Plug&Play-Aspekte und Integration zustandsbehafteter Daten in Prozess-Management-Systeme

Diplomarbeit

Diplomand:	Kevin Göser
Fachbereich:	Informatik
Fachrichtung:	Medieninformatik
Betreuer:	Prof. Dr. Peter Dadam, Hilmar Acker
Korrektoren:	Prof. Dr. Peter Dadam, Dr. Manfred Reichert
Abgabedatum:	30. September 2005

Danksagung

An erster Stelle möchte ich meinen Betreuern Prof. Dr. Peter Dadam und Hilmar Acker für die enge und fruchtbare Zusammenarbeit danken. Sie haben sich stets viel Zeit für mich genommen und waren auch zu langen Diskussionen bereit.

Hilmar Acker danke ich besonders dafür, dass er stets ein offenes Ohr für meine Fragen und Anliegen hatte.

Mein besonderer Dank geht an meine Freundin, Linh Thao Ly, deren Liebe und Unterstützung mir während der gesamten Zeit Rückhalt geboten haben.

Kevin Göser

Ulm, den 30.09.2005

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufgabenstellung 2

1.2 Aufbau der Arbeit 3

2 Relevante Grundlagen **5**

2.1 Workflow-Management-Systeme 5

2.2 ADEPT 8

2.2.1 Überblick über ADEPT 8

2.2.2 Kontroll- und Datenfluss 9

2.2.3 Ausführungszustände von Aktivitäten 11

2.2.4 Konzepte zur Unterstützung von Adaptivität 11

2.2.5 Definitionen und Notationen 12

2.3 Komponententechnologie 12

3 Plug & Play **15**

3.1 Plug & Play und Softwarekomponenten 16

3.2 Plug & Play mit Komponenten in Workflow-Management-Systemen 17

3.3 Abhängigkeiten zwischen Aktivitäten 19

3.4 Beispiele 24

3.5 Weitere Aspekte und Anforderungen 26

3.6 Integration versteckter Datenflüsse 27

3.7 Semantische und funktionale Aspekte 29

3.8 Zusammenfassung und weitere Inhalte der Diplomarbeit 30

4 Zustandsbehaftete Daten **33**

4.1 Anforderungen 33

4.2 Ein erster Lösungsansatz 34

4.3 Bewertung und weitere Anforderungen 37

5 Zustandsbeschreibungen mittels Zustandsautomaten **41**
5.1 Definition des Zustandsgraphen 42
5.2 Erweiterung der Parameter von Aktivitäten 44
5.3 Beispiele zur Demonstration des Konzepts 47
5.4 Ansätze zur Sicherstellung der Zustandsforderungen 48
5.4.1 Anforderungen an die Validierung der Zustandsforderungen 51
5.4.2 Algorithmus für synchronisationsfreie Prozesse 51
5.4.3 Synchronisation 54
5.4.4 Lösungsansätze für Prozesse mit Synchronisation 59
5.5 Diskussion und Ausblick 61
5.5.1 Parallele Zustandswechsel 62
5.5.2 Verarbeitungswege 63
5.5.3 Allgemeine Zustandsobjekte 64
5.5.4 Globale Zustände 65
5.5.5 Mehrere Zustände 65
5.5.6 Bezugspunkt für Zustände 67
5.6 Zusammenfassung 68

6 Ausblick **69**

A Algorithmen **75**

Kapitel 1

Einleitung

Bemühungen, Geschäftsprozesse und beteiligte Anwendungen durch Informationssysteme zu unterstützen, stießen in den letzten Jahren auf reges Interesse [5]. Flexibilität und Adaptivität der Prozess- und der Anwendungslogik sollen durch systemseitige Unterstützung verbessert werden. In diesem Zusammenhang stechen besonders drei Schlagwörter heraus: *Workflow-Management-Systeme* (WfMS) und *Web Services* bzw. *Service Oriented Architecture* (SOA). *Workflow-Management-Systeme* trennen die Prozess- von der Anwendungslogik und erlauben so eine anwendungsunabhängige Modellierung und Ausführung von Abläufen. *Web Services* und SOA beschäftigen sich hingegen mit der Aufsplittung von Anwendungssystemen und Komponenten in einzelne Dienste, um deren Integration in die Betriebs-IT zu vereinfachen.

Beide Ansätze haben ihre Vor- und Nachteile. Bei *Web Services* ist es durch Einsatz von Beschreibungsmodellen möglich, Methoden von Komponenten mit entsprechendem Aufwand als separate Dienste in die Anwendungslandschaft zu integrieren und miteinander zu verschalten. Aufgrund der Spezifikation ihrer Schnittstellen können Dienste dynamisch durch andere ersetzt werden. Die Modellierung von Geschäftsprozessen ist jedoch nicht, oder nur in geringem Maße, möglich. *Workflow-Management-Systeme* hingegen legen den Schwerpunkt auf die Organisation und Ausführung von Abläufen. Letzendlich werden jedoch auch bei WfMS bei der Ausführung der Geschäftsprozesse dahinterliegende Komponenten und Dienste angesprochen. Die angestrebte Flexibilität und Adaptivität wurde bei WfMS bisher höchstens bezüglich der Abläufe umgesetzt, nicht jedoch bezüglich der angesprochenen Komponenten und Dienste. Auswahl und Komposition der Dienste sind daher auch in aktuellen Forschungsprojekten noch zu statisch. An ein benutzerfreundliches Verschalten oder Auswechseln von Diensten und Komponenten ist daher bis heute noch nicht zu denken.
Die Vereinigung beider Herangehensweisen ist deshalb für die Zukunft von Informationssystemen im betrieblichen Umfeld von großer Bedeutung. Die Komponententechnologie, die sich u.a. mit der Beschreibung von Komponenten beschäftigt, kann wertvolle Impulse dazu einbringen. Im Rahmen des *AristaFlow*-Projekts, das vom Land Baden-Württemberg gefördert

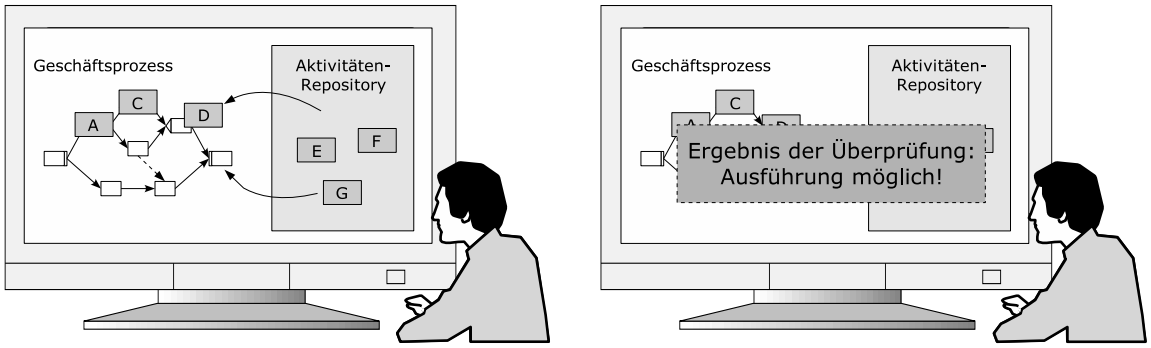

Abbildung 1.1: Zuweisen von Arbeitsschritten mittels Plug & Play

wird, sollen Konzepte für *Workflow-Management-Systeme* der nächsten Generation erarbeitet werden [1]. Der Schwerpunkt des Projekts ist die Vereinigung von Prozess- und Komponententechnologie für eine zeitnahe Umsetzung von Geschäftsprozessen in WfMS. Zur Beschreibung des Ablaufs eines Geschäftsprozesses wird ein Prozessmodell verwendet. Die vom Geschäftsprozess aufzurufenden Komponenten werden in einem sog. Aktivitäten-Repository verwaltet und dem Prozess zur Verfügung gestellt. Diese sollen, nach der Metapher von Plug & Play, auf einfache Art und Weise mit dem Prozessmodell verbunden werden, um einen ausführbaren Prozess zu erzeugen. Neben einer raschen Umsetzung von Geschäftsprozessen soll jedoch auch die fehlerfreie Ausführung der Prozesse sichergestellt werden (siehe Abbildung 1.1). Dazu muss die korrekte Verwendung der Komponenten anhand ihrer Spezifikation sichergestellt werden.
Mit der Vision der Integration von Plug & Play in WfMS sind jedoch vielfältige Herausforderungen verbunden. Einige davon sollen in der vorliegenden Diplomarbeit vorgestellt werden.

1.1 Aufgabenstellung

Die vorliegende Arbeit hat sich zweierlei Aufgaben zum Ziel gemacht. Zum einen geht es darum, grundlegende Probleme für die Integration „Plug & Play“-Konzepten in *Workflow-Management-Systemen* zu erörtern. Dabei sollen im Speziellen Abhängigkeiten, die bei der Umsetzung von „Plug & Play“-Konzepten berücksichtigt werden müssen, systematisch aufgezeigt werden.

Der zweite Teil der Arbeit beschäftigt sich mit der Erarbeitung eines im Bereich von Plug & Play einsetzbaren Konzeptes zur Integration von Zustandsinformationen in das Prozessmodell des WfMS ADEPT. Damit sollen auf zustandsbehaftete Daten basierende Abhängigkeiten bei der Auswahl und Komposition von Aktivitäten bzw. Komponenten sowie bei der Prozessanalyse berücksichtigt werden können.

1.2 Aufbau der Arbeit

Die vorliegende Diplomarbeit ist wie folgt gegliedert. Kapitel 2 führt relevante Grundlagen für das Verständnis der vorliegenden Arbeit ein. Dabei wird in Abschnitt 2.1 auf wichtige Begriffe im Kontext von WfMS eingegangen. In Abschnitt 2.2 wird das ADEPT WfMS als Ausgangsbasis für die Konzepte der vorliegenden Arbeit vorgestellt. Abschnitt 2.3 führt Grundlagen zu Komponententechnologie ein.

Kapitel 3 motiviert den Einsatz von Plug & Play-Konzepten im Kontext von WfMS und erörtert systematisch bestehende Probleme bei dessen Umsetzung.

Kapitel 4 geht genauer auf das Problem von zustandsbehafteten Daten ein. Es wird ein erster Ansatz zur Integration von zustandsbehafteten Daten in ADEPT vorgestellt. Ein ausgereifterer Lösungsansatz wird in Kapitel 5 eingeführt. Dieser erweitert ADEPT um die Verwendung von Zustandsautomaten zur Beschreibung von möglichen Zustandsübergängen.

In Kapitel 6 schließt die vorliegende Arbeit mit einem Ausblick ab.

Kapitel 2

Relevante Grundlagen

Im Folgenden werden wichtige Grundlagen für die vorliegende Diplomarbeit eingeführt. Im nächsten Abschnitt wird ein Überblick über Workflow-Management (WfM) gegeben. In Abschnitt 2.2 wird dabei das Workflow-Management-System ADEPT vorgestellt, welches die Grundlage für Konzepte dieser Arbeit darstellt. Des Weiteren wird ein Überblick über Komponenten gegeben.

2.1 Workflow-Management-Systeme

Workflow-Management-Systeme (WfMS) werden heutzutage vermehrt eingesetzt, um die Ausführung von Geschäftsprozessen in Unternehmen zu unterstützen. Sie versprechen eine zuverlässigere und effizientere Prozessausführung.
Workflow-Management umfasst unterschiedliche Aufgabenbereiche und Schnittstellen, wie Abbildung 2.1 verdeutlicht. Dazu gehören Schnittstellen und Werkzeuge, um Geschäftsprozesse zu modellieren und um diese auszuführen. Bei der Ausführung von Geschäftsprozessen müssen anfallende Arbeitsschritte an die jeweiligen Bearbeiter auf *Workflow-Clients* verteilt werden. Des Weiteren muss der Aufruf von benötigten Anwendungen in den Arbeitsschritten unterstützt werden. Ferner zählen auch die Überwachung der Abläufe sowie Schnittstellen zu anderen WfMS zu weiteren Aspekten von *Workflow-Management* [9].

Geschäftsprozesse stellen unterschiedliche Anforderungen an die Modellierung. So müssen in der Regel sequentielle, alternative, parallele und zyklische Abläufe möglich sein. Für die Darstellung von Geschäftsprozessen wird im Allgemeinen ein Graph als Repräsentation gewählt. Knoten im Graphen stellen dabei Arbeitsschritte dar. Über ausgehende Kanten werden die nachfolgenden Arbeitsschritte bestimmt.
Ein Arbeitsschritt in einem Prozess wird auch Aktivität genannt. Die Aktivität steht für unterschiedliche Aktionen, wie z.B. einen manuellen Arbeitsschritt oder den Aufruf einer Me-

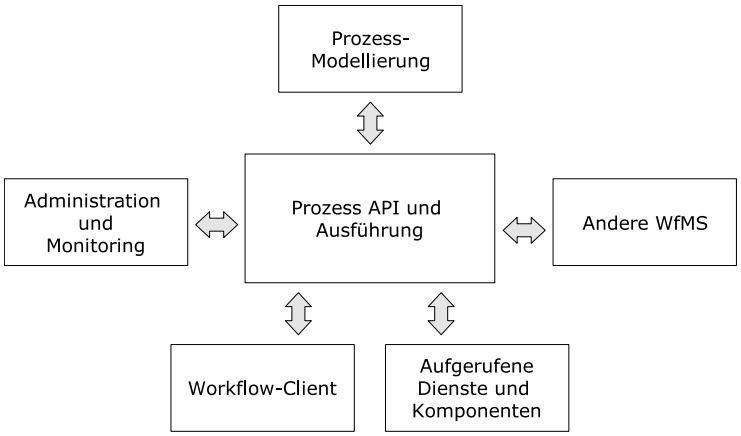

Abbildung 2.1: Workflow Reference Model [9]

thode in einer Softwarekomponente. Durch eine Zuordnung zwischen Aktivitäten und Knoten wird festgelegt, welche Arbeitsschritte durch den Geschäftsprozess ausgeführt werden. Die Zuordnung zwischen Aktivitäten und Knoten im Graph erlaubt eine flexible Modellierung sowie die Wiederverwendung von bereits bekannten Aktivitäten in Form von Aktivitätenvorlagen. Prozess-Designer wählen bereits vorgefertigte Aktivitätenvorlagen aus einem Repository aus und setzen diese direkt in einen Prozess ein.

Ergebnis der Modellierung eines Geschäftsprozesses ist eine Prozessvorlage. Um einen Prozess zu starten, wird die Vorlage logisch kopiert. Als Ergebnis erhält man eine ausführbare Prozessinstanz. Bei auf Graphen basierenden Systemen, wie z.B. ADEPT (s. Abschnitt 2.2), wird der Zustand der Ausführung durch Markierungen an Knoten und Kanten dargestellt. Damit eine Aktivität gestartet werden kann, muss ihr Knoten zunächst ein entsprechendes Signal durch die eingehenden Kanten erhalten. Ein Knoten mit einer fertig ausgeführten Aktivität gibt dieses Signal über ausgehende Kanten weiter.

Durch bisher aufgezeigte Konzepte kann ein Prozess modelliert und ausgeführt werden. In der Praxis ist dieses Vorgehen für viele Anwendungsdomänden, wie z.B. klinische Prozesse, jedoch zu statisch [4]. Abläufe müssen dynamisch angepasst werden können, um z.B. die Behandlung von Ausnahmefällen zu ermöglichen. So muss es möglich sein, Adhoc-Änderungen an einer laufenden Instanz vorzunehmen, um z.B. einen weiteren Arbeitsschritt, wie eine zusätzliche Untersuchung für einen Patienten, einzufügen. Änderungen können auch durch Optimierung der Geschäftsprozesse oder durch geänderte gesetzliche Rahmenbedingungen motiviert sein. In diesen Fällen sind nicht nur einzelne Prozessinstanzen betroffen sondern vielmehr die Prozessvorlage. Es ist offentsichtlich, dass die Änderungen an der Prozessvorlage auch auf laufende Instanzen übertragen werden sollten, wo dies möglich ist. Dieser Vorgang wird auch als Schemaevolution bezeichnet. Durch Modellierung, Ausführung, erneute Analyse und Optimierung der Prozesse, entsteht so ein Kreislauf, auch unter *Business Process Lifecycle*

(Lebenszyklus von Geschäftsprozessen) bekannt (s. Abb. 2.2).

Angesichts der bestehenden Anforderungen muss ein modernes WfMS sowohl Adhoc-Änderungen an Prozessinstanzen als auch Änderungen an der Prozessvorlage (Schemaevolution) nicht nur in einer effizienten Art und Weise ermöglichen, sondern auch für die zuverlässige Ausführung der Prozesse garantieren. So dürfen die angesprochenen Änderungen keine Fehler in der Ausführung der Prozesse verursachen, z.B. durch fehlende Daten nach dem Löschen einer Aktivität. Aktuelle auf dem Markt verfügbare WfMS sind im Moment noch nicht in der Lage, diese Flexibilität und Robustheit zu gewähren. In Forschungsprojekten wie ADEPT können jedoch Ansätze dafür gefunden werden.

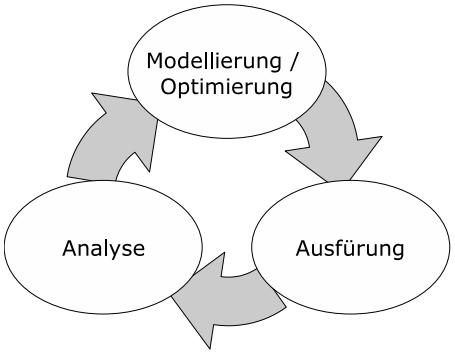

Abbildung 2.2: Business Process Lifecycle

Neben den genannten Änderungen am Ablauf von Prozessen müssen moderne WfMS jedoch auch Änderungen in der Auswahl der im Prozess verwendeten Aktivitäten unterstützen. Dies bedeutet in erster Linie, dass Aktivitäten ausgetauscht werden können. Hinter diesen Aktivitäten stehen jedoch oft komplexe Anwendungssysteme, wie z.B. ein Warenwirtschaftssystem. Neue Anforderungen an diese Anwendungssysteme, welche z.B. durch Optimierungen oder einer geänderten gesetzlichen Rahmenbedingung motiviert sein können, betreffen daher u.U. auch ganze Softwarekomponenten. Bei Bedarf müssen diese auf den neuesten Stand gebracht oder vollständig ausgetauscht werden können.

Ein WfMS soll die angesprochenen Änderungen unterstützen. Dabei muss es den Forderungen nach einer effizienten und fehlerfreien Ausführung jedoch weiterhin genügen. Die Vorgänge, um Änderungen durchzuführen, müssen daher nicht nur performant ablaufen. Bei den modifizierten Prozessen darf es auch weiterhin nicht zu einem Fehler während der Ausführung kommen.

2.2 ADEPT

Grundlage für alle in dieser Arbeit vorgestellten Konzepte stellt das ADEPT WfMS und dahinter liegende Konzepte dar. Darum soll an dieser Stelle eine Einführung in die Notation und Konzepte von ADEPT gegeben werden. Da eine detaillierte Behandlung von ADEPT den Rahmen dieser Arbeit sprengen würde, wird sich der Abschnitt auf für das Verständnis der vorliegenden Arbeit grundlegende Aspekte beschränken. Der interessierte Leser wird für weiterführende Informationen auf [12] bzw. für Details zu konkreten Definitionen und Algorithmen auf [11] verwiesen.

2.2.1 Überblick über ADEPT

Hinter ADEPT stehen Konzepte für ein adaptives WfMS, die an der Universität Ulm in der Abteilung Datenbanken und Informationssysteme entwickelt wurden. Bei der Entwicklung von ADEPT wurden auf verschiedene Aspekte besonders großen Wert gelegt. Prozesse sollten bereits zur Modellierungszeit hinsichtlich ihrer Ausführbarkeit analysierbar sein. Dennoch sollten sie auch zur Laufzeit angepasst werden können. Weitere Aspekte, die bei ADEPT eine große Rolle spielen, sind Performanz und Skalierbarkeit.

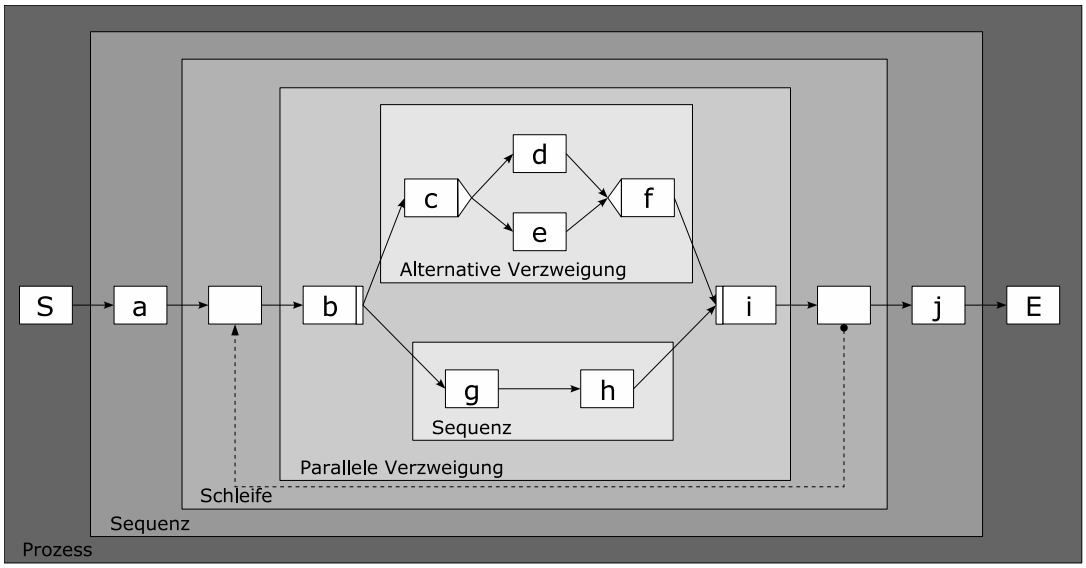

Abbildung 2.3: Knotenkonstrukte und Blockstruktur von Prozessen in ADEPT

Zur Modellierung von Prozessen verwendet ADEPT gerichtete Graphen, denen eine Blockstruktur zugrunde liegt. Den Knoten im Graphen sind Aktivitäten des Prozesses zugeordnet. Spezielle blockorientierte Knotenkonstrukte erlauben die Modellierung paralleler, alternativer sowie zyklischer Abläufe.
Die Verwendung der Blockstruktur erlaubt somit auf einfache Weise die Modellierung von

Sub-Prozessen, da ein einzelner Knoten einen Sub-Prozess darstellen kann. Abbildung 2.3 zeigt die erwähnten Knotenkonstrukte und veranschaulicht die Blockstruktur von Prozessen in ADEPT.

Da Aktivitäten des Prozesses Knoten im Prozessmodell zugeordnet werden, ist es möglich, Aktivitäten auszutauschen, ohne Knoten des Prozessmodells entfernen und wieder einfügen zu müssen. Die an Knoten hinterlegten Aktivitäten wiederum können für einen tatsächlichen Dienst stehen, sei es die Methode einer Komponente, den Aufruf eines Programms oder einen manuellen Arbeitsschritt. Die Aktivität dient dazu, interpretierbare Informationen über den dahinter stehenden Dienst erhalten zu können, wie z.B. Eingabe- und Ausgabeparameter. Abbildung 2.4 veranschaulicht dieses Prinzip.

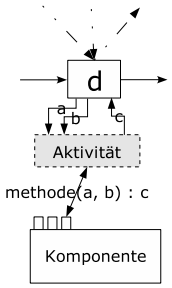

Abbildung 2.4: Zusammenhang zwischen Knoten, Aktivität und dahinterliegender Komponente

2.2.2 Kontroll- und Datenfluss

In ADEPT werden Kontroll- und Datenfluss explizit getrennt. Zunächst wird im Folgenden auf den Kontrollfluss eingegangen. Im Anschluss daran wird der Datenfluss in ADEPT behandelt.

Zum Kontrollfluss (kurz KF) gehören alle Konstrukte, welche die Abfolge der Aktivitäten bestimmen. Wie bereits erwähnt, handelt es sich bei ADEPT um einen blockbasierten Ansatz. Blöcke werden durch ganze Prozesse sowie durch parallele, alternative und zyklische Abläufe gebildet.
Als alternative Verzweigung erlaubt ADEPT XOR-Splits. Die Ausgangssemantik des XOR-Split-Knotens wird auch als ONE_OF_ALL bezeichnet, da nur genau einer der ausgehenden Kanten gefolgt wird. In Abbildung 2.5 stellt der Knoten d ein XOR-Split und der Knoten g den dazugehörigen XOR-Join des Blocks der alternativen Verzweigung dar.

Ein paralleler Block wird durch einen AND-Split-Knoten eingeleitet, d.h. es wird allen ausgehenden Kanten des AND-Splits gefolgt (ALL_OF_ALL Ausgangssemantik). In Abbildung 2.5 stellt Knoten c ein AND-Split dar, während Knoten j den dazugehörigen AND-Join darstellt.

Zwischen Knoten in unterschiedlichen parallelen Zweigen besteht keine vorgegebene Ausführungsreihenfolge. In einigen Anwendungsszenarien ist jedoch eine Synchronisation paralleler Vorgänge erforderlich. Diese Synchronisation kann allerdings unter Berücksichtigung der Blockstruktur nicht erfolgen. Kontrollkanten zwischen den Knoten von parallelen Zweigen sind nicht zulässig. Daher wurden in ADEPT sog. Synchronisationskanten eingeführt. Diese erlauben einen kontrollierten Bruch der Blockstruktur und ermöglichen so eine Synchronisation zwischen parallelen Ausführungszweigen. Knoten mit einer eingehenden Synchronisationskante können erst ausgeführt werden, wenn der Knoten, von dem die Kante ausgeht, beendet oder abgewählt wurde. In Abbildung 2.5 wird Aktivität i mit Aktivität f unter Verwendung einer Synchronisationskante (gestrichelter Pfeil) synchronisiert. Aktivität i kann daher erst ausgeführt werden, wenn Aktivität h und Aktivität f ausgeführt wurden oder wenn Aktivität h ausgeführt wurde und Aktivität f nicht mehr ausführbar ist.

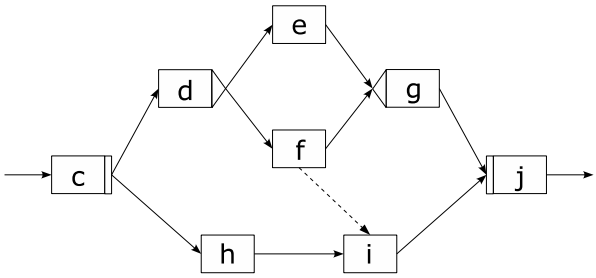

Abbildung 2.5: Alternative Verzweigung, parallele Verzweigung und Synchronisationskante

Ein spezieller Block kann durch einen einleitenden AND-Split und einen abschließenden XOR-Join gebildet werden. Diese Kombination bildet eine parallele Verzweigung mit finaler Auswahl. Dies bedeutet, dass zwar alle Zweige parallel gestartet werden, jedoch nur die Ergebnisse (Daten) von einem Zweig ausgewählt werden. Die Ergebnisse der abgewählten Zweige werden verworfen oder ignoriert.

Der Datenfluss (kurz DF) wird in ADEPT mit Datenkanten und Datenelementen explizit modelliert. Eine Datenkante ist eine gerichtete Kante zwischen Knoten und Datenelement. An einem Knoten eingehende Kanten bedeuten einen lesenden Zugriff, ausgehende Kanten bedeuten einen schreibenden Zugriff. Der an einem Knoten hinterlegten Aktivität werden zur Laufzeit die Werte der mit lesenden Kanten referenzierten Datenelemente als Eingabeparameter zur Verfügung gestellt. Analog werden die durch schreibende Kanten referenzierten Datenelemente mit den Ausgabeparametern der Aktivität geschrieben. Abbildung 2.6 zeigt einen Ausschnitt eines Prozesses mit Datenfluss.
Die explizite Modellierung des Datenflusses ermöglicht Analysen auf Basis der Prozessvorlage. Dadurch können Datenzugriffe bereits vor der Ausführung sichergestellt werden. Das Lesen eines Datenelements, welches noch keine Daten enthält, wird so bereits zur Modellierungszeit ausgeschlossen. In Abbildung 2.6 liest die Aktivität von Knoten a das Datenelement d_1.

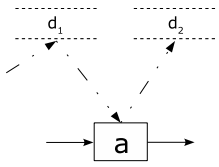

Abbildung 2.6: Ein Knoten mit lesender und schreibender Datenkante

Anhand des Kontrollflusses und der explizit modellierten Datenzugriffe, kann festgestellt werden, ob eine weitere Aktivität das Datenelement vor dem lesenden Zugriff schreibt. Ist dies der Fall, gilt der Eingabeparameter als versorgt und die Ausführung des Prozesses ist daher möglich.

2.2.3 Ausführungszustände von Aktivitäten

Zur Laufzeit einer Prozessinstanz befinden sich Aktivitäten in unterschiedlichen Zuständen. Der Zustand der Ausführung wird durch Markierungen an Knoten und Kanten dargestellt. In Abbildung 2.7 sind die wichtigsten Zustände in ADEPT veranschaulicht. Zunächst befindet sich eine Aktivität im Zustand *NOT_ACTIVATED*. Im Zustand *ACTIVATED* befindet sich eine Aktivität, wenn sie ausgeführt werden kann. Der Zustand *STARTED* gibt an, dass sich eine Aktivität in der Bearbeitung befindet. In den Zustand *COMPLETED* wird eine Aktivität überführt, wenn ihre Ausführung erfolgreich verlief. Für eine vollständige Übersicht über alle möglichen Zustände wird auf die Arbeit von Reichert in [11, S. 90] verwiesen.

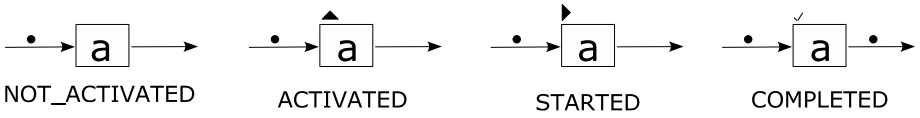

Abbildung 2.7: Darstellung der Zustände von Aktivitäten

2.2.4 Konzepte zur Unterstützung von Adaptivität

Anpassungen an Prozessen werden in ADEPT auf drei Ebenen realisiert. ADEPT ermöglicht es Prozessvorlagen zu ändern. Des Weiteren können diese Änderungen auch auf laufende Instanzen der geänderten Vorlage übertragen werden (Schemaevolution). Die dritte Ebene sind Änderungen an einzelnen laufenden Instanzen (Adhoc-Änderungen).

Als Grundlage dienen generische Änderungsoperationen, welche auf Prozesse und Instanzen angewendet werden können. Diese Operationen lassen sich unter wohl definierten Voraussetzungen anwenden und liefern ausschließlich korrekte Graphen als Ergebnis.
Im Falle einer Schemaevolution wird so für jede Instanz individuell und effizient festgestellt,

ob eine Migration der Instanz auf die modifizierte Prozessvorlage möglich ist. Ist dies der Fall, kann die Instanz mit Hilfe der Änderungsoperationen automatisch an die geänderte Vorlage angepasst werden. An dieser Stelle soll nicht näher auf das Thema eingegangen werden. Ein einführender Überblick über das Thema findet sich in [15], näheres zu Adaptivität und prototypischer Implementierung in ADEPT findet sich in [13]. Für Details wird auf [16] verwiesen.

2.2.5 Definitionen und Notationen

Definitionen, die in dieser Diplomarbeit vorgestellt werden, erweitern oft Definitionen von Reichert aus [11]. Da eine Ausführung der vollständigen Definition den Rahmen der Arbeit sprengen würde, wird der Leser an gegebener Stelle auf die Arbeit von Reichert verwiesen. Neue Definitionen in dieser Diplomarbeit lehnen sich in vereinfachter Form an die Notation von Reichert an. Die Zugehörigkeit zu einem Element wird durch die hochgestellte Abkürzung des Elements gekennzeichnet. So hat z.B. das Feld $Type$ eines Parameters P die Bezeichnung $Type^P$.

2.3 Komponententechnologie

Komponenten sind Anwendungen, welche sich als Teil in ein größeres System einfügen. Insbesondere in Betrieben kommen dadurch oft unterschiedlichste Anwendungen zum Einsatz. Die Vorteile liegen dabei auf der Hand: Die Verwendung bereits vorgefertigter Komponenten verspricht günstiger zu sein als die Entwicklung von eigenen Anwendungen.

Damit Komponenten und ihre Methoden verwendet werden können, müssen diese genau beschrieben werden. Die Spezifikation einer Komponente kann als Vertrag angesehen werden, an welchen sich die Komponente hält und worauf sich der Anwender verlassen kann. Entwickler können dadurch die Funktionen unterschiedlicher Komponenten verstehen und durch eigenen Code zusammenarbeiten lassen. Durch Komposition der Einzelteile entsteht eine neue Anwendung, wobei der Code der Entwickler die Zusammenarbeit der Komponenten sichergestellt [2].

Für die Spezifikation von Komponenten wurden unterschiedliche Beschreibungsmodelle entwickelt. Dazu gehören z.B. die *Interface Description Language* (IDL) von CORBA [7] und *Enterprise JavaBeans* [10]. Ziel dieser Beschreibungsmodelle ist eine Spezifikation, welche die Verwendung der Komponente ermöglicht, ohne ihre interne Funktionsweise kennen zu müssen.

Parallel zu den Bemühungen, die Zusammenarbeit und Verwendung von Komponenten zu vereinfachen, haben sich die Beschreibungsmodelle auch zur allgemeinen Entwicklung von Programmen etabliert. Durch eine möglichst genaue Spezifikation des zu erstellenden Programms sollen mögliche Probleme bereits auf Ebene der Architektur gelöst bzw. verhindert werden. Das heute wohl wichtigste Modell für die Softwareentwicklung ist die *Unified Mode-*

ling Language (UML) der *Object Management Group* (OMG), welche inzwischen in Version 2.0 verabschiedet wurde [8].

An dieser Stelle werden unterschiedliche Zielsetzungen bei der Spezifikation sichtbar. Auf der einen Seite sollen Komponenten mit ihren Methoden möglichst präzise spezifiziert werden, um ihre Verwendung ohne Kenntnis über deren internen Aufbau zu ermöglichen. Auf der anderen Seite soll ein Programm, dieses als Komponente betrachtet, in seiner internen Funktionsweise präzise beschrieben werden, um eine möglichst fehlerfreie Entwicklung zu garantieren. Im Zusammenhang mit UML werden diese zwei Sichten auch als *Black Box* und *White Box* bezeichnet.

Es gibt unterschiedliche Wege eine Komponente zu spezifizieren. Grundlegend sind hierbei die Fragen, welche Methoden zur Verfügung stehen und wie sie aufgerufen werden können. Letzteres schließt die Signatur der Methoden mit ein, also Informationen über Parameter und Datentypen. Ein einfaches Beispiel für eine Spezifikation sind *Application Programming Interface* (API) Dokumentationen.

Kapitel 3

Plug & Play

Ein wichtiger Schritt bei der Umsetzung von Geschäftsprozessen in WfMS ist die Zuordnung bzw. Auswahl der Arbeitsschritte, welche an den Knoten des Prozessmodells ausgeführt werden sollen. Da in ADEPT zwischen den Knoten im Prozessmodell und Aktivitäten unterschieden wird, müssen Aktivitäten an den Knoten hinterlegt werden. Dafür müssen Aktivitäten neu erstellt oder bestehende Aktiväten wiederverwendet werden. Dieser Vorgang soll sich für den Benutzer möglichst einfach gestalten. Aktivitäten sollen sich nach dem Prinzip des Plug & Plays einfach an einen Knoten „einstecken“ lassen. In Abbildung 3.1 werden u.a. Methoden einer Lagerverwaltung aus einem Aktivitäten-Repository (ein Pool von bereits bestehenden Aktivitäten bzw. Komponenten) als Aktivitäten an Knoten hinterlegt.

Unter Plug & Play versteht man das dynamische und fehlerfreie Zusammenspiel unterschiedlicher Komponenten. Im Kontext von Betriebsanwendungen bilden unterschiedliche Systeme die verwendeten Komponenten. Dazu gehören z.B. Systeme zur Warenwirtschaft, zur Lagerverwaltung oder Buchhaltung. Das Zusammenspiel dieser Komponenten wird heute

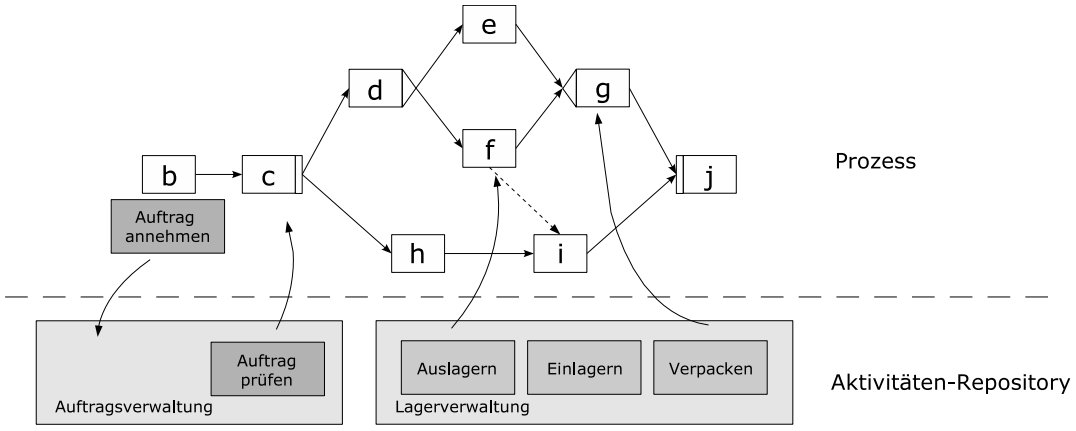

Abbildung 3.1: Aktivitäten werden mittels Plug & Play an Knoten hinterlegt

oft durch statische Programmierung erreicht. Unter dem Hintergrund von Arbeitsabläufen, welche aus unterschiedlichen Gründen dynamisch geändert werden müssen, ist diese feste Verschaltung jedoch hinderlich. Ebenso problematisch ist dies, wenn Komponenten geändert oder ausgewechselt werden sollen. Abhilfe versprechen hier adaptive WfMS wie ADEPT, welche nicht nur Änderungen an den Abläufen sondern auch den dynamischen Austausch der Komponenten erlauben. Mittels Plug & Play soll dies einfach und und vor allem auch fehlerfrei möglich sein. Denn trotz der erlaubten Dynamik müssen diese Systeme zuverlässig arbeiten [2].

In den nächsten beiden Abschnitten wird auf die Bedeutung von Plug & Play bei Softwarekomponenten und in WfMS näher eingegangen. Abschnitt 3.3 wird sich näher mit Zusammenhängen und Abhängigkeiten zwischen Komponenten beschäftigen, welche das Zusammenspiel der Aktivitäten beeinträchtigen könnten. Nachdem in Abschnitt 3.5 weitere Aspekte und Anforderungen erläutert werden, werden in Abschnitt 3.7 semantische und funktionale Aspekte voneinander abgegrenzt. Das Kapitel schließt in Abschnitt 3.8 mit einer Zusammenfassung und einem Ausblick auf weitere Inhalte der Diplomarbeit.

3.1 Plug & Play und Softwarekomponenten

Um zwischen Softwarekomponenten Plug & Play zu ermöglichen, ist eine API Dokumentation nicht ausreichend. Hier wird eine durch Maschinen verarbeitbare Komponentenbeschreibung benötigt sowie ein System, welches die Kommunikation zwischen den Komponenten vereinheitlicht (sog. *Middleware*). Zur Spezifikation der Komponenten gibt es unterschiedliche Modelle und Sprachen. Erwähnenswert sind aus diesem Bereich die *Interface Definition Language* (IDL) aus CORBA, die *Web Service Description Language* (WSDL) aus dem Bereich der *Web Services*, die Beschreibung von *Enterprise Java Beans* (EJB) sowie die IDL von Microsofts DCOM.

Mit diesen Modellen können zumindest Komponenten, Typen, Methoden und Parameter beschrieben werden. Dabei werden Methoden der Komponenten zumeist sehr isoliert betrachtet. Die Methoden werden meistens als zustandslos angesehen und hängen ausschließlich von den übergebenen Daten ab. In vielen Fällen reicht diese Betrachtungsweise jedoch nicht aus: Da es sich bei Komponenten um komplexe Anwendungssysteme handelt, kann die Ausführung der Methoden vom Zustand der Komponente oder von zusätzlichen internen Daten abhängig sein. Dadurch können Methoden nur unter bestimmten Voraussetzungen aufrufbar sein, also z.B. Aufrufreihenfolgen voraussetzen. Einige der genannten Beschreibungsmodelle bieten unteschiedliche Mittel an, um solchen Abhängigkeiten zu begegnen. Speziell für *Web Services* kann z.B. die *Web Services Conversation Language* (WSCL), eine Erweiterung zur WSDL, eingesetzt werden. Diese bietet u.a. eine Möglichkeit zur Modellierung von Aufrufabfolgen an [14].

Über entsprechende Beschreibungen erhofft man sich möglichst reibungslose Komposition und Austausch von Komponenten. *Middleware* bietet hierfür eine gemeinsame Platform und die notwendigen Kommunikationswege. Hierbei muss jedoch beachtet werden, dass die Spezifikation der Komponenten mit unterschiedlichen Beschreibungsmodellen erfolgt sein kann. In der Forschung werden daher weiterführende Spezifikationen entwickelt, welche zusätzliche Abhängigkeiten beschreiben können und vorhandene Beschreibungsmodelle vereinheitlichen. In [17] werden weitere Informationen zum Thema Komponenten und *Middleware* gegeben und ein Meta-Beschreibungsmodell für Komponenten vorgeschlagen. Ein umfassendes Modell zur Spezifikation von Komponenten wird in [3] vorgestellt, ein Überblick über den Ansatz ist in [6] zu finden.

3.2 Plug & Play mit Komponenten in Workflow-Management-Systemen

Werden Arbeitsabläufe durch ein WfMS unterstützt, müssen Komponenten und Methoden ausgewählt und mit Hilfe des Prozessmodells miteinander verschaltet werden. Durch „Plug & Play“-Techniken soll das Auffinden und Verschalten der Komponenten für den Benutzer vereinfacht werden. Dabei ist es wichtig, dass die Komposition fehlerfrei funktioniert und dafür notwendige Prüfungen für den Benutzer transparent vonstatten gehen.

Die Abläufe werden i.A. von Prozess-Designern erstellt. Normale Benutzer des Systems können die Abläufe jedoch, z.B. durch Adhoc-Änderungen, beeinflussen. In beiden Fällen müssen die Anwender die gewünschten Aktivitäten in einem Repository auffinden, um diese dann in den Prozess einzusetzen. Daraufhin muss überprüft werden, ob die Aktivitäten in der vom Benutzer vorgeschlagenen Komposition ausgeführt werden können, oder ob z.B. noch Daten zur Ausführung fehlen. Diese Überprüfung ist notwendig, da die Komposition nicht durch Entwickler geschieht, die sich in APIs, formelle und informelle Spezifikationen einlesen können. Insbesondere Benutzer des WfMS haben keinen Überblick über Zusammenhänge zwischen den Aktivitäten oder gar über Details ihrer internen Funktionsweise. Während der Prozess-Modellierung und bei Adhoc-Änderungen zur Laufzeit, muss eine Überprüfung, ob eine Aktivität zu den gegebenen Bedingungen ausführbar ist, möglich sein. Die dafür benötigten Informationen müssen für das WfMS aus einer Beschreibung der Komponenten bzw. der einzelnen Aktivitäten ableitbar sein.

An dieser Stelle muss jedoch zwischen zwei unterschiedlichen Aspekten im Einsatz der Komponentenbeschreibungen in WfMS unterschieden werden. Auf der einen Seite müssen die Benutzer in der Auswahl der Aktivitäten unterstützt werden. Dafür muss das WfMS dem Benutzer verfügbare Informationen zur Verfügung stellen, jedoch gleichzeitig auch die Beschreibungen selbst auswerten, um sinnvolle Vorschläge bzgl. der Aktivitätenauswahl ma-

chen zu können. Auf der anderen Seite ist es notwendig, dass das WfMS die Komposition der Aktivitäten auf Fehler überprüft. Diese Überprüfung muss erfolgen, wenn der Prozess durch Komposition von Aktivitäten erstellt wird oder Änderungen (z.B. Adhoc-Änderungen) erfährt.

Für Softwarekomponenten existieren unterschiedliche formale Beschreibungsmodelle. Für den Einsatz der Komponenten in einem WfMS muss das System diese Beschreibung auswerten können. Eine minimale Voraussetzung ist hier die Interpretation von Schnittstellen-Beschreibungen. Während es bei der Entwicklung von Software möglich ist, sich auf ein Beschreibungsmodell festzulegen, muss ein WfMS die Beschreibung unterschiedlichster Komponenten interpretieren können. Dafür gibt es grundsätzlich zwei Möglichkeiten: Komponenten müssen für den Einsatz erneut für das WfMS beschrieben werden. Die Alternative ist die Formulierung einer abstrakten Schnittstelle für Anfragen an unterschiedliche Beschreibungsmodelle. In beiden Fällen wird ein abstraktes Beschreibungsmodell benötigt, auf dem das WfMS arbeiten kann. Im Vergleich zur Softwareentwicklung, stellen WfMS jedoch weitere Anforderungen an das Beschreibungsmodell.

Ein Beschreibungsmodell zur Softwareentwicklung sollte eine umfangreiche und detaillierte Modellierung ermöglichen, vor allem auch über die internen Zusammenhänge der Komponente (*White Box*). Diese Informationen dienen der Erstellung der Komponente selbst. Sie werden in bestimmten Situationen zur Analyse eingesetzt, i.A. jedoch von den Entwicklern verwendet.

Je nachdem, ob es um das Auffinden von Aktivitäten oder die formale Analyse der Komposition geht, sind die Anforderungen an ein Beschreibungsmodell für ein WfMS unterschiedlich. Von Seiten des Aktivitäten-Repositorys, welches die Benutzer bei der Auswahl der Aktivitäten unterstützt, besteht die Anforderung von informellen Beschreibungen. Neben informellen sollten jedoch auch formale Beschreibungen vorhanden sein. Diese können vom WfMS interpretiert werden, um sinnvolle Vorschläge für den Benutzer zu generieren. Dies ist bei Adhoc-Änderungen für normale Benutzer hilfreich, jedoch ebenso eine Unterstützung für Prozess-Designer. Letztere müssen jedoch auch tiefere Einblicke in die Funktion der Komponente haben. Für sie können z.B. API Dokumentationen aus der Softwareentwicklung durchaus hilfreich sein. Während der Modellierungszeit ist Performanz zweitrangig, daher sind hier auch umfangreiche Beschreibungen einsetzbar.

Geht es um die Analyse der Komposition, so muss das Beschreibungsmodell eine formale Spezifikation ermöglichen. Die Informationen werden dabei hauptsächlich vom WfMS selbst verwendet. Dies bedeutet, dass hier das Augenmerk auf eine performante und automatische Verarbeitung gelegt werden muss. Die Spezifikation sollte daher nur die notwendigsten Informationen zur Verwendung der Komponente bereitstellen (*Black Box*). Zudem müssen die erforderlichen Informationen während der Analyse effizient abrufbar sein. Dies spielt insbesondere bei Schemaevolution eine Rolle, wenn Änderungen vieler Instanzen eines Prozesses eine effiziente Überprüfung unumgänglich machen. Eine zeitaufwändige Anfrage an die Ak-

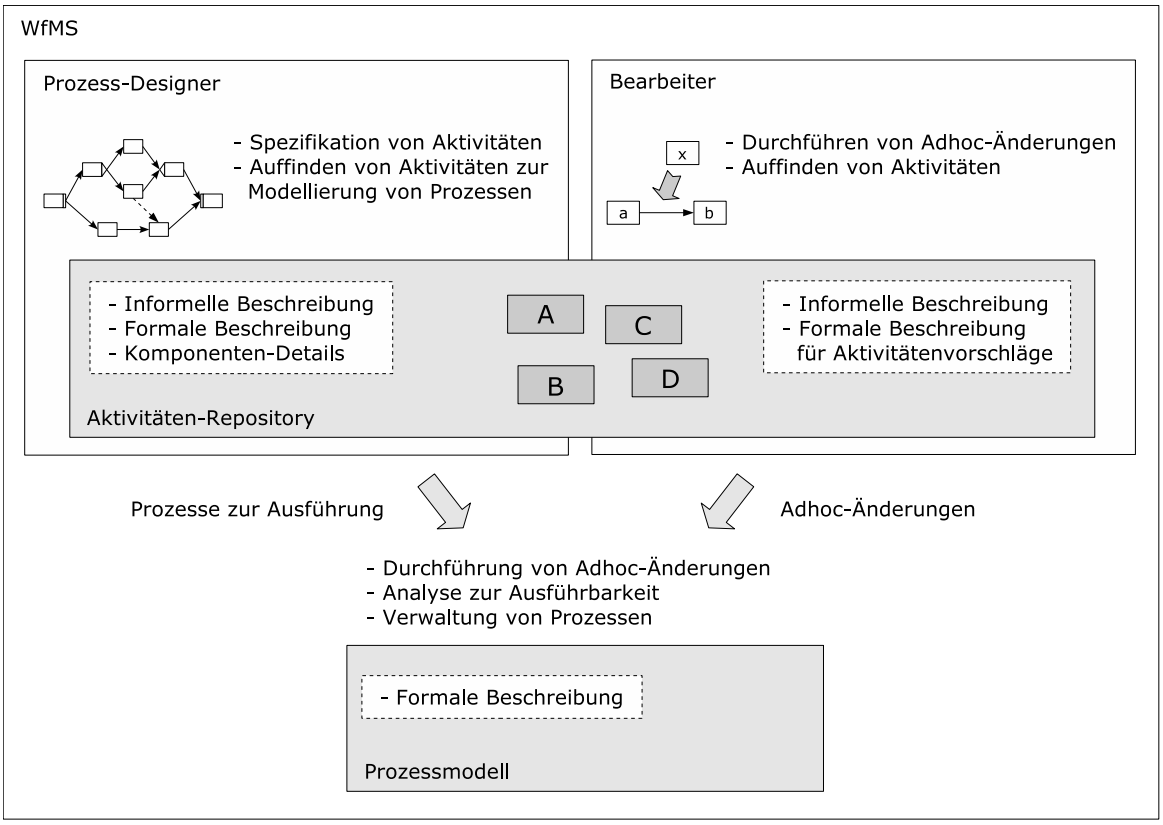

Abbildung 3.2: Zusammenhänge im WfMS bezüglich Aktivitätenbeschreibungen

tivität bzw. das Aktivitäten-Repository darf hier nicht notwendig sein. Vielmehr müssen die entsprechenden Informationen in das Prozessmodell des WfMS integriert sein. Für das Bereitstellen der formalen Beschreibungen, sollten Prozess-Designer detaillierte Informationen über Komponenten und ihre Methoden im Aktivitäten-Repository verwalten. Abbildung 3.2 fasst die Zusammenhänge zwischen WfMS, Aktivitäten-Repository und den Aufgaben der Benutzer zusammen.

Im Folgenden Abschnitt wird ein Überblick darüber gegeben, welche Aspekte von einem Beschreibungsmodell für Komponenten in einem WfMS abgedeckt bzw. berücksichtigt werden müssen. Da es Kern darum geht, wovon die Ausführbarkeit einer Aktivität abhängig sein kann, wird hier allgemein von „Abhängigkeiten“ gesprochen.

3.3 Abhängigkeiten zwischen Aktivitäten

Zur fehlerfreien Ausführung einer Aktivität müssen alle notwendigen Randbedingungen gegeben sein. Dies bedeutet in erster Linie, dass die Funktionalität jeder Aktivität, zu jedem ihrer möglichen Ausführungszeitpunkte im Prozess, gewährleistet sein muss. So muss z.B. sicher-

gestellt werden, dass die von ihr benötigten Daten zur Verfügung stehen. Die Abhängigkeit einer Aktivität von Daten ist jedoch nur einer von vielen Aspekten, die die Ausführbarkeit von Aktivitäten beeinflussen und daher bei der Umsetzung von Plug & Play berücksichtigt werden müssen.

In Geschäftsprozessen sind unzählige Abhängigkeiten zwischen Aktivitäten denkbar, insbesondere wenn rein semantische Abhängigkeiten hinzugenommen werden. Um dieses weite Feld etwas übersichtlicher zu gestalten und gleichzeitig einen systematischen Überblick über verschiedene Abhängigkeiten zu bieten, wird in diesem Abschnitt eine Klassifikation vorgeschlagen. Das Augenmerk soll hier hauptsächlich auf funktionale Abhängigkeiten gerichtet sein, also auf Abhängigkeiten, welche die Ausführbarkeit der Aktivität direkt betreffen.

Eine Aktivität kann für ihre Ausführung verschiedene Voraussetzungen an ihre Umgebung stellen. Diese lassen sich grob in Abhängigkeiten von Daten, Ressourcen, anderen Arbeitsschritten, Zeit, Zugriffsrechte sowie semantische Abhängigkeiten einteilen. Im Weiteren werden diese Abhängigkeitsbeziehungen näher beschrieben und feiner unterteilt.

1. Abhängigkeit von Daten
   Eine grundlegende Abhängigkeit stellt die Versorgung von Parametern dar. Darüber hinaus können jedoch weitere Abhängigkeiten bezüglich Daten bestehen. Dabei muss es sich nicht notwendigerweise um konkrete, im Sinne von durch das WfMS verwaltbare Daten handeln. So könnte es sich bei Daten z.B. ebenso gut um ein Papierdokument handeln, welches in einem manuellen Arbeitsschritt erzeugt wurde. Auch andere für das WfMS unsichtbare Datenflüsse (versteckte Datenflüsse), wie sie auch unter Komponentenkompositionen vorkommen können, können die Ausführbarkeit von Aktivitäten bestimmen. Ebenso zu berücksichtigen sind Daten, die lediglich in Form von Referenzen vom WfMS im Datenfluss weitergereicht werden.

   (a) Existenz der Daten
   Beispiel: *Die Aktivität „E-Mail-Versand“ braucht die Daten „E-Mail-Adresse“ und „Nachricht“.*
   Die Aktivität hängt also davon ab, dass bestimmte Daten verfügbar sind. Ob diese als durch das WfMS kontrollierbare Parameter übergeben werden, ist hierbei irrelevant.

   (b) Typ der Daten
   Beispiel: *Das Datum „E-Mail-Adresse“ der Aktivität „E-Mail-Versand“ muss eine E-Mail-Adresse sein.*
   Die Aktivität hängt davon ab, dass Daten einem bestimmten Typ entsprechen.

   (c) Zustand der Daten
   Beispiel: *Für die Aktivität „Sicherer E-Mail-Versand“ müssen sich die Daten „Nachricht“ im Zustand „verschlüsselt“ befinden.*

Die Aktivität hängt davon ab, dass die Daten einer bestimmten Verarbeitung unterzogen wurden. Dies wird i.A. durch die Verarbeitung durch andere Aktivitäten erreicht.

(d) Ort der Daten
Beispiel: *Die Aktivität „Bestellung SAP“ muss ihre Daten vom SAP-Backend beziehen können.*
Beispiel: *Die durch den Parameter „Rechnungs-Nr.“ referenzierten Daten müssen sich auf die DB2 Datenbank „Archivierte_Rechnungen“ beziehen.*
Die vom WfMS gelieferten Daten sind lediglich eine Referenz auf die tatsächlichen Daten: Die Aktivität kann Einschränkungen auf den Ort der Daten haben. Sie muss sich darauf verlassen können, dass sich die Referenz auf ein bestimmtes *Backend* bezieht. Als *Backend* sind hier u.a. Datenbanken, unterschiedliche Informationssysteme und Dateisysteme denkbar.

(e) Abhängigkeiten unter den benötigten Daten
Beispiel: *Der Parameter „Lieferadresse“ muss sich auf die Bestellung „Bestell-ID“ beziehen.*
Es bestehen Abhängigkeiten unter den Daten, die von einer Aktivität benötigt werden. Dies tritt z.B. auf, wenn mehrere Daten übergeben werden, welche Teil einer Relationen aus einer Datenbank sind. Falls im Kontext eines Prozesses unterschiedliche Kombinationen von Datenelementen die Parameter erfüllen würden, muss sichergestellt werden, dass die Daten den richtigen Bezug haben.

(f) Abhängigkeiten unter Parametern
Beispiel: Vergleichbar mit Shell-Kommandos: Ein optionaler Parameter macht einen weiteren Parameter obligat.
Eine Aktivität hat unterschiedliche Ausführungsmöglichkeiten. Die Auswahl, welche Ausführungsart wahrgenommen wird, erfolgt durch das Setzen entsprechender Parameter. Je nach Ausführungsart werden unterschiedliche Parameter optional oder obligat sein. Eine Aktivität mit solchen Abhängigkeiten lässt sich in mehrere Aktivitäten aufteilen. Dieser Lösungsansatz kann jedoch bei komplexen Parameter-Konstrukten schnell unübersichtlich werden.

2. Abhängigkeit von Ressourcen
Die Verwaltung von Ressourcen ist ein sehr weites Gebiet und umfasst Bereiche, wie die Verwaltung von Gütern und Maschinen. Im Allgemeinen sind Ressourcen an einen Ort gebunden, können jedoch verschiedenste relevante Eigenschaften besitzen. Diese Eigenschaften können Bedingungen zur Ausführung einer Aktivität sein. Dadurch sind für alle Ressourcen komplexe Abhängigkeiten möglich, vergleichbar mit Bearbeiterzuordnungsregeln. Hier sind sehr viele Abhängigkeiten denkbar. In dieser Diplomarbeit soll jedoch nicht weiter darauf eingegangen werden.

(a) Güter
Beispiel: *Die Waren aus der Artikelliste müssen zum Versand bereit sein.*
Die Aktivität hängt also davon ab, dass bestimmte Güter an einem bestimmten Ort vorhanden sind (z.B. Lager) und versendet werden können.

(b) Geräte/Werkzeuge/Maschinen
Beispiel: *Die Aktivität „Serienbrief drucken“ benötigt einen „Laser-Drucker“*
Die Aktivität verwendet bestimmte Objekte. Diese müssen daher verfügbar sein.

(c) Personen
Beispiel: *„Ein Mitarbeiter der Verwaltung mit Prozessänderungsrechten“*
Die Aktivität ist davon abhängig, dass Bearbeiter mit entsprechenden Qualifikationen auch vor Ort sind.

(d) Ressourcen-Eigenschaften
Jede Ressource kann unterschiedliche Eigenschaften haben, welche von Aktivitäten gefordert werden können. So kann z.B. ein Drucker Farbe oder nur Schwarz drucken, einen bestimmten Seitendurchsatz haben und unterschiedliche Papierformate und -stärken bedrucken. Ähnlich wie zustandsbehaftete Daten können auch Ressourcen Zustände besitzen. Ein Drucker hat u.a. die Zustände „Aus“, „Bereit“ und „Fehler“.

3. Abhängigkeiten von Vorgängen
Allgemeine Vorgänge können ebenfalls Abhängigkeiten bewirken, die von den bisher genannten Abhängigkeiten nicht abgedeckt werden.

(a) Abhängigkeit von anderen Vorgängen: Vorher/Nachher-Beziehungen
Beispiel: *Vor „Rechnung versenden“ muss „Rechnung drucken“ erfolgen.*
Die Aktivität ist davon abhängig, dass andere Vorgänge/Aktivitäten auch ausgeführt werden. Dabei handelt es sich nicht um eine Abhängigkeit durch einen Datenfluss oder durch Ressourcen.

(b) Abhängigkeit von anderen Vorgängen: Ergebnis
Beispiel: *Die Aktivität „Kredit erteilen“ setzt das Ergebnis „Antrag genehmigt“ der Aktivität „Kreditantragsprüfung“ voraus.*
Die Ausführbarkeit einer Aktivität kann auch von Ergebnissen anderer Aktivitäten abhängen. So kann eine Aktivität fordern, dass eine weitere Aktivität mit einem bestimmten Status beendet wurde.

(c) Abhängigkeit durch interne Vorgänge: Ausführungshäufigkeit
Beispiel: *„Mahnung versenden“ darf höchstens 3 mal ausgeführt werden.*
Die Aktivität darf aus internen Gründen nicht mehrmals ausgeführt werden. Dies kann sich z.B. aus einem internen Zustand ergeben. Im Beispiel ist ein solcher interner Zustand dadurch gegeben, dass höchstens drei Male gemahnt wird.

4. Zeitliche Abhängigkeiten
   Zeitliche Abhängigkeiten können grundlegend in zwei Klassen unterteilt werden: Abhängigkeiten, welche im Zusammenhang mit anderen Abhängigkeiten oder Aktivitäten auftreten, und aktivitäteninterne Abhängigkeiten. Insbesondere durch Schleifen und periodische Zeitangaben können sehr komplexe Abhängigkeiten entstehen. Beispielhaft sollen im Folgenden zwei typische Abhängigkeiten erläutert werden.

   (a) Min/Max-Abstände
   Beispiel: *Die Daten des Parameters „Blutbild“ dürfen nicht älter als ein Tag sein.*
   Beispiel: *Die Ressource „Drucker“ muss für mindestens drei Stunden verfügbar sein.*
   Beispiel: *Der Vorgang „Mahnung versenden“ darf frühestens drei Wochen nach „Rechnung versenden“ ausgeführt werden.*
   Die aufgeführten Beispiele zeigen zeitliche Minimal- und Maximalabstände in Kombination mit unterschiedlichen anderen Abhängigkeiten.

   (b) Aktivitäteninterne zeitliche Abhängigkeiten
   Beispiel: *„Löcher bohren“ darf max. drei Stunden durchgehend laufen (sonst läuft der Bohrer heiß).*
   Beispiel: *„Datenversand an das DATEV RZ“ ist nur zwischen 6 und 8 Uhr möglich.*
   Die Ausführung von Aktivitäten kann zeitliche Einschränkungen haben. Möglich sind z.B. Laufzeiten, konkrete Termine und Wiederholungs- sowie Ruhezeiten.

5. Abhängigkeiten durch Rechte
   Auch Rechte stellen ein sehr weitreichendes Gebiet dar. Hier sollen zunächst nur einige wenige Anforderungen angesprochen werden, welche für die Ausführung funktionalen Charakter haben. Das allgemeine „Recht zur Ausführung“ bzw. die Bearbeiterzuordnung gehört hier nicht dazu, da diese nicht unbedingt die Funktionalität einer Aktivität beeinträchtigen. Eine grundlegende funktionale Voraussetzung ist jedoch z.B., dass der Bearbeiter die Rechte haben muss, um benötigte Daten zu lesen bzw. zu ändern.

   (a) Rechte zum Lesen/Ändern von Daten
   Wenn eine Aktivität Daten benötigt oder schreibt, muss der Bearbeiter die entsprechenden Rechte besitzen.

   (b) Entscheidungsgewalt
   Falls eine Aktivität bei einer alternativen Verzweigung über den weiteren Verlauf entscheidet, muss der Bearbeiter auch die entsprechenden Rechte für die Entscheidung besitzen. Gleiches gilt auch bei vormodellierten Sprung- und Fehlerkanten. Beispiele für Entscheidungsbefugnis sind einfach zu finden, so z.B. in einem Krankenhaus im Prozess zur Behandlung eines Patienten. Dieser Aspekt ist insofern funktional, da die Aktivität selbst vielleicht von jedem Bearbeiter ausgeführt wer-

den könnte, es jedoch bestimmte Qualifikationen für eine korrekte Entscheidung bedarf.

6. Semantische Abhängigkeiten
   Als semantische Abhängigkeiten werden hier Abhängigkeiten bezeichnet, welche für die Funktion einzelner Aktivitäten keine Rolle spielen. Ein Beispiel hierfür ist die Unverträglichkeit zwischen unterschiedlichen Medikamenten.

Unter den vorgestellten Abhängigkeitsklassen sind unterschiedliche Kombinationen möglich. So kann es für die Ausführung einer Aktivität z.B. erforderlich sein, dass ihre Eingabedaten einem bestimmten Typ entsprechen und sich darüber hinaus noch in einem bestimmten Zustand befinden. Ein weiteres Beispiel sind Anforderungen an die Ausführungshäufigkeit, die sich auf eine andere Aktivität beziehen: Die Aktivität, um Geld über eine Rechnung gerichtlich einzufordern, hängt davon ab, dass zuvor die Aktivität, um Mahnungen zu verschicken, drei Male aufgerufen wurde.

Die Relevanz der in diesem Abschnitt vorgestellten Abhängigkeiten muss an realen Prozessen überprüft werden. Da die Integration der Abhängigkeiten auch einen Mehraufwand in der Modellierung und Analyse von Prozessen bedeutet, scheint eine Beschränkung auf die im realen Fall benötigten Abhängigkeiten sinnvoll. Die vorgestellte Klassifikation erhebt keinen Anspruch auf Vollständigkeit. Dennoch bietet sie eine Grundlage, um Probleme oder Teilprobleme einzuordnen.

3.4 Beispiele

Nachdem im vorhergehenden Abschnitt Abhängigkeiten vorgestellt wurden, werden einige von ihnen im Folgenden an Beispielprozessen veranschaulicht.

Vorlauf Der Prozess beschreibt den Teil eines Buchhaltungsvorgangs in der Bearbeitung eines Vorlaufs.

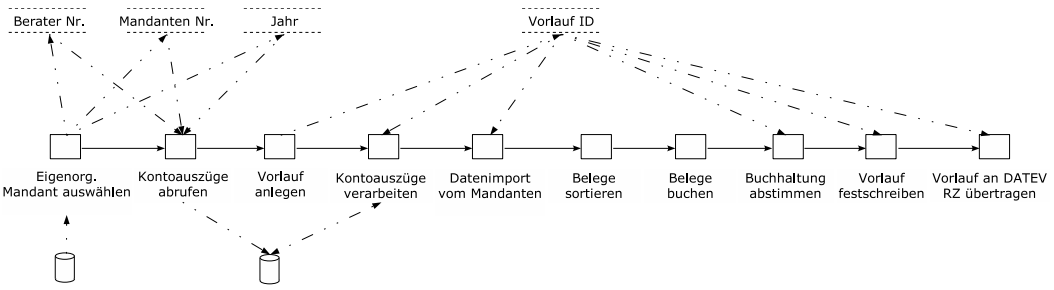

Abbildung 3.3: Bearbeitung eines Vorlaufs in einem Buchhaltungsvorgang

Wie in Abbildung 3.3 angedeutet, verwendet die Aktivität „Kontoauszüge verarbeiten“ Daten, die von der Aktivität „Kontoauszüge einlesen“ bereitgestellt werden. Der Datenfluss wird jedoch nicht durch das WfMS verwaltet sondern erfolgt intern. Des Weiteren werden die Daten (hier die Kontoauszüge) durch die verarbeitende Aktivität gelöscht. Wenn die Kontoauszüge also ein weiteres Mal verarbeitet werden sollen, müssen sie zuerst erneut eingelesen werden.
Eine besondere Bedeutung hat die Aktivität „Vorlauf festschreiben“. Nach ihrer Ausführung können und dürfen die Daten eines Vorlaufs nicht mehr geändert werden. Dies ist Voraussetzung, um den Vorlauf an DATEV übertragen zu können. Dies bedeutet, dass keine Aktivität mehr eingefügt werden darf, welche die Daten ändert. Auf der anderen Seite muss das WfMS es weiter erlauben, die Daten zu lesen, z.B. durch eine Aktivität, welche den Vorlauf ausdruckt.
Eine zeitliche Abhängigkeit ist in der Aktivität zur Übertragung an DATEV zu finden. Die Übertragung darf ausschließlich zu bestimmten Tageszeiten stattfinden.

Angebot Bei diesem Prozess (Abbildung 3.4) handelt es sich um einen Ausschnitt aus einem Ablauf für ein Angebot aus dem B2B-Bereich. Auf Anfrage des Kunden wird ein Angebot erstellt, von höherer organisatorischer Instanz kontrolliert und schließlich dem Kunden übermittelt. Dies wiederholt sich, bis der Kunde das Angebot akzeptiert und keine weiteren Änderungswünsche hat. Daraufhin wird abgewartet, ob der Kunde aufgrund des Angebots eine Bestellung in Auftrag gibt.
In diesem Beispiel sind semantische Abhängigkeiten zu erkennen. So sollte es nicht möglich sein, einem Kunden ein Angebot zuzusenden, welches noch keiner Durchsicht unterzogen wurde. Aus funktionaler Sicht ist diese Aktivität nicht notwendig, dennoch ist es ein für die Firma wichtiger Vorgang.

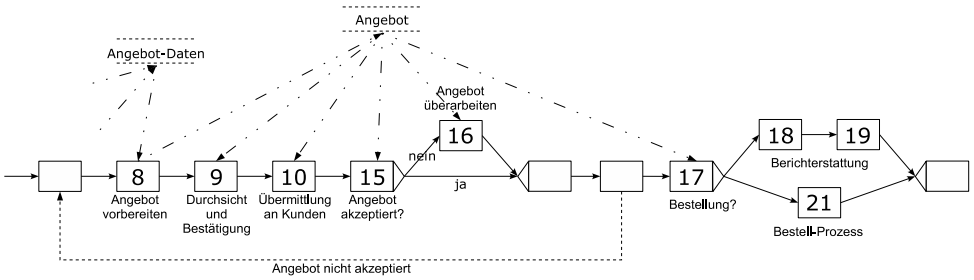

Abbildung 3.4: Erstellen eines Angebots

3.5 Weitere Aspekte und Anforderungen

Neben den vorgestellten Abhängigkeiten können weitere Forderungen von Seiten der Aktivitäten bestehen. So kann z.B. die Ausführung einer Aktivität einer bestimmten Version vorausgesetzt werden. Ebenso können „weiche“ Abhängigkeiten bestehen, d.h. der Benutzer kann im konkreten Fall entscheiden, ob die Abhängigkeit für die Ausführung relevant ist.

Bedingte Abhängigkeiten Abhängigkeiten können unter Voraussetzung einer anderen Abhängigkeit bestehen. Beispiel: *Falls „Ware nachbestellt“ ausgeführt wurde, muss vor „Bestellte Artikel versenden“ auch „Wareneingang bestätigen“ ausgeführt worden sein.* Um Abhängigkeiten dieser Art zu berücksichtigen, ist ein Konzept mit gemeinsamen Schnittstellen zur Repräsentation notwendig. Auf dieser Basis ließen sich zusammengesetzte Abhängigkeiten durch entsprechende Regeln definieren. Auf die Definition und Interpretation solcher Regeln soll in dieser Diplomarbeit jedoch nicht weiter eingegangen werden, da die entsprechenden Grundlagen noch geschaffen werden müssen.

Erklärungen von Seiten des Systems Die Abhängigkeiten und v.a. auch die Begründung, warum sie im konkreten Fall verletzt werden, müssen für den Benutzer von Seiten des Systems verständlich gemacht werden. Je komplexer die Abhängigkeiten gestaltet werden, desto besser müssen sie auch erklärt werden. Das spielt auch für den Prozess-Designer beim Modellieren komplexer Prozesse eine Rolle. Besonders wichtig sind Erklärungen jedoch bei Adhoc-Änderungen, welche von normalen Benutzern durchgeführt werden. Zusätzlich zur Erklärung sollten bei Verletzung von Abhängigkeiten auch Lösungsmöglichkeiten vorgeschlagen werden.

Kontrolle über die Einhaltung von Abhängigkeiten Manche Abhängigkeiten lassen sich zur Laufzeit nicht durch das WfMS kontrollieren. Im Gegensatz dazu lässt sich z.B. das Schreiben eines vom WfMS verwalteten Datums überprüfen, so dass im Fehlerfall die Ausführung ausgesetzt werden kann. Dies ist jedoch nicht bei allen Abhängigkeiten möglich. Die Abhängigkeiten können beschrieben werden. Anhand der Beschreibung können korrekte Prozesse modelliert werden. Das WfMS muss sich jedoch darauf verlassen, dass sich die Aktivitäten entsprechend ihrer Beschreibung verhalten. Wird die Aktivität „Nachricht verschlüsseln“ ohne Fehlermeldung beendet, muss das WfMS davon ausgehen, dass die Verschlüsselung erfolgreich ausgeführt wurde. Bei sicherheitskritischen Vorgängen sollten jedoch zusätzliche Prüfroutinen den tatsächlichen Status der Abhängigkeiten überprüfen können.

3.6 Integration versteckter Datenflüsse

Wie in Abschnitt 3.3 geschildert, können Aktivitäten bestimmte Daten zur Ausführung voraussetzen. In diesem Abschnitt werden versteckte Datenflüsse, also Datenflüsse, die nicht durch das WfMS verwaltet werden, etwas näher betrachtet. Das Problem der versteckten Datenflüsse wird auch in [2] beschrieben. In [2] schlagen Acker et al. zudem eine Lösung mittels virtueller Datenflüsse vor. In diesem Abschnitt wird der Einsatz des vorgeschlagenen Lösungsansatzes in ADEPT untersucht. Der Abschnitt zeigt, wie die Beschreibung einer Abhängigkeit, durch minimale Anpassungen des Prozessmodells, in ein bestehendes WfMS integriert werden kann.

Wie das Beispiel des Vorlauf-Prozesses (s. Abschnitt 3.4) der Steuerberatung zeigt, können eine Reihe von Daten unter Aktivitäten und Komponenten außerhalb des WfMS ausgetauscht werden. Im konkreten Beispiel liest z.B. die Aktivität „Kontoauszüge abrufen“ Kontoauszüge in eine Datenbank ein. Die Aktivität „Kontoauszüge verarbeiten“ liest diese Daten wieder aus. Der Datenfluss erfolgt absolut transparent für das WfMS. Falls ein Benutzer die Aktivität „Kontoauszüge abrufen“ z.B. bei einer Adhoc-Änderung löschen möchte, würde das WfMS dies zulassen. Das WfMS ist nicht in der Lage, die Verletzung der Abhängigkeit zu entdecken und Vorschläge zur Auflösung des Konflikts zu machen. Es ist leicht auszumalen, dass solche verletzten Abhängigkeiten kritische Fehler während der Prozessausführung zur Folge haben können.
Soll das WfMS mit versteckten Datenflüssen umgehen können, müssen Abhängigkeiten, welche durch versteckte Datenflüsse entstehen, ebenso wie konkrete Datenflüsse, auf Ebene der Aktivitäten modelliert werden.

Versteckte Datenflüsse haben die gleichen Eigenschaften wie normale Datenflüsse: Die Daten werden von Aktivitäten gelesen und geschrieben. Sie haben einen Datentyp und könnten über einen Bezeichner identifiziert werden. Um das Problem der versteckten Datenflüsse zu lösen, liegt für ADEPT daher eine Lösung auf Basis des vorhandenen Datenflussmodells nahe. Wie in Abschnitt 2.2 geschildert, umfasst das Beschreibungsmodell der Aktivitäten von ADEPT eine vollständige Lösung für normale Datenflüsse. Das Modell wird daher um virtuelle Datenelemente und virtuelle Parameter erweitert, wodurch ein sogenannter virtueller Datenfluss modelliert werden kann. Konzepte und Eigenschaften, wie obligat vs. optional oder die Nachforderbarkeit von Parametern, lassen sich ohne weiteres auf die virtuellen Konstrukte übertragen. Dadurch lassen sie sich in vorhandenen Algorithmen genauso wie die nicht virtuellen Konstrukte behandeln. Implementierungstechnisch bietet es sich an, die betroffenen Konzepte um ein Attribut zu erweitern, welches über die Virtualität Aufschluss gibt. Von dem Attribut unabhängige Algorithmen können so ohne Änderung übernommen werden.

Zur Integration von virtuellen Datenflüssen wird die Definition eines Eingabe- bzw. Aus-

gabeparameters P nach [11] um folgendes Attribut erweitert:
$IsVirtual^P$ = { TRUE oder FALSE }

Mit der Definition von Datenelementen wird analog verfahren. Abbildung 3.5 zeigt einen Ausschnitt aus dem Beispielprozess der Steuerberatung (vgl. 3.4), in dem die Datenübergabe der Kontoauszüge über einen virtuellen Datenfluss modelliert wird. Für die Darstellung von virtuellen Datenelementen wird eine grau hinterlegte Variante vorgeschlagen.

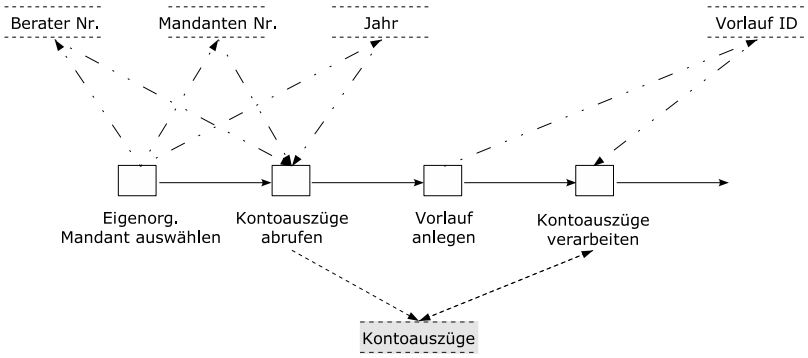

Abbildung 3.5: Modellierung eines versteckten Datenflusses mit einem virtuellen Datenelement

Mit Hilfe der virtuellen Datenelemente kann das WfMS über versteckte Datenflüsse in Kenntnis gesetzt werden. Die Korrektheit des Datenflusses kann mit vorhandenen Algorithmen sichergestellt werden. Zudem ist es aufgrund der Parameterbeschreibungen auch möglich, dem Benutzer erklärende Rückmeldungen zu Fehlern im Datenfluss zu geben.
Ein wichtiger Aspekt bleibt jedoch anzumerken: Im Unterschied zum normalen Datenfluss entzieht sich der versteckte Datenfluss immer noch der Kontrolle des WfMS. Dies bedeutet, dass sich das WfMS darauf verlassen muss, dass die Aktivitäten die modellierten schreibenden Zugriffe tätigt. Es kann nicht überprüft werden, ob die Daten auch tatsächlich vorhanden sind.

Neben der Modellierung von versteckten Datenflüssen können virtuelle Datenflüsse auch für die Modellierung anderer Abhängigkeiten eingesetzt werden. Damit ließen sich auf Aktivitätenebene auch explizite Forderungen über vorher ausgeführte Aktivitäten stellen. Es könnte ein spezieller Ausgabeparameter eingeführt werden, der für jede Aktivität einen schreibenden Zugriff auf ein virtuelles Datenelement beschreibt. Der Typ des virtuellen Datenelements wäre dann genau der eindeutige Bezeichner der Aktivität. Damit wäre es möglich zu modellieren, dass eine Aktivität die Ausführung einer anderen Aktivität erfordert. Dazu ist lediglich ein obligater virtueller Eingabeparameter mit dem Typ der erforderlichen Aktivität notwendig. Im Unterschied zu einer expliziten Modellierung eines solchen Zusammenhangs,

z.B. als kleiner Sub-Prozess, lassen sich auf diese Weise lose Abfolgen auf Ebene der Aktivitäten modellieren. Die Ausführungsreihenfolge wird in diesem Fall durch den Datenfluss bestimmt, wodurch nur die Reihenfolge der beteiligten Aktivitäten festgelegt wird.
Da es sich in diesem Fall noch nicht einmal um einen versteckten Datenfluss handelt, ist dies als eine rein semantische Erweiterung des Modells zu sehen. Dennoch können die dadurch modellierten Abhängigkeiten auch funktionalen Charakter haben. So könnte z.B. eine Aktivität „Paket versenden“ davon abhängig sein, dass die Aktivität „Paket zuschnüren“ vorher ausgeführt wurde. Abbildung 3.6 veranschaulicht die Aktivitätenbeschreibungen und das dazugehörige Prozessfragment.

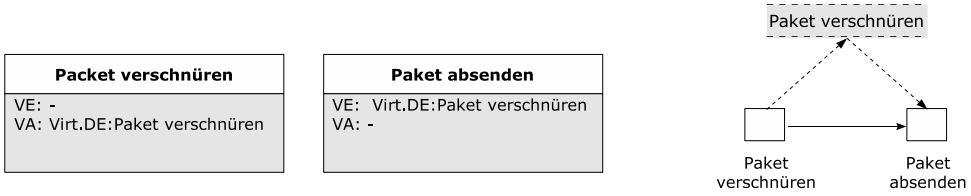

Abbildung 3.6: Modellierung einer semantischen Abhängigkeit durch einen virtuellen Datenfluss

Da es nicht möglich ist, den Datenfluss über virtuelle Datenelemente durch das WfMS zu kontrollieren, kann es zur Ausführungszeit zu Ausnahmesituationen kommen. Sobald einer Aktivität, z.B. durch einen Fehler in einer Komponente, Daten nicht zur Verfügung stehen, ist die weitere korrekte Ausführung des Prozesses nicht mehr sichergestellt. Nicht zuletzt bei geschäfts- oder sicherheitskritischen Prozessen kann dies fatale Auswirkungen haben. Eine Erweiterung des Modells ist daher empfehlenswert. So könnten benutzerdefinierte Prüfroutinen zugelassen werden, welche die für das WfMS sonst transparenten Datenflüsse und Parameterübergaben sicherstellen. Zeigt die Prüfroutine einen Fehler an, kann das WfMS die Ausführung des Prozesses an dieser Stelle aussetzen und den Prozessverantwortlichen informieren.

3.7 Semantische und funktionale Aspekte

Die zu lösenden und zu modellierenden Abhängigkeiten umspannen ein sehr breit gefächertes Gebiet. Schon allein durch funkionale Aspekte wird ein großer Problemraum aufgespannt. Da eine fehlerfreie Ausführung eine hohe Priorität hat, ist zunächst eine Konzentration auf die Funktionalität der Aktivitäten angebracht.
Ob es sich bei einer Abhängigkeit um einen funktionalen oder semantischen Aspekt handelt, lässt sich in vielen Fällen erst bei einer konkreten Implementierung sagen. Dadurch ergibt sich eine sehr große Grauzone von Abhängigkeiten, welche je nach Implementierung einen semantischen oder funktionalen Charakter besitzen. Im Allgemeinen lässt sich sagen, dass je mehr

sich eine Abhängigkeit von syntaktischen Beschreibungen entfernt, desto unwahrscheinlicher ist es, dass diese Abhängigkeit funktionaler Natur ist.

Semantisch höherstehende Abhängigkeiten ergeben sich z.B. durch notwendiges Hintergrundwissen. So könnten zwei Aktivitäten zur Verabreichung von Medikamenten aus funktionaler Sicht einwandfrei hintereinander ausgeführt werden. Bei entsprechenden Unverträglichkeiten zwischen den Medikamenten kann dies dennoch ein schwerwiegendes Problem darstellen. Die Last, das entsprechende Hintergrundwissen bereitszustellen und zu gebrauchen, liegt bis heute in den meisten Fällen bei den Benutzern. Eine Unterstützung durch das WfMS ist jedoch wünschenswert, da, vor allem bei umfangreichen Prozessen, die wenigsten Benutzer einen Überblick über den gesamten Prozess haben. Insbesondere da Prozesse durch Adhoc-Änderungen auch von dem erwarteten Verlauf abweichen können.

Weitere Beispiele für semantische Abhängigkeiten lassen sich im Bereich allgemeiner Geschäftsregeln (*Business Rules*) finden. Eine Aktivität um Mahnungen zu verschicken kann, rein funktional betrachtet, eine beliebige Anzahl an Mahnungen erstellen. Dass nach genau drei Mahnungen weitere Schritte eingeleitet werden sollen, kann rechtlich oder auch firmenpolitisch bedingt sein.

Die Möglichkeiten, um funktionale und semantische Abhängigkeiten zu integrieren, können sich überschneiden. Eine abstrakte semantische Beschreibung kann durchaus zur Modellierung einer funktionalen Abhängigkeit verwendet werden. Es muss daher unterschieden werden, ob die Abhängigkeit oder das Mittel zur Modellierung der Abhängigkeit einen funktionalen bzw. semantischen Charakter haben.

Ein typisches Beispiel für eine Abhängigkeit, welche in einer Grauzone zwischen Semantik und Funktionalität liegt, findet sich auch im Beispiel des Vorlauf-Prozesses (s. 3.4). Ein Vorlauf muss am Ende eines Prozesses entweder gelöscht oder an DATEV übertragen worden sein. Durch entsprechende Adhoc-Änderungen ließe sich eine Prozessinstanz beenden, ohne dass dieses Kriterium erfüllt ist. Das WfMS würde dies auch zulassen, da diese semantische Information nicht bekannt ist. Dennoch kann dies in anderen Prozessinstanzen, z.B. bei der Abrechnung, auch zu funktionalen Problemen führen.

3.8 Zusammenfassung und weitere Inhalte der Diplomarbeit

Im bisherigen Teil der vorliegenden Diplomarbeit wurden Ziele der Anwendung von “Plug & Play”-Konzepten in WfMS erläutert. Darüber hinaus wurden Probleme bei der Umsetzung dieser Konzepte erörtert. Insbesondere wurden mögliche Abhängigkeiten, die Aktivitäten bzw. Komponenten betreffen, identifiziert und klassifiziert.

Im Weiteren Verlauf dieser Arbeit soll eine der identifizierten Abhängigkeiten näher untersucht werden. Es sollen Konzepte erarbeitet werden, die eine Integration von zustands-

behafteten Daten in WfMS erlauben. Eine Lösung in dieser Richtung kann Grundlage für die Integration weiterer Abhängigkeiten sein, da es auf Zustände basierende Abhängigkeiten bzgl. Daten, Ressourcen und Komponenten gibt. Zustandsbehaftete Daten sind ein guter Startpunkt, da auf Konzepte zur Datenflussmodellierung zurückgegriffen werden kann.
Die erarbeiteten Konzepte werden auf das WfMS ADEPT aufbauen. Durch die klare Strukturierung, die Unterscheidung zwischen Knoten und ihnen zugeordneten Aktivitäten sowie der expliziten Modellierung des Datenflusses bietet ADEPT eine hervorragende Grundlage.

Kapitel 4

Zustandsbehaftete Daten

Wie bereits in Kapitel 3 erörtert, kann die Funktionalität einer Aktivität davon abhängig sein, dass benötigte Daten einer bestimmten Verarbeitung unterlaufen sind und sich dadurch in einem bestimmten Zustand befinden. Dies zeigt sich auch in einem einfachen Beispiel, wie dem Vorlauf-Prozess der Buchhaltung (s. Abschnitt 3.4). Bevor ein solcher Vorlauf an DATEV übermittelt werden kann, muss er festgeschrieben werden („Vorlauf festschreiben“). Das Festschreiben bedeutet, dass die Daten von keiner Aktivität mehr geändert werden dürfen. Dadurch ergeben sich zwei Datenzustände: „festgeschrieben“ und „nicht festgeschrieben“. Solche Abhängigkeiten mögen dem Prozess-Designer wohl bewusst sein, jedoch steht diese Information dem WfMS nicht zur Verfügung. Dies hat zur Folge, dass das WfMS auch die Ausführung eines Prozesses zulässt, welcher, aufgrund einer verletzten Abhängigkeit, keine Aussicht auf eine korrekte Beendigung hat.

Das oben genannte Beispiel stellt sicherlich keinen Einzelfall dar. Vielmehr ist es nicht schwer auszumalen, dass zustandsbehaftete Daten in vielen Prozessen eine Rolle spielen. Um in vielen realen Anwendungszenarien eingesetzt werden zu können, ist es daher notwendig, dass WfMS mit zustandsbehafteten Daten umgehen können. Im Folgenden wird in 4.1 auf die daraus resultierenden Anforderungen eingegangen. Ein erster Lösungsansatz wird in 4.2 vorgestellt.

4.1 Anforderungen

Aus dem Beispiel mit dem Vorlauf ergeben sich zwei grundlegende Anforderungen zur Umsetzung in einem WfMS. Zum einen muss es für Aktivitäten möglich sein, einen Zustand von Daten zu fordern. Das bedeutet, die Ausführung darf nur dann erfolgen, wenn sich die Daten im geforderten Zustand befinden. Des Weiteren müssen Aktivitäten den Zustand von Daten ändern können. Um den Anforderungen zu entsprechen, ergeben sich weitere Anforderungen

von Seiten des WfMS. So muss es für das WfMS möglich sein, den momentanen Zustand der Daten zu ermitteln. Von den Aktivitäten muss es zudem den neuen Datenzustand erfahren und diesen für die Daten setzen können.
Dies würde soweit genügen, um zur Laufzeit festzustellen, ob eine Aktivität ausführbar ist oder nicht. Da es während der Ausführung eines Prozesses jedoch möglichst zu keinen Überraschungen kommen soll, muss es schon zur Modellierungszeit möglich sein, genaue Aussagen zur Ausführbarkeit der Aktivitäten zu treffen. Dies bedeutet, dass die Aktivitäten vorgeben müssen, unter welchen Datenzuständen sie ausführbar sind und welche Datenzustände sie bewirken.

4.2 Ein erster Lösungsansatz

Ausgehend von den beschriebenen Anforderungen lässt sich ein einfacher Ansatz zur Integration von Datenzuständen formulieren. Dazu wird zunächst die Definition des Datenelements erweitert. Es werden zwei neue Attribute benötigt: Eines, welches während der Ausführung den momentanen Zustand speichert, und ein weiteres, welches alle möglichen Zustände des Datenelements enthält. Das Speichern aller existierenden Zustände ist nicht unbedingt notwendig. Dies ist jedoch hilfreich zur Erhaltung der Konsistenz und um das Überblicken der vorhandenen Zustände zu erleichtern.
Parameter und Datenkanten müssen um eine Zustandsforderung und eine optionale Zustandsänderung erweitert werden. Da für Parameter und Datenkanten gleiches gilt, wird an dieser Stelle davon abstrahiert. Beschreibungen werden sich im Folgenden auf Parameter beziehen, müssen jedoch ebenso auf Datenkanten angewandt werden. Die Datenkanten spielen nur in der graphischen Darstellung eine weitere Rolle. Sie werden mit Zustandsforderung und Zielzustand annotiert. Ein Zustand besteht ausschließlich aus einem eindeutigen Bezeichner.
Das Fordern und Setzen eines Zustands kann mit dem Lesen und Schreiben eines Datenelements verglichen werden. Fordert eine Aktivität einen Zustand, so möchte sie exakt diesen Wert vom Zustandsattribut des Datenelements lesen. Setzt eine Aktivität einen neuen Zustand, so schreibt sie diesen Wert in das Attribut. Dadurch ist eine Erweiterung vorhandener Algorithmen zur Sicherstellung der Korrektheit des Datenflusses mit wenig Aufwand verbunden.

Erweiterung des Datenelements ($DataElement$, DE):
$CurrentState^{DE}$ = { UNDEFINED oder $State$ }
$States^{DE}$ = {{ $State$ }}

Erweiterung des Parameters P (bzw. Datenkanten):
$State^{P}$ = { UNDEFINED oder $State$ }

Handelt es sich bei dem Parameter um einen Eingabeparameter, so ist $State^P$ eine Zustandsforderung. Im Falle eines Ausgabeparameters wird dieser Zustand gesetzt. Ist der Wert von $State^P$ UNDEFINED, bedeutet dies, dass die Aktivität vom Zustand der Daten unabhängig ist bzw. diesen nicht ändert.
Abbildung 4.1 stellt eine abstrakte Darstellung der Beschreibung einer Aktivität dar. Zu der Beschreibung von Eingabe (E) und Ausgabe (A) kommt der Bezeichner des Zustands hinzu.

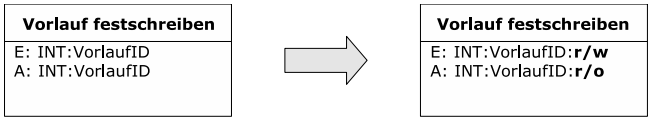

Abbildung 4.1: Die Ergänzung der Beschreibung einer Aktivität um Zustandsinformationen

Abbildung 4.2 zeigt anhand eines allgemeinen Beispiels, wie der Graph mit Zustandsforderungen dargestellt wird. In diesem Fall wird von Knoten a ein Zustand Z_1 im Datenelement d_1 erzeugt. Knoten b fordert diesen Zustand.

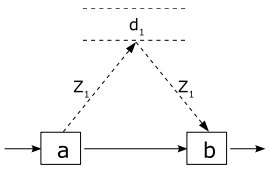

Abbildung 4.2: Setzen und Fordern von Zuständen

Der Ansatz bringt eine Einschränkung der Modellierung des Datenflusses mit sich: Am abschließenden Knoten eines Blocks darf nur ein Zustand möglich sein. Dies bedeutet, dass der letzte Zustand innerhalb eines alternativen Verzweigungsblocks immer der gleiche sein muss. Notwendig ist dies, da ansonsten zur Modellierungszeit nicht sicher festzustellen ist, welcher Zustand nach dem Block gesetzt ist. Eine darauffolgende Aktivität kann jedoch einen einzelnen Zustand voraussetzen. In Anlehnung an die Datenfluss-Strukturierungsregeln DF-1 bis DF-3 aus [11] wird im Folgenden eine ergänzende Regel DF-4* definiert.

Definition 4.1 (DF-4*). *Wird in einer Oder-Verzweigung der Zustand eines Datenelements d geändert, muss am Ende des Blocks in allen Zweigen der gleiche Zustand gesetzt sein.*

Für parallele Verzweigungen sind keine weiteren Einschränkungen notwendig, da parallele Schreibzugriffe durch DF-2 (s. [11, S. 122]) verhindert werden. Erlaubt hingegen sind parallele Schreibzugriffe, deren Reihenfolge durch Synchronisationskanten festgelegt ist. Durch diese Festlegung ist auch sichergestellt, dass am Ende eines parallelen Blocks immer der gleiche Zustand gesetzt ist, auch wenn nicht jeder schreibende Zugriff den Zustand ändert.

Auch im Falle von Schleifen muss der Datenfluss eingeschränkt werden. Hier gilt, dass am Ende der Schleife der gleiche Zustand gelten muss wie zu Anfang. So kann sichergestellt werden, dass der Zustand für jede Schleifenausführung passend ist.

Definition 4.2 (DF-5*). *Wird innerhalb eines Schleifenblocks der Zustand eines Datenelements d geändert, muss am Ende eines Durchlaufs der Zustand des Datenelements wieder gleich dem Zustand sein, der am Anfang des Blocks gegolten hat.*

Die Versorgung eines Parameters mit Zustandsforderung muss algorithmisch sichergestellt werden. Die Erweiterung des Algorithmus von Reichert in [11] ist minimal. Zum Wert der Versorgung (Supplied, Counter) muss zusätzlich noch der Wert des geschriebenen Zustands weitergereicht werden. Da durch DF-4* am Ende eines Blocks immer nur ein einzelner Zustand möglich ist, wird an dieser Stelle das Attribut höchstens mit dem gleichen Wert überschrieben. Auch muss der Algorithmus weiterhin nicht explizit mit Schleifen umgehen. Da durch DF-5* zu Beginn und am Ende einer jeden Schleifenausführung der gleiche Zustand vorliegt, genügt ein einfaches Folgen des Kontrollflusses ohne Wiederholung, um den Zustand zu ermitteln.
Bei dem hier aufgeführten Algorithmusteil handelt es sich um eine Erweiterung des Algorithmus 5-1 aus [11, S. 126]. Ergänzungen sind fett gesetzt. Dieser Teil des Algorithmus beschäftigt sich mit Prozessen ohne Synchronisationskanten und dient hier dazu, zu zeigen, wie gering die notwendigen Änderungen sind. Auch der vollständige Algorithmus erfährt im Prinzip die gleichen Anpassungen.

$WriterExists(CFS, DFS, n_{read}, d, state)$

1. Bestimme die Menge $c_pred*(n_{read})$ aller Vorgänger des Knotens n_{read} (bzgl. Kontrollkanten des Typs CONTROL_E) Initialisiere für alle Knoten $n \in N$ das Attribut $Supplied(n)$ mit $False$ und den Zähler $Counter(n)$ mit 0 **und das Attribut** $State(n)$ **mit UNDEFINED**.

2. Initialisiere die Menge der aktuell zu betrachtenden Knoten mit der leeren Menge: $NodeList = \varnothing$. Für jeden obligaten Schreibzugriff auf das Datenelement d durch einen Knoten $n \in Pred$ (bzw. einen vor-/nachgeschalteten Hilfsdienst von n) führe die folgende Anweisung aus: Inkrementiere für jeden direkten Nachfolgerknoten $n*$ von n (bzgl. Kanten des Typs CONTROL_E), der innerhalb der Menge $Pred \cup \{n_{read}\}$ liegt, den Zähler $Counter(n*)$ um 1, **falls** $State^P$ **von** n **nicht UNDEFINED ist**. **Setze** $State(n*)$ **auf** $State^P$ **des Ausgabeparameters von** n und füge den Knoten $n*$ der Menge $NodeList$ hinzu. ($NodeList$ enthält im Anschluß nur Knoten, für die gilt: d wird in mindestens einem der in diesen Knoten einmündenden Ausführungszweige sicher versorgt).

3. Falls $NodeList = \varnothing$ gilt, gehe zu Schritt 5. Andernfalls: Initialisiere die Menge der in der nächsten Iteration zu betrachtenden Knoten mit der leeren Menge ($NewNodeList :=$

$\varnothing$). Überprüfe für jeden Knoten der Menge $NodeList$, ob bei seiner Aktivierung das Datenelement d sicher versorgt ist: Für einen Knoten $n \in NodeList$ trifft dies genau dann zu, wenn er entweder eine der beiden Eingangssemantiken ONE_Of_ONE bzw. ALL_Of_ALL aufweist [vgl. (1)] oder wenn er die Eingangssemantik ONE_Of_ALL besitzt und sein Zähler $Counter(n)$ gleich der Anzahl der in n einmündenden Kontrollkanten bzw. Teilzweige ist [vgl. (2)]. Ist eine dieser beiden Bedingungen erfüllt, so setze das Attribut $Supplied(n)$ auf den Wert $True$ und führe für jeden direkten Nachfolger $n*$ von n wiederum die folgende Anweisung aus: Falls $n* \in Pred \cup n_{read}$ und $Supplied(n*) = False$ gilt, so erhöhe den Zähler $Counter(n*)$ um 1, **setze** $State(n*)$ **auf** $State^P$ **des Ausgabeparameters von** n und füge den Knoten $n*$ der Menge $NewNodeList$ ($\rightarrow$ duplikatfrei) hinzu.

4. Falls $NewNodeList = \varnothing$ oder $Supplied(n_{read}) = True$ gilt, so gehe zu Schritt 5. Andernfalls setze $NodeList := NewNodeList$ und wiederhole Schritt 3 (d.h. propagiere die Markierungen weiter entlang des Kontrollflusses).

5. Die Datenversorgung des obligaten Lesezugriffs von n_{read} auf d ist genau dann sicher gestellt, wenn für die Markierung dieses Knotens $Supplied(n_{read}) = True$ gilt ($\rightarrow$ Algorithmus liefert Rückgabewert $WriterExists(CFS, DFS, n_{read}, d) = True$). **Falls** $State^P$ **von** n_{read} **ungleich UNDEFINED ist, muss zustäzlich** $State(n_{read})$ **gleich** $State^P$ **von** n_{read} **sein.** Andernfalls kann der Lesezugriff nicht in jedem Fall von Vorgängerknoten (bzgl. normaler Kontrollkanten) versorgt werden ($\rightarrow$ Rückgabewert $WriterExists(CFS, DFS, n_{read}, d) = False$)

Durch das Festlegen der Zustände auf der Ebene von Aktivitäten- und Prozessvorlagen kann der momentane Zustand zu einem bestimmten Ausführungsschritt immer festgestellt werden. Dies ermöglicht nicht nur eine Prüfung zur Modellierungszeit, sondern ebenso auch bei der Migration von laufenden Instanzen sowie Adhoc-Änderungen durch den Benutzer.

4.3 Bewertung und weitere Anforderungen

Der vorgestellte Ansatz ist für eine überschaubare Menge von Zuständen mit wenigen beteiligten Aktivitäten gut einsetzbar. Abbildung 4.3 zeigt einen Ausschnitt des Vorlauf-Prozesses, modelliert mit Zustandsforderungen und -änderungen. Die Aktivität „Vorlauf festschreiben“ fordert den Zustand „r/w“ und ändert diesen auf „r/o“. Es handelt sich hierbei um zwei Datenkanten; „i:“ und „o:“ stehen hier für Eingabe bzw. Ausgabe.

Die Einsetzbarkeit des vorgestellten Ansatzes ist begrenzt. Probleme bereitet unter anderem, dass es nicht möglich ist, die Übergänge zwischen Zuständen einzugrenzen. So wäre z.B. eine Aktivität einsetzbar, welche den Zustand der Daten auf „r/w“ setzt, obwohl dies aus dem

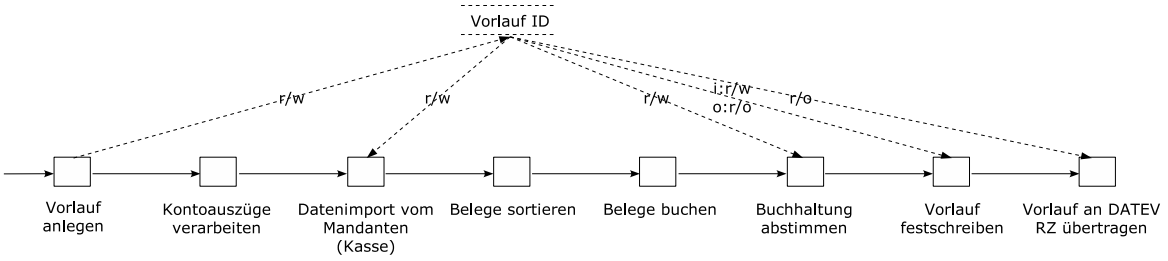

Abbildung 4.3: Vorlaufprozess mit Zustandsattribut

Zustand „r/o“ heraus nicht zulässig wäre. Daher auch die Einschränkung auf wenige Zustände und Aktivitäten. Der Modellierer muss die Zusammenhänge zwischen den Zuständen selbst im Auge behalten. Dazu gehört nicht nur, dass bestimmte Zustände nicht mehr erreicht werden dürfen. Bei mehreren Zuständen ist auch eine Reihenfolge im Erreichen der Zustände von Bedeutung. Dies kann schnell unübersichtlich werden und sollte daher durch das WfMS unterstützt werden.
Eine weitere Schwäche ist die Begrenzung der Parameterbeschreibungen auf einzelne Zustände. So ist es z.B. denkbar, dass eine Aktivität zur Prüfung eines Kreditantrags, je nach Ausgang der Beurteilung, ein Datum „Antrag“ in einen anderen Zustand versetzt. Eine daraufhin folgende Aktivität wiederum muss die Möglichkeit haben, auf eine Auswahl dieser möglichen Zustände reagieren zu können. Diese Beschränkung macht auch die Strukurierungsregeln zur Verhinderung mehrfacher Zustände am Ende eines Blocks notwendig. Jedoch sind gerade an dieser Stelle unterschiedliche Zustände typisch, dienen doch z.B. Oder-Verzweigungen auch der Auswahl unterschiedlicher Verarbeitungswege der Daten.

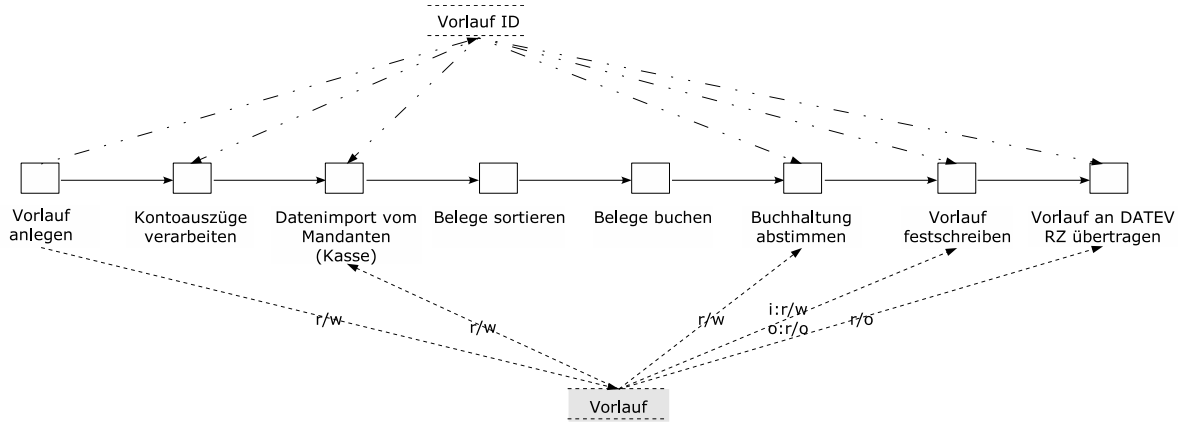

Abbildung 4.4: Vorlaufprozess mit Zuständen in einem virtuellen Datenelement

Eine Alternative zur Erweiterung des Datenelements soll der Vollständigkeit halber erwähnt werden. Anstelle eines neuen Attributs mit dem Zustand wäre es auch denkbar, den bisher nicht verwendeten Wert eines virtuellen Datenelements zu verwenden. Diese Idee wurde jedoch

schnell verworfen, da sie Probleme birgt. Falls Daten und Zustand in zwei unterschiedlichen Datenelementen gespeichert würden, müssten diese einander Zugeordnet werden. So lange es nur ein Datenelement vom gleichen Typ gibt, ist dies kein Problem. Sobald jedoch, z.B. durch eine Adhoc-Änderung, ein weiteres Datenelement eingefügt wird, muss diese Zuordnung bei weiteren Änderungen u.U. von Hand geschehen. Ein weiterer Nachteil ist, dass beinahe doppelt so viele Datenkanten benötigt werden, was die graphische Darstellung schnell unübersichtlich werden lässt. Abbildung 4.4 verdeutlicht die Situation.

Der Ansatz zur Formulierung virtueller Datenflüsse bietet in der vorliegenden Form keine Möglichkeit, bereits gesetzte Zustände zurückzusetzen. In einer Erweiterung könnte dies über ein spezielles Schlüsselwort in $State^P$ bei einem schreibenden Parameter geschehen. Inwiefern diese Möglichkeit sinnvoll ist, müssen jedoch zunächst konkrete Anwendungsfälle zeigen.

Kapitel 5

Zustandsbeschreibungen mittels Zustandsautomaten

Nachdem in Kapitel 4 ein erster Lösungsansatz eingeführt und bewertet wurde, wird im Folgenden ein weiterer Ansatz vorgestellt. Dieser nimmt Probleme in Angriff, mit denen der einfache Lösungsansatz nicht umgehen kann. Die Zustandsdefinition, die im vorangehenden Kapitel vorgestellt wurde, umfasste eine lose Menge von Zuständen ($States^{DE}$). Sollen jedoch Zusammenhänge zwischen den Zuständen dargestellt werden, bieten sich Digraphen zur Modellierung an. Mittels Digraphen kann festgelegt werden, zwischen welchen Zuständen Übergänge stattfinden dürfen (Abbildung 5.1). Ein Zustandsübergang von „festgeschrieben“ nach „nicht festgeschrieben“ (s. 4) kann somit durch das WfMS als ungültig erkannt und dadurch verhindert werden. Die neu eingeführte Zustandsdefinition entspricht daher einem einfachen Zustandsautomaten. Durch die Verkettung der Zustände kann ein Verarbeitungsweg für Daten vorgeschrieben werden. Typischerweise haben Wege auch einen Beginn und ein Ende. Auch in diesem Fall ist dies eine Anforderung an das Modell. So kann eine Forderung für den Vorlauf-Prozess lauten, dass die Daten am Ende des Prozesses entweder an DATEV übertragen wurden oder der Vorlauf gelöscht wurde. Das Beispiel mag nach einer rein semantischen Erweiterung aussehen, da es nicht direkt die Funktionalität von Aktivitäten betrifft. Dennoch ist es für einen korrekten Gesamtablauf essentiell.

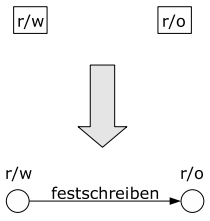

Abbildung 5.1: Von losen Zuständen zu Zustandsautomaten

Eine weitere Anforderung ist, dass sich der Zustand auch durch lesenden Zugriff ändern lassen muss. Wichtig ist dies vor allem, wenn nur eine Referenz der Daten verwaltet wird. Der schreibende Zugriff ist dem WfMS in diesem Fall nicht bekannt. Ein weiterer Anwendungsfall sind Zustände, welche tatsächlich durch lesenden Zugriff erreicht werden. Sie können sich z.B. auf einen Druckvorgang beziehen, bei dem die Daten selbst nicht geändert werden, jedoch der Zustand „gedruckt“ erzeugt wird.

Die feste Verbindung zwischen Zustandsdefinition und Datenelement muss gelöst werden. Da Aktivitäten im Allgemeinen in unterschiedlichen Prozessen verwendet werden, wird die dazugehörige Zustandsdefinition in mehreren Prozessen benötigt. Daher bietet sich eine zentrale Verwaltung von Zustandsdefinitionen an, so dass sie aus mehreren Prozessen heraus referenziert werden können. Ein Datenelement muss dann nur noch eine Referenz auf eine solche Definition halten.

Des Weiteren muss es möglich sein, mehr als einen Zustand als Zustandsforderung und Zielzustand angeben zu können. Mehrere Zustandsforderungen bedeuten, dass einer dieser Zustände eintreten muss. Die Situation, dass an einer Position im Kontrollfluss unterschiedliche Zustände erreicht werden, tritt z.B. durch alternative Verzweigungen ein. Durch mehrfache Zustandsforderungen ist es, anders als beim ersten Lösungsansatz, nicht mehr notwendig, dass am Ende eines Blocks ein einzelner Zustand feststeht. Zudem kann dadurch eine Aktivität für jeden geforderten Zustand einen anderen Folgezustand anzugeben. Aktivitäten, wie z.B. eine Auftragsprüfung, können unterschiedliche Ausführungsergebnisse besitzen. Um dies auch in den Zuständen widerspiegeln zu können, ist es notwendig, dass für je einen geforderten Zustand, mehrere mögliche Folgezustände angegeben werden können. Hier muss erst zur Laufzeit entschieden werden, welcher der angegebenen Zustände tatsächlich eintritt.
Eine Aktivität kann bzgl. eines Parameters also aktiviert werden, sobald die Daten im richtigen Zustand sind. Die Aktivität entscheidet während ihrer Ausführung, welcher konkrete Folgezustand eintritt. Um eine Ausführbarkeitsanalyse zu ermöglichen, muss die Menge der möglichen Folgezustände jedoch schon zur Modellierungszeit feststehen.

Im Folgenden wird in Abschnitt 5.1 auf die erweiterte Zustandsdefinition und in Abschnitt 5.2 auf die Parameterbeschreibung eingegangen. Nach einigen Beispielen wird das dynamische Verhalten in 5.3 erläutert. Im Anschluss werden in 5.4 Kriterien und Algorithmen für den Ansatz vorgestellt. Eine kritische Betrachtung des Ansatzes folgt in Abschnitt 5.5. Mit einer Zusammenfassung in 5.6 schließt dieses Kapitel ab.

5.1 Definition des Zustandsgraphen

Die Zustandsdefinition beinhaltet alle Informationen bezüglich der Zustände. Dazu gehören unter anderem ein Bezeichner, eine Beschreibung und ein Zustandsgraph. Der Zustandsgraph

ist ein Digraph, dessen Knoten Zustände und dessen Kanten Zustandsübergänge (Transitionen) darstellen. In Anlehnung an einen Verarbeitungsweg für die Daten wird der Bezeichner für die Zustandsdefinition *DataProcessingTemplate* (Abkürzung sei DPT) genannt.
Da eine Zustandsdefinition von mehreren Prozessen verwendet wird, sind ein Name und eine Beschreibung angebracht. Diese unterstützen den Modellierer bei der Suche nach einer passenden Zustandsdefinition und erleichtern das Verstehen eines bereits vorhandenen Prozesses. Um die Verwendung des *DataProcessingTemplate* weiter zu erleichtern, kann die Zustandsdefinition mit typischen benutzerdefinierten Datentypen und typischen Datenelementbezeichnern versehen werden.

$DataProcessingTemplate = \{Name^{DPT}, Description^{DPT}, States^{DPT}, Transitions^{DPT}, StartStates^{DPT}, EndStates^{DPT}, Usage^{DPT}\}$

$States^{DPT}$	Eine nicht leere Menge von Zuständen (*State*). Jeder Zustand muss über eine Transition mit mindestens einem weiteren Zustand verbunden sein.
$Transitions^{DPT}$	Eine nicht leere Menge von Zustandsübergängen (*Transition*) mit Zustandstupeln aus $States^{DPT}$: $Transitions^{DPT} \subseteq \{States^{DPT} \times States^{DPT}\}$.
$StartStates^{DPT}$	Eine nicht leere Teilmenge von $States^{DPT}$, welche als Startzustände dienen können.
$EndStates^{DPT}$	Eine Teilmenge von $States^{DPT}$, die Endzustände repräsentieren.
$Usage^{DPT}$	Eine Liste von typischen benutzerdefinierten Datentypen und Datenelementbezeichnern, für welche die Zustandsdefinition verwendet wird.
State	Ein Zustand (Abk. S) bestehend aus einem eindeutigen Bezeichner und einer optionalen Beschreibung.
Transition	Ein Zustandsübergang (Abk. T) , mit einem Bezeichner, einer optionalen Beschreibung und einem Tupel unterschiedlicher Zustände: $\{RequiredState^T, GoalState^T\}$.

Jeder der Zustände in der Menge $States^{DPT}$ enthält einen Bezeichner und kann durch eine Beschreibung ergänzt werden. Dies ist wichtig, um dem Benutzer informative Erklärungen geben zu können, falls z.B. eine Forderung nach einem Zustand verletzt wird. Gleiches gilt für die Beschreibung der Transitionen. Zusätzlich muss für eine Transition auch ein geforderter Zustand und ein Zielzustand angegeben werden. Im Zustandsgraph wird dieses Tupel mit einer gerichteten Kante dargestellt. Mit der Beschreibung der Transitionen können Alternativen gefunden und dem Benutzer erklärt werden, wie die Verarbeitung weiter gehen kann, um

z.B. einen Konflikt zu lösen.
Durch die $StartStates^{DPT}$ können mögliche Startzustände festgelegt werden. Die Startzustände sind beliebige Zustände aus $States^{DPT}$, welche durch eine Aktivität initial gesetzt werden können. Da jedes zustandsbehaftete Datum auch einen anfänglichen Zustand haben muss, ist die Angabe eines Startzustandes Pflicht.
Optional können Zustände auch als Endzustände gekennzeichnet werden. Diese befinden sich dann in der Menge $EndStates^{DPT}$. Endzustände erlauben eine Kontrolle darüber, ob sich die Verarbeitung der Daten am Ende eines Prozesses in einem für sie vorgesehenen Zustand befindet. Dies dient als Hilfe für den Bearbeiter des Prozesses und darf nicht vom System erzwungen werden. So können Benutzer darüber informiert werden, wenn der Zustand nicht erreicht wird, aber dennoch die Entscheidungsgewalt über die weitere Ausführung behalten. Da der Endzustand unter Umständen keine Rolle spielt oder die Daten keinen finalen Zustand besitzen, ist die Angabe von Endzuständen optional.
Abbildung 5.2 zeigt eine Zustandsdefinition, wie sie für einen Vorlauf aus dem Prozess der Steuerberatung (s. Abschnitt 3.4) aussehen könnte. Im Unterschied zum einfachen Ansatz kann hier ein Zustandswechsel von „*r/w*“ nach „*r/o*“ ausgeschlossen werden. Ein festgeschriebener Vorlauf kann somit nicht mehr weiter geändert werden.

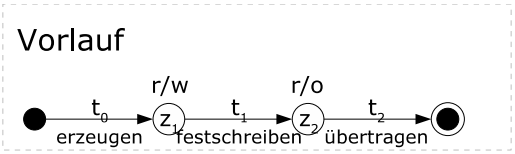

Abbildung 5.2: Zustandsdefinition für den Vorlauf

5.2 Erweiterung der Parameter von Aktivitäten

Die Beschreibung der Eingabe- und Ausgabeparameter von Aktivitäten nach [11] muss um Zustandsforderungen und Folgezustände erweitert werden, um die Datenzustände zu integrieren. In ADEPT wird explizit zwischen Eingabe- und Ausgabeparametern unterschieden. Wenn eine Aktivität ein Datenelement liest und schreibt, drückt sich dies daher in zwei Parametern aus. Werden an beiden Parametern Zustandsforderungen erlaubt, müssten diese in einer bestimmten Reihenfolge abgearbeitet werden. Dadurch wären auch Zustandsübergänge während der Ausführung möglich, wie Abbildung 5.3 veranschaulicht. Der Einfachheit halber wird deshalb festgelegt, dass nur einer dieser Parameter eine Erweiterung um eine Zustandsbeschreibung enthalten darf. Falls optionale und obligate Parameter vorliegen, ist die Zustandsbeschreibung über den obligaten Parameter zu finden.
Da unterschiedliche Zustandsdefinitionen für die Daten existieren können, kann ein Parameter

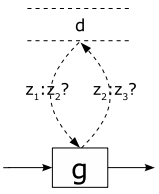

Abbildung 5.3: Mehrere Parameter für ein einzelnes Datenelement

auch mehrere Zustandsbeschreibungen besitzen. Es darf jedoch nur genau eine Zustandsbeschreibung für eine Zustandsdefinition hinterlegt werden .

$StateRequirements^P = \{\{Template^{SR}, StateList^{SR}\}\}$

$Template^{SR}$	Eine Referenz auf das *DataProcessingTemplate*, auf welches sich die Zustandsforderungen und Folgezustände in der $StateList^{SR}$ beziehen.
$StateList^{SR}$	Eine Menge bestehend aus $\{RequiredState, GoalStates\}$. *RequiredState* ist eine Voraussetzung, um die Aktivität ausführen zu können. Als spezieller Zustand kann *Any* angegeben werden, falls die Aktivität für jeden Zustand des Datenelements ausführbar ist. *GoalStates* ist eine Menge von Zuständen, welche durch die Ausführung erreicht werden können, sofern der geforderte Zustand galt.

Aktivitäten können in der Zustandsbeschreibung der Parameter mehrere Zustände fordern. Damit die Aktivität ausführbar ist, muss einer dieser geforderten Zustände eintreten. Als besonderer Zustand kann das Schlüsselwort *Any* angegeben werden. In diesem Fall ist jedoch die Angabe eines Folgezustands nicht erlaubt. Diese Option ist besonders nützlich, wenn eine Aktivität zwar immer aufrufbar ist, den Zustand der Daten jedoch nur in bestimmten Situationen ändert.

Für jeden geforderten Zustand können mehrere Folgezustände angegeben werden. Dies bedeutet, dass für den entsprechenden geforderten Zustand einer der *GoalStates* gesetzt wird. Welcher dieser Zustände tatsächlich eintritt, entscheidet sich erst zur Laufzeit. Dadurch ist es notwendig, dass die Aktivität das WfMS von dem gesetzten Zustand in Kenntnis setzt.

Als mögliche Folgezustände dürfen nur direkte Folgezustände des geforderten Zustands oder der geforderte Zustand selbst angegeben werden. Dadurch wird die Einhaltung des durch den Zustandsgraphen beschriebenen Verarbeitungsweg sichergestellt. Die Angabe des gefor-

derten Zustands als Zielzustand macht das Setzen eines neuen Zustands optional. Im Falle eines hierarchisch aufgebauten Prozesses muss es jedoch eine Ausnahme geben. Ist einem Knoten keine Aktivität sondern ein Sub-Prozess hinterlegt, so sind in dessen Parameterbeschreibungen Zustandssprünge erlaubt. Voraussetzung hierfür ist, dass im Zustandsgraphen mindestens ein Weg zwischen dem geforderten und dem Zielzustand existiert. Zudem muss der Sub-Prozess die Einhaltung der Zustandsübergänge sicherstellen. Um solche transitiven Zustandsübergänge in der graphischen Darstellung schnell erkennen zu können, werden sie mit einem $*$ versehen.

Anstelle einer Menge von Folgezuständen könnte ebenso eine Menge von Transitionen angegeben werden. Wie gezeigt, ist jedoch spätestens bei Sub-Prozessen ein Zustandssprung unumgänglich. Die Angabe von Transitionen wäre an dieser Stelle hinderlich. Des Weiteren haben Zustände den Vorteil, dass der Folgezustand direkt ersichtlich ist und nicht erst aus dem Zustandsgraphen abgelesen werden muss. Dadurch kann die Konsistenz der Zustandsübergänge in einer graphischen Darstellung schnell erkannt und nachvollzogen werden.

Von Datenkanten wird weiterhin abstrahiert. Für sie gelten jedoch auch die bisher angesprochenen Erweiterungen der Parameter. Wie schon im ersten Ansatz, werden sie in der graphischen Darstellung mit Zustandsforderung und Zielzuständen annotiert. Dabei wird für jede Forderung eine eigene Zeile verwendet. In jeder Zeile steht der *RequiredState*. Falls *GoalStates* existieren, stehen diese hinter dem *RequiredState* und einem Doppelpunkt in einer kommaseparierten Liste. Transitive Zustandsübergänge werden durch ein $*$ gekennzeichnet. Beispiel: „$RequiredState : State1, State2*$“.
Eine abstrakte Darstellung einer Aktivität mit einer Zustandsforderung nach dem erweiterten Ansatz wird in Abbildung 5.4 gezeigt.

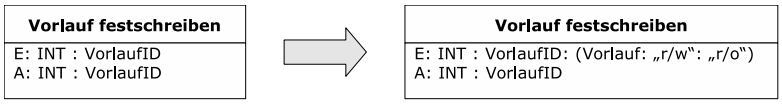

Abbildung 5.4: Aktivität mit erweiterter Zustandsbeschreibung

Die Parameterbeschreibung muss noch um den Ausgabeparameter für den Folgezustand erweitert werden. Dieser Wert spielt ausschließlich zur Ausführungszeit eine Rolle. Falls die Aktivität den Zustand der Daten ändert, gibt dieser spezielle Ausgabeparameter den neuen Zustand zurück. Der Wert muss spätestens dann vorliegen, wenn die Aktivität in den Zustand COMPLETED wechselt.

$OutputState^P$	Enthält den neuen Folgezustand nach Ausführung der Aktivität. Der Wert ist UNDEFINED, falls die Aktivität den Zustand der Daten nicht ändert. Wird der Zustand geändert, muss der neue Zustand aus der Menge $GoalStates$ aus der $StateList^{SR}$ sein, welche zum entsprechenden und eingetretenen Eingangszustand $RequiredState$ gehört.

5.3 Beispiele zur Demonstration des Konzepts

Bevor auf konkrete Algorithmen zur Überprüfung der Einhaltung von Zustandsforderungen eingegangen wird, soll an dieser Stelle der Ansatz anhand von Beispielen erläutert werden. Die bereits vorgestellte Abhängigkeit im Vorlauf-Prozess der Steuerberatung lässt sich mit dem neuen Zustandsmodell sauberer modellieren. Unter Verwendung der bereits in Abbildung 5.2 gezeigten Zustandsdefinition, lässt sich Modellierung der Aktivitäten an die benötigten Zustände anpassen. Soll an einer gegebenen Stelle im Prozess eine Aktivität eingefügt werden, so kann das System die Zustände, welche zur Ausführung an dieser Stelle gelten, ermitteln. Entsprechen diese Zustände nicht den geforderten Zuständen der Aktivität, so wird das Einfügen nicht ermöglicht. Das WfMS kann also verhindern, dass unpassende Aktivitäten, wie z.B. eine Änderung an einem Vorlauf nach dem Festschreiben, eingefügt werden. Durch die Forderung zur Einhaltung der Zustandsübergänge werden schon während der Spezifikation einer Aktivität Fehler unterbunden. Das Erstellen einer Aktivität, welche den Zustand von „r/o“ auf „r/w“ zurücksetzt, ist daher nicht möglich.

Das vorgestellte Modell kann auch zur Modellierung semantischer Abhängigkeiten verwendet werden. Als Beispiel dient hier der bereits beschriebene Angebot-Prozess (s. Abb. 3.4). Die Verarbeitung eines „Angebot“ erfordert, dass das Angebot nach der Erstellung einer Durchsicht unterzogen wird, bevor es zum Kunden geschickt werden kann. Dieser kann Wünsche bzgl. des Angebots äußern oder es akzeptieren. Falls das Angebot noch einmal überarbeiet wird, muss erneut eine Durchsicht erfolgen. Abbildung 5.5 zeigt eine zur Situation passende Zustandsdefinition. Über die Zustandsparameter können Aktivitäten beschreiben, welchem Verarbeitungsschritt sie entsprechen. Dadurch können passende Aktivitäten über Zustand der Daten und ein Aktivitäten-Repository, durch entsprechende Anfragen, gefunden werden. Abbildung 5.6 zeigt den fertig modellierten Angebot-Prozess mit Zustandsinformationen. Es ist nicht mehr möglich, dass ein Angebot an den Kunden versendet wird, ohne dass es vorher durchgesehen und bestätigt wurde. Allerdings zeigt sich in der Modellierung auch eine Schwäche des Konzepts. Aktivität 15 ändert den Zustand des Datenelements in Abhängigkeit des Ausführungsergebnisses. Ebenso hängt davon die Entscheidung ab, welchem alternativen

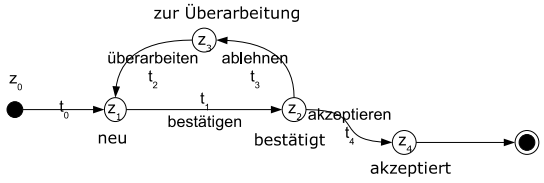

Abbildung 5.5: Zustandsdefinition zur Verarbeitung eines Angebots

Zweig gefolgt wird. Dieser Zusammenhang ist mit bisherigen Mitteln leider noch nicht modellierbar, weshalb die Aktivität an Knoten 16 auf beide Folgezustände von der Aktivität an Knoten 15 eingehen muss, obwohl nur einer eintreten wird. Auf dieses Problem wird in Abschnitt 5.5 nochmals näher eingegangen.

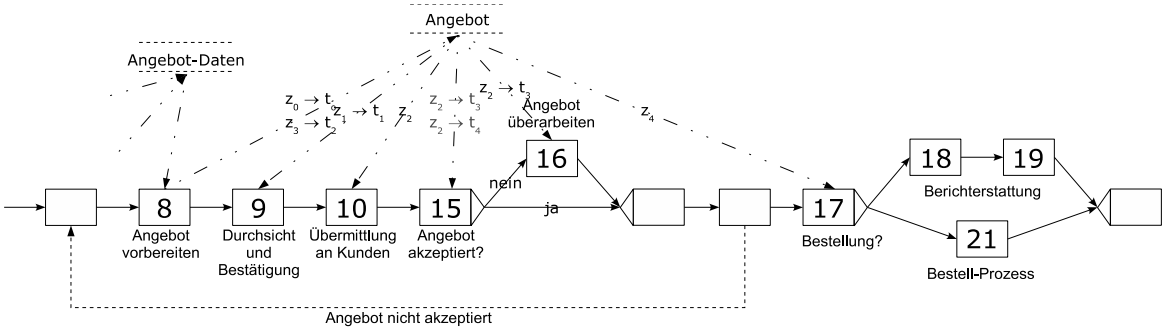

Abbildung 5.6: Verarbeitung eines Angebots mit Zustandsinformationen

Das Verhalten von Graph und Zustandsdefinition während der Ausführung eines Prozesses wird in Abbildung 5.7 dargestellt. Die schematische Darstellung zeigt, wie Zustandswechsel der Aktivität (z.B. „aktiviert“, „gestartet“) und Zustandswechsel in der Zustandsdefinition zusammenspielen. Eine konkrete Implememtierung müsste diese Informationen jedoch nicht während der Ausführung verarbeiten. Es würde genügen, die Zustandsinformationen ausschließlich während der Modellierung und bei Änderungen von Instanzen zu verwenden. Durch die Validierung der Prozessvorlage können während der Laufzeit keine falschen Zustände eintreten. Welcher Zustand zu einem bestimmten Zeitpunkt nun tatsächlich eintritt, ist daher für die Ausführung zunächst nicht relevant.

5.4 Ansätze zur Sicherstellung der Zustandsforderungen

Bevor algorithmisch festgestellt werden kann, ob die Zustandsforderungen eingehalten werden, muss zunächst definiert werden, wann eine Aktivität ausführbar ist. Danach wird auf grundlegende Anforderungen an die Validierung eingegangen. Im Anschluss daran wird ein

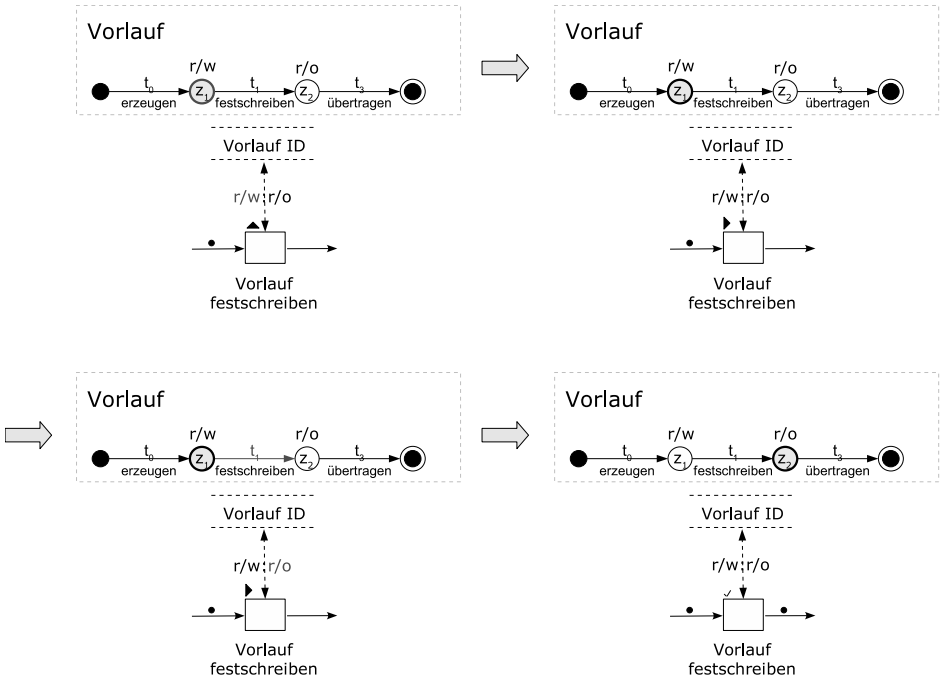

Abbildung 5.7: Ausführung einer Aktivität mit zustandsbehafteten Daten

Algorithmus für synchronisationsfreie Prozesse vorgestellt. Im Weiteren wird analysiert, welchen Einfluss Synchronisation auf die Validierung hat, um daraufhin mögliche Lösungsansätze zu diskutieren.

Für die Definition, wann ein Parameter mit einer Zustandsbeschreibung als versorgt gilt, müssen unterschiedliche Kriterien festgelegt werden. Um die Definition zu vereinfachen, werden zunächst einige Prädikate bezüglich eines Parameters P eingeführt. Das *DataProcessingTemplate*, also die Zustandsdefinition, ist in einem konkreten Prozess festgelegt. Daher wird an dieser Stelle von der Referenz $Template^{SR}$ abstrahiert. Es werden also immer ausschließlich Zustandsbeschreibungen in Betracht gezogen, welche für die gewählte Zustandsdefinition gelten.

$InputStates(P)$	Alle Zustände aus *RequiredState* der $StateList^{SR}$ des Parameters P.
$OutputStates(P)$	Alle Zustände aus *GoalStates* der $StateList^{SR}$ des Parameters P.

Damit ein Parameter P mit einer Zustandsbeschreibung als versorgt gilt, darf kein Zustand eintreten, welcher nicht als *RequiredState* in der $StateList^{SR}$ von P auftaucht. In einem Prozess können zum Ausführungszeitpunkt der Aktivität immer höchstens die Zustände eintreten, die durch die Vorgänger erzeugt werden können. Daher ist es möglich, dass nicht

alle Zustände *RequiredState* in einem Prozess erreicht werden. Eine Aktivität kann z.B. nur einen bestimmten Zustand in ihren *GoalStates* haben. Daher können durch eine darauf folgende Aktivität nur die *GoalStates* erreicht werden, welche diesen einzelnen Zustand als *RequiredState* besitzen. Auf der anderen Seite, können bei einer fehlerhaften Modellierung auch Zustände erreichbar sein, welche nicht in *RequiredStates* vorkommen. Um diese Fehler durch eine Analyse erkennen bzw. ausschließen zu können, werden auch diese Zustände in die Menge der erreichbaren Zustände aufgenommen. Für die in einem konkreten Prozess tatsächlich erreichbaren Eingangs- und Ausgangszustände an einem Parameter P werden die Prädikate $ReachableInputStates(P)$ und $ReachableOutputStates(P)$ definiert.

$ReachableInputStates(P)$	Alle Zustände, welche bei Vorgängerknoten in *ReachableOutputStates* auftauchen können.
$ReachableOutputStates(P)$	Alle Zustände, welche bei Vorgängerknoten in *ReachableOutputStates* auftauchen können.

Auf dieser Grundlage lässt sich ein einfaches Kriterium formulieren, wann eine Prozessvorlage, auch unter dem Gesichtspunkt von zustandsbehafteten Parametern, ausführbar ist.

$Versorgt(P)$	$ReachableInputStates(P)$ ist eine Teilmenge von $InputStates(P)$ oder $InputStates(P)$ enthält den speziellen Zustand *Any*. Ein zustandsbehafteter Parameter gilt genau dann als versorgt, wenn durch Vorgängerknoten ausschließlich Zustände erzeugt werden, die auch gefordert wurden.

Abbildung 5.8 zeigt, wie sich die geforderten und erreichbaren Zustände zusammensetzen. In diesem Beispiel beinhaltet einen Fehler in der Modellierung. Die erreichbaren Eingabezustände sind keine Teilmenge der geforderten Zustände. Es ist dadurch möglich, dass bei Knoten h Zustand z_2 eintritt, für den die hinterlegte Aktivität jedoch nicht ausführbar ist.

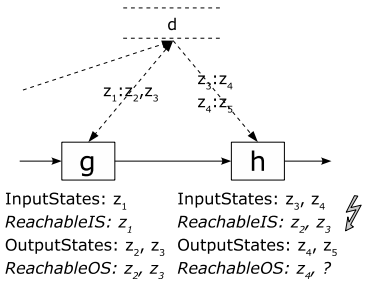

Abbildung 5.8: Geforderte und erreichbare Zustände in einer fehlerhaften Modellierung

5.4.1 Anforderungen an die Validierung der Zustandsforderungen

Die grundlegende Anforderung an die Validierung ist, dass die Parameter einer Aktivität zum Ausführungszeitpunkt mit Daten in den richtigen Zuständen versorgt sind. Um dies sicherzustellen, müssen diejenigen Zustände, die über alle möglichen Ausführungswege zustande kommen können, ermittelt werden.
Erschwert wird dies dadurch, dass die möglichen Zustände nicht alleine davon abhängen, welche Zustände in den *GoalStates* der zuvor ausgeführten Aktivität stehen. Welche dieser *GoalStates* erreicht werden, hängt auch davon ab, welche Zustände vor dieser Aktivität galten (vgl. *ReachableOutputStates*). Wird dieser Gedankengang fortgesetzt, wird schnell klar, dass die Aktivitäten von Beginn an in ihrer tatsächlichen Ausführungsreihenfolge betrachtet werden müssen. Teile dieser Reihenfolge sind jedoch nicht festgelegt. Bei Oder-Verzweigungen wird alternativ genau einem von mehreren möglichen Zweigen gefolgt. Als Konsequenz sind am Ende eines Oder-Blocks, alle Zustände möglich, welche am Ende einer jeden alternativen Verzweigung gelten.
Im Hinblick auf diesen Hintergrund scheint die Ermitllung der Zustände aufwändig zu sein. Es bietet sich an, die gefundenen Werte für die Zustandsmengen der *ReachableInputStates* und *ReachableOutputStates* an bestimmten Knoten zu speichern. Dadurch ist bei Adhoc-Änderungen schneller ersichtlich, ob eine Aktivität an einer bestimmten Stelle eingesetzt werden kann. Zudem erleichtert die Information das algorithmische Vorgehen. Um die Werte nicht an jedem Knoten halten zu müssen, bietet sich eine Auswahl bestimmter Knoten an. Sinnvoll sind Knoten, welche Zustandsforderungen besitzen, und Knoten, welche einen Block einleiten oder abschließen.
Ein weitere Anforderung an den Validierungsalgorithmus ist eine performante Ausführung. So muss z.B. bei einer Schemaevolution ein effizientes Prüfen aller zu migrierenden Instanzen möglich sein.

5.4.2 Algorithmus für synchronisationsfreie Prozesse

Bei Verwendung von Synchronisationskanten verspricht dies jedoch komplizierter zu werden, da diese die Blockstruktur von Prozessen in ADEPT aufbrechen. Daher wird im Folgenden zunächst ein Algorithmus für synchronisationsfreie Prozesse beschrieben, bevor näher auf das Problem mit Synchronisationskanten eingegangen wird. Die Vorgehensweise ist an den $WriterExists$ Algorithmus (s. 4.2) angelehnt. Anders als beim $WriterExists$ Algorithmus wird jedoch der gesamte Graph durchlaufen. Dies ist notwendig, da die Knoten in ihrer tatsächlichen Ausführungsreihenfolge abgearbeitet werden müssen, diese jedoch nicht gegeben ist. Im $WriterExists$ Algorithmus wird der Status, ob ein Knoten versorgt ist, von Knoten zu Knoten weiter gegeben. Um die Knoten-Semantik berücksichtigen zu können, wird ein Counter verwendet um die Anzahl der eingehenden und versorgten Kanten zu zählen.

Die grundsätzliche Idee wird hier übernommen. Da jedoch auch die möglichen Eingangszustände (*ReachableInputStates*) gefunden werden müssen, werden auch diese von Knoten zu Knoten weiter propagiert. Der Counter wird verwendet, um die bereits abgearbeiteten Kanten zu zählen. Ein Knoten wird nur dann weiter bearbeitet, wenn der Verlauf über alle relevanten eingehenden Kontrollfluss-Kanten betrachtet wurde. Wie bereits im vorigen Abschnitt erklärt, ist dies notwendig, um die Menge der möglichen Zustände bestimmen zu können. Damit das Verfahren nicht alle Knoten durchlaufen muss, werden Blöcke ohne Zustandsforderung übersprungen.
Um zu bestimmen, welche Zweige und welche Blöcke relevant sind, wird die topologische Sortierung des Graphen zunutze gemacht. Die topologische Sortierung von ADEPT-Graphen hat folgende Eigenschaften: Liegen Knoten in einer Sequenz, so sind sie aufsteigend in Ausführungsreihenfolge nummeriert. Liegt ein Knoten b zwischen zwei Knoten a und c einer Sequenz, so muss die Knoten-ID von b zwischen den Knoten-IDs der umgebenden Knoten a und c liegen. Ein Block wird übersprungen, wenn kein Knoten mit Zustandsforderung zwischen Split- und Join-Knoten liegt. Analog dazu werden Zweige ignoriert, wenn kein Knoten mit Zustandsforderung zwischen dem ersten und letzten Knoten, einschließlich des ersten und des letzten Knotens, im Zweig liegt.
Die von Knoten zu Knoten weiter propagierten Zustände werden an bestimmten Stellen in den Knoten-Eigenschaften RIS und ROS gespeichert, wobei ersteres für die erreichbaren Eingangszustände und letzteres für die erreichbaren Ausgangszustände steht. Knoten, an denen die Information gespeichert wird, sind Block-Knoten und Knoten mit Zustandsforderungen.

1. Initialisierung von Variablen und Setzen von initialen Zuständen. Für den Algorithmus werden zusätzliche Knoten-Attribute und weitere Variablen benötigt. Für jeden Knoten n werden die Attribute $RIS(n)$ (mögliche Eingangs-Zustände), $ROS(n)$ (mögliche Ausgangszustände) und $Counter(n)$ (abgearbeitete eingehende Kanten) jeweils mit 0 bzw. $\varnothing$ initialisiert. Der Graph wird anhand eines Stacks durchlaufen. Zu bearbeitende Knoten und ihre möglichen Eingangszustände werden auf den Stack gelegt. Während der Initialisierung werden Zustände aus den Eingangsparametern des Prozesses auf den Start-Knoten übertragen und dieser auf den Stack gelegt. Beginnend bei Schritt 2 wird dieser Stack abgearbeitet und gleichzeitig um Folgeknoten erweitert. Der Algorithmus terminiert, wenn der Stack leer ist, also der gesamte Graph durchlaufen wurde, oder wenn eine Verletzung von Zustandsforderungen gefunden wurde.

2. Nächsten Knoten (*node*) und mögliche Zustände ($States0$) vom Stack holen und betrachten.

3. Falls der Knoten Zustandsforderungen bzgl. des Datenelements besitzt:

   (a) Überprüfung von Zuständen: Um die Einhaltung der Zustandsforderungen sicher-

zustellen, sind unterschiedliche Fälle zu beachten. Da der Knoten Zustände fordert, darf die Menge der möglichen Zustände nicht leer sein. Ist die Menge nicht-leer, so muss sie entweder eine Teilmenge der geforderten Zustände sein, oder das Schlüsselwort *Any* muss in der Menge der geforderten Zustände enthalten sein.

(b) Finden der möglichen Folgezustände: Die Folgezustände ergeben sich aus allen Mengen der Zielzustände, deren geforderter Zustand in der Menge der möglichen Zustände auftaucht. Hier muss die Menge $RIS(node)$ verwendet werden, diese ist gesetzt, da sie an Knoten mit Zustandsforderungen gespeichert wird. Die möglichen Folgezustände werden in der Menge $ROS(node)$ gespeichert. Enthält die Zustandsforderung das spezielle Schlüsselwort *Any*, werden auch die nicht explizit geforderten Zustände der Menge $ROS(node)$ hinzugefügt. Die Menge der möglichen Folgezustände wird für die weitere Verarbeitung zusätzlich in *SuccStates* gespeichert.

4. Falls der Knoten keine Zustandsforderungen besitzt:

   (a) Gegebenenfalls Übertragung von $RIS(node)$ auf $ROS(node)$: Da die möglichen Zustände nicht nur an Knoten mit Zustandsforderungen gespeichert werden, kann auch in diesem Fall die Menge $RIS(node)$ nicht-leer sein. Da der Knoten den Zustand nicht ändert, erhält die Menge $ROS(node)$ die gleichen Zustände.

   (b) Der Zustand wird nicht geändert. Die Menge der möglichen Zustände zur Weiterverarbeitung bleibt also gleich. *SuccStates* wird auf den Wert von *States*0 gesetzt.

5. Finden der Nachfolger: Es müssen relavante Nachfolger bezüglich Kontrollfluss-Kanten gefunden werden.

6. Aktueller Knoten ist der Anfangsknoten eines Blockes (z.B. OR-Split):

   (a) Falls kein Knoten im Block eine Zustandsforderung besitzt, wird der zugehörige Join-Knoten als relevanter Nachfolger verwendet. Dadurch wird der Block übersprungen.

   (b) Falls Knoten im Block Zustandsforderungen besitzen, werden alle Zweige betrachtet. Nachfolger des Block-Knotens, welche auf einem Zweig mit Zustandsforderungen liegen, werden der Menge der relevanten Nachfolgern hinzugefügt.

7. Aktueller Knoten ist kein Block-Anfang-Knoten: Der (einzige) direkte Nachfolger wird verwendet.

8. Der nächste (relevante) Nachfolger *sn* wird betrachtet.

9. Anpassung der Folgezustände und Setzen von $RIS(sn)$: Je nach Knotensemantik muss die Menge *SuccStates* angepasst werden. Im gleichen Zug wird die Menge $RIS(sn)$ gesetzt, wo dies erwünscht ist. Folgende Sonderfälle für *sn* sind zu beachten:

(a) AND-Join: Falls der Folgeknoten ein AND-Join ist und im Block Zustandsforderungen existieren, wird $RIS(sn)$ auf $SuccStates$ gesetzt. Da der Prozess frei von Sync-Kanten ist, wurde höchstens einem Zweig gefolgt.

(b) OR-Join (und Zustandsforderung im Block): Am Ende einer alternativen Verzweigung sind alle Zustände möglich, die am Ende jeder einzelnen Verzweigung gelten. Daher werden diese der Menge $RIS(sn)$ und $SuccStates$ hinzugefügt. Der Knoten wird jedoch erst dann auf den Stack gelegt, wenn alle eingehenden Kanten relevanter Zweige abgearbeitet wurden.

(c) LOOP-End (und Zustandsforderung im Block): Nach einer Schleife werden die möglichen Zustände in $RIS(sn)$ gespeichert. Zusätzlich wird hier noch kontrolliert, ob die Zustände am Ende der Schleife eine Teilmenge der Zustände zu Anfang der Schleife sind.

(d) Knoten ändert den Zustand: In diesem Fall muss nur $RIS(sn)$ auf den Wert von $SuccStates$ gesetzt werden.

10. Knoten auf den Stack: Der Nachfolger *sn* wird genau dann auf den Stack gelegt, wenn *node* der letzte noch nicht betrachtete relevante Vorgänger ist. Ansonsten wird der Zähler $Counter(sn)$ um eins erhöht. Da es ausschließlich bei Oder-Verzweigungen mehrere relevante eingehende Zweige gibt, muss der Zähler nur im Falle eines OR-Joins berücksichtigt und erhöht werden.

11. Den nächsten Nachfolger betrachten, zurück zu Schritt 8.

12. Den nächsten Knoten im Stack betrachten, zurück zu Schritt 2.

Bei der textuellen Erläuterung des Algorithmus wurden bewusst einige Details ausgelassen, um nur den prinzipiellen Ablauf zu erläutern. Im Anhang A wird ein ein vollständiger Algorithmus als kommentierter Pseudo-Code vorgestellt. Der Algorithmus bietet ein effizientes Verfahren zur Überprüfung von Zuständen in einem synchronisationsfreien Prozess. Der Graph muss nur einmal durchlaufen werden, wobei sämtliche irrelevante Blöcke und Teilzweige übersprungen werden. Währenddessen werden die geltenden Zustände zu bestimmten Ausführungszeitpunkten zwischengespeichert, was eine effiziente Revalidierung zur Laufzeit erlaubt.

5.4.3 Synchronisation

In diesem Abschnitt wird analysiert, welche Anforderungen an den Algorithmus für den Umgang mit synchronisierten Graphen bestehen. Synchronisationskanten sind ein bewusster Bruch der Blockstruktur, der eine Festlegung der Ausführungsreihenfolge ansonsten paralleler

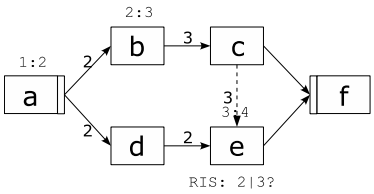

Abbildung 5.9: Synchronisation von KF und DF

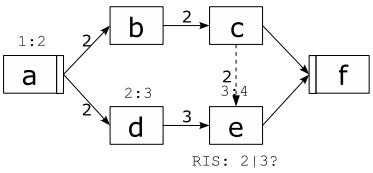

Abbildung 5.10: Vom DF unabhängige Synchronisation

Aktivitäten erlaubt. Damit kann die Ausführung von zwei parallelen Zweigen eines Blocks synchronisiert werden. Teil eines Zweigs sind auch Aktivitäten aus Blöcken, welche auf dem Zweig liegen. Dadurch ist Synchronisation aus Blöcken heraus, sowie in Blöcke hinein, möglich, sofern Start- und Zielknoten zu unterschiedlichen Zweigen eines gemeinsamen Und-Blocks gehören. Hintergrund von Sync-Kanten ist die Kontrollflussmodellierung, um die Synchronisation von paralleler Ausführung an bestimmten Stellen im Prozess zu ermöglichen. Die Synchronisation von Zugriffen auf Datenelementen findet daher nicht explizit statt, sondern ergibt sich implizit aus dem Kontrollfluss.
Diese implizite Synchronisation bereitet Schwierigkeiten bei einer Erweiterung des im vorigen Abschnitt beschriebenen Algorithmus. Da Zustandsforderungen in parallelen Zweigen durch Sync-Kanten ermöglicht werden, muss der Algorithmus auch diesen folgen und Zustände auch darüber weiter propagieren. Die Entscheidung, ob und welche Zustände über eine Sync-Kante propagiert werden müssen, ist jedoch nicht trivial. Gründe hierfür werden anhand von ein paar einfachen Beispielen aufgezeigt. Um die Graphiken übersichtlich zu halten, wird auf die Darstellung von Datenelementen verzichtet. Die Beispiel-Graphen beeinhalten ein einzelnes Datenelement. Sofern ein Knoten eine Zustandsforderung und Zielzustände besitzt, werden diese direkt am Knoten annotiert: $\{Eingabezustaende\} : \{Ausgabezustaende\}$. Übertragene Zustände werden ggf. an den Kontrollfluss-Kanten angezeigt.

Übertragung von Zuständen über eine Sync-Kante Abbildungen 5.9 und 5.10 zeigen das grundsätzliche Problem der Entscheidung, ob Zustände über Sync-Kanten propagiert werden müssen oder nicht. Alleine durch Betrachtung der synchronisierten Knoten lässt sich nicht bestimmen, welcher Zustand übernommen werden muss. Ein naiver Ansatz wäre z.B. zu untersuchen oder zu speichern, auf welchem Zweig sich die letzte Zustandsforderung befand. Wie jedoch Abbildung 5.11 zeigt, genügt dies nicht: Die Synchronisation des Datenflusses kann auch transitiv über mehrere Sync-Kanten und parallele Zweige geschehen.

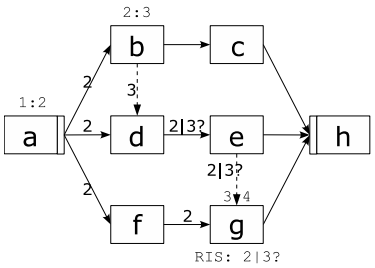

Abbildung 5.11: Transitive Synchronisation des DF

Auswahl der bei Sync-Kanten zu übertragenden Zustände Werden Zustände über Sync-Kanten übertragen, muss untersucht werden, welche Zustände bei der Synchronisation eine Rolle spielen. Dies ist mit Komplikationen verbunden, wenn die Synchronistion aus Oder-Blöcken heraus geschieht. Sync-Kanten können in diesem Fall abgewählt werden: Wird der Zweig mit der Sync-Kante abgewählt, wird die Sync-Kante als nicht mehr ausführbar markiert. Zur Ausführung des Zielknotens spielt die Sync-Kante von diesem Moment an keine Rolle mehr. Dadurch sind implizit mehrere Knoten mit dem Zielknoten synchronisiert: Der Ausgangsknoten der Sync-Kante und jeder Knoten, der über die Ausführung des Ausgangsknotens entscheidet. Der Graph in Abbildung 5.12 zeigt eine solche Situation. Es nehmen sowohl Knoten c als auch Knoten e Einfluss auf die Synchronisation. Im synchronisierten Zweig sind im weiteren Verlauf beide Mengen der möglichen Ausgangszustände in Betracht zu ziehen.

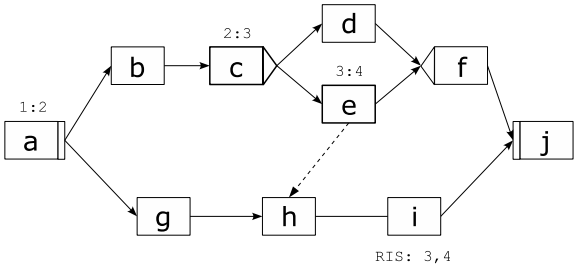

Abbildung 5.12: Synchronisation durch abwählbare Sync-Kante

Eine Untersuchung des direkt umschließenden Oder-Blockes reicht jedoch nicht aus. Vielmehr müssen alle verschachtelten Blöcke, bis zum synchronisierten Und-Block, untersucht werden. Durch jede weitere Verschachtelung nimmt ein weiterer Knoten Einfluss auf die Synchronisation. In Abbildung 5.13 wird diese Situation verdeutlicht. An der Synchronisation sind in diesem Fall Knoten b, c und e beteiligt. Dementsprechend müssen auch die möglichen Ausgangszustände berücksichtigt werden.
Ließen sich in den bisher gezeigten Beispielen die beteiligten Knoten auf die Split-Knoten zurückführen, so gibt es dennoch einen weiteren zu berücksichtigenden Fall: Sind alle Zweige

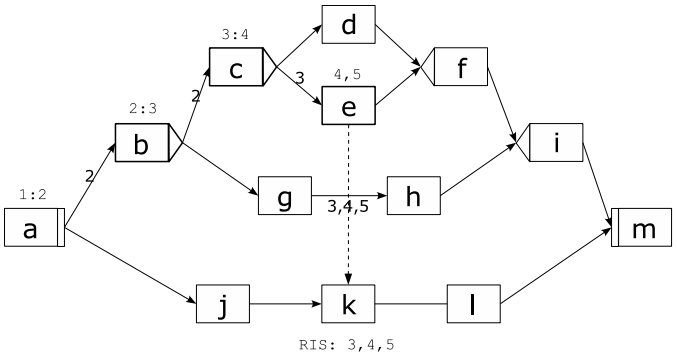

Abbildung 5.13: Synchronisation aus einem verschachtelten Oder-Block

des Oder-Blocks synchronisiert, spielt der Zustand beim Split-Knoten keine Rolle mehr. In diesem Fall müssen die Zustände aus allen Knoten, von denen die Sync-Kanten ausgehen, übernommen werden.

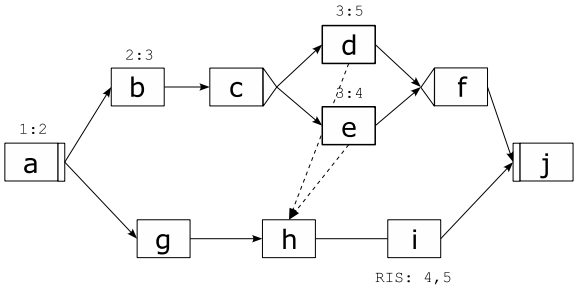

Abbildung 5.14: Synchronisation aller alternativen Zweige

Falls alle alternativen Zweige synchronisiert werden, muss dies nicht an einem einzelnen Knoten geschehen. Abbildung 5.15 zeigt, wie alle Zweige eines Oder-Blocks indirekt synchronisiert werden. Die Beispiele lassen sich beliebig durch Kombinationen der Fälle entsprechend verkomplizieren. Die vorgestellen Graphen veranschaulichen jedoch alle möglichen grundlegenden Situationen.

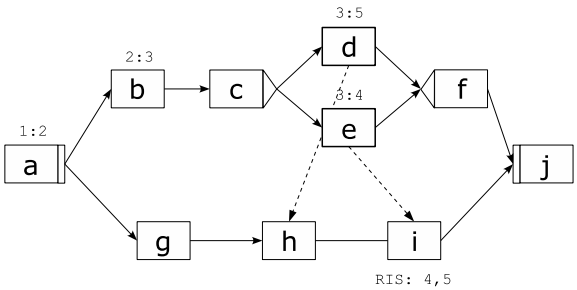

Abbildung 5.15: Indirekte Synchronisation aller alternativen Zweige

Behandlung von mehreren Sync-Kanten Gehen an einem Knoten mehrere Synchronisationskanten ein, müssen diese weiter untersucht werden. Grundsätzlich ist festzustellen, ob die Kanten für den Datenfluss relevant sind oder nicht. Für je zwei Kanten können folgende Fälle unterschieden werden:

1. Die Reihenfolge, in welcher die Sync-Kanten bewertet werden, ist eindeutig festgelegt. Dies kann durch transitive Synchronisation geschehen, oder die Ausgangsknoten der Kanten liegen in einer Sequenz. Die Kante, die zuerst bewertet wird, kann daher ignoriert werden.

2. Die Kanten kommen aus unterschiedlichen Zweigen und ihre Reihenfolge ist nicht eindeutig festgelegt. In diesem Fall kann es nur sein, dass über eine Kante Zustände übertragen werden. Ansonsten wären die Zustandsforderungen nicht synchronisiert. Die Entscheidung, von welcher Kante Zustände entgegengenommen werden, ist analog zu der Frage, ob über eine Kontrollfluss- oder über eine Sync-Kante Zustände entgegengenommen werden.

3. Die Kanten kommen aus unterschiedlichen Zweigen von einem Oder-Block oder sind eine transitive Synchronisation aus diesen Zweigen. Die Zustände von allen Kanten müssen berücksichtigt werden. Falls nicht alle Zweige des Oder-Blocks synchronisiert wurden, müssen zudem auch die Zustände der entsprechenden OR-Splits berücksichtigt werden (vgl. Abbildungen 5.12, 5.13 und 5.14).

Zusammenfassung Wie einfache Beispiele zeigen, ist die bisher verfolgte Heransgehensweise zur Validierung nicht auf einfache Weise auf synchronisierte Graphen anwendbar. Ein weiteres Problem stellt das Speichern der Zustände an Block-Knoten dar. Durch den Bruch der Blockstruktur ist dies nicht mehr ohne weiteres möglich. So müssen entweder falsche Werte an bestimmten Knoten in Kauf genommen werden oder die Tatsache, dass diese nicht an jedem Block mit Zustandsforderungen gespeichert werden. In Abbildung 5.12 ist es z.B. nicht möglich einen korrekten Wert an Knoten f zu speichern. Durch Synchronisation und parallele Ausführung sind die Zustände zu dessen Ausführungszeitpunkt nicht mehr eindeutig.
Auch wenn die Modellierung einiger der vorgestellten Graphen wenig sinnvoll erscheint, so sind sie dennoch ADEPT konform und müssen bei der Analyse berücksichtigt werden. Zu bedenken ist auch, dass Graphen durch Adhoc-Änderungen, Schemaevolution und semantisch höherwertige Änderungsoperationen modifiziert werden können. Dadurch können in realen Fällen auch weniger ansehnliche Graphen entstehen.

5.4.4 Lösungsansätze für Prozesse mit Synchronisation

Im Folgenden werden drei Lösungsansätze umrissen, die eine Ausführbarkeitsanalyse unter Berücksichtigung von Synchronisationskanten ermöglichen. Zunächst wird ein Ansatz, der auf Knoten-Beziehungen basiert, vorgestellt. Als zweites wird ein Ansatz, der mit Erreichbarkeitsgraphen arbeitet, angedacht. Ein dritter Lösungsansatz basiert auf Techniken zur Graph-Reduktion.

Ein Ansatz auf Grundlage von Knoten-Relationen Unter der Voraussetzung, dass bestimmte Beziehungen unter den Knoten, die Zustandsforderungen besitzen, bekannt sind, lassen sich die Zustände einfacher überprüfen. Dazu müssen Fragen, wie z.B. „Kommt ein Knoten direkt vor einem anderen?“ oder „Kommt ein Knoten optional direkt vor einem anderen?“, beantwortet werden können. Die erforderlichen Knoten-Beziehungen müssen auch Sync-Kanten berücksichtigen. Tabelle 5.1 gibt eine Übersicht über benötigte Beziehungen.

Tabelle 5.1: Knotenbeziehungen

$a < b$	a wird direkt vor b ausgeführt.
$a \lhd b$	a wird direkt vor b ausgeführt, jedoch ist die Ausführung von b optional.
$a \mid\!< b$	a wird direkt vor b ausgeführt, jedoch ist die Ausführung von a optional.
$a \ ?\ b$	Zwischen a und b existiert eine der Beziehungen bzw. a kommt vor b.

Sind die Beziehungen unter den Knoten mit Zustandsforderungen bekannt, kann wie folgt vorgegangen werden:

1. Betrachte alle Tupel mit einem Knoten a, zu dem es kein Knoten b gibt, mit $b\ ?\ a$.
2. Falls $a < b$ oder $a \lhd b$ und nicht $a \mid\!< b$: b wird (optional) nach a ausgeführt. Wird b ausgeführt, so ist jedoch a per Definition der Vorgänger. RIS von b erhält also ROS von a. $RIS(b) := ROS(a)$.
3. Falls $a \mid\!< b$: a wird optional vor b ausgeführt. Die Zustände, die über diese Beziehung übertragen werden, müsen zu RIS von b hinzugefügt werden. $RIS(b) := RIS(b) \cup ROS(a)$.
4. Entferne das Tupel aus der Menge und fahre mit dem nächsten Tupel fort. Terminiere, falls keine weiteren Tupel vorhanden sind.

Werden Knoten-Beziehungen bereitgestellt, so ist eine algorithmische Überprüfung der Zustände mit minimalem Aufwand möglich. Die algorithmische Hauptarbeit liegt jedoch im Fin-

den der Knoten-Beziehungen. Die Frage nach der tatsächlichen Ausführungsreihenfolge einer Knotenmenge wird jedoch auch für andere Analysen von Bedeutung sein. Beispiele hierfür sind semantische Abhängigkeiten zwischen Aktivitäten, Ressourcenmanagement und weitere komponenteninterne Abhängigkeiten. Auch um Sichten auf eine Teilmenge von Knoten eines Graph zu erstellen, kann diese Information interessant sein. Unter Umständen ist es daher sinnvoll, Datenstrukturen zu erstellen, die eine effiziente Feststellung der Knoten-Beziehungen ermöglichen.

Ein Ansatz auf Grundlage von Erreichbarkeitsgraphen Eine weitere Vorgehensweise zur Berücksichtigung von Synchronisationskanten ist die Erstellung eines Erreichbarkeitsgraphen. In diesem Graphen werden alle möglichen Kombinationen von Aktivitätenzuständen (*STARTED*, *COMPLETED* etc., siehe Abschnitt 2.2.3) dargestellt, die während der Ausführung des Prozesses erreicht werden können. Damit lassen sich auch alle möglichen Ausführungsreihenfolgen herausfinden. Die Zustandswechsel der Aktivitäten spielen dabei jedoch keine Rolle. Dadurch reicht ein bedeutend kleinerer Erreichbarkeitsgraph aus, indem z.B. nur der Aktivitätenzustand *STARTED* berücksichtigt wird. Durch die kombinatorische Vielfalt, die durch die verschiedenen Ausführungsreihenfolgen bei parallelen Zweigen zu erwarten ist, erscheint eine anschließende Analyse jedoch trotzdem als zu aufwändig. Der Aufwand ließe sich jedoch erheblich verkleinern, wenn Knoten und Blöcke aus dem Graphen entfernt würden, welche zur Analyse der Zustandsforderungen keine Rolle spielen. Dieser Vorgang, auch Graph-Reduktion genannt, könnte jedoch auch genutzt werden, um einen neuen und einfach analysierbaren Graphen zu erzeugen.

Ein Ansatz unter Verwendung von Graph-Reduktion Aktivitäten mit Zustandsforderungen dürfen nicht parallel ausgeführt werden. Dadurch kann die Ausführungsreihenfolge dieser Aktivitäten in einem Graphen ohne Parallelität dargestellt werden. Graph-Reduktion ist eine Möglichkeit, um einen solchen Graphen zu erzeugen. Insbesondere unter Verwendung der topologischen Sortierung können so ganze Blöcke und einzelne Knoten nach und nach aus dem Original-Graphen entfernt werden. Dadurch würden z.B. Sync-Kanten, die für eine indirekte Synchronisation verantwortlich sind (siehe Abbildung 5.15), auf einen Knoten zusammenfallen. Und-Blöcke könnten dadurch so lange vereinfacht werden, bis die tatsächliche Ausführungsreihenfolge der betrachteten Knoten eindeutig ist. Daraufhin könnten die Block-Knoten entfernt und die Sync-Kanten in normale Kontrollfluss-Kanten umgewandelt werden, um den gewünschten Graphen zu erhalten. Grundlage können bereits vorhandene Algorithmen bieten. In [11, S. 111] wird z.B. eine einfache Graphreduktion vorgeschlagen, um die Erreichbarkeitsanalyse für ADEPT-Graphen zu optimieren.

5.5 Diskussion und Ausblick

Die in diesem Kapitel vorgestellte Lösung zur Integration von auf zustandsbehafteten Daten basierende Abhängigkeiten bietet dem Prozess- und Aktivitäten-Designer ein mächtiges Werkzeug. Damit lassen sich zahlreiche Zusammenhänge zwischen Aktivitäten modellieren. Durch eine zentrale Verwaltung der Zustandsgraphen ist auch eine Wiederverwendung bereits modellierter Zusammenhänge möglich. Die wichtigste Neuerung ist jedoch die Festlegung von Zustandsabfolgen. Die Realisierung als Digraph entspricht einem einfachen Zustandsautomaten, welcher als Eingabe nur den aktuellen Zustand und den direkten Folgezustand kennt. Durch die Möglichkeit der Aktivitäten, auf unterschiedliche Zustände zu reagieren und unterschiedliche Folgezustände angeben zu können, sind auch sehr komplexe Zusammenhänge modellierbar. Da jedoch i.A. kleine Graphen für die Datenzustände zu erwarten sind, verspricht die Modellierung dennoch übersichtlich zu bleiben. Auch semantische Abhängigkeiten, die sich in Form einer Zustandsdefinition ausgedrückt werden kann, können auf diesem Wege ins WfMS integriert werden.
Vorteile einer Modellierung mit Zuständen liegen auch in der Ausführung von Vorwärtssprüngen (z.B. wegen einer Notoperation) vor. So können nicht nur unversorgte Parameter erkannt werden, sondern zudem auch die Ausführung von weiteren Aktivitäten vorgeschlagen werden, um den benötigten Datenzustand bei der Zielaktivität zu erreichen. Dadurch kann der Ausführungserfolg von Aktivitäten auch bei Vorwärtssprüngen sichergestellt werden.

Ein durch den vorgestellten Ansatz nicht lösbares Problem verbleibt an Oder-Verzweigungen. Da Aktivitäten unterschiedliche Zustände fordern können, werden am Ende einer Verzweigung mehrere mögliche Zustände erlaubt. Wie bereits gezeigt, ist dies notwendig, da Verzweigungen typischerweise auch zur unterschiedlichen Verarbeitung von Daten verwendet werden. Die Entscheidung, welcher Verarbeitungsweg eingeschlagen wird, kann jedoch auch vom Zustand der Daten abhängen und sich dadurch auch in diesem widerspiegeln. Daraus ergeben sich zwei Anforderungen: Der Zustand der Daten muss im Entscheidungsprädikat bei Oder-Verzweigungen verwendet werden können. Darüber hinaus muss die Menge möglicher Zustände, je nach Verzweigung, eingeschränkt werden können. Eine naive Lösung dafür wäre eine Zuordnung zwischen Ausgabeparametern und den vom Knoten ausgehenden Verzweigungen. Ein allgemeinerer Ansatz wäre jedoch sinnvoller. So könnte die Entscheidung, welchem Zweig gefolgt wird, auch vom Wert des Statusparameters einer Aktivität abhängig gemacht werden. Der Statusparameter ist ein spezieller Ausgabeparameter, welcher weitere Informationen über die Beendigung der Aktivität gibt. Er ist Vergleichbar mit *exit codes* von Prozessen eines Betriebssystems [11, Abschnitt 3.2.2.3]. Würde eine Zuordnung zwischen Statusparameter und Verzweigung existieren, könnten auf deren Grundlage auch Zustände aus den *GoalStates* einzelnen Zweigen fest zugeordnet werden. Da bisher jedoch noch keine Möglichkeit existiert, das Verzweigungsverhalten einer Aktivität zu formulieren, müssen entsprechende Voraussetzungen auf einer höheren Ebene in Aktivitätenbeschreibungen und Prozessmodell geschaffen

werden.

Die Ausführung der Prozesse bleibt weiterhin effizient. Da alle Zustandsinformationen, die zur Validierung benötigt werden, bereits in der Prozessvorlage enthalten sind, ist kein Zugriff auf das Aktivitäten-Repository notwendig. Tatsächlich könnten diese Zustandsinformationen sogar ausschließlich zur Modellierung verwendet werden. Zur Ausführungszeit wird dann nur die Information über momentanen Zustand verarbeitet. Da der Prozess vor der Ausführung validiert wurde, ist ein Weiterschalten und Verwalten aller Zustände nicht notwendig.
Dies gilt natürlich nur unter der Voraussetzung, dass alle beteiligten Aktivitäten und Komponenten fehlerfrei arbeiten und korrekt modelliert wurden. Damit das WfMS gegenüber entsprechender Fehler robuster ist, sollte die Einhaltung der Zustände revalidierbar sein. Dazu muss auch bei laufenden Instanzen die Zustandsinformation verarbeitet und gespeichert werden. Wenn der tatsächliche Zustand der Daten überprüft wird, sollte eine Berechnung des Soll-Zustandes aus Gründen der Performanz nicht notwendig sein. Ähnlich wie für versteckte Datenflüsse vorgeschlagen (s. Abschnitt 3.6), ist in bestimmten Fällen eine Überprüfung der Daten auf den korrekten Zustand wünschenswert.

Im Folgenden wird ein Ausblick auf weiterführende Aspekte gegeben, für deren Lösung der vorgestellte Ansatz als Grundlage dienen kann.

5.5.1 Parallele Zustandswechsel

Zustandswechsel können auch bei paralleler Ausführung mit Einschränkungen erlaubt werden. In der in diesem Kapitel vorgestellten Lösung sind nur Änderungen in einem einzelnen parallelen Zweig erlaubt. Zudem müssen alle anderen Zweige, welche Zustandsforderungen enthalten mit diesem Zweig synchronisiert sein. Diese Restriktion lässt sich jedoch auflockern: Unter dem Vorbehalt, dass die Ausführungsreihenfolge der beteiligten Aktivitäten durch Synchronisation festgelegt ist, können in allen parallelen Zweigen Zustandswechsel erlaubt werden. Zusätzlich können auch Zustandsforderungen unsynchronisiert zugelassen werden, sofern sie alle möglichen, auch durch parallele Zweige verursachten, Zustände abdecken. In diesem Fall muss jedoch über weitere Mechanismen nachgedacht werden, um das Weiterschalten des Zustands zu verhindern, wenn zu dem Zeitpunkt eine weitere Aktivität ausgeführt wird, die den Zustand gefordert hat.

Auch bezüglich Zustandswechsel in einem parallel laufenden Sub-Prozess können die bisher gestellten Anforderungen gelockert werden. Zum Beispiel unter Betrachtung folgender Situation: Eine Aktivität fordert einen Zustand, der während der Ausführung eines parallel laufenden Sub-Prozesses verursacht wird. Der geforderte Zustand sei dabei nicht der Endzustand des Sub-Prozesses, sondern tritt während dessen Ausführung auf. In diesem Fall muss die Aktivität darauf warten, dass der geforderte Zustand durch den Sub-Prozess gesetzt wird. Die Aktivität kann daher genau dann ausgeführt werden, wenn eine Aktivität im Sub-Prozess

ausgeführt wurde. Dies ist gleichbedeutend mit Synchronisation über die Prozesshierarchie hinweg.
Zu beachten sind dabei folgende Anforderungen. Der Sub-Prozess muss unbedingt einen der geforderten Zustände im Laufe seiner Ausführung erreichen. Dies bedeutet für den Sub-Prozess, dass alle möglichen Wege zwischen geforderten Zuständen und möglichen Zielzuständen mindestens einen der geforderten Zustände der parallelen Aktivität enthalten. Der Sub-Prozess muss eine Referenz auf das Datenelement erhalten, so dass Zustandsänderungen auch für die darüber liegende Instanz ersichtlich sind. Das WfMS muss bei Zustandswechseln überprüfen, ob ein geforderter Zustand der Aktivität erreicht wurde, und deren Ausführung gegebenenfalls freigeben.

5.5.2 Verarbeitungswege

Der Zustandsgraph kann als ein Verarbeitungsweg für Daten eines bestimmten Typ angesehen werden. So soll ein Datum einer bestimmten Verarbeitung unterlaufen, indem Aktivitäten das Datum lesen, ändern und schreiben. Dabei ändern die Daten bei jedem Schritt ihren Zustand. Legt man den Verarbeitungsweg über einen Zustandsgraphen fest, so erhält man eine abstrakte Beschreibung davon, was mit den Daten passieren soll.
Zusammen mit der Information, welche Aktivitäten den Zustand der Daten ändern, lässt sich ein durch das WfMS und das Aktivitäten-Repository gestützter Modellierungsprozess für den Kontrollfluss vorstellen. Dies ist ein Schritt in Richtung Datengetriebener WfMS: Die Modellierung orientiert sich an der Verarbeitung der Daten; der Kontrollfluss ergibt sich daraus. So könnte die Modellierung eines Prozesses bei den Datenelementen beginnen. Für jedes Datenelement kann danach ein Verarbeitungsweg ausgewählt werden. Dabei könnte bereits eine erste Eingrenzung geschehen, indem der Benutzer relevante Wege im Zustandsgraphen markiert. So könnte der Modellierer festlegen, welche Daten verarbeitet werden, auf welchem Weg und mit welchem Ziel sie verarbeitet werden. Danach könnten Aktivitäten zur Erreichung von jedem Zustand angeboten werden. Denkbar wäre auch mit einer Aktivität für den Zielzustand zu beginnen und durch Auflösen von Abhängigkeiten den Prozess iterativ zu vervollständigen. Je mehr Aktivitäten ausgewählt wurden, desto mehr Abhängigkeiten ergeben sich, wodurch die Auswahl zunehmend eingeschränkt wird. Dadurch können die Abhängigkeiten mit Hilfe des Systems mehr und mehr semi-automatisch gelöst werden können. So können durch jede Abhängigkeit, welche sich durch Daten, geforderte Zustände und dem festgelegten Verarbeitungsweg ergibt, Aktivitäten zur Lösung vorgeschlagen werden.
Die Modellierung verspricht dadurch schneller und übersichtlicher zu werden, da der Verarbeitungsweg eines einzelnen Datums i.A. viel kleiner sein wird als ein vollständiger Prozess.

5.5.3 Allgemeine Zustandsobjekte

Der Bezugspunkt für Zustände in den bisherigen Überlegungen waren Daten. In WfMS sind jedoch auch von Daten unabhängige Objekte denkbar, die ebenso einen Zustand besitzen können. Zum einen sind hier allerlei Ressourcen, wie z.B. Drucker, vorstellbar. Auf der anderen Seite aber auch komponenteninterne Zustände, welche nicht an konkrete Daten gebunden sind. Ein Drucker könnte zum Beispiel die Zustände „Aus“, „Bereit“ und „Druck“ besitzen. Eine Aktivität könnte den Druckerzustand „Bereit“ einfordern.
Tatsächlich ließe sich das Konzept auch auf allgemeine Zustandsobjekte übertragen. Es müssten hierfür nur entsprechende neue Elemente in die Prozessmodellierung eingeführt werden. Dazu gehören neben den Zustandsobjekten selbst z.B. auch Zustandskanten und eigene Zustandsparameter. Alternativ wäre jedoch auch denkbar, anstelle neuer Zustandsobjekte virtuelle Datenelemente zu verwenden. Aufgrund der gleichen Verhaltensweise von virtuellen und konkreten Datenelementen und -flüssen kann das Zustandsmodell für beide Zwecke eingesetzt.
Eine wichtige Überlegung zu diesem Punkt ist jedoch die Benutzerfreundlichkeit. So ist es für den Anwender nicht gerade einleuchtend, warum z.B. ein Drucker durch ein zustandsbehaftetes virtuelles Datenelement repräsentiert werden soll. Eine explizite Modellierung als allgemeines Objekt, welches einen Zustand besitzt, scheint daher sinnvoller. Auch in der graphischen Repräsentation eines Prozesses muss der Unterschied zwischen Datenelementen und anderen Objekte, wie einem Drucker, einfach ersichtlich sein. Auf der anderen Seite wiederum muss anhand der Darstellung der beiden Elemente ersichtlich sein, ob sie zustandsbehaftet sind oder nicht.

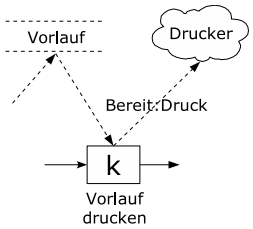

Abbildung 5.16: Ein Drucker als allgemeines Zustandsobjekt

Abbildung 5.16 zeigt einen Vorschlag zur Darstellung allgemeiner Zustandsobjekte. Im Beispiel wird ein Drucker durch ein wolkenförmiges Objekt dargestellt. Die Aktivität verlangt vom Drucker den Zustand „Bereit“. An dieser Stelle wird auch sichtbar, dass weitere Konzepte benötigt werden: Der Drucker besitzt einen globalen Zustand, welcher nicht durch den aktuellen Prozess verursacht wird. Im folgenden Abschnitt wird näher darauf eingegangen.

5.5.4 Globale Zustände

Das Beispiel mit dem Drucker als eine zustandsbehaftete Ressource bringt einen weiteren Aspekt ins Spiel: Globalität bezüglich unterschiedlicher Prozesse und Instanzen. Da ein Drucker ein konkretes, einmalig existierendes Objekt ist, muss auch das ihn repräsentierende Zustandsobjekt dem entsprechen. Dies zieht eine Reihe von Konsequenzen mit sich.
Da das Zustandsobjekt global ist, muss es auch vom WfMS zentral verwaltet werden. Eine Prozessinstanz wiederum kann nur eine Referenz auf das (einmalige) Objekt erhalten. Da nun mehrere Instanzen parallel auf das Objekt zugreifen können, kann aus Sicht einer Instanz das Objekt einen spontanen Zustandswechsel vollziehen. Die zu Beginn des Kapitels gestellte Forderung nach der Information über mögliche Zustandsübergänge zur Modellierungszeit kann dadurch nicht mehr eingehalten werden.
Da nun nicht mehr davon ausgegangen werden kann, dass sich die Daten, bzw. bei allgemeinen Zuständen das Objekt, zum Ausführungszeitpunkt in einem geforderten Zustand befinden, müssen neue Mechanismen eingeführt werden. Dazu gehört z.B. auch, dass eine Instanz auf das Eintreten eines bestimmten Zustands warten muss. Hier zeigt sich eine Stärke des Modells mit einem Zustandsgraphen: Es ist zumindest feststellbar, ob der geforderte und erwartete Zustand noch erreichbar ist oder nicht. So ist es im Ernstfall möglich, eine Warnung und Aufforderung zum manuellen Eingriff an den Prozessverantwortlichen zu senden.
Um *Race Conditions* weitgehend verhindern zu können, müssen hier Synchronisationskonzepte eingesetzt werden. So ist es durchaus wahrscheinlich, dass mehrere Instanzen gleichzeitig auf den Zustand „Bereit“ eines Druckers warten. Schlussendlich darf dieser Zustand jedoch nur von einer einzigen Instanz geändert werden.
Bei Zustandsforderungen muss darüber hinaus weiter unterschieden werden. Eine Aktivität kann z.B. Daten in einem bestimmten Zustand benötigen. Sobald diese jedoch als Parameter erhalten wurden, kann sich der Zustand der Daten ändern. Bei allgemeinen Zustandsobjekten kann ein geforderter Zustand auch bedeuten, dass der Zustand während der gesamten Ausführung erhalten bleiben muss und daher erst nach Beendigung der Aktivität wieder geändert werden darf.
Insbesondere wenn es sich um globale Zustände handelt, muss zudem noch beachtet werden, dass sich der Zustand auch „von außen“ ändern kann. Dies kann z.B. durch einen manuellen Eingriff geschehen (einen Drucker ausschalten), jedoch auch vom konkreten Objekt selbst ausgehen (Drucker geht in Zustand „Papierstau“).

5.5.5 Mehrere Zustände

Im vorgestellten Zustandsmodell kann für ein Datenelement eine Zustandsdefinition hinterlegt werden. Dies bedeutet jedoch, dass alle Aktivitäten und Komponenten mit den gleichen Zuständen arbeiten müssen. Realistisch sind jedoch auch Situationen, in denen eine Kom-

ponente bezüglich eines Datums unterschiedliche Zustände besitzt. Ist dies bei mehreren beteiligten Komponenten innerhalb eines Prozesses der Fall, müssten daher unterschiedliche Zustände parallel gelten. So könnten zum Beispiel in einem Bestellprozess mehrere Aktivitäten der Komponenten Lagerverwaltung und Auftragsverwaltung beteiligt sein. Die Aktivität „Artikel auslagern“ der Lagerverwaltung könnte bezüglich eines Datums „Auftrag“ den Zustand „Verfügbarkeit überprüft“, welcher durch eine Aktivität der gleichen Komponente verursacht wird, verlangen. Gleichzeitig kann die Aktivität „Lieferschein erstellen“ der Auftragsverwaltung den Zustand „Auftrag angenommen“ voraussetzen. Zustände dieser Komponenten können gleichzeitig gelten. Abbildung 5.17 zeigt die beschriebene Situation.

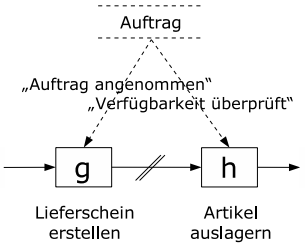

Abbildung 5.17: Mehrere Zustände?

Zur Lösung dieses Problems könnten die Zustandsgraphen zu Petri-Netzen aufgebläht werden. Die Komplexität würde sich dadurch allerdings erheblich vergrößern. Das ganze Modell würde höchst unübersichtlich werden.
Für voneinander unabhängige Zustände könnte das Problem z.B. dadurch gelöst werden, dass mehrere Zustandsattribute mit voneinander unterschiedlichen Zustandsdefinitionen an einem Datenelement erlaubt würden. Auch diese Lösung würde jedoch die Darstellung unübersichtlicher werden lassen, da die an den Datenkanten annotierten Zustandsinformationen sich auf unterschiedliche Zustandsdefinitionen beziehen.

Eine ähnliche Lösung könnte mit Hilfe der bereits angesprochenen allgemeinen Zustandsobjekte erreicht werden. Wie Abbildung 5.18 zeigt, könnte, anstelle eines Zustandsattributs im Datenelement, letzteres mit einem Zustandsobjekt verbunden werden. Das Zustandsobjekt würde in diesem Fall den Zustand der Daten repräsentieren. Für mehrere voneinander unabhängige Zustandsdefinitionen könnten so mehrere Zustandsobjekte mit einem Datenelement verbunden werden. Dies würde die graphische Repräsentation insofern auflockern, dass Zustandsforderungen und -änderungen durch Kanten zum jeweiligen Zustandsobjekt dargestellt würden.

Sind mehrere Zustandsdefinitionen für ein Datenelement erlaubt, können Zusammenhänge zwischen Zuständen auch indirekt modelliert werden. Die indirekte Modellierung würde so aussehen, dass Aktivitäten z.B. in Folge einer Forderung aus einer Zustandsdefinition, einen Zustandsübergang einer weiteren Definition verlangen könnte. Ein großer Nachteil ist jedoch,

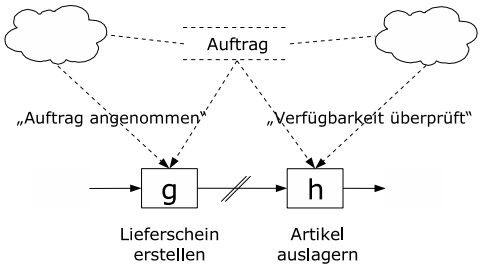

Abbildung 5.18: Mehrere Zustände durch Zustandsobjekte

dass das WfMS den Zusammenhang nicht kennt und auch nicht kontrollieren kann. Er ergibt sich nur indirekt aus den Zustandsforderungen und Zielzuständen der Aktivitäten.

5.5.6 Bezugspunkt für Zustände

Die Zustände im vorgestellten Ansatz beziehen sich ausschließlich auf die Daten selbst. Ein weiterer Aspekt sind jedoch Zustände, die innerhalb einer Komponente bestehen. Je nach Zustand der Komponente sind unterschiedliche Methoden aufrufbar. In einem Bestellprozess ist z.B. der Aufruf der Aktivität „Verfügbarkeit prüfen“ erst nach der Aktivität „Bestellung aufnehmen“ möglich. Dies lässt sich als Zustand des dahinterliegenden Warenwirtschaftssystem ausdrücken [2].

Der vorgestellte Ansatz könnte mit Hilfe der allgemeinen Zustandsobjekte erweitert werden. Ein Zustandsobjekt repräsentiert in diesem Fall die Komponente. Aktivitäten dieser Komponente könnten den Zustand der Komponente über Zustandsparameter fordern und ändern. Hier könnten neue Darstellungen helfen, das Zustandsobjekt als Komponente zu erkennen. Abbildung 5.19 erweitert das Beispiel aus vorigem Abschnitt. Es stellt eine mögliche Darstellung von Komponenten, welche bzgl. eines Datums Zustände besitzen, vor.

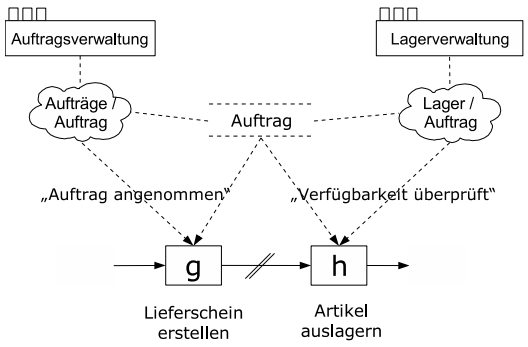

Abbildung 5.19: Komponenten-Zustände durch Zustandsobjekte

Auch hier muss unterschieden werden, ob der Zustand nur innerhalb eines Prozesses gilt oder ob die Komponente einen globalen Zustand besitzt. Zusätzlich kann der Zustand auch

noch an einen Datenkontext gebunden sein. So besitzt das Warenwirtschaftssystem bezüglich einer bestimmten Bestellung z.B. den Zustand „Bestellung angenommen“. Dieser Zustand würde global gelten. Das bedeutet, dass die Komponente sich, bezüglich einer Bestellung, in diesem Zustand befindet, unabhängig davon, welche Prozessinstanz ihre Aktivitäten aufruft. Dadurch existieren Zustände, die sowohl an eine Komponente als auch an Daten gebunden sind.

Der Bezug zur Komponente ist in diesem Fall wichtig für die Funktionalität ihrer Aktivitäten. Würde der Zustand über ein allgemeines Zustandsobjekt bzw. ein Datenelement realisiert, welches auch durch Aktivitäten anderer Komponenten Zustandsänderungen erfahren könnte, ist die Konsistenz zwischen der Komponente und dem Zustand ihres Zustandsobjekts nicht mehr sichergestellt. Dies bedeutet, dass es möglich sein muss, ein Zustandsmodell zu definieren, welches nur seinen Zustand nur durch die Aktivitäten einer einzelnen Komponente ändern kann.

5.6 Zusammenfassung

Das vorgestellte Beschreibungsmodell ermöglicht die Darstellung von komplexen Zusammenhängen zwischen Aktivitäten, die auf Zustände von Daten beruhen. Auch für weiterführende Konzepte, wie z.B. globale Zustandsobjekte, ist das Modell offen und erweiterbar. Inwiefern die angesprochenen Aspekte im praktischen Einsatz eine Rolle spielen, muss jedoch zunächst durch weitere Anforderungsanalysen erarbeitet werden.

Kapitel 6

Ausblick

Das Thema Plug & Play in WfMS umfasst eine Vielzahl von Aspekten. In der vorliegenden Arbeit wurden vor allem Aspekte betrachtet, die das Prozessmodell betreffen. Dazu gehören hauptsächlich mögliche Abhängigkeiten zwischen Aktivitäten. Die Integration der Beschreibungen von Abhängigkeiten ist ein wichtiger Schritt, um Plug & Play anbieten zu können. Die Information, ob eine Aktivität überhaupt an einer bestimmten Stelle im Prozess ausgeführt werden kann, stellt eine notwendige Grundlage dar. Ausgehend von dieser Grundlage können weiterführende „Plug & Play“-Konzepte in WfMS eingeführt werden. Viele weiterführende Fragestellungen, z.B. die Gestaltung von benutzerfreundlichen Oberflächen in diesem Kontext, sind noch offen.

Ein Hauptziel der Integration von Plug & PlayKonzepten in WfMS ist, die Erstellung von Prozessen zu unterstützen und zu beschleunigen. Um dies zu ermöglichen, soll die Auswahl und Komposition von Aktivitäten und Komponenten systemseitig durch das Aktivitäten-Repository unterstützt werden. Dies soll über einen durch Vorschläge des WfMS gestützten, interaktiven Prozess mit dem Benutzer erfolgen. Ebenso gehört dazu, dass das WfMS den Benutzer über fehlgeschlagene Abhängigkeitsprüfungen aufklärt. Neben Erklärungen, die für jeden Benutzer verständlich sind, sollten auch noch Vorschläge zur Lösung des Problems angeboten werden. Ein wichtiges Mittel zur Modellierung und Erklärung von Zusammenhängen sind auch graphische Darstellungen. Sie sollten Prozesse und Abhängigkeiten anschaulich vermitteln können, ohne dass der Benutzer viel über verwendete Notationen lernen muss. Dadurch können die Modellierung von neuen Prozessen sowie Änderungen durch Benutzer zeitnah und effizient umgesetzt werden.

Abgesehen von der Verwendung von Plug & Play innerhalb des WfMS, ist auch die Integration neuer Komponenten ein wichtiger Aspekt. Diese müssen dem WfMS zunächst bekannt gemacht werden. Das bedeutet, dass Dienste und Methoden als Aktivitäten im Beschreibungsmodell des WfMS spezifiziert werden müssen. Auch dieser Vorgang sollte vom WfMS in „Plug & Play“-Manier unterstützt werden. Da bei den Komponenten jedoch heterogene Be-

schreibungmodelle (EJB, UML, WSDL, etc.) zu erwarten sind, müssen hier unterschiedliche Schnittstellen angesprochen werden. Dieser „Import“ neuer Komponenten ist kein zeitkritischer Vorgang. Daher können auch aufwändige Analysen verwendet werden, um dem Benutzer möglichst viel Arbeit bei der Integration zu ersparen.

Abhängigkeiten, welche auf zustandsbehafteten Daten basieren, wurden in der vorliegenden Arbeit ausführlich untersucht und erläutert. Die vorgestellten Ansätze bieten eine einfache bzw. eine umfangreiche Lösung, um die Abhängigkeiten in das Beschreibungsmodell von ADEPT zu integrieren. Bei der Entwicklung der Ansätze wurde eine effiziente Analyse der Zustände im Prozess bedacht. Es wurden auch weitere zustandsbasierte Abhängigkeiten, wie z.B. durch Ressourcen, betrachtet. Der Ansatz mit Zustandsautomaten bietet hier eine gute Grundlage für weiterführende Konzepte.

Literaturverzeichnis

[1] Aristaflow Projekt. `http://www.aristaflow.de`. Abgerufen am 26. Sept. 2005.

[2] H. Acker, C. Atkinson, P. Dadam, S. Rinderle und M. Reichert. Aspekte der komponentenorientierten Entwicklung adaptiver prozessorientierter Unternehmenssoftware. In *Tagungsband 1*, 2004.

[3] Jörg Ackerman, Frank Brinkop, Stefan Conrad, Peter Fettke, Andreas Frick, Elke Glistau, Holger Jaekel, Otto Kotlar, Peter Loos, Heike Mrech, Erich Ortner, Ulrich Raape, Sven Overhage, Stephan Sahm, Andreas Schmietendorf, Thorsten Teschke und Klaus Turowski. Vereinheitlichte Spezifikation von Fachkomponenten. *Memorandum des GI-Arbeitskreises 5.10.3 Komponentenorientierte betriebliche Anwendungssysteme*, 2002.

[4] Peter Dadam, Manfred Reichert und Klaus Kuhn. Clinical Workflows - The Killer Application for Process-oriented Information Systems? Seiten 36–59, 2000.

[5] Peter Dadam, Manfred Reichert, Stefanie Rinderle und Colin Atkinson. Auf dem Weg zu prozessorientierten Informationssystemen der nächsten Generation - Herausforderungen und Lösungskonzepte. 2005.

[6] Peter Fettke und Peter Loos. Entwicklung eines Metamodells für die 'Vereinheitlichte Spezifikation von Fachkomponenten'. *Rundbrief der GI-Fachgruppe WI-MobIS*, 10(2):121–130, 2003.

[7] Object Management Group. Common Object Request Broker Architecture (CORBA), v3.0, Chapter 3. 2002. `http://www.omg.org/cgi-bin/doc?formal/02-06-07`.

[8] Object Management Group. UML 2.0 Superstructure Specification. 2004. Abgerufen am 26. Sept. 2005.

[9] D. Hollingsworth. Workflow Management Coalition - The Workflow Reference Model. Technical report, The Workflow Management Coalition, 1995.

[10] Sun Microsystems. EJB 2.1 Specification Final Release 2.1 FR. 2003. `http://java.sun.com/products/ejb/docs.html`.

[11] Manfred Reichert. *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* Dissertation, Universität Ulm, 2000.

[12] Manfred Reichert und Peter Dadam. ADEPT*flex* - Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[13] Manfred Reichert, Stefanie Rinderle und Peter Dadam. ADEPT Workflow Management System: Flexible Support For Enterprise-wide Business Processes (Tool Presentation). Seiten 371–379.

[14] Manfred Reichert und Dietmar Stoll. Komposition, Choreograhpie und Orchestrierung von Web Services - Ein Überblick. *EMISA Forum*, 24(2):21–32, 2004.

[15] Stefanie Rinderle und Peter Dadam. Schemaevolution in Workflow-Management-Systemen. *Informatik-Spektrum*, 26(1):17–19, 2003.

[16] Steffanie Rinderle. *Schema Evolution in Process Management Systems.* Dissertation, Universität Ulm, 2004.

[17] Jochen Rütschlin, Günter Sauter, Jürgen Sellentin, Klaudia Hergula und Bernhard Mitschang. Komponenten-Middleware - Der nächste Schritt zur Interoperabilität von IT-Systemen. 2001.

Abbildungsverzeichnis

1.1 Zuweisen von Arbeitsschritten mittels Plug & Play 2

2.1 Workflow Reference Model [9] 6
2.2 Business Process Lifecycle 7
2.3 Knotenkonstrukte und Blockstruktur von Prozessen in ADEPT 8
2.4 Zusammenhang zwischen Knoten, Aktivität und dahinterliegender Komponente 9
2.5 Alternative Verzweigung, parallele Verzweigung und Synchronisationskante . 10
2.6 Ein Knoten mit lesender und schreibender Datenkante 11
2.7 Darstellung der Zustände von Aktivitäten 11

3.1 Aktivitäten werden mittels Plug & Play an Knoten hinterlegt 15
3.2 Zusammenhänge im WfMS bezüglich Aktivitätenbeschreibungen 19
3.3 Bearbeitung eines Vorlaufs in einem Buchhaltungsvorgang 24
3.4 Erstellen eines Angebots 25
3.5 Modellierung eines versteckten Datenflusses mit einem virtuellen Datenelement 28
3.6 Modellierung einer semantischen Abhängigkeit durch einen virtuellen Datenfluss 29

4.1 Die Ergänzung der Beschreibung einer Aktivität um Zustandsinformationen . 35
4.2 Setzen und Fordern von Zuständen 35
4.3 Vorlaufprozess mit Zustandsattribut 38
4.4 Vorlaufprozess mit Zuständen in einem virtuellen Datenelement 38

5.1 Von losen Zuständen zu Zustandsautomaten 41
5.2 Zustandsdefinition für den Vorlauf 44
5.3 Mehrere Parameter für ein einzelnes Datenelement 45

5.4 Aktivität mit erweiterter Zustandsbeschreibung 46
5.5 Zustandsdefinition zur Verarbeitung eines Angebots 48
5.6 Verarbeitung eines Angebots mit Zustandsinformationen 48
5.7 Ausführung einer Aktivität mit zustandsbehafteten Daten 49
5.8 Geforderte und erreichbare Zustände in einer fehlerhaften Modellierung . . . 50
5.9 Synchronisation von KF und DF 55
5.10 Vom DF unabhängige Synchronisation 55
5.11 Transitive Synchronisation des DF 56
5.12 Synchronisation durch abwählbare Sync-Kante 56
5.13 Synchronisation aus einem verschachtelten Oder-Block 57
5.14 Synchronisation aller alternativen Zweige 57
5.15 Indirekte Synchronisation aller alternativen Zweige 57
5.16 Ein Drucker als allgemeines Zustandsobjekt 64
5.17 Mehrere Zustände? 66
5.18 Mehrere Zustände durch Zustandsobjekte 67
5.19 Komponenten-Zustände durch Zustandsobjekte 67

Anhang A

Algorithmen

Listing A.1: CacheAndCheckStates

```
1  CacheAndCheckStates(CFS, DFS, d)
2
3  Speichern der Zustände an Knoten welche Zustandsforderungen/-änderungen
4  bezüglich Datenelement d haben sowie an Block-Ende-Knoten.
5
6  Zusätzliche Knotenattribute, Knoten n:
7      RIS(n): ReachableInputStates (duplikatfreie Menge v. Zuständen)
8      ROS(n): ReachableOutputStates (duplikatfreie Menge v. Zuständen)
9      Counter(n): Anzahl bereits bearbeiteter eingehender Kontrollfluss-
10         Kanten. Wird für OR-Joins benötigt, bei parallelen Splits
11         wird eh höchstens ein zweig berücksichtigt.
12     Alle Attribute mit 0 bzw. ∅ initialisiert.
13
14 Prädikate / Funktionen:
15     IS(n): InputStates von n
16     correspondingBlockNode(n): Der Join-Knoten bzgl. eines Splits und
17         Umgekehrt
18     succ(n): Nachfolger von n bzgl. Kontrollfluss-Kanten
19     id(n): ID des Knotens nach topologischer Sortierung.
20     branch(n): Die Verzweigung eines Knotens.
21     c_pred_branch(n, branch): Der Vorgängerknoten von n, welcher auf der
22         Verzweigung branch liegt. Diese Methode ist notwendig, um die
23         Knoten auf einer Sequenz in einem Zweig zu finden.
24
25 Einige wichtige Variablen
26     StateNodes:
27         Menge der Knoten welche von d Zustände fordert/ändert (gegeben).
28     NodeStack:
29         Stack aus Tupeln (node, States0) mit zu bearbeitender
30         Knoten und möglicher Eingangs-Zustände (mit Methoden push und pop).
31     node:
32         Momentan betrachteter Knoten
33     States0:
34         Menge der möglichen Eingangs-Zustände, des momentan betrachteten
35         Knotens.
```

```
36      SuccStates:
37          Menge der möglichen Ausgangs-Zustände, welche nach dem momentan
38          betrachteten Knoten erreichbar sind.
39
40  // Beginn: Prozess-Parameter auf den Startknoten übertragen, diesen
41  // auf den Knoten-Stack legen.
42  RIS(startNode) := Zustände über Eingabeparameter des Prozesses bzw. ∅
43  push NodeStack (startNode, RIS(startNode), ∅)
44
45  // so lange iterieren, bis der Stack leer ist
46  while( (node, States0) := pop NodeStack ) {
47      // 1. Schritt: Überprüfen von Zuständen, Finden und setzen von
48      // Folgezuständen.
49      // Falls n den Zustand von d ändert, oder einen Zustand fordert:
50      if(node ∈ StateNodes){
51          // prüfen ob die Zustandsforderung verletzt wird,
52          // das Ergebnis ist OK, falls:
53          // - Die möglichen Eingangs-Zustände eine Teilmenge der geforderten
54          //   Zustände sind
55          // - oder der spezielle Zustand Any in den Forderungen enthalten ist
56          // - Da der Knoten Zustände fordert, darf die Menge der möglichen
57          //   Eingangs-Zustände nicht leer sein
58          if( RIS(node) ≠ ∅ &&
59              (RIS(node) ⊆ IS(node) || Any ∈ IS(node))
60              ) {
61              // Zustände OK
62          }
63          else {
64              // ungültiger DF
65          }
66
67          // setzen der möglichen Ausgangs-Zustände
68          ROS(node) := alle GoalStates mit RequiredState in RIS(node)
69
70          // falls Any verwendet wird, auch feforderte Zustände weiterreichen
71          if(Any ∈ IS(node))
72              ROS(node) := ROS(node) ∪ (RIS(node) \ RequiredStates)
73
74          // die folgenden möglichen Zustände entsprechen den möglichen
75          // Ausgangszuständen
76          SuccStates := ROS(node)
77      }
78      else {
79          // Falls der Knoten keine Zustandsforderung besitzt, jedoch Zustände
80          // in RIS gesetzt sind, diese auf ROS übertragen (z.B. bei Block-
81          // Anfang-Knoten).
82          if(RIS(node) ≠ ∅)
83              ROS(node) := RIS(node)
84
85          // keine Änderung in der neuen Zustandsmenge
86          SuccStates := States0
87      }
88
```

```
89      // 2. Schritt: Finden von relevanten Nachfolgern und Überspringen von
90      // irrelevanten Blöcken.
91      // Initialisieren der Menge der relevanten Nachfolger.
92      RelevantSuccessors := ∅
93
94      // Block-Anfang-Knoten:
95      // - Falls kein Knoten mit Zustandsforderung innerhalb des Blocks ist,
96      // den zugehörigen Join-Knoten als Nachfolger nehmen.
97      // - Sonst: Untersuchen welche Zweige weiter verfolgt werden müssen.
98      if(node AND-Split, OR-Split, Loop-Start) {
99          // der zum Split gehörende Join-Knoten
100         joinNode := correspondingBlockNode(node)
101
102         // Überprüfe ob ein Knoten mit Zustandsforderung zwischen Split-
103         // und Join-Knoten liegt (unter Verwendung der topologischen
104         // Sortierung):
105         if(∃ n ∈ StateNodes with: id(node) < id(n) < id(joinNode)) {
106             // RIS setzen (falls nicht schon geschehen)
107             if(n ∉ StateNodes)
108                 RIS(node) := States0
109
110             // Zweigen mit Zustandsforderung folgen. Es werden alle
111             // Nachfolger untersucht, im Falle eines OR-Splits können
112             // Knoten (bzw. Zweige) relevant sein. Ansonsten wird
113             // in RelevantSuccessors genau ein Knoten sein.
114             AllSuccessors := c_succ(node)
115             for(sn ∈ AllSuccessors) {
116                 // letzten Knoten der Verzweigung von sn finden
117                 lastBranchNode := c_pred_branch(joinNode, branch(sn))
118
119                 // falls ein Knoten mit Zustandsforderung auf dem Zweig
120                 // liegt, dem Zweig folgen (Aufnahme in RelevantSuccessors).
121                 if(∃ n ∈ StateNodes with: id(sn) <= id(n) <= id(
                        lastBranchNode))
122                     RelevantSuccessors := RelevantSuccessors ∪ sn
123             }
124
125             // Falls es sich um einen OR-Block handelt, und nicht
126             // alle Verzweigungen Zustandsforderungen enthalten,
127             // müssen die aktuellen Zustände ebenfalls übertragen werden.
128             // Zustände der relevanten Verzweigungen werden hinzugefügt.
129             // Korrektur des Counters, da nicht allen Zweigen gefolgt wird.
130             if(node OR-Split && #RelevantSuccessors < #AllSuccessors) {
131                 RIS(joinNode) := ROS(node)
132                 Counter(joinNode) := #RelevantSuccessors
133             }
134         }
135
136         // Es gibt keine Zustandsforderung im Block: Nächster zu
137         // betrachtender Knoten ist der zugehörige Join-Knoten
138         else
139             RelevantSuccessors := joinNode
140     }
```

```
141
142     // Kein Block-Anfang-Knoten: Genau ein Nachfolger in c_succ(node).
143     else {
144         RelevantSuccessors := c_succ(node)
145     }
146
147     // 3. Schritt: Zustände auf relevante Nachfolger übertragen, Nachfolger
148     // auf den Stack legen.
149     for(sn ∈ RelevantSuccessors) {
150         // Falls sn ein Block-Ende-Knoten ist, feststellen ob es im Block
151         // eine Zustandsforderung gab. Ist dies der Fall, muss RIS gesetzt
152         // werden.
153         blockChange := false
154
155         if(sn OR-Join, AND-Join, Loop-End) {
156             // Überprüfen ob Knoten mit Zustandsforderungen zwischen
157             // Split- und Join-Knoten liegen:
158             if(∃ n ∈ StateNodes with: id(correspondingBlockNode(sn)) < id(n)
                    < id(sn)) {
159                 blockChange := true
160             }
161         }
162
163         // Falls in einem AND-Block eine Änderung war:
164         // RIS auf die Folgezustände setzen.
165         if(sn AND-Join && blockChange) {
166             RIS(sn) := SuccStates
167         }
168
169         // OR-Join: Die möglichen Zustände der momentanen Verzweigung der
170         // möglichen Eingangs-Zuständen des Knotens hinzufügen.
171         else if(sn OR-Join && blockChange) {
172             RIS(sn) := RIS(sn) ∪ SuccStates
173             SuccStates := RIS(sn)
174         }
175
176         // Schleifen, Forderung: Am Ende des Loops müssen die gleichen
177         // Zustände wie zu Beginn möglich sein.
178         else if(sn LOOP-End && blockChange) {
179             RIS(sn) := SuccStates
180             if(RIS(sn) ⊈ RIS(correspondingBlockNode(sn))) {
181                 // Fehler im Datenfluss!
182             }
183         }
184
185         // Der Knoten besitzt eine Zustandsforderung, Momentane Zustände
186         // in RIS(sn) speichern.
187         else if(sn ∈ StateNodes) {
188             RIS(sn) := SuccStates
189         }
190
191         // OR-Join: Beim Abarbeiten der letzten relevanten Oder-Verzweigung
192         // den Nachfolger auf den Knoten-Stack legen, ansonsten den Counter
```

```
193             // erhöhen.
194             // Andere Knoten: Nachfolger auf den Knoten-Stack legen.
195             if(node !OR-Join || Counter(sn) = # eing. KF-Kanten - 1)
196                 push NodeStack (sn, SuccStates)
197             else
198                 Counter(sn) := Counter(sn) + 1
199         }
200     }
```

Erklärung

Hiermit erkläre ich, Kevin Göser (Matrikel-Nr. 448692), dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kevin Göser

Ulm, den 30.09.2005