nach dem Zustand `RUNNING` schneller als nach dem Zustand `COMPLETED`. Hier macht sich der Rekonstruktionsaufwand aufgrund der größeren Knotenmenge bemerkbar. ZR-3 dagegen hat erwartungsgemäß immer dieselbe Laufzeit unabhängig vom angefragten Knoten und Knotenzustand (dunkelgraue und schwarze Säulen).

Insgesamt zahlt sich bei der XML-Speicherung eine möglichst kleine Datenmenge aus, womit ZR-2 Vorteile gegenüber ZR-3 hat. Der Rekonstruktionsaufwand für Zustände (*StartComplete* und *EndComplete*) ist zwar vorhanden, aber die Laufzeiten sind immer noch deutlich schneller als bei ZR-3.

PostgreSQL Im Fall des relationalen DBMS PostgreSQL werden für die Anfragen an Kollektionen von Prozessinstanzen keine gespeicherten Prozeduren (*Stored Procedure*) verwendet. Dementsprechend müssen beim CLOB-Datentyp alle Prozessinstanzen eingelagert und im Primärspeicher durchsucht werden. Lediglich bei ZR-2 und den Anfragen nach dem Zustand `RUNNING` (ggf. auch beim Zustand `SKIPPED`) können die entsprechende Prozessinstanzen auch beim CLOB-Datentyp im DBMS ermittelt werden, da diese Information direkt als Tupelattribut vorliegt. Dagegen werden beim XML-Datentyp die entsprechenden Möglichkeiten von PostgreSQL genutzt und so die Einlagerung vermieden.

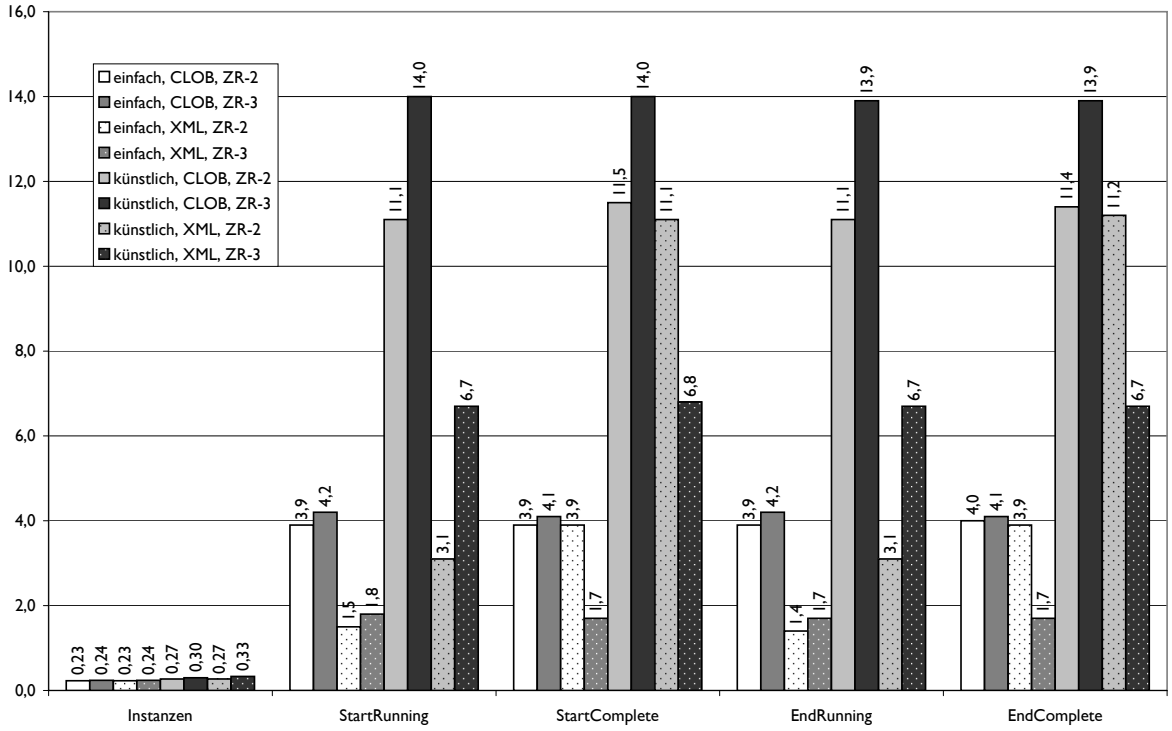


Abbildung 8.18: Anfragen gegen PostgreSQL

Bei den Messergebnissen (s. Abb. 8.18) fällt die kurze Laufzeit von Anfragen nach den Prozessinstanzen einer bestimmten Prozessvorlage auf (ganz links). Für diese Information kann bei der Speicherung in PostgreSQL aus dem Index ermittelt werden. Im Gegensatz zur Speicherung als XML-Dateien (s. Abb. 8.17) müssen keine Daten eingelagert werden.

Bei Anfragen nach bestimmten Knoten und deren Zustände, ist die Speicherung als XML-Datentyp (gepunktete Säulen) meistens wesentlich schneller, höchstens jedoch genauso langsam wie der CLOB-Datentyp (Säulen derselben Farbe ohne Punkte). Dies zeigt, dass die entsprechenden Anfragen von PostgreSQL direkt beantwortet und die Daten nicht eingelagert werden. Die Laufzeiten beim CLOB-Datentyp sind bei einer Zustandsrepräsentation und einem Prozessinstanztyp bei allen vier Anfragen ähnlich: ZR-3 ist bei einfachen Prozessinstanzen (dunkelgraue Säule ohne Punkte) etwas langsamer als ZR-2 (weiße Säule ohne Punkte), bei komplexen Prozessinstanzen ist der Unterschied wesentlich deutlicher. Auch sind die komplexen Prozessinstanzen wesentlich langsamer als die einfachen Prozessinstanzen. Der angefragte Knoten und der angefragte Zustand haben beim CLOB-Datentyp nur geringfügige Auswirkungen auf die Laufzeit; der Aufwand für das Einlagern der Daten überwiegt das Durchsuchen der Daten und Rekonstruieren der Knotenzustände bei ZR-2.

Beim XML-Datentyp gibt es keine wesentlichen Unterschiede zwischen den angefragten Knoten, erwartungsgemäß sind auch die Laufzeiten bei ZR-3 unabhängig vom angefragten Knotenzustand (dunkelgraue und schwarze Säulen mit und ohne Punkte). Allerdings fällt der Knotenzustand bei ZR-2 (weiße und hellgraue Säulen mit und ohne Punkte) signifikant ins Gewicht: während bei *StartRunning* und *EndRunning* ZR-2 (weiße und hellgraue Säulen mit Punkten) etwas schneller ist als ZR-3 (dunkelgraue und schwarze Säulen mit Punkten), ist ZR-2 bei *StartComplete* und *EndComplete* ungefähr um den Faktor 2 langsamer als ZR-3. Bei einfachen Prozessinstanzen ist der Unterschied zwischen ZR-2 (weiße Säulen mit Punkten) und ZR-3 (dunkelgraue Säulen mit Punkten) sogar noch größer im Vergleich zu komplexen Prozessinstanzen. Das liegt daran, dass bei komplexen Prozessinstanzen der Aufwand der Zustandsrepräsentation im Verhältnis zum Aufwand des Einlagerns niedriger ist, was auf die größere Datenmenge bei komplexen Prozessinstanzen zurückzuführen ist.

8.1.4.4 Weiterschalten von Prozessinstanzen

Während die vorherigen Messungen hauptsächlich Laufzeiten bezüglich Primär- und Sekundärspeicher untersucht haben, geht es in diesem Abschnitt um die Laufzeit des Weiterschaltens im Primärspeicher. Da hier nur Datenstrukturen im Primärspeicher betroffen sind, sind die Laufzeiten entsprechend klein, und die Größe der Daten hat weit weniger Einfluss. Es finden keine Transformationen zwischen Datenrepräsentationen statt.

Einen großen Einfluss auf die Laufzeit beim Weiterschalten hat die Prozessstruktur, etwa die Anzahl der ein- und ausgehenden Kanten bei Verzweigungen oder die Anzahl der zu überspringenden Prozessschritte bei alternativen Verzweigungen. Wichtig ist dabei auch die Repräsentation der Prozessstruktur. Dementsprechend erfolgen die Laufzeitmessungen des Weiterschaltens auf den Blockrepräsentationen BS-1 (Blockstrukturen als Objekte) und BS-2 (Blockstrukturen als Relation) sowie den Zustandsrepräsentationen ZR-2 (Implizite Markierungen) und ZR-3 (Explizite Markierungen). Daneben unterscheiden sich die Messungen im Typ des Knotens, der weitergeschaltet wird. Neben normalen Knoten wird die Laufzeit paralleler und alternativer Verzweigungen sowie der parallelen Vereinigung gemessen.

Individuelle Prozessinstanzen Abb. 8.19 zeigt die Messergebnisse des Weiterschaltens unterschiedlicher Knotentypen bei verschiedenen Block- und Zustandsrepräsentationen. Die Messungen der mit * markierten Säulen sind Mittelwerte von Messungen mit starken Schwankungen (großer Varianz). Dementsprechend sind diese nur begrenzt aussagefähig und zeigen nur Tendenzen. Bis auf eine Messung mit starken Schwankungen ist BS-1 (weiße und dunkelgraue

Säulen) immer langsamer ist als BS-2 (hellgraue und schwarze Säulen). Dies kann darauf zurückgeführt werden, dass für das Weiterschalten die Ermittlung direkter Vorgänger- und Nachfolgerknoten wesentlich wichtiger ist als Blockstrukturen. Bei BS-1 erfolgt die Repräsentation der Blockstrukturen jedoch zu Lasten der Vorgänger-/Nachfolgerermittlung, d. h. die Repräsentation der Blockstrukturen verschlechtert die Laufzeit der Vorgänger-/Nachfolgerermittlung. Dagegen sind bei BS-2 Blockstrukturen nur zusätzliche Information zu den normalen Graphstrukturen ohne Auswirkungen auf die Vorgänger-/Nachfolgerermittlung.

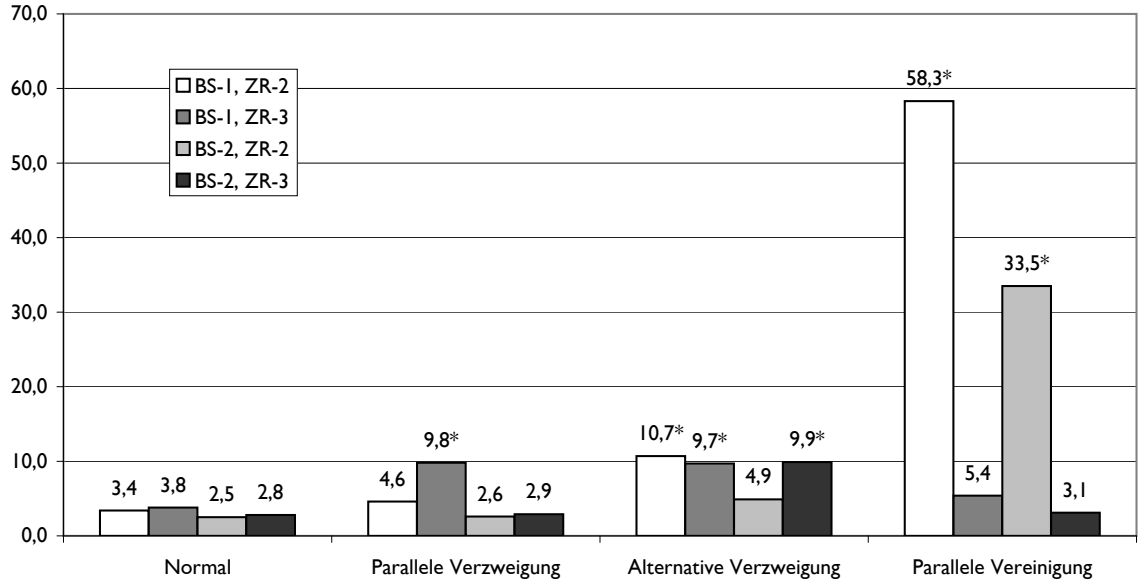


Abbildung 8.19: Weiterschalten mit individuellen Prozessinstanzen

Tendenziell ist ZR-2 (weiße und hellgraue Säulen) bei normalen und Verzweigungsknoten schneller als ZR-3 (dunkelgraue und schwarze Säulen). Hier müssen weniger Daten geändert werden. Dies fällt insbesondere beim Weiterschalten der alternativen Verzweigung auf. Bei ZR-2 wird dabei lediglich der ausgewählte Zweig gespeichert, während bei ZR-3 die Zustände aller Knoten auf abgewählten Zweigen angepasst werden. Allerdings ist ZR-2 bei der parallelen Vereinigung wesentlich langsamer als ZR-3. Bei ZR-3 liegen die Zustände der Vorgängerknoten des parallelen Vereinigungsknotens direkt vor, bei ZR-2 müssen diese meistens erst rekonstruiert werden. Dies führt zur ungefähr elffachen Laufzeit für ZR-2. Die Messergebnisse zeigen, dass sowohl ZR-2 als auch ZR-3 von der Prozessstruktur abhängig sind: ZR-3 bei alternativen Verzweigungen, ZR-2 in noch höherem Maße bei parallelen Vereinigungen.

Zustandsclusterung Die Zustandsclusterung verbessert die Laufzeit der Schemaevolution. Varianten, die keine vollständige Zustandsrepräsentation darstellen, etwa die grobe Clusterung (s. [Glö09] und Abschnitt 6.4.2.1), benötigen jedoch zusätzliche Datenstrukturen. Hierdurch verschlechtert sich naheliegenderweise die für das Weiterschalten benötigte Laufzeit, da zusätzliche Daten angepasst werden müssen. Abb. 8.20 zeigt die Messergebnisse für das Weiterschalten für individuelle Prozessinstanzen wie eben wenn zusätzlich noch die grobe Clusterung (s. [Glö09] und Abschnitt 6.4.2.1) verwendet wird. Das heißt, bei jedem Weiterschalten wird der Zustand einer Prozessinstanz individuell angepasst und sie außerdem bei Bedarf im entsprechenden Clusterbaum umgehängt.

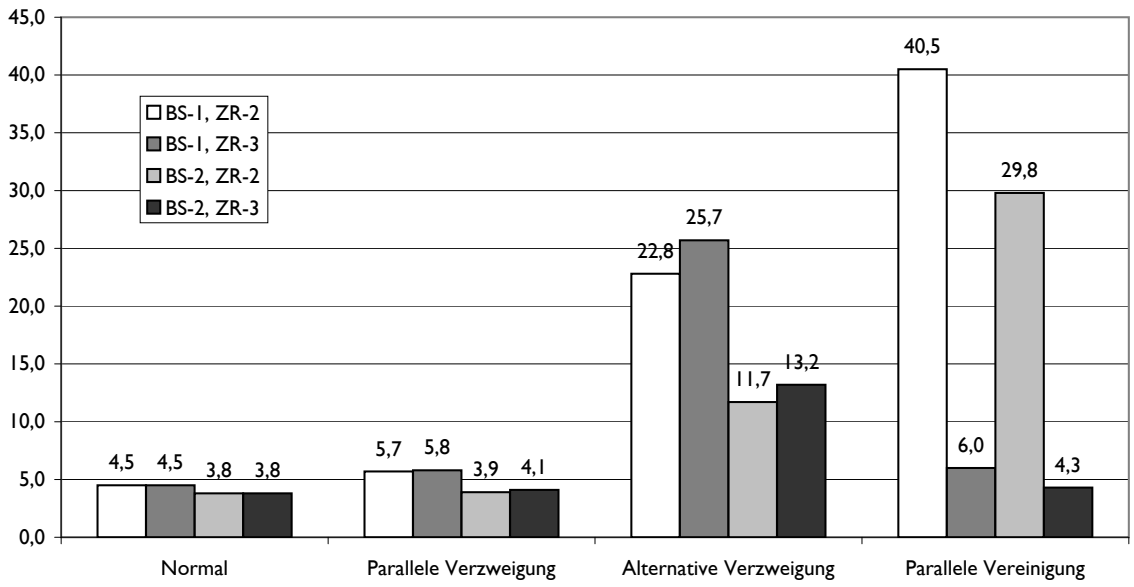


Abbildung 8.20: Weiterschalten mit Clusterung

Da die grobe Clusterung zusätzlich zur individuellen Zustandsanpassung einer Prozessinstanz existiert, ist die Laufzeit immer langsamer als das Weiterschalten ohne Clusterung – von den stark schwankenden und mit * markierten Messungen (s. Abb. 8.19) abgesehen. Auch hier zeigt sich wieder, dass BS-1 (weiße und dunkelgraue Säulen) immer langsamer als BS-2 (hellgraue und schwarze Säulen) ist. ZR-2 (weiße und hellgraue Säulen) ist bei normalen Knoten und einer parallelen Verzweigung wieder etwas langsamer als ZR-3 (dunkelgraue und schwarze Säulen). Bei der alternativen Verzweigung und der parallelen Vereinigung ist der Unterschied von ZR-2 zu ZR-3 im Vergleich zu den anderen Knotentypen größer, er ist jedoch nicht mehr so groß wie ohne Clusterung. Hier macht sich die Laufzeit für die Clusterung bemerkbar, die unabhängig von der Zustandsrepräsentation ist.

Den Laufzeitunterschied der einzelnen Messungen zwischen dem Weiterschalten mit und ohne grobe Clusterung zeigt Abb. 8.21. Dabei ist die Laufzeit des Weiterschaltens mit Clusterung ins Verhältnis zur Laufzeit des Weiterschaltens ohne Clusterung ins Verhältnis gesetzt. Verhältnisse von unter 100% sagen aus, dass das Weiterschalten mit Clusterung schneller ist als ohne. Dies tritt jedoch nur bei den stark schwankenden Messungen aus Abb. 8.19 auf und kann deshalb als Messfehler interpretiert werden. Bei normalen Knoten und parallelen Verzweigungen hat die Clusterung einen größeren Einfluss bei ZR-2 (weiße und hellgraue Säulen) als ZR-3 (dunkelgraue und schwarze Säule). Das liegt daran, dass ZR-2 eine kürzere Laufzeit hat als ZR-3 und damit der zusätzliche Aufwand für die Clusterung mehr ins Gewicht fällt.

Die Messungen bei der alternativen Verzweigung beruht auf stark schwankenden Messungen. Tendenziell ist die Clusterung hier jedoch wesentlich langsamer. In diesem Fall ist die Zuordnung zum Clusterbaum auch am aufwendigsten.

8.1.5 Fazit

Die Messergebnisse zeigen, dass die Betrachtung der Konzepte idealerweise durch eine Simulation konkreter Umsetzungen der Datenstrukturen ergänzt wird. Hier offenbaren sich teilweise überraschende Ergebnisse, die nicht mit den qualitativen Untersuchungen der Kon-

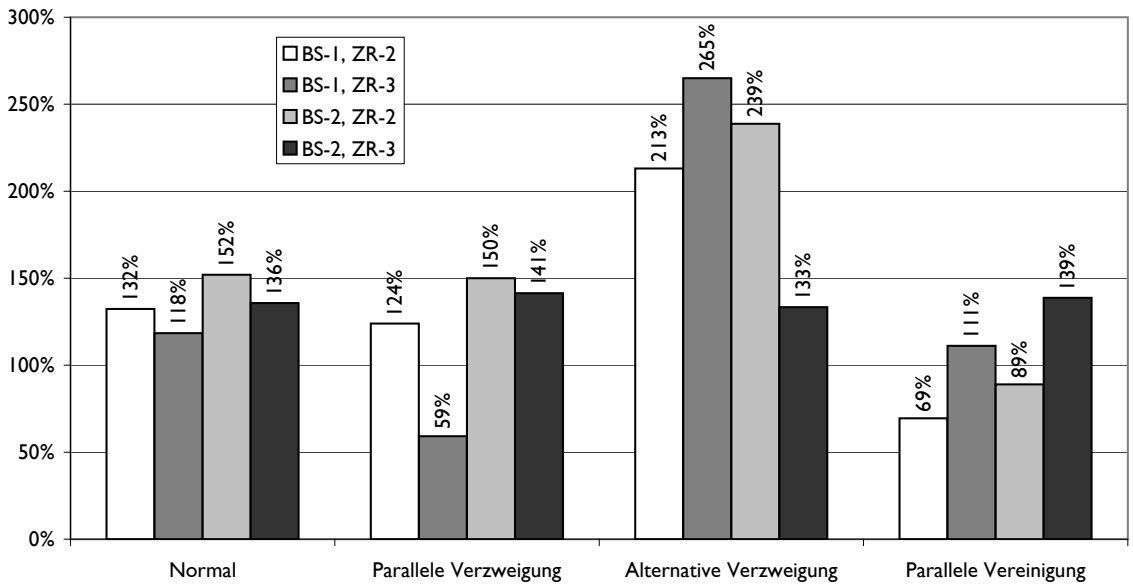


Abbildung 8.21: Laufzeit von Clusterung gegenüber individuellen Prozessinstanzen

zepte übereinstimmen. Ein Beispiel hierfür ist die längere Laufzeit komplexer Prozessvorlagen beim Ein- und Auslagern von BS-2 im Sekundärspeicher mit PostgreSQL (s. Abb. 8.11 und Abb. 8.12). Solche Diskrepanzen können allerdings auch auf die Umsetzung der Testumgebung zurückzuführen sein. So können bereits geringfügig unterschiedliche Implementierungen eines Konzepts zu völlig unterschiedlichen Ergebnissen führen.

Wichtig bei der Interpretation der Messergebnisse ist der Zusammenhang mit dem normalen Betriebs eines PrMS. So werden Prozessvorlagen kaum ausgelagert, aber oft eingelagert. Dementsprechend haben die Messergebnisse für das Einlagern eine größere Bedeutung als die für das Auslagern. Im Gegensatz dazu findet bei Prozessinstanzen Ein- und Auslagern ungefähr gleich oft statt, womit das Auslagern wesentlich höher zu gewichten ist als bei Prozessvorlagen. Zudem werden Prozessvorlagen von mehreren Prozessinstanzen genutzt, so dass Prozessvorlagen im Vergleich zu Prozessinstanzen selten eingelagert werden. Für das Weiterschalten gilt, dass normale Knoten häufiger vorkommen als Verzweigungs- und Vereinigungsknoten.

Aussagen zur Datenrepräsentation im Primärspeicher ergeben sich durch die Messergebnisse für das Weiterschalten. Hier empfiehlt sich BS-2 (Blockstrukturen als Relation) für die Repräsentation von Blockstrukturen. Für die Zustandsrepräsentation ergibt sich keine eindeutig bessere Variante. ZR-2 (implizite Markierungen) ist zwar meist geringfügig schneller als ZR-3 (explizite Markierungen), jedoch hat ZR-2 bei parallelen Vereinigungen eine schlechte Laufzeit. Hier hängt es von der Prozessstruktur ab. Für sequentielle Prozesse ist ZR-2, für parallele Prozesse ZR-3 besser geeignet.

Für die Datenrepräsentation im Sekundärspeicher ist bei der Speicherung als XML-Dateien BS-2 besser geeignet als BS-1. Bei PostgreSQL ist BS-1 bei der Speicherung als CLOB am besten geeignet. Bei der Speicherung von BS-1 als XML-Datentyp ist das Auslagern zwar etwas schneller als bei der Speicherung als CLOB, Prozessvorlagen werden jedoch öfter ein- als ausgelagert, so dass das schnellere Einlagern beim CLOB-Datentyp höher zu gewichten ist.

Für die Zustandsrepräsentation wird ZR-3 im Primärspeicher empfohlen. Diese ist zwar meistens geringfügig langsamer als ZR-2, jedoch wird diese längere Laufzeit durch das wesentlich schnellere Weiterschalten bei parallelen Vereinigungen gegenüber ZR-2 mehr als ausgeglichen. Beim Ein-/Auslagern ist bei der XML-Speicherung ZR-2 im Sekundärspeicher am besten, bei PostgreSQL ist es ZR-2 als XML-Datentyp.

Etwas überraschend ist die wesentlich kürzere Laufzeit beim Ein-/Auslagern von XML-Dateien gegenüber PostgreSQL. Auch die Anfragen an Kollektionen von Prozessinstanzen ist nicht so viel langsamer, dass die Speicherung in XML-Dateien generell ausgeschlossen werden sollte. Dies hängt stark davon ab, wie zuverlässig das für die Speicherung genutzte Dateisystem ist und ob benutzerdefinierte Anfragen an Kollektionen von Prozessinstanzen möglich sein sollen. Während solche benutzerdefinierten Anfragen bei PostgreSQL bei entsprechender Kenntnis von SQL und den zugrundeliegenden Datenstrukturen relativ einfach erstellt werden können, müssen sie bei Speicherung als XML-Dateien aufwendig programmiert werden.

Schemaevolution ist im Vergleich zum Weiterschalten eine eher selten benötigte Funktion. Aufgrund der großen Auswirkungen der groben Clusterung auf die Laufzeit des Weiterschaltens ist von ihrer Verwendung abzuraten. Für eine genauere Abschätzung ob, und ab wann sich die grobe Clusterung lohnt, etwa bei häufiger Schemaevolution oder komplexen Änderungen an Prozessvorlagen mit dementsprechend aufwendigen Zustandsüberprüfungen, sind detailliertere Untersuchungen nötig.

Um die Laufzeiten weiter zu minimieren, ist auch ein Wechsel der verschiedenen Repräsentationsvarianten während der Verwendung von Prozessvorlagen und -instanzen möglich. So bietet sich BS-2 bei einfachen Prozessvorlagen und BS-1 bei komplexen Prozessvorlagen an. Wann zwischen den beiden Repräsentationsvarianten gewechselt wird, muss detailliert simuliert werden. Bei einer solchen Simulation werden mehr Prozessvorlagen und -instanzen unterschiedlicher Größe untersucht als im Rahmen dieser Arbeit. So kann ermittelt werden, ab welchen Größen ein Wechsel der Repräsentationsvariante sinnvoll ist.

Weitere Informationen zur Simulation und zusätzliche Messergebnisse finden sich in [Glö09]. Dazu gehören etwa andere relationale DBMS neben PostgreSQL und insbesondere deren unterschiedliche Fähigkeiten bezüglich dem XML-Datentyp.

8.2 Veränderte Prozessinstanzen

Bei der Simulation veränderter Prozessinstanzen werden die vorgestellten Realisierungskonzepte für individuelle Instanzgraphen, operationale Instanzdeltas und strukturelle Instanzdeltas in den beiden Implementierungsvarianten Substitutionsvorlagen und gekapselte Vorlagen miteinander verglichen [KR10]. Die Messungen werden bezüglich Speicherbedarf sowie Laufzeit von Weiterschalten und der transitiven Vorgängerbestimmung durchgeführt. Sie variieren bezüglich der Anzahl der Änderungen an Prozessinstanzen. Bei der Simulation gehen einige bedeutende Einflüsse der Implementierung mit ein, etwa die laufzeitkritische Durchführung von Mengenoperationen, was in den qualitativen Betrachtungen (s. Kapitel 5) nicht direkt möglich ist. Bevor wir die Messergebnisse vorstellen und interpretieren, erläutern wir zuerst die Testumgebung, in der wir die Messungen durchgeführt haben, sowie den Ablauf der Messungen.

8.2.1 Testumgebung

Die Implementierung der Realisierungskonzepte erfolgt wie bei unveränderten Prozessinstanzen (s. Abschnitt 8.1) in Java. Auch bei veränderten Prozessinstanzen werden die Einflüsse der Laufzeitumgebung auf die Simulation minimiert; es gibt mehrere Messungen pro Testfall und die ersten Testdurchläufe gehen nicht in die Messung mit ein. Die Messungen finden ausschließlich auf Datenstrukturen im Primärspeicher statt. Prozessvorlagen, geänderte Prozessvorlagen (d. h. individuelle Instanzgraphen, Teilvorlagen und Wrapper) und Prozessinstanzen sind jeweils als eigene Javaobjekte realisiert. Wird eine Prozessinstanz geändert, wird das referenzierte Prozessvorlagenobjekt durch ein eigenes Objekt des entsprechenden Realisierungskonzepts ersetzt. Dieses referenziert dann die ursprüngliche Prozessvorlage und ist damit der Prozessinstanz und der ursprünglichen Prozessvorlage zwischengeschaltet. Dies gilt nicht für ÄR-1 (individuelle Instanzgraphen, s. Abschnitt 5.1.1). Hier wird eine neues normales Vorlagenobjekt erzeugt, das die ursprüngliche Prozessvorlage komplett ersetzt. Allerdings werden in der Implementierung zusätzlich die gelöschten Knoten mit im ersetzenden Vorlagenobjekt abgelegt. Dies ist für diese Änderungsrepräsentation eigentlich nicht notwendig, es erleichtert aber die Ermittlung der Unterschiede zwischen dem individuellen Instanzgraphen und der ursprünglichen Prozessvorlage. Die Ermittlung der Unterschiede wird beispielsweise bei Schemaevolution veränderter Prozessinstanzen benötigt (s. Abschnitt 2.3.2.3).

8.2.2 Ablauf der Messung

Ausgangspunkt der Messungen ist eine unveränderte Prozessinstanz. Die entsprechende Prozessvorlage besitzt 100 Knoten, die sequentiell angeordnet sind. Diese Anordnung vereinfacht die Durchführung von instanzspezifischen Änderungen. Deren Auswirkungen sind leichter nachvollziehbar, das Ändern ganzer Blockstrukturen hätte große Auswirkungen auf das Messergebnis und wäre entsprechend schwieriger zu interpretieren. Die Auswirkungen von Blockstrukturen auf die Repräsentation unveränderter Prozessvorlagen ist Gegenstand der vorherigen Simulation (s. Abschnitt 8.1).

Von der unveränderten Prozessinstanz wird der Speicherbedarf für die Datenstrukturen gemessen, die für die Repräsentation der instanzspezifischen Änderungen notwendig sind. Der Speicherplatz für die Instanzdaten (z. B. Zustandsinformation) sowie der von der ursprünglichen Prozessvorlage belegte Speicher sind nicht in der Messung enthalten, da diese unabhängig von der gewählten Änderungsrepräsentation sind. Der in diesen Messungen ermittelte Speicherbedarf wird absolut angegeben.

Daneben wird die Zeit für die Bestimmung aller Vorgänger des letzten Knotens der Sequenz sowie die Zeit für das Weiterschalten eines Knotens ermittelt. Um Laufzeitunterschiede einzelner Messungen zu minimieren, wird die Vorgängerbestimmung 10.000 Mal durchgeführt und alle Ergebnisse arithmetisch gemittelt. Auch das Weiterschalten wird 10.000 Mal durchgeführt. Allerdings wird zusätzlich jeder Knoten der Prozessinstanz weitergeschaltet, d. h. jeder Knoten der Prozessinstanz wird pro Durchlauf einmal gestartet und einmal beendet. Das Ergebnis wird anschließend durch die Anzahl der Knoten (100) dividiert, um somit den Mittelwert für das Weiterschalten eines Knotens dieser Prozessinstanz zu erhalten, bevor auch hier der Mittelwert über alle 10.000 gemessenen Zeiten gebildet wird. Die Implementierung der Prozessinstanz beruht auf Vorlagenreferenzen mit expliziten Markierungen (VG-2, s. Abschnitt 4.1.3.1, und ZR-3, s. Abschnitt 4.1.3.2). Die transitive Vorgängerbestimmung ist ein komplexer Datenzugriff. Dieser erfolgt iterativ, wobei in jedem Schritt die jeweiligen Vorgänger eines Knotens ermittelt

und der Ergebnismenge hinzugefügt werden. Dabei dürfen die Vorgänger jedes Knotens nur einmal ermittelt werden, um nicht in eine Endlosschleife zu geraten.

Die Zeitmessungen bei der Vorgängerbestimmung und beim Weiterschalten werden jeweils in Relation zur Laufzeit einer unveränderten Instanz gesetzt. Das heißt, ohne Änderungen ist der relative Zeitbedarf sowohl für das Weiterschalten als auch für die transitive Vorgängerbestimmung jeweils 1. Bei der Implementierung von ÄR-2 (operationale Instanzdeltas, s. Abschnitt 5.1.2) wird vor einem Zugriff auf die Prozessinstanz die Änderungsoperationen auf die ursprünglicher Prozessvorlage angewendet und somit jedesmal temporär der entsprechend resultierende individuelle Instanzgraph erstellt. Beim Weiterschalten passiert dies bei jeder Zustandsänderung eines Knotens, bei der transitiven Vorgängerbestimmung wird derselbe individuelle Instanzgraph für alle transitiven Vorgänger verwendet.

Da die Messungen bezüglich der Anzahl an Änderungen ermittelt werden, wird nach dem Durchlaufen aller Messungen wie eben beschrieben bei jedem Realisierungskonzept eine Instanzänderung durchgeführt. Diese beinhaltet das Hinzufügen eines Knotens in die Sequenz sowie das Entfernen eines bereits vorhandenen Knotens. Dadurch bleibt die Anzahl der Knoten der Prozessvorlage über alle Messungen konstant. Bei jedem Durchlauf wandert jedoch ein Knoten von der ursprünglichen Prozessvorlage auf die geänderte Prozessvorlage (s. Abb. 8.22). Dies erfolgt für jeden Knoten der ursprünglichen Sequenz einmal, so dass am Ende zwar dieselben Graphstrukturen wie zu Beginn des Tests existieren, diese jedoch ausschließlich über Änderungen erzeugt wurden und die ursprüngliche Prozessvorlage komplett überlagert ist. Damit sind pro Realisierungskonzept 100 Messdurchläufe nötig. Hierdurch erhalten wir die gewünschten Messergebnisse pro Realisierungskonzept abhängig von der Anzahl an durchgeführten Änderungen.

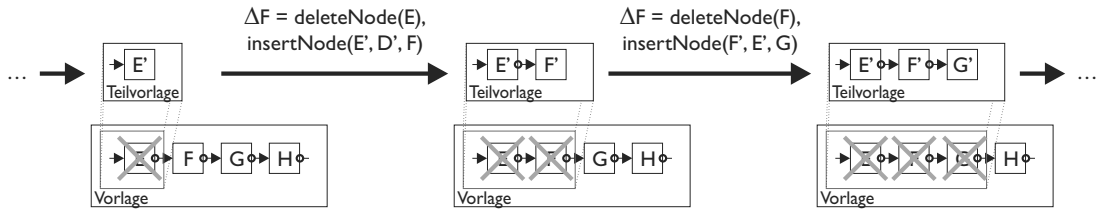


Abbildung 8.22: Knotenwandern im Laufe der Messung

8.2.3 Messergebnisse für den Speicherbedarf

Abb. 8.23 zeigt den absoluten Speicherbedarf der verschiedenen Implementierungen in Abhängigkeit von der Anzahl der vorgenommenen Instanzänderungen. Am Speicherbedarf von 0 Byte bei 0 Änderungen ÄR-2 wird deutlich, dass sich der gemessene Speicherbedarf nur auf die Repräsentation von instanzspezifischen Änderungen bezieht. Alle anderen Änderungsrepräsentationen benötigen bereits bei 0 Änderungen Speicher für Datenstrukturen, etwa die Zuordnung eines Knotens zu seiner Teilvorlage bei ÄR-3-1 (s. Abschnitt 5.4.1). Diese Datenstrukturen werden jedoch sinnvollerweise erst dann angelegt, wenn die entsprechende Prozessinstanz zum ersten Mal geändert wird.

Auffällig an der Messung ist der schnelle Anstieg bei Realisierungskonzept ÄR-2 (operationale Instanzdeltas), obwohl man intuitiv hierfür eigentlich mit weniger Speicherbedarf als bei den anderen Realisierungskonzepten rechnet. Das liegt in dieser Implementierung daran, dass für jede durchgeführte Änderungsoperation ein eigenes Änderungsobjekt angelegt wird, was

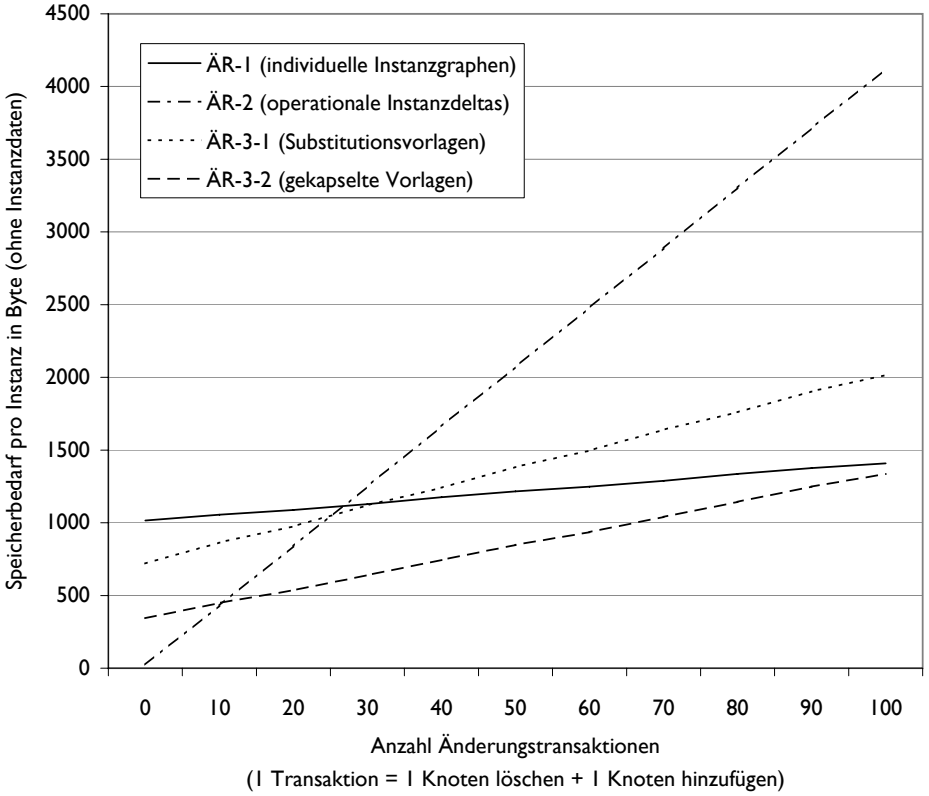


Abbildung 8.23: Speicherbedarf für Realisierungskonzepte

relativ viel Speicher für Objektmetainformationen benötigt. Eine andere, potentiell bessere Realisierungsmöglichkeit ist, Zeichenketten anstelle von Änderungsobjekten zu verwenden (z. B. „DeleteNode B“), vergleichbar einem Änderungsprotokoll. Eine Anpassung der Umsetzung führte jedoch überraschenderweise zu einem noch schnelleren Anstieg des Speicherbedarfs. Der Grund hierfür könnte in der relativ aufwendigen Repräsentation von Zeichenketten in Java (2 Byte pro Zeichen) liegen. Deshalb ist Realisierungskonzept ÄR-2, wie in Abb. 8.23 zu sehen, nur bis zu einer mittleren Anzahl für die Repräsentation von Änderungen geeignet.

Realisierte Graphstrukturen (Realisierungskonzept ÄR-1, Implementierungsvarianten ÄR-3-1 und ÄR-3-2) benötigen ab einer mittleren Anzahl von Änderungen wesentlich weniger Speicherplatz als ÄR-2: Implementierungsvariante ÄR-3-2 ist im vorliegenden Szenario ab 10 Änderungstransaktionen (d. h. 10 Einfügungen und 10 Löschungen) besser, ab 25 Änderungen ist Implementierungsvariante ÄR-3-1 und ab 30 Änderungen sogar Realisierungskonzept ÄR-1 (vollständige individuelle Instanzgraphen) besser als Implementierungsvariante ÄR-1. Der Speicherbedarf für die Implementierungsvarianten ÄR-3-1 (Substitutionsvorlagen) und ÄR-3-2 (gekapselte Vorlagen, s. Abschnitt 5.4.2) besitzt eine vergleichbare Steigung. Die Implementierungsvariante ÄR-3-1 ist aber aufgrund der redundanten Information (z. B. Kontextknoten) insgesamt größer.

Realisierungskonzept ÄR-1 (individuelle Instanzgraphen) ist erwartungsgemäß für wenige Änderungen sehr groß. Es benötigt jedoch bei steigender Anzahl von Änderungen nur wenig zusätzlichen Speicherbedarf. Dieser Anstieg ist ausschließlich darauf zurückzuführen, dass gelöschte Knoten bei dieser Implementierung explizit verwaltet werden, d. h. sie werden

als gelöscht markiert. Kanten werden dagegen physisch gelöscht. Würde man auch Knoten physisch löschen, wäre der Speicherbedarf von individuellen Instanzgraphen in dem verwendeten Anwendungsfall konstant, da pro Änderungstransaktion ein Knoten hinzugefügt und einer entfernt wird.

8.2.4 Messergebnisse für die Laufzeit von Zugriffen

Abb. 8.24 zeigt das Messergebnis für das Weiterschalten abhängig von der Anzahl Änderungen für die verschiedenen Änderungsrepräsentationen jeweils relativ zum Weiterschalten bei einer unveränderten Prozessinstanz. Abb. 8.25 zeigt die Ergebnisse für die Bestimmung aller transitiven Vorgänger des letzten Knotens der getesteten Knotensequenz.

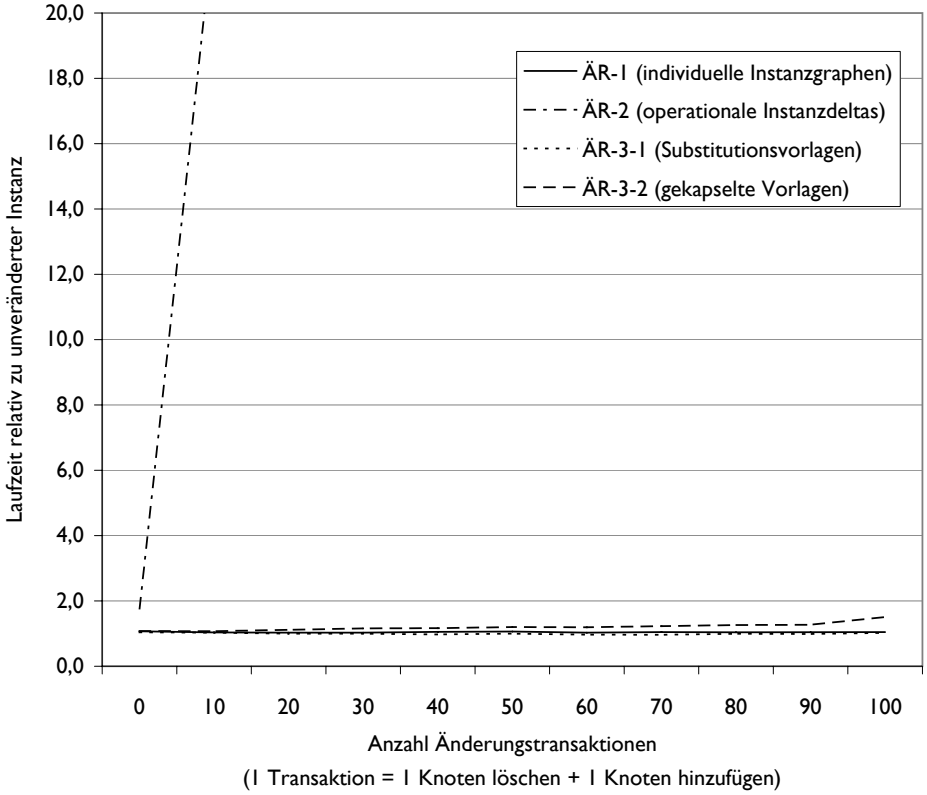


Abbildung 8.24: Zugriffsdauer beim Weiterschalten

Auffällig ist in beiden Fällen die hohe Laufzeit von Realisierungskonzept ÄR-2. Insbesondere beim Weiterschalten (s. Abb. 8.24) ist der Anstieg bezüglich der Anzahl der Änderungen stark ausgeprägt. Dies liegt an der aufwendigen temporären Rekonstruktion der aus den Änderungen resultierenden logischen Prozessvorlage. Diese erfolgt im Falle des Weiterschaltens vor jedem Zugriff, während sie bei der Vorgängerbestimmung nur einmal erfolgt. Deshalb fällt der Aufwand für die Rekonstruktion gegenüber dem Aufwand für den Zugriff wesentlich mehr ins Gewicht als bei der Vorgängerbestimmung.

Bei Realisierungskonzept ÄR-1 gibt es sowohl beim Weiterschalten als auch bei der transitiven Vorgängerbestimmung kaum Unterschiede in der Laufzeit verglichen mit einer unveränderter Instanz. Dies ist einleuchtend, da der Zugriff beides Mal identisch abläuft, d. h. es gibt keinerlei

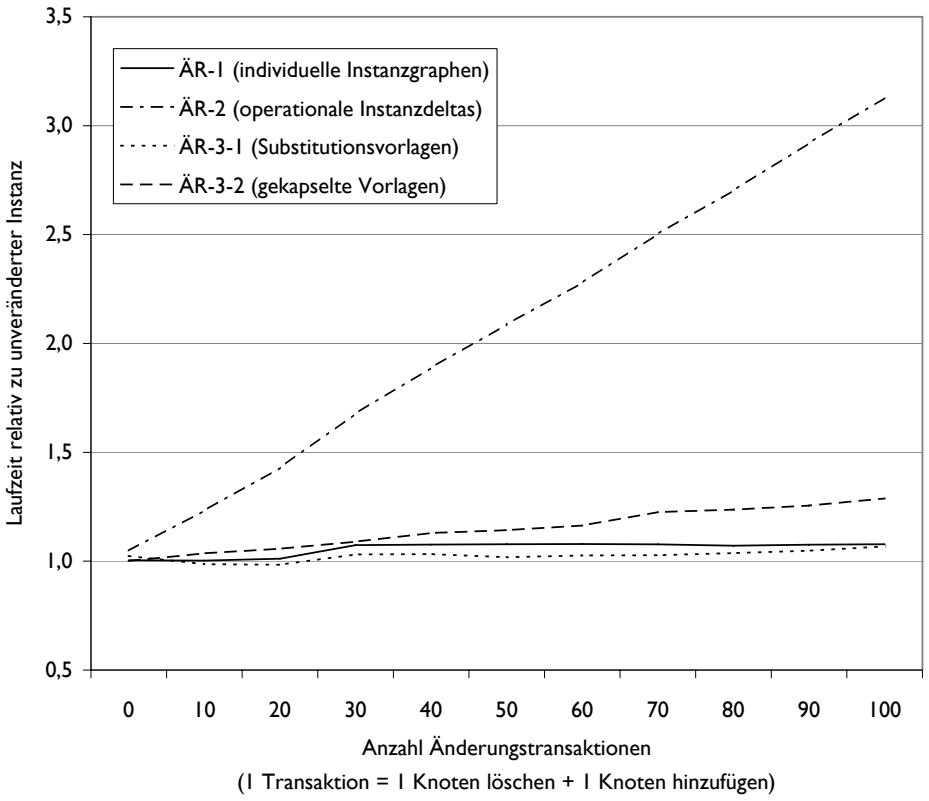


Abbildung 8.25: Zugriffsdauer bei der transitiven Vorgängerbestimmung

Unterschied zwischen dem Zugriff auf eine unveränderte Prozessinstanz und einer veränderten Prozessinstanz mit individuellem Instanzgraphen. Ein ähnliches Laufzeitverhalten besitzt Implementierungsvariante ÄR-3-1. Dies zeigt, dass die Bestimmung der gültigen Vorlage vor jedem Knotenzugriff keine wesentlichen Auswirkungen hat. Der leichte Anstieg der Zugriffszeit sowohl bei individuellen Instanzgraphen als auch im Zusammenhang mit Substitutionsvorlagen bei größerer Anzahl an Änderungen kann mit dem höheren Speicherbedarf erklärt werden. Bei beiden Verfahren steigt dieser gegenüber einer unveränderten Instanz an (s. Abschnitt 8.2.3), wodurch die Laufzeitumgebung (z. B. Speicherverwaltung) beeinträchtigt wird. Dieser wiederum beeinflusst indirekt die Laufzeit der ausgeführten Funktion (Weiterschalten und Vorgängerbestimmung).

Bei Implementierungsvariante ÄR-3-2 ist der Zugriff etwas langsamer als bei Implementierungsvariante ÄR-3-1 und Realisierungskonzept ÄR-1, da aufwendigere Mengenoperationen nötig werden, z. B. zum Filtern von gelöschten Knoten. Mit steigender Anzahl an Änderungen werden diese Operationen noch aufwendiger, da die zu durchsuchende Knoten- und Kantenmenge größer wird, was den relativen Anstieg gegenüber den beiden schnelleren Verfahren erklärt.

8.2.5 Fazit

Die Realisierung instanzspezifischer Änderungen mittels Realisierungskonzept ÄR-2 (operationale Instanzdeltas) ist aufgrund der Zugriffszeit und des Speicherbedarfs nicht für die Repräsentation im Primärspeicher geeignet. Aufgrund der Zugriffszeiten ist Implementierungs-

Tabelle 8.1: Vergleich der Simulation von Blockstrukturen

Variante im Sek.-Sp.	BS-1 im Pr.-Sp. (O)		BS-2 im Pr.-Sp. (+)	
	Einl.	Ausl.	Einl.	Ausl.
BS-1 in XML-Dateien	O	-	O	+
BS-2 in XML-Dateien	+	+	+	O
BS-1 als CLOB in DBMS	+	O	+	O
BS-1 als XML in DBMS	O	+	O	+
BS-2 in DBMS	O	O	O	O

Bewertung:
+: gut
O: neutral
-: schlecht

variante ÄR-3-1 eine sehr gutes Konzept bei geringem Speicherbedarf. Die prototypische Implementierung offenbarte jedoch eine komplexe und damit fehleranfällige Realisierung. Insbesondere die Verwaltung der Kanten hat sich als problematisch herausgestellt, da bei zu ändernden oder zu löschenden Kanten vorab nicht immer festgestellt werden kann, ob die Kante bereits Bestandteil des Substitutionsobjekts oder nur in der ursprünglichen Vorlage vorhanden ist. Im ersten Fall muss sie ggf. physisch gelöscht werden, im zweiten Fall nicht, da dies Auswirkungen auf alle Instanzen der entsprechenden Vorlage hätte. Bei komplexen Änderungen ist zudem eine Verschlechterung des Speicherbedarfs zu erwarten, da ein größerer Änderungskontext zu speichern ist.

Realisierungskonzept ÄR-1 benötigt bei wenigen Änderungen vergleichsweise viel Speicherplatz, weshalb sich diese Variante erst ab einer Vielzahl von Instanzänderungen empfiehlt. Dies lässt sich gut in Kombination mit gekapselten Vorlagen (ÄR-3-2) realisieren, da von ÄR-3-2 leicht zu ÄR-1 gewechselt werden kann, sobald eine gewisse Anzahl an Änderungen erreicht ist. Variante ÄR-3-2 benötigt zudem am wenigsten Speicherplatz (bis 100 einfache Änderungen). Sie hat zwar etwas schlechtere Zugriffszeiten als Implementierungsvariante ÄR-3-1 und Realisierungskonzept ÄR-1, die Zugriffszeiten sind jedoch immer noch effizient. Zudem ist Variante ÄR-3-2 einfach zu implementieren.

8.3 Zusammenfassung

In diesem Kapitel haben wir verschiedene Repräsentationsvarianten aus den vorherigen Kapiteln im Rahmen von Simulationen quantitativ hinsichtlich Speicherbedarf und Laufzeit verglichen. Einige Messergebnisse weichen dabei von den durch die abstrakten Betrachtungen erwarteten Ergebnissen ab. Zudem liegen die Messergebnisse oft dicht beieinander, so dass Einflüsse der Testumgebung großen Einfluss haben und keine eindeutigen Interpretationen möglich sind. Dies trifft insbesondere auf den Vergleich der Repräsentation von Blockstrukturen und der Zustandsrepräsentation zu.

Tabelle 8.1 fasst die Messergebnisse der Simulation von Blockstrukturen zusammen. Variante BS-2 (Blockstrukturen als Relation) ist die beste Repräsentation für den Primärspeicher. Für den Sekundärspeicher gilt dies nur für XML-Dateien. Für die Sekundärspeicherrepräsentation als CLOB ist Variante BS-1 (Blockstrukturen als Objekte) am besten, inklusive einer Transformation zu BS-2 beim Einlagern in den Primärspeicher. Dabei wird das Einlagern höher gewichtet, da Prozessvorlagen öfter ein- als ausgelagert werden.

Bei der Simulation der Zustandsrepräsentation erweist sich Variante ZR-2 (implizite Markierungen) als die beste Repräsentation, sowohl für Primär- als auch Sekundärspeicher (s. Tabelle 8.2). Das gilt sowohl für das Ein-/Auslagern als auch für Anfragen gegen Kollektionen von

Tabelle 8.2: Vergleich der Simulation von Zustandsrepräsentationen

Variante im Sek.-Sp.	ZR-2 im Pr.-Sp. (+)		ZR-3 im Pr.-Sp. (-)		Anfr. g. Koll.
	Einl.	Ausl.	Einl.	Ausl.	
ZR-2 in XML-Dateien	+	+	+	+	+
ZR-3 in XML-Dateien	+	-	+	O	O
ZR-2 als CLOB in DBMS	+	-	+	-	O
ZR-2 als XML in DBMS	+	+	+	+	+
ZR-3 als CLOB in DBMS	O	O	O	+	-
ZR-3 als XML in DBMS	O	O	O	+	+

Bewertung:
+: gut
O: neutral
-: schlecht

Tabelle 8.3: Vergleich von Block- und Zustandsrepräsentation am Anwendungsfall Weiterschalten

	ZR-2	ZR-3
BS-1	-	-
BS-2	+	+

Bewertung:
+: gut
-: schlecht

Prozessinstanzen. Sowohl bei der Speicherung in Dateien als auch in einem DBMS empfiehlt sich die Verwendung von XML.

Den Einfluss der Repräsentation von Blockstrukuren und der Zustandsrepräsentation auf die Laufzeit des Weiterschaltens fasst Tabelle 8.3 zusammen. Variante BS-2 (Blockstrukuren als Relationen) hat dabei die kürzeste Laufzeit. Zwischen den Zustandsrepräsentationen gibt es nur geringe Unterschiede, so dass die Laufzeit des Ein-/Auslagerns als Entscheidung für die Zustandsrepräsentation herangezogen werden kann.

Tabelle 8.4 fasst das Ergebnis der Simulation verschiedener Repräsentationen für instanzspezifische Änderungen zusammen. Hier ist Variante ÄR-2 (operationale Instanzdeltas) überraschend schlecht, während Variante ÄR-1 (individuelle Instanzgraphen) gut abschneidet und selbst ihr Speicherbedarf relativ gering ist. Variante ÄR-3-1 (Substitutionsvorlagen) ist insgesamt die beste. Allerdings hat sich deren Implementierung als sehr komplex und fehleranfällig erwiesen. Insofern stellt ÄR-3-2 (gekapselte Vorlagen) einen guten Kompromiss dar, der zudem den dynamischen Wechsel auf Variante ÄR-1 zulässt, sobald eine Prozessinstanz eine gewisse Anzahl an instanzspezifischen Änderungen aufweist.

Tabelle 8.4: Vergleich der Simulation von Änderungsrepräsentationen

Variante	Speicherbedarf	Weiterschalten	Vorgängerbest.
ÄR-1	O	+	+
ÄR-2	-	-	-
ÄR-3-1	O	+	+
ÄR-3-2	+	O	O

Bewertung:
+: gut
O: neutral
-: schlecht

9 Machbarkeitsimplementierung

In diesem Kapitel stellen wir anhand von Bildschirmfotos (Screenshots) eine Machbarkeitsimplementierung vor. Diese realisiert die vorgestellte Architektur und setzt die funktionalen Anforderungen um. Daneben geben wir Einblicke in andere Forschungsprojekte, deren Machbarkeitsimplementierungen auf der Architektur aufbauen. Diese nutzen etwa die Programmierschnittstelle (FA 12) zur Umsetzung eigener prozessorientierter Anwendungen.

9.1 Prozessausführung

Abb. 9.1 zeigt die Ausführung von Prozessschritten (FA 1) in einem Klientenprogramm. Ganz oben befinden sich eine Auswahlliste mit der Stelle aus dem Organisationsmodell, die der angemeldete Benutzer aktuell verwendet. Daneben befinden sich einige Einstellmöglichkeiten, etwa das Setzen von Abwesenheit. Darunter befindet sich die Arbeitsliste, die die Prozessschritte gerade laufender Prozessinstanzen anzeigt, gefiltert nach aktuellem Benutzer. Das heißt, das sind die Prozessschritte, die für den aktuell angemeldeten Benutzer und die aktuell verwendete Stelle verfügbar, unterbrochen, zugeordnet oder laufend sind. Unterhalb der Arbeitsliste befinden sich zusätzliche Informationen zum ausgewählten Arbeitslisteneintrag, etwa eine genauere Beschreibung des entsprechenden Prozessschritts. Ganz unten wird eine Prozessinstanz mit ihren Prozessschritten und deren Zuständen angezeigt.

Die Ausführung eines Prozessschritts mit Formular ist in Abb. 9.2 zu sehen. Das entsprechende Formular wird im rechten Teil angezeigt. Im unteren Bereich des Formulars gibt es verschiedene Interaktionsmöglichkeiten für das Beenden des Anwendungsbausteins, wobei jede Interaktionsmöglichkeit zu einem anderen Zustand des entsprechenden Prozessschritts führt. *Confirm* beendet den Prozessschritt, *Suspend* speichert die Eingaben und ermöglicht die spätere Fortsetzung der Ausführung, *Reset* verwirft die eingegebenen Daten und macht den Prozessschritt neu verfügbar. *Fail and discard* lässt den Prozessschritt fehlschlagen. Das Formular selbst bietet Rückmeldung bezüglich der notwendigen Daten. So erfordert der ausgeführte Prozessschritt einen Wert für den Ausgabeparameter *Requested quantity*. Im Formular ist noch kein entsprechender Wert eingegeben, was zum einen graphisch beim entsprechenden Formularfeld angezeigt wird. Zum anderen lässt sich der Prozessschritt über das Formular

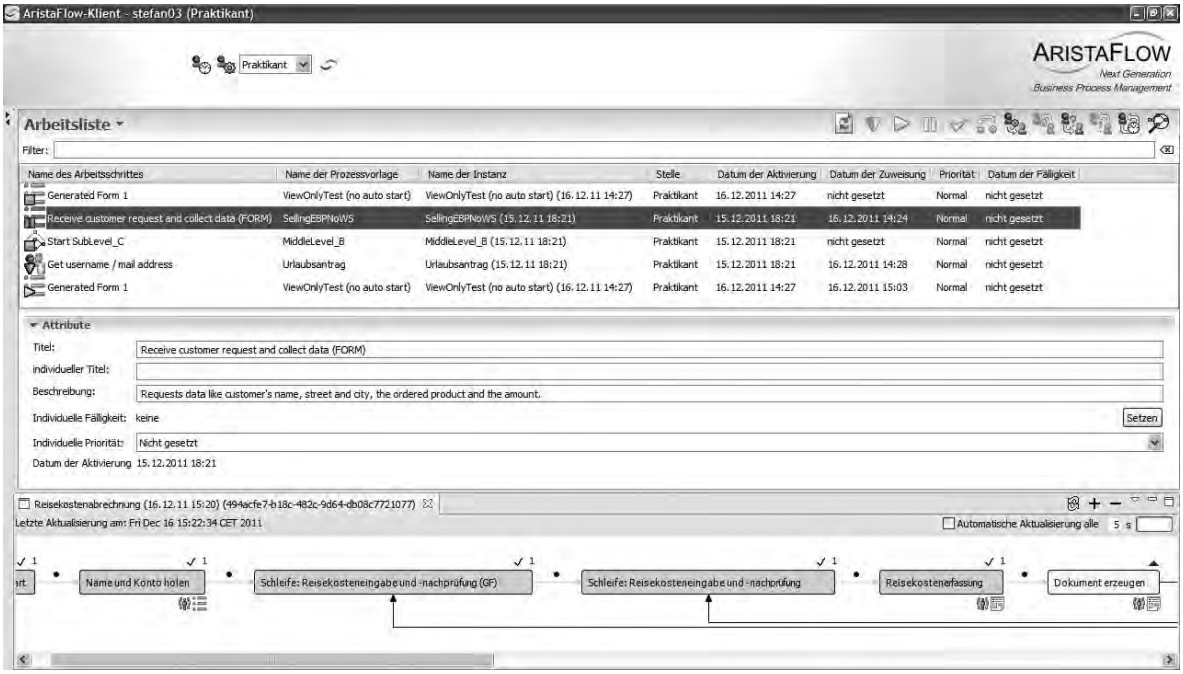


Abbildung 9.1: Prozessausführung mit Arbeitsliste und Anzeige einer laufenden Prozessinstanz

nicht mittels *Confirm* beenden, solange es noch unversorgte notwendige Ausgabeparameter gibt. Würde der Prozessschritt in diesem Zustand beendet werden, würde er fehlschlagen.

Bevor ein Prozessschritt ausgeführt werden kann, muss eine entsprechende Prozessinstanz erzeugt und gestartet werden. Dies erfolgt über eine Liste wählbarer Prozessvorlagen (s. Abb. 9.3). Die Liste enthält alle Prozessvorlagen des Systems, die zum gegenwärtigen Zeitpunkt vom aktuellen Benutzer aufgrund der von ihm aktuell eingenommenen Stelle per Doppelklick instanziert werden können.

9.2 Protokollierung

Abb. 9.4 zeigt die Monitoranwendung. Damit kann ein berechtigter Benutzer die Prozessausführung (FA 2) überwachen und bei Bedarf eingreifen, etwa indem er Prozessschritte zurücksetzt oder die Ausführung der gesamten Prozessinstanz unterbricht oder abbricht. Rechts oben sieht man den aktuellen Zustand einer Prozessinstanz, vergleichbar der Ansicht im normalen Klientenprogramm (s. Abb. 9.1). Darunter befindet sich die Ausführungshistorie. Diese protokolliert sämtliche Zustandsänderungen von Prozessschritten, den Zeitpunkt der Zustandsänderung, die Art der Änderung und den Benutzer, der die Zustandsänderung ausgelöst hat.

Abb. 9.5 zeigt eine andere Aufbereitung der Ausführungshistorie. Hier sind im Klientenprogramm nur die Elemente der Ausführungshistorie zu sehen, die den aktuell angemeldeten Benutzer betreffen. Neben der Ausführungshistorie werden alle von einem Prozessschritt gelesenen und geschriebenen Daten in einer Datenhistorie protokolliert. Bei geeigneten Anwendungsbausteinen, etwa Formularschritten, ist es mit den beiden Ausführungshistorien möglich, einen bereits ausgeführten Prozessschritt erneut zu öffnen. Dabei werden die vom Prozessschritt gelesenen und geschriebenen Daten direkt im entsprechenden Formular angezeigt. Abb. 9.6 zeigt dies für den in Abb. 9.2 normal ausgeführten Prozessschritts. Naheliegenderweise können

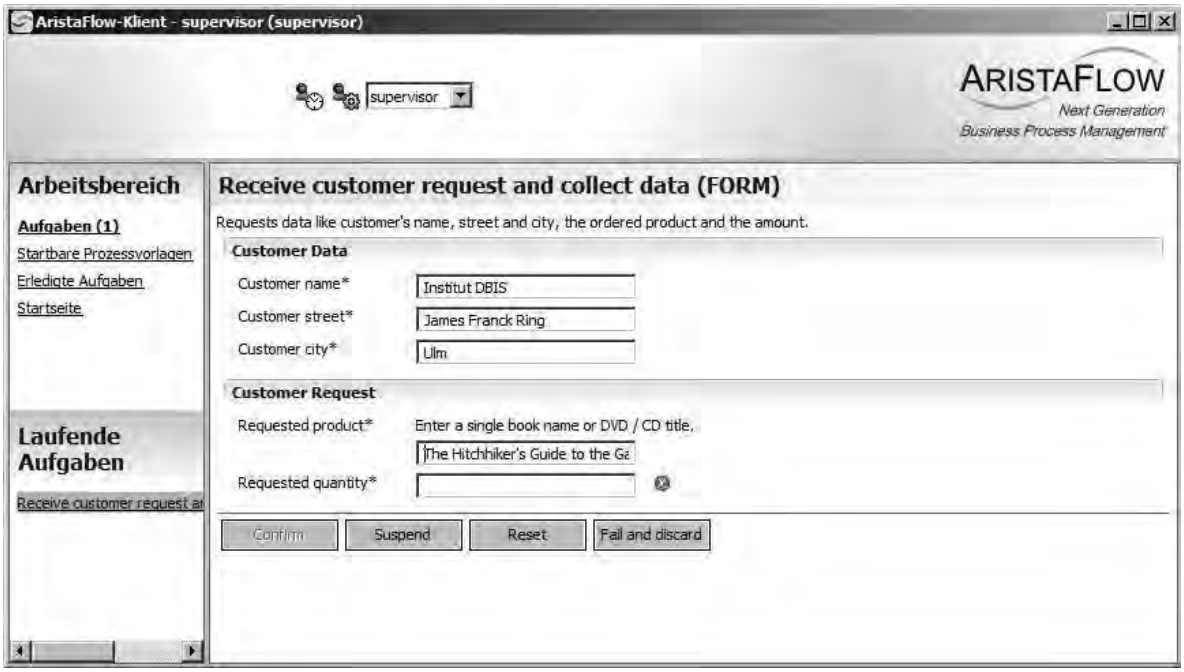


Abbildung 9.2: Ausführung eines Prozessschritts mit Formular

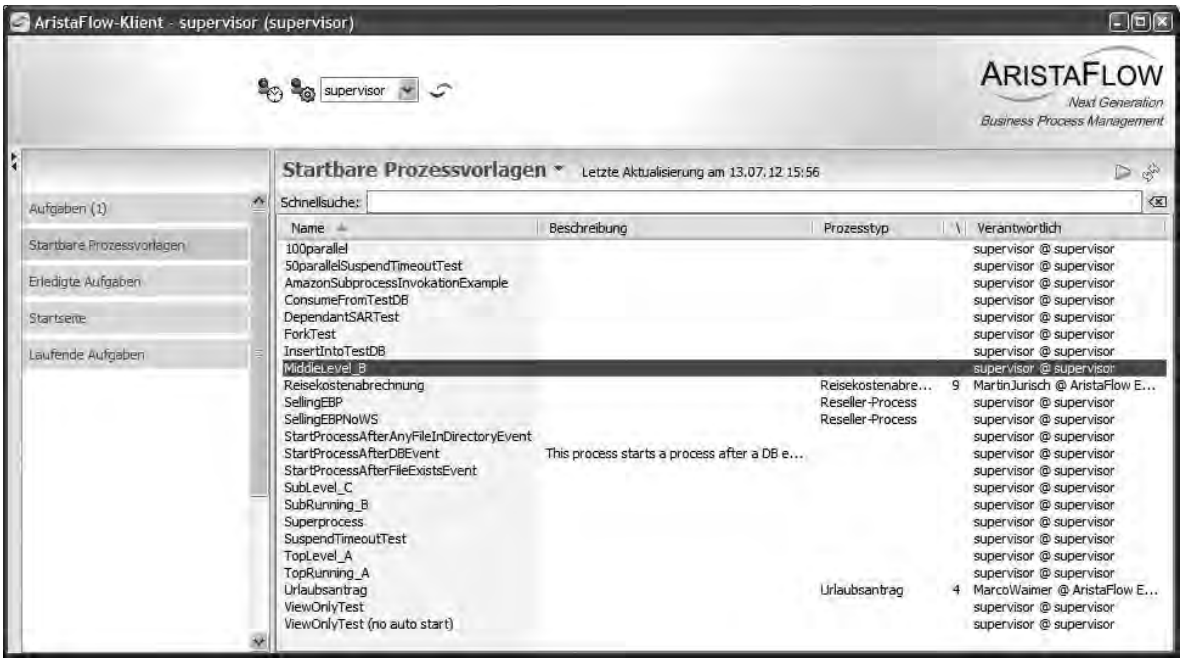


Abbildung 9.3: Liste von startbaren Prozessvorlagen

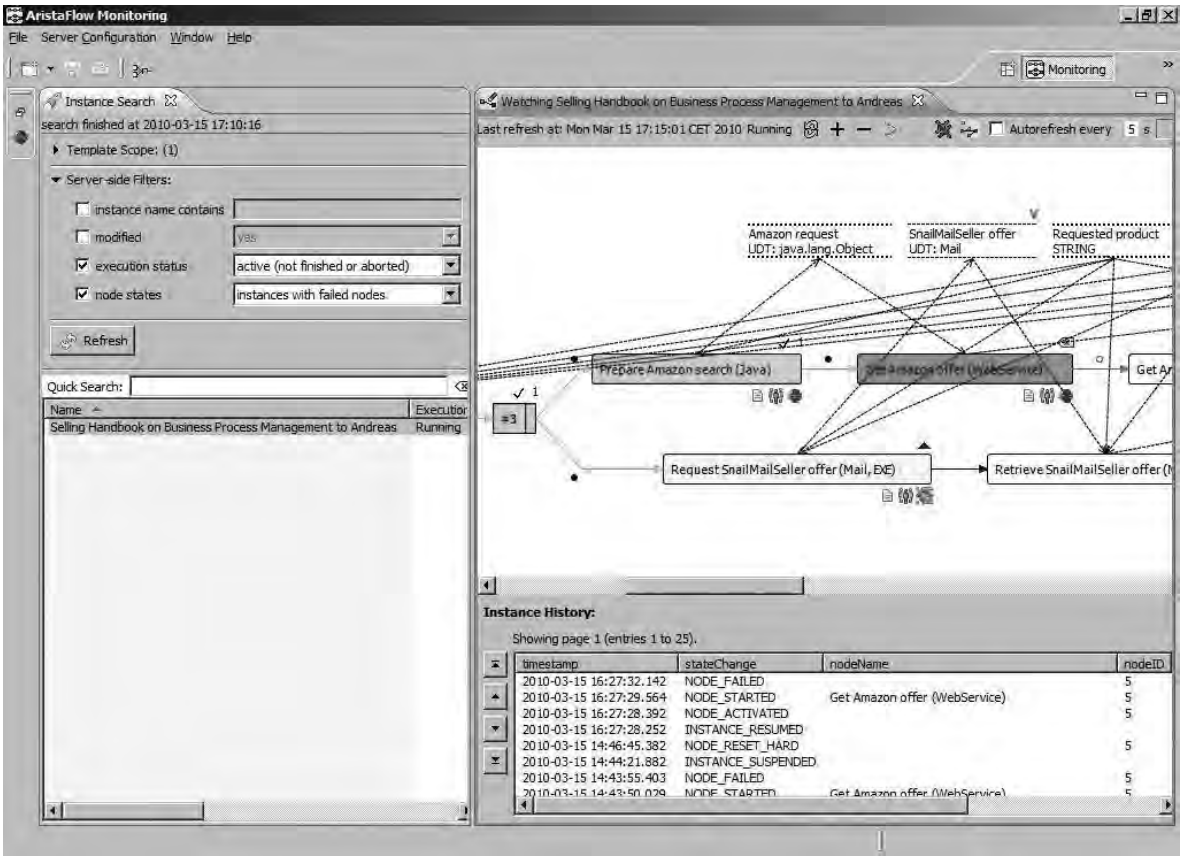


Abbildung 9.4: Monitoranwendung mit Prozessinstanz und deren Ausführungshistorie

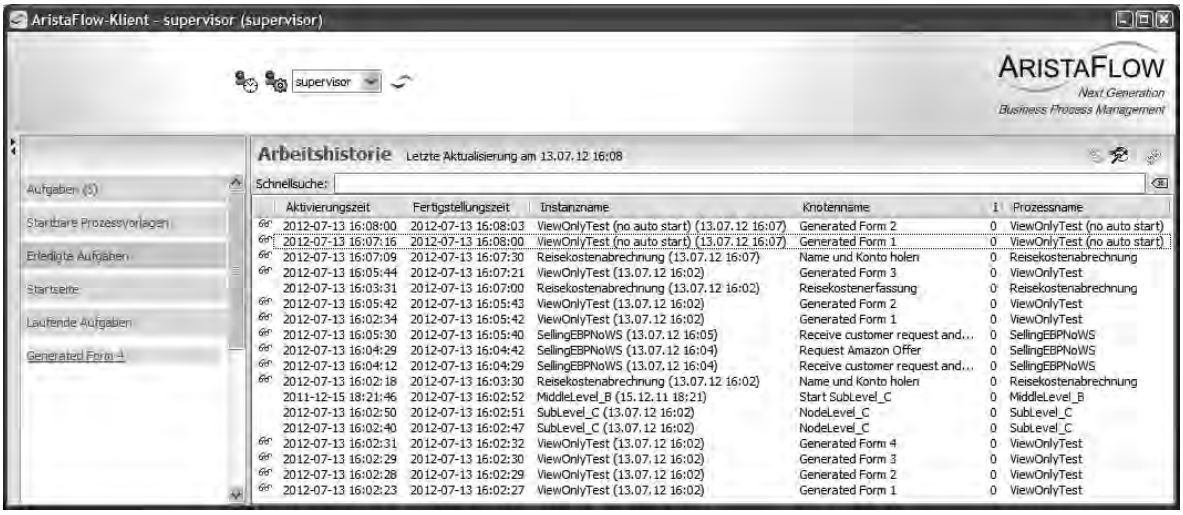


Abbildung 9.5: Historie aller ausgeführten Prozessschritte eines Bearbeiters

beim Öffnen eines abgeschlossenen Prozessschritts keine Daten geändert werden. Außerdem kann der Prozessschritt nur geschlossen werden (*Close* in der Abbildung). Andere Interaktionsmöglichkeiten gibt es nicht, da der Prozessschritt selbst bereits abgeschlossen und damit in einem Endzustand ist.

9.3 Organisationsmodell

Den Editor für die Erstellung und Verwaltung des Organisationsmodells (FA 3, *OrgModelManager*, s. Abschnitt 7.3.3) zeigt Abb. 9.7. Auf der linken Seite sind die verschiedenen Entitätstypen des Organisationsmodells abgebildet. Hier können Attribute für Entitätstypen hinzugefügt, geändert und gelöscht werden. An dieser Stelle wird auch hinterlegt, welche Attribute von einem vorhandenen LDAP-Verzeichnis übernommen werden. Im mittleren Bereich sind alle Entitäten und deren Attributwerte aufgelistet. Auf der rechten Seite kann man oben die Attribute der aktuell ausgewählten Entität bearbeiten. Darunter werden die Beziehungen zu anderen Entitäten festgelegt, etwa welche Stellen (*OrgPosition*) ein Mitarbeiter (*Agent*) innerhalb des Organisationsmodells hat.

Ein erstelltes Organisationsmodell wird für die Zuordnung von Prozessschritten zu den Bearbeitern (genauer deren Arbeitslisten) verwendet. Dazu wird an jedem Prozessschritt eine entsprechende Bearbeiterzuordnungsregel hinterlegt. Abb. 9.8 zeigt die Erstellung einer solchen Regel. Im gezeigten unterstützen Modus (*Assisted Policy Editing*) kann man nur einzelne Entitäten verschiedenen Typs auswählen. Dies dient der einfachen Erstellung von Bearbeiterzuordnungsregeln. Über den manuellen Modus (*Manual Policy Editing*) können alle Freiheiten des Modells genutzt werden. So lassen sich z. B. sämtliche Attribute eines Entitätstyps sowie alle vorhandenen Entitätstypen in Regeln verwenden.

9.4 Arbeitslistenverwaltung

Die Arbeitslistenverwaltung (FA 4, *WorklistManager*, s. Abschnitt 7.3.7) bereitet die aktuell ausführbaren und ausgeführten Prozessschritte in Form von Arbeitslisten auf. Dies bietet

Abbildung 9.6: Bereits abgeschlossener Formularschritt, der zur nachträglichen Betrachtung „nur lesend“ geöffnet ist

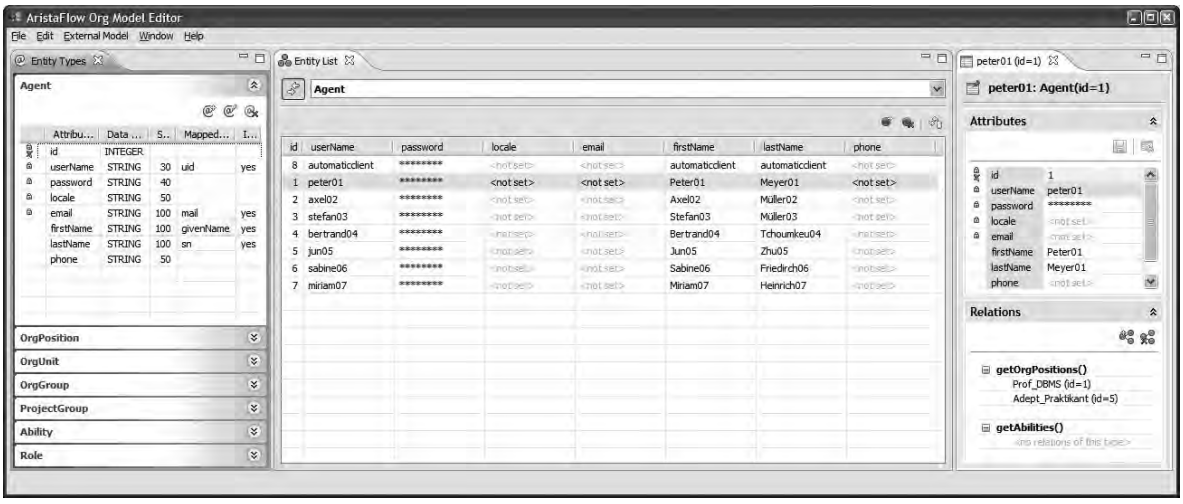


Abbildung 9.7: Editor für die Erstellung und Verwaltung des Organisationsmodells

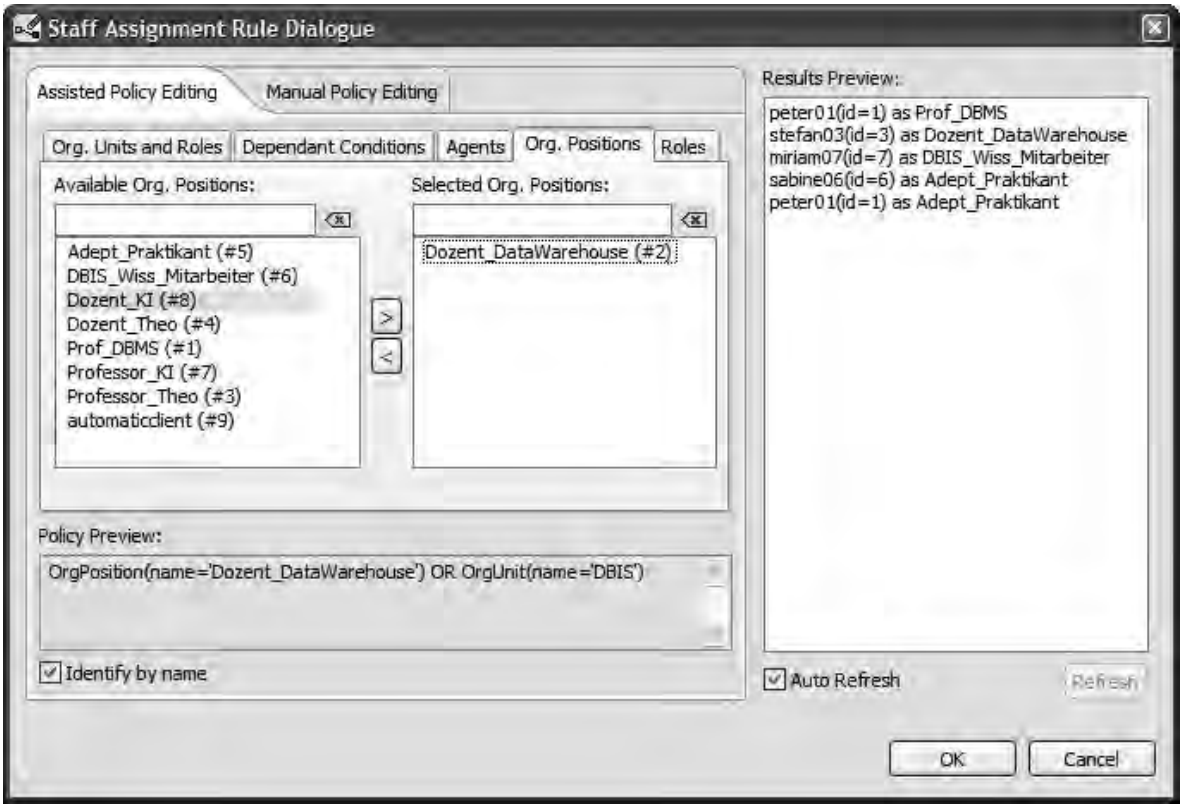


Abbildung 9.8: Dialog zur Erstellung von Bearbeiterregeln für die Hinterlegung an Prozessschritten

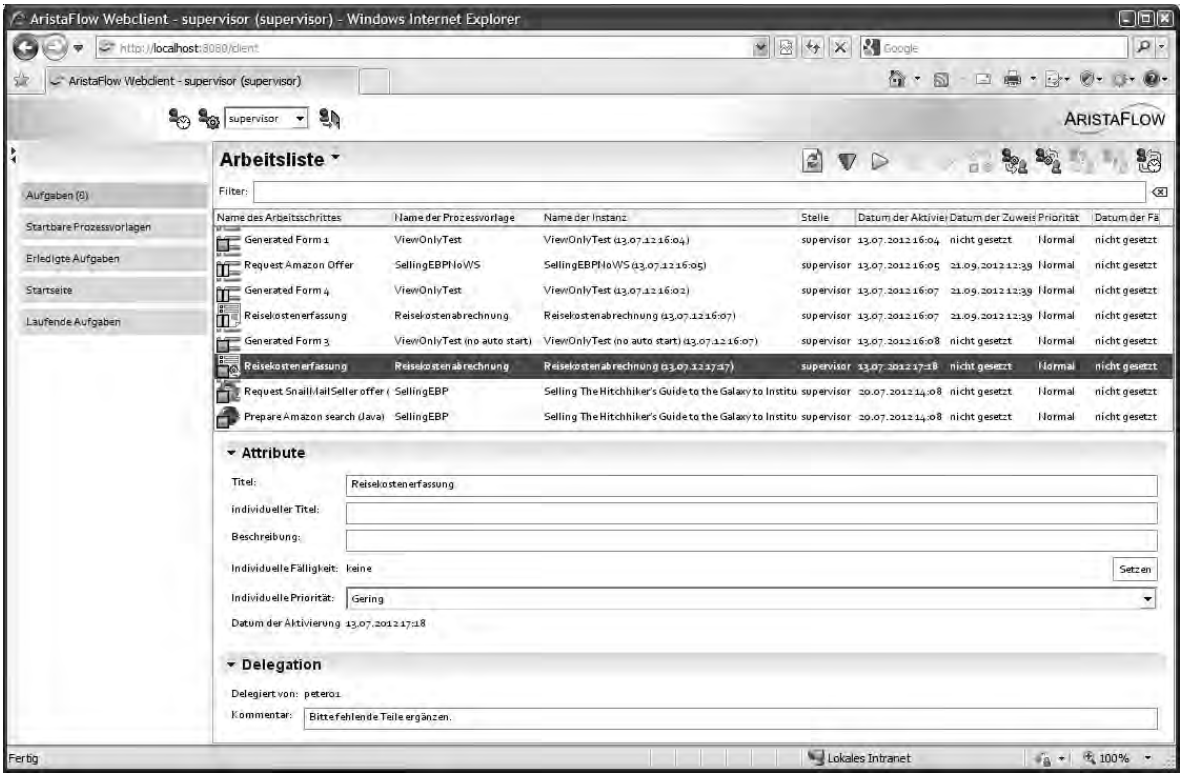


Abbildung 9.9: Arbeitsliste mit delegiertem Prozessschritt im Klientenprogramm für das Web

Bearbeitern einen guten Überblick über die jeweils anstehenden Aufgaben. Abb. 9.1 zeigt eine Arbeitsliste in einem Klientenprogramm mit zusätzlichen Informationen zum ausgewählten Arbeitslisteneintrag. Abb. 9.9 zeigt eine vergleichbare Arbeitsliste in einem Klientenprogramm für das Web. Dieses beruht auf denselben Oberflächen und weitgehend auf derselben Implementierung wie das Klientenprogramm aus Abb. 9.1.

Über die Arbeitslistenverwaltung können Arbeitslisteneinträge delegiert werden. Das heißt, sie werden einem anderen Bearbeiter zugewiesen, etwa weil dem aktuellen Bearbeiter notwendige Informationen fehlen. Delegationen können mit einem Kommentar versehen werden. In Abb. 9.9 ist dies unterhalb der Arbeitsliste für den aktuell ausgewählten Arbeitslisteneintrag zu sehen.

9.5 Prozessüberwachung

Neben der Prozessüberwachung (FA 5) durch einen Benutzer, etwa über die Monitoranwendung (s. Abb. 9.4), kann die Prozessausführung systemseitig durch Eskalationen überwacht werden. Abb. 9.10 zeigt die Definition einer Eskalation im Prozessvorlageneditor für den Prozessschritt *Reisekosten buchen oder ablehnen (GF)*. Eskalationen werden zu einem bestimmten absoluten Zeitpunkt oder nach Ablauf einer Zeitspanne relativ zu einem Ereignis während der Prozessausführung ausgelöst, etwa der Aktivierung des entsprechenden Prozessschritts. Tritt die Eskalation ein, kann sie beispielsweise die Priorität eines Prozessschritts erhöhen, bestimmte Personen per E-Mail benachrichtigen oder eine neue Prozessinstanz starten. Es können beliebig viele Eskalationen unabhängig voneinander hinterlegt werden. In Abb. 9.10 sieht man im

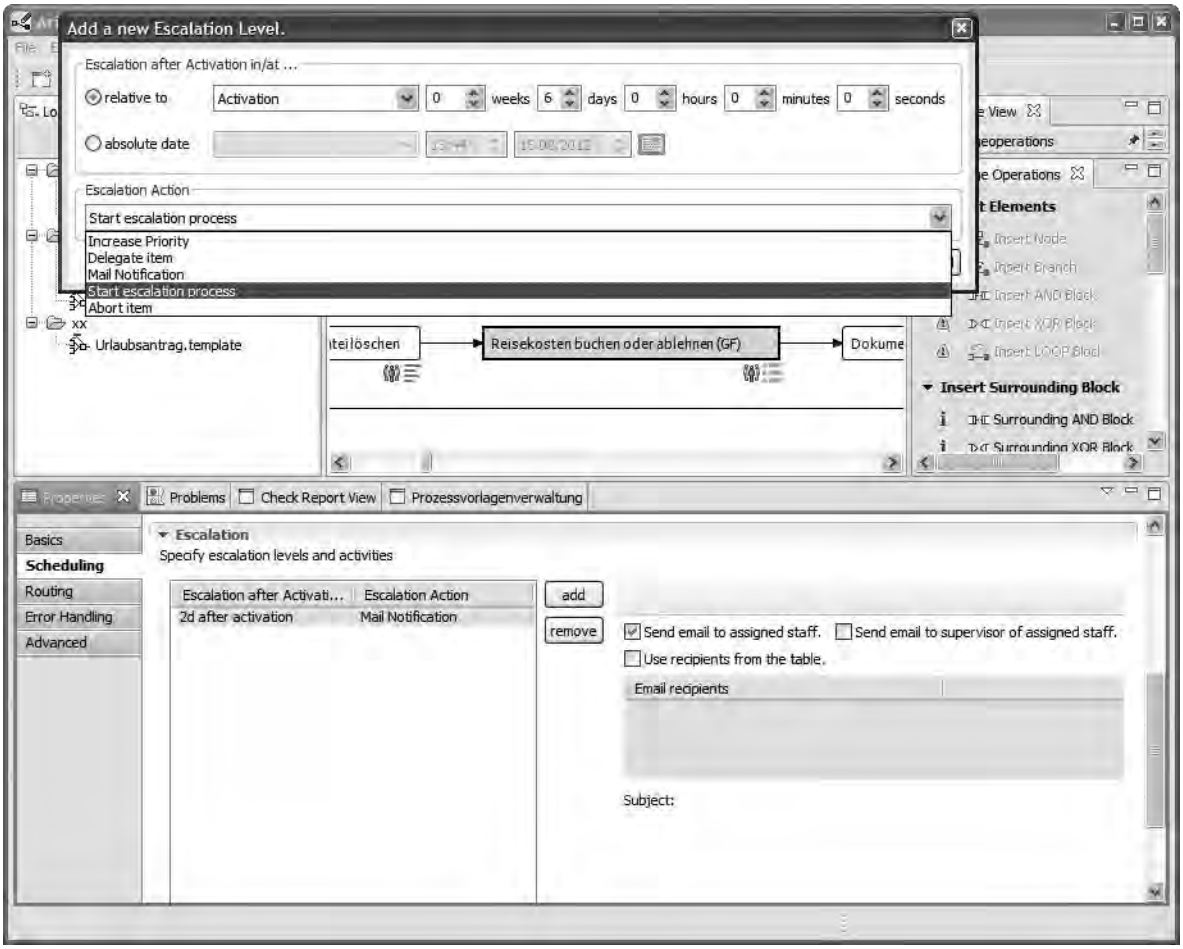


Abbildung 9.10: Definition von Eskalation an einem Prozessschritt im Prozessvorlageneditor

unteren Teil des Bildschirmfotos beispielsweise eine bereits existierende Eskalation. Diese benachrichtigt die zugeordneten Bearbeiter zwei Tage nach Aktivierung des Prozessschritts per E-Mail.

Die Prozessüberwachung anhand von Leistungskennzahlen (KPI) erfolgt sowohl auf den Basis der Historiendaten als auch auf der Ausführungshistorie (s. Abb. 9.4). Dabei können die Leistungskennzahlen regelmäßig oder bei Änderungen an den Historiendaten neu berechnet und anschließend ggf. entsprechende Maßnahmen getroffen werden.

9.6 Anwendungsintegration und -aufruf

Die Architektur der Laufzeitumgebung (*RuntimeManager*, s. Abschnitt 7.3.8) ermöglicht die Integration und Ausführung weitgehend beliebiger Anwendungsbausteine (FA 7). Abb. 9.11 zeigt die Ausführung eines Anwendungsbausteins für HTML-Formulare. In diesem Fall wird das Formular in einem Klientenprogramm für das Web ausgeführt, das selbst auf HTML beruht. Das Formular überprüft Eingaben auf syntaktische Korrektheit und Plausibilität, etwa das richtige Zahlenformat. Ferner stellt es sicher, dass Anfangszeiten nicht nach Endzeiten liegen. Weiter können dynamisch Daten und Eingabefelder aktiviert/deaktiviert oder angezeigt/

ausgeblendet werden. Durch Trennung der graphischen Oberflächen vom Klientenprogramm können dieser Anwendungsbaustein und das erzeugte HTML-Formular auch in einem anderen Klientenprogramm ausgeführt werden, etwa dem SWT-Klientenprogramm aus Abb. 9.2. Dieses SWT-Klientenprogramm führt einen in Abb. 9.2 einen Anwendungsbaustein aus, der automatisch ein SWT-Formular generiert, das die Ein-/Ausgabeparameter des entsprechend Prozessschritts anzeigt und manipuliert.

Der Anwendungsbaustein in Abb. 7.41 ruft *OpenOffice.org* über die entsprechende API (*Universal Network Objects* [uno]) auf. In diesem Fall ermöglicht die Trennung der graphischen Oberfläche vom Klientenprogramm die direkte Integration der AWT-Oberfläche von *OpenOffice.org* in das ausführende SWT-Klientenprogramm. Die Ausführung eines speziell für den Einsatz in PrMS implementierten Anwendungsbaustein zeigt Abb. 9.12. Der Anwendungsbaustein dient zur Auswahl von Ressourcen für die Einsatzplanung beim Katastrophenschutz [WR09].

9.7 Prozessmodellierung

Die Prozessmodellierung (FA 8) mit dem Prozessvorlageneditor zeigt Abb. 9.13. Die Bildschirmaufteilung ist wie folgt: In der Mitte ist die modellierte Prozessvorlage zu sehen, rechts unten befindet sich eine Übersicht. Links unten werden Details zum aktuell markierten Objekt angezeigt, in diesem Fall Details zum Prozessschritt *Fill out Order Form*. Die Details können an dieser Stelle auch bearbeitet werden. Am linken Bildschirmrand befinden sich alle Objekte des Aktivitätenrepository, u. a. Aktivitätenvorlagen wie *XOR Predicate*. Auf der rechten Seite sieht man oben Werkzeuge für die Auswahl von Objekten der Prozessvorlage und darunter Änderungsoperationen für die Prozessmodellierung. Dabei sind nur die Änderungsoperationen aktiv, die auf die momentane Objektmarkierung anwendbar sind. In der Abbildung sind lediglich das Verschieben von Knoten und das Hinzufügen eines Datenelements aktiviert. Für deaktivierte Änderungsoperationen erhält der Benutzer über das gelbe Ausrufezeichen eine Kurzinformation zum Grund der Deaktivierung.

Über die Objekdetails links unten in Abb. 9.13 können Bearbeiterzuordnungen (*Staff Assignment*) an Knoten hinterlegt und bearbeitet werden. Beim Drücken der entsprechenden Schaltfläche (*Edit*) öffnet sich der Dialog in Abb. 9.14. Diese Bearbeiterzuordnungen werden zur Laufzeit auf Grundlage des Organisationsmodells (s. Abschnitt 9.3) aufgelöst. Im gezeigten Dialog können Bearbeiterzuordnungen sowohl manuell (*Manual Policy Editing*) als auch assistiert (*Assisted Policy Editing*) erstellt und bearbeitet werden. In diesem Fall handelt es sich um eine abhängige Bearbeiterzuordnung (s. Abschnitt 2.2.3.5), bei der zur Laufzeit der Bearbeiter zugeordnet wird, der die entsprechende Prozessinstanz gestartet hat (*InstanceInitiator*). Auf der rechten Seite des Dialogs werden, wenn gewünscht, alle Bearbeiter aufgeführt, die sich momentan im Organisationsmodell ergeben. Bei einer abhängigen Bearbeiterzuordnung wie in diesem Fall funktioniert dies nicht, da es zum Modellierungszeitpunkt keine Prozessinstanz und damit keinen startenden Bearbeiter gibt, weshalb die Abhängigkeit nicht aufgelöst werden kann.

Abb. 9.15 zeigt eine fortgeschrittene Prozessmodellierung. Gegenüber Abb. 9.13 wurde die Detailanzeige von der Problemanzeige und die Prozessvorlagenübersicht rechts unten durch eine Liste aller Datenelemente mit entsprechenden Zusatzinformationen ersetzt. Wie in der zentralen Prozessvorlagenansicht und der Problemanzeige ersichtlich, gibt es ein Problem mit dem Datenfluss (s. Abschnitt 2.2.2.2). Der Prozessschritt *Write Customer Offer (Java)* liest

Abbildung 9.11: HTML-Formular zur Reisekostenabrechnung im Klientenprogramm für das Web

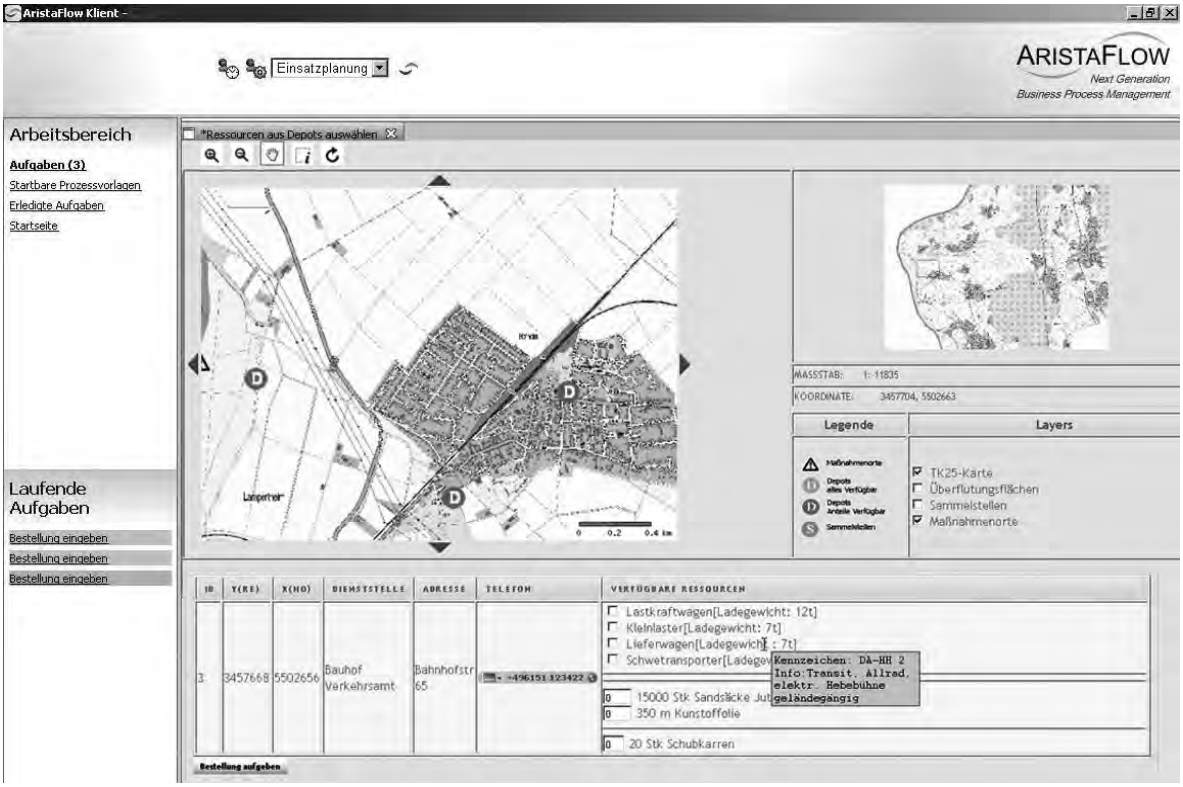


Abbildung 9.12: Ausführung einer Anwendung aus dem Katastrophenschutz

das Datenelement *Customer Price per Unit* obligat. Zu diesem Datenelement führt aber keine schreibende Kante, so dass es bei der Prozessausführung beim des Prozessschritts keinen Wert haben kann.

Eine frühzeitige Rückmeldung von an der Prozessausführung beteiligten Personen bereits während der Modellierungsphase ist hilfreich für die Prozesserstellung. Hierbei hilft die testweise Ausführung von in der Erstellung befindlichen Prozessvorlagen im *TestClient* (s. Abb. 9.16). Dieser entspricht von der Benutzeroberfläche weitgehend einem normalen Klientenprogramm (s. Abb. 9.1 und Abb. 9.2). Wie im Bildschirmfoto ersichtlich, besitzt er eine Arbeitsliste (oben), einen Bereich für die Oberfläche des ausgeführten Anwendungsbausteins (in der Mitte), in diesem Fall eines automatisch generierten Formulars, und eine Übersicht über die ausgeführte Prozessinstanz (unten).

Neben der Oberfläche besitzt der *TestClient* alle Dienste des Systems und ist somit ein komplettes PrMS mit eigener Oberfläche. Es sind lediglich einige Dienste gegenüber dem produktiven Einsatz geringfügig verändert. So werden die Daten nicht dauerhaft gespeichert. Außerdem wird das Organisationsmodell im Normalfall nicht verwendet, damit alle Prozessschritte für den Test in derselben Arbeitsliste erscheinen. Dies erspart das Wechseln von Benutzern in der Benutzeroberfläche und beschleunigt so die testweise Prozessausführung. Es kann jedoch auch das normale Organisationsmodell verwendet werden (durch Austausch des Dienstes *OrgModelManager*). So können im *TestClient* die in der Prozessvorlage hinterlegten Bearbeiterzuordnungen überprüft werden. Ähnliches gilt auch für die Ausführung von automatischen Schritten. Im Normalfall werden diese im *TestClient* manuell ausgeführt. Dadurch

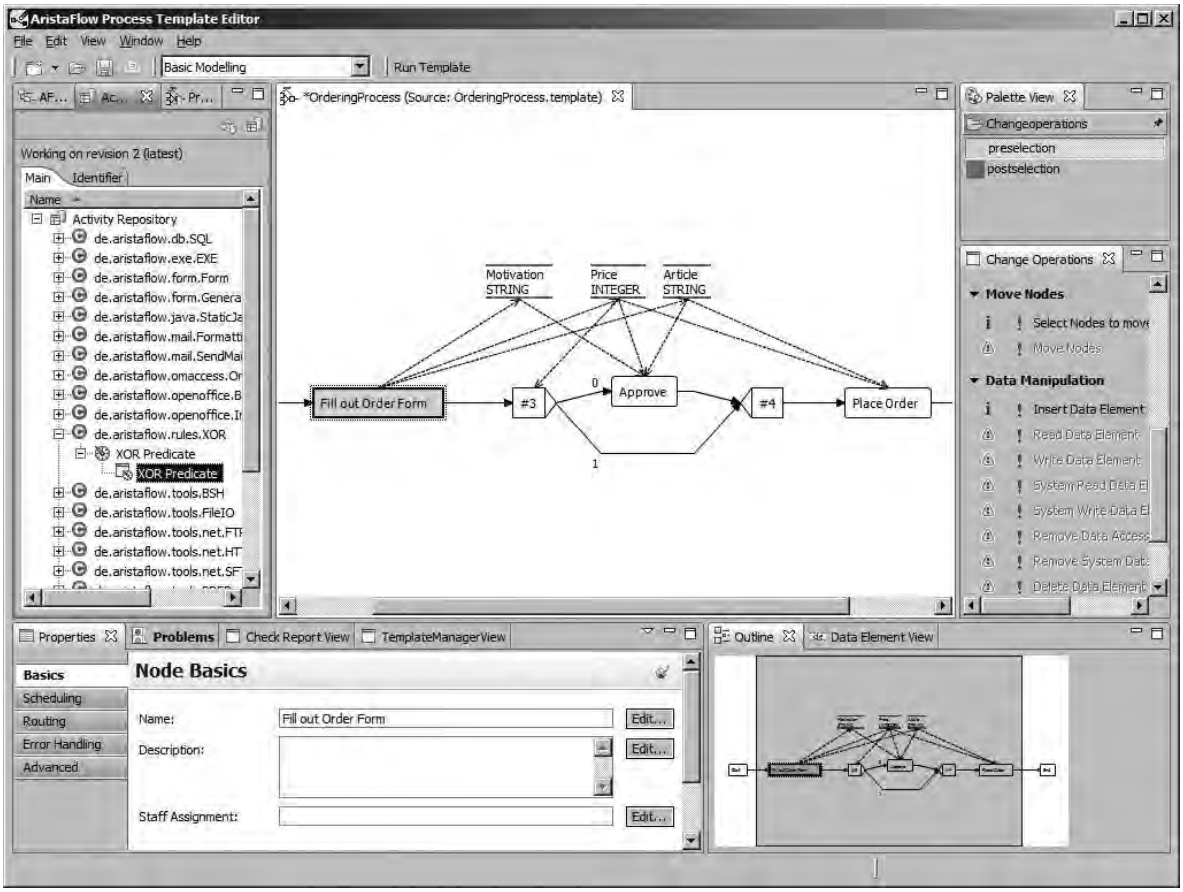


Abbildung 9.13: Hinterlegung einer Aktivitätenvorlage bei der Erstellung einer Prozessvorlage

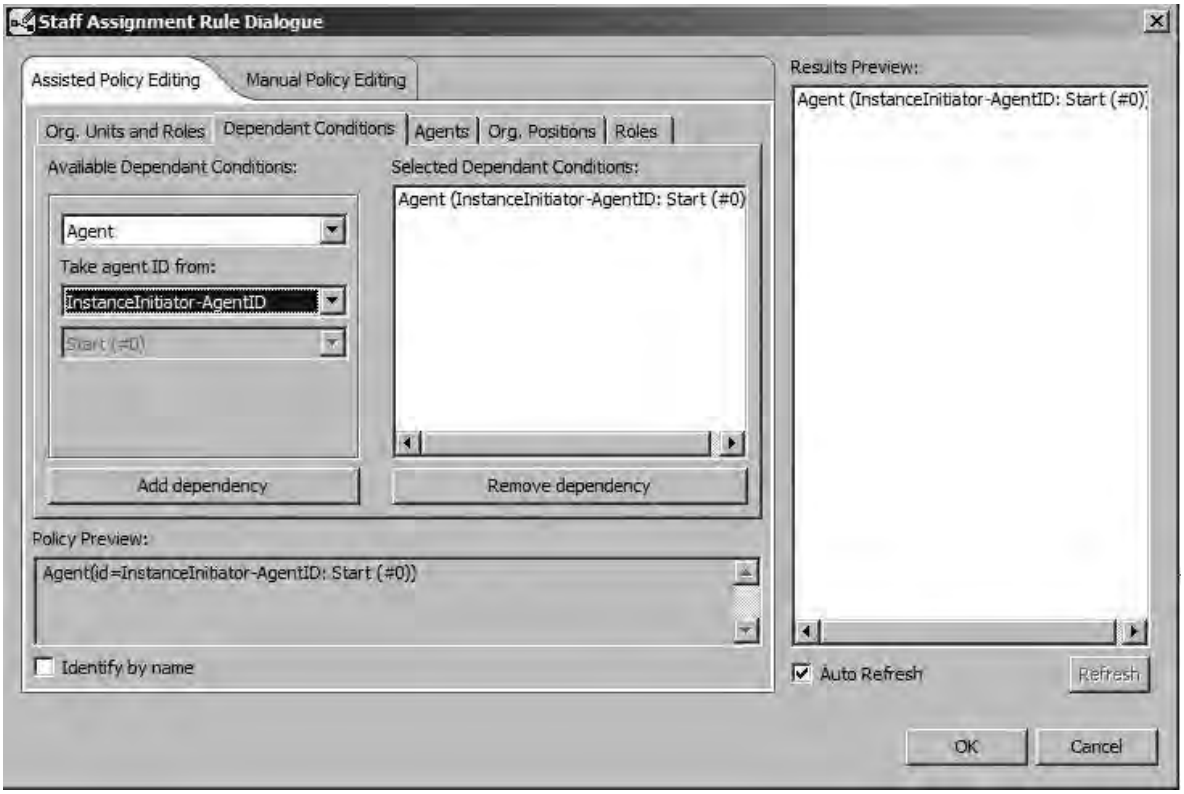


Abbildung 9.14: Unterstützte Erstellung einer Bearbeiterzuordnung

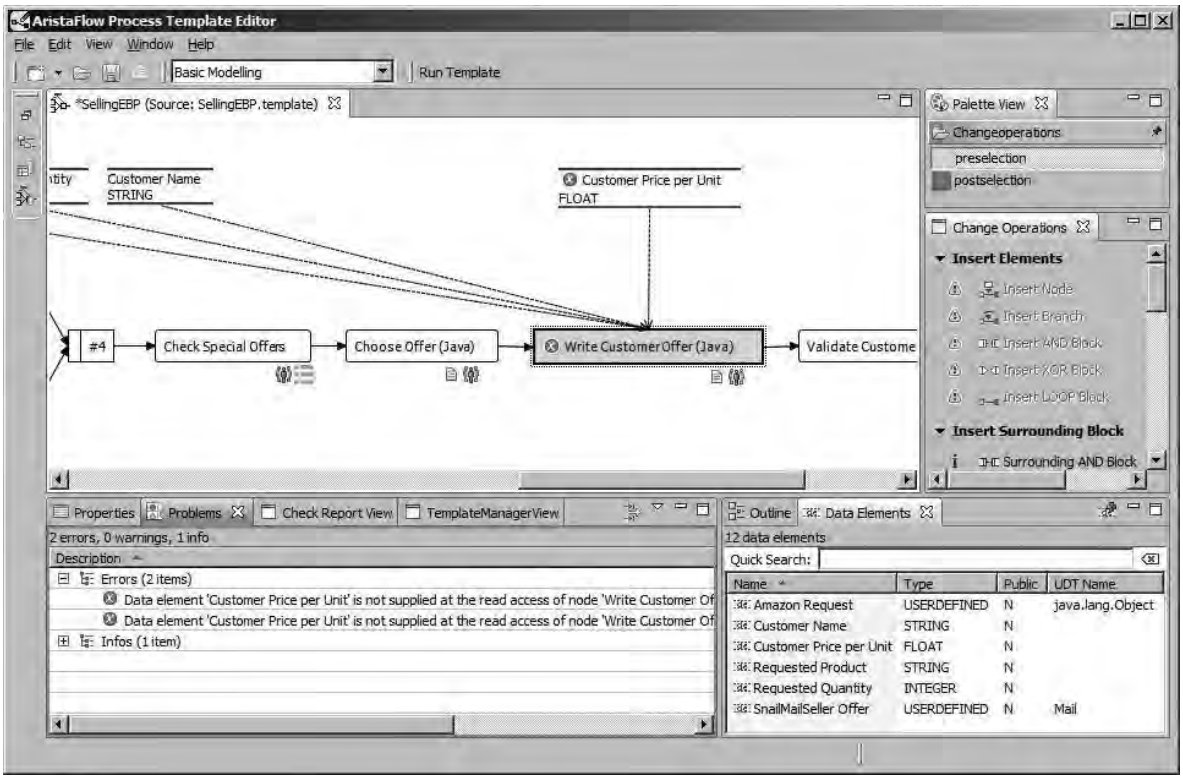


Abbildung 9.15: Unzulässiger lesender Zugriff auf ein noch nicht geschriebenes Datenelement bei der Erstellung einer Prozessvorlage

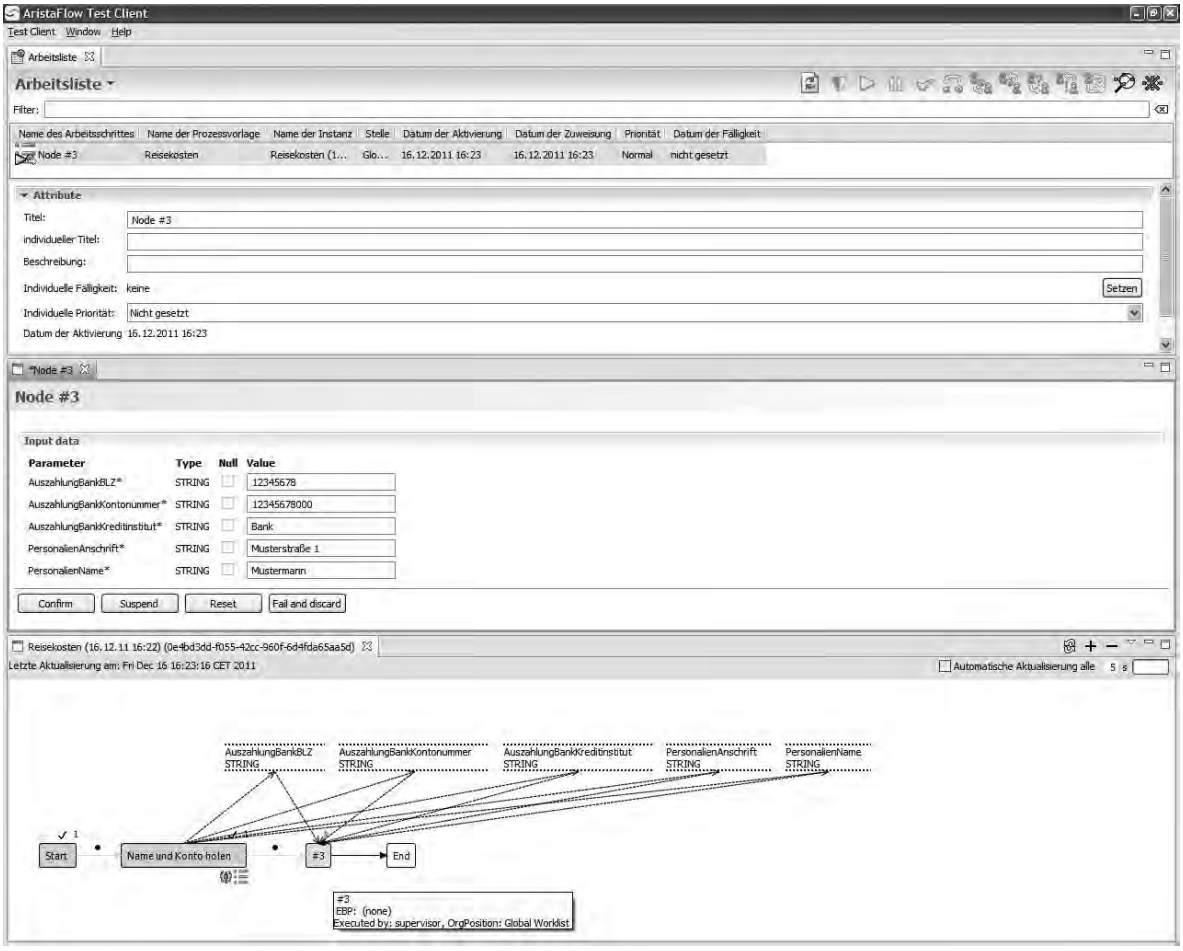


Abbildung 9.16: Testweise Ausführung einer unvollständig modellierten Prozessvorlage

sind Verzweigungsentscheidungen beim Testen besser nachvollziehbar. Sie können jedoch auch im *TestClient* automatisch ausgeführt werden.

Der wesentliche Unterschied der Ausführung im *TestClient* im Vergleich zur produktiven Ausführung liegt darin, dass Prozessschritte nicht vollständig modelliert sein müssen. So kann aufgrund des fehlenden Organisationsmodells auf Bearbeiterzuordnungen verzichtet werden. Es müssen nicht einmal Aktivitäten an Prozessschritten hinterlegt sein. Für solch „leere“ Prozessschritte erzeugt der *TestClient* automatisch generierte Formulare. Ein Beispiel zeigt Abb. 9.2, wo eine solche Formularaktivität explizit hinterlegt worden ist. Die Formulare zeigen sämtliche Ein-/Ausgabeparameter des leeren Knotens und bieten zudem Eingabemöglichkeiten für alle Ausgabeparameter. Zusätzlich wird gewährleistet, dass alle erforderlichen Ausgabeparameter mit Werten versehen sind, wenn das Formular geschlossen und der entsprechende Prozessschritt damit beendet wird. Abb. 9.16 zeigt das für den leeren Prozessschritt *#3* generierte Formular. In der Prozessansicht unten wird deutlich, dass an diesem Prozessschritt momentan keine Aktivität hinterlegt ist (*EBP: (none)*).

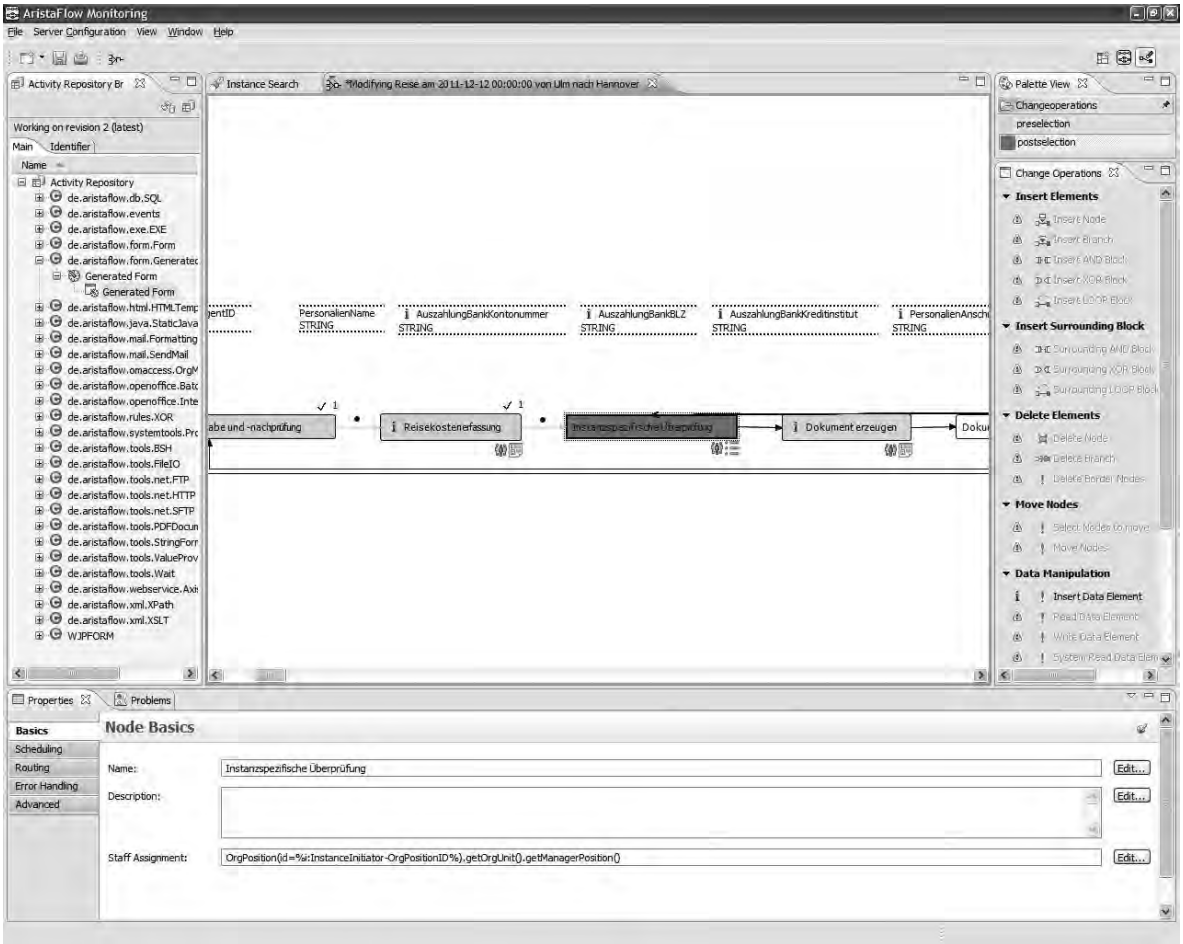


Abbildung 9.17: Instanzspezifische Änderung (Einfügen eines Prozessschritts)

9.8 Instanzspezifische Änderungen

Abb. 9.17 zeigt ein dem Prozessvorlageneditor (s. Abb. 9.13) ähnliches Bildschirmfoto. Hier handelt es sich allerdings um die Monitoranwendung, in der eine Prozessinstanz geändert wird (FA 9.1). Dies ist an den entsprechend ihres aktuellen Zustands farblich markierten Prozessschritten zu erkennen. So ist etwa der Prozessschritt *Reisekostenerfassung* abgeschlossen. Im Rahmen der Änderung wurde der neue Prozessschritt *Instanzspezifische Überprüfung* zwischen *Reisekostenerfassung* und *Dokument erzeugen* eingefügt. *Dokument erzeugen* war vor der Änderung aktiviert, aufgrund der Änderung wird die Aktivierung zurückgenommen. Die Neubewertung des Zustands der Prozessinstanz erfolgt am Ende der Änderungstransaktion. Der neu eingefügte Prozessschritt besteht aus einem automatisch generierten Formular (*Generated Form*, links im Aktivitätenrepository) und wird dem Vorgesetzten desjenigen Bearbeiters zugewiesen, der die Prozessinstanz gestartet hat (Bearbeiterzuordnungsregel in den Objektdetails unten). Beide Eigenschaften werden im Rahmen der instanzspezifischen Änderung genauso festgelegt wie bei der Prozessmodellierung (s. Abschnitt 9.7).

Die Fortsetzung der Prozessausführung nach der vorgenommenen instanzspezifischen Änderung zeigt Abb. 9.18. Im oberen Teil sieht man das generierte Formular, das an dem neuen

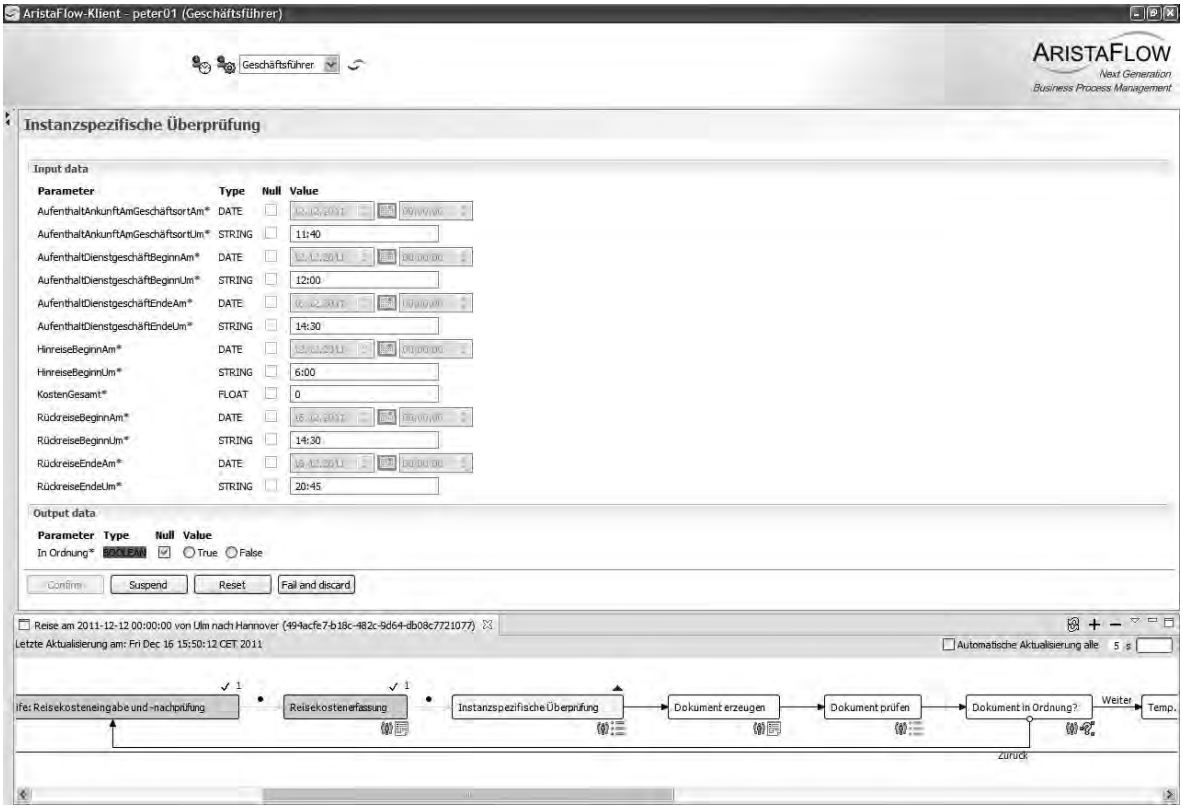


Abbildung 9.18: Ausführung des instanzspezifisch eingefügten Prozessschritts

Prozessschritt hinterlegt worden ist. Der untere Teil zeigt den Zustand der Prozessinstanz direkt nach Abschluss der instanzspezifischen Änderung mit dem neu eingefügten Prozessschritt.

9.9 *Plug and Play*

Die funktionale Anforderung *Plug and Play* (FA 10) wird zu einem Teil durch das Aktivitätenrepository (*ActivityRepository*, s. Abschnitt 7.3.1) umgesetzt. Abb. 9.19 zeigt den Aktivitätenvorlageneditor zum Verwalten von Aktivitätenvorlagen gemäß dem in Abschnitt 7.2.2 vorgestellten Aktivitätenmodell. Auf der linken Seite sieht man alle im Aktivitätenrepository vorhandenen Objekte als Baum. Derselbe Baum wird auch im Prozessvorlageneditor genutzt, um Aktivitätenvorlagen mittels Ziehen mit der Maus (*Drag and Drop*) mit Prozessschritten zu verknüpfen (s. Abb. 9.22 und Abb. 9.13). Auf der rechten Seite des Aktivitätenvorlageneditors werden die Attribute des aktuellen Objekts angezeigt; in diesem Fall ist es die Aktivitätenvorlage *Send Mail*. Diese repräsentiert einen Anwendungsbaustein zum automatischen Versand einer E-Mail innerhalb eines Prozesses. In den Attributen der Aktivitätenvorlage findet man unter anderem die übliche Bearbeiterzuordnung (*Staff Assignment Rule*) für den automatischen Klienten, der den Prozessschritt ohne Benutzereingriff ausführt. Aufgrund der Versionierung des Aktivitätenrepository können zum gezeigten Zeitpunkt keine Änderungen vorgenommen werden.

Die Ein-/Ausgabeparameter der Aktivitätenvorlage *Send Mail* zeigt Abb. 9.20. Sie bestimmen, welche Daten der Anwendungsbaustein zur Laufzeit aus dem Prozess erhält. In diesem

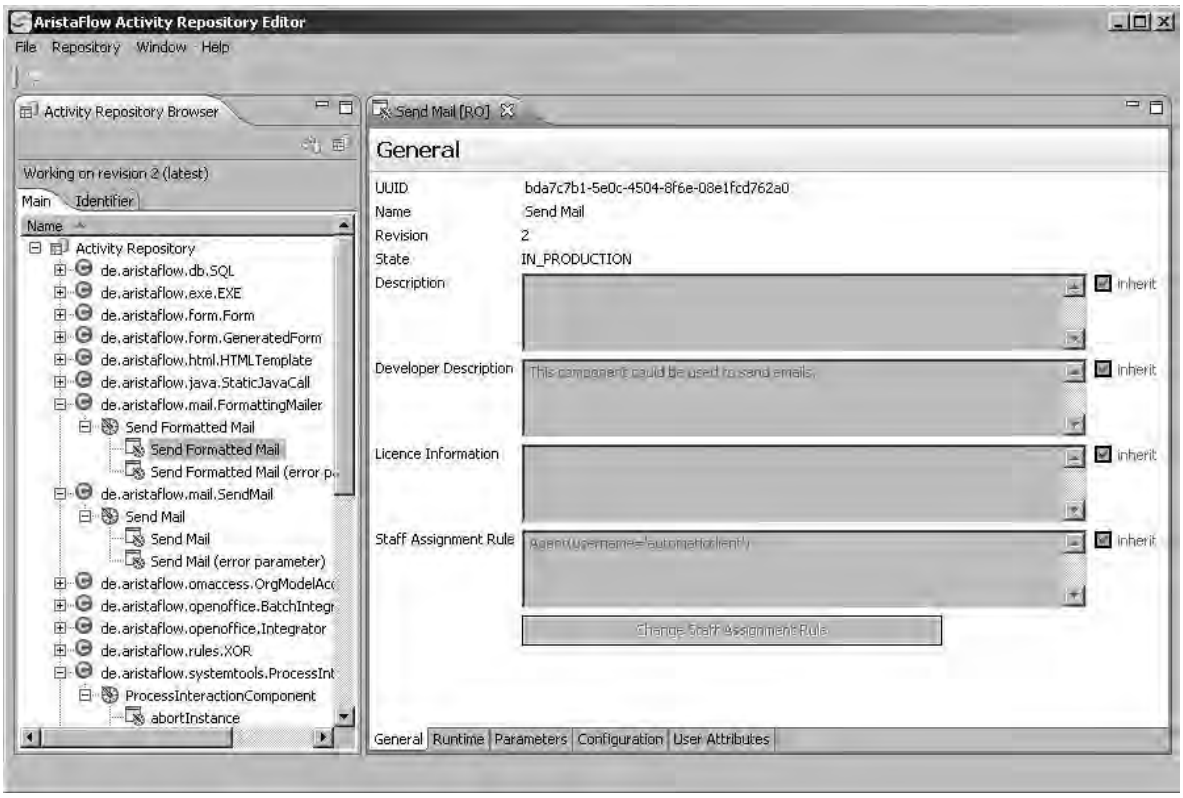


Abbildung 9.19: Aktivitätenvorlage im Aktivitätenrepository

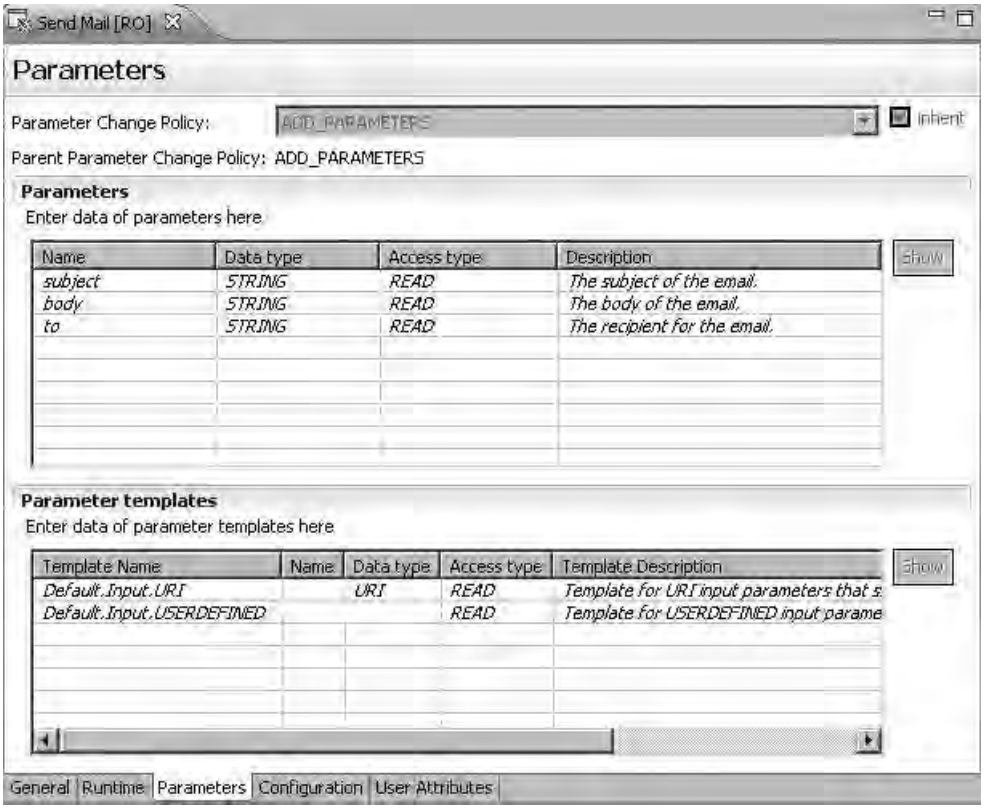


Abbildung 9.20: Ein-/Ausgabeparameter einer Aktivitätenvorlage im Aktivitätenrepository

Fall sind dies der Betreff (*subject*), der Inhalt (*body*) sowie der oder die Empfänger (*to*) der zu versendenden E-Mail. Der Anwendungsbaustein erlaubt es dem Prozessmodellierer, zusätzliche Eingabeparameter hinzuzufügen (*ADD_PARAMETERS*). Es handelt sich also um einen generischen Anwendungsbaustein, der bei seiner Ausführung die tatsächlich vorhandenen Parameter überprüft und zusätzliche, durch den Prozessmodellierer definierte Parameter entsprechend verarbeitet. Da die zusätzlichen Parameter im allgemeinen Einschränkungen aufweisen können, etwa bezüglich des Datentyps, kann man im Aktivitätenrepository entsprechende Parametervorlagen (*Parameter templates*, im unteren Teil von Abb. 9.20) hinterlegen. Diese erleichtern dem Prozessmodellierer den Umgang mit generischen Anwendungsbausteinen. Im Fall des E-Mail-Versands gibt es die Möglichkeit, zusätzliche Eingabeparameter für Dateianhänge oder Referenzen auf Dateianhänge aus dem Prozess zu übergeben. Diese Anhänge werden dann mit der E-Mail versandt.

Neben den Daten aus den Eingabeparametern besitzen Anwendungsbausteine noch Konfigurationswerte. Zu *Send Mail* gehören unter anderem Einstellungen des zu nutzenden SMTP-Server (*Simple Mail Transfer Protocol*) wie Serveradresse und Benutzername. Die jeweils erforderliche Konfiguration wird für jede Aktivitätenvorlage im Aktivitätenrepository hinterlegt (s. Abb. 9.21). Zusätzlich können Vorgabewerte für die Konfigurationen spezifiziert werden, wo dies sinnvoll ist. Die Konfigurationswerte werden letztlich durch den Prozessmodellierer festgelegt, wobei er unpassende Vorgabewerte ändern kann. Konfigurationswerte unterscheiden sich von Eingabeparameter durch ihren Gültigkeitsrahmen: während Konfigurationswerte für

Send Mail [RO]

Configuration

Configuration description

Shows detailed information about the possible configuration values

Name	Description	Datatype	Inheritance	Optional
Mail5MTPPort	The port for of the SMTP server.	INTEGER	inherited	true
Mail5MTPUsername	The user name for the SMTP server.	STRING	inherited	true
MailFrom	The sender used for sending mails.	STRING	inherited	false
Mail5MTPUseTLS	Whether TLS should be used for t...	BOOLEAN	inherited	true
Mail5MTPHostname	The host name of the SMTP server.	STRING	inherited	false
Mail5MTPOvertrustful	Whether the connection to the S...	BOOLEAN	inherited	true
Mail5MTPPassword	The password for the SMTP server.	STRING	inherited	true
Mail5MTPUseSSL	Whether SSL should be used for t...	BOOLEAN	inherited	true

Configuration

Shows detailed information about the concrete values of the configuration

Name	Datatype	Direct value	From input parameter	From process configurat
Mail5MTPUseTLS	BOOLEAN	false	N/A	N/A
Mail5MTPOvertrustful	BOOLEAN	false	N/A	N/A
Mail5MTPUseSSL	BOOLEAN	false	N/A	N/A

General | Runtime | Parameters | Configuration | User Attributes

Abbildung 9.21: Konfiguration einer Aktivitätenvorlage im Aktivitätenrepository

die jeweilige Prozessvorlage und damit alle Prozessinstanzen der Prozessvorlage gelten, sind Werte von Eingabeparameter spezifisch für die jeweilige Prozessinstanz.

Die Aktivitätenvorlagen aus dem Aktivitätenrepository werden mittels Ziehen mit der Maus (*Drag and Drop*) mit Prozessschritten verknüpft (s. Abb. 9.22). Dabei öffnet sich der *Plug and Play*-Assistent, der die Einbindung der Aktivitätenvorlage als Aktivität in den Prozess vornimmt. Hierzu gehört unter anderem die Konfiguration des Anwendungsbausteins. Abb. 9.23 zeigt die entsprechende allgemeine Seite des *Plug and Play*-Assistenten. Der Assistent sorgt dafür, dass alle benötigten Konfigurationswerte vom Prozessmodellierer angegeben werden. Dabei werden auch die bereits im Aktivitätenrepository spezifizierten Vorgabekonfigurationswerte berücksichtigt. Naheliegenderweise muss der Prozessmodellierer umso weniger konfigurieren, je mehr Konfigurationswerte bereits hinterlegt und im aktuellen Prozesskontext sinnvoll sind. Im Idealfall sind bereits alle Konfigurationswerte im Prozessrepository vorhanden, so dass er den Konfigurationsschritt des *Plug and Play*-Assistenten überspringen kann.

Im nächsten und letzten Schritt des *Plug and Play*-Assistenten werden die Ein-/Ausgabeparameter des Anwendungsbausteins mit den Datenelementen der Prozessvorlage verknüpft und somit der Datenfluss etabliert (s. Abb. 9.24). Der Assistent schlägt für jeden Parameter eine Verknüpfung vor. Diese berücksichtigt zahlreiche Informationen. Dazu gehören der Datentyp, Name von Parametern und Datenelementen sowie, falls vorhanden, speziellere Typinformationen wie *Identifier* (s. Abschnitt 7.2.2.1). Ist der entsprechende Prozessschritt bereits über Datenkanten mit Datenelementen verbunden. werden diese verbundenen Datenelemente bevorzugt. Im Idealfall entspricht die vorgeschlagene Verknüpfung der Parameter mit den

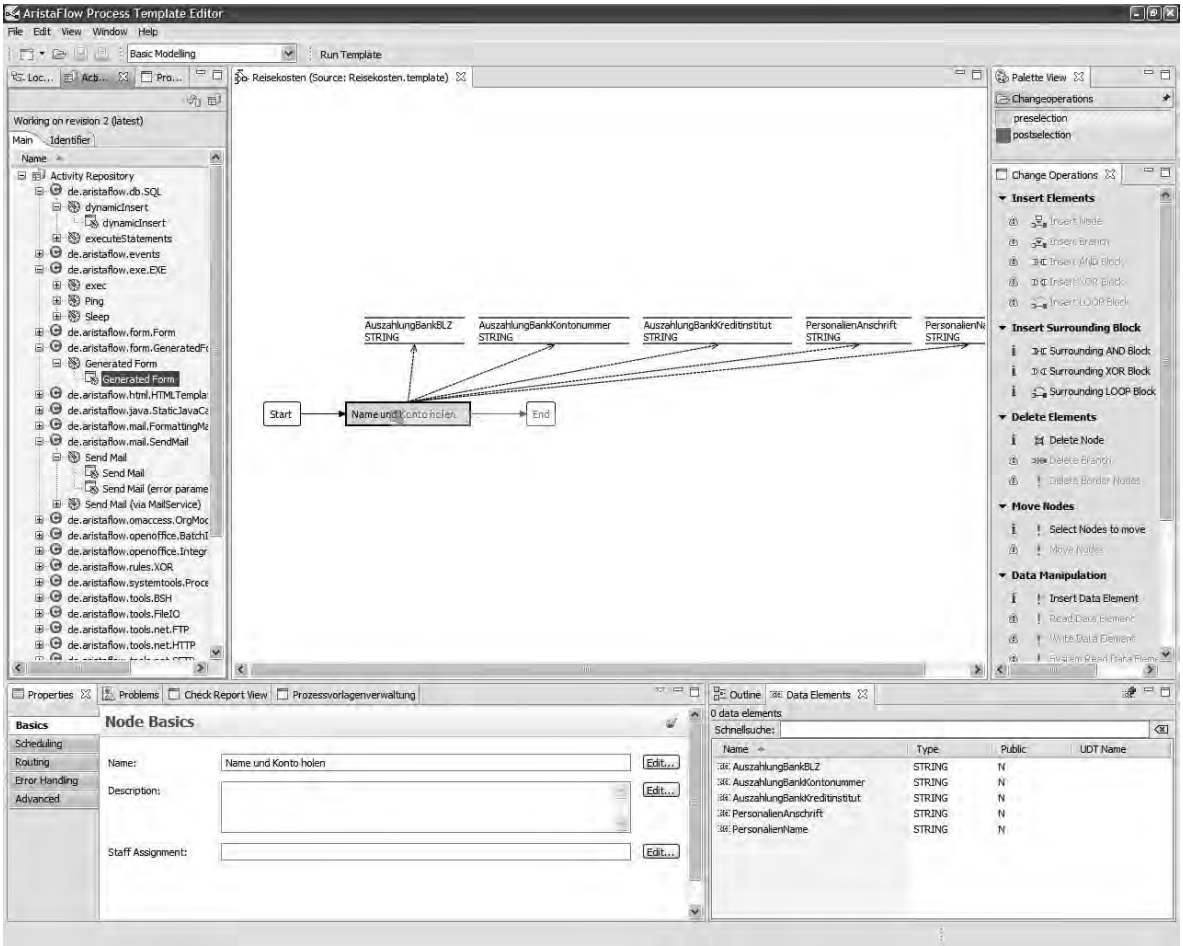


Abbildung 9.22: Verknüpfung einer Aktivitätenvorlagen mit einem Prozessschritt durch Ziehen mit der Maus (*Drag and Drop*)

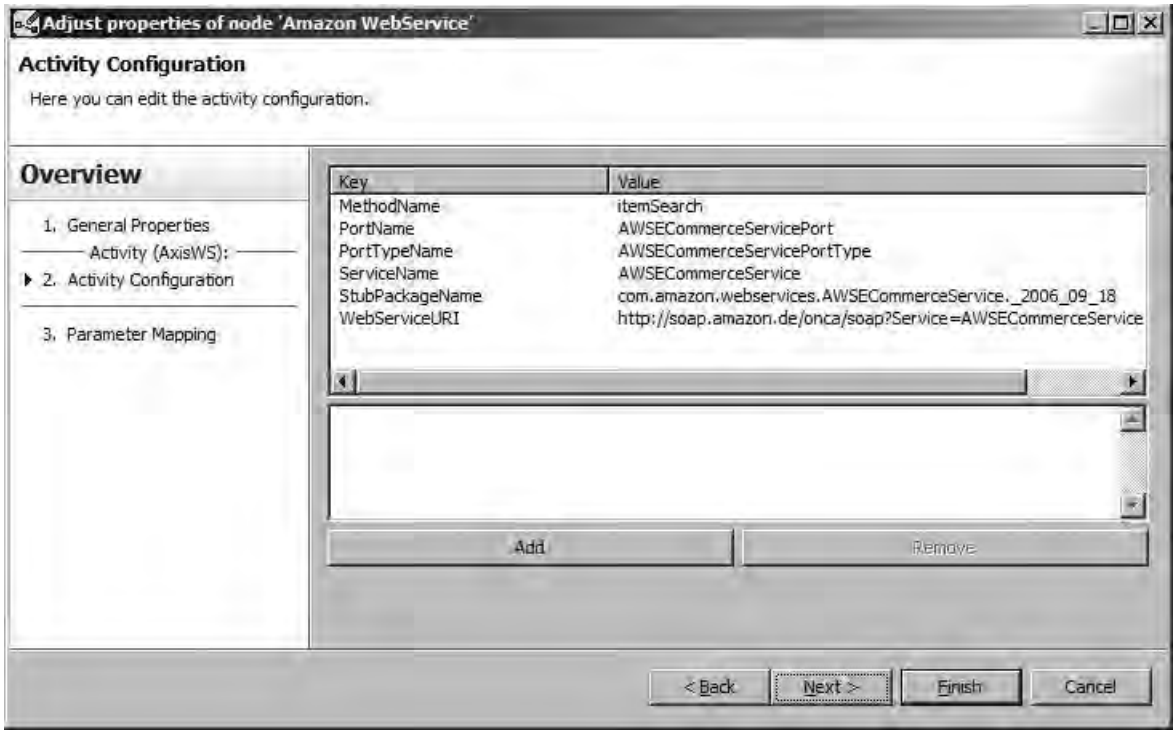


Abbildung 9.23: Konfiguration einer Aktivität bei der Prozessmodellierung

Datenelementen der Intention des Prozessmodellierers und er kann diesen Schritt des *Plug and Play*-Assistenten sofort beenden oder überspringen.

Die bisher gezeigten Seiten des *Plug and Play*-Assistenten sind allgemeingültig und unabhängig von der Aktivitätenvorlage. In diesen Fällen gibt die Aktivitätenvorlage nur die Inhalte der Seiten vor. Der Assistent kann jedoch auch aktivitätsspezifische Seiten haben, die die allgemeine Konfigurationsseite ersetzen. So zeigt Abb. 9.25 die Erstellung von Verzweigungsentscheidungen wie sie für alternative Verzweigungen und Schleifenendknoten benötigt werden. In diesem Fall wird die spezifizierte Verzweigungsentscheidung auf Vollständigkeit und Korrektheit überprüft. In Abb. 9.25 ist beispielsweise nicht der komplette Wertebereich des ganzzahligen Eingabeparameters *Price* abgedeckt. Für einen solchen nicht-abgedeckten Wert ist die Verzweigungsentscheidung nicht definiert. Würde ein solcher Wert bei der Prozessausführung auftreten, gelangt die entsprechende Prozessinstanz in einen ungültigen Zustand.

9.10 Prozessverwaltung und -versionierung

Die graphische Oberfläche für die Prozessverwaltung und -versionierung (FA 11) ist Teil des Prozessvorlageneditors (s. Abb. 9.26). Im linken Teil sieht man den Inhalt des *ProcessRepository* (s. Abschnitt 7.3.2). Jeder Eintrag auf oberster Ebene des Baums (gelbe Quadrate) repräsentiert einen Prozesstyp. Darunter hängen die einzelnen Versionsbäume und die Prozessvorlagen. Die Symbole für die einzelnen Prozessvorlagen zeigen dabei den jeweiligen Zustand bezüglich des Lebenszyklus. Aktive Prozessvorlagen können als Subprozesse verwendet werden. Dazu werden sie bei der Prozessmodellierung wie Aktivitätenvorlagen behandelt (s. Abschnitt 9.9). Mittels

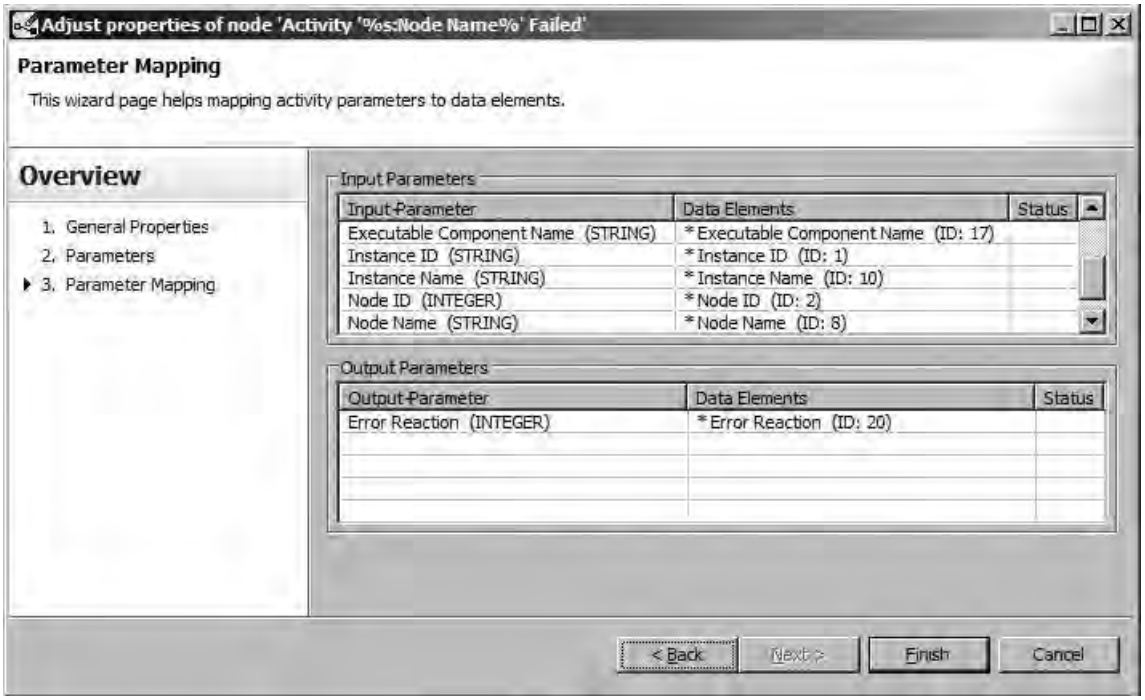


Abbildung 9.24: Einbinden der Ein-/Ausgabeparameter einer Aktivität in den Datenfluss einer Prozessvorlage

Ziehen werden die Prozessvorlagen mit Prozessschritten verknüpft und mit dem *Plug and Play*-Assistenten in den Prozesskontext eingebunden.

Im unteren Teil des Bildschirmfotos befindet sich die Sicht auf die Verwaltung der startbaren und in Ausführung befindlichen Prozessvorlagen (*ProcessManager*, s. Abschnitt 7.3.4). Neue Prozessvorlagen können dort mittels Ziehen aus dem *ProcessRepository* hinzugefügt werden.

9.11 Programmierschnittstelle

Der modulare, dienstorientierte Aufbau des Gesamtsystems bietet einfachen Zugriff auf die Funktionen des adaptiven PrMS. Die Programmierschnittstelle (FA 12) ergibt sich damit durch die Schnittstellen aller Dienste. Die in den vorigen Abschnitten vorgestellten Implementierungen beruhen auf dieser Programmierschnittstelle. Sie decken die naheliegenden Funktionen eines adaptiven PrMS ab. In diesem Abschnitt stellen wir weitere Softwareanwendungen vor, die selbst kein PrMS realisieren, aber eines nutzen. Diese Anwendungen konzentrieren sich auf die Umsetzung konkreter Probleme aus spezifischen Bereichen. Dementsprechend sind Prozesse dort oftmals unter der Oberfläche versteckt und für den Benutzer nicht auf den ersten Blick ersichtlich. Die vorgestellten Beispiele erweitern ihre Funktionen entweder durch die in diesem Kapitel vorgestellte Machbarkeitsimplementierung, oder sie bauen in großem Maße darauf auf.

Ein umfassendes System ist das ERP-System (*Enterprise Resource Planning*) *Microsoft Dynamics NAV*. Dieses dient der Verwaltung und Planung betrieblicher Abläufe durch die Bereitstellung der entsprechenden Daten. Prozessmanagement verbessert den Nutzen signifikant, etwa indem Aufgaben zwischen verschiedenen Benutzern koordiniert werden oder nur noch die Daten bereitgestellt werden, die für eine bestimmte Aufgabe notwendig sind. Abb. 9.27 zeigt

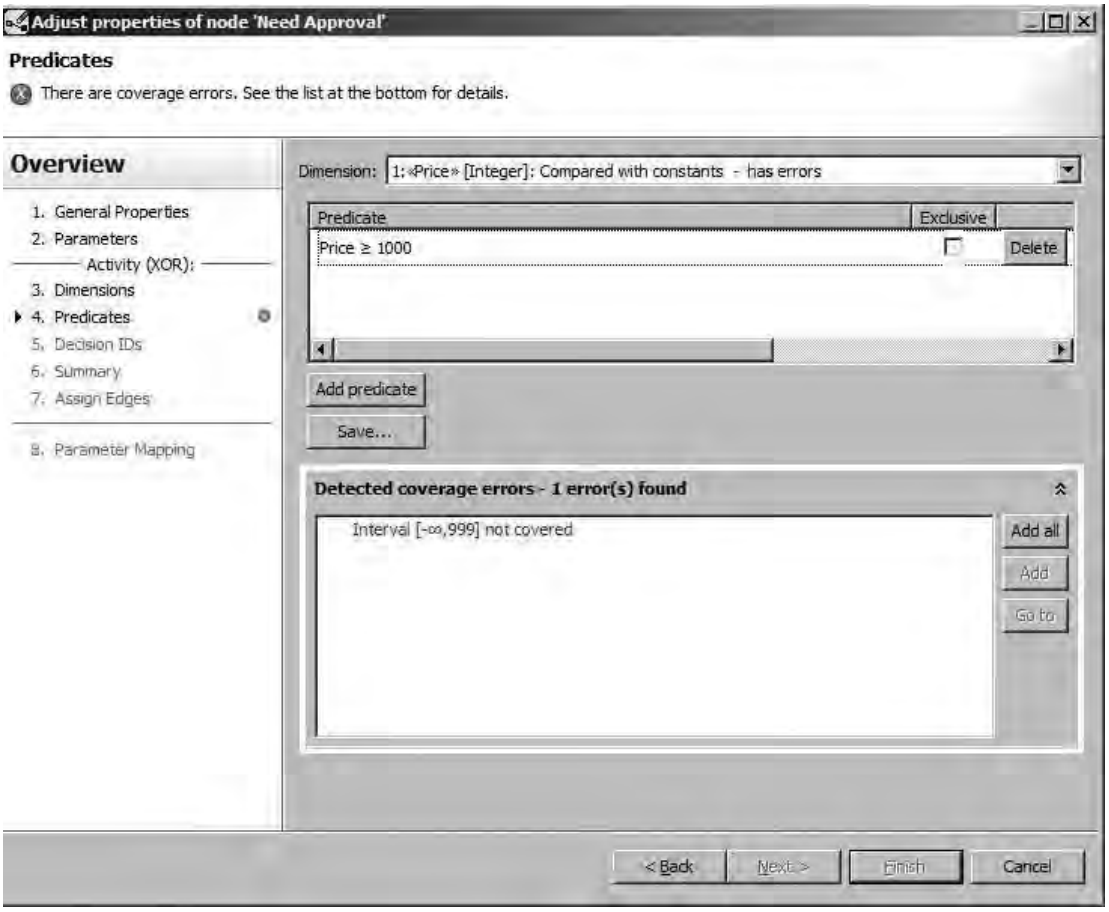


Abbildung 9.25: Aktivitätsspezifische Seite des *Plug and Play*-Assistenten

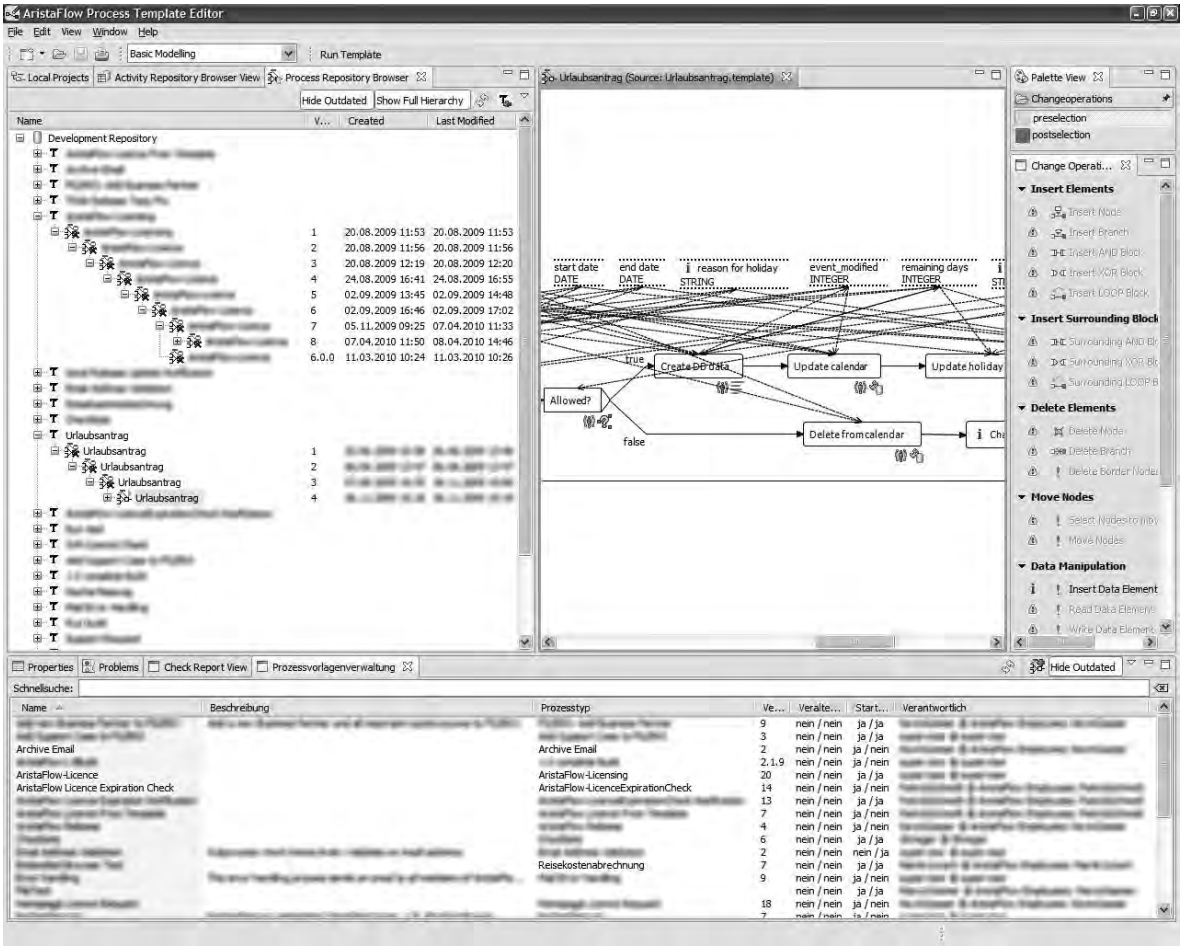


Abbildung 9.26: Prozessvorlagenverwaltung im Prozessvorlageneditor

Abbildung 9.27: *Microsoft Dynamics NAV* mit zusätzlichen Prozessfunktionen

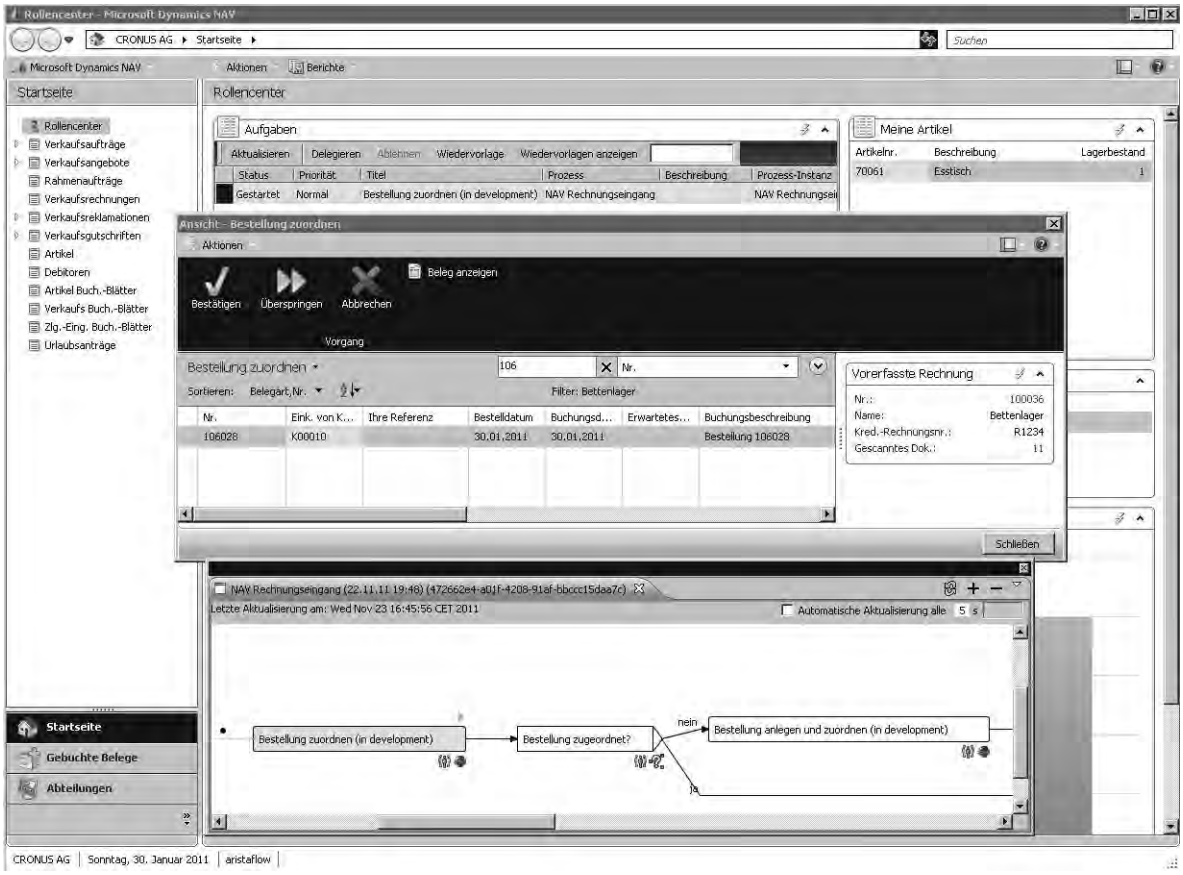


Abbildung 9.28: Ausführung eines Prozessschritts in *Microsoft Dynamics NAV*

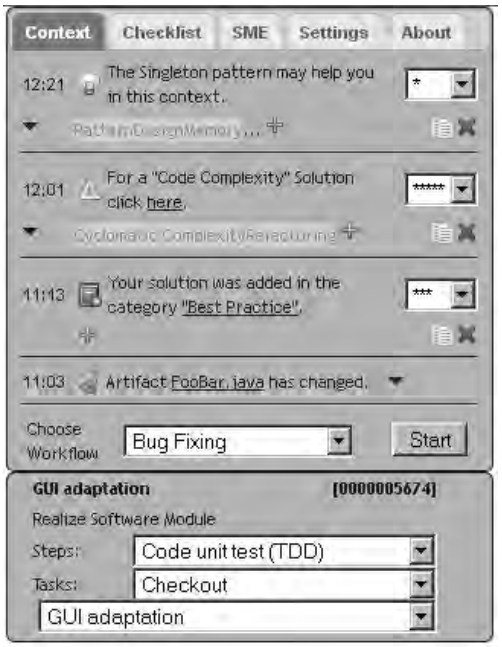
die graphische Oberfläche von *Microsoft Dynamics NAV* in die eine Arbeitsliste (s. Abb. 9.1) sowie eine Liste der startbaren Prozessvorlage (s. Abb. 9.3) integriert ist. Ein separates Fenster zeigt eine laufende Prozessinstanz (s. Abb. 9.1). Die Ausführung eines Prozessschritts zeigt Abb. 9.28. Dabei wird ein vorhandenes Formular von *Microsoft Dynamics NAV* aufgerufen. Beim Schließen des Formulars wird das PrMS informiert und ggf. Daten an den Prozess übergeben (über die Ausgabeparameter des entsprechenden Prozessschritts).

Abb. 9.29 zeigt ein mobiles Gerät, auf dem eine Anwendung aus dem medizinischen Bereich für die Unterstützung von Patienten läuft [spo]. Die Anwendung nutzt das PrMS unter anderem für Behandlungspläne. Instanzspezifische Änderungen der Prozesse ermöglicht die Berücksichtigung von Besonderheiten einzelner Patienten oder Ausnahmefälle. Daneben werden über die Prozesse und das PrMS zahlreiche existierende Softwaresysteme integriert und so die entsprechenden Abläufe vereinfacht.

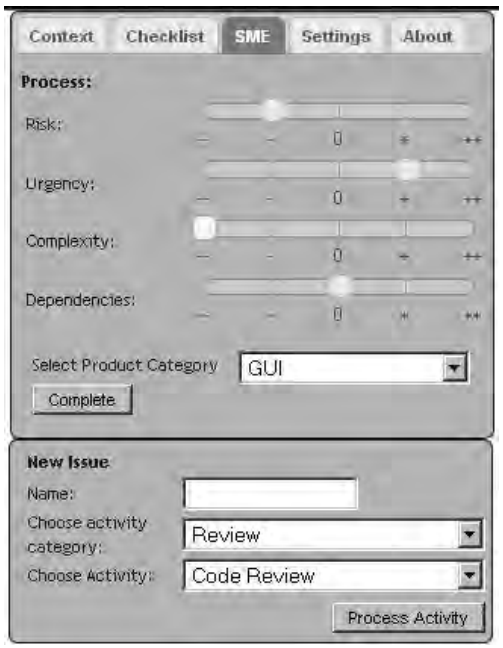
Abb. 9.30 zeigt ein Werkzeug zur Unterstützung von Softwareentwicklung [GOR11a]. Prozesse dienen dabei zur Verbesserung der Zusammenarbeit und Kommunikation sowie zur Koordination und Nachvollziehbarkeit bei auftretenden Probleme der entwickelten Software. Da diese Probleme meist spezielle und individuelle Lösungen erfordern, gibt es dafür keine allgemeingültigen Prozesse. Entsprechende Prozessmodelle wären umfangreich und kompliziert. Aus diesem Grund verwendet das Werkzeug sehr einfache Prozessvorlagen. Diese werden im

Abbildung 9.29: Mobiles Klientenprogramm aus dem medizinischen Bereich [spo]

Laufe der Prozesse über den instanzspezifischen Änderungen über die Programmierschnittstelle angepasst. So entstehen individuelle lösungsorienterte Prozessinstanzen.



(a) Äquivalent einer Arbeitsliste



(b) Start einer neuen Prozessinstanz

Abbildung 9.30: Unterstützung der Softwareentwicklung durch Prozessmanagement [GOR11a]

9.12 Zusammenfassung

Die vorgestellten Machbarkeitsimplementierungen demonstrieren die Umsetzbarkeit der zahlreichen funktionalen Anforderungen eindrucksvoll. Bei der Implementierung haben sich zudem die Vorteile der modularen und dienstorientierten Architektur gezeigt. Damit validieren die Machbarkeitsstudien und insbesondere auch die Verwendung in anderen Forschungsprojekten die Architektur des adaptiven PrMS.

Teil IV

Diskussion und Zusammenfassung

10 Stand der Technik

In diesem Kapitel vergleichen wir das vorgestellte adaptive PrMS bezüglich der wichtigsten funktionalen und nicht-funktionalen Anforderungen mit existierenden PrMS. Dabei betrachten wir akademische Ansätze ebenso wie PrMS aus der Praxis. Den verglichenen Systemen liegen unterschiedliche Konzepte und Metamodelle zugrunde, etwa Petrinetze, regelbasierte Beschreibungssprachen oder deklarative Ansätze. Die Konzepte und Metamodelle selbst spielen beim Vergleich eine untergeordnete Rolle, da der Schwerpunkt auf technischen Aspekten liegt. Die Konzepte haben jedoch Auswirkungen auf die Funktionen der Systeme. Während die Systeme aus der Praxis funktional umfassend sind, beschränken sich die akademischen Ansätze meist auf die Demonstration ausgewählter Aspekte, etwa die Machbarkeit bestimmter fortschrittlicher Konzepte. Der Vergleich beruht auf Literaturstudien, der Evaluation von Systemen, praktischen Erfahrungen sowie der Befragung von Systemexperten.

Bevor wir im Detail auf die verschiedenen akademischen Ansätze (s. Abschnitt 10.2) und Systeme aus der Praxis (s. Abschnitt 10.3) sowie deren Unterschiede eingehen, stellen wir die von uns verwendeten Vergleichskriterien vor (s. Abschnitt 10.1). Am Ende des Kapitels fassen wir den Vergleich kompakt zusammen (s. Abschnitt 10.4).

10.1 Vergleichskriterien

Tabelle 10.1 zeigt die für den Vergleich existierender PrMS relevanten funktionalen und nicht-funktionalen Anforderungen. Die funktionale Anforderung der Prozessausführung (FA 1) wird nicht explizit berücksichtigt, da sie von allen betrachteten Systemen erfüllt wird; der Vergleich umfasst keine reinen Prozessmodellierungswerkzeuge. Ebenso wird die Persistenz (FA 6) nicht betrachtet, da ohne sie ein Produktiveinsatz nicht möglich ist. Neben den funktionalen Anforderungen ist die wichtigste nicht-funktionale Anforderung die der Skalierbarkeit/ Leistungsfähigkeit (NA 1).

Die Systeme werden dahingehend verglichen, ob und wie gut sie die jeweilige Anforderung erfüllen. Gibt es bei den akademischen Ansätzen und Forschungsprototypen zu einer funktionalen Anforderung keine verlässlichen Informationen, wird diese Anforderung nicht bewertet.

Tabelle 10.1: Anforderungen für Vergleich von PrMS

FA 2 Protokollierung
FA 3 Organisationsmodell
FA 4 Arbeitslistenverwaltung
FA 5 Prozessüberwachung (Eskalation, Prozessanalysen)
FA 7 Anwendungsintegration und -aufruf
FA 8 Prozessmodellierung
FA 9.1 Instanzspezifische Änderungen
FA 9.2 Schemaevolution
FA 10 *Plug and Play* (Anwendungsintegration bei der Modellierung)
FA 11 Prozessverwaltung und -versionierung
FA 12 Programmierschnittstelle (API), Entwicklungsplattform
NA 1 Skalierbarkeit/Leistungsfähigkeit

Dies gilt nicht für Skalierbarkeit/Leistungsfähigkeit (NA 1). Hier gehen wir bei keiner Aussage davon aus, dass dieser Aspekt nicht ausreichend berücksichtigt wurde.

Die funktionalen Anforderungen FA 7, FA 8 und FA 10 sind eng miteinander verwandt und werden wie folgt verwendet: FA 7 bewertet, wie einfach sich zusätzliche Anwendungen oder neue Funktionen für Prozessschritte implementieren lassen (analog zu Anwendungsbausteinen in ADEPT [Rei00]). Die komplementäre Anforderung FA 10 bezieht sich darauf, wie einfach eine solche zusätzliche Anwendung oder neue Funktion bei der Prozessmodellierung berücksichtigt werden kann. Dabei spielt es auch eine Rolle, wie weit die Integration bei der Modellierung geht, etwa ob die Ein-/Ausgabedaten berücksichtigt werden und die Datenversorgung des entsprechenden Prozessschritts überprüft wird. Die Überprüfung einzelner Prozessschritte im Rahmen von FA 7 wird durch Verifikation des globalen Datenflusses sowie weitere Korrektheitsprüfungen der Prozessmodelle bei FA 8 ergänzt. FA 8 betrachtet darüber hinaus die Benutzerfreundlichkeit bei der Erstellung von Prozessvorlagen.

In den Abschnitten 10.2 und 10.3 stellen wir die einzelnen Ansätze und Systeme vor. Ist eine Aussage für eine Anforderung relevant und damit für den Vergleich zu berücksichtigen, referenzieren wir jeweils die entsprechende Anforderung.

10.2 Akademische Ansätze und Forschungsprototypen

Bei den akademischen Ansätzen ist das verwendete Metamodell ein wesentlicher Aspekt. Häufig sind Systeme auch Machbarkeitsstudien für entwickelte Metamodelle. Das zugrundeliegende Metamodell hat insbesondere Auswirkungen auf die mögliche Adaptivität (FA 9.1 und FA 9.2) sowie weitergehend auch auf die (Benutzerfreundlichkeit bei der) Prozessmodellierung (FA 8). Wir betrachten in diesem Abschnitt Systeme, denen Petrinetze zugrunde liegen (Chautauqua, Milano, YAWL), einen deskriptiven Ansatz (MOBILE), einen Graphgrammatikansatz (DYNAMITE) sowie graphbasierte Ansätze (MOKASSIN, WASA, PHILharmonicFlows).

10.2.1 Chautauqua

Chautauqua [EM97] beruht auf dem Konzept *Information Control Network* (ICN), welches (eingefärbte) Petrinetze verwendet. ICN sind gegenüber Petrinetzen zusätzlich blockstrukturiert,

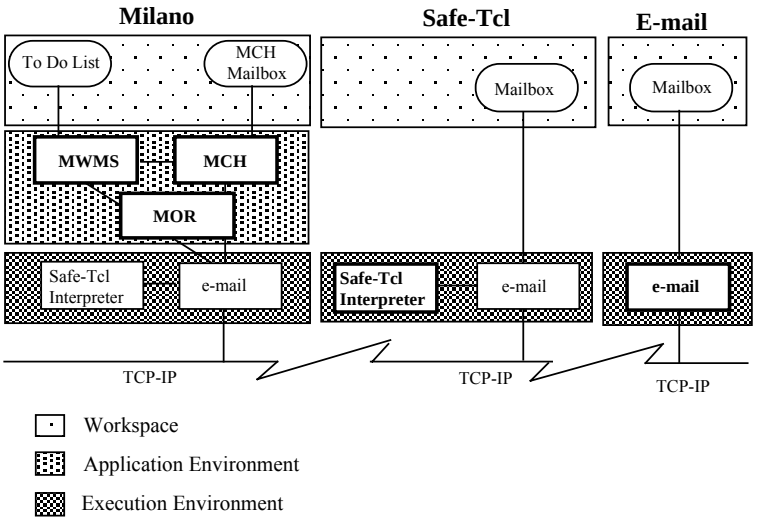


Abbildung 10.1: Architektur von Milano [AdM00b]

und sie haben explizite Start- und Endknoten (FA 8). Der Schwerpunkt dieser Netze sind weniger Prozessabläufe als vielmehr die Kommunikation zwischen beteiligten Personen, auch wenn beide Aspekte miteinander verwandt sind. Daher rührt auch die Beschränkung auf Formulare und Dokumentenfelder als Prozessschritte (FA 7). Die Petrinetztoken in ICN repräsentieren Dokumente, die weitergereicht werden und so den Prozessablauf sowie den Ablauf der Kommunikation repräsentieren. Die „Formularausführung“ ist nur rudimentär gekapselt (FA 10).

An den Token hängt auch die Ausführungs- und Datenhistorie (FA 2) sowie eine Entscheidungsliste für alternative Verzweigungen. Die Entscheidungsliste wird jeweils vom direkten Vorgängerknoten einer alternativen Verzweigung geschrieben. Laufende Prozesse können mittels Graphoperationen und durch das Verschieben von Token (FA 9.1) geändert werden. Die Leistungsfähigkeit von Chautauqua wird lediglich durch Aussagen zur Verwendung von Python [pyt] und des hohen Leistungsbedarfs eines integrierten Webservers für die Objektserialisierung (NA 1) erwähnt.

10.2.2 Milano

Milano [AdM00a, AdM00b] ist eine Kooperationsplattform, in der das PrMS (*Milano Workflow Management System* oder *MWMS* genannt) als Komponente integriert ist (FA 12). Eine weitere Komponente neben MWMS ist *MOR* (*Milano Object Repository*), in dem Organisationsstrukturen und Berechtigungen abgelegt sind (FA 3). Diese werden von der gesamten Plattform genutzt (s. Abb. 10.1).

Durch Integration des PrMS in eine Kooperationsplattform wird der Ansatz verfolgt, dass nur der Kontrollfluss vom PrMS gesteuert wird. Die eigentliche Ausführung von Prozessschritten und der Datenfluss obliegen anderen Teilen der Plattform. Der Kontrollfluss wird über zyklenfreie Petrinetze definiert. Die Flexibilität wird durch explizite Vorwärts- und Rückwärtssprünge erhöht. Diese Sprünge sind auch die einzige Form von Änderungen, die auf laufende Prozessinstanzen angewandt werden können (FA 9.1). Diese bleiben jedoch weit hinter den konzeptionellen Möglichkeiten von ADEPT zurück [RDB03]. Auf Ebene von Prozessvorlagen

kennt MILANO Parallelisierung, Sequentialisierung und Reihenfolgeänderungen (FA 9.2). Für diese Änderungen gibt es (einfache) Korrektheitskriterien für Schemaevolution (FA 9.2). So werden Prozessinstanzen nicht migriert, wenn die Migration in einen unsicheren Zustand führt. Stattdessen wird gewartet, bis die betroffenen Prozessinstanzen so weit gelaufen sind, dass der unsichere Zustand nicht mehr auftritt. Für die Korrektheitsprüfungen wird ein „minimales Kriterium“ benötigt. Dieses muss formal erfasst werden. Es ist ein Prozessmodell, das die minimalen Einschränkungen bezüglich kausaler und temporaler Einschränkungen aufweist. Diese Einschränkungen werden dann bei Änderungen neben der Verklemmungsfreiheit berücksichtigt.

Die Architekten von MILANO plädieren für eine starke Berücksichtigung formaler Grundlagen bei Prozessänderungen. Zumindest für den Kontrollfluss werden sie diesem Anspruch gerecht. Allerdings ignorieren sie den Datenfluss bei Änderungen und verschenken so großes Potential formaler Grundlagen für die Praxis. Zudem muss das minimale Kriterium definiert werden. Ist dieses unvollständig, kann es zu unerwünschten Prozesszuständen kommen.

Es gibt in MILANO eine separate Komponente für die Prozessmodellierung (FA 8), die Anzeige laufender Prozessinstanzen (FA 5) und die Durchführung benutzerdefinierter Sprünge innerhalb einzelner Prozessinstanzen (FA 9.1). Die Sprünge sind allerdings nur für Formularprozesse möglich, nicht aber für allgemeine Prozesse, die auch mit anderen Teilen der Plattform interagieren. Bei solchen Prozessen werden die Prozessschritte durch externe Komponenten ausgeführt (FA 7). Der Benutzer sieht die für ihn anstehenden Schritte in einer Arbeitsliste (FA 4). Benutzer, die keinen entsprechenden Klienten für die Arbeitslistenanzeige haben, erhalten pro Arbeitslisteneintrag eine Mail. Diese Mail repräsentiert auch gleichzeitig den Prozessschritt selbst und enthält Verweise zu den Daten, die für die Ausführung des Prozessschritts notwendig sind. Obwohl MILANO lediglich ein Prototyp ist, gibt es Überlegungen von Tcl [tcl] zu Java zu wechseln, um die Leistungsfähigkeit zu erhöhen (NA 1).

10.2.3 YAWL

YAWL [YAW10, YAW12] ist aus einer prototypischen Implementierung der Kontrollflussmuster aus [vdAtHKB03] entstanden. Diese Implementierung wurde nach und nach zu einem vollwertigen PrMS erweitert. Dabei wurden zahlreiche Architekturaspekte berücksichtigt. Einige der Erweiterungen entstammen wissenschaftlichen Arbeiten zu bestimmten Aspekten, etwa *Worklet*s [AtHEvdA06] für die Flexibilisierung der Prozessausführung.

Naheliegenderweise deckt der Funktionsumfang von YAWL viele der auch in dieser Arbeit vorgestellten Dienste ab. Ein großer Unterschied liegt allerdings darin, dass der Kern von YAWL lediglich Prozesse und Prozessinstanzen interpretiert. Das entspricht den in dieser Arbeit vorgestellten Diensten *ProcessManager*, *DataManager* und *ExecutionManager*. Alle weiteren Funktionen werden durch sog. *Custom Service*s umgesetzt. Hierzu gehört insbesondere die Arbeitslistenverwaltung (FA 4) und die Ausführung von Prozessschritten (FA 7). Für Benutzerschritte müssen somit Anwendungsbausteine als *Custom Service*s implementiert werden. Es gibt einen vordefinierten *Custom Service*, der eine einfache Arbeitsliste implementiert und automatisch Formulare für Prozessschritte erzeugt. Auch für die automatische Ausführung von Prozessschritten gibt es einen vordefinierten *Custom Service*, der dynamisch Anwendungsfunktionen in Form sog. *Codelet*s aufrufen kann. Diese entsprechen Anwendungsbausteinen, sie funktionieren allerdings nur bei automatischer und nicht bei interaktiver Ausführung.

Auf diese Weise stellen sich bei YAWL zahlreiche Probleme einer allgemeingültigen Anwendungsintegration nicht. Dagegen fällt erheblich mehr Implementierungsaufwand an, da die Implementierung der Anwendungsbausteine immer auch eine Arbeitsliste umfasst. Die

Schnittstelle des Systems ist als Java-API realisiert. Dazu gehört auch die Schnittstelle zu einem *Custom Service* (FA 12).

Das umfangreiche Organisationsmodell von YAWL umfasst Personen, Rollen, Stellen und Fähigkeiten (FA 3). Auch abhängige Bearbeiterzuordnungen sind möglich, also solche mit Bezug zu anderen Prozessschritten. Der Kontrollfluss beruht auf Kontrollflussmustern. Er wird auf statische und dynamische Korrektheit überprüft (FA 8). Dazu gehört etwa, dass es keine Knoten ohne ausgehende Kontrollkanten gibt, dass keine Verklemmungen auftreten können und dass es keine unsinnigen Verzweigungen gibt. Außerdem kann der Kontrollfluss durch entsprechende Reduktionsregeln vereinfacht werden. Der Datenfluss umfasst prozessweite Variablen, Ein-/Ausgabeparameter und lokale Variablen an Prozessschritten. Er ist streng getypt und beruht auf XML. Für den Datenaustausch mit entsprechenden *Custom Service* wird DOM verwendet. Ein-/Ausgabeparameter von Prozessschritten werden mittels XQuery (*XML Query Language*) mit prozessweiten und lokalen Variablen verknüpft (s. Abb. 10.2). Dementsprechend benötigt der Modellierer gute Kenntnisse zu XML, XSD und XQuery. Zur Modellierzeit wird XSD auf Syntaxfehler überprüft (FA 8), eine Validierung findet jedoch erst zur Laufzeit statt.

An Prozessschritten werden *Custom Service* hinterlegt. Sind die entsprechenden Implementierungen bei der Modellierung verfügbar, können darüber die Ein-/Ausgabeparameter automatisch ermittelt werden. Anschließend wird überprüft, dass alle Parameter entsprechend mit prozessweiten Variablen verknüpft sind (FA 10). *Codelet*s werden nicht auf diese Weise geprüft. Dies kann dazu führen, dass zur Laufzeit Eingabewerte fehlen und die *Codelet*s nicht ausgeführt werden. Allerdings gilt der entsprechende Prozessschritt dann trotzdem als abgeschlossen [YAW10].

Flexibilitätsaspekte deckt YAWL mittels *Worklet*s ab [AtHEvdA06]. Dies sind eigenständige (Sub-)Prozesse, vergleichbar mit Fehlerbehandlungsprozessen. Im Fehlerfall ersetzt ein *Worklet* zur Laufzeit dynamisch einen fehlgeschlagenen Prozessschritt (FA 9.1). Dazu muss es explizit im Prozessmodell hinterlegt sein. Ein *Worklet* unterscheidet sich somit signifikant von einer frei definierbaren instanzspezifischen Änderung. Allerdings können verschiedene *Worklet*s an einem Prozessmodell hinterlegt werden. Über Regeln können Bedingungen definiert werden, wann welches *Worklet* auszuwählen ist. Einerseits können so abhängig vom aufgetretenen Fehler unterschiedliche *Worklet*s einen Prozessschritt ersetzen. Andererseits können *Worklet*s nicht nur im Fehlerfall aufgerufen werden, sondern die Regeln können sich auch auf bestimmte Daten beziehen (FA 5). Weisen die referenzierten Variablen dann bestimmte Werte auf, wird das entsprechende *Worklet* ausgeführt. Diese Regeln werden zu definierten Zeitpunkten ausgewertet, etwa beim Starten und Beenden von Prozessinstanzen und Prozessschritten sowie bei Zeitüberschreitungen [YAW12].

10.2.4 MOBILE

MOBILE [Jab97, Jab98, Jab00] ist aus der wissenschaftlichen Betrachtung der Architektur von PrMS entstanden. So kritisiert [Jab97], dass die Architektur von PrMS zu wenig Beachtung seitens der Wissenschaft erfährt und die existierenden Architekturbetrachtungen mehr Erfahrungsberichte als wissenschaftliche Untersuchungen seien.

[Jab97] sieht PrMS erstmals als Middleware-Plattform für die Nutzung durch andere Software (FA 12), sie bieten normalerweise aber auch eigene Werkzeuge an, etwa zur Prozessmodellierung. Ein PrMS muss nach diesem Verständnis den Kontroll- und Datenfluss der auszuführenden Prozesse sowie ein Organisationsmodell (FA 3) kennen. Außerdem muss man zusätzliche An-

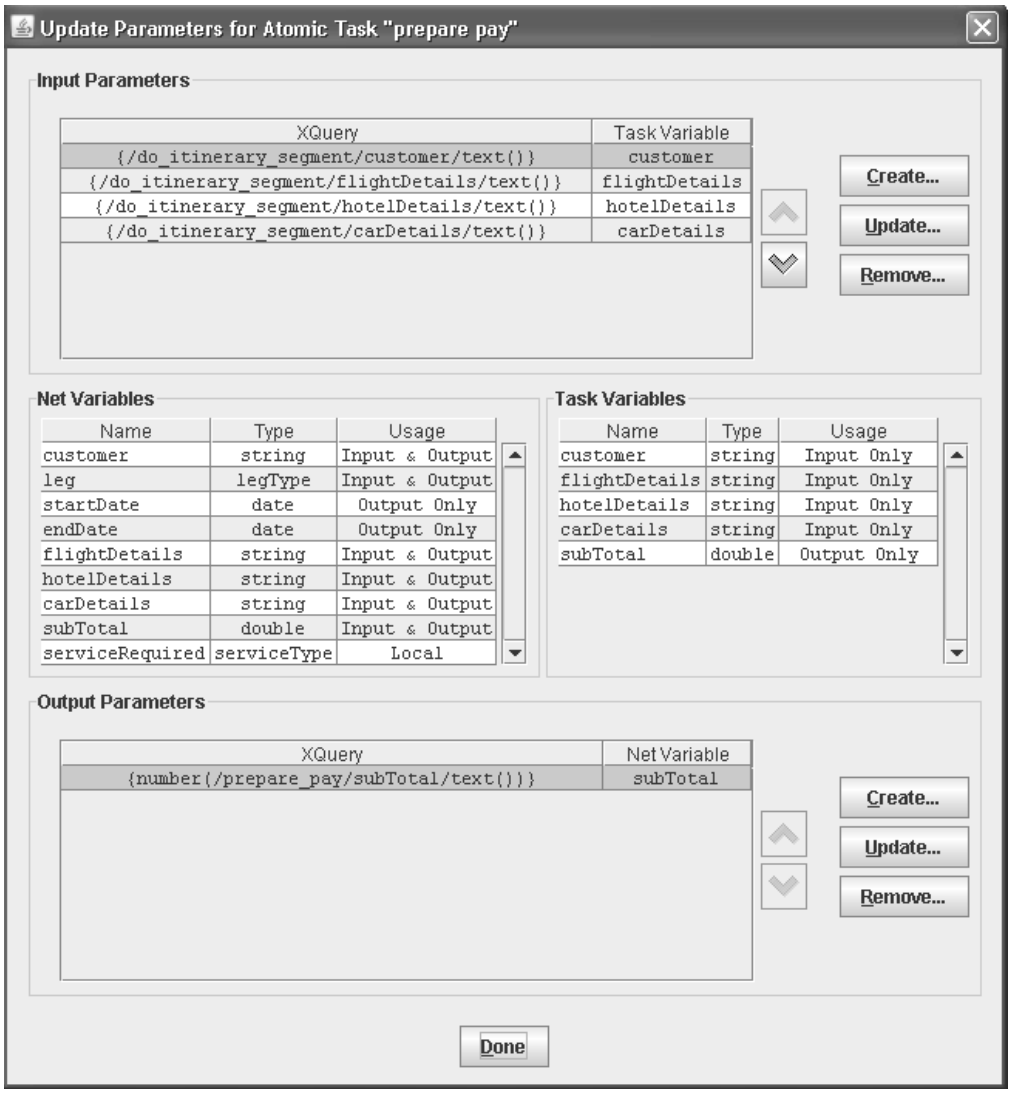


Abbildung 10.2: Anbindung an den Datenfluss in YAWL [YAW12]

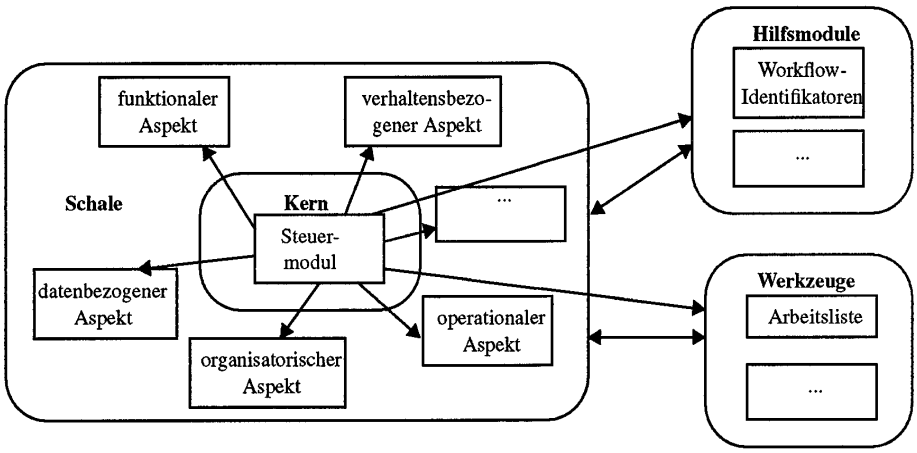


Abbildung 10.3: Architektur eines PrMS aus Implementierungssicht [Jab97]

wendungen in das PrMS einbinden können (FA 7). Sicherheitsaspekte sind für [Jab97] optional. Weitere Forderungen betreffen die dynamische Anpassbarkeit von Prozessen, Versionsverwaltung sowie für die verschiedenen Module des PrMS Skalierbarkeit, Modularität, Orthogonalität (Redundanzvermeidung) und Unabhängigkeit. Besonderer Wert liegt auf der flexiblen Erweiterbarkeit eines PrMS, um es bei Bedarf an spezifische Anforderungen anpassen zu können. Dies ist im Kontext von MOBILE notwendig, da hier Anwendungsbausteine nicht losgelöst sind vom PrMS, sondern aus Architektursicht direkt in dieses integriert werden. Damit erweitern sie das PrMS um die entsprechenden Anwendungsfunktionen (FA 7, FA 10). Als gute Möglichkeit für die Integration wird CORBA (*Common Object Request Broker Architecture*) vorgeschlagen, was zur damaligen Zeit den Stand der Technik widerspiegelte.

Die einzelnen Module der PrMS-Architektur in [Jab97] (s. Abb. 10.3) entsprechen weitgehend den in dieser Arbeit vorgestellten Diensten eines PrMS. Der funktionale Aspekt entspricht dem *ExecutionManager* (s. Abschnitt 7.3.6), der verhaltensbezogene Aspekt dem *SubprocessManager* (s. Abschnitt 7.3.8.5), der datenbezogene Aspekt dem *DataManager* (s. Abschnitt 7.3.5), der organisatorische Aspekt dem *OrgModelManager* (s. Abschnitt 7.3.3) und der operationale Aspekt dem *RuntimeManager* (s. Abschnitt 7.3.8). Das Steuermodul aus [Jab97] ist bei der in dieser Arbeit vorgestellten Architektur im *ExecutionManager* anzusiedeln, da dieser die anderen Dienste steuert. Die Aktivitätsträger (*Threads*) ordnet [Jab97] aus Gründen der besten Skalierbarkeit den Modulen zu (NA 1). Aktivitätsträger können darüber hinaus auch modulübergreifend sein; dies entspricht unserer Zuordnung.

Zur Korrektheit von Prozessen gehören nach [Jab00] nicht nur die Korrektheit bezüglich der Ausführbarkeit, wie dies oft bei auf Petrinetzen beruhenden Konzepten der Fall ist, sondern auch Aspekte wie Daten- und Materialfluss. Dabei wird aufgrund der theoretischen Unentscheidbarkeit zahlreicher Aspekte zu einem pragmatischen, werkzeugunterstützten Ansatz geraten. So soll ein Prozessmodellierer zwar auf eine potentielle Endlosschleife hingewiesen werden, es liegt aber in der Verantwortung des Modellierers, ob diese terminiert oder nicht (FA 8). Das heißt, das System akzeptiert auch eine Endlosschleife.

Die Architekturbetrachtungen in [Jab97, Jab98, Jab00] wurden in der Machbarkeitsstudie MOBILE technisch umgesetzt.

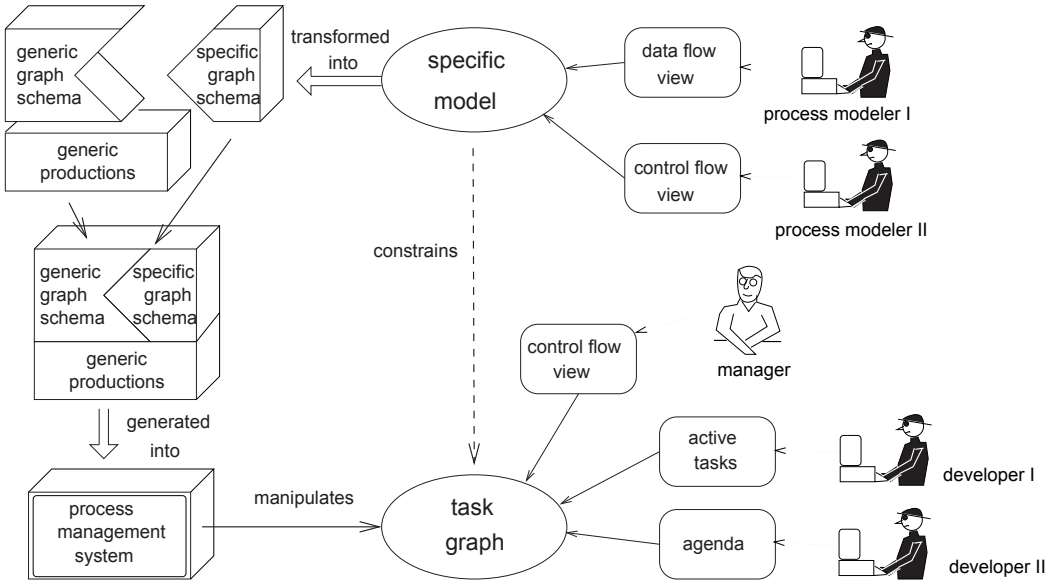


Abbildung 10.4: DYNAMITE-Ansatz zur Erstellung von Prozessen/eines entsprechenden PrMS [HJKW96]

10.2.5 DYNAMITE

DYNAMITE [HJKW96] und das Nachfolgesystem AHEAD [Wes01b] beruhen auf einem Graphgrammatikansatz, die Modellierung erfolgt mittels Graphtransformation (FA 8). Dabei wird aus einer Menge von allgemeingültigen Regeln, domänenspezifischen Parametern sowie ggf. weiteren domänenspezifischen Regeln ein domänenspezifisches Modell und hieraus dann das PrMS erzeugt (s. Abb. 10.4). Das PrMS ist somit selbst nicht allgemeingültig, d. h. man kann beispielsweise nicht einfach beliebige neue Prozesse hinzufügen und ausführen (FA 11). Neue Prozesse müssen erst die Transformation durchlaufen, was dann zu einer neuen Version des PrMS führt.

Durch den Graphgrammatik- und Graphtransformationsansatz sind Modellierung und Ausführung eng verflochten. Somit sind instanzspezifische Änderungen und Schemaevolution inhärenter Bestandteil des DYNAMITE-Ansatzes (FA 9.1, FA 9.2). Allerdings ist für komplexe Änderungen entsprechendes Wissen über Graphtransformationen notwendig (FA 8). Außerdem liegen die Prozesse nicht als Graphen, sondern als komplexe Datenobjekte vor, die nicht direkt angezeigt werden können. Schon aus diesem Grund werden die Prozesszustände den beteiligten Personen in Form von Arbeitslisten dargestellt.

Das Metamodell kennt sowohl Kontroll- als auch Datenfluss. Es gibt keine impliziten Schleifen sondern nur explizite Rücksprünge (FA 8). Korrektheitskriterien sind für Kontroll- und Datenfluss definiert (FA 8). Für den Datenfluss gelten lockere Bedingungen. So ist es möglich, unvollständige Daten an nachfolgenden Schritte weiterzureichen, d. h. der die Daten produzierende Schritt muss noch nicht vollständig abgeschlossen sein, bevor nachfolgende Schritte die Daten lesen. Allerdings können die nachfolgenden Schritte erst dann beendet werden, wenn der produzierende Schritt abgeschlossen ist und die nachfolgenden Schritte die endgültigen Datenwerten gelesen haben. Dies ist sehr ähnlich zu dem Ausführungsparadigma von *Case Handling*-Systemen [KWR11, KR11b, GRvdA08, WMR10, MWR08]. Der Kontrollfluss hat

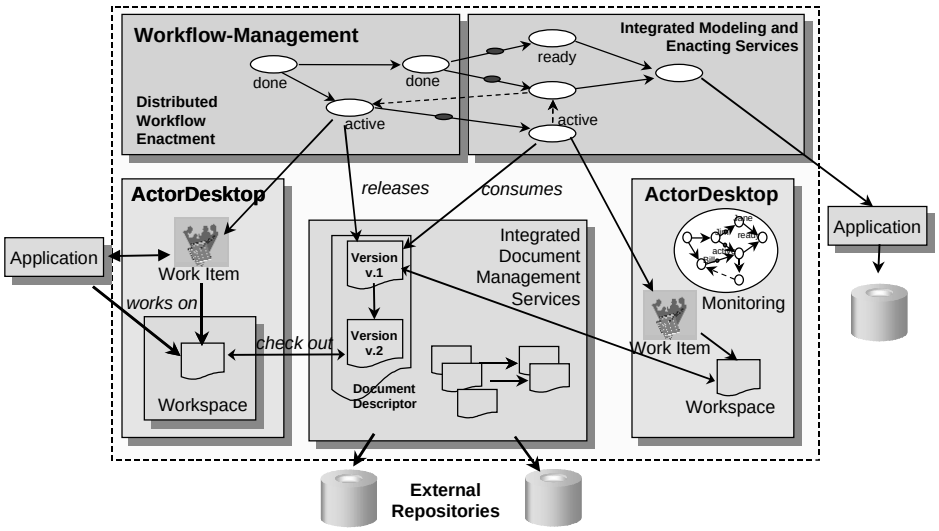


Abbildung 10.5: Architektur von MOKASSIN [GJS+99]

ein definiertes Zustandsmodell. Dessen Transitionsregeln können jedoch für die Transformation zum domänenspezifischen Modell angepasst werden. So lassen sich etwa Bedingungen bezüglich der Abgeschlossenheit eines Prozessschritts aufweichen.

Prozesse in DYNAMITE können hierarchisch geschachtelt sein. Bei der Modellierung wird zwischen dem Was und dem Wie getrennt (FA 7, FA 10), d. h. die Modellierung ist diesbezüglich mit der Verwendung von Anwendungsbausteinen vergleichbar. Neben der Graphtransformation für Änderungen an Prozessvorlagen und -instanzen bietet DYNAMITE für einfache Fälle eine „Reaktivierung“ bereits abgeschlossener Prozessschritte. Dabei werden abgeschlossene Prozessschritte einfach erneut in den weiteren Prozessablauf eingefügt und ermöglichen so eine Vorwärtskompensation. Die Schritte werden geeignet versioniert, damit diese Änderungen nachvollziehbar sind (FA 5).

10.2.6 MOKASSIN

MOKASSIN [JH98, GJS+99, JWH00] ist ein graphbasierter Ansatz mit dem Ziel der verteilten, dynamischen und unternehmensübergreifenden Ausführung von Prozessen. Das PrMS ist ähnlich wie Milano mit einem DMS (Dokumentenmanagementsystem) gekoppelt. In MOKASSIN werden Aktivitäten durch eigenständige Agenten repräsentiert. Sie kommunizieren mittels Nachrichten und Ereignissen [GJS+99], was die verteilte Ausführung erleichtert. Dementsprechend hat MOKASSIN auch keine fest definierten Vorgaben für die Prozessausführung. In der Architektur und im System wird nicht zwischen Modellier- und Laufzeit getrennt. MOKASSIN unterscheidet aber zwischen dem Was und Wie eines Prozessschritts und ist damit bezüglich der Modellierung mit der Verwendung von Anwendungsbausteinen vergleichbar (FA 7, FA 10). MOKASSIN ist mittels CORBA implementiert, worüber auch externe Systeme eingebunden werden können [JWH00]. Auch die Integration von Web-Anwendungen ist möglich (s. Abb. 10.5).

Für [JH98] ist Flexibilität für die Verwaltung der verschiedenen Prozessversionen wichtig. MOKASSIN kennt verschiedene Varianten der Schemaevolution: *lazy*, *eager* und *selective*. Dies bezieht sich darauf, wie und wann die Änderungen an der Prozessvorlage propagiert werden.

Dabei werden auch ggf. bereits vorhandene instanzspezifische Änderungen berücksichtigt (FA 9.1, FA 9.2). In jedem Fall muss die Konsistenz gewährleistet sein. Hierfür wird das Kriterium der *Trace Equivalence* [RW12] verwendet, also dass die momentane Ausführungshistorie auch nach Anwendung der Änderungen vom geänderter Prozess hätte erzeugt werden können.

10.2.7 WASA

WASA ist aus der wissenschaftlichen Betrachtung der Architektur von PrMS entstanden [Wes99, Wes01a], vergleichbar zu MOBILE und der vorliegenden Arbeit. Die wichtigsten Anforderungen an die Architektur nach [Wes99] sind Skalierbarkeit (NA 1), Flexibilität (im Sinne von FA 9) und Integration existierender Anwendungen (FA 7). Allerdings wird die Architektur auf abstrakter Ebene nur kurz diskutiert und daraus dann entsprechende Datenmodelle sowie das Metamodell (s. Abb. 10.6) entwickelt. Dementsprechend wird das Metamodell auch nicht so umfassend wissenschaftlich diskutiert wie etwa in [RD98, Rei00, RRD04c, Rin04].

[Wes01a] hält die Auswahl einer geeigneten Infrastruktur für die Architektur und Implementierung für wichtig und wählt selbst Objektorientierung, CORBA und Java (s. Abb. 10.7). Damit ist das System programmiersprachenunabhängig, und die verwendeten Implementierungsobjekte können zur Laufzeit transparent zwischen verschiedenen Knoten und Plattformen verteilt werden (FA 12). Leider wird dieses ebenso wie weitere Architekturaspekte in [Wes99] und [Wes01a] nicht diskutiert, sondern technische Konzepte, Standards und Implementierungen einfach als Lösung akzeptiert. So wird CORBA als Lösung für die verteilte Ausführung sowie für die Integration existierender Anwendungen (FA 7) betrachtet, ohne dabei genauer auf Probleme und Lösungsdetails einzugehen. Dass hier durchaus eine wissenschaftliche Betrachtung sinnvoll ist, zeigen [BRD03, Bau01].

WASA hat ein einfaches Rollenkonzept für die Bearbeiterzuordnung. Bearbeiterzuordnungen können von Daten abhängen; es gibt Delegation sowie Vertreterregeln (FA 3). Bei der Modellierung werden sowohl Kontroll- als auch Datenfluss in einem Prozessgraphen spezifiziert. Die Prozessgraphen werden vom PrMS interpretiert, was einen einfachen Wechsel zwischen Modellier- und Laufzeit ermöglicht. Prozessvorlagen können ebenso geändert werden wie individuelle Prozessinstanzen (FA 9.2, FA 9.1). Für die interne Repräsentation werden Vorlagenreferenzen (s. Abschnitt 4.1.3.1) zur Vermeidung von Redundanz und zur einfacheren Realisierung von Schemaevolution verwendet. Allerdings wird dies in [Wes99] nicht weiter diskutiert. Durch die Verwendung von Vorlagenreferenzen und einem einfachen Versionierungsmechanismus für Änderungen – beide Änderungsarten unterscheiden sich nur in der Sichtbarkeit – ist die Kombination von Schemaevolution und instanzspezifischen Änderungen in WASA nicht möglich. Für die Versionierung wird ein Prozessrepository verwendet (FA 11).

WASA bietet allgemeine Konsistenzkriterien für Prozessmodelle. Diese sind jedoch eher einfacher Natur, etwa dass der Datenfluss keinen Zyklus mit dem Kontrollfluss bilden darf. Korrektheitskriterien für PrMS, etwa analog dem Serialisierbarkeitsprinzip in DBMS, sind für [Wes99] nicht adäquat. Diese über einfache und grundlegende Dinge hinausgehende „semantische Korrektheit“ muss durch den Prozessmodellierer festgelegt und eingehalten werden. Für Prozessänderungen gibt es ähnlich einfache Kriterien, etwa dass die Vergangenheit nicht geändert werden darf (FA 9.1, FA 9.2). Diese Kriterien werden bei einer Schemaevolution automatisch geprüft (siehe auch [RRD03c, RRD04a]).

Die prototypische Implementierung von WASA hat einige zusätzliche Funktionen und Anwendungsprogramme. Hierzu gehören eine Prozesseditor (FA 8, eine Arbeitslistenverwaltung (FA 4) sowie eine Monitoranwendung (FA 5).

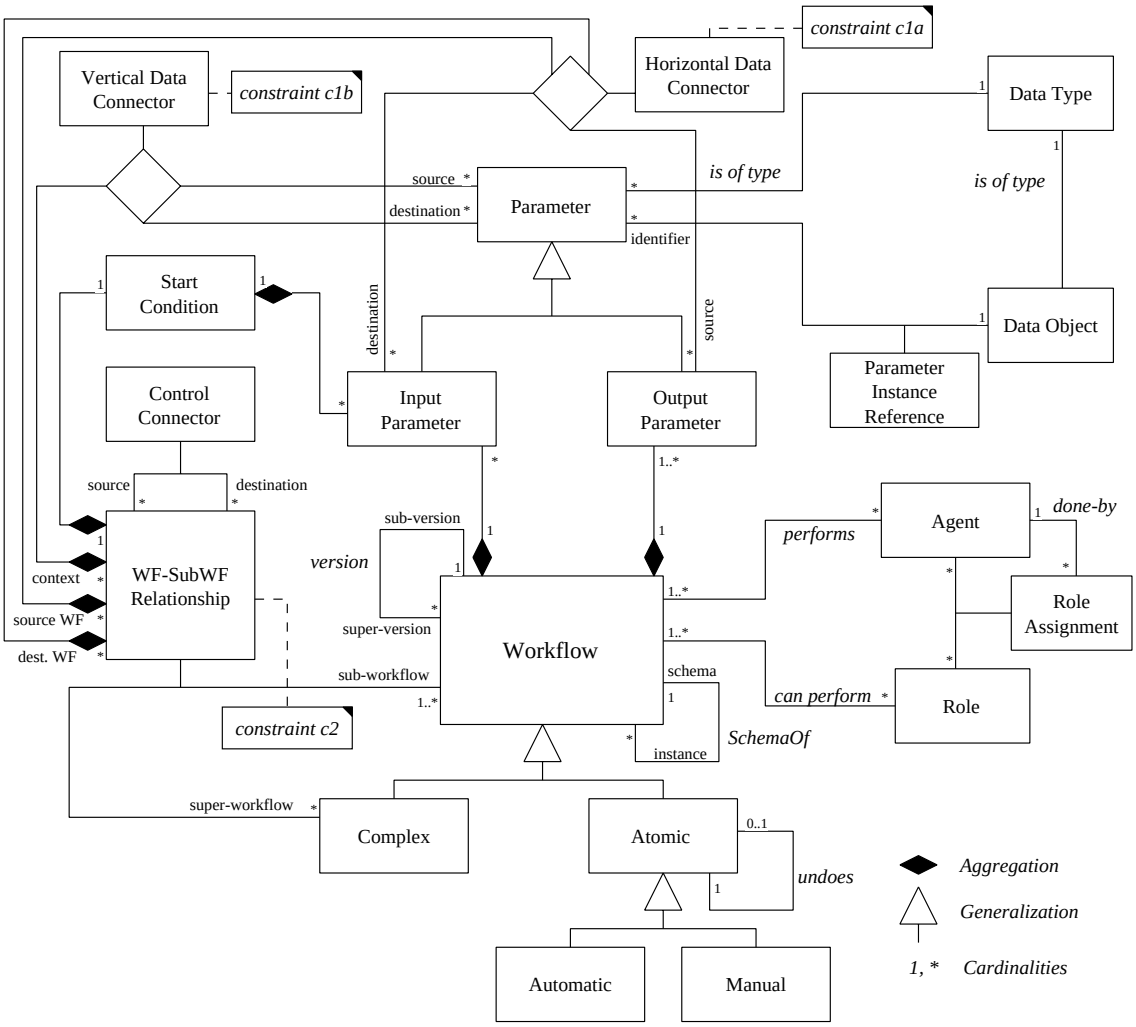


Abbildung 10.6: Metamodell von WASA [Wes99]

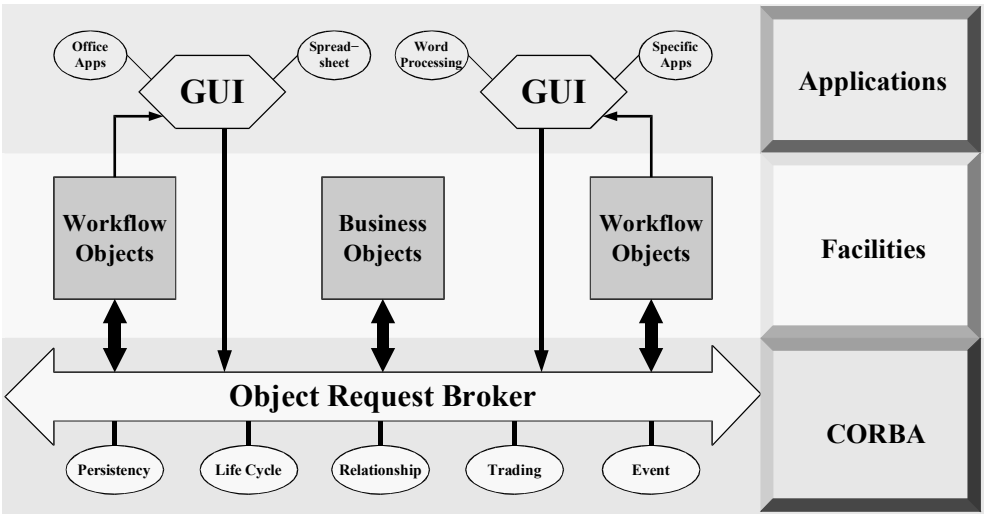


Abbildung 10.7: Abstrakte Architektur von WASA [Wes99]

In [Wes99] und [Wes01a] nimmt die Darstellung und Diskussion der Speicherung von Implementierungsobjekten des PrMS viel Raum ein. Allerdings werden interessante Diskussionen mit technischen Lösungen abgekürzt. So soll die Verwendung eines DBMS transaktionales Verhalten und die Möglichkeit zur Wiederherstellung eines konsistenten Zustands im Fehlerfall ermöglichen. [Wes99] beschreibt dazu, wie eine Transaktion zum Laden eines Objekts verwendet wird. Im Kontext von PrMS reicht dies jedoch nicht aus. Transaktionen müssen hier auf einer höheren Ebene gelten und umfassen wesentlich mehr. Das Starten eines Prozessschritts erfordert beispielsweise das Laden mehrerer Objekte, deren Manipulation und das Zurückschreiben dieser Objekte innerhalb derselben Transaktion. Auf dieser Ebene muss das System auch eine automatische Wiederherstellung beherrschen. Es reicht nicht aus, wenn die einzelnen Objektdaten im DBMS konsistent sind. Ist ein Prozessschritt etwa gestartet, muss das PrMS auch dafür sorgen, dass die entsprechende Anwendung auch läuft, oder das PrMS muss den Prozessschritt entsprechend zurücksetzen, so dass er neu gestartet werden kann.

10.2.8 PHILharmonicFlows

PHILharmonicFlows [Kün13, KR11b, KR11a, KR09b, KR09a] ist ein umfassendes Konzept, das über reines Prozessmanagement hinausgeht und die Prozessen wesentlich enger mit Daten integriert. In PHILharmonicFlows bilden Daten und Prozesse verschiedene Sichten desselben Modells. Während bei klassischem Prozessmanagement die Daten lediglich durch Prozesse manipuliert und verwendet werden, haben sie in PHILharmonicFlows eigene Lebenszyklen. So können Zustände und Zustandsübergänge für Daten definiert werden, die bei der Manipulation der Daten durch Prozesse mit berücksichtigt werden [KR11a]. Daneben können Daten auch außerhalb von Prozessen manipuliert werden. Die entsprechenden Zustandsänderungen haben auch Auswirkungen auf die jeweiligen Prozesse. PHILharmonicFlows ist also die Vereinigung von klassischem Prozessmanagement mit *Case Handling* [KWR11, KR11b, KR12, GRvdA08, WMR10], wobei die Konsistenz zwischen Prozess- und Datenzuständen gewährleistet ist.

Aufgrund dieses Schwerpunkts betrachtet PHILharmonicFlows einige Aspekte von adaptivem Prozessmanagement bisher nur am Rande oder gar nicht. Hierzu gehören die Protokollierung in Form von Ausführungs- und Datenhistorien, instanzspezifische Änderungen und Schemaevolution sowie die Prozessverwaltung und -versionierung. Diese Aspekte sollen zukünftig im Kontext von PHILharmonicFlows allerdings vertiefend untersucht werden oder sind bereits Gegenstand aktueller Forschung (z. B. [CKR12a]). Die Leistungsfähigkeit wird an zahlreichen Stellen erwähnt, jedoch nicht im Detail untersucht, da sich PHILharmonicFlows bisher auf die entsprechenden theoretischen Konzepten und deren Machbarkeit konzentriert hat und nicht deren praktische Umsetzung im Vordergrund stand. Nichtsdestotrotz gibt es bereits eine umfassende prototypische Implementierung.

Das Organisationsmodell ist in PHILharmonicFlows inhärent im Konzept integriert, da die Organisationsstrukturen direkt im Datenmodell abgebildet werden [KR09a]. Auf diese Weise lassen sich sehr komplexe Funktionen direkt umsetzen. Hierzu gehören Zugriffsberechtigungen, die von Attributwerten im Datenmodell abhängen. So darf ein Benutzer beispielsweise nur dann auf ein bestimmtes Objekt zugreifen oder einen entsprechenden Prozessschritt ausführen, wenn ein bestimmtes Objektattribut einen vorgegeben Wert hat. Mittels der Organisationsstrukturen werden anstehende Aufgaben Benutzern zugeordnet und in Form von Arbeitslisten angezeigt. Allerdings betrachtet PHILharmonicFlows keine weiteren Aspekte bezüglich der Arbeitslistenverwaltung wie Delegation und Eskalation. Die Überwachung von Prozessen be-

schränkt sich auf die Behandlung von Ausnahmen sowie die Anzeige von Daten und laufenden Prozessinstanzen.

Die Integration existierender Anwendungen mittels Anwendungsbausteinen ist im Modell in Form sog. *Black-Box Activities* vorgesehen. Diese sind jeweils an ein Datenobjekt des entsprechenden Prozesses gebunden und werden damit zur Laufzeit aufgerufen. PHILharmonicFlows macht jedoch keinen expliziten Datenaustausch mit dem Anwendungsbaustein über explizite Parameter. Dementsprechend gibt es auch kein *Plug and Play*. Wie Datenobjekte haben Anwendungsbausteine ein Zustandsmodell. Dieses wird zur Laufzeit und bei der Modellierung unter anderem für Korrektheitsüberprüfungen verwendet. Die Korrektheitsüberprüfungen gewährleisten, dass zur Laufzeit keine Inkonsistenzen zwischen Prozess- und Datenzuständen auftreten. Sie gehen daher über die in Kapitel 2 vorgestellten Überprüfungen von ADEPT [Rei00] hinaus.

PHILharmonicFlows ist als allgemeines Konzept für eine umfassende Entwicklungsplattform gedacht. Dabei werden Informationssysteme direkt durch die Daten- und Prozessmodellierung mit PHILharmonicFlows erstellt. Die Entwicklungsplattform wird dadurch selbst zum Informationssystem, PHILharmonicFlows wird nicht in andere Informationssysteme integriert. Aus diesem Grund gibt es keine Programmierschnittstelle (API). Das entwickelte Konzept wurde in verschiedenen Anwendungsdomänen validiert, etwa im Personal- [Kün13] und Gesundheitswesen [CKR12b, CKR13].

10.3 Systeme aus der Praxis

Nachdem wir im vorherigen Abschnitt zahlreiche akademische Ansätze vorgestellt haben, gehen wir in diesem Kapitel auf Systeme aus der Praxis ein. Bei der Entwicklung dieser Systeme steht nicht die Machbarkeit ausgewählter Konzepte im Vordergrund sondern die möglichst umfassende Abdeckung funktionaler Anforderungen für den praktischen Einsatz. Wir gehen deshalb in diesem Abschnitt bei jedem System nur auf die wichtigsten erfüllten und nicht erfüllten Anforderungen ein. Im anschließenden Vergleich werden alle Anforderungen berücksichtigt.

Activiti [acta] ist ein quelloffenes System und eine Neuentwicklung von jBPM [jbp], einer PrMS-Erweiterung des JBoss-Anwendungsserver. Aufgrund der engen Verwandtschaft diskutieren wir hier nur das mittlerweile verbreitetere Activiti. Als letztes System betrachten wir das wohl umfassendste PrMS *IBM WebSphere* [ibm].

10.3.1 Activiti

Activiti [acta] ist ein quelloffenes, in Java implementiertes PrMS, das aufgrund seiner Historie große Ähnlichkeit mit jBPM hat (s. Abb. 10.8). Es ist für die Verwendung als Plattform entwickelt (FA 12).Die Prozessmodellierung erfolgt einerseits im *Activiti Modeler* von Signavio [sig]. Da dieser nicht alle Funktionen abdeckt, gibt es andererseits den *Activiti Designer* zum Vervollständigen von Prozessmodellen (FA 8). Dort kann auch direkt Java-Code in Form von Eclipse-Plugins [ecl] (Zusatzmodule für die Eclipse-Plattform) für die Integration eigener Anwendungslogik implementiert werden (FA 7). Auch HTML-Formulare werden dort erstellt. Diese werden direkt über die Benennung der (HTML-)Variablen an den Datenfluss angebunden (FA 10). Ebenso wie der entwickelte Anwendungscode liegen diese Formulare in Form von Eclipse-Plugins vor. Diese Zusatzmodule sind allerdings an den jeweiligen Prozess gebunden, so dass keine Wiederverwendung wie bei der Verwendung eines Aktivitätenrepository möglich

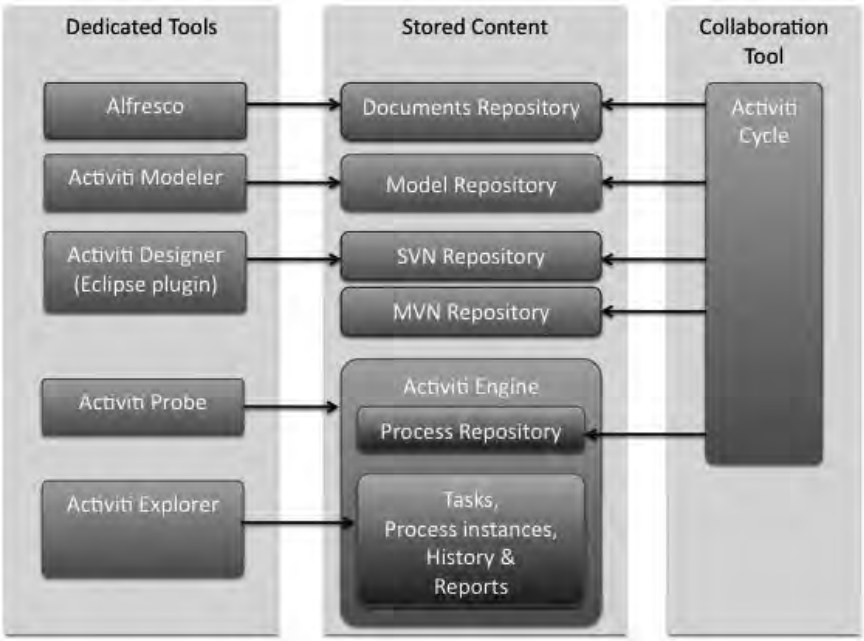


Abbildung 10.8: Architektur von Activiti [actb]

ist (FA 10). Sie müssen nach Ihrer Fertigstellung als Java-Archivdatei beim Activiti-Server abgelegt werden, der danach neu gestartet wird. Vordefinierte Anwendungsbausteine sind der Versand von E-Mails sowie die Ausführung von Skripten (JavaScript und Groovy) [Hal11].

Das Organisationsmodell von Activiti ist sehr eingeschränkt; es gibt nur Benutzer und nicht-hierarchische Gruppen. Bei der Prozessmodellierung gibt es keinerlei Fehlerprüfung (FA 8). Modellierte Prozesse können lediglich mittels JUnit-Tests und damit nicht interaktiv auf ihre erfolgreiche Ausführung getestet werden. Die Versionierung von Prozessvorlagen erfolgt im Editor auf Ebene der entsprechenden Dateien im Dateisystem (FA 11). Wird eine neue Prozessvorlage in den Server eingespielt – mittels Kopieren der Datei – so ist auch hier ein Serverneustart notwendig.

Zur Laufzeit können Prozessinstanzen und Prozessschritte über *Activiti Probe* abgerufen werden. Dort werden auch die aktuellen Werte der Datenelemente angezeigt (FA 2). Für Arbeitslisten werden im *Activiti Explorer* (FA 4) angezeigt. Daneben kann Activiti Ad-hoc-Prozesse ausführen. Solche Prozesse werden dynamisch vom Endbenutzer modelliert und danach gleich ausgeführt (FA 9.1).

10.3.2 *IBM WebSphere*

WebSphere ist eine Produktlinie von IBM, die unter anderem auch ein PrMS umfasst. Das PrMS beruht zum Teil auf anderen Produkten der WebSphere-Produktlinie, etwa *WebSphere MQ*[1]. Skalierbarkeit ist inhärent in der WebSphere-Produktlinie und damit auch in der PrMS-Architektur berücksichtigt. Das PrMS in *IBM WebSphere*, der *IBM WebSphere Process Server* besteht selbst aus mehreren Anwendungsprogrammen, die zusammen genommen den ganzen Prozesslebenszyklus abdecken [DBJ+13, AGSBC+12]. So erfolgen, ähnlich wie bei

[1] Wir betrachten die WebSphere-Produktlinie Version 7. Ab Version 8 ist das PrMS mit dem von Lombardi und der *WebSphere Lombardi Edition* zum *IBM Business Process Manager* vereinigt.

Activiti, auch bei *IBM WebSphere* die Prozesserfassung und technische Prozessmodellierung in verschiedenen Anwendungsprogrammen. Für die Prozesserfassung und abstrakte Modellierung in BPMN (*Business Process Model and Notation*) gibt es den *WebSphere Business Modeler*. Dort werden Prozessschritte, Kontrollfluss, Datenfluss, Rollen (FA 3), Kosten und Zeiten sowie Geschäftskennzahlen (KPI) festgelegt. Innerhalb des *WebSphere Business Modeler* gibt es einen graphischen Editor für die Modellierung von Datenstrukturen (FA 10). Für diese Datenstrukturen können automatisch Formulare für Prozessschritte generiert werden. Das Layout der Formulare kann direkt im *WebSphere Business Modeler* Layout geändert werden (FA 7), weitere Änderungen sind jedoch nicht möglich. Außerdem können Webdienste direkt aus UDDI oder entsprechenden WSDL-/XSD-Dateien an Prozessschritten hinterlegt werden. Auch Geschäftsregeln können in Form einfacher Bedingungen erstellt werden. Diese werden jedoch erst zur Laufzeit auf Fehler überprüft (FA 7, FA 8). Für komplexere Regeln gibt es das separate ILOG-Anwendungsprogramm.

Im *WebSphere Business Modeler* können Prozessmodelle von mehreren Benutzern gleichzeitig bearbeitet werden. Über CVS (*Concurrent Versions System*) werden die Prozessmodelle versioniert (FA 11). Zusätzlich können sie bezüglich einiger Aspekte validiert werden (FA 8), etwa dass alle Pfade zu einem Endknoten führen. Daneben lassen sich Prozesse zusammen mit der Ressourcenauslastung umfangreich simulieren und analysieren. Hieraus wiederum lassen sich Berichte erzeugen. Dies ermöglicht die Prozessoptimierung bereits bei der Prozesserfassung. Ebenso können Daten von ausgeführten Prozessen zum entsprechenden Prozessmodell in den *WebSphere Business Modeler* importiert und wie nach einer Simulation analysiert werden. Auf diese Weise ist eine nachträgliche Prozessoptimierung möglich.

Die technische Prozessmodellierung erfolgt im *WebSphere Integration Developer*. Hier werden Anwendungsbausteine hinterlegt und der Prozess um technische Aspekte erweitert, etwa die Ausnahmebehandlung. Für die Anwendungsintegration bietet der *WebSphere Integration Developer* zahlreiche Möglichkeiten: Webdienste, Session-EJB (*Enterprise JavaBeans*), JavaBean, HTTP, diverse Nachrichtenbusse, verschiedene vorgegebene Benutzerschnittstellen (z. B. Portlets, JSF (*JavaServer Faces*) und *Lotus Forms*) bis hin zur Verwendung eigener Adapter für beliebige Anwendungen (FA 7). Diese Anwendungsbausteine können teilweise mittels Ziehen und Ablegen mit der Maus (*Drag and Drop*) erstellt werden, beispielsweise Webdienste aus einem UDDI-Verzeichnis. Im Datenfluss können komplexe Datentypen verwendet werden. Passen die verwendeten Datentypen nicht direkt zur Schnittstelle eines Anwendungsbausteins, kann mit einem graphischen Editor eine entsprechende Datentransformation spezifiziert werden. Der Datenfluss wird jedoch nicht auf Korrektheit überprüft (FA 10). Die technisch modellierten Prozesse können im *WebSphere Integration Developer* zu Testzwecken ausgeführt werden. Dies ähnelt der Ausführung zu Debugzwecken in einer integrierten Umgebung für die Softwareentwicklung (s. Abb. 10.9).

Dynamik zur Laufzeit ist beim *IBM WebSphere Process Server* auf verschiedene Arten möglich. Für IBM fallen hierunter beispielsweise auch Bearbeiterzuordnungen, die von Prozessdaten abhängen. So lässt sich unter anderem das 4-Augen-Prinzip umsetzen (FA 3). Prozessschritte können delegiert und wieder vorgelegt werden. Das Überspringen von Prozessschritten ist möglich, allerdings bleiben dann ggf. Daten unversorgt. Das heißt, das PrMS übernimmt im Kontext solcher dynamischer Änderungen keine Korrektheitsgarantien. Außerdem können Vorgängerschritte wiederholt werden. Daneben können Benutzer ähnlich wie bei Activiti einfache Ad-hoc-Prozesse modellieren und ausführen (FA 9.1). Bei bestimmten Anwendungsbausteinen, etwa Webdiensten [ACKM04], kann der aufgerufene Dienst (seine URL) zur Laufzeit geändert werden. Ebenso können Geschäftsregeln jederzeit geändert werden. In beiden Fällen überprüft

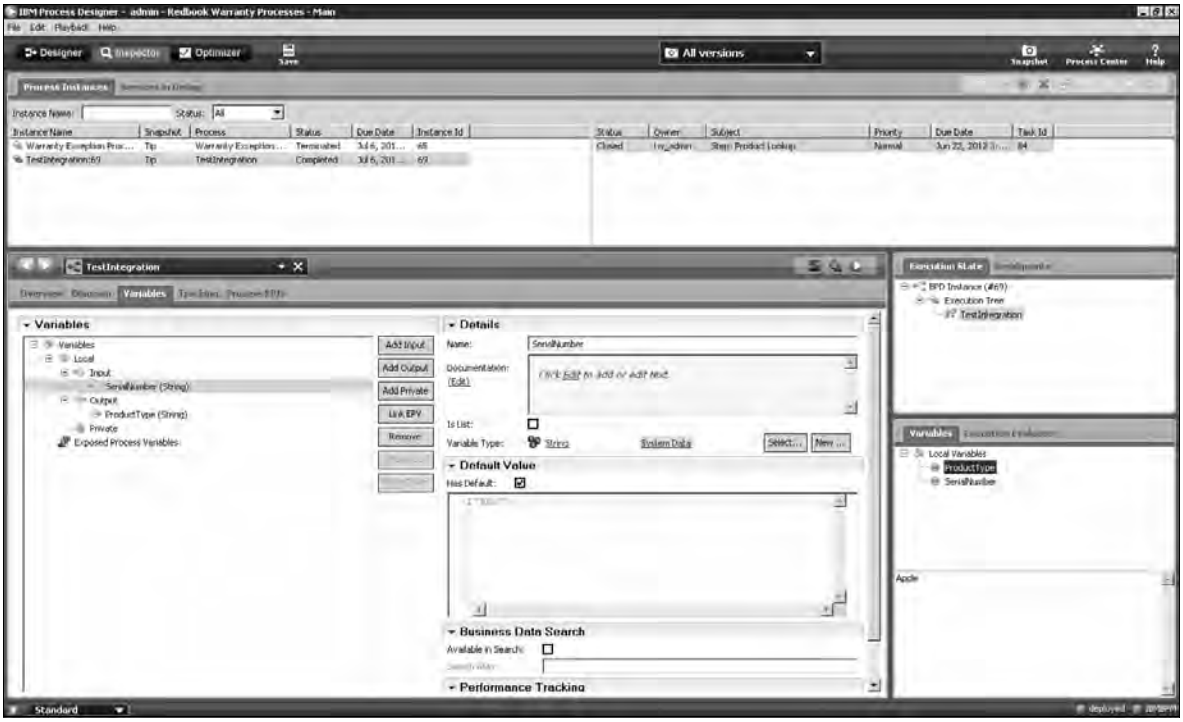


Abbildung 10.9: Testweise Ausführung im *Process Designer* (Vorgänger von *WebSphere Integration Developer*) [AGSBC+12]

das System die Änderung aber nicht auf Korrektheit, beispielsweise ob der neue Webdienst dieselbe Schnittstelle und das gewünschte Verhalten hat.

Zur Laufzeit kann sich ein Benutzer die von ihm gestarteten Prozessinstanzen anschauen. Dabei werden allerdings nur die Benutzerschritte angezeigt; automatisch ausgeführte Schritte sind ausgeblendet. Die eigentliche Monitoranwendung ist separat. Diese kann auch noch Daten anderer WebSphere-Produkte anzeigen. Dazu werden die relevanten Ereignisse unter anderem an die Monitoranwendung versandt. Beispielsweise teilt das PrMS auf diese Weise mit, wann welcher Prozessschritt gestartet und beendet wird. Auch Informationen über die bei der Modellierung spezifizierten Geschäftskennzahlen (KPI) werden der Monitoranwendung auf diese Weise mitgeteilt. Dort können diese Informationen umfangreich analysiert werden. Ferner gibt es Anzeige- und Benachrichtigungsmöglichkeiten beim Erreichen bestimmter Grenzwerte (FA 2, FA 5).

Neben der Implementierung eigener Adapter für Anwendungen können die Klientenprogramme mittels Portaltechnik integriert und angepasst aber auch durch selbst implementierte Klientenprogramme ersetzt werden (FA 12).

10.4 Zusammenfassung

Tabelle 10.2 vergleicht die vorgestellten akademische Ansätze und PrMS aus der Praxis bezüglich der eingeführten Kriterien (s. Abschnitt 10.1). Die Bewertungen sind wie folgt: - (Anforderung nicht erfüllt), O (Anforderung grundlegend erfüllt), + (Anforderung umfas-

Tabelle 10.2: Vergleich verschiedener PrMS

Ansatz	FA 2	FA 3	FA 4	FA 5	FA 7	FA 8	FA 9.1	FA 9.2	FA 10	FA 11	FA 12	NA 1
AristaFlow	O	+	+	O	+	+	+	+	+	+	+	+
Chautauqua	-	+	O	-	-	O	-		O			-
MILANO		(+)	O	-		O	-	O				-
YAWL	O	+	O	O	+	+	O		O	O	+	
MOBILE		O	O		-	-					+	O
DYNAMITE			O	O	-	O	O	-	O	-		-
MOKASSIN					+		+	+		+		
WASA		+	O	O	O	+	+	+		+	+	+
PHILharmonicFlows		+	O	-	(O)	+						O
Activiti	-	-	O		O	O	(O)		O	+	+	
IBM Websphere	+	+	+	+	+	+	O		-	+		+

Bewertung:
+: gut
O: neutral
-: schlecht
( ): Funktion eingeschränkt
□: Funktion nicht vorhanden

send erfüllt). Bewertungen in Klammern gelten nur eingeschränkt, d. h. nur für Teile der Anforderung.

Die Tabelle verdeutlicht, dass die akademischen Ansätze und Forschungsprototypen meist nur einige ausgewählte Anforderungen abdecken. Dies liegt daran, dass diese oft nur die Umsetzbarkeit ausgewählter wissenschaftlicher Konzepte zeigen wollen. Dagegen müssen Systeme aus der Praxis viele Anforderungen abdecken, um überhaupt für den Produktiveinsatz geeignet zu sein. Lediglich WASA (s. Abschnitt 10.2.7) setzt fast alle Anforderungen um. Dies ist für eine sinnvolle wissenschaftliche Betrachtung der Architektur von PrMS aber auch unbedingt erforderlich.

Die Systeme aus der Praxis sind aufgrund ihres Funktionsumfangs für den Produktivbetrieb geeignet. Allerdings ist bei Activiti viel Aufwand für die Integration notwendig. Sowohl Activiti als auch *IBM WebSphere* zeigen außerdem Defizite in Bezug auf Korrektheitsprüfungen der Modelle (FA 10) und bei der Flexibilität hinsichtlich Schemaevolution (FA 9.2). Instanzspezifische Änderungen (FA 9.1) sind meist in grundlegender Form möglich, allerdings wird dabei nicht in ausreichendem Maße auf Korrektheit geachtet. Der Schwerpunkt dieser Arbeit, Flexibilität im Kontext von Skalierbarkeit, wird lediglich ansatzweise bei WASA betrachtet, alle anderen Systeme betrachten höchstens einen der beiden Aspekte.

11 Zusammenfassung und Ausblick

Prozeßmanagementsysteme besitzen ein ähnlich großes Potential im Hinblick auf Produktivitätssteigerungen bei der Softwareentwicklung wie DBMS. DBMS machen durch die Abstraktion der Daten die jeweilige Softwareimplementierung unabhängig von der physischen Datenrepräsentation. Zusätzlich bieten sie nützliche Funktionen, die über einen reinen Datenzugriff hinausgehen, etwa im Hinblick auf Synchronisation und Mehrbenutzerzugriff. PrMS erlauben dasselbe für Kontroll- und Datenflüsse komplexer verteilter Informationssysteme. Durch die Loslösung von Prozessaspekten aus den Implementierungen ändert sich die Softwareentwicklung signifikant: Monolithische Systeme werden durch Dienste ersetzt, wie dies im Kontext von serviceorientierten Architekturen (SOA) [Erl05] propagiert wird. Mittels eines PrMS können Anwendungsbausteine leicht neu kombiniert und so neue Geschäftsprozesse realisiert werden. Hierzu reicht es aus, die Prozesse zu modellieren; es fällt kein weiterer Implementierungsaufwand an. Auf diese Weise werden die Aufgaben von Fach- und IT-Abteilungen in Unternehmen klar voneinander getrennt. Fachabteilungen müssen nicht mehr aufwendig Spezifikationen für ein Gesamtsystem formulieren. Vielmehr reicht es aus, die Spezifikation überschaubarer Softwaremodule (Anwendungsbausteine) vorzunehmen. Aus diesen wiederum können die Fachabteilungen die benötigten Prozessimplementierungen selbst zusammenstellen. Im Gegensatz dazu implementieren IT-Abteilungen überschaubare Softwaremodule. Hierdurch verringert sich das Risiko, dass die Softwaremodule nicht den Wünschen der Fachabteilungen entsprechen. Somit bilden PrMS eine wichtige Ergänzung von SOA [Man, Buc12].

Die Flexibilität des ADEPT-Metamodells [Rin04, Rei00, RRMD09] ermöglicht es darüber hinaus, spontan auf Ausnahmefälle zu reagieren und den Prozessablauf dynamisch anzupassen. Das entspricht vom Prinzip her der dynamischen Änderung von implementiertem Programmcode während der Ausführungszeit. Ähnliches gilt in Bezug auf Schemaevolution. Die durch sie bewirkten Änderungen sind dauerhaft, was wiederum kurze Entwicklungszyklen für Prozesse ermöglicht. So kann die Rückmeldung von Benutzern zu einem produktiven Prozess schnell in Prozessänderungen umgesetzt und sogar in bereits laufenden Prozessinstanzen berücksichtigt werden. Dies verkürzt Entwicklungszyklen signifikant und ist damit eine ideale Ergänzung zu agilen Entwicklungsmethoden [BBvB+].

Die vorliegende Arbeit leistet einen grundlegenden Beitrag zu Architektur- und Implementierungskonzepten von PrMS. Dadurch sollen diese einen Schritt näher an den Entwicklungsstand

von DBMS gebracht und somit ihre Eignung für den breiten praktischen Einsatz verbessert werden. Der Arbeit liegt ein ganzheitlicher und umfassender Ansatz zugrunde und somit die explizite Berücksichtigung von Abhängigkeiten der betrachteten Aspekte sowie deren gegenseitige Beeinflussung :

- Wichtigstes Kriterium für den praktischen Einsatz ist die Skalierbarkeit und Leistungsfähigkeit des PrMS. Hierzu werden Lösungskonzepte für die Repräsentation von Prozessvorlagen und -instanzen innerhalb eines PrMS vorgestellt und diskutiert. Dies umfasst sowohl unveränderte als auch veränderte Prozessinstanzen. Auch für Schemaevolution wird ein umfangreiches Lösungskonzept diskutiert. Insbesondere werden die verschiedenen Aspekte nicht isoliert, sondern im Zusammenspiel miteinander betrachtet. Dies ist unabdingbar, da die Lösungskonzepte nicht voneinander unabhängig betrachtet werden können.

- Ergänzt werden die angestellten theoretischen Betrachtungen durch umfassende Simulationen der verschiedenen Lösungskonzepte. Dabei werden sowohl der Speicherbedarf als auch die Laufzeit von Funktionen gemessen, die für den praktischen Einsatz eines PrMS besonders wichtig sind. Beim Ein-/Auslagern der Datenrepräsentationen zwischen Primär- und Sekundärspeicher gibt es zudem Variationen in den Repräsentationen. Die vorgestellten Simulationen ermöglichen ein tiefergehendes Verständnis in Bezug auf Speicherbedarf und Laufzeit. Insbesondere lassen sich dadurch die Auswirkungen der Repräsentationen auf Skalierbarkeit und Leistungsfähigkeit messen.

- Neben innovativen Implementierungskonzepten wird in der Arbeit eine umfassende Gesamtarchitektur eines adaptiven PrMS entwickelt. Insbesondere setzt diese alle diskutierten funktionalen Anforderungen um. Die Gesamtarchitektur, die Datenmodelle und die einzelnen Dienste der Architektur werden darin im Detail vorgestellt. Ferner kann die Architektur direkt als Programmierschnittstelle (API) verwendet werden; d. h. das PrMS kann, ähnlich wie ein DBMS, in andere Systeme integriert werden. Darauf bauen auch die Anwendungen des PrMS selbst auf, durch die das PrMS eigenständig genutzt werden kann. Neben den umfangreichen funktionalen Anforderungen erfüllt die vorgestellte Architektur auch folgende nicht-funktionalen Anforderungen: Skalierbarkeit und Leistungsfähigkeit, Robustheit, Erweiterbarkeit, Einfachheit, Unabhängigkeit und Modularität, Wiederverwendung und Redundanzvermeidung, Wartbarkeit sowie Veränderbarkeit und Flexibilität.

- Die vorgestellte Architektur bildet das Fundament der *AristaFlow BPM Suite* [DRRM+09, RDRM+09b, LKRD10, LRD10a, GJA+07]. Diese umfassende Implementierung setzt die behandelten Funktionen in die Praxis um und demonstriert dabei sowohl die Machbarkeit der Architektur als auch der entwickelten Konzepte.

Ausblick

Neben der Validierung der in dieser Arbeit diskutierten Aspekte zeigen sich im praktischen Einsatz verschiedene Möglichkeiten zur Verbesserung und Erweiterung des Systems, der Architektur und der Konzepte. Einige wurden bereits in der Arbeit berücksichtigt, einige gehen jedoch weit darüber hinaus und sind somit Themen für zukünftige Arbeiten. Hierzu gehören:

- Erweiterung des Prozessmetamodells hinsichtlich der Modellierung von Kontroll- und Datenfluss (s. [Wol08, For09, RR06] für Details): Durch flexible Endknoten, die einen (Sub-)Prozess an beliebigen Stellen beenden, nicht-exklusive Verzweigungen sowie strukturierte Datentypen kann die Modellierung vereinfacht werden. Dies wiederum erfordert Anpassungen der Korrektheitskriterien. Weitergehend können die Korrektheitskriterien bezüglich des Datenflusses durch die Erkennung von Abhängigkeiten zwischen verschiedenen Verzweigungsentscheidungen abgeschwächt werden. So kann erkannt werden, dass ein Datenelement in einer alternativen Verzweigung obligat gelesen werden kann, weil es in einem Zweig einer vorausgehenden alternativen Verzweigung mit derselben Verzweigungsentscheidung obligat geschrieben wird. Eine weitere sinnvolle Erweiterung ist das in der Praxis häufig anzutreffende *konsumierende Lesen*. Dabei wird ein Datenwert nach dem Lesen ungültig und darf in nachfolgenden Prozessschritten nicht mehr verwendet werden.

- Eine Zustandsänderung eines Prozessschritts ist aus Sicht des Kontrollflusses die kleinste atomare Einheit. Zu dieser Zustandsänderung des Prozessschritts gehören ggf. noch das Lesen und Schreiben der jeweiligen Datenelemente. Aus semantischer Sicht kann es sinnvoll sein, größere Einheiten atomar aufzufassen und zusammengehörige Prozessschritte in einer Transaktion zu klammern [GMS87, WR92, Ley95]. Bei Abbruch der Transaktion wird ggf. eine Kompensation notwendig, etwa in Form eines separaten Kompensationsprozesses.

- Aufgrund der Zustandsänderung als kleinste atomare Einheit, gibt es keine Möglichkeit, adäquat mit Datenströmen umzugehen [CM12]. Diese ermöglichen z. B. die Integration von Sensordaten in Prozesse. Eng verwandt mit Datenströmen sind externe Ereignisse. Während Ereignisse einmalig sind, lassen sich Datenströme als fortlaufende Ereignisse betrachten. Ereignisse können momentan entweder oberhalb der Programmierschnittstelle verarbeitet werden, beispielsweise indem ein Ereignis eine bestimmte Prozessinstanz startet, oder ein Prozessschritt kann explizit auf ein definiertes Ereignis warten. Es gibt jedoch keine Möglichkeit, nur eine bestimmte Zeit auf das Ereignis zu warten oder das Ereignis bereits vor der Ausführung des Prozessschritts zu erkennen und nicht zu warten.

- Durch Arbeitslisten pro Mitarbeiter-Stelle-Paar können Arbeitsschritte feingranular verteilt werden. Für den praktischen Einsatz sind auch zusammengefasste Arbeitslisten sinnvoll, etwa Arbeitslisten für Organisationseinheiten, -gruppen oder andere Entitäten des Organisationsmodells. Ein Anwendungsfall ist das Zusammenarbeiten von Benutzern über eine solche Arbeitsliste. Hier dürfen nur Einträge enthalten sein, die der entsprechenden Entität zugeordnet sind, etwa einer Projektgruppe. Ein anderer Anwendungsfall ist die Übersicht eines Vorgesetzten über Arbeit der ihm untergebenen Mitarbeiter. In diesem Fall enthält die aggregierte Arbeitsliste alle Einträge, die die Mitarbeiter dieser Organisationseinheit haben, die aber nicht notwendigerweise auch dieser Organisationseinheit zugewiesen sind. Hierfür ist es notwendig, dass die Arbeitslistenverwaltung mehr Informationen zu den Bearbeiterzuordnungsregeln enthält. Außerdem fallen hier große Datenmengen an, die laufzeit- und speichereffizient repräsentiert werden müssen.

- Die Zuordnung von Aufgaben zu Benutzern mittels Bearbeiterzuordnungsregeln deckt Sicherheitsaspekte bezüglich der Ausführung von Prozessschritten explizit ab. Implizit definieren sie dadurch auch Berechtigungen bezüglich Lesen und Schreiben von Daten.

Gerade auf Daten sind jedoch oft umfassende und explizite Zugriffsberechtigungen notwendig [KR09a]. Weitergehend können diese Zugriffsberechtigungen sogar vom Inhalt der Daten abhängen, etwa eine Person darf ein Datum nur lesen, wenn der Wert unterhalb einer bestimmten Schwelle liegt. Sind solche Zugriffsberechtigungen vorhanden, müssen sie bei den Bearbeiterzuordnungsregeln berücksichtigt werden. Dadurch dürfen keine Inkonsistenzen entstehen, etwa dass der Bearbeiter eines Prozessschritts ein von diesem Schritt gelesenes Datum nicht sehen darf [PDA08]. Entweder würde in diesem Fall die Zugriffsberechtigung verletzt oder die Ausführung des Prozessschritt blockiert werden. Beides muss das PrMS verhindern. Aufgrund seiner Komplexität hat dies weitreichende Auswirkungen sowohl auf die Prozessmodellierung als auch die Prozessausführung. So lassen sich Zugriffsberechtigungen, die von Datenwerten abhängen, erst dann auswerten, wenn die Werte feststehen.
Weitergehende Sicherheitsaspekte ergeben sich durch Funktionen zur Steigerung der Flexibilität in Form instanzspezifischer Änderungen und Schemaevolution [WRWR05a, PDA08]. Hierbei ist relevant, welche Bearbeiter Änderungen durchführen dürfen, welche Änderungen dies sind und in welchem Prozesskontext sie durchgeführt werden. Beispielsweise sind nur Einfügungen zulässig und dies auch nur von bestimmten Aktivitäten und an bestimmten Stellen des Prozessgraphen. Insbesondere mit Zugriffsberechtigungen auf Daten erhöht sich die Komplexität um ein Vielfaches. Dabei sind bestimmte Änderungen nur zulässig, wenn ein Datum einen bestimmten Wert nicht überschreitet. Auch bei instanzspezifischen Änderungen dürfen keine Zugriffsberechtigungen verletzt werden und damit auch keine Inkonsistenzen bezüglich Berechtigungen auftreten.

- Sehr ähnlich zu Zugriffsberechtigungen sind globale Integritätsbedingungen (*Compliance*) [LKRM+10, Ly13]. Dies sind Regeln, die für die Ausführung von Prozessschritten definiert sind. Dazu gehören etwa Reihenfolgeabhängigkeiten oder zeitliche Bedingungen, die zwischen Prozessschritten einzuhalten sind [Ly13, Hei00]. Die Integritätsbedingungen können, ebenso wie die Zugriffsberechtigungen, von Daten abhängen. Beispielsweise lässt sich so eine Unverträglichkeit zweier Medikamente spezifizieren, die bei der Modellierung, Ausführung und Änderung medizinischer Prozesse berücksichtigt werden muss. Dementsprechend haben die Integritätsbedingungen ähnlich weitgehende Anforderungen an Modellierung und Ausführung wie Zugriffsberechtigungen. Dementsprechend sind sie ähnlich komplex.

- Großes Potential hat die Berücksichtigung zeitlicher Aspekte [LWR10, LWR14, LKR13, LPCR13] und Ressourcen in PrMS. Hiermit lassen sich Prozesse hinsichtlich des Zeit- und Ressourcenbedarfs planen. Die Neuplanung während der Prozessausführung ermöglicht darüber hinaus die vorausschauende Erkennung von Zeit- und Ressourcenengpässen.

- Durch den Einsatz von PrMS für betriebsinterne Prozesse wächst der Wunsch, PrMS auch im Kontext unternehmensübergreifender Prozesse zu nutzen [RWR06, KPR12, KRP+13, KRFRM13, CKCR13, FRMR12]. Durch die enge Verzahnung von externen und internen Prozessen kann die Zusammenarbeit zwischen Unternehmen automatisiert und signifikant verbessert werden. Grundlage für betriebsübergreifende Prozesse kann beispielsweise die verteilte Ausführung von Prozessen sein [Bau01, BRD03, RB07]. Zusätzlich müssen Sicherheitsaspekte berücksichtigt werden, so dass betriebsinterne Daten oder interne Prozessabläufe nicht ohne Grund von außerhalb des Unternehmens eingesehen werden können.

- Instanzspezifische Änderungen können durch zahlreiche Mittel vereinfacht werden. Naheliegend ist die Einschränkung der Änderungen auf wenige vordefinierte Fälle, vergleichbar der Einschränkung durch Sicherheitsaspekte (s. o.). Dies geht aber einher mit der Einschränkung der Flexibilität. Ein besseres Verfahren ist die Wiederverwendung instanzspezifischer Änderungen aufgrund von Informationen aus dem Anwendungskontext. Dies leistet beispielsweise *Fallbasiertes Schließen* (*Case-Based Reasoning*) [WRWR05b, WRW06, WRWRM09, GRvdA08]. Dabei gibt es zu vordefinierten Änderungstransaktionen eine Menge an Fragen und Antworten. Tritt ein Problem auf, beantwortet der Benutzer die entsprechenden Fragen. Seine Antworten werden mit den vorhandenen Fällen verglichen, und es werden vom PrMS die Änderungen angeboten, die in ähnlichen Fällen geholfen haben. Hierdurch werden die technische und die operationale Sicht auf instanzspezifische Änderungen vor dem Benutzer verborgen. Fallbasiertes Schließen erfordert die Verwaltung zusätzlicher Daten zu instanzspezifischen Änderungen, es kann aber weitgehend auf der vorgestellten Schnittstelle eines adaptiven PrMS aufgesetzt werden.

- Durch Fortschritte in der technischen Entwicklung werden Geräte immer kleiner und leistungsfähiger. Der Einsatz von Prozessmanagement auf mobilen Geräten hat großes Potential, etwa im medizinischen Bereich [PMR13, PTKR10, PLRH12, PMLR14]. So können durch den Einsatz von mobilen Geräten in Verbindung mit Prozessmanagement sehr leicht Aufgaben (z. B. Untersuchungen) festgelegt und dem verantwortlichen Personal zugeordnet werden. Dies führt zu erheblichen Kosteneinsparungen bei gleichzeitiger Verbesserung der Qualität. Besonderheiten auf mobilen Geräten gegenüber normalen PrMS-Klientenprogrammen ergeben sich zum einen durch die Größe der Anzeige. Formulare für die Prozessschritte müssen dies ebenso berücksichtigen wie weitergehende Funktionen des PrMS (z. B. Prozessmodellierung oder -änderung). Zum anderen muss bei mobilen Geräten mit Einschränkungen der Verbindung zum PrMS gerechnet werden. So kann eine Verbindung abbrechen oder von vornherein nicht möglich sein. Dazu müssen die notwendigen Daten auf das Gerät übertragen und anschließend synchronisiert werden. Hierdurch ergeben sich zahlreiche zusätzliche Anforderungen an die Architektur bis hin zur Implementierung leichtgewichtiger Dienste für die Ausführung von Prozessen direkt auf dem mobilen Gerät.

Literaturverzeichnis

[AAD+04] H. Acker, C. Atkinson, P. Dadam, S. Rinderle und M. Reichert: *Aspekte der komponentenorientierten Entwicklung adaptiver prozessorientierter Unternehmenssoftware.* In: *Proceedings 1. Verbundtagung Architekturen, Komponenten, Anwendungen (AKA 2004)*, Nummer P-57 in *Lecture Notes in Informatics (LNI)*, Seiten 7–24, Augsburg, 2004. Koellen-Verlag.

[ACKM04] G. Alonso, F. Casati, H. Kuno und V. Machiraju: *Web Services: Concepts, Architectures and Applications.* Springer, 2004.

[acta] *Activiti.* http://www.activiti.org/. Abgerufen am 28.3.2014.

[actb] *Activiti-Architektur.* http://activiti.org/components.html. Abgerufen am 28.3.2014.

[AdM00a] A. Agostini und G. de Michelis: *Improving Flexibility of workflow Management Systems.* In: *Business Process Management*, Band 1806 der Reihe *LNCS*, Seiten 218–234. Springer, 2000.

[AdM00b] A. Agostini und G. de Michelis: *A Light Workflow Management System Using Simple Process Models.* Computer Supported Cooperative Work (CSCW), 9(3-4):335–363, 2000.

[AGSBC+12] A. Abdel-Gayed, K. Singh Bhogal, D. Carr, R. Davies, A.P. Dutta, M. Correia Lima, A. Martinez Hernandez Magro, Y. Musashi, M. Norris, F. Pistorius und M. Keen: *Implementing an Advanced Application Using Processes, Rules, Events, and Reports.* IBM Redbooks, 2012.

[App87] A.W. Appel: *Garbage collection can be faster than stack allocation.* Information Processing Letters, 25(4):275–279, 1987.

[AtHEvdA06] M. Adams, A.H.M. ter Hofstede, D. Edmond und W.M.P. van der Aalst: *Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows.* In: *On The Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Band 4275 der Reihe *LNCS*, Seiten 291–308, Montpellier, Oktober 2006. Springer.

[ATW+13] C. Ayora, V. Torres, B. Weber, M. Reichert und V. Pelechano: *Enhancing Modeling and Change Support for Process Families through Change Patterns.* In: *Proceedings Enterprise, Business-Process, and Information Systems Modelling: 14th International Workshop BPMDS*, Band 147 der Reihe *LNBIP*, Seiten 246–260, Valencia, Juni 2013. Springer.

[Bau96] B. Baumgarten: *Petri-Netze – Grundlagen und Anwendungen.* Spektrum Akademischer Verlag, 1996.

[Bau01] T. Bauer: *Effiziente Realisierung unternehmensweiter Workflow-Management-Systeme.* Dissertation, Universität Ulm, Februar 2001.

[BBKK02] S. Bassil, M. Benyoucef, R.K. Keller und P. Kropf: *Addressing Dynamism in E-negotiations by Workflow Management Systems.* In: *13th International Workshop on Database and Expert Systems Applications (DEXA)*, Seiten 655–659, Aix-en-Provence, 2002.

[BBvB+] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland und D. Thomas: *Manifesto for Agile Software Development.* http://agilemanifesto.org/. Abgerufen am 28.3.2014.

[Ben08] R. Bentele: *Konzepte, Technologien und Realisierungsvariangen eines webbasierten, intelligenten Workflow-Klienten.* Diplomarbeit, Universität Ulm, Ulm, Oktober 2008.

[Ber05] M. Berroth: *Konzeption und Entwurf einer Komponente für Organisationsmodelle.* Diplomarbeit, Universität Ulm, Ulm, Juni 2005.

[BG04] A. Bauer und H. Günzel: *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung.* dpunkt, 2004.

[BHG87] P.A. Bernstein, V. Hadzilacos und N. Goodman: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BKK04] S. Bassil, R.K. Keller und P. Kropf: *A Workflow-Oriented System Architecture for the Management of Container Transportation.* In: *Business Process Management*, Band 3080 der Reihe *LNCS*, Seiten 116–131, Potsdam, Juni 2004. Springer.

[Bla96] R. Blaser: *Konfiguration verteilter Anwendungen aus vorgefertigten Programmbausteinen.* Diplomarbeit, Universität Ulm, Ulm, 1996.

[BM72] R. Bayer und E.M. McCreight: *Organization and Maintenance of Large Ordered Indexes.* Acta Informatica, 1:173–189, 1972.

[BRD03] T. Bauer, M. Reichert und P. Dadam: *Intra-subnet Load Balancing in Distributed Workflow Management Systems.* International Journal of Cooperative Information Systems (IJCIS), 12(3):295–323, September 2003.

[Buc12] S. Buchwald: *Erhöhung der Durchgängigkeit und Flexibilität prozessorientierter Applikationen mittels Service-Orientierung.* Dissertation, Universität Ulm, Oktober 2012.

[CCPP98] F. Casati, S. Ceri, B. Pernici und G. Pozzi: *Workflow Evolution.* Data & Knowledge Engineering, 24(3):211–238, 1998.

[CKCR13] S. Catalkaya, D. Knuplesch, C. Chiao und M. Reichert: *Enriching Business Process Models with Decision Rules.* In: *International Workshop on Decision Mining & Modeling for Business Processes (DeMiMoP 13)*, Seiten 344–362, Bejing, August 2013. Springer.

[CKR12a] C.M. Chiao, V. Künzle und M. Reichert: *Schema Evolution in Object and Process-Aware Information Systems: Issues and Challenges.* In: *Business Process Management Workshops*, Band 132 der Reihe *LNBIP*, Seiten 328–339, Tallinn, September 2012. Springer.

[CKR12b] C.M. Chiao, V. Künzle und M. Reichert: *Towards Object-aware Process Support in Healthcare Information Systems.* In: *4th International Conference on eHealth, Telemedicine, and Social Medicine (eTELEMED 2012)*, Seiten 227–236. IARIA, Februar 2012.

[CKR13] C.M. Chiao, V. Künzle und M. Reichert: *Object-aware Process Support in Healthcare Information Systems: Requirements, Conceptual Framework and Examples.* International Journal on Advances in Life Sciences, 5(1&2):11–26, Juli 2013.

[CLRS09] T.H. Cormen, C.E. Leiserson, R.L. Rivest und C. Stein: *Introduction to Algorithms.* MIT Press, 3 Auflage, September 2009.

[CM12] G. Cugola und A. Margara: *Processing flows of information: From data stream to complex event processing.* ACM Computing Surveys (CSUR), 44(3), Juni 2012.

[Cod70] E.F. Codd: *A relational model of data for large shared data banks.* Communications of the ACM, 13(6):377–387, Juni 1970.

[CP10] C. Combi und R. Posenato: *Towards Temporal Controllabilities for Workflow Schemata.* In: *17th International Symposium on Temporal Representation and Reasoning (TIME 2010)*, Seiten 129–136, Paris, September 2010. IEEE.

[Dad96] P. Dadam: *Verteilte Datenbanken und Client/Server-Systeme: Grundlagen, Konzepte und Realisierungsformen.* Springer, 1996.

[Dad97] P. Dadam: *Business Information Systems: Trends and Technological Challenges.* In: *Proceedings of the 1st International Conference on Business Information Systems (BIS '97)*, Seiten 509–524, Poznan, April 1997.

[Dad13] P. Dadam: *Skript zur Vorlesung Informationssysteme.* http://www.uni-ulm.de/in/iui-dbis/lehre/ss-2013/informationssysteme.html, Sommersemester 2013. Abgerufen am 3.9.2013.

[DBJ+13] L. Dyer, F. Bento, S. Jones, S. Leibowitz und J. Michalec: *Empowering your Ad Hoc Business with IBM Business Process Manager.* IBM Redbooks, 2013.

[DKR+95] P. Dadam, K. Kuhn, M. Reichert, T. Beuter und M. Nathe: *ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender*

Assistenzsysteme in klinischen Anwendungsumgebungen. In: *Proceedings GI-Jahrestagung (GISI '95)*, Seiten 677–686, Zürich, September 1995.

[DR09] P. Dadam und M. Reichert: *The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support – Challenges and Achievements.* Computer Science – Research and Development, 23(2):81–97, Mai 2009.

[DRK00] P. Dadam, M. Reichert und K. Kuhn: *Clinical Workflows – The Killer Application for Process-oriented Information Systems?* In: *Proceedings of the 4th International Conference on Business Information Systems (BIS '00)*, Seiten 36–59, Poznan, April 2000. Springer.

[DRR+07] P. Dadam, M. Reichert, S. Rinderle, H. Acker, M. Jurisch, U. Kreher, K. Göser und M. Lauer: *ADEPT2 – Next Generation Process Management Technology.* In: *Proceedings Fourth Heidelberg Innovation Forum*, Heidelberg, April 2007.

[DRRM+09] P. Dadam, M. Reichert, S. Rinderle-Ma, K. Göser, U. Kreher und M. Jurisch: *Von ADEPT zur AristaFlow BPM Suite – Eine Vision wird Realität: „Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen.* EMISA-Forum, 29(1):9–28, 2009.

[DRRM11] P. Dadam, M. Reichert und S. Rinderle-Ma: *Prozessmanagementsysteme: Nur ein wenig Flexibilität wird nicht reichen.* Informatik-Spektrum, 34(4):364–376, August 2011.

[ecl] *Eclipse.* http://www.eclipse.org/. Abgerufen am 28.3.2014.

[EM97] C. Ellis und C. Maltzahn: *The Chautauqua Workflow System.* In: *Proceedings 30th Hawaii International Conference on System Sciences*, Band 4, Seiten 427 – 436. IEEE Computer Society Press, Januar 1997.

[Erl05] T. Erl: *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall International, 2005.

[For09] A. Forschner: *Fortschrittliche Datenflusskonzepte für flexible Prozessmodelle.* Diplomarbeit, Universität Ulm, Ulm, Juni 2009.

[FRMR12] W. Fdhila, S. Rinderle-Ma und M. Reichert: *Change Propagation in Collaborative Processes Scenarios.* In: *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom '12)*, Seiten 452–461, Pittsburgh, Oktober 2012. IEEE Computer Society Press.

[GCC+04] D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal und M.-C. Shan: *Business Process Intelligence.* Computers in Industry, 53(3):321–343, 2004.

[GHJV95] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.

[GJA+07] K. Göser, M. Jurisch, H. Acker, U. Kreher, M. Lauer, S. Rinderle-Ma, M. Reichert und P. Dadam: *Next-generation Process Management with ADEPT2*. In: *Business Process Management 2007 Demo Proceedings*, Band 272 der Reihe *CEUR-WS.org Workshop Proceedings*, Seiten 3–6, Brisbane, September 2007. CEUR-WS.

[GJS+99] B. Gronemann, G. Joeris, S. Scheil, M. Steinfort und H. Wache: *Supporting cross-organizational engineering processes by distributed collaborative workflow management – The MOKASSIN approach*. In: *Proceedings of 2nd Symposium on Concurrent Mulitdisciplinary Engineering (CME '99)*, 1999.

[Glö09] B. Glöckle: *Qualitative und quantitative Evaluation von Varianten zur performanten Verwaltung von Prozessdaten in einem Hochleistungs-PMS*. Diplomarbeit, Universität Ulm, Ulm, Dezember 2009.

[GMB+81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu und I. Traiger: *The Recovery Manager of the System R Database Manager*. ACM Computing Surveys (CSUR), 13(2):223–242, 1981.

[GMS87] H. Garcia-Molina und K. Salem: *Sagas*. In: U. Dayal und I.L. Traiger (Herausgeber): *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, Band 16 der Reihe *SIGMOD '87*, Seiten 249–259. ACM, Dezember 1987.

[GMUW02] H. Garcia-Molina, J.D. Ullman und J.D. Widom: *Database Systems: The Complete Book*. Pearson, 2002.

[GOR10a] G. Grambow, R. Oberhauser und M. Reichert: *Employing Semantically Driven Adaptation for Amalgamating Software Quality Assurance with Process Management*. In: *2nd International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE '10)*, Seiten 58–67, Lissabon, November 2010. Xpert Publishing Services.

[GOR10b] G. Grambow, R. Oberhauser und M. Reichert: *Semantic Workflow Adaption in Support of Workflow Diversity*. In: *4th International Conference on Advances in Semantic Processing (SEMAPRO '10)*, Seiten 158–165, Florence, Oktober 2010. Xpert Publishing Services.

[GOR11a] G. Grambow, R. Oberhauser und M. Reichert: *Contextual Generation of Declarative Workflows and their Application to Software Engineering Processes*. International Journal On Advances in Intelligent Systems, 4(3&4):158–179, 2011.

[GOR11b] G. Grambow, R. Oberhauser und M. Reichert: *Towards a Workflow Language for Software Engineering*. In: *Proceedings of the 10th IASTED International Conference on Software Engineering (SE '11)*, Innsbruck, Februar 2011.

[GR92] J. Gray und A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[Gri97] M. Grimm: *ADEPT-TIME: Temporale Aspekte in flexiblen Workflow-Management-Systemen.* Diplomarbeit, Universität Ulm, Ulm, April 1997.

[GRMR+08] C.W. Günther, S. Rinderle-Ma, M. Reichert, W.M.P. van der Aalst und J. Recker: *Using Process Mining to Learn from Process Changes in Evolutionary Systems.* Journal of Business Process Integration and Management, Special Issue on Business Process Flexibility, 3(1):61–78, 2008.

[GRRvdA06] C.W. Günther, S. Rinderle, M. Reichert und W.M.P. van der Aalst: *Change Mining in Adaptive Process Management Systems.* In: *On The Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Band 4275 der Reihe *LNCS*, Seiten 309–326, Montpellier, Oktober 2006. Springer.

[GRvdA08] C.W. Günther, M. Reichert und W.M.P. van der Aalst: *Supporting Flexible Processes with Adaptive Workflow and Case Handling.* In: *Proceedings of the 17th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '08)*, Seiten 229–234, Rom, Juni 2008. IEEE Computer Society Press.

[Hal10] A. Hallerbach: *Management von Prozessvarianten.* Dissertation, Universität Ulm, Februar 2010.

[Hal11] S. Haller: *Chancen und Risiken der Prozessautomatisierung am Finanzsektor am Beispiel des BPMS Activiti.* Bachelorarbeit, Universität Ulm, Ulm, Dezember 2011.

[HBR10] A. Hallerbach, T. Bauer und M. Reichert: *Capturing Variability in Business Process Models: The Provop Approach.* Journal of Software Maintenance and Evolution: Research and Practice, 22(6–7):519–546, November 2010.

[Hei00] C. Heinlein: *Workflow- und Prozesssynchronisation mit Interaktionsausdrücken und -graphen.* Dissertation, Universität Ulm, Juli 2000.

[HJKW96] P. Heimann, G. Joeris, C.A. Krapp und B. Westfechtel: *DYNAMITE: Dynamic Task Nets for Software Process Management.* In: *Proceedings of the 18th ICSE*, Seiten 331–341, Berlin, 1996.

[HKP11] J. Han, M. Kamber und J. Pei: *Data Mining: Concepts and Techniques.* Elsevier, 2011.

[HR01] T. Härder und E. Rahm: *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, 2001.

[hsq] *HyperSQL DataBase (HSQLDB).* http://www.hsqldb.org/. Abgerufen am 28.3.2014.

[ibm] *IBM Websphere.* http://www.ibm.com/software/de/websphere/. Abgerufen am 26.8.2013.

[ISO] ISO: *ISO/IEC 9075: Information technology – Database languages – SQL.*

[ISO01] ISO: *ISO/IEC 9126-1: Software engineering – Product quality – Part 1: Quality model*, 2001.

[jaa] *Java™ Authentication and Authorization Service (JAAS) Reference Guide.* http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html. Abgerufen am 28.3.2014.

[Jab97] S. Jablonski: *Architektur vonWorkflow-Management-Systemen.* Informatik – Forschung und Entwicklung, 12(2):72–81, 1997.

[Jab98] S. Jablonski: *A Software Architecture for Workflow Management Systems.* In: *9th International Workshop on Database and Expert Systems Applications (DEXA)*, Seiten 739–744, Wien, August 1998.

[Jab00] S. Jablonski: *Workflow Management between Formal Theory and Pragmatic Approaches.* In: *Business Process Management Models, Techniques, and Empirical Studies*, Band 1806 der Reihe *LNCS*, Seiten 345–358. Springer, 2000.

[jaf] *JavaFX.* http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html. Abgerufen am 28.3.2014.

[jav] *Oracle Technology Network – Java.* http://www.oracle.com/technetwork/java/index.html. Abgerufen am 28.3.2014.

[jbp] *jBPM.* http://www.jboss.org/jbpm/. Abgerufen am 28.3.2014.

[jdb] *Java™ Database Connectivity.* http://www.oracle.com/technetwork/java/overview-141217.html. Abgerufen am 28.3.2014.

[JH98] G. Joeris und O. Herzog: *Managing evolving workflow specifications.* In: M. Halper (Herausgeber): *Proceedings of the 3rd IFCIS Conference on Cooperative Information Systems (CoopIS '98)*, Seiten 310–319, New York, 1998. IEEE, IEEE Computer Society Press.

[Jon96] R. Jones: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[Jur06] M. Jurisch: *Konzeption eines Rahmenwerkes zur Erstellung und Modifikation von Prozessvorlagen und -instanzen.* Diplomarbeit, Universität Ulm, Ulm, März 2006.

[JWH00] G. Joeris, H. Wache und O. Herzog: *MOKASSIN – Innovative Technologien zum Prozessmanagement und zur semantischen Datenintegration.* KI, 14(1):52, 2000.

[Kar94] B. Karbe: *Flexible Vorgangssteuerung mit ProMInanD.* In: U. Hasenkamp, S. Kirn und M. Syring (Herausgeber): *CSCW – Computer Supported Cooperative Work: Informationssysteme für dezentralisierte Unternehmensstrukturen*, Seiten 117–133. Addison-Wesley, 1994.

[KGD99] M. Kradolfer, A. Geppert und K.R. Dittrich: *Workflow Specification in TRAMs.* In: J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau und E. Métais (Herausgeber): *Proceedings of the 18th International Conference on Conceptual Modeling (ER '99)*, Band 1728 der Reihe *LNCS*, Seiten 263–278, Paris, November 1999. Springer.

[KPR12] D. Knuplesch, R. Pryss und M. Reichert: *Data-Aware Interaction in Distributed and Collaborative Workflows: Modeling, Semantics, Correctness.* In: *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom '12)*, Seiten 223–232, Pittsburgh, Oktober 2012. IEEE Computer Society Press.

[KR09a] V. Künzle und M. Reichert: *Integrating Users in Object-aware Process Management Systems: Issues and Challenges.* In: S. Rinderle-Ma, S. Sadiq und F. Leymann (Herausgeber): *Business Process Management Workshops*, Band 43 der Reihe *LNBIP*, Seiten 29–41, Ulm, September 2009. Springer.

[KR09b] V. Künzle und M. Reichert: *Towards Object-Aware Process Management Systems: Issues, Challenges, Benefits.* In: *Proceedings of the 10th International Workshop on Business Process Modeling, Development, and Support (BPMDS '09)*, Band 29 der Reihe *LNBIP*, Seiten 197–210, Amsterdam, Juni 2009. Springer.

[KR10] U. Kreher und M. Reichert: *Speichereffiziente Repräsentation instanzspezifischer Änderungen in Prozess-Management-Systemen.* Technical Report UIB-2010-02, Universität Ulm, Ulm, 2010.

[KR11a] V. Künzle und M. Reichert: *A Modeling Paradigm for Integrating Processes and Data at the Micro Level.* In: *Proceedings Enterprise, Business-Process, and Information Systems Modelling: 12th International Workshop BPMDS and 16th International Conference EMMSAD at CAiSE 2011*, Band 81 der Reihe *LNBIP*, Seiten 201–215, London, Juni 2011. Springer.

[KR11b] V. Künzle und M. Reichert: *PHILharmonicFlows: towards a framework for object-aware process management.* Journal of Software Maintenance and Evolution: Research and Practice, 23(4):205–244, Juni 2011.

[KR12] V. Künzle und M. Reichert: *Striving for Object-Aware Process Support: How Existing Approaches Fit Together.* In: *First International Symposium on Data-driven Process Discovery and Analysis (SIMPDA '11)*, Band 116 der Reihe *LNBIP*, Seiten 169–188, Campione d'Italia, Juni 2012. Springer.

[KRFRM13] D. Knuplesch, M. Reichert, W. Fdhila und S. Rinderle-Ma: *On Enabling Compliance of Cross-Organizational Business Processes.* In: *Business Process Management*, Band 8094 der Reihe *LNCS*, Seiten 146–154. Springer, Peking, August 2013.

[KRP+13] D. Knuplesch, M. Reichert, R. Pryss, W. Fdhila und S. Rinderle-Ma: *Ensuring Compliance of Distributed and Collaborative Workflows.* In: *Proceedings of the 9th International Conference on Collaborative Computing:*

Networking, Applications and Worksharing (CollaborateCom '13), Seiten 133–142, Austin, Oktober 2013. IEEE Computer Society Press.

[KRW90] B. Karbe, N. Ramsperger und P. Weiss: *Support of Cooperative Work by Electronic Circulation Folders.* In: *Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems*, Band 11 der Reihe *Conference on Supporting Group Work*, Seiten 109–117, Cambridge, Massachusetts, 1990. ACM New York, NY, USA.

[Kub98] M. Kubicek: *Organisatorische Aspekte in flexiblen Workflow-Management-Systemen.* Diplomarbeit, Universität Ulm, Ulm, März 1998.

[Kün13] V. Künzle: *Object-Aware Process Management.* Dissertation, Universität Ulm, Juli 2013.

[KWR11] V. Künzle, B. Weber und M. Reichert: *Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches.* International Journal of Information System Modeling and Design (IJISMD), 2(2):19–46, April 2011.

[Lan08] A. Lanz: *Realisierung einer Zeitmanagementkomponente eines adaptiven Prozess-Management-Systems.* Diplomarbeit, Universität Ulm, Ulm, August 2008.

[Ley95] F. Leymann: *Supporting Business Transactions via Partial Backward Recovery In Workflow Management Systems.* In: *GI-Fachtagung*, Seiten 51–70, Dresden, März 1995.

[LHHR05] R. Lenz, U. Hasenkamp, W. Hasselbring und M. Reichert (Herausgeber): *EAI 2005: Enterprise Application Integration – Tagungsband des 2. GI-/GMDS-Workshops zum Thema Enterprise Application Integration.* Springer, Philipps-Universität Marburg, Juni 2005.

[Li10] C. Li: *Mining Process Model Variants: Challenges, Techniques, Examples.* Dissertation, University of Twente, November 2010.

[Lit80] W. Litwin: *Linear Hashing: A New Tool for File and Table Addressing.* In: *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, Seiten 212–223, Montreal, Oktober 1980. IEEE Computer Society Press.

[LKR13] A. Lanz, J. Kolb und M. Reichert: *Enabling Personalized Process Schedules with Time-Aware Process Views.* In: *Proceedings of the CAiSE 2013 International Workshops*, Band 148 der Reihe *LNBIP*, Seiten 205–216, Valencia, Juni 2013. Springer.

[LKRD10] A. Lanz, U. Kreher, M. Reichert und P. Dadam: *Enabling Process Support for Advanced Applications with the AristaFlow BPM Suite.* In: *Proceedings of the Business Process Management 2010 Demonstration Track*, Hoboken, New Jersey, September 2010.

[LKRM+10] L.T. Ly, D. Knuplesch, S. Rinderle-Ma, K. Göser, H. Pfeifer, M. Reichert und P. Dadam: *SeaFlows Toolset – Compliance Verification Made Easy for Process-aware Information Systems*. In: *Proceedings CAiSE '10 Forum, Information Systems Evolution*, Band 72 der Reihe *LNBIP*, Seiten 76–91, Hammamet, Juni 2010. Springer.

[LPCR13] A. Lanz, R. Posenato, C. Combi und M. Reichert: *Controllability of Time-Aware Processes at Run Time*. In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, Band 8185 der Reihe *LNCS*, Seiten 39–56, Graz, September 2013. Springer.

[LR99] F. Leymann und D. Roller: *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999.

[LR05] R Lenz und M. Reichert: *IT Support for Healthcare Processes*. In: *Business Process Management*, Band 3649 der Reihe *LNCS*, Seiten 354–363. Springer, September 2005.

[LR07] R Lenz und M. Reichert: *IT Support for Healthcare Processes – Premises, Challenges, Perspectives*. Data & Knowledge Engineering, 61(1):39–58, 2007.

[LR12] M. Lohrmann und M. Reichert: *Efficacy-aware Business Process Modeling*. In: *On the Move to Meaningful Internet Systems: OTM 2012*, Band 7565 der Reihe *LNCS*, Seiten 38–55, Rom, September 2012. Springer.

[LR13] M. Lohrmann und M. Reichert: *Understanding Business Process Quality*. In: *Business Process Management*, Band 444 der Reihe *Studies in Computational Intelligence*, Seiten 41–73. Springer, 2013.

[LRD10a] A. Lanz, M. Reichert und P. Dadam: *Making Business Process Implementations Flexible and Robust: Error Handling in the AristaFlow BPM Suite*. In: *CAiSE '10 Demos*, Hammamet, Juni 2010.

[LRD10b] A. Lanz, M. Reichert und P. Dadam: *Robust and Flexible Error Handling in the AristaFlow BPM Suite*. In: *Proceedings CAiSE '10 Forum, Information Systems Evolution*, Nummer 72 in *LNBIP*, Seiten 174–189, Hammamet, Juni 2010. Springer.

[LRR04] M. Lauer, S. Rinderle und M. Reichert: *Repräsentation von Schema- und Instanzobjekten in adaptiven Prozess-Management-Systemen*. In: *GI-Jahrestagung*, Nummer 2, Seiten 555–560, Ulm, 2004.

[LRW08] C. Li, M. Reichert und A. Wombacher: *Discovering Reference Process Models by Mining Process Variants*. In: *Proceedings 6th International Conference on Web Services (ICWS '08)*, Seiten 45–53, Beijing, September 2008. IEEE Computer Society Press.

[LRW09a] C. Li, M. Reichert und A. Wombacher: *Discovering Reference Models by Mining Process Variants Using a Heuristic Approach*. In: *Business Process Management*, Band 5701 der Reihe *LNCS*, Seiten 344–362, Ulm, September 2009. Springer.

[LRW09b] C. Li, M. Reichert und A. Wombacher: *Mining Based on Learning from Process Change Logs.* In: *Business Process Management Workshops*, Band 17 der Reihe *LNCS*, Seiten 121–133, Mailand, 2009. Springer.

[LRW10] C. Li, M. Reichert und A. Wombacher: *The MinAdept Clustering Approach for Discovering Reference Process Models out of Process Variants.* International Journal of Cooperative Information Systems (IJCIS), 19(3 & 4):159–203, 2010.

[LRW11] C. Li, M. Reichert und A. Wombacher: *Mining Business Process Variants: Challenges, Scenarios, Algorithms.* Data & Knowledge Engineering, 70(5):409–434, 2011.

[LWR10] A. Lanz, B. Weber und M. Reichert: *Workflow Time Patterns for Process-aware Information Systems.* In: *Proceedings Enterprise, Business-Process, and Information Systems Modelling: 11th International Workshop BPMDS and 15th International Conference EMMSAD at CAiSE 2010*, Band 50 der Reihe *LNBIP*, Seiten 94–107, Hammamet, Juni 2010. Springer.

[LWR14] A. Lanz, B. Weber und M. Reichert: *Time patterns for process-aware information systems.* Requirements Engineering, 19(2):113–141, 2014.

[LY05] H. Li und Y. Yang: *Dynamic checking of temporal constraints for concurrent workflows.* Electronic Commerce Research and Applications, 4(2):124–142, 2005.

[Ly13] L.T. Ly: *SeaFlows -– A Compliance Checking Framework for Supporting the Process Lifecycle.* Dissertation, Universität Ulm, April 2013.

[Man] A. T. Manes: *SOA is Dead; Long Live Services.* http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html. Abgerufen am 28.3.2014.

[MHHR06] D. Müller, J. Herbst, M. Hammori und M. Reichert: *IT Support for Release Management Processes in the Automotive Industry.* In: *Business Process Management*, Band 4102 der Reihe *LNCS*, Seiten 368–377, Wien, Oktober 2006. Springer.

[MRB08] B. Mutschler, M. Reichert und J. Bumiller: *Unleashing the Effectiveness of Process-oriented Information Systems: Problem Analysis, Critical Success Factors and Implications.* In: *IEEE Transactions on Systems, Man, and Cybernetics (Part C)*, Band 38, Seiten 280–291, 2008.

[MRH07] D. Müller, M. Reichert und J. Herbst: *Data-driven Modeling and Coordination of Large Process Structures.* In: *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS '07)*, Band 4803 der Reihe *LNCS*, Seiten 131–149, Vilamoura, 2007. Springer.

[MRH08] D. Müller, M. Reichert und J. Herbst: *A new Paradigm for the Enactment and Dynamic Adaptation of Data-driven Process Structures.* In: *Proceedings 20th International Conference on Advanced Information Systems Engineering (CAiSE '08)*, Band 5074 der Reihe *LNCS*, Seiten 48–63, Montpellier, 2008. Springer.

[MS02] P. Mangan und S. Sadiq: *On Building Workflow Models for Flexible Processes.* In: *Proceedings of the 13th Australasian database conference – Volume 5*, ACM International Conference Proceeding Series, Seiten 103–109, Melbourne, Victoria, 2002. Australian Computer Society, Inc.

[MWR08] B. Mutschler, B. Weber und M. Reichert: *Workflow Management Versus Case Handling: Results from a Controlled Software Experiment.* In: *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, Seiten 82–89, Fortaleza, Brasilien, 2008. ACM.

[net] *.NET Framework.* http://www.microsoft.de/net. Abgerufen am 28.3.2014.

[NYHR05] C. Neuman, T. Yu, S. Hartman und K. Raeburn: *RFC 4120: The Kerberos Network Authentication Service (V5).* Technischer Bericht, IETF – Network Working Group, Juli 2005.

[OAS07] OASIS: *Web Services Business Process Execution Language.* http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007. Abgerufen am 28.3.2014.

[OMG09] OMG: *Business Process Modeling Notation (BPMN)*, Januar 2009.

[Opi07] R. Opitz: *Konzeption und Entwurf einer Komponente für die Arbeitslistenverwaltung.* Masterarbeit, Universität Ulm, Ulm, Juni 2007.

[PDA08] M. Predeschly, P. Dadam und H. Acker: *Security Challenges in Adaptive e-Health Processes.* In: *Proceedings of the 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2008)*, Band 5219 der Reihe *LNCS*, Seiten 181–192, Newcastle upon Tyne, September 2008. Springer.

[PLRH12] R. Pryss, D. Langer, M. Reichert und A. Hallerbach: *Mobile Task Management for Medical Ward Rounds – The MEDo Approach.* In: *Business Process Management Workshops*, Band 132 der Reihe *LNBIP*, Seiten 43–54, Tallinn, September 2012. Springer.

[PMLR14] R. Pryss, N. Mundbrod, D. Langer und M. Reichert: *Supporting medical ward rounds through mobile task and process management.* Information Systems and e-Business Management, Seiten 72–83, März 2014.

[PMR13] R. Pryss, S. Musiol und M. Reichert: *Collaboration Support Through Mobile Processes and Entailment Constraints.* In: *Proceedings of the 9th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom '13)*, Seiten 178–187, Austin, Oktober 2013. IEEE Computer Society Press.

[pos] *PostgreSQL.* http://www.postgresql.org/. Abgerufen am 28.3.2014.

[Pry05] R. Pryss: *Enterprise Application Integration: Anforderungen, Ansätze und Technologien.* Diplomarbeit, Universität Ulm, Ulm, Januar 2005.

[PTKR10] R. Pryss, J. Tiedeken, U. Kreher und M. Reichert: *Towards Flexible Process Support on Mobile Devices.* In: *Proceedings CAiSE '10 Forum, Information Systems Evolution*, Band 72 der Reihe *LNBIP*, Seiten 150–165, Hammamet, Juni 2010. Springer.

[pyt] *Python.* https://www.python.org/. Abgerufen am 28.3.2014.

[RB07] M. Reichert und T. Bauer: *Supporting Ad-hoc Changes in Distributed Workflow Management Systems.* In: *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS '07)*, Band 4803 der Reihe *LNCS*, Seiten 150–168, Vilamoura, 2007. Springer.

[RCWR04] M. Reichert, U. Catrinescu-Wiedemuth und S. Rinderle: *Evolution von Zugriffsregelungen in Informationssystemen.* In: *Proceedings Conference eBusiness Processes (EBP'04)*, Seiten 100–114, September 2004.

[RD97] M. Reichert und P. Dadam: *A Framework for Dynamic Changes in Workflow Management Systems.* In: *8th International Workshop on Database and Expert Systems Applications (DEXA)*, Seiten 42–48, Toulouse, September 1997.

[RD98] M. Reichert und P. Dadam: *ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control.* Journal of Intelligent Information Systems, 10(2):93–129, 1998.

[RD00] M. Reichert und P. Dadam: *Geschäftsprozessmodellierung und Workflow-Management – Konzepte, Systeme und deren Anwendung.* Industrie Management, 16(3):23–27, 2000.

[RD08] M. Reichert und P. Dadam: *Realizing Adaptive Proceess-aware Information Systems with ADEPT2.* Technical Report 13, University of Ulm, Ulm, September 2008.

[RD09] M. Reichert und P. Dadam: *Enabling Adaptive Process-aware Information Systems with ADEPT2.* In: *Handbook of Research on Business Process Modeling*, Seiten 173–203. Hershey, New York, New York, 2009.

[RDB03] M. Reichert, P. Dadam und T. Bauer: *Dealing with Forward and Backward Jumps in Workflow Management Systems.* Software and Systems Modeling, 2(1):37–58, 2003.

[RDRM+09a] M. Reichert, P. Dadam, S. Rinderle-Ma, M. Jurisch, U. Kreher und K. Göser: *Architectural Principles and Components of Adaptive Process Management Technology.* In: A. Heinzl, P. Dadam, S. Kirn und P. Lockemann (Herausgeber): *PRIMIUM – Process Innovation for Enterprise Software*, Lecture Notes in Informatics (LNI), Seiten 81–97. Koellen-Verlag, 2009.

[RDRM+09b] M. Reichert, P. Dadam, S. Rinderle-Ma, A. Lanz, R. Pryss, M. Predeschly, J. Kolb, T. Ly, M. Jurisch, U. Kreher und Göser K.: *Enabling Poka-Yoke Workflows with the AristaFlow BPM Suite.* In: *Proceedings of the Business Process Management 2009 Demonstration Track*, Ulm, September 2009.

[Rei00] M. Reichert: *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* Dissertation, Universität Ulm, Mai 2000.

[Rei11] M. Reichert: *What BPM Technology Can Do for Healthcare Process Support.* In: *Proceedings of the 13th Conference on Artificial Intelligence in Medicine (AIME 2011)*, Band 6747, Seiten 2–13, Bled, Juli 2011. Springer.

[RHD98] M. Reichert, C. Hensinger und P. Dadam: *Supporting Adaptive Workflows in Advanced Application Environments.* In: *EDBT Workshop on Workflow Management Systems (in conjunction with EDBT '98 conference)*, Seiten 100–109, Valencia, März 1998.

[Rin04] S. Rinderle: *Schema Evolution in Process Management Systems.* Dissertation, Universität Ulm, Oktober 2004.

[RJR07] S. Rinderle, M. Jurisch und M. Reichert: *On Deriving Net Change Information From Change Logs – The DELTALAYER-Algorithm.* In: *Proceedings 12th Conference Datenbanksysteme in Business, Technologie und Web (BTW '07)*, Nummer P-103 in *Lecture Notes in Informatics (LNI)*, Seiten 364–381, Aachen, 2007. Koellen-Verlag.

[RKL+06] S. Rinderle, U. Kreher, M. Lauer, P. Dadam und M. Reichert: *On Representing Instance Changes in Adaptive Process Management Systems.* In: *First IEEE Workshop on Flexibility in Process-aware Information Systems (ProFlex)*, Nummer 1, Seiten 297–302, Manchester, 2006.

[RMR08] S. Rinderle-Ma und M. Reichert: *Managing the Life Cycle of Access Rules in CEOSIS.* In: *Proceedings of the 12th IEEE International Enterprise Computing Conference (EDOC '08)*, Seiten 257–266, München, September 2008. IEEE Computer Society Press.

[RMR09] S. Rinderle-Ma und M. Reichert: *Comprehensive Life Cycle Support for Access Rules in Information Systems: The CEOSIS Project.* Enterprise Information Systems, 3(3):219–251, 2009.

[RMR10] S. Rinderle-Ma und M. Reichert: *Advanced Migration Strategies for Adaptive Process Management Systems.* In: *12th IEEE Conference on Commerce and Enterprise Computing (CEC '10)*, Seiten 56–63, Shanghai, November 2010. IEEE Computer Society Press.

[RMRW08a] S. Rinderle-Ma, M. Reichert und B. Weber: *On the Formal Semantics of Change Patterns in Process-aware Information Systems.* In: *Proceedings of the 27th International Conference on Conceptual Modeling (ER '08)*, Band 5231 der Reihe *LNCS*, Seiten 279–293, Barcelona, Oktober 2008. Springer.

[RMRW08b] S. Rinderle-Ma, M. Reichert und B. Weber: *Relaxed Compliance Notions in Adaptive Process Management Systems.* In: *Proceedings of the 27th International Conference on Conceptual Modeling (ER '08)*, Band 5231 der Reihe *LNCS*, Seiten 232–247, Barcelona, Oktober 2008. Springer.

[RR05] S. Rinderle und M. Reichert: *On the Controlled Evolution of Access Rules in Cooperative Information Systems.* In: *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, Band 3760 der Reihe *LNCS*, Seiten 238–255. Springer, November 2005.

[RR06] S. Rinderle und M. Reichert: *Data-Driven Process Control and Exception Handling in Process Management Systems.* In: *Proceedings 18th International Conference on Advanced Information Systems Engineering (CAiSE '06)*, Band 4001 der Reihe *LNCS*, Seiten 273–287, Luxembourg, 2006. Springer.

[RR07] S. Rinderle und M. Reichert: *A Formal Framework for Adaptive Access Control Models.* Journal on Data Semantics IX, 4601:82–112, 2007.

[RRD02] S. Rinderle, M. Reichert und P. Dadam: *Effiziente Verträglichkeitsprüfung und automatische Migration von Workflow-Instanzen bei der Evolution von Workflow-Schemata.* Informatik – Forschung und Entwicklung, 17(4):177–197, 2002.

[RRD03a] M. Reichert, S. Rinderle und P. Dadam: *ADEPT Workflow Management System: Flexible Support for Enterprise-wide Business Processes.* In: *Business Process Management*, Band 2678 der Reihe *LNCS*, Seiten 370–379, Eindhoven, 2003. Springer.

[RRD03b] M. Reichert, S. Rinderle und P. Dadam: *On the Common Support of Workflow Type and Instance Changes under Correctness Constraints.* In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, Band 2888 der Reihe *LNCS*, Seiten 407–425, Catania, 2003. Springer.

[RRD03c] S. Rinderle, M. Reichert und P. Dadam: *Evaluation of Correctness Criteria for Dynamic Workflow Changes.* In: *Proceedings of the Business Process Management 2003*, Band 2678 der Reihe *LNCS*, Seiten 41–57, Eindhoven, 2003. Springer.

[RRD04a] S. Rinderle, M. Reichert und P. Dadam: *Correctness Criteria for Dynamic Changes in Workflow Systems – A Survey.* Data & Knowledge Engineering, 50(1):9–34, 2004.

[RRD04b] S. Rinderle, M. Reichert und P. Dadam: *Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications.* In: *On The Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, Band 3290 der Reihe *LNCS*, Seiten 101–120, Agia Napa, 2004. Springer.

[RRD04c] S. Rinderle, M. Reichert und P. Dadam: *Flexible Support of Team Processes by Adaptive Workflow Systems.* Distributed and Parallel Databases, 16(1):91–116, 2004.

[RRD04d] S. Rinderle, M. Reichert und P. Dadam: *On Dealing with Structural Conflicts between Process Type and Instance Changes.* In: *Business Process Management*, Band 3080 der Reihe *LNCS*, Seiten 274–289, Potsdam, Juni 2004. Springer.

[RRJK06] S. Rinderle, M. Reichert, M. Jurisch und U. Kreher: *On Representing, Purging, and Utilizing Change Logs in Process Management Systems.* In: *Business Process Management*, Band 4102 der Reihe *LNCS*, Seiten 241–256, Wien, Oktober 2006. Springer.

[RRKD05] M. Reichert, S. Rinderle, U. Kreher und P. Dadam: *Adaptive Process Management with ADEPT2.* In: *Proceedings International Conference on Data Engineering (ICDE)*, Seiten 1113–1114, Tokyo, 2005.

[RRMD09] M. Reichert, S. Rinderle-Ma und P. Dadam: *Flexibility in Process-aware Information Systems.* In: *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, Band 5460 der Reihe *LNCS*, Seiten 115–135, Eindhoven, 2009. Springer.

[RS04] M. Reichert und D. Stoll: *Komposition, Choreographie und Orchestrierung von Web Services: Ein Überblick.* EMISA-Forum, 24(2):21–32, 2004.

[RtHEvdA05] N. Russell, A.H.M. ter Hofstede, D. Edmond und W.M.P. van der Aalst: *Workflow Data Patterns: Identification, Representation and Tool Support.* In: *Conceptual Modeling – ER 2005*, Band 3716 der Reihe *LNCS*, Seiten 353–368. Springer, 2005.

[RvdAtHE05] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede und D. Edmond: *Workflow Resource Patterns: Identification, Representation and Tool Support.* In: *Advanced Information Systems Engineering*, Band 3520 der Reihe *LNCS*, Seiten 216–232. Springer, 2005.

[RW12] M. Reichert und B. Weber: *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies.* Springer, 2012.

[RWR06] S. Rinderle, A. Wombacher und M. Reichert: *Evolution of Process Choreographies in DYCHOR.* In: *On The Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Band 4275 der Reihe *LNCS*, Seiten 273–290, Montpellier, Oktober 2006. Springer.

[RWRW05] S. Rinderle, B. Weber, M. Reichert und W. Wild: *Integrating Process Learning and Process Evolution – A Semantics Based Approach.* In: *Business Process Management*, Band 3649 der Reihe *LNCS*, Seiten 252–267, Hoboken, New Jersey, September 2005. Springer.

[SH99] G. Saake und A. Heuer: *Datenbanken: Implementierungstechniken.* MITP-Verlag, 1999.

[sig] *Signavio GmbH.* http://www.signavio.com/. Abgerufen am 28.3.2014.

[SM95] D.M. Strong und S.M. Miller: *Exceptions and Exception Handling in Computerized Information Processes.* ACM Transactions on Information Systems, 13(2):206–233, April 1995.

[Smø03] D.E. Smørgrav: *Pluggable Authentication Modules.* Technischer Bericht, FreeBSD, 2003. Abgerufen am 28.3.2014.

[spo] *Fraunhofer SPOT*. http://www.spot.fraunhofer.de/. Abgerufen am 28.3.2014.

[Sto81] M. Stonebraker: *Operating System Support for Database Management*. Communications of the ACM, 27(7):412–418, 1981.

[Sto04] D. Stoll: *Komposition von Web Services*. Masterarbeit, Universität Ulm, Ulm, August 2004.

[svn] *Subversion®*. http://subversion.apache.org/. Abgerufen am 28.3.2014.

[tcl] *Tcl Developer Xchange*. http://www.tcl.tk/. Abgerufen am 28.3.2014.

[tib] *TIBCO Software Inc.* http://www.tibco.com/. Abgerufen am 28.3.2014.

[TRI09] L. Thom, M. Reichert und C. Iochpe: *Activity Patterns in Process-aware Information Systems: Basic Concepts and Empirical Evidence*. International Journal of Business Process Integration and Management (IJBPIM), 4(2):93–110, 2009.

[TvS06] A. Tanenbaum und M. van Steen: *Distributed Systems: Principles and Paradigms*. Prentice Hall International, 2nd Auflage, 2006.

[uml] *Unified Modeling Language™ (UML®)*. http://www.omg.org/spec/UML/. Abgerufen am 28.3.2014.

[uno] *Universal Network Objects*. http://wiki.openoffice.org/wiki/Uno. Abgerufen am 28.3.2014.

[vdA11] W.M.P. van der Aalst: *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[vdARD07] W.M.P. van der Aalst, M. Roseman und M. Dumas: *Deadline-based Escalation in Process-Aware Information Systems*. Decision Support Systems, 43(2):492–511, 2007.

[vdAtHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski und A.P. Barros: *Workflow Patterns*. Distributed and Parallel Databases, 14(1):5–51, 2003.

[vdAVH04] W.M.P. van der Aalst und K. Van Hee: *Workflow Management: Models, Methods, and Systems*. MIT Press, 2004.

[VVK08] J. Vanhatalo, H. Völzer und J. Koehler: *The Refined Process Structure Tree*. In: *Proceedings of the Business Process Management 2008*, Band 5240 der Reihe *LNCS*, Seiten 100–115. Springer, 2008.

[VVK09] J. Vanhatalo, H. Völzer und J. Koehler: *The Refined Process Structure Tree*. Data & Knowledge Engineering, 68(9):793–818, September 2009.

[Wes99] M. Weske: *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects*. Habilitation, Universität Münster, Dezember 1999.

[Wes01a] M. Weske: *Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System.* In: R. Sprague (Herausgeber): *Proceedings 34th Hawaii International Conference on System Sciences (HICSS-34).* IEEE Computer Society Press, Januar 2001.

[Wes01b] B. Westfechtel: *Ein graphbasiertes Managementsystem für dynamische Entwicklungsprozesse.* Informatik – Forschung und Entwicklung, 16(3):125–144, 2001.

[Wil92] P.R. Wilson: *Uniprocessor garbage collection techniques.* In: *Memory Management*, Band 637/1992 der Reihe *LNCS*, Seiten 1–42. Springer, 1992.

[WMR10] B. Weber, B. Mutschler und M. Reichert: *Investigating the effort of using business process management technology: Results from a controlled experiment.* Science of Computer Programming, 75(5):292–310, Mai 2010.

[Wol08] J. Wolz: *Erweiterte Kontroll- und Datenflusskonzepte in ADEPT.* Diplomarbeit, Universität Ulm, Ulm, Juli 2008.

[wpf] *Windows Presentation Foundation.* http://msdn.microsoft.com/de-de/netframework/aa663326.aspx. Abgerufen am 28.3.2014.

[WR92] H. Wächter und A. Reuter: *The ConTract Model.* In: A.K. Elmagarmid (Herausgeber): *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[WR09] A. Wagenknecht und U. Rüppel: *Improving Resource Management in Flood Response With process Models and webGIS.* In: *Proceedings of the 16th TIEMS Annual Conference 2009*, Seiten 141–151, Istanbul, Juni 2009.

[WRMR07] B. Weber, S. Rinderle-Ma und M. Reichert: *Change Support in Process-Aware Information Systems – A Pattern-Based Analysis.* Technical Report 76, University of Twente, Enschede, 2007.

[WRRM08] B. Weber, M. Reichert und S. Rinderle-Ma: *Change Patterns and Change Support Features – Enhancing Flexibility in Process-Aware Information Systems.* Data & Knowledge Engineering, 66(3):438–466, 2008.

[WRW06] B. Weber, M. Reichert und W. Wild: *Case-Base Maintenance for CCBR-based Process Evolution.* In: *Proceedings of the 8th European Conference (ECCBR 2006)*, Band 4106 der Reihe *LNCS*, Seiten 106–120, Fethiye, September 2006. Springer.

[WRWR05a] B. Weber, M. Reichert, W. Wild und S. Rinderle: *Balancing Flexibility and Security in Adaptive Process Management Systems.* In: *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, Band 3760 der Reihe *LNCS*, Seiten 59–76. Springer, November 2005.

[WRWR05b] B. Weber, S. Rinderle, W. Wild und M. Reichert: *CCBR–Driven Business Process Evolution.* In: *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR 2005)*, Band 3620 der Reihe *LNCS*, Seiten 610–624, Chicago, August 2005. Springer.

[WRWRM09] B. Weber, M. Reichert, W. Wild und S. Rinderle-Ma: *Providing Integrated Life Cycle Support in Process-Aware Information Systems.* International Journal of Cooperative Information Systems (IJCIS), 18(1):115–165, 2009.

[YAW10] The YAWL Foundation: *YAWL-Technical Manual*, Version 2.1 Auflage, 2010. Abgerufen am 28.3.2014.

[YAW12] The YAWL Foundation: *YAWL-User Manual*, Version 2.3 Auflage, 2004–2012. Abgerufen am 28.3.2014.

[YHK95] W. Yeong, T. Howes und S. Kille: *RFC 1777: Lightweight Directory Access Protocol.* Technischer Bericht, IETF – Network Working Group, März 1995.

Abkürzungsverzeichnis

ADEPT *Application development based on encapsulated process templates* [DKR+95, Rei00]

API *Application Programming Interface*

AWT *Abstract Window Toolkit*

BLOB *Binary Large Object*

BPMN *Business Process Model and Notation*

CLOB *Character Large Object*

CORBA *Common Object Request Broker Architecture*

CVS *Concurrent Versions System*

DBMS Datenbankmanagementsystem

DMS Dokumentenmanagementsystem

DNS *Domain Name System*

DOM *Document Object Model*

EAI *Enterprise Application Integration*

EJB *Enterprise JavaBeans*

ERP *Enterprise Resource Planning*

GTK+ GIMP-Toolkit

HTML *Hypertext Markup Language*

HTTP *Hypertext Transfer Protocol*

ICN *Information Control Network*

JAAS *Java Authentication and Authorization Service* [jaa]

JDBC *Java Database Connectivity* [jdb]

JIT Just-in-time-Kompilierung

JSF *JavaServer Faces*

JVM *Java Virtual Machine*

KPI *Key Performance Indicator*

LDAP *Lightweight Directory Access Protocol* [YHK95]

PAM *Pluggable Authentication Modules* [Smø03]

PrMS Prozeßmanagementsystem

RPC *Remote Procedure Call*

SMTP *Simple Mail Transfer Protocol*

SOA Service-Orientierte Architektur

SSL *Secure Sockets Layer*

SWT *Standard Window Toolkit*

TCP *Transmission Control Protocol*

UDDI *Universal Description, Discovery and Integration*

UDF *User-Defined Function*

UDT *User-Defined Data Type*

UML *Unified Modeling Language* [uml]

URI *Uniform Resource Identifier*

URL *Uniform Resource Locator*

USB *Universal Serial Bus*

WSDL *Web Services Description Language*

XML *Extensible Markup Language*

XSD *XML Schema*

XPath *XML Path Language*

XQuery *XML Query Language*

XSLT *XSL Transformation*

Teil V

Anhang

A
Definitionen

Teile der folgenden Definitionen sind an [Rei00] angelehnt; von Datenelementen und -kanten wird im folgenden abstrahiert. Bei Vorlagen muss auf Typebene zwischen vollständigen Vorlagen ($\mathsf{Template}$) und (einfachen) Vorlagenobjekten ($\mathsf{Template}^{\Delta}$) unterschieden werden. Vollständige Vorlagen umfassen die Einheit aus Deltaschicht (in der jeweiligen Repräsentation) und dem einfachen Vorlagenobjekt, das die ursprüngliche und unveränderte Vorlage repräsentiert. Bei der algorithmischen Umsetzung von Operationen wird teilweise explizit unterschieden, ob die verwendeten Primitive auf die ursprüngliche Vorlage oder die Repräsentation der Deltaschicht angewendet werden. Primitive selbst können sowohl auf ursprüngliche Vorlagen als auch auf Deltaschichtrepräsentationen angewendet werden, da beide Objekttypen auf denselben Datenstrukturen beruhen (Knoten, Kanten, ...). Primitive haben einen undefinierten Rückgabewert, falls ein angefragtes Objekt nicht Bestandteil des entsprechenden einfachen Vorlagenobjekts/der entsprechenden Deltaschicht ist.

Aufzählungstypen

- $\mathsf{NodeType}$: `STARTFLOW`|`ENDFLOW`|`ACTIVITY`|`NULL`|`OR_SPLIT`|`OR_JOIN`|`AND_SPLIT`|`AND_JOIN`|`STARTLOOP`|`ENDLOOP`
- $\mathsf{NodeState}$: `NOT_ACTIVATED`|`ACTIVATED`|`SELECTED`|`RUNNING`|`COMPLETED`|`SKIPPED`|`FAILED`

Einfache Zugriffsoperationen

- Menge der Knoten einer Instanz:
  $Nodes : \mathsf{Instance} \longmapsto \{\mathsf{Node}\}$
- Knotentyp:
  $NodeType : \mathsf{Instance}, \mathsf{Node} \longmapsto \mathsf{NodeType}$
- Menge der direkten Vorgänger-/Nachfolgerknoten eines Knotens:
  $Pred : \mathsf{Instance}, \mathsf{Node} \longmapsto \{\mathsf{Node}\}$, $Succ : \mathsf{Instance}, \mathsf{Node} \longmapsto \{\mathsf{Node}\}$

- Menge der transitiven Vorgänger-/Nachfolgerknoten eines Knotens (ohne den Knoten selbst):
  $Pred^* : \mathsf{Instance}, \mathsf{Node} \longmapsto \{\mathsf{Node}\}$, $Succ^* : \mathsf{Instance}, \mathsf{Node} \longmapsto \{\mathsf{Node}\}$

- korrespondierender Verzweigungs-/Vereinigungsknoten eines Vereinigungs-/Verzweigungsknoten:
  $Join : \mathsf{Instance}, \mathsf{Node} \longmapsto \mathsf{Node}$, $Split : \mathsf{Instance}, \mathsf{Node} \longmapsto \mathsf{Node}$

- Menge der Verzweigungs-/Vereinigungsknoten einer Instanz:
  $SplitNodes : \mathsf{Instance} \longmapsto \{\mathsf{Node}\}$, $JoinNodes : \mathsf{Instance} \longmapsto \{\mathsf{Node}\}$

- Knotenzustand:
  $NodeState : \mathsf{Instance}, \mathsf{Node} \longmapsto \mathsf{NodeState}$
  $SetNodeState : \mathsf{Instance}, \mathsf{Node}, \mathsf{NodeState} \longmapsto \mathsf{Instance}$
  wobei $NodeState(SetNodeState(i, n_1, s), n_2)) = s, n_1 = n_2$
  und $f(SetNodeState(i, n_1, s), n_2) = f(i, n_2)$,
  $\forall f : \mathsf{Instance}, \mathsf{Node} \longmapsto \mathsf{Instance} \backslash NodeState \vee n_1 \neq n_2$

- Vorlage einer gegebenen Instanz $inst : \mathsf{Instance}$:
  $Template : \mathsf{Instance} \longmapsto \mathsf{Template}$

Komplexe Zugriffsoperationen

- Menge der Knoten einer Instanz in einem bestimmten Zustand:
  $NodesInState : \mathsf{Instance}, \mathsf{NodeState} \longmapsto \{\mathsf{Node}\}$
  $NodesInState(inst, state) := \{n \mid n \in Nodes(inst) \wedge NodeState(inst, n) = state\}$

- Menge der aktiven Knoten einer Instanz $inst : \mathsf{Instance}$:
  $ActiveNodes : \mathsf{Instance} \longmapsto \{\mathsf{Node}\}$
  $$\begin{aligned} ActiveNodes(inst) := \; & NodesInState(inst, \texttt{ACTIVATED}) \\ & \cup NodesInState(inst, \texttt{SELECTED}) \\ & \cup NodesInState(inst, \texttt{RUNNING}) \end{aligned}$$

- Menge der Knoten aus Teilzweige alternativer Verzweigungspfade einer Instanz $inst : \mathsf{Instance}$:
  $AltBranches : \mathsf{Instance} \longmapsto \{\mathsf{Node}\}$
  $$\begin{aligned} AltBranches(inst) := \{n \mid & s \in SplitNodes(inst) \wedge NodeType(inst, s) = \texttt{OR_SPLIT} \\ & \wedge n \in (Succ^*(inst, s) \cap Pred^*(inst, Join(inst, s)))\} \end{aligned}$$

- Menge der Knoten gewählter Pfade alternativer Verzweigungen bei der bisherigen Ausführung einer Instanz
  $inst : \mathsf{Instance}$:
  $ChosenBranches : \mathsf{Instance} \longmapsto \{\mathsf{Node}\}$
  $$\begin{aligned} ChosenBranches(inst) := \{n \mid & n \in AltBranches(inst) \\ & \wedge NodeState(inst, n) = \texttt{COMPLETED}\} \end{aligned}$$

- Menge der Knoten aus parallelen Teilzweigen eines bestimmten Knotens einer Instanz $inst : \mathsf{Instance}$:

$ParNodes$: Instance, Node $\longmapsto$ {Node}
$ParNodes(inst, node) := \{n \mid s \in SplitNodes(Pred^*(inst, node))$
$\wedge NodeType(inst, s) = \texttt{AND_SPLIT}$
$\wedge n \in (Succ^*(inst, s) \cap Pred^*(inst, Join(inst, s)))$
$\wedge n \notin Pred^*(inst, node) \wedge n \notin Succ^*(inst, node)\}$

- Menge aller Instanzen einer gegebenen Vorlage $templ$: Template:
  $Instances$: Template $\longmapsto$ {Instance}
  $Instances(templ) := \{inst \mid Template(inst) = templ\}$

Zugriffsprimitive

- Menge der Knoten eines Vorlagenobjekts:
  $PrimNodes$: Template$^{\Delta}$ $\longmapsto$ {Node}

- Menge der direkten Vorgänger-/Nachfolgerknoten eines Knotens bezüglich eines Vorlagenobjekts:
  $PrimPred$: Template$^{\Delta}$, Node $\longmapsto$ {Node}, $PrimSucc$: Template$^{\Delta}$, Node $\longmapsto$ {Node}

Relationen bezüglich Verträglichkeit der Änderungen einer Schemaevolution bei Zustandsclusterung

- Menge der Knoten einer Instanz $inst$: Instance in einem gegebenen Zustand $state$: NodeState vor (oder parallel zu) einer gegebenen Knotenmenge (in diesem Kontext ein gegebener Änderungskontext $changeContext$: {Node}):
  $BeforeContext$: Instance, NodeState, {Node} $\longmapsto$ {Node}
  $BeforeContext(inst, state, nodes) :=$
  $\{n \mid n \in NodesInState(inst, state) \wedge (\forall s \in nodes \backslash \{n\} : n \notin Succ^*(inst, s))\}$

- Menge der Knoten einer Instanz $inst$: Instance in einem gegebenen Zustand $state$: NodeState nach (nicht parallel zu) einer gegebenen Knotenmenge (in diesem Kontext ein gegebener Änderungskontext $changeContext$: {Node}):
  $AfterContext$: Instance, NodeState, {Node} $\longmapsto$ {Node}
  $AfterContext(inst, state, nodes) :=$
  $\{n \mid n \in NodesInState(inst, state)$
  $\wedge(\forall s \in nodes \backslash \{n\} : (n \notin Pred^*(inst, s) \wedge n \notin ParNodes(inst, s))$

- Menge der Instanzen, bezüglich deren Zustand des Änderungskontexts $changeContext$: {Node} die Änderungen einer Schemaevolution verträglich sind, da sich alle laufenden Knoten der Instanzen vor (oder parallel zu) dem Änderungskontext befinden:
  $CompliantInstances$: Template, {Node} $\longmapsto$ {Instance}
  $CompliantInstances(templ, changeContext) :=$
  $\{inst \mid inst \in Instances(templ)$
  $\wedge BeforeContext(inst, \texttt{RUNNING}, changeContext)$
  $= NodesInState(inst, \texttt{RUNNING})\}$

- Menge der Instanzen, bezüglich deren Zustand des Änderungskontexts $changeContext$: {Node} die Änderungen einer Schemaevolution nicht verträglich sind, da sich mindestens ein laufender oder abgeschlossener Knoten nach dem Änderungskontext befindet:

$NonCompliantInstances$: Template, {Node} $\longmapsto$ {Instance}
$NonCompliantInstances(templ, changeContext) :=$
$\{inst \mid inst \in Instances(templ)$
$\wedge(AfterContext(inst, \texttt{RUNNING}, changeContext)$
$\cup AfterContext(inst, \texttt{COMPLETED}, changeContext)$
$\cup AfterContext(inst, \texttt{SKIPPED}, changeContext)$
$\cup AfterContext(inst, \texttt{FAILED}, changeContext)) \neq \emptyset\}$

- Menge der Instanzen, deren Zustand bezüglich der Verträglichkeit einer Schemaevolution genauer überprüft werden müssen:
  $IndeterminedInstances$: Template, {Node} $\longmapsto$ {Instance}
  $IndeterminedInstances(templ, changeContext) :=$
  $\{inst \mid inst \in Instances(templ)$
  $\wedge BeforeContext(inst, \texttt{RUNNING}, changeContext)$
  $\neq NodesInState(inst, \texttt{RUNNING})\}$
  $\wedge(AfterContext(inst, \texttt{RUNNING}, changeContext)$
  $\cup AfterContext(inst, \texttt{COMPLETED}, changeContext)$
  $\cup AfterContext(inst, \texttt{SKIPPED}, changeContext)$
  $\cup AfterContext(inst, \texttt{FAILED}, changeContext)) = \emptyset\}$

Änderungsoperationen

- $InsertNodeSerial$: Instance, Node, Node, Node $\longmapsto$ Instance
- $DeleteSerialNode$: Instance, Node $\longmapsto$ Instance

Änderungsprimitive

- $AddControlEdge$: Template$^{\Delta}$, Node, Node $\longmapsto$ Template$^{\Delta}$
- $RemoveControlEdge$: Template$^{\Delta}$, Node, Node $\longmapsto$ Template$^{\Delta}$
- $AddNode$: Template$^{\Delta}$, Node $\longmapsto$ Template$^{\Delta}$
- $RemoveNode$: Template$^{\Delta}$, Node $\longmapsto$ Template$^{\Delta}$
- $ChangeNodeType$: Template$^{\Delta}$, Node, NodeType $\longmapsto$ Template$^{\Delta}$
- $CreateNodeData$: Instance, Node $\longmapsto$ Instance
- $DropNodeData$: Instance, Node $\longmapsto$ Instance

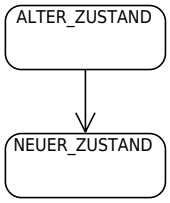
Weitere

- Ursprüngliche Vorlage einer Instanz:
  $BaseTemplate$: Instance $\longmapsto$ Template$^{\Delta}$
- Daten (Attributwerte) eines Knotens auf der entsprechenden (Substitutions-)Vorlage, diese sind nur definiert, falls der Knoten Bestandteil der (Substitutions-)Vorlage ist:
  $NodeData$: Template$^{\Delta}$, Node $\longmapsto$ Node

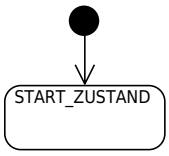
B
UML-Referenz

In diesem Kapitel erläutern wir die in UML-2-Diagrammen (Zustands-, Klassen- und Sequenzdiagrammen) verwendeten Elemente. Dies ist keine vollständige UML-2-Referenz.

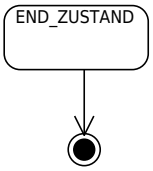
B.1 Zustandsdiagramm



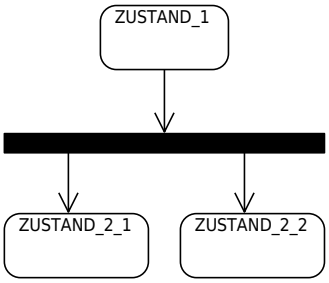
Zustände werden durch abgerundete Rechtecke dargestellt, Zustandsübergänge durch gerichtete Kanten. Normalerweise werden Zustandsübergänge explizit durch Ereignisse oder implizit ausgelöst. Im Rahmen der Arbeit werden sie meistens durch den Benutzer oder durch ein Ereignis (z. B. im Rahmen das Weiterschalten oder eine Schemaevolution) ausgelöst. Da dies aus den beteiligten Zuständen hervorgeht, sind die Zustandsübergänge nicht weiter detailliert. Das bei einem Zustandsübergang mögliche Verhalten ist im Rahmen dieser Arbeit irrelevant; hier interessieren nur die Zustände selbst und die Auswirkungen, die ein Zustand auf die Verwendung eines Objekts hat. So gibt ein Zustand beispielsweise an, ob eine Prozessvorlage als Subprozess in einer anderen Prozessvorlage verwendet werden darf.



Der initiale Zustand ist durch einen Zustandsübergang von einem schwarzen gefüllten Kreis gekennzeichnet. Diesen Zustand nimmt das entsprechende Objekt (z. B. eine Prozessvorlage oder eine Prozessinstanz) direkt nach der Erzeugung ein.

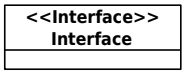


Ein Zustand mit einem Zustandsübergang zu einem schwarzen gefüllten Kreis mit einem umgebenden Kreis ist ein Endzustand. Vom Endzustand aus sind keine weiteren Zustandsänderungen mehr möglich. Dies schließt auch parallele Zustände mit ein, d. h. auch parallele Zustände können sich nicht mehr ändern, auch wenn hier kein Endzustand erreicht wurde.

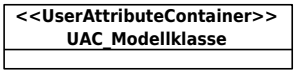


Da ein Objekt bezüglich verschiedener Aspekte einen Zustand einnehmen kann, können Zustände parallel und unabhängig voneinander sein – vom Erreichen eines Endzustands abgesehen. Solche Zustände liegen hinter einer Gabelung wie links abgebildet. Ein Beispiel für zwei getrennte Aspekte ist die Änderbarkeit auf Vorlagen- und Instanzebene bei einer Prozessvorlage (Schemaevolution und instanzspezifische Änderung).

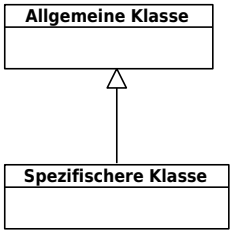
B.2 Klassendiagramm



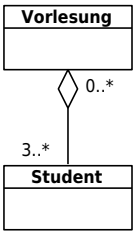
Der Stereotyp `Interface` dient im Rahmen dieser Arbeit dazu, Klassen mit gekapselter Logik von Klassen zu unterscheiden, die nur Daten bereitstellen. Während Datenklassen in erster Linie Attribute besitzen, haben als `Interface` markierte Klassen Methoden. Wie in der Einleitung zu Abschnitt 7.2 erläutert, werden Datenklassen technisch auch über Java-Interfaces realisiert, sie bieten keinen direkten Zugriff auf ihre Attribute.



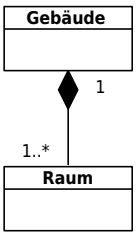
Der Stereotyp `UserAttributeContainer` kennzeichnet eine Klasse, die benutzerdefinierte Attribute zur Verfügung stellt und deren Manipulation ermöglicht. Auch hier ist der Zugriff auf die Attribute technisch über Methoden gelöst.



Die Generalisierung zwischen zwei Klassen bedeutet, dass eine allgemeine Klasse (die Klasse in Pfeilrichtung) durch eine speziellere Klasse erweitert wird. In der Regel besitzt die speziellere Klasse zusätzliche Attribute und Methoden. Sie kann aber auch Funktionen der allgemeinen Klasse einschränken.

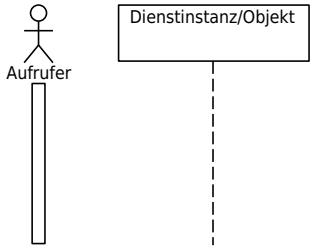


Eine Aggregation zwischen zwei Klassen zeigt an, dass eine Klasse aus anderen Klassen zusammengesetzt wird. Die offene Raute ist bei der zusammengesetzten Klasse. Zusätzlich kann eine Aggregation mit Kardinalitäten versehen werden. Das Beispiel links bedeutet, dass eine Vorlesung mindestens drei und bis zu beliebig viele Studenten hat. Ein Student dagegen kann keine oder beliebig viele Vorlesungen besuchen. Aus Gründen der besseren Übersichtlichkeit werden nicht immer Kardinalitäten angegeben. In so einem Fall ist die Kardinalität genau 1. Zusätzlich können Aggregationen noch benannt sein. Dieser Name identifiziert die Aggregation seitens der zusammengesetzten Klasse. Dies ist für die technische Umsetzung relevant: Aggregationen werden meistens über Referenzen gelöst, die wiederum Attribute in der zusammengesetzten Klasse sind. Der Name der Aggregation ist der Name des entsprechenden Attributs.

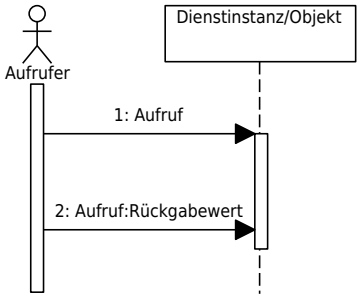


Eine Komposition zwischen zwei Klassen ist wie eine Aggregation. Der einzige Unterschied ist, dass die Beziehung zwischen den Klassen stärker ist. Die Existenz der Teilklasse (gegenüber der gefüllten Raute) ist an die Existenz der zusammengesetzten Klasse gebunden. Das Beispiel links sagt aus, dass ein Gebäude ein oder mehrere Räume haben kann, jeder Raum eindeutig einem Gebäude zugeordnet ist und dass es keine Räume ohne ein Gebäude gibt.

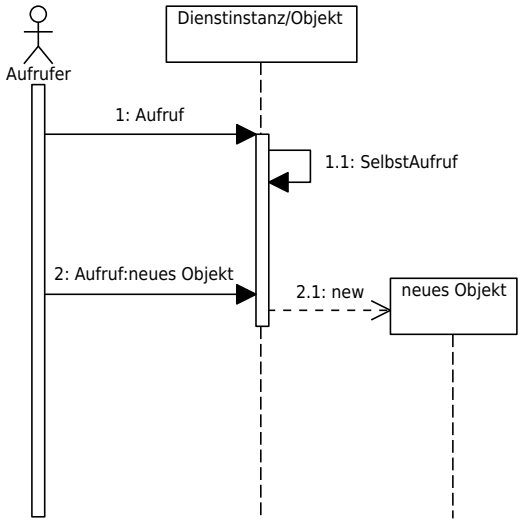
B.3 Sequenzdiagramm



Interaktionen gehen immer von einem Aufrufer aus. Dieser verwendet die Schnittstelle von Dienstinstanzen und Objekten. Unterhalb von Aufrufer und Dienstinstanzen bzw. Objekten sind Lebenslinien. Ist eine Dienstinstanz bzw. ein Objekt aktiv, d. h. gerade bei der Ausführung eines Aufrufs, so ist dies durch ein überlagertes Rechteck gekennzeichnet. Der Aufrufer ist ständig aktiv.



Pfeile zwischen Aufrufer und Dienstinstanzen/Objekten sowie Pfeile zwischen verschiedenen Dienstinstanzen/Objekten sind Aufrufe, d. h. eine Operation des Ziels wird aufgerufen. Die Beschriftung des Pfeils entspricht dem Operationsnamen. Hat die Operation einen Rückgabewert, steht dieser hinter dem Operationsnamen.



Selbstaufrufe sind komplexe Aufrufe, bei der das Aufrufziel zur Ausführung umfangreicher Logik Operationen von sich selbst verwendet. Daneben gibt es noch Aufrufe, die neue Objekte erzeugen. Die eigentliche Objekterzeugung erfolgt in einem Aufruf und ist durch einen gestrichelten Pfeil mit offener Spitze gekennzeichnet. Dieser führt zu einem Objekt, dessen Lebenslinie direkt nach dem gestrichelten Pfeil beginnt. Das erzeugte Objekt wird normalerweise beim auslösenden Aufruf zurückgegeben.

C
Algorithmen

C.1 Zustandsbestimmung bei impliziten Zuständen

Gesucht

- Knotenzustand eines bestimmten Knoten $node$: Node einer Instanz $inst$: Instance: $NodeState(inst, node)$

Algorithmus

(1) $node \in ActiveNodes(inst)$

(1.1) $NodeState(inst, node) \in \{\texttt{ACTIVATED}, \texttt{SELECTED}, \texttt{RUNNING}\}$
Ist $node$ ein aktiver Knoten, so wird sein Zustand explizit verwaltet und ist damit bekannt, Algorithmus abbrechen.

(2) $node \in \{n \mid a \in ActiveNodes(inst) \wedge n \in Succ^*(inst, a)\}$

(2.1) $NodeState(inst, node) := \texttt{NOT_ACTIVATED}$
Ist $node$ ein Nachfolger eines aktiven Knotens, so ist sein Zustand NOT_ACTIVATED.

(3) $node \in \{n \mid a \in ActiveNodes(inst) : \wedge n \in Pred^*(inst, a)\}$
Ist $node$ ein Vorgänger eines aktiven Knotens, so sind weitere Überprüfungen notwendig.

(3.1) $node \in \{n \mid n \in AltBranches(inst) : \wedge n \notin ChosenBranches(inst)\}$?
$NodeState(inst, node) := \texttt{SKIPPED}$
Befindet sich $node$ in einem Teilzweig, jedoch nicht in einem ausgeführten Teilzweig, so ist sein Zustand SKIPPED.

(3.2) $NodeState(inst, node) := \texttt{COMPLETED}$
In allen anderen Fällen befindet sich $node$ im Zustand COMPLETED.

Anmerkung

- Explizite Schleifenkanten werden bei diesem Algorithmus ignoriert. Die rekonstruierten Knotenzustände geben innerhalb einer Schleife den Zustand beim letztmaligen (bzw. dem aktuellen) Schleifendurchlauf an. Dies reicht aus, wenn zusätzliche Iterationszähler in den Zustandsinformationen einer Instanz mitgeführt werden (s. Abschnitt 2.2.3.2).

C.2 Instanzspezifische Änderungen

Um die Funktionsweise der Implementierungsvarianten ÄR-3-1 (Substitutionsvorlagen) und ÄR-3-2 (Gekapselte Vorlagen) vergleichen zu können, stellen wir in diesem Abschnitt vor, wie drei häufig benötigte Operationen für jede Variante algorithmisch umgesetzt werden. Dies umfasst die Ermittlung des/der direkten Nachfolgerknoten(s) sowie die Änderungsoperationen zum Einfügen eines Knotens in eine Knotensequenz und das Löschen eines Knotens.

Um die Algorithmen für die Änderungsoperationen verständlich zu halten, stellen wir hier nur die Durchführung der strukturellen Änderung dar. Die notwendigen Vorbedingungen bezüglich der Struktur und ggf. des Instanzzustands seien jeweils erfüllt. Diese entsprechen den in [Rei00] beschriebenen. Des weiteren verzichten wir auf die Darstellung weitergehender Anpassungen, wie notwendige Zustandsanpassungen und das Ändern von Verzweigungsprädikaten alternativer Verzweigungen, wenn durch das Löschen eines Knotens ein Zweig einer solchen Verzweigung wegfällt.

Definitionen

- Deltaschicht einer Instanz:
  $DeltaLayer$: $\mathsf{Instance} \longmapsto \mathsf{Template}^{\Delta}$
  $CreateDeltaLayer$: $\mathsf{Instance} \longmapsto \mathsf{Instance}$

- Überprüfung, ob eine Instanz bereits eine Deltaschicht hat:
  $HasDeltaLayer$: $\mathsf{Instance} \longmapsto \mathtt{TRUE}|\mathtt{FALSE}$

Anlegen der Deltaschicht-Datenstrukturen

Da Änderungen in der Deltaschicht verwaltet werden, müssen die entsprechenden Datenstrukturen vor der Durchführung einer Änderungsoperation angelegt werden, sofern diese noch nicht existieren. Dies erfolgt folgendermaßen:

(1) $\neg HasDeltaLayer(inst)$
Falls die Instanz noch keine Datenstruktur für die Deltaschicht besitzt, muss sie erst angelegt werden. Gegebenenfalls muss jeder Knoten noch ein neues Attribut bekommen, das die Vorlage angibt. Dieses kann aber auch schon vorher existieren.

(1.1) $inst := CreateDeltaLayer(inst)$

C.2.1 Substitutionsvorlagen

Definitionen

- (Substitutions-)Vorlage eines Knotens:
  $SubstTemplate$: $\mathsf{Instance}, \mathsf{Node} \longmapsto \mathsf{Template}^{\Delta}$

$SetSubstTemplate$: Instance, Node, Template$^{\Delta}$ $\longmapsto$ Instance,
wobei $SubstTemplate(SetSubstTemplate(i, n_1, t), n_2) = t, n_1 = n_2$
und $f(SetSubstTemplate(i, n_1, t), n_2) = f(i, n_2)$,
$\forall f$: Instance, Node $\longmapsto$ Template$^{\Delta} \backslash SubstTemplate \vee n_1 \neq n_2$

Nachfolgerbestimmung

Da Kanten bei Änderungen sowohl in der ursprünglichen als auch in der Substitutionsvorlage vorhanden sind, muss für den aktuellen Knoten lediglich die gültige (Substitutions-)Vorlage festgestellt werden. Mit dieser können dann direkt die Nachfolgerknoten ermittelt werden.

$Succ(inst, node)$

(1) $HasDeltaLayer(inst)$

(1.1) $substTempl := DeltaLayer(inst)$

(1.2) $node \in PrimNodes(substTemplate)$

(1.2.1) $currTempl := substTempl$

(1.3) $node \notin PrimNodes(substTemplate)$

(1.3.1) $currTempl := BaseTemplate(inst)$

(2) $\neg HasDeltaLayer(inst)$

(2.1) $currTempl := BaseTemplate(inst)$

(3) $Succ(inst, node) := PrimSucc(currTempl, node)$

Einfügen eines Knotens in eine Knotensequenz

$InsertSerialNode(inst, newNode, predNode, succNode)$

(1) $inst := CreateNodeData(inst, newNode)$

(2) $inst := SetNodeState(inst, newNode, \texttt{NOT_ACTIVATED})$

(3) $substTempl := DeltaLayer(inst)$

(4) $substTempl := AddNode(substTempl, node)$

(5) $substTempl := AddControlEdge(substTempl, predNode, node)$

(6) $substTempl := AddControlEdge(substTempl, node, succNode)$

(7) $inst := SetSubstTempl(inst, node, substTempl)$

(8) $predNode \in PrimNodes(substTemplate)$
Falls der Vorgängerknoten Bestandteil der Substitutionsvorlage ist, so ist es auch die Kante, die $predNode$ mit dem Nachfolgerknoten $succNode$ verbindet. Diese muss entfernt werden.

(8.1) $RemoveControlEdge(substTempl, predNode, succNode)$

(9) $predNode \notin PrimNodes(substTemplate)$
Falls der Vorgängerknoten noch nicht Bestandteil der Substitutionsvorlage ist, muss er mitsamt aller seiner Kanten übernommen werden.

(9.1) $substTempl := AddNode(substTempl, predNode)$

(9.2) $\forall ppN \in Pred(inst, predNode)$:
$substTempl := AddControlEdge(substTempl, ppN, predNode)$

(9.3) $substTempl := AddControlEdge(substTempl, predNode, node)$

(9.4) $\forall spN \in Succ(inst, predNode) \setminus succNode$:
$substTempl := AddControlEdge(substTempl, predNode, spN)$

(9.5) $inst := SetSubstTempl(inst, predNode, substTempl)$

(10) $succNode \in PrimNodes(substTemplate) \wedge succNode \in Succ(predNode)$
Falls der Nachfolgerknoten Bestandteil der Substitutionsvorlage ist, so ist oder war es auch die Kante, die *succNode* mit dem Vorgängerknoten *predNode* verbindet. Diese ist bereits entfernt, wenn der Vorgängerknoten vor der Änderung bereits Bestandteil der Substitutionsvorlage war (s. o.).

(10.1) $RemoveControlEdge(substTempl, predNode, succNode)$

(11) $succNode \notin PrimNodes(substTemplate)$
Falls der Nachfolgerknoten noch nicht Bestandteil der Substitutionsvorlage ist, muss er mitsamt aller seiner Kanten übernommen werden.

(11.1) $substTempl := AddNode(substTempl, succNode)$

(11.2) $\forall psN \in Pred(inst, succNode) \setminus predNode$:
$substTempl := AddControlEdge(substTempl, psN, succNode)$

(11.3) $substTempl := AddControlEdge(substTempl, node, succNode)$

(11.4) $\forall ssN \in Succ(inst, succNode)$:
$substTempl := AddControlEdge(substTempl, succNode, ssN)$

(11.5) $inst := SetSubstTempl(inst, succNode, substTempl)$

Löschen eines Knotens

$DeleteSerialNode(inst, node)$

(1) $inst := DropNodeData(inst, node)$

(2) $substTempl := DeltaLayer(inst)$

(3) $templ := BaseTemplate(inst)$

(4) $predNode := Pred(inst, Node)$

(5) $succNode := Succ(inst, Node)$

(6) $node \in PrimNodes(substTemplate)$
Falls der zu löschende Knoten Bestandteil der Substitutionsvorlage ist, wird er physisch gelöscht.

(6.1) $substTempl := RemoveNode(substTempl, node)$

(6.2) $substTempl := RemoveControlEdge(substTempl, predNode, node)$

(6.3) $substTempl := RemoveControlEdge(substTempl, node, succNode)$

(6.4) $succNode \in PrimSucc(substTemplate, pred)$
Ist die neue Kante zwischen dem Vorgänger- und dem Nachfolgerknoten nicht Bestandteil der ursprünglichen Vorlage, so muss sie der Substitutionsvorlage hinzugefügt werden. Dazu müssen auch die beiden Knoten Bestandteil der Substitutionsvorlage sein.

(6.4.1) $predNode \notin PrimNodes(substTemplate)$
Falls der Vorgängerknoten noch nicht Bestandteil der Substitutionsvorlage ist, muss er mitsamt aller seiner Kanten übernommen werden.

(6.4.1.1) $substTempl := AddNode(substTempl, predNode)$

(6.4.1.2) $\forall ppN \in Pred(inst, predNode)$:
$substTempl := AddControlEdge(substTempl, ppN, predNode)$

(6.4.1.3) $\forall spN \in Succ(inst, predNode) \setminus node$:
$substTempl := AddControlEdge(substTempl, predNode, spN)$

(6.4.1.4) $inst := SetSubstTempl(inst, predNode, substTempl)$

(6.4.2) $succNode \notin PrimNodes(substTemplate)$
Auch beim Nachfolgerknoten müssen alle Kanten übernommen werden.

(6.4.2.1) $substTempl := AddNode(substTempl, succNode)$

(6.4.2.2) $\forall psN \in Pred(inst, succNode) \setminus node$:
$substTempl := AddControlEdge(substTempl, psN, succNode)$

(6.4.2.3) $\forall ssN \in Succ(inst, succNode)$:
$substTempl := AddControlEdge(substTempl, succNode, ssN)$

(6.4.2.4) $inst := SetSubstTempl(inst, succNode, substTempl)$

(6.4.3) $substTempl := AddControlEdge(substTempl, predNode, succNode)$

(6.5) $(NodeType(inst, predNode) = \texttt{OR_SPLIT}$
$\vee NodeType(inst, predNode) = \texttt{AND_SPLIT}$
$\vee NodeType(inst, predNode) = \texttt{STARTLOOP})$
$\wedge Join(inst, predNode) \in Succ(inst, predNode)$
Falls der Vorgänger des zu löschenden Knotens ein Verzweigungsknoten ist und nach dem Löschen eine leere Kante zum entsprechenden Vereinigungsknoten besitzt, muss diese Kante entfernt werden. Bei alternativen Verzweigungen muss anschließend noch das Verzweigungsprädikat entsprechend angepasst werden.

(6.5.1) $substTempl := RemoveControlEdge(substTempl, predNode, succNode)$

(6.5.2) $\| Succ(inst, predNode) \| = 1$
Falls der Verzweigungsknoten jetzt nur noch einen Nachfolgerknoten hat, müssen die Knotentypen angepasst werden. In diesem Fall handelt es sich dann beim Vorgänger- und Nachfolgerknoten um zusammengehörige Verzweigungs- und Vereinigungsknoten, da zwischen ihnen eine Verzweigung entfallen ist. Außerdem sind beide Knoten Bestandteil der Substitutionsvorlage, da sonst

die ursprüngliche Vorlage ein ungültiges Prozessmodell ist (Verzweigungs-/ Vereinigungsknoten hätten nur eine Verzweigung).

(6.5.2.1) $substTempl := ChangeNodeType(substTempl, predNode, \texttt{NULL})$

(6.5.2.2) $substTempl := ChangeNodeType(substTempl, succNode, \texttt{NULL})$

(6.6) $predNode \in PrimNodes(substTemplate)$
$\wedge NodeData(substTempl, predNode) = NodeData(templ, predNode)$
$\wedge PrimPred(substTempl, predNode) = PrimPred(templ, predNode)$
$\wedge PrimSucc(substTempl, predNode) = PrimSucc(templ, predNode)$
Falls der Vorgängerknoten des auf der Substitutionsvorlage gelöschten Knotens selbst Bestandteil der Substitutionsvorlage ist, seine Attribute sowie seine ein- und ausgehenden Kanten auf der Substitutionsvorlage gleich sind wie die in der ursprünglichen Vorlage, kann er aus der Substitutionsvorlage entfernt werden. Dadurch wird die Substitutionsvorlage minimal gehalten.

(6.6.1) $substTempl := RemoveNode(substTempl, predNode)$
Wir entfernen den Vorgängerknoten.

(6.6.2) $\forall sppN \in Pred(inst, predNode)$
Nun können noch alle Kanten des Vorgängerknotens aus der Substitutionsvorlage entfernt werden, wenn der entsprechende Vorgängerknoten auch nicht Bestandteil der Substitutionsvorlage ist.

(6.6.2.1) $sppN \notin PrimNodes(substTempl)?$
$substTempl := RemoveControlEdge(substTempl, sppN, predNode)$

(6.6.3) $\forall sspN \in Succ(inst, predNode)$
Dasselbe gilt auch für die Kanten des Vorgängerknotens zu weiteren Nachfolgerknoten.

(6.6.3.1) $sspN \notin PrimNodes(substTempl)?$
$substTempl := RemoveControlEdge(substTempl, predNode, sspN)$

(6.7) $succNode \in PrimNodes(substTemplate)$
$\wedge NodeData(substTempl, succNode) = NodeData(templ, succNode)$
$\wedge PrimPred(substTempl, succNode) = PrimPred(templ, succNode)$
$\wedge PrimSucc(substTempl, succNode) = PrimSucc(templ, succNode)$
Auch der Nachfolgerknoten und dessen Kanten können bei Erfüllung der entsprechenden Bedingungen aus der Substitutionsvorlage entfernt werden.

(6.7.1) $substTempl := RemoveNode(substTempl, succNode)$

(6.7.2) $\forall spsN \in Pred(inst, succNode)$

(6.7.2.1) $spsN \notin PrimNodes(substTempl)?$
$substTempl := RemoveControlEdge(substTempl, spsN, succNode)$

(6.7.3) $\forall sssN \in Succ(inst, succNode)$

(6.7.3.1) $sssN \notin PrimNodes(substTempl)?$
$substTempl := RemoveControlEdge(substTempl, succNode, sssN)$

(7) $node \notin PrimNodes(substTemplate)$
Falls der zu löschende Knoten nicht Bestandteil der Substitutionsvorlage ist, müssen

der Vorgänger- und der Nachfolgerknoten mit einer Kante in der Substitutionsvorlage verbunden werden. Dazu müssen beide Knoten ggf. mit allen ihren ein- und ausgehenden Kanten in die Substitutionsvorlage übernommen werden. Das gilt auch, wenn der zu löschende Knoten zwar Bestandteil der Substitutionsvorlage war, aber weder der Vorgänger- noch der Nachfolgerknoten.

(7.1) $substTempl := AddControlEdge(substTempl, predNode, succNode)$

(7.2) $predNode \in PrimNodes(substTempl)$
Gegebenenfalls muss der Vorgängerknoten neu in die Substitutionsvorlage übernommen werden. Ist er bereits Bestandteil der Substitutionsvorlage, so muss lediglich die Kante zum gelöschten Knoten entfernt werden.

(7.2.1) $substTempl := RemoveControlEdge(substTempl, predNode, node)$

(7.3) $predNode \notin PrimNodes(substTemplate)$
Falls der Vorgängerknoten noch nicht Bestandteil der Substitutionsvorlage ist, muss er mitsamt aller seiner Kanten übernommen werden.

(7.3.1) $substTempl := AddNode(substTempl, predNode)$

(7.3.2) $\forall ppN \in Pred(inst, predNode)$:
$substTempl := AddControlEdge(substTempl, ppN, predNode)$

(7.3.3) $\forall spN \in Succ(inst, predNode) \setminus node$:
$substTempl := AddControlEdge(substTempl, predNode, spN)$

(7.3.4) $inst := SetSubstTempl(inst, predNode, substTempl)$

(7.4) $succNode \in PrimNodes(substTemplate)$
Gegebenenfalls muss auch der Nachfolgerknoten neu in die Substitutionsvorlage übernommen werden. Ist er bereits Bestandteil der Substitutionsvorlage, so muss lediglich die Kante zum gelöschten Knoten entfernt werden.

(7.4.1) $substTempl := RemoveControlEdge(substTempl, node, succNode)$

(7.5) $succNode \notin PrimNodes(substTemplate)$
Auch beim Nachfolgerknoten müssen alle Kanten übernommen werden.

(7.5.1) $substTempl := AddNode(substTempl, succNode)$

(7.5.2) $\forall psN \in Pred(inst, succNode) \setminus node$:
$substTempl := AddControlEdge(substTempl, psN, succNode)$

(7.5.3) $\forall ssN \in Succ(inst, succNode)$:
$substTempl := AddControlEdge(substTempl, succNode, ssN)$

(7.5.4) $inst := SetSubstTempl(inst, succNode, substTempl)$

(7.6) $(NodeType(inst, predNode) = \texttt{OR_SPLIT}$
$\vee NodeType(inst, predNode) = \texttt{AND_SPLIT}$
$\vee NodeType(inst, predNode) = \texttt{STARTLOOP})$
$\wedge Join(inst, predNode) \in Succ(inst, predNode)$
Falls der Vorgänger des zu löschenden Knotens ein Verzweigungsknoten ist und nach dem Löschen eine leere Kante zum entsprechenden Vereinigungsknoten besitzt, muss diese Kante entfernt werden. Bei alternativen Verzweigungen muss anschließend noch das Verzweigungsprädikat entsprechend angepasst werden.

(7.6.1) $substTempl := RemoveControlEdge(substTempl, predNode, succNode)$

(7.6.2) $\| Succ(inst, predNode) \| = 1$
Falls der Verzweigungsknoten jetzt nur noch einen Nachfolgerknoten hat, müssen die Knotentypen angepasst werden. In diesem Fall handelt es sich dann beim Vorgänger- und Nachfolgerknoten um zusammengehörige Verzweigungs- und Vereinigungsknoten, da zwischen ihnen eine Verzweigung entfallen ist.

(7.6.2.1) $substTempl := ChangeNodeType(substTempl, predNode, \texttt{NULL})$

(7.6.2.2) $substTempl := ChangeNodeType(substTempl, succNode, \texttt{NULL})$

C.2.2 Gekapselte Vorlagen

Definitionen

- Ursprüngliche Vorlage einer Instanz:
  $BaseTemplate : \mathsf{Instance} \longmapsto \mathsf{Template}^{\Delta}$

- Gelöschte Knoten einer (gekapselten) Vorlage:
  $IsNodeDeleted : \mathsf{Template}^{\Delta}, \mathsf{Node} \longmapsto \texttt{TRUE}|\texttt{FALSE}$,
  $SetNodeDeleted : \mathsf{Template}^{\Delta}, \mathsf{Node} \longmapsto \mathsf{Template}^{\Delta}$, wobei
  $IsNodeDeleted(SetNodeDeleted(t, n_1), n_2) = \begin{cases} \texttt{TRUE} & n_1 = n_2 \\ \texttt{FALSE} & sonst \end{cases}$,
  und $f(SetNodeDeleted(t, n_1), n_2) = f(n_2)$,
  $\forall f : \mathsf{Template}^{\Delta}, \mathsf{Node} \longmapsto \mathsf{Template}^{\Delta} \backslash IsNodeDeleted \vee n_1 \neq n_2$

- Gelöschte Kanten einer (gekapselten) Vorlage:
  $IsEdgeDeleted : \mathsf{Template}^{\Delta}, \mathsf{Node}, \mathsf{Node} \longmapsto \texttt{TRUE}|\texttt{FALSE}$,
  $SetEdgeDeleted : \mathsf{Template}^{\Delta}, \mathsf{Node}, \mathsf{Node} \longmapsto \mathsf{Template}^{\Delta}$, wobei
  $IsEdgeDeleted(SetEdgeDeleted(t, n_1, n_2), n_3, n_4) = \begin{cases} \texttt{TRUE} & n_1 = n_3 \wedge n_2 = n_4 \\ \texttt{FALSE} & sonst \end{cases}$,
  und $f(SetEdgeDeleted(t, n_1, n_2), n_3, n_4) = f(t, n_3, n_4)$,
  $\forall f : \mathsf{Template}^{\Delta}, \mathsf{Node}, \mathsf{Node} \longmapsto \mathsf{Template}^{\Delta} \backslash IsEdgeDeleted$
  und $f(SetEdgeDeleted(t, n_1, n_2), n_3, n_4) = f(t, n_3, n_4)$,
  $\forall f : \mathsf{Template}^{\Delta}, \mathsf{Node}, \mathsf{Node} \longmapsto \mathsf{Template}^{\Delta} \backslash IsEdgeDeleted \vee n_1 \neq n_3 \vee n_2 \neq n_4$

Nachfolgerbestimmung

$Succ(inst, node)$

(1) $Succ(inst, node) := \emptyset$

(2) $HasDeltaLayer(inst)$

(2.1) $wrap := DeltaLayer(inst)$

(2.2) $\forall potSucc \in PrimSucc(BaseTemplate(inst), node)$
Wir fügen die Knoten hinzu, die die ursprüngliche Vorlage als Nachfolger liefert, bei denen die entsprechende Kante in der gekapselten Vorlage nicht als gelöscht markiert ist.

(2.2.1) $\neg IsEdgeDeleted(wrap, node, potSucc)$?
$Succ(inst, node) := Succ(inst, node) \cup \{potSucc\}$

(2.3) $Succ(inst, node) := Succ(inst, node) \cup PrimSucc(DeltaLayer(inst), node)$
Zusätzlich müssen noch alle Nachfolgerknoten der gekapselten Vorlage hinzugefügt werden.

(3) $\neg HasDeltaLayer(inst)$

(3.1) $Succ(inst, node) := PrimSucc(BaseTemplate(inst), node)$

Einfügen eines Knotens in eine Knotensequenz

$InsertSerialNode(inst, newNode, predNode, succNode)$

(1) $inst := CreateNodeData(inst, newNode)$

(2) $inst := SetNodeState(inst, newNode, \texttt{NOT_ACTIVATED})$

(3) $wrap := DeltaLayer(inst)$

(4) $wrap := AddNode(wrap, node)$

(5) $wrap := AddControlEdge(wrap, predNode, node)$

(6) $wrap := AddControlEdge(wrap, node, succNode)$

(7) $succNode \notin PrimSucc(wrap, predNode)$
Existiert die ursprüngliche Kante zwischen $predNode$ und $succNode$ noch nicht in der gekapselten Vorlage (der Nachfolger kann nicht als direkter Nachfolger des Vorgängerknotens bezüglich der gekapselten Vorlage ermittelt werden), so muss die Kante in der gekapselten Vorlage als gelöscht markiert werden.

(7.1) $wrap := SetEdgeDeleted(wrap, predNode, succNode)$

(8) $succNode \in PrimSucc(wrap, predNode)$
Ist die Kante zwischen $predNode$ oder $succNode$ Bestandteil der gekapselten Vorlage, wird sie entfernt und nicht gelöscht. Dadurch wird die gekapselte Vorlage minimal gehalten.

(8.1) $wrap := RemoveControlEdge(wrap, predNode, succNode)$

Löschen eines Knotens

$DeleteSerialNode(inst, node)$

(1) $inst := DropNodeData(inst, node)$

(2) $wrap := DeltaLayer(inst)$

(3) $templ = BaseTemplate(inst)$

(4) $predNode := Pred(inst, Node)$

(5) $succNode := Succ(inst, Node)$

(6) $node \in PrimNodes(templ)$
Der Knoten ist Bestandteil der ursprünglichen Vorlage und wird deshalb als gelöscht markiert.

(6.1) $wrap := SetNodeDeleted(wrap, node)$

(6.2) $predNode \in PrimPred(templ, node)$
Ist die Kante zum gelöschten Knoten auf der ursprünglichen Vorlage, so wird sie als gelöscht markiert.

(6.2.1) $wrap := SetEdgeDeleted(wrap, predNode, node)$

(6.3) $predNode \in PrimPred(wrap, node) \wedge \neg IsEdgeDeleted(wrap, predNode, node)$
Ist die Kante zum gelöschten Knoten Bestandteil der gekapselten Vorlage und noch nicht als gelöscht markiert, so wird sie entfernt.

(6.3.1) $wrap := RemoveControlEdge(wrap, predNode, node)$

(6.4) $succNode \in PrimSucc(templ, node)$
Dasselbe gilt für den Nachfolgerknoten.

(6.4.1) $wrap := SetEdgeDeleted(wrap, node, succNode)$

(6.5) $succNode \in PrimSucc(wrap, node) \wedge \neg IsEdgeDeleted(wrap, node, succNode)$

(6.5.1) $wrap := RemoveControlEdge(wrap, node, succNode)$

(6.6) $wrap := AddControlEdge(wrap, predNode, succNode)$

(7) $node \in PrimNodes(wrap) \wedge \neg IsNodeDeleted(wrap, node)$
Ist der Knoten Bestandteil der gekapselten Vorlage und noch nicht als gelöscht markiert (durch den vorherigen Fall), wird er entfernt.

(7.1) $wrap := RemoveNode(wrap, node)$

(7.2) $succNode \in PrimSucc(templ, predNode)$
Existiert die Kante zwischen dem Vorgänger- und dem Nachfolgerknoten des gelöschten Knotens in der ursprünglichen Vorlage, so ist sie in der gekapselten Vorlage als gelöscht markiert. Diese Markierung wird entfernt, indem die Kante von der gekapselten Vorlage entfernt wird.

(7.2.1) $wrap := RemoveControlEdge(wrap, predNode, succNode)$

(7.3) $succNode \notin PrimSucc(templ, predNode)$
Existiert die Kante zwischen dem Vorgänger- und dem Nachfolgerknoten des gelöschten Knoten nicht in der ursprünglichen Vorlage, so muss sie auf der gekapselten Vorlage hinzugefügt werden.

(7.3.1) $wrap := AddControlEdge(wrap, predNode, succNode)$

(7.4) $node \in PrimNodes(templ)$
Wurden an dem zu löschenden Knoten Attribute geändert, können seine umgebenden Kanten auf der ursprünglichen Vorlage existieren.

(7.5) $wrap := SetEdgeDeleted(wrap, predNode, node)$

(7.6) $wrap := SetEdgeDeleted(wrap, node, succNode)$

(8) $(NodeType(inst, predNode) = \texttt{OR_SPLIT}$
$\vee NodeType(inst, predNode) = \texttt{AND_SPLIT}$
$\vee NodeType(inst, predNode) = \texttt{STARTLOOP})$
$\wedge Join(inst, predNode) \in Succ(inst, predNode)$
Falls der Vorgänger des zu löschenden Knotens ein Verzweigungsknoten ist und nach den Löschen eine leere Kante zum entsprechenden Vereinigungsknoten besitzt, muss diese Kante entfernt werden. Bei alternativen Verzweigungen muss anschließend noch das Verzweigungsprädikat entsprechend angepasst werden.

(8.1) $succNode \in PrimSucc(templ, predNode)$
Ist die Kante nur in der ursprünglichen Vorlage, so wird sie als gelöscht markiert.

(8.1.1) $wrap := SetEdgeDeleted(wrap, predNode, succNode)$

(8.2) $succNode \in PrimSucc(wrap, predNode)$
$\wedge \neg IsEdgeDeleted(wrap, predNode, succNode)$
Ist die Kante in der gekapselten Vorlage und noch nicht als gelöscht markiert, so wird sie entfernt.

(8.2.1) $wrap := RemoveControlEdge(wrap, predNode, succNode)$

(8.3) $\| Succ(inst, predNode) \| = 1$
Falls der Verzweigungsknoten jetzt nur noch einen Nachfolgerknoten hat, müssen die Knotentypen angepasst werden.

(8.3.1) $wrap := ChangeNodeType(wrap, predNode, \texttt{NULL})$

(8.3.2) $wrap := ChangeNodeType(wrap, succNode, \texttt{NULL})$

(8.3.3) $succNode \in PrimSucc(templ)$
Ist diese einzige Kante Bestandteil der ursprünglichen Vorlage, sind die Knoten in der ursprünglichen Vorlage bereits vom richtigen Typ. Entsprechen ihre anderen Attribute in der gekapselten Vorlage denen der ursprünglichen Vorlage, können die Knoten von der ursprünglichen Vorlage entfernt werden.

(8.3.3.1) $NodeData(templ, predNode) = NodeData(wrap, predNode)?$
$wrap := RemoveNode(wrap, predNode)$

(8.3.3.2) $NodeData(templ, succNode) = NodeData(wrap, succNode)?$
$wrap := RemoveNode(wrap, succNode)$

D

Verwendete Abkürzungen für qualitativen Vergleich

$join_i$: Vereinigungsknoten der Verzweigung i

$C_{SchemaEvo}$: Gesamtkosten für Schemaevolution einer Prozessvorlage

$C_{StateAdapt}$: Kosten für die Anpassungen des Zustands einer Prozessinstanz; diese hängen von der Art der Änderung und dem momentanen Zustand der Prozessinstanz ab.

$C_{StateCompl}$: Kosten für die Überprüfung der Verträglichkeit bezüglich des Zustands einer Prozessinstanz; diese hängen von der Art der Änderung und dem momentanen Zustand der Prozessinstanz ab.

$C_{StructAdapt}$: Kosten für die strukturellen Anpassungen einer Instanzvorlage; diese hängen von der Art der Änderung ab.

$C_{StructCompl}$: Kosten für die Überprüfung der Verträglichkeit bezüglich der Struktur einer Prozessinstanz; diese hängen von der Art der Änderung ab.

$M_{ActiveNodes}$: Speicherbedarf (pro Prozessinstanz) für Informationen zur Verwaltung aller aktiven Knoten
Bei Variante ZR-2 (s. Abschnitt 4.1.3.2) werden nur die aktiven Knoten einer Prozessinstanz sowie einige weitere Informationen zur vollständigen Rekonstruktion des Zustands einer Prozessinstanz gespeichert.

$M_{ChangeEntry}$: Speicherbedarf für einen Eintrag in der Änderungshistorie

M_{Edge}: Speicherbedarf für eine Kante einer Prozessvorlage, unabhängig von ihrer Repräsentation

$M_{ExecEntry}$: Speicherbedarf für einen Eintrag in der Ausführungshistorie

$M_{InstanceChanges}$: Speicherbedarf für die Repräsentation der geänderten Strukturen einer Prozessinstanz gegenüber der ursprünglichen Prozessvorlage

$M_{InstanceNode}$: Speicherbedarf für die Repräsentation eines Instanzknotens

$M_{IterationCount}$: Speicherbedarf für den Iterationszähler für einen Instanzknoten

M_{Node}: Speicherbedarf für einen Knoten einer Prozessvorlage

$M_{NodeState}$: Speicherbedarf für die Repräsentation eines Knotenzustands

$M_{Reference}$: Speicherbedarf für eine Referenz, etwa von einer Prozessinstanz zu der entsprechenden Prozessvorlage bei instanzspezifischen Änderungen oder dem ihren Zustand repräsentierenden Knoten im entsprechenden Zustandsgraphen

$M_{StateNodes}$: Speicherbedarf eines Knotens eines Zustandsgraphen

M_{Total}: Gesamter Speicherbedarf für alle Prozessvorlagen- und Prozessinstanzdaten

$MSeq_{StateNodes}$: Speicherbedarf eines (Teil-)Zustandsgraphen für sequentielle Prozessschritte

$nodes(x, y)$: Menge der Prozessschritte von Prozessschritt x bis Prozessschritt y (einschließlich)

$N_{AddedNodes}$: Anzahl der hinzugefügten Knoten einer Prozessinstanz

$N_{AllInsts}$: Anzahl aller Prozessinstanzen der geänderten Prozessvorlage

$N_{Branches}$: Anzahl der Zweige einer parallelen Verzweigung (konstant)

$N_{Branches}(i)$: Anzahl der Zweige der alternativen Verzweigung i

$N_{ComplInsts}$: Anzahl der Prozessinstanzen, zu deren Zustand die Änderungen einer Schemaevolution verträglich sind und die dementsprechend migriert werden können.

N_{Edges}: Anzahl der Kanten pro Prozessvorlage

$N_{ExecEntriesPerNode}$: Anzahl der Einträge in der Ausführungshistorie pro Knoten
Da ein Knoten mehrere Zustände durchläuft werden mehrere Einträge in der Ausführungshistorie protokolliert.

$N_{Instances}$: Anzahl aller Prozessinstanzen im System unabhängig von der jeweiligen Prozessvorlage

$N_{MovedNodes}$: Anzahl der verschobenen Knoten einer Prozessinstanz

N_{Nodes}: Anzahl der Knoten pro Prozessvorlage

$N_{Nodes}(x, y)$: Anzahl der Prozessschritte zwischen Prozessschritt x (einschließlich) und Prozessschritt y (ausschließlich)

$N_{NodesPerBranch}$: Anzahl der Knoten eines Zweigs einer parallelen Verzweigung (konstant)

$N_{ParallelBlocks}$: Anzahl der parallelen Blöcke in einer Prozessvorlage

$N_{RemNodes}$: Anzahl der gelöschten Knoten einer Prozessinstanz

$N^*_{StateNodes}$: Anzahl aller Knoten eines Zustandsgraphen unter Berücksichtigung der Struktur der entsprechenden Prozessvorlage

$N_{StateNodes}(x, y)$**:** Anzahl der Knoten eines Zustandsgraphen für alle Prozessschritte zwischen Prozessschritt x (einschließlich) und Prozessschritt y (ausschließlich)

N_{States}**:** Anzahl aller Zustände des verwendeten Prozessmetamodells

$N_{Templates}$**:** Anzahl aller Prozessvorlagen im System

$NAlt_{StateNodes}(x, y)$**:** Anzahl aller Knoten eines Zustandsgraphen für alle Prozessschritte zwischen Prozessschritt x (einschließlich) und Prozessschritt y (ausschließlich) einer Prozessvorlage mit mindestens einer alternativen Verzweigung

$NPar_{StateNodes}$**:** Anzahl aller Knoten eines Zustandsgraphen für Prozessschritte in einer Sequenz

$NParBlock_{StateNodes}$**:** Anzahl aller Knoten eines Zustandsgraphen für Prozessschritte in einer parallelen Verzweigung

$NSeq_{StateNodes}$**:** Anzahl aller Knoten eines Zustandsgraphen für Prozessschritte in einer Sequenz

$NSeq_{StateNodes}(x, y)$**:** Anzahl aller Knoten eines Zustandsgraphen für alle Prozessschritte in einer Sequenz zwischen x (einschließlich) und y (ausschließlich)