Diplomarbeit

Erweiterte Kontroll- und Datenflusskonzepte in ADEPT

Jonas Wolz

Juli 2008

1. Gutachter: Prof. Dr. Manfred Reichert
2. Gutachter: Dr. Stefanie Rinderle-Ma
Kurzfassung


ADEPT wurde mit dem Ziel entworfen, die in der Praxis relevanten Prozessabläufe bestmöglich zu unterstützen und dabei gleichzeitig die Robustheit und Korrektheit der Prozesse zu garantieren. Wie sich im Laufe der Zeit immer wieder gezeigt hat, lassen sich die meisten in der Praxis vorkommenden Prozesse mit ADEPT2 problemlos modellieren, dieses Ziel wurde also im Großen und Ganzen erreicht.

Allerdings sind in den letzten Jahren in der Praxis auch einige Prozessabläufe aufgefallen, die bislang vom System nicht optimal unterstützt werden, so dass hier ein gewisser Verbesserungsbedarf besteht. Langfristig wird selbstverständlich eine bestmögliche Unterstützung auch dieser Abläufe angestrebt, so dass untersucht werden muss, wie eine Unterstützung der dafür nötigen Ablaufkonzepte sowohl in das Metamodell als auch in die konkrete Implementierung des Systems integriert werden kann.

In dieser Arbeit wird genau dieses für die folgenden Konzepte anhand von Anforderungen aus der Praxis untersucht:

- Variable Parallelität bzw. Multiinstanzaktivitäten
- Lose Synchronisation paralleler Verzweigungen
- Flexible Endknoten
- Abhängige XOR-Entscheidungen
- Verzögerte XOR-Entscheidungen
- ODER-Verzweigungen

Zu jedem dieser Konzepte wird die prinzipielle Umsetzbarkeit diskutiert und wenn nötig die erforderlichen Erweiterungen des ADEPT-Metamodells beschrieben. Desweiteren wird ein Weg zur Implementierung des jeweiligen Konzepts skizziert und ein Abgleich mit den Workflow Patterns vorgenommen.

Als Abrundung wird schließlich die Verfügbarkeit der untersuchten Konzepte in zwei anderen Workflowsprachen (YAWL und WS-BPEL 2.0) untersucht.
Inhaltsverzeichnis

1. Einleitung .......................................................... 9
   1.1. Motivation ....................................................... 9
   1.2. Aufgabenstellung und Zielsetzung .......................... 11
   1.3. Aufbau der Arbeit ........................................... 12

2. Grundlagen ....................................................... 13
   2.1. ADEPT-Metamodell ............................................ 13
       2.1.1. Kontrollfluss ............................................ 15
       2.1.2. Datenfluss .............................................. 18
       2.1.3. Korrektheit .............................................. 19
       2.1.4. Dynamisches Verhalten ................................ 21
       2.1.5. Instanzspezifische Änderungen ....................... 22
       2.1.6. Bearbeiterzuordnung ................................... 23
       2.1.7. Zusammenfassung ....................................... 24
   2.2. Workflow Patterns ........................................... 24

3. Anforderungen .................................................. 29
   3.1. Variable Parallelität ...................................... 29
   3.2. Lose Synchronisation paralleler Verzweigungen .......... 31
   3.3. Flexible Endknoten ......................................... 33
   3.4. Abhängige XOR-Entscheidungen ............................ 34
   3.5. Verzögerte XOR-Entscheidungen ........................... 35
   3.6. ODER-Verzweigungen ....................................... 36

4. Variable Parallelität .......................................... 39
   4.1. Problemstellung ............................................. 39
   4.2. Umsetzungsvarianten ....................................... 40
       4.2.1. Vervielfältigter Teil .................................. 40
       4.2.2. Instanzanzahl .......................................... 44
       4.2.3. Datenübergabe ......................................... 47
       4.2.4. Bearbeiterzuordnung ................................... 49
       4.2.5. Diskussion ............................................. 51
   4.3. Existierendes Metamodell .................................. 54
       4.3.1. Kontrollfluss ........................................... 54
       4.3.2. Datenfluss .............................................. 56
       4.3.2.1. Anforderungen an Listen ......................... 56
       4.3.2.2. Parametermapping ................................... 57
       4.3.2.3. Zugriff auf einzelne Listenelemente ............. 60
       4.3.2.4. Instanzanzahl bei mehreren Listen ............. 62
<table>
<thead>
<tr>
<th>Seite</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4. Zusammenfassung</td>
</tr>
<tr>
<td>5. Lose Synchronisation paralleler Verzweigungen</td>
</tr>
<tr>
<td>5.1. Problembeschreibung</td>
</tr>
<tr>
<td>5.2. Lose Synchronisation mit Weiterlaufen</td>
</tr>
<tr>
<td>5.2.1. Problembeschreibung</td>
</tr>
<tr>
<td>5.2.2. Existierendes Metamodell</td>
</tr>
<tr>
<td>5.2.2.1. Kontrollfluss</td>
</tr>
<tr>
<td>5.2.2.2. Datenfluss</td>
</tr>
<tr>
<td>5.2.3. Zusammenfassung</td>
</tr>
<tr>
<td>5.3. AND-Join mit partiellem Abbruch</td>
</tr>
<tr>
<td>5.3.1. Problembeschreibung</td>
</tr>
<tr>
<td>5.3.2. Existierendes Metamodell</td>
</tr>
<tr>
<td>5.3.2.1. Kontrollfluss</td>
</tr>
<tr>
<td>5.3.2.2. Datenfluss</td>
</tr>
<tr>
<td>5.3.3. Zusammenfassung</td>
</tr>
<tr>
<td>5.4. Zusammenfassung</td>
</tr>
<tr>
<td>6. Flexible Endknoten</td>
</tr>
<tr>
<td>6.1. Problembeschreibung</td>
</tr>
<tr>
<td>6.2. Umsetzungsvarianten</td>
</tr>
<tr>
<td>6.2.1. Flexible Endknoten</td>
</tr>
<tr>
<td>6.2.2. Fehlerendknoten</td>
</tr>
<tr>
<td>6.2.3. Diskussion</td>
</tr>
<tr>
<td>6.3. Existierendes Metamodell</td>
</tr>
<tr>
<td>6.3.1. Kontrollfluss</td>
</tr>
<tr>
<td>6.3.2. Datenfluss</td>
</tr>
<tr>
<td>6.4. Zusammenfassung</td>
</tr>
<tr>
<td>7. Erweiterungen bedingter Verzweigungen</td>
</tr>
<tr>
<td>7.1. Abhängige XOR-Entscheidungen</td>
</tr>
<tr>
<td>7.1.1. Problembeschreibung</td>
</tr>
<tr>
<td>7.1.2. Umsetzungsvarianten</td>
</tr>
<tr>
<td>7.1.2.1. Erkennen der Abhängigkeit</td>
</tr>
<tr>
<td>7.1.2.2. Analyse des Datenflusses</td>
</tr>
<tr>
<td>7.1.3. Existierendes Metamodell</td>
</tr>
<tr>
<td>7.1.3.1. Kontrollfluss</td>
</tr>
<tr>
<td>7.1.3.2. Datenfluss</td>
</tr>
<tr>
<td>7.1.4. Zusammenfassung</td>
</tr>
<tr>
<td>7.2. Verzögerte XOR-Entscheidungen</td>
</tr>
<tr>
<td>7.2.1. Problembeschreibung</td>
</tr>
<tr>
<td>7.2.2. Existierendes Metamodell</td>
</tr>
<tr>
<td>7.2.2.1. Kontrollfluss</td>
</tr>
<tr>
<td>7.2.2.2. Datenfluss</td>
</tr>
<tr>
<td>7.2.3. Zusammenfassung</td>
</tr>
<tr>
<td>7.3. ODER-Verzweigungen</td>
</tr>
<tr>
<td>7.3.1. Problembeschreibung</td>
</tr>
</tbody>
</table>
1. Einleitung

1.1. Motivation


Dies bedeutet insbesondere, dass die Verknüpfung verschiedener Anwendungen nicht mehr manuell durch die einzelnen Mitarbeiter oder durch für jeden Spezialfall unterschiedliche ad-hoc-Lösungen vorgenommen wird, sondern einheitlich unter der Kontrolle des Prozessmanagementsystems stattfindet. Dies erleichtert Änderungen des Prozesses deutlich, da sich möglicherweise ergebende Anpassungsschwierigkeiten sofort erkennen und behoben werden können. Zudem existieren Prozesse durch die explizite Modellierung nicht mehr länger nur implizit in den Köpfen der Mitarbeiter (wobei jeder nur einen kleinen Ausschnitt des Prozesses kennt), womit eine globale Sicht auf den Prozess ermöglicht wird. Durch eine solche globale Sicht können mögliche Optimierungen, wie beispielsweise überflüssige oder doppelte Bearbeitungsschritte, deutlich leichter erkannt werden. Insgesamt können betriebliche Prozesse durch den Einsatz von Prozessmanagementsystemen also deutlich verbessert und anpassungsfähiger gemacht werden [DR05].

Nichtsdestotrotz weisen die meisten der derzeit am Markt befindlichen Systeme noch einige Defizite in Bezug auf Flexibilität und Adaptivität der Prozesse auf, was in der Praxis oft eine nicht unbedeutende Hürde für ihren Einsatz darstellt [DRRA05]. Innerhalb der Abteilung für Datenbanken und Informationssysteme der Universität Ulm wird deshalb seit weit über zehn Jahren auf dem Gebiet des Workflowsystems geforscht, um diese Situation durch die Entwicklung neuer Technologien zum Prozessmanagement langfristig zu verbessern.
Im Rahmen dieser Forschung wurde unter Anderem das ADEPT-Metamodell [Rei00] entworfen. Das ADEPT-Metamodell definiert die Grundlagen der ADEPT-Workflowsprache. Es beinhaltet eine Beschreibung der zur Modellierung eines Prozesses verfügbaren Konstrukte, Korrektheitsbedingungen für den Kontroll- und Datenfluss und darauf aufbauend Änderungsoperationen für sogenannte „instanzspezifische Änderungen“ zur Laufzeit. Eine instanzspezifische Änderung ermöglicht es, auch bereits laufende Prozessinstanzen noch zu ändern, ohne dass das Prozessschema modifiziert werden muss oder die Korrektheit und Konsistenz der Instanz gefährdet ist.

Später wurde dieses Metamodell, vor allem im Rahmen des ADEPT2-Projekts\(^1\), in einem konkreten Prototyp eines Workflowmanagementsystems implementiert, dem ADEPT2-Workflowmanagementsystem [DAG+06, DRRA05]. Durch die fundierte theoretische Grundlage in Form des ADEPT-Metamodells kann dieser Prototyp die korrekte Ausführung eines Prozesses zur Laufzeit schon zur Entwurfszeit weitgehend garantieren. Außerdem ist es möglich, im Rahmen instanzspezifischer Änderungen auch bereits gestartete Prozessinstanzen ohne Beeinträchtigung der Korrektheit und Konsistenz zu modifizieren.

Das ADEPT2-System implementiert aber nicht nur die Konstrukte des ADEPT-Metamodells aus [Rei00], sondern auch einige seitens vorgenommene Erweiterungen, die im Rahmen der Forschungsanstrengungen in der Abteilung entstanden sind. Ein Beispiel dafür ist die Erweiterung des System um semantische Konstrukte, die eine komponentenbasierte Modellierung von Prozessen erleichtern und dabei gleichzeitig die Möglichkeiten zur Korrektheitsprüfung erweitern („Plug & Play“, siehe [AAD+04]).

Eine andere solche Erweiterung ist die sogenannte „Schemaevolution“, das heißt die Möglichkeit, Änderungen an der Prozessvorlage auch auf bereits laufende Instanzen dieser Vorlage zu propagieren – auch, wenn diese Instanzen bereits instanzspezifische Änderungen aufweisen [RRD02, Rin04]. Auch in diesem Fall ist natürlich die Konsistenz und Korrektheit der Prozessinstanzen weiterhin jederzeit sichergestellt.

Insgesamt kann ADEPT also die Korrektheit von Prozessen sowohl zur Entwurfs- als auch zur Laufzeit jederzeit weitgehend garantieren und gleichzeitig eine hohe Flexibilität und Adaptivität insbesondere auch zur Laufzeit bieten.

Trotz bzw. gerade wegen dieser Korrektheitszusicherungen und hohen Flexibilität ist es in ADEPT möglich, die meisten in der Praxis vorkommenden Prozesse auf einfache Weise zu modellieren; die bislang definierten Konstrukte des ADEPT-Metamodells sind also ausreichend, um die meisten praktisch auftretenden Kontrollflussabläufe zu realisieren. Die hohe Flexibilität und zugesicherte Korrektheit geht in ADEPT also nicht zu Lasten der Ausdrucksstärke des Metamodells.

Dass dies nicht selbstverständlich ist, zeigt eine Untersuchung anderer Workflowmanagementsysteme wie in [RRD04]: Die meisten anderen Systeme bieten keine Möglichkeit, mit der sich eine mit ADEPT vergleichbare Flexibilität bei gleichzeitiger Zusicherung der Korrektheit realisieren lässt (womit einer der Vorteile des Einsatzes von Workflowmanagementsystemen, die leichtere Änderbarkeit von Prozessen, teilweise konterkariert wird).

Trotz der erwiesenermaßen hohen Praxistauglichkeit von ADEPT sind im Laufe der Zeit einige Praxisanforderungen an den Prozessablauf aufgefallen, die sich in ADEPT bislang nicht völlig

\(^1\)http://www.uni-ulm.de/in/iui-dbis/forschung/projekte/adept2.html (letzter Besuch: 16.06.2008)
zufriedenstellend umsetzen lassen. Diese Praxisanforderungen lassen sich mit den bisherigen Metamodellkonstrukten nur schwer modellieren bzw. es ist unklar, ob sie sich überhaupt mittels der bisherigen Konstrukte darstellen lassen. Langfristig wird selbstverständlich eine optimale Unterstützung auch dieser Anforderungen angestrebt, so dass eine nähere Untersuchung, wie diese Anforderungen umgesetzt werden können, erforderlich ist.

1.2. Aufgabenstellung und Zielsetzung


Daher wird hier untersucht, wie das Konzept am besten in das ADEPT-Metamodell integriert werden kann bzw. ob es überhaupt sinnvoll in das Metamodell aufgenommen werden kann, wenn es zu konkreten Anforderungen auf die relevantesten Konzepte.

Kонкрет wurden für diese Arbeit die folgenden Konzepte ausgewählt\(^2\):

- Variable Parallelität bzw. Multiinstanzaktivitäten (Kapitel 4)
- Lose Synchronisation paralleler Verzweigungen (Kapitel 5)
- Flexible Endknoten (Kapitel 6)
- Abhängige XOR-Entscheidungen (Abschnitt 7.1)
- Verzögerte XOR-Entscheidungen (Abschnitt 7.2)
- ODER-Verzweigungen (Abschnitt 7.3)

Diese Konzepte haben die Gemeinsamkeit, dass bei allen das Ziel klar ist, der Weg dorthin allerdings nicht: Da die Konzepte auf konkreten Praxisanforderungen beruhen, ist die gewünschte Semantik (das Ziel) relativ klar. Es ist allerdings bislang unklar, wie diese Konzepte im Rahmen des ADEPT-Metamodells sinnvoll umgesetzt werden können (der Weg).

\(^2\) Eine genauere Beschreibung der Konzepte findet sich in Kapitel 3 oder zu Beginn der entsprechenden Kapitel bzw. Abschnitte.
oder, ob eine Integration potentiell mehr Probleme schafft, als sie löst. Die Integration in das Metamodell kann dabei entweder durch eine Erweiterung (beispielsweise einen neuen Knotentyp) oder durch ein Zurückschneiden auf bestehende Konzepte geschehen. Sofern es gewisse Freiheitsgrade bei der Realisierung des entsprechenden Konzepts gibt, werden die verschiedenen Varianten untersucht und die günstigste Variante ausgewählt (diejenige, die die meisten Korrektheitszusicherungen bei möglichst hoher Flexibilität für den Benutzer erlaubt).

Da die Praxisanforderungen alle im Wesentlichen einen bestimmten Kontrollflussablauf vorgeben, die zu erziehende Kontrollflussemantik also klar ist, muss hier vor allem untersucht werden, welche Auswirkungen der gegebene Kontrollflussablauf auf den Datenfluss hat und, wie die Datenflusskorrektheit garantiert werden kann. Als Ergebnis dieser Diskussion ergibt sich jeweils ein Vorschlag, durch welche Maßnahmen das untersuchte Konzept in ADEPT unterstützt werden kann und wie ggf. die Datenflussanalyse daran angepasst werden kann.

1.3. Aufbau der Arbeit

Zunächst werden in dieser Arbeit einige Grundlagen erörtert (Kapitel 2) und die zu erfüllenden Anforderungen für die diskutierten Konzepte erarbeitet (Kapitel 3).


Nach der Diskussion der einzelnen Konzepte wird am Ende dieser Arbeit die Unterstützung der beschriebenen Konzepte in verwandten Arbeiten bzw. anderen Workflowsprachen (YAWL und WS-BPEL 2.0) untersucht (Kapitel 8), die Diskussion nochmals zusammengefasst und ein Ausblick auf noch nicht endgültig geklärte Punkte gegeben (Kapitel 9).
2. Grundlagen

In diesem Kapitel werden die für die Diskussion in den nachfolgenden Kapiteln nötigen Grundlagen in kompakter Form vorgestellt. Da diese Arbeit thematisch stark auf dem ADEPT-Metamodell aufbaut, werden hier zunächst in Abschnitt 2.1 die grundlegenden Konzepte dieses Metamodells dargestellt.

Danach erfolgt in Abschnitt 2.2 eine Einführung in die Workflow Patterns. Die Workflow Patterns stellen eine systemübergreifende Zusammenstellung von immer wieder auftretenden, gemeinsamen „Mustern“ in Workflowsystemen dar, die vor allem als gemeinsame Grundlage dazu dienen sollen, Vergleiche verschiedener Systeme zu erleichtern.

2.1. ADEPT-Metamodell

ADEPT ist ein blockstrukturierter Ansatz zur Modellierung von Workflows. Der Kontrollfluss zur Laufzeit wird über (nicht mengenwertige) Markierungen der Knoten und Kanten des Graphen gesteuert (und nicht über die Weitergabe von Token wie in Petrinetzen [Pet62]). Der Vorteil dieses Ansatzes ist, dass die Laufzeitmarkierungen des Graphen (außer bei Schleifen, s.u.) die Ausführungshistorie des Prozesses jederzeit abbilden und dadurch ohne aufwendige Analysen anhand der Markierung erkennbar ist, welche Knoten bereits ausgeführt wurden, welche gerade ausgeführt werden und welche noch auszuführen sind.

Wie die meisten gängigen Workfowmanagementsysteme unterscheidet ADEPT zwischen dem sogenannten Workflowschema (oder Prozessvorlage) und den konkreten Prozessinstanzen: Das Workflowschema wird zur Entwurfszeit vom Modellierer editiert und beschreibt den Ablauf des Prozesses (vergleichbar mit dem Quelltext eines Programms). Zur Laufzeit können von einem Schema beliebig viele Instanzen mit jeweils unterschiedlichem Ausführungsstatus und verschiedenen konkreten Daten existieren (ein Prozess kann also wie das fertile Programm mehrfach gestartet werden). In ADEPT ist es darüber hinaus möglich, dass einzelne Instanzen instanzspezifische Änderungen aufweisen, der Ablauf also gegenüber der Vorlage modifiziert werden kann, ohne die Ausführung der Prozessinstanz zu beeinträchtigen (dies entspricht dem Austausch des Quelltextes einer laufenden Programminstanzt, ohne diese zu beenden).

Abbildung 2.1: Prozessvorlage und mehrere Instanzen
Das ADEPT-Metamodell (inklusive Erweiterungen) umfasst Konstrukte zur Modellierung der statischen Struktur des Kontrollflusses (Abschnitt 2.1.1), des Datenflusses (Abschnitt 2.1.2) und Korrektheitsbedingungen dafür (Abschnitt 2.1.3). Desweiteren definiert es das dynamische Verhalten dieser Konstrukte (Abschnitt 2.1.4) und Regeln, mit denen Flexibilität zur Laufzeit, das heißt insbesondere instanzspezifische Änderungen, möglich sind (Abschnitt 2.1.5). Zudem ist für ADEPT ein Organisationsmodell mit einer Sprache zur Bearbeiterzuordnung definiert (Abschnitt 2.1.6).

Als Grundlage für diese Arbeit sind vor allem die Kontroll- und Datenflusskonstrukte, ihr dynamisches Verhalten und die Korrektheit wichtig (entsprechend den Kapiteln 3 - 5 in [Rei00]). Die instanzspezifischen Änderungen und das Organisationsmodell sind dagegen von eher untergeordneter Bedeutung, weshalb hier lediglich ein kurzer Überblick über diese Aspekte gegeben wird.

### 2.1.1. Kontrollfluss

Der schematische Aufbau eines ADEPT-Workflowschemas ist in Abbildung 2.2 zu sehen. Wie man erkennt, setzt sich ein ADEPT-Workflow bzw. -Prozess aus ineinander geschachtelten Blöcken verschiedener Art zusammen (vergleichbar mit dem Programmaufbau bei strukturiert der Programmierung [DDH72]).

**Abbildung 2.2:** Schematische Übersicht über das ADEPT-Metamodell
Die wichtigsten blockorientierten Konstrukte in ADEPT sind in Abbildung 2.2 zu sehen:
Das grundlegendste und am häufigsten verwendete Konstrukt ist die Sequenz (z.B. an den Knoten H, I und J). Wie der Name schon nahelegt, werden bei einer Sequenz die Knoten zur Laufzeit sequentiell nacheinander ausgeführt (im Beispiel zuerst H, dann I, dann J).


Die Auswahl des auszuführenden Nachfolgers erfolgt über die Ausgabe eines Entscheidungscode durch eine dem Verzweigungsknoten zugeordnete Entscheidungsaktivität. Jeder ausgehenden Kante des Verzweigungsknotens (und damit jedem Zweig) ist ein solcher Entscheidungscode zugeordnet, so dass durch den Entscheidungscode eindeutig ein Zweig ausgewählt wird. Falls ein nicht zugeordneter Code ausgegeben wird, wird ein vorher bestimmter Defaultzweig ausgewählt

Im Beispiel wird also abhängig von der Entscheidung an D immer nur entweder E oder F ausgeführt; vergleichbar mit einer if-Anweisung oder einem switch ... case in Java [GJSB05].


Das heißt, im Beispiel werden C, H und K parallel gestartet, P wird allerdings erst ausgeführt, wenn sowohl G, J als auch Ò fertig sind. Im Rahmen der gängigen Programmiersprachen ist dies am ehesten mit der Erstellung mehrerer Threads vergleichbar, wobei in einem Hauptthread an einem Punkt auf die Fertigstellung der übrigen Threads gewartet wird (fork und join).

Der dritte in [Rei00] definierte Verzweigungstyp ist die finale Auswahl bei Parallelverzweigungen (Knoten K bis Ò). Dort werden wie bei einer parallelen Verzweigung am Verzweigungsknoten alle Nachfolger gestartet, am Vereinigungsknoten wird jedoch nur ein gültiger Zweig ausgewählt und die anderen Zweige zurückgesetzt (i.d.R. „gewinnt“ der zuerst beendete Zweig; es ist aber auch eine Auswahl durch den Benutzer zur Laufzeit möglich).

Im Beispiel werden also zunächst L, M und N parallel ausgeführt. Sobald der erste Zweig fertig ist, werden die beiden anderen Zweige zurückgesetzt und als übersprungen markiert (insbesondere also in ihrer Ausführung abgebrochen) und Ò aktiviert. Insgesamt ist die finale Auswahl in der Praxis aber gegenüber der bedingten und der parallelen Verzweigung von untergeordneter Bedeutung, so dass sie im ADEPT2-Prototyp derzeit noch nicht implementiert ist.

Neben den verschiedenen Verzweigungstypen definiert ADEPT auch ein Schleifenkonstrukt. Eine Schleife (Knoten A bis Q) wird durch einen Schleifenstart- (A) und einen Schleifenendknoten (Q) begrenzt. Der von der Schleife umgebene Block wird dann zur Laufzeit solange

1Das ADEPT2-System implementiert eine etwas davon abweichende Logik ohne die Notwendigkeit eines Defaultzweigs; für Details siehe Abschnitt 7.1.3.
2.1. ADEPT-Metamodell

ausgeführt, bis die Abbruchbedingung am Schleifenendknoten erfüllt ist. Der Schleifenblock wird also mindestens einmal ausgeführt, ADEPT implementiert demnach eine nicht abweisende Schleife (repeat ... until bzw. do ... while in Java).

Im Beispiel bedeutet dies, dass die Knoten A bis Q zur Laufzeit beliebig oft, aber mindestens einmal, ausgeführt werden können. Ist dabei die Abbruchbedingung bei Ausführung von Q erfüllt, so wird Knoten R aktiviert, die Schleifenausführung somit abgeschlossen. Ist die Bedingung nicht erfüllt, so wird über die Schleifenrücksprungkante Q→A Knoten A aktiviert und eine weitere Iteration der Schleife durchgeführt.

Da sich mit einer strengen Blockstruktur nicht alle sinnvollen Prozessabläufe damit ausdrücken lassen, gibt es zusätzlich blockübergreifende Konstrukte wie beispielsweise Synchronisationskanten:


Unter der Annahme, dass die Synchronisationskante von Knoten F nach I in Abbildung 2.2 eine einfache Synchronisationskante ist, bedeutet diese Semantik, dass I erst dann ausgeführt werden kann, wenn sowohl F als auch der direkte Vorgänger von I (Knoten H) ihre Ausführung beendet haben. Wird an D der obere Zweig ausgewählt, F also übersprungen, so wird I sofort nach Ende der Ausführung von H aktiviert.

Innerhalb dieser Arbeit bezieht sich der Begriff der „Synchronisationskante“ (oder kurz „Synchronisationskante“) ohne Zusatz immer auf einfache Synchronisationskanten, da die strikte Semantik in der Praxis nicht benötigt wird und demnach im existierenden ADEPT2-System derzeit nicht implementiert ist.


Eine elementare Aktivität ist in einer beliebigen Programmiersprache implementiert und stellt aus Sicht des Systems einen atomaren Bearbeitungsschritt dar. „Atomar“ bedeutet hier, dass eine elementare Aktivität die feinste dem System bekannte Verfeinerungsstufe des Prozesses ist. Insbesondere sind dem System die interne Ausführungslogik und internen Zustände unbekannt, eine elementare Aktivität ist aus Sicht des Workflowsystems also eine Art „Black
Nichtsdestotrotz können die von einer elementaren Aktivität durchgeführten Aktionen praktisch beliebig komplex sein.


### 2.1.2. Datenfluss


Ein Beispiel hierfür ist in Abbildung 2.2 zu sehen: Dort schreibt beispielsweise Knoten $\text{H}$ auf das Datenelement $d1$, während Knoten $\text{R}$ dieses Datenelement liest.


Im ADEPT2-System wird diese Zuordnung noch durch weitere Eigenschaften der Parameter und Datenelemente eingeschränkt, um die Korrektheit zu erhöhen und dem Modellierer die Zuordnung weiter zu erleichtern, wie z.B. die Eigenschaft, ob ein Datenelement virtuell ist oder nicht. Diese Details sind für diese Arbeit allerdings nicht weiter relevant und können unter Anderem in [AAD+04] nachgelesen werden.

Bei einer Aktivität wird außerdem zwischen optionalen und obligaten Parametern und dementsprechend am Knoten zwischen optionalen und obligaten Lese- und Schreibzugriffen unterschieden. Der Unterschied zwischen beiden Typen besteht darin, dass bei einem *obligaten* Zugriff zur Laufzeit auf jeden Fall ein gültiger Wert vorhanden sein muss, während bei einem *optionalen* Zugriff auch ein leerer ungültiger Wert, ein sogenannter *NULL*-Wert, vorliegen darf.

Dies impliziert insbesondere, dass zur Laufzeit für einen obligaten Lesezugriff auf ein Datenelement vorher ein obligater Schreibzugriff darauf stattgefunden haben muss (sofern vom Modellierer kein Defaultwert für das Datenelement angegeben wurde). Wenn an einem Knoten sichergestellt ist, dass ein dort obligat gelesenes Datenelement in jedem möglichen Kontrollflussablauf vorher geschrieben wurde, es also auf jeden Fall zur Laufzeit einen gültigen Wert enthält, bezeichnet man das Datenelement und den Lesezugriff an dieser Stelle als „sicher versorgt“. Die sichere Versorgtheit muss schon zur Entwurfszeit vom System überprüft werden.

---

2 Eine Rekursion ist bei ADEPT allerdings nicht möglich. 
3 Ein virtuelles Datenelement repräsentiert einen Datenfluss, der außerhalb des ADEPT2-Systems stattfindet, enthält also keine konkreten Daten.
können, um einen fehlerfreien Ablauf zur Laufzeit sicherzustellen. Dies stellt kein triviales Problem dar (wir werden gleich noch darauf zurück kommen).

Innerhalb dieser Arbeit ist mit einem „Lesezugriff“ oder „Schreibzugriff“ (ohne Zusatz) immer ein obligater Zugriff gemeint, da die obligate Semantik eher der Intuition entspricht und zudem auch aus theoretischer Sicht (Sicherstellung der Versorgtheit von Lesezugriffen) die Interessantere ist.

**2.1.3. Korrektheit**

Mit den bislang vorgestellten Konstrukten lassen sich zwar schon Prozesse modellieren, es ist allerdings noch nicht sichergestellt, dass sie sich zur Laufzeit in jedem Fall korrekt verhalten (es könnten beispielsweise mittels Synchronisationskanten noch Zyklen modelliert werden, die zu Verklemmungen führen). Deshalb werden in [Rei00] Korrektheitsregeln sowohl für die Kontrollflusskonstrukte als auch den Datenfluss definiert.

Informell lassen sich die Korrektheitsregeln für die Kontrollflusskonstrukte wie folgt zusammenfassen (für eine formale Definition siehe die Regeln KF-1 bis KF-8 in [Rei00]):

1. Es gibt genau einen Startknoten ohne eingehende Kanten und genau einen Endknoten ohne ausgehende Kanten.

2. Alle anderen Knoten haben mindestens eine eingehende und mindestens eine ausgehende Kontrollflusskante.

3. Die Blockstruktur wird strikt eingehalten (d.h. ein Graph wie beispielsweise in Abbildung 2.3 a), b) oder c) ist ungültig).

4. Abgesehen von Schleifenrücksprungkanten ist der Graph bezüglich Kontrollfluss- und Synchronisationskanten azyklisch (d.h. ein Verlauf von Kontrollfluss- oder Sync-Kanten wie beispielsweise in Abbildung 2.3 d), e) oder f) ist nicht erlaubt).

5. Synchronisationskanten dürfen nicht zwischen Knoten innerhalb einer Schleife und Knoten außerhalb derselben Schleife verlaufen.

Insgesamt schreiben diese Regeln vor allem eine bestimmte syntaktische Struktur des Graphen vor und lassen sich demnach einfach prüfen. Zudem ist es durch einen entsprechenden syntagmgesteuerten Editor möglich, die Modellierung eines diesen Regeln widersprechenden Graphen von vornherein zu verhindern. Aus diesem Grund wird im Folgenden immer davon ausgegangen, dass diese Strukturierungsregeln für alle vorkommenden Graphen bereits erfüllt sind; die Einhaltung dieser Regeln wird in den in dieser Arbeit vorgestellten Algorithmen also nicht nochmals geprüft.
Abbildung 2.3: Nach den Korrektheitsregeln ungültige Workflowgraphen (ohne Schleifen; aus [Rei00])

Die Korrektheitsbedingungen für den Datenfluss (DF-1 bis DF-3 in [Rei00]) sind:

1. Wenn an einem Knoten K ein Datenelement d obligat gelesen wird, muss unabhängig vom zur Laufzeit gewählten Ausführungspfad sichergestellt sein, dass d vorher mindestens einmal von einem anderen Knoten obligat geschrieben wurde (das Datenelement also an K sicher versorgt sein).


Im Vergleich zu den Korrektheitsbedingungen für den Kontrollfluss ist die Überprüfung der Datenflussregeln nicht ganz trivial (insbesondere, da die Einhaltung dieser Regeln von der Laufzeitsemantik der Knoten abhängt). In [Rei00] werden zwei Algorithmen zur Überprüfung dieser Regeln vorgestellt: WriteExists zur Überprüfung der sicheren Versorgtheit von Datenelementen und ParallelWriterExists zur Überprüfung, ob unerlaubte parallele Schreibzugriffe vorhanden sind.

Die Idee hinter WriteExists ist, dass, wenn ein Datenelement an einem Knoten sicher versorgt ist, es auch an seine Nachfolgern sicher versorgt ist – es sei denn, der Nachfolger ist ein
2.1. ADEPT-Metamodell


Die Funktionsweise von ParallelWriterExists ist dagegen nahe an die zweite Regel angelehnt: Er prüft direkt, ob zwei Knoten, die auf dasselbe Datenelement schreiben, sich innerhalb zweier verschiedener Zweige einer Parallelverzweigung befinden, und, wenn ja, ob ihre Ausführungsreihenfolge durch Sync-Kanten (transitiv) festgelegt wurde.

Zum korrekten Lesen eines Datenelements wird ein weiterer Algorithmus, ReadDataElement, verwendet. Dieser Algorithmus sorgt dafür, dass nicht immer der chronologisch zuletzt geschriebene Wert gelesen wird, sondern dass innerhalb einer parallelen Verzweigung und einer finalen Auswahl Schreibzugriffe eines anderen Zweigs nicht sichtbar sind (und somit deterministisch klar ist, von welchem Knoten der gelesene Wert stammt).

Eine detaillierte Beschreibung der Funktionsweise dieser Algorithmen würde hier allerdings zu weit führen, weshalb dafür auf [Rei00] verwiesen sei.

2.1.4. Dynamisches Verhalten


Abbildung 2.4: Knoten- und Kantenmarkierungen in ADEPT (aus [Rei00])

Der prinzipielle Ablauf zur Laufzeit ist relativ naheliegend: Beim Start des Prozesses sind (bis auf den Startknoten) alle Knoten mit NOT_ACTIVATED markiert. Sobald ein Knoten seine Ausführung abgeschlossen hat (COMPLETED), werden seine Nachfolger in die Arbeitslisten gelegt, also mit ACTIVATED markiert, und später ausgeführt (RUNNING), bis sie selbst wieder COMPLETED sind. Bei einer bedingten Verzweigung werden die nicht ausgewählten Zweige übersprungen und dementsprechend mit SKIPPED markiert.

Für Sync-Kanten sind kompliziertere, zusätzliche Regeln erforderlich, die an dieser Stelle aus Gründen der Übersicht aber nicht näher beschrieben werden.
Der Zustand **SELECTED** tritt nur bei manuellen Aktivitäten auf und signalisiert, dass ein Benutzer die Aktivität bereits für sich zur Bearbeitung reserviert, aber noch nicht mit der Bearbeitung begonnen hat. Ein Knoten wird mit **FAILED** markiert (und die Prozessausführung angehalten), wenn die Ausführung der zugeordneten Aktivität unter Rückgabe eines Fehlercodes gescheitert ist.

Desweiteren ist es jederzeit möglich, die Ausführung einer Aktivität zu unterbrechen (Suspend) und später wieder aufzunehmen (Resume). Zur Unterscheidung dieser beiden Zustände kann ein Knoten mit **SUSPENDED** bzw. **STARTED** markiert sein. In vielen Fällen kann davon allerdings durch den übergeordneten Zustand **RUNNING** abstrahiert werden (vgl. auch Abbildung 2.5).

Wenn an einem Schleifenendknoten die Abbruchbedingung nicht erfüllt ist und demnach eine neue Iteration der Schleife ausgeführt wird, werden die Markierungen des Schleifenkörpers auf **NOT_ACTIVATED** zurückgesetzt und danach der Schleifenanfangsknoten zum Starten der neuen Iteration mit **ACTIVATED** markiert.

**Abbildung 2.5:** Zustandsübergänge zur Laufzeit (aus [Rei00], vereinfacht)

Die dabei an einem Knoten auftretenden Zustände und die möglichen Zustandsübergänge sind in Abbildung 2.5 nochmals als Zustandsdiagramm dargestellt.

Die Semantik der **Kantenmarkierungen** ist wie folgt: Solange der Quellknoten der Kante noch ausgeführt werden kann oder gerade ausgeführt wird, ist die Kante nicht signalisiert (**NOT_SIGNALED**). Hat der Quellknoten seine Ausführung erfolgreich abgeschlossen, wird die Kante mit **true** signalisiert (**TRUE_SIGNALED**). Wurde der Quellknoten übersprungen, ist also nicht mehr ausführbar, wird die Kante mit **false** signalisiert (**FALSE_SIGNALED**).

### 2.1.5. Instanzspezifische Änderungen

gelöscht werden (vgl. auch Abbildung 2.1). Insgesamt sind dabei die selben Änderungen wie auch zur Entwurfszeit möglich.

Die Herausforderung dabei ist, dass durch solche Änderungen weder die oben angegebenen Korrektheitsbedingungen verletzt werden dürfen, noch ein Graph entstehen darf, mit dem der bisherige Prozessablauf unmöglich wäre, da sonst der Zustand des Prozesses inkonsistent werden würde. Es dürfen also insbesondere keine bereits ausgeführten Knoten gelöscht oder ein neuer Knoten in bereits komplett ausgeführte Bereiche des Prozesses eingefügt werden. Zudem müssen möglicherweise die Laufzeitmarkierungen der Knoten im Zuge der Änderungsoperation angepasst werden (z.B. beim Einfügen eines Knotens vor einen bereits aktivierten aber noch nicht gestarteten anderen Knoten: Der bereits aktivierte Knoten muss deaktiviert und stattdessen der neue Knoten aktiviert werden).

In [Rei00] werden deshalb Änderungsoperationen (z.B. zum Einfügen, Löschen oder Verschieben von Knoten) mit zugeordneten zusätzlichen Korrektheitsbedingungen, die definieren, wann eine Operation zulässig ist und wann nicht, vorgestellt.

Darauf aufbauend wurde in [Rin04] die sogenannte „Schemaevolution“ definiert. Eine Schemaevolution erlaubt es, Änderungen an der Prozessvorlage auf die bereits laufenden Instanzen zu propagieren, sogar wenn diese bereits instanzspezifische Änderungen aufweisen. Auch in diesem Fall wird die Konsistenz der modifizierten Graphen anhand von Korrektheitsbedingungen sichergestellt (es werden nur solche Änderungen propagiert, durch die die Korrektheit und Konsistenz der Instanz nicht gefährdet ist).

Insgesamt ist die genaue Funktionsweise instanzspezifischer Änderungen und der Schemaevolution für diese Arbeit allerdings im Detail nicht relevant, weshalb hier auf eine ausführlichere Darstellung verzichtet und für Details auf [Rei00] und [Rin04] verwiesen wird.

2.1.6. Bearbeiterzuordnung

Neben den bereits vorgestellten Kontrollfluss- und Datenflusskonstrukten weist ADEPT zusätzlich ein Organisationsmodell auf [Ber05].

Im ADEPT-Organisationsmodell kann die gesamte Hierarchie und Struktur eines Unternehmens abgebildet und zur Bearbeiterzuordnung genutzt werden.

Das heißt, jedem Knoten wird über eine spezielle Sprache zur Bearbeiterzuordnung ein einzelner oder eine Gruppe von Bearbeitern (z.B. alle Personen mit der Rolle „Sachbearbeiter“) zugewiesen, die diesen Knoten dann zur Laufzeit in ihre Arbeitsliste gelegt bekommen und somit später bearbeiten (der Bearbeiter kann darüber hinaus auch ein automatischer Klient sein). Über diese Sprache sind allerdings nicht nur solche direkten Zuordnungen von Bearbeitern möglich, sondern die Bearbeiter eines Knotens können auch in Abhängigkeit der Bearbeiterzuordnung eines Vorgängers gesetzt werden (und noch viel mehr).

Wie schon bei den instanzspezifischen Änderungen sind weitere Details aber auch hier für diese Arbeit nicht weiter relevant und können in [Ber05] nachgelesen werden.
2. Grundlagen

Datenelement

Beginn einer Parallelverzweigung
("AND-Split")

Beginn einer bedingten Verzweigung
mit Auswahl einer Kante
("XOR-Split")

Beginn einer bedingten Verzweigung
mit Auswahl mehrerer Kanten
("ODER-Split")

Kontrollflusskante

Ende einer Parallelverzweigung
("AND-Join")

Ende einer bedingten Verzweigung
mit Auswahl einer Kante
("XOR-Join")

Ende einer bedingten Verzweigung
mit Auswahl mehrerer Kanten
("ODER-Join")

Einfache Synchronisationskante oder
Datenzugriffskante

Abbildung 2.6: Graphische Darstellung der hier verwendeten Metamodelllemente

2.1.7. Zusammenfassung

In Abbildung 2.6 ist nochmals eine Übersicht über die in dieser Arbeit verwendeten Meta-
modellelemente und ihre graphische Darstellung gegeben. Neben den Notationen aus [Rei00]
wurde für diese Arbeit zusätzlich eine in Abbildung 2.6 ebenfalls dargestellte Notation für be-
dingte Verzweigungen mit Auswahl mehrerer Kanten (sogenannte „ODER-Verzweigungen“, vgl.
Abschnitt 7.3) aus einem internen Arbeitspapier übernommen.

2.2. Workflow Patterns

Die Workflow-Patterns-Initiative wurde um das Jahr 2000 ins Leben gerufen, um für Vergle-
che verschiedener Workflowsysteme eine gemeinsame Diskussionsgrundlage zu schaffen. Ziel war
dabei, Gemeinsamkeiten („gemeinsame Muster“) in den Fähigkeiten verschiedener Workflows-
systeme zu identifizieren und diese abstrakt zu beschreiben, um bei Vergleichen verschiedener
Systeme von der konkreten Implementierung besser abstrahieren zu können.

Dies ist in der Zielsetzung teilweise vergleichbar mit der der Software-Entwurfsmustern (engl.
„Design Patterns“) [GHJV94]: Auch bei diesen ist einer der bedeutendsten Vorteile, dass je-
dem Entwurfsmuster ein eindeutiger Namen gegeben wird, so dass sich zwei Entwickler leichter
über die Muster unterhalten können (da sie sich nicht erst auf einen Namen einigen müssen).
Außerdem können Vor- und Nachteile und mögliche Probleme der Entwurfsmuster zentral do-
kumentiert werden, so dass diese nicht von jedem Entwickler erneut untersucht werden müssen.
Es gibt allerdings auch einen gravierenden Unterschied: Die Workflow Patterns wurden primär
geschaffen, um die Ausdrucksstückigkeit von Workflowsmanagementsystemen zu vergleichen.
Die Entwurfsmuster wurden dagegen entwickelt, um Entwicklern durch vorgegebene Lösungen
häufig auftretender Probleme bei der Erstellung qualitativ hochwertiger Software zu helfen; ein
Vergleich verschiedener Programmiersprachen war und ist kein Ziel der Entwurfsmuster.

Die Verwendung als Vergleichsgrundlage stellt allerdings die wichtigste Anwendungsmöglichkeit
der Workflow Patterns dar: Durch einen Abgleich eines Systems mit den Workflow Patterns

Innerhalb der Workflow-Patterns sind nicht nur Pattern für den Kontrollfluss definiert (20 ursprüngliche Patterns in [AHKB03]; später Erweiterung auf 43 „Workflow Control-Flow Pattern“ in [RHAM06])

5
5Die zusätzlichen Pattern in [RHAM06] sind allerdings oft keine völlig neuen Pattern, sondern vielmehr Aufspaltungen bzw. Verfeinerungen der alten Pattern, so dass sich die Namen und Nummern der Pattern gegenüber [AHKB03] teilweise geändert haben.

, sondern auch „Data Pattern“ [RHEA04], „Resource Pattern“ [RHE04] und „Exception Pattern“ [RAH06]. Es gibt also für alle Perspektiven eines Workflows (Kontrollfluss, Daten, Ressourcen und Exceptions) entsprechende Pattern.

In [RHAM06] (und analog in den anderen Papieren) werden für jedes Pattern eine Beschreibung, ein Beispiel und die nötigen Kontextbedingungen – die Anforderungen, die für eine Unterstützung des Patterns erfüllt sein müssen – angegeben. Zusätzlich werden mögliche Probleme bei der Implementierung des Patterns und die zugehörigen Lösungen diskutiert. Im Anhang von [RHAM06] werden außerdem einige bestehende Workflowsysteme und -sprachen auf ihre Unterstützung der Workflow Patterns hin evaluiert.

In Abbildung 2.7 sind die in einem konkreten ADEPT-Beispielprozesses verwendeten Workflow Patterns zu sehen. Wie man dort leicht erkennt, werden innerhalb der Workflow Patterns Verzweigungs- und Vereinigungsknoten isoliert voneinander betrachtet. Die Workflow Patterns verwenden im Gegensatz zu ADEPT also keine Blockstruktur.

Abbildung 2.7: Beispiel für die in einem ADEPT-Prozess verwendeten Workflow Patterns

Bei Betrachtung aller definerter Patterns fällt auf, – wie am Ende von [RHAM06] im Detail beschrieben wird – dass sich viele Pattern durch eine Kombination mehrerer anderer Pattern ausdrücken lassen (z.B. WCP-6 (Multi-Choice) durch WCP-2 (Parallel Split) und WCP-4 (Exclusive Choice)) bzw. einige Pattern als Generalisierung anderer Pattern betrachtet werden können (z.B. sind WCP-19 (Cancel Activity) und WCP-20 (Cancel Case) Spezialfälle von WCP-25 (Cancel Region)). Bei der Evaluierung eines Systems scheint zu gelten, dass eine Unterstützung eines allgemeineren Patterns als Unterstützung der spezielleren Form gilt – eine
Unterstützung der zusammengesetzten Form eines Patterns dagegen nicht als Unterstützung gewertet wird⁶. Da in [RHAM06] aber keine explizite diesbezügliche Regel aufgestellt wird, bleibt hier bei einer selbst durchgeführten Evaluierung eines Systems eine gewisse Unklarheit bestehen.


Außerdem werden in den Workflow Patterns Aspekte der Korrektheit und Robustheit von Prozessen (wie die Versorgtheit von Lesezugriffen) bislang generell nur eher am Rande betrachtet. Ein möglicher Grund hierfür ist, dass die einzelnen Perspektiven eines Workflows (Kontrollfluss, Daten, Ressourcen und Exceptions) innerhalb der Workflow Patterns immer möglichst unabhängig voneinander betrachtet werden, um ein – für die Verwendung als Vergleichsgrundlage durchaus sinnvolles – hohes Abstraktionsniveau zu erreichen. Viele Korrektheitsprobleme fallen aber gerade erst durch eine integrierte Sicht dieser Perspektiven auf, so dass diese Probleme bei einer isolierten Betrachtung nicht in vollem Umfang erkannt werden können. Somit kann mit Hilfe der Workflow Patterns vor allem die Ausdrucksmächtigkeit von Systemen, nicht aber die Unterstützung von Robustheit, Korrektheit und ähnlichen Eigenschaften, verglichen werden.

⁶Was Systeme wie ADEPT, die sich aus Gründen der Einfachheit des Metamodells in diesem Fall meist mit einer Unterstützung der zusammengesetzten Form zufriedengeben, tendenziell benachteiligt.
⁷ohne die Erweiterungen aus dieser Arbeit
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Unterstützt?</th>
<th>Anmerkungen</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCP-1 (Sequence)</td>
<td>Ja</td>
<td>—</td>
</tr>
<tr>
<td>WCP-2 (Parallel Split)</td>
<td>Ja</td>
<td>AND-Split</td>
</tr>
<tr>
<td>WCP-3 (Synchronization)</td>
<td>Ja</td>
<td>AND-Join</td>
</tr>
<tr>
<td>WCP-4 (Exclusive Choice)</td>
<td>Ja</td>
<td>XOR-Split</td>
</tr>
<tr>
<td>WCP-5 (Simple Merge)</td>
<td>Ja</td>
<td>XOR-Join</td>
</tr>
<tr>
<td>WCP-6 (Multi-Choice)</td>
<td>Nein</td>
<td>ODER-Split (vgl. Abschnitt 7.3)</td>
</tr>
<tr>
<td>WCP-7 (Structured Synchronizing Merge)</td>
<td>Nein</td>
<td>ODER-Join (vgl. Abschnitt 7.3)</td>
</tr>
<tr>
<td>WCP-8 (Multi-Merge)</td>
<td>Nein</td>
<td>Nur bei tokenbasierten Ansätzen sinnvoll.</td>
</tr>
<tr>
<td>WCP-9 (Structural Discriminator)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-10 (Arbitrary Cycles)</td>
<td>Nein</td>
<td>Durch Blockstruktur unmöglich.</td>
</tr>
<tr>
<td>WCP-11 (Implicit Termination)</td>
<td>Nein</td>
<td>Wegen Blockstruktur wenig sinnvoll; Praxistauglichkeit des Patterns allgemein fraglich.</td>
</tr>
<tr>
<td>WCP-12 (Multiple Instances without Synchronization)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-13 (Multiple Instances with a priori Design-Time Knowledge)</td>
<td>Teilweise</td>
<td>Parallelverzweigung mit manuell duplizierten Zweigen.</td>
</tr>
<tr>
<td>WCP-14 (Multiple Instances with a priori Run-Time Knowledge)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-15 (Multiple Instances without a priori Run-Time Knowledge)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-16 (Deferred Choice)</td>
<td>Nein</td>
<td>Verzögerte XOR-Entscheidung (vgl. Abschnitt 7.2)</td>
</tr>
<tr>
<td>WCP-17 (Interleaved Parallel Routing)</td>
<td>Nein</td>
<td>Angabe einer partiellen Ordnung ist durch Sync-Kanten möglich; Aktivitäten ohne Relation zueinander können jedoch immer nebeneinander ausgeführt werden.</td>
</tr>
<tr>
<td>WCP-18 (Milestone)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-19 (Cancel Activity)</td>
<td>Teilweise</td>
<td>Vorwärtssprung ohne Nachholen (Einschränkungen durch Blockstruktur; nur Zurücksetzen, aber kein Abbruch)</td>
</tr>
<tr>
<td>WCP-20 (Cancel Case)</td>
<td>Nein</td>
<td>Nicht im Metamodell vorgesehen; möglich durch manuellen Aufruf der entsprechenden Schnittstellenmethode.</td>
</tr>
<tr>
<td>WCP-21 (Structured Loop)</td>
<td>Ja</td>
<td>—</td>
</tr>
<tr>
<td>WCP-22 (Recursion)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-23 (Transient Trigger)</td>
<td>Nein</td>
<td>Nicht im Metamodell; Aktivitäten können im implementierten System jedoch Signale empfangen.</td>
</tr>
<tr>
<td>WCP-24 (Persistent Trigger)</td>
<td>Nein</td>
<td>Siehe WCP-23.</td>
</tr>
<tr>
<td>WCP-25 (Cancel Region)</td>
<td>Teilweise</td>
<td>Vorwärtssprung ohne Nachholen (Einschränkungen durch Blockstruktur; nur Zurücksetzen, aber kein Abbruch)</td>
</tr>
<tr>
<td>WCP-26 (Cancel Multiple Instance Activity)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>Pattern</td>
<td>Unterstützt?</td>
<td>Anmerkungen</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>WCP-27 (Complete Multiple Instance Activity)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-28 (Blocking Discriminator)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-29 (Cancelling Discriminator)</td>
<td>Teilweise</td>
<td>Finale Auswahl (dafer kein Abbruch ohne Zurücksetzen bereits ausgeführter Aktivitäten möglich)</td>
</tr>
<tr>
<td>WCP-30 (Structured Partial Join)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-31 (Blocking Partial Join)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-32 (Cancelling Partial Join)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-34 (Static Partial Join for Multiple Instances)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-35 (Cancelling Partial Join for Multiple Instances)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-36 (Dynamic Partial Join for Multiple Instances)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-38 (General Synchronizing Merge)</td>
<td>Nein</td>
<td>Durch dieses Pattern gelöstes Problem tritt nur bei tokenbasierten Ansätzen auf; bei ADEPT daher äquivalent zu WCP-37.</td>
</tr>
<tr>
<td>WCP-39 (Critical Section)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-40 (Interleaved Routing)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-41 (Thread Merge)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-42 (Thread Split)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-43 (Explicit Termination)</td>
<td>Teilweise</td>
<td>Entspricht dem Endknoten; dieser kann aber erst erreicht werden, wenn alle Aktivitäten abgeschlossen sind und nicht schon vorher.</td>
</tr>
</tbody>
</table>

Tabelle 2.1: Bislang unterstützte Workflow Patterns
3. Anforderungen

Nachfolgend werden die in dieser Arbeit behandelten Konzepte und die daran gestellten, aus der Praxis stammenden Anforderungen beschrieben. Jedem Konzept ist dafür ein eigener Abschnitt zugeordnet, in dem die zugehörigen Anforderungen diskutiert werden.

Quelle der behandelten Anforderungen sind dabei sowohl einige vorangegangene Arbeiten (hauptsächlich [Bre06, Beu02, Ste05]) als auch diverse aus der Praxis stammende Prozesse, die sich bislang nicht zufriedenstellend modellieren ließen.

3.1. Variable Parallelität

Auf abstrakter Ebene beschreibt der Begriff der „variablen Parallelität“ das Konzept, dass die Anzahl der Zweige in einer Parallelverzweigung nicht statisch zur Entwurfszeit festgelegt wird, sondern, dass diese Anzahl dynamisch zur Laufzeit bestimmt wird.

![Abbildung 3.1: Konzeptuelle Idee hinter variabler Parallelität (nach [Beu02])](image)

Um dieses Konzept gegenüber allgemeinen instanzspezifischen Änderungen noch etwas weiter abzugrenzen, wird dies in der Regel so interpretiert, dass zur Entwurfszeit ein Zweig als Vorlage vormodelliert wird, der dann zur Laufzeit vervielfältigt wird (so auch hier und in [Beu02]). Dies ist in Abbildung 3.1 beispielhaft illustriert.

Da bei dieser Interpretation ein einzelner Zweig mehrfach instanziert wird, kann man die variable Parallelität auch als ein Multiinstanzkonzept bezeichnen (vgl. „multiple instances“ in den Workflow Patterns). Dies kann soweit gehen, dass die variable Parallelität als *Multiinstanzaktivität* implementiert ist.\(^1\)

\(^1\) Wie z.B. in YAWL (siehe [HA05] oder Abschnitt 8.2)
Da es oft nicht sinnvoll ist, dass jede Instanz des Zweiges exakt dasselbe tut, muss es eine Möglichkeit geben, den einzelnen Instanzen jeweils eigene Ein- und Ausgabeparameter zuzuweisen [Beu02]. Da diese Daten vor der Stelle der variablen Parallelität erzeugt werden müssen und hinterher ggf. weiterverarbeitet werden sollen, muss es in diesem Zusammenhang auch irgendeine Möglichkeit geben, zur Laufzeit eine dynamische Anzahl von Objekten in einem Datenelement zu speichern. Als eine naheliegende Möglichkeit bieten sich hier Listen dynamischer Länge an (oder noch allgemeiner dynamische Baumstrukturen wie in [Beu02]).

Im Rahmen der Variablen Parallelität kann es außerdem sinnvoll sein, den verschiedenen Zweiginstanzen verschiedene Bearbeiter zuzuordnen [Beu02, Ste05]. Dieses ist insbesondere bei manuellen Tätigkeiten wie in Abbildung 3.1 der Fall, da es dort oft nicht sinnvoll ist, wenn ein Bearbeiter sämtliche Zweiginstanzen bearbeitet muss, sondern jede Instanz einem anderen Bearbeiter zugewiesen werden muss.

Sofern sich eine Stelle variabler Parallelität innerhalb einer Schleife befindet, ist es wünschenswert, dass die Anzahl der Zweige je Iteration variabel und nicht etwa nach der ersten Iteration unveränderlich ist [Beu02]. Beispielsweise ist der in Abbildung 3.1 dargestellte Ausschnitt eines Entwicklungsprozesses typischerweise in eine Schleife eingebettet, in der vorher die Liste der noch zu entwickelnden Teile bestimmt wird. Je Iteration sind dies oft verschiedene (und insbesondere verschieden viele) Teile.

Da für die variable Parallelität und insbesondere den Listenzugriff in ADEPT bislang noch keine graphischen Notationen vorhanden sind, beide aber für Beispiele im Rahmen der variablen Parallelität benötigt werden, wird im Rahmen dieser Arbeit die in Abbildung 3.2 dargestellte Notation verwendet.

Abbildung 3.2: Neu eingeführte Notationselemente

Die (informelle) Bedeutung der einzelnen Notationselemente ist dabei wie folgt:

a) **Splitknoten für variable Parallelität**: Markiert den Beginn eines Blocks mit variabler Parallelität.

b) **Joinknoten für variable Parallelität**: Markiert das Ende eines Blocks mit variabler Parallelität.

c) **Lesender indizierter Listenzugriff mit konstantem Index**: Liest das Listenelement mit dem angegebenen, nullbasierten Index \( \text{get}(1) \) gibt z.B. das zweite Element zurück.

d) **Lesender indizierter Listenzugriff mit variablen Index**: Liest das Listenelement mit dem im angegebenen Datenelement gespeicherten Index \( \text{get}(\$d) \) liest z.B. das Element, dessen Index im Datenelement mit Namen \$d gespeichert ist.

\[ a) \quad \begin{array}{c}
\text{c)} \quad \begin{array}{c}
\text{set}(X) \\
\text{e)} \quad \begin{array}{c}
\text{d)} \quad \begin{array}{c}
\text{f)} \quad \begin{array}{c}
\text{g)} \quad \begin{array}{c}
\text{b)} \quad \begin{array}{c}
\text{a)} \\
\text{b)} \\
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array} \]
3.2. Lose Synchronisation paralleler Verzweigungen


![Abbildung 3.3: Beispiel für lose Synchronisation](image-url)

Am Ende des Prozesses findet ein globales Qualitätsmanagement statt, das die Ergebnisse der einzelnen Qualitätsmanagementschritte verwendet, sofern diese vorhanden sind. Auch dieses globale Qualitätsmanagement darf nicht durch die vorhergehenden Qualitätsmanagement- schritte verzögert werden.

In Folge dessen muss die Möglichkeit bestehen, die Qualitätsmanagementaktivität in jedem Schritt parallel zur Beschwerdebearbeitung zu starten, ohne, dass am Joinknoten am Ende jeden Schrittes darauf synchronisiert wird (z.B. darf das Qualitätsmanagement des zweiten Schritts auch nicht auf das des ersten warten!).

In diesem Prozess gibt es hierbei – wie auch generell bei loser Synchronisation – zwei Möglichkeiten, wie mit den unsynchronisierten Zweigen umgegangen werden kann: Diese können entweder bis zu ihrem Ende weiterlaufen, oder sie können abgebrochen werden, sobald alle Zweige, die synchronisiert werden, beendet sind. Diese beiden Varianten haben ihre spezifischen Vor- und Nachteile, die in Kapitel 5 entsprechend diskutiert werden. Im oben beschriebenen Praxisbeispiel ist es von untergeordneter Bedeutung, ob das Qualitätsmanagement weiterlaufen darf oder abgebrochen wird, es ist also gleichgültig, welche dieser Varianten gewählt wird. Dennoch kann man davon ausgehen, dass es in der Praxis auch Prozesse gibt, bei denen eine der beiden Varianten deutlich sinnvoller ist als die andere (so dass die Unterstützung möglichst beider Semantiken angestrebt wird).

Abbildung 3.4: Für die lose Synchronisation verwendete Notationen

Wie schon bei der variablen Parallelität müssen auch hier einige neue Notationen eingeführt werden. Diese werden in Abbildung 3.4 aufgeführt und kurz beschrieben.
Zusammengefasst sind folgende Anforderungen im Rahmen der loser Synchronisation festzuhalten:

(1) Innerhalb einer Parallelverzweigung muss es möglich sein, zwei verschiedene Arten von Zweigen zu definieren:
   
a) **Masterzweige:** Auf diese Zweige wird wie bisher am Joinknoten gewartet, das heißt, der Joinknoten wird erst aktiviert, wenn diese Zweige komplett ausgeführt wurden.

   b) **Slavezweige:** Auf diese Zweige wird am Joinknoten nicht gewartet, das heißt, für die Aktivierung des Joinknotens ist es belanglos, ob diese Zweige komplett ausgeführt wurden oder nicht.

(2) Entweder dürfen die Slavezweige in jedem Fall zu Ende laufen (auch wenn der Joinknoten bereits ausgeführt wurde)

(3) Oder die Slavezweige werden bei Aktivierung des Joinknotens abgebrochen.

Bei den Anforderungen (2) und (3) reicht es aus, wenn *eine* der beiden Anforderungen erfüllt ist.

### 3.3. Flexible Endknoten

Der Begriff der „flexiblen Endknoten“ beschreibt die Möglichkeit, einen Endknoten an jeder Stelle des Graphen einfügen zu können, also mehr als nur einen einzigen Endknoten am Ende des Graphen zu unterstützen. Falls sich die Einfügestelle dabei innerhalb einer Parallelverzweigung befindet, müssen die parallel laufenden Aktivitäten bei Ausführung des Endknotens abgebrochen werden (u.a. da dies die intuitive Semantik ist: Ein *Endknoten beendet* den Prozess.).

![Abbildung 3.5: Beispiel für flexible Endknoten](image)

Dies wird beispielsweise in Entwicklungsprozessen wie in [Bre06] benötigt, da dort die Möglichkeit bestehen muss, den Prozess an fast jeder beliebigen Stelle abbrechen zu können. Ein vereinfachtes Beispiel hierfür findet sich in Abbildung 3.5: Wenn bei der Kostenprüfung eines Teils festgestellt wird, dass die Entwicklung dieses Teils zu teuer ist, muss der gesamte Entwicklungsprozess, also insbesondere auch die Entwicklung der übrigen Teile, angehalten werden.

Da ein flexibler Endknoten aus Benutzersicht einfach ein Endknoten ist, der an jeder beliebigen Stelle des Prozesses eingefügt werden kann, muss keine neue Notation eingeführt werden. Flexible Endknoten werden daher nachfolgend graphisch wie gewöhnliche Endknoten dargestellt.

Zusammengefasst gibt es folgende Anforderungen an flexible Endknoten:

1. Es muss möglich sein, einen Endknoten an einer quasi beliebigen Stelle des Workflowgraphen einzufügen zu können (überall dort, wo auch ein gewöhnlicher Knoten möglich wäre).

2. Sobald ein flexibler Endknoten gestartet wurde, ist der Prozess beendet, das heißt, parallel laufende Aktivitäten werden abgebrochen.

### 3.4. Abhängige XOR-Entscheidungen

Als „abhängige XOR-Entscheidungen“ werden solche XOR-Entscheidungen bezeichnet, deren Entscheidung von einer früheren abhängig ist, d.h. deren getroffene Entscheidung zur Laufzeit immer identisch zu einer früheren ist. Ein einfaches Beispiel für einen solchen Fall ist in Abbildung 3.6 dargestellt: Die Zahlungsart ändert sich im Verlauf des Bestellprozesses nicht, also entscheiden sich beide XOR-Verzweigungen (Zahlungsart?) gleich.

Bislang wird dieser Fall bei der Datenflussanalyse in [Rei00] nicht berücksichtigt. Beispielsweise wird (obligates Lesen angenommen) der Datenfluss des Prozesses in Abbildung 3.6 bislang wegen nicht sicher versorgter Lesezugriffe an den Knoten Lastschrift erstellen und Kreditkartenbuchung zurückgewiesen.

Durch eine statische Analyse zur Entwurfszeit kann man hier aber eigentlich feststellen, dass der Knoten Lastschrift erstellen nur dann zur Ausführung kommt, wenn die Zahlungsart...
3.5. Verzögerte XOR-Entscheidungen

*Lastschrift* gewählt wurde. In diesem Fall wird aber auch immer *Kontodaten erfassen* ausgeführt, so dass der Lesezugriff in jedem Fall sicher versorgt ist. Für den Knoten *Kreditkartenz. buchen* gilt Entsprechendes.

Da eine solche Konstellation in der Praxis häufig anzutreffen ist, ist es wünschenswert, dass ADEPT solche Kontrollflussabhängigkeiten automatisch erkennt und bei der Datenflussanalyse berücksichtigt, in unserem Beispiel also den Datenfluss als sicher akzeptiert.

Desweiteren ist es sinnvoll, wenn solche Abhängigkeiten explizit modelliert werden können, so dass möglich Fehler (Übersehen eines zu ändernden Prädikats) bei der Aktualisierung von Entscheidungsprädikaten vermieden werden können und außerdem ein kleiner Geschwindigkeitsgewinn erzielt wird (da das selbe Prädikat nicht mehr mehrfach ausgewertet werden muss).


Zusammengefasst bleiben folgende Anforderungen festzuhalten:

1. Entscheidungsabhängigkeiten zwischen XOR-Entscheidungen müssen zur Entwurfszeit erkannt und bei der Datenflussanalyse berücksichtigt werden (wenn Zweig B einer abhängigen XOR-Entscheidung immer ausgeführt wird, wenn in einer früheren XOR-Verzweigung Zweig A ausgewählt wurde, so dürfen alle innerhalb von Zweig A obligat geschriebenen Datenelemente in Zweig B obligat gelesen werden)

2. Es muss möglich sein, diese Abhängigkeiten explizit zu modellieren

### 3.5. Verzögerte XOR-Entscheidungen

Als „verzögerte XOR-Entscheidung“ werden XOR-Entscheidungen bezeichnet, bei denen die Auswahl des Nachfolgezweigs implizit durch den Start einer Nachfolgeaktivität des Splitknotens aus der Arbeitsliste erfolgt (und nicht explizit durch eine Entscheidungsaktivität). Dieses Konzept wurde auch schon als „implizite Auswahl“ bei XOR-Splitknoten in [Rei00] beschrieben, allerdings dort nicht im Detail ausgearbeitet.


Dieses Konzept lässt sich in der Praxis vielfältig einsetzen, beispielsweise neben einem Szenario wie in Abbildung 3.7 auch zur Modellierung einfacher Zeitbedingungen (ein automatischer Klient startet eine Aktivität nach einer gewissen Zeitspanne).
Um verzögerte XOR-Entscheidungen in Diagrammen zu kennzeichnen, werden diese in dieser Arbeit wie in Abbildung 3.7 wie gewöhnliche XOR-Splitknoten, die kursiv mit „Verzögert“ beschriftet sind, dargestellt.

Zusammengefasst bleibt lediglich eine Anforderung zu erfüllen:

(1) Bei XOR-Entscheidungen soll es wahlweise möglich sein, die Entscheidung nicht explizit durch eine Entscheidungsaktivität zu treffen, sondern implizit durch den Start des gewünschten Zweigs innerhalb der Arbeitslisten.

### 3.6. ODER-Verzweigungen

Als „ODER-Verzweigung“ wird eine bedingte Verzweigung bezeichnet, bei der zur Laufzeit eine beliebige Teilmenge der Zweige ausgewählt werden kann, die dann parallel ausgeführt werden. Hier werden also am Verzweigungsknoten $m$ von $n$ Zweigen ausgewählt – im Gegensatz zu 1 von $n$ Zweigen bei XOR-Verzweigungen oder $n$ von $n$ Zweigen bei AND-Verzweigungen. Am zugehörigen Vereinigungsknoten wird bei einer ODER-Verzweigung dementsprechend darauf gewartet, bis alle $m$ ausgewählten Zweige ihre Ausführung beendet haben.

Wie schon verzögerte XOR-Entscheidungen wurden auch ODER-Verzweigungen in [Rei00] erwähnt, allerdings nicht detalliert behandelt. Außerdem ist in diesem Zusammenhang zu beachten, dass das hier als ODER-Verzweigung bezeichnete Konzept nicht das selbe ist wie die OR-Verzweigung in [Rei00], obwohl die Bezeichnung das nahe legen könnte. Bei der OR-Verzweigung aus [Rei00] wird nur ein Zweig ausgewählt und nicht wie hier mehrere, es handelt sich also um ein *exklusives* Oder, während die ODER-Verzweigung ein *inklisches* Oder darstellt. Um Verwechslungen zu vermeiden, wird eine 1 aus $n$-Auswahl hier konsequent als „XOR-Verzweigung“ bezeichnet, während die $m$ aus $n$-Auswahl „ODER-Verzweigung“ genannt wird.


Für die Darstellung von ODER-Verzweigungen wird die schon in den Abbildungen 2.6 und 3.8 gezeigte Notation übernommen.
Zusammengefasst haben ODER-Verzweigungen folgende Anforderungen zu erfüllen:

(1) Eine Entscheidungsaktivität wählt nicht nur einen, sondern beliebig viele, Zweige einer ODER-Verzweigung aus

(2) Die ausgewählten Zweige werden nebenläufig ausgeführt

(3) Der Joinknoten wird erst ausgeführt, wenn alle ausgewählten Zweige ihre Ausführung beendet haben
4. Variable Parallelität

Das Konzept der variablen Parallelität beschreibt die Möglichkeit, zur Entwurfszeit einen Zweig vorzumodellieren, der dann zur Laufzeit mehrfach instanziert bzw. vervielfältigt wird (mit einer erst zur Laufzeit festgelegten Anzahl von Instanzen).

Je nach genauer Implementierung kann man die variable Parallelität auch als Multiinstanzaktivität bezeichnen. Bei einer solchen Implementierung als Multiinstanzaktivität wird einem speziellen Mehrinstanzknoten eine mehrfach instanzierte Aktivität (i.d.R. ein Subprozess) zugewiesen.


4.1. Problembeschreibung


Abbildung 4.1: Beispiel für variable Parallelität (angelehnt an [Bre06])


Außerdem müssen in jedem Schleifendurchlauf potentiell verschieden viele Reviewer befragt werden – je nachdem, wie umfangreich die in der entsprechenden Iteration zu untersuchende Änderung ist.

Welchen Workflow Patterns die variable Parallelität genau entspricht, hängt stark von der konkreten Interpretation und Implementierung dieses Konzeptes ab (vgl. auch Abschnitt 4.2). Im Wesentlichen kommen dafür die Pattern WCP-14 oder WCP-15 (Multiple instances with/without a priori run-time knowledge), ggf. kombiniert mit WCP-34 (Static Partial Join for Multiple Instances), in Frage. Daher werden wir in der Zusammenfassung zu diesem Kapitel nochmals auf die Workflow Patterns zurückkommen und klären, welchen Pattern die in diesem Kapitel für ADEPT gewählte Implementierung entspricht.

4.2. Umsetzungsvarianten

Bei variabler Parallelität gibt bezüglich der Implementierung einiger Aspekte Freiheitsgrade, die in diesem Abschnitt diskutiert werden. Für jeden Aspekt werden in einem eigenen Unterabschnitt die möglichen Varianten diskutiert, bevor anschließend am Ende dieses Abschnitts die vorteilhafteste Kombination ausgewählt wird. Diese wird später in diesem Kapitel in Hinblick auf eine Implementierung in ADEPT genauer untersucht.

4.2.1. Vervielfältigter Teil

Der für den Benutzer augenfälligste Aspekt der Modellierung ist die Art und Weise, wie der zur Laufzeit zu vervielfältigende Teil bzw. „Zweig“ des Prozesses modelliert wird. Nichtsdestotrotz kann die Modellierung des vervielfältigten Teil aus theoretischer Sicht oft orthogonal zu den nachfolgend diskutierten anderen Aspekten der variablen Parallelität betrachtet werden. Im Folgenden wird daher ein beliebiges Konstrukt zur Modellierung variabler
4.2. Umsetzungsvarianten

Abbildung 4.2: Varianten der Modellierung des vervielfältigten Teils
Parallelität (unabhängig von der hier gewählten Variante) als „variabler Parallelitätsblock“ bezeichnet.

Grundsätzlich gibt die folgenden Varianten, wie der zu vervielfältigende Teil modelliert werden kann:

1. **Modellierung als gewöhnlicher Zweig** innerhalb je eines Split- und Joinknotens für variable Parallelität, wie in Abbildung 4.2a (oder 3.1) dargestellt


3. **Modellierung als Scope**, mit eigenem Gültigkeitsbereich für Variablen bzw. Datenlemente, sofern das Workflowsystem ein solches Konzept unterstützt (z.B. bei WS-BPEL 2.0 [JEAn+07, BBn+07]), quasi als Zwischenstufe zwischen Zweig und Subprozess

**Modellierung als Zweig** Eine Modellierung als Zweig ist für Benutzer die vermutlich intuitivste Variante, da die Modellierung völlig analog zu der normaler Parallelverzweigungen erfolgen kann. Insbesondere ist es sehr einfach, auf beliebige Datenlemente des umgebenden Prozesses zuzugreifen, da auf diese mittels gewöhnlicher Datenzugriffskanten zugegriffen werden kann.


In diesem Zusammenhang ergibt sich ein weiteres Problem: Wenn eine Aktivität im vervielfältigten Zweig schreibend auf ein nicht-listenwertiges Datenelement zugreift, wird dies zur Entwurfsszeit zunächst erlaubt werden, da zu diesem Zeitpunkt nur ein einziger Zweig modelliert wurde, womit der Schreibzugriff zulässig ist. Zur Laufzeit wird diese Aktivität dann mitsamt der Schreibkante vervielfältigt, so dass nun mehrere Zweige nebeneinander auf das selbe Datenelement zugreifen möchten. Um einen korrekten Datenfluss zu gewährleisten, ist dies bei
ADEPT allerdings nicht zulässig. Dementsprechend wird in diesem Fall aufgrund der durchgeführten Korrektheitsanalysen die entsprechende instanzspezifische Änderung (zur Laufzeit!) vom System zurückgewiesen. Damit scheitert zur Laufzeit die Instanziierung des Zweiges und die Ausführung des Prozesses muss mit einem Fehler angehalten werden, um eine inkorrekte Ausführungssemantik der variablen Parallelität zu verhindern. Dies ist selbstverständlich nicht akzeptabel, so dass man (zumindest innerhalb des Editors) eine Analyse vorsehen muss, die solche Zugriffe bereits zur Entwurfszeit erkennt und beanstandet.

Ein weiteres Problem sind „temporäre Variablen“ innerhalb des vervielfältigten Zweiges, also Variablen, die nur innerhalb des Zweigs zur Speicherung von Zwischenergebnissen benötigt werden (wie z.B. das Datenelement Bestellmenge in Abbildung 4.2). Durch die parallele Ausführung können dafür nicht einfach nicht-listenwertige Datenelemente verwendet werden. Daher muss der Modellierer (wie in Abbildung 4.2a) explizit Listen zur Ablage temporärer Werte zu verwenden. Dies macht die Modellierung allerdings relativ aufwendig und wenig intuitiv.


Nicht zuletzt ergeben sich bei ADEPT durch die bei einer Modellierung als Zweig nötigen instanzspezifischen Änderungen und die damit verbundenen, zur Laufzeit durchgeführten, aufwendigen Korrektheitsanalysen merkliche Beeinträchtigungen der Ausführungsgeschwindigkeit des Prozesses.


Im Falle von ADEPT ist hierbei auch eine deutlich bessere Ausführungsgeschwindigkeit als mit instanzspezifischen Änderungen zu erwarten.

Ein weiterer Vorteil besteht darin, dass ein Subprozess einen eigenen Gültigkeitsbereich für Datenelemente bereitstellt, so dass es keine Probleme mit temporären Variablen innerhalb des vervielfältigten Teils gibt. Außerdem werden die Ein- und Ausgabeparameter explizit definiert, wodurch zur Laufzeit die Übergabe der Daten an die einzelnen Instanzen deutlich vereinfacht wird, da keine aufwendigen Analysen erforderlich sind, um die übergebenen (listenwertigen) Datenelemente herauszufinden.

Wie später in diesem Kapitel noch genauer begründet wird, können die nötigen Maßnahmen zur Verteilung der Ein- und Ausgabedaten bei dieser Variante vollständig innerhalb des Mehrinstanzknotens durchgeführt werden, so dass dieser aus Sicht des umgebenden Prozesses wie
ein gewöhnlicher Knoten behandelt werden kann. Auch innerhalb des Subprozesses sind keine Modifikationen zur Unterstützung der variablen Parallelität erforderlich.

Der bedeutendste Nachteil dieser Variante besteht darin, dass die Modellierung ohne eine spezielle Editorunterstützung deutlich unkomfortabler ist als bei einer Modellierung der variablen Parallelität als Zweig. So muss zur Modellierung des Subprozesses in vielen Editoren beispielsweise ein separates Fenster geöffnet werden. Zudem müssen, um innerhalb des Subprozesses auf ein weiteres Datenelement zugreifen zu können, aufgrund der explizit definierten Ein- und Ausgabeparameter sowohl die Parameter des Subprozesses als auch die Parameterzuordnung am Mehrinstanzknoten angepasst werden. Für solche Einzelaktionen müssen also häufig manuell mehrere Bearbeitungsschritte durchgeführt werden.

Für eine benutzerfreundlichere Modellierung ist daher eine spezielle Unterstützungsfunktion im Editor erforderlich. Es ist beispielsweise sinnvoll, eine Funktion anzubieten, die einem Mehrinstanzknoten einen neuen Parameter hinzufügt und dies auch gleich beim zugewiesenen Subprozess erledigt. Darüber hinaus ist sogar denkbar, dem Benutzer eine Darstellung ähnlich zur Modellierung als Zweig zu präsentieren, die variable Parallelität intern aber als Subprozess zu implementieren.

**Modellierung als Scope** Die Modellierung als Scope ist bezüglich Ihrer Vor- und Nachteile zwischen der Modellierung als Zweig und der als Subprozess angesiedelt. Ein Scope bzw. Gültigkeitsbereich in der hier verwendeten Bedeutung ist (analog zu WS-BPEL 2.0) ein bezüglich des Kontrollflusses abgeschlossener Block (d.h. es gibt eine einzige Stelle, an der der Kontrollfluss den Block betritt und eine einzige Stelle, an der er ihn wieder verlässt) mit eindeutig zugeordneten Knoten. Innerhalb eines Scopes können (wie der Name schon sagt) ähnlich wie in einem Subprozess eigene Datenelemente definiert werden, auf die von außerhalb des Scopes nicht zugegriffen werden kann. Allerdings ist es im Gegensatz zu einem Subprozess problemlos möglich, auf weiter außen definierte Datenelemente zuzugreifen.


Da ADEPT keine Scopes unterstützt und Scopes zudem, wie oben dargestellt, im Zusammenhang mit der variablen Parallelität in vielen Aspekten Subprozessen ähneln, wird diese Variante im Weiteren nicht mehr näher betrachtet.

**4.2.2. Instanzanzahl**

Eine weitere wichtige Frage bei der variablen Parallelität ist, *wie* und *wann* die Anzahl der zu erstellenden Instanzen bestimmt wird.
4.2. Umsetzungsvarianten

Abbildung 4.3: Varianten der Bestimmung der Instanzenzahl

Bei der Bestimmung der Anzahl („wie“) lassen sich folgende Möglichkeiten angeben:

1. **Explizit festgelegt**, z.B. durch einen ganzzahligen Ausgabeparameter der Verzweigungsaktivität (siehe Abbildung 4.3a)

2. Bestimmt aus der **Datenstruktur**, z.B. aus der Anzahl der Listenelemente (siehe Abbildung 4.3b)

3. Als Sonderfall: Aus der erforderlichen **Anzahl der Bearbeiter** (beispielsweise bei einem Entwicklungsprozess)

Für den **Zeitpunkt der Festlegung** der Anzahl („wann“) gibt es hingegen folgende Möglichkeiten:

1. **Statische Festlegung**, d.h. nach Beginn der Ausführung des variablen Parallelitätsblocks ist die Anzahl der Instanzen fest und es können keine weiteren hinzukommen (entspricht WCP-14 (Multiple instances with a priori run-time knowledge)).

2. **Dynamische Festlegung**, d.h. solange der variable Parallelitätsblock nicht beendet wurde, können jederzeit neue Instanzen hinzukommen (entspricht WCP-15 (Multiple instances without a priori run-time knowledge)).

**Anzahl explizit festgelegt** Die explizite Festlegung der Instanzenzahl ist die weitaus flexiblere und zunächst vielleicht naheliegendere Variante, da hier der Modellierer alle Freiheiten hat, die Instanzenzahl auf einen für ihn sinnvollen Wert festzulegen.

Auch eine Bestimmung der Instanzenzahl aus der Datenstruktur lässt sich hierauf abbilden, indem eine geeignete Aktivität definiert wird, die die Länge einer Liste ausliest (vgl. Abbildung 4.3a). Die Anzahl kann aber auch problemlos auf eine andere Art aus den Daten bestimmt werden, beispielsweise aus der Anzahl der erforderlichen Bearbeiter bei einem Review-Prozess wie in Abbildung 4.1.


Genau diese Flexibilität stellt aber auch das größte Manko dar, da es dadurch sehr schwer bis unmöglich ist, irgendwelche Korrektheitsaussagen bezüglich des Datenflusses zu treffen. In
4. Variable Parallelität

Abbildung 4.3a könnte die Splitaktivität beispielsweise zur Laufzeit 600 Instanzen von Hole Review einfordern, obwohl die Reviewerliste lediglich 6 Elemente aufweist und somit für die Instanzen Nr. 7 bis 600 keine Daten vorliegen. Zur Entwurfszeit kann ein solches Verhalten praktisch nicht erkannt und präventiv unterbunden werden.

**Anzahl aus Datenstruktur** Der größte Vorteil der Bestimmung der Instanzanzahl aus der Datenstruktur besteht darin, dass das System die volle Kontrolle über die Instanzanzahl hat, da der Modellierer lediglich die Eingabeparameter festlegt und das System daraus automatisch eine passende Instanzanzahl erzeugt.

Dadurch lassen sich viele potentielle Datenfehler vermeiden und man kann zur Entwurfszeit wesentlich besser Korrektheitsaussagen treffen. Insbesondere kann der Modellierer prinzipbedingt nicht den Fehler machen, eine unpassende Anzahl von Instanzen anzufordern. Eine triviale Möglichkeit, hier die Korrektheit zu garantieren ist es, wie bei manchen Workflowsystemen nur je eine Liste (bzw. einen Baum) als Eingabeparameter zu erlauben, da so die Instanzanzahl immer gleich der Elementanzahl der Liste gewählt werden kann.


**Statistische Festlegung** Bei der statischen Festlegung der Instanzanzahl wird, wie schon oben beschrieben, zu Beginn der Ausführung des variablen Parallelitätsblocks sowohl die gewünschte Instanzanzahl bestimmt als auch entsprechend viele Instanzen erstellt. Danach können keine neuen Instanzen mehr erzeugt werden – es sei denn, der Block befindet sich innerhalb einer Schleife, dies kann aber als erneute Ausführung betrachtet werden.

Dies stellt für ADEPT eigentlich keine Einschränkung dar, da sich die Eingabedaten nach dem Start einer Aktivität nicht mehr ändern können. Bei ADEPT ist es demnach unmöglich, dass nach Start des variablen Parallelitätsblocks auf neue Eingabedaten eintreffen, die neue Instanzen zu ihrer Bearbeitung benötigen.

1 Die variable Parallelität funktioniert hier wie ein paralleles for each-Konstrukt (vgl. moderne Programmiersprachen wie Java [GJSB05]).

2 z.B. WEP [Beu02]
4.2. Umsetzungsvarianten

Dynamische Festlegung  
Die offensichtlich flexiblere Variante ist die dynamische Festlegung der Instanzanzahl. Hier wird wie bei der statischen Festlegung zunächst eine bestimmte Anzahl von Instanzen gestartet, es können aber jederzeit neue Instanzen hinzugefügt werden, solange der variable Parallelitätsblock noch nicht beendet ist.

Die zentrale Schwierigkeit bei einer dynamischen Instanzanzahl ist, wie die Erstellung neuer Instanzen angestoßen werden kann und wie die neuen Instanzen mit Eingabedaten versorgt werden können. Eine naheliegende Möglichkeit ist es, beides durch den Benutzer erledigen zu lassen, der dann auch gleich die fehlenden Instanzdaten nachversorgt (wobei fraglich ist, ob dies immer als wirklich benutzerfreundlich angesehen kann). In Systemen, in denen zur Laufzeit neue Daten(qualitäten) aus früheren Schritten bereitgestellt werden können, wie z.B. WEP [Beu02], ist auch dieses Eintreffen neuer Daten als Auslöser zur Erstellung neuer Instanzen (die dann die neuen Daten bearbeiten) denkbar.

Insgesamt bleibt es aber (insbesondere bei ADEPT) unklar, wie die Erstellung der neuen Instanzen in der Praxis auf sinnvolle Weise ausgelöst werden kann.

Überrlagert sind bislang keine praxisrelevanten Szenarien bekannt, in denen eine dynamische Festlegung der Instanzanzahl wirklich benötigt wird. Da eine dynamische Festlegung der Instanzanzahl viele konzeptionelle Ähnlichkeiten zu einer instanzspezifischen Änderung aufweist, ist es aufgrund der beschriebenen Unklarheiten naheliegend, in ADEPT kein Konstrukt zur expliziten Unterstützung dieser Semantik vorzusehen, sondern eine dynamische Festlegung der Instanzanzahl bei Bedarf durch instanzspezifische Änderungen zu erledigen.

4.2.3. Datenübergabe

Ein weiterer wichtiger Aspekt der variablen Parallelität ist die Frage, wie eine Instanz an die ihr zugeordneten Daten kommt, das heißt, wie den einzelnen Instanzen die jeweils zu bearbeitenden Listenelemente zugewiesen werden.

Auch hier lassen sich im Wesentlichen zwei Möglichkeiten identifizieren:

1. **Zuteilung eines Indexes** an jede Instanz (siehe Abbildung 4.4a). Hier greift jede Instanz mit dessen Hilfe selbst indiziert auf die Liste(n) (bzw. andere Datenstrukturen) zu und besorgt sich so ihre Eingabedaten bzw. schreibt die Ausgabe.

2. **Zuteilung der konkreten Daten** an jede Instanz (siehe Abbildung 4.4b). In diesem Fall wird jeder Instanz das konkrete zu bearbeitende Eingabedatum zugewiesen bzw. es werden aus den Ausgabedaten automatisch entsprechende Ausgabelisten erzeugt. Die Instanzen greifen also nicht direkt auf die Listen zu, sondern es wird eine automatische Zuordnung der Listen auf die Instanzdaten und umgekehrt durchgeführt.

**Zuteilung eines Indexes**  
Diese Variante hat den großen Vorteil, dass sich das System nur um die Verteilung des Indexes kümmern muss und die eigentlichen Zugriffe auf die Listen als gewöhnliche indizierte Zugriffe ablaufen. Dies ist insbesondere bei einer Modellierung als Scope ein Vorteil, da man sich so die Analysen zur Bestimmung der Zugriffe auf listenwertige Daten (die bei einer konkreten Zuteilung der Daten modifiziert werden müssten) sparen kann.
Abbildung 4.4: Varianten der Datenübergabe am Beispiel eines Subprozesses
Gleichzeitig gibt es aber den Nachteil, dass hier keinerlei Sicherung dagegen besteht, dass eine Instanz (unabsichtlich) die Daten einer anderen Instanz manipuliert bzw. einen ungültigen Index verwendet, da dies bei einem gewöhnlichen indizierten Zugriff nur schwer überprüft werden kann.

An Abbildung 4.4a kann man die Funktionsweise dieser Variante nachvollziehen: Jede Instanz verwendet den zugeteilten Instanzindex, um auf den zugeordneten Artikel zugreifen. Bei der ersten Instanz ist dies beispielsweise das erste Element der Artikelliste, also „Haken“ ("Haken" = Artikel.get(0) = Artikel.get(Instanzindex)). Auf die selbe Weise werden der zweiten Instanz „Ösen“ und der dritten Instanz „Nägel“ zugeordnet.

Es ist aber ebenso problemlos möglich, in einer Instanz unbeabsichtigt mit einem zu großen Index auf die Liste zugreifen oder über einen Schreibzugriff die Artikeliste zu manipulieren (wodurch sich je nach Implementierung ggf. ungewollte Seiteneffekten auf die anderen Instanzen ergeben können).

Zuteilung der konkreten Daten  
Bei einer Zuteilung konkreter Daten kann eine Instanz nicht auf die Daten einer anderen Instanz zugreifen oder einen ungültigen Index zum Listenzugriff verwenden, was aus Korrektheitsgesichtspunkten vorteilhaft ist.

Außerdem muss sich der Modellierer hier nicht um den Zugriff auf Listen kümmern, sondern kann direkt auf die den einzelnen Elementen entsprechenden Daten zugreifen. Dies ist in vielen Fällen komfortabler und übersichtlicher, als wenn er jedesmal die korrekte Indexvariable mit angeben muss. Dies kann man gut an Abbildung 4.4 erkennen: Die Modellierung im Fall (b) ist deutlich übersichtlicher und potentiell weniger fehlerträchtig als die Modellierung im Fall (a)\(^3\). Jede Instanz bekommt ohne den Umweg über einen Zugriff auf die Artikeliste direkt den entsprechenden Artikel zugeteilt und kann die Artikeliste somit auch nicht manipulieren oder einen ungültigen Index verwenden.

Allerdings setzt diese Variante voraus, dass dem System die referenzierten listenwertigen Datenelemente vollständig bekannt sind, um die Verteilung zur Laufzeit korrekt vornehmen zu können.

Bei der Modellierung des vervielfältigten Teils als Zweig oder Scope ist dafür eine mehr oder weniger aufwendige Analyse erforderlich, was in diesen Fällen ein gewisser Nachteil dieser Variante ist.

Bei der Modellierung als Subprozess sind die referenzierten Daten durch die Ein- und Ausgabeparameter dagegen prinzipbedingt explizit gegeben, so dass keine Analyse der Prozessstruktur erforderlich ist. Dadurch bietet sich diese Variante natürlich besonders für diesen Fall an.

4.2.4. Bearbeiterzuordnung

Zur Zuordnung der Bearbeiter lassen sich folgende Alternativen angeben (es sind allerdings auch Mischformen denkbar):

\(^3\)Aus diesem Grund bieten viele moderne Programmiersprachen zusätzlich zu normalen while-Schleifen eine for each-Schleife an, obwohl sich jede for each-Schleife natürlich in eine äquivalente while-Schleife umformen lässt.
1. Über spezielle Konstrukte einer Sprache zur Bearbeiterzuordnung: Hier besitzt die Sprache, die zur Bearbeiterzuordnung verwendet wird, spezielle Konstrukte, die eine Fallunterscheidung im Rahmen der variablen Parallelität erlauben (siehe Abbildung 4.5a).

2. Behandlung der Bearbeiterzuordnung wie gewöhnliche Daten (ggf. mit eigenem Datentyp): In diesem Fall besitzen Aktivitäten einen speziellen Systemeingabeparameter, über den die Bearbeiterzuordnung gesetzt werden kann (siehe Abbildung 4.5b).

Abbildung 4.5: Varianten der Bearbeiterzuordnung

Spezielle Konstrukte einer Sprache zur Bearbeiterzuordnung Wenn bei einem Workflow-System (wie bei ADEPT) bereits eine Sprache zur Bearbeiterzuordnung vorhanden ist, liegt es auf den ersten Blick nahe, diese zur Unterstützung der variablen Parallelität zu erweitern.

Dies bringt aber auch einige Probleme mit sich: Zuerst einmal muss in der Sprache ein Mechanismus implementiert werden, der es erlaubt, eine Fallunterscheidung für die einzelnen Instanzen durchzuführen. Dazu wird in vielen Fällen allein der Index der Instanz als Kriterium nicht ausreichen, sondern es werden weitere Instanzen Daten mit einbezogen werden müssen. Das heißt, dass die Sprache um Konstrukte erweitert werden muss, mit denen beliebige Daten untersucht werden können. Zudem ist anzunehmen, dass eine solche Fallunterscheidung oft zu langen und damit unübersichtlichen Ausdrücken führt.

Da der Bearbeiterausdruck einem Knoten im variablen Parallelitätensblock zugeordnet ist, während die Daten in einer vorgelagerten Aktivität erzeugt werden, findet bei dieser Variante die Erzeugung der Daten und die Zuordnung der Bearbeiter immer an zwei verschiedenen Orten.

Allerdings ist hier im Prozesseditor auch eine Unterstützung des Modellierers durch einen entsprechenden Assistenten denkbar.
4.2. Umsetzungsvarianten

statt. Dies stellt eine potentielle Fehlerquelle dar, da bei einer Änderung der Aktivität, die die Daten für die einzelnen Instanzen erzeugt, immer auch der Bearbeiterausdruck angepasst werden muss, der aus den konkreten Daten den gewünschten Bearbeiter ableitet. Dies kann leicht vergessen werden und somit zu schwer auffindbaren Fehlern führen.

**Bearbeiterzuordnung als gewöhnliche Daten** In diesem Fall erfolgt die Zuordnung der Bearbeiter auf exakt dieselbe Weise wie die Zuordnung gewöhnlicher Instanzdaten. Der Nachteil dieser Variante ist natürlich, dass das System die Bearbeiterzuordnung aus Datenelementen – z.B. über Systemeingabeparameter – unterstützen muss.

Wird eine solche Zuordnung aber unterstützt, bietet sie den großen Vorteil, dass sowohl seitens des Metamodells als auch seitens der Implementierung praktisch keinerlei Zusatzaufwand für die Unterstützung einer variablen Bearbeiterzuordnung entsteht. Auch aus Benutzerobericht ist es vorteilhaft, dass die Zuordnung der Bearbeiter völlig analog zu der Zuordnung der übrigen Daten erfolgt, so dass keine Konstrukte zur Fallunterscheidung in der Sprache zur Bearbeiterzuordnung erlernt werden müssen.

Zudem kann die Erzeugung der Liste mit den Bearbeitern und der Liste mit den zu verarbeitenden Daten in der selben Aktivität stattfinden, was bei Änderungen die Wahrscheinlichkeit von Fehlern senkt. Es ist auch problemlos möglich, die Bearbeiterzuordnung in einem vor der variablen Parallelität liegenden Regelschritt aus den Daten zu erzeugen, wobei (über den Einsatz entsprechender benutzerdefinierter Aktivitäten) praktisch beliebig komplexe Regeln angewandt werden können.

4.2.5. Diskussion

Wie die vorangegangenen Abschnitte zeigen, gibt es zur Umsetzung der variablen Parallelität viele mögliche Varianten, die alle ihre spezifischen Vor- und Nachteile aufweisen. Eine kurze Übersicht darüber ist in Tabelle 4.1 nochmals in kompakter Form gegeben.

<table>
<thead>
<tr>
<th>Aspekt</th>
<th>Variante</th>
<th>Vor- und Nachteile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vervielfältigter Teil</td>
<td>Als Zweig</td>
<td>+ Für den Benutzer intuitiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Zugriff auf Daten des umgebenden Prozesses einfach</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Theoretisch über instanzspezifische Änderungen ohne Metamodellmodifikationen realisierbar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− Analysen der Graphstruktur zur Laufzeit nötig</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− Sonderbehandlung für Zugriffe auf Listen nötig</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− Temporäre Variablen nur wenig intuitiv zu realisieren</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− Ggf. Korrektheitsprobleme mit nicht-listenwertigen Schreibzugriffen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− Bei Realisierung über instanzspezifische Änderungen Beeinträchtigungen der Ausführungsgeschwindigkeit</td>
</tr>
<tr>
<td>Aspekt</td>
<td>Variante</td>
<td>Vor- und Nachteile</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
|                        | Als Subprozess    | + Keine Analysen oder Modifikationen der Graphstruktur zur Laufzeit  
|                        |                   | + Gute Performance  
|                        |                   | + Keine Probleme mit temporären Variablen  
|                        |                   | + Keine Spezialbehandlung im Subprozess oder dem umgebenden Prozess nötig  
|                        |                   | − Für den Benutzer (ohne Editorunterstützung) weniger komfortabel  
|                        | Als Scope         | + Zugriff auf Daten des umgebenden Prozesses einfach  
|                        |                   | + Keine Probleme mit temporären Variablen  
|                        |                   | ± Laufzeitanalysen nötig, aber sehr einfach  
|                        |                   | − Ähnliche Korrektheitsprobleme wie bei Modellierung als Zweig  
|                        | Instanzenzahl    | + Extrem flexibel  
| (Art der Bestimmung)   | Explizit festgelegt | + Variable Parallelität ohne Eingabedaten möglich  
|                        |                   | – Korrektheitszusicherungen für die Versorgtheit von Lesezugriffen praktisch unmöglich  
|                        | Aus Daten         | + Korrektheitszusicherungen relativ einfach  
|                        |                   | − Immer mindestens ein Eingabeparameter nötig  
|                        | Aus Anzahl        | * Kann immer auf explizite Festlegung oder Bestimmung aus den Daten zurückgeführt werden  
| Bearbeiter              | Statisch          | + Für ADEPT keine Einschränkung  
|                        |                  |  
|                        | Dynamisch         | + Flexibler als eine statische Festlegung  
|                        |                   | – Wie wird Erstellung neuer Instanzen angestoßen?  
|                        |                   | – Woher kommen die Daten für dynamisch erstellte Instanzen?  
|                        |                   | – Praktischer Nutzen fraglich  
|                        | Datenübergabe     | + Außer Zuteilung des Indexes keine systemseitige Spezialbehandlung nötig  
|                        | Zuteilung eines Index | – Korrektheitszusicherungen schwierig  

4. Variable Parallelität
4.2. Umsetzungsvarianten

<table>
<thead>
<tr>
<th>Aspekt</th>
<th>Variante</th>
<th>Vor- und Nachteile</th>
</tr>
</thead>
</table>
| Zuteilung konkreter Daten |                                  | + Korrektheitszusicherungen einfach  
|                       |                                  | + Modellierer muss sich nicht um Listenzugriffe kümmern  
|                       |                                  | ± System muss referenzierte Listen kennen  
|                       |                                  | − System benötigt eine spezielle Routine zum Verteilen/ Einsammeln der Daten  
| Bearbeiterzuordnung   | Über Sprachkonstrukte             | + Zunächst naheliegendere Variante  
|                       |                                  | − Sprache zur Bearbeiterzuordnung muss stark erweitert werden  
|                       |                                  | − Ausdrücke zur Bearbeiterzuordnung potentiell sehr komplex  
|                       |                                  | − Zuordnung der Bearbeiter und Erzeugung der Eingabedaten an verschiedenen Stellen (potentielle Fehlerquelle bei Änderungen)  
| Wie gewöhnliche Daten |                                  | + Kaum Zusatzaufwand gegenüber Verteilung gewöhnlicher Daten  
|                       |                                  | + Kein zusätzlicher Lernaufwand für Benutzer nötig  
|                       |                                  | − System muss das Lesen von Bearbeiterzuordnungen aus Datenelementen unterstützen  

Tabelle 4.1: Übersicht über die Umsetzungsvarianten

Wie man an der Tabelle leicht erkennen kann, lässt sich keine ideale Kombination angeben, die allen anderen möglichen Varianten in jedem denkbaren Fall überlegen ist, also keinerlei Nachteile aufweist. Dies spiegelt sich auch darin wider, dass sich bei anderen, variable Parallelität unterstützenden Workflowsystemen bei jedem System verschiedene Kombinationen der hier vorgestellten Umsetzungsvarianten finden.


Neben den anderen oben schon diskutierten Vorteilen der einzelnen Aspekte liegt der Grund für die Auswahl dieser Kombination vor allem darin, dass eine Bestimmung der Instanzanzahl aus den Eingabedaten bei einer konkreten Zuteilung der einzelnen Listenelemente Datenflussanalysen gegenüber den anderen Möglichkeiten wesentlich erleichtert. Zudem kann durch die Verwendung von Subprozessen die bestmögliche Ausführungsgeschwindigkeit erreicht werden und die bestehenden Algorithmen zur Datenflussanalyse müssen nicht angepasst werden (da die

---

5 Die Umsetzung in anderen System wird in Kapitel 8 untersucht.
Zuordnung der Daten an die einzelnen Instanzen vollständig innerhalb des Mehrinstanzknotens stattfinden kann).

4.3. Existierendes Metamodell


Abbildung 4.6: Symbol für einen Mehrinstanzknoten

Falls einem Mehrinstanzknoten keine Aktivität zugewiesen wurde, schaltet der Knoten zur Laufzeit einfach ohne jede Aktion weiter. Da diese Semantik intuitiv naheliegend und einfach umzusetzen ist, wird dieser Fall zur Vereinfachung der Diskussion im Folgenden nicht mehr besonders berücksichtigt.

Selbstverständlich kann ein Subprozess, der einem Mehrinstanzknoten zugewiesen wurde, selbst wieder Mehrinstanzknoten enthalten. Es ist also eine beliebige Schachtelung möglich. Da ein Mehrinstanzknoten von außen aber wie ein gewöhnlicher Knoten behandelt werden kann, muss auch dieser Fall im Folgenden nicht weiter separat diskutiert werden.

In den folgenden Abschnitten werden die Abweichungen im Kontroll- und Datenflussverhalten gegenüber gewöhnlichen Knoten behandelt und im Abschnitt „Datenfluss“ (4.3.2) die Möglichkeiten zur Garantie der Datenflusskorrektheit diskutiert. Die Datenflusskorrektheit wird (wie wir noch sehen werden) maßgeblich durch die gewählte Instanzanzahl beeinflusst. Für eine Diskussion der Bestimmung einer geeigneten Instanzanzahl ist es aber vorteilhaft, wenn die Funktionsweise des Mehrinstanzknotens vorher geklärt wurde. Um die Diskussion zu vereinfachen, wird die Bestimmung der Instanzanzahl daher zunächst ausgeklammert und als gegeben betrachtet.

4.3.1. Kontrollfluss

Aus Sicht des umgebenden Prozesses verhält sich ein Mehrinstanzknoten identisch zu einem gewöhnlichen Knoten mit Ein- und Ausgangssemantik \texttt{ONE\_OF\_ONE}, dem ein Subprozess zugewiesen wurde. Genau wie dieser wechselt ein Mehrinstanzknoten bei seiner Aktivierung sofort vom Zustand \texttt{ACTIVATED} in den Zustand \texttt{STARTED}, um die nötigen Initialisierungsarbeiten
4.3. Existierendes Metamodell

durchzuführen und schließlich die Instanzen der zugewiesenen Aktivität zu erstellen und zu aktivieren.

Dazu wird zunächst die nötige Instanzenzahl bestimmt, die zugewiesene Aktivität wird entsprechend mehrfach instanziert und die einzelnen Instanzen werden mit Eingabedaten versorgt. Danach werden alle Instanzen aktiviert (Zustand \textit{ACTIVATED}), damit also gleichzeitig in den Arbeitslisten angeboten und dementsprechend später auch (potentiell) parallel ausgeführt.

Sobald die Ausführung aller Instanzen abgeschlossen ist (sich also alle Instanzen im Zustand \textit{COMPLETED} befinden), werden die Ausgabelisten des Mehrinstanzknotens aus den Ausgabedaten der abgeschlossenen Instanzen erzeugt und der Mehrinstanzknoten wechselt in den Zustand \textit{COMPLETED}.

Wenn die Ausführung des Mehrinstanzknotens suspendiert wird (Übergang \textit{STARTED} \rightarrow \textit{SUSPENDED}), wird dies nahezu gleichzeitig auf alle Instanzen propagiert. Dementsprechend wird bei der Wiederaufnahme der Ausführung eines suspendierten Mehrinstanzknoten (Übergang \textit{SUSPENDED} \rightarrow \textit{STARTED}) auch die Ausführung aller Instanzen wieder fortgesetzt.

Sollte die Ausführung \textit{irgendeiner} der Aktivitätsinstanzen scheitern (Zustand \textit{FAILED}), wird die Ausführung der anderen Instanzen abgebrochen und der Mehrinstanzknoten selbst wechselt in den Zustand \textit{FAILED}. Dabei wird der Error Code der gescheiterten Instanz nach außen weitergereicht. Analog dazu werden alle Instanzen abgebrochen, wenn der Mehrinstanzknoten selbst von außen abgebrochen wird.

Aus dieser Semantik folgt insbesondere, dass nicht jede Statusänderung einer Instanz zu einer Änderung des nach außen sichtbaren Status des Mehrinstanzknotens führt, sondern dass dabei immer auch der Status der anderen Instanzen betrachtet werden muss. Das heißt also, dass zur Laufzeit vom System der Status einer jeden Instanz separat zum globalen Status des Knotens verwaltet werden muss.

\textbf{Abbildung 4.7:} Ausführungszustände eines Mehrinstanzknoten

Die Ausführungslogik eines Mehrinstanzknotens ist in Abbildung 4.7 nochmals in Form eines (an UML angelehnten) Zustandsdiagramms zu sehen. Zusätzlich ist diese Logik in Listing B.1 in Anhang B.1 als Pseudocode dargestellt.
4.3.2. Datenfluss

4.3.2.1. Anforderungen an Listen

Als Ein- und Ausgabeparameter werden, wie oben schon beschrieben, bei variabler Parallelität listenwerte Daten verwendet. Die Definition eines Listenkonzepts war zum Zeitpunkt der Erstellung dieser Arbeit allerdings noch Gegenstand der Forschung, so dass noch kein endgültiges Konzept zur Verfügung stand. Daher werden in diesem Abschnitt die in dieser Arbeit als gegeben angenommenen Eigenschaften und Operationen der Listen definiert, um die nachfolgende Diskussion nicht unnötig kompliziert zu machen.

Die angenommenen Eigenschaften der Listen sind:

- Die Länge der Listen ist dynamisch, d.h. kann sich zur Laufzeit ändern.
- Auf die Listen kann indiziert zugegriffen werden und die Indizes sind nullbasiert (d.h. das erste Element besitzt den Index 0).
- Die Listen haben immer einen zur Entwurfszeit bekannten inneren Datentyp, das heißt eine Liste enthält immer nur Objekte eines bestimmten Datentyps. Eine Liste, die Objekte vom Typ `TYPE` enthält wird dementsprechend als `List of TYPE` bezeichnet.
- Es ist möglich, über gewöhnliche Datenlesekanten bzw. Datenschreibkanten auf die Liste als Ganzes zuzugreifen\(^6\).

Außerdem wird davon ausgegangen, dass auf den Listen die Operationen aus Tabelle 4.2 mit der dort angegebenen Semantik möglich sind.

<table>
<thead>
<tr>
<th>Annahmen:</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ <code>int</code> beschreibt einen genügend großen Ganzzahltyp</td>
</tr>
<tr>
<td>◦ <code>TYPE</code> ist der innere Datentyp der Liste</td>
</tr>
<tr>
<td>◦ Indizes haben Werte $\geq 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semantik</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TYPE list.get(int index)</code></td>
<td>Gibt das Element mit dem entsprechenden Index zurück. Löst einen Laufzeitfehler aus, wenn der übergebene Index zu groß ist.</td>
</tr>
<tr>
<td><code>TYPE List.getOptional(int index)</code></td>
<td>Gibt das Element mit dem entsprechenden Index zurück. Gibt <code>NULL</code> zurück, wenn der übergebene Index zu groß ist.</td>
</tr>
<tr>
<td><code>List.set(int index, TYPE element)</code></td>
<td>Setzt das Listenelement an der entsprechenden Indexposition auf <code>element</code>.</td>
</tr>
<tr>
<td><code>List.add(TYPE element)</code></td>
<td>Fügt das angegebene Element am Ende der Liste an.</td>
</tr>
<tr>
<td><code>List.clear()</code></td>
<td>Löscht den Inhalt der Liste (setzt die Länge auf 0).</td>
</tr>
<tr>
<td><code>int list.getLength()</code></td>
<td>Gibt die Anzahl der Elemente in der Liste zurück.</td>
</tr>
</tbody>
</table>

\(^6\)Das schließt selbstverständlich die Existenz anderer, hier aber nicht benötigter Kanten zum Lesen bzw. Schreiben einzelner Elemente nicht aus!
4.3. Existierendes Metamodell

4.3.2.2. Parametermapping

Bei der hier gewählten Variante der variablen Parallelität wird jede Instanz der zugewiesenen Aktivität zur Laufzeit durch den Mehrinstanzknoten mit ihren Eingabeparametern versorgt bzw. es werden die Ausgabeparameter zusammengetragen und entsprechende Ausgabelisten daraus erzeugt (wie bei der Diskussion der Umsetzungsvarianten schon beschrieben wurde).

Dies bedeutet, dass bei Zuweisung einer Aktivität an einen Mehrinstanzknoten die Datentypen der Parameter der zugewiesenen Aktivität (nachfolgend auch als „innerer Parameter“ bezeichnet) nicht eins zu eins nach außen weitergegeben werden können, sondern, dass ein Mapping des inneren Datentyps auf den entsprechenden, listenwertigen äußeren Datentyp erfolgen muss (Beispiel: Innerer Typ INTEGER → Äußerer Typ list of INTEGER).


Darüber hinaus kann es sinnvoll sein, wenn eine Instanz weiß, welchen Index in den Eingabelisten sie bearbeitet (beispielsweise für Ausgaben an den Benutzer). Daher wird auch die Möglichkeit angeboten, einen Eingabeparameter der inneren Aktivität vom Typ INTEGER als Indexparameter zu markieren, der zur Laufzeit als Wert den Index der Instanz erhält (0 für die erste Instanz, 1 für die zweite, usw.; praktisch als „Zählvariable“). Ein solcher Indexparameter erscheint nach außen hin nicht als Eingabeparameter des Mehrinstanzknotens (da die Daten intern erzeugt werden).

Durch eine Kombination dieser beiden Erweiterungen ist es darüber hinaus möglich, eine Zuordnung der Instanzdaten über einen Index zu realisieren, falls dies in einem konkreten Anwendungsfall angebracht ist (durch Übergabe einer Liste ohne Mapping und Zugriff darauf unter Einsatz des Indexparameters).

Insgesamt bietet das soeben beschriebene Mapping den Vorteil, dass für den Mehrinstanzknoten weder zur Entwurfs- noch zur Laufzeit bezüglich des Datenflusses irgendwelche Ausnahmen gegenüber anderen Knoten gemacht werden müssen, also insbesondere die Datenflussanalyse nicht angepasst werden muss:

- Nach innen kann die zugewiesene Aktivität beliebige Ein- und Ausgabeparameter aufweisen (und muss sich darüber hinaus nicht um Zugriffe auf Listen kümmern).
- Nach außen besitzt der Mehrinstanzknoten gewöhnliche Parameter, die über die üblichen Mechanismen versorgt werden bzw. deren Versorgtheit über die bereits vorhandenen Algorithmen sichergestellt werden kann.

{Ein anderes Verhalten wäre auch inkonsistent.}

Wie man in Abbildung 4.8 erkennt, werden die Datentypen der inneren Parameter beim Mappingtyp LIST nach außen auf den entsprechenden listenwertigen Typ erweitert, während der Name des Parameters erhalten bleibt. Parameter mit dem Mappingtyp NONE werden einfach eins zu eins nach außen weitergereicht, während Indexparameter nach außen nicht erscheinen.

In Abbildung 4.9 erkennt man, dass beim Mappingtyp LIST jeder Instanz ein Element der Liste zugeordnet wird. Beispielsweise wird der ersten Instanz der erste Artikel (Haken) zugeordnet, der zweiten der zweite (Ösen), usw. Beim Mappingtyp NONE erhält jede Instanz einfach eine Kopie der Daten (beispielsweise die Rechnungsnummer). Der Indexparameter schließlich versorgt jede Instanz mit einer laufenden Nummer (0 für die erste Instanz, 1 für die zweite, usw.; beachte, dass ArtikelListe.get(index) genau den der entsprechenden Instanz zugeordneten Artikel liefern würde).

Die Algorithmen zum Durchführen des Mappings zur Entwurfszeit bzw. zum Handling der Daten zur Laufzeit sind im Anhang B.1 in den Listings B.2, B.3 und B.4 als Pseudo code beschrieben.

<table>
<thead>
<tr>
<th>Mappingtyp</th>
<th>Nur Eingabeparameter?</th>
<th>Datentyp nach außen (innerer Typ InnerType)</th>
<th>Wert zur Laufzeit (i-te Instanz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listenelement (LIST)</td>
<td>Nein</td>
<td>list of InnerType</td>
<td>outerValue.get(i)</td>
</tr>
<tr>
<td>Kein (NONE)</td>
<td>Ja</td>
<td>InnerType</td>
<td>outerValue</td>
</tr>
<tr>
<td>Indexparameter (INDEX)</td>
<td>Ja</td>
<td>— (nach außen unsichtbar)</td>
<td>1</td>
</tr>
</tbody>
</table>

Tabelle 4.3: Typen des Parametermappings

Abbildung 4.8: Beispiel für das Parametermapping zur Entwurfszeit
4.3. Existierendes Metamodell

(a) Daten aus Sicht des umgebenden Prozesses (das jeweils 1. Element der Listen wird unten Instanz 1 zugeordnet, das 2. Instanz 2, usw.)

(b) Daten aus Sicht der einzelnen Instanzen

Abbildung 4.9: Beispiel für das Parametermapping zur Laufzeit
4.3.2.3. Zugriff auf einzelne Listenelemente

Bevor in Abschnitt 4.3.2.4 diskutiert wird, wie sich die Instanzanzahl geeignet bestimmen lässt, werden in diesem Abschnitt die Probleme bei einem indizierten Zugriff auf einzelne Listenelemente beschrieben. Der Grund hierfür liegt darin, dass sich die grundlegenden Probleme in beiden Fällen kaum unterscheiden und sich durch die Betrachtung des indizierten Zugriffs die Vor- und Nachteile des hier gewählten Ansatzes, die konkreten Daten zu verteilen, deutlich besser beurteilen lassen.

Die grundlegende Schwierigkeit beim Zugriff auf Listen besteht darin, dass die Länge der Listen im Allgemeinen dynamisch, also zur Entwurfszeit unbekannt ist. Bei einem indizierten Zugriff mit variablen Index kommt verschärfend hinzu, dass auch der Wert der Indexvariablen erst zur Laufzeit bekannt ist.

Zum korrekten obligaten Lesen eines Listenelements muss der Index immer positiv und kleiner als die Länge der Liste sein. Sowohl die Listenlänge als auch der Wert eines variablen Indexes sind aber zur Entwurfszeit unbekannt und es sind im Allgemeinen auch keine Informationen über ihre relative Größe bekannt, was eine statische Korrektheitsanalyse hier sehr problematisch bis unmöglich macht. Auch Zusicherungen über den Wert von Datenelementen und die (relative) Länge von Listen sind im allgemeinen nicht bekannt und schaffen im Fall ihrer Implementierung sehr wahrscheinlich wieder neue Probleme.

Das Problem beim indizierten Zugriff auf eine Liste ist in Abbildung 4.10 nochmals illustriert: Dort soll auf das sechste Element zugegriffen werden, die Liste enthält aber lediglich vier Elemente. Auch wenn im Beispiel ein statischer (also zu Entwurfszeit bekannter) Index zum Zugriff auf die Liste verwendet wird, ist der Inhalt der Liste zur Entwurfszeit unbekannt. Das heißt, dass sich ein Auftreten des abgebildeten Falls kaum verhindern lässt und sich demnach die Frage stellt, wie man zur Laufzeit mit einer solchen Situation umgehen kann.

Für das indizierte obligate Lesen lassen sich folgende drei grundlegende Strategien identifizieren:

1. Obligates Lesen einzelner Listenelemente *immer zulassen*, sofern die Liste vorher geschrieben wurde. Damit riskiert man allerdings Laufzeitfehler, wenn mit einem zu großen Index

---

8Im Fall von Listen statischer Länge, auf die nur mittels statischer Indizes zugegriffen wird, wären natürlich ohne Weiteres Analysen möglich (z.B. Behandlung jedes Listenelements als eigenes Datenelement). Dieser Fall ist aber im Kontext der variablen Parallelität eher uninteressant.

9Es ist beispielsweise unklar, wie man eine solche Zusicherung implementieren kann, wenn Aktivität A die Liste erzeugt und Aktivität B den Index.
zugegriffen wird.
Diese Lösung hat den Vorteil, dass sie dem Benutzer am wenigsten Einschränkungen auferlegt und der Vorgehensweise bei den meisten gängigen Programmiersprachen entspricht. Allerdings wird hierdurch die erwartete Semantik des obligaten Lesens verletzt, da ein zu großer Index zur Laufzeit jederzeit auftreten kann und dann zu Laufzeitfehlern führt. Diese Lösung kommt für ADEPT also nicht in Frage.

2. Obligates Lesen einzelner Listenelemente *nie erlauben*. Dies vermeidet mögliche Fehler, bedeutet aber eine massive Einschränkung, da nur noch optionale Lesezugriffe auf einzelne Listenelemente möglich sind.

3. Angeben eines *Defaultwerts*, der für nicht vorhandene Elemente zurückgegeben wird. Diese Lösung vermeidet die massiven Einschränkungen der zweiten Variante, es gibt aber in der Praxis nicht in jedem Fall einen passenden bzw. sinnvollen Defaultwert, so dass diese Variante oft nicht anwendbar ist.

Es lässt sich also keine Strategie angeben, mit der sich das Problem der dynamischen Länge von Listen in jedem Fall kompensieren lässt. Für die Praxis ist in den meisten Fällen daher eine Kombination der zweiten und dritten Strategie am brauchbarsten: Es werden keine obligaten indizierten Lesezugriffe erlaubt, es sei denn, der Modellierer hat einen Defaultwert vergeben, der im Fall eines ungültigen Index verwendet wird.

Im Falle *optionalen Lesens* oder bei Schreibzugriffen hingegen kann die Korrektheit (zumindest theoretisch) durch eine geschickte Definitionen der Zugriffsoperationen garantiert werden (solange die Indizes ≥ 0 sind, was durch einen geeigneten Datentyp sichergestellt werden kann). Bei *optionalem Lesen* gelingt dies einfach dadurch, dass im Fall eines zu großen Index ein NULL-Wert zurückgegeben wird (vgl. obige Definition von `getOptional()` in Tabelle 4.2).
Beim *Schreiben* kann der Erfolg sichergestellt werden, indem die Liste bei einem zu großen Index einfach auf die passende Größe erweitert und mit einem Defaultwert aufgefüllt wird (hier sind in der Praxis für sehr große Indizes aus Speicherplatzgründen dennoch Einschränkungen sinnvoll).

Da parallele Schreibzugriffe aber im Gegensatz zum Lesen nicht vorhandener Werte nicht zwangsläufig zu Fehlern führen, ist es vermutlich dennoch vertretbar, parallele indizierte Schreibzugriffe auf die selbe Liste zuzulassen (mit der oben angesprochenen Einschränkung bezüglich der verwendeten Indexdatenelemente).

Dies muss aber bei der genauen Definition des Listendatentyps (der in dieser Arbeit nicht vorgenommen werden soll) genauso wie das Problem des obligaten Lesens noch einmal genauer untersucht werden.
4.3.2.4. Instanzanzahl bei mehreren Listen

Auch bei der Zuordnung konkreter Daten im Rahmen der variablen Parallelität bleibt das grundlegende Problem der zur Entwurfszeit unbekannten Länge von Listen bestehen. Da hier jedem Eingabeparameter jeder Instanz der zugewiesenen Aktivität jeweils ein Element der korrespondierenden Eingabeliste zugewiesen werden muss, kommt der Wahl der Instanzanzahl die zentrale Bedeutung bei der Datenflusskorrektheit zu: Ist die Instanzanzahl zu groß, gibt es nicht für jede Instanz ein korrespondierendes Listenelement, einige Eingabeparameter sind dort also unversorgt. Ist die Instanzanzahl zu klein, werden einige Listenelemente nicht bearbeitet bzw. ignoriert, was in vielen Fällen ebenso unerwünscht ist.

\[
\begin{align*}
\text{Liste1} &= \{A, B, C, D, E\} \\
\text{Liste2} &= \{1, 2, 3\}
\end{align*}
\]

Abbildung 4.11: Variable Parallelität mit mehreren Eingabelisten

Solange alle Eingabelisten gleich lang sind, gibt es kein Problem: Hier wählt man einfach die Instanzanzahl gleich der gemeinsamen Länge aller Listen.

Sofern mehr als eine Liste als Eingabeparameter vorhanden ist, kann aufgrund der dynamischen Länge der Listen im Allgemeinen aber nicht garantiert werden, dass alle Listen zur Laufzeit gleich lang sind. Daher stellt sich das Problem, wie die Instanzanzahl zu wählen ist, wenn festgestellt wird, dass die Eingabelisten verschieden viele Elemente enthalten. Dieses Problem ist in Abbildung 4.11 dargestellt. Unter der Annahme, dass sowohl \text{Liste1} als auch \text{Liste2} obligat gelesen werden, ist zunächst unklar, wie viele Instanzen im dargestellten Fall zu erstellen sind.

Eine schon genannte triviale Möglichkeit, dieses Problem zu vermeiden, ist es daher, nur eine einzige Liste als Eingabeparameter zuzulassen, so dass in jedem Fall der obige Spezialfall auftritt (bei genau einer Liste haben trivialerweise immer alle Listen die selbe Länge). Solange der Subprozess nur höchstens einen Eingabeparameter aufweist, ist dies völlig unproblematisch.

In der Regel weisen Subprozesse allerdings mehr als einen Eingabeparameter auf. Daher müssen in diesem Fall die Daten aller Parameter in einem einzigen Listenelement enthalten sein. Um dies zu erreichen, muss die Liste strukturierte Daten („records“) enthalten. Diese können dann entweder an einen einzigen, ebenso strukturierten Eingabeparameter des Subprozesses weitergegeben werden oder aber auf mehrere primitivwertige Parameter des Subprozesses abgebildet werden.

Dieser Ansatz verschiebt das Problem in vielen Fällen allerdings nur: Wenn die Werte der einzelnen Felder des strukturierten Datentyps von verschiedenen Aktivitäten erzeugt werden, muss die Liste mit den strukturierten Daten zuerst aus den einzelnen Listen zusammengesetzt werden, bevor sie am Mehrinstanzknoten verwendet werden kann. Bei diesem Zusammensetzen stellt sich bei verschiedenen langen Eingabelisten aber genau das selbe Problem wie hier, nur dass die Instanzanzahl indirekt über die Länge der resultierenden Liste gewählt werden muss.
4.3. Existierendes Metamodell

Wenn Bearbeiterzuordnungen an die einzelnen Instanzen weitergegeben werden sollen, müssen diese hier zusammen mit den übrigen Daten in einem gemeinsamen strukturierten Datensatz gespeichert werden. Auch dies kann – vor allem aus Sicherheitsgründen – in der Praxis durchaus problematisch sein.

Ein weiteres Manko ist, dass der Zwang zum Zusammensetzen einer solchen Liste im häufig vorkommenden Fall mehrerer Eingabeparameter den Komfort für den Modellierer vermindert und immer etwas den Anschein einer Behelfslösung hat.

Aus diesen Gründen ist angebracht, diese Lösung zunächst zu verwerfen und mögliche Strategien im Falle mehrerer verschieden langer Listen zu diskutieren10.


Sollten nur optionale Eingabeparameter vorhanden sein, besteht desweiteren die Möglichkeit, umgekehrt die Länge der längsten Liste als Instanzanzahl zu übernehmen. Die anderen (optional zu lesenden) Listen werden in diesem Fall logisch gesehen mit NULL-Werten aufgefüllt (das heißt, es wird, wie bei getOptional() definiert, für zu große Indizes einfach NULL gelesen).


Für obligates Lesen ist diese Variante aber nicht direkt anwendbar, hier besteht höchstens die Möglichkeit, für alle obligat zu lesenden Listen jeweils einen Defaultwert vorzuschreiben, mit dem die Liste dann jeweils aufgefüllt wird. In der Praxis stellt sich in diesem Fall aber wieder das Problem, dass es unter Umständen sehr schwer sein kann, einen sinnvollen Defaultwert anzugeben.

Da zu kurze Listen bei optionalem Lesen kein Problem darstellen, ist eine weitere naheliegende Möglichkeit, die Länge der kürzesten, obligat zu lesenden Liste als Instanzanzahl zu verwenden (siehe Abbildung 4.12c). Auch hier bleibt allerdings das Problem bestehen, dass unerwartet viele Elemente ignoriert werden können und zur Entwurfszeit unklar ist, welche Listen davon betroffen sind.

Der gravierendste Nachteil der bisher diskutierten Varianten war, dass zur Entwurfszeit unklar ist, aus welchen Listen Elemente zur Laufzeit ignoriert werden können. Eine Möglichkeit, dem Modellierer eine bessere Kontrolle darüber zu geben, besteht darin, dass zur Entwurfszeit eine Liste als „führende Liste“ markiert wird. Die Instanzanzahl wird in diesem Fall immer gleich der Anzahl der Elemente in dieser Liste gewählt. Es ist allerdings weiterhin nicht sichergestellt, dass die führende Liste nicht länger ist als die übrigen Listen. Daher Ansatz ist daher nur dann praktikabel, wenn die führende Liste entweder gleichzeitig die einzige obligat zu lesende Liste ist, oder, wenn für die übrigen obligat zu lesenden Listen ein Defaultwert vergeben wurde. Da man nicht immer einen sinnvollen Defaultwert angeben kann, ist die praktikablere Variante daher, nur eine obligat zu lesende Liste zu erlauben, die gleichzeitig die Anzahl von Elementen in dieser Liste darstellt.

Tabelle 4.4 enthält nochmals eine tabellarische Übersicht über die Möglichkeiten zur Bestimmung der Instanzanzahl und die individuellen Vor- und Nachteile. Da mehrere obligate Eingabeparameter in der Praxis relativ häufig sind, ist die Variante, die Instanzanzahl aus der

---

10 Zudem unterstützt ADEPT zum Zeitpunkt der Erstellung dieser Arbeit noch keine strukturierten Daten
4. Variable Parallelität

**Abbildung 4.12: Varianten der Bestimmung der Instanzanzahl**
### 4.3. Existierendes Metamodell

<table>
<thead>
<tr>
<th>Variante</th>
<th>Vor- und Nachteile</th>
</tr>
</thead>
</table>
| Nur eine einzige Liste erlaubt | + Kein Ignorieren von Daten  
− System muss strukturierte Daten unterstützen  
− Verschiebt das Problem in vielen Fällen nur |
| Aus kürzester Liste | + Auch mehrere obligat zu lesender Listen anwendbar  
− Möglicherweise werden viele Listenelemente ignoriert  
− Zur Entwurfszeit ist unklar, aus welchen Listen potentiell Daten ignoriert werden |
| Aus längster Liste | + Kein Ignorieren von Daten  
− Für obligat zu lesende Listen muss ein Defaultwert angegeben werden  
− In der Praxis existiert nicht immer ein sinnvoller Defaultwert |
| Aus kürzester, obligat zu lesender Liste | + Auch mehrere obligat zu lesender Listen anwendbar  
+ Die Anzahl der ignorierten Listenelemente wird minimiert  
− Immer noch potentiell viele ignorierte Listenelemente  
− Zur Entwurfszeit ist unklar, aus welchen Listen potentiell Daten ignoriert werden |
| Führende Liste mit Defaultwert für obligat zu lesende Listen | + Auch mehreren obligat zu lesender Listen anwendbar  
+ Zur Entwurfszeit ist klar, aus welchen Listen potentiell Daten ignoriert werden  
− In der Praxis existiert nicht immer ein sinnvoller Defaultwert |
| Nur eine obligat zu lesende, führende Liste erlaubt | + Zur Entwurfszeit ist klar, aus welchen Listen potentiell Daten ignoriert werden  
− Nur ein obligat zu lesender Eingabeparameter möglich |

Tabelle 4.4: Möglichkeiten zur Bestimmung der Instanzanzahl
4. Variable Parallelität

kürzesten, obligat zu lesenden Liste zu bestimmen, die praxistauglichste. Zudem besteht dadurch für den Modellierer jederzeit die Möglichkeit, das Ignorieren von Elementen genau zu steuern, indem er sich freiwillig auf eine einzige obligate Eingabeliste beschränkt.

Listing B.5 in Anhang B.1 implementiert dementsprechend die Bestimmung der Instanzanzahl aus der kürzesten, obligat zu lesenden Liste. Sofern alle Listen nur optional zu lesen sind, wird dort stattdessen die Länge der längsten optional zu lesenden Liste verwendet, um ein Ignorieren von Daten zu vermeiden.

Obwohl aufgrund der erst zur Laufzeit bekannten Länge von Listen gewisse Nachteile bestehen bleiben, ist es bei einer Zuordnung der konkreten Daten dennoch deutlich einfacher als bei einem direkten indizierten Zugriff, die Datenflusskorrektheit zu garantieren. Bei einer Weitergabe der konkreten Daten kann der eigentliche indizierte Zugriff vollständig innerhalb des Codes des Laufzeitsystems stattfinden. Wenn die einzelnen Aktivitäten direkt auf die Listen zugreifen, stellt sich das Problem der geeigneten Wahl der Instanzanzahl genauso, es entsteht aber zusätzlich die Schwierigkeit, dass zu große Indizes nicht zuverlässig verhindert werden können, da die zugewiesene Aktivität die volle Freiheit bei der Wahl des verwendeten Indexwertes hat (vgl. auch die Diskussion im vorigen Abschnitt).

In diesem Zusammenhang ist ein weiterer Aspekt interessant: Die variable Parallelität kann, analog zu WS-BPEL 2.0, auch als ein paralleles for each-Konstrukt angesehen werden. Aufgrund der beschriebenen Vorteile bei der Datenflusskorrektheit ist es daher erwägenswert, die variable Parallelität dahingehend zu erweitern, dass die Prozessinstanzen nicht nur parallel, sondern auch sequentiell ausgeführt werden können (um eine for each-Semantik wie in gängigen Programmiersprachen zu erreichen). Dies vereinfacht die Listenverarbeitung gegenüber normalen Schleifen deutlich und ist mit minimalen, naheliegenden Änderungen der vorgestellten Algorithmen realisierbar\(^\text{11}\). Auf der anderen Seite erhöht dies die Ausdrucksstärke des Metamodells in keiner Weise, so dass die genaue Implementierung einer solchen Erweiterung hier nicht weiter verfolgt wurde.

4.4. Zusammenfassung

Bei variabler Parallelität gibt es viele Freiheitsgrade bezüglich der genauen Umsetzung einiger Aspekte und demnach eine große Anzahl von potentiellen Varianten, wie dieses Konzept umgesetzt werden kann. Dies betrifft die Art der Modellierung des zur Laufzeit vervielfältigten Prozessesteils (als Zweig oder Subprozess), die Bestimmung der Instanzanzahl (explizit durch eine Aktivität oder implizit aus den Daten abgeleitet; statisch oder dynamisch), die Art der Datenübergabe an die einzelnen Instanzen (Übergabe lediglich eines Indexes oder Übergabe der konkret zu bearbeitenden Daten) und nicht zuletzt die Art der Bearbeiterzuordnung (Konstrukte einer Sprache zur Bearbeiterzuordnung oder Behandlung der Bearbeiterzuordnung wie gewöhnliche Daten).

Durch Betrachtung der individuellen Vor- und Nachteile dieser Varianten lässt sich eine im Kontext von ADEPT optimale Lösung angeben: Die Modellierung erfolgt als Subprozess, die Instanzanzahl wird statisch (d.h. zu Beginn der Ausführung der variablen Parallelität) aus den

\(^\text{11}\)Die Instanzen müssen im Wesentlichen sequentiell erstellt und ausgeführt werden und nicht mehr parallel.
Eingabedaten bestimmt, die einzelnen Instanzen erhalten die konkreten Daten zugeordnet und die Bearbeiterzuordnung erfolgt analog zu gewöhnlichen Daten.

Die Modellierung als Subprozess bietet eine optimale Ausführungsgeschwindigkeit und ist mit einer entsprechenden Unterstützung durch den Editor ähnlich komfortabel wie eine Modellierung als Zweig. Die Bestimmung der Instanzenzahl aus den Eingabedaten und die Übergabe der konkreten Daten an die einzelnen Instanzen weisen einige Vorteile in Bezug auf die Datenflusskorrektheit auf und verbessern den Komfort für den Modellierer. Die Bearbeiterzuordnung analog zu gewöhnlichen Daten hat schließlich den Vorteil, dass weder die Sprache zur Bearbeiterzuordnung noch das Workflowsystem selbst erweitert werden müssen, so dass der Implementierungsaufwand gering gehalten werden kann.

Um diese Semantik konkret umzusetzen, wurde ein sogenannter Mehrinstanzknoten beschrieben, dem eine gewöhnliche Aktivität (d.h. eine elementare Aktivität oder ein Subprozess) zugewiesen werden kann. Diese Aktivität wird dann zur Laufzeit mehrfach instanziert und ausgeführt, wobei sich der Mehrinstanzknoten um die Verteilung der Eingabedaten und das Einsammeln der Ausgabe kümmert.

Dazu wird nach außen zum umgebenden Prozess hin zur Entwurfszeit eine Abbildung der Datentypen der Parameter der zugewiesenen Aktivität auf entsprechende listenwertige Typen durchgeführt (aus \texttt{TYPE} wird \texttt{list of TYPE}). Nach innen werden dann zur Laufzeit die Listenelemente automatisch auf die einzelnen Instanzen verteilt. Bei Eingabeparametern besteht darüber hinaus die Möglichkeit, diese Abbildung für bestimmte Parameter zu deaktivieren (wodurch sämtliche Instanzen die selben Eingabedaten erhalten) oder den Index der jeweiligen Instanz zu übergeben. Dadurch wird die Flexibilität deutlich erhöht, da damit bei Bedarf auch eine Übergabe der kompletten Listen und eines direkten indizierten Zugriffs darauf möglich ist.

Die größte Herausforderung besteht allerdings in der korrekten Bestimmung der Instanzenzahl: Sofern mehrere Listen als Eingabeparameter gegeben sind, ist zur Laufzeit nicht sichergestellt, dass diese auch in jedem Fall die gleiche Länge aufweisen. Dem kann begegnet werden, indem nur eine einzige Liste als Eingabeparameter erlaubt wird, diese Lösung verschiebt aber in vielen Fällen das Problem nur auf frühere Schritte.

Falls mehrere Eingabelisten vorhanden sind, deren Länge sich zur Laufzeit unterscheidet, besteht eine grundlegende Möglichkeit damit umzugehen darin, die Instanzenzahl anhand der Länge der längsten Liste zu wählen und die übrigen Listen aufzufüllen. Die andere grundlegende Möglichkeit ist, die Länge anhand der Länge der kürzesten Liste zu wählen und überschüssige Elemente in den anderen Listen zu ignorieren.

Da beide Varianten in der Praxis nicht optimal sind, wurde hier ein Mittelweg gewählt: Die Anzahl der Instanzen wird anhand der Länge der kürzesten obligat zu lesenden Liste bestimmt.

Diese gesamte Logik kann allein im Mehrinstanzknoten implementiert werden, der sich dann sowohl nach außen hin als auch nach innen analog zu den bereits bestehenden Knoten verhält – außer der Einführung des Mehrinstanzknotens selbst sind also keinerlei Metamodellanpassungen erforderlich.

Damit ist es theoretisch sogar denkbar, einen Mehrinstanzknoten ohne jegliche Metamodelländerung als „Wrapperaktivität“ zu realisieren, das heißt als eine gewöhnliche Aktivität, der ein Aktivitätstemplate zugewiesen wird (das dann mehrfach instanziert wird). Dadurch wäre das
Vorhandensein einer Mehrfachinstanzierung allerdings nicht mehr sofort aus dem Prozessmodell zu erkennen und es müssten zur Zuordnung des zu instanzierenden Aktivitätstemplates voraussichtlich dennoch Anpassungen am Editor vorgenommen werden. Daher ist eine Implementierung als eigenes Metamodellkonstrukt deutlich vorteilhafter.

Insgesamt wird durch die Unterstützung variable Parallelität die Flexibilität zur Laufzeit weiter erhöht und damit die Modellierung vieler denkbarer Anwendungsfälle deutlich erleichtert, da nun in vielen Fällen auf instanzspezifische Änderungen verzichtet werden kann. Außerdem verbessert ADEPT damit die Kompatibilität zu Standards wie WS-BPEL 2.0 und einigen bestehenden Workflowsystem, wie auch in Kapitel 8 nochmals genauer dargestellt wird.

Die von dem hier vorgestellten Mehrinstanzknoten (nicht) unterstützten Workflow Patterns sind in Tabelle 4.5 aufgeführt: Die Pattern WCP-14 (Multiple Instances \emph{with} a priori Run-Time Knowledge) und das Datenpattern 4 (Multiple Instance Data) werden durch den hier vorgestellten Knoten voll unterstützt. WCP-15 (Multiple Instances \emph{without} a priori Run-Time Knowledge) und WCP-34 bis 36 (Partial Join for Multiple Instances) allerdings nicht. Bei WCP-15 liegt der Grund darin, dass dieses Pattern einer dynamischen Festlegung der Instanzanzahl entspricht, die wie wir gesehen haben, einige Schwierigkeiten auf wirft\textsuperscript{12}. Bei einem Partial Join hingegen besteht das Problem darin, dass dieser effektiv einer losen Synchronisation entspricht, die in ADEPT bezüglich des Datenflusses problematisch ist (vgl. auch Kapitel 5), so dass eine Unterstützung dieses Konzepts in diesem Kapitel ausgeklammert wurde – insbesondere ein Cancelling Partial Join ist aber auch in ADEPT durch einige kleine Erweiterungen realisierbar.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Anmerkungen</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCP-14 (Multiple Instances \emph{with} a priori Run-Time Knowledge)</td>
<td>— (Alle Kontextbedingungen voll erfüllt.)</td>
</tr>
<tr>
<td>Data Pattern 4 (Multiple Instance Data)</td>
<td>—</td>
</tr>
<tr>
<td>\textbf{Nicht} unterstützte verwandte Pattern:</td>
<td></td>
</tr>
<tr>
<td>WCP-15 (Multiple Instances \emph{without} a priori Run-Time Knowledge)</td>
<td>* Kann über instanzspezifische Änderungen realisiert werden * Datenherkunft für neue Instanzen in ADEPT unklar (siehe Absatz „Dynamische Festlegung“ in Abschnitt 4.2.2)</td>
</tr>
<tr>
<td>WCP-34 (Static Partial Join for Multiple Instances)</td>
<td>Entspricht einer Form der losen Synchronisation (siehe Kapitel 5)</td>
</tr>
<tr>
<td>WCP-35 (Cancelling Partial Join for Multiple Instances)</td>
<td>Entspricht einer Form der losen Synchronisation</td>
</tr>
<tr>
<td>WCP-36 (Dynamic Partial Join for Multiple Instances)</td>
<td>* Variante von WCP-15 * Entspricht einer Form der losen Synchronisation</td>
</tr>
</tbody>
</table>

\textbf{Tabelle 4.5: Unterstützte Workflow Patterns}

\textsuperscript{12}Es ist unklar, wie die Erstellung neuer Instanzen während der Ausführung des Multininstanzknotens ausgelöst werden soll und woher die Eingabedaten für die neuen Instanzen kommen sollen.
5. Lose Synchronisation paralleler Verzweigungen

Ein weiteres im Rahmen dieser Arbeit untersuchtes Konzept ist die lose Synchronisation paralleler Verzweigungen. Eine lose Synchronisation bedeutet, dass am Vereinigungsknoten nicht gewartet wird, bis alle Zweige fertig sind, sondern die Ausführung schreitet fort, sobald einige ausgewählte Zweige ihre Ausführung beendet haben. Dies hat einige Auswirkungen auf Kontroll- und vor allem den Datenfluss, die nachfolgend untersucht werden. Wie wir am Ende des vorigen Kapitels gesehen haben, kommt diesem Konzept auch im Rahmen der variablen Parallelität eine gewisse Bedeutung zu, da es auch dort sinnvoll sein kann, nicht auf die Fertigstellung aller Instanzen zu warten.

Bezüglich der Zweige, auf die nicht gewartet wird, lassen sich zwei in ADEPT mögliche Vorgehensweisen unterscheiden: Man kann diese Zweige bis zu ihrem Ende weiterlaufen lassen, oder man kann ihre Ausführung abbrechen, sobald der Vereinigungsknoten ausgeführt wird. Die Konsequenzen eines Weiterlaufs der Zweige werden in Abschnitt 5.2 untersucht, während ein Abbruch der Zweige in Abschnitt 5.3 diskutiert wird.

Bevor in diesen Abschnitten die beiden Varianten abgehandelt werden, wird in Abschnitt 5.1 zunächst das zugrundeliegende Problem nochmals ausführlicher dargestellt.

5.1. Problembeschreibung

Bislang wird an Vereinigungsknoten von Parallelverzweigungen gewartet, bis alle Zweige ihre Ausführung beendet haben. Es wird also eine vollständige Synchronisation aller Zweige durchgeführt. Für einige praxisrelevante Prozesse ist diese Synchronisationssemantik an AND-Joinknoten allerdings zu strikt. Bei diesen Prozessen ist es stattdessen notwendig, dass der Vereinigungsknoten ausgeführt wird und somit weiterläuft, sobald bestimmte Zweige fertig sind, unabhängig davon, ob die übrigen Zweige noch ausgeführt werden oder nicht.

Ein Beispiel für einen solchen Prozess ist in Abbildung 5.1 zu sehen. Dort ist ein mehrstufiger Prozess zur Beschwerdebearbeitung zu sehen. In jedem Schritt wird parallel zur eigenen Beschwerdebearbeitung ein Qualitätsmanagement durchgeführt, das das Fortschreiten des Gesamtprozesses aber nicht verzögern darf. Daher kann am Joinknoten am Ende des Schrittes nicht auf die Fertigstellung des Qualitätsmanagements gewartet werden, sondern der Knoten muss weiterlaufen, sobald die eigentliche Beschwerdebearbeitung abgeschlossen ist. Am Ende wird ein Gesamtqualitätsmanagement durchgeführt, das Daten aus dem Qualitätsmanagement der einzelnen Schritte verwendet – sofern solche vorhanden sind.

Das heißt, dass es zur Unterstützung dieses Prozesses die Möglichkeit geben muss, zur Entwurfszeit am Ende einer Parallelverzweigung auszuwählen, auf welche Zweige zur Laufzeit gewartet werden soll.
wird und auf welche nicht gewartet wird. Man kann auch sagen, dass die Zweige an dieser Stelle dann nicht mehr strikt, sondern „lose“ miteinander synchronisiert werden.

Da alleine die Zweige, auf die gewartet wird, den weiteren Kontrollfluss ablaufen bestimmen und daher aus Prozesssicht die wichtigeren Zweige sind, werden diese nachfolgend auch als „Masterzweige“ bezeichnet (z.B. der Zweig **Beschwerde bearbeiten** in Abbildung 5.1). Die unwichtigeren Zweige, auf die nicht gewartet wird, werden dementsprechend „Slavezweige“ genannt (Zweig **QM** in Abbildung 5.1).

Beim Weiterlaufen gibt es verschiedene Möglichkeiten, wie mit den Slavezweigen umgegangen werden kann, sobald alle Masterzweige fertig sind: Die Slavezweige können entweder bis zu Ihrem Ende weiterlaufen, abgebrochen oder abgebrochen und zurückgesetzt werden.

Beim Weiterlaufen gibt es darüberhinaus zwei mögliche Semantiken, wenn einer (bzw. alle) der Slavezweige seine Ausführung beendet hat: In diesem Fall kann der Vereinigungsknoten entweder nochmals schalten oder die Fertigstellung von Slavezweigen nicht beachten (der durch die Masterzweige ausgelöste Schaltvorgang ist in letzterem Fall also der einzige). „Nochmaliges Schalten“ bedeutet hier, dass der AND-Joinknoten und seine Nachfolger nochmals ausgeführt werden (in Petrinetzen würde dafür ein weiteres Kontrollflusstoken weitergegeben). Diese Möglichkeiten sind im Rahmen der Workflow Patterns in den Pattern WCP-8 (nochmaliges Schalten) und WCP-9 (ein einziger Schaltvorgang) berücksichtigt (vgl. auch Tabelle 5.1).


Somit ergibt sich aus Implementierungssicht das Problem, dass ein zweiter (oder dritter, vierter, . . . ) Satz von Graphmarkierungen verwaltet werden muss (insbesondere, da es sich nicht ausschließen lässt, dass der zweite Kontrollflussstrang den ersten überholt). Im Endeffekt führt dies dazu, dass sich die Ausführungssemantik von ADEPT stark in Richtung eines petrinetz-basierten Systems bewegt, was wiederum vielfältige Probleme bei Korrektheitsanalysen und instanzspezifischen Änderungen mit sich bringt (vgl. auch [Rei00]). Aus diesen Gründen wird

**Abbildung 5.1:** Beispiel für eine lose Synchronisation paralleler Verzweigungen
nachfolgend immer von einem einmaligen Schaltvorgang des Joinknotens ausgegangen (es sei denn, es wird ausdrücklich auf eine andere Semantik hingewiesen).

Die Wahlfreiheit bezüglich der genauen Interpretation loser Synchronisation spiegelt sich auch innerhalb der Workflow Patterns wider: Außer für das Zurücksetzen gibt es für jede der soeben beschriebenen Möglichkeiten Pattern, die eine entsprechende Semantik implementieren, wie in Tabelle 5.1 dargestellt.

<table>
<thead>
<tr>
<th>Semantik</th>
<th>Pattern mit Reaktion auf ersten Zweig</th>
<th>Pattern mit Reaktion auf erste n Zweige</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abbruch der Slavezweige</td>
<td>WCP-29 (Cancelling Discriminator)</td>
<td>WCP-32 (Cancelling Partial Join)</td>
</tr>
<tr>
<td>Weiterlaufen lassen mit einmaliger Aktivierung des Joinknotens</td>
<td>WCP-9 (Structured Discriminator)</td>
<td>WCP-30 (Structured Partial Join)</td>
</tr>
<tr>
<td>Weiterlaufen lassen mit mehrmaliger Aktivierung des Joinknotens</td>
<td>WCP-8 (Multi-Merge)</td>
<td></td>
</tr>
</tbody>
</table>

Tabelle 5.1: Im Rahmen loser Synchronisation relevante Workflow Patterns


Auch wenn die Workflow Patterns alle drei soeben beschriebenen Möglichkeiten unterstützen, implementiert kein Pattern genau die hier beschriebene Semantik genau:

In dieser Arbeit wird bei der losen Synchronisation explizit zwischen Master- und Slavezweigen unterschieden – also Zweigen, auf die gewartet wird, und Zweigen, auf die nicht gewartet wird. Innerhalb der Workflow Patterns sind dagegen alle Zweige semantisch gleichwertig, der Joinknoten schaltet, sobald der erste (oder die ersten $n$) Zweige fertig sind – und zwar unabhängig davon, welche Zweige dies sind. Analog zur finalen Auswahl bei Parallelverzweigungen aus [Rei00], WCP-29 entspricht daher wie in Tabelle 2.1 erwähnt recht genau der finalen Auswahl. Der einzige Unterschied besteht darin, dass bei der finalen Auswahl die zu spät kommenden Zweige zurückgesetzt und nicht wie bei WCP-29 lediglich abgebrochen werden.

Zudem beziehen sich diese Pattern (wie bei den Workflow Patterns üblich) lediglich auf den Vereinigungsknoten, an dem die eigentliche lose Synchronisation stattfindet. Der Verzweigungsknoten verhält sich immer wie ein gewöhnlicher AND-Splitknoten, weshalb ihm in allen Fällen WCP-2 (Parallel Split) zugeordnet werden kann.

Bei näherer Untersuchung stellt man allerdings fest, dass sich die hier verwendete Semantik relativ einfach auf die genannten Pattern zurückführen lässt: Wie in Abbildung 5.2 graphisch dargestellt, werden alle Masterzweige und alle Slavezweige in jeweils einen mittels eines gewöhnlichen parallelen Vereinigungsknoten synchronisierten Zweig einer Verzweigung mit der Semantik des entsprechenden Patterns gelegt. Zusätzlich muss sichergestellt werden, dass die
Slavezweige nicht vor den Masterzweigen fertig sind, diese müssen also auf deren Ende warten.

Abbildung 5.2: Modellierung einer losen Synchronisation mit Hilfe der Workflow Patterns

Der umgekehrte Weg (also eine Zurückführung der Patterns auf die hier untersuchte Semantik) ist allerdings nicht ohne Weiteres möglich. Die hier untersuchte Semantik stellt also eine Spezialisierung gegenüber den entsprechenden Workflow Patterns dar. Diese ist jedoch aus Gründen des Datenflusses sinnvoll – wie wir in diesem Kapitel noch sehen werden.

5.2. Lose Synchronisation mit Weiterlaufen

In diesem Abschnitt wird als erste der beiden in ADEPT sinnvollen Varianten die lose Synchronisation mit Weiterlaufen der Slavezweige behandelt. Dazu werden zunächst in Abschnitt 5.2.1 die bei dieser Variante charakteristischen Probleme beschrieben. Anschließend werden in Abschnitt 5.2.2 die sich daraus ergebenden Auswirkungen einer Umsetzung einer solchen losen Synchronisation auf das existierende Metamodell untersucht. Schließlich wird in Abschnitt 5.2.3 eine kurze Zusammenfassung gegeben.

5.2.1. Problem beschreibung

Bei einer losen Synchronisation mit Weiterlaufen der Slavezweige kann sich zur Laufzeit eine Graphmarkierung wie in Abbildung 5.1 ergeben. Es kann also vorkommen, dass eine Aktivität in einem Slavezweig nebenläufig zu einem ihrer Nachfolger ausgeführt wird (z.B. der vordere QM-Knoten zusammen mit Beschwerde bearbeiten im zweiten Schritt). Da nicht sicher ist, dass eine Slaveaktivität beendet ist, bevor ihre Nachfolger hinter der losen Synchronisation ausgeführt werden, führt dies zu der offensichtlichen Einschränkung im Datenfluss, dass in einem Slavezweig geschriebene Daten nicht obligat gelesen werden können.

Im diesem konkreten Beispiel ist dies auch die einzige sich durch die lose Synchronisation ergebende Einschränkung; Gesamt-QM darf weder QM-Daten 1 noch QM-Daten 2 obligat, sondern nur optional, lesen – dies ist im gegebenen Prozess aber kein Problem.

Es lassen sich allerdings komplexere Beispiele (insbesondere im Zusammenhang mit Schleifen) angeben, die zeigen, dass eine direkte, naive Implementierung dieses Konzepts prinzipbedingt
zu einigen schwer zu lösenden Problemen führt. Diese werden im nachfolgenden Abschnitt zunächst aus Kontrollflussssicht und anschließend aus Datenflussssicht anhand einiger Beispiele diskutiert.

5.2.2. Existierendes Metamodell

5.2.2.1. Kontrollfluss


Im abgebildeten Fall steht man dabei aber vor der Schwierigkeit, dass Knoten D noch ausgeführt wird. Werden die Markierungen in diesem Fall einfach naiv zurückgesetzt, hat die Markierung des Knotens D nichts mehr mit dem Ausführungsstatus der zugewiesenen Aktivität zu tun. Solange D seinen Ausführungsstatus nicht ändert, ist dies nicht wirklich problematisch. Sobald aber ein Ereignis eintritt, das eigentlich eine Statusänderung nach sich zieht, ist zunächst einmal unklar, ob D und sein Nachfolger E direkt neu markiert werden dürfen oder nicht.

Konkret können die folgenden problematischen Ereignisse eintreten:

1. Knoten B ist in einer neuen Iteration fertig:
   In diesem Fall muss eine weitere Instanz von D gestartet werden, die parallel zur alten Instanz dieses Knotens ausgeführt wird. Es ergibt sich also eine Art impliziter variabler Parallelität bezüglich der Knoten D und E, da von diesen Knoten potentiell so viele Instanzen gleichzeitig aktiv sein können, wie die Schleife Iterationen durchläuft.

2. D hat seine Ausführung beendet:
   Hier muss Knoten E ausgeführt werden, wobei er aus d1 diejenigen Daten liest, die B in der Startiteration von D dort hinein geschrieben hat. E darf nicht den Wert aus der aktuellen Iteration lesen, da sonst ein korrekter Ablauf in der früheren Iteration nicht mehr gewährleistet wäre.
Da der Kontrollfluss aus früheren Iterationen von dem der aktuellen Iteration unabhängig sein muss, dürfen für ein korrektes Verhalten im zweiten Fall nicht einfach D oder E direkt neu markiert werden. Zudem dürfen D und E nicht einfach den neuesten Wert aus den Datenelementen lesen bzw. diesen überschreiben.


Außerdem ist unklar, was zu tun ist, wenn der Kontrollfluss den Endknoten erreicht, aber noch Instanzen von D und E aktiv sind. In diesem Fall können entweder die noch laufenden Instanzen abgebrochen werden oder man lässt den Prozess solange weiter laufen, bis alle Instanzen von D und E beendet sind, implementiert also eine Art impliziter Terminierung.

Man kann argumentieren, dass ein Abbruch der Instanzen zu bevorzugen ist, da dadurch eine implizite Terminierung des Prozesses sichergestellt ist und sie der intuitiven Semantik eines Endknotens näher kommt. Umgekehrt lassen sich aber auch Fälle angeben, in denen eine implizite Terminierung vorteilhafter ist (beispielsweise, wenn es wichtig ist, dass die Slavezweige auf jeden Fall zu Ende laufen dürfen).

Letztlich ist dies daher im Wesentlichen ein Definitionsproblem, da man sich für eine der beiden Alternativen entscheiden kann (und muss) – ohne, dass sich dadurch wirklich gravierende Vor- oder Nachteile gegenüber der jeweils anderen Alternative ergeben.

5.2.2.2. Datenfluss

Die gravierenderen Probleme ergeben sich bei loser Synchronisation mit Weiterlaufen allerdings beim Datenfluss und nicht beim Kontrollfluss.

Da nicht sicher ist, dass ein Knoten in einem Slavezweig bereits ausgeführt wurde, wenn ein Leseknoten hinter der losen Synchronisation ausgeführt wird, dürfen lediglich innerhalb eines Slavezweigs geschriebene Datenelemente hinter dem Joinknoten, an dem die lose Synchronisation stattfindet, nicht obligat gelesen werden (wie in der Einleitung schon beschrieben wurde). So dürfen die Knoten H und I in Abbildung 5.3 das Datenelement d2 nicht obligat lesen, da nicht sicher ist, dass der Knoten E bereits ausgeführt wurde und d2 mit Daten versorgt hat, wenn H ausgeführt wird.


Desweiteren muss dafür gesorgt werden, dass jeder Instanz der lose synchronisierten Zweige die richtige Schattenkopie der Datenelemente bei den Schreib- und Lesezugriffen zugeordnet wird. Dies lässt sich durch eine kleine Erweiterung der ursprünglich für die finale Auswahl vorgesehenen Logik in ReadDataElement (siehe [Rei00]) relativ problemlos erreichen.

\footnote{Ein Endknoten beendet den Prozess; vgl. auch die flexiblen Endknoten in Kapitel 6.}
Außerdem muss die korrekte (Un-)Sichtbarkeit der Schattenkopien von außerhalb der losen Synchronisation beachtet werden: Ein Knoten hinter der losen Synchronisation muss Daten, die innerhalb eines Slavezweigs geschrieben wurden, (optional) lesen können. Dabei muss immer der logisch gesehen neueste vorhandene Wert sichtbar sein, d.h. wenn die Knoteninstanz aus der ersten Iteration bereits fertig ist, die aus der zweiten aber noch nicht, muss der Wert aus der ersten Iteration (nicht etwa NULL oder ähnlich) gelesen werden. Umgekehrt muss aber, sobald die zweite Instanz fertig ist, auch der von dieser geschriebene Wert – und nicht etwa ein Wert aus einer früheren Instanz – gelesen werden.

Nun ergibt sich allerdings das Problem, dass es innerhalb einer Schleife nicht sicher ist, dass die einzelnen Zweiginstanzen ihre Ausführung chronologisch auch in der selben Reihenfolge beenden, in der sie gestartet wurden. Daher muss sichergestellt werden, dass nicht die chronologisch zuletzt durchgeführten Zugriffe sichtbar sind, sondern dass die der höchsten Schleifeniteration zugewiesene Schattenkopie, auf die bereits ein Schreibzugriff erfolgt ist, sichtbar ist.

So ist es in Abbildung 5.3 nicht sicher, dass die Instanz von E, die in der ersten Iteration der Schleife gestartet wurde, auch vor derjenigen Instanz von E beendet ist, die in der zweiten Iteration gestartet wurde. Das heißt, dass der zur ersten Iteration gehörende Wert von d2 nach demjenigen geschrieben werden kann, der zur zweiten Iteration gehört. Der zur ersten Iteration gehörende Wert darf in diesem Fall den Wert der zweiten Iteration aus Sicht von H aber nicht überschreiben.


Abbildung 5.4: Mögliches Überschreiben von Daten bei loser Synchronisation

Ein Beispiel dafür ist in Abbildung 5.4 zu sehen. Hier muss sichergestellt werden, dass Knoten D den Wert des Datenelements d nicht überschreibt, so dass H in jedem Fall den von G geschriebenen Wert liest. Umgekehrt muss aber E den von D geschriebenen Wert lesen und nicht den von G geschriebenen.

Letztlich können aber auch diese beiden Anforderungen mittels entsprechender Erweiterungen an ReadDataElement ohne größere Schwierigkeiten realisiert werden.

Das schwerwiegendste Datenflussproblem stellt allerdings der sich ergebende Nichtdeterminismus des Werts von Datenelementen dar: Es ist zur Entwurfszeit völlig unklar, wann (relativ zur


**Abbildung 5.5:** Nichtdeterminismus des Werts von Datenelementen bei loser Synchronisation

Eine Möglichkeit, diesen Nichtdeterminismus etwas zu reduzieren, ist es, die Semantik so zu definieren, dass nur diejenigen Werte hinter der losen Synchronisation lesbar sind, die bei Aktivierung des Joinknotens bereits geschrieben waren. Dadurch ist in Abbildung 5.3 zwar immer noch nicht klar, aus welcher Iteration der Wert von d2 stammt, aber es ist sicher, dass H und I den selben Wert lesen. Ohne das Vorhandensein einer Schleife ist der Nichtdeterminismus damit sogar vollständig beseitigt (beispielsweise in Abbildung 5.5).

Allerdings widerspricht diese Semantik der Intuition, da man bei einem Weiterlaufen der Slavezweige eigentlich erwartet, dass immer der neueste verfügbare Wert gelesen werden kann. Effektiv wird damit auch der Nutzen des Weiterlaufens von Slavezweigen stark vermindert, da diese nach Abschluss der Ausführung der Masterzweige keine Daten mehr schreiben können,
die vom Rest des Prozesses gelesen werden können. Aus Sicht des übrigen Prozesses kommt diese Semantik in ihren Datenflusseffekten daher mehr oder weniger einem Abbruch der Slave-
zwiege gleich (die Slavezweige laufen zwar weiter, dies hat allerdings keinerlei Effekte auf den
restlichen Prozess).

5.2.3. Zusammenfassung

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Anmerkungen</th>
</tr>
</thead>
</table>
| WCP-2 (*Parallel Split*) | * Entspricht Splitknoten  
* Wie normaler AND-Split |
| WCP-9 (*Structured Discriminator*) | * Entspricht Joinknoten  
* Hier untersuchtes Konzept ist Spezialisierung des Patterns (ausgrund von Problemen beim Datenfluss) |
| WCP-30 (*Structured Partial Join*) | * Entspricht Joinknoten  
* Generalisierung von WCP-9  
* Hier untersuchtes Konzept ist Spezialisierung des Patterns (ausgrund von Problemen beim Datenfluss) |

Tabelle 5.2: Unterstützte Workflow Patterns

Wie in diesem Abschnitt gezeigt wurde, führt eine direkte Implementierung des Weiterlaufens von Slavezweigen bei loser Synchronisation zu einigen schwerwiegenden prinzipiellen Proble-
men. Insbesondere der kaum zu verhindernende Nichtdeterminismus des Werts von Datenele-
menten durch die implizite Parallelanführung lose synchronisierter Knoten stellt eine ständige
potentielle Fehlerquelle dar. Vor diesem Hintergrund ist eine Implementierung dieser Art der
loosen Synchronisation nur wenig sinnvoll und wird daher in der in diesem Abschnitt beschrie-
benen Form nicht mehr weiter verfolgt.

Obwohl eine konkrete Implementierung nicht näher untersucht wird, enthält Tabelle 5.2 der
Vollständigkeit halber eine kurze Übersicht, welchen Workflow Patterns die soeben untersuchte
Lose Synchronisation mit Weiterlaufen entspricht.

\footnote{Gerade wegen solcher Nichtdeterminismen ist ja das unsynchronisierte parallele Schreiben in ADEPT nicht zulässig!}
5. Lose Synchronisation paralleler Verzweigungen

5.3. AND-Join mit partiellen Abbrüchen

5.3.1. Problembeschreibung


5.3.2. Existierendes Metamodell

5.3.2.1. Kontrollfluss


Dies ist in Abbildung 5.6 illustriert: In Teil (a) werden noch alle Verzwege ausgeführt, da der Masterverzweig C•D noch nicht fertig ist. In Teil (b) ist der Knoten D schließlich mit seiner Ausführung fertig, und der noch laufende Slaveverzweig E•F wird abgebrochen und mit SKIPPED markiert. Die Markierung des bereits beendeten Slaveknotens G bleibt allerdings unverändert.

Falls ein leerer Masterverzweig vorhanden ist, gilt dieser Zweig sofort als fertig und wird dementsprechend behandelt. Daraus folgt, dass alle Slaveverzweige sofort abgebrochen werden, sofern nur ein einziger, leerer Masterverzweig vorhanden ist. Dieses Verhalten ist in den meisten Fällen nicht besonders sinnvoll, so dass in diesem Fall der Modellierer eine Warnung erhalten muss. Es stellt aber auch keinen Fehler im eigentlichen Sinne dar, da dieses Verhalten klar aus der oben
genannten Semantik hervorgeht und zu keinem überraschenden oder inkonsistenten Verhalten des Prozesses führt (ähnlich wie ein `if (false)` in Java).

Wenn an einem AND-Join mit partiellem Abbruch nur Masterzweige vorhanden sind, ist die Semantik nach der obigen Beschreibung ebenfalls klar: Es wird gewartet, bis alle Masterzweige fertig sind und dann der Joinknoten aktiviert. Da keine Slavezweige existieren, brauchen natürlich auch keine solchen abgebrochen werden. In diesem Fall verhält sich ein AND-Block mit partiellem Abbruch also genau wie ein gewöhnlicher AND-Block und könnte eigentlich ohne Weiteres durch einen solchen ersetzt werden. Der AND-Join mit Abbruch ist in diesem Fall also streng genommen überflüssig\(^3\). Da die Semantik hier klar und konsistent ist, kann ein AND-Join mit partiellem Abbruch ohne Slavezweige aber dennoch problemlos erlaubt werden.


**Sync-Kanten** Synchronisationskanten können bei einem AND-Block mit partiellem Abbruch ohne Schwierigkeiten hinzugefügt werden:

Sync-Kanten zwischen zwei Masterzweigen oder zwei Slavezweigen sind offensichtlich unproblematisch, da in diesem Fall beide Zweige bezüglich der Synchronisation semantisch gleichwertig sind und sich daher keine anderen Auswirkungen der Sync-Kanten als bei einem gewöhnlichen AND-Block ergeben können.

\(^3\)Was aber z.B. auch für AND-Blöcke mit nur einem Zweig gilt.
Dies kann man sich durch folgende Überlegung klar machen: Die Semantik eines AND-Join mit Abbruch ändert sich nicht, wenn, wie in Abbildung 5.7b, alle Master- und alle Slavezweige nochmals in je einen AND-Block geschachtelt werden. Dadurch werden Sync-Kanten, die zwischen zwei Masterzweigen oder zwei Slavezweigen verlaufen, zu Sync-Kanten innerhalb dieses AND-Blocks, sind also offensichtlich zulässig (wie beispielsweise die Kanten $E \rightarrow D$ und $G \rightarrow F$ in Abbildung 5.7).


**Abbildung 5.7**: Synchronisationskanten bei einem AND-Join mit Abbruch

Insgesamt können Sync-Kanten also – unabhängig vom Zweigtyp – unter genau den gleichen Einschränkungen wie bei einem gewöhnlichen AND-Block zu einem AND-Block mit partiell Abbruch hinzugefügt werden (das heißt Zyklusfreiheit und nur zwischen potentiell parallel ausgeführten Knoten).
5.3. AND-Join mit partiellen Abbruch

5.3.2.2. Datenfluss


Hier wird auch klar, weshalb die hier vorgenommene Unterscheidung zwischen Master- und Slavezweigen sinnvoll ist: Wären alle Zweige (wie bei den Workflow Patterns) semantisch gleichwertig, so könnten aus keinem Zweig heraus Daten obligat geschrieben werden, da man bei keinem Zweig sicher davon ausgehen könnte, dass er vollständig ausgeführt wird. Aus Datenflussicht ist die hier vorgestellte Variante daher die sinnvollere und ausdrucksmächtigere.

Falls es Sync-Kanten von einem Slaveknoten zu einem Masterknoten gibt, ist allerdings eine Spezialbehandlung nötig:

In diesem Fall gilt, dass ein Datenelement, das von den Vorgängerknoten der Sync-Kante im Slavezweig obligat geschrieben wird, auch am Joinknoten des AND-Blocks mit partiellen Abbruch obligat gelesen werden darf, sofern dieses Datenelement am Zielknoten der Synchronisationskante obligat gelesen werden kann (was der Regelfall ist).

Das Datenelement darf am Zielknoten der Sync-Kante nur dann nicht gelesen werden, wenn sich der Slaveknoten innerhalb einer XOR-Verzweigung befindet, die innerhalb des AND-Blocks mit partiellen Abbruch geschachtelt ist. In diesem Fall kann der Zielknoten im Masterzweig auch ausgeführt werden, obwohl der Slaveknoten aufgrund der Wahl eines anderen XOR-Zweigs überspringen wurde, so dass die Versorgtheit nicht sichergestellt ist (OR-Splitkriterium aus [Rei00]).

Dies alles lässt sich durch entsprechende Erweiterungen des WriterExists-Algorithmus berücksichtigen, die in Anhang B.2 dargestellt sind.

Ein Beispiel für den Datenfluss bei einem AND-Join mit partiellen Abbruch findet sich in Abbildung 5.8. d1 darf an I obligat gelesen werden, da es an C innerhalb eines Masterzweigs und somit in jedem Fall geschrieben wird. d2 hingegen nicht, da sich G in einem Slavezweig befindet und somit nicht sicher ist, dass G ausgeführt wurde, wenn I gestartet wird. d3 hingegen darf an I wiederum gelesen werden, obwohl sich der schreibende Knoten E innerhalb eines Slavezweigs befindet. Der Grund dafür ist, dass durch die Sync-Kante E→D sichergestellt ist, dass E vor D ausgeführt wird und somit der Masterzweig C→D erst dann beendet wird, wenn E beendet ist. Da Slavezweige aber erst abgebrochen werden, wenn alle Masterzweige fertig sind, wird E somit immer ausgeführt und d3 darf an I obligat gelesen werden. Oder anders formuliert: d3 darf durch die Sync-Kante am Knoten D, also am Ende eines Masterzweigs, obligat gelesen werden, also auch seinen Nachfolgern H und I.
Parallele Schreibzugriffe auf Datenelemente  Im Gegensatz zur sicheren Versorgung von Datenelementen können Master- und Slavezweige bezüglich parallelter Schreibzugriffe identisch behandelt werden. Es werden unabhängig vom Zweigtyp wie in einem gewöhnlichen AND-Block alle Knoten potentiell parallel zueinander ausgeführt. Daher muss auch hier die selbe Einschränkung gelten, nämlich, dass ein Schreibzugriff zweier Knoten in zwei verschiedenen Zweigen auf das selbe Datenelement nicht erlaubt ist, es sei denn, die Ausführungsreihefolge wurde durch eine Sync-Kante eindeutig festgelegt.

Bei der Analyse paralleler Schreibzugriffe (ParallelWriterExists) können AND-Blöcke mit partiellem Abbruch daher identisch zu gewöhnlichen AND-Blöcken ohne Abbruch behandelt werden.

Weiterlaufen Lassen über Entkoppeln der Slavezweige  Aus Sicht des umgebenden Prozesses ist es gleichgültig, ob innerhalb einer losen Synchronisation die Slavezweige abgebrochen werden, oder ob diese weiterlaufen dürfen, aber keinerlei Daten mehr so schreiben können, dass diese von außerhalb gelesen werden können (wie am Ende von Abschnitt 5.2.2 schon festgestellt wurde). Diese beiden Semantiken können aus dieser externen Sicht also als äquivalent betrachtet werden, so dass sie insbesondere bei der Datenflussanalyse nicht separat behandelt werden müssen. Da die Slavezweige – analog zu einer unabhängigen Prozessinstanz – weiterlaufen, ohne den umgebenden Prozess zu beeinflussen, kann man in diesem Fall auch von einem „Entkoppeln“ der Slavezweige sprechen.

Aufgrund dieser Äquivalenz liegt es auf den ersten Blick nahe, in einem AND-Block mit Abbruch wahlweise die Semantik zu bieten, dass Slavezweige (statt abgebrochen zu werden) weiterlaufen können – allerdings ohne die Möglichkeit zu haben, Daten an den umgebenden Prozess zu schreiben.

Die Äquivalenz beider Semantiken illustriert Abbildung 5.9: An Knoten B wird für das Datenelement d der Wert 1 gelesen, unabhängig davon, ob der Slavezweig abgebrochen wird oder
5.3. AND-Join mit partiellen Abbruch

Abbildung 5.9: Äquivalenz zwischen dem Abbruch von Slavezweigen und dem Weiterlaufen ohne Daten zu schreiben ohne nach außen sichtbare Schreibzugriffe weiterläuft. Im Fall (b) ist zu erkennen, dass diese Schreibzugriffe nur nach außen unsichtbar sein dürfen; nach innen müssen diese für eine korrekte Semantik des Zweigs weiterhin sichtbar sein (A muss in jedem Fall d=4 lesen). Das heißt, dass die Schreibzugriffe im Allgemeinen nicht einfach ignoriert werden dürfen, sondern dass die Slavezweige eigene Schattenkopien der Datenelemente verwenden müssen.

Es müssen also auch bei einem solchen wahlweisen Weiterlaufen der Zweige innerhalb eines AND-Block mit partiellem Abbruch wie in Abschnitt 5.2.2 Schattenkopien der Knotenmarkierungen der Slavezweige und der referenzierten Datenelemente angelegt werden (insbesondere, falls sich die lose Synchronisation innerhalb einer Schleife befindet). Aufgrund der vollkommenen Unsichtbarkeit dieser Schattenkopien nach außen hin werden hier aber die sich Abschnitt 5.2.2 ergebenden Nichtdeterminismen im Datenfluss vermieden. Der Implementierungsaufwand wird allerdings nicht wesentlich geringer sein als in jenem Fall.

Bei der Implementierung ist desweiteren zu beachten, dass Sync-Kanten zwischen mehreren Slavezweigen weiterhin die intendierte Semantik aufweisen müssen, so dass die einzelnen Slavezweige nicht als (vollig) unabhängig voneinander betrachtet werden dürfen.

Bei einer Implementierung dieser alternativen Semantik muss zunächst herausgefunden werden, von welchen Knotenmarkierungen und von welchen Datenelementen Schattenkopien angelegt werden müssen. Es genügt dabei, Schattenkopien der Knotenmarkierungen derjenigen Knoten innerhalb der Slavezweige anzulegen, die ihre Ausführung noch nicht beendet haben, also deren Markierung ≠ COMPLETED ist. Bezüglich der Datenelemente müssen nur von jenen Schattenkopien angelegt werden, die von einem der noch nicht beendeten Slaveknoten gelesen werden, da in diesem Fall der Originalwert erhalten bleiben muss. Falls einige Datenelemente lediglich geschrieben werden, müssen keine Schattenkopien davon angelegt werden, da Schreibzugriffe in diesem Fall einfach ignoriert werden können.

Sobald diese Schattenkopien angelegt sind, müssen nur noch sämtliche Markierungsvorgänge und Datenzugriffe der betroffenen Slaveknoten auf die entsprechende Schattenkopie umgeleitet werden, was durch Erweiterungen der bisherigen Algorithmen machbar ist.

\footnote{Natürlich kann es bei der Implementierung aus Orthogonalitätsgründen praktischer sein, auch in diesem Fall Schattenkopien anzulegen.}

\footnote{Beachte dabei, dass auch geschachtelte lose Synchronisationen möglich sind, so dass ggf. auch Schattenkopien von Schattenkopien angelegt werden müssen.}
Eine denkbare Möglichkeit, für Datenzugriffe solche Schattenkopien zu implementieren, ist es, \texttt{ReadDataElement} dahingehend zu modifizieren, dass nicht mehr der chronologisch aktuellste Wert eines beliebigen Vorgängers gelesen wird, sondern dass Vorgänger, die näher am lesenden Knoten liegen, bevorzuge werden („Nahe“ bezieht sich in diesem Zusammenhang auf die Anzahl der Kontrollflusskanten auf dem Weg von einem Vorgänger zum Leseknoten). In diesem Fall reicht es zur Implementierung von Schattenkopien für Datenelemente aus, wenn der Joinknoten bei seinem Schalten alle zu diesem Zeitpunkt gültigen Werte der von den Slavezweigen geschriebenen Datenelemente nochmals schreibt. Dadurch lesen Nachfolger des Joinknotens immer höchstens die Werte, die der Joinknoten geschrieben hat, unabhängig davon, ob ein Slavezweig später noch Daten schreibt oder nicht (der Joinknoten „schneidet“ seine Nachfolger dadurch also von den Schreibzugriffen der Vorgänger ab). Innerhalb der Slavezweige sind dementsprechend ebenfalls nur die Werte von Vorgängern (aus der selben oder einer früheren Schleifeniteration) sichtbar, so dass auch hier eine korrekte Semantik sichergestellt ist.

Vor einer Implementierung einer solchen Semantik muss allerdings noch im Detail untersucht werden, wie (und ob) eine solche Semantik in \texttt{ReadDataElement} umgesetzt werden kann\textsuperscript{6}, ohne dass die Korrektheit des Datenflusses gefährdet ist.

Eine Implementierung von Schattenkopien ist also prinzipiell machbar, bedeutet aber einen nicht unerheblichen Implementierungsaufwand. Demgegenüber steht eine erweiterte Semantik, bei der man allerdings einwenden kann, dass ein Weiterlaufen in dieser Form in der Praxis möglicherweise nicht benötigt wird bzw. es verwirrend sein kann, wenn manche Schreibzugriffe sichtbar sind (jene vor Beendigung der Masterzweige) und manche nicht (jene nach Beendigung der Masterzweige).

Es kann daher sinnvoller sein, die Slavezweige für das Weiterlaufen komplett zu entkoppeln, also unabhängig vom Status der Masterzweige zu verhindern, dass Schreibzugriffe der Slavezweige im Rest des Prozesses sichtbar sind. Allerdings ist dies ein völlig anderes Konzept als ein AND-Join mit partiellem Abbruch und auch keine lose Synchronisation im eigentlichen Sinne mehr, sondern eher ein unsynchronisiertes Starten einer Subprozessinstanz, so dass dieses vollständige Entkoppeln an dieser Stelle nicht weiter untersucht wird.

5.3.3. Zusammenfassung

Wie wir gesehen haben, lässt sich eine lose Synchronisation mit Abbruch der Slavezweige in ADEPT ohne größere Schwierigkeiten implementieren. Es sind lediglich die Einführung eines neuen Knotentyps (AND-Join mit partiellem Abbruch) und damit verbundene Anpassungen der Datenflussanalyse erforderlich.


\textsuperscript{6}Ein Ansatz ist beispielsweise, die Historie eines Datenelements nicht mehr chronologisch, sondern nach der strukturellen Position des schreibenden Knotens und der Nummer der Schleifeniteration, in der der Schreibzugriff stattfand, zu sortieren.
5.4. Zusammenfassung


Aufgrund dieser Datenflussprobleme und der Tatsache, dass in praxisrelevanten Prozessen üblicherweise schon zur Entwurfszeit bekannt ist, welche Zweige abgebrochen werden dürfen und welche komplett ausgeführt werden müssen, wurde hier mit dem AND-Join mit Abbruch eine Spezialisierung von WCP-29 bzw. WCP-32 betrachtet. Durch die Unterscheidung zwischen Master- und Slavezweigen wird die Menge der potentiell abgebrochenen Zweige auf die Slavezweige eingeschränkt, wodurch ein obligates Schreiben aus den Masterzweigen heraus völlig unproblematisch ist. Lediglich die Slavezweige können keine Daten obligat schreiben, was allerdings aufgrund der Semantik unvermeidlich ist.

Ein AND-Join mit Abbruch schafft es also, die beschriebenen Schwierigkeiten im Datenfluss soweit wie möglich zu vermeiden, ohne die Praxistauglichkeit zu beeinträchtigen. Eine Zusammenfassung der einer losen Synchronisation mit Abbruch entsprechenden Patterns ist in Tabelle 5.3 zu sehen. Der Splitknoten entspricht hier wie schon bei einer losen Synchronisation mit Weiterlaufen einem gewöhnlichen AND-Split.

5.4. Zusammenfassung

Wie wir in diesem Kapitel gesehen haben, lässt sich eine lose Synchronisation in ADEPT nur dann zufriedenstellend implementieren, wenn die unsynchronisierten Zweige abgebrochen werden.

\[ WCP-32 \text{ schaltet sobald } n \text{ Zweige fertig sind; setze } n = 1 \text{ für ein Verhalten wie } WCP-29 \]
### Tabelle 5.3: Unterstützte Workflow Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Anmerkungen</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCP-2 (Paralleled Split)</td>
<td>* Entspricht Splitknoten</td>
</tr>
<tr>
<td></td>
<td>* Wie normaler AND-Split</td>
</tr>
<tr>
<td>WCP-29 ( Cancelling Discriminator)</td>
<td>* Entspricht Joinknoten</td>
</tr>
<tr>
<td></td>
<td>* AND-Join mit partiellen Abbruch ist Spezialisierung des Patterns (zur Vermeidung von Datenflussproblemen)</td>
</tr>
<tr>
<td>WCP-32 ( Cancelling Partial Join)</td>
<td>* Entspricht Joinknoten</td>
</tr>
<tr>
<td></td>
<td>* Generalisierung von WCP-29</td>
</tr>
<tr>
<td></td>
<td>* AND-Join mit partiellen Abbruch ist Spezialisierung des Patterns (zur Vermeidung von Datenflussproblemen)</td>
</tr>
</tbody>
</table>

Werden die Zweige nicht abgebrochen, sondern dürfen bis zu ihrem Ende weiterlaufen und Daten schreiben, ergibt sich eine nichtdeterministische Datenflusssemantik. Diesem Nichtdeterminismus kann letztlich nur dadurch abgeholfen werden, dass ein Synchronisationspunkt für den Datenfluss erzwungen wird, also ein Punkt, ab dem die Slavezweige keine Daten mehr schreiben dürfen. Eine Möglichkeit dies zu tun, ist der Abbruch der Slavezweige, sobald die Masterzweige fertig sind. Eine andere Möglichkeit ist das Entkoppeln der Slavezweige (also Ausführen, ohne die Möglichkeit Daten zu schreiben) – im Fall eines vollständigen Entkoppels von Zweigen ist es allerdings fraglich, inwiefern man dies noch als „lose Synchronisation“ bezeichnen kann, da dies eher einem unsynchronisierten Start einer separaten Subprozessinstanz gleichkommt (vgl. auch Pattern WCP-12 (Multiple Instances without Synchronisation)).

Außerdem ergeben sich durch eine lose Synchronisation ohne Abbruch bei einem markierungs-basierten Verfahren wie ADEPT generell diverse Probleme mit der korrekten Markierung der Knoten, insbesondere beim Einsatz innerhalb von Schleifen (wodurch Schattenkopien der Markierungen nötig werden). Diese treten bei tokenbasierten Verfahren (Petrinetze) prinzipbedingt nicht auf, dafür weisen diese Verfahren aber andere Nachteile auf, die eine Implementierung einer näherungsweise tokenbasierten Semantik in ADEPT nicht erstrebenswert macht. Beispielsweise ist die Ausführungshistorie bei solchen Verfahren nicht mehr aus dem aktuellen Zustand des Graphen ableitbar, womit korrekte instanzspezifische Änderungen deutlich erschwert werden.

Vor diesem Hintergrund verwundert es nicht, dass eine lose Synchronisation mit Weiterlaufen der Slavezweige innerhalb der Workflow Patterns als eher unproblematisch gesehen wird – die Semantik wird über Petrinetze definiert und die Korrektheit des Datenflusses wird außen vor gelassen.

6. Flexible Endknoten


In diesem Kapitel wird die Implementierung solcher flexibler Endknoten in ADEPT untersucht. Dazu wird zunächst nochmals die Notwendigkeit flexibler Endknoten im Detail motiviert (Abschnitt 6.1), dann die sinnvollen Varianten flexibler Endknoten beschrieben (Abschnitt 6.2) und die Auswirkungen flexibler Endknoten auf den Kontroll- und Datenfluss untersucht (Abschnitt 6.3). Schließlich wird in Abschnitt 6.4 ein Zusammenfassung dieses Kapitels gegeben und ein Fazit gezogen.

6.1. Problembeschreibung

In vielen in der Praxis auftretenden Prozessen kann es vorkommen, dass die Ausführung des Prozesses vorhersehbar an mehreren Stellen vorzeitig beendet werden muss – sei es, weil sich in einem hypothetischen Beschwerdeprozess wie in Abbildung 6.1 der Grund für die Beschwerde erledigt hat oder sei es, weil ein Fehler aufgetreten ist, der eine weitere Ausführung des Prozesses unmöglich macht. Im letzteren Fall kann es bei Subprozessen sinnvoll sein, dem Elternprozess den Fehler auch als solchen zu signalisieren, um eine Fehlerbehandlung auf der höheren Ebene durchzuführen.

Sofern noch nebenläufige Aktivitäten ausgeführt werden, müssen diese durch den Start eines Endknotens abgebrochen werden, so dass der Prozess nach der Aktivierung des ersten Endknotens auch wirklich beendet ist – wie man es intuitiv erwartet.

Dieses lässt sich in vielen Fällen schon mit den bestehenden Mitteln modellieren, so kann man beim Beispiel aus Abbildung 6.1 statt der Verwendung eines flexiblen Endknoten auch wie in Abbildung 6.2 die weiteren Eskalationsstufen im nein-Zweig des Entscheidungsknoten Kunde...
zufrieden? unterbringen und mit einem leeren ja-Zweig direkt bis zum Ende des Prozesses springen.


Allerdings lässt sich am angegebenen Beispiel auch erkennen, dass eine solche Modellierung trotz theoretischer Machbarkeit nicht immer zu guten Modellen führt. So würden sich im Falle vieler Eskalationsstufen viele XOR-Blöcke quer über fast den gesamten Prozess erstrecken, was den Prozess sehr unübersichtlich und intuitiv schwer verständlich macht.


Im Kontext der Workflow Pattern können flexible Endknoten sowohl dem Pattern WCP-43 (Explicit Termination) als auch WCP-20 (Cancel Case) entsprechen. Der Unterschied zwi-
6.1. Problembeschreibung

schen beiden Pattern besteht darin, dass WCP-43 semantisch eine normale Prozessbeendigung beschreibt (und somit auch den gewöhnlichen, nicht flexiblen Endknoten umfasst), während WCP-20 semantisch einen Abbruch im Fehlerfall darstellt.

Im Kontext des Workflow Patterns WCP-43 wurde auch die Frage der Transformierbarkeit von Modellen mit flexiblen Endknoten in solche mit einem einzigen Endknoten gestellt und in [KHA03] diskutiert. Dort wurde der hier explizit geforderte Abbruch parallel laufender Aktivitäten allerdings ausdrücklich als nicht erwünscht betrachtet und dementsprechend nicht berücksichtigt, so dass es sich um einen etwas anderen Fall als hier handelt. In diesem abgewandelten Fall (ohne Abbruch) wurde festgestellt, dass auch Modelle mit Parallelität und mehreren flexiblen Endknoten immer mit einem einzigen Endknoten modellierbar sind, dies die Modelle unter Umständen aber stark kompliziert. Allerdings wurden in [KHA03] nur Workflowmodelle ohne strikte Blockstruktur betrachtet, so dass die dortigen Ergebnisse nicht direkt auf ADEPT übertragbar sind


Dies ist insbesondere dann häufig der Fall, wenn der Prozess in Rahmen einer planbaren Abweichung vorzeitig abgebrochen werden muss (vgl. auch [RBFD02]). Ohne flexible Endknoten muss zur Modellierung eines vorzeitigen Abbruchs die Struktur des Workflowgraphen stark modifiziert werden, so dass der normale Ablauf nur noch schwer aus der graphischen Darstellung ersichtlich ist. Mit flexiblen Endknoten kann in diesem Fall die Struktur des Prozesses den normalen Ablauf viel besser widerspiegeln, was für den Überblick und das intuitive Verständnis des Prozesses äußerst vorteilhaft ist.

Es kann allerdings auch das Gegenteil der Fall sein: Durch die übermäßige Verwendung von flexiblen Endknoten lassen sich deutlich leichter Prozesse erstellen, deren Semantik sich intuitiv nur noch schwer erschließt, analog zur übertriebenen Verwendung von goto oder return-Anweisungen in imperativen Programiersprachen.

Auch vor dem Hintergrund der im Rahmen der strukturierten Programmierung [DDH72] propagierten Philosophie des „Single Exit Point“ can die Einführung flexibler Endknoten in ADEPT daher durchaus auch kritisch gesehen werden, da durch die Einführung flexibler Endknoten die Einhaltung eines Single Exit Point in ADEPT nicht mehr länger erzwungen wird.


1In [KHA03] wurde zusammengefasst so argumentiert, dass man statt Verwendung mehrerer Endknoten immer einfach eine Kante zu einem einzigen Endknoten ziehen kann – in ADEPT ist dies aufgrund der Blockstrukturierung aber offensichtlich so nicht möglich.

2Die Einhaltung eines Single Exit Point bedeutet, dass jede Prozedur eines Programms nur einen einzigen Punkt hat, an dem sie verlassen wird – analog zur Verwendung nur eines einzigen Endknotens bei ADEPT.
6. Flexible Endknoten

Aus diesem Grund kann das erhöhte Risiko, unübersichtliche und damit schlechte Modelle zu erstellen, kein stichhaltiges Argument gegen die Einführung flexibler Endknoten sein, zumal flexible Endknoten wie dargestellt auch Chancen zur Verbesserung von Prozessmodellen eröffnen. Es muss also in die Zuständigkeit des Modellierers belassen werden, mit der durch flexible Endknoten erhöhten Flexibilität verantwortungsvoll umzugehen und diese zur Verbesserung der Modelle zu nutzen.

6.2. Umsetzungsvarianten

Wie in der Einleitung schon beschrieben wurde, ist es naheliegend, zwei Arten flexibler Endknoten anzubieten:
Gewöhnliche flexible Endknoten, um ein reguläres (d.h. keinen Fehler darstellendes) vorzeitiges Ende anzuzeigen und flexible Fehlerendknoten, mit denen ein vorzeitiges Ende im Fehlerfall modelliert werden kann.

Abbildung 6.4: a) Flexible Endknoten und b) Fehlerendknoten

6.2.1. Flexible Endknoten

Gewöhnliche flexible Endknoten sind in ihren Eigenschaften und ihrer Semantik so weit wie möglich äquivalent zum nicht flexiblen Endknoten. Das heißt, die einzigen Unterschiede (aus Benutzersicht) bestehen in der Flexibilität, den Knoten an jeder beliebigen Stelle im Prozess (und nicht nur an dessen Ende) einfügen zu können, und den sich daraus ergebenden Konsequenzen (parallel laufende Aktivitäten werden abgebrochen). Er stellt also aus Benutzersicht einfach eine Kopie des normalen Endknotens dar, die an beliebiger Stelle eingefügt werden kann, analog zur return-Anweisung in Java oder der exit-Aktivität in WS-BPEL 2.0.

Daraus folgt, dass es aus Sicht des Elternprozesses keinerlei Unterschied machen darf, ob der Subprozess über einen flexiblen Endknoten oder den nicht flexiblen Endknoten beendet wurde. Dies bedeutet im Fall eines Subprozesses, dass ein flexibler Endknoten genau wie der nicht flexiblen Endknoten die Ausgabeparameter des Subprozesses versorgen muss (der flexible Endknoten muss bei ADEPT also entsprechende Eingabeparameter aufweisen).

Innerhalb der Workflow Patterns entspricht dieser Typ somit zusammen mit dem nicht flexiblen Endknoten dem Pattern WCP-43 (Explicit Termination).

Wegen der weitgehenden Äquivalenz werden gewöhnliche flexible Endknoten nachfolgend wie in Abbildung 6.4 a) mit dem selben Symbol wie nicht flexible Endknoten dargestellt (also als ein in Fettschrift mit „Ende“ beschrifteter Knoten).
6.2. Umsetzungsvarianten

6.2.2. Fehlerendknoten

Fehlerendknoten dienen dazu (wie ihr Name schon nahelegt), einen vorhersehbaren Fehler im Ablauf eines Prozesses zu modellieren, der zu seiner vorzeitigen Beendigung führt (was in der Praxis vermutlich sogar der häufigste Grund für einen vorzeitigen Abbruch sein dürfte). Dies kann insbesondere innerhalb von Subprozessen sinnvoll sein, um eine Fehlerbehandlung auf höherer Ebene anzustoßen, wenn eine Fehlerbehandlung auf Ebene des Subprozesses keinen Sinn ergibt. Ein Fehlerendknoten kann daher in seiner Funktion auch mit der throw-Anweisung in Java bzw. der throw-Aktivität in WS-BPEL 2.0 verglichen werden.

Dementsprechend ist die Semantik eines Fehlerendknotens einfach die, dass der Prozess genau wie bei einem gewöhnlichen flexiblen Endknoten beendet wird, aber zusätzlich ein Fehlercode signalisiert wird.

Da im Fehlerfall nicht davon auszugehen ist, dass alle Ausgabeparameter eines Subprozesses vorliegen (die Fehlerursache könnte ja gerade die Unmöglichkeit sein, diese zu erzeugen), muss ein Fehlerendknoten auch keine dafür vorgesehenen Eingabeparameter aufweisen. Als kleine Erweiterung kann es aber dennoch sinnvoll sein, einen optionalen Eingabeparameter vorzusehen, über den der signalisierte Fehlercode zur Laufzeit gesetzt werden kann, um dynamisch genauere Fehlerinformationen geben zu können. Diese Erweiterung ist allerdings als rein optional zu betrachten und berührt die sonstige Funktionsweise des Knotens innerhalb des hier betrachteten Rahmens in keiner Weise.

Aufgrund seiner Semantik (nicht erfolgreiche Prozessbeendigung) lässt dieser Typ dem Workflow Pattern WCP-20 (Cancel Case) zuordnen.

Nachfolgend werden flexible Fehlerendknoten wie in Abbildung 6.4 b) als ein in kursiver Fettschrift mit „Abbruch“ beschrifteter Knoten dargestellt.

6.2.3. Diskussion

Da sich viele verschiedene Gründe für eine vorzeitige Prozessbeendigung angeben lassen und es für die Verständlichkeit ohne Zweifel vorteilhaft ist, auf Modellebene zwischen einem regulären Ende und einem fehlerbedingten Abbruch zu unterscheiden, müssen nach Möglichkeit beide Typen von flexiblen Endknoten angeboten werden.

Außer der verschiedenen Semantik beschränken sich die Unterschiede zwischen den beiden Typen aus Sicht des Prozesses, in den sie eingefügt werden, zudem im Wesentlichen darin, dass sie eine unterschiedliche Anzahl von Eingabeparametern aufweisen. Daher lassen sich die meisten der nachfolgenden Überlegungen (solange nicht explizit auf das Gegenteil hingewiesen wird) auf beide Fälle anwenden.
6. Flexible Endknoten

6.3. Existierendes Metamodell

6.3.1. Kontrollfluss

Die prinzipielle Kontrollflussessemantik eines flexiblen Endknotens lässt sich mit einem Satz beschreiben: Sobald der flexible Endknoten gestartet wird, muss die Ausführung der Prozessinstanz beendet und sämtliche laufenden Aktivitäten innerhalb dieser Instanz abgebrochen werden.

Im Fall eines normalen flexiblen Endknotens müssen dabei noch ggf. die Ausgabeparameter des Subprozesses mit den Werten der Eingabeparameter des flexiblen Endknotens versorgt werden bzw. im Fall eines Fehlerendknotens muss der gewünschte Fehler signalisiert werden.


Beide soeben beschriebenen Semantiken stellen technisch gesehen keine größere Herausforderung dar (bzw. sind in den vorhandenen Schnittstellen teilweise sogar schon vorgesehen), so dass an dieser Stelle auf eine detailliertere Darstellung verzichtet werden kann.

Ein flexibler Endknoten ist aus theoretischer Sicht ein anderer Knotentyp als der gewöhnliche, nicht flexible Endknoten – auch wenn die Unterschiede aus Benutzersicht minimal sind. So hat der nicht flexible Endknoten die Ausgangssemantik NONE, während ein flexibler Endknoten die Ein- und Ausgangssemantik ONE_Of_ONE besitzt. Bezüglich der Strukturierungsregeln eines Graphen („KF-Regeln") verhält sich ein flexibler Endknoten daher wie ein gewöhnlicher Aktivitätenknoten und nicht wie ein Endknoten.

Im Laufzeitverhalten besteht allerdings der bedeutende Unterschied, dass eine von einem flexiblen Endknoten ausgehende Kontrollflusskante niemals mit true signalisiert wird. Somit gilt, dass direkte (Kontrollfluss-) Nachfolgeknoten eines flexiblen Endknotens niemals aktiviert und ausgeführt werden bzw. „tot“ sind, sofern sie die Eingangssemantik ONE_Of.ONE oder ALL.Of.ALL aufweisen (also insbesondere keine XOR-Joinknoten sind).

Bezüglich XOR-Joinknoten (Eingangssemantik ONE.Of.ALL) gilt davon abweichend die Regel, dass ein XOR-Joinknoten nur dann tot ist, wenn alle seine Vorgänger tot oder flexiblen Endknoten sind (da eine true-signalisierte Kante zu seiner Aktivierung ausreicht).

Eine weitere Abweichung ergibt sich, wenn der Nachfolger ein AND-Join mit partiellen Abbruch ist: Ein AND-Join mit partiellem Abbruch ist nur dann tot, wenn das Ende von mindestens einem seiner Masterzweige tot ist. Bei einem toten Slavezweig ist hingegen nicht si-
Existerendes Metamodell

6.3. Existierendes Metamodell


Die soeben festgestellten Regeln gelten transitiv natürlich auch entsprechend für die direkten Nachfolger toter Knoten, die Nachfolger der Nachfolger, usw.


Abbildung 6.5: Beispiel für tote Knoten


Sofern der flexible Endknoten wie in Abbildung 6.7 ein direkter Nachfolger eines AND-Splitknotens (auch mit partielen Abbruch) ist und keine eingehenden Sync-Kanten vorhanden sind, können alle anderen Zweige dieses AND-Blocks als tot betrachtet werden, da sie praktisch zeitgleich mit ihrem Start wieder abgebrochen werden.

³Vergleiche hier auch das OR-Splitkriterium in [Rei00]: Wenn der Nachfolger nach diesem Kriterium vom flexiblen Endknoten geschriebene Daten obligat lesen darf, ist er hier tot.
Auch in diesen beiden Fällen gibt es natürlich wieder eine beschriebene Transitivität bezüglich der Nachfolger von toten Knoten (die Regeln gelten also auch für tote Knoten und nicht nur flexible Endknoten).

Da es durch diese Transitivität passieren kann, dass (vom Modellierer) unerwartet große Teile des Prozesses unerreichbar werden, ist es erstrebenswert, zur Entwurfszeit eine Erkennung solcher toter Knoten durchzuführen und dem Benutzer einen Fehler zu melden, sofern einem toten Knoten eine Aktivität zugewiesen wurde. Tote Nullaktivitäten oder ein toter nicht flexibler Endknoten sind allerdings aus syntaktischen Gründen oft unvermeidbar und müssen daher zugelassen werden.


Zusammengefasst sind dabei die folgenden Markierungsregeln anzuwenden:

Wenn der Knoten $n$ tot oder ein flexibler Endknoten ist, dann sind (auch) die folgenden Knoten tot:

1. Alle Kontrollflussnachfolger $n_{\text{succ}}$, wenn eine der folgenden Bedingungen zutrifft:
   a) $n_{\text{succ}}$ hat die Eingangssemantik $\text{ONE}_0\text{f}_\text{ONE}$ oder $\text{ALL}_0\text{f}_\text{ALL}$ und ist kein AND-Join mit partiellem Abbruch
   b) $n_{\text{succ}}$ hat die Eingangssemantik $\text{ONE}_0\text{f}_\text{ALL}$ und alle seine Vorgänger sind tot oder flexible Endknoten
   c) $n_{\text{succ}}$ ist ein AND-Join mit partiellem Abbruch und $n$ befindet sich in einem Masterzweig
2. Alle Nachfolger bezüglich Sync-Kanten, wenn sie sich im selben Zweig der innersten, \( n \) umgebenden XOR-Verzweigung befinden bzw. es keine solche XOR-Verzweigung gibt.

3. Wenn \( n \) direkter Nachfolger eines AND-Splitknotens ist und keine eingehenden Sync-Kanten aufweist: Alle anderen Knoten innerhalb des AND-Blocks.

### 6.3.2. Datenfluss


Auf der anderen Seite gilt, dass Datenelemente, die nur innerhalb eines solchen Zweigs versorgt sind, hinter dem XOR-Join obligat gelesen werden dürfen, da der Kontrollfluss an den Leseknoten nur ankommt, wenn der Zweig mit dem flexiblen Endknoten nicht genommen wurde.

In der Summe bedeutet dies, dass ein Zweig, der einen flexiblen Endknoten enthält, am XOR-Joinknoten bezüglich des Datenusses vollständig ignoriert werden kann.

Daraus folgt, dass der obligate Lesezugriff auf \( d \) an Knoten \( J \) in Abbildung 6.5 zulässig ist, obwohl \( d \) nur an \( D \) im oberen Zweig des XOR-Splits geschrieben wird. Der Grund hierfür ist, dass \( J \) (und damit auch \( J )\) nie zur Ausführung kommt, sofern ein anderer als der obere Zweig gewählt wird. Daher ist \( d \) in allen Fällen, in denen \( J \) bzw. \( J \) ausgeführt werden können, sicher versorgt.

Befindet sich der flexible Endknoten in einem Slavezweig eines AND-Join mit partiellem Abbruch, so können seine Vorgänger ebenfalls keine Werte schreiben, die hinter dem AND-Join obligat lesbar sind. Da dies aber ohnehin schon aufgrund der Semantik eines Slavezweigs der Fall ist, müssen flexible Endknoten hier nicht gesondert berücksichtigt werden.

Sollte sich der flexible Endknoten nicht innerhalb eines XOR-Blocks oder eines Slavezweigs befinden, sind ohnehin alle Kontrollflussnachfolger des Knotens tot und können daher bezüglich des Datenflusses ignoriert werden.

Diese Feststellungen können auf einfache Weise in einer Datenflussanalyse wie im WriterE-xists-Algorithmus aus [Rei00] berücksichtigt werden, wie in Anhang B.3 beschrieben ist.

**Abbildung 6.8:** Modellierung einer finalen Auswahl mit flexiblen Endknoten

**Abbildung 6.9:** Modellierung eines AND-Join mit partiellem Abbruch mit flexiblen Endknoten
6.4. Zusammenfassung


6.4. Zusammenfassung

Flexible Endknoten erhöhen die Flexibilität in der Modellierung beträchtlich und können die Qualität eines Modells verbessern. Bei unbedachten Einsatz kann allerdings auch das Gegenteil der Fall sein: Wenn zu viele Stellen modelliert werden, an denen ein Prozessabbruch stattfindet, können Modelle durch den Einsatz flexibler Endknoten leicht unnötig kompliziert und schwer verständlich werden.

Da flexible Endknoten die Ausdruckssteifigkeit des Metamodells aber echt erhöhen, spricht dies nicht gegen die Einführung flexibler Endknoten und muss in der Verantwortung des Modelliers belassen werden.\footnote{Zudem gibt es auch schon ohne flexible Endknoten viele Möglichkeiten, ein Modell durch ungeschickte Modellierung unübersichtlich zu gestalten.}

Die Herausforderung bei der Implementierung von flexiblen Endknoten steckt weniger in der genauen Festlegung der Semantik – diese ist relativ eindeutig – oder der technischen Umsetzung der Knoten selbst, sondern in der Tatsache, dass es tote Knoten geben kann und den sich daraus ergebenden Konsequenzen auf den Kontroll- und vor allem den Datenfluss. Letztlich sind diese aber auch gut beherrschbar: Tote Knoten können durch gängige Algorithmen
(Dead Path Elimination) bestimmt werden und die sich daraus ergebenden Modifikationen der Datenflußanalyse sind naheliegend und unkompiliert.


<table>
<thead>
<tr>
<th>Pattern</th>
<th>Anmerkungen</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCP-43 (Explicit Termination)</td>
<td>* Entspricht gewöhnlichen flexiblen Endknoten</td>
</tr>
<tr>
<td></td>
<td>* Pattern umfasst auch den nicht flexiblen Endknoten</td>
</tr>
<tr>
<td></td>
<td>* Kontextbedingungen bei weichem Abbruch vollständig erfüllt</td>
</tr>
<tr>
<td>WCP-20 (Cancel Case)</td>
<td>* Entspricht Fehlerendknoten</td>
</tr>
<tr>
<td></td>
<td>* Kontextbedingungen bei weichem Abbruch vollständig erfüllt</td>
</tr>
</tbody>
</table>

**Tabelle 6.1: Unterstützte Workflow Patterns**
7. Erweiterungen bedingter Verzweigungen

Bedingten Verzweigungen kommt im Rahmen der meisten Geschäftsprozesse eine zentrale Bedeutung zu. Erst durch den Einsatz bedingter Verzweigungen kann sich der Prozessablauf dynamisch an verschiedene vorhersehbare Situationen anpassen, ohne dass ein Benutzereingriff in Form instanzspezifischer Änderungen erforderlich ist.

Bislang ist nur eine Form der bedingten Verzweigung verfügbar, die explizite XOR-Entscheidung. Obwohl diese Semantik zur Modellierung der meisten praktisch auftretenden Prozessabläufe ausreicht, gibt es auch viele Abläufe, die sich mit einer expliziten XOR-Entscheidung nur indirekt oder unter Einsatz von manuell zu modellierenden Behelfslösungen realisieren lassen. Eine Unterstützung erweiterter Semantiken für bedingte Verzweigungen kann in diesen Fällen Abhilfe schaffen und die Modellierung deutlich erleichtern. Aufgrund dessen werden in diesem Kapitel drei praxisrelevante Erweiterungen verschiedener Aspekte von bedingten Verzweigungen untersucht:

Zunächst eine detaillierte Datenflussanalyse, die Abhängigkeitsbeziehungen zwischen verschiedenen Entscheidungen berücksichtigt (abhängige XOR-Entscheidungen in Abschnitt 7.1), und somit die Modellierung des Datenflusses bei Vorhandensein mehrerer ähnlicher bedingter Verzweigungen erleichtert.

Danach wird die Realisierung einer impliziten Entscheidungssemantik für bedingte Verzweigung (verzögerte XOR-Entscheidungen in Abschnitt 7.2) untersucht. Dadurch wird es möglich, dass Entscheidungen implizit durch den Start der gewünschten Aktivität getroffen werden können, die Entscheidungsfindung also nicht mehr in jedem Fall explizit durch eine Entscheidungsaktivität erfolgen muss.

Zum Abschluss dieses Kapitels wird die Realisierung einer bedingten Mehrfachauswahl untersucht (ODER-Verzweigungen in Abschnitt 7.3). Damit können an einer bedingten Verzweigung nun auch wahlweise mehrere Zweige zur Ausführung ausgewählt werden und nicht nur wie bislang lediglich ein einziger.

7.1. Abhängige XOR-Entscheidungen

Bei in der Praxis vorkommenden Prozessen lässt es sich oft beobachten, dass wie in Abbildung 7.1 mehrere XOR-Verzweigungen vorkommen, die in jedem Fall zur Laufzeit die selbe Entscheidung treffen (sich also nicht unabhängig voneinander entscheiden), und deshalb auch als „abhängige XOR-Entscheidungen“ bezeichnet werden können. Solche Abhängigkeitsbeziehungen werden derzeit bei der Datenflussanalyse noch nicht berücksichtigt. Bei Berücksichtigung
von Abhängigkeitsbeziehungen lässt sich häufig folgern, dass ein bislang als unsicher abgeleichter Datenfluss aufgrund der sich ergebenden Kontrollflussabhängigkeiten eigentlich sicher ist (so auch im angegebenen Beispiel).


7.1.1. Problembeschreibung

Abbildung 7.1: Abhängige XOR-Entscheidungen

In Abbildung 7.1 erkennt ein menschlicher Betrachter leicht, dass die Versorgtheit der beiden Lesezüge an den Knoten *Lastschrift erstellen* und *Kreditkartenz. buchen* eigentlich sichergestellt ist, da alle Lesezüge im zweiten XOR-Block im gleichen Zweig stattfinden, innerhalb dessen das entsprechende Datenelement im ersten XOR-Block geschrieben wurde. Beispielsweise wird *Lastschrift erstellen* nur ausgeführt, wenn die Zahlungsart *Lastschrift* gewählt wurde. In diesem Fall wurde vorher aber immer auch *Kontodaten erfassen* ausgeführt, womit das Datenelement *Kontodaten* an *Lastschrift erstellen* sicher versorgt ist. ADEPT bemängelt *Kontodaten* aber dennoch als nicht sicher versorgt, da es bislang alle XOR-Entscheidungen als voneinander unabhängig betrachtet.

Durch diese Unabhängigkeitsannahme folgt zwangsläufig, dass (wie im bisherigen *WriterExists*-Algorithmus implementiert) ein an einem bestimmten Knoten, z.B. *Liefern*, nicht sicher versorgtes Datenelement auch an allen Nachfolgern dieses Knotens (also auch im zweiten XOR-Block) nicht sicher versorgt sein kann - sofern kein weiterer Schreibzugriff erfolgt. Wie dieses Beispiel zeigt, kann diese Annahme aber nicht mehr in jedem Fall aufrechterhalten werden, wenn mehrere XOR-Entscheidungen voneinander abhängig sind und man diese Abhängigkeiten berücksichtigt.
7.1. Abhängige XOR-Entscheidungen


Abbildung 7.2: Der transformierte Prozess aus Abbildung 7.1

Wie man aber schon an diesem einfachen Beispiel erkennen kann, wird der Prozessgraph dadurch um einiges komplexer und intuitiv deutlich schlechter verständlich. Zudem sind die nötigen Modifikationen bei komplexeren und damit realistischeren Prozessen oft nicht gerade einfach, so dass die meisten Benutzer höchstwahrscheinlich damit überfordert sind, diese manuell durchzuführen (man denke dabei z.B. an einem Prozess mit mehreren geschachtelten abhängigen XOR-Entscheidungen, wie er später in diesem Abschnitt noch vorkommen wird).

Das heißt, man muss in einem benutzerfreundlichen Editor zu diesem Zweck eine automatische Funktion anbieten, die den Graphen entsprechend transformiert. Um eine automatische Transformation des Graphen vornehmen zu können, müssen abhängige XOR-Entscheidungen erkannt und die Kontrollflussabhängigkeiten zwischen den Zweigen festgestellt werden. Danach kann der Graph entsprechend transformiert werden.

Wenn man aber die Kontrollflussabhängigkeiten zwischen den Zweigen ohnehin feststellen muss, kann man diese Information auch direkt bei der Datenflussanalyse berücksichtigen und sich somit die Graphtransformation und die damit verbundene schwerere Verständlichkeit des Graphen ersparen. Zudem ist es gerade bei geschachtelten abhängigen XOR-Entscheidungen unklar, ob eine solche Transformation in jedem denkbaren Fall gelingt oder ob sich Gegenbeispiele angeben lassen, bei denen keine semantikerhaltende Transformation möglich ist.

Aus diesem Grund wird eine solche Transformation in dieser Arbeit gar nicht näher betrachtet, sondern es wird direkt untersucht, welche Erweiterungen des bestehenden WriterExists-
Algorithmus aus [Rei00] nötig sind, um abhängige XOR-Entscheidungen zu erkennen und die Datenflussbeschränkungen in diesem Fall entsprechend zu lockern.

7.1.2. Umsetzungsvarianten

Bevor der Algorithmus aber beschrieben wird, wird in diesem Abschnitt auf abstrakter Ebene untersucht, welche Möglichkeiten es gibt, die Abhängigkeit von XOR-Entscheidungen zueinander zu erkennen (Unterabschnitt 7.1.2.1) und, wie solche Abhängigkeiten in einer Datenflussanalyse berücksichtigt werden können (Unterabschnitt 7.1.2.2). Nach der jeweiligen Diskussion wird in jedem Unterabschnitt eine der Möglichkeiten für die Implementierung ausgewählt.

7.1.2.1. Erkennen der Abhängigkeit

Die erste größere Herausforderung bei der Entwicklung eines Algorithmus zur Berücksichtigung abhängiger XOR-Entscheidungen ist, auf welche Weise überhaupt erkannt werden kann, dass mehrere XOR-Entscheidungen voneinander abhängig bzw. aneinander gekoppelt sind, d.h. wie das System zur Entwurfszeit feststellen kann, dass zwei XOR-Entscheidungen zur Laufzeit in jedem Fall die gleiche Entscheidung treffen.

Da möglichst allgemeine Entscheidungsaktivitäten unterstützt werden müssen, fällt ein Vergleich der inneren Logik der Entscheidungsaktivitäten als Möglichkeit, eine Abhängigkeit festzustellen, aus:
Zum Einen ist die innere Entscheidungslogik einer Aktivität im allgemeinen Fall unbekannt. Zum Anderen ist eine solche vergleichende Analyse, selbst wenn die Entscheidungslogik bekannt wäre, im Allgemeinen unmöglich. Im allgemeinist möglichen Fall (z.B. zwei beliebige (Java-)Programme) läuft ein solcher Vergleich nämlich letztlich darauf hinaus, dass das bewesermaßige unentscheidbare Problem der funktionalen Äquivalenz zweier Turing-Maschinen [HU90] gelöst werden muss.


Eine naheliegende Möglichkeit, solche Abhängigkeitsbeziehungen explizit zu modellieren, ist es, einen speziellen Entscheidungsaktivitätstyp zur Verfügung zu stellen, der einen Eingabe-parameter besitzt, der aus einem Datenelement den Wert des Auswahlparameters einer früheren Entscheidungsaktivität liest. Der über diesen Eingabeparameter gelesene Wert wird dann einfach an den Auswahlparameter der Aktivität kopiert. Eine solche Entscheidungsaktivität wird nachfolgend auch kurz als „Koppelaktivität“ bezeichnet, das referenzierte Datenelement als „Koppeldatenelement“.

7.1. Abhängige XOR-Entscheidungen

Aus der beschriebenen Semantik folgt, dass die das Koppeldatenelement schreibende allgemeine Entscheidungsaktivität und alle nachfolgenden Koppelaktivitäten, die dieses Datenelement (bis zum strukturell nächsten Schreizzugriff) lesen, voneinander abhängig sind. Dies kann im Rahmen der Datenflussanalyse leicht festgestellt werden. Dabei ist zu beachten, dass es für manche Anwendungen sinnvoll sein kann, wenn die schreibende Entscheidungsaktivität nicht zwingend einem XOR-Splitknoten zugewiesen werden muss, sondern einem beliebigen Knoten zugewiesen sein kann.\footnote{Ein Beispiel hierfür findet sich später in diesem Kapitel bei den ODER-Verzweigungen (Abschnitt 7.3)}

Die ein Koppeldatenelement schreibende Entscheidungsaktivität, also diejenige Entscheidungsaktivität, die die eigentliche Entscheidung für eine Gruppe abhängiger Entscheidungen trifft, wird nachfolgend auch als „bestimmende Entscheidungsaktivität“ bezeichnet.

Abbildung 7.3: Beispiel für die Abhängigkeit von Entscheidungen

Ein Beispiel für eine auf diese Weise hergestellte Abhängigkeitsbeziehung findet sich in Abbildung 7.3. In diesem Beispiel sind die Knoten A, B, C voneinander abhängig und genauso die Knoten D, E; C und E beispielsweise aber nicht, da das Datenelement decision am dazwischen liegenden Knoten D neu geschrieben wurde. Dementsprechend sind A und C bestimmende Entscheidungsaktivitäten, während B, C und E Koppelaktivitäten zugewiesen wurden.


7.1.2.2. Analyse des Datenflusses

Lesezugriff ist dann sicher versorgt, wenn beim Vergleich festgestellt wird, dass die Vorbedingung für die Versorgung (wann wird das Datenelement geschrieben?) immer gilt, wenn auch die Vorbedingung für die Knotenausführung (wann wird das Datenelement gelesen?) erfüllt ist.

Abbildung 7.4: Beispiel für eine Datenflussanalyse mit Vorbedingungen

Ein Beispiel für eine solche Analyse ist in Abbildung 7.4 zu sehen. Dort wird überprüft, ob der Lesezugriff auf das Datenelement **Kontodaten** am Knoten **Letzte Schrift erstellen** zulässig ist. Dieses Datenelement wird im vorderen XOR-Block geschrieben, sofern **Zahlungsart = Letzte Schrift** gilt. Unter dieser Vorbedingung ist es daher sicher versorgt. Im hinteren XOR-Block wird der Knoten **Letzte Schrift erstellen** nur dann ausgeführt, wenn auch **Zahlungsart = Letzte Schrift** gilt. Daher ist die Vorbedingung für die sichere Versorgung von **Kontodaten** dort immer erfüllt und der Lesezugriff somit zulässig.

Selbstverständlich lässt sich dieses Verfahren auch bei komplexeren Graphen mit mehreren abhängigen XOR-Entscheidungen anwenden (wobei sich natürlich u.U. deutlich komplizierere Vorbedingungen ergeben). Falls sich ein Knoten innerhalb mehrerer XOR-Verzweigungen befindet, müssen zur Ausführung des Knotens die Bedingungen aller Zweige auf dem Weg zu diesem Knoten erfüllt sein. Die Vorbedingung ergibt sich also in diesem Fall aus einer Und-Verknüpfung (Konjunktion) der Bedingungen jener XOR-Zweige, die zu dem Knoten führen.

Für die sichere Versorgung eines Datenelements ist es ausreichend, wenn es an einem Knoten obligat geschrieben wird, also wenn mindestens einer der „Schreiberknoten“ ausgeführt wird. Am Leseknoten muss also lediglich mindestens eine der Vorbedingungen der Schreiberknoten erfüllt sein. Die Vorbedingung zur Versorgung eines Datenelements ergibt sich daher aus der Oder-Verknüpfung (Disjunktion) der Vorbedingungen aller Knoten, an denen das betreffende Datenelement geschrieben wird.

**Durch Elimination unausführbarer Pfade** Eine andere, auf den ersten Blick durchaus attraktive Möglichkeit ist es, diejenigen Pfade im Graphen zu bestimmen, die unter Garantie niemals ausgeführt wurden, wenn der Kontrollfluss an dem Knoten angekommen ist, der den zu untersuchenden Lesezugriff durchführt. Diese „unausführbaren Pfade“ (engl. „infeasible paths“) werden
Abhängige XOR-Entscheidungen

dann aus dem Graphen entfernt. Anschließend kann die unmodifizierte Datenflussanalyse auf den Restgraphen angewandt werden.

Die Idee hinter dieser Vorgehensweise stammt aus dem Kontext von White Box Tests in der Softwaretechnik und ist in [BGS97] beschrieben. Im Kontext des Softwaretests ist das Ziel, möglichst viele mögliche Ausführungspfade durch entsprechende Testfälle abzudecken. Es gibt aber immer einige Pfade im Programm, die aufgrund der Programmlogik niemals genommen werden können. Durch die Erkennung unausführbarer Pfade anhand der statischen Struktur des Programms können daher einige solcher Fälle von vornherein ausgeschlossen werden, was die Arbeit des Testers deutlich erleichtert.

![Abbildung 7.5: Datenflussanalyse mit Elimination unausführbarer Pfade](attachment:diagram1.png)


![Abbildung 7.6: Problematischer Prozess bei Elimination unausführbarer Pfade](attachment:diagram2.png)

Auch wenn diese Variante auf den ersten Blick recht elegant wirkt, lassen sich leicht Gegenbeispiele angeben, bei denen sie nicht mehr korrekt funktioniert. Ein solches Gegenbeispiel ist in
Abbildung 7.6 zu sehen.

Im dargestellten Fall kann bei der Analyse des Leseszugs des Knotens X auf das Datenelement d kein einziger Pfad an den XOR-Blöcken D1 und D2 ausgeschlossen werden. Da X ausgeführt wird, wenn A ∨ C gilt, kann an D1 sowohl der obere Zweig (mit der Bedingung A ∨ B) als auch der untere Zweig (mit der Bedingung C) genommen worden sein; d wird aber nur im oberen Zweig geschrieben. Entsprechendes gilt an D2.

Demnach kann der Graph nicht reduziert werden (außer ggf. um den unteren Zweig von D3, dieser ist hier aber nicht relevant). Der bisherige Datenflussanalysealgorithmus würde bei Anwendung auf den Graphen daher feststellen, dass d an X nicht sicher versorgt sei.

Bestimmt man allerdings die Vorbedingungen, so stellt man fest, dass d an X eigentlich sicher versorgt ist: X wird ausgeführt, wenn A oder C gilt. Wenn A gilt, wurde aber vorher der obere Zweig an D1 ausgeführt und d ist versorgt. Wenn C gilt, wurde der untere Zweig an D2 ausgeführt und d ist ebenfalls versorgt. d wird also in allen Fällen, in denen X ausgeführt werden kann, versorgt.

Oder formaler: d ist versorgt, wenn (A ∨ B) ∨ (B ∨ C) = A ∨ B ∨ C gilt. Da X aber nur ausgeführt wird, wenn A ∨ C gilt, ist d dort offensichtlich sicher versorgt.

Daher ist das Ergebnis dieser Variante der Datenflussanalyse in diesem Beispiel nicht korrekt (bzw. stellt keine Verbesserung gegenüber der bisherigen dar).

Zudem ist der Aufwand für diese Variante in der Praxis kaum geringer als bei der Bestimmung der Vorbedingungen. Um die zu streichenden Zweige an den XOR-Verzweigungen herausfinden, muss auch hier die Vorbedingung für den untersuchten Knoten bestimmt werden und mit den Ausführungsbedingungen der möglicherweise zu streichenden XOR-Zweige verglichen werden. Wie wir später in Abschnitt 7.1.3.2 noch feststellen werden, hat ein solcher Vergleich zweier Vorbedingungen im Allgemeinen eine exponentielle Komplexität.

Daher stellt die Elimination ausführbarer Pfade keine deutliche Verbesserung gegenüber der Bestimmung der Vorbedingungen dar – es müssen lediglich einfachere, aber dafür deutlich mehr Bedingungen miteinander verglichen werden.

Aus den genannten Gründen wird im Folgenden die zuerst vorgestellte Variante mit Bestimmung der Vorbedingung zur Analyse des Datenflusses verwendet.

7.1.3. Existierendes Metamodell

Im Folgenden wird gegenüber dem Metamodell aus [Rei00] eine kleine Erweiterung aus dem implementierten ADEPT2-System übernommen.

Während in [Rei00] der Entscheidungsparameter einer Entscheidungsaktivität direkt die zu aktivierende Kante angibt, existiert im ADEPT2-System eine Indirektion bzw. ein „Mapping“ zwischen dem Wert des Entscheidungsparameters und den verfügbaren Kanten.

Das heißt, dass der von der Entscheidungsaktivität geschriebene Entscheidungsparameter nicht direkt die zu selektierenden Kante, den sog. „Edge Code“, angibt, sondern eine abstrakte Nummer, die sog. „Decision ID“, die eine der Entscheidungsmöglichkeiten symbolisiert. Der Modellierer muss dann beim Zuweisen der Aktivitätenvorlage jedem Edge Code ein oder mehrere
7.1. Abhängige XOR-Entscheidungen

Decision IDs zuordnen. Zusätzlich hierzu ist dem System die Menge der möglichen Decision IDs bekannt, so dass sichergestellt werden kann, dass jeder Decision ID genau eine Kante zugewiesen wurde. Somit können zur Laufzeit keine nicht zugeordneten Decision IDs auftreten und es entfällt die Notwendigkeit eines Defaultzweigs. Umgekehrt gibt es so auch keine nie ausgeführten Zweige (solange man davon ausgeht, dass die von der Entscheidungsaktivität angegebene Menge der möglichen Decision IDs auch korrekt ist).

Dies bedeutet insbesondere, dass eine Entscheidungsaktivität problemlos mehr Decision IDs besitzen kann, als der zugeordnete XOR-Split Zweige hat.

Diese Eigenschaft ist im Kontext abhängiger XOR-Entscheidungen besonders nützlich (aber auch keine zwingend notwendige Voraussetzung): Durch diese Indirektion ist es möglich, dass strukturell unterschiedliche XOR-Entscheidungen (also XOR-Entscheidungen mit verschiedenen vielen Zweigen) voneinander abhängig sein können.

Alle voneinander abhängigen XOR-Entscheidungen geben zur Laufzeit notwendigerweise den selben Entscheidungswert aus\(^2\). Bildet man diese eins zu eins auf die Kanten ab, so sind verschieden viele Zweige bei abhängigen Entscheidungen nur unter Verwendung von Defaultzweigen realisierbar, was für viele Anwendungen aber zu ungenau ist, da alle nicht anderweitig zugeordneten Werte den Defaultzweig aktivieren.

Mit dieser Indirektion ist dies hingegen kein Problem, da das Mapping der Decision IDs auf die Edge Codes an jedem XOR-Splitknoten (trotz identischer Decision IDs) unterschiedlich sein kann. Wenn es mehr IDs als Kanten gibt, müssen die überschüssigen IDs nicht alle einer Kante zugewiesen werden, sondern verschiedene Gruppen von IDs können verschiedenen Kanten zugeordnet werden. Der Modellierer muss nur darauf achten, dass die Entscheidungsaktivität zwischen genügend vielen Decision IDs unterscheidet, um bei der maximal gewünschten Zweiganzahl jedem Zweig mindestens eine ID zuordnen zu können, da das Mapping ansonsten vom System als ungültig bemängelt wird. Dies ergibt sich jedoch in den meisten Fällen schon von selbst aus der Semantik der Entscheidung, so dass diese Einschränkung vermutlich meist gar nicht als solche wahrgenommen wird (es ist für einen Modellierer offensichtlich, dass er eine Entscheidung zwischen vier Fällen nicht zum Split in fünf Zweige verwenden kann).

7.1.3.1. Kontrollfluss

Da die Abhängigkeiten durch spezielle Koppelaktivitäten modelliert werden, ergeben sich die Kontrollflussauswirkungen abhängiger XOR-Entscheidungen aus der Semantik dieser Aktivitäten.


\(^2\)Bei der hier verwendeten Definition der Abhängigkeit.
Bei einer Implementierung ist allerdings das technische Problem zu lösen, wie die Menge der möglichen Decision IDs zur Entwurfszeit von der bestimmenden Aktivität zu den einzelnen Koppelaktivitäten transportiert werden kann, um das Mapping der Decision IDs auf die Edge Codes korrekt durchführen zu können.

Das heißt, für jede Koppelaktivität muss die bestimmende Entscheidungsaktivität herausgefunden werden und dann die Menge der Decision IDs von dort übernommen werden. Außerdem müssen Koppelaktivitäten über Änderungen an der bestimmenden Aktivität bzw. über eine Ersetzung dieser Aktivität informiert werden können. Wie dies am effizientesten geschieht, ist allerdings stark implementierungsabhängig und kann hier daher nicht abschließend geklärt werden (denkbar wäre beispielsweise eine Implementierung über Enumerations, die die Menge der Decision IDs als Wertebereich erhalten).

Ein weiteres technisches Problem ist, wie das System erkennen kann, dass einem Knoten eine Koppelaktivität zugewiesen wurde. Da das ADEPT2-System die Definition und Abfrage von Aktivitätstypen erlaubt, ist dies technisch trivial lösbar (man definiere einfach einen eigenen Aktivitätstyp für Koppelaktivitäten). Im Folgenden wird davon allerdings abstrahiert und davon ausgegangen, dass es eine Funktion in der Form isCoupledDecision(node) gibt, mit der dies für einen gegebenen Knoten effizient (in $O(1)$) festgestellt werden kann.

### 7.1.3.2. Datenfluss


Die Idee hinter diesem Algorithmus versteht man am besten, wenn man sich nochmals überlegt, unter welcher Voraussetzung ein Lesezugriff eigentlich als sicher versorgt gilt: Ein Lesezugriff auf ein Datenelement $d$ an einen Knoten $n$ ist dann sicher versorgt, wenn $d$ bei jedem möglichen zur Laufzeit auftretenden Kontrollflussablauf, in dem $n$ zur Ausführung kommt, von mindestens einem Vorgängerknoten geschrieben wurde.


Der nachfolgend vorgestellte Algorithmus macht diese Kontrollflusseinschränkungen durch die Bestimmung der Vorbedingungen greifbar und verwendet sie, um die Datenflussanalyse im Fall

Informelle Beschreibung des erweiterten **WriterExists-Algorithmus** Der vorgestellte Algorithmus stellt eine Erweiterung des bestehenden **WriterExists-Algorithmus** dar und besitzt daher wie dieser zwei Eingabeparameter (der Kontrollflussgraph und das Datenflusschema werden hier als implizit gegeben vorausgesetzt):

1. \(n_{\text{target}}\): Der Knoten, der den obligaten Lesezugriff durchführt
2. \(d_{\text{target}}\): Das Datenelement, das an \(n_{\text{target}}\) obligat gelesen wird.

Als erstes wird zur Datenflusssanalyse der bisherige **WriterExists-Algorithmus** ausgeführt. Sofern dieser das Ergebnis \(\text{supplied}(n_{\text{target}}) = \text{true}\) ergibt, ist der Algorithmus fertig (\(d_{\text{target}}\) ist auf jeden Fall sicher versorgt).

Ansonsten wird die nachfolgend beschriebene verfeinerte Analyse durchgeführt. Die verfeinerte Analyse wird unabhängig davon durchgeführt, ob sich \(n_{\text{target}}\) innerhalb einer abhängigen Verzweigung befindet, da es auch Fälle wie in Abbildung 7.7 geben kann, in denen eine Datenelement aufgrund abhängiger XOR-Verzweigungen versorgt ist, obwohl sich der lesende Knoten nicht innerhalb einer solchen befindet.

![Abbildung 7.7: Der Lesezugriff an Knoten J ist sicher versorgt](image)

Das Ziel des Algorithmus ist zunächst, sowohl die Vorbedingung, wann \(d_{\text{target}}\) sicher versorgt wurde, als auch die Vorbedingung, wann \(n_{\text{target}}\) zur Ausführung kommt, zu bestimmen. Die Vorbedingung für \(d_{\text{target}}\) ergibt sich dabei aus der Disjunktion der Vorbedingungen für die Ausführung aller Knoten, die \(d_{\text{target}}\) obligat schreiben\(^3\).

Daher muss zur Bestimmung beider Werte zunächst nur das Problem gelöst werden, wie die Vorbedingung für die Ausführung eines Knotens effizient bestimmt werden kann.

Eine Vorbedingung eines Knotens \(n\) hat dabei die Form \(d_1 \in \{a, b, c\} \land d_2 \in \{x, y\} \land \ldots\) mit der Bedeutung „der Knoten \(n\) wird ausgeführt, wenn \(d_1\) zur Laufzeit den Wert \(a, b\) oder \(c\) hat,\(^3\)

---

\(^3\)Es reicht für die sichere Versorgung aus, wenn das Datenelement einmal geschrieben wird, also wenn ein beliebiger Schreiberknoten ausgeführt wird.
und, wenn \( d_2 \) den Wert \( x \) oder \( y \) hat und, wenn . . . “.

Da die Vorbedingung für \( d_{target} \) eine Oder-Verknüpfung mehrerer solcher Knotenvorbedingungen darstellt, wird eine einzelne Knotenvorbedingung in diesem Zusammenhang nachfolgend auch als „Klausel“ bezeichnet.

Die Vorbedingungen der Knoten können im Rahmen einer Breitensuche iterativ bestimmt werden. Dabei wird jeder Nachfolgeknoten \( n \) des gerade untersuchten Knotens \( n_{current} \) nach den folgenden Regeln mit seiner Vorbedingung \( Precondition(n) \) markiert. Dabei wird immer die erste zutreffende Regel angewandt und die darauf folgenden Regeln für den gerade untersuchten Knoten ignoriert:

(PC1) Als Initialisierung (Vorbedingung des Startknotens) wird \( Precondition(Start) = \text{true} \) verwendet, da der Startknoten immer ausgeführt wird.

(PC2) Sollte \( n_{current} \) tot sein, so tue nichts (ignoriere tote Knoten).

(PC3) Ist \( n \) ein AND-Join mit partiellem Abbruch und gehört \( n_{current} \) zu einem Slavezweig: Tue nichts (Slavezweige werden bezüglich des Datenflusses ignoriert).

(PC4) Sollte der Nachfolgeknoten \( n \) ein XOR-Join sein und \( n_{current} \) selbst kein XOR-Split: Setze die Vorbedingung des XOR-Joinknotens \( n \) auf die Vorbedingung von \( n_{current} \), ohne das zuletzt hinzugefügte Element

(PC5) Ist \( n_{current} \) selbst ein XOR-Split: Setze die Vorbedingung jedes Nachfolgers auf die des Splitknotens plus der Bedingung für die Ausführung des Zweigs, in dem sich der Nachfolger befindet (die Bedingung des Zweigs wird mit einem logischen Und an die Vorbedingung des Splitknotens angehängt).

(PC6) Ansonsten (also insbesondere bei normalen Knoten, AND-Splits und AND-Joins) entspricht die Vorbedingung der Nachfolger einfach der des Knotens \( n_{current} \) selbst (da in diesem Fall die Nachfolger immer ausgeführt werden, wenn der Knoten selbst ausgeführt wird)

Abbildung 7.8: Beispiel für die Bestimmung der Vorbedingungen

Ein Beispiel für die Bestimmung der Vorbedingungen nach den vorstehenden Regeln ist in Abbildung 7.8 zu sehen. Der am weitesten links stehende Knoten DX1 hat die Vorbedingung \( \text{true} \), da er immer ausgeführt wird. Da er ein XOR-Split ist, werden seine beiden Nachfolger aber nicht immer ausgeführt, sondern nur, wenn die Ausführungsbedingung des entsprechenden
7.1. Abhängige XOR-Entscheidungen

XOR-Zweigs erfüllt ist. \( \bar{u} \) erhält also die Vorbedingung \( \text{true} \land X \in \{1, 2\} \) (Regel (PC5)). Der Nachfolger von \( \bar{u} \) ist ein XOR-Join und wird daher immer dann ausgeführt, wenn auch der zugehörige XOR-Split ausgeführt wurde – also im Beispiel immer. Dies wird in Regel (PC4) berücksichtigt, der Nachfolger von \( \bar{u} \) erhält also korrekterweise die Vorbedingung \( \text{true} \). \( D_X2 \) ist ein gewöhnlicher Kontrollflussnachfolger bezüglich des gerade betrachteten XOR-Joins, also wird die Vorbedingung einfach kopiert und er erhält \( \text{true} \) (Regel (PC6)).

Durch eine wiederholte Anwendung dieser Regeln auf den übrigen Knoten erhält man nach und nach die in Abbildung 7.8 zu sehenden Vorbedingungen. Wie man leicht durch Nachvollziehen der Logik des Prozesses erkennen kann, sind diese korrekt.

Zusätzlich zu seiner Vorbedingung wird jeder Knoten mit einer Markierung \( \text{SuppliedCondition}(n) \) versehen, die die Vorbedingung für die Versorgung von \( d_{\text{target}} \) am Knoten \( n \) enthält (analog zur Markierung \( \text{supplied}(n) \) im bisherigen \text{WriterExists}-Algorithmus). Auch diese Markierung wird in einem Breitendurchlauf iterativ bestimmt. Dazu muss kein separater Durchlauf durchgeführt werden, sondern diese Markierung kann gleichzeitig zur Vorbedingung bestimmt werden. Die Regeln zur Markierung des Nachfolger \( n \) des gerade untersuchten Knotens \( n_{\text{current}} \) sind dabei wie folgt:

(SC1) \( \text{SuppliedCondition}(n) \) wird zunächst überall mit \( \text{false} \) vorinitialisiert.

(SC2) Ist \( n_{\text{current}} \) ein flexibler Endknoten, tot oder befindet sich \( n_{\text{current}} \) in einem Slavezweig und der Nachfolger \( n \) ist der zugehörige AND-Join mit partiell Abbruch, so wird die Markierung des Nachfolgers nicht erweitert.

(SC3) Ansonsten wird \( \text{SuppliedCondition}(n) \) duplikatfrei um die Markierung von \( n_{\text{current}} \) erweitert: Hänge also \( \text{SuppliedCondition}(n_{\text{current}}) \) mit einem logischen Oder an und entferne ggf. doppelt vorkommende Klauseln.

(SC4) Sofern an \( n_{\text{current}} \) ein Schreibzugriff auf \( d_{\text{target}} \) festgestellt wird:
   Hänge an die \( \text{SuppliedCondition} \) des Nachfolgers \( n \) zusätzlich die Vorbedingung des schreibenden Knotens \( n_{\text{current}} \) an (wieder mit einem logischen Oder).

Eine Markierung nach den vorstehenden Regeln für die \text{Precondition} und die \text{SuppliedCondition} würde bereits genügen, wenn sicher gestellt wäre, dass Entscheidungselemente nur genau einmal geschrieben werden. Dies ist in ADEPT allerdings nicht der Fall, so dass Maßnahmen ergriffen werden müssen, um die Vorbedingungen auch im Fall mehrfacher Schreibzugriffe korrekt zu behandeln. Es ist klar, dass ein Koppeldatenelement nach einem Schreibzugriff nicht einfach in der Vorbedingung belassen werden kann (nach einem weiteren Schreibzugriff enthält es ja einen anderen Wert), es darf aber auch nicht einfach aus der Vorbedingung entfernt oder die gesamte Vorbedingung als ungültig markiert werden, da manche Fälle dann nicht mehr richtig behandelt werden können.

Ein Beispiel für einen solchen Fall zeigt Abbildung 7.9. So ist der Lesezugriff an Knoten \( G \) auf \( d \) zulässig, obwohl \text{Decision} vorher geschrieben wurde. Wäre \text{Decision} an \( F \) einfach aus der Vorbedingung entfernt oder die ganze Vorbedingung als ungültig deklariert worden, könnte dieser Fall nicht korrekt erkannt werden. Es sind also offensichtlich komplexere Maßnahmen zur Berücksichtigung mehrfacher Schreibzugriffe nötig.

Um mehrfache Schreibzugriffe korrekt zu berücksichtigen, wird jeder Knoten \( n \) daher als dritte Markierung zusätzlich mit einer Menge \( \text{LogicalElements}(n) \) markiert. Diese Menge enthält
Abbildung 7.9: Mehrere Schreibzugriffe auf ein Entscheidungsdatenelement

Tupel \((\text{Datenelement, Schreibzähler})\). Sobald ein Entscheidungsdatenelement das erste Mal geschrieben wird, wird es (mit \(\text{Schreibzähler} = 1\)) am Nachfolgeknoten des schreibenden Knotens in diese Menge aufgenommen. Zusätzlich gibt es eine globale Menge \(\text{AllLogicalElements}\), die Tupel \((\text{Datenelement, Schreibzähler, Anzahl Decision IDs})\) enthält. Diese Menge enthält einen Eintrag für sämtliche im gesamten Prozess bekannten Einträge einer \(\text{LogicalElements}\)-Menge und speichert für jeden solchen Eintrag zusätzlich die Anzahl der Decision IDs der Entscheidungsaktivität, die das Datenelement geschrieben hat (dieser Wert wird später für Vereinfachungen der Vorbedingung benötigt).

Sofern ein beliebiger Knoten \(n\) auf irgendein Datenelement schreibt, für das in \(\text{LogicalElements}(n)\) bereits ein Eintrag vorhanden ist, wird der maximale Wert des Schreibzählers für das gegebene Datenelement aus \(\text{AllLogicalElements}\) bestimmt und am Nachfolger ein Eintrag \((\text{Datenelement, Maximum des Schreibzählers} + 1)\) hinzugefügt bzw. ein bereits vorhandener Eintrag für das selbe Datenelement ersetzt. Außerdem wird ein entsprechender Eintrag zu \(\text{AllLogicalElements}\) hinzugefügt. Wenn der schreibende Knoten eine Entscheidungsaktivität und das geschriebene Datenelement das Entscheidungsdatenelement, wird dabei die entsprechende Anzahl der Decision IDs eingetragen, ansonsten \(\text{UNDEFINED}\).

Alle anderen Einträge in \(\text{LogicalElements}(n_{\text{current}})\) werden ohne Veränderung an den Nachfolger kopiert, sofern noch kein Eintrag für das entsprechende Datenelement vorhanden ist. Ist bereits ein Eintrag vorhanden, wird am Nachfolger das Maximum der beiden Schreibzählerwerte verwendet.

In der Vorbedingung wird dann nicht nur eine Referenz auf ein Datenelement, sondern ein Tupel \((\text{Datenelement, Schreibzähler})\) gespeichert. Dies führt dazu, dass ein Datenelement, logisch gesehen, mit jedem Schreibzugriff einen neuen Namen bekommt, was die oben dargestellten Probleme behebt und die Analyse insgesamt deutlich vereinfacht\(^4\). Ein solches Tupel \((\text{Datenelement, Schreibzähler})\) wird daher nachfolgend auch als „logisches Datenelement“ bezeichnet.

Auch \(\text{LogicalElements}\) wird im selben Breitendurchlauf wie die beiden anderen Markierungen bestimmt. Dabei ergeben sich zusammengefasst die folgenden Markierungsregeln:

\[(\text{LE1})\] Initialisiere überall \(\text{LogicalElements}(n)\) und \(\text{AllLogicalElements}\) mit \(\emptyset\) vor.

\(^4\) vgl. auch die „Static Single Assignment Form“ [AWZ88, CFR+91] aus dem Compilerbau
(LE2) Ist \( n_{\text{current}} \) ein flexibler Endknoten oder tot, so wird die Markierung des Nachfolgers nicht erweitert.

(LE3) Ansonsten füge alle Einträge aus \( \text{LogicalElements}(n_{\text{current}}) \) zur Markierung von \( n \) hinzu. Sollte für einen Eintrag bereits ein Eintrag in \( \text{LogicalElements}(n) \) mit dem entsprechenden Datenelement vorhanden sein, so ändere den Schreibzählerwert auf das Maximum der beiden Einträge (\( \text{LogicalElements} \) enthält also für jedes Datenelement höchstens einen Eintrag).

(LE4) Wurde \( n_{\text{current}} \) eine Entscheidungsaktivität zugewiesen, die auf ein Entscheidungsdatenelement schreibt, das noch nicht in \( \text{LogicalElements}(n_{\text{current}}) \) enthalten ist:

a) Das Entscheidungsdatenelement hat noch keinen Eintrag in \( \text{AllLogicalElements} \): Füge \( \text{LogicalElements}(n) \) einen Eintrag für das Datenelement mit Schreibzähler \( = 1 \) hinzu und erstelle einen entsprechenden Eintrag in \( \text{AllLogicalElements} \)

b) Das Entscheidungsdatenelement hat bereits einen Eintrag in \( \text{AllLogicalElements} \): Füge \( \text{LogicalElements}(n) \) einen Eintrag für das Datenelement mit Schreibzähler gleich „Maximum des Schreibzählerwerts in \( \text{AllLogicalElements} + 1 \)“ hinzu und erstelle einen entsprechenden Eintrag in \( \text{AllLogicalElements} \).

(LE5) Wenn \( n_{\text{current}} \) auf ein Datenelement schreibt, das bereits einen Eintrag in \( \text{LogicalElements}(n_{\text{current}}) \) hat: Füge \( \text{LogicalElements}(n) \) einen Eintrag für das Datenelement mit Schreibzähler gleich „Maximum des Schreibzählerwerts in \( \text{AllLogicalElements} + 1 \)“ hinzu und erstelle einen Eintrag in \( \text{AllLogicalElements} \). Verwende für den Eintrag in \( \text{AllLogicalElements} \) als Anzahl der Decision IDs UNDEFINED, wenn \( n_{\text{current}} \) keine Entscheidungsaktivität oder das geschriebene Datenelement nicht das Entscheidungsdatenelement ist.

Bei (LE4) und (LE5) bedeutet dabei „Hinzufügen“ wie bei (LE3): „Füge hinzu, falls noch kein Eintrag für das entsprechende Datenelement vorhanden und verwende ansonsten das Maximum der beiden Schreibzähler.“

Abbildung 7.10: Bestimmung logischer Datenelemente

Abbildung 7.10 enthält ein Beispiel für die Bestimmung logischer Datenelemente anhand des Prozesses aus Abbildung 7.9.
Wie man erkennt, wird \textit{LogicalElements} am Anfang an Knoten $A$ mit der leeren Menge vor-
initialisiert (Regel (LE1)). Da $A$ eine Entscheidungsaktivität ist, die auf das Datenelement \textit{Decision} schreibt, wird an den beiden Nachfolgern $B$ und $C$ ein Eintrag (\textit{Decision},1) hinzugefügt. \textit{AllLogicalElements} wird entsprechend um (\textit{Decision},1,2) erweitert, da es an $B$ zwei mögliche Decision IDs gibt (Regel (LE4)).

Der Schreibzugriff auf $d$ am Knoten $B$ erzeugt \textit{keinen} neuen Eintrag, da $B$ keine Entscheidungs-
aktivität zugewiesen wurde und in \textit{LogicalElements}($B$) noch kein Eintrag für $d$ vorhanden ist.

Ansonsten wird die \textit{LogicalElements}-Menge zunächst einfach entsprechend Regel (LE3) an

Die Nachfolger weiterkopiert.\par

Durch den Schreibzugriff an $F$ wird an dessen Nachfolger $G$ der Schreibzähler für Decision um

ein erhöht, da bereits ein Eintrag für \textit{Decision} vorgefunden wurde. Da $F$ keine Entscheidungs-
aktivität zugewiesen wurde, wird \textit{AllLogicalElements} ein Eintrag (\textit{Decision},2,UNDEFINED)
hinzugefügt (Regel (LE5)).\par

An I schließlich wird für den Schreibzähler von Decision nach Regel (LE3) das Maximum

der Schreibzählerwerte der beiden Vorgänger verwendet, da dort nicht sicher ist, welcher Zweig

genommen wurde, man also davon ausgehen muss, dass Decision geschrieben wurde.

Bezüglich der Vorbedingungen ergeben sich dadurch die folgenden Auswirkungen: Wie in der

Abbildung dargestellt, bezieht sich die Vorbedingung an $B$ und damit die \textit{SuppliedCondition}
an dessen Nachfolger $D$ auf das Tupel (\textit{Decision},1), also auf das Datenelement \textit{Decision}

nach dem ersten Schreibzugriff. Auch die abhängige Entscheidung an $E$ hängt von Decision

nach dem ersten Schreibzugriff ab, so dass sich auch die Vorbedingungen seiner Nachfolger auf
das Tupel (\textit{Decision},1) beziehen, so auch an $F$ und $G$. Es ist dabei zu beachten, dass sich ein

Schreibzugriff auf Decision wie an $F$ nicht auf die Vorbedingung auswirkt – dieser Schreibzugriff
ändert ja nichts an der Tatsache, dass $F$ und $G$ ausgeführt werden. Dementsprechend wird bei

dem Vergleich der Vorbedingung mit der \textit{SuppliedCondition} an $G$ festgestellt, dass sich beide

auf die selbe Version von Decision beziehen, der Zugriff also erlaubt werden kann.

Wenn mittels der vorgestellten Regeln schließlich alle Vorgänger von $n_{\text{target}}$ und $n_{\text{target}}$ selbst

markiert wurden, wird die Vorbedingung von $n_{\text{target}}$ mit der Markierung für die Versorgung von $d_{\text{target}}$

an $n_{\text{target}}$ (also \textit{Precondition}($n_{\text{target}}$) mit \textit{SuppliedCondition}($n_{\text{target}}$)) verglichen.

Wenn bei diesem Vergleich herauskommt, dass die \textit{SuppliedCondition}($n_{\text{target}}$) immer erfüllt

ist, wenn die \textit{Precondition}($n_{\text{target}}$) gilt, so ist der Lesezugriff an $d_{\text{target}}$ sicher versorgt.

Zur Vorbereitung dieses Vergleichs wird aber zunächst die Vorbedingung für die Versorgung
von $d_{\text{target}}$ (\textit{SuppliedCondition}($n_{\text{target}}$)) vereinfacht: Alle mit Oder verknüpften Klauseln, die

sich nur in der Bedingung eines logischen Datenelements unterscheiden, werden verschmolzen,

indem die Vereinigungsmenge der Bedingungen gebildet wird (aus $(X \in \{1\} \land Y \in \{1,2\}) \lor (X \in \{1\} \land Y \in \{3,4\})$ wird so $X \in \{1\} \land Y \in \{1,2,3,4\}$). Dieser Vorgang wird solange wiederholt,
bis keine zwei Klauseln mehr gefunden werden, die verschmolzen werden können.

Als nächstes wird in allen Klauseln untersucht, ob es nun logische Datenelemente gibt, deren

Bedingung gleich viele Elemente enthält, wie es Decision IDs für dieses Datenelement gibt
(hierfür wird der entsprechende Wert aus \textit{AllLogicalElements} verwendet). Solche Einträge
werden durch \textit{true} ersetzt, da diese trivialerweise immer erfüllt sind.

 Dieser Fall tritt insbesondere dann ein, wenn $d_{\text{target}}$ in allen Zweigen einer XOR-Verzweigung
geschrieben wurde (aber nicht nur in diesem Fall, wie Abbildung 7.7 zeigt). Eine Erkennung
dieser Situation ist wichtig, da Einträge, die ein logisches Datenelement enthalten, das nicht in der Vorbedingung von $n_{target}$ vorkommt, beim folgenden Vergleich als unerfüllt betrachtet werden. Durch die Verschmelzung der Klauseln und die Elimination der immer erfüllten Einträge enthält die Vorbedingung für die Versorgung in ihren Einträgen nur noch jene Datenelemente, bezüglich derer wirklich eine Bedingung erfüllt sein muss.

Schließlich muss nur noch überprüft werden, ob die vereinfachte Vorbedingung für die Versorgung auch immer erfüllt ist, wenn $n_{target}$ ausgeführt wird, also wenn die Vorbedingung für die Ausführung von $n_{target}$ gilt. Dazu werden alle Wertekombinationen, bei denen die Vorbedingung für $n_{target}$ erfüllt ist, in die vereinfachte Bedingung für $d_{target}$ eingesetzt und jeweils überprüft, ob diese ebenfalls erfüllt ist. Klauseln, die ein logisches Datenelement enthalten, das nicht in der Vorbedingung für $n_{target}$ vorkommt, werden dabei wie $false$ behandelt.

Sofern die Vorbedingung für die Versorgung von $d_{target}$ dabei für alle Werteeinordnungen erfüllt ist, so ist $d_{target}$ sicher versorgt. Ansonsten ist $d_{target}$ nicht sicher versorgt.

Zusammenfassend lässt sich die Ausführung dieses Algorithmus grob in vier Phasen einteilen:

(i) Führe bisherigen $WriterExists$-Algorithmus aus. Liefert dieser als Ergebnis, dass $d_{target}$ sicher versorgt ist, so ist der Algorithmus fertig.

(ii) Führe einen Breitendurchlauf über den Graphen durch und markiere dabei alle Vorgänger von $n_{target}$ und $n_{target}$ selbst mit $Precondition$, $SuppliedCondition$ und $LogElement$.

(iii) Vereinfache die Bedingung für die Versorgung von $d_{target}$ an $n_{target}$ (also die $SuppliedCondition(n_{target})$)

(iv) Prüfe, ob $SuppliedCondition(n_{target})$ immer erfüllt ist, wenn $n_{target}$ ausgeführt wird (also wenn $Precondition(n_{target})$ gilt). Ist dies der Fall, so ist $d_{target}$ sicher versorgt, sonst nicht.

In der vorhergehenden Darstellung wurden aus Gründen der Verständlichkeit einige Details ausgelassen, die für eine Implementierung des Algorithmus noch geklärt werden müssen. Deshalb findet sich in Anhang B.4 eine formale Darstellung dieses Algorithmus als Pseudo Code, die als Vorlage für eine Implementierung dienen kann.

Ein Beispiel für eine Anwendung des gesamten Algorithmus auf einen Prozess ist in Abbildung 7.11 zu sehen. Dort soll untersucht werden, ob der Lesezugriff auf $d$ am Knoten $Target$ sicher versorgt ist. Zur besseren Übersicht wurden die Vorbedingungen der einzelnen Knoten (siehe Abb. 7.8) bis auf einige Ausnahmen weggelassen und die Datenelemente nicht als logische Datenelemente dargestellt, da dies im angegebenen Prozess unnötig ist (man müsste einfach überall $(X, 1)$ statt $X$ und $(Y, 1)$ statt $Y$ schreiben).

Wie man leicht erkennt, betrachtet der bisherige $WriterExists$-Algorithmus im abgebildeten Fall den Zugriff auf $d$ an $Target$ als nicht sicher versorgt (nicht explizit dargestellt). Es muss also der erweiterte Algorithmus angewendet werden.

Die einzelnen Grafiken zeigen, wie die Bedingung für die Versorgung von $d$ im Rahmen des Durchlaufs durch den Graphen sukzessive erweitert wird. Zunächst ist diese $false$ (Bild (a); Regel (SC1)). Da $U$ auf $d$ schreibt, wird sie an dessen Nachfolger um die Vorbedingung für $U$ erweitert, ist dort also $false \lor (true \land X \in \{1, 2\})$ (Bild (b); Regel (SC4)). Ansonsten wird
sie einfach an den Nachfolger weiter kopiert (Regel (SC3); nicht dargestellt). Am Nachfolger von \( V \) wird diese Vorbedingung wieder um die Vorbedingung von \( V \) erweitert (Bild (c); Regel (SC4)) und danach nach (SC3) einfach weiterkopiert.

Schließlich sind (in Bild (d)) alle Vorgänger und \textbf{Target} selbst markiert und es müssen die Vorbedingung für die Ausführung von \( \text{Target} \) (\( \text{Precondition} \text{(Target)} = \text{true} \wedge X \in \{1, 2, 3\} \wedge Y \in \{A, B\} \)) mit der Vorbedingung für die Versorgung von \( d \) verglichen werden (\( \text{SuppliedCondition}\text{(Target)} = \text{false} \vee (\text{true} \wedge X \in \{1, 2\}) \vee (\text{true} \wedge X \in \{2, 3\} \wedge Y \in \{A, B\}) \)). Dazu setzt man alle Wertekombination von \( X \) und \( Y \), für die \( \text{Target} \) ausgeführt wird, in die \( \text{SuppliedCondition}\text{(Target)} \) ein und prüft, ob die Evaluierung \text{true} ergibt. Konkret sind dies alle Tupel aus der Menge \( \{1, 2, 3\} \times \{A, B\} = \{(1, A), (1, B), (2, A), \ldots, (3, B)\} \). In diesem konkreten Fall ergibt dies überall \text{true}, also ist \( d \) an \text{Target} sicher versorgt.

\textbf{Mögliche Erweiterungen} Der soeben vorgestellte Algorithmus kann auf analoge Weise dazu verwendet werden, um die Beschränkungen für \textit{parallele Schreibzugriffe} zu lockern, sofern parallel ausgeführte, voneinander abhängige XOR-Verzweigungen betroffen sind (dieser Fall kommt vor allem bei den später vorgestellten ODER-Verzweigungen häufig vor).

Dabei wird zunächst festgestellt, ob überhaupt ein (möglicher) paralleler Schreibzugriff vorliegt und, wenn ja, ob sich die betroffenen Knoten innerhalb zweier abhängiger XOR-Verzweigungen befinden. Falls dies zutrifft, werden die Vorbedingungen der parallel schreibenden Knoten nach dem oben beschriebenen Schema bestimmt (allerdings nicht die Vorbedingung für das Datenelement) und dann miteinander verglichen. Sollte bei diesem Vergleich herauskommen, dass es keine Wertebelieferung gibt, die beide Vorbedingungen gleichzeitig erfüllt, also beide Knoten niemals parallel ausgeführt werden, so kann der Schreibzugriff zugelassen werden. Ein Beispiel für diese Situation ist in Abbildung 7.12 zu sehen. Dort kann durch eine verfeinerte Analyse festgestellt werden, dass die Schreibzugriffe auf \( d \) an den Knoten \( C \) und \( F \) niemals parallel stattfinden und somit erlaubt werden können.

Eine weitere mögliche Erweiterung ergibt sich, sofern, wie im implementierten ADEPT2-System, auch Schleifenendknoten gewöhnliche Entscheidungsaktivitäten zugewiesen werden. In diesem Fall kann es, wie in Abbildung 7.13 dargestellt, auch \textit{abhängige Schleifenendknoten} geben. In diesem Beispiel kann durch eine verfeinerte Analyse festgestellt werden, dass \( d \) am Knoten \( F \) obligat gelesen werden kann, da die Schleife nur terminiert (und somit \( F \) ausgeführt wird), wenn an \( B \) die Decision ID 1 gewählt wurde. In diesem Fall wird aber immer Knoten \( C \) ausgeführt, der \( d \) obligat schreibt.

Schleifenendknoten können dann innerhalb des vorgestellten Algorithmus ähnlich wie XOR-Splitknoten behandelt werden. Da der Nachfolger des Schleifenendknotens (bezüglich normaler Kontrollflusskanten) nur aktiviert wird, wenn die Bedingung für das Verlassen der Schleife erfüllt ist, kann diese Bedingung zur Vorbedingung dieses Knotens hinzugefügt werden. Der Rest der verfeinerten Analyse kann dann völlig unverändert ablaufen. Da die Schleifenrücksprungkante für die Datenfluss sicherheit unerheblich ist, kann diese (und die zugehörige Bedingung) ignoriert werden.

Außerdem werden im oben vorgestellten Algorithmus \textit{Versorgungen über Sync-Kanten} bislang noch nicht berücksichtigt. Dies kann auf ähnliche Weise wie im bisherigen WriterExists-Algorithmus berücksichtigt werden: Technisch gesehen muss dafür im Wesentlichen die selbe Logik
7.1. Abhängige XOR-Entscheidungen

(a) Zunächst wird die Bedingung für die Versorgtheit von d mit false vorninitialisiert

(b) Beim ersten Schreibzugriff auf d an Knoten U wird diese Bedingung um die Vorbedingung von U erweitert.

(c) Genauso beim zweiten Schreibzugriff an V

(d) Schließlich wird die Bedingung für einen zulässigen Lesezugriff an d mit der Vorbedingung für die Ausführung von Target verglichen

**Abbildung 7.11:** Beispiel für eine erweiterte Datenflussanalyse
Abbildung 7.12: Beispiel parallele abhängige XOR-Entscheidungen

Abbildung 7.13: Beispiel für abhängige Schleifenendknoten
implementiert werden, wie sie schon in [Rei00] zur Erweiterung des WriterExists-Algorithmus um die Unterstützung von Sync-Kanten vorgestellt wurde. Der wesentliche Unterschied besteht darin, dass statt der Markierung $supplied(n)$ die Markierung $SuppliedCondition(n)$ an die sich ergebenden neuen Nachfolger propagiert (d.h. zur $SuppliedCondition$ des Nachfolgers hinzugefügt) wird. Analog dazu wird auch $LogElement(n)$ neu propagiert. Die Vorbedingung der Knoten $Precondition(n)$ muss hingegen nicht verändert werden, da Sync-Kanten die Ausführung eines Knoten höchstens verzögern können, aber nicht die Vorbedingung für seine Ausführung beeinflussen.

**Komplexitätsanalyse** Im Folgenden wird die Laufzeitkomplexität der einzelnen Phasen des vorgestellten Algorithmus untersucht. Dabei sei $|N|$ als die Zahl der Knoten im Prozess definiert und $|D|$ als die Zahl der Datenelemente.

Die erste Phase des Algorithmus besteht aus der Ausführung des bisherigen WriterExists-Algorithmus und weist daher wie dieser eine Komplexität von $O(|N|^2)$ auf.

Die Markierungsphase (ii) des Algorithmus hat grob nach oben abgeschätzt eine Komplexität von $O(|N|^2|D|)$. Der Faktor von $|N|^2$ entsteht dadurch, dass jeder der $|N|$ Knoten potentiell $|N| - 1$ Vorgänger besitzt, und dadurch potentiell an jedem Knoten $|N| - 1$ Markierungsvorgänge stattfinden. Der Faktor $|D|$ entsteht durch die Vereinigung der die beiden LogicalElements-Mengen. Dabei muss jeder Eintrag aus der Menge des aktuellen Knotens zu der des Nachfolgers hinzugefügt werden, es muss also jedes Element der LogicalElements des aktuellen Knotens untersucht werden. Diese kann potentiell einen Eintrag für jedes Datenelement, also maximal $|D|$ Einträge, enthalten. Der dabei nötige Test, ob bereits ein Eintrag für das hinzuzufügende Datenelement existiert, kann durch eine geeignete Datenstruktur (Hash tabelle) in $O(1)$ erledigt werden. Insgesamt kann die Komplexität dieser Phase im Vergleich zur Komplexität von $O(|N|^2)$ beim unmodifizierten WriterExists-Algorithmus also durchaus als noch akzeptabel angesehen werden.

Ein Vereinfachungsschritt in Phase (iii) (Funktion $mergeSuppConds$ in Listing B.7) weist eine Komplexität von $O(|N|^3)$ auf. Der Grund dafür liegt darin, dass jeder Knoten höchstens eine Klausel zur Vorbedingung für $d_{large}$ beitragen kann, diese Vorbedingung also höchstens $|N|$ Klauseln enthält. Jede dieser Klauseln enthält höchstens $\frac{|N|}{2}$ Elemente, da für jedes in der Vorbedingung eines Knotens vorkommende Datenelement mindestens ein XOR-Split Vorgänger und ein XOR-Join Nachfolger dieses Knotens sein muss. Die Länge der Vorbedingung eines Knotens entspricht also genau der Schachtelungstiefe bezüglich XOR-Verzweigungen, innerhalb derer sich der Knoten befindet.

In jedem Vereinfachungsschritt muss untersucht werden, ob es für eine gegebene Klausel eine andere gibt, mit der diese verschmolzen werden kann. Dazu muss jede der höchstens $|N|$ Klauseln mit den anderen $O(|N|)$ Klauseln verglichen werden, es werden also $O(|N|^2)$ Vergleiche durchgeführt. In jedem solchen Vergleich müssen (bei einer naiven Implementierung) zur Feststellung der Verschmelzbarkeit die einzelnen Elemente der beiden Klauseln miteinander verglichen werden, was einen weiteren Faktor von $\frac{|N|}{2}$ hinzufügt. Zusammen ergibt dies die oben genannte Komplexität von $O(|N|^3)$.

Zusätzlich muss nach jeder Verschmelzung von Klauseln untersucht werden, ob nun einige Elemente daraus entfernt werden können. Dazu muss die Anzahl der dem entsprechenden lo-
gischen Datenelement zugeordneten Decision IDs aus AllLogicalElements ausgelesen werden. Dieses Auslesen ist allerdings durch eine Hashtabelle in $O(1)$ machbar, fällt also bezüglich der Gesamtkomplexität nicht weiter ins Gewicht.

Insgesamt muss diese Vereinfachung so lange durchgeführt werden, bis keine zwei Klauseln mehr gefunden werden, die sich miteinander verschmelzen lassen. Da es im Worst Case $|N|$ Klauseln gibt und in jedem Schritt mindestens eine Klausel eliminiert werden muss (da die Schleife sonst vorher terminiert), wird dieser Vereinfachungsschritt höchstens $O(|N|)$ mal ausgeführt.

Insgesamt weist die Vereinfachung der Vorbedingung im Worst Case also eine Komplexität von $O(|N|^4)$ auf. Diese Komplexität kann zwar nicht mehr wirklich als „gut“ bezeichnet werden, kann aber, auch in Hinblick auf die hier vorgenommene sehr grobe Abschätzung (der dargestellte Worst Case dürfte in der Praxis äußerst selten sein), durchaus noch als akzeptabel betrachtet werden.

Problematisch ist allerdings – auch bedingt durch den zugegebenermaßen naiven Ansatz – die Komplexität des Vergleichs der Vorbedingung für die Versorgung von $d_{\text{target}}$ mit der Vorbedingung für die Ausführung von $n_{\text{target}}$ in Phase (iv) (Funktion EvaluateSuppCond in Listing B.8).

Wenn $\maxIDCount$ die maximale Anzahl Decision IDs darstellt, die einem Zweig zugewiesen wurde, beträgt die Komplexität dieser Funktion $O(\maxIDCount |N| - |N|^2)$: Eine Vorbedingung besteht wie oben festgestellt aus höchstens $|N|$ Elementen mit jeweils höchstens $\maxIDCount$ Decision IDs. Das heißt, dass sich daraus höchstens $\maxIDCount |N|$ Wertebelegungen ableiten lassen. Das Einsetzen einer solchen Wertebelegung in die SuppliedCondition lässt sich dann in $O(|N|^2)$ erledigen (Die Werte müssen in $O(|N|)$ Klauseln mit je $O(|N|)$ Elementen eingesetzt werden; der Existenztest auf den einzelnen Mengen von Decision IDs wird als $O(1)$ angenommen).

Insgesamt beträgt die Gesamtkomplexität des Algorithmus über alle vier Phasen demnach $O(|N|^2 + O(|N|^2|D|) + O(|N|^4) + O(\maxIDCount |N| - |N|^2) = O(|N|^2 (\maxIDCount |N| - |N|^2 + |D|))$, ist also wegen des in Phase (iv) hinzugekommenen Faktors von $\maxIDCount |N|$ exponentiell.

Zur Verbesserung der Gesamtkomplexität ist es daher von Interesse, wie und ob der Vergleich in Phase (iv) effizienter gestaltet werden kann. Ein möglicher Ansatz dafür ergibt sich aus der Interpretation der Vorbedingung und der SuppliedCondition als aussagenlogische Formeln:

Bei einer solchen Interpretation ist $d_{\text{target}}$ an $n_{\text{target}}$ sicher versorgt, wenn $\text{Precondition}(n_{\text{target}}) \rightarrow \text{SuppliedCondition}(n_{\text{target}}) = \neg \text{Precondition}(n_{\text{target}}) \vee \text{SuppliedCondition}(n_{\text{target}})$ immer erfüllt, also eine Tautologie ist. Dies ergibt sich aus der Überlegung, dass $\text{Precondition}(n_{\text{target}})$ eine hinreichende Bedingung für die Versorgung von $d_{\text{target}}$ darstellen muss, wenn diese an $n_{\text{target}}$ sichergestellt sein soll\(^5\).

Eine Interpretation einer Vorbedingung als aussagenlogische Formel ergibt sich, indem man jeden möglichen Wert eines Datenelements als eigenes Literal interpretiert und zusätzlich die Bedingung formuliert, dass die sich so ergebenden Literale eines Datenelements niemals gleichzeitig erfüllt sind, diese also mit einem Exklusiv-Oder verknüpft.

Beispiel ($\oplus$ beschreibe ein logisches Exklusiv-Oder): Aus $X \in \{1, 2\} \land Y \in \{1, 2, 3, 4\}$ ergibt

\(^5\) Ist die Precondition erfüllt, so muss die SuppliedCondition erfüllt sein. Ist die Precondition nicht erfüllt, so ist es gleichgültig, ob die SuppliedCondition erfüllt ist, da $n_{\text{target}}$ dann nicht ausgeführt wird.
7.1. Abhängige XOR-Entscheidungen

sich: \((x_1 \oplus x_2) \land (y_1 \oplus y_2 \oplus y_3 \oplus y_4)\)

Die Interpretation der \textit{SuppliedCondition} ergibt sich analog (durch Oder-Verknüpfung ihrer nach dem selben Schema interpretierten Klauseln).

Allerdings ist das Problem, ob eine aussagenlogische Formel eine Tautologie ist, NP-hart [Sch05]. Daher besitzen alle bislang bekannten Algorithmen exponentielle Komplexität (in der Zahl der Literale), so dass hiervon leider auch keine grundsätzliche Verbesserung zu erwarten ist. Allerdings ist dieses Problem innerhalb der theoretischen Informatik gut untersucht, so dass sich mit Hilfe der aussagenlogischen Interpretation der Vorbedingungen aller Voraussicht nach bessere Algorithmen als der oben angegebene naive Algorithmus finden lassen.

Zudem ist es bislang noch nicht bewiesen, dass der Vergleich der Vorbedingungen nur in exponentieller Zeit möglich ist. Es liegt also durchaus noch im Bereich des Möglichen, dass es über einige Kniffe gelingen kann, den Vergleich der Vorbedingungen in polynomialer Zeit zu realisieren.

7.1.4. Zusammenfassung


Darüber hinaus kann durch die Möglichkeit der expliziten Modellierung von Abhängigkeiten eine noch größere Robustheit bei Änderungen erzielt werden, da das System bei Änderungen an der bestimmenden Entscheidungsaktivität diese entweder automatisch oder an die davon abhängigen Aktivitäten propagieren kann bzw. den Benutzer darauf hinweisen kann, wo er noch überall Änderungen vornehmen muss.

Ohne eine solche explizite Modellierung, also, wenn die bestimmende Entscheidungsaktivität einfach manuell an die abhängigen Positionen kopiert wird, kann bei solchen Änderungen leicht die ein oder andere eigentlich ebenfalls zu modifizierende abhängige Aktivität übersehen werden, was insbesondere in Kombination mit optionalen Lesezugriffen schnell zu einem Fehlverhalten zur Laufzeit führen kann.

Trotz all dieser Vorteile hat der vorgestellte Algorithmus einen gravierenden Nachteil, nämlich dass ein Teil davon – der Vergleich der Vorbedingungen – eine exponentielle Komplexität aufweist. Allerdings kann man argumentieren, dass dieses Problem für die Praxis vermutlich weit weniger gravierend ist als es zunächst scheinen mag (also dieses Komplexitätsproblem nicht zwingend zu Effizienzproblemen führt).

Sowohl die Vorbedingung für die Ausführung des untersuchten Knotens als auch die Vorbedingung für die sichere Versorgtheit des obligat gelesenen Datenelements sind oft wahrscheinlich deutlich kürzer als es theoretisch im Worst Case möglich ist. So ist die Länge der Vorbedingung
eines Knotens durch die Anzahl XOR-Verzweigungen, in die er geschachtelt ist, bestimmt. Diese Schachtelungstiefe wird vermutlich meistens nicht allzu groß sein, so dass die Vorbedingungen der Knoten und damit auch die Vorbedingung für die Versorgtheit kurz bleiben. Auch die Zahl der einem Zweig zugewiesenen Decision IDs wird meistens eher klein sein.

Die beschriebene Exponentialität ergibt sich aus der Anzahl der Wertbelegungen, an denen eine Vorbedingung erfüllt ist (Max. Anzahl IDs \( \text{Schachtelungstiefe} \)). Da beide Werte in der Praxis eher klein sein werden, hat diese Exponentialität auf die Ausführungsgeschwindigkeit voraussichtlich geringere Auswirkungen, als es zunächst scheint (insbesondere verschwindet sie ganz, wenn jedem Zweig höchstens eine Decision ID zugewiesen wurde).

Daher kann eine Implementierung des Algorithmus in der hier beschriebenen Form trotzdem in Betracht gezogen werden, wobei aber ggf. eine Einschränkung bezüglich der Länge der unterstützten Vorbedingungen gemacht werden sollte. Dadurch kann man dem Benutzer in den häufig vorkommenden einfachen Fällen (wie in Abbildung 7.1) einen erhöhten Komfort bieten, ohne bei pathologischen Beispielen eine unangemessen lange Laufzeit der Analyse zu riskieren.

7.2. Verzögerte XOR-Entscheidungen

In vielen Situation kann es vorteilhaft sein, die Entscheidung an einer XOR-Verzweigung nicht explizit über eine Entscheidungsaktivität zu treffen, sondern dies implizit durch den Start der gewünschten Aktivität aus der Arbeitsliste zu erledigen. Eine solche Semantik wird durch die in diesem Abschnitt beschriebenen verzögerten XOR-Entscheidungen implementiert.

Zunächst aber wird das zugrundeliegende Problem nochmals detailliert beschrieben (Abschnitt 7.2.1). Danach wird untersucht, wie verzögerte XOR-Entscheidungen implementiert werden können und welche Gemeinsamkeiten und Unterschiede es zu nicht verzögerten, expliziten Entscheidungen gibt (Abschnitt 7.2.2). Zum Abschluss wird eine kurze Zusammenfassung dieses Abschnitts gegeben (Abschnitt 7.2.3).

7.2.1. Problembeschreibung


Abbildung 7.14: Verzögerte XOR-Entscheidungen

In diesem Beispiel kann einzig die Entscheidung zwischen **Normale Bearbeitung** und **Bearbeitung durch Chef** unter Umständen auch mittels einer expliziten Benutzerauswahl getroffen werden, indem bestimmte Benutzer über ein Dialogfeld gefragt werden, welcher Bearbeitungs- typ gewünscht wird (oder über eine automatische Entscheidungsaktivität, falls es harte Kriterien gibt, anhand derer über die Wichtigkeit eines Vorgangs entschieden werden kann). Dennoch ist eine solche Lösung suboptimal, da der oder die Benutzer, die die Entscheidung treffen, sich jeden Vorgang anschauen müssen, bevor er den eigentlichen Bearbeitern vorgelegt wird, und dadurch stark belastet sind.

Die zeitliche Überwachung des Timeouts kann dagegen überhaupt nicht auf eine explizite Auswahl abgebildet werden, da während der Ausführung des Splitknotens noch nicht bekannt ist, welcher Zweig genommen werden muss.

Mit Hilfe eines verzögerten Entscheidungsknotens lässt sich diese Semantik dagegen ohne Schwierigkeiten umsetzen: Der Start von **Timeout aufgetreten** nach einer bestimmten Wartezeit ist problemlos machbar und es müssen darüber hinaus keine Benutzer benannt werden, die entscheiden, ob eine Bearbeitung durch den Vorgesetzten zu erfolgen hat oder nicht. Zudem ergibt sich durch die Semantik einer verzögerten Entscheidung eine Art implizite Vertreterregelung bzw. Lastverteilung (wenn der Vorgesetzte keine Zeit hat, werden die Vorgänge normal bearbeitet).

Allerdings ist es zur Unterstützung verzögerter XOR-Entscheidungen nötig, das Metamodell um einen entsprechenden Knotentyp zu erweitern, da für die temporäre Aktivierung der Nachfolger des Splitknotens die Markierungsregeln des Metamodells angepasst werden müssen (und man erkennt schon an diesem einfachen Beispiel, dass sich eine implizite Auswahl in der Regel nicht auf eine äquivalente explizite Auswahl abbilden lässt).


Insgesamt entspricht dieses Konzept innerhalb der **Workflow Patterns** genau WCP-16 (Deferred Choice).

**7.2.2. Existierendes Metamodell**

Da für die Implementierung verzögerter XOR-Entscheidungen die Graphmarkierungsregeln modifiziert werden müssen, ist für dieses Konzept eine Anpassung des **Workflowsystems** zwingend
7.2. Verzögerte XOR-Entscheidungen

notwendig.

Es ist also naheliegend, einen neuen XOR-Splitknotentyp zu definieren, der die verzögerte Semantik implementiert (aber ansonsten im Wesentlichen identisch zu normalen XOR-Splitknoten ist). Diesem Knotentyp darf allerdings keine Aktivität zugewiesen werden, da er eine spezielle Ausführungssemantik implementiert. Zusätzlich weist er einen (System-)Ausgabeparameter auf, über den als Entscheidungsparameter die implizit gewählte Decision ID ausgegeben wird.

Im Rahmen dieser Arbeit werden diese Knoten wie schon in Abbildung 7.14 als XOR-Splitknoten mit der kursiven Aufschrift „Verzögert“ dargestellt.

7.2.2.1. Kontrollfluss

Der Kontrollflussablauf bei verzögerten XOR-Entscheidungen ist an sich relativ naheliegend und größtenteils schon durch das Konzept selbst festgelegt: Sobald die Ausführung des verzögerten XOR-Splitknotens \( n_{\text{split}} \) beginnt, müssen alle seine direkten Nachfolger in den Arbeitslisten angeboten werden, also in den Zustand \textit{ACTIVATED} wechseln. Sobald einer dieser Nachfolger gestartet werden soll (also unmittelbar bevor er in den Zustand \textit{RUNNING} wechselt), wurde die Entscheidung getroffen. Die anderen Zweige werden daher wie bei einer explizit getroffenen Entscheidung mit \textit{SKIPPED} markiert (und somit übersprungen) und \( n_{\text{split}} \) wechselt in den Zustand \textit{COMPLETED}.

Dies alles macht nur dann Sinn, wenn keiner der Nachfolger von \( n_{\text{split}} \) ein NULL-Knoten ist, da ansonsten dieser sofort ausgeführt und somit immer ausgewählt wird. Für eine korrekte Funktionsweise einer verzögerten XOR-Entscheidung muss also sichergestellt sein, dass jedem direkten Nachfolger eines verzögerten XOR-Splitknotens eine Aktivität zugewiesen wurde.

Ein ähnlicher Fall ergibt sich, wenn direkten Nachfolgern von \( n_{\text{split}} \) automatische Aktivitäten zugewiesen wurden: Auch diese werden typischerweise sofort gestartet, so dass anderen Nachfolgern zugewiesene manuelle Aktivitäten in der Praxis niemals ausgeführt werden. Zudem ist es bei mehreren sofort startenden Aktivitäten zur Entwurfszeit völlig unklar bzw. nicht deterministisch, welche Aktivität schneller ist und daher zur Laufzeit ausgeführt wird.

Dieser Nichtdeterminismus und vor allem die Nichtausführung manueller Aktivitäten wird vom Modellierer häufig nicht beabsichtigt sein, stellt aber aus Systemsicht keine Abweichung gegenüber der Semantik bei manuellen Aktivitäten dar (auch dort ist es durch die implizite Semantik aus Systemsicht nichtdeterministisch, welche Aktivität ausgeführt wird). Da sich mit Hilfe von automatischen Aktivitäten zudem auch sinnvolle Szenarien modellieren lassen (z.B. die zeitliche Überwachung in Abbildung 7.14), ist es nicht sinnvoll, automatische Aktivitäten an direkten Nachfolgern von \( n_{\text{split}} \) zu verbieten. Nichtsdestotrotz ist es empfehlenswert, dass der Modellierer in diesem Fall vom System eine Warnung vor einer möglicherweise unerwarteten Semantik erhält.

Darüber hinaus ist noch der Spezialfall eines leeren Zweigs zu beachten (der nach der soeben festgestellten Korrektheitsbedingung nur dann zulässig ist, wenn dem Joinknoten eine Aktivität zugewiesen wurde): In diesem Fall muss der zu \( n_{\text{split}} \) gehörende Joinknoten \( n_{\text{join}} \) zunächst wie die anderen direkten Nachfolger auch mit \textit{ACTIVATED} markiert werden. Sofern aber einer der anderen Nachfolger von \( n_{\text{split}} \) gestartet wird, darf \( n_{\text{join}} \) nicht mit \textit{SKIPPED} markiert werden, sondern die Markierung muss auf \textit{NOT\_ACTIVATED} zurückgesetzt werden (da dieser Knoten nach
Definitionen:
- **n_split**: Der verzögerte XOR-Splitknoten.
- **d_decision**: Das Datenelement, in dem die getroffene Verzweigungsentscheidung abgelegt wird.
- **Succ:: c_succ(n_split)**: Die Menge der direkten Nachfolgeknoten von *n_split* bezüglich Kontrollflusskanten.

<table>
<thead>
<tr>
<th>Knoten</th>
<th>von Zustand</th>
<th>nach Zustand</th>
<th>Aktion</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>n_split</em></td>
<td>ACTIVATED oder SUSPENDED</td>
<td>STARTED</td>
<td>Setze Zustand aller Knoten ∈ Succ auf ACTIVATED</td>
</tr>
<tr>
<td><em>n_split</em></td>
<td>STARTED</td>
<td>SUSPENDED</td>
<td>Setze Zustand aller Knoten ∈ Succ auf NOT_ACTIVATED</td>
</tr>
<tr>
<td><em>n_split</em></td>
<td>Beliebig</td>
<td>FAILED</td>
<td>Setze Zustand aller Knoten ∈ Succ auf NOT_ACTIVATED</td>
</tr>
<tr>
<td><em>n ∈ Succ</em></td>
<td>SELECTED</td>
<td>ACTIVATED</td>
<td>Setze Zustand aller Knoten ∈ Succ \ {n} auf ACTIVATED</td>
</tr>
<tr>
<td><em>n ∈ Succ</em></td>
<td>SELECTED oder RUNNING⁷</td>
<td>ACTIVATED</td>
<td>Setze Zustand aller Knoten ∈ Succ \ {n} auf ACTIVATED</td>
</tr>
</tbody>
</table>

⁷Diese Aktion muss direkt nach dem Start des Knoten *n* durch den Benutzer, aber vor Beginn der Ausführung der *n* zugewiesenen Aktivität (und Lesen der Eingabeparameter) durchgeführt werden.

**Tabelle 7.1: Zustandsübergänge und Markierungsregeln bei verzögerten XORs**

Ende der Ausführung des ausgewählten Zweigs auf jeden Fall noch ausgeführt wird). Alternativ kann man die Existenz von leeren Zweigen bei verzögerten XOR-Entscheidungen ganz verbieten. In diesem Fall tritt dieser Spezialfall niemals auf, so dass bei den Markierungsregeln keine Ausnahme gemacht werden muss.

In jedem Fall impliziert diese Semantik eine beachtenswerte Abweichung gegenüber den bisherigen Markierungsregeln, da die Nachfolger des Splitknotens bereits aktiviert werden, obwohl der Splitknoten selbst noch ausgeführt wird. Dies mutet auf den ersten Blick vielleicht etwas seltsam an, ist allerdings völlig konsistent zur Semantik expliziter Entscheidungen: Der Entscheidungsknoten wird in beiden Fällen solange ausgeführt, bis die Entscheidung getroffen wurde.

Ein anderes Verhalten wäre im Übrigen auch problematisch, da sonst zum Speichern einer die Entscheidung repräsentierenden Decision ID in einem Datenelement ein Schreibzugriff stattfinden müsste, obwohl der schreibende Splitknoten seine Ausführung bereits beendet hat.


Einem verzögerten XOR-Splitknoten wird keine Entscheidungsaktivität zugewiesen, die die
7.2. Verzögerte XOR-Entscheidungen

Menge der möglichen Decision IDs definiert, so dass die Menge der Decision IDs automatisch generiert werden muss. Da überdies jede Kante nur genau eine Decision ID zugewiesen werden darf, wird hier dementsprechend auch das Mapping der IDs auf die Kanten automatisch und nicht manuell durch den Benutzer durchgeführt. Bei einer Zuordnung mehrerer Decision IDs zu einer Kante gäbe es das Problem, dass unklar ist, welche der zugewiesenen IDs bei Wahl dieser Kante geschrieben werden soll.

Die Decision IDs und ihre Zuordnung zu den Kanten können nach dem folgenden Schema automatisch generiert werden: Wenn ein verzögerter XOR-Splitknoten \( m \) ausgehende Kanten besitzt, so verwende die Menge \( \{0, 1, 2, \ldots, m - 1\} \) als Menge der möglichen Decision IDs und weise der von oben gesehen ersten Kante (mit Edge Code 0) die Decision ID 0 zu, der zweiten Kante die ID 1, usw.

Ein Beispiel für solches Mapping ist in Abbildung 7.14 zu sehen: Die von oben gesehen erste Kante hat die Decision ID 0, die zweite Kante die ID 1 und die dritte Kante die ID 2.


Trotz aller dargestellten Abweichungen sind verzögerte XOR-Entscheidungen aus Kontrollflussicht aber insgesamt fast äquivalent zu nicht verzögerten XOR-Entscheidungen:

Bevor der Splitknoten \( n_{\text{split}} \) ausgeführt wird oder nachdem der ausgewählte direkte Nachfolger von \( n_{\text{split}} \) gestartet wurde, kann man allein anhand der Knotenmarkierungen nicht unterscheiden, ob an \( n_{\text{split}} \) eine explizite oder implizite Auswahl getroffen wird: Vor der Ausführung von \( n_{\text{split}} \) ist der komplette XOR-Block in beiden Fällen mit NOT_ACTIVATED markiert und der Wert des Entscheidungsdatenelements undefiniert. Nach Start des ausgewählten direkten Nachfolgers von \( n_{\text{split}} \) wird in beiden Fällen der ausgewählte Zweig ausgeführt, \( n_{\text{split}} \) ist beendet, die anderen Zweige sind mit SKIPPED markiert und das Entscheidungsdatenelement enthält die diesem Zweig zugewiesene Decision ID.

Die einzigen Unterschiede treten also zwischen dem Start von \( n_{\text{split}} \) und dem Start des ausgewählten Nachfolgers, hier \( n_{\text{selected}} \) genannt, auf. In beiden Fällen wird \( n_{\text{split}} \) beendet, bevor \( n_{\text{selected}} \) gestartet wird und es werden auch keine anderen Aktivitäten des XOR-Blocks zwischen beiden Knoten ausgeführt. Bezüglich dieser beiden Knoten besteht der einzige Unterschied also darin, dass bei einer expliziten Auswahl \( n_{\text{selected}} \) erst nach Ende der Ausführung von \( n_{\text{split}} \) mit
aktiviert markiert wird, während dies bei einer impliziten Auswahl schon währenddessen der Fall ist.

Bezüglich der nicht ausgewählten Nachfolgeknoten stellt man ähnliches fest: Diese Knoten werden in beiden Fällen nicht ausgeführt und der Unterschied besteht nur darin, dass sie bei der impliziten Auswahl während der Ausführung von n_split zeitweise mit ACTIVATED markiert sind, während sie bei der expliziten Auswahl zu dieser Zeit weiterhin NOT_ACTIVATED sind.

Aus Sicht der Aktivitäten an den Nachfolgeknoten von n_split sind diese Unterschiede in beiden Fällen aber völlig unerheblich, da sie nicht aktiv werden, wenn ihr Knoten lediglich aktiviert oder deaktiviert wird, sondern erst, sobald die Markierung auf STARTED wechselt.

Zusammengefasst lassen sich drei Phasen der Ausführung von XOR-Entscheidungen unterscheiden:

(i) Vor Ausführung von n_split:
   Keine Unterschiede zwischen expliziter und impliziter Entscheidung.

(ii) Während der Ausführung von n_split:
   Bei expliziter Entscheidung sind weiterhin alle Zweige mit NOT_ACTIVATED markiert. Bei impliziter Entscheidung sind die direkten Nachfolger von n_split ACTIVATED.

(iii) Nach Ende der Ausführung von n_split:
   Keine Unterschiede zwischen expliziter und impliziter Entscheidung.

Dies ist nochmals in Abbildung 7.16 illustriert. Wie man sieht, sind die Graphen bezüglich der Knotenmarkierung in den Phasen (i) und (iii) völlig identisch, nur während der Ausführung des Splitknotens in Phase (ii) besteht ein Unterschied.

Abbildung 7.16: Äquivalenz expliziter und impliziter Auswahl
7.2.2.2. Datenfluss

Ein Datenfluss kann an einem Knoten nur im Zeitraum zwischen Start und Ende der Ausführung der zugewiesenen Aktivität auftreten, also insbesondere nicht, solange der Knoten lediglich aktiviert ist und in einer Arbeitsliste erscheint.

Wie im vorigen Abschnitt festgestellt wurde, besteht der einzige Unterschied zwischen verzögerten und nicht verzögerten XOR-Entscheidungen aber lediglich im ACTIVATED-Status der direkten Nachfolgeknoten des Splitknotens in Phase (ii). Sobald eine dieser Nachfolgeaktivitäten in Phase (iii) gestartet wurde, sobald also Daten fließen können, sind diese beiden Fälle aber ununterscheidbar. Damit dies auch für die direkten Nachfolger sichergestellt ist, ist es daher wichtig, dass der Wert des Entscheidungsdatenelements bei einer verzögerten Entscheidung geschrieben wird, bevor die ausgewählte Aktivität die Eingabeparameter liest.

Aufgrund dieser Unterscheidbarkeit beider Semantiken ist der Datenfluss bei verzögerten XOR-Entscheidungen vollständig äquivalent zu dem bei gewöhnlichen, nicht verzögerten XOR-Entscheidungen, so dass keine Anpassungen der Datenflussanalyse vorgenommen werden müssen.

7.2.3. Zusammenfassung

Wie wir in diesem Abschnitt gesehen haben, lassen sich verzögerte XOR-Entscheidungen und somit eine implizite Benutzerauswahl ohne größere Schwierigkeiten implementieren, wenn auch die Einführung eines neuen Knotentyps hierfür notwendig ist. Ansonsten sind aufgrund der weitgehenden Äquivalenz zu nicht verzögerten XORs keine weiteren Anpassungen nötig.

Nichtsdestotrotz lassen sich mit verzögerten XOR-Entscheidungen einige interessante Szenarien auf einfache Weise modellieren, die vorher nicht oder nur unbefriedigend modellierbar waren. Insbesondere ist es durch die implizite Semantik möglich, dass mehrere Benutzer quasi unabhängig bzw. autonom voneinander in die Entscheidungsfindung eingebunden werden können, ohne dass bestimmten Benutzern die Rolle eines Entscheiders zugewiesen werden muss. Indem die Bearbeiterzuordnung der Nachfolger des verzögerten XOR-Splitknotens entsprechend gewählt wird, ist es zudem auf einfache Weise möglich, bestimmten Benutzern nur bestimmte Entscheidungsoptionen anzubieten.

Zudem ist möglich, die Benutzer eine Entscheidung treffen zu lassen, ohne dass sie sich dessen in vollem Umfang bewusst sind, was möglicherweise aus psychologischen Gründen zu einer schnelleren Entscheidungsfindung beiträgt (viele Menschen zögern, wenn sie eine Entscheidung treffen müssen bzw. schieben das Entscheiden, wenn möglich, auf andere ab).

Daher ergeben sich gerade im Zusammenhang mit manuellen Aktivitäten in der Praxis vielfältige Einsatzmöglichkeiten für verzögerte XOR-Entscheidungen.

Eine Übersicht über die verzögerten XOR-Entscheidungen entsprechenden Workflow Patterns ist in Tabelle 7.2 zu sehen. Verzögerte XOR-Entscheidungen in der hier vorgestellten Semantik unterstützen die Kontextbedingungen des Workflow Patterns WCP-16 (Deferred Choice) vollständig – sobald ein Zweig gewählt wurde, wird die Aktivierung der übrigen Zweige sofort
Erweiterungen bedingter Verzweigungen zurückgezogen. Sie gehen aber auch nicht darüber hinaus (entsprechen also genau diesem Pattern). Da der zugehörige Joinknoten ein gewöhnlicher XOR-Joinknoten ist, wird diesem das Pattern WCP-5 zugeordnet.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Anmerkungen</th>
</tr>
</thead>
</table>
| WCP-16 (*Deferred Choice*) | * Splitknoten  
* Kontextbedingungen vollständig erfüllt |
| WCP-5 (*Simple Merge*)    | * Joinknoten  
* Entspricht gewöhnlichem XOR-Join              |

**Tabelle 7.2: Unterstützte Workflow Patterns**
7.3. ODER-Verzweigungen

In vielen Anwendungsfällen kann es nötig sein, an einer bedingten Verzweigung nicht nur einen, sondern beliebig viele Zweige auszuwählen zu können, die dann parallel zueinander ausgeführt werden. Es ist demnach notwendig, neben der bislang unterstützten $1$ aus $n$ Auswahl eine $m$ aus $n$ Auswahl zu unterstützen.

Dieser Abschnitt beschreibt, wie ODER-Verzweigungen (also eine solche $m$ aus $n$ Auswahl) in ADEPT umgesetzt werden können. Dazu wird zunächst eine Beschreibung des zugrundeliegenden Problems gegeben (Abschnitt 7.3.1) und untersucht, wie ODER-Verzweigungen mittels bereits vorhandener Metamodellkonstrukte umgesetzt werden können (Abschnitt 7.3.2). Danach wird für eine dieser Varianten beschrieben, wie eine automatische Abbildung von ODER-Verzweigungen auf die vorhandenen Konstrukte gelingt und welche Konsequenzen dies für den Kontroll- und Datenfluss hat (Abschnitt 7.3.3). Schließlich wird eine kurze Zusammenfassung gegeben und es werden mögliche Erweiterungen beschrieben (Abschnitt 7.3.4).

7.3.1. Problembeschreibung


Dies lässt sich – wie später in diesem Abschnitt noch ausführlich dargestellt wird – zwar problemlos mit den bereits bestehenden Verzweigungstypen (AND und XOR) modellieren, allerdings entspricht eine ODER-Verzweigung in vielen Fällen eher der Denkweise der meisten
7. Erweiterungen bedingter Verzweigungen

Menschen und macht die Logik des Prozesses damit leichter verständlich als bei einer Modellierung unter direkter Verwendung paralleler und bedingter Verzweigungen.

An diesem Beispiel lässt sich aber auch ein prinzipbedingtes Manko von ODER-Verzweigungen erkennen: Hinter dem ODER-Joinknoten (am Knoten Reise antreten) darf keines der Datenelemente obligat gelesen werden, da jedes nur in jeweils einem Zweig geschrieben wird und es (wie bei einer XOR-Verzweigung) nicht sicher ist, dass der entsprechende Zweig immer ausgeführt wird. Beispielsweise ist es nicht sicher, dass Fluginfo einen Wert (ungleich NULL) enthält, da bei einem nahe gelegenen Ziel überhaupt kein Flug gebucht wird, weshalb in diesem Fall auch keine Buchungsinformationen über einen Flug vorliegen können. Somit kann Fluginfo am Knoten Reise antreten nur optional gelesen werden (was in diesem Fall aber semantisch kein Problem darstellt). Für Hotelinfo, Zuginfo und Mietautoinfo gilt entsprechendes.


Insgesamt weisen ODER-Verzweigungen also sowohl die Datenflussbeschränkungen von XOR-Verzweigungen als auch die Beschränkungen von AND-Verzweigungen auf. Aus der Kombination beider Beschränkungen folgt, dass in einer ODER-Verzweigung kein Datenelement so beschrieben werden kann, dass es hinter der Verzweigung obligat gelesen werden darf (zumindest ohne eine detaillierte Analyse wie bei abhängigen XOR-Verzweigungen). Optionales Lesen ist natürlich kein Problem.
Diese Beschränkung ist zudem unabhängig von der konkreten Umsetzung, da sie sich direkt aus der Semantik von ODER-Verzweigungen ableiten lässt (ein Zweig wird nicht in jedem Fall ausgeführt, aber wenn, dann möglicherweise parallel zu anderen).

Innerhalb der Workflow Patterns entsprechen die hier vorgestellten ODER-Verzweigungen einer Kombination von WCP-6 (Multi-Choice) für den Splitknoten mit WCP-7 (Structured Synchronizing Merge) für den Joinknoten.

7.3.2. Umsetzungsvarianten

Wie schon im Kontext der Workflow Patterns festgestellt wurde [RHAM06], lässt sich jede ODER-Verzweigung als Workaround problemlos mittels einer Kombination von AND- und XOR-Verzweigungen darstellen. Daher ist es – um das Metamodell möglichst einfach zu halten – naheliegend, in ADEPT ODER-Verzweigungen intern auf diese Weise zu implementieren und lediglich dem Benutzer über ein Sichtenkonzept („Views“) einen ODER-Knoten zu präsentieren (vgl. auch [BBR06, BRB07]).
7.3. ODER-Verzweigungen

Bevor wir aber eine solche automatische Abbildung von ODER-Verzweigungen definieren können, muss zunächst einmal diskutiert werden, auf welche Art und Weise eine ODER-Verzweigungen überhaupt in eine Darstellung mit AND- und XOR-Knoten transformiert werden kann und welche Vor- und Nachteile die einzelnen Varianten aufweisen.

Zusammengefasst gibt es genau zwei grundlegende Schemata für eine solche Transformation:

1. Es wird außen eine XOR-Verzweigung verwendet, die eine passende AND-Verzweigung mit der auszuführende Zweigkombination auswählt.

2. Es wird außen eine AND-Verzweigung verwendet, die parallel für jeden Zweig eine XOR-Verzweigung ausführt, die entscheidet, ob der jeweilige Zweig ausgeführt wird oder nicht.

Natürlich sind auch Kombinationen davon denkbar (ein Teil der Zweige wird auf die eine Art transformiert und ein anderer auf die andere), für eine Diskussion der Vor- und Nachteile ist es aber zweckmäßiger, sich auf diese beiden Extreme zu beschränken, zumal sich die so gewonnenen Erkenntnisse dann ggf. auch einfach auf eine Mischform übertragen lassen. Aus diesem Grund werden nachfolgend lediglich diese beiden Möglichkeiten diskutiert, bevor anschließend eine der Varianten für die Implementierung in ADEPT ausgewählt wird.

7.3.2.1. XOR außen

Bei dieser Variante wird für eine ODER-Verzweigung mit \( n \) Zweigen eine XOR-Verzweigung mit \( 2^n - 1 \) Zweigen verwendet, die die auszuführende Zweigkombination auswählt. Jeder Zweig dieser XOR-Verzweigung repräsentiert dabei eine mögliche parallele Ausführungskombination der ursprünglichen Zweige. Jeder XOR-Zweig enthält also innerhalb einer AND-Verzweigung eine Kopie einer (jeweils anderen) Teilmenge der ursprünglichen Zweige der ODER-Verzweigung.

Formal lässt sich dies auch so ausdrücken: Wenn \( Z \) die Menge der ODER-Zweige ist, so enthält die XOR-Verzweigung einen Zweig für jedes Element der Potenzmenge von \( Z \) ohne die leere Menge, also einen Eintrag für jedes Element in \( P(Z) \setminus \{\emptyset\} \).

Eine Transformation des Beispiels aus Abbildung 7.17 nach diesem Schema ist in Abbildung 7.18 dargestellt. Der dargestellte Graph ergibt sich dabei über folgende Überlegungen (von oben nach unten betrachtet):

- Es kann nur ein einzelner ODER-Zweig ausgewählt werden: Je ein XOR-Zweig mit einer Kopie eines einzelnen ODER-Zweigs ist nötig.

- Es kann auch eine beliebige Kombination zweier ODER-Zweige ausgewählt werden: XOR-Zweige für jede mögliche „Zweierkombination“ sind nötig.

- Es kann eine beliebige Kombination dreier ODER-Zweige ausgewählt werden: XOR-Zweige für jede mögliche „Dreierkombination“ sind nötig.

Abbildung 7.18: Abbildung 7.17 mittels XOR außen und AND innen modelliert
Diesen Zweigen müssen dann noch die entsprechenden Ausführungsbedingungen zugeordnet werden, was hier allerdings aus Gründen der Übersicht weggelassen wurde.

Im allgemeinen Fall gilt, dass jeder der $n$ ODER-Zweige im transformierten Graphen nun in $2^{n-1}$ Kopien vorkommt und zusätzlich jeweils $2^n - n - 1$ AND-Split- und Joinknoten (je einer für jede mehr als einelementige Teilmenge) benötigt werden. Dies sind offensichtlich exponentiell viele Knoten mehr als bei einer direkten Modellierung nötig würden.

Um aus den Bedingungen der Zweige der ODER-Verzweigung die korrekten Ausführungsbedingungen für die Zweige der äußeren XOR-Verzweigung abzuleiten, ist zudem eine genaue Analyse erforderlich. Es muss untersucht werden, bei welcher Ausführungsbedingung welche Teilmenge von ODER-Zweigen ausgeführt wird, und dann der dieser Teilmenge von ODER-Zweigen entsprechenden Zweig der XOR-Verzweigung die jeweilige Ausführungsbedingung zugewiesen werden.


7.3.2.2. AND außen

Für eine ODER-Verzweigung mit $n$ Zweigen werden bei dieser Variante $n$ XOR-Verzweigungen parallel ausgeführt, die jeweils entscheiden, ob der entsprechende Zweig übersprungen oder ausgeführt wird. Dadurch kommt auch im transformierten Graphen jeder Zweig weiterhin genau einmal vor, womit der Graph nach der Transformation strukturell ähnlich zum untransformierten Graphen bleibt.

Dies erleichtert die Bestimmung der Ausführungsbedingungen für die Zweige der XOR-Verzweigungen deutlich, da die Bedingung für die Ausführung eines Zweigs direkt übernommen werden kann und zur Bestimmung der Bedingung für das Überspringen lediglich das Komplement davon gebildet werden muss. Es muss zur Bestimmung der korrekten Ausführungsbedingungen also (anders als bei XOR außen) keine aufwändige Analyse der ausführbaren Zweigkombinationen durchgeführt werden.


\[\text{Wenn alle Zweige zusammen } m_Z \text{ Knoten enthalten, sind dies genau } 2 \cdot (2^n - n - 1) + 2^{n-1} m_Z - m_Z = 2^{n-1} (m_Z + 4) - m_Z - 2n - 2 \text{ zusätzliche Knoten.}\]
7. Erweiterungen bedingter Verzweigungen

Im allgemeinen Fall werden für die Modellierung einer ODER-Verzweigung mit \( n \) Zweigen also nur die Split- und Joinknoten der XOR-Verzweigungen zusätzlich benötigt (der ODER-Split und -Join wird durch je einen AND-Knoten ersetzt und die Zweige sind weiterhin genau einmal vorhanden). Dies sind genau \( 2n \) – also nur linear viele – Knoten mehr als bei einer direkten Modellierung.

Dadurch, dass zur Ableitung der Ausführungsbedingungen im Gegensatz zur Modellierung mit XOR außen keine Analyse der ausführbaren Zweigkombinationen durchgeführt werden muss, ist bei dieser Darstellung weiterhin nicht direkt erkennbar, welche Zweigkombinationen niemals ausgeführt werden, so dass die Datenflussbeschränkungen der ODER-Verzweigung zunächst in vollem Umfang erhalten bleiben. Dies ist aber kein Nachteil, da eine solche Analyse natürlich bei Bedarf trotzdem durchgeführt werden kann und dann als Vereinfachung niemals gemeinsam ausgeführte Zweige in gemeinsame XOR-Verzweigungen eingebettet werden können (wodurch diese Beschränkungen wieder etwas gelockert werden). Zudem kann bei systemseitiger Unterstützung abhängiger XOR-Entscheidungen durch eine geschickte Implementierung dafür gesorgt werden, dass solche Lockierungen der Datenflussbeschränkungen automatisch im Rahmen der Analyse abhängiger XOR-Entscheidungen erkannt werden, so dass im Kontext der ODER-Verzweigung keine separate Analyse erforderlich ist.

7.3.2.3. Diskussion

Bei der Variante mit der XOR-Verzweigung außen werden (potentiell) exponentiell viele zusätzliche Knoten benötigt und die Graphstruktur ändert sich stark, was eine automatische Abbildung mittels Sichten erschwert. Zudem ist eine verhältnismäßig aufwändige Analyse der
Verzweigungsbedingungen erforderlich. Allerdings bietet diese Analyse der Verzweigungsbedingungen den Vorteil, dass möglicherweise einige Ausführungskombinationen der Zweige ausgeschlossen werden können, was die einer ODER-Verzweigung inhärenten Datenflussbeschränkungen etwas lockern kann.

Bei der Variante mit der AND-Verzweigung außen hingegen werden lediglich linear viele zusätzliche Knoten benötigt, die Graphstruktur bleibt sehr ähnlich und die Bestimmung der Verzweigungsbedingungen ist sehr einfach. Durch die regelmäßige Struktur ist es zudem verhältnismäßig einfach, die interne Implementierung vor dem Benutzer zu verbergen, da nur die Teile des Graphen, die die AND- und XOR-Knoten vor und nach den ODER-Zweigen enthalten, zusammengefasst dargestellt werden müssen. Die Darstellung der Zweigknoten selbst kann dagegen eins zu eins erfolgen, ohne dass der Status mehrerer Knoten zusammengefasst werden muss.

Ohne eine detaillierte Analyse der Verzweigungsbedingungen bleiben bei dieser Variante allerdings die Datenflussbeschränkungen zunächst in vollem Umfang erhalten. Bei Unterstützung abhängiger XOR-Entscheidungen oder durch eine zusätzlich durchgeführte Analyse wie bei XOR außen, kann dieser Nachteil allerdings vollständig behoben werden.

Daher ist für eine Implementierung in ADEPT eindeutig die letztere Variante zu bevorzugen und wird dementsprechend nachfolgend näher untersucht.

7.3.3. Existierendes Metamodell

Anmerkung: Im folgenden wird für einige Optimierungen davon ausgegangen, dass die in Abschnitt 7.1 beschriebenen abhängigen XOR-Entscheidungen unterstützt werden – dies ist allerdings keine zwingende Voraussetzung für die Implementierung von ODER-Verzweigungen.

7.3.3.1. Kontrollfluss

Zunächst einmal werden wir klären, wie die Transformation einer gegebenen ODER-Verzweigung in eine Darstellung mit einer AND-Verzweigung außen genau ablaufen muss. Intuitiv ist nach Abschnitt 7.3.2.2 klar, was prinzipiell getan werden muss: Jeder ODER-Zweig muss in einen XOR-Verzweigung eingebettet werden, die die Ausführungsbedingung für den ODER-Zweig beibehält. Zusätzlich erhält diese XOR-Verzweigung einen leeren Zweig, der den ODER-Zweig überspringt, falls seine Ausführungsbedingung nicht erfüllt ist. Schließlich muss der ODER-Split durch einen AND-Splitknoten und der ODER-Join durch einen ODER-Joinknoten ersetzt werden.

Für eine genaue Untersuchung ist es allerdings nötig, diesen informellen Ablauf etwas formaler zu fassen. Daher definieren wir die folgenden Variablen:

Gegeben sei eine ODER-Verzweigung mit (vgl. auch Abbildung 7.20a):

- ODER-Splitknoten $OR_{split}$
- ODER-Joinknoten $OR_{join}$
7. Erweiterungen bedingter Verzweigungen

- k Zweigen $z_{\text{start}}^1 \rightarrow \cdots \rightarrow z_{\text{end}}^1$, $z_{\text{start}}^2 \rightarrow \cdots \rightarrow z_{\text{end}}^2$, bis $z_{\text{start}}^k \rightarrow \cdots \rightarrow z_{\text{end}}^k$ ($z_{\text{start}}^i$ ist also der erste Knoten des i-ten Zweigs und $z_{\text{end}}^i$ der letzte). Leere Zweige seien nicht vorhanden.

- Entscheidungsdatenelement $d_{\text{decision}}$ (wird von der $O_{\text{split}}$ zugewiesenen Entscheidungsaktivität geschrieben)

- Außerdem sei AllIDs die Menge aller möglichen Decision IDs, die von der Entscheidungsaktivität an $O_{\text{split}}$ geschrieben werden können.

 Dann kann diese ODER-Verzweigung wie folgt transformiert werden:

1. Für jeden Zweig $l \in 1, \ldots, k$:
    a) Speichere die Zweig $l$ zugewiesenen Decision IDs in einer Variablen $IDs^l$.
    b) Erstelle einen Zweig $l$ umgebenden XOR-Block: Füge einen XOR-Splitknoten $XOR_{\text{split}}^l$ zwischen $O_{\text{split}}$ und $z_{\text{start}}^l$ ein und einen XOR-Joinknoten $XOR_{\text{join}}^l$ zwischen $z_{\text{end}}^l$ und $O_{\text{join}}$.
    c) Füge dem XOR-Block einen leeren Zweig $XOR_{\text{split}}^l \rightarrow XOR_{\text{join}}^l$ hinzu
    d) Weise $XOR_{\text{join}}^l$ eine Koppeaktivität\(^7\) zu, die die Entscheidung aus $d_{\text{decision}}$ übernimmt.
    e) Weise der Kante $XOR_{\text{split}}^l \rightarrow z_{\text{start}}^l$ die Decision IDs $IDs^l$ zu und dem leeren Zweig $XOR_{\text{split}}^l \rightarrow XOR_{\text{join}}^l$ die Decision IDs $\text{AllIDs} \setminus IDs^l$ (also das „Komplement“ der dem entsprechenden ODER-Zweig zugewiesenen IDs).

2. Ersetze $O_{\text{split}}$ durch einen AND-Splitknoten $AND_{\text{split}}$ und $O_{\text{join}}$ durch einen AND-Joinknoten $AND_{\text{join}}$. Die diesen Knoten zugewiesenen Aktivitäten bleiben dabei erhalten, d.h. $AND_{\text{split}}$ ist hinterher die Entscheidungsaktivität von $O_{\text{split}}$ zugewiesen und $AND_{\text{join}}$ diejenige Aktivität, die ggf. vorher $O_{\text{join}}$ zugewiesen war (bzw. NULL, falls $O_{\text{join}}$ ein NULL-Knoten war).

Diese Transformation ist anhand eines Beispiels in Abbildung 7.20 illustriert: In Teil (a) ist zunächst die zu transformierende ODER-Verzweigung zu sehen.

Für die Transformation wird als Erstes jeder Zweig in einen XOR-Block eingebettet, aber die ODER-Knoten zunächst beibehalten. Desweiteren wird jeder solchen XOR-Verzweigung ein Leerzweig hinzugefügt (Teil (b)).

Jedem XOR-Splitknoten wird dann eine Koppeaktivität zugewiesen und der Zweig mit den ODER-Knoten erhält als Ausführungsbedingung eine Kopie der Ausführungsbedingung für den ODER-Zweig, in dem er sich befindet. Der Leerzweig erhält als Ausführungsbedingung das „Komplement“ davon (also die Menge aller möglichen Decision IDs, ohne jene, die den ODER-Zweig auswählen). Dies ist in Teil (c) zu sehen.

Schließlich werden die beiden ODER-Knoten noch durch entsprechende AND-Knoten ersetzt und die Transformation ist fertig (Teil (d)).

\(^7\)Bzw. eine andere, vordefinierte Aktivität mit der selben Semantik (lies eine Decision ID über einen Eingabe- parameter ein und gib sie unverändert über den Entscheidungsparameter aus), falls keine Koppeaktivitäten definiert sind.
7.3. ODER-Verzweigungen

(a) Die zu transformierende ODER-Verzweigung

(b) Bette jeden Zweig in einen umgebenden XOR-Block ein und füge einen Leerzweig zu diesem hinzu.

(c) Weise jedem XOR-Spliknoten eine Koppelaktivität zu und setze entsprechende Decision IDs

(d) Ersetze die ODER-Knoten durch entsprechende AND-Knoten

Abbildung 7.20: Transformation einer ODER-Verzweigung

Hierdurch bleibt die Kontrollflusssemantik der ursprünglichen ODER-Verzweigung vollständig erhalten: Die Menge der Decision IDs, die zur Ausführung eines bestimmten Zweigs führen, hat sich nicht geändert und es ist durch die Verwendung von Koppelaktivitäten sichergestellt, dass jede der Entscheidungsaktivitäten an den XOR-Splits die selbe ID wie die bestimmende, ursprünglich dem ODER-Splitknoten zugewiesene Entscheidungsaktivität ausgibt. Daher werden bei einer gegebenen ID die selben Zweige wie vorher ausgeführt. Durch den AND-Join ist es weiterhin sichergestellt, dass die dem ODER-Joinknoten zugewiesene Aktivität (bzw. nachfolgende Knoten) erst dann ausgeführt wird, wenn alle Zweige fertig sind – auch dieses hat sich also nicht geändert.


Wenn dem Benutzer auch zur Laufzeit ODER-Knoten präsentiert werden sollen, müssen die Statusmarkierungen der transformierten Form in die untransformierte Darstellung rückübertragen werden können. Für Zweigknoten ist dies trivial: Da diese eins zu eins übernommen werden, können die Markierungen der Zweigknoten im transformierten Graphen einfach direkt
auf die entsprechenden Knoten im untransformierten Graphen übertragen werden.

Um zu bestimmen, woraus die Markierung des ODER-Split- und des ODER-Joinknotens abgeleitet werden kann, überlegt man sich, wie die erwartete Laufzeitsemantik der ODER-Knoten aussieht und wie im Vergleich dazu der Ablauf in der transformierten Form ist.

Die erwartete Ausführungssemantik eines ODER-Splitknotens ist, dass dieser Knoten solange ausgeführt wird, wie die Entscheidungsfindung durch die Entscheidungsaktivität dauert. Der Knoten muss daher beendet sein, bevor die ersten Zweige aktiviert werden. Dies entspricht im Wesentlichen dem Ausführungstatus des AND-Splitknotens, so dass einfach der Status dieses Knotens übernommen werden kann.

Allerdings kann es dadurch eine kurze, in ungünstigen Fällen sichtbare Verzögerung zwischen dem Abschluss der Ausführung des ODER-Splitknotens und der Aktivierung der ausgewählten Zweige geben, da in der transformierten Form zwischen beiden Aktionen noch die XOR-Splitknoten ausgeführt werden müssen. Da die Ausführung dieser Knoten aber sehr schnell von statten gehen sollte, kann dies in der Praxis vernachlässigt werden.


### 7.3.3.2. Datenfluss

Analysiert man die transformierte Form einer ODER-Verzweigung, so stellt man die gleichen Datenflussbeschränkungen fest, wie sie sie auch bei der untransformierten Form zu erwarten sind:

Ein Datenelement, das in der transformierten Form innerhalb einer der XOR-Verzweigungen obligat geschrieben wird, darf am entsprechenden XOR-Joinknoten nicht obligat gelesen werden (da es im leeren Zweig dieser Verzweigung offensichtlich nicht geschrieben wird). Dementsprechend darf es auch am AND-Joinknoten und dessen Nachfolgern nicht obligat gelesen werden.

Da jeder Zweig aus der untransformierten ODER-Verzweigung in eine solche XOR-Verzweigung eingebettet wird, darf ein dort geschriebenes Datenelement also wie erwartet hinter der ODER-Verzweigung nicht obligat gelesen werden.

Auch dürfen (ohne Synchronisation durch Sync-Kanten) keine zwei Zweige aus der ursprünglichen ODER-Verzweigungen auf das selbe Datenelement schreiben, da sich diese in der transformierten Form in unterschiedlichen Zweigen der AND-Verzweigung befinden. Auch hier ändern sich...
sich die bestehenden Einschränkungen also nicht.
Wird in der ODER-Verzweigung die Ausführungsreihenfolge durch Sync-Kanten festgelegt, um ein Schreiben aus verschiedenen Zweigen zu ermöglichen, so findet auch in der transformierten Darstellung eine Synchronisation zwischen den verschiedenen AND-Zweigen statt, so dass hier nun wie gewünscht ein Schreibzugriff möglich ist.

Abbildung 7.21: Äquivalenz der Datenflussbeschränkungen in untransformierter und transformierter Form

Die Äquivalenz der Datenflussbeschränkungen ist nochmals in Abbildung 7.21 anhand eines Beispiels illustriert: So darf das Datenelement singlewrite weder in der transformierten noch in der untransformierten Form am Knoten read gelesen werden, da es nur in einem einzelnen Zweig geschrieben wurde. Genauso ist der parallele Schreibzugriff auf das Datenelement parwrite in beiden Fällen unzulässig, während die Zugriffe auf syncread aufgrund der Sync-Kante $z_{\text{end}}^1 \rightarrow z_{\text{end}}^2$ zulässig sind.

Außerdem lässt sich feststellen, dass sich durch eine Analyse abhängiger XOR-Entscheidungen gerade beim Datenfluss von ODER-Verzweigungen einige Verbesserungen ergeben können. So kann oftmals beobachtet werden, dass zwei Zweige eigentlich niemals parallel ausgeführt werden (also immer entweder der eine oder der andere Zweig), so dass Schreibzugriffe auf das selbe Datenelement aus diesen Zweigen eigentlich zulässig sind. Außerdem kann es gerade in solchen
Fällen häufig vorkommen, dass die Zweige komplementäre Fälle der Decision IDs abdecken, so dass das entsprechende Datenelement hinter dem ODER-Joinknoten eigentlich sicher versorgt ist.

Diese beiden Fälle können durch die weiter vorne beschriebene Analyse abhängiger XOR-Entscheidungen erkannt und korrekt behandelt werden, ohne dass dort für transformierte ODER-Verzweigungen irgendwelche Erweiterungen vorgenommen werden müssen.

Ein Beispiel hierfür ist am Datenelement `depwrit` in Abbildung 7.21 zu sehen. So kann durch eine Analyse abhängiger XOR-Entscheidungen in der transformierten Form erkannt werden, dass die Schreibzugriffe von $z_{end}^1$ und $z_{end}^3$ auf `depwrit` auch ohne Sync-Kante zulässig sind, da diese beiden Knoten aufgrund der Ausführungsbedingungen der beiden Zweige niemals parallel ausgeführt werden. Zudem kann der obligate Lesezugriff des Knotens `read` auf `depwrit` erlaubt werden, da bei jeder möglichen Decision ID immer entweder $z_{end}^1$ oder $z_{end}^3$ ausgeführt wird und `depwrit` somit sicher versorgt ist ($z_{end}^1$ wird für $d_{decision} \in \{1, 2\}$ ausgeführt, $z_{end}^3$ für $d_{decision} \in \{3, 4\}$, mögliche Werte sind $d_{decision} \in \{1, 2, 3, 4\}$).

### 7.3.4. Zusammenfassung

ODER-Verzweigungen lassen sich auf einfache Weise auf AND- und XOR-Verzweigungen abbilden und sich somit ohne Erweiterung des Metamodells realisieren. Für eine benutzerfreundliche Realisierung ist es jedoch wünschenswert, dem Benutzer innerhalb des Editors und der Überwachungswerkzeuge die untransformierte Form der ODER-Verzweigung zu präsentieren, so dass hier ein Sichtenkonzept implementiert werden sollte. Im Fall von ODER-Verzweigungen ist dies jedoch relativ simpel, da im Wesentlichen nur die XOR-Verzweigungen verborgen und die beiden AND-Knoten mit einem anderen Symbol dargestellt werden müssen. Eine solcherart implementierte Sicht bietet auch den Vorteil, dass ADEPT dann auch im Kontext der Workflow Pattern eine Unterstützung von WC-6 und WC-7 für sich reklamieren kann, da eine direkte, manuelle Modellierung mittels AND und XOR dort ausdrücklich als nicht ausreichend erachtet wird.

Eine weiterer interessanter Nebeneffekt dieser Modellierungsart ist es, dass sich darüber präuboloh ohne Zusatzaufwand erweiterte ODER-Semantiken modellieren lassen. Wenn man beispielsweise die bisher verwendeten nicht verzögerten XOR-Splitknoten durch verzögerte XOR-Splits ersetzt, so erhält man ein verzögerstes ODER, d.h. der Benutzer kann durch Auswahl aus der Arbeitsliste für jeden Zweig wählen, ob er ihn ausführen oder überspringen möchte.

Falls in Zukunft weitere erweiterte XOR-Semantiken (oder auch AND-Semantiken) definiert werden, lassen sich diese auf analoge Weise auf ODER-Verzweigungen übertragen. Man gewinnt durch die indirekte Modellierung also Einiges an Flexibilität und Erweiterbarkeit.

Durch Verwendung verzögerter XOR-Knoten innerhalb des ODER-Splits ist dann allerdings sogar ohne größeren Zusatzaufwand eine erweiterte Semantik möglich, die innerhalb der Workflow Patterns bislang nicht (direkt) auftaucht, nämlich ein verzögertes ODER (bzw. „Deferred Multi-Choice“ in der Terminologie der Workflow Patterns). Die Möglichkeiten einer solcherart implementierten ODER-Verzweigung gehen also potentiell über die Unterstützung von WCP-6 und 7 hinaus.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Anmerkungen</th>
</tr>
</thead>
</table>
| WCP-6 (Multi-Choice)                 | * Splitknoten  
* Kontextbedingungen vollständig erfüllt  
* Gilt nur als unterstützt, wenn Knoten direkt (über eine Sicht) angeboten wird |
| WCP-7 (Structured Synchronizing Merge) | * Joinknoten  
* Kontextbedingungen vollständig erfüllt  
* Gilt nur als unterstützt, wenn Knoten direkt (über eine Sicht) angeboten wird |

*Tabelle 7.3: Unterstützte Workflow Patterns*
7. Erweiterungen bedingter Verzweigungen
8. Verwandte Arbeiten

Nachfolgend wird die Unterstützung der hier behandelten Konzepte in einigen anderen verwan- 
deten Arbeiten bzw. Workflowsprachen untersucht. Zunächst wird nochmals ein tabella-
rischer Überblick darüber gegeben, welche Workflow Patterns vom ADEPT-Metamodell un-
ter Berücksichtigung der hier vorgenommenen Erweiterungen unterstützt werden (Abschnitt 
8.1).

Danach wird untersucht, ob und wie die beschriebenen Konzepte von YAWL (Abschnitt 8.2) 
und WS-BPEL 2.0 (Abschnitt 8.3) umgesetzt sind. Schließlich wird in Abschnitt 8.4 ein kurzes 
Fazit gezogen.

8.1. Workflow Patterns

Da die Zuordnung der hier vorgestellten Konzepte zu den entsprechenden Workflow Pattern 
sehen in den jeweiligen Kapiteln behandelt wurde, gibt dieser Abschnitt lediglich eine tabella-
rische Zusammenfassung in Form von Tabelle 8.1.

Diese Tabelle gibt Auskunft darüber, welche Workflow Patterns vom ADEPT-Metamodell unter 
Berücksichtigung der Erweiterungen aus dieser Arbeit unterstützt werden und stellt daher im 
Wesentlichen eine modifizierte Kopie von Tabelle 2.1 dar. Gegenüber Tabelle 2.1 zusätzlich 
unterstützte Pattern sind in Tabelle 8.1 **fett** und neu hinzugekommene Anmerkungen **kursiv** 
markiert.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Unterstützt?</th>
<th>Anmerkungen</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCP-1 (Sequence)</td>
<td>Ja</td>
<td>—</td>
</tr>
<tr>
<td>WCP-2 (Parallel Split)</td>
<td>Ja</td>
<td>AND-Split</td>
</tr>
<tr>
<td>WCP-3 (Synchronization)</td>
<td>Ja</td>
<td>AND-Join</td>
</tr>
<tr>
<td>WCP-4 (Exclusive Choice)</td>
<td>Ja</td>
<td>XOR-Split</td>
</tr>
<tr>
<td>WCP-5 (Simple Merge)</td>
<td>Ja</td>
<td>XOR-Join</td>
</tr>
<tr>
<td>WCP-6 (Multi-Choice)</td>
<td>Ja</td>
<td>ODER-Split (Abschnitt 7.3)</td>
</tr>
<tr>
<td>WCP-7 (Structured Synchronizing Merge)</td>
<td>Ja</td>
<td>ODER-Join (Abschnitt 7.3)</td>
</tr>
<tr>
<td>WCP-8 (Multi-Merge)</td>
<td>Nein</td>
<td>Nur bei tokenbasierten Ansätzen sinnvoll.</td>
</tr>
<tr>
<td>WCP-9 (Structural Discriminator)</td>
<td>Nein</td>
<td>Datenfluss problematisch, Generalisierung loser Synchronisation mit Weiterlaufen der Slavezweige (Abschnitt 5.2)</td>
</tr>
<tr>
<td>WCP-10 (Arbitrary Cycles)</td>
<td>Nein</td>
<td>Durch Blockstruktur unmöglich.</td>
</tr>
<tr>
<td>WCP-11 (Implicit Termination)</td>
<td>Nein</td>
<td>Wegen Blockstruktur wenig sinnvoll; Praxistauglichkeit des Patterns allgemein fraglich.</td>
</tr>
<tr>
<td>Pattern</td>
<td>Unterstützt?</td>
<td>Anmerkungen</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>WCP-12 (Multiple Instances without Synchronization)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-13 (Multiple Instances with a priori Design-Time Knowledge)</td>
<td>Ja</td>
<td>Über WCP-14</td>
</tr>
<tr>
<td>WCP-14 (Multiple Instances with a priori Run-Time Knowledge)</td>
<td>Ja</td>
<td>Variable Parallelität (Kapitel 4)</td>
</tr>
<tr>
<td>WCP-15 (Multiple Instances without a priori Run-Time Knowledge)</td>
<td>Nein</td>
<td>Entspricht dynamischer Bestimmung der Instanzanzahl bei variabler Parallelität; Praxisrelevanz dieses Konzepts fraglich (vgl. Abschnitt 4.2.2).</td>
</tr>
<tr>
<td>WCP-16 (Deferred Choice)</td>
<td>Ja</td>
<td>Verzögerte XOR-Entscheidung (Abschnitt 7.2)</td>
</tr>
<tr>
<td>WCP-17 (Interleaved Parallel Routing)</td>
<td>Nein</td>
<td>Angabe einer partiellen Ordnung ist durch Sync-Kanten möglich; Aktivitäten ohne Relation zueinander können jedoch immer nebeneinander ausgeführt werden.</td>
</tr>
<tr>
<td>WCP-18 (Milestone)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-19 (Cancel Activity)</td>
<td>Teilweise</td>
<td>Vorwärtsprung ohne Nachholen (Einschränkungen durch Blockstruktur; nur Zurücksetzen, aber kein Abbruch)</td>
</tr>
<tr>
<td>WCP-20 (Cancel Case)</td>
<td>Ja</td>
<td>Flexible Endknoten (Fehlerendknoten; Kapitel 6)</td>
</tr>
<tr>
<td>WCP-21 (Structured Loop)</td>
<td>Ja</td>
<td>—</td>
</tr>
<tr>
<td>WCP-22 (Recursion)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-23 (Transient Trigger)</td>
<td>Nein</td>
<td>Nicht im Metamodell; Aktivitäten können im implementierten System jedoch Signale empfangen.</td>
</tr>
<tr>
<td>WCP-24 (Persistent Trigger)</td>
<td>Nein</td>
<td>Siehe WCP-23.</td>
</tr>
<tr>
<td>WCP-25 (Cancel Region)</td>
<td>Teilweise</td>
<td>Vorwärtsprung ohne Nachholen (Einschränkungen durch Blockstruktur; nur Zurücksetzen, aber kein Abbruch)</td>
</tr>
<tr>
<td>WCP-26 (Cancel Multiple Instance Activity)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-27 (Complete Multiple Instance Activity)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-28 (Blocking Discriminator)</td>
<td>Nein</td>
<td>—</td>
</tr>
<tr>
<td>WCP-29 (Cancelling Discriminator)</td>
<td>Teilweise</td>
<td>Finale Auswahl (kein Abbruch ohne Zurücksetzen bereits ausgeführter Aktivitäten möglich) oder AND-Join mit partiellem Abbruch (Abschnitt 5.3; nicht alle Zweige gleichwertig)</td>
</tr>
<tr>
<td>WCP-30 (Structured Partial Join)</td>
<td>Nein</td>
<td>Datenfluss problematisch, Generalisierung loser Synchronisation mit Weiterlaufen der Slavezweige (Abschnitt 5.2)</td>
</tr>
<tr>
<td>WCP-31 (Blocking Partial Join)</td>
<td>Nein</td>
<td>—</td>
</tr>
</tbody>
</table>
8.2. YAWL

YAWL (Yet Another Workflow Language) [HA05] ist eine von den für die Workflow Patterns verantwortlichen Gruppen\(^1\) entwickelte Workflowsprache. YAWL wurde mit dem Ziel entwickelt, eine Referenzimplementierung eines Workflowsystems bereitzustellen, das die Modellierung aller definierten Workflow Patterns unterstützt. Dieses Ziel wurde auch erreicht, YAWL bietet (nach [HA05]) eine Unterstützung für alle definierten Pattern aus [AHKB03] mit Ausnahme von WCP-11 (Implicit Termination).


\(^{1}\)An der Technischen Universität Eindhoven (Niederlande) und der Queensland University of Technology (Brisbane, Australien)
Eine Open-Source-Implementierung von YAWL in Java kann unter http://www.yawl-system.com/ heruntergeladen werden; eine Beschreibung dieser Implementierung findet sich auf der Website bzw. in [AADH04, KBP07, Ouy05, KRAA06].

YAWL ist ein formaler, petrinetzbasierter, aber (im Gegensatz zu ADEPT) nicht blockbasierter Ansatz. Innerhalb der Modelle sind also unter Anderem beliebige Kombination von Split- und Joinknoten und beliebige Schleifen erlaubt. Dies erhöht im Vergleich zu ADEPT die Flexibilität der spezifizierbaren Modelle, erschwert aber Analysen des Prozesses und ermöglicht es, Prozesse mit Verklemmungen, nicht terminierende Workflows oder tote Aktivitäten zu modellieren (was bei ADEPT teilweise schon aufgrund der Syntax ausgeschlossen ist). Dank der vorhandenen formalen Grundlage können solche Probleme allerdings auch in YAWL schon zur Entwurfszeit erkannt und (mit Hilfe des Modellierers) beseitigt werden.

Ein Überblick über die in YAWL unterstützten Modellierungskonstrukte ist in Abbildung 8.1 gegeben. *Conditions* sind mit den Stellen in Petrinetzen vergleichbar, während *Tasks* Transitionen entsprechen. Im Gegensatz zu Petrinetzen dürfen bei YAWL allerdings auch zwei Tasks ohne zwischengeschaltete Condition miteinander verbunden werden (nicht jedoch zwei Conditions ohne zwischengeschalteten Task). Die informelle Semantik der Konstrukte ist wie folgt (für eine exakte formale Beschreibung siehe [HA05]):

**Condition:** Gewöhnliche Stelle als „Zwischenlager“ für Token

**Input Condition:** *Startknoten:* Hier wird beim Start des Prozesses ein einzelnes Token erzeugt.

**Output Condition:** *Endknoten:* Eine Senke für Token, die beliebig viele Token aufnehmen kann. Ein Prozess terminiert, wenn keine Token mehr im Umlauf sind (also nicht, wenn kein Schaltvorgang mehr möglich ist, aber noch Token vorhanden sind (also eine Verklemmung auftritt) und auch nicht, wenn das erste Token die Output Condition erreicht).

**Atomic Task:** Knoten für elementare Aktivitäten

**Composite Task:** Knoten für Subprozesse

**Multiple Instances of . . .:** *Mehrinstanzknoten:* Wie . . . , nur, dass mehrere Instanzen der Aktivität bzw. des Subprozesses erstellt werden können

**AND/XOR/OR-Split Task:** Im Wesentlichen analog zu ADEPT (also Ausgabe eines Tokens über alle/eine/m viele ausgehende Kanten)

**AND/XOR/OR-Join Task:** Im Wesentlichen analog zu ADEPT (also Schalten, wenn über alle/eine/m viele eingehende Kanten ein Token empfangen wurde)

**Remove Tokens:** Der zugewiesene Task entfernt alle Token aus dem markierten Bereich, bricht diesen Bereich also ab.

Zur Implementierung des Datenflusses unterstützt YAWL sowohl sich auf den gesamten Prozess beziehende Netzwerkvariablen (*net variables*; analog zu Datenelementen in ADEPT) als auch Variablen, die nur innerhalb einer einzelnen Aktivität gültig sind (*task variables*). Der Wert einer einzelnen Variable wird zur Laufzeit durch ein XML-Dokument repräsentiert. Um eine (gewisse) Typsicherheit zu gewährleisten, wird daher jeder Variablen zur Entwurfszeit entweder ein vordefinierter primitiver XML-Schemadatentyp (wie *string*, *long*, *double*, ...) oder ein
Abbildung 8.1: Übersicht über die von YAWL unterstützten Modellierungskonstrukte (aus [HA05])

komplettes XML-Schema [BM04, TBMM04] zugewiesen, womit auch komplexe, hierarchisch strukturierte Daten unterstützt werden.


Zum benutzerfreundlichen Entwurf von Prozessmodellen bietet YAWL einen graphischen Editor, mit dem auch die unterstützten Korrektheitsanalysen durchgeführt werden können. Die Ausführung der so modellierten Prozesse wird davon getrennt mittels einer entsprechenden Serveranwendung realisiert. YAWL besitzt also genau wie ADEPT2 eine Client/Server-Architektur. Ein Überblick über die Architektur von YAWL findet sich in Abbildung 8.2 bzw. detaillierter in [AADH04].

Wie man bei einem Vergleich der den hier behandelten Konzepten entsprechenden Workflow Patterns mit den von YAWL unterstützten Patterns feststellt2, werden die den hier behandelten

2Eine Analyse, welche der neuen Pattern aus [RHAM06] der von YAWL unterstützt werden, ist beispielsweise
Konzepten entsprechenden Workflow Patterns – bis auf WCP-43 – alle von YAWL unterstützt. Daraus lässt sich folgern, dass die in dieser Arbeit beschriebenen Konzepte alle prinzipiell auch in YAWL modellierbar sind (bis auf einen Fall der flexiblen Endknoten). Dies ist im Großen und Ganzen auch der Fall, im Detail gibt es aber dennoch einige Unterschiede, die nachfolgend erläutert werden:

**Variable Parallelität** Die Umsetzung der Variablen Parallelität in YAWL ist sehr ähnlich zu der hier Vorgeschlagenen: Auch YAWL besitzt Mehrinstanzknoten, die die zugewiesene Aktivität mehrfach instanzieren. Bei YAWL gibt es allerdings zwei verschiedene Typen, einen Mehrinstanzknoten für elementare Aktivitäten (Multiple Instances of Atomic Tasks) und einen für Subprozesse (Multiple Instances of Composite Tasks). Außer der unterschiedlichen graphischen Darstellung und der Art der zugewiesenen Aktivität ist die Funktionsweise dieser beiden Typen allerdings weitgehend äquivalent.

Die initiale Anzahl von Instanzen wird bei YAWL aus den Eingabedaten abgeleitet und die Ausgabedaten der Instanzen werden wieder zu einem einzelnen Datum verschmolzen. Dies geschieht jedoch nicht vollautomatisch, sondern der Modellierer muss jeweils XQuery-Ausdrücke angeben, die die Zerlegung der Eingabe und das Zusammensetzen der Ausgabe erledigen.

Im Gegensatz zu den hier vorgestellten Mehrinstanzknoten ist es bei YAWL möglich, anzugeben, wie viele Instanzen minimal und wie viele Instanzen maximal erstellt werden dürfen. Außerdem kann ein *Continuation Threshold* angegeben werden, also die Zahl fertiger Instanzen, ab der der Mehrinstanzknoten weiterarbeitet, wenn nicht auf alle gewartet werden soll (die noch laufenden Instanzen werden in diesem Fall abgebrochen). Zudem ist es möglich, die Anzahl

---

wahlweise entweder wie hier statisch zu bestimmen oder es zu ermöglichen, dass jederzeit dynamisch neue Instanzen hinzugefügt werden können. Im letzteren Fall kann dann nach [KRAA06] der Benutzer über ein Webinterface entsprechendes XML für die Eingabedaten eingeben und mittels eines Button *Create new Instance* neue Instanzen anlegen.

**Lose Synchronisation mit Weiterlaufen** Eine Lose Synchronisation mit Weiterlaufen kann in YAWL ähnlich wie in Abbildung 8.3 modelliert werden.

Die in Abschnitt 5.2 beschriebenen Datenflussprobleme treten natürlich auch bei YAWL auf; diese werden dort allerdings für eine größere Kontrollflussflexibilität in Kauf genommen.

![Abbildung 8.3: Die Lose Synchronisation mit Weiterlaufen aus Abbildung 5.1 mit YAWL modelliert](image)

**Lose Synchronisation mit Abbruch** Eine Lose Synchronisation mit Abbruch kann in YAWL ähnlich wie in Abbildung 8.4 modelliert werden. Wie man erkennt, ist das Grundprinzip im Wesentlichen das selbe wie bei einer losen Synchronisation mit Weiterlaufen, nur dass die Slavezweige zum Abbruchbereich des Joinknotens für die Masterzweige hinzugefügt werden.

![Abbildung 8.4: Die Lose Synchronisation mit Abbruch aus Abbildung 5.6 mit YAWL modelliert](image)

**Flexible Endknoten** Laut [RHAE07] werden von YAWL Fehlerendknoten (WCP-20) unterstüzt, nicht jedoch gewöhnliche flexible Endknoten (WCP-43).
Nach [HA05] kann WCP-20, also ein Fehlerendknoten, wie in Abbildung 8.5 modelliert werden, indem einem Knoten einfach der gesamte Prozess als Abbruchbereich zugewiesen wird.

Abbildung 8.5: Ein Fehlerendknoten (WCP-20) mit YAWL modelliert (aus [HA05])

Da dieser Knoten jedoch nicht automatisch einen Fehler signalisiert, kann darüber eigentlich auch der hier beschriebene gewöhnliche flexible Endknoten modelliert werden (indem der Knoten keinen Fehler signalisiert).


³Beachte, dass der gewöhnliche flexible Endknoten auf keinen Fall WCP-20 entspricht, da WCP-20 eine nicht erfolgreiche Prozessausführung impliziert.
8.2. YAWL

**Abbildung 8.6:** Die verzögerte XOR-Entscheidung aus Abbildung 7.14 mit YAWL modelliert


**Abbildung 8.7:** Die ODER-Verzweigung aus Abbildung 7.17 mit YAWL modelliert


**Zusammenfassung** Insgesamt unterstützt YAWL im Vergleich zu ADEPT mehr Workflows und bietet eine größere Ausdrucksmächtigkeit bezüglich der modellierbaren Kontrollflussabläufe.

Auf der anderen Seite implementiert YAWL vergleichsweise wenige Korrektheitsanalysen zur Entwurfszeit, so dass viele Probleme – insbesondere beim Datenfluss – erst zur Laufzeit auffallen und dann zu Laufzeiteigenschaften und Prozessabbrüchen führen. Auch werden keine instanzenbezogenen Änderungen unterstützt und es ist (wenn man solche Änderungen unterstützen wollte) durch die relativ freie Graphstruktur und den tokenbasierten Ansatz verhältnismäßig schwer
festzustellen, ob ein gegebener Knoten bereits ausgeführt wurde oder nicht, und damit, ob eine bestimmte Änderung noch zulässig ist.

Da erst zur Laufzeit auftretende Fehler in der Praxis sehr ärgerlich und teuer sind, stellt sich daher die Frage, ob die größere Ausdrucksmächtigkeit von YAWL diesen Nachteil wirklich ausgleicht, oder ob die geringere Ausdrucksmächtigkeit von ADEPT bei einer weit höheren Korrektheit und Robustheit nicht in vielen Fällen doch der bessere und brauchbarere Ansatz ist.4

8.3. WS-BPEL 2.0

BPEL ist eine XML-basierte Beschreibungssprache zur „Orchestrierung“ von Web Services, d.h. eine Sprache, die es ermöglicht, Web Services ähnlich wie Aktivitäten in ADEPT zu benutzen. Die Aufrufe der Web Services können also parallel und sequentiell angeordnet werden, es können Schleifen und bedingte Verzweigungen modelliert werden, usw. (vgl. auch [RS04]).


Die Syntax von BPEL ist grundsätzlich blockorientiert, mit ähnlichen grundlegenden Konstrukten (bei BPEL „Aktivitäten“ genannt) wie bei ADEPT: Es gibt eine Sequenz (sequence), bedingte Verzweigungen (if), Parallelverzweigungen (flow) und Schleifen (while und repeat-Until) (und natürlich noch einige weitere speziellere Aktivitäten).

Innerhalb von flow-Aktivitäten sind beliebige Kanten („Links“) zwischen den normalerweise parallel ausgeführten Untertätigkeiten möglich, um eine bestimmte Ausführungsreihenfolge festzulegen, ähnlich wie (harte) Sync-Kanten in ADEPT. Anders als bei Sync-Kanten können

4Dabei ist auch zu bedenken, dass nach [RHAM06] viele kommerziell erfolgreiche Systeme nur eine nicht allzu große Teilmenge (oft weniger als 50%) der Workflow Patterns unterstützen, also keine besonders hohe Ausdrucksmächtigkeit aufweisen.

5Es gibt allerdings eine „BPEL4People“ genannte Erweiterung, die dies tut [AAD+07].
für diese Links aber Transitionsbedingungen angegeben werden und am Zielknoten kann eine Joinbedingung definiert werden (z.B. Aktivierung erst wenn alle oder schon wenn nur eine eingehenden Kante = true ist). Damit können XOR- und AND-Splits (bzw. -Joins) auch vollständig innerhalb einer flow-Aktivität modelliert werden (und nicht nur mittels der zusätzlich vorhandenen blockorientierten Konstrukte).


Subprozesse werden von BPEL nicht direkt unterstützt – als hierarchisches Strukturierungselement sind lediglich die Scopes verfügbar. Allerdings ist jeder BPEL-Prozess selbst wieder ein aufrufbarer Webservice, kann also insbesondere Ein- und Ausgabeparameter besitzen, womit indirekt ein hierarchischer Aufruf von Prozessen möglich ist.

Die hier beschriebenen erweiterten Konzepte werden von BPEL teilweise unterstützt. Nachfolgend werden die Gemeinsamkeiten und Unterschiede im Detail beschrieben:

**Variable Parallelität** Eine variable Parallelität wird in WS-BPEL 2.0 direkt über die forEach-Aktivität unterstützt.


Im Unterschied zu ADEPT (aber ähnlich wie YAWL) kann eine forEach-Aktivität eine completionCondition besitzen. Diese wird ausgewertet, sobald eine Instanz beendet wurde. Falls die Auswertung true ergibt, wird die Ausführung der Aktivität (inklusive aller anderen gestarteten Instanzen im Parallelfall) vorzeitig abgebrochen.

Indizierte Zugriffe auf listenwertige Daten außerhalb des Scopes können mittels der Zähllvariablen vorgenommen werden; eine direkte Zuweisung von Listenelementen findet nicht statt (was aber keine Funktionalitätseinschränkung darstellt).
Lose Synchronisation Eine lose Synchronisation in genau der hier beschriebenen Semantik wird von BPEL nicht unterstützt.


Flexible Endknoten Beide Typen flexibler Endknoten werden direkt unterstützt: Es gibt analog zum normalen flexiblen Endknoten eine exit-Aktivität, die die gesamte Prozessausführung abbricht und analog zum Fehlerendknoten eine throw-Aktivität, mittels der ein Fehler signalisiert werden kann.

Die exit-Aktivität beendet bei BPEL immer hart den gesamten Prozess. Bei der throw-Aktivität hingegen werden nur diejenigen Scopes abgebrochen, die den entsprechenden Fehler nicht abfangen. Wenn kein Scope (einschließlich des Prozess-Scopes) dies tut, wird der gesamte Prozess abgebrochen (analog zum Exceptionhandling in Java [GJSB05]).

Abhängige XOR-Entscheidungen Da abhängige XOR-Entscheidungen im Wesentlichen bei Vorhandensein einer statischen Datenflußanalyse von Interesse sind und BPEL eine solche nicht bietet, wird dieses Konzept von BPEL nicht unterstützt.


Zusammenfassung  Insgesamt bietet BPEL eine etwa vergleichbare (direkte) Ausdrucksmächtigkeit wie ADEPT, aber kaum Prüfungen der Datenflusskorrektheit, keine instanzspezifischen Änderungen und keine standardisierte graphische Darstellung. Aufgrund der eher informellen Definition von BPEL in [IEA+07] ist es darüber hinaus schwierig, Korrektheitsprüfungen und korrekte instanzspezifische Änderungen sauber zu definieren (vgl. in diesem Zusammenhang auch [RR06]).

Nichtsdestotrotz kann BPEL als Industriestandard nicht einfach ignoriert werden und stellt unabhängig davon einen guten „kleinsten gemeinsamen Nenner“ für die in einem Workflowsystem im Zusammenhang mit Web Services zu unterstützenden, sinnvollen Kontrollflusskonzepte dar. Zudem gibt es mittlerweile einige Workflowsysteme, die BPEL nativ ausführen können. Diese Implementierungen nehmen dabei oft Erweiterungen gegenüber dem Standard vor, um fehlende, aber sinnvolle Funktionalität zu ergänzen, so dass die im vorigen Absatz genannten Einschränkungen dort ggf. nicht in vollem Umfang zum Tragen kommen.


Darüber hinaus ist es selbstverständlich auch denkbar, BPEL in einer konkreten Implementierung durch Korrektheitsprüfungen und instanzspezifische Änderungen im Stile von ADEPT zu ergänzen, beispielsweise durch einen Ansatz wie in [RR06].

8.4. Fazit

Die beiden in diesem Kapitel untersuchten Workflowsprachen (YAWL und WS-BPEL 2.0) unterstützen die meisten der hier vorgestellten Konzepte in der ein oder anderen Form bereits, ADEPT kann also durch die in dieser Arbeit beschriebenen Konzepte seine direkte Ausdrucksmächtigkeit im Vergleich zu den beiden untersuchten Systemen verbessern. Im Fall von BPEL ist dies besonders wichtig, da es für den Praxiseinsatz langfristig unumgänglich sein wird, einen zufriedenstellenden Import von Prozessen zu unterstützen, die in diesem Industriestandard modelliert wurden. Durch eine direkte Unterstützung der in BPEL vorhandenen Konstrukte wird ein solcher Import natürlich deutlich erleichtert. Allerdings müssen auch nach Implementierung aller Erweiterungen aus dieser Arbeit für eine Unterstützung aller in BPEL vorhandenen Konstrukte (insbesondere Ereignisse) noch einige weitere Erweiterungen von ADEPT vorgenommen werden müssen.

6Beim Datenfluss kommt hinzu, dass Datenzugriffe über beliebige benutzerdefinierte Ausdrücke in einer (potentiell) beliebigen Sprache definiert werden, was Analysen deutlich erschwert.

7http://de.wikipedia.org/wiki/Business_Process_Execution_Language#BPEL_Engines (Stand 13.05.08) bietet einen kurzen Überblick.
Auf der anderen Seite kann man auch erkennen, dass statische Korrektheitsanalysen des Datenflusses und instanzspezifische Änderungen in anderen Systemen weiterhin größtenteils nicht unterstützt werden. In dieser Hinsicht weist ADEPT daher weiterhin einen großen Vorsprung gegenüber vergleichbaren Systemen auf (bzw. kann diesen Vorsprung durch die Unterstützung abhängiger XOR-Verzweigungen sogar noch ausbauen).
9. Zusammenfassung und Ausblick

Dieses Kapitel fasst die zentralen Aussagen dieser Arbeit in Abschnitt 9.1 kurz zusammen und gibt in Abschnitt 9.2 einen Ausblick auf noch offene Fragestellungen, die in dieser Arbeit nicht abschließend behandelt werden konnten.

9.1. Zusammenfassung

Mit ADEPT war es auch schon bislang ohne Schwierigkeiten möglich, die meisten in der Praxis vorkommenden Prozesse zu modellieren. Im Laufe der Zeit sind jedoch Anforderungen aus der Praxis aufgefallen, die sich mit den bisher vorhandenen Konstrukten des ADEPT-Metamodells nur schwer modellieren ließen. Ziel dieser Arbeit war daher, zu untersuchen, wie sich eine Unterstützung für einige dieser Anforderungen in das ADEPT-Metamodell integrieren lässt und welche Herausforderungen (und Lösungen dafür) es bei der konkreten Implementierung geben kann.

Konkret lassen sich die untersuchten Praxisanforderungen durch Unterstützung der folgenden Konzepte erfüllen:

- **Variable Parallelität bzw. Multiinstanzaktivitäten:**

- **Lose Synchronisation paralleler Verzweigungen:**
  Bei einer losen Synchronisation werden am Ende von Parallelverzweigungen nicht alle Zweige synchronisiert, sondern es gibt einige ausgezeichnete Zweige, auf die nicht gewartet wird. Diese Zweige können dann entweder bis zu ihrem Ende weiterlaufen oder aber beim
7. Zusammenfassung und Ausblick

Schalten des Vereinigungsknotens abgebrochen werden.
Wenn die Zweige bis zu ihrem Ende weiterlaufen, ergeben sich einige schwerwiegende Probleme mit der Datenflusskorrektheit, so dass diese Semantik in ADEPT nicht sinnvoll umsetzbar ist. Im Gegensatz dazu ist ein Abbruch der Zweige unproblematisch und kann über die Einführung eines speziellen Vereinigungsknotens realisiert werden. Darüber hinaus sind Erweiterungen denkbar, die die nicht synchronisierten Zweige vom Rest des Prozesses entkoppeln (so dass diese Zweige keine vom Rest des Prozesses lesbaren Daten mehr schreiben können). Durch ein solches Entkoppeln kann ein Weiterlaufen ermöglicht werden, ohne dass die Datenflusskorrektheit gefährdet ist.

- **Flexible Endknoten:**
Flexible Endknoten sind aus Benutzersicht Endknoten, die an jeder beliebigen Stelle des Prozesses und nicht nur an seinem Ende eingefügt werden können und bei ihrer Aktivierung den gesamten Prozess abbrechen. Da es in der Praxis häufig vorkommt, dass die Ursache für einen solchen Abbruch eine Fehlsituation ist, wurden zwei Typen von Endknoten definiert, gewöhnliche flexible Endknoten, die für eine erfolgreiche Beendigung des Prozesses sorgen, und Fehlerendknoten, die einen fehlerbedingten Abbruch implementieren.
Flexible Endknoten sind in ADEPT durch Einführung entsprechender neuer Knotentypen problemlos realisierbar. Die einzige Herausforderung ist, dass manche Nachfolger flexibler Endknoten niemals ausgeführt werden, also „tot“ sind, was einige Auswirkungen auf den Datenfluss hat. Diese Auswirkungen können auf einfache Weise durch eine Erweiterung der bisherigen Datenflussanalyse berücksichtigt werden.

- **Abhängige XOR-Entscheidungen:**
In dieser Arbeit wurde eine Erweiterung des bisherigen Algorithmus zur Datenflussanalyse vorgestellt, die die Auswirkungen abhängiger XOR-Verzweigungen berücksichtigt, wodurch viele bislang fälschlich als unsicher abgelehnte Datenflüsse als korrekt akzeptiert werden.

- **Verzögerte XOR-Entscheidungen:**

- **ODER-Verzweigungen:**
Eine ODER-Verzweigung implementiert eine bedingte Auswahl, bei der mehrere Nach-
folger (und nicht nur ein Nachfolger wie bei XOR-Verzweigungen) ausgewählt werden können. Diese Semantik lässt sich auf einfache Weise mittels der bereits vorhandenen AND- und XOR-Verzweigungen modellieren, es ist also nicht nötig, eine Metamodeller-weiterung zu definieren.

Allerdings ist es zur Steigerung der Benutzerfreundlichkeit angebracht, diese Modellierung nicht manuell vom Benutzer vornehmen zu lassen, sondern dies automatisch durchzuführen und dem Benutzer über eine Sicht entsprechende ODER-Knoten zu präsentieren.

Insgesamt hat diese Arbeit also gezeigt, dass sich alle untersuchten Konzepte – ggf. mit leichten Einschränkungen wie bei der loser Synchronisation – im Rahmen des ADEPT-Metamodells umsetzen lassen.

Allerdings können die meisten Konzepte nicht allein mittels der bisherigen Metamodellkonstrukte modelliert werden, sondern es mussten Erweiterungen des Metamodells vorgenommen werden. Entweder ist eine Umsetzung mit bisherigen Konstrukten einfach prinzipiell nicht möglich (wie bei verzögerten XOR-Entscheidungen), oder dies ist zwar theoretisch machbar, aber für den Praxiseinsatz zu komplex (wie bei den abhängigen XOR-Entscheidungen) oder zu ineffizient (wie bei variabler Parallelität).

Das einzige Konzept, das problemlos nur mittels der bisherigen Konstrukte modelliert werden kann, ist die ODER-Verzweigung. Aber auch in diesem Fall ist es sinnvoll, zumindest den Prozesseditor so zu erweitern, dass der Benutzer eine solche Modellierung nicht selbst vornehmen muss, sondern dass er direkt ODER-Knoten verwenden kann und die Transformation in eine Darstellung mit den bisherigen Konstrukten benutzerfreundlich automatisch im Hintergrund geschieht.


Genauso ist es problemlos möglich, die hier beschriebenen Konzepte sukzessive einzeln im ADEPT2-System zu implementieren; es ist nicht so, dass eines der neuen Konzepte als zwingende Grundvoraussetzung auf die Unterstützung eines anderen aufbaut.

Schließlich hat es sich gezeigt, dass die untersuchten Konzepte (bis auf die abhängigen XOR-Entscheidungen) auf Kontrollflussseite in den betrachteten anderen Workflowsprachen (YAWL und WS-BPEL 2.0) bereits mehr oder weniger unterstützt werden, diese aber die Korrektheit des Datenflusses weiterhin nur am Rande berücksichtigen.

In diesem Lichte ist auch die fehlende Unterstützung für abhängige XOR-Entscheidungen in YAWL und BPEL nicht verwunderlich: Solange die Sicherheit des Datenflusses nicht (oder kaum) überprüft wird, ist die Existenz abhängiger XOR-Entscheidungen nur eher von untergeordnetem Interesse, da diese sich aus der reinen Kontrollflussicht kaum von unabhängigen XOR-Entscheidungen unterscheiden. Wenn man wie bei ADEPT aber auch den Datenfluss (und insbesondere die Sicherheit desselben) berücksichtigt, ergeben sich durchaus bemerkenswerte Auswirkungen – wie diese Arbeit gezeigt hat.
Auch im Kontext der Workflow Patterns erhöht eine Implementierung der untersuchten Konzepte die Ausdrucksmächtigkeit von ADEPT erheblich. Statt wie vorher nur 6 voll und 5 teilweise unterstützten Kontrollflusspattern (von 43) werden nun 13 Pattern voll und 4 Pattern teilweise von ADEPT unterstützt.

Insgesamt stellen die vorgestellten Erweiterungen eine Ergänzung zu ADEPT dar, mit Hilfe derer die Ausdrucksmächtigkeit deutlich erhöht werden kann, ohne die Korrektheit des Datenflusses zu beeinträchtigen (im Gegensatz zu den untersuchten anderen Systemen). Durch die Unterstützung abhängiger XOR-Entscheidungen wird die Analyse des Datenflusses sogar noch verfeinert. ADEPT bleibt also auch mit diesen Erweiterungen weiterhin eines der wenigen Systeme, die es schaffen, eine hohe Ausdrucksmächtigkeit mit einer garantierten Korrektheit des Prozesses zu kombinieren.

9.2. Ausblick

Nichtsdestotrotz gibt es noch einige offene Fragestellungen, die in dieser Arbeit nicht endgültig geklärt werden konnten.

Zuallererst muss für eine sinnvolle Implementierung der variablen Parallelität untersucht werden, wie Listen - und insbesondere eine Datenflussanalyse dafür - in ADEPT implementiert werden können. Im Rahmen der variablen Parallelität wurde dieses Thema hier zwar schon teilweise behandelt und einige Anforderungen an die Listen definiert, dennoch bleibt insbesondere die genaue Implementierung indizierter Zugriffe außerhalb der variablen Parallelität weiterhin unklar.

Ein weiterer Punkt ist, welche Möglichkeiten es gibt, eine lose Synchronisation mit Weiterläufen einiger Zweige auf eine mit einem korrekten Datenfluss verträgliche Weise in ADEPT zu implementieren. Wie hier gezeigt wurde, ist eine direkte, naive Implementierung aufgrund von auftretenden Nichtdeterminismen im Datenfluss kaum zu gebrauchen.

Eine Implementierung einer externen Abbruchsemantrik ist dagegen problemlos machbar, allerdings sind hier Schattenkopien der Markierungen von Knoten und des Wertes von Datenelementen nötig. Bevor diese Semantik konkret umgesetzt werden kann, muss daher noch geklärt werden, wie solche Schattenkopien am besten im ADEPT2-System implementiert werden können.

Eine weitere Möglichkeit ist es, die Slavezweige vollständig vom Rest des Prozesses zu entkoppeln, so dass diese keinerlei Daten an den Rest des Prozesses schreiben können. Hierbei muss aber noch im Detail untersucht werden, ob dies nicht doch an einigen Stellen zu Problemen führt, ob es überhaupt Szenarien in der Praxis gibt, für die eine solche Semantik sinnvoll ist, und nicht zuletzt, wie man ein solches Entkoppeln sinnvoll implementieren kann.

Was hier auch nicht abschließend geklärt werden konnte, ist, welche Möglichkeiten es gibt, den Vergleich der beiden Vorbedingungen in abhängigen XOR-Entscheidungen effizienter als hier beschrieben zu gestalten. Der vorgestellte, etwas naive Algorithmus ist exponentiell in der Schachtelungstiefe der abhängigen XOR-Verzweigungen. Hier ist unklar, ob diese Exponentialität wirklich diesem Problem inhärent ist oder ob diese Überprüfung auch in polynomieller
9.2. Ausblick

Zeit möglich ist. Falls die Exponentialität nicht vermieden werden kann, stellt sich die Frage, mittels welcher Algorithmen diese so klein wie möglich gehalten werden kann\(^1\).

Nicht zuletzt ist es für die Implementierung von ODER-Verzweigungen und die variable Parallelität sinnvoll, wenn ein Sichtenkonzept für Workflows existiert, so dass dem Benutzer eine benutzerfreundlichere Sicht als die interne Implementierung präsentiert werden kann. Für die beiden hier genannten Spezialfälle ist dies zwar verhältnismäßig einfach zu realisieren, letztlich ist aber ein generelles Konzept deutlich sinnvoller. Hierbei ist es insbesondere interessant, wie die Markierungen zur Laufzeit auf die in der Sicht sichtbaren Knoten übertragen werden können, wie instanzspezifische Änderungen auf solchen Sichten realisiert werden können und nicht zuletzt, wie eine solche Sicht (ggf. sogar durch den Modellierer selbst) auf eine generische Weise spezifiziert werden kann.


\(^1\)Vgl. z.B. den Unterschied zwischen $O(2^n)$ für den naive Algorithmus vs. ca. $O(1.4^n)$ für optimierte Algorithmen, die das Erfüllbarkeitsproblem der Aussagenlogik (SAT) lösen [Sch01].
9. Zusammenfassung und Ausblick
Literaturverzeichnis


[AAD+07] Agrawal, Ashish; Amend, Mike; Das, Manoj; Ford, Mark; Keller, Chris; Kloppmann, Matthias; König, Dieter; Leymann, Frank; Müller, Ralf; Pfaus, Gerhard; Plösser, Karsten; Rangaswamy, Ravi; Rickayzen, Alan; Rowley, Michael; Schmidt, Patrick; Trickovic, Ivana; Yiu, Alex; Zeller, Matthias: WS-BPEL Extension for People (BPEL4People), Version 1.0, Juni 2007. https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/30c6f5b5-ef02-2a10-c8b5-cc1147f4d58c


[BBE+07] Barreto, Charlon; Bullard, Vaughn; Erl, Thomas; Evdemon, John; Jordan, Diane; Kand, Khanderao; König, Dieter; Moser, Simon; Stout, Ralph; Ten-Hove, Ron; Trickovic, Ivana; Rijn, Danny van d.; Yiu, Alex; Organization for the Advancement of Structured Information Standards (OASIS) (Hrsg.): Web Service Business Process Execution Language Version 2.0 - Primer. : Organization for the Advancement of Structured Information Standards (OASIS), Mai 2007. http://docs.oasis-open.org/wsbpel/2.0/Primer/ wsbpel-v2.0-Primer.pdf


[Ber05] Berroth, Marco: Konzeption und Entwurf einer Komponente für Organisationsmodelle, Universität Ulm, Diplomarbeit, 2005


[BRB07] Bobrik, Ralph ; Reichert, Manfred ; Bauer, Thomas: Parameterizable Views for Process Visualization / University of Twente. 2007. – Technical Report


[DRRA05] Dam, Peter; Reichert, Manfred; Rinderle, Stefanie; Atkinson, Colin: Auf dem Weg zu prozessorientierten Informationssystemen der nächsten Generation – Herausforderungen und Lösungskonzepte. In: Tagungsband doIT-Forschungstag, 2005

[GHJV94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns. Addison-Wesley, 1994


[JEAV07] Jordan, Diane; Evdemon, John; Alves, Alexandre; Arkin, Assaf; Askary, Sid; Barreto, Charlton; Bloch, Ben; Curbera, Francisco; Ford, Mark; Goland, Yaron; Guizar, Alejandro; Kartha, Neelakantan; Liu, Canyang K.; Khalaf, Rania; König, Dieter; Marin, Mike; Mehta, Vinod; Thatte, Satish; Rijn, Danny van d.; Yendluri, Prasad; Yiu, Alex; Organization for the Advancement of Structured Information Standards (OASIS) (Hrsg.): Web Service Business Process Execution Language Version 2.0 –


[Rei00] REICHERT, Manfred: Dynamische Ablaufänderungen in Workflow-Management-Systemen, Universität Ulm, Diss., Mai 2000


Literaturverzeichnis


[RRD04] Rinderle, Stefanie; Reichert, Manfred; Dadam, Peter: Correctness criteria for dynamic changes in workflow systems—a survey. In: Data & Knowledge Engineering 50 (2004), Nr. 1, S. 9–34


[Ste05] Steinle, Martin: Einsatz von Workflow Engines zur effizienten Anpassung standardisierter Branchensoftware an unterschiedliche Kundenanforderungen, Universität Ulm, Masterarbeit, Oktober 2005

Abbildungsverzeichnis

2.1. Prozessvorlage und mehrere Instanzen .............................................. 14
2.2. Schematische Übersicht über das ADEPT-Metamodell ............................. 15
2.3. Nach den Korrektheitsregeln ungültige Workflowgraphen (ohne Schleifen; aus [Rei00]) .......................................................... 20
2.4. Knoten- und Kantennarkierungen in ADEPT (aus [Rei00]) ....................... 21
2.5. Zustandsübergänge zur Laufzeit (aus [Rei00], vereinfacht) .................... 22
2.6. Graphische Darstellung der hier verwendeten Metamodelllemente .......... 24
2.7. Beispiel für die in einem ADEPT-Prozess verwendeten Workflow Patterns ... 25

3.1. Konzeptuelle Idee hinter variabler Parallelität (nach [Beu02]) ............... 29
3.2. Neu eingeführte Notationselemente .................................................. 30
3.3. Beispiel für lose Synchronisation ..................................................... 31
3.4. Für die lose Synchronisation verwendete Notationen ............................ 32
3.5. Beispiel für flexible Endknoten ....................................................... 33
3.6. Beispiel für abhängige XOR-Entscheidungen ...................................... 34
3.7. Beispiel einer verzögerten XOR-Entscheidung .................................... 36
3.8. Beispiel einer ODER-Verzweigung (nach [RHAM06]) ............................. 37

4.1. Beispiel für variable Parallelität (angelehnt an [Bre06]) ....................... 39
4.2. Varianten der Modellierung des vervielfältigten Teils ........................... 41
4.3. Varianten der Bestimmung der Instanzzahl ........................................ 45
4.4. Varianten der Datenübergabe am Beispiel eines Subprozesses ............... 48
4.5. Varianten der Bearbeiterzuordnung .................................................. 50
4.6. Symbol für einen Mehrinstanzknoten .............................................. 54
4.7. Ausführungszustände eines Mehrinstanzknoten ................................... 55
4.8. Beispiel für das Parametermapping zur Entwurfszeit ............................ 58
4.9. Beispiel für das Parametermapping zur Laufzeit .................................. 59
4.10. Indizierter Zugriff auf eine Liste .................................................. 60
4.11. Variable Parallelität mit mehreren Eingabelisten ............................... 62
4.12. Varianten der Bestimmung der Instanzzahl ...................................... 64

5.1. Beispiel für eine lose Synchronisation paralleler Verzweigungen ........... 70
5.2. Modellierung einer losen Synchronisation mit Hilfe der Workflow Patterns 72
5.3. Lose Synchronisation paralleler Verzweigungen: Probleme bei Schleifen ... 73
5.4. Mögliches Überschreiben von Daten bei loser Synchronisation ................ 75
5.5. Nichtdeterminismus des Werts von Datenelementen bei loser Synchronisation 76
5.6. Kontrollflussablauf bei einem AND-Join mit Abbruch .......................... 79
5.7. Synchronisationskanten bei einem AND-Join mit Abbruch ..................... 80
5.8. Datenfluss bei einem AND-Join mit partiellem Abbruch ........................ 82
5.9. Äquivalenz zwischen dem Abbruch von Slavezweigen und dem Weiterlaufen ohne Daten zu schreiben .................................................. 83

6.1. Beispiel für den Einsatz flexibler Endknoten .................................. 87
6.2. Modellierung des Prozesses in Abbildung 6.1 ohne flexible Endknoten .... 88
6.3. Flexible Endknoten in Parallelverzweigung ........................................... 88
6.4. a) Flexible Endknoten und b) Fehlerendknoten .................................... 90
6.5. Beispiel für tote Knoten ...................................................................... 93
6.6. Tote Knoten aufgrund einer Synchronisationskante ................................ 94
6.7. Tote Knoten bei einem flexiblen Endknoten direkt hinter einem AND-Split ... 94
6.8. Modellierung einer finalen Auswahl mit flexiblen Endknoten .................. 96
6.9. Modellierung eines AND-Join mit partiellem Abbruch mit flexiblen Endknoten 96

7.1. Abhängige XOR-Entscheidungen ......................................................... 100
7.2. Der transformierte Prozess aus Abbildung 7.1 ...................................... 101
7.3. Beispiel für die Abhängigkeit von Entscheidungen ................................. 103
7.4. Beispiel für eine Datenflussanalyse mit Vorbedingungen ....................... 104
7.5. Datenflussanalyse mit Elimination unausführbarer Pfade ....................... 105
7.6. Problematischer Prozess bei Elimination unausführbarer Pfade ............... 105
7.7. Der Lesezugriff an Knoten J ist sicher versorgt .................................... 109
7.8. Beispiel für die Bestimmung der Vorbedingungen .................................. 110
7.9. Mehrere Schreibzugriffe auf ein Entscheidungsdatenelement .................. 112
7.10. Bestimmung logischer Datenelemente ............................................... 113
7.11. Beispiel für eine erweiterte Datenflussanalyse ..................................... 117
7.12. Beispiel parallele abhängige XOR-Entscheidungen ............................. 118
7.13. Beispiel für abhängige Schleifenendknoten ....................................... 118
7.14. Verzögerte XOR-Entscheidungen ..................................................... 123
7.15. Zustandsdiagramm für verzögerte XOR-Entscheidungen ...................... 127
7.16. Äquivalenz expliziter und impliziter Auswahl .................................... 128
7.17. Beispiel für eine ODER-Verzweigung .............................................. 131
7.18. Abbildung 7.17 mittels XOR außen und AND innen modelliert ................ 134
7.19. Abbildung 7.17 mittels AND außen und XOR innen modelliert ............... 136
7.20. Transformation einer ODER-Verzweigung .......................................... 139
7.21. Äquivalenz der Datenflussbeschränkungen in untransformierter und transformierter Form .................................................. 141

8.1. Übersicht über die von YAWL unterstützten Modellierungskonstrukte (aus [HA05]) 149
8.3. Die Lose Synchronisation mit Weiterlaufen aus Abbildung 5.1 mit YAWL modelliert ................................................................. 151
8.4. Die Lose Synchronisation mit Abbruch aus Abbildung 5.6 mit YAWL modelliert 151
8.5. Ein Fehlerendknoten (WCP-20) mit YAWL modelliert (aus [HA05]) .......... 152
8.6. Die verzögerte XOR-Entscheidung aus Abbildung 7.14 mit YAWL modelliert 153
8.7. Die ODER-Verzweigung aus Abbildung 7.17 mit YAWL modelliert ............ 153
# Tabellenverzeichnis

2.1. Bislang unterstützte Workflow Patterns ........................................ 28

4.1. Übersicht über die Umsetzungsvarianten .................................. 53
4.2. Zu unterstützende Listenoperationen ......................................... 56
4.3. Typen des Parametermappings ............................................... 58
4.4. Möglichkeiten zur Bestimmung der Instanzanzahl ..................... 65
4.5. Unterstützte Workflow Patterns ................................................. 68

5.1. Im Rahmen loser Synchronisation relevante Workflow Patterns .......... 71
5.2. Unterstützte Workflow Patterns .............................................. 77
5.3. Unterstützte Workflow Patterns .............................................. 86

6.1. Unterstützte Workflow Patterns .............................................. 98

7.1. Zustandsübergänge und Markierungsregeln bei verzögerten XORs .... 126
7.2. Unterstützte Workflow Patterns .............................................. 130
7.3. Unterstützte Workflow Patterns .............................................. 143

8.1. Mit den Erweiterungen aus dieser Arbeit unterstützte Workflow Patterns . 147

B.1. In den Listings B.1 bis B.5 verwendete Definitionen ..................... 178
B.2. In den Listings B.6 bis B.8 verwendete Definitionen ..................... 186
A. Glossar


**Datenelement, Variable:** Ein Konstrukt im Metamodell, in dem zur Laufzeit Daten eines bestimmten Datentyps abgelegt werden. Variablen können (je nach System) entweder immer global im gesamten Prozess sichtbar sein oder aber einem bestimmten Sichtbarkeitsbereich zugeordnet sein. Datenelemente in ADEPT sind dagegen immer global sichtbar. Oft muss in einem Workflowsystem jedem Parameter einer Aktivität eine Variable zugewiesen werden (die Aktivitäten tauschen ihre Daten also nicht direkt aus, sondern indirekt über die Variablen bzw. Datenelemente).

**Elementare Aktivität:** Eine elementare Aktivität stellt eine einzelne, atomare Aktion dar. „Atomar“ bedeutet in diesem Zusammenhang, dass die Aktivität nicht aus anderen Elementen des Workflowmetamodells zusammengesetzt ist, sondern in einer anderen Programmiersprache implementiert ist – die durchgeführte Aktion kann aber durchaus sehr komplex sein.

**Entwurfszeit:** Der Zeitraum, in dem das Prozessmodell bearbeitet wird. Zu diesem Zeitpunkt sind i.d.R. beliebige Änderungen des Modells möglich, da nicht auf die Ausführungshistorie der Instanz Rücksicht genommen werden muss. Ein „Prozess zur Entwurfszeit“ ist daher normalerweise ein Prozessmodell.

**Instanzspezifische Änderung:** Eine strukturelle Änderung, die zur Laufzeit an einer Prozessinstanz (und nicht zur Entwurfszeit am entsprechenden Workflowschema) durchgeführt wird. Bei Unterstützung von instanzspezifischen Änderungen kann eine Prozessinstanz demnach strukturell von der entsprechenden Vorlage abweichen. Dennoch unterliegen instanzspezifische Änderungen stärkeren Einschränkungen als Änderungen zur Entwurfszeit, da nur solche instanzspezifischen Änderungen erlaubt sind, die den bisherigen Prozessablauf nicht unmöglich machen (es dürfen also beispielsweise keine bereits ausgeführten Knoten gelöscht oder neue Knoten vor bereits ausgeführten eingefügt werden).

**Laufzeit:** Der Zeitraum, in dem eine Prozessinstanz ausgeführt wird. Änderungen sind zur Laufzeit i.d.R. nur über instanzspezifische Änderungen (oder gar nicht) möglich. Der Begriff eines „Prozesses zur Laufzeit“ bezieht sich normalerweise auf eine Prozessinstanz.
**Metamodell:** Ein „Metamodell“ ist ein Modell, das andere Modelle beschreibt. Es definiert also insbesondere die in den beschriebenen Modellen verfügbaren Konstrukte, die Regeln, nach denen aus diesen ein gültiges Modell zusammengebaut werden kann, und nicht zuletzt die Semantik der Modellkonstrukte. Beispiele dafür sind das ADEPT-Metamodell (siehe Abschnitt 2.1) oder das UML-Metamodell [OMG07b, OMG07a].

**Prozessinstanz:** Eine konkrete Ausprägung eines Prozessmodells, das gerade ausgeführt wird. Eine Prozessinstanz weist typischerweise zusätzliche Markierungen auf, die den Ausführungsstatus der einzelnen Aktivitäten und die konkreten Datenwerte von Datenelementen enthalten. Zu jedem Prozessmodell kann es beliebig viele Instanzen mit unterschiedlichem Ausführungsstatus, Daten, usw. geben.

**Prozessmodell, Workflowschema, Prozessvorlage:** Ein Prozessmodell oder Workflowschema ist ein aus den Konstrukten eines entsprechenden Metamodells modellierter Ablauf. Typischerweise ist ein Prozessmodell als (gerichteter, knoten- und kantenmarkierter) Graph definiert und kann daher auf einfache Weise graphisch dargestellt werden. Zur Laufzeit wird eine Instanz des Prozessmodells vom Workflowsystem interpretiert, um den modellierten Ablauf auszuführen\(^1\).

**Subprozess, Subworkflow:** Ein Prozessmodell, das allerdings (i.d.R.) nicht selbständig ausführbar ist, sondern dafür gedacht ist, wie eine Aktivität aus einem anderen Prozessmodell heraus aufgerufen zu werden. Ein Subprozess ist also in etwa das Äquivalent zu einem Unterprogramm in anderen Programmiersprachen. Ein Subprozess besitzt typischerweise mehrere Ein- und Ausgabeparameter, über die Daten vom bzw. zum Aufrufer übergeben werden können.

**Workflowsystem:** Software, die die Semantik der Konstrukte eines Workflownmetamodells versteht und somit Prozessmodelle ausführen kann. Bietet neben der eigentlichen Ausführungsunterlage meistens zusätzlich noch weitere Werkzeuge, wie z.B. graphische Editoren zur Erstellung und Bearbeitung von Workflownmodellen oder Überwachungswerkzeuge für die Workflownausführung zur Laufzeit.

---

\(^1\) Man kann ein Prozessmodell (mit Einschränkungen) also auch als graphische Darstellung eines Programms sehen und die Prozessmodellierung dementsprechend als graphische Programmierung.
B. Algorithmen


B.1. Variable Parallelität

In diesem Abschnitt werden die zur Implementierung der variablen Parallelität benötigten Algorithmen als Pseudocode dargestellt. Dabei werden die nachfolgend in Tabelle B.1 aufgeführten Variablen, Eigenschaften und Methoden mit der dort angegebenen Semantik verwendet. Desweiteren wird davon ausgegangen, dass das Mapping bereits gültig ist, das heisst, dass keine ungültigen Werte für den Mappingtyp angetroffen werden, und, dass die Listenoperationen aus Tabelle 4.2 definiert sind.

<table>
<thead>
<tr>
<th>Ausdruck</th>
<th>Semantik</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignedActivityTemplate</td>
<td>Das dem Mehrinstanzknoten zugewiesene Aktivitätstemplate.</td>
</tr>
<tr>
<td>InputParams</td>
<td>Die Menge aller Eingabeparameter der zugewiesenen Aktivität (die inneren Eingabeparameter).</td>
</tr>
<tr>
<td>OutputParams</td>
<td>Die Menge aller Ausgabeparameter der zugewiesenen Aktivität (die inneren Ausgabeparameter).</td>
</tr>
<tr>
<td>InputParamsOuter</td>
<td>Die Menge der Eingabeparameter des Mehrinstanzknotens (die äußeren Eingabeparameter).</td>
</tr>
<tr>
<td>OutputParamsOuter</td>
<td>Die Menge der Ausgabeparameter des Mehrinstanzknotens (die äußeren Ausgabeparameter).</td>
</tr>
<tr>
<td>createInstance(template)</td>
<td>Erstellt eine neue Instanz des angegebenen Templates.</td>
</tr>
<tr>
<td>waitForAnyStateChange(instanceList)</td>
<td>Wartet darauf, dass sich der Ausführungsstatus irgendeiner Instanz aus der übergebenen Liste ändert und gibt diese Instanz zurück.</td>
</tr>
<tr>
<td>[instance.]state</td>
<td>Setzt zur Laufzeit den Ausführungsstatus bzw. gibt diesen zurück (bezieht sich ohne Instanzangabe auf den Mehrinstanzknoten, sonst auf die angegebene Instanz).</td>
</tr>
<tr>
<td>Ausdruck</td>
<td>Semantik</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>[instance.]start()</td>
<td>Startet die entsprechende Instanz (setzt den Status der angegebenen Instanz auf ACTIVATED).</td>
</tr>
<tr>
<td>[instance.]abort(errorCode)</td>
<td>Bricht die Ausführung der angegebenen Instanz mit dem angegebenen Error Code ab und setzt den Status auf FAILED.</td>
</tr>
<tr>
<td>[instance.]getValue(param)</td>
<td>Gibt zur Laufzeit den dem Parameter param zugewiesenen Wert zurück.</td>
</tr>
<tr>
<td>[instance.]setValue(param, value)</td>
<td>Setzt zur Laufzeit den Wert des Parameters param auf value.</td>
</tr>
<tr>
<td>getValue(param)</td>
<td>Gibt zur Laufzeit den Wert des Parameters param zurück.</td>
</tr>
<tr>
<td>setValue(param, value)</td>
<td>Setzt zur Laufzeit den Wert des Parameters param auf value.</td>
</tr>
<tr>
<td>param.dataType</td>
<td>Gibt den Datentyp des Parameters param zurück bzw. setzt diesen.</td>
</tr>
<tr>
<td>param.isMandatory</td>
<td>Bestimmt, ob der Parameter param obligat ist.</td>
</tr>
</tbody>
</table>

Tabelle B.1: In den Listings B.1 bis B.5 verwendete Definitionen
Listing B.1: ExecMultiInstanceNode: Ausführungslogik eines Mehrinstanzknotens (ohne Suspend/Resume)

// Implementiert die Ausführungslogik des Mehrinstanzknotens

procedure ExecMultiInstanceNode();
state = RUNNING;

// Berechne nötige Anzahl von Instanzen (siehe Listing B.5)
instanceCount = CalcInstanceCount();
instanceList = new(list of Instance);
for i = 1 to instanceCount:
    instanceList . add( createInstance( assignedActivityTemplate ));
end for;

// Versorge die Instanzen mit ihren Eingabeparametern (siehe Listing B.3)
SupplyInstances( instanceList );

// Stäre die Ausführung der Instanzen
for each instance ∈ instanceList:
    instance . start ();
end for;

n_completed = 0;

// Warte, bis alle Instanzen fertig sind
while n_completed < instanceCount:
    // Wartet darauf, dass sich der Status irgendeiner Instanz ändert
    instance = waitForAnyStateChange(instanceList);
    // Prüfe ob anhand des neuen Status eine Aktion nötig ist:
    case instance . state of
        if COMPLETED:
            n_completed += 1;
        if FAILED:
            for each inst in instanceList:
                if ( inst = instance) then
                    inst . abort(instance . errorCo de);
                end if;
            end for;
            abort(instance . errorCo de);
            return;
        end case;
    end while;

// Sammle die Ausgabeparameter der Instanzen und erzeuge daraus die
// Ausgabeparameter des Mehrinstanzknotens (siehe Listing B.4)
CollectOutput( instanceList );
state = COMPLETED;
end procedure;
Listing B.2: MapParameters: Mapping zur Entwurfszeit

// Erzeugt die Ein- und Ausgabeparameter des Mehrinstanzknotens aus den
// Ein- und Ausgabeparametern der zugewiesenen Aktivität
function MapParameters(InputParams, OutputParams): (InputParamsOuter,
   OutputParamsOuter);

InputParamsOuter = ∅;
OutputParamsOuter = ∅;

for each param ∈ InputParams:
    // Erstelle Kopie des Parameters
    paramOut = clone(param);
    case param.mappingType of
        LIST:
            paramOut.dataType = list of param.dataType;
        NONE:
            paramOut.dataType = param.dataType;
        INDEX:
            // Parameter nicht nach außen weitergeben
            continue for;
    end case;

    // Füge Parameter zur Menge der Eingabeparameter hinzu
    InputParamsOuter = InputParamsOuter ∪ {paramOut};
end for;

for each param ∈ OutputParams:
    // Erstelle Kopie des Parameters
    paramOut = clone(param);
    // Keine Fallunterscheidung, da nur mappingType = LIST auftreten darf
    paramOut.dataType = list of param.dataType;
    OutputParamsOuter = OutputParamsOuter ∪ {paramOut};
end for;

return (InputParamsOuter, OutputParamsOuter);
end function:
Listing B.3: SupplyInstances: Versorgung der Instanzen mit den Eingabedaten zur Laufzeit

// Versorgt die übergebene Liste von Instanzen mit ihren Eingabeparametern
define SupplyInstances(InstanceList):
  for each param ∈ InputParams:
    // Bestimme korrespondierenden äußeren Parameter
    outerParam = getOuterParameter(param);
    for i = 0 to InstanceList.getLength() - 1:
      // Bestimme i-te Instanz
      instance = InstanceList.get(i);
      case param.mappingType of
        LIST: // Setze Wert auf das i-te Listenelement
          if (param.isMandatory) then
            instance.setValue(param, getValue(outerParam).get(i));
          else
            // Bei optionalen Parametern dürfen ggf. zu kurze Listen auftreten
            instance.setValue(param, getValue(outerParam).getOptional(i));
          end if;
        NONE: // Kopiere Wert des äußeren Parameters
          instance.setValue(param, getValue(outerParam));
        INDEX: // Setze Wert auf Index der Instanz
          instance.setValue(param, i);
        end case;
    end for;
  end for;
end define;

Listing B.4: CollectOutput: Erzeugen der Ausgabelisten nach Ende der Ausführung aller Instanzen

// "Sammelt" die Ausgabedaten der übergebenen Liste von Instanzen ein
// und erzeugt daraus die Ausgabelisten des Mehrinstanknotens
define CollectOutput(InstanceList):
  for each param ∈ OutputParams:
    // Bestimme korrespondierenden äußeren Parameter
    outerParam = getOuterParameter(param);
    // Erzeuge neue Liste zur Ausgabe
    outerList = new(list of param.dataType);
    outerList.clear();
    for i = 0 to InstanceList.getLength() - 1:
      // Bestimme i-te Instanz
      instance = InstanceList.get(i);
      outerList.add(instance.getValue(param));
    end for;
    setValue(outerParam, outerList);
  end for;
end define;
Listing B.5: CalcInstanceCount: Berechnet die Anzahl der Instanzen

// Berechnet die Anzahl der Instanzen aus der kürzesten obligat zu lesenden Liste
// bzw. aus der längsten optional zu lesenden, falls keine obligaten Parameter
// vorhanden sind.

function CalcInstanceCount : int;
    // Initialisiere Variablen
    obligateMin = ∞;
    optionalMax = −∞;
    for each param ∈ InputParams:
        // Untersuche nur listenwertige Eingabeparameter
        if (param.mappingType == LIST) then
            // Bestimme korrespondierenden äußeren Parameter
            outerParam = getOuterParameter(param);
            // Bestimme die Länge der Eingabeliste
            listLength = getValue(outerParam).getLength;
            if (param.isMandatory) then
                // Bestimme die kürzeste obligate Liste
                if (listLength < obligateMin) then
                    obligateMin = listLength;
                end if
            else
                // Bestimme die längste optionale Liste
                if (listLength > optionalMax) then
                    optionalMax = listLength;
                end if
            end if
        end if
    end for;
    if (obligateMin < ∞) then
        // Sofern es mindestens eine obligate Liste gibt, verwende
        // die Länge der kürzesten
        return obligateMin;
    else
        // Sonst verwende die Länge der längsten optionalen Liste
        return optionalMax;
    end if
end function
B.2. AND-Join mit partiellem Abbruch


Die einfachste Möglichkeit, dies zu tun, ist es, innerhalb von WriterExits statt $c_{\text{succ}}[*]$ und $c_{\text{pred}}[*]$ speziell definierte Funktionen $df_{\text{succ}}[*]$ und $df_{\text{pred}}[*]$ zu verwenden. Diese berücksichtigen wie die $c_{\ldots}$-Funktionen nur Kontrollflusskanten, allerdings ohne die Kanten zwischen dem letzten Knoten eines Slavezweigs und dem zugehörigen AND-Join mit partiellem Abbruch, da an diesen Kanten kein Datenfluss stattfindet.

Formal:

\[
\begin{align*}
    df_{\text{succ}}(n) &= \{ m \in c_{\text{succ}}(n) \mid m \text{ ist kein AND-Join mit Abbruch oder } n \text{ befindet sich in einem Masterzweig} \} \\
    df_{\text{pred}}(n) &= \{ m \in c_{\text{pred}}(n) \mid m \text{ ist kein AND-Join mit Abbruch oder } n \text{ befindet sich in einem Masterzweig} \} 
\end{align*}
\]

$df_{\text{succ}}[*]$ und $df_{\text{pred}}[*]$ werden entsprechend als transitive Hüllen davon definiert.

Durch die Verwendung dieser Funktionen wird verhindert, dass in Schritt 3 der Supplied(n)-Status aus Slavezweigen heraus an den zugehörigen AND-Joinknoten mit Abbruch propagiert wird und zusätzlich erreicht, dass Knoten aus Slavezweigen, die das gewünschte Datenelement schreiben, in Schritt 6 automatisch in die Menge Writers aufgenommen werden. Dennoch ist in Schritt 7 weiterhin zu beachten, dass auch dort der Supplied(n)-Status weiterhin nicht entlang von Kanten zwischen dem letzten Knoten eines Slavezweigs und dem zugehörigen AND-Join mit partiellem Abbruch propagiert werden darf (also diese Kanten dort nicht wie gewöhnliche Kontrollflusskanten behandelt werden dürfen).
B.3. Flexible Endknoten

Der WriterExists-Algorithmus kann nach dem folgenden Schema erweitert werden, um flexible Endknoten bei der Analyse korrekt zu berücksichtigen:

1. Bestimme zunächst für alle flexiblen Endknoten, die (bezüglich Kontrollfluss- und Sync-Kanten) Vorgänger des untersuchten Knotens sind, die toten Knoten.

2. Sollte der untersuchte Knoten selbst tot sein\footnote{Beachte, dass flexible Endknoten selbst nicht tot sind!}: Versorgtheit nicht relevant → Fertig!

3. Ansonsten führe normale Analyse durch, ignoriere dabei aber direkte Nachfolgeknoten von flexiblen Endknoten und tote Knoten (d.h. füge keine solchen Knoten zu NodeList hinzu und erhöhe auch nicht Counter(n)). Ändere außerdem die Logik an XOR-Joinknoten in den Schritten 3 und 8: Überprüfe hier nicht, ob die Zahl der versorgten einmündenden Kanten (Counter(n)) gleich der Zahl aller einmündenden Kanten ist, sondern prüfe, ob die Zahl der versorgten einmündenden Kante gleich der Zahl aller lebendigen einmündenden Kanten ist. Eine Kante ist \textit{lebendig}, wenn der Knoten, von dem sie ausgeht, weder tot noch ein flexibler Endknoten ist (diese Kante also zur Laufzeit aktiviert werden kann.

Eine Kante ist \textit{lebendig}, wenn der Knoten, von dem sie ausgeht, weder tot noch ein flexibler Endknoten ist (diese Kante also zur Laufzeit aktiviert werden kann.)
### B.4. Abhängige XOR-Verzweigungen

In diesem Abschnitt wird der in Abschnitt 7.1 informell beschriebene Algorithmus zur Behandlung abhängiger XOR-Verzweigungen nochmals formal als Pseudo-Code dargestellt. Dabei werden die nachfolgend in Tabelle B.2 aufgeführten Definitionen verwendet (teilweise aus [Rei00] übernommen).

<table>
<thead>
<tr>
<th>Ausdruck</th>
<th>Semantik</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>Menge aller Knoten im Workflowgraphen</td>
</tr>
<tr>
<td>(D)</td>
<td>Menge aller Datenelemente</td>
</tr>
<tr>
<td>(n_{\text{target}})</td>
<td>Der Knoten, der den zu untersuchenden Leseszugriff durchführt.</td>
</tr>
<tr>
<td>(d_{\text{target}})</td>
<td>Das Datenelement, das an (n_{\text{target}}) obligat gelesen wird.</td>
</tr>
<tr>
<td>(#(M))</td>
<td>Gibt die Anzahl der Elemente in der Menge (M) zurück.</td>
</tr>
<tr>
<td>(V_{\text{in}}(n))</td>
<td>Gibt die Eingangssemantik des Knoten (n) zurück (ONE_OF_ONE, ONE_OF_ALL oder ALL_OF_ALL)</td>
</tr>
<tr>
<td>(V_{\text{out}}(n))</td>
<td>Gibt die Ausgangssemantik des Knoten (n) zurück (ONE_OF_ONE, ONE_OF_ALL oder ALL_OF_ALL)</td>
</tr>
<tr>
<td>(c_{\text{succ}}(n))</td>
<td>Gibt alle direkten Nachfolger des Knoten (n) bezüglich Kontrollflusskanten zurück</td>
</tr>
<tr>
<td>(c_{\text{pred}}^*(n))</td>
<td>Gibt alle (direkten und transitiven) Vorgänger des Knoten (n) bezüglich Kontrollflusskanten zurück</td>
</tr>
<tr>
<td>(\text{ActivityWrites}(n, d, \text{mode}))</td>
<td>Gibt zurück, ob Knoten (n) Datenelement (d) im angegebenen Modus (\text{mode} \in {\text{OPTIONAL, MANDATORY, ANY}}) schreibt.</td>
</tr>
<tr>
<td>(\text{Value}(d, \text{numwrite}))</td>
<td>Wert des Datenelements (d) zur Laufzeit nach Schreibzugriff Nummer (\text{numwrite}).</td>
</tr>
<tr>
<td>(\text{isDecisionActivity}(n))</td>
<td>Gibt zurück, ob Knoten (n) eine Entscheidungsaktivität zugewiesen wurde.</td>
</tr>
<tr>
<td>(\text{isCoupledDecision}(n))</td>
<td>Gibt zurück, ob Knoten (n) eine Koppelaktivität zugewiesen wurde.</td>
</tr>
<tr>
<td>(\text{getDecisionElement}(n_{\text{decision}}))</td>
<td>Gibt das Entscheidungsdatenelement zurück, das Knoten (n) schreibt. Im Fall einer Koppelaktivität gibt diese Funktion stattdessen das gelesene Koppelsdatenelement zurück.</td>
</tr>
<tr>
<td>(\text{getDecisionIDs}(n_{\text{decision}}))</td>
<td>Gibt die Menge aller Decision IDs zurück, die an der (n_{\text{decision}}) zugewiesenen Entscheidungsaktivität als Ausgabe möglich sind.</td>
</tr>
<tr>
<td>(\text{getBranchDecisionIDs}(n_{\text{decision}}, n))</td>
<td>Gibt die Menge der Decision IDs zurück, die am XOR-Splitknoten (n_{\text{decision}}) demjenigen Zweig zugeordnet wurden, der mit Knoten (n) beginnt.</td>
</tr>
<tr>
<td>(\text{isFlexibleEndnode}(n))</td>
<td>Gibt zurück, ob Knoten (n) ein flexibler Endknoten ist.</td>
</tr>
<tr>
<td>(\text{isDead}(n))</td>
<td>Gibt zurück, ob Knoten (n) tot ist.</td>
</tr>
<tr>
<td>(\text{isSlaveEdge}(n_1, n_2))</td>
<td>Gibt true zurück, wenn sich (n_1) in einem Slavezweig einer AND-Verzweigung mit partiellem Abbruch befindet und (n_2) gleichzeitig der zugehörige Joinknoten ist.</td>
</tr>
<tr>
<td>(\text{LogicalElements}(n))</td>
<td>Gibt die am Knoten (n) bekannten logischen Datenelemente zurück. Dies ist eine Menge von Tupeln der Form ((d, \text{numwrite})). Dabei ist (d) ein Datenelement und (\text{numwrite}) der zugehörige Schreibzähler.</td>
</tr>
<tr>
<td>(\text{AllLogicalElements})</td>
<td>Die Menge aller bekannten logischen Datenelemente. Dies ist eine Menge von Tupeln der Form ((d, \text{numwrite}, \text{numids})). Dabei ist (d) ein Datenelement und (\text{numwrite}) der zugehörige Schreibzähler. (\text{numids}) gibt Zahl der Decision IDs der Entscheidungsaktivität an, deren Schreibzugriff auf (d) die Erhöhung des Schreibzählers auf (\text{numwrite}) ausgelöst hat. Falls der Schreibzugriff nicht durch eine Entscheidungsaktivität erfolgte, ist (\text{numids} = \text{UNDEFINED}).</td>
</tr>
<tr>
<td>Ausdruck</td>
<td>Semantik</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Precondition(n)          | Die Vorbedingung des Knotens \( n \). Dies ist eine Liste von Tupeln der Form \((d, num\text{write}, IDs)\). Dabei ist \( d \) ein Datenelement, \( num\text{write} \) der Wert des Schreibzählers und \( IDs \) eine Menge von Decision IDs.  
Semantische Interpretation: Knoten \( n \) wird ausgeführt, wenn gilt:  
\[
\bigwedge_{(d, nw, IDs) \in \text{Precondition}(n)} \text{Value}(d, nw) \in IDs
\]  
Für \( \text{Precondition}(n) = \emptyset \) wird der Knoten immer ausgeführt (\( \emptyset \) entspricht semantisch also true). |
| SuppliedCondition(n)     | Die Vorbedingung für die Versorgung von \( d_{\text{target}} \) am Knoten \( n \). Dies ist eine Menge von Mengen von Tupeln der Form \((d, num\text{write}, IDs)\) (also eine Menge von Vorbedingungen in der gerade beschriebenen Form)  
Semantische Interpretation: Der Zugriff auf \( d_{\text{target}} \) ist an \( n \) sicher versorgt, wenn gilt:  
\[
\bigvee_{\text{Cond} \in \text{SuppliedCondition}(n)} \bigwedge_{(d, nw, IDs) \in \text{Cond}} \text{Value}(d, nw) \in IDs
\]  
Für \( \text{SuppliedCondition}(n) = \emptyset \) ist dieser Zugriff nie sicher versorgt (\( \emptyset \) entspricht semantisch also false). |
| \( l++ \)                 | Beschreibt im Folgenden die Listenkatenation \((L++\{x\})\) hängt \( x \) hinten an die Liste \( L \) an.                                                                                                                                                                                                                     |
| initial(List)            | Gibt alle Elemente der Liste \( List \) bis auf das letzte zurück (es gilt \( \text{initial}(L++\{x\}) = L \)).                                                                                                                                                                                      |
| last(List)               | Gibt das letzte Elemente der Liste \( List \) zurück (es gilt \( \text{last}(L++\{x\}) = x \)).                                                                                                                                                                                                 |
| getMaxNumWrites(        | Gibt den maximalen Schreibzählerwert der logischen Datenelemente, die sich auf \( d \) beziehen, zurück oder 0, falls noch kein solcher existiert.                                                                                                                                                 |
| \AllLogicalElements, d) |                                                                                                                                                                                                                                                                                                                                                         |
| findNumIDsFor(           | Gibt den Wert \( \text{numids} \) des dem Datenelement \( d \) mit Schreibzähler \( \text{num\text{writes}} \) zugeordneten Eintrags in \( \AllLogicalElements \) zurück bzw. UNDEFINED, falls kein solcher existiert. Das Ergebnis gibt also die Anzahl der dem logischen Datenelement \( (d, \text{num\text{writes}}) \) zugeordneten Decision IDs zurück. |
| \AllLogicalElements, d,  |                                                                                                                                                                                                                                                                                                                                                         |
| num\text{writes})       |                                                                                                                                                                                                                                                                                                                                                         |
| findIDsFor(Precondition, | Gibt die Decision ID-Menge \( IDs \) des Eintrags in \( \text{Precondition} \) zurück, der sich auf das Datenelement \( d \) mit dem Schreibzähler \( \text{num\text{writes}} \) bezieht. Gibt UNDEFINED zurück, falls kein solcher Eintrag existiert.  |
| \text{Precondition}, d, |                                                                                                                                                                                                                                                                                                                                                         |
| num\text{writes})       |                                                                                                                                                                                                                                                                                                                                                         |

**Tabelle B.2:** In den Listings B.6 bis B.8 verwendete Definitionen
Listing B.6: Erweiterter WriterExists-Algorithmus als Pseudo code

```java
// Eingaben: n_{target}: Der Knoten, der d_{target} obligat lesen möchte.
// d_{target}: Das zu untersuchende Datenelement
// Ausgabe: true, falls der Lesezugriff sicher versorgt ist.
// false sonst
function ExtendedWriterExists(n_{target}, d_{target}): boolean
    supplied = WriterExists(n_{target}, d_{target});
    if supplied then
        return true;
    end if

    // Nach bisherigem Algorithmus nicht sicher versorgt, also 
    // führe die restlichen Phasen durch

    // Initialisierung der Markierungen (Phase (ii)):
    AllLogicalElements = ∅;
    for each n ∈ N:
        LogicalElements(n) = ∅; // Regel (LE1)
        Precondition(n) = ∅; // Regel (PC1)
        SuppliedCondition(n) = ∅; // Regel (SC1)
    end for:

    // Bestimme die Menge der Knoten für die die Vorbedingung bestimmt werden muss
    // Dies sind alle Vorgänger von n_{target} und natürlich dieser Knoten selbst
    Pred = c_{pred}(n_{target}) ∪ n_{target};

    // Breitensuche, beginnend mit Startknoten, um Vorbedingungen der Knoten
    // und die Vorbedingung für die Versorgtheit von d_{target} zu bestimmen
    NodeList = {Start};
    while NodeList ≠ ∅:
        NewNodeList = ∅;
        for each n_{current} ∈ NodeList:
            if isFlexibleEndnode(n_{current}) then
                // Nachfolger eines flexiblen Endknotens brauchen
                // nicht aktualisiert zu werden (Regeln (PC2), (SC2), (LE2))
                continue for;
            end if;

            // Zwischenspeichern der Markierungen des aktuellen Knoten
            CurLogElem = LogicalElements(n_{current});
            CurPrecond = Precondition(n_{current});
            CurSuppCond = SuppliedCondition(n_{current});

            // Erweitere CurSuppCond, falls n_{current} d_{target} schreibt (SC4)
            if ActivityWrites(n_{current}, d_{target}, MANDATORY) then
                CurSuppCond = CurSuppCond ∪ {CurPrecond};
            end if;

        end for;
        // Aktualisiere CurLogElem:
```
// Bestimme Entscheidungsdatenelement, falls n\(\_\)current eine
// bestimmende Entscheidungsaktivität zugewiesen wurde
if isDecisionActivity(n\(\_\)current) and ¬isCoupledActivity(n\(\_\)current) then
  \(d_{\text{decision}}\) = getDecisionElement(n\(\_\)current);
else
  \(d_{\text{decision}}\) = UNDEFINED;
end if;

// Prüfe, ob der Knoten irgendeines der Datenelemente schreibt,
// die zur Modellierung einer Abhängigkeitsbeziehung verwendet wurden
writesExistingDecisionElement = false;
for each \((d,\text{numwrites})\) \(\in\) CurLogElem:
  if ActivityWrites(n\(\_\)current, d, ANY) then
    // Falls irgendein Datenelement geschrieben wird, das bereits in einer
    // Vorbedingung verwendet wurde, so erhöhe den Schreibzähler (LE5)
    newnumwrites = getMaxNumWrites(AllLogicalElements, d) + 1;
    // Wenn das Entscheidungsdatenelement geschrieben wird,
    // so bestimme die Zahl der Decision IDs
    if \(d = d_{\text{decision}}\) then
      // Bestimme Anzahl der Decision IDs an n\(\_\)current
      NewIDCount = #(getDecisionIDs(n\(\_\)current));
      writesExistingDecisionElement = true;
    else
      NewIDCount = UNDEFINED;
    end if;
    // Ersetze alten Eintrag in CurLogElem
    CurLogElem = (CurLogElem \ (d,\text{numwrites})) \(\cup\) \{(d, newnumwrites)\};
    // Füge das neue logische Datenelement zur Menge
    // aller logischen Datenelemente hinzu
    AllLogicalElements = AllLogicalElements \(\cup\) \{(d, newnumwrites, NewIDCount)\};
  end if;
end for;

// Falls n\(\_\)current eine Entscheidungsaktivität zugewiesen wurde
// und auf ein in CurLogElem noch nicht vorhandenes Entscheidungsdatenelement
// schreibt, so füge es hinzu (LE4)
if not writesExistingDecisionElement then
  // Bestimme Zahl der Decision IDs
  NewIDCount = #(getDecisionIDs(n\(\_\)current));
  // Bestimme Wert des Schreibzählers (verwende getMaxNumWrites()).
  // da in einem "parallelen" Zweig bereits ein Eintrag für \(d_{\text{decision}}\)
  // existieren könnte
  numwrites = getMaxNumWrites(AllLogicalElements, d\(\_\)decision) + 1;
  // Erstelle neue Einträge
  CurLogElem = CurLogElem \(\cup\) \{(d\(\_\)decision, numwrites)\};
  AllLogElem = AllLogElem \(\cup\) \{(d\(\_\)decision, numwrites, NewIDCount)\};
end if;

// Aktualisiere Markierungen der Nachfolgeknoten
for each \(n \in\) \(\_\)suc\(\_\)e(n\(\_\)current):
  if \(n \notin\) Pred or isDead(n) or isSlaveEdge(n\(\_\)current, n) then
Ignoriere tote Knoten, die Kante vom letzten Knoten eines Slavezweigs zum zugehörigen AND−Join mit partiell abgebrochen oder Knoten, die keine Vorgänger von \( n_{\text{target}} \) sind ((SC2) und (PC2)).

```c
continue for;
end if;
```

Aktualisiere Vorbedingung Precondition des Nachfolgers:
```
case \( V_{\text{out}}(n_{\text{current}}) \) of
ONE_OF_ALL: // \( n_{\text{current}} \) ist XOR−Split
  // Erweitere Vorbedingung des Nachfolgers um die des Zweigs, in dem er sich befindet (PC5)
  IDs = getBranchDecisionIDs(\( n_{\text{current}} \), n);
  NewPrecond = CurPrecond ++ \{ (\( d_{\text{decision}} \), numwrites, IDs) \};
default: // Sonst koppere einfach Vorbedingung des aktuellen Knotens (PC6)
  NewPrecond = CurPrecond;
end case;
```
```
case \( V_{\text{in}}(n) \) of
ONE_OF_ALL: // Nachfolger ist XOR−Join
  // Entferne letztes Glied aus der Vorbedingung (PC4)
  Precondition(n) = initial(NewPrecond);
default: // Sonst kann die Vorbedingung einfach ersetzt werden.
  // Dies gilt auch für AND−Joins, da in diesem Fall alle Vorgänger eine identische Vorbedingung aufweisen (PC6)
  Precondition(n) = NewPrecond;
end case;
```

Aktualisiere SuppliedCondition des Nachfolgers:
```
case \( V_{\text{in}}(n) \) of
ONE_OF_ALL: // Nachfolger ist XOR−Join
  // Verschmelze Einträge in SuppliedCondition, die sich nur in der Bedingung eines Knotens unterscheiden
  // (als Optimierung; es würde auch ausreichen, dies einmalig vor Auswertung der Vorbedingung zu erledigen)
  // mergeSuppConds wird nachfolgend beschrieben (Listing B.7)
  SuppliedCondition(n) =
    mergeSuppConds(SuppliedCondition(n), CurSuppCond, AllLogicalElements);
default: // Sonst kann die Vorbedingung einfach hinzugefügt werden (SC3)
  SuppliedCondition(n) = SuppliedCondition(n) ∪ CurSuppCond;
end case;
```

Aktualisiere LogicalElements des Nachfolgers:
```
case \( V_{\text{in}}(n) \) of
ONE_OF_ONE: // Nachfolger ist gewöhnlicher Knoten
  // LogicalElements kann einfach kopiert werden, da \( n_{\text{current}} \) als einziger Vorgänger die einzige mögliche Quelle dafür darstellt
  LogicalElements(n) = CurLogElem;
default: // Sonst müssen gegebenen vorhandene Einträge mit niedrigerem Schreibzähler ersetzt bzw. ein neuer Eintrag hinzugefügt werden, wenn noch kein solcher existiert (LE3)
  for each \( (d_{1}, numwrites_{1}) \) ∈ CurLogElem
```
// Prüfe, ob bereits ein Eintrag für Datenelement \(d_1\) vorhanden ist
if \(\exists (d_2, \text{numwrites}_2) \in \text{LogicalElements}(n) : d_1 = d_2\) then
  if \(\text{numwrites}_1 > \text{numwrites}_2\) then
    // Es existiert bereits ein Eintrag mit niedrigerem Schreibzähler
    // → ersetzen
    \(\text{LogicalElements}(n) = (\text{LogicalElements}(n) \setminus \{(d_2, \text{numwrites}_2)\}) \cup \{(d_1, \text{numwrites}_1)\}\)
  end if
else
  // Es existiert noch kein Eintrag für \(d_1\) → Neuen hinzufügen
  \(\text{LogicalElements}(n) = \text{LogicalElements}(n) \cup \{(d_1, \text{numwrites}_1)\}\)
end if;
end case;

// Füge Knoten zur Untersuchung in nächsten Iteration hinzu
\(\text{NewNodeList} = \text{NewNodeList} \cup \{n\}\);
end for;
end for;
// Setze NodeList für nächste Iteration
\(\text{NodeList} = \text{NewNodeList}\);
end while;

// Ende von Phase (ii). überprüfe nun, ob die Vorbedingung für die Versorgtheit von \(d_{\text{target}}\) immer erfüllt ist, wenn die Vorbedingung für \(n_{\text{target}}\) gilt
\(\text{TargetPrecondition} = \text{Precondition}(n_{\text{target}})\);
\(\text{NewSuppCond} = \text{SuppliedCondition}(n_{\text{target}})\);

// Vereinfache Vorbedingung für \(d_{\text{target}}\) (Phase (iii))
// Dies ist notwendig, um die Fälle zu erkennen, in denen \(d_{\text{target}}\) für alle Decision IDs einer Verzweigung erfüllt ist, von der \(n_{\text{target}}\) aber nicht abhängig ist. In diesem Fall kann die Bedingung für das entsprechende Datenelement weggelassen werden, da \(d_{\text{target}}\) unabhängig von dieser Verzweigungsentscheidung versorgt ist. Würden diese Bedingungen nicht entfernt, so würden die entsprechenden Klauseln später fälschlicherweise als erfüllt betrachtet werden.
do
  \(\text{TargetSuppCond} = \text{NewSuppCond}\);
  //mergeSuppConds ist in Listing B.7 definiert
  \(\text{NewSuppCond} = \text{mergeSuppConds}(\text{TargetSuppCond}, \text{TargetSuppCond}, \text{AllLogicalElements})\);
while \(\text{NewSuppCond} \neq \text{TargetSuppCond}\);
end do

// Prüfe, ob \(\text{TargetSuppCond}\) für alle Wertbelegungen aus
// TargetPrecondition erfüllt ist (Phase (iv))
// EvaluateSuppCond wird nachfolgend beschrieben (Listing B.8)
return EvaluateSuppCond(\emptyset, \text{TargetPrecondition}, \text{TargetSuppCond});
end function;
Listing B.7: Erweiterter WriterExists-Algorithmus: Funktion mergeSuppConds

```plaintext
// Vereinigt in Phase ( iii ) die Vorbedingungen SuppCond₁ und SuppCond₂ und führt
// dabei Vereinfachungen durch, indem Vorbedingungen für die Versorgung
// die sich nur in einer Bedingung an einem Datenelement unterscheiden.
// verschmolzen werden
function mergeSuppConds(SuppCond₁, SuppCond₂, LogicalElements): SuppliedCondition
ResultCond = SuppCond₂;
for each Clause₁ ∈ SuppCond₁:
    NewResultCond = {};
    for each Clause₂ ∈ ResultCond:
        // Untersuche nur gleich lange Klauseln näher.
        // Denn: Falls die Klauseln unterschiedlich lang sind, so beziehen sie sich
        // auf jeden Fall auf unterschiedliche Datenelemente
        if #(Clause₁) = #(Clause₂) then
            numDiff = 0;
            ResultClause = {};
            for each (d, numwrites, IDs₁) ∈ Clause₁:
                IDs₂ = findIDsFor(Clause₂, d, numwrites):
                if IDs₂ = UNDEFINED then
                    // beide Klauseln beziehen sich auf unterschiedliche Datenelemente
                    exit for;
                else
                    if IDs₁ ≠ IDs₂ then
                        numDiff += 1;
                        if numDiff > 1 then
                            // Mehr als ein Unterschied → Keine Vereinigung möglich
                            exit for;
                        end if;
                    end if;
                end for;
            end if;
        // Vereinige IDs, falls dies der erste Unterschied ist
        NewIDs = IDS₁ ∪ IDS₂;
        // Prüfe, ob NewIDs alle für das entsprechende logische
        // Datenelement verfügbaren Decision IDs enthält.
        // Füge den Eintrag nur zur Resultatklausel hinzu, wenn dies
        // nicht der Fall ist (ansonsten ist dieser Eintrag immer erfüllt
        // und kann weggelassen werden)
        if #(NewIDs) ≠ getNumIDsFor(LogicalElements, d, numwrites) then
            ResultClause = ResultClause ∪ {(d, numwrites, NewIDs)};
        end if;
    end for;
    if numDiff = 1 then
        // Falls es genau einen Unterschied gab, ersetze Clause₁ durch
        // die verschmolzene Klausel
        Clause₁ = ResultClause;
    end if;
end for;
if numDiff > 1 then
    // Falls die Klauseln unterschiedlich sind und nicht verschmolzen wurden,
    // behalte Clause₂ im Ergebnis bei
end if;
```

NewResultCond = NewResultCond ∪ \{\text{Clause}_2\};

\textbf{end if}:

\textbf{else} // Fall \#(\text{Clause}_1) ≠ \#(\text{Clause}_2) (\text{also insbesondere Clause}_1 ≠ \text{Clause}_2)

// Falls die Klauseln unterschiedlich sind, behalte \text{Clause}_2 \text{ im Ergebnis bei}

\text{NewResultCond} = \text{NewResultCond} ∪ \{\text{Clause}_2\};

\textbf{end if}:

\textbf{end for}; // for each \text{Clause}_2 ∈ ResultCond

// Füge (ggf. verschmolzene) Klausel zum Ergebnis hinzu.

\text{ResultCond} = \text{NewResultCond} ∪ \{\text{Clause}_1\};

\textbf{end for}; // for each \text{Clause}_1 ∈ \text{SuppCond}_1

\textbf{return} \text{ResultCond};

\textbf{end function}:
Listing B.8: Erweiterter WriterExists-Algorithmus: Funktion EvaluateSuppCond

// Bestimmt in Phase (iv) rekursiv alle Wertebelegungen, die sich aus der restlichen Vorbedingung // RestPrecond ergeben. und setzt diese in TargetSuppCond ein. // values enthält eine Menge von Tupeln (d, numwrites, id), die die bereits // vorgegebenen Wertebelegungen angibt.

function EvaluateSuppCond(values, RestPrecond, TargetSuppCond): boolean;

if RestPrecond ≠ ∅ then // Noch nicht am Ende der Vorbedingung

(d, numwrites, IDs) = last(RestPrecond);

for each id ∈ IDs:

if ¬EvaluateSuppCond(values ++ (d, numwrites, id), initial(RestPrecond), TargetSuppCond) then

// Falls TargetSuppCond für eine Wertebelegung nicht erfüllbar ist, so breche ab

return false;

end if;

end for;

// Für alle Wertebelegungen erfüllt, die sich aus RestPrecond ergeben
return true;

else // In values steht eine komplette Wertebelegung (Rekursionsabbruch)

// Setze in values gegebene Belegung in TargetSuppCond ein:

for each Clause ∈ TargetSuppCond:

result = true;

for each (d, numwrites, IDs) ∈ Clause:

// Sucht den Eintrag (d, numwrites, id) in values

// und gibt id zurück oder UNDEFINED, falls kein solcher

// Eintrag existiert

id = findValueFor(values, d, numwrites)

if id = UNDEFINED then

// Falls kein Wert für das Datenelement vorhanden ist, so ist die // gesamte Klausel unerfüllt (da Und-Verknüpfung innerhalb der Klausel) // Dies ist dann der Fall, wenn die Ausführung von n_target von d abhängt

result = false;

exit for;

else

// Falls id kein gültiger Wert ist, so ist die gesamte Klausel unerfüllt // (da Und-Verknüpfung innerhalb der Klausel)

if id ∉ IDs then

result = false;

exit for;

end if;

end if;

end for;

// Falls eine einzige Klausel erfüllt ist, so ist die gesamte Vorbedingung erfüllt // (da die einzelnen Klauseln Oder-verbunden sind)

if result then

return true;

end if;

end for;

// Keine einzige Klausel wurde erfüllt → gesamte Vorbedingung unerfüllt

return false;

end if;

end function;
C. Erklärung

Matrikelnummer: 514520

Hiermit erkläre ich, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 8.7.2008

........................
Jonas Wolz