„Die Realisierung eines graphischen Workflow-Editors in Java“

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

vorgelegt von:

Ralf Michel

1. Gutachter: Prof. Dr. Peter Dadam
2. Gutachter: Prof. Dr. Franz Schweiggert

1999

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Modellierungsunterstützungen

1. Einleitung

Das Thema Workflow-Management ist in den letzten Jahren eines der am meisten diskutierten Themen im Softwarebereich. Der Grund dafür ist, daß durch die zunehmende Globalisierung der Wettbewerbsdruck für die Unternehmen immer größer wird. Um ihre Kosten zu senken, verkleinern die Unternehmen ihre Organisationsstrukturen und optimieren ihre Prozesse. Die Überarbeitung der Aufbau- und Ablauforganisation wird auch unter dem Stichwort „Business Process Reengineering" (BPR) zusammengefaßt. Auch anderen Bereiche, wie z.B. Kliniken, führen wegen des zunehmenden Kostendrucks BPR-Maßnahmen durch.

Um Prozesse optimieren zu können, müssen vorhandene Abläufe erfaßt, analysiert und optimiert werden. Um dies zu vereinfachen, wurden Geschäftsprozeßoptimierungswerkzeuge entwickelt, die helfen, überflüssige und zu arbeitsintensive Abläufe aufzudecken und zu verbessern.

Die BPR-Maßnahmen können jedoch nur zum Erfolg führen, wenn es gelingt, die optimierten Prozesse anschließend durch geeignete Kommunikations- und Informationssysteme zu unterstützen. Diese Anforderungen versuchen Workflow-Management-Systeme (WfMS) [Jab95, Jab95a, JBS97] zu erfüllen, indem sie Geschäftsprozesse eines Unternehmens mit Hilfe von Computern automatisieren. WfMS können dabei als Middleware betrachtet werden, die verschiedene Benutzer koordiniert, die räumlich verteilt an der Lösung von Aufgaben arbeiten.

Die durch diese Automatisierung erhofften Vorteile für das Unternehmen und die Endanwender sind:

- Assistenz bei der Bearbeitung von Prozessen, d.h. beispielsweise die Meldung neuer vorliegender Prozeßschritte und Erinnerung bei drohenden Terminüberschreitungen
- Verbesserte Durchlaufzeiten
- Bessere Möglichkeiten zur Kommunikation und zum Datenaustausch zwischen verschiedenen Beteiligten eines Prozesses
- Vermeidung unnötiger Mehrfacheingaben von Daten

Um diese Vorteile für den Endbenutzer und das Unternehmen zu erreichen, müssen auch Ansprüche an ein WfMS gestellt werden. Diese Ansprüche sind nach [Jab95a]:

- *Skalierbarkeit*, da zu erwarten ist, daß die Anzahl der WfMS-Benutzer und auch die Zahl der auszuführenden Workflows ständig steigen werden.
- *Integration von Alt-Software.* Trotz Restrukturierung eines Anwendungssystems wird die weitere Verwendung bestehender Software Systeme (legacy software) unabdingbar sein. Derartige Alt-Systeme müssen in WfMS integrierbar sein.
- *Transparenz*. WfMS werden in verteilten und heterogenen Hard- und Softwareumgebungen eingesetzt. Sowohl Verteilung als auch Heterogenität sollen dem Benutzer verborgen werden.

Auf dem Markt gibt es mittlerweile eine fast unüberschaubare Anzahl von WfMS, die sich doch wesentlich voneinander unterscheiden. Diese Unterschiedlichkeit beginnt schon bei der Definition der auszuführenden Workflows. Hierzu werden verschiedene Ansätze zur Modellierung wie Petri-Netze,

State- und Activity-Charts, Aktivitätennetze, etc. verwendet. All diese Ansätze beschreiben den Kontroll- bzw. Datenfluß eines Prozesses in unterschiedlicher Weise, wobei bestimmte Aspekte in den Vordergrund treten und andere vernachlässigt werden.

Ein weiterer Nachteil der verschiedenen Workflow-Editoren als auch der WfMS an sich ist, daß sie nicht die Qualität erreichen, die die Nutzer von anderen Softwaresystemen her kennen. Neben der, trotz graphischen Beschreibungsmöglichkeiten, schweren Erlernbarkeit der Bedienung vieler Workflow-Editoren überlassen sie die korrekte und fehlerfreie Modellierung von Arbeitsprozessen meist dem Benutzer. D.h. sie bieten keine oder nur einfache Korrektheitstests des Arbeitsablaufs an. Dadurch wird die Modellierung eines Workflows sehr fehleranfällig und die schnelle Veränderbarkeit der Prozesse eines Unternehmens ist nicht mehr gegeben.

1.1 Aufgabenstellung der Diplomarbeit

Diese Arbeit beschreibt Konzepte, Grundlagen und Implementierung eines komfortablen, graphischen Workflow-Editors, der den Modellierer bei der Definition eines Workflows unterstützt. D.h. der Editor hilft Fehler bei der Prozeßdefinition zu vermeiden, indem er Fehler in Kontroll- und Datenfluß erkennt und anzeigt.

Grundlage für den Editor WFEdit2 ist das an der Universität Ulm entwickelte ADEPT[1]-Basismodell [DKR95, DaR97]. Das ADEPT-Basismodell entstand im Rahmen des Forschungsprojekts „**O**ffenes **k**linisches Datenbank- und **I**nformationssystem zur Integration autonomer **S**ysteme“ (OKIS). Ein Hauptmerkmal des ADEPT-Basismodells ist seine formale Definition von Syntax und (Ausführungs-) Semantik.

Die Modellierung eines Workflows trennt das ADEPT-Basismodell in Kontroll- und Datenflußmodellierung. Zur Kontrollflußmodellierung stehen Konstrukte wie verschiedene Verzweigungen, Schleifen oder Parallelbearbeitung von Aktivitäten zur Verfügung. All diese Konstrukte haben das Konzept der symmetrischen Blockstrukturierung gemeinsam. Diese Konzept garantiert, daß jedes Konstrukt nur einen Start- und Endknoten besitzt und ermöglicht so die beliebig häufige Schachtelung dieser Konstrukte. Außerdem besitzt das ADEPT-Basismodell die Möglichkeit, Abweichungen und Kompensationen vorzumodellieren. Auch die Synchronisation von Aktivitäten in parallelen Zweigen eines Workflow-Graphen ist möglich.

Der Datenfluß im ADEPT-Basismodell wird über globale Prozeßvariablen, sogenannte Datenslots, modelliert. Dazu besitzt jede Aktivität typisierte Ein- und Ausgabeparameter, die über diese Prozeßvariablen miteinander verbunden werden und so Daten austauschen können.

Eine herausragende Eigenschaft des ADEPT-Basismodells ist die Möglichkeit, Prozesse bei der Ausführung zu modifizieren, ohne die Korrektheit des Prozesses zu verletzen. Das Einfügen einer neuen Aktivität in einen laufenden Workflow ist ein Beispiel für eine solche Änderung. Genauere Erläuterungen von dynamischen Änderungen finden sich z.B. in [Hen97, DaR97].

[1] ADEPT ist die Abkürzung für **A**pplication **D**evelopment Based On **E**ncapsulated **P**re-Modelled Process-**T**emplates.

1.2 Einordnung des Editors in das bestehende WfMS

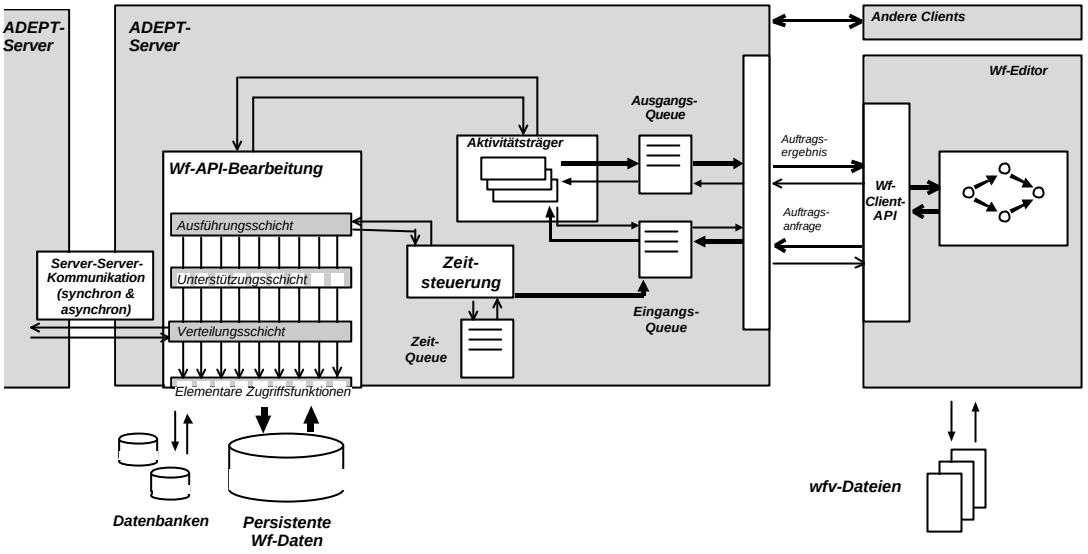


Abbildung 1: *Schematische Darstellung des WfMS der Abteilung DBIS*

Abbildung 1 stellt die Komponenten des WfMS der Abteilung DBIS[1] dar. Der Workflow-Editor WFEdit2 hat dabei die gleiche Stellung wie alle anderen Clients und wird über das WF-Client-API mit dem Workflow-Server verbunden. Ein anderer Workflow-Client ist beispielsweise eine Monitoring-Komponente, die zur Überwachung des Status aktueller Workflow-Instanzen (Arbeitsablauf-Monitoring) und zur Analyse von ausgeführten Workflow-Instanzen (Arbeitsablauf-Controlling) dient. Eine Aufgabe des Workflow-Servers ist die Speicherung von Vorlagen und Instanzen eines Prozesses in verschiedenen Datenbanken. Desweiteren realisiert er die Server-Server-Kommunikation.

1.3 Standardisierungen

Da sehr viele unterschiedliche Workflow-Management-Ansätze im wissenschaftlichen und kommerziellen Bereich vorhanden sind, wurde 1993 die Workflow-Management-Coalition (WfMC) gegründet. Die WfMC besteht aus Anbietern und Anwendern von Workflow-Management-Technologie. Die Ziele der WfMC sind:

- WfMS auf dem Markt zu verbreiten
- Das Risiko beim Einsatz von Workflow-Produkten zu minimieren

[1] DBIS = **D**aten**b**ank und **I**nformations-**S**ysteme

- Unterschiedliche Workflow-Produkte zu vereinheitlichen
- Standards für WfMS zu schaffen und deren Umgebung zu definieren
- Ein allgemein gültiges Referenzmodell zu entwickeln

Aufgrund der vielen Workflow-Management-Ansätze definierte die WfMC zuerst eine klare Begriffswelt. Die wichtigsten Begriffe in Zusammenhang mit der Modellierung von Workflows werden in Abschnitt 1.3.1 erläutert. Weitere Definitionen finden sich in dem Dokument [WMC96].

Ebenso definierte die WfMC ein Basismodell, das die wesentliche Charakteristik eines WfMS zum Ausdruck bringt und die Beziehungen zwischen den Funktionen eines WfMS verdeutlicht. Dieses Modell wird in Abschnitt 1.3.2 beschrieben.

1.3.1 Begriffe

Für den Bereich Workflow-Modellierung sind folgende Begriffe von Wichtigkeit:

- **Geschäftsprozeß**

Ein Geschäftsprozeß ist eine Menge von Aktivitäten, die gemeinsam ein Geschäftsziel verwirklichen. Dabei kann ein Geschäftsprozeß mehrerer Organisationseinheiten überspannen. Als Synonym für das Wort Geschäftsprozeß wird auch das Wort *Prozeß* benutzt.

- **Workflow**

Ein Workflow ist die Automatisierung eines Geschäftsprozesses mit Hilfe von Computern. Als Workflow wird auch die Gesamtheit alle ablaufender Aktivitäten bezeichnet

- **Workflow-Management-System (WfMS)**

Ein WfMS unterstützt mit seinen Komponenten die Entwicklung (Modellierungskomponente), die Steuerung und die Ausführung (Laufzeitkomponente) von Workflows. Die Hauptaufgabe eines WfMS ist die Steuerung des Arbeitsflusses zwischen den beteiligten Stellen nach den Vorgaben einer Ablaufspezifikation.

- **Prozeßdefinition**

Die Prozeßdefinition ist eine besondere Darstellung eines Geschäftsprozesses. Diese Darstellung ermöglicht die Modellierung bzw. Ausführung eines Geschäftsprozesses durch ein WfMS. Ein Synonym für Prozeßdefinition ist das Wort Prozeßvorlage. In dieser Arbeit wird eine Prozeßdefinition auch als *Workflow-Schema* bezeichnet.

- **Prozeßinstanz**

Als Prozeßinstanz wird eine bestimmte Ausführung einer Prozeßdefinition durch ein WfMS bezeichnet. Jede Instanz besitzt dabei einen eigenen Ausführungsstatus und eigene Daten.

- **Aktivität**

Eine Aktivität beschreibt einen logischen Schritt innerhalb eines Prozesses. Eine Aktivität kann dabei eine manuelle Aktivität, die durch eine Person ausgeführt wird, oder eine automatisierte Aktivität sein. Automatisierte Aktivitäten können durch einen Computer ausgeführt werden. Aktivitäten beschreiben meist eine durchzuführende Aktion. Diese Aktion kann manuell bzw. durch ein Programm ausgeführt werden. Synonyme für eine Aktivität sind die Wörter *Arbeitsschritt* oder *Schritt*. In Zusammenhang mit einem Graphen wird auch das Wort *Knoten* für eine Aktivität benutzt.

- **Aktivitäteninstanz**

Eine Aktivitäteninstanz ist die Darstellung einer Aktivität in einer bestimmten Ausführung eines Prozesses. Dadurch kann die Aktivität in unterschiedlichen Workflows jeweils unabhängig voneinander ausgeführt werden. Einer Aktivitäteninstanz wird bei der Ausführung eines Workflows ein bestimmter Ausführungszustand zugeordnet und jede Instanz besitzt weitere Daten, wie z.B. einen Akteur, der die Aktivität manuell ausführen kann.

- **Akteur**

Als Akteur wird eine Person bezeichnet, die zur Laufzeit die Arbeit einer Aktivitäteninstanz durchführt. Die WfMC verwendet für den Begriff Akteur den Ausdruck Workflow-Teilnehmer. Ein Synonym für Akteur ist das Wort *Bearbeiter*.

- **Rolle**

Für jede Aktivität wird vom Ersteller der Ablaufvorlage eine Gruppe von Akteuren definiert, die bestimmte Qualifikationen, Attribute bzw. Fähigkeiten besitzen. Diese Gruppe erfüllt die festgelegte Rolle einer Aktivität und kann diese ausführen. Die WfMC bezeichnet diesen Rollenbegriff als „*Organisatorische Rolle*“.

1.3.2 Die Referenzarchitektur für WfMS

Aufgabe der Referenzarchitektur der WfMC ist es, Schnittstellen zu definieren und zu standardisieren. Der Grund für die Definition der Schnittstellen ist die Forderung nach der Zusammenarbeit mehrerer verschiedener WfMS. Außerdem sollen Module eines WfMS durch Module eines anderen Herstellers ersetzt bzw. ergänzt werden können. Über den inneren Aufbau der Komponenten wird innerhalb der Referenzarchitektur nichts ausgesagt, da dies Sache der einzelnen Hersteller sein soll. Abbildung 2 zeigt das Referenzmodell der WfMC.

Zentraler Teil der Referenzarchitektur ist die Komponente zur Abwicklung und Koordination laufender Workflows (Workflow Enactment Service). Die Komponente kann durch eine oder mehrere Workflow-Engines realisiert sein.

Der Workflow-Enactment-Service besitzt fünf Schnittstellen mit denen verschiedene Komponenten angebunden werden können. Jedoch ist anzumerken, daß einige Schnittstellen noch nicht vollständig definiert sind.

Interface 1 definiert die Schnittstelle, mit der Workflow-Modellierungswerkzeuge an die zentrale Komponente angeschlossen werden können. Um Workflow-Schemata auszutauschen, muß eine allgemein verbindliche Workflow-Sprache definiert werden, die es erlaubt auch zwischen Modulen

verschiedener Hersteller Schemata auszutauschen. Aus diesem Grund entwickelt die WfMC die Workflow Process Definition Language (WPDL).

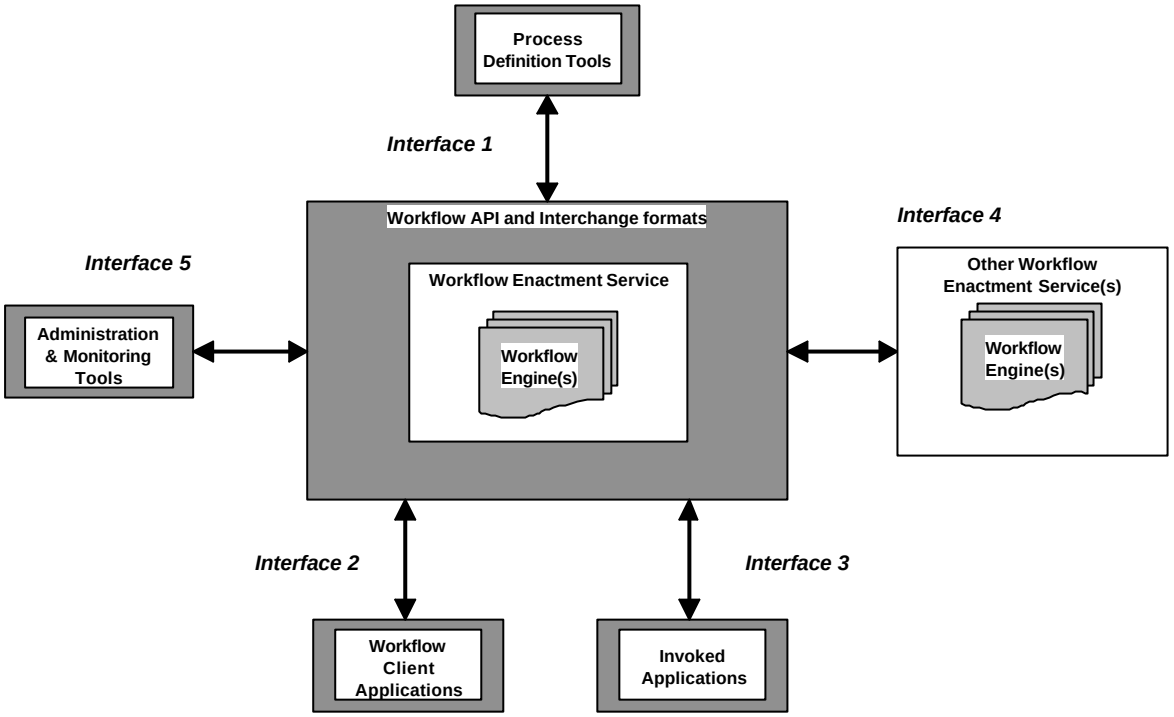


Abbildung 2: *WfMC-Referenzarchitektur*

Interface 2 ist für den Notifikationsdienst auf Clientseite spezifiziert worden. Ein Notifikationsdienst ist vergleichbar mit einem Posteingangskorb. Interface 2 ist notwendig, da ein Benutzer in der Realität meist nicht nur mit einem Notifikationsdienst verbunden ist.

Interface 3 ist die Schnittstelle für den Applikationsaufruf.

Interface 4 dient dem Austausch workflow-relevanter Daten zwischen verschiedenen Workflow-Engines.

Interface 5 bietet eine Schnittstelle, die es erlaubt, Informationen über den aktuellen Ausführungszustand von Workflows zu erfahren. Desweiteren bietet sie die Möglichkeit andere Administrations- und Monitoringwerkzeuge an die zentrale Komponente anzuschließen.

1.4 Gliederung der Arbeit

Die vorliegende Arbeit gliedert sich in folgende Kapitel:

In Kapitel 2 werden die Grundzüge der Geschäftsprozeß- und Workflowmodellierung erläutert. Dabei werden Möglichkeiten zur Modellierung von Prozessen aufgezeigt und existierende Geschäftsprozeß- und Workflow-Modellierungswerkzeuge werden mit ihren Modellierungsansätzen vorgestellt.

Das ADEPT-Basismodell wird als Grundlage des in dieser Arbeit entwickelten Workflow-Editors WFEdit2 in Kapitel 3 vorgestellt. Dabei werden nur die für den Editor wesentlichen Aspekte betrachtet.

In Kapitel 4 werden einige wichtige Algorithmen vorgestellt, die zur Implementierung des Editors notwendig waren. Zu diesen Algorithmen gehören beispielsweise das Verfahren zur Visualisierung des Prozeßgraphen und verschiedenen Korrektheitsprüfungen des Kontroll- und Datenflusses.

Die Modellierungsmöglichkeiten des Editors WFEdit2 werden in Kapitel 5 aufgezeigt. Dabei wird ausführlich auf die Bedienung des Editors eingegangen. Außerdem wird ein Schwerpunkt auf die Unterstützung der korrekten Workflow-Definition durch den Editor gelegt.

Die interne Struktur und die Zusammenarbeit der einzelnen Komponenten des Editors wird in Kapitel 6 beschrieben. Innerhalb des Kapitels wird auch die zugrundeliegende Architektur und das darauf basierende Package-Konzept des Workflow-Editors erläutert. Weitere Internas wie die wichtigsten Klassen und ihre Methoden werden ebenfalls erläutert.

Kapitel 7 gibt eine Zusammenfassung der Ergebnisse der Arbeit und eine Ausblick auf mögliche Erweiterungen des Editors, die zu einer komfortablen Modellierung eines Workflows notwendig sind.

2. Geschäftsprozeß- und Workflowmodellierung

In diesem Kapitel werden verschiedene Ansätze bzw. Formalismen zur Modellierung von Arbeitsabläufen sowie darauf basierende Werkzeuge beschrieben. Zuerst wird auf allgemeine Aspekte der Modellierung eingegangen. Anschließend werden in Abschnitt 2.2 verschiedene Ablaufbeschreibungssprachen vorgestellt. Dabei werden Stärken und Schwächen dieser Ansätze im Hinblick auf ihre Modellierungsmöglichkeiten näher betrachtet. In den nachfolgenden Abschnitten wird eine Trennung zwischen der Modellierung von Geschäftsprozessen und der Modellierung von Workflows durchgeführt. Gemeinsamkeiten, Unterschiede und Zusammenhänge dieser zwei verwandten Gebiete werden erläutert. Für beide Modellierungsaufgaben werden existierende Modellierungswerkzeuge, die zum Teil die Ablaufbeschreibungssprachen aus Abschnitt 2.2 verwenden, vorgestellt.

2.1 Grundlagen

Zuerst muß die Frage geklärt werden, was unter Ablaufmodellierung zu verstehen ist. Das Modellieren bedeutet in diesem Kontext, daß ein Modell eines Arbeitsvorgangs erzeugt wird. Ein Arbeitsvorgang ist ein Ablauf der realen Welt. Ein Modell ist eine Vorstellung von diesem Arbeitsvorgang und wird als Arbeitsablauf bezeichnet [JBS97]. Genauer betrachtet, ist dieses Modell eine abstrakte Darstellung der Realität, die viele Details ausklammert. Es werden Dinge ausgeschlossen, die das relevante Verhalten nicht beeinflussen. Daher zeigt ein Modell nur die Dinge, die der Entwerfer des Modells für notwendig erachtet, um das modellierte Phänomen zu verstehen. Ein gutes Modell fängt daher die entscheidenden Aspekte ein und läßt irrelevante Aspekte weg.

Der Zweck eines Modells besteht darin die Komplexität eines realen Arbeitsvorgangs zu reduzieren. Dies geschieht, um eine Untersuchung dieses Ablaufs zu erleichtern oder überhaupt erst zu ermöglichen. Die Fähigkeit zur Abstraktion ist eine fundamentale Fähigkeit, die es uns ermöglicht, mit Komplexität fertig zu werden.

Modelle dienen dem Erreichen eines bestimmten Ziels, wie z.B. der Optimierung eines Arbeitsablaufs. Daher ist die Modellierung ein iterativer Vorgang, der immer wieder ein Modell verändert, bis das Modell den bestehenden Anforderungen entspricht.

Da die Modellierung ein iterativer Vorgang ist, muß auch der Vorgang der Modellierung als solcher betrachtet werden. Um ein Modell mit anderen Personen zu diskutieren und zu verändern, müssen die verwendeten Modellierungskonzepte und Begriffe bekannt sein. Diese Anforderung führt zu einer Vorstellung des Modellierens, die als Metamodell bezeichnet wird. Eine anschauliche Darstellung dieses Metamodells wird Metaschema genannt. [JBS97] definiert den Begriff Metaschema folgendermaßen: Ein Metaschema definiert Konzepte, die bei der Erstellung und Handhabung der Problemschemata verwendet werden. Metaschemata werden heute nicht nur bei der Modellierung von Arbeitsabläufen benötigt, sondern auch bei der Modellierung von Datenbanken, wo Entity-Relationship-Diagramme als Metaschema bezeichnet werden können.

Um bei der Modellierung eines Arbeitsablaufs zusätzlich eine Animation oder Simulation durchzuführen, reicht die Strukturbeschreibung eines Metaschemas nicht aus. Zusätzlich muß die

Ablaufsemantik der durch das Metamodell generierbaren Arbeitsabläufe beschrieben werden. Die Ablaufsemantik beschreibt, in welcher Weise die modellierten Abläufe ausgeführt werden können.

Für die Modellierung von Arbeitsabläufen gibt es heute viele Gründe. Die wichtigsten sind:

- *Ausführung des Ablaufs durch ein WfMS*
  Die Modellierung eines Arbeitsablaufs wird durchgeführt, um diesen Arbeitsablauf rechnerunterstützt durchzuführen. Dazu ist eine genaue formale Definition des Ablaufs und der Daten notwendig.

- *Analyse und Reorganisation*
  Ein Modell eines Arbeitsablaufs wird erzeugt, um eine Reorganisation und evtl. eine Optimierung eines Ablaufs durchführen[1]. Dabei ist der Zweck des Modells, mit Anwendern Möglichkeiten der Neugestaltung und Optimierung auf sachlicher Ebene zu diskutieren.

- *Dokumentation*
  Die Ablaufbeschreibung dient in diesem Fall nur dem Zweck einen realen Ablauf zu dokumentieren. Solche Dokumente sind z.B. Organisationsrichtlinien und Handlungsanweisungen. Dabei kann jedes Modell je nach Zweck eine andere Abstraktionsebene besitzen.

Trotz diesen unterschiedlichen Verwendungszwecken von Ablaufmodellen liegt der Modellierung eines Arbeitsablaufs eine einfache Fragestellung zugrunde. Die meisten Personen möchten primär aus einem Arbeitsablauf erfahren, wer, was, wann, wie tut bzw. tun soll. Aus diesem Grund wird die Modellierung eines vollständigen Arbeitsablaufs in drei Teile gespalten [CKO92]:

1. *Modellierung des Kontrollflusses (Ablauforganisation):*

   Die Modellierung des Kontrollflusses legt fest, was wann getan wird bzw. getan werden soll. Dazu müssen die einzelnen Funktionen identifiziert werden und in eine Reihenfolge gebracht werden.

2. *Modellierung des Datenflusses*

   Die Modellierung des Datenflusses legt fest, welche Daten die Funktionen des Kontrollflusses benötigen bzw. erzeugen.

3. *Modellierung der Organisation (Aufbauorganisation):*

   Bei der Aufbauorganisation werden die Benutzer mit ihren Rollen definiert. Mit Hilfe der Rollen werden die Benutzer den Funktionen im Kontrollfluß zugeordnet. Ein Benutzer kann dabei verschiedene Rollen annehmen.[2]

Um einen Arbeitsablauf zu modellieren, müssen nicht immer alle drei Teilbereiche spezifiziert werden. So kann beispielsweise ein Arbeitsablauf auch nur aus dem Kontrollfluß bestehen. Welche Teilbereiche modelliert werden müssen, hängt von den Anforderungen ab, die an das Modell gestellt werden. Dies bedeutet ebenfalls, daß je nach Anwendung auch weitere Modellaspekte wie beispielsweise Zeit bzw. Ressourcen wichtig sein können.

[1] Dieses Einsatzgebiet der Ablaufmodellierung wird auch als „Business Process Reengineering“ bezeichnet.
[2] Die Aufbauorganisation wird in dieser Arbeit nicht weiter betrachtet, da dieses Thema in einer anderen Diplomarbeit betrachtet wird.

Die Modellierung eines Arbeitsablaufs wird meistens durch eine Diagrammsprache unterstützt. Der Arbeitsablauf wird dabei durch graphische Symbole beschrieben. Graphische Symbole haben eine hohe Anschaulichkeit und Selbsterklärungsfähigkeit. Dadurch kann ein Arbeitsablauf leichter diskutiert werden. Ein weiterer Vorteil ist, daß auch Experten, die ein großes Wissen über den realen Ablauf besitzen, zur Spezifikation des Arbeitsablaufs hinzugenommen werden. Dies ist trotz der Unkenntnis der Modellierungsmethode möglich.

Eine weitere Möglichkeit, einen Arbeitsablauf zu beschreiben, ist die Verwendung einer Spezifikationssprache. Diese Möglichkeit existiert vor allem bei der Modellierung von Workflows, die durch ein WfMS koordiniert werden sollen. Die Beschreibung durch eine Spezifikationssprache wird benötigt, um einen Workflow innerhalb eines WfMS darzustellen. Die Definition eines Ablaufes ist dadurch ähnlich der Programmierung einer Anwendung. Eine Spezifikationssprache muß programmiersprachliche Konstrukte zur Definition eines Ablaufs besitzen. Ein Nachteil gegenüber den Diagrammsprachen ist die textuelle Darstellung des Arbeitsablaufs, die es erfordert, daß alle an der Modellierung beteiligten Personen die Modellierungsmöglichkeiten der Spezifikationssprache kennen. Dies erschwert die Hinzunahme von Experten aus anderen Bereichen, die zwar mit dem zu spezifizierenden Sachverhalt vertraut sind, aber keine Kenntnisse der Modellierung haben. Ein Beispiel für eine Spezifikationssprache ist die FlowMark Definition Language (FDL), die in dem WfMS der Firma IBM [LeA94] verwendet wird.

2.2 Repräsentationsformen von Arbeitsabläufen

Arbeitsabläufe müssen nicht immer explizit definiert werden. Vielmehr können Repräsentationen von Arbeitsabläufen zwei allgemeinen Ansätzen zugeordnet werden. Diese Ansätze sind die schon erwähnte explizite und die implizite Definition. Bei der expliziten Definition wird eine genaue Beschreibung des Kontrollflusses z.B. durch eine Ablaufvorlage erstellt. Die implizite Definition dagegen verwendet ein anderes Konzept. Hier wird die Ablaufreihenfolge implizit, beispielsweise durch eine Menge von Ausführungsregeln festgelegt. Einige Vertreter dieser Ansätze werden im nachfolgenden Abschnitt detaillierter vorgestellt.

Zur **impliziten** Definition von Arbeitsabläufen gehören:

- Regelbasierte Ansätze
- Formularbasierte Ansätze

Zur **expliziten** Definition von Arbeitsabläufen gehören:

- Netzbasierte Verfahren
- State- und Activity-Charts
- Blockbasierten Verfahren.
- Flußdiagramme

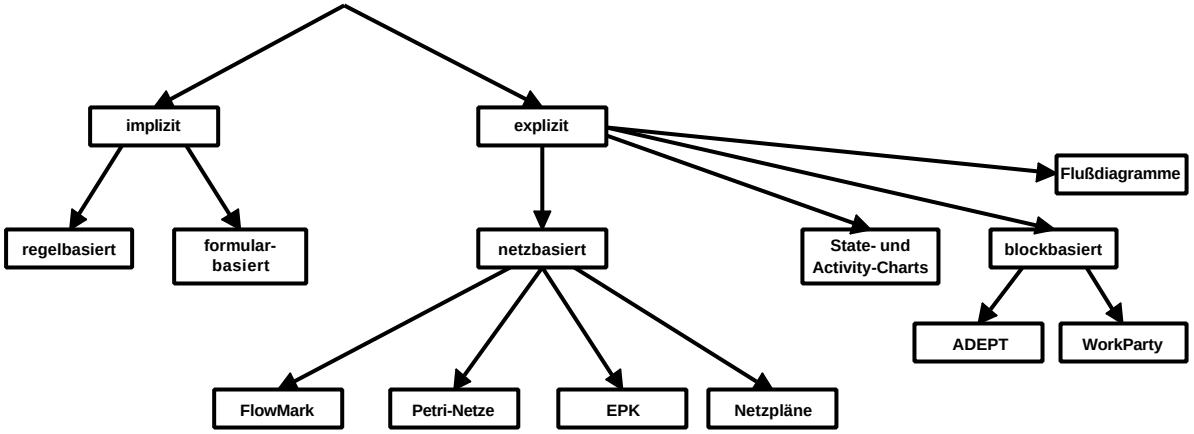


Abbildung 3: *Mögliche Repräsentationsformen von Abläufen*

2.2.1 Die implizite Modellierung von Arbeitsabläufen

- **Regelbasierte Verfahren**

Eine regelbasierte Beschreibung eines Prozesses besteht aus einer Menge von ausführbaren Regeln. Ein auf dieser Beschreibung basierendes WfMS wertet bei Auftreten eines Ereignisses (z.B. Änderung im Status einer Aktivität) die Menge der Regeln aus und führt eine bestimmte Aktion (z.B. Start einer Aktivität aus). Die häufigste Regelbeschreibungsart ist das Event/Condition/Action (ECA)-Modell. Eine Regel setzt sich in diesem Modell aus einem Ereignis, einer Bedingung und einer Aktion zusammen. Bei der Ausführung eines Prozesses prüft die Laufzeitkomponente zuerst welches Ereignis eingetreten ist. Daraufhin wird eine Bedingung, wie z.B. die Belegung von Prozeßvariablen mit bestimmten Werten, geprüft. Ergibt diese Prüfung als Ergebnis den boolschen Wert „Wahr“, so wird die in dieser Regel spezifizierte Aktion ausgeführt.

Die Mächtigkeit der Beschreibungsmöglichkeiten durch das ECA-Modell ist sehr groß, da verschiedene Ereignisse und Bedingungen kombiniert werden können. Außerdem unterscheidet sich das ECA-Modell von anderen regelbasierten Beschreibungsmöglichkeiten dadurch, daß bei der Ausführung eines Arbeitsablaufs nicht alle Ausführbarkeitsbedingungen der Regeln evaluiert werden müssen.

Trotz dieser Vorteile eignen sich regelbasierte Systeme schlecht für die Modellierung von Arbeitsabläufen. Der größte Nachteil regelbasierter Ablaufbeschreibungen ist, daß die formale Analyse und Validation durch die implizite Modellierung von Kontroll- und Datenfluß des Arbeitsablaufs äußerst schwierig ist. Daneben erzeugen komplexe Prozesse eine unübersichtliche Regelmenge, die von einem Modellierer kaum noch zu überblicken und zu pflegen ist. Die Einführung von temporalen Abhängigkeiten verschärft diese Situation noch. Durch diese hohe Komplexität eines Prozesses sind Änderungen der Ablauflogik schwierig durchzuführen und führen häufig zu Fehlern. Solche Fehler sind z.B. unerwünschte Regelaktionen, die bei der Ausführung des Prozesses zu Seiteneffekten oder Verklemmungen führen. Schwierigkeiten bereitet auch das Hinzufügen von neuen Regeln, da evtl. bestehende Regeln aufgrund neuer Ereignisse bzw. geänderter Bedingungen modifiziert werden müssen. Ein weiteres Problem regelbasierter Systeme ist, daß Effekte aus der Ausführung mehrerer Regeln hintereinander für den Modellierer äußerst schwer nachzuvollziehen sind.

Durch diese Nachteile müssen daher bei regelbasierten Ablaufbeschreibungen Punkte wie Strukturiertheit und Korrektheit ausgeklammert werden.

- **Formularbasierte Verfahren**

Bei formularbasierten Ansätzen zur Prozeßbeschreibung stehen die Anwendungsdaten, die bei einem einzelnen Prozeßschritt anfallen, im Vordergrund. Ein Prozeß wird in diesem Kontext als Dokument betrachtet, in dem alle Anwendungsdaten eines Prozesses gehalten werden. Dieses Dokument wird in einer gemeinsamen Datenbank gespeichert und den verschiedenen Akteuren über verschieden Bildschirmformulare angezeigt. Welche Daten des Dokuments den einzelnen Akteuren angezeigt werden, läßt sich bei der Modellierung der Formulare des Prozesses bestimmen.

Prozesse werden modelliert, indem der Ablauf durch Steuer- und Statusfelder wie „nächster Bearbeiter“ oder „nächster Bearbeitungsschritt“ im Dokument festgelegt wird. Das Weiterschalten im Arbeitsablauf erfolgt durch das Ausführen bestimmter Formeln oder Skripte, die bei Beendigung der Arbeit in einem Formular aktiviert werden. Die im Formular enthaltene Formel bestimmt dann im Dokument, welcher Schritt als nächster ausgeführt werden soll. Die Aktivierung kann z.B. durch eine Schaltfläche oder durch Schließen des Formulars geschehen.

Der Hauptnachteil dieses Ansatzes ist wie bei Regeln die implizite Definition des Arbeitsablaufs. Da die Bestimmung des nächsten Schrittes in jedem Formular „versteckt“ ist, sind Änderungen in der Ablaufreihenfolge der Formulare aufwendig und schwer durchzuführen. Aus diesem Grund entstehen aus solchen Änderungen häufig Fehler.

Da das System keine Kenntnisse über den Ablauf des Prozesses besitzt, kann es auch keine Analyse des Kontroll- und Datenflusses durchführen. Dadurch kann der Modellierer bei der Spezifikation eines Arbeitsablaufs vom System nicht unterstützt werden. Beispielsweise ist es so möglich, daß es bei der parallelen Bearbeitung von Daten in verschiedenen Formularen zu Schreibkonflikten und Lost Updates kommen kann.

Formularbasierte Ablaufbeschreibungen eignen sich nur zur Spezifikation von Arbeitsabläufen mit geringer Komplexität, da die Implementierung und Wartung mit ansteigender Größe immer aufwendiger wird.

Die Realisierung dieses Konzeptes findet man z.B. in Groupware-Systemen. Lotus Notes [ReM96] ist ein Beispiel für die formularbasierte Modellierung eines Arbeitsablaufs in einem Groupware-System.

2.2.2 Die explizite Modellierung von Arbeitsabläufen

- **Netzbasierten Verfahren**

Das wohl am häufigsten benutzte netzbasierte Verfahren zur Modellierung von Arbeitsabläufen sind **Petri-Netze** [RoW91, Bau90, Obe96]. Petri-Netze sind ein graphischer Formalismus zur Spezifikation und Analyse von Systemen. Sie ermöglichen die Beschreibung sequentieller, sich gegenseitig ausschließende sowie nebenläufiger (voneinander unabhängiger) Aktivitäten. Dabei werden nur wenige graphische Beschreibungskonstrukte benötigt: Kreise zur Repräsentation von Zuständen oder statischen Aspekten (z.B. Dokumente, Ressourcen, Daten) und Vierecke zur Repräsentation von Ereignissen, Aktivitäten oder lokalen Zustandsübergängen [Obe96].

Formal ist ein Petri-Netz ein Tripel N = (S,T,F), für das gilt:

1. S, T endliche Mengen
2. $S \cap T = \emptyset$
3. $S \cup T \neq \emptyset$
4. $F \subseteq (S \times T) \cup (T \times S)$

Die Elemente aus S werden Stellen (graphische Darstellung: Kreise) und die Elemente aus T werden Transitionen (graphische Darstellung: Vierecke) genannt. Beide zusammen werden als Knoten bezeichnet. F ist die Flußrelation von N. Diese Flußrelation beschreibt die in der graphischen Darstellung vorhandenen Pfeile zwischen Kreisen und Vierecken.

Petri-Netze sind in ihrer allgemeinen Form nichts anderes als ein bipartiter Graph. Allerdings können nicht alle Knoten untereinander verbunden werden, sondern nur Stellen mit Transitionen bzw. Transitionen mit Stellen (siehe Abbildung 4).

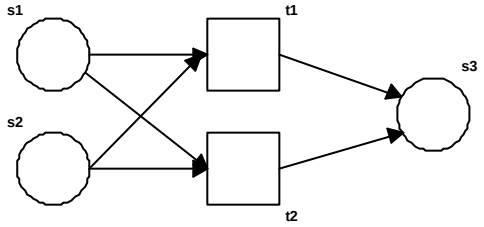


Abbildung 4: *Einfaches Petri-Netz*

Viele Systeme für die Prozeßmodellierung basieren auf der Theorie der Petri-Netze. In der Grundform sind Petri-Netze aber ein „low-level“ Konzept. Aus diesem Grund sind die allgemeinen Petri-Netze weiterentwickelt worden. Diese Weiterentwicklung reicht bis zur Entwicklung von „höheren“ Petri-Netzen wie z.B. FUNSOFT-Netzen (siehe Abschnitt 2.4.1). Dabei ist zu anzumerken, daß alle diese Verfahren nur die elementaren Elemente Stelle und Transition benutzen. Dieses Festhalten an diesen Elementen erschwert die Modellierung bestimmter Sachverhalte von Arbeitsabläufen.

Eine einfache Art von Petri-Netzen, die ein dynamisches Verhalten darstellen können, sind *Bedingungs/Ereignis-Netze.* In diesen Petri-Netzen ist für jeden Zustand nur maximal eine Marke erlaubt. Aus diesem Grund werden diese Netze auch „Ein-Marken-Petri-Netze“ genannt. Die Stellen in Bedingungs/Ereignis-Netzen werden Bedingungen genannt, die entweder wahr (der Kreis ist markiert) oder falsch (der Kreis ist leer) sein können. Die Transitionen repräsentieren Ereignisse, die stattfinden können, wenn alle Eingangsbedingungen markiert und alle Ausgangsbedingungen unmarkiert sind. Wenn ein Ereignis stattfindet, d.h. eine Transition schaltet, so werden alle Marken aus den Eingangsbedingungen entfernt und alle Ausgangsbedingungen werden mit Marken versehen.

Stellen/Transitions-Netzen arbeiten im Gegensatz zu Bedingungs/Ereignis-Netzen mit mehreren Marken. Die Stellen werden dabei als Objektbehälter für „anonyme“ Marken betrachtet. Jede Stelle besitzt eine Kapazitätsgrenze, die angibt, wieviel Marken zu einem Zeitpunkt in einer Stelle vorkommen können. Diese Obergrenze kann auch unendlich sein. Zusätzlich hat jede Kante eine Kantengewichtsfunktion, die jeder Kante eine natürliche Zahl zuordnet. Transitionen können in

Stellen/Transitions-Netzen schalten, wenn in jeder Eingangsstelle genügend Marken entsprechend der jeweiligen Kantengewichtung vorhanden sind und wenn in jeder Ausgangsstelle genügend Platz für die zusätzlichen Marken sind.

Bei Bedingungs/Ereignis-Netzen und Stellen/Transitions-Netzen sind die verwendeten Marken nicht unterscheidbar, d.h. die Marken in diesen Petri-Netzen sind weder identifizierbar noch enthalten sie Informationen. Aus diesem Grund können Stellen/Transitions-Netze nur den quantitativen Fluß von Ressourcen modellieren, während Bedingungs/Ereignis-Netze nur bei der kausalen Systemanalyse Verwendung finden.

Diese vorher genannten Nachteile werden durch *Prädikat/Transitions-Netze* vermieden. Prädikats/Transitions-Netze werden zu den „höheren“ Petri-Netze gezählt. Die in diesen Netzen verwendeten Marken tragen Informationen. Daneben besitzt jede Marke eine Struktur und eine Identifikation. Die Transitionen sind mit Vor- und Nachbedingungen versehen, deren freie Variablen durch die Marken beim Schalten der Transition aktualisiert werden. Dadurch ist es möglich, daß eine Transition erst bei Vorliegen einer bestimmten Marke schaltet. Beim Vorgang des Schaltens werden die Marken im Eingangsbereich verbraucht und neuen Marken für den Ausgangsbereich erzeugt. Prädikat/Transitions-Netze erlauben auch die Modellierung von bedingten Verzweigungen.

Petri-Netze besitzen eine sehr große Ausdrucksmächtigkeit bezüglich der statischen Struktur und des dynamischen Systemverhaltens. Durch die mathematische Basis kann eine formale Analyse bestimmter Eigenschaften der Petri-Netze durchgeführt werden [AAH98]. Grundlagen für diese Analysen sind der Erreichbarkeitsgraph und die Inzidenzmatrix. Ein Erreichbarkeitsgraph ist ein gerichteter Graph, dessen Knoten die Markierungen darstellen, die von einer Anfangsmarkierung durch Schalten von Transitionen erreicht werden können. Die Inzidenzmatrix stellt Stellen und Transitionen eines Petri-Netzes als Zeilen und Spalten einer Matrix dar und enthält als Matrixelemente die Gewichte des Markenflusses zwischen den Stellen und Transitionen. Aus diesen beiden Strukturen können wichtige formale Netzeigenschaften abgeleitet werden.

In Petri-Netzen können beispielsweise Eigenschaften wie Lebendigkeit, Erreichbarkeit und Sicherheit analysiert werden. Unter Lebendigkeit ist folgendes zu verstehen: Eine Transition ist lebendig, wenn es für alle möglichen Markierungen eine Folgemarkierung gibt, in der die Transition aktiviert ist. Daraus folgt, daß ein Petri-Netz lebendig ist, wenn alle Transitionen lebendig sind. Desweiteren kann die Sicherheit eines Petri-Netzes bestimmt werden, indem analysiert wird, ob für ein Netz mit einer Anfangsmarkierung gilt, daß es für alle möglichen folgenden Markierungen keinen Markenüberfluß gibt. Auch läßt sich die Erreichbarkeit einer Markierung von einer gegebenen Markierung aus bestimmen. Dabei muß sich die Markierung durch wiederholtes Schalten von aktivierten Transitionen erreichen lassen.

Ein Nachteil dieser Analysealgorithmen ist, daß sie sehr aufwendig sind, da die kombinatorische Vielfalt der Markierungen eines Petri-Netzes sehr groß sein kann. Ist diese Vielfalt unendlich groß, so müssen manche Verfahren mit einem Überdeckungsgraph arbeiten. Ein weiterer Nachteil der Petri-Netze ist, daß sie bei zunehmender Größe sehr schnell unübersichtlich werden. Aus diesem Grund wurden Möglichkeiten zur Vergröbergung bzw. Verfeinerung eingeführt. Dabei müssen allerdings auch hier die Strukturvorschriften der Petri-Netze eingehalten werden. Bei der Darstellung eines Ablaufs eines Petri-Netzes zeigt sich ein weiteres Manko. Petri-Netze können nur ihren momentanen Systemzustand anzeigen. Der Ausführungspfad kann daher nicht rekonstruiert werden.

Um weitere Aspekte von Arbeitsabläufen darstellen zu können, wurden Petri-Netze um Zeitinformationen erweitert. Erweiterungen von Petri-Netzen um Zeitaspekte sind beispielsweise Timer-Netze [Gri97]. Um vormodellierte Ausnahmen zu unterstützen, müssen Petri-Netze ebenfalls erweitert werden.

Ein weiteres netzbasiertes Verfahren ist die Darstellung eines Arbeitsablaufs durch **Ereignisprozeßketten** (EPK) [Wäc95]. EPK sind gerichtete, bipartite Graphen mit Ereignissen und Funktionen als Knotentypen. Abbildung 5 zeigt eine EPK einer Chemotherapie [SMM95]. Ein Ereignis beschreibt das Eintreten eines Zustands, von dem der weitere Verlauf des Prozesses abhängt. Ereignisse können mehrere Funktionen auslösen und auch das Resultat einer oder mehrerer Funktionen sein. Graphisch werden Ereignisse als Sechsecke dargestellt.

Die Funktionen der EPK dienen zur Verarbeitung von Input- und Output-Daten. Funktionen werden immer durch Ereignisse ausgelöst und werden als Vierecke mit abgerundeten Ecken visualisiert. EPK unterstützen die Modularisierung eines Prozesses durch Funktionen, die Subworkflows (hierarchische Verfeinerung) darstellen und durch Prozeßwegweiser, die auf einen weiteren Prozeßteil zeigen (horizontale Unterteilung).

EPK besitzen drei Arten von Verknüpfungsoperatoren: „AND“, „OR“ und „XOR“. Diese Operatoren werden zur Modellierung von nichtsequentiellen Abläufen benutzt. Dabei ist zu bemerken, daß die Operatoren im EPK-Modell sowohl als Verteiler als auch als Verbinder genutzt werden. Noch komplexere Verzweigungen können über sogenannte Entscheidungstabellen realisiert werden. Dazu wird bei EPK ein ET-Operator benutzt.

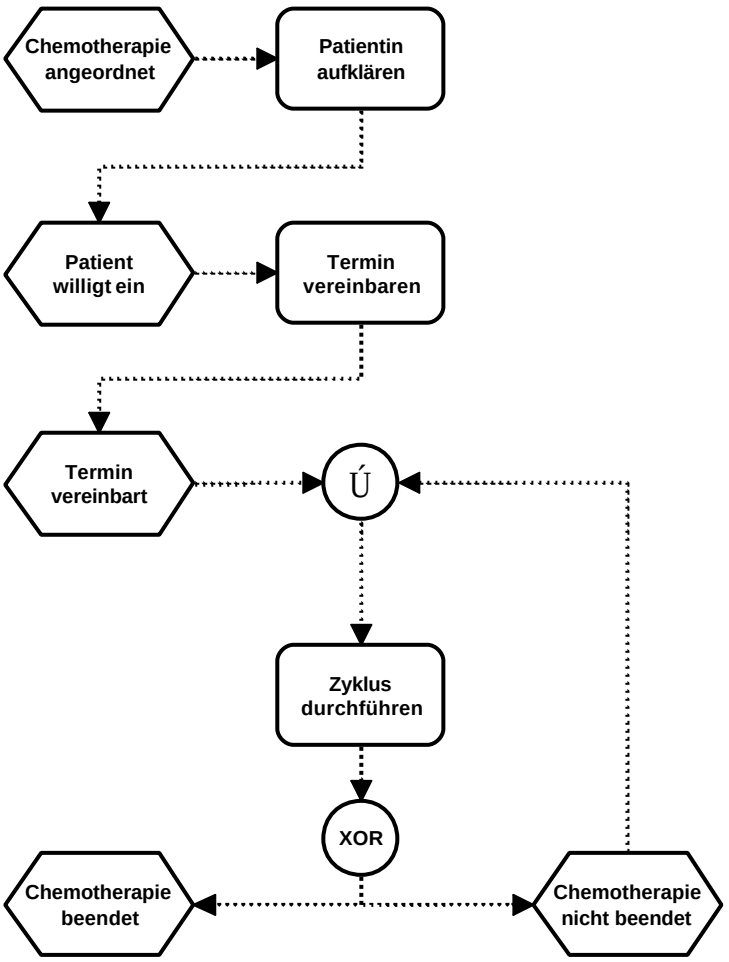


Abbildung 5: *Ereignisprozeßkette (EPK)*

Alle vorgestellten Elemente werden bei EPK über gerichtete Kanten verknüpft. Diese Elemente und Kanten bilden den Kontrollfluß. Der Datenfluß läßt sich mit EPK explizit modellieren. Für jede Funktion lassen sich zwar Ein-und Ausgabedaten angeben, aber dies erschwert die Überprüfung der Korrektheit des Datenflusses. Neben dieser Angabe der Nutzdaten können einer Funktion Organisationseinheiten zugeordnet werden. Dabei unterstützen EPK nicht den Rollenbegriff.

Ein Nachteil der EPK ist, daß Zeiträume zwischen zwei Funktionen nicht modelliert werden können. Zeiträume müssen daher durch entsprechende Ereignisse nachgebildet werden.

Zu den netzbasierten Verfahren wird auch die **Netzplantechnik** gezählt. Sie ist ein Verfahren zur Planung von Arbeitsabläufen und wurde ursprünglich nur für die Zeit- und Terminplanung eingesetzt. Im Laufe der Zeit kamen aber Gebiete wie die Ressourcen- und die Kostenplanung hinzu. Die Netzplantechnik wird in dieser Arbeit nicht weiter betrachtet. Eine Beschreibung der verschiedenen Methoden der Netzplantechnik findet sich in [NeM93].

- **State- und Activity-Charts**

State- und Activity-Charts wurden zur Beschreibung von reaktiven System wie Echtzeitsystemen entwickelt. Sie erlauben eine saubere Trennung von Kontroll- und Datenfluß. Ein weiterer Vorteil der State- und Activity-Charts ist, daß die Modellierung eines Systems graphisch vorgenommen werden kann.

Activity-Charts betrachten die funktionale Sicht eines Systems. Sie spezifizieren den Datenfluß zwischen Aktivitäten durch einen gerichteten Graphen, wobei die Datenelemente die Beschriftungen der Kanten sind. Activity-Charts können durch Subaktivitäten hierarchisch verfeinert werden. Activity-Charts machen keine Aussage über die Ausführungsreihenfolge und die Häufigkeit der Ausführung einer Aktivität. Daneben kann aus einem Activity-Chart auch nicht auf Nebenläufigkeit oder die Dauer der Ausführung geschlossen werden.

Die Ablauflogik des Systems wird durch State-Charts beschrieben. State-Charts basieren auf der Theorie der erweiterten endlichen Automaten. Diese Theorie wurde für die Modellierung von Abläufen um verschiedene Konzepte erweitert. Die wichtigsten Erweiterungen sind die Verfeinerung durch hierarchische Zerlegung in Subautomaten und die Nebenläufigkeit. Ein State-Chart wird graphisch als Zustandsgraph mit Transitionen beschrieben.

Diese Transitionen werden in der Form von Event/Condition/Action (ECA)-Regeln definiert. Im Gegensatz zu den regelbasierten Definitionen sind die ECA-Regeln eines State-Charts in das Zustands-Transitions-Netz eingebettet. D.h. eine Transition mit der Beschreibung E[C]A wird nur dann aktiviert, wenn der Quellzustand betreten, das Ereignis E aufgetreten ist und die Bedingung C erfüllt ist. In diesem Fall wird die Aktion A ausgeführt. Die Aktion A kann dabei der Start einer Aktivität oder die Generierung eines Ereignisses sein.

Bei der Modellierung eines Arbeitsablaufs durch State- und Activity-Charts wird folgendermaßen verfahren: Zuerst werden die Aktivitäten und der Datenfluß zwischen ihnen in den Activity-Charts festgelegt. Anschließend wird ausgehend von einem Activity-Chart der Kontrollfluß in einem State-Chart modelliert.

Ein Vorteil der State- und Activity-Charts ist, daß ihnen eine formal definierte operationale Semantik zugrundeliegt [JBS97]. Dadurch ergibt sich die Semantik eines Workflow-Schemas nicht erst durch die Implementierung einer Laufzeitumgebung eines WfMS. Ebenso kann die Korrektheit eines durch State- und Activity-Charts modellierten Ablaufs verifiziert werden. Systeme, wie z.B. Statemate [BES96], erlauben statische und dynamische Test. Mögliche statische Test sind die Suche nach

Syntaxfehlern, unvollständig deklarierten Variablen und unbeschrifteten Transitionen. Dynamische Test des Systemverhaltens erlauben die Deadlockerkennung, die Ermittlung nicht deterministischer Übergänge zwischen Zuständen und die Prüfung der Erreichbarkeit von Zuständen. Außerdem können parallele Schreiboperationen auf eine Variable gefunden werden.

- **Blockbasierte Verfahren**

Blockbasierte Verfahren beruhen auf der Idee der symmetrischen Blockstrukturierung. Dieses Konzept orientiert sich an den von Programmiersprachen bekannten Kontrollstrukturen (Verzweigungen, Konstrukte zur parallelen Ausführung von Aktivitäten, etc.). Jedes Konstrukt besitzt bei einem blockbasierten Beschreibungsverfahren genau einen Eingangs- und Ausgangsknoten. Dadurch können diese Konstrukte einfach geschachtelt werden.

Zu den blockbasierten Verfahren gehören die Modellierungsmöglichkeiten des ADEPT-Modells bzw. ADEPT-Basismodells. Beide Modelle wurden allerdings um wichtige Konstrukte erweitert, etwa zur Synchronisation nebenläufiger Aktivitäten und zum Ausdruck von Ausnahme- und Fehlerbehandlung. Das ADEPT-Basismodell wird in vereinfachter Form in Kapitel 3 erläutert. Daneben wendet Siemens WorkParty dasselbe Konzept an. Das Verfahren zur Erstellung eines Workflows durch die Modellierungskomponente von WorkParty wird in Abschnitt 2.4.1 beschrieben.

- **Flußdiagramme**

Flußdiagramme ermöglichen die graphische Darstellung von Abläufen. Sie erlauben die gute Visualisierung von Aspekten des Kontrollflusses mit sequentiellen und bedingten Verzweigungen. Flußdiagramme sind beispielsweise Struktogramme oder Nassi-Shneiderman-Diagramme. Nachteile der Flußdiagramme sind die fehlenden Möglichkeiten zur Repräsentation dynamischen Verhaltens. Ebenso können Flußdiagramme nicht formal auf ihre Gültigkeit geprüft werden Eine genauere Beschreibung der Flußdiagramme wird in [Rei93] gegeben.

2.3 Geschäftsprozeßmodellierung

Die Geschäftsprozeßmodellierung ist meistens ein Teil der Geschäftsprozeßoptimierung (GPO). Ein Geschäftsprozeß ist ein Prozeß, der sich an der betrieblichen Wertschöpfungskette des Unternehmens orientiert. Ein Beispiel für einen Geschäftsprozeß ist die Auftragsabwicklung vom Auftragseingang bis zur Zahlung. Bei der GPO werden Prozesse erfaßt, analysiert und ggf. optimiert. Mögliche Ziele dabei sind die Effizienzsteigerung bzgl. der Kosten und Zeit oder die Qualitätverbesserung der Produkte.

Die GPO teilt sich in drei Phasen. In der ersten Phase wird der zu analysierende Prozeß identifiziert. D.h. der Prozeß wird zu anderen Prozessen abgegrenzt und Zielvorgaben für seine Optimierung werden fixiert. In der zweiten Phase wird der Ist-Zustand des Prozesses erfaßt und Schwachstellen werden lokalisiert. Auf der Grundlage des erfaßten Prozesses wird ein Soll-Konzept entwickelt. Die Schwachstellen werden durch Vergleich des Soll-Konzeptes mit dem Ist-Zustand erkannt. Bei dieser Arbeit können GPO-Werkzeuge eingesetzt werden, um einen Prozeß in ein formales Modell zu überführen. Das formale Modell stellt dabei verschiedene Aspekte des Geschäftsprozesses, wie die Ablauflogik, die Informationsflüsse, die Funktionen oder die Organisation, dar. GPO-Werkzeuge

erlauben die Modellierung, Simulation und Analyse dieser Aspekte. Sie benutzen meist eine graphische Beschreibung der Prozesse, damit die erstellten Modelle auch mit Personen aus anderen Bereichen des Unternehmens diskutiert werden können. Basierend auf der Simulation und Analyse des Geschäftsprozesses durch das GPO-Werkzeug wird ein optimierter Prozeß generiert und anschließend können in der dritten Phase der GPO die gefunden Erkenntnisse auf den realen Prozeß übertragen werden.

Auf dem Markt gibt es viele GPO-Werkzeuge, die unterschiedliche Anforderungen erfüllen. Sie unterscheiden sich durch ihre Methoden und den Simulations- bzw. Analysemöglichkeiten. Auch bieten verschiedene GPO-Werkzeuge unterschiedliche Schnittstellen zu WfMS an. Welches GPO-Werkzeug benutzt werden kann, hängt von der Aufgabe ab und muß vor einer GPO geprüft werden.

2.3.1 Modellierungsmöglichkeiten kommerzieller GPO-Werkzeuge

- **ARIS**

Das ARIS-Toolset [Sch96, Sch98] ist eine der bekanntesten und am weitesten verbreiteten Lösungen für den Bereich der Modellierung und Gestaltung von Geschäftsprozessen. Ein Grund für die Verbreitung des ARIS-Toolset ist die Verbindung des Produkts mit Standardsoftware wie SAP R/3. ARIS stammt von der Firma IDS Prof. Scheer.

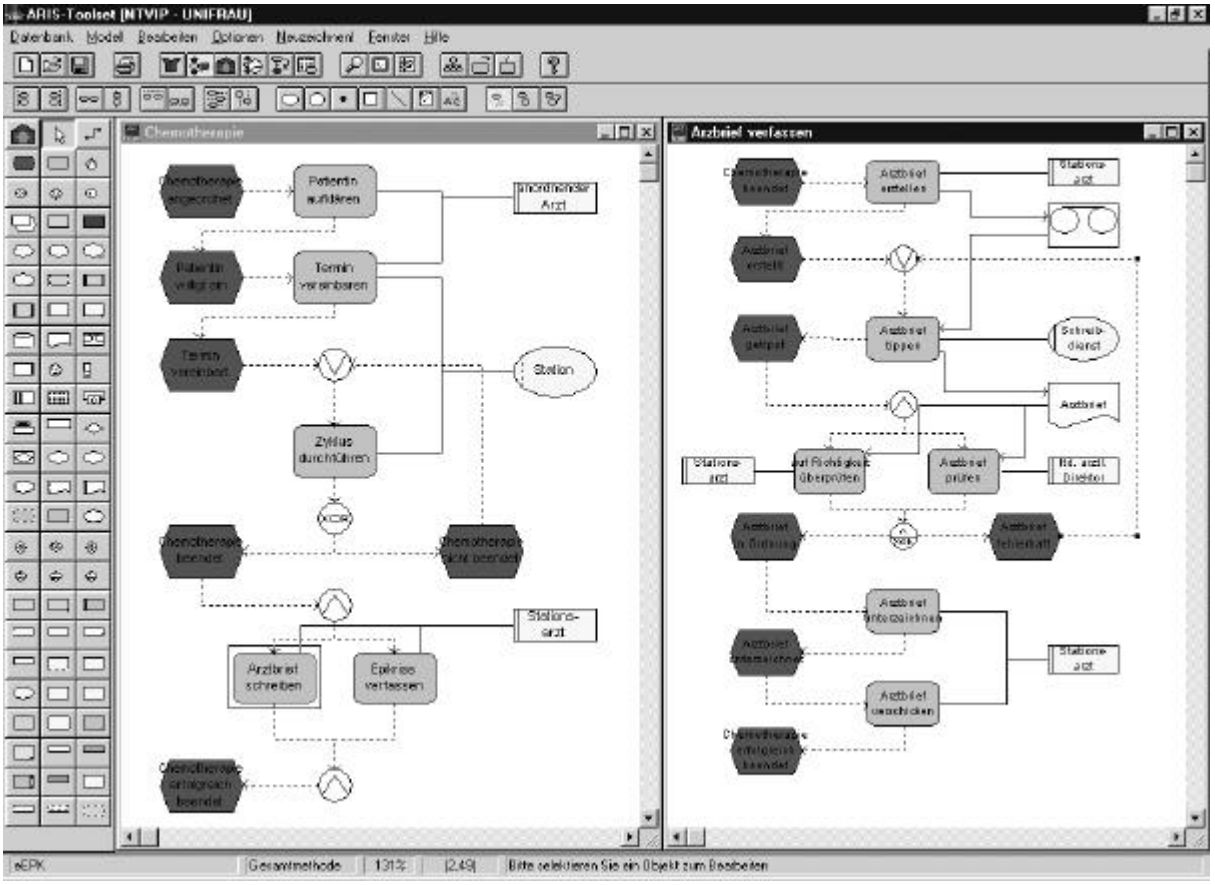


Abbildung 6: *ARIS-Toolset*

Das ARIS-Toolset beruht auf dem von A. W. Scheer entwickelten Konzept der „Architektur integrierter Informationssysteme“. Diese Architektur des ARIS-Toolsets gliedert sich in die vier Komponenten Funktionssicht, Datensicht, Organisationssicht und Steuerungssicht. Die Funktionssicht beschreibt die auszuführenden Funktionen eines Unternehmens sowie ihre hierarchischen Zusammenhänge. Die Datensicht beschreibt die Ereignisse und Zustände des Bezugsumfelds von Unternehmen. Für die Modellierung der Datensicht benutzt ARIS erweiterte Entity-Relationship-Diagramme. Die Organisationsschicht beschreibt die Organisationseinheiten und Bearbeiter sowie ihre Beziehungen und Strukturen.

Die Steuerungsschicht verbindet die einzelnen Komponenten der ARIS-Architektur miteinander. Abläufe werden hauptsächlich mit EPK (siehe Abschnitt 2.2.2) definiert. Abbildung 6 zeigt das Fenster der Steuerungsschicht zur Modellierung eines Arbeitsablaufs. Neben der Modellierung durch EPK bietet die neuste ARIS-Version die Modellierung mit Hilfe von UML-Diagrammen an. Diese UML-Diagramme werden aber wieder auf EPK abgebildet [Ver98].

Das ARIS-Toolset bietet verschiedene Möglichkeiten zur Analyse von Geschäftsprozessen an. Ein Beispiel für die Verwendung des ARIS-Toolsets wird in [SMM95] gegeben.

- **Bonapart**

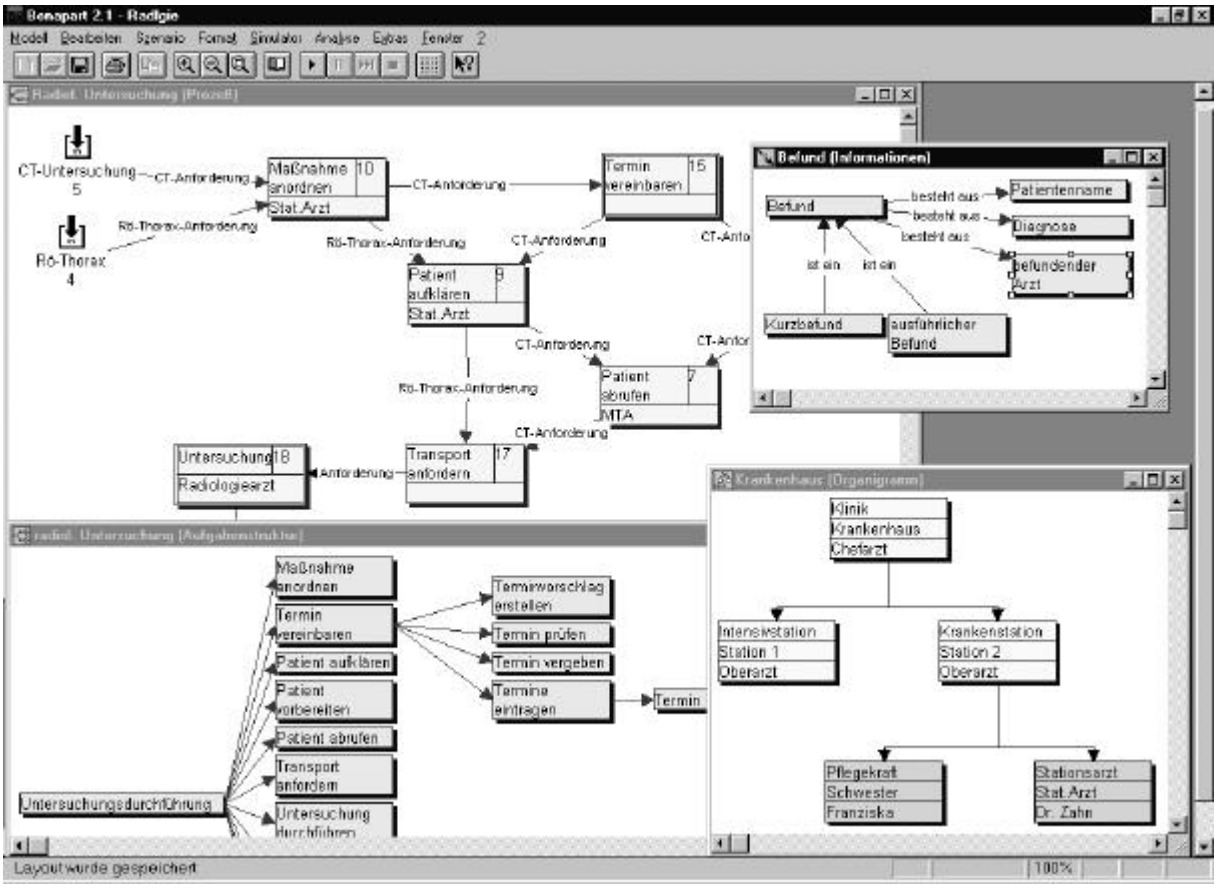


Abbildung 7: *Bonapart*

Das GPO-Werkzeug Bonapart stammt von der Firma UBIS und besitzt eine intuitive, objektorientierte Modellierungstechnik. Bei der Modellierung eines Geschäftsprozesses wird in Bonapart zwischen der Aufbauorganisation und der Prozeßmodellierung unterschieden. Bei der Prozeßmodellierung wird ein Funktionsmodell, ein Prozeßmodell und das Datenmodell erstellt. Die Funktionsmodellierung erfolgt durch eine Vorgehensweise, die die Funktionen schrittweise verfeinert. D.h. man beginnt mit einer Grobeinteilung der Funktionen und verfeinert jede Funktion dann sukzessive. Dadurch entsteht ein Funktionsbaum. Das Prozeßmodell in Bonapart ist Petri-Netz-basiert. Dadurch orientiert sich die Modellierung eines Prozesses am Datenfluß. Um einen Datenfluß herstellen zu können, müssen die zwischen den Schritten des Prozeßmodells gezogenen Kanten mit der dort fließenden Information beschriftet werden. Diese Information muß vorher im Datenmodell definiert werden. Abbildung 7 zeigt die verschiedenen Fenster zur Erstellung der beschriebenen Modelle.

Neben der Modellierung eines Arbeitsablaufs bietet Bonapart umfangreiche Möglichkeiten zur Simulation und Analyse von Prozeßmodellen. Ein Vorteil, den Bonapart besitzt, ist die für den Benutzer leicht verständliche Darstellung der modellierten Prozesse.

2.4 Workflowmodellierung

Die Workflowmodellierung unterscheidet sich von der Geschäftsprozeßmodellierung in einigen wesentlichen Punkten. Bei der Workflowmodellierung steht die rechnerunterstützte Ausführung von Prozessen im Vordergrund, während bei der Geschäftsprozeßmodellierung hauptsächlich Aspekte wie Analyse, Gestaltung und Optimierung eines Prozesses betrachtet werden.

Ein modellierter Workflow besitzt demzufolge gegenüber einem Geschäftsprozeßmodell meist eine größere Genauigkeit, Detailliertheit und daher eine höhere formale Qualität. Beispielsweise müssen bei der Workflowmodellierung die Typen und Strukturen der auszutauschenden Daten festgelegt werden. Im Gegensatz zu dieser Vorgehensweise verläuft die Geschäftsprozeßmodellierung auf einer abstrakteren Ebene. Zeitlich gesehen wird meist vor einer Workflowmodellierung eine GPO durchgeführt, damit nicht Schwachstellen im Prozeßablauf oder der Organisation in den Workflow eingehen.

Da die separate Durchführung einer Geschäftsprozeßmodellierung und einer Workflowmodellierung einen doppelten Aufwand bedeutet, bieten einige GPO-Werkzeuge eine Verbindung zu WfMS an. Ein Beispiel für eine solche Verbindung ist das GPO-Werkzeug Bonapart und das WfMS FlowMark. Durch diese Kopplung soll die Übernahme eines Prozeßmodells in ein WfMS ermöglicht werden. In der Praxis muß ein so erstelltes Modell für die Ausführung durch ein WfMS meist nachbearbeitet werden. Eine dieser notwendigen Nachbearbeitungen ist das Zusammenlegen von Aktivitäten eines Benutzers aus dem Geschäftsprozeßmodell. Würde dies nicht geschehen, so müßte der Benutzer bei jeder abgeschlossenen Aktivität seine Zustimmung geben. Dies würde bei zu hoher Frequenz den Benutzer zu sehr demotivieren.

In den Abschnitt 2.4.1 werden als Beispiele Modellierungskomponenten von verschiedenen Workflow-Systeme wie LEU, WorkParty und FlowMark vorgestellt.

2.4.1 Modellierungsmöglichkeiten kommerzieller WfMS

- **LEU (FUNSOFT-Netze)**

Ein Werkzeug zur Modellierung, Analyse, Simulation und Durchführung von Geschäftsprozessen ist die Workflow-Management-Umgebung LEU [Gru93, DGS94, Gru96].

Die Aufbauorganisation wird wie in ARIS oder Bonapart durch organisatorische Einheiten (Stellen) und deren Beziehungen modelliert. Dadurch ergibt sich ein hierarchischer Aufbau eines Organisationsmodells. Der Zusammenhang zwischen den abstrakten Stellen und konkreten Prozeßbeteiligten wird mit Hilfe des Rollenbegriffs modelliert. Datenmodelle werden in LEU durch erweiterte Entity-Relationship-Diagramme beschrieben. Sie legen die Struktur von Objekten und deren Beziehungen zueinander fest.

Der interessanteste Punkt in LEU ist die Beschreibung eines Prozesses durch FUNSOFT-Netze, die speziell für die Prozeßmodellierung entwickelt wurden [Gru96]. Abbildung 8 zeigt das Beispiel eines FUNSOFT-Netzes. FUNSOFT-Netze sind „höhere“ Petri-Netze, wobei Transitionen als Aktivitäten und Stellen als Kanäle bezeichnet werden. Kanäle repräsentieren Objektspeicher und sind fest mit einem Objekttyp verknüpft. Sie können mehrere Objekte speichern. Die Kanten eines FUNSOFT-Netzes beschreiben die Ein- bzw. Ausgabebeziehungen zwischen Aktivitäten und Kanälen, d.h. Aktivitäten lesen aus Kanälen, mit denen sie über eingehende Kanten verbunden sind. Ebenso schreiben Aktivitäten in Kanäle, die über ausgehende Kanten erreichbar sind. Neben dem zerstörenden Lesen aus einem Kanal gibt es die Möglichkeit ein Objekt aus einem Kanal zu kopieren, d.h. das Objekt wird gelesen und verbleibt im Kanal. Neben dem normalen Petri-Netz-Schaltverhalten gibt es noch weitere Schaltverhalten in FUNSOFT-Netzen, die die Modellierung eines Prozesses vereinfachen. Das Schaltverhalten einer Aktivität kann als Attribut festgelegt werden. FUNSOFT-Netze bieten außerdem verschiedene Strukturierungsmittel. Aktivitäten lassen sich verfeinern, indem sie wiederum durch ein FUNSOFT-Netz beschrieben werden. FUNSOFT-Netze bieten auch die Möglichkeit Schnittstellenkanäle zu definieren. Durch Schnittstellenkanäle lassen sich weitere Geschäftsprozeßmodelle anbinden.

FUNSOFT-Netze ermöglichen es, wie allgemeine Petri-Netze auch, die Korrektheit eines Prozesses zu prüfen. Ein Beispiel für die Korrektheitsprüfung in FUNSOFT-Netzen ist die Deadlock-Erkennung. Die Semantik der FUNSOFT-Netze wird durch eine Abbildung auf Prädikat/Transitions-Netze definiert. Um bestimmte Eigenschaften eines Netzes nachzuweisen kann neben der FUNSOFT-Netz-Repräsentation auch die Prädikat/Transitions-Netz-Darstellung des jeweiligen Netzes benutzt werden [Gru96]. Ein Nachteil des Nachweises bestimmter Netzeigenschaften auf Basis von Prädikats/Transitions-Netzen ist, daß die Algorithmen äußerst laufzeitintensiv sind.

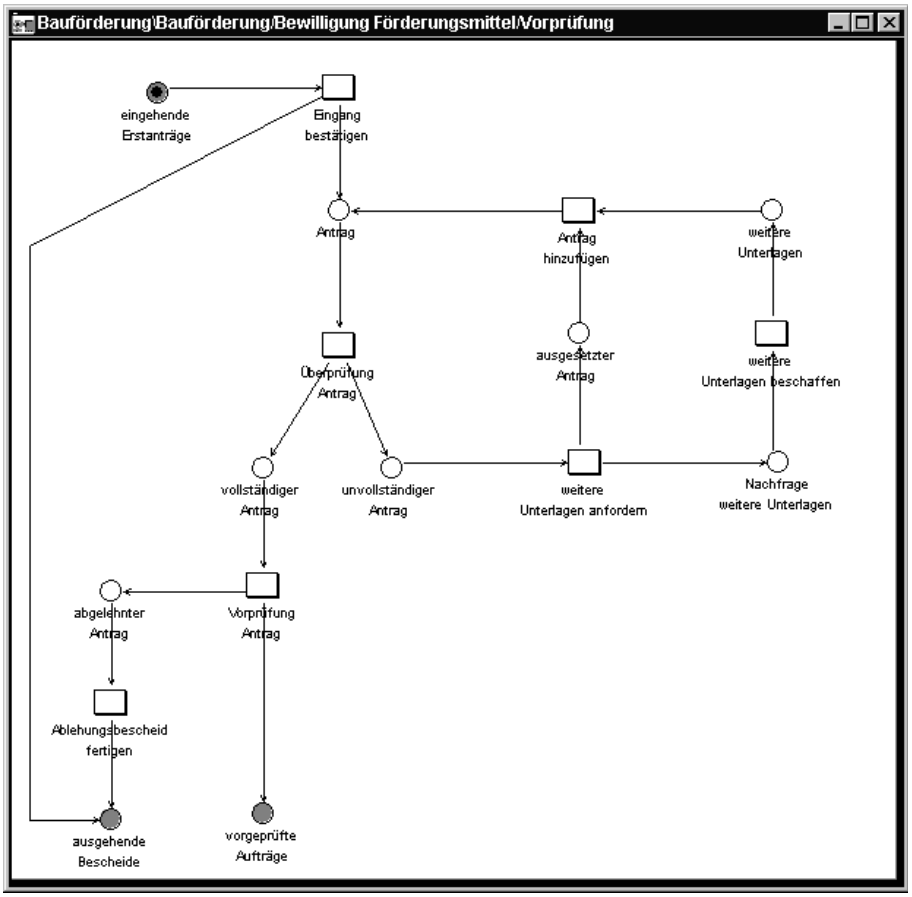


Abbildung 8: *FUNSOFT-Netz*

- **WorkParty:**

Das prozeßorientierte WfMS WorkParty [FrS97] ist eine Entwicklung der Firma Siemens-Nixdorf. WorkParty unterscheidet zwischen der Modellierung der Ablauflogik (Kontroll- und Datenfluß), der Modellierung der Organisation und der Erstellung des Anwendungscodes. Bei der Erstellung des Anwendungscodes werden Aktivitätenprogramme erstellt, die bei Aufruf eines Prozeßschrittes automatisch gestartet werden.

Der Kontrollfluß eines Prozesses (in WorkParty: Ablauf) wird mit Hilfe eines graphischen, syntaxgesteuerten Modellierungswerkzeugs spezifiziert. Der Editor unterstützt die Modellierung mit den folgenden blockbasierten Konstrukten: elementare Aktivitäten, Subprozesse (komplexe Aktivitäten), Schleifen, alternative und parallele Verzweigungen. Eine der Stärken von WorkParty ist die übersichtliche Darstellung des Prozeßgraphen. Dieses Layout des Graphen wird vom Editor generiert und entlastet so den Modellierer. Ein weiterer Vorteil ist auch die leichte Bedienbarkeit des Editors.

Aktivitäten (in WorkParty: Tätigkeiten) können mit dem Tätigkeitseditor definiert werden. Dabei wird festgelegt, wer die Tätigkeit ausführen darf, welche Programme aufzurufen sind und welche Daten benötigt werden.

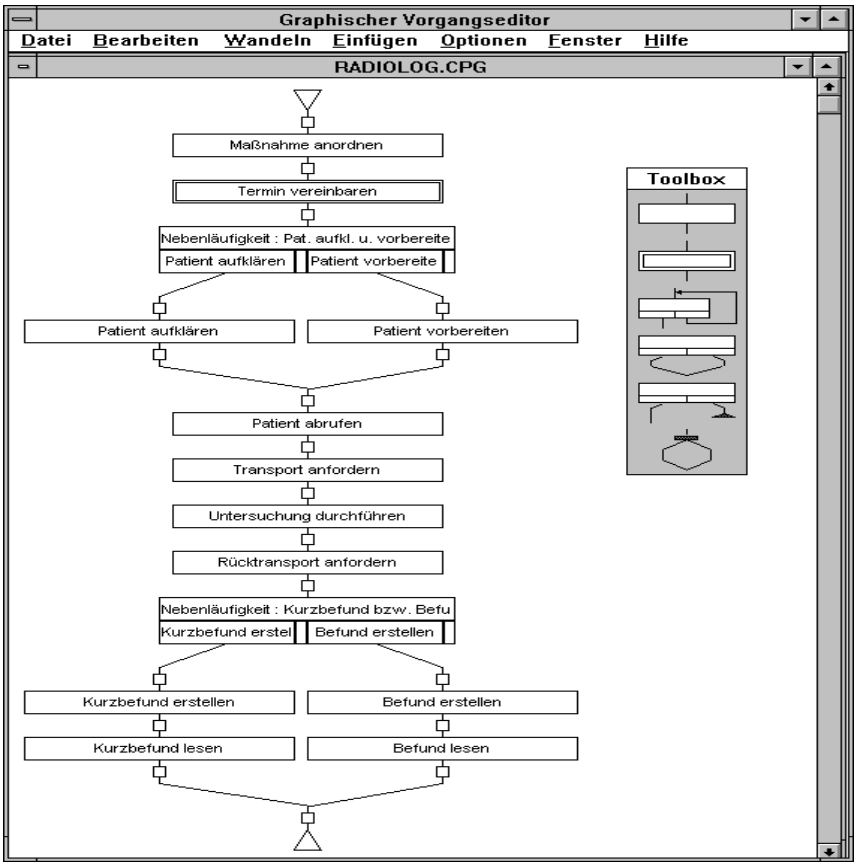


Abbildung 9: *WorkParty*

Der WorkParty-Editor erlaubt nur eine sehr eingeschränkte Modellierung von temporalen Aspekten. D.h. es können beispielsweise keine Zeitabstände zwischen Tätigkeiten im Editor modelliert werden. Außerdem ist das Granulat für die Angabe eines Startzeitpunktes sehr grob. D.h. es können für eine Aktivität nur die Werte genau, spätestens oder frühestens n Tage nach dem Erscheinen der Aktivität in einer Arbeitsliste spezifiziert werden. Eine Erweiterung des WfMS WorkParty um verschiedene temporale Aspekte wird in [Gri96] beschrieben.

Die Datenflußspezifikation in WorkParty erfolgt durch die Definition von Prozeßvariablen und deren Verknüpfung mit Tätigkeiten. Da WorkParty den Austausch von typisierten und komplexen Strukturen nicht ermöglicht, werden über das WfMS nur Workflow-relevante Daten und Referenzen auf Anwendungsobjekte ausgetauscht.

Ein weiterer Nachteil der Prozeßmodellierung in WorkParty ist, daß nicht einzelne Teile des Prozeßgraphen als Ausnahmebehandlung definiert werden können. Dadurch können keine verschiedenen Ansichten, beispielsweise mit ausgeblendeten Fehlerkanten wie im ADEPT-Basismodell möglich, dargestellt werden. WorkParty ermöglicht nur die Spezifikation von Ausnahmeprogrammen für Fehler, Abbruch oder Unterbrechung eines Prozesses und erlaubt daher nur die nochmalige Ausführung des aktuellen Prozeßschritts. Bereits beendete Prozeßschritte können nicht zurückgesetzt werden.

Die Aufbauorganisaton wird in WorkParty durch das Organisations- und Ressourcenmanagement (ORM)-Werkzeug definiert. Während der Ausführung des Prozesses werden aus diesem Modell Bearbeiter den einzelnen Tätigkeiten zugeordnet. ORM spezifiziert bei der Modellierung

organisatorische Entitäten wie Organisationseinheit, Stelle, Rolle, Mitarbeiter, Kompetenz und deren Beziehungen zueinander.

- **FlowMark**:

Das prozeßorientierte WfMS FlowMark ist ein Produkt der Firma IBM [LeA94]. Es benutzt zur Modellierung eines Prozesses eine graphischen, netzbasierten Ansatz. Neben der graphischen Modellierung ist in FlowMark auch die textuelle Spezifikation eines Workflows in der FlowMark Definition Language (FDL) möglich.

Prozeßdefinitionen in FlowMark bestehen im wesentlichen nur aus Aktivitäten und werden als azyklischen Graphen repräsentiert. Die Knoten des Graphen repräsentieren dabei die Aktivitäten, die ausgeführt werden sollen und die Kanten stellen den Kontroll- und Datenfluß dar. Ein so gestaltetes Netz wird als Aktivitätennetz bezeichnet.

FlowMark unterscheidet bei der graphischen Darstellung drei Knotentypen:

- *Programmaktivitäten* repräsentieren ein aufzurufendes Programm und sind somit atomare Aktivitäten.

- *Prozeßaktivitäten* enthalten eine Referenz auf eine weiteres Workflow-Schema. Dieses Workflow-Schema kann im Gegensatz zu Blöcken mehrmals wiederverwendet werden.

- *Blöcke* bestehen aus mehreren Aktivitäten und besitzen Ähnlichkeiten mit Prozeßaktivitäten. Blöcke können jedoch nur einmal verwendet werden. Da Prozeßgraphen in FlowMark azyklisch sein müssen, können mit Hilfe von Blöcken Schleifen realisiert werden. Jeder Block besitzt dafür eine Austrittsbedingung und bei Nichterfüllung kann der gesamte Block wiederholt werden.

Die einzelnen Aktivitäten werden über Kontrollkonnektoren (graphisch: durchgezogene Linien) verbunden und definieren so einen zeitlichen Ablauf der Aktivitäten. FlowMark erlaubt die Modellierung von Verzweigungen durch die Zuweisung eines boolschen Ausdrucks zu Kontrollkonnektoren. Mit Hilfe sogenannter Defaultkonnektoren (graphisch: durchgezogene Linie mit Kreis am Startpunkt) lassen sich bei Verzweigungen Zweige implementieren, die durchlaufen werden, wenn die Transitionsbedingungen sämtlicher Kontrollkonnektoren nicht erfüllt werden.

Für die Modellierung des Datenflusses besitzt jede Aktivität Ein- und Ausgabeparameter. Die Menge aller Ein- bzw. Ausgabeparameter wird in FlowMark Eingabecontainer bzw. Ausgabecontainer einer Aktivität genannt. Die Parameter einer Aktivität sind typisiert und können komplexe Werte aufnehmen. Um Daten auszutauschen werden Verbindungen, sogenannte Datenkonnektoren (graphisch: gestrichelte Linien), zwischen zwei Aktivitäten spezifiziert. Die Abbildung der einzelnen Ausgabeparameter auf die jeweiligen Eingabeparameter muß dabei durchgeführt werden. Da die Datenkonnektoren ebenfalls in den Prozeßgraph eingezeichnet werden, wird eine komplexe Prozeßdefinition schnell unübersichtlich.

In FlowMark können hierarchische Organisationen modelliert werden. FlowMark unterstützt auch die Zusammenfassung mehrerer Personen durch den Rollen-Begriff.

Temporale Aspekte unterstützt FlowMark nur unzureichend, da Zeitspannen zwischen Aktivitäten nicht modelliert werden können. Auch die Angabe eines Startzeitpunktes einer Aktivität ist nicht möglich. Jedoch ermöglicht FlowMark die Spezifikation zeitlicher Begrenzungen für die Ausführungsdauer einer Aktivität.
Zusätzlich zur Modellierung bietet FlowMark eine Animationskomponente, die die unterschiedlichen Möglichkeiten der Ausführung eines Workflow darstellt, ohne eine Instanz des Workflows zu erzeugen. Damit kann eine Reihe logischer Fehler schon bei der Spezifikation eines Workflows

entdeckt und behoben werden. Eine formale Verifikation bestimmter Eigenschaften von Workflow-Schemata bietet FlowMark bisher nicht, da die Semantik erst durch das FlowMark-Laufzeitsystem festgelegt wird. FlowMark kann allenfalls einfache Konsistenzüberprüfungen, wie z.B. die Erkennung von isolierten Aktivitäten, durchführen.

Ein weiterer Nachteil von FlowMark ist, daß die Beschreibung planbarer Abweichungen im Kontrollfluß nicht mit Hilfe eigener Konstrukte, wie z.B. Fehlerkanten, erfolgen kann. Diese Abweichungen müssen mit den vorhandenen Konstrukten vom Modellierer selbst implementiert werden. Dadurch können wie bei WorkParty keine verschiedenen Ansichten auf den Ablaufgraph realisiert werden. Durch die zusätzlichen Konstrukte für die Kompensation wird der Ablaufgraph noch unübersichtlicher.

Ein Nachteil bei der Bedienung von FlowMark ist die große Anzahl von Fenstern bei der Modellierung eines Prozesses. Diesen Nachteil kann der Modellierer fast als „Fenster-Terror“ empfinden.

Einen ähnlichen Ansatz wie FlowMark verfolgt das in [ReW92] vorgestellte System ActMan. Dieses System benutzt zur Modellierung ebenfalls Aktivitätennetze.

3. Das ADEPT-Basismodell

Das folgende Kapitel beschreibt die formalen Grundlagen des ADEPT-Basismodells. Dies geschieht hier allerdings nur so tief, wie die Theorie für das Verständnis der Modellierung und der Funktionsweise des Editors notwendig ist.

Das ADEPT-Basismodell ist ein graphischer, netzbasierter Ansatz zur Modellierung von Arbeitsabläufen. Es besitzt im Vergleich zum ADEPT-Modell eine kleinere Anzahl von Konstrukten und Operationen [Kir96, Hen97, Gri97, DKR95, DaR97]. Der Grund hierfür ist, daß die Verifikation und die Ausführung des Arbeitsablaufs durch das interpretierende Laufzeitsystem zu aufwendig wären. Um dies alles zu vereinfachen, wird das ADEPT-Modell auf das ADEPT-Basismodell abgebildet.

Grundlage für die Beschreibung eines Prozesses ist der Kontrollfluß. Er legt zur Ausführungszeit die Reihenfolge der einzelnen Arbeitsschritte fest. Die Darstellung des Kontrollflusses im ADEPT-Basismodell geschieht durch einen gerichteten Graphen. Die Arbeitsschritte werden im ADEPT-Basismodell durch Aktivitäten oder Aktivitätenblöcke[1] beschrieben. Aktivitäten werden durch sogenannte Aktivitätenvorlagen definiert. Das Konzept der Vorlagen erlaubt es, eine Spezifikation für eine Aktivität mehrmals in einem oder verschiedenen Arbeitsabläufen zu verwenden. Aktivitäten werden entweder manuell, d.h. mit Benutzerinteraktion, oder durch einen Computer automatisch ausgeführt, z.B. durch Aufruf von Applikationen oder Formularen. Das ADEPT-Basismodell wurde so gewählt, daß es relativ einfach ist, die syntaktische Korrektheit des Kontrollflusses zu gewährleisten. Analysen zur Korrektheit des Kontrollflusses im ADEPT-Basismodell können sowohl bei der Modellierung des Ablaufs (statische Prüfung) als auch bei der Ausführung des Prozesses gemacht werden (dynamische Prüfung). Die dynamische Prüfung erfolgt z.B. bei der Änderung der Struktur eines Ablaufs zur Laufzeit (z.B. in medizinischen Umgebungen).

Ein besonderes Merkmal für die Spezifikation und Ausführung von Abläufen im ADEPT-Basismodell ist das Konzept der symmetrischen Kontrollstrukturen [Rei93]. Sequenzen, Verzweigungen (mit unterschiedlichen Semantiken) und Schleifen werden als symmetrische Blöcke modelliert, die jeweils nur einen Start- und Endknoten besitzen. Diese Blöcke können beliebig oft verschachtelt werden, dürfen sich jedoch niemals überlappen. Abbildung 10 illustriert das Konzept der symmetrischen Blöcke.

[1] Aktivitätenblöcke werden in dieser Arbeit nicht betrachtet, da sie nicht vom Editor unterstützt werden. [Hen97] beschreibt Aktivitätenblöcke näher.

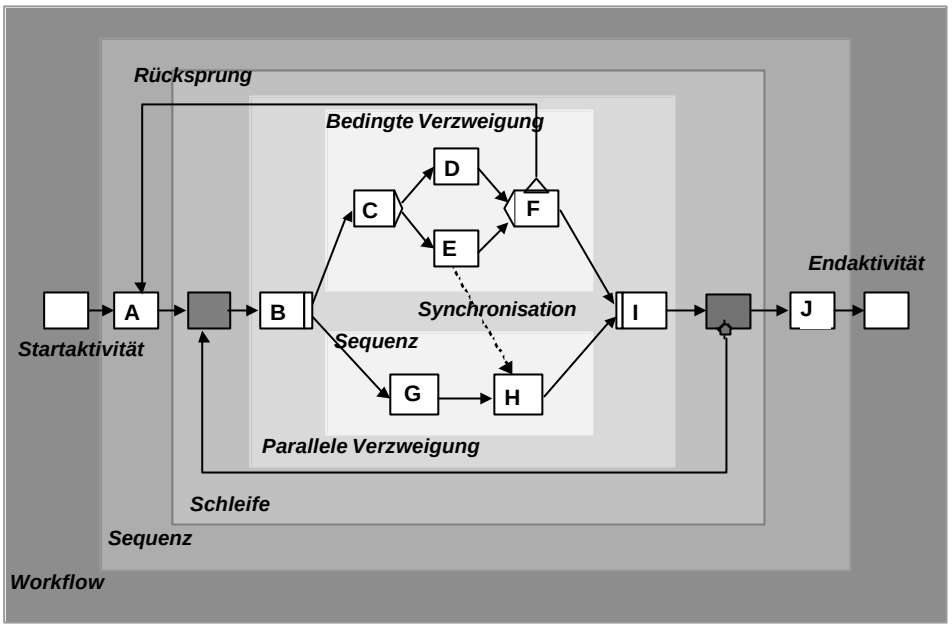


Abbildung 10: *Symmetrische Blockstrukturierung*

Neben dem Kontrollfluß wird im ADEPT-Basismodell der Datenfluß explizit modelliert. Dazu kann jede Aktivitätenvorlage Ein- und Ausgabeparameter besitzen, die den Informationsfluß zwischen den Arbeitsschritten sicherstellen. Die Ein- und Ausgabeparameter der Aktivitäten werden über sogenannte Datenslots verbunden. Eine Verbindung zwischen einem Parameter und einem Datenslot bedeutet, daß der Parameter einer Aktivität aus diesem Datenslot entweder ließt oder schreibt. Datenslots sind vergleichbar mit globalen Prozeßvariablen und verwalten verschieden Versionen eines Datums. Diese Versionen entstehen während der Ausführung des Workflows durch das Schreiben in diesen Datenslot durch mehrere Aktivitäten. Wie beim Kontrollfluß erlaubt die explizite Modellierung die statische und dynamische Korrektheitsprüfung.

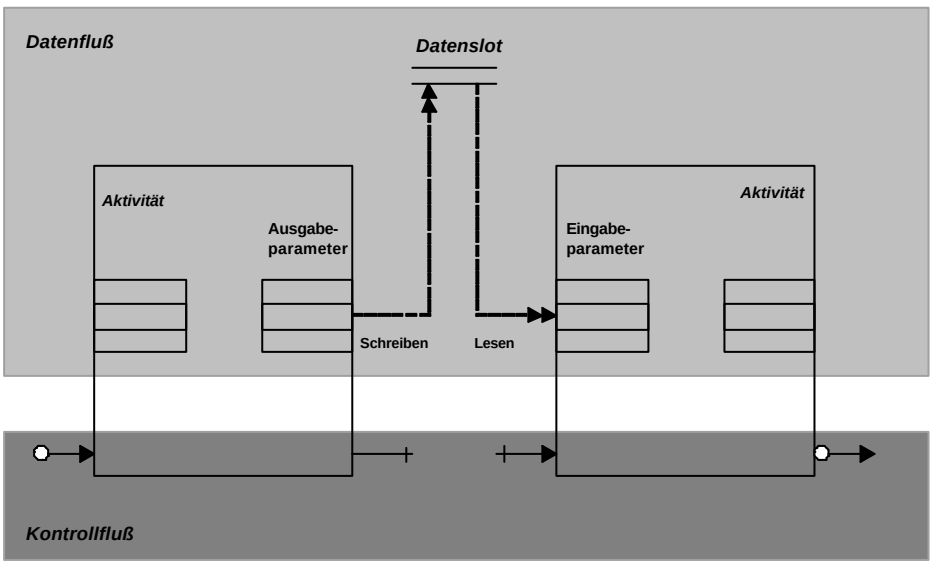


Abbildung 11: *Integration von Kontroll- und Datenfluß*

3.1 Die Konstrukte des ADEPT-Basismodells

Um einen Überblick über das ADEPT-Basismodell zu bekommen, werden hier die Konstrukte informell eingeführt. Neben der Semantik der Konstrukte werden die graphischen Darstellungen vorgestellt.

3.1.1 Die Sequenz

Das einfachste Konstrukt im ADEPT-Basismodell ist die Sequenz. Jede Aktivität der Sequenz besitzt genau einen Nachfolger. Vorgänger- und Nachfolgerknoten sind jeweils über eine Kontrollkante miteinander verbunden. Wird der Vorgängerknoten beendet, so wird automatisch der Nachfolgerknoten aktiviert (sequentielle Ausführung).

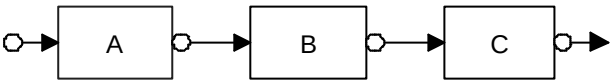


Abbildung 12: *Sequenz*

3.1.2 Die Verzweigungen

Das ADEPT-Basismodell kennt drei Arten von Verzweigungen. Dies sind die parallele Verzweigung, die parallele Verzweigung mit finaler Auswahl und die bedingte Verzweigung. Die Blocksymmetrie erzwingt immer, daß alle Zweige am Synchronisationsknoten wieder zusammengeführt werden.

- **Die parallele Verzweigung**

Bei parallelen Verzweigungen werden alle Zweige parallel gestartet und auch parallel ausgeführt. Der Synchronisationsknoten kann erst nach Beendigung aller Aktivitäten der Zweige gestartet werden. Dieses Konstrukt ermöglicht es, daß voneinander unabhängige Arbeitsschritte parallel (nebenläufig) ausgeführt werden können. In der Logik entspricht die UND-Parallelität dem Konstrukt der parallelen Verzweigung.

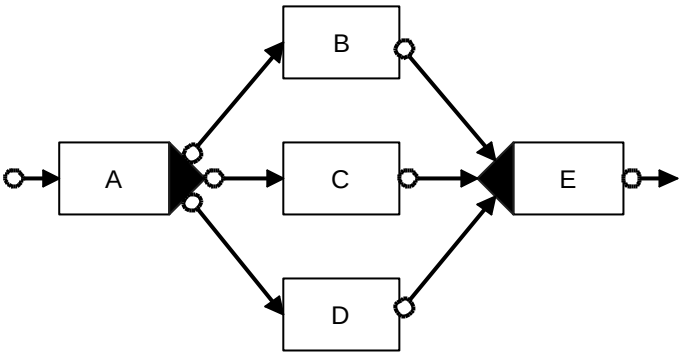


Abbildung 13: *Parallele Verzweigung*

- **Die parallele Verzweigung mit finaler Auswahl**

Bei der parallelen Verzweigung mit finaler Auswahl werden, wie bei der parallelen Verzweigung, alle Zweige parallel gestartet. Der zuerst beendete Zweig ermöglicht dann das Weiterarbeiten am Synchronisationsknoten (implizite Entscheidung) und die anderen Zweige werden zurückgesetzt. Eventuell gemachte Schreiboperationen dieser Zweige auf Datenslots werden rückgängig gemacht, d.h. nur die Informationen des „schnellsten" Zweiges überleben. Die parallele Verzweigung entspricht der ODER-Parallelität in der Logik.

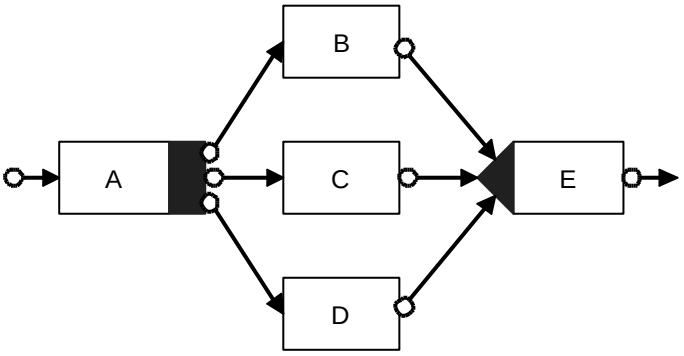


Abbildung 14: *Parallele Verzweigung mit finaler Auswahl*

- **Die bedingte Verzweigung**

Bei der bedingten Verzweigung wird im Gegensatz zu den anderen beiden Verzweigungsarten nur die Abarbeitung eines Zweigs gestartet. Eine Analogie dazu ist die aus den Programmiersprachen bekannte if ... then – Verzweigung.

Das ADEPT-Basismodell bietet bei der Ausführung einer Workflow-Instanz zwei Möglichkeiten bei der Auswahl eines Zweiges.

1. Die Auswahl eines Zweiges geschieht explizit nach dem Verzweigungsknoten aufgrund eines Wertes aus einem Datenslot. Dieser Datenslot kann von vorhergehenden Knoten oder vom Verzweigungsknoten selbst beschrieben worden sein. Diese Vorgehensweise wird vom Editor unterstützt.

2. Die möglichen Zweige bzw. deren Anfangsaktivitäten werden den Benutzern in deren Arbeitslisten angeboten. Wählt ein Benutzer nun eine Anfangsaktivität, so wird der Zweig, der die Anfangsaktivität enthält, als weiterer Weg benutzt. Die anderen alternativen Aktivitäten werden aus den Arbeitslisten entfernt. Dieses Vorgehen entspricht der impliziten Auswahl eines Zweiges.

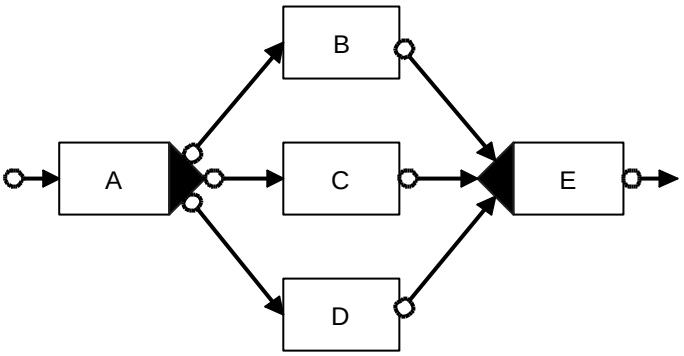


Abbildung 15: *Bedingte Verzweigung*

3.1.3 Die Schleife

Die Schleife im ADEPT-Basismodell entspricht der in der Welt der Programmiersprachen häufig benutzen repeat ... until-Schleife. Schleifen bilden einen Block mit dem Startknoten der Schleife vom Typ EMPTY und dem Endknoten der Schleife vom Typ LOOP. Der Schleifenkörper selbst bildet wiederum einen Block.

Nach jedem Durchlauf der Schleife wird im Schleifenendknoten die Abbruchbedingung evaluiert. Wird die Schleife nicht abgebrochen, so erfolgt der Rücksprung zum Startknoten der Schleife über eine spezielle Schleifenkante. Mit Hilfe der Konstrukte des ADEPT- Basismodells lassen sich andere Schleifentypen, wie z. B. die while ... do-Schleife ebenfalls realisieren.

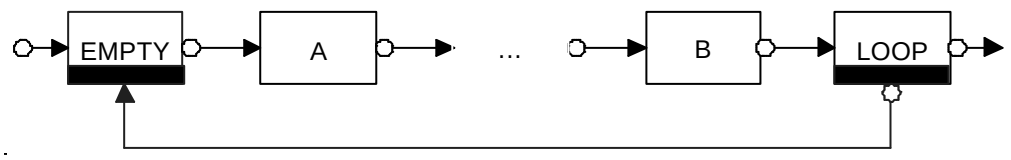


Abbildung 16: *Schleife*

3.1.4 Die Kantentypen des ADEPT-Basismodells

- **Kontrollkanten**

Kontrollkanten beschreiben den Kontrollfluß innerhalb des Ablaufgraphen. Kontrollkanten können unterschiedlich Prioritäten haben. Die Wert für die Priorität einer Kontrollkante ist entweder 1 oder 2. Per Default haben alle Kontrollkanten die Priorität 1. Die Priorität bewirkt, daß der nachfolgende Arbeitsschritt entweder als Standardarbeitsschritt (Priorität 1) oder als Abweichungsschritt (Priorität 2) in den Arbeitslisten der Benutzer erscheint. Kontrollkanten mit Priorität 2 sind vor allem zu betrachten im Zusammenhang mit Priorisierungskanten.

- **Schleifenkanten**

Schleifenkanten verbinden den Schleifenstart- und Schleifenendknoten miteinander und erlauben die Ausführung einer Schleifeniteration.

- **Fehlerkanten**

Tritt beim Ablauf des Prozesses ein Ereignis ein, das es erfordert, einen Teil des Ablaufs zurückzusetzen (z.B. eine Aktivität kann nicht erfolgreich beendet werden), so wird von der fehlgeschlagenen Aktivität ein Fehlerwert gesetzt. Aufgrund des Fehlerwertes wird eine dazu korrespondierende Fehlerkante gesucht und der Geschäftsprozeß wird, falls eine solche vorhanden ist, zu der Aktivität zurückgesetzt, auf die die Fehlerkante zeigt[1]. Die Fehlerwerte, die den Kanten zugeordnet werden, müssen aufgrund der Auswahlfunktion eindeutig sein.

- **Synchronisationskanten**

Synchronisationskanten (Sync-Kanten) dienen der Synchronisation von verschiedenen Zweigen einer parallelen Verzweigung. Eine Aufgabe von Sync-Kanten ist z.B. die Vermeidung von parallelen Schreiboperationen zweier Knoten in unterschiedlichen Zweigen einer parallelen Verzweigung.

Das ADEPT-Basismodell besitzt zwei unterschiedliche Typen von Synchronisationskanten:

1. Eine „weiche“ Sync-Kante zwischen zwei Aktivitäten n_1 und n_2 bedeutet, daß n_2 nur ausgeführt werden kann, wenn n_1 abgeschlossen worden ist oder n_1 nicht mehr aktiviert werden kann. Dies ist z.B. der Fall, wenn sich n_1 in einem nicht bearbeiteten Zweig einer bedingten Verzweigung befindet.

2. Eine „harte“ Sync-Kante (strikte Synchronisation) zwischen den beiden Aktivitäten n_1 und n_2 erfordert, daß n_2 nur nach der erfolgreichen Ausführung von n_1 gestartet werden kann.

[Hen97] beschreibt Sync-Kanten sehr ausführlich.

[1] Zurücksetzen bedeutet, daß bei der Ausführung der Prozeßvorlage gemachte Markierungen zurückgesetzt und Schreiboperationen von Knoten auf Datenslots kompensiert werden. Die Arbeiten [Wei97] und [Hen97] vertiefen diese Thematik.

- **Priorisierungskanten**

Priorisierungskanten sind eine besondere Art von Kanten. Sie bewirken, daß die Priorität einer auf den Endknoten der Priorisierungskante zeigenden Kontrollkante von 2 auf 1 gesetzt wird, wenn der Startknoten der Priorisierungskante abgearbeitet worden ist. Priorisierungskanten können nur in parallelen Verzweigungen auftreten. Die Nutzung von Priorisierungskanten macht sich deutlich bei der Ausführung eines Workflows bemerkbar. Dabei wird dem Benutzer bei Verwendung einer Priorisierungskante in einer parallelen Verzweigung, der Startknoten der Priorisierungskante als Standardarbeitsschritt und der Endknoten als Ausnahmeschritt angeboten. [Hen97] zeigt ausführlich die Verwendung vom Priorisierungskanten.

- **Zeitkanten**

Zeitkanten definieren den zeitlichen Abstand zwischen zwei Aktivitäten n_1 und n_2. Diese Kanten besitzen 2 Attribute, die den minimalen und den maximalen Abstand zwischen den zwei Aktivitäten n_1 und n_2 angeben. Der minimale Abstand beschreibt den kleinstmöglichen Abstand zwischen dem Ende von n_1 und dem Anfang von n_2. Der maximale Abstand gibt die größte Zeitspanne zwischen Ende von n_1 und Start von n_2 an. Zeitkanten können nur zu nachfolgenden Aktivitäten von n_1 gezogen werden.

Zeitkanten werden in der Arbeit [Gri97] umfassend beschrieben.

3.2 Die formalen Grundlagen des ADEPT-Basismodells

Der Formalismus des ADEPT-Basismodells ist die Grundlage für Korrektheitsprüfungen in Kontroll- bzw. Datenfluß und für die in anderen Arbeiten beschriebenen dynamischen Änderungen einer Ablaufvorlage [Hen97, DaR9?].

Ein Arbeitsablauf, der durch Konstrukte des ADEPT-Basismodells definiert worden ist, wird durch den gerichteten Graphen P = (N, E, D, DF) repräsentiert. N ist eine endliche Menge von Knoten, die elementare Aktivitäten $a \in A$ darstellen, E ist die endliche Menge $E = \{e_1, e_2, \ldots\}$ von Kanten, D ist die endliche Menge der Datenslots und DF ist die endliche Menge von Datenkonnektoren. Datenkonnektoren definieren den Datenfluß, indem sie Ein- und Ausgabeparameter über Datenslots miteinander verknüpfen.

Eine Aktivität $a \in A \subseteq N$ wird durch das 8-Tupel $(Id^a, T^a, V_{in}{}^a, V_{out}{}^a, Dp^a, IPars^a, OPars^a, Time^a)$ dargestellt.

Die einzelnen Elemente des Tupels für eine Aktivität sind:

- $Id^a \in I$ ist die eindeutige Bezeichnung der Aktivität, wobei I die Menge aller möglichen Bezeichnungen ist.

- $T^a \in T$ definiert den Typ der Aktivität. Mögliche Werte der Menge T sind:

  - *START* für den Startknoten
  - *ENDWF* für den Endknoten
  - *MANUAL* für normale Arbeitsschritte

- *APPLICATION* für normale Arbeitsschritte, die von einem Computer ausgeführt werden (z.B. externe Programme)

- *LOOP* für Schleifenendknoten

- *EMPTY* für leere Knoten

⌦ V_{in}^{a} beschreibt die Eingangssemantik (Synchronisationssemantik) des Knotens n.
Die Eingangssematik gibt an, wieviel abgeschlossene Zweige zum Fortfahren im Arbeitsablauf notwendig sind.
Als Werte für V_{in}^{a} sind die Elemente der Menge V möglich:

- *„1-aus-1“* – Wert für alle Knoten des Ablaufs, die nur einen Vorgängerknoten besitzen.
- *„1-aus-n“* – Wert für Synchronisationsknoten einer parallelen Verzweigung mit finaler Auswahl oder einer bedingten Verzweigung. Bei der bedingten Verzweigung wird der hier abzuarbeitende Zweig schon im Voraus gewählt.
- *„n-aus-n“* – Wert für den Synchronisationsknoten einer parallelen Verzweigung.

⌦ V_{out}^{a} beschreibt die Ausgangssemantik (Verzweigungssemantik) des Knotens n.
Die Ausgangssemantik gibt an, wieviele mögliche Alternativzweige gestartet werden sollen.

Als Werte für V_{in}^{a} sind möglich:
- *„1-aus-1“* – Wert für Knoten des Ablaufs, nach denen nur eine einzige mögliche Aktivität gestartet werden kann.
- *„1-aus-n“* – Wert für den Verzweigungsknoten einer bedingten Verzweigung. Ein Zweig muß ausgewählt werden.
- *„n-aus-n“* – Wert für alle Arten von parallelen Verzweigungen. Es werden alle möglichen Zweige gestartet.

⌦ Der Entscheidungsparameter Dp^{a} („Decisionparameter“) enthält die eindeutige Bezeichnung eines Datenslots, aufgrund dessen Wert die Entscheidung für einen alternativen Zweig einer bedingten Verzweigung (Verzweigungssemantik V_{out}^{a} = *„1-aus-n“*) gefällt wird.

⌦ $IPars^{a}$ und $OPars^{a}$ sind die endlichen Mengen von Eingabe- und Ausgabeparametern der Aktivität a. Die Elemente der Mengen $IPars^{a}$ und $OPars^{a}$ werden beschrieben durch das 4-Tupel par = (B^{par}, T^{par}, Dm^{par}, Nf^{par}):

- B^{par} ist der Bezeichner des Parameters. Der Bezeichner muß nur in den Mengen $IPars^{a}$ oder $OPars^{a}$ eindeutig sein.

- T^{par} bezeichnet den Typ des Parameters. Die möglichen Typen der Parameter sind STRING, INTEGER, REFERENCE[1].

- Die Eingabeparameter des gesamten Arbeitsablaufs sind die Ausgabeparameter des Startknotens. Als Ausgabeparameter des Workflows werden die Eingabeparameter des Endknotens benutzt.

- Dm^{a} gibt an, ob ein Parameter obligat oder optional ist. Für Ausgabeparameter bedeutet dies, daß die Aktivität den Ausgabeparameter zwingend schreiben muß oder im Falle optional auch keinen Wert zu liefern braucht. Bei Eingabeparametern bedeutet obligat und optional, daß der Parameter entweder sicher versorgt sein muß oder auch auf einen Wert für den Parameter verzichtet werden kann.

[1] Der Typ REFERENCE steht für eine Referenz auf Applikationsdaten, die nicht vom WfMS verwaltet werden, aber ihm bekanntgemacht werden sollten.

- Nf^a beschreibt die Nachforderbarkeit eines Parameters einer Aktivität. Die Werte für Nf^a sind NOEXTRADEMAND, SPECIALEXTRADEMAND und EXTRADEMAND. Die Aufgabe von Nf^a ist anzugeben, ob der Parameter nachforderbar ist und ob er durch eine Aktivität oder durch automatisch generierte Dienste (z.B. einen Dialog des Systems) mit einem Wert versorgt werden kann.

⌦ $Time^a$ gibt Zeitwerte für die Aktivität a an. $Time^a$ wird beschrieben durch das 6-Tupel $Time^a = (FAZ^a, SAZ^a, FEZ^a, SEZ^a, D_{min}{}^a, D_{max}{}^a)$. Die Attribute des Tupels bedeuten:

- FAZ^a ist der früheste Anfangszeitpunkt von a
- SAZ^a ist der späteste Anfangszeitpunkt von a
- FEZ^a ist der früheste Endzeitpunkt von a
- SEZ^a ist der späteste Endzeitpunkt von a
- $D_{min}{}^a$ ist die minimale Dauer von a
- $D_{max}{}^a$ ist die maximale Dauer von a

Das andere elementare Konstrukt des ADEPT-Basismodells zur Definition des Kontrollflusses ist die Kante. Alle Kanten werden durch das 6-Tupel $(Id_{start}{}^e, Id_{dest}{}^e, Et^e, Sc^e, Fc^e, Time^e)$ definiert.

⌦ $Id_{start}{}^e, Id_{dest}{}^e \in I$ geben den Start- und Endknoten einer Kante an. Der Start- und Endknoten des Ablaufs besitzt keine eingehenden bzw. ausgehenden Kanten.

⌦ $Et^e \in ET$ gibt den Kantentyp an. Die Werte der Menge ET sind:

- *CONTROL_E* für Kontrollkanten
- *LOOP_E* für Schleifenkanten
- *ERROR_E* für Fehlerkanten
- *HSYNC_E* und *SSYNC_E* für „harte“ und „weiche“ Sync-Kanten
- *PRIOR_E* für Priorisierungskanten
- *TIME_E* für Zeitkanten

⌦ $Sc^e \in SC$ gibt den Auswahlwert für eine Kante in Form einer positiven ganzen Zahl an. Dieser Wert wird benötigt bei einer bedingten Verzweigung, die den Auswahlwert der Kante mit dem Inhalt von Dp^a vergleicht.

⌦ $Fc^e \in FC$ definiert den mit einer Fehlerkante verbundenen Fehlerwert. Für andere Kanten als Fehlerkante ist dieser Wert grundsätzlich undefiniert.

⌦ Die Zeitinformation $Time^e$ wird durch das Tupel $Time^e = (Za_{min}{}^e, Za_{max}{}^e)$ beschreiben.

- $Za_{min}{}^e$ gibt den minimalen Zeitabstand zwischen dem Ende von $Id_{start}{}^e$ und dem Anfang von $Id_{dest}{}^e$ an.

- Za_{max}^{e} gibt den maximalen Zeitabstand zwischen dem Ende von Id_{start}^{e} und dem Anfange von Id_{dest}^{e} an.

Der Kontrollfluß wird im ADEPT-Basismodell wird durch den Graphen P = (N, E) beschrieben. Zur Definition des Datenflusses wird dieses Tupel um die Menge D der Datenslots und um die Menge DF zur Verknüpfung der Ein- und Ausgabeparameter über Datenslots erweitert.

Ein Datenslot wird durch das Tupel $d = (Id^{d}, T^{d})$ beschrieben. Jeder Datenslot verwaltet ein Datenobjekt. Wird nun ein Datenslot beschrieben, so wird das aktuelle Datum nicht verworfen. Es wird eine neue Version erzeugt, die von nachfolgenden Knoten gelesen werden kann. Dies erlaubt das Rücksetzen bei der Ausführung des Prozesses mit einer Fehlerkante und die parallele Abarbeitung von Verzweigungen.

- Id^{d} ist der eindeutige Bezeichner des Datenslots.
- T^{d} ist der Typ des Datenslots. Die möglichen Typen sind identisch mit den Typen für Ein- und Ausgabeparameter.

Der eigentliche Datenfluß wird definiert durch die Menge DF. Die Menge DF besteht aus Tupel der Form $df = (Z^{df}, n^{df} \in N, par^{df} \in IPars^{a} \cup Opars^{a}, d^{df} \in D)$.

- $Z^{df} \in Z$ bezeichnet den Zugriffstyp. Wenn der Parameter in der Menge $IPars^{a}$ enthalten ist, dann ist der Zugriffstyp r_in, im anderen Fall ist der Typ w_out.
- n^{df} gibt den zugreifenden Knoten an
- $par^{df} \in IPars^{a} \cup OPars^{a}$ definiert den Parameter des Knotens n^{df}
- $d^{df} \in D$ gibt den Datenslot an, aus dem gelesen oder in den geschrieben wird

Das hier beschriebene ADEPT-Basismodell wird in den folgenden Kapiteln als Grundlage für die Definition, Darstellung und Analyse der Geschäftsprozesse verwendet.

4. Die Speicherung und Analyse von Graphen

Graphendarstellungen werden mittlerweile für unzählige Anwendungen benutzt. Beispiele dafür sind z.B. Editoren zum Erstellen von Entity-Relationship-Diagrammen oder zur Darstellung von elektronischen Schaltkreisen. Wegen ihrer einfachen Darstellung komplexer Sachverhalte sind sie heute auch bei der Darstellung von Unternehmensabläufen nicht mehr wegzudenken.

Um einen graphischen Editor zu realisieren, müssen verschiedene Probleme in Zusammenhang mit der Darstellung des Graphen bewältigt werden. Offene Fragestellungen treten bei der internen Repräsentation des Graphen, der externen Repräsentation auf dem Bildschirm und bei den Operationen auf der internen Darstellung des Graphen auf. All diese Probleme lassen sich in allgemeine Probleme, die bei der Erstellung eines Editors für Graphen immer auftauchen, und spezifische Probleme, die nur bei der Erstellung eines Workflow-Editors auftreten, trennen. Ein Beispiel für das erste Aufgabengebiet ist die effiziente interne Speicherung des Graphen mit seinen Knoten und Kanten. Das zweite Gebiet sind z.B. die Korrektheitsprüfungen, die nur in Ablaufgraphen, die aus den Konstrukten des ADEPT-Basismodells bestehen, auftreten. Viele dieser Fragen lassen sich mit den Erkenntnissen der Graphentheorie lösen.

In den nachfolgenden Abschnitten werden viele Problemstellungen bei der Erstellung eines Grapheditors aufgezeigt und Lösungsmöglichkeiten vorgestellt. Die gezeigten Lösungen sind nur eine kleine Auswahl von Verfahren, um die Probleme zu lösen.

4.1 Die interne Speicherung des Graphs

Zu den allgemeinen Problemen gehört die Speicherung des Graphs innerhalb eines Programms. Vor dieser Schwierigkeit steht jeder Entwickler, der einen Sachverhalt mit Hilfe eines Graphen darstellen möchte. Dabei macht es keinen Unterschied, ob der Graph visualisiert oder nur als interne Repräsentation verwendet wird.

Die vorgestellten Lösungen beziehen sich auf einen gerichteten Graphen, den sogenannten Digraphen („Directed Graph“). In einem Digraph können die Kanten nur in eine Richtung durchlaufen werden. Abbildung 17 zeigt einen Digraphen.

Für die verschiedenen Algorithmen auf Graphen ist es wichtig, in welcher Form der zu bearbeitende Graph gespeichert ist. Die gewählte Speicherungsart beeinflußt auch die Komplexität des Algorithmus. Deswegen werden hier die zwei bekanntesten und am häufigsten benutzten Speicherungsarten für Graphen vorgestellt.

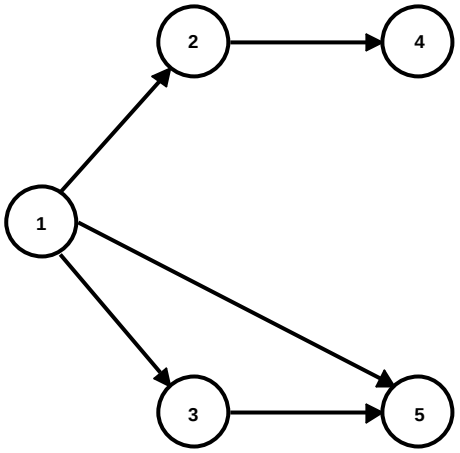


Abbildung 17: *Einfacher Graph*

Die einfachste Darstellungsweise für einen Graphen G = (V,E) innerhalb eines Programms ist die Adjazenzmatrix. Zuerst muß bei der Speicherung innerhalb einer Adjazenzmatrix jedem Knoten ein eindeutiger Zahlenwert zugeordnet werden. Die Zahlenwerte bewegen sich dabei in der Spanne von 1 bis zur Anzahl der Knoten |V|. Der Grund dafür ist, daß es möglich sein soll, mittels Feldindizes schnell auf die jedem Knoten entsprechende Information zuzugreifen.

Danach wird eine quadratische Matrix der Größe $|V| \times |V|$ erzeugt und jedes Element der Matrix wird mit 0 bzw. False initialisiert. Das Element der Matrix an der Position (x,y) wird auf 1 bzw. True gesetzt, wenn eine gerichtete Kante vom Knoten mit der Nummer x zum Knoten mit der Nummer y verläuft. Abbildung 18 zeigt die zu einem gerichteten Graphen aus Abbildung 17 gehörende Adjazenzmatrix.

	1	2	3	4	5
1	0	1	1	0	1
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	0
5	0	0	0	0	0

Abbildung 18: *Adjazenzmatrix*

Eine Adjazenzmatrix wird meist bei dicht besetzten Graphen verwendet. Die Anzahl der Kanten ist bei der Speicherung in einer Adjazenzmatrix nicht von Bedeutung, da immer $\Theta(|V|^2)$ Speicherplätze benötigt werden.

Eine andere Art der Speicherung eines Graphs ist die Adjazenzstruktur. Dabei besitzt jeder Knoten v_1 eine Adjazenzliste mit seinen Nachfolgerknoten. Zu jedem Knoten v_2 aus der Nachfolgerliste führt eine gerichtete Kante von v_1 nach v_2.

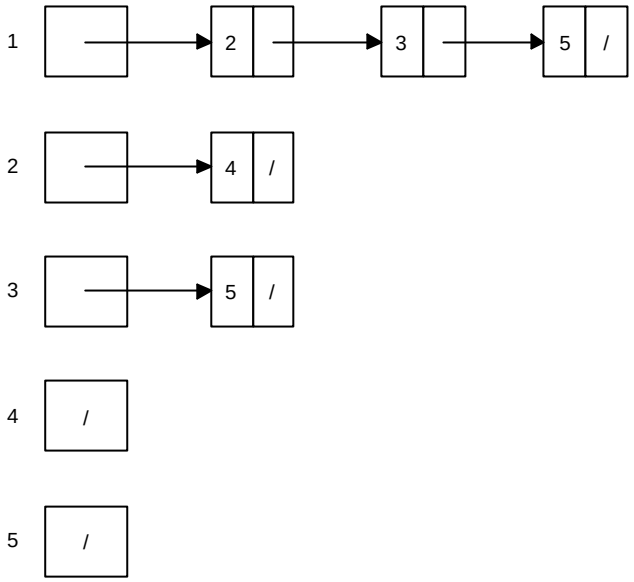


Abbildung 19: *Adjazenzstruktur*

Adjazenzstrukturen benötigen im Gegensatz zur Adjazenzmatrix O(|V|+|E|) Speicherplätze. Adjazenzstrukturen bieten damit die Möglichkeit, einen dünn besetzten Graph, bei dem |E| deutlich kleiner ist als $|V|^2$, effizient zu speichern. Die meisten Graphen, die ein Workflow-Schema repräsentieren, gehören zu der Gruppe, die sich am besten mit Adjazenzstrukturen speichern lassen. Der Nachteil der Adjazenzstruktur ist, daß bestimmte Operationen nicht so effizient wie bei der Adjazenzmatrix ausgeführt werden können. Zu diesen Operationen gehört z. B. die Bestimmung der Vorgängerknoten eines Knotens. Um die Vorgängerknoten von v_2 zu bestimmen muß jede Adjazenzliste eines Knotens v_1 nach v_2 durchsucht werden. Wird v_2 in einer Adjazenzliste gefunden, so kann v_1 in die Vorgängermenge aufgenommen werden.

In der Literatur werden verschiedene Implementierungsarten der Adjazenzstruktur beschrieben. Neben der offensichtlichen Implementierungsart mit einem Feld für die Knoten und mehreren Listen (Adjazenzlisten) für die Nachfolgeknoten wird in [NeM93] eine Implementierungsart vorgestellt, die mit zwei Felder auskommt. Eine Feld *from* ist für die Knoten und das andere Feld *succ* ist für alle Nachfolgerlisten der Knoten vorgesehen. Dabei werden die Nachfolgerlisten hintereinander im Feld *succ* gespeichert und das Feld *from* enthält im Feld *succ* die Anfangsadressen der Nachfolgerlisten für einen Knoten.

Bei der Implementierung von WFEdit2 wurden eine Speicherungsart ähnlich der Adjazenzstruktur gewählt. Dabei werden die Kanten in einer Hashtabelle gespeichert. Als Schlüssel dient der Startknoten der Kante. Ebenso werden die Knoten mit ihren Informationen in einer Hashtabelle verwaltet. Diese Speicherungsart erlaubt einen schnellen Zugriff auf Knoten und Kanten. Außerdem ist es möglich, mit Hilfe eines Knoten alle ausgehenden Kanten aus diesem Knoten zu bestimmen. Ein Vorteil dieser Implementierung ist, daß auch Informationen der Kanten (wie z.B. die minimale und

maximale Dauer, der Entscheidungswert, der Typ der Kante) so gespeichert werden können und ein schneller Zugriff auf diese Attribute gewährleistet ist.

4.2 Algorithmen zur Darstellung von Graphen

Eine Aufgabe, die innerhalb dieser Arbeit zu lösen war, ist die Darstellung eines Graphen auf dem Bildschirm. Dabei stellte sich das Problem, daß sich bekannte Algorithmen-Bücher wie z.B. [CLR90] über dieses Problem ausschweigen und somit spezielle Literatur benötigt wurde.

In diesem Abschnitt werden zwei Algorithmen vorgestellt, die Graphen, die vergleichbar mit den Graphen des ADEPT-Basismodells sind, darstellen. Da diese Algorithmen jedoch einige Nachteile besitzen, wurde in dieser Arbeit ein eigener Algorithmus für die Darstellung des Prozeßgraphen entwickelt. Dabei konnten grundlegende Konzepte der beiden Algorithmen verwendet werden..

Ein Algorithmus, der das Problem der Darstellung löst, muß vor allem zwei Dinge leisten. Er muß erstens die Position eines jeden Knoten des Graphen auf dem Bildschirm bestimmen und zweitens den Verlauf jeder Kante festlegen. In [TBB89] werden zwei allgemeine Möglichkeiten zur Anordnung der Knoten auf dem Bildschirm beschrieben. Die erste Möglichkeit der Knotenplazierung ist die Anordnung der Knoten in einem Gitter, wobei die Kantenwege oft mit Hilfe des Gitters berechnet werden. Die zweite Möglichkeit ist die uneingeschränkte, wahlweise Plazierung der Knoten auf dem Bildschirm und die Verbindung der Knoten über gerade Linien. Die Eigenschaften dieser beiden Möglichkeiten können auch miteinander vermischt werden, z.B. können Knoten wahlfrei plaziert werden und durch Kanten, die aus verschiedenen Segmenten bestehen, verbunden werden.

Der Vorteil eines automatischen Layouts eines Graphen auf dem Bildschirm ist, daß der Benutzer von der fehleranfälligen, ermüdenden und zeitraubenden manuellen Plazierung der Knoten auf dem Bildschirm entlastet wird. Ein Layout-Algorithmus verbessert damit auch die Übersichtlichkeit des Graphen, da der Algorithmus im Gegensatz zur manuellen Plazierung der Knoten immer eine optimale Darstellung anstrebt. Solch eine optimale Anordnung der Knoten ist sehr wichtig für einen Workflow-Editor, da der auf dem Bildschirm dargestellte Arbeitsablauf auch mit Personen diskutiert werden muß, die nicht unmittelbar an der Modellierung mit dem Editor beteiligt sind.

In [FNP93] werden verschiedene Bedingungen genannt, die das Aussehen eines Graphen bestimmen können. Wichtige Anforderungen an die Darstellung eines Graphen sind:

- Minimierung der Kantenkreuzungen
- Knoten mit gleichen Tiefe werden auf gleicher Ebene im Graph dargestellt
- Hierarchische Anordnung der Knoten des Graphen, d.h. maximieren der Kanten, die in eine Richtung zeigen
- Kanten des Graphen sollen alle ungefähr gleich lang sein
- Plazierung der Knoten, so daß der Raum zwischen den Knoten ungefähr gleich groß ist
- Zu seinen Nachfolgerknoten zentrierter Vorgängerknoten

Natürlich können diese Anforderungen an die Graphendarstellung nicht alle gleichzeitig erfüllt werden, da es zu einem Konflikt zwischen den Anforderungen kommen kann. So kann das festhalten

an einer hierarchischen Darstellung des Graphen zu Kantenkreuzungen führen, die in einer anderen Darstellung vermeidbar wären (Abbildung 20). Welche Anforderungen an die Darstellung gestellt werden, hängt von der Anwendung, vom Typ des Graphen und vom Benutzer ab.

Eine weitere sehr wichtige Anforderung an den Darstellungsalgorithmus ist die Stabilität des Layouts. Darunter versteht man, daß bei einer Änderung am Graphen das Aussehen des Graphen nicht vollständig verändert wird. Dies würde die Benutzer eines Grapheditors verwirren. Besonders auffallend ist die fehlende Stabilität des Layouts, wenn ein gerade editierter Teilgraph aus dem Darstellungsfenster verschwindet.

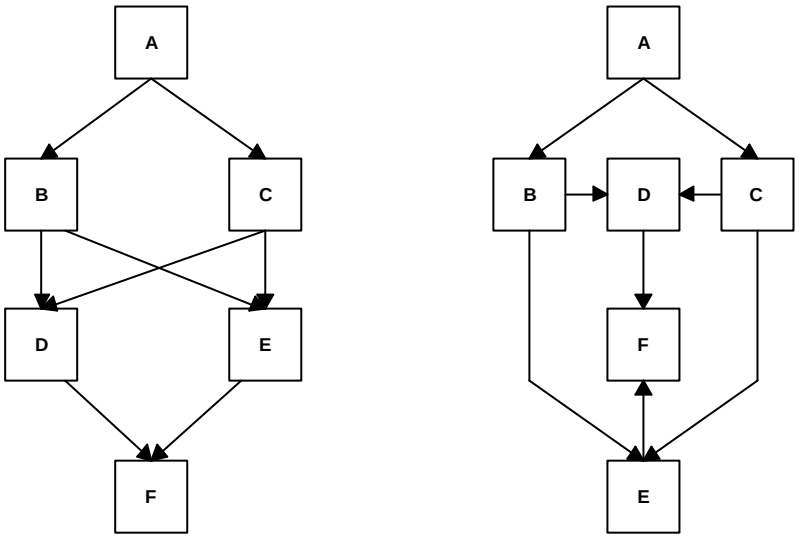


Abbildung 20: *Unterschiedliche Darstellungen eines Graphen*

Ein wichtiges Verfahren zur Darstellung eines gerichteten Graphen ist der Sugiyama-Algorithmus [FNP93]. Die ausführliche Beschreibung der Methode findet sich in der Arbeit [STT81]. Der Sugiyama-Algorithmus erzeugt eine hierarchische Darstellung eines Graphen und läßt sich drei Phasen aufteilen:

- **1. Phase**: Sie ordnet den Knoten verschiedene Ebenen im Graphen zu. Dies geschieht in der Weise, daß jeder Knoten unter der Ebene seiner Vorgängerknoten eingefügt wird. Diese Zuordnung geschieht durch eine topologische Sortierung der Knoten. Durch diese Sortierung war der ursprüngliche Sugiyama-Algorithmus nicht in der Lage, Zyklen zu verarbeiten. Für Zyklen werden deswegen Stellvertreterknoten („proxy nodes“) für den gesamten Zyklus eingeführt. Für „lange“ Kanten werden bei der Darstellung des Graphen unsichtbare Dummy-Knoten eingesetzt, um sicherzustellen, daß jede Kante im Graph nur eine Ebene überspannt. D.h. Kanten können nur von Knoten der Ebene e zu Knoten der Ebene e + 1 führen.

- **2. Phase**: Nun werden die Knoten auf einer Ebene so angeordnet, daß die Anzahl der Kantenkreuzungen zwischen den Ebenen minimiert wird.

- **3. Phase**: Sie dient dazu, die Darstellung des Graphen nochmals zu verbessern, indem die gesamte Kantenlänge des Graphen minimiert wird.

Ein Nachteil des Sugiyama-Algorithmus ist, daß er nicht immer zu einer optimalen Darstellung des Graphen führt. In einigen Fällen werden eliminierbare Kantenkreuzungen nicht entfernt und die topologische Sortierung in der ersten Phase des Algorithmus findet nicht immer eine optimale Zuordnung der Knoten zu den Ebenen. [FNP93] zeigt eine Reihe von Verbesserungen des Sugiyama-Algorithmus, die von verschiedenen Entwicklern im Laufe der Zeit vorgeschlagen wurden.

Ein weiterer Nachteil des Sugiyama-Algorithmus ist die Sortierung der Knoten auf den Ebenen in der zweiten Phase des Algorithmus. Dadurch kann ein stabiles Layout nicht gewährleistet werden, da die Knoten beim Neuaufbau des Graphen andere Positionen bekommen können.

Ein anderer Algorithmus, um eine hierarchische Struktur wie einen gerichteten Graphen darzustellen, ist der Algorithmus von Vilela, Maldonado und Jino [VMJ97] zur Darstellung von Programmgraphen. Ein Programmgraph stellt den Kontrollfluß eines Programms dar und hat einen Start- und einen Endknoten. Solch ein Kontrollflußgraph ist ähnlich dem Kontrollflußgraphen eines Workflows.

Verschiedene ästhetische Aspekte werden von den Entwicklern als Ziele für die Darstellung genannt:

- Hierarchische Struktur des Programmgraphen.
- Die semantische Informationen des Programms sollen erhalten bleiben, d.h. Strukturen wie z.B. eine if ... then-Verzweigung sollen im Graphen sofort erkennbar sein.
- Die visuelle Harmonie soll gewährleistet sein. Darunter verstehen die Entwickler des Algorithmus, daß z.B. der Graph möglichst kurze Kanten besitzt, Knoten sich nicht überlappen und Vorgängerknoten über ihren Nachfolgern zentriert sein.

Wie arbeitet der Algorithmus nun? Zwei Hauptaufgaben lassen sich identifizieren. Erstens muß der Algorithmus x- und y-Koordinaten für die Knoten finden und zweitens müssen geeignete Wege zwischen den Knoten für die Kanten gefunden werden.

Im ersten Teil des Algorithmus werden den Knoten, wie im Sugiyama-Algorithmus, bestimmten Ebenen zugeordnet. Im Unterschied zum Sugiyama-Algorithmus werden dabei die Schleifenkanten einfach ignoriert. Durch diesen Teil des Algorithmus werden die y-Koordinaten der Knoten bestimmt.

Die Bestimmung der x-Koordinaten erfolgt im nächsten Schritt. Dies passiert durch ein Konzept mit dem Namen scope. Es gibt an, wieviel Platz auf der virtuellen x-Achse benötigt wird, um einen Knoten mit seinen Nachfolgern ohne Überlappung zu positionieren. Abbildung 21 verdeutlicht das Konzept scope. Dabei geben die kursiven Zahlen rechts oberhalb den scope des Knotens an. Die Berechnung des scope eines Knotens innerhalb des Algorithmus erfolgt rekursiv. Mit diesem Wert kann dann die endgültige Position eines Knotens berechnet werden. Eine Sortierung der Knoten wie im Sugiyama-Algorithmus wird in diesem Algorithmus nicht durchgeführt, da durch Änderungen der Knotenpositionen semantische Informationen verloren gehen könnten.

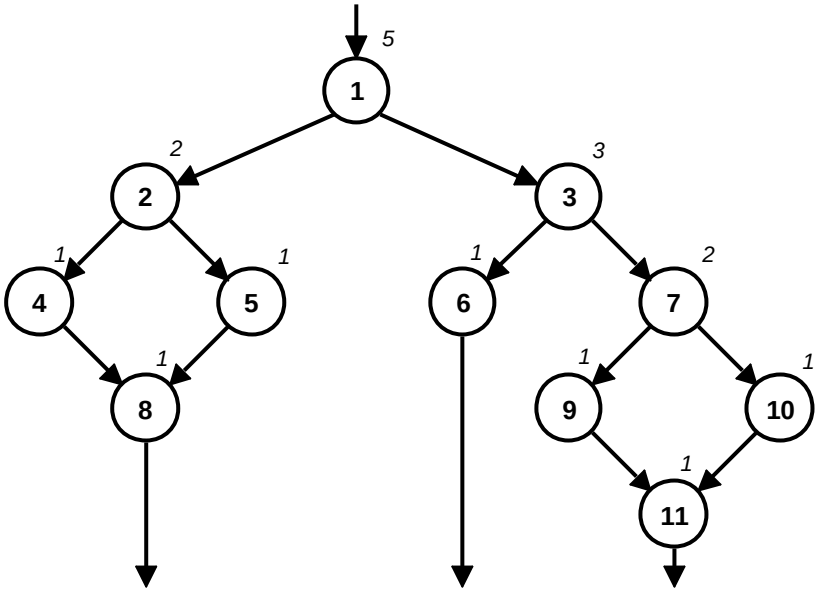


Abbildung 21: *Darstellung des scope eines jeden Knoten*

Nachdem nun die x- und y-Koordinaten der Knoten bestimmt worden sind, muß der Verlauf der Kanten festgelegt werden. Dazu wird jeder Knoten in ein Gitter eingefügt. Durch dieses Gitter wird nun der Verlauf der Kanten durch ein aufwendiges Verfahren berechnet. Dabei muß beachtet werden, daß verschiedene Positionen im Gitter schon durch die Knoten belegt sind.

Die ausführliche Beschreibung der obigen Verfahren zur Bestimmung der x- und y-Koordinaten eines Knotens und zum Festlegen des Verlaufs einer Kante ist in [VMJ97] zu finden.

Beide vorgestellten Algorithmen haben Nachteile, die es erschweren, einen Ablaufgraphen, der aus Konstrukten des ADEPT-Basismodells besteht, darzustellen. So ist ein Nachteil beider Algorithmen, daß sie nur mit einer Kantenart arbeiten. Das ADEPT-Basismodell besitzt aber mehrere Kantenarten (z.B. Kontrollkanten, Fehlerkanten,…). Außerdem kann der vorgestellte Sugiyama-Algorithmus kein stabiles Layout des Graphen liefern. Das Verfahren aus [VMJ97] hat den Nachteil, daß es eine für Modellierungszwecke unübersichtliche Darstellung des Prozeßgraphen erzeugen würde. So könnten Schleifenkanten nicht schnell im Graph identifiziert werden.

Aufgrund dieser Nachteile wurde in dieser Arbeit ein eigener Algorithmus entwickelt. Einige Konzepte, wie z.B. unsichtbare Dummy-Knoten und die Anordnung der Knoten in einem Gitter, konnten den vorgestellten Algorithmen entlehnt werden. Allerdings mußten diese Konzepte den Anforderungen des eigenen Algorithmus angepaßt werden.

Die Anforderungen an den eigenen Algorithmus zur Darstellung des Kontrollflußgraphen sind:

- Hierarchische Darstellung des Graphen von links nach rechts, d.h. alle Kontrollflußkanten zeigen mit ihrer Pfeilspitze nach rechts. Schleifen und Verzweigungen, die neu eingefügt werden, vergrößern den Graph nach unten.

- Keine Kreuzungen zwischen Kontrollflußkanten und Schleifenkanten. Kreuzungen anderer Kantenarten des ADEPT-Basismodells lassen sich unter diesen Voraussetzungen nicht verhindern.

- Knoten mit der gleichen Tiefe im Graphen werden auf gleicher Ebene angeordnet.

- Stabiles Layout

Die Abbildung 22 zeigt die gewünschte Darstellung des Graphen an einem Beispiel.

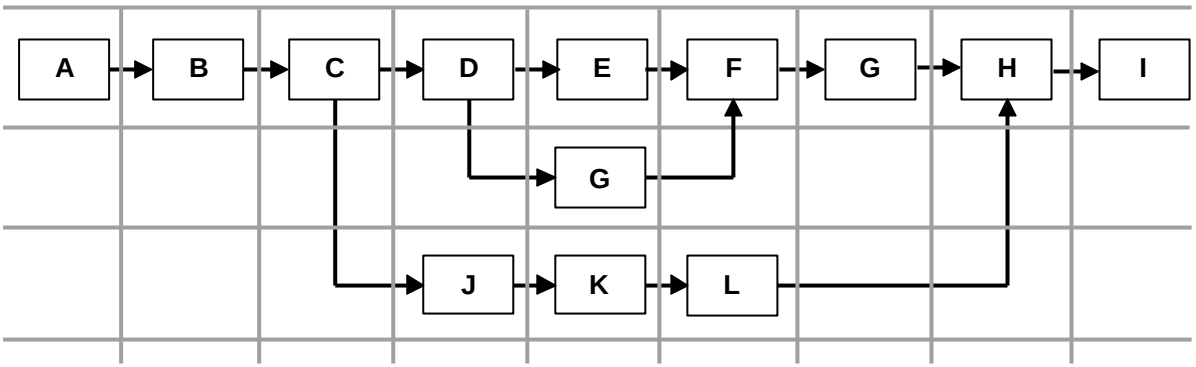


Abbildung 22: *Darstellung eines Graphen im Gitter*

Die Grundlage für die Darstellung ist die Anordnung der Knoten des Graphen in einem Gitter. Ähnliche Darstellungsformen haben sich in Arbeiten der Abteilung DBIS [Kir96] und in der Literatur bewährt. Das Gitter wird innerhalb des Programms als Matrix implementiert. Dabei bilden Gitterpositionen, die keinen Knoten enthalten, später einen Leerraum auf dem Bildschirm. Nachdem alle Knoten des Graphen in das Gitter eingefügt worden sind, werden die x- und y-Positionen der Knoten auf dem Bildschirm berechnet. Die Kanten des ADEPT-Basismodells werden erst nach dem Positionieren der Knoten auf dem Bildschirm eingefügt. Dabei wird mit Hilfe der Koordinaten der Knoten der Verlauf der Kanten bestimmt. Die hier dargestellten Kanten haben nur die Aufgabe, die Abfolge der Knoten im Graphen zu illustrieren.

Der Algorithmus, der die Aktivitäten des Workflows in das Gitter einfügt, gliedert sich in zwei Teile. Der erste Teil ist die Bestimmung der „Basislinie“ des Graphen. Die „Basislinie“ ist die oberste Zeile im Gitter bzw. bei der späteren Darstellung, die auch den Start- und Endknoten enthält. Das Einfügen dieser Knoten geschieht in der Funktion `fillMatrix`. Trifft der Algorithmus auf eine Verzweigung oder Schleife, so wird die Funktion `fillBranch` aufgerufen. Die Aufgabe dieser Funktion ist es einen abzweigenden Ast in des Gitter einzufügen. Stößt die Funktion `fillBranch` wieder auf eine Verzweigung oder eine Schleife, so wird wiederum die Funktion `fillBranch` rekursiv aufgerufen.

Nachdem der Algorithmus nun kurz beschrieben wurde, läßt sich seine Funktionsweise am besten an einem Beispiel verdeutlichen. In diesem Beispiel soll der Graph aus Abbildung 22 in das Darstellungsgitter eingefügt werden.

Voraussetzung für den Algorithmus ist, daß jedem Knoten ein bestimmter Typ zugeordnet ist. Dieser Typ darf nicht verwechselt werden mit den Aktivitätentypen, die das ADEPT-Basismodell definiert! Die hier beschriebenen Typen dienen allein der Generierung der Darstellung des Graphen.

Folgende Knotentypen sind für den Aufbau des Graphen notwendig:

- *start* für den Startknoten
- *end* für den Endknoten

- *loopstart* für einen Schleifenstartknoten
- *loopend* für einen Schleifenendknoten
- *verz* für einen Verzweigungsknoten
- *sync* für einen Synchronisationsknoten
- *norm* für alle anderen Knoten

Als weiterer Vorbereitungsschritt müssen alle Schleifen aus dem Graphen entfernt werden. Diese Operation wird nicht im Originalgraphen durchgeführt, sondern in einer Kopie des Graphen, die nur Kontroll- und Schleifenkanten enthält. Der Grund für die Ersetzung der Schleifen ist, daß bei der Positionierung der Knoten nicht im Graph zurückgelaufen werden darf und damit Knoten noch mal betrachtet werden. Da Schleifenkanten in der späteren Darstellung auf der selben Höhe wie die Knoten verlaufen sollen, werden die Schleifen durch eine Verzweigung ersetzt. Die Schleifenkante wird durch zwei Kontrollkanten und einen Dummy-Knoten substituiert, der später bei der Darstellung auf dem Bildschirm nicht sichtbar ist. Die Typen des früheren Schleifenstartknotens (*loopstart*) und Schleifenendknotens (*loopend*) bleiben bei der Ersetzung der Schleifen durch Verzweigungen erhalten. Abbildung 23 zeigt das Vorgehen beim Ersetzen von Schleifen durch eine Verzweigung.

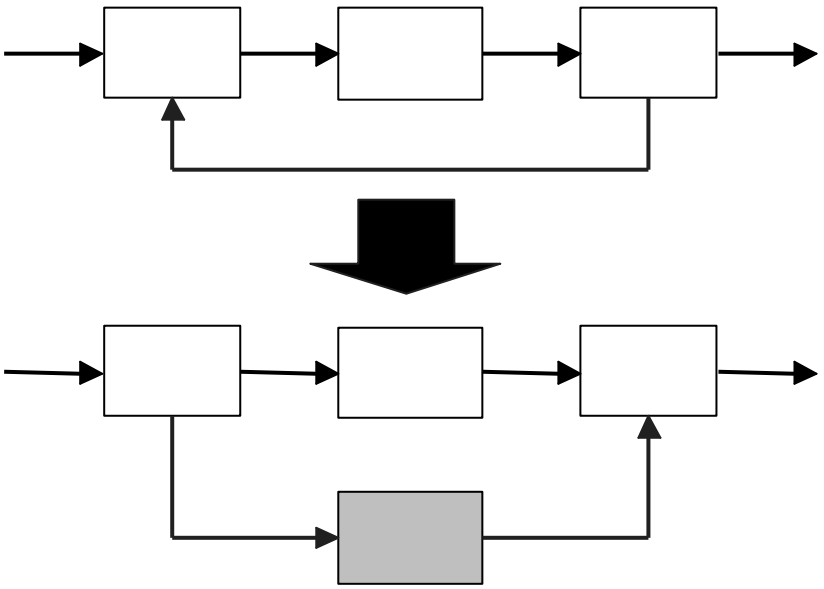


Abbildung 23: *Schleifen werden durch Verzweigungen mit einem Dummy-Knoten ersetzt*

Zuerst wird damit begonnen, die „Basislinie“ des Ablaufgraphen in das Gitter einzufügen. Der Anfangsknoten ist dabei immer der Startknoten des Workflow (im Beispiel der Knoten A). Trifft nun der Algorithmus auf einen Knoten des Typs *verz* oder *loopstart*, so wird der Knoten mit seiner Position im Gitter auf einem Stack gespeichert. Dies ist notwendig, um später abzweigende Äste in das Gitter einzufügen. Alle folgenden Knoten vom Typ *norm* werden wie gewohnt in das Gitter eingefügt. Im Beispiel sind solche Verzweigungsknoten die Knoten C und D. Der

Verzweigungsknoten C bildet mit dem Synchronisationsknoten H eine Verzweigung und zum Verzweigungsknoten D gehört der Synchronisationsknoten F (siehe Abbildung 22).

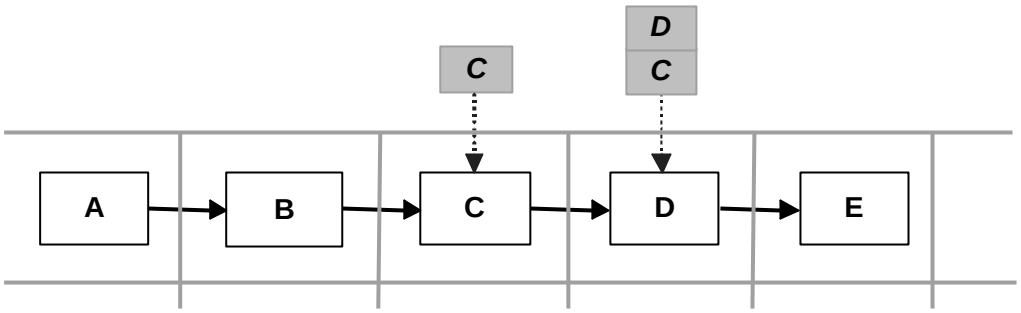


Abbildung 24: *Einfügen der Knoten in das Gitter durch die Methode fillMatrix*

Wenn der Algorithmus auf einen Knoten vom Typ *sync* oder *loopend* trifft, wird die Funktion `fillBranch` zum Einfügen eines Astes aufgerufen. Die Abbildung 24 zeigt den Zustand des Gitters in dem Moment, in dem der Knoten F gelesen wird. Zu beachten ist, daß der Knoten F zwar gelesen, aber noch nicht in das Gitter eingefügt wird, da zuerst wird die Funktion `fillBranch` aufgerufen wird. Der Funktion `fillBranch` werden die Spalte und die Zeile im Gitter für den ersten Knoten des neuen Astes übergeben. Um an diese Informationen zu gelangen, muß der oberste Stackeintrag gelesen und gelöscht werden. Dieser oberste Eintrag enthält immer den letzten Verzweigungsknoten, der zum gelesene Synchronisationsknoten gehört.

Die Funktion `fillBranch` ordnet nun die Knoten des neuen Astes im Gitter an. Der erste Knoten wird dabei an die durch den übergebenen Spalten- und Zeilenwert gekennzeichnete Position im Gitter gesetzt. Für eine Schleife wird dabei ein später unsichtbarer Dummy-Knoten in die Matrix eingeordnet. Falls in diesem Ast wieder eine Verzweigung zu finden ist, wird die Funktion `fillBranch` rekursiv aufgerufen. Die nachfolgende Abbildung zeigt die belegten Gitterpositionen nach Ablauf der Funktion `fillBranch`.

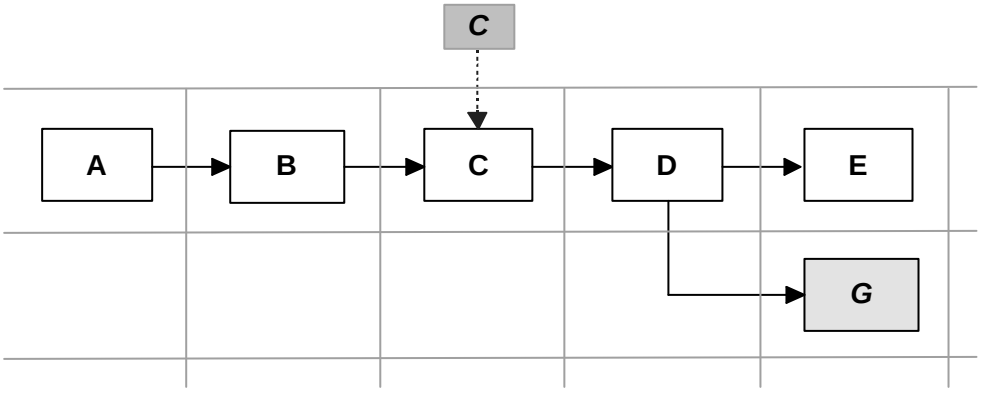


Abbildung 25: *Einfügen eines Knotens G durch die Methode fillBranch*

Nachdem die Funktion `fillBranch` beendet wurde, wird nun der gelesene Verzweigungsknoten von der Funktion `fillMatrix` in das Gitter gesetzt. Die Funktion `fillBranch` gibt für diesen Knoten die richtige Spaltenposition im Gitter zurück. Dieses Vorgehen ist notwendig, da der längste Ast einer Verzweigung die Spaltenposition des Sychronisationsknotens der Verzweigung bestimmt. Außerdem gibt die Funktion `fillBranch` noch ihre maximale Zeilenposition an. Diese Information ist zum Einfügen von umschließenden Ästen einer neuen Verzweigung in das Gitter notwendig.

Die Funktion `fillMatrix` übernimmt nun wieder die Kontrolle und ordnet die Aktivitäten auf der „Basislinie“ an. Dies geschieht solange, bis ein neuer Synchronisationsknoten einer Verzweigung gefunden wird, wobei das Einfügen einer neuen Verzweigung in das Gitter dann wieder so abläuft wie im vorherigen Schritt beschrieben. Die Abbildung 26 zeigt den Stand nach dem Einfügen der zweiten Verzweigung durch die Funktion `fillBranch`.

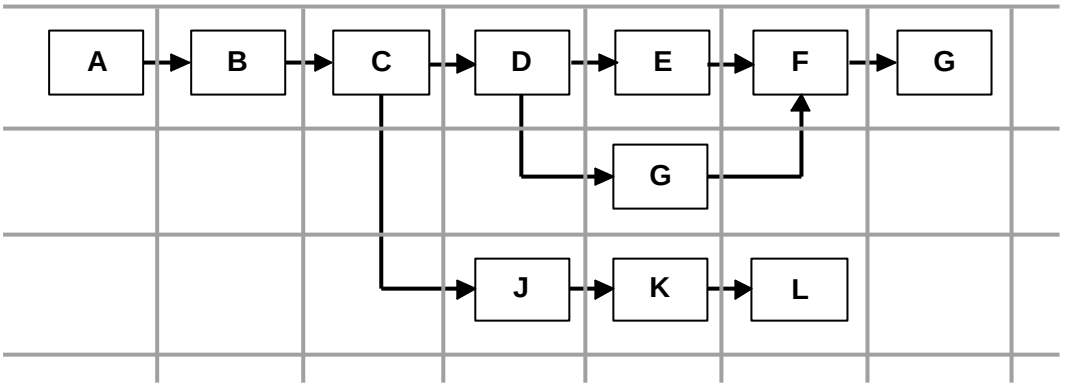


Abbildung 26: *Einfügen eines zweiten Zweigs*

Wenn das Einfügen der letzten Verzweigung abgeschlossen ist, verarbeitet die Funktion `fillMatrix` den Synchronisationsknoten und die restlichen Knoten des Ablaufgraphen. Dieser Prozeß endet, wenn der Endknoten des Workflows (im Beispiel der Knoten I) erreicht wird. Die Funktion `fillMatrix` gibt als Ergebnis die maximale Spalten- und Zeilennummer des Graphen im Gitter zurück. Aufgrund dieser Information läßt sich dann die benötigte Zeichenfeldgröße für den gesamten Graphen berechnen.

4.3 Algorithmen zur Graphanalyse

4.3.1 Zyklensuche

Ein Problem, das bei der Modellierung eines Workflow-Schemas auftaucht, ist die mögliche Entstehung von Zyklen. Entsteht beim Einfügen einer Sync-Kante in den Graphen ein Zyklus der aus Kontroll- und Sync-Kanten besteht, so darf die neue Sync-Kante nicht eingefügt werden. Würde die

neue Sync-Kante doch eingefügt, so würde es bei der Ausführung des Workflows zu einer Verklemmung kommen.

Diese Problemstellung läßt sich durch eine Zyklensuche im gesamten Graphen lösen. Dabei müssen im Editor nur die Kontroll- und Sync-Kanten betrachtet werden. Mögliche Zyklen durch andere Kantenarten können zu keiner Verklemmung bei der Abarbeitung des Workflows führen.

Ein Algorithmus für die Zyklensuche in einem Graphen, der als Datenstruktur eine Adjazenzliste benutzt, wird in [NeM93] vorgestellt. Er basiert auf der topologischen Sortierung der Knoten eines Graphen und kann auch zur Zyklensuche in einem Graphen verwendet werden. Der Algorithmus stoppt dabei beim ersten Auftreten eines Zyklusses. Wird die topologische Sortierung der Knoten nicht benötigt, so kann der Algorithmus vereinfacht werden.

Die topologische Sortierung eines gerichteten Graphen ist die Anordnung seiner Knoten $V = \{v_1,v_2,...,v_n\}$, so daß für alle Kanten gilt (v_i, v_j) gilt: $i < j$. Der Algorithmus zur Zyklensuche macht sich zunutze, daß Graphen die Zyklen enthalten, nicht topologisch sortiert werden können. Eine topologische Sortierung ist nur bei azyklischen Graphen möglich [Shö96]!
Vereinfacht ausgedrückt entfernt der Algorithmus solange Knoten ohne Vorgänger aus dem Graphen, bis entweder der Graph keinen Knoten mehr besitzt oder es keinen Knoten ohne Vorgänger mehr gibt. Stoppt der Algorithmus und sind noch Knoten im Graphen vorhanden, so enthält der Graph einen Zyklus.

Der vereinfachte Algorithmus zur Zyklensuche gliedert sich in zwei Schritte. Der erste Schritt ist die Initialisierungsphase, auf die dann die eigentliche Zyklensuche in Schritt 2 folgt.

Die Initialisierungsphase (Schritt 1) verläuft folgendermaßen:

CycleCheck() → success ∈ {TRUE, FALSE}

input:
P : Prozeßgraph

output:
success : Boolscher Wert

begin
Q = ∅ ❶
forall (u ∈ V) **do**
$d^-(u) := 0$ ❷
end
forall (u ∈ V) **do**
forall ((u,v) ∈ V) **do**
$d^-(v) := d^-(v) + 1$ ❸
end
end

s := Bestimme Startknoten des Graphen G
Q.**push**(s) ❹
z := 1

Zuerst wird eine FIFO[1]-Queue Q erzeugt, die die Knoten ohne Vorgänger aufnimmt ❶. Dann wird der Eingangsgrad d^- eines jeden Knoten mit 0 initialisiert ❷. Der Eingangsgrad d^- eines Knotens gibt an, wieviele direkte Vorgänger ein Knoten hat bzw. wieviele gerichtete Kanten an dem Knoten enden. In ❸ wird für jeden Knoten v aus der Nachfolgermenge des Knotens u der Eingangsgrad $d^-(v)$ inkrementiert. Danach wird der einzige Knoten ohne Vorgänger (der Startknoten des Workflows) in Q geschrieben ❹. Die Anzahl der verarbeiteten Knoten wird anschließend auf 1 gesetzt und die Zyklensuche gestartet.

Die Zyklensuche (Schritt 2) sieht folgendermaßen aus:

```
while (Q ≠ ∅) do                                    ❺
    u = Q.pop()
    forall ((u,v) ∈ E) do
        d⁻(v) = d⁻(v) − 1                           ❻
        if (d⁻(v) = 0) then
            Q.push(v)                               ❼
            z = z + 1
        end
    end
end

if (z = |V|) then                                   ❽
    return TRUE
else
    return FALSE
end
end
```

Die while-Schleife der Zyklensuche läuft solange, bis keine Knoten ohne Vorgänger mehr vorhanden sind ❺, d.h. die Menge Q ist leer. Dies kann der Fall sein, wenn alle Knoten verarbeitet worden sind oder der Algorithmus auf einen Zyklus im Graphen stößt.

In der Schleife wird zuerst das erste Element u der Queue Q gelesen und aus Q entfernt. Danach wird für jeden Knoten v aus der Nachfolgermenge von u der Eingangsgrad dekrementiert ❻. Ist der Wert des Eingangsgrads eines Knotens gleich 0, so wird er in Q geschrieben und die Anzahl der verarbeiteten Knoten wird erhöht ❼.

Wird die Schleife beendet und ist die Anzahl der verarbeiteten Knoten gleich der Anzahl der Knoten des Graphen ❽, so ist der Graph zyklenfrei. Andernfalls enthält der Graph einen Zyklus.

4.3.2 Breitensuche

Ein sehr wichtiger Algorithmus bei der Implementierung eines Workflow-Editors ist die Breitensuche (breadth-first-search). Auch sonst ist die Breitensuche ein Algorithmus, der in fast jedem Programm, das mit Graphen arbeitet, benötigt wird. Die Breitensuche bildet die Grundlage für verschieden

[1] FIFO = first in, first out

Algorithmen wie z.B. die Bestimmung von Vorgänger- und Nachfolgermengen eines Knotens und für die Bestimmung des kürzesten Wegs von u nach v. Um die Vorgängermenge über die Breitensuche bestimmen zu können, müssen die gerichteten Kanten des Graphen logisch gedreht werden, da nun alle Knoten bis zum Startknoten gesucht werden.

Bei der Breitensuche werden zuerst alle Knoten bearbeitet, die sich in einer bestimmten Entfernung vom Startknoten befinden.

Bei der Bestimmung der Nachfolgermenge eines Knotens kann folgender auf der Breitensuche basierende Algorithmus benutzt werden:

GetSuccessorSet(n) $\rightarrow S \subset V$

```
input:
   P          : Prozeßgraph

output:
   S          : Nachfolgermenge
begin
   Q := ∅                                                    ❶
   Q.push(n)

   S := ∅;                                                   ❷

   while (Q ≠ ∅) do                                          ❸
        u := Q.pop();
        forall ((u,v) ∈ E) do                                ❹
             if (v ∉ S) then
                  S := S ∪ {v}                               ❺
                  Q.push(v)
             end
        end
   end

   return S                                                  ❻
end
```

Der Algorithmus zur Bestimmung legt zuerst eine FIFO-Queue Q an, in der alle Knoten gespeichert werden, die schon „entdeckt“ wurden ❶. Als erster Knoten wird der Startknoten in Q gespeichert. Dann wird die Menge S der Nachfolger mit ∅ initialisiert ❷.

In der Schleife ❸, die abgebrochen wird, wenn kein Knoten im Graph mehr vorhanden ist wird das jeweils erste Element u aus Q gelesen und entfernt. Danach werden alle Kanten mit Startknoten u gesucht ❹. Alle Endknoten v der gefundenen Kanten, werden, sofern sie noch nicht in der Nachfolgerliste S gespeichert sind, dort gespeichert und in Q geschrieben. Wenn alle Knoten, die Nachfolgerknoten des Ursprungsknoten n sind, gefunden worden sind, wird die Nachfolgerlist S als Ergebnis zurückgegeben.

Eine weitere Anwendung des Breitensuche-Algorithmus ist die Bestimmung des kürzesten Pfades zwischen den Knoten u und v in einem Graph. Eine Bedingung, damit der Algorithmus funktioniert, ist, daß der Knoten v ein Nachfolger des Knotens u ist. Beim Einfügen von Fehlerkanten muß ein Pfad vom Endknoten zum Startknoten der Fehlerkante bestimmt werden. Der Grund für die Notwendigkeit

des Pfades ist, daß zwischen dem Start- und Endknoten der Fehlerkante kein offener Synchronisationsknoten einer Verzweigung mit „1 aus n“-Semantik sein darf. Um die Existenz eines solchen Knotens festzustellen, müssen alle Knoten aus dem Pfad betrachtet werden. Natürlich ließe sich das Problem auch mit jedem anderen Pfad vom Endknoten zum Startknoten der Fehlerkante lösen, aber der kürzeste Pfad enthält die geringste Anzahl Knoten.

Die Beschreibung des Algorithmus zur Bestimmung des kürzesten Pfades zwischen zwei Knoten bezieht sich auf einen Graphen mit nur einer Kantenart. Im Gegensatz dazu müssen im Editor für das ADEPT-Basismodell nur Kontrollkanten betrachtet werden. Alle anderen Kantenarten werden ignoriert.

FindPathAToB(n_1,n_2) $\rightarrow$ Path $\subset$ V

```
input
   P          : Prozeßgraph

output
   Pfad       : Feld, das den Pfad von u nach v repräsentiert
begin
   forall (u ∈ (V \ n1)) do                                    ❶
        farbe(u) := weiß
   end

   farbe(n1) := grau                                           ❷
   Q.push(n1)

   VorgängerVon := ∅                                           ❸

   while (Q ≠ ∅) do                                            ❹
      u := Q.pop()
   end

   forall ((u,v) ∈ E) do                                       ❺
      if (farbe(v) = weiß) do                                  ❻
         v := grau
         VorgängerVon := VorgängerVon ∪ {(v,u)}                ❼
         Q.push(v)
      end

      u := schwarz                                             ❽
   end

   // Code, der kürzesten Pfad von n1 nach n2 aus VorgängerVon-Menge ermittelt
   s := n2
   Pfad.add(s)                                                 ❾
   while (s ≠ n1) do
      s = VorgängerVon(s).second()                             ❿
      Pfad.addFront(s)
   end

   return Pfad
end
```

Der Algorithmus beginnt damit, daß zuerst alle Knoten des Graphen, außer dem Startknoten des Pfades, weiß gefärbt werden ❶. Die Färbungen der Knoten, die im Algorithmus auftauchen, haben folgende Bedeutungen: Weiß bedeutet, daß der Knoten noch nicht besucht wurde. Grau bedeutet, daß der Knoten zwar besucht wurde, aber die Betrachtung noch nicht abgeschlossen ist. Die Farbe Schwarz steht für alle besuchten und vollständig betrachteten Knoten. Der Startknoten des Pfades n_1 wird anschließend grau gefärbt und in die FIFO-Queue Q geschrieben ❷. Q nimmt alle Knoten auf, die noch betrachtet werden müssen. Danach wird die VorgängerVon-Menge mit ∅ initialisiert ❸. Die VorgängerVon-Menge nimmt Knotenpaare auf. Die Bedeutung eines Knotenpaares (x,y) ist, daß Knoten y ein Vorgänger von Knoten x ist. Aufgrund dieser Menge wird später der kürzeste Pfad vom Knoten n_1 nach n_2 bestimmt.

Die while-Schleife ❹ wird solange durchlaufen, bis keine Knoten mehr betrachtet werden müssen. In jeder Iteration der while-Schleife wird das erste Element von Q gelesen und aus der Queue gelöscht. In ❺ werden alle Nachfolger v des Knotens u bestimmt. Ist ein Nachfolgerknoten noch weiß ❻, so wird er grau gefärbt und in der VorgängerVon-Menge mit dem Paar (v, u) gespeichert ❼. Außerdem wird der Knoten v noch in Q zur Bearbeitung gespeichert. Jeder Knoten, dessen Betrachtung abgeschlossen ist, wird am Ende der while-Schleife schwarz gefärbt ❽.

Nun beginnt die eigentliche Bestimmung der Knoten des kürzesten Pfads von n_1 nach n_2. Zuerst wird der Endknoten des Pfads n_2 dem Feld Pfad als erstes Element hinzugefügt ❾. Dann wird der Pfad mit Hilfe der Elemente der VorgängerVon-Menge aufgebaut. In der VorgängerVon-Menge wird dann das Knotenpaar gesucht, dessen erstes Element der Knoten s ist ❿. Das zweite Element aus dem Knotenpaar (der Vorgängerknoten von s) ist dann der nächste Knoten des kürzesten Pfades. Damit wird der Pfad vom Endknoten des Pfades aus über die Vorgängerknoten aufgebaut. Der Algorithmus macht sich dabei eine besondere Eigenschaft der VorgängerVon-Menge zunutze. In ❻ läßt sich ein neues Knotenpaar nur erzeugen, wenn der Nachfolgerknoten weiß ist. Bei einem Synchronisationsknoten einer Verzweigung wird ein Knotenpaar mit dem Synchronisationsknoten der Verzweigung nur für den kürzesten Pfad durch die Verzweigung aufgebaut. Ein längerer Pfad durch die Verzweigung wird nicht hinzugenommen, da der Synchronisationsknoten schon eine andere Farbe als Weiß besitzt. Deswegen wird nur der kürzeste Pfad durch eine Verzweigung in den kürzesten Pfad zwischen den Knoten n_1 und n_2 aufgenommen.

Jeder weitere Knoten des kürzesten Pfads von n_1 nach n_2 muß nun als erstes Element in das Feld Pfad eingefügt werden, damit die richtige Reihenfolge der Knoten von n_1 nach n_2 erhalten bleibt. Nachdem dies alles geschehen ist, kann das Feld Pfad als Ergebnis zurückgegeben werden.

4.3.3 Tiefensuche

Neben der Breitensuche ist die Tiefensuche ein weiterer wichtiger Graphalgorithmus. Die Bestimmung der Nachfolger- bzw. Vorgängermenge eines Knotens kann auch über die Tiefensuche implementiert werden.

Wie der Name schon sagt, sucht die Tiefensuche zuerst „tiefer“ als die Breitensuche in den Graph hinein. Bei der Tiefensuche werden im Gegensatz zur Breitensuche alle Kanten betrachtet, die vom letzten besuchten Knoten ausgehen. Wenn alle ausgehenden Kanten eines Knotens n betrachtet worden sind, geht der Algorithmus zu dem Knoten zurück, von dem aus der Knoten n erreicht worden ist, und untersucht die weiteren ausgehende Kanten. Dieser Prozeß geht so lange, bis alle von einem vorgegebenen Startknoten erreichbaren Knoten entdeckt worden sind.

Eine einfache rekursive Implementierung der Tiefensuche findet sich in [Shö96]. Ein Algorithmus der die Nachfolgersuche bewerkstelligt und auf der Tiefensuche basiert, sieht folgendermaßen aus:

GetSuccessorSet(u)

input
P : Prozeßgraph
S : Nachfolgermenge (initialisiert mit ∅)

begin
farbe(u) := grau ❶
forall ((u,v) ∈ E) do
if (farbe(v) = weiß) then DepthFirstSearch(v) ❷
end
farbe(u) := schwarz ❸
S := S ∪ {u} ❹
end

Der rekursive Algorithmus läuft wie folgt ab: Zuerst sind alle Knoten des Graphen weiß. Die im Algorithmus benutzten Farben für die Knoten entsprechen in ihrer Bedeutung genau den Farben im Algorithmus FindPathAToB. Dann wird jeder besuchte Knoten u grau gefärbt ❶. Anschließend wird die direkte Nachfolgermenge des grau gefärbten Knotens u bestimmt. Beim ersten Knoten v der Nachfolgermenge, der noch weiß ist, wird DepthFirstSeach mit dem Knoten v aufgerufen ❷. Kehrt dieser Aufruf zurück, werden die restlichen Knoten v der direkten Nachfolgermenge von u untersucht. Wenn kein weißer Nachfolgerknoten mehr vorhanden ist, dann wird der Knoten u schwarz gefärbt ❸ und in die Nachfolgermenge des Ursprungsknotens n aufgenommen ❹.

Eine iterative Implementierung der Tiefensuche findet sich in [Sed93].

4.4 Datenfluß-Analysealgorithmen

In diesem Kapitel werden verschiedene Algorithmen vorgestellt, die zur Gewährleistung der Korrektheit des Prozesses benutzt werden. Dabei handelt es sich vor allem um Algorithmen zur Korrektheitsprüfung des Datenflusses. Die hier vorgestellten Algorithmen wurden in [Hen97] formal definiert. Sie dienten als Basis für die im Editor verwendeten Algorithmen, da sie den Anforderungen des Editors angepaßt werden mußten.

4.4.1 Die Vermeidung überflüssiger Schreiboperationen

Der Zweck dieses Algorithmus ist zu prüfen, ob ein Wert in einem Datenslot unbenutzt überschrieben wird. Dies passiert, wenn nach einem Schreibzugriff einer Aktivität wieder ein Schreibzugriff einer anderen Aktivität auf den gleichen Datenslot erfolgt. Dabei wird der erste geschriebene Wert nicht gelesen und somit auch nicht gebraucht! Solch eine Konstellation wird nicht als Fehler betrachtet, sondern wird nur als Warnung angezeigt, da die Ausführung des Prozesses nicht behindert wird. Aufeinanderfolgende Schreiboperationen, die durch eine Schleifeniteration entstehen können, werden von dem Algorithmus nicht betrachtet.

unnecessaryWrites() → E

input:
D: Menge der Datenslots
P: Prozeßgraph

output:
E: Menge der Knoten die hintereinander einen Datenslot beschreiben

```
begin
  forall (d ∈ D) do                                                    ❶
    Writers := suche alle Schreiberknoten des Datenslots d
    Readers := suche alle Leseknoten des Datenslots d

    if (|Writers| > 1) do                                              ❷

      forall (w1 ∈ Writers) do
        Succ_n := bestimme alle Nachfolger von w1 über Kontroll- und Sync-Kanten

        forall (w2 ∈ Writers) do
          Pred_n := bestimme alle Vorgänger von w2 über Kontroll- und Sync-Kanten

          //Alle Knoten zwischen w1 und w2 bestimmen
          Nodes := Succ_n ∩ Pred_n                                     ❸

          //Test, ob in Nodes ein Schreiber von d ist
          //⇒ Abbrechen der Analyse
          existsWriter := FALSE
          forall (n ∈ Nodes) do                                        ❹
            if (n ∈ Writers) do
              existsWriter := TRUE
              break
            end
          end

          if (existsWriter = FALSE) do                                 ❺
            //w2 der Menge Nodes hinzufügen, da auch
            //er ein Leser sein kann
            Nodes := Nodes ∪ {w2}

            existsReader := FALSE

            forall (r ∈ Readers) do                                    ❻
              if (r ∈ Nodes) do
                existsReader := TRUE
                break
              end
            end

            if (existsReader = FALSE) do                               ❼
              E := E ∪ {(w1,w2,d)}
            end
          end
      end
  end
```

return E

end

In der ersten Schleife des Algorithmus werden jeweils für eine Datenslot die Schreib- und Leseknoten bestimmt ❶. Erst wenn mindestens zwei Schreibknoten für einen Datenslot existieren, kann ein überflüssiges Schreiben überhaupt auftreten ❷. Ansonsten muß der Test für den Datenslot nicht durchgeführt werden. Muß der Test auf unnötiges Schreiben durchgeführt werden, so werden nun alle Knoten über Kontroll- und Sync-Kanten zwischen den zwei Schreibknoten bestimmt ❸. Sollte sich zwischen diesen Knoten ein weiterer Schreibknoten befinden ❹, so kann die Analyse des Knotenpaares beendet werden, da zur Analyse die zwei Schreibknoten aufeinander folgen müssen und kein weiterer Schreibknoten sich zwischen ihnen befinden darf. Würde dieser Test nicht ausgeführt werden, dann würden redundante Konfliktmeldungen ausgegeben werden.

Sind alle Bedingungen erfüllt ❺, so wird der Knoten w_2 der Menge der Zwischenknoten hinzugefügt, da auch er ein Leser des Datenslots sein kann. Nun wird geprüft ob die Menge der Zwischenknoten einen Leser enthält ❻. Wenn die Menge der Zwischenknoten keine Leser enthält, so werden die zwei Knoten und der Datenslot der Ergebnismenge E hinzugefügt ❼. Als Ergebnis der Funktion werden alle unnötigen Schreiboperationen in der Menge E zurückgeliefert.

4.4.2 Die Vermeidung paralleler Schreiboperationen

Der in diesem Abschnitt vorgestellte Algorithmus prüft in jeder parallelen Verzweigung des Ablaufs, ob Aktivitäten in unterschiedlichen Zweigen eine Datenslot gleichzeitig beschreiben. Dies darf nicht vorkommen, da sonst nicht festgelegt werden kann, welcher Wert von nachfolgenden Knoten gelesen wird. Besonderes Augenmerk bei diesem Algorithmus muß auf die Sync-Kanten gelegt werden, da sie es in parallelen Verzweigungen wieder erlauben, daß zwei Knoten in unterschiedlichen Zweigen einen Datenslot beschreiben. Durch die Sync-Kante wird ein Knoten wieder zum Nachfolger des anderen Knotens.

Der Algorithmus besteht aus folgenden formal dargestellten Schritten:

parallelWrites() $\rightarrow$ E

input:
P: Prozeßgraph
D: Menge der Datenslots

output:
E: Menge der Knoten die eine Datenslot parallel beschreiben

begin
PBranch := Suche alle parallelen Verzweigungen und
speichere alle ihre Knoten ❶

//Wenn eine parallele Verzweigung im Graphen exisitiert
if ($\exists$ p $\in$ Pbranch) **do**

```
PBranch := Pbranch \ (alle von parallelen Verzweigungen eingeschlossenen parallelen
            Verzweigungen)                                                    ❷

forall (p ∈ Pbranch) do
  forall (d ∈ D) do
    Writers := alle Knoten suchen, die den Datenslot d beschreiben

    if (|Writers| > 1) do                                                     ❸
      for (i = 1,...,|Writers| -1) do
        for (j = i + 1,...,|Writers|) do
          if (Writers[i] und Writers[j] in unterschiedlichen Zweigen einer
              Verzweigung) do                                                 ❹
            bNode := Verzweigungsknoten der gefundenen Verzweigung
            sNode := Synchronisationsknoten der gefundenen Verzweigung

            if (V_in^sNode = „n aus n“) do                                    ❺
              Succ_n1:= bestimme Nachfolger des Writers[i]
              if (Writers[j] ∉ Succ_n1) do                                    ❻
                Succ_n2 := bestimme Nachfolger des Writers[j]
                if (Writers[i] ∉ Succ_n2) do                                  ❼
                  E := E ∪ {(Writers[i], Writers[j], d, bNode, sNode)}
                end
              end
            end

          end
        end
      end

    end
  end

end

return E

end
```

Die wichtigsten Schritte des Algorithmus sind: Zuerst werden alle parallele Verzweigungen im Graphen gesucht und die Knoten der jeweiligen Verzweigung werden in der Menge PBranch gespeichert ❶. Da der Algorithmus den Test nur bei mindesten einer parallelen Verzweigung ausführt, wird die Anzahl der parallelen Verzweigungen bestimmt. Um den Test auszuführen, ist mindestens eine parallele Verzweigung im Graphen notwendig. Sind mehrere parallele Verzweigungen gefunden worden, so werden parallele Verzweigungen, die in anderen parallelen Verzweigungen enthalten sind, entfernt ❷. Dann werden die Schreibknoten für jeden einzelnen Datenslot in der jeweiligen parallelen Verzweigung bestimmt und in einem Feld gespeichert, damit über einen Index auf die Schreibknoten zugegriffen werden kann. Wenn die Anzahl der Schreibknoten in einer parallelen Verzweigung größer eins ist ❸, dann wird für jedes Schreibknotenpaar geprüft, ob die beiden Knoten sich in einer Verzweigung in unterschiedlichen Zweigen befinden ❹. Dabei werden auch andere Verzweigungsarten (parallele Verzweigung mit finaler Auswahl, bedingte Verzweigung) beachtet. Dies ist notwendig, da in einer parallelen Verzweigung noch andere Verzweigungen enthalten sein können. Nun wird festgestellt, ob die beiden Knoten in einer parallelen Verzweigung sind. Besitzt der Synchronisationsknoten der Verzweigung die „n aus n“-Eingangssemantik, so handelt es sich bei der Verzweigung um eine parallele Verzweigung ❺. Um die Synchronisationskanten zu beachten, wird

die Nachfolgermenge des Knotens Writers[i] bestimmt ❻. Ist der Knoten Writers[j] nicht in dieser Menge, so wird die Nachfolgermenge von Writers[j] berechnet ❼. Ist nun Writers[i] nicht in dieser Menge, so beschrieben die Knoten den Datenslot d parallel und ein 5-Tupel (Writers[i], Writers[j], Datenslot, Verzweigungsknoten, Synchronisationsknoten) wird in die Ergebnismenge für die spätere Fehlermeldung hinzugefügt. Am Ende des Algorithmus wird diese Ergebnismenge als Resultat zurückgegeben.

4.4.3 Die Sicherstellung der Versorgung der obligaten Eingabeparameter

Der Algorithmus zur Sicherstellung der Versorgung aller obligaten Eingabeparameter der Aktivitäten ist ein sehr wichtiger und aufwendiger Algorithmus zur Korrektheitsprüfung. Der Grund für seine Wichtigkeit ist, daß eine Aktivität innerhalb des Workflows später nicht korrekt ausgeführt werden kann, wenn nicht alle obligaten Eingabeparameter versorgt sind.

In [Hen97] wird ein Korrektheitskriterium definiert, das angibt, wann die Datenversorgung einer Aktivität sichergestellt ist:

> In einer Aktivitätenvorlage ist die Datenversorgung eines Knotens n sichergestellt, falls für jeden obligaten Lesezugriff des Knotens n ein oder mehrere obligate Schreibzugriffe auf den gleichen Datenslot existieren, von dem in jeder möglichen Ablaufreihenfolge mindestens einer zwingend vor der Ausführung des Knotens n stattfindet.

Bei der Korrektheitsprüfung für die Versorgung der Eingabeparameter sind erstens die parallelen Verzweigungen mit finaler Auswahl und die bedingten Verzweigungen zu beachten, da hier jeweils nur die Aktivitäten eines Zweiges als ausgeführt gelten und damit nur die Daten des Zweiges zur Verfügung stehen. Zweitens müssen auch die Sync-Kanten beachtet werden, da die Leseparameter eines Knotens auch über Sync-Kanten versorgt werden können. Dies geschieht durch vor den Sync-Kanten liegende Knoten eines anderen Zweiges. Dabei stellt sich für „weiche“ Sync-Kanten ein Problem, das eine gesonderte Prüfung notwendig macht. Dazu muß für jeden Knoten vor einer Sync-Kante geprüft werden, ob er Teil einer bedingten Verzweigung oder Verzweigung mit finaler Auswahl ist, wobei die Verzweigung den Zielknoten der Sync-Kante nicht enthält. Trifft diese Bedingung zu, so können diese Knoten, die oder den nur über Sync-Kanten erreichbaren Nachfolgerknoten nicht mit Daten versorgen.

Die Funktionsweise des Algorithmus wird ausführlich in [Hen97] beschrieben. Deswegen werden hier nur kurz die einzelnen Schritte des Algorithmus aufgezeigt. Für die Verwendung des Algorithmus in dieser Arbeit mußte der Algorithmus modifiziert werden. So benötigt der Überprüfungsalgorithmus in dieser Arbeit keine Analysen der Versorgung der obligaten Eingabeparameter zur Laufzeit, da zur Modellierungszeit noch nicht bestimmt werden kann, in welcher Reihenfolge die Aktivitäten des Workflows ausgeführt werden. Aus dem gleichen Grund müssen auch Nachforderungsdienste nicht betrachtet werden.

Der Algorithmus prüft für genau einen obligaten Lesezugriff eines Knotens n auf einen Datenslot d, ob dieser Datenslot d obligat versorgt wird. Grundlage dieses Verfahrens ist die Markierung einzelner Knoten mit verschiedenen Werten. Der Algorithmus gliedert sich in zwei Teile:

1. Im ersten Teil des Algorithmus wird geprüft, ob der obligate Eingabeparameter des Knotens n von Vorgängerknoten, die mit n nur über Kontrollkanten verbunden sind, versorgt werden kann. D.h. Sync-Kanten werden hier noch nicht betrachtet! Die Knoten werden in diesem Teil mit dem Wahrheitswert „Check“ und dem Zähler „Counter“ markiert. Das Attribut „Check“ eines Knotens

wird bei dem Verfahren auf Wahr gesetzt, wenn ein Knoten den Datenslot d beschreibt oder ein Knoten (unbeachtet, ob er den Datenslot d liest oder nicht) sicher mit einem Wert versorgt werden kann. Der Zahler „Counter“ ist notwendig, um festzustellen, ob bei einer späteren Ausführung der Ablaufvorlage eine bedingte oder parallele Verzweigung mit finaler Auswahl einen Wert für den Lesezugriff des Knotens n sicher liefert. Solch eine Verzweigung kann nur sicher einen Wert für den Datenslot d bereitstellen, wenn der Datenslot d innerhalb der Verzweigung in jedem alternativen Zweig obligat beschrieben wird. Wenn das Attribut „Check“ für den Knoten n auf Wahr gesetzt ist, ist die Versorgung des obligaten Eingabeparameters des Knotens n sichergestellt und der Algorithmus kann beendet werden. Ansonsten muß die Versorgung über Sync-Kanten geprüft werden.

2. Bei der Überprüfung der Datenversorgung über Sync-Kanten besitzt jeder Knoten neben den Attributen „Check“ und „Counter“ noch das Attribut „SyncCheck“. Der Wahrheitswert „SyncCheck“ gibt an, ob ein Knoten über eine Sync-Kante sicher mit einem Wert für den Datenslot d versorgt werden kann (unabhängig davon, ob der Knoten den Datenslot auch wirklich liest). Aufgrund des Attributs „SyncCheck“ wird geprüft, ob das Attribut „Check“ eines Knotens auf Wahr gesetzt werden kann. Ist nach Ablauf dieses Teils der Analyse das Attribut „Check“ des Leseknotens n auf Wahr gesetzt, so ist die Datenversorgung über Sync-Kanten sichergestellt, ansonsten wird der obligate Lesezugriff des Knotens n nicht bei jeder Ausführung des Workflows sicher mit einem Wert versorgt.

Die Komplexität des im Workflow-Editor verwendeten Algorithmus ist O(|N|), wobei N die Menge der Knoten des Kontrollflußgraphen beschreibt.

5. Der Editor

In diesem Kapitel wird der Funktionsumfang des Editors WFEdit2 vorgestellt. WFEdit2 ist ein Editor für das ADEPT-Basismodell, das an der Universität Ulm entwickelt wurde. Der Editor dient zur Spezifikation von Workflows in der Modellierungsphase [Jab95]. Die Modellierung findet dabei vor der Ausführung des Workflows (Ausführungsphase) statt. Wesentliche Anforderungen an den Editor waren benutzerfreundliche, schnelle und einfache Modellierbarkeit von Arbeitsabläufen sowie die Prüfung der Korrektheit des Kontroll- und Datenflusses. D.h. der Editor soll den Benutzer bei der Modellierung von korrekten Workflow-Schemata unterstützen, indem er mögliche Fehlerquellen schon vor der Ausführung des Workflows zu vermeiden hilft oder sie entdeckt und anzeigt. Neben der ausführlichen Erklärung dieser Modellierungsunterstützung werden deren wesentlichen Inhalte nach jedem relevanten Modellierungsschritt in einer kurzen Zusammenfassung dargestellt.

Im ersten Abschnitt wird die Oberfläche vorgestellt und die Details wie die Spezifikation des Kontroll- und Datenflusses vertiefend erläutert. Die Modellierung eines Arbeitprozesses läßt sich im ADEPT-Basismodell in Modellierung von Kontrollfluß und Datenfluß trennen. Die Kontrollflußmodellierung wird ausführlich in Abschnitt 5.2 beschrieben. Im darauffolgenden Abschnitt erfolgt die Beschreibung der Datenflußmodellierung. Die Möglichkeiten zur Modellierung der Zeit im Editor werden in Abschnitt 5.4 vorgestellt. In Abschnitt 5.5 werden die Korrektheitsprüfungen des Abschlußtests beschrieben.

5.1 Die Funktionsweise und Bedienung des Editors

Nach dem Start von WFEdit2 ist zuerst das Hauptfenster sichtbar. Abbildung 27 zeigt das Hauptfenster, das sehr ähnlich zu anderen Windows-Anwendugen ist. Der Kontrollfluß ist vom Datenfluß getrennt, um eine übersichtliche Modellierung des gesamten Workflow-Schemas zu ermöglichen.

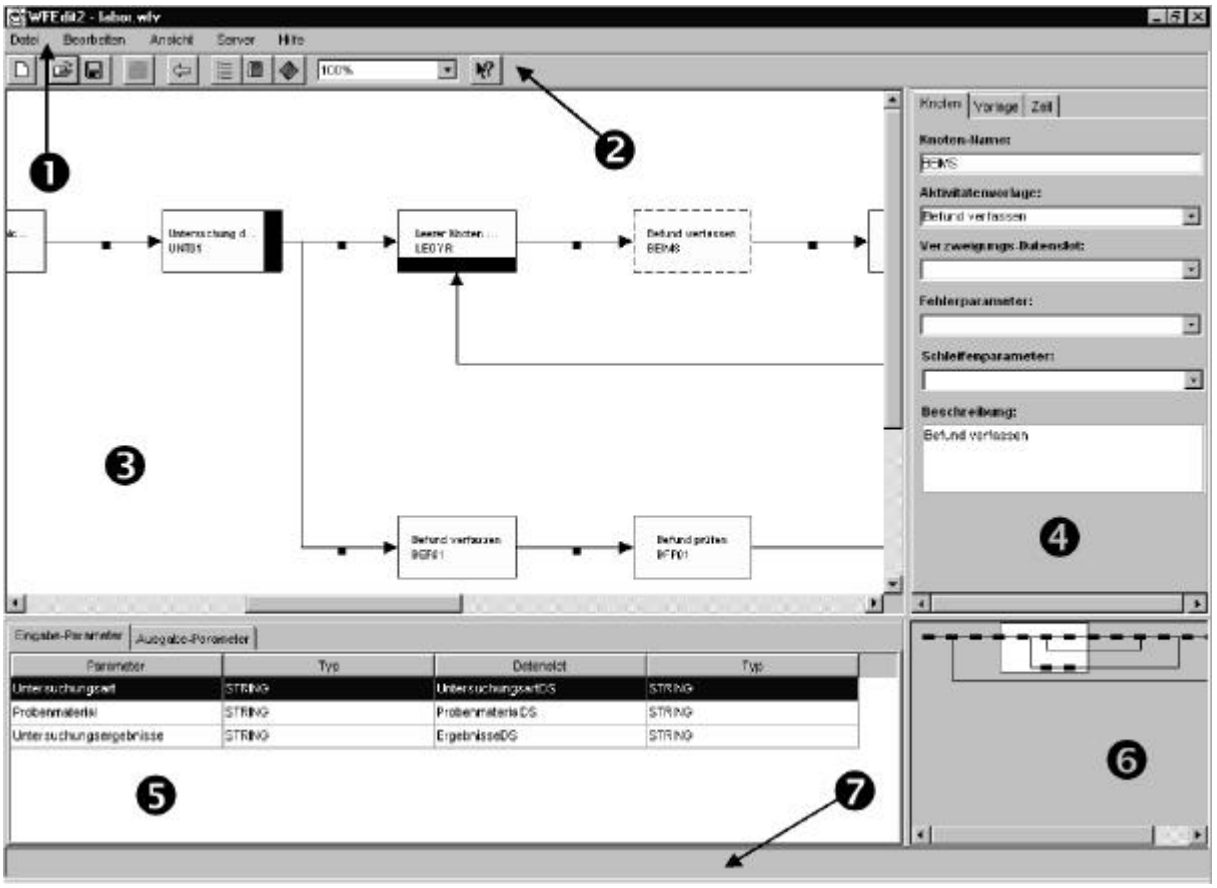


Abbildung 27: *Das Hauptfenster des Editors WFEdit2*

Das Hauptfenster besteht aus sieben Elementen: Am oberen Rand des Fensters befindet sich die Hauptmenüleiste ❶ und darunter eine Symbolleiste ❷. Die Hauptmenüleiste ermöglicht in hierarchisch dargestellter Textform den Zugriff auf die Befehle von WFEdit2. Die Symbolleiste zeigt die gebräuchlichsten Befehle als Icons an.

Unter der Symbolleiste befindet sich das große Fenster zur Darstellung des Kontrollflusses ❸. Rechts davon ist ein Register, welches auf verschiedenen Registerkarten die Eigenschaften des gewählten Knoten und dessen Aktivitätenvorlage darstellt ❹.

Unter den zwei Fenstern ist links das Fenster für den Datenfluß plaziert ❺. Das Fenster enthält die Registerkarten „Eingabeparameter“ und „Ausgabeparameter“. Hier werden die Ein- und Ausgabeparameter der gewählten Aktivität angezeigt, und diesen Parametern können Datenslots zugeordnet werden. Rechts daneben ist eine Placemarkerbox [FNP93], die das gesamte Workflow-Schema in einer Übersicht darstellt ❻ und den momentan sichtbaren Bereich markiert. Mit der Placemarkerbox können verschiedene Stellen des Prozeßgraphen mit einem Mausklick erreicht werden, d.h. mit dieser Box wird die Navigation in großen Graphen für den Benutzer stark vereinfacht.

Unter all diesen Fenstern befindet sich eine Statusleiste, die wichtige Meldungen des Programms anzeigt ❼.

Neben den sichtbaren Teilen des Hauptfensters bietet WFEdit2 noch eine andere Möglichkeit auf Befehle zuzugreifen – lokale Pop-up-Menüs, die in einzelnen Fenstern des Programms zur Verfügung

stehen. Solch ein kontextabhängiges Pop-up-Menü wird z.B. im Kontrollflußfenster zum Einfügen von neuen Aktivitäten, Schleifen, etc. genutzt.

5.1.1 Die Hauptmenü- und Symbolleiste

Die Hauptmenüleiste enthält zahlreiche Befehle, die für die Modellierung eines Prozesses notwendig sind. Alle allgemeinen Befehle, die nicht ausschließlich mit der Modellierung des Kontroll- und Datenfluß zu tun haben, werden in den nächsten Abschnitten erklärt. Die Erläuterungen der spezifischen Befehle für Kontroll- und Datenfluß befinden sich in den Abschnitten 5.2. bzw. 5.3.

Abbildung 28: *Die Menüleiste des Programms*

Die Icons in der Symbolleiste repräsentieren folgende Befehle:

- Erzeugt eine neue Prozeßvorlage
- Lädt eine Prozeßvorlage aus einer Datei (nicht in der Datenbank!)
- Speichern einer Prozeßvorlage in einer Datei
- Druckt die Prozeßvorlage (nicht implementiert)
- Rückgängig machen der letzten Änderungen
- Bearbeitet die Parameter einer bestehenden Aktivitätenvorlage
- Erzeugt eine neue Aktivitätenvorlage
- Startet den Abschlußtest für eine Prozeßvorlage
- 100% Erlaubt eine Darstellung des Kontrollflußgraphen in normaler (100%) oder halber (50%) Größe
- Ruft die Hilfe für das Programm auf

5.1.2 Das Datei-Menü

Das Datei-Menü enthält alle Befehle zur Verwaltung von Prozeßvorlagen in der Datenbank oder in einer Datei. Dies sind die Befehle zum Laden und Speichern und zum Erzeugen einer neuen Prozeßvorlage. Im durch das Dateimenü erreichbaren Eigenschaftsdialog (Abbildung 29) können der Name, die Beschreibung und die Kategorie der Vorlage editiert werden. Die Ein- und Ausgabeparameter der Prozeßvorlage mit ihren Typen und ihren Relevanz-Attributen können im selben Dialog in einer Baumübersicht betrachtet, aber nicht editiert werden.

Daneben ist noch der Menübefehl zum abschließenden Test der Korrektheit von Kontroll- und Datenfluß. Damit soll verhindert werden, daß inkonsistente Prozeßvorlagen zur Ausführung freigegeben werden[1]. Einen Korrektheitstest der Prozeßvorlage kann der Benutzer auch schon während der Modellierung des Workflow-Schemas ausführen. Dies vereinfacht die Modellierung für den Benutzer erheblich, da er mögliche Fehler schon frühzeitig erkennen kann.

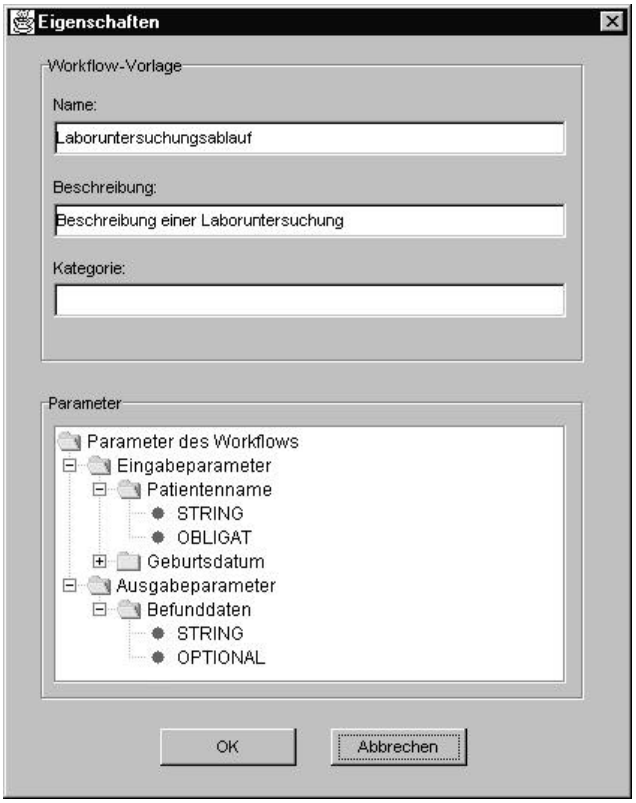


Abbildung 29: *Eingenschaftsdialog*

5.1.3 Das Bearbeiten-Menü

Das Bearbeiten-Menü enthält die Befehle zur Manipulation von Aktivitätenvorlagen und Datenslots. Außerdem befindet sich in diesem Menü auch der Rückgängig-Befehl. Er dient dazu, Veränderungen am Kontrollfluß und Datenfluß rückgängig zu machen. Der Benutzer kann seine letzten 20 Schritte rückgängig machen.

[1] Es besteht für den Benutzer kein Zwang, die Korrektheits-Checks zu aktivieren. In manchen Fällen kann es vorteilhaft sein, Workflow-Schemata ohne Korrektheitstest zu speichern. Allerdings sollten diese unvollständigen Workflow-Schemata nicht ausgeführt werden! In der Datenbank besteht die Möglichkeit die Freigabe einer Prozeßvorlage zu steuern.

Für Aktivitätenvorlage gibt es folgende Möglichkeiten zur Manipulation:

- Erzeugen einer neuen Aktivitätenvorlage
- Ändern des Namens einer Vorlage
- Manipulation der Ein- und Ausgabeparameter einer Vorlage
- Löschen einer Aktivitätenvorlage

Zur Manipulation von Datenslots stehen folgende Aktionen zur Verfügung:

- Erzeugen von neuen Datenslots
- Ändern des Namens eines Datenslots
- Löschen eines Datenslots

5.1.4 Das Ansicht-Menü

Die Befehle des Ansicht-Menüs verändern die Größendarstellung des Kontrollflußfensters und aktivieren bzw. deaktivieren die Ansicht von speziellen Kanten des ADEPT-Basismodells.

Der erste Befehl Größe erlaubt es, die Darstellung zwischen 50% und 100% zu variieren. Bei der Einstellung 50% verdoppelt sich die Darstellungsfläche des Kontrollflußfensters im Vergleich zur 100%-Darstellung. Die 50%-Darstellung bietet sich vor allem an, um einen Überblick über einen größeren Ausschnitt des Workflow-Schemas zu bekommen.

Unter dem Befehl zur Größenveränderung sind mehrere Befehle zum Ein- und Ausschalten von besonderen Kantenarten plaziert. Dies ermöglicht eine übersichtliche Darstellung des Graphen.

Folgende Kantenarten können ein- und ausgeschaltet werden:

- Fehlerkanten
- Priorisierungskanten
- Sync-Kanten, wobei zwischen „weichen“ Sync-Kanten und „harten“ Sync-Kanten unterschieden wird
- Zeitkanten

Neben der Möglichkeit die Anzeige der verschiedenen Kantenarten jeweils einzeln ein- und auszuschalten gibt es auch die Möglichkeit die Darstellung aller Kantenarten zu aktivieren und zu deaktivieren.

5.1.5 Das Server-Menü

Das Server-Menü dient, dazu eine Verbindung zu einem Workflow-Server, der Prozeßvorlagen enthält, herzustellen. Für die Verbindung muß ein Servername, der Benutzername und ein Paßwort in ein Dialogfenster (Abbildung 30) eingegeben werden. Tritt bei der Verbindung ein Fehler auf oder ist kein Server vorhanden, so wird eine Fehlermeldung ausgegeben.

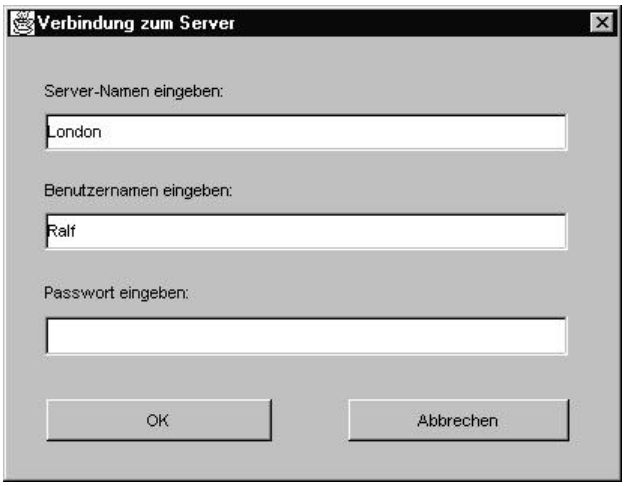


Abbildung 30: *Dialog zum Aufbau der Verbindung zum Workflow-Server*

5.2 Die Modellierung des Kontrollflusses

Die Modellierung des Kontrollflusses ist eine der Hauptaufgaben des Editors. Die Aufgabe soll vom Benutzer möglichst einfach, schnell und komfortabel ausgeführt werden können. Im ADEPT-Basismodell werden verschiedene vormodellierte Aktivitätenvorlagen für die Knoten eines Workflow-Schemas benutzt. Somit ist ein hoher Grad an Wiederverwendbarkeit der Aktivitätenvorlagen gesichert.

Der Kontrollfluß definiert die Reihenfolge der Abarbeitung der Aktivitäten bei der Ausführung des Workflows. Dies kann sequentiell oder parallel geschehen. Damit wird auch ein Teil des Verhaltens definiert. Daher wird dieser Sachverhalt in der Literatur [JBS97, Jab95] als Verhaltensaspekt beschrieben.

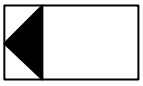
Die Modellierung des Kontrollflusses eines Workflow-Schemas erfolgt durch den syntax-gesteuerten Editor mit Hilfe einer netzbasierten Beschreibungssprache für das ADEPT-Basismodell. Komplexere Konstrukte werden in Abschnitt 3.1 beschrieben. Sie sind aus den graphischen Basiselementen zusammengesetzt, die über verschiedene Kantenarten miteinander verbunden werden.

Die Grundelemente zur Modellierung des ADEPT-Basismodells sind:

- **Aktivitäten (Knoten):**

Normaler Knoten

Verzweigungsknoten mit „1 aus n“-Semantik



Synchronisationsknoten mit „1 aus n“-Semantik

Verzweigungsknoten mit „n aus n“-Semantik

Synchronisationsknoten mit „n aus n“-Semantik

Schleifenstart- und Schleifen-endknoten

- **Kanten:**

- Kontrollkanten	Schwarz
- Fehlerkanten	Rot
- Priorisierungskanten	Gelb
- „harte“ Sync-Kanten	Magenta
- „weiche“ Sync-Kanten	Grün
- Zeitkanten	Blau

5.2.1 Das Einfügen und Löschen von Aktivitäten, Schleifen und Verzweigungen

Elementare Operationen auf dem Ablaufgraphen sind das Einfügen und Löschen von Konstrukten des ADEPT-Basismodells. Der Editor bietet bei verschiedenen Konstrukten mehrere Wege, um sie in das Workflow-Schema einzufügen. Dies ist z.B. bei Schleifen davon abhängig, ob die Schleife als ganzes eingefügt werden soll oder ob die Schleife nachträglich um einen bzw. mehrere Knoten gelegt werden soll. Im ersten Fall kommt hier ein Pop-up-Menü zum Einsatz, während im zweiten Fall zuerst Start- und Endknoten gewählt werden müssen.

Um das Pop-up-Menü zum Einfügen von Konstrukten und zum Löschen von Kanten zu aktivieren, ist ein Mausklick mit der rechten Maustaste auf den sogenannten „Hotspot“ der Kante notwendig. Der Hotspot ist ein Quadrat, mit dem sich die Kante manipulieren läßt. Abbildung 31 zeigt den „Hotspot“ einer Kante. Der „Hotspot“ befindet sich je nach Kantentyp an verschiedenen Stellen. Bei Kontroll-

und Fehlerkanten ist er in der Mitte der Kante, während er bei Zeitkanten in der Nähe des Kantenursprungs zu finden ist. Diese Anordnung des „Hotspots“ ist notwendig, um eine übersichtliche Darstellung trotz der unterschiedlichen Verläufe der Kanten zu gewährleisten.

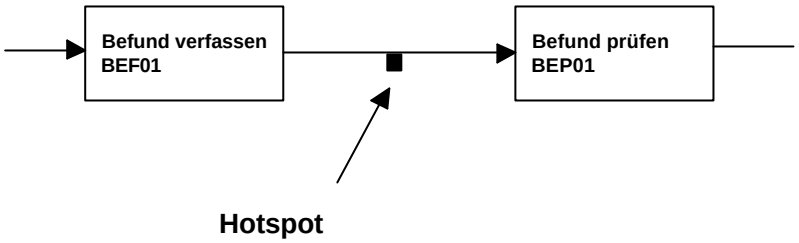


Abbildung 31: *Darstellung des „Hotspots“ einer Kante*

Nachdem das Pop-up-Menü sichtbar ist, kann nun zwischen dem Einfügen der verschiedenen Konstrukte gewählt werden. Man kann sich das Einfügen über das Pop-up-Menü so vorstellen, als ob die Kante mit dem gewählten „Hotspot“ vom ihrem Endknoten getrennt wird und dann auf den neu erzeugten Knoten gesetzt wird. Anschließend wird eine Kante von diesem neuen Knoten auf den alten Endknoten der Kante mit dem angeklickten Hotspot gesetzt. Die Abbildung 32 zeigt das Einfügen einer Schleife über einen „Hotspot“ der Kante in den Teilgraphen aus Abbildung 31.

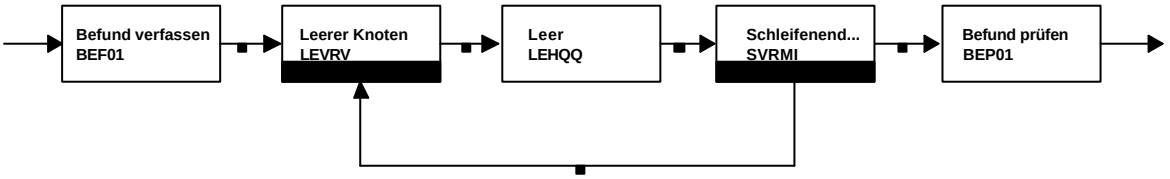


Abbildung 32: *Teilgraph mit einer Schleife anstelle des „Hotspots“*

Wird der Befehl „Aktivität einfügen“ gewählt, so öffnet sich ein Dialog mit allen Aktivitätenvorlagen. Ausgenommen sind die Vorlagen vom Typ START, ENDWF und LOOP. Die Aktivitätenvorlagen vom Typ START und ENDWF sind fest dem Start- bzw. Endknoten zugeordnet und können nicht neu erzeugt oder gelöscht werden, da sie genau einmal in einer Prozeßvorlage vorkommen müssen. Bei Aktivitätenvorlagen vom Typ LOOP verhält es sich anders. Diese Vorlagen können nur Schleifenendknoten zugeordnet werden.

☞ ***Modellierungsunterstützung 1:***

Bei der Aktivierung des Befehls „Aktivität einfügen“ können nur Vorlagen folgenden Typs eingefügt werden:

- APPLICATION
- EMPTY
- MANUAL

Soll eine Schleife eingefügt werden, so öffnet sich der Dialog aus Abbildung 33. Im oberen Listenfeld wird die Aktivitätenvorlage für den Schleifenknoten gewählt. Dieser Schleifenknoten repräsentiert den Schleifenkörper innerhalb der Schleife. Hier sind wieder alle Aktivitätenvorlagen mit allen Typen außer START, ENDWF und LOOP möglich. Darunter werden die Attribute für den Schleifenendknoten gesetzt. Im mittleren Listenfeld wird eine Vorlage für den Schleifenendknoten bestimmt. Hier sind nur Aktivitätenvorlagen zur Auswahl, die vom Typ LOOP sind. Jede Prozeßvorlage besitzt per Default die Aktivitätenvorlage „Schleifenendknoten (Default)“ vom Typ LOOP. Im unteren Listenfeld werden alle Ausgabeparameter des Schleifenendknotens angezeigt, die vom Typ INTEGER sind und obligat geschrieben werden. Obligate Ausgabeparameter müssen immer mit einem Wert, den die Aktivität generiert, versorgt sein. Aufgrund des Wertes des gewählten Ausgabeparameters wird bei der Ausführung einer Workflow-Instanz entschieden, ob die Schleife verlassen werden soll oder ob eine weitere Iteration der Schleife gestartet werden soll.

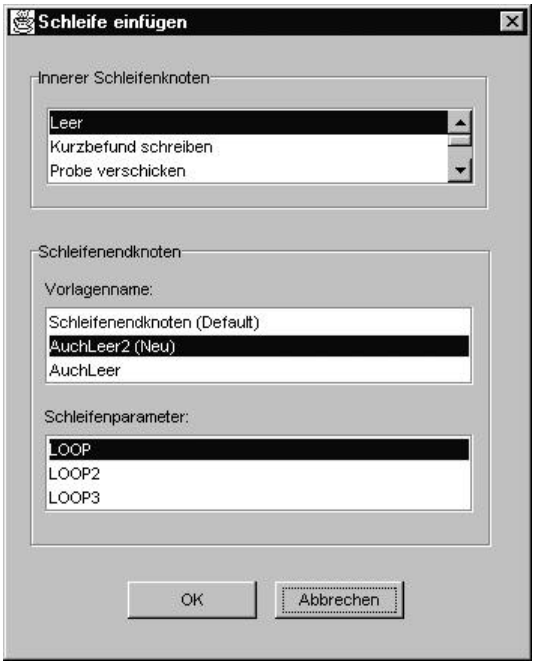


Abbildung 33: *Dialog zum Einfügen einer Schleife*

☞ ***Modellierungsunterstützung 2:***

Beim Einfügen einer Schleife müssen folgende Attribute der Schleife angegeben werden:

1. Der innere Schleifenknoten (Schleifenkörper) muß spezifiziert werden. Hier sind nur Vorlagen folgender Typen erlaubt:

   - APPLICATION
   - EMPTY
   - MANUAL

2. Es kann nur eine Aktivitätenvorlage des Typs LOOP für den Schleifenendknoten gewählt werden.

3. Es kann nur ein obligater Integer-Ausgabeparameter des Schleifenendknotens als Schleifenparameter gewählt werden.

Bei Auswahl von „Verzweigung einfügen“ muß eine der drei Verzweigungsarten des ADEPT-Basismodells (bedingte Verzweigung, parallele Verzweigung, parallele Verzweigung mit finaler Auswahl) gewählt werden. Der selektierte Verzweigungstyp bestimmt das Aussehen des Verzweigungsdialogs aus Abbildung 34. Während bei der bedingten Verzweigung alle Eingabefelder aktiviert sind, ist bei der parallelen Verzweigung und der parallelen Verzweigung mit finaler Auswahl das Listenfeld „Auswahl-Datenslot“ deaktiviert, da hier keine Entscheidung für einen Zweig aufgrund eines Datenslots getroffen wird. Bei der bedingten Verzweigung werden alle obligaten Datenslots mit dem Typ INTEGER angezeigt. Bei Datenslots, die vorher von einem anderen Knoten oder dem Verzweigungsknoten mit einem Wert obligat beschrieben werden, wird das Attribut „[obligat]“angezeigt. Bei diesen Datenslots ist die Versorgung mit einem Wert sichergestellt und bei der Ausführung des Workflows kann aufgrund dieses Wertes sicher eine Entscheidung für einen Zweig der bedingten Verzweigung getroffen werden.

Im Gegensatz zur bedingten Verzweigung werden bei den parallelen Verzweigungen alle vorhandenen Zweige mit ihren Aktivitäten gestartet. Dies führt dazu, daß die Zweige parallel abgearbeitet werden. Im obersten Listenfeld wird die Aktivitätenvorlage für den Startknoten der Verzweigung ausgesucht. Dabei werden wieder aus obengenannten Gründen nur Vorlagen vom Typ EMPTY, MANUAL und APPLICATION angezeigt. Für alle anderen Knoten der Verzweigung wird automatisch die Aktivität „Leerer Knoten (Default)“ vom Typ EMPTY eingefügt. Dieser Knoten ist in allen Prozeßvorlage per Default vorhanden. In dem Eingabefeld darunter kann bestimmt werden, wieviel Zweige die Verzweigung besitzen soll. Es können minimal zwei und maximal 20 Zweige für die Verzweigung generiert werden.

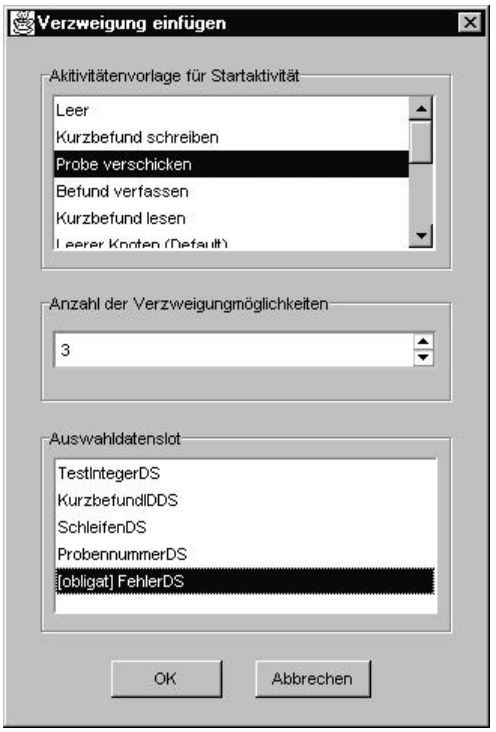


Abbildung 34: *Dialog zum Einfügen einer Verzweigung*

☞ ***Modellierungsunterstützung 3:***

Beim Einfügen einer Verzweigung wird folgendes Korrektheitskriterium immer erzwungen:

1. Die Vorlage des Verzweigungsknotens kann nur folgenden Typs sein:

   - APPLICATION
   - EMPTY
   - MANUAL

Zusätzlich gilt noch für eine bedingte Verzweigung:

2. Ein Integer-Datenslot muß als Lieferant für den Entscheidungswert spezifiziert werden. Wird der Datenslot von einem dem Verzweigungsknoten vorhergehenden Knoten obligat beschrieben, so wird er mit „[obligat]“ markiert.

Beim Einfügen einer bedingten Verzweigung bleiben die Entscheidungswerte der Kanten, die den Verzweigungsstartknoten mit den ersten Knoten der alternativen Zweige verbinden, undefiniert. Diese müssen über den Eigenschaftsdialog für Kanten vom Benutzer gesetzt werden. Dieser Eigenschaftsdialog befindet sich am unteren Ende des vorher beschriebenen Pop-up-Menüs. Falls die Werte während der Modellierung nicht spezifiziert wurden, so erkennt die Abschlußprüfung deren fehlen und gibt eine Meldung aus, welcher Kante nicht mit einem Entscheidungswert ausgestattet wurde.

Um bei einer bestehenden Verzweigung einen weiteren Zweig hinzuzufügen, ist ein Mausklick mit der rechten Maustaste auf den Verzweigungsknoten notwendig. Daraufhin wird ein Pop-up-Menü

angezeigt. Bei der Auswahl von „Verzweigungsast hinzufügen“ wird dann automatisch ein der Verzweigungsart entsprechender Zweig mit einem Knoten (Vorlage: „Leere Knoten (Default)“) angelegt. Ist die Verzweigung eine bedingte Verzweigung, so ist der Entscheidungsparameter der neuen Kante zum ersten Knoten des neuen Zweigs undefiniert.

Das Löschen von Aktivitäten bzw. Konstrukten des ADEPT-Basismodells erfolgt auch über das im letzten Abschnitt erwähnte Pop-up-Menü, das mit einem Klick der rechten Maustaste auf einen beliebigen Knoten aktiviert wird. Ausnahmen hiervon sind nur der Start- und Endknoten der Prozeßvorlage, die nicht gelöscht werden können. Beim Löschen von Schleifen und Verzweigungen gibt es je nach Konstrukt unterschiedliche Verhaltensweisen.

Eine Aktivität wird gelöscht, indem im Pop-up-Menü der Befehl „Aktivität löschen“ gewählt wird. Es entsteht dabei keine Lücke im Kontrollfluß, da die gelöschte Aktivität durch eine Kante vom Vorgängerknoten zum Nachfolgerknoten der entfernten Aktivität ersetzt wird.

Beim Löschen eines Schleifenstartknotens bzw. Schleifenendknotens über das Pop-up-Menü und „Aktivität löschen“ wird die gesamte Schleife mit Schleifenstartknoten, Schleifenkörper und Schleifenendknoten gelöscht. D.h. es gehen alle Aktivitäten zwischen den beiden Schleifenknoten verloren. Um nur die Schleife ohne Schleifenkörper zu löschen, gibt es eine andere Vorgehensweise. Um den Schleifenkörper zu erhalten muß der „Hotspot“ der Schleifenkante, die vom Schleifenendknoten zum Schleifenstartknoten zurück verläuft, mit der rechten Maustaste angeklickt werden. Daraufhin erscheint das Pop-up-Menü, das auch zum Einfügen der Konstrukte benutzt wird, und mit „Kante löschen“ kann die Schleifenkante dann entfernt werden. Dabei werden nur der Schleifenstartknoten und der Schleifenendknoten gelöscht!

☞ ***Modellierungsunterstützung 4:***

Folgende Unterstützungen werden dem Benutzer beim Löschen von Schleifen und Verzweigungen gewährt:

- Beim Löschen eines Schleifenstartknotens bzw. Schleifenendknotens wird die gesamte Verzweigung gelöscht.
- Wird bei einer Schleife die Schleifenkante gelöscht, so wird der Schleifenstart- und der Schleifenendknoten entfernt. Der ehemalige Schleifenkörper bleibt aber erhalten.
- Bei Löschen eines Verzweigungs- bzw. Synchronisationsknotens wird die gesamte Verzweigung entfernt
- Besteht eine Verzweigung nur noch aus zwei Ästen und wird einer davon entfernt, wird die Ausgangsemantik des Verzweigungsknotens bzw. die Eingangssemantik des Synchronisationsknotens zurückgesetzt.

Das Löschen eines Verzweigungs- bzw. Synchronisationsknoten einer Verzweigung verhält sich ähnlich wie das Löschen von Schleifen. Wird ein Verzweigungs- oder ein Synchronisationsknoten gelöscht, so verschwindet die gesamte Verzweigung. Einzelne Zweige können nur gelöscht werden, indem alle Aktivitäten des Zweiges einzeln gelöscht werden. Besteht der Zweig dann nur noch aus einer einzigen Aktivität und wird diese gelöscht, so verschwindet damit der ganze Zweig. Wenn nun die gesamte Verzweigung nur noch aus zwei Ästen besteht und ein Ast wird wie oben beschrieben gelöscht, so verschwindet beim Verzweigungs- und Synchronisationsknoten die Verzweigungs- und Synchronisationssemantik. D.h. der frühere Verzweigungsknoten besitzt dann die „1 aus 1“-Ausgangssemantik und der frühere Synchronisationsknoten besitzt dann die „1 aus 1“-Eingangssemantik. Dies ist ebenfalls an der graphischen Darstellung der Knoten im Editor zu sehen.

Neben dem Einfügen von ganzen Konstrukten ist es auch notwendig, verschiedene Konstrukte des ADEPT-Basismodells noch nachträglich in den Graphen einzufügen. Diese Konstrukte, sind Schleifen und Verzweigungen. Unter dem nachträglichen Einfügen von Schleifen und Verzweigungen versteht man, daß um schon bestehende Knoten ein Schleife oder eine Verzweigung konstruiert wird. Dazu müssen zwei Knoten mit jeweils einem Mausklick bei gedrückter Umschalt-Taste markiert werden. Nach dem Markieren öffnet sich dann der in Abbildung 35 gezeigte Dialog.

Abbildung 35: *Dialog zum Einfügen verschiedener Konstrukte*

Im Dialog „Konstrukt einfügen“ kann dann eine Schleife oder Verzweigung gewählt werden. Wie die anderen Auswahlmöglichkeiten schon zeigen, verläuft das Einfügen von Kanten bis hierhin ähnlich.

Wird das Einfügen einer Schleife gewählt, so wird der Dialog aus Abbildung 36 sichtbar. Die vorher markierten Knoten sind dabei der Anfangsknoten und der Endknoten des inneren Schleifenblocks. Im obersten Listenfeld des Dialogs wird die Vorlage des Schleifenendknotens gewählt. Dies können nur Aktivitätenvorlagen vom Typ LOOP sein. Darunter wird aus den obligaten Integer-Parametern der selektierten Aktivität ein Ausgabeparameter ausgewählt, der den Wert enthalten soll, aufgrund dem entschieden wird, ob die Schleife beendet oder fortgeführt werden soll. Als Schleifenstartknoten wird automatisch ein leerer Knoten mit der Vorlage „Leerer Knoten (Default)“ eingefügt. Beim Einfügen von Schleifen kann es zu verschiedenen Fehlersituationen kommen. So darf sich nie eine Schleife mit einer anderen Schleife oder mit einer Verzweigung überschneiden. Außerdem gibt es verschiedene Einschränkungen beim Einfügen in bezug auf Fehlerkanten. Diese Einschränkungen werden im Abschnitt 5.2.2 beschrieben. Solche Situationen werden vom Editor entdeckt und dem Benutzer wird eine Fehlermeldung angezeigt.

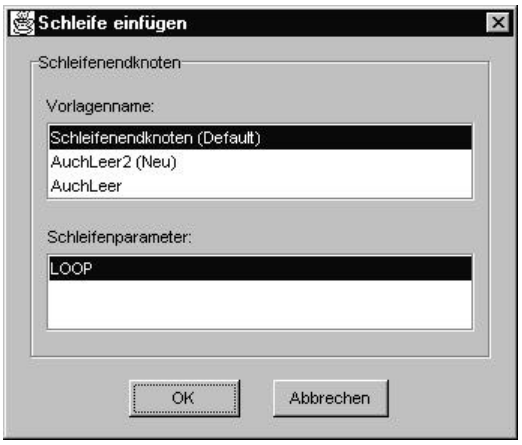


Abbildung 36: *Dialog zum nachträglichen Einfügen einer Schleife*

Soll eine Verzweigung eingefügt werden, so muß zuerst bestimmt werden, welche Art von Verzweigung vom Benutzer gewünscht wird (Abbildung 37). Die zuvor markierten Knoten sind der Verzweigungs- bzw. Synchronisationsknoten der späteren Schleife. Daher muß zwischen dem Schleifenstart- und dem Schleifenendknoten noch mindestens eine weiterer Knoten liegen. Außerdem muß noch der Entscheidungs-Datenslot bestimmt werden. Das Listenfeld „Auswahl-Datenslot“ enthält die gleichen Attribute wie beim Einfügen einer kompletten Schleife. Beim Einfügen darf es keine Überschneidungen mit anderen Konstrukten geben. Verzweigungen und schon in der Prozeßvorlage bestehende Fehlerkanten unterliegen auch bestimmten Einschränkungen. Sie werden wie bei Schleifen vom Editor erkannt und dem Benutzer gemeldet. Welche Einschränkungen existieren, wird in Abschnitt 5.2.2 beschrieben.

Abbildung 37: *Dialog zum nachträglichen Einfügen einer Verzweigung*

☞ ***Modellierungsunterstützung 5:***

Beim nachträglichen Einfügen bietet der Editor folgende Unterstützung an:

- Schleifen und Verzweigungen dürfen sich nicht mit anderen Schleifen bzw. Verzweigungen überschneiden. Ein solcher Konflikt wird als Fehler erkannt und angezeigt.
- Konflikte mit Fehlerkanten werden erkannt und angezeigt.
- Beim Einfügen einer Schleife muß ein Schleifenendknoten vom Typ LOOP gewählt werden und ein obligater Integer-Ausgabeparameter des Schleifenendknotens muß als Schleifenparameter bestimmt werden.
- Beim Einfügen einer bedingten Verzweigung muß ein Integer-Datenslot spezifiziert werden. Wird der Datenslot vorher obligat beschrieben, so wird er mit „[obligat]“ markiert.

5.2.2 Das Einfügen, Löschen und Markieren von Kanten

Das Einfügen von Kanten erfolgt, wie schon beim nachträglichen Einfügen von komplexeren Konstrukten wie Schleifen und Verzweigungen, über das Markieren von Knoten mit Umschalt-Taste + Mausklick. Danach wird der Dialog aus Abbildung 35 gezeigt, mit dem man die Kantenart zwischen den zwei Knoten auswählen kann. Kontrollkanten können nicht auf diese Weise eingefügt oder gelöscht werden, da sie automatisch beim Einfügen oder Löschen von Aktivitäten oder komplexen Konstrukten des ADEPT-Basismodells generiert oder entfernt werden.

Um den Verlauf der verschiedenen Kantenarten zu visualisieren, können diese Kanten markiert werden. Dies funktioniert mit allen Kantenarten außer Kontrollkanten und Schleifenkanten. Diese hervorgehobene Anzeige von Kanten dient dazu, die Darstellung übersichtlicher zu machen, da sich Kanten ganz oder teilweise überlagern können. Überlagern sich zwei oder mehrere Kanten, so wird ein Dialog der alle in Frage kommenden Kanten anzeigt, geöffnet.

Fehlerkanten können nur zu Knoten gezogen werden, die im Kontrollfluß vor dem Ursprungsknoten der Fehlerkante liegen, d.h. im Fehlerfall wird die gesamte Ausführung des Workflows bis zum Endknoten der Fehlerkante zurückgesetzt. Damit eine Fehlerkante aktiviert werden kann, muß sie einen zwischen allen Fehlerkanten eines Knotens eindeutigen Fehlerwert besitzen. Der Fehlerwert wird vom Knoten geschrieben. Damit die Auswahl durchgeführt werden kann erscheint ein Dialog, der in einer Auswahlbox alle obligaten Integer-Ausgabeparameter anzeigt. Daraufhin muß ein Ausgabeparameter aus dieser Box gewählt werden und ein Fehlerwert eingegeben werden. Existiert schon eine den Knoten verlassende Fehlerkante, so kann der Fehlerwert nicht neu gewählt werden. Jeder Knoten kann in bezug auf Fehlerkanten nur einen Fehlerwert besitzen.

Folgende Einschränkungen sind bei Fehlerkanten noch zu beachten:

- Eine Fehlerkante darf niemals zwischen dem Start- (inklusiv) und dem Zielknoten (exklusiv) der Fehlerkante einen offenen Synchronisationsknoten mit „1 aus n“-Semantik haben Vereinfacht ausgedrückt heißt das, daß eine Fehlerkante nie innerhalb einer bedingten oder parallelen Verzweigung enden darf. Würde diese Bedingung verletzt werden, so könnte der Workflow bei der Abarbeitung in einen nicht existierenden Prozeßzustand zurückgesetzt werden.

- Außerdem dürfen keine Fehlerkanten von außerhalb der Schleife in die Schleife gezogen werden. Wäre dies möglich, so könnte bei der Ausführung des Prozesses nicht bestimmt werden, in welche Schleifeniteration der Prozeß zurückgesetzt werden muß. Im Gegensatz dazu sind Fehlerkanten von innerhalb einer Schleife nach außen erlaubt, da hier bei der Ausführung der Fehlerkante alle Iterationen der Schleife zurückgesetzt werden können.

Diese Regeln müssen im Falle der Existenz von Fehlerkanten auch beim nachträglichen Einfügen von Verzweigungen und Schleifen beachtet werden, da die Verzweigungen und Schleifen in Konflikt mit einer bestehenden Fehlerkante geraten können.

Priorisierungskanten sind einfacher zu handhaben. Sie dürfen nur zwischen zwei Knoten einer parallelen Verzweigung eingefügt werden. Für „harte“ und „weiche“ Sync-Kanten gilt diese Bedingung ebenfalls. Die Bedingung wird beim Einfügen der Kanten geprüft und wird sie verletzt erscheint eine Fehlermeldung. Beim Einfügen von „harten“ und „weichen“ Sync-Kanten wird außerdem geprüft, ob durch sie ein Zyklus im Kontrollflußgraph entsteht könnte. Falls ein Zyklus entsteht, kann die Sync-Kante nicht eingefügt werden und dem Benutzer ein Fehler gemeldet.

Das Löschen der verschiedenen Kantenarten (außer Kontrollkanten) geschieht wie das schon vorher beschriebene Löschen von Schleifenkanten. Mit der rechten Maustaste wird eine Kante durch einen Mausklick auf den „Hotspot“ der Kante selektiert und anschließend mit „Kante löschen“ entfernt.
Liegen mehrere Kanten übereinander, so wird ein Auswahldialog ähnlich dem bei Markieren der Kanten angezeigt. Daraufhin können eine oder auch mehrere Kanten zum Löschen ausgewählt werden.

☞ ***Modellierungsunterstützung 6:***

1. Fehlerkanten:
   - Können nur zu Vorgängerknoten gezogen werden.
   - Konflikte mit Schleifen bzw. Verzweigungen werden erkannt und angezeigt.
   - Fehlerparameter können nur obligate Integer-Ausgabeparameter einer Aktivität sein.
2. Priorisierungskanten:
   - Können nur in parallele Verzweigungen eingefügt werden.
3. Sync-Kanten:
   - Können nur in parallele Verzweigungen eingefügt werden.
   - Erkennung von unerwünschten Zyklen im Graphen.

5.2.3 Die Verwaltung von Aktivitätenvorlagen und Knoten

Hinter jedem Knoten im Prozeßgraph verbirgt sich eine vormodellierte Aktivitätenvorlage. Diese Vorlagen können mehrmals benutzt werden. Knoten und Aktivitätenvorlagen besitzen verschiedene Attribute.

Die Attribute für Knoten sind:

- Eindeutiger Knotenname
- Aktivitätenvorlage
- Verzweigungs-Datenslot
- Fehlerparameter
- Schleifenparameter
- Beschreibung des Knotens

Für Aktivitätenvorlage gibt es folgende Attribute:

- Beschreibung
- Bearbeiter-Formel
- Application-Service
- Unterbrechungsmodus
- Aktivitätenvorlagentyp
- Zeitinformationen für Aktivitäten

Im Menü Bearbeiten | Aktivitätenvorlagen befinden sich mehrere Befehle zum Manipulieren von Aktivitätenvorlagen. Dort können Vorlagen erzeugt, der Name und die Parameter der Vorlage verändert werden und dort können die Vorlagen auch gelöscht werden.

Soll eine Vorlage neu erzeugt werden, so öffnet sich der Dialog aus Abbildung 38. In diesem Dialog ist es möglich eine Vorlage komplett neu zu erzeugen oder eine schon existierende Vorlage als Kopie zu verändern. Der Name einer neuen Aktivitätenvorlage darf nicht in der Prozeßvorlage existieren, d.h. es wird keine Vorlage überschrieben. Außerdem ist zu beachten, daß Aktivitäten vom Typ EMPTY bestimmte Attribute wie z.B. die Parameter und die Zeitwerte nicht besitzen können, da sie bei der Ausführung des Workflows gestartet und sofort wieder beendet werden. Schleifenendknoten vom Typ LOOP müssen mindestens einen obligaten Integer-Ausgabeparameter besitzen, damit bei der Ausführung bestimmt werden kann, ob die Schleife fortgesetzt oder beendet werden soll. All diese Bedingungen werden vom Editor geprüft und es wird gegebenenfalls eine Fehlermeldung ausgegeben.

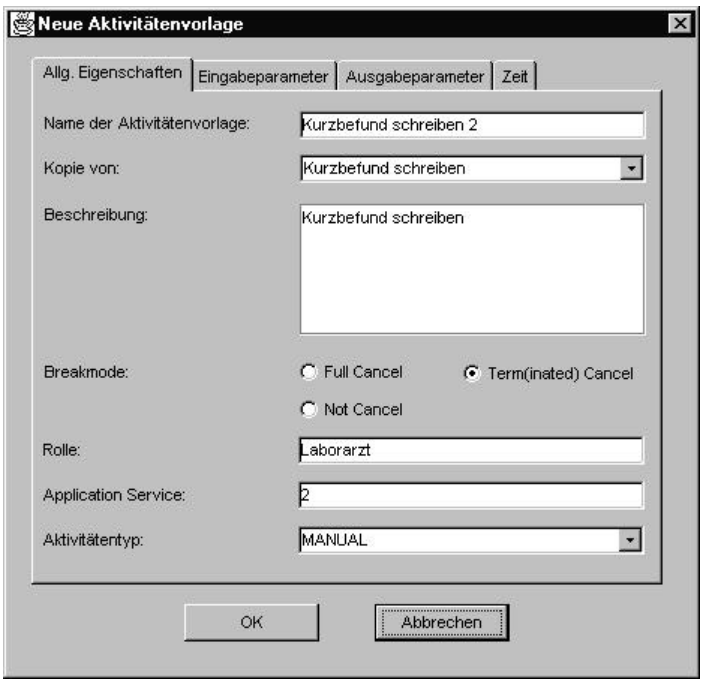


Abbildung 38: *Dialog zum Erstellen einer neuen Aktivitätenvorlage*

Zum Festlegen der Ein- und Ausgabeparameter werden zwei weitere Registerkarten verwendet. Diese Registerkarten werden in Abbildung 39 dargestellt. Noch während der Spezifikation der Aktivitätenvorlage können hier Ein- und Ausgabeparameter erzeugt, verändert und gelöscht werden. Die Bedeutung der einzelnen Attribute für einen Parameter werden in Abschnitt 3.2 erläutert.

Natürlich ist es auch möglich, den Namen und die Attribute der in der Prozeßvorlage vorhandenen Aktivitätenvorlagen zu ändern. Der Name läßt sich über Bearbeiten | Aktivitätenvorlage | Ändern | Name neu setzen. Um Attribute einer Attributvorlage zu verändern, wird ein Dialog über Bearbeiten | Aktivitätenvorlage | Ändern | Parameter aufgerufen. Dieser Dialog ähnelt stark dem Dialog aus Abbildung 38. Der Unterschied ist, daß zuerst eine Aktivitätenvorlage zur Bearbeitung gewählt werden muß. Natürlich können Ein- und Ausgabeparameter von Aktivitätenvorlagen, die in der Prozeßvorlage benutzt werden, nicht einfach gelöscht werden! Außerdem sind noch einige Einschränkungen in bezug auf Parameter der Aktivitätenvorlage definiert worden. All diese Einschränkungen werden bei der Modellierung vom Editor geprüft.

Folgende Einschränkungen sind vorhanden:

- Der Typ des Parameters kann nicht verändert werden, wenn er mit einem Datenslot verbunden ist.
- Eine Aktivitätenvorlage vom Typ LOOP benötigt mindestens einen obligaten Integer-Ausgabeparameter.
- Obligate Integer-Ausgabeparameter, die als Fehlerparameter, Schleifenparameter oder Verzweigungsparameter benutzt werden, können nicht verändert werden oder gelöscht werden. Eine Ausnahme bei der Veränderung der Parameter ist nur die Nachforderbarkeit.

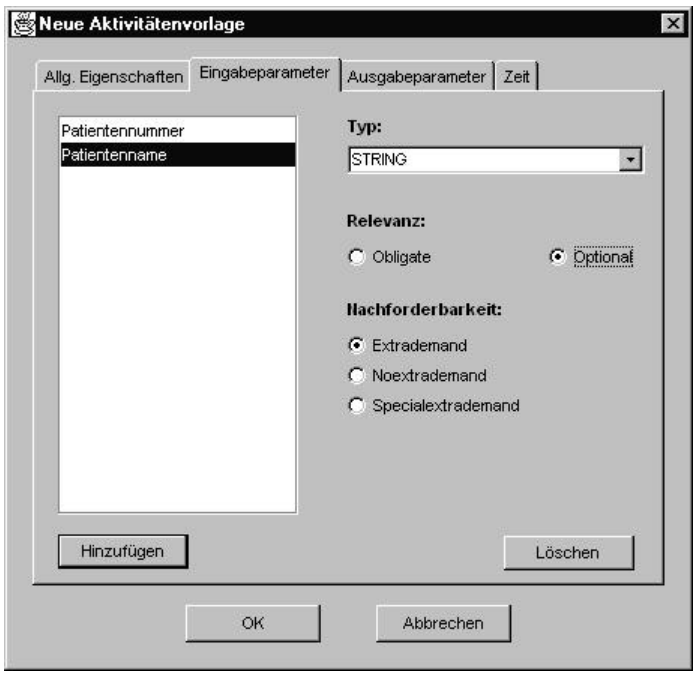


Abbildung 39: *Registerkarte zum Einfügen eines neuen Parameters*

☞ ***Modellierungsunterstützung 7:***

Bei der Modellierung von Aktivitätenvorlagen werden folgende Hilfen angeboten:

- Bestehende Aktivitätenvorlagen können kopiert und dann verändert werden.
- Vorlagen vom Typ EMPTY können keine Parameter und Zeitinformationen besitzen.
- Vorlagen vom Typ LOOP müssen mindestens einen obligaten Integer-Ausgabeparameter besitzen.
- Ein- und Ausgabeparameter, die mit Datenslots verbunden sind, können nicht gelöscht werden.
- Obligate Integer-Ausgabeparameter, die als Fehler- bzw. Schleifenparameter von einem Knoten benutzt werden, können nicht gelöscht oder verändert werden (Ausnahme: Nachforderbarkeit).

Einzelne Attribute der Knoten und der Aktivitätenvorlage werden über das Register aus Abbildung 40 geändert. Das Register befindet sich links neben dem Fenster, daß den Kontrollfluß darstellt. Sofort sichtbar ist immer die aktuelle Aktivitätenvorlage des Knotens. Hier kann auch die Aktivitätenvorlage, auf der der Knoten basiert, geändert werden. Zur Auswahl angezeigt werden nur die Aktivitätenvorlagen, die für den Knoten in Frage kommen. Z.B. werden für einen Schleifenendknoten nur Aktivitätenvorlagen vom Typ LOOP zur Auswahl gestellt. Alle Attribute auf der Registerkarte „Vorlage“ beziehen sich auf diese Aktivitätenvorlage.

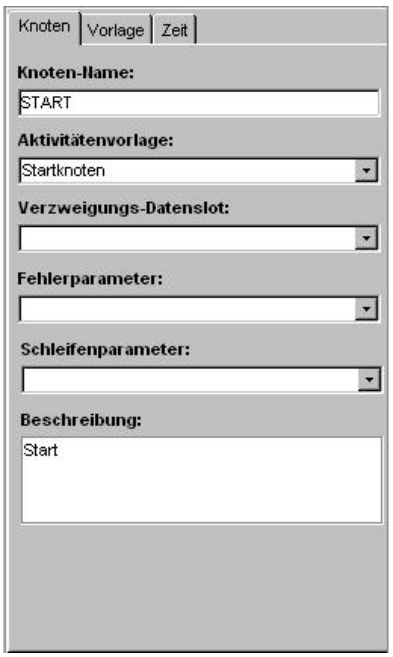


Abbildung 40: *Anzeige der Eigenschaften eines Knotens*

Desweiteren können noch andere Attribute des Knotens geändert werden. Dies sind der Verzweigungs-Datenslot bei einem Verzweigungsknoten einer bedingten Verzweigung, der Fehlerparameter eines Knotens, aus dem Fehlerkanten hinauslaufen und der Schleifenparameter eines Schleifenendknotens. Alle aktuellen Werte der Attribute sind sofern vorhanden bei Markierung des Knotens sofort sichtbar. Für den Verzweigungs-Datenslot werden nur Integer-Datenslots angezeigt. Wird eine Datenslot von einem Vorgängerknoten des Verzweigungsknoten oder dem Verzweigungsknoten selbst sicher beschrieben, so wird „[obligat]“ vor dem Namen des Datenslots angegeben. Für den Fehlerwert des Knoten werden bei ausgehenden Fehlerkanten alle obligaten Integer-Ausgabeparameter angezeigt. Des gleiche gilt für den Schleifenparameter bei Schleifenendknoten.

Auf der nächsten Registerkarte befinden sich alle Eigenschaften der Aktivitätenvorlage. Dies sind die Beschreibung der Vorlage, die Bearbeiter-Formel, die dazu verwendet wird einen Akteur für die Aktivität zu bestimmen, und der Unterbrechungsmodus. Daneben wird noch der Aktivitätentyp, der hier nicht editierbar ist, angezeigt. Zu beachten ist, daß Änderungen auf der Registerkarte sich auf die Aktivitätenvorlage auswirken! D.h. ändert man an einem Knoten z.B. den Unterbrechungsmodus für die Vorlage, so besitzen alle Aktivitäten, die auf der Aktivitätenvorlage basieren, den gleichen Unterbrechungsmodus.

Änderungen der Zeitattribute auf der nächsten Registerkarte werden in Abschnitt 5.4. dargestellt

☞ ***Modellierungsunterstützung 8:***

Um Fehler zu vermeiden werden auf der Registerkarte für einen Knoten bzw. für eine Aktivität folgende Einschränkungen gemacht:

- Es werden nur Vorlagen angezeigt, die für den markierten Knoten den passenden Typ besitzen.
- Bei einem Verzweigungsknoten werden im Eingabefeld „Verzweigungs-Datenslot“ alle Integer-Datenslots angezeigt und ggf. als „[obligat]“ beschrieben markiert.
- Im Eingabefeld „Fehlerparameter“ bzw. „Schleifenparameter“ werden alle obligaten Integer-Ausgabeparameter der gewählten Aktivität angezeigt.

5.3 Die Modellierung des Datenflusses

Die Modellierung des Datenflusses besteht im wesentlichen aus der Verbindung der Ein- und Ausgabeparameter der Aktivitätenvorlagen mit den Datenslots. Diese Datenslots müssen wie die Aktivitätenvorlagen verwaltet werden. Komplexere Verbindungen zwischen Datenslots und Parametern werden in der Arbeit [Bla96] beschrieben.

Datenslots können über das Menü Bearbeiten | Datenslot manipuliert werden. Die Möglichkeiten zur Manipulation sind erzeugen und löschen von Datenslots und das Ändern des Namens eines Datenslots. Beim Erzeugen eines Datenslot muß ein eindeutiger Name und der Typ des Datenslots gewählt werden. Zur Auswahl stehen die drei Typen STRING, INTEGER, REFERENCE. Diese Typen entsprechen genau den möglichen Typen der Ein- und Ausgabeparameter.

Zum Löschen eines Datenslots muß nur der Name angegeben werden. Wird ein Datenslot von einem Parameter einer Aktivität beschrieben oder wird aus ihm gelesen, so kann der Datenslot nicht gelöscht werden. Dies wird automatisch vom Editor verhindert und eine Fehlermeldung wird angezeigt.

Das Ändern des Namens eines Datenslots ist unproblematisch, da nur ein unter allen Datenslots der Prozeßvorlage eindeutiger Name angegeben werden muß.

Die eigentliche Modellierung, d.h. die Verbindung zwischen den Parametern der Aktivitäten und den Datenslots, geschieht über die in Abbildung 41 dargestellte Tabelle. Die Tabelle ist für Ein- und Ausgabeparameter einer Aktivität nahezu identisch. Die Ein- und Ausgabeparameter für eine Aktivität sind auf zwei verschiedenen Registerkarten getrennt aufgelistet.

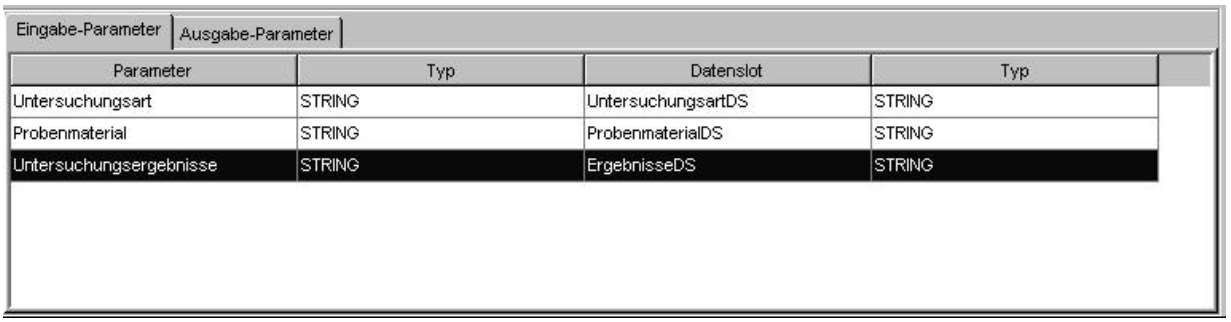


Abbildung 41: *Fenster zur Manipulation des Datenflusses*

In dieser Tabelle werden in der ersten Spalte die Ein- bzw. Ausgabeparameter der selektierten Aktivität angezeigt. In der zweiten Spalte befindet sich der Typ des Parameters. In der dritten Spalte ist der dem Parameter zugeordnete Datenslot und dessen Typ in der vierten Spalte zu sehen. Um Parameter mit Datenslots zu verknüpfen, müssen Parameter und Datenslot vom gleichen Typ sei.

Um die Verknüpfung durchzuführen, ist ein Doppelklick mit der linken Maustaste auf das Datenslot-Feld des Parameters, der verbunden werden soll, notwendig. Daraufhin öffnet sich der in Abbildung 42 dargestellte Dialog. Die Datenslot-Felder der einzelnen Parameter befinden sich in der dritten Spalte von links in der Registerkarte „Eingabeparameter“ bzw. „Ausgabeparameter“ (siehe Abbildung 41).

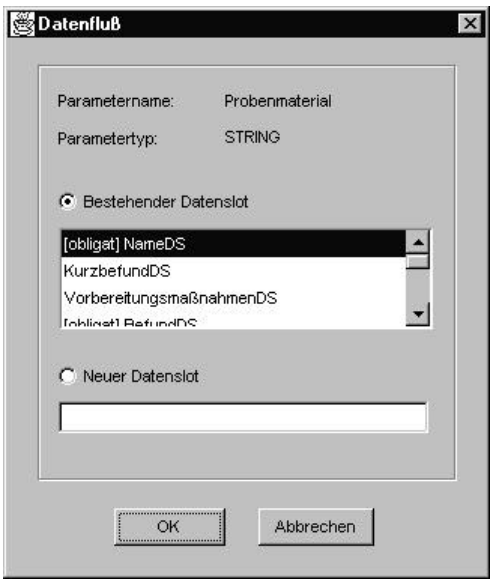


Abbildung 42: *Dialog zum Erstellen einer Verbindung zwischen Parameter und Datenslot*

In dem Dialog werden alle Datenslots vom gleichen Typ wie der Parameter angezeigt. Es kann ein bestehender Datenslot ausgewählt werden oder ein neuer Datenslot erzeugt werden. Ein Unterschied zwischen dem Dialog bei Eingabeparametern zu dem Dialog bei Ausgabeparameter besteht. Bei Eingabeparametern wird angezeigt, welcher Datenslot von Vorgängerknoten des gewählten Knoten

obligat beschreiben wird. Wird der Datenslot neu erzeugt, so besitzt er den gleichen Typ wie der Parameter, der dem Datenslot zugeordnet werden soll. Die Verbindung zwischen einem Datenslot und einem Parameter kann auch ganz aufgehoben werden, indem im Listenfeld „Bestehender Datenslot“ der Eintrag „Keine Zuordnung“ gewählt wird. Der Parameter ist dann keinem Datenslot zugeordnet.

Mit einem Doppelklick mit der rechten Maustaste auf einen Datenslot eines Parameters wird angezeigt, welche Knoten im Prozessgraphen den Datenslot lesen oder schreiben. Die Farben für die Knoten sind: Grün für lesende Knoten, Blau für schreibende Knoten und die Farbe Magenta für Knoten, die den Datenslot lesen und wieder beschreiben. Durch diese Markierungsmöglichkeit können die Abhängigkeiten verschiedener Aktivitäten von einem Datenslot visualisiert werden. Beispielsweise können so alle Knoten bestimmt werden, die vor einem lesenden Knoten liegen und den gleichen Datenslot beschreiben.

☞ ***Modellierungsunterstützung 9:***

- Von der Prozeßvorlage benutzte Datenslots können nicht gelöscht werden.
- Ein- und Ausgabeparameter einer Aktivität können nur mit Datenslots gleichen Typs verbunden werden.
- Vorher obligate beschriebene Datenslots werden bei der Verknüpfung mit Eingabeparametern als „[obligat]“ markiert.
- Alle Knoten, die aus einem bestimmten Datenslot lesen oder ihn beschreiben, können angezeigt werden.

5.4 Die Modellierung von Zeitinformationen

Für die Modellierung von Zeitinformationen gibt es im Ablaufgraphen zwei Stellen. Dies sind Aktivitätenvorlagen und spezielle Zeitkanten, die zwei Knoten miteinander verbinden.

Für Aktivitätenvorlagen sind sechs Zeitattribute möglich. Diese Attribute sind die minimale und maximale Dauer der Aktivität, der früheste Anfangszeitpunkt (FAZ), der späteste Anfangszeitpunkt (SAZ), der früheste Endzeitpunkt (FEZ) und der späteste Endzeitpunkt (SEZ).
Um diese sechs Zeitinformationen für Aktivitätenvorlagen festzulegen, gibt es zwei Möglichkeiten im Programm. Die erste Stelle ist das Eigenschaftsfenster für Knoten und Aktivitätenvorlagen rechts neben dem Kontrollflußfenster. Hier können auf der dritten Registerkarte „Zeit“ diese Zeitwerte verändert werden. Zu beachten ist hier wieder, daß alle Knoten die auf der Aktivitätenvorlage basieren anschließend neue Zeitwerte besitzen.

Beim Erzeugen einer Aktivitätenvorlage können in dem Dialog aus Abbildung 40 die sechs Zeitwerte auf der Registerkarte mit dem Namen „Zeit“ eingegeben werden.

Um Zeitkanten zwischen zwei Knoten zu ziehen, muß wie in Abschnitt 5.2.2 vorgegangen werden. Zeitkanten können nur zu Knoten gezogen werden, die in der Nachfolgermenge des Startknotens der Zeitkante zu finden sind. Deshalb ergibt sich beim Löschen von Sync-Kanten ein Sonderfall. Hier kann es passieren, daß durch das Löschen der Sync-Kante ein Zielknoten einer Zeitkante nicht mehr in der Nachfolgermenge des Startknotens der Zeitkante sich befindet. Die Zeitkante wird dann automatisch mit der Sync-Kante gelöscht.

Um die Zeitwerte einer Zeitkante zu ändern, muß mit der rechten Maustaste der „Hotspot“ der Zeitkante angeklickt werden. Daraufhin erscheint ein Dialog, der die minimale und maximale Dauer der Zeitkante anzeigt. Das Markieren der Zeitkanten verläuft genauso wie bei anderen Kanten. Es muß wiederum der „Hotspot“ der Kante mit der linken Maustaste angeklickt werden. Liegen mehrere Kanten übereinander, so wird ein Auswahldialog für die Kanten angezeigt.

Das Löschen der Zeitkanten verläuft auch wie bei anderen Kanten. Mit der rechten Maustaste muß der „Hotspot“ der Zeitkante angeklickt werden und im anschließend erscheinenden Pop-up-Menü muß der Befehl “Kante löschen“ gewählt werden. Daraufhin wird die Kante gelöscht oder ein Auswahldialog wird bei mehreren übereinanderliegenden Kanten angezeigt. In diesem Dialog können wie schon vorher beschrieben eine oder mehrere Kanten gelöscht werden.

☞ ***Modellierungsunterstützung 10:***

- Zeitkanten können nur zu Nachfolgerknoten gezogen werden.
- Beim Löschen von Sync-Kanten werden ungültig gewordene Zeitkanten automatisch gelöscht.

5.5 Der Abschlußtest

Beim Abschlußtest für eine Prozeßvorlage werden verschiedene Tests für Kontroll- und Datenfluß durchgeführt, um die Korrektheit der Prozeßvorlage sicherzustellen. Das Ergebnisfenster des Abschlußtests kann z.B. wie in Abbildung 43 aussehen. Dabei kann unterschieden werden zwischen dem Test für den Kontrollfluß und dem Tests für den Datenfluß.

Für den Kontrollfluß wird getestet, ob Fehlerkanten und Verzweigungskanten einen Entscheidungswert besitzen. Besitzt eine solche Kante keinen Wert, so ist dies ein Fehler, der die Ausführung der Prozeßvorlage unmöglich macht. Alle anderen Fehlermöglichkeiten im Kontrollfluß werden schon bei der Modellierung ausgeschlossen. Diese zwei beschriebenen Tests sind die Punkte 2 und 3 in Abbildung 43.

In bezug auf den Datenfluß werden mehr Tests als beim Kontrollfluß vorgenommen. Der Grund dafür ist, daß bei der Modellierung des Kontrollflusses schon viele Fehlermöglichkeiten aufgrund der Konstruktion des Kontrollflußgraphen im Editor ausgeschlossen werden können. Die Tests zur Korrektheit des Datenflusses sind die Punkte 1, 4, 5 und 6 in Abbildung 43.

Im ersten Punkt des Abschlußtests wird ermittelt, ob alle obligaten Eingabeparameter der Aktivitäten sicher mit einem Wert, der von den Vorgängeraktivitäten geschrieben wird, versorgt werden. Dieser Wert muß auf jeden Fall beim Start der Aktivität vorhanden sein, da sonst der Workflow nicht ausgeführt werden kann. Wird das Fehlen der Versorgung eines obligaten Eingabeparameters beim Abschlußtest entdeckt, so wird ein Fehler gemeldet.

Abbildung 43: *Resultatsfenster des Abschlußtests*

In Punkt 4 wird geprüft, ob in der Prozeßvorlage parallele Schreiboperationen zweier Knoten auf einen Datenslot existieren. Solche Schreiboperationen sind Modellierungsfehler, die abgefangen werden müssen. Die Vermeidung von parallelen Schreiboperationen bedeutet, daß in parallelen Verzweigungen ein Datenslot nicht von Aktivitäten unterschiedlicher Zweige gleichzeitig beschrieben werden darf. Ansonsten wäre für nachfolgende Knoten nicht eindeutig festgelegt, welcher Wert gelesen wird. Eine Situation, in der zwei Knoten in unterschiedlichen Ästen einer parallelen Verzweigung, einen Datenslot beschreiben, kann mit Hilfe von Synchronisationskanten implementiert werden. Durch die Synchronisationskante können die beiden Knoten nicht mehr parallel ausgeführt werden und beide Knoten werden sequentiell gestartet.

In Punkt 5 wird getestet, ob alle Parameter der benutzten Aktivitätenvorlagen auf einen Datenslot abgebildet sind. Dabei wird beim Fehlen einer Abbildung eines Parameters auf eine Datenslot nur eine Warnung angezeigt, da bei der Ausführung des auf der Prozeßvorlage basierenden Workflows nicht alle Parameter mit Datenslots verbunden sein müssen.

Der Punkt 6 sucht im Prozeßgraphen nach überflüssigen Schreiboperationen von Aktivitäten auf Datenslots. Überflüssige Schreiboperationen entstehen, wenn auf einen Datenslot direkt nach einem Schreibzugriff einer Aktivität ein weiterer Schreibzugriff einer anderen Aktivität stattfindet. Solche überflüssigen Schreiboperationen behindern nicht die Ausführung des Workflows und müssen daher nicht vermieden werden. Falls überflüssige Schreiboperationen entdeckt werden, so ist deren Anzeige nur als Warnung zu verstehen.

☞ ***Modellierungsunterstützung 11:***

Der Abschlußtest prüft folgende Kriterien auf Korrektheit:

- Sind alle obligaten Eingabeparameter der Aktivitäten versorgt?
- Sind die Entscheidungswerte aller Kanten einer bedingten Verzweigung gesetzt?
- Sind die Fehlerwerte aller Fehlerkanten gesetzt?
- Existieren parallele Schreiboperationen in parallelen Verzweigungen?
- Sind alle Parameter mit einem Datenslot verbunden?
- Existieren überflüssige Schreiboperationen?

Hinweis:

Die korrekte Struktur des Kontrollflusses (Kontrollkanten) wird durch den syntaxgesteuerten Editor sichergestellt.

6. Der Entwurf und Aufbau des Editors

Dieses Kapitel beschreibt die Analyse, das Design und die Implementierung des Workflow-Editors WFEdit2.

In Abschnitt 6.1 werden zuerst die durch die Analyse des Problems gefundenen Anforderungen an den Editor erläutert. Anschließend wird das Design dargestellt. Die während der Designphase gewählte Architektur des Editors spiegelt sich auch in der Aufteilung der Klassen des Editors auf die verschiedenen Module wieder. Die Modularisierung wird in Abschnitt 6.3.1 beschrieben. Nachdem die Module kurz vorgestellt worden sind, wird in den nächsten Abschnitten das Zusammenspiel der einzelnen Module und deren wichtigste Klassen aufgezeigt. Dabei erfolgt auch die Vorstellung der wichtigsten Methoden der Klassen.

Die Realisierung des in dieser Arbeit entstandenen Workflow-Editor WFEdit2 erfolgte in der Programmiersprache Java mit Hilfe des Borland JBuilder2. Die Java-Version, die dem JBuilder2 zur Entstehungszeit zugrunde lag, war das JDK 1.1.6 der Firma Sun. Die Verwendung einer visuellen Entwicklungsumgebung bietet einige wichtige Vorteile, die den Entwicklungsprozeß einer Anwendung entscheidend beschleunigen. So besitzt der JBuilder2 einen Quellcode-Editor, ein visuelles Entwurfswerkzeug, eine Komponentenpalette (Java-Beans [Fla98]), einen Projekt-Browser, einen Eingenschafts-Inspektor für Komponenten, einen integrierten Debugger und einen Compiler. Eine herausragende Fähigkeit des JBuilder2 ist die Synchronisation zwischen dem Quellcode und dem visuellen Entwurf. D.h., daß Änderungen am Quellcode sofort in den visuellen Entwurf der Oberfläche eingehen. Umgekehrt gilt dasselbe. Diese Fähigkeit erlaubt die Entwicklung einer Anwendung in dem Modus, der für das jeweilige Programmelement am besten geeignet ist. Entwicklungumgebungen, die eine solche Fähigkeit besitzen, werden als „Two-Way-Tools“ bezeichnet[1].

Die Anwendung ist als eigenständige Java-Applikation realisiert worden. D.h. das Programm wurde nicht web-basiert als Java-Applet entwickelt. Ein Java-Applet ist eine Java-Anwendung, die eine spezielle Schnittstelle implementiert und so durch andere Programme, wie z.B. einen Web-Browser, ausgeführt werden kann. Die Implementierung als Java-Applet wurde nicht in Betracht gezogen, da Applets bestimmten Einschränkungen unterliegen, die aus dem Sicherheitskonzept der Programmiersprache Java resultieren. So kann ein Applet keine Zugriffe auf ein lokales Dateisystem machen. Dieser Nachteil würde die Nutzung des Workflow-Editors stark einschränken.

6.1 Die Anforderungen an den Editor

Die Hauptaufgabe dieser Arbeit war, einen Workflow-Editor basierend auf der Architektur des ADEPT-Basismodells zu realisieren. Bei der Analyse dieser Aufgabe zeigten sich folgende Anforderungen, die vom Editor erfüllt werden müssen:

[1] Eine weiterer Vertreter dieser Art von Programmierumgebungen ist Borland Delphi.

1. Modellierung von Kontroll- und Datenfluß:

Der Editor muß alle Konstrukte des ADEPT-Basismodells (siehe Kapitel 3) modellieren können. Die Modellierung soll für den Benutzer schnell und einfach möglich sein. Um dies sicherzustellen, muß eine geeignete Darstellung des Kontroll- (siehe Kapitel 4) und Datenflusses (siehe Kapitel 5) gefunden werden.

2. Konsistenzprüfungen von Kontroll- und Datenfluß

Ein sehr wichtiger Teil des Editors sind die Konsistenzprüfungen des Kontroll- und Datenflusses einer Prozeßvorlage, da der Editor den Benutzer bei der Modellierung unterstützen soll. D.h. die syntaktische Korrektheit der mit dem Editor modellierten Prozeßgraphen soll gewährleistet sein. Außerdem soll auch die korrekte Datenflußmodellierung sichergestellt werden. Dazu müssen neben der Sicherstellung der Korrektheit bei der Modellierung (z.B. Vorauswahl bestimmter Möglichkeiten durch die Anwendung) auch verschiedene Algorithmen für Korrektheitstests des fertig modellierten Workflows entwickelt werden (siehe Kapitel 4).

3. Weitere Funktionalitäten

Der Editor muß noch weitere notwendige Funktionalitäten besitzen. Um eine einfach und schnelle Modellierung zu gewährleisten, muß der Prozeßgraph in verschiedenen Größen dargestellt werden können.

Außerdem soll der Editor zwei Möglichkeiten bieten, eine Prozeßvorlage zu speichern. Erstens müssen Prozeßvorlagen in einer Datenbank, die über einen Workflow-Server angesprochen wird, gespeichert werden können. Zweitens soll die Möglichkeit bestehen, eine Prozeßvorlage in einer wfv-Datei (siehe Anhang A) zu speichern. Dies soll die Verwendung des Modellierungswerkzeugs auch ohne die Infrastruktur einer Datenbank bzw. Workflow-Servers ermöglichen.

4. Betrachtung von Aspekten der Software-Ergonomie

Da ein Workflow-Editor ein Programm mit sehr vielen Benutzerinteraktionen ist, müssen die Aspekte der Software-Ergonomie betrachtet werden, da bei deren Mißachtung die Modellierung eines Workflows durch den Benutzer nicht mehr schnell und einfach durchgeführt werden kann.

Was ist Software-Ergonomie? Nach [EOO94] ist die Software-Ergonomie die benutzergerechte Gestaltung der Mensch-Computer-Interaktion. Dabei werden Ergebnisse einer disziplinübergreifenden Forschung benutzt. Wichtige Fachbereiche, die zur Software-Ergonomie beitragen, sind Informatik, Psychologie und die Arbeitswissenschaft.

Die Software-Ergonomie teilt sich in drei Schwerpunkte [Maa93]:

- Der technische Schwerpunkt:
  In diesem Bereich wird versucht, die technischen Komponenten der MCI durch Neu- und Weiterentwicklung zu verbessern.

- Der kognitiv-psychologische Schwerpunkt:
  Hier wird die Systemgestaltung unter Gesichtspunkten der menschlichen Wahrnehmung, des Denkens, Problemlösens und Lernens analysiert.

- Der arbeitspsychologische Schwerpunkt:
  Dieser Schwerpunkt sieht Technikeinsatz im größeren Kontext von organisierter Arbeit. Es werden dazu hauptsächlich die Faktoren Aufgabe und Organisation betrachtet.

Für die Entwicklung einer Anwendung wie einen Workflow-Editor ist der kognitiv-psychologische Schwerpunkt der Software-Ergonomie von Wichtigkeit, da hauptsächlich Aspekte der Oberflächengestaltung betrachtet werden müssen. Der Artikel [Hob95] zeigt einige Prinzipien einer guten Oberflächengestaltung auf und eine ausführliche Behandlung dieses Themas findet sich in [Shn92].

Ein Grund für die besondere Betrachtung der Möglichkeiten einer ergonomische Oberflächengestaltung ist, daß das Design der Programmoberfläche von vielen Entwicklern vernachlässigt wird. Das spiegelt sich in der Meinung vieler Entwickler wieder, daß eine ergonomische Benutzeroberfläche intuitiv implementierbar sei. Daß dies nicht so ist, zeigen viele Programme mit einer schlecht bedienbaren Benutzeroberfläche. Außerdem wurde in einer Studie [Mol90] festgestellt, daß ein Designer oder Programmierer durchschnittlich nur 37% der Designschwächen eines in dieser Studie vorgegebenen Dialogs erkennt. Der beste Teilnehmer an dieser Studie fand 60% der Schwächen des Dialogs.

Die ideale Interaktionsform für die Modellierung des Prozeßgraphen im Editor ist das Konzept der direkten Manipulation (DM) [FäZ88]. Eine Benutzeroberfläche, die folgende Eigenschaften besitzt, kann als DM-Oberfläche[1] bezeichnet werden:

- Ständige Darstellung der interessierenden Objekte
- Physische Handlungen oder Drücken beschrifteter Knöpfe an Stelle von komplexer Syntax
- Schnelle, inkrementelle und reversible Operationen mit sofort sichtbarem Effekt an den interessierenden Objekten

Die wesentlichen Vorteile der DM sind die leichte Erlernbarkeit und auch gelegentliche Benutzer können die Interaktionsmöglichkeiten gut im Gedächtnis behalten. Außerdem sehen die Benutzer, ob eine Aktion sie ihrem Ziel näherbringt. Auch haben die Nutzer der Anwendung weniger Angstgefühle, da DM-Systeme es ermöglichen, daß Aktionen gefahrlos ausprobiert und rückgängig gemacht werden können. Ein wesentlicher Nachteil der DM-Oberflächen ist, daß das Hantieren mit einer Maus langsamer ist als die geübte Bedienung einer Tastatur.

Neben diesem allgemeinen Entwurf der Benutzerschnittstelle eines Programms gibt es weitere Dinge, die beim Design der Oberfläche beachtet werden sollten. Überwacht werden muß das Design der Meldungen des Systems, da der Benutzer durch schlecht formulierte Meldungen von einem Anwendungssystem schnell enttäuscht sein kann. [Shn82] beschreibt ausführlich, wie gut formulierte und aussagekräftige Meldungen aussehen sollten. Neben den Meldungen spielt auch das Menüdesign eine Rolle, um eine Anwendung schnell und einfach bedienen zu können. Die Arbeit [PaR88] gibt eine Anleitung für die Entwicklung eines gut zu bedienenden Menüs. Zusätzlich muß auch das Aussehen von Icons und Symbolleisten durchdacht sein. Das benutzerfreundliche Aussehen von Icons und Symbolleisten wird in [MoB95] erläutert. Ein weiterer Punkt den es zu beachten gilt, ist die Antwortzeit eines Programms. Vor allem Personen, die ein Programm häufig nutzen, sind nicht zufrieden mit einem Programm, das lange Antwortzeiten besitzt oder bei langen Operationen keine Rückmeldung liefert. Die Thematik der Antwortzeiten wird in [Shn84] beschrieben. In dem in dieser Arbeit entstandenen Editor wurden die hier genannten Punkte beachtet, wobei die Antwortzeitproblematik nur sekundär war, da es sich bei dem Editor um eine Prototypen handelt.

Neben den genannten Punkten gibt es vor allem bei der Entwicklung einer kommerziellen Systems noch weitere Anforderungen. So muß für ein kommerzielles System auch eine benutzergerechte Hilfe implementiert werden.

[1] DM-Oberflächen werden häufig als WYSIWYG („What you see is what you get“)-Oberflächen bezeichnet.

6.2 Die grundlegende Architektur des Editors

Die Entwicklung großer Software-Projekte läßt sich nicht auf der Intuition eines oder mehrerer Entwickler aufbauen. Solch eine unstrukturierte Vorgehensweise führt bei der Entwicklung einer Anwendung zu Problemen. Eine so entstandenes Programm ist häufig schwer zu verstehen und damit auch schwer zu warten. Auf Grund dessen sind einzelne Teile oder auch die ganze Software nur schwerlich wiederzuverwenden.

Ein strukturierter Softwareentwicklungsprozeß hilft viele Probleme zu vermeiden. Die Softwareentwicklung läßt sich in drei Stadien einteilen:

- **Analyse:**
  Beschreibung des Problems, das mit Hilfe der Software gelöst werden soll.

- **Design:**
  Zusammenfassen der Ergebnisse der Analyse um eine globale Struktur für das System zu erzeugen.

- **Implementierung:**
  Codieren und Testen der Software bzw. der einzelnen Softwarekomponenten.

Diese drei Stufen werden nicht nur einmal ausgeführt, sondern jedes Stadium wird bei der Softwareentwicklung mehrfach durchlaufen. Jeder Durchlauf betrachtet dabei notwendige Veränderungen des Systems und verfeinert das Endprodukt. Einige wichtige, umfangreiche Punkte, der Softwareentwicklung sind nicht als gesondertes Stadium berücksichtigt worden, da sie im gesamten Softwareentwicklungsprozeß immer wieder betrachtet werden sollten. Diese Punkt sind das Experimentieren, das Testen und die bei jedem Schritt begleitend durchgeführte Analyse des Designs und der Implementierung. Weitere Punkte die bei der Entwicklung von Software sind außerdem die Dokumentation und das Management des Entwicklungsprozesses.

Gerade das Design der Software ist ein wichtiger Punkt, der über das Gelingen eines Projekts entscheidet. Ein schlechtes Design erschwert die Entwicklung der Anwendung, da durch das schlechte Design Veränderungen nur schwer und mit hohem Zeitaufwand durchzuführen sind. Welche Kriterien sind nun wichtig für das Design der Software? Durch den iterativen Prozeß der Softwareentwicklung macht das System eine Veränderung durch. Es wird auf verschieden Art und Weise erweitert und geändert. Daraus folgt, daß die Applikation so entworfen werden sollte, daß sie leicht veränderbar ist, d.h. beim Design muß auf folgende drei Punkte geachtet werden:

- Flexibilität

- Erweiterbarkeit

- Portabilität

Dies läßt sich am besten erreichen, indem man die Bestandteile, die sich mit einer gewissen Wahrscheinlichkeit ändern werden, voneinander trennt. Dies Trennung bedeutet, daß beim Design der Anwendung die einzelnen Teile so gestaltet werden sollten, daß ein Teil möglichst wenig Kenntnis von den anderen Teilen hat. Eine Art des Designs einer Anwendung, die diese Vorgaben erfüllt, ist die Model-View-Controller-Architektur (MVC-Architektur) [Bur92]. Die MVC-Architektur diente auch als Grundlage für das Design des in dieser Arbeit entstandenen Workflow-Editors WFEdit2. Die Beschreibung des Systementwurfs in den nachfolgenden Abschnitten basiert auf dem MVC-Konzept und einige Designentscheidungen werden durch die Wahl dieses Konzepts begründet.

6.2.1 Die Model-View-Controller-Architektur

Die MVC-Architektur ist eine elegante und einfache Architektur zur Erstellung von graphischen Anwendung. Aber durch ihre besondere Entwicklungsphilosphie im Vergleich zu traditionellen Anwendungen, wird das MVC-Paradigma in diesem Abschnitt erläutert.

Die Grundidee der MVC-Architektur ist die Trennung zwischen Benutzereingaben, dem visuellen Feedback auf dem Bildschirm und dem Modell des Problems. Dadurch gliedert sich die MVC-Architektur in drei Komponenten, die jeden genannten Teil verkörpern. Die drei Komponenten lassen sich folgendermaßen charakterisieren:

- Die View (Präsentationsschicht):
  Die Aufgabe der Präsentationsschicht ist es, die Objekte der Modellschicht auf dem Bildschirm darzustellen. Dabei kann es mehrere problemorientierte graphische bzw. textuelle Ansichten desselben Model-Objekts geben.

- Der Controller (Anwendungslogik):
  Die Anwendungslogik wird für die Interaktion des Benutzer mit der graphischen Oberfläche angewandt. Die Anwendungslogik interpretiert die Eingaben des Benutzers und transformiert die Eingaben in Operationen auf Objekte der Präsentationsschicht bzw. der Datenbasis.

- Das Model (Datenbasis):
  Die dem Problembereich zugrundeliegenden Informationen werden in Model-Objekten repräsentiert. Ein Model-Objekt hat dabei keine Kenntnisse über seine Darstellung auf dem Bildschirm. Ändert der Controller Model-Objekte, so werden alle Views, die diese Model-Objekte darstellen, über die Änderungen informiert. Ebenfalls muß ein verändertes View-Objekt sein zugehöriges Model-Objekt über die Änderung informieren.

Durch diese Trennung eignet sich die MVC-Architektur vor allem für interaktive Anwendungen, denen ein objektorientiertes Design zugrundeliegt. Ursprünglich stammt das MVC-Paradigma aus dem Umfeld der Programmiersprache Smalltalk. Mittlerweile wird es aber in vielen Gebieten eingesetzt. So verwendet die Programmiersprache Java dieses Konzept bei der Implementierung bestimmter Klassen. Als Beispiele für die Verwendung des MVC-Paradigmas sind einige Komponenten wie `JList`, `JComboBox` der Swing-Bibliothek [Sun98] zur Erstellung von graphischen Benutzeroberflächen zu nennen. Weitere Beispiele für das Vorkommen der MVC-Architektur in Java sind das Observer/Observable-Model und das Event-Model zur Steuerung von graphischen Anwendung [Fla98].

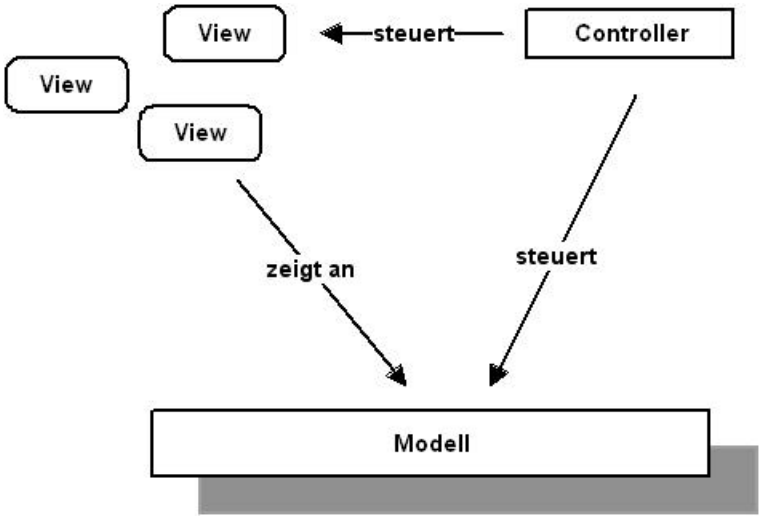


Abbildung 44: *Die Model-View-Controller Architektur*

Der Vorteil der MVC-Architektur ist die Trennung der einzelnen Schichten voneinander. In der View ist das Wissen über die Modelldaten enthalten. Umgekehrt gilt dies jedoch nicht! D.h. ein Model-Objekt weiß nichts über seine Darstellung auf dem Bildschirm. Daher gibt es nur eine Abhängigkeitsrichtung, was das Hinzufügen und Entfernen von View-Objekten ermöglicht, ohne die Implementierung der Model-Objekte zu verändern. Dadurch können in einer Anwendung die Model- und View-Objekte getrennt betrachtet werden und auch in einer verteilten Umgebung als verteilte Objekte realisiert werden.

6.3 Der Aufbau des Editors

Das verwendete MVC-Paradigma für das Design des Workflow-Editors zeigt sich auch in der Modularisierung der Anwendung. So besitzt das Programm Module, die der View-Ebene und der Model-Ebene und der Controller-Ebene entsprechen.

Die Module werden in der Programmiersprache Java durch sogenannte Packages repräsentiert. Diese Packages sind hierarchisch organisiert und können mehrere Klassen enthalten. Ein Package-Name wird auf das Dateisystem des Rechners abgebildet, d.h. ein Package mit dem Namen `WFEdit2.View.Dialogs` wird z.B. auf das Verzeichnis D:\WFEdit2\View\Dialogs umgesetzt. In diesem Verzeichnis befinden sich alle Klassen des benannten Packages im Java-Quellcode.

6.3.1 Übersicht über die Packages

Die Einteilung der Klassen in Packages wird neben der MVC-Architektur zum Teil auch von der verwendeten Entwicklungsumgebung bestimmt. So legt der JBuilder2 immer zwei Klassen an, die als

Grundgerüst für das Programm dienen. Eine Klasse dient dazu, das eigentliche Programm zu realisieren (im Editor: `WFEdit2Frame`). Diese Klasse implementiert das Hauptfenster und wird immer von einer Java-Fensterklasse wie z.B. `JFrame` abgeleitet. Die Aufgabe der anderen Klasse ist es, dieses Hauptfenster auf dem Bildschirm zu erzeugen (im Editor: `WFEdit2Andwendung`).

Die weiteren Klassen sind nach dem MVC-Paradigma angeordnet. Dabei zeigte sich, daß es auch sinnvoll ist, weitere Bereiche wie z.B. den Zugriff auf Dateien bzw. auf die Datenbank in einem eigenen Package zu realisieren, da dies die Übersichtlichkeit des Programms erhöht.

Der Workflow-Editor gliedert sich in die im folgenden kurz beschriebenen Packages:

- **`WFEdit2`**
  In diesem Package befinden sich zwei Klassen, die durch den JBuilder2 erzeugt wurden. Dies sind die oben beschriebenen Klassen `WFEdit2Frame` und `WFEdit2Anwendung`. Die Klasse `WFEdit2Frame` realisiert verschiedene Aufgaben des Controllers der MVC-Architektur.

  - **`WFEdit2.View`**
    Dieses Package enthält alle Klassen, die zur Darstellung verschiedener Elemente der Anwendung auf dem Bildschirm notwendig sind. In diesem Package wird auch die Darstellung der Model-Objekte realisiert.

    - **`WFEdit2.View.Dialogs`**
      In dieser Kategorie enthalten sind alle Klassen, die Dialoge des Programms erzeugen

    - **`WFEdit2.View.Util`**
      Dieses Package enthält Hilfsklassen, die für die Darstellung auf dem Bildschirm notwendig sind.
      Bsp.: Zentrieren einer Komponente auf dem Bildschirm.

  - **`WFEdit2.Model`**
    Dieses Package gruppiert alle Klassen, die die Datenbasis des Workflow-Editors darstellen.
    Bsp.: Klassen für Aktivitäten, Kanten.

    - **`WFEdit2.Util`**
      Hier befinden sich Hilfsklassen für das Model der Anwendung.
      Bsp.: Schnittmengenbildung zweier Felder.

  - **`WFEdit2.DataAccess`**
    Durch die Klassen dieses Packages werden die Daten eines Workflow-Schemas gespeichert bzw. gelesen. Dies kann in einer Datei oder in einer Datenbank geschehen. Die Datenbank ist dabei über einen Workflow-Server mit der Anwendung gekoppelt. Daneben wird in diesem Package ein Parser für die Workflow-Beschreibung in einer Datei implementiert.

    - **`WFEdit2.DataAccess.Util`**
      Dieses Package realisiert Hilfsklassen für den Zugriff auf die gespeicherten Daten.
      Bsp.: Bestimmung der IP-Adresse des Workflow-Servers.

6.3.2 Das Zusammenspiel der einzelnen Komponenten der Anwendung

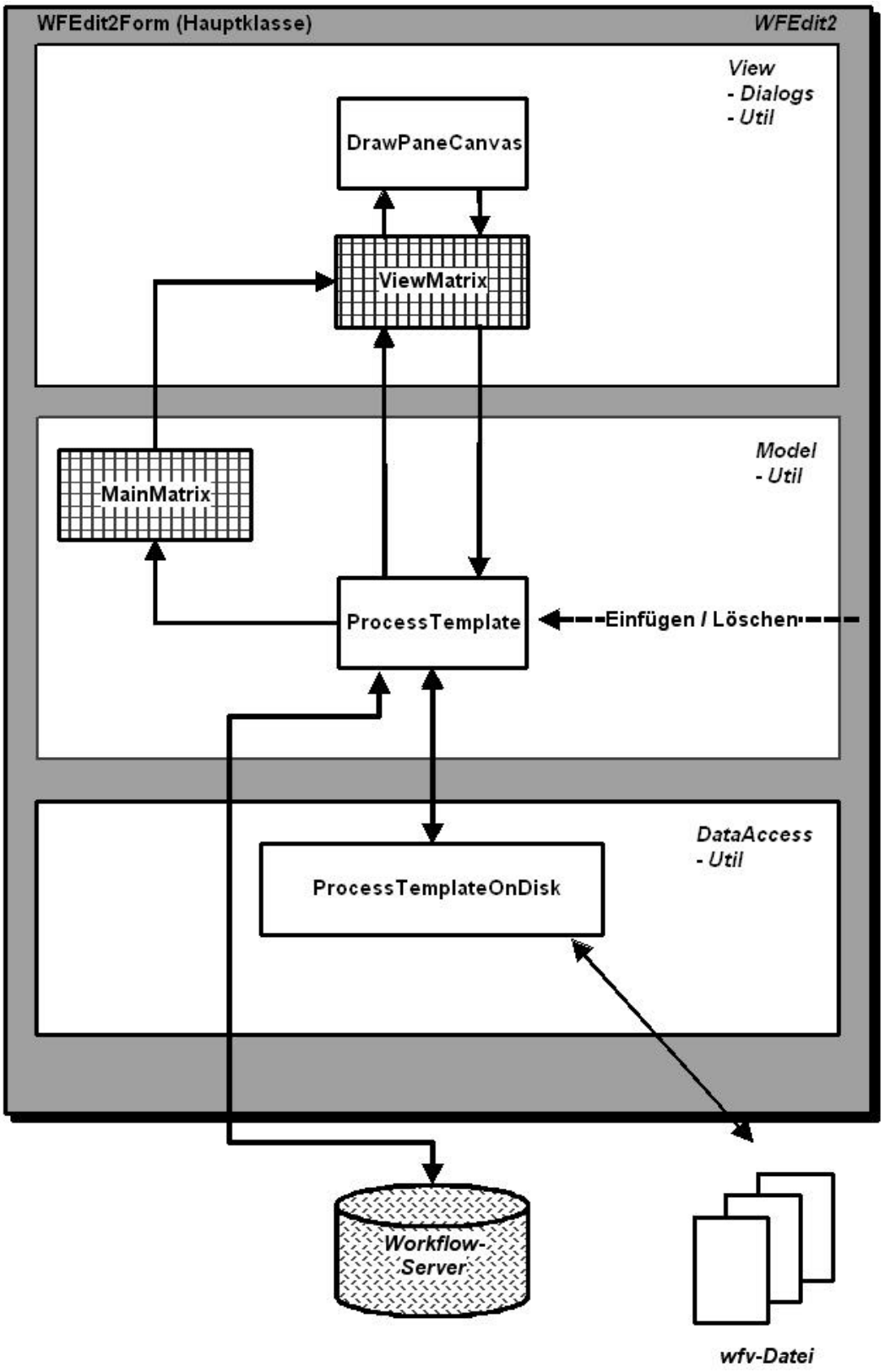


Abbildung 45: *Aufbau des Editors WFEdit2*

Die Funktionsweise des Programms läßt sich anhand des Schaubildes auf der vorhergehenden Seite (Abbildung 45) verdeutlichen. Das Schaubild zeigt die einzelnen Packages des Programms und die wichtigsten Klassen, die für das Verständnis des Programmablaufs innerhalb des Workflow-Editors notwendig sind. Auf Details der Oberflächenprogrammierung wird in dieser Beschreibung nicht eingegangen (siehe Quelltext), sondern es soll vielmehr auf den Aufbau bzw. die Veränderung eines Kontrollflußgraphen eingegangen werden. Die Benutzeroberfläche wird hauptsächlich durch die „Hauptklasse" der Anwendung `WFEdit2Frame` realisiert. Auf Details wie z.B. die Implementierung von Korrektheitschecks, wird bei der näheren Beschreibung der Packages und deren Klassen eingegangen, da diese Operationen als Methoden bestimmter Klassen implementiert sind.

Um eine Ablaufvorlage zu bearbeiten, muß diese zuerst aus einer Datei oder aus einer Datenbank, die über einen Workflow-Server zugänglich ist, geladen werden. Diese Aufgabe übernimmt die Klasse `DataAccess` aus dem gleichnamigen Package `DataAccess`. Von der Klasse `DataAccess` werden keine Instanzen erzeugt, da alle Methoden für den Zugriff auf die Daten als statische Methoden (Klassenmethoden) implementiert worden sind. Der Begriff statische Methode bedeutet, daß die Methode durch die Klasse aufgerufen wird und nicht durch eine Instanz der Klasse. Klassenmethoden sind in der Sprache Java am ehesten mit „globalen" Methoden anderer Programmiersprachen vergleichbar [Fla98]. Wird auf eine Prozeßdefinition in einer wfv-Datei zugegriffen, so wird diese Definition in einer Instanz der Klasse `ProcessTemplateOnDisk` gespeichert. Die Klasse `ProcessTemplateOnDisk` entspricht genau der Beschreibung eines Prozesses in einer wfv-Datei. Die Darstellung eines Prozesses in einer Instanz der Klasse `ProcessTemplateOnDisk` hat einige Nachteile, die z.B. Korrektheitstests wesentlich erschweren würden. Deswegen wird der Inhalt der Klasse `ProcessTemplateOnDisk` aufbereitet und in einer Instanz der Klasse `ProcessTemplate` gespeichert. Die Klasse `ProcessTemplate` bildet die Datenbasis der Anwendung und wird dem Model aus der MVC-Architektur zugeordnet. Die Darstellung einer Prozeßvorlage in einem Objekt, das auf der Klasse `ProcessTemplate` basiert, erleichtert wesentlich die Bearbeitung und Analyse einer Prozeßvorlage. Im Gegensatz zum Zugriff auf eine wfv-Datei wird der Inhalt einer Prozeßvorlage aus der Datenbank direkt in ein Objekt der Klasse `ProcessTemplate` gelesen. Der Zugriff erfolgt über ein API[1] für den Zugriff über einen Workflow-Server auf eine Datenbank. Die Speicherung einer Prozeßvorlage in der Datenbank bzw. in einem wfv-File verläuft in die entgegengesetzte Richtung durch Methoden der Klasse `DataAccess`. Dabei werden wiederum die gleichen Datenstrukturen wie beim Lesen der Daten benutzt.

Aus der so erzeugten Instanz der Klasse `ProcessTemplate` wird die Darstellung des Graphen auf dem Bildschirm generiert. Dazu wird ein Objekt, das auf der Klasse `MainMatrix` basiert, erzeugt. Die Klasse `MainMatrix` realisiert eine 100×100 Felder große Matrix, in die die Knoten des Kontrollflußgraphen eingefügt werden. Das Einfügen der Knoten wird durch den für diese Arbeit entwickelten Algorithmus aus Abschnitt 4.2 realisiert. Dabei werden nur die Knotennamen in die Matrix eingefügt, um die Positionen der Knoten zu berechnen. Mit Hilfe der Datenstruktur `MainMatrix` wird die maximale Zeilen- und Spaltenzahl des darzustellenden Kontrollflußgraphen ermittelt.

Basierend auf der maximalen Zeilen- und Spaltenanzahl wird eine Instanz der Klasse `ViewMatrix` in entsprechender Größe erzeugt. In diese Matrix werden Objekte eingefügt, die die Knoten des Ablaufgraphen des Prozesses visualisieren. Dieses Vorgehen entspricht der Beschreibung der MVC-Architektur. Die Knoten werden dabei in Objekten der Klasse `ViewNode` gespeichert und diese Objekte werden in ein Objekt der Klasse `ViewMatrix` eingetragen. Die Objekte der Klasse `ViewNode` enthalten die berechneten x- und y-Koordinaten der Knoten auf dem Bildschirm.

Nachdem die Knotenpositionen auf dem Bildschirm feststehen, werden die Knoten mit ihren entsprechenden Darstellungen auf dem Bildschirm gezeichnet. Dabei wird das Matrix-Objekt, das auf der Klasse `ViewMatrix` basiert, ausgelesen und jeder Knoten wird auf dem Zeichenfeld gezeichnet. In diesem Schritt dürfen Dummy-Knoten für Schleifen nicht auf dem Zeichenfeld angezeigt werden.

[1] API = **A**pplication **P**rogramming **I**nterface

Die Realisation des Zeichenfeldes erfolgt durch eine Instanz der Klasse `DrawPaneCanvas`. Mit Hilfe der x- und y-Koordinaten der Knoten werden die Kantenverläufe berechnet. Die Kanten werden dabei in Objekten der Klasse `ViewEdge` gespeichert. Diese Objekte enthalten alle x- und y-Koordinaten einer Kante, um sie auf dem Bildschirm darzustellen. Außerdem wird die x- und y-Position des „Hotspots“ einer jeden Kante berechnet und in dem entsprechenden Objekt der Klasse `ViewEdge` gespeichert.

Werden Kanten bzw. Knoten aus dem Kontrollflußgraphen gelöscht oder eingefügt, so geschehen diese Veränderungen zuerst immer in dem Objekt der Klasse `ProcessTemplate`. Aufgrund dieser Änderung des Objekts wird automatisch eine neue Bildschirmdarstellung des Graphen auf dem Bildschirm erzeugt. Der Aufbau der Darstellung verläuft dann genau so, wie vorher beschrieben.

Änderungen an Attributen von Kanten oder Knoten bewirken keinen Aufbau einer neuen Bildschirmdarstellung. Es werden dabei nur die entsprechenden Attribute der View- bzw. der Model-Objekte aktualisiert.

6.3.3 Der Controller

Die Aufgaben des Controller aus der MVC-Architektur werden durch die Klasse `WFEdit2.WFEdit2Anwendung` übernommen. Diese Klasse kann als Hauptprogramm der Anwendung betrachtet werden, da sie die Benutzeroberfläche generiert und die Verbindung zwischen den Model-Objekten und den dazugehörigen View-Objekten herstellt. Durch sie werden fast alle Interaktionsmöglichkeiten zwischen dem Benutzer und dem Programm implementiert. Weiter Interaktionsmöglichkeiten finden sich nur noch in den Dialogen, deren Code in dem Package `WFEdit2.View.Dialogs` zu finden ist.

Das Gerüst der Klasse `WFEdit2.WFEdit2Anwendung` wurde beim Anlegen eines neuen Projekts durch die Entwicklungsumgebung JBuilder erzeugt. Diese Gerüst enthält aber nur die allernotwendigste Funktionalität für das Hauptfenster der Anwendung und muß durch den Entwickler erweitert werden.

Die Klasse `WFEdit2.WFEdit2Anwendung` enthält ca. 7000 Zeilen. Ein Grund für die Größe dieser Klasse ist, daß neben den oben genannten Aufgaben noch verschiedene Operationen auf dem Workflowgraphen implementiert worden sind. Diese Operationen konnten mit Hilfe der View-Objekte wesentliche einfacher und effizienter verwirklicht werden als nur auf den Model-Objekten einer Prozessvorlage. Beispiele für diese Operationen sind die Korrektheitschecks des Abschlußtests und mehrere Prüfungen, die festlegen, ob eine Kante eingefügt werden kann oder nicht.

Die Klasse `WFEdit2.WFEdit2Anwendung` erweitert die Klasse `JFrame` der Swing-Bibliothek. Die Klasse `JFrame` ist im Package `java.swing` zu finden[1].

Die **wichtigsten** Methoden der Klasse `WFEdit2.WFEdit2Anwendung` sind:

[1] `JFrame` befindet sich in Java 2 im Package `javax.swing`.

`obligateParametersProvidedWithValues`	Diese Methode des Abschlußtests prüft, ob alle obligaten Eingabeparameter aller Aktivitäten mit einem Wert versorgt sind.
`existsAllErrorParameters`	Die Methode des Abschlußtests prüft, ob alle Fehlerkanten einen Wert besitzen.
`existsAllDecisionParameters`	Die Methode des Abschlußtests prüft, ob alle Kanten einer bedingten Verzweigung mit einem Wert versehen sind.
`existsParallelWritings`	Die Methode des Abschlußtests prüft, ob parallele Schreiboperationen in einer parallelen Verzweigung der Ablaufvorlage auftreten
`allParametersMapped`	Die Methode prüft, ob alle Parameter der in der Ablaufvorlage verwendeten Aktivitäten auf einen Datenslot abgebildet sind.
`finalCheck`	Diese Methode ruft alle Prüfungen des Abschlußtests auf.
`branchNodes`	Diese Methode bestimmt, ob sich zwei Knoten in einer Verzweigung befinden und sie gibt ggf. den dazugehörigen Verzweigungs- und Synchronisationsknoten aus.
`cycleCheck`	Die Methode prüft, ob der gesamte Graph einen Zyklus enthält. Es werden dabei nur Kontroll- und Sync-Kanten betrachtet.
`findAllObligateWrittenDS`	Die Methode sucht alle Knoten, die bis zu einem bestimmten Knoten einen Datenslot obligat beschreiben.
`findBranchNodeOfSyncNode`	Die Methode sucht zu einem gegebenen Synchronisationsknoten einer Verzweigung den dazugehörigen Verzweigungsknoten
`findWritingNodes`	Die Methode sucht alle Knoten, die einen Datenslot beschreiben. Ein obligates oder optionales Beschreiben ist unwesentlich.
`insertBranch`	Die Methode fügt eine Verzweigung in die Prozeßvorlage ein. Die Methode kann alle Arten von Verzweigungen verarbeiten.

insertBranch2	Die Methode fügt zwischen zwei bestehenden Knoten eine Verzweigung ein. Die zwei Knoten werden in eine Verzweigungs- bzw. Synchronisationsknoten umgewandelt.
branchCheck	Die Methode testet, ob zwischen zwei bestehenden Knoten nachträglich eine Verzweigung eingefügt werden kann.
checkErrorEdgeForBranch	Die Methode prüft, ob eine neue bedingte Verzweigung oder eine neue parallele Verzweigung beim nachträglichen Einfügen mit einer bestehenden Fehlerkante in Konflikt geraten würde.
deleteLoopEndNode	Löscht eine Schleifenendknoten mitsamt der Schleife.
deleteLoopStartNode	Löscht eine Schleifenstartknoten mitsamt der Schleife.
deleteSyncNode	Löscht einen Synchronisationsknoten einer Verzweigung mitsamt der Verzweigung.
deleteVerzNode	Löscht einen Verzweigungsknoten mitsamt der Verzweigung.
insertLoop	Die Methode fügt eine Schleife nachträglich in den Prozeßgraphen ein.
errorCheck	Die Methode testet, ob eine Fehlerkante eingefügt werden kann.
insertError	Die Methode fügt eine Fehlerkante ein.
deleteEdge	Löscht eine Kante aus der Prozeßvorlage und prüft z.B. ob eine Zeitkante beim Löschen einer Sync-Kante noch gültig ist.
deleteNode	Löscht einen Knoten und setzt die bestehenden Kanten so, daß keine Lücke im Graph entsteht. Verschieden Konstrukte wie z.B. Schleifen müssen betrachtet werden.

errorEdgeCheck	Die Methode testet , ob eine Fehlerkante von außen in eine bedingte oder parallele Verzweigung mit finaler Auswahl hineinführt.
insertTime	Die Methode fügt eine Zeitkante in die Prozeßvorlage ein.
insertSync	Die Methode fügt eine Sync-Kante in die Prozeßvorlage ein. Der Typ der Sync-Kante wird der Methode übergeben.
insertPriority	Fügt eine Priorisierungskante in die Prozeßvorlage ein.
priorityCheck	Die Methode prüft, ob eine Priorisierungskante zwischen zwei gewählten Knoten eingefügt werden kann.
syncCheck	Die Methode prüft, ob eine Synchronisationskante zwischen zwei gewählten Knoten eingefügt werden kann.
timeCheck	Die Methode testet, ob eine Zeitkante zwischen zwei gewählten Knoten eingefügt werden kann.
loopCheck	Die Methode führt einen Test aus, der bestimmt, ob eine Schleife nachträglich um zwei gewählten Knoten eingefügt werden kann.
searchDependentNodes	Die Methode sucht alle von einem Datenslot abhängigen Knoten und liefert die Menge der Schreibknoten bzw. Leseknoten zurück.
updateDisplay	Erzeugt eine neue Bildschirmdarstellung des Kontrollflußgraphen. Dabei wird die Größe der Zeichenfläche angepaßt und die Placemarkerbox aktualisiert.
insertNode	Fügt einen Knoten in die Prozeßvorlage ein. Verbindungskanten zu diesem Knoten werden automatisch erzeugt.

Neben diesen wichtigen Methoden besitzt die Klasse Methoden, die bestimmte Ereignisse der Benutzeroberfläche, wie z.B. Mausklicks verarbeiten. Mit Hilfe des JBuilders lassen sich deren Aufrufpunkte und deren Arbeitsweise bestimmen (siehe Quelltext).

6.3.4 Das Model

Das Model enthält die Daten einer Prozeßbeschreibung und bildet die Datenbasis der Anwendung. Die wichtigste Klasse des Models ist `WFEdit2.Model.ProcessTemplate`. In dieser Klasse werden die Attribute einer Prozeßvorlage gespeichert und verschiedene Operationen auf der Prozeßvorlage definiert.

Die wichtigsten öffentlichen Attribute der Klasse `ProcessTemplate` sind:

- ID der Prozeßvorlage
- Der Name, die Beschreibung und die Kategorie der Prozeßvorlage
- Aktivitätenvorlagen
- Knoten
- Kanten
- Datenslots

Die Aktivitätenvorlage, die Knoten, die Kanten und die Datenslots werden in jeweils einer Hashtabelle gespeichert, um eine schnellen Zugriff auf diese Daten über ihren Namen zu gewährleisten. Kanten werden dabei mit dem Namen des Startknotens als Schlüssel gespeichert und bestimmt..

Die Aktivitätenvorlagen basieren alle auf der Klasse `ActivityCore` aus dem Package `WFEdit2.DataAccess`. Die Klasse `NodeModel` beschreibt Knoten des Workflowgraphen, Kanten werden durch die Klasse `EdgeModel` beschrieben und Datenslots durch die Klasse `Dataslot`. Die Implementierungen dieser Klassen befinden sich im Package `WFEdit2.Model`.

Die **wichtigsten** Methoden der Klasse `ProcessTemplate` sind:

`writerExists`	Die Methode prüft für genau eine Paar bestehend aus einem Knoten n und einem Datenslot d, ob dieser Datenslot d von Vorgängerknoten des Knotens n obligat versorgt wird. Siehe Abschnitt 4.4.3.
`getSuccessorSet`	Die Methode liefert alle Nachfolgerknoten eines bestimmten Knotens über Kontroll- und Sync-Kanten.
`getPredecessorSetControl`	Die Methode liefert alle Vorgängerknoten eines bestimmten Knotens, der nur über Kontrollkanten von diesen Knoten aus erreichbar ist.
`getSyncNodeOfBranchNode`	Die Methode liefert den Synchronisationsknoten einer Verzweigung, deren Verzweigungsknoten bekannt ist.

findPathAToB	Die Methode findet den kürzesten Pfad zwischen zwei Knoten. Siehe Abschnitt 4.3.2.
findAllNodesBetweenTwoNodes	Diese Methode liefert die Menge aller Knoten, die zwischen zwei angegebenen Knoten liegen. Die Methode wird benutzt beim Löschen von Schleifen bzw. Verzweigungen.
getSurroundingBranch	Diese Methode bestimmt ggf. eine Verzweigung in der der angegebenen Knoten enthalten ist. Ansonsten liefert die Methoden einen Wert, der angibt das keine umgebende Verzweigung gefunden wurde.

Eine weitere wichtige Klasse für den Editor ist die Klasse `ModelMatrix`. Diese Klasse realisiert den ersten Schritt zum Aufbau der Bildschirmdarstellung eines Kontrollflußgraphen. In eine Instanz der Klasse `ModelMatrix` werden die Knotenname der Knoten des Graphen eingetragen, um festzulegen welche Positionen im Darstellungsgitter die einzelnen Knoten haben. Aufgrund dieser Einordnung werden später in der View die x- und y-Koordinaten berechnet. Der Ablauf des Algorithmus zum Einfügen der Knoten in das Darstellungsgitter wird in Abschnitt 4.2 beschrieben.

Die **wichtigsten** Methoden der Klasse `ModelMatrix` sind:

fillNodesAndEdges	In dieser Methode werden die Knoten und Kanten für die Speicherung der Knoten in einer Instanz der Klasse `MainMatrix` aufbereitet. Z.B. werden hier den Knoten die besonderen Typen für die Positionierung zugewiesen und die Schleifenkanten durch Kanten und Dummy-Knoten ersetzt.
fillMatrix	Diese Methode schreibt die „Basislinie“ des Workflow-Graphen in die Matrix. Trifft diese Methode auf eine Verzweigung, so wird die Methode `fillBranch` aufgerufen.
fillBranch	Die Methode `fillBranch` schreibt einen Verzweigungsast in die Matrix. Sie kann mehrmals rekursiv aufgerufen werden.

6.3.5 Die View

Die Aufgabe der View-Objekte ist es die Model-Objekte zu visualisieren. Die Klassen des Packages `WFEdit2.View` definieren die Zeichenfläche (Klasse `DrawPaneCanvas`), die Placemarkerbox (Klasse `PlaceMarkerBox`), die Knoten (Klasse `ViewNode`) und die Kanten (Klasse `ViewEdge`) zur Darstellung auf der Zeichenfläche. Die Datenstruktur, die im Hintergrund der Zeichenfläche liegt, ist eine Instanz der Klasse `ViewMatrix`. In dieser Klasse werden die einzelnen Knoten mit ihren x- und y-Koordinaten gespeichert. Mit Hilfe einer Instanz der Klasse `ViewMatrix` wird z.B. bestimmt, ob und welcher Knoten bei einem Mausklick aktiviert werden muß. In einer Instanz der Klasse `ViewMatrix` werden Knoten-Objekte, die auf der Klasse `ViewNode` basieren, gespeichert. Zusätzlich existiert für jede Kante ein Objekt der Klasse `ViewEdge`. Die Objekte der Klasse `ViewEdges` werden in einem Feld innerhalb der Klasse `WFEdit2Frame` gespeichert.

Die Klasse `DrawPaneCanvas` basiert auf der Klasse `java.awt.Canvas` und stellt eine Zeichenfläche zur Verfügung. Die Implementierung der Ereignisbehandlung innerhalb des Zeichenfeldes befindet sich in der Klasse `WFEdit2Frame`. Die **wichtigste** Methode der Klasse `DrawPaneCanvas` ist:

`paint`	Die Methode zeichnet alle Knoten, Kanten. Durch Aufrufe anderer Methoden der Klasse drawPaneCanvas werden Knoten bzw. Kanten markiert oder farbig dargestellt.

Die Klasse `ViewMatrix` verkörpert die hinter der Zeichenfläche verborgene Datenstruktur die die einzelnen Knoten enthält.

Die **wichtigsten** Methoden der Klasse `ViewMatrix` sind:

`findInViewMatrix`	Durch diese Methode, der die x- und y-Position eines Mausklicks übergeben wird, wird bestimmt welcher Knoten durch den Benutzer ausgewählt wurde.
`findInViewMatrix`	Dieser Methode muß ein Knotenname übergeben werden. Daraufhin wird eine Instanz der Klasse `ViewMatrix` nach diesem Knoten durchsucht.
`findPositionInViewMatrix`	Die Methode sucht einen Knoten in einer Instanz der Klasse `ViewMatrix` und gibt dessen x- und y-Koordinaten zurück.

Die Klasse `Placemarkerbox` basiert ebenso wie die Klasse `DrawPaneCanvas` auf der Klasse `java.awt.Canvas`. Die Realisierung der Ereignisbehandlung dieses Fensters befindet sich in der Klasse `WFEdit2Frame`. Die wichtigste Methode ist auch hier:

`paint`	In dieser Methode wird der Graph verkleinert und nur mit Kontrollkanten auf der Zeichenfläche der Placemarkerbox dargestellt. Daneben wird noch die Markierung für den sichtbaren Teil des Graphen erzeugt.

6.3.6 Das Package DataAccess

Neben der Einteilung der Klassen des Programms in Packages entsprechend der MVC-Architektur zeigte es sich, daß es sinnvoll ist, den Zugriff auf eine wfv-Datei bzw. auf eine Datenbank in einem separaten Package zu realisieren. Dies erhöht zusätzlich die Übersichtlichkeit des Quelltextes der Applikation.

Das Package `DataAccess` enthält zwei wichtige Klassen. Dies sind die Klassen `DataAccess` und `ProcessTemplateOnDisk`.

Die Klasse `DataAccess` implementiert das Lesen bzw. Schreiben in eine Datenbank oder in eine wfv-Datei. Eine Hauptaufgabe der Klasse `DataAccess` ist es, den Parser für das Lesen aus einer wfv-Datei zu realisieren. Dieser Parser erkennt die Struktur der gewählten wfv-Datei und ermöglicht so die Speicherung des Inhalts der Datei im Programm. Daneben erkennt der Parser schwerwiegende Fehler innerhalb der Struktur der wfv-Datei und meldet ggf. einen Fehler[1].

Die Methoden für den Zugriff auf eine Datenbank führen diesen Zugriff über Funktionen eines Workflow-API durch. Dieses API wurde durch Mitarbeiter der Abteilung DBIS größtenteils verwirklicht und stand für die Implementierung des Workflow-Editors zur Verfügung.

Die **wichtigsten** Methoden der Klasse `DataAccess` sind:

`readFromFile`	Die Aufgabe der Methode ist das Lesen einer Prozeßbeschreibung aus einer Datei. Innerhalb dieser Methode wird der Parser für eine wfv-Datei realisiert.
`saveToFile`	Diese Methode implementiert die Speicherung einer Prozeßvorlage in einer Datei. Dabei wird die Struktur einer wfv-Datei erzeugt.
readFromDB	Die Methode liest eine Prozeßvorlage in einer Datenbank.
saveToDB	Die Aufgabe dieser Methode ist die Speicherung einer Prozeßvorlage in einer Datenbank.

Der durch den Parser als relevant erkannte Inhalt einer wfv-Datei wird in einer Instanz der Klasse `ProcessTemplateOnDisk` gespeichert. Ein Objekt dieser Klasse ist ein genaues Abbild der vom Benutzer gewählten wfv-Datei. Die einzelnen Attribute der Klasse `ProcessTemplateOnDisk` entsprechen genau den einzelnen Sektionen innerhalb einer wfv-Datei.

Die Attribute der Klasse `ProcessTemplateOnDisk` sind:

- Name, Beschreibung, Kategorie der Prozeßvorlage

[1] Der in dieser Arbeit verwendete Parser fängt nicht alle möglichen Fehler in einer wfv-Datei ab, da dessen Realisierung innerhalb dieser Arbeit zu aufwendig gewesen wäre.

- Aktivitätenvorlagen der Prozeßvorlage. Innerhalb einer Aktivitätenvorlage sind alle Knoten des Workflow-Graphen gespeichert, die auf dieser Aktivitätenvorlage basieren.
- Kanten
- Datenslots

Die Aktivitätenvorlagen, die Kanten und die Datenslots einer Prozeßvorlage werden jeweils in einem Feld gespeichert. Die einzelnen Inhalte der Feldpositionen werden realisiert durch Klassen, die ebenfalls im Package `WFEdit2.DataAccess` implementiert sind.

Aktivitätenvorlagen basieren auf der Klasse `Acitivty`. Die Klasse `Activity` ist eine Subklasse der im Package `WFEdit2.Model` verwendeten Klasse `AcitivtyCore`. Der Unterschied zwischen den beiden Klasse besteht darin, daß die Klasse `Acitivty` zusätzlich alle Knoten speichern kann, die auf der jeweiligen Aktivitätenvorlage basieren. Kanten einer Prozeßvorlage werden durch die Klassen `Edge` und `EndNode` dargestellt. Der Aufbau der Klasse `Edge` orientiert sich stark an der entsprechenden Sektion innerhalb einer wfv-Datei. In einer wfv-Datei werden für einen Knoten alle ausgehenden Kanten in einer Zeile beschrieben. Analog dazu speichert die Klasse `Edge` für einen bestimmten Knoten alle Endknoten der ausgehenden Kanten in einem Feld. Die Klasse `EndNode` enthält zusätzlich zum Namen des Endknotens noch weitere Attribute wie den Typ der Kante und optional einen Entscheidungswert bzw. die minimale und maximale Dauer zwischen Ende und Start zweier durch die Kante verbundenen Aktivitäten. Die in einer Prozeßvorlage vorhandene Datenslots durch die Klasse `Dataslot` beschrieben.

6.4 Die Erweiterbarkeit des Editors

Da das ADEPT-Basismodell auch heute noch Gegenstand der Forschung ist, ist es denkbar, daß in der Zukunft weiterer Elemente dem ADEPT-Basismodell hinzugefügt werden. Aus diesem Grund muß die Erweiterbarkeit des Workflow-Editors betrachtet werden.

Denkbar ist z.B., daß neue Attribute den Aktivitätenvorlagen hinzugefügt werden. Dazu müssen die Klassen `Acitivty` und `ActivityCore` um weitere Attribute erweitert werden. Methoden dieser Klassen müssen eventuell für die Verarbeitung der neuen Attribute angepaßt werden. Diese Erweiterung der Klassen hat zur Folge, daß die Zugriffsmethoden auf eine wfv-Datei und die Datenbank ebenfalls geändert werden müssen. Dabei wird vorausgesetzt, daß die Struktur der wfv-Datei und die Tabellen der Datenbank an diese neuen Attribute angepaßt worden sind. Damit die neuen Attribute angezeigt und bearbeitet werden können, muß die Oberfläche um entsprechende Anzeigeelemente für diese Attribute erweitert werden.

Das Hinzufügen eines neuen Kantenattributs erfolgt durch Änderungen an den Klassen `EndNode` und `EdgeModel`. Eine Änderung der Klasse `ViewEdge` ist nicht notwendig, da `ViewEdge` die Attribute einer Kante aus der Klasse `EdgeModel` erbt. Diese Erweiterung der Klassen für Kanten führt wiederum zu Änderungen an der Datenzugriffsschicht, an der wfv-Datei und an der Datenbank. Außerdem muß auch die Oberfläche für die Bearbeitung des neuen Kantenattributs erweitert werden.

Die Definition einer neuen Kantenart erfordert keine Änderungen der Klassen die Kanten beschreiben. Es müssen allerdings die Oberflächenelemente für die Verarbeitung dieser neue Kantenart erweitert werden und ggf. müssen weitere Korrektheitstest implementiert werden bzw. bestehende Test müssen angepaßt werden. Um die neue Kantenart auf dem Bildschirm darzustellen, muß die Klasse

`DrawPaneCanvas` um Funktionen erweitert werden, die die neue Kantenart zeichnen und ggf. markiert darstellen.

Eine Erweiterung, wie z.B. die Hinzunahme eines weiteren Aktivitätentyps, ist leichter zu bewerkstelligen, da die Oberfläche nicht verändert werden muß. Ein neuer Typ kann in den vorhandenen Oberflächenelementen für die Anzeige der Typen eines Attributs dargestellt werden. Allerdings muß ggf. die Applikationslogik erweitert werden, um die Korrektheit eines Prozeßgraphen weiterhin garantieren zu können.

Auf komplexere Erweiterungen des Editors wird im Ausblick (Abschnitt 7.2) eingegangen.

7. Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde ein graphischer Workflow-Editor für das ADEPT-Basismodell konzipiert und prototypisch implementiert. Dabei hat sich gezeigt, daß es auch mit der Programmiersprache Java möglich ist, einen komfortablen und visuell ansprechenden Editor zu realisieren. Leider ist Java infolge der Interpretation des Codes durch die virtuelle Maschine immer noch langsamer als andere compilierte Programmiersprachen. Die fehlende Leistung macht sich vor allem bei der Interaktion mit der Oberfläche des Programms bemerkbar.

Die Idee für diese Diplomarbeit stammt aus der Forderung nach einer schnellen Möglichkeit, Workflow-Schemata modellieren zu können. Bisher mußten diese Schemata, die durch eine bereits vorhandene Workflow-Engine ausgeführt werden sollen, direkt in das ADEPT-Repository eingetragen werden. Dieses Vorgehen ist natürlich zeitraubend und fehleranfällig. Daneben ist der Workflow-Editor auch das Bindeglied zwischen der Speicherung einer Prozeßdefinition in einer Datenbank und der Speicherung in einer Datei. Der Editor unterstützt beides und ermöglicht so die Modellierung eines Workflows auch ohne Datenbank. Der Vorteil dieser Eigenschaft ist, daß die Modellierung fast überall stattfinden kann. Außerdem ermöglicht diese Eigenschaft, daß andere Werkzeuge, wie z.B. Geschäftsprozeßmodellierungs-Tools, entsprechende Dateien generieren und diese dann in den Editor übernommen und weiterentwickelt werden können.

Ein wesentliches Ziel bei der Realisierung des Workflow-Editors war es, bereits bei der Modellbildung, d.h. noch vor Inbetriebnahme des Systems oder idealerweise sogar vor der Implementierung der eigentlichen Anwendungskomponenten, die Struktur und das Verhalten von Workflows studieren und validieren zu können. Dabei soll der Editor bestimmte Fehler im Kontroll- und Datenfluß schon bei der Modellierung verhindern oder bei Fertigstellung einer Prozeßdefinition mit Hilfe von Tests erkennen und anzeigen. Aufgrund der syntaxgesteuerten Modellierung und der Blockstrukturierung besitzt der Editor diese Fähigkeit. Ein Beispiel für die Verhinderung von Fehlern ist, daß bei jeder Kontrollflußaufsplittung automatisch auch eine Zusammenführung generiert wird. Neben dieser Eigenschaft besitzt der Editor noch weitere Modellierungseigenschaften, die die Korrektheit des Kontrollflusses (z.B. Zyklenfreiheit bzgl. bestimmter Kantentypen) sicherstellen. Die Korrektheit des Datenflusses wird vom Editor ebenfalls gewährleistet. Beispielsweise können Datenflußschemata auf Vollständigkeit geprüft werden, d.h. es kann per Analyse festgestellt werden, ob alle obligaten Eingabeparameter einer beliebigen Aktivität zur Laufzeit sicher versorgt werden können. Um die verschiedenen Korrektheitseigenschaften prüfen zu können, wurden verschiedene Algorithmen implementiert. Dabei konnte auf Erkenntnisse aus dem Bereich der Graphentheorie zurückgegriffen werden. Die dabei gefundenen Algorithmen mußten jedoch den speziellen Anforderungen des Editors angepaßt werden.

Ein weiterer aufwendiger Punkt war die Erstellung der Oberfläche des Programms. Diese Entwicklungsarbeit dauerte länger als vorgesehen, da auch der Prototyp eine für den Benutzer konsistente Oberfläche besitzen sollte. Die Schwierigkeit lag in der Vielzahl an Möglichkeiten zur Manipulation der Attribute von Aktivitäten, Knoten, Kanten, usw. Auch die Entwicklung des Algorithmus, der für die visuelle Darstellung von Workflow-Graphen benötigt wird, nahm viel Zeit in Anspruch.

Die Entwicklungsumgebung Borland JBuilder2 zeigte mit zunehmender Größe des Projekts Schwächen. So dauerte der Aufbau des Hauptfenster in der Designanzeige des JBuider2 sehr lange und aus diesem Grund mußte das Hinzufügen von visuellen Elementen in das Hauptfenster per Hand im Java-Code ausgeführt werden. Trotz dieses Nachteils ist der JBuilder2 zweifellos ein guter Ansatz, der die Java-Programmierung erleichtert, indem er den Benutzer von vielen Routineaufgaben entlastet. Leider kann der JBuilder2 (noch) nicht mit der Qualität einer Entwicklungsumgebung wie Delphi mithalten.

7.2 Ausblick

Ein graphischer Workflow-Editor ist für jedes WfMS fast schon eine Notwendigkeit, da der Benutzer durch die graphischen Beschreibungsmöglichkeiten des Editors viele Fehler bei der Prozeßmodellierung vermeiden kann. Diese Vorteile ermöglichen die schnelle Änderung von Prozeßdefinitionen, um ggf. auf Veränderungen der Umwelt reagieren zu können. Da WfMS meist in der Industrie eingesetzt werden, können solche Veränderungen z.B. notwendige Effizienzsteigerungen in den Abläufen des Unternehmens sein, die durchgeführt werden müssen, um konkurrenzfähig zu bleiben.

Da die Implementierung des Editors in dieser Arbeit ein Prototyp ist, sind verschiedene Erweiterungen denkbar, die der Editor für eine möglichen kommerziellen Einsatz benötigen würde.

Folgende Erweiterungen des Workflow-Editors wären wünschenswert:

- **Subworkflows:**

Eine notwendige Erweiterung ist die Möglichkeit zur Einbindung von Subworkflows in einen neuen Workflow. Dies würde die Verfeinerung eines Workflows ermöglichen und so die Übersichtlichkeit erhöhen. Ein weiterer Vorteil wäre auch die Wiederverwendbarkeit von schon definierten Workflows. Um Subworkflows zu nutzen, muß garantiert werden, daß alle Elemente einer Prozeßbeschreibung eine global eindeutige ID besitzen.

- **Blöcke:**

Blöcke bestehen aus mehreren Aktivitäten und gehören fest zu einer einzigen Prozeßvorlage. Der Editor sollte bei der Blockerzeugung eine top-down- und eine bottom-up-Vorgehensweise unterstützen. Top-down bedeutet, daß ein Block in den Prozeßgraphen eingefügt werden kann und später mit Aktivitäten ausgestaltet wird. Bottom-up erlaubt das Markieren mehrerer Knoten, die dadurch zu einem Block zusammengefaßt werden. Bei der bottom-up-Vorgehensweise muß der Editor den Modellierer unterstützen, da ein Block nur einen Eingangsknoten und einen Ausgangsknoten haben darf. Verletzt der Modellierer diese Bedingung muß ein Fehler angezeigt werden. Auch stellt sich bei der Implementierung die Frage nach der Schachtelungstiefe. Eine Schachtelung über mehrere Ebenen ist schwerer zu implementieren als eine Schachtelungstiefe, die es erzwingt, daß ein Block nur noch Aktivitäten enthält und keine weiteren Blöcke.

Bei der vorher beschriebenen Art der Implementierung kann ein Block nur einmal in einer Prozeßvorlage benutzt werden. Eine andere Definitionsmöglichkeit von Blöcken wäre eine Art „Blockvorlage“, die mehrmals in einem Prozeß verwendet werden könnte. Hier muß jedoch beachtet werden, daß es bei einer Schachtelungstiefe > 1 zu einer zyklischen Blockdefinition kommen könnte. D.h. die „Blockvorlage“ A enthält den Block B. B ist aber so definiert, daß er wiederum Block A enthält. Dadurch würde es beim Einfügen des Blocks A in eine Prozeßvorlage zu einer unendlichen

Schachtelung von Blöcken kommen. Aufgabe des Editors wäre es, rekursive Aufrufstrukturen, die für die Modellierung realer Arbeitsabläufe ohnehin nicht benötigt werden, zu erkennen und auszuschließen.

- **Animation:**

Eine weitere wünschenswerte Komponente wäre die Animation des Kontroll- bzw. Datenflusses, die die Ausführung einer Prozeßdefinition simuliert, ohne eine Instanz des Prozesses zu erzeugen. Dadurch ließen sich Fehler in der Ablauflogik des Prozesses schon bei der Modellierung erkennen. Dabei muß entschieden werden, ob neben dem Kontrollfluß auch der Datenfluß animiert werden soll.

Bei der Animation des Kontroll- und Datenflusses muß der Modellierer bei allen Ausgabeparametern der Aktivität Werte angeben, da die Aktivitäten bei der Animation nicht ausgeführt werden können. Bei der Animation nur des Kontrollflusses entfällt die Angabe von Werten. Allerdings muß der Modellierer bei bedingten Verzweigungen und parallelen Verzweigungen mit finaler Auswahl bestimmen, welcher Zweig benutzt wird. Der Aufwand, um eine Simulation des Kontroll- und Datenflusses zu implementieren, ist sicher höher als der Programmieraufwand für die alleinige Simulation des Kontrollflusses.

- **Aufbauorganisation:**

Um ein vollständiges Modellierungswerkzeug darzustellen, benötigt der Editor noch eine Komponente zur Spezifikation einer Organisation. Diese Komponente ist notwendig, um den Aktivitäten Bearbeiter zuzuordnen. Dabei müßte der Editor ebenfalls um einen Dialog erweitert werden, der eine komfortable Eingabe von Bedingungen zur Auswahl eines Bearbeiters bietet, da zum jetzigen Zeitpunkt im Editor nur ein Eingabefeld für die Spezifikation einer Rolle vorhanden ist. Die Verwaltung der Aufbauorganisation wird zur Zeit im Workflow-Praktikum 98/99 der Abteilung DBIS implementiert.

- **Prüfung temporale Aspekte:**

Im Workflow-Editor ist bis jetzt nur die Eingabe von Zeitwerten implementiert. Es wird noch keine Korrektheitsprüfung vorgenommen. Eine Überprüfung der Zeitwerte ist aber auf jeden Fall notwendig, da die Zeitangaben ein wichtiges Kriterium für die Korrektheit eines Workflow-Schema sind. Um Prüfungen temporaler Aspekte zu implementieren, kann auf die Erkenntnisse und Verfahren aus [Gri97] zurückgegriffen werden.

- **Late Modeling:**

Eine denkbare Erweiterung des Editors wäre auch die Möglichkeit, Prozeßfragmente zu modellieren, die erst zur Laufzeit mit Aktivitäten ausgefüllt werden. Dieses Vorgehen ist vor allem im Bereich evolutionärer Workflows von Wichtigkeit.

Neben den Erweiterungen des Workflow-Editors gibt es noch Veränderungen, die das WfMS als ganzes betreffen. So könnte die relationale Datenbank durch eine objektorientierte Datenbank ersetzt werden. Dies würde die Umsetzung Java-Klassen der Workflowbeschreibung in Tabellen der relationalen Datenbank unnötig machen. Eine weitere Veränderung betrifft die Kommunikation zwischen dem Editor und dem Workflow-Server. Der Editor wird zur Zeit noch per RMI[1] [Fla98] mit dem Workflow-Server verbunden. Mit Hilfe von RMI lassen sich nur auf Java basierende Client/Server-Systeme realisieren. Die Verbindung zwischen den Clients, wie dem Workflow-Editor, und dem Workflow-Server ließe sich auch über CORBA[2] [OMG92] realisieren. Dies würde neben der von Java ermöglichten Plattformunabhängigkeit auch die Möglichkeit eröffnen, Programme, die mit unterschiedlichen Programmiersprachen erstellt wurden, zusammenarbeiten zu lassen.

[1] RMI = Remote Method Invocation
[2] CORBA = Common Object Request Broker Architecture

Abschließend ist zu sagen, daß die Modellierung von Workflows in der Zukunft immer wichtiger wird, da die Workflow-Technologie immer weiter vordringen wird (z.B. in den medizinischen Bereich). Da dann immer mehr „Nicht-Experten“ mit der Problemstellung der Spezifikation eines Workflows konfrontiert sind, muß die Modellierung von Prozessen schnell und einfach zu erledigen sein. Diesen Vorteil haben graphische Modellierungswerkzeuge sicherlich, doch müssen die Workflow-Editoren auch die Korrektheit eines modellierten Workflow-Schemata garantieren. Dadurch läßt sich die erste Hürde beim Einsatz eines WfMS von den Verantwortlichen leichter nehmen und das WfMS wird nicht als Bedrohung betrachtet.

Anhang

A Dateiformat der Ablaufvorlagen (.wfv)*

Das ursprüngliche Dateiformat der Ablaufvorlagen wurde in [Hen97] festgelegt. Im Rahmen des Workflow-Praktikums im Wintersemester 97/98 wurde des Format um weitere Attribute ergänzt. Für diese Arbeit mußte es nochmals erweitert werden, da der Editor mehr Attribute spezifizieren kann, als in der früheren Versionen des Dateiformats vorgesehen waren.

Nachfolgend werden die verschiedenen Erweiterungen kurz erläutert:

- **Abschnitt [ACTIVITIES]**

Die Attribute InParameter und OutParameter wurden um einen Datenslotnamen erweitert. Dies ist notwendig, da nun Parameter mit Datenslots verbunden werden können. Ein OutParameter-Eintrag in einer Datei kann nun folgendermaßen aussehen:

☞ **OutParameter** = Patientenname, STRING, OBLIGATE, NOEXTRADEMAND, NameDS

In diesem Fall wird der Ausgabeparameter Patientenname auf den Datenslot NameDS abgebildet. Der Workflow-Editor ist jedoch weiterhin in der Lage InParameter und OutParameter, die nur aus vier Attributen bestehen, zu lesen.

Die nächste Veränderung betrifft die Attribute FAZ, SAZ, FEZ, SEZ. Sie werden nun nicht mehr einem Knoten zugeordnet, sondern einer Aktivitätenvorlage. Dies entspricht auch der Zuordnung in der Datenbank. Aus diesem Grund wurde das Attribut Timestamps eingeführt. Ein Beispiel für Timestamps ist:

☞ **Timestamps** = **FAZ:** 1998-03-05 17:20:10

In diesem Beispiel wird der früheste Anfangszeitpunkt (FAZ) auf den 03.05.1998 17:20:10 festgelegt. Daneben sind keine weitere Zeitangaben (SAZ, FEZ, SEZ) definiert worden. Für das Attribut Timestamps einer Aktivität können entweder alle oder nur ein Teil der Menge der Zeitwerte angegeben werden. Sind keine Zeitstempel für definiert, so fehlt das Attribut Timestamps in der Aktivitätenvorlage.

Dem Instance-Tag können nun die Attribute loopPara und errorPara zugeordnet werden. Das Attibut loopPara gibt einen obligaten Integer-Ausgabeparameter an, aufgrund dessen Wert entschieden wird, ob eine Schleife beendet oder weitergeführt wird. Dieses Attribut können nur Knoten besitzen, deren Aktivitätenvorlage vom Typ LOOP ist. Eine Instance-Zeile einer Aktivitätenvorlage kann folgendes Aussehen besitzen:

☞ **Instance** = EMP03, Schleifenentscheidung, **loopPara:** LOOP

Knoten, die ausgehende Fehlerkanten besitzen, wird das Attribut errorPara zugeordnet. Es enthält den Namen eines obligaten Integer-Ausgabeparameters und ermöglicht über den Wert des Parameters die Auswahl einer Fehlerkante. Die Anordnung des Attributs errorPara ist ähnlich dem Attribut loopPara und stellt sich wie folgt dar:

☞ **Instance** = UNT01, Untersuchung durchführen, **errorPara:** Untersuchungsfehlerwert;

- **Abschnitt [DATASLOTS]**

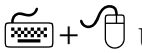
Dieser Abschnitt spezifiziert alle Datenslots einer Prozeßvorlage. Der Editor kann Vorlagen mit und ohne diese Sektion lesen. Ein Eintrag in diesen Abschnitt hat folgendes Aussehen:

☞ KurzbefundDS, STRING

An ersten Stelle befindet sich in der Zeile der Name des Datenslots. Darauffolgend wird der Typ des Datenslots angegeben. Wie bei den Parametern einer Aktivität sind hier die Typen INTEGER, STRING und REFERENCE erlaubt.

B Die Kurzbedienungsanleitung des Editors

☞ *Kontrollflußfenster:*

 Rechtsklick auf einen Knoten	Markieren eines Knotens.
 Linksklick auf einen Knoten	Aufruf eines Pop-up-Menüs zum Löschen eines Knotens bzw. zum Hinzufügen eines Verzweigungsastes.
 Rechtsklick auf einen „Hotspot“	Aufruf eines Pop-up-Menüs zum Einfügen von Konstrukten, zum Löschen von Kanten und zur Anzeige der Kanteneigenschaften.
 Linksklick auf eine Kante (außer Kontrollkanten)	Markieren des Verlaufs einer Kante.
 Umschalttaste und Mausklick	Auswahl des Start- bzw. Endknotens einer einzufügenden Kante oder eines nachträglich einzufügenden Konstruktes.

☞ *Datenflußfenster:*

Einfachklick auf eine Zeile	Markiert die Zeile.
Doppelklick mit der *linken* Maustaste auf das Datenslot-Feld einer Zeile	Erstellen/Ändern einer Verbindung zwischen einem Parameter einer Aktivität und einem Datenslot.
Doppelklick mit der *rechten* Maustaste auf das Datenslot-Feld einer Zeile	Anzeige aller Knoten, die einen Datenslot schreiben bzw. lesen. **Farben:** Grün = lesen Blau = schreiben Magenta = lesen und schreiben

☞ *Placemarkerbox:*

Klick mit der *linken* Maustaste	Festlegen, welcher Teilgraph im Kontrollfluß-fenster angezeigt wird.

Literaturverzeichnis

[AAH98] **N. Adam, V. Atluri, W. Huang**
„Modeling and Analysis of Workflows Using Petri Nets“
In: *Journal of Intelligent Inf. Systems, Vol. 10, No. 2, S. 131-158, März/April 1998*

[Bau90] **B. Baumgarten**
„Petri-Netze“
BI-Wiss.-Verl., 1990

[BES96] **B. Biechele, D. Ernst, J. Schmid, W. Schulte, F. Houdek**
„Erfahrungen bei der Modellierung eingebetteter Systeme mit verschiedenen SA/RT-Ansätzen“
Ulmer Informatik-Bericht, Nr. 96-09, Dezember 1996

[Bla96] **R. Blaser**
„Konfiguration verteilter Anwendungen aus vorgefertigten Programmbausteinen“
Diplomarbeit, Universität Ulm, 1996

[Bur92] **S. Burbeck**
„Applications Programming in Smalltalk-80: How to use the Model-View-Controller (MVC)“
http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[CKO92] **B. Curtis, M. I. Kellner, J. Over**
„Process Modeling“
In: *Communications of the ACM, Vol. 35, No. 9, S. 75-90, September 1992*

[CLR90] **T. H. Cormen, C. E. Leiserson, R. L. Rivest**
„Introductions to Algorithms“
MIT Press, 1990

[DaR97] **P. Dadam, M. Reichert**
„ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows without loosing Control“
In: *Journal of Intelligent Information Systems, Special Issue on Workflow and Process Management*

[DGS94] **G. Dinkhoff, V. Gruhn, A. Saalmann, M. Zielonka**
„Business Process Modeling in the Workflow Management Environment Leu“
In: *Lecture Notes in Computer Science 881, S. 46-63*
Springer, 1994

[DKR95] **P. Dadam, K. Kuhn, M. Reichert, T. Beuter, M. Nathe**
„ADEPT: Ein integrierter Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Anwendungssysteme in klinischen Anwendungsumgebungen“
In: *F. Huber-Wäschle, H. Schauer, P. Widmayer (Hrsg.): Proc. 25. GI-Jahrestagung und 13. Schweizer Informatikertag, Zürich, S. 677-686*
Springer, 1995

[EOO94] **E. Eberleh, H. Oberquelle, R. Oppermann**
„Einführung in die Software-Ergonomie“
De Gruyter, 1994

[FäZ88] **K.-P. Fähnrich, J.E. Ziegler**
„Direct Manipulation“
In: *M. Helander (ed.): Handbook of Human Computer Interaction, S. 123-133*
Elsevier Science Publishers B. V. (North-Holland), 1988

[Fla98] **D. Flanagan**
„Java in a Nutshell“
O’Reilly, 1998

[FNP93] **Frances Newbery Paulisch**
„The Design of an Extendible Graph Editor“
Lecture Notes in Computer Science 704, Springer, 1993

[FrS97] **S. Frank, B. Schultheiß**
„Prozeßmodellierung und -steuerung mit WorkParty“
Interner Ulmer Informatik-Bericht, Abteilung DBIS, April 1997

[Gri97] **M. Grimm**
„ADEPT-TIME: Temporale Aspekte in flexiblen Workflow-Management-Systemen“
Diplomarbeit, Universität Ulm, 1997

[Gru93] **V. Gruhn**
„Entwicklung von Informationssystemen in der LION-Entwicklungsumgebung“
In: *G. Scheschonk, W. Reisig (Hrsg.): Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen, S. 31-45*
Springer, 1993

[Gru96] **V. Gruhn**
„Geschäftsprozeß-Management als Grundlage der Software-Entwicklung“
In: *Informatik - Forschung und Entwicklung (1996) 11, S. 94-101*
Springer, 1996

[Hen97] **C. Hensinger**
„ADEPTflex-Dynamische Modifikation von Workflows und Ausnahmebehandlung in WfMS“
Diplomarbeit, Universität Ulm, 1997

[Hob95] **J. Hobart**
„Principles of Good GUI Design“
In: *UNIX Review September 1995*

[Jab95] **S. Jablonski**
„Workflow-Management-Systeme: Modellierung und Architektur“
Thomson Publ. 1995

[Jab95a] **S. Jablonski**
„Workflow-Management-Systeme: Motiviation, Modellierung, Architektur“
In: *Informatik-Spektrum 18 (1995), S. 13-24*
Springer, 1995

[JBS97] **S. Jablonski, M. Böhm, W.Schulze**
„Workflow-Management: Entwicklung von Anwendungen und Systemen“
dpunkt-Verlag, 1997

[Kir96] **M. Kirsch**
„Realisierung einer Entwicklungsumgebung für die Modellierung und Animation flexibler Workflows“
Diplomarbeit, Universität Ulm, 1996

[LeA94] **F. Leymann, W. Altenhuber**
„Managing Business Processes as an Information Ressource“
In: *IBM Systems Journal, Vol. 33, No. 2, 1994, S. 326-348*

[Maa93] **S. Maaß**
„Software-Ergonomie“
In: *Informatik-Spektrum 16 (1993), S. 191-205*
Springer, 1993

[MoB95] **L. Mohan, J. Byrne**
„Designing Intuitive Icons and Toolbars“
In: *UNIX Review September 1995*

[MoN90] **R. Molich, J. Nielsen**
„Improving a Human-Computer Dialog“
In: *Communications of the ACM, Vol. 33, No. 3, S. 338-348, März 1990*

[NeM93] **K. Neumann, M. Morlock**
„Operations Research“
Carl Hanser, 1993

[Obe96] **A. Oberweiß**
„Modellierung und Ausführung von Workflows mit Petri-Netzen“
Teubner, 1996

[OMG92] **Object Management Group**
„The Common Object Request Broker Architecture and Specification“
Object Management Group (OMG), Framingham, MA, 1992

[PaR88] **K. R. Paap, R. J. Roske-Hofstrand**
„Design of Menus“
In: *M. Helander (ed.): Handbook of Human Computer Interaction*
Elsevier Science Publishers B. V. (North-Holland), 1988

[Rei93] **B. Reinwald**
„Workflow-Management in verteilten Systemen“
Teubner, 1993

[ReM96] **B. Reinwald, C. Mohan**
„Structured Workflow Management with Lotus Notes Release 4“
Proc. 41th IEEE Computer Society Int’l Conf. (CompCon96), Santa Clara, CA, S. 451-457, Februar 1996

[ReW92] **B. Reinwald, H. Wedekind**
„Integrierte Aktivitäten- und Datenverwaltung zur systemgestützten Kontroll- und Datenflußsteuerung“
In: *Informatik in Forschung und Entwicklung (1992) 7, S. 73-82*
Springer, 1992

[RoW91] **B. Rosenstengel, U. Winand**
„Petri-Netze“
Vieweg 1991

[Sch96] **A. W. Scheer**
„ARIS-Toolset: Vom Forschungsprototypen zum Produkt“
In: *Informatik-Spektrum 19 (1996), S.71-78*

[Sch98] **A. W. Scheer**
„ARIS - Modellierungsmethoden, Metamodelle, Anwendungen“
Springer, 1998

[Sed92] **R. Sedgewick**
„Algorithmen“
Addison-Wesley, 1993

[Shn82] **B. Shneiderman**
„Designing Computer System Messages“
In: *Communications of the ACM, Vol. 25, No.9, September 1982*

[Shn84] **B. Shneiderman**
„Response Time and Display Rate in Human Performance with Computers“
In: *Computing Surveys, Vol. 16, No. 3, September 1984*

[Shn92] **B. Shneiderman**
„Designing the User Interface“
Addison-Wesley, 1992

[Shö96] **U. Schöning**
„Vorlesungsskript Algorithmen“
Universität Ulm 1996

[SMM95] **B. Schultheiß, J. Meyer, R. Mangold, T. Zemmler, M. Reichert**
„Prozeßentwurf für den Ablauf einer stationären Chemotherapie“
Interner Ulmer Informatik-Bericht, Abteilung DBIS, 22. November 1995

[STT81] **K. Sugiyama, S. Tagawa, M. Toda**
„Automatic graph drawing and readability of diagrams“
In: *IEEE Transactions on Systems, Man and Cybernetics, SMC-11(2), Februar 1981, S. 109-125*

[Sun98] **Sun Microsystems**
„Creating a GUI with JFC/Swing“
http://www.javasoft.com/docs/books/tutorial/ui/index.html

[TBB89] **R. Tamassia, G. Di Battista, C. Batini**
„Automatic graph drawing and readability of diagrams“
In: *IEEE Transactions on Systems, Man, Cybernetics, SMC-18(1), Januar/Februar 1989, S. 61-79*

[Ver98] **G. Versteegen**
„Fit mit UML - Geschäftsabläufe objektorientiert optimiert“
In: *iX – Magazin für professionelle Informationstechnik 12/98, S.143-147*
Heise, 1998

[VMJ97] **P. Vilela, J. Maldonado, M.Jino**
„Program Graph Visualization“
In: *Software-Practice and Experience, Vol.27(11) (November 1997), S. 1245-1262*
John Wiley & Sons, Ltd., 1997

[Wäc95] **H. Wächter et al.**
„Modellierung und Ausführung flexibler Geschäftsprozesse mit SAP Business Workflow 3.0“
In: *Proc. 25. GI-Jahrestagung und 13. Schweizer Informatikertag (GIST’95), Zürich, September 1995 (Informatik-Aktuell, Springer), S. 197-204*

[Wei97] **P. Weilbach**
„Implementierungsaspekte zur Verwaltung und Synchronisation dynamischer Änderungen in prozeßorientierten Workflow-Management-Systemen“
Diplomarbeit, Universität Ulm, 1997

[WMC96] **Workflow Management Coalition**
„Workflow Management Coalition Terminology & Glossary“
WfMC, Brüssel, Belgien, 1996

Erklärung

Name: Ralf Michel **Matr. Nr.: 0279935**

Ich erkläre, daß ich die Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidenheim, den 18.01.1999 **.....**

Unterschrift