ulm university universität uulm

Universität Ulm
Fakultät für Ingenieurswissenschaften und Informatik
Institut für Datenbanken und Informationssysteme (DBIS)

Integration von Geschäftsregeln in Prozess-Management-Systeme

Diplomarbeit

vorgelegt von

Johannes Stöhr

1. Gutachter: Prof. Dr. P. Dadam
2. Gutachter: Dr. S. Rinderle-Ma

11. Juli 2007

Erklärung

Ich erkläre, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Ulm, den 11. Juli 2007

(Johannes Stöhr)

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben. Insbesondere möchte ich mich bei meinem Professor, Herrn Prof. Dr. P. Dadam und meiner Betreuerin, Frau Dipl.-Inf. Thao Ly, für die ausgezeichnete Betreuung bedanken. Weiterhin danke ich Frau Dr. S. Rinderle-Ma, dass Sie sich bereit erklärt hat, die Rolle der Zweitgutachterin zu übernehmen. Armin Butscher und Daniel Mangold danke ich für die konstruktiven Anregungen zur Ausarbeitung. Schließlich möchte ich mich bei meiner Familie für die Unterstützung bedanken, die ich während des gesamten Studiums erfahren durfte.

Inhaltsverzeichnis

Erklärung ii

Danksagung iii

1 Einleitung 2

1.1 Semantische Prozessverifikation 2

1.2 Aufgabenstellung der Diplomarbeit 3

1.3 Aufbau der Diplomarbeit 3

2 Grundlagen Prozess-Meta-Modell 4

2.1 Blockstrukturierung 4

2.2 Kontrollstrukturen 5

2.3 Dynamisches Verhalten 6

2.4 Datenflussmodellierung 7

2.5 API-Funktionen von ADEPT 9

2.6 Adaptivität 9

3 Geschäftsregeln 11

3.1 Begriffsklärung Geschäftsregeln 11

3.2 Kategorien von Geschäftsregeln 12

3.2.1 Strukturaussagen 12

3.2.2 Aktionsaussagen 13

3.2.3 Ableitungen 14

3.3 Der Business Rules-Ansatz 14

3.3.1 Ausgangslage 14

3.3.2 Grundkonzepte 15

3.4 Zusammenfassung 15

4 Ansätze zur Integration von Geschäftsregeln in Prozess-Management-Systeme 17

4.1 Prozessmodellierung mit Geschäftsregeln 17

4.2 Integration von Geschäftsregeln in PMS mit Business Rules Engines 18

4.3 Einschränkung und Automation der Prozessadaption 23

4.3.1 Einschränkung der Adaptions- und Kompositionsmöglichkeiten von Prozessen . . . 23
4.3.2 Automatische Adaption von Prozessen . . . 25
4.4 Semantische Verifikation von Prozessen mit Geschäftsregeln . . . 26
4.4.1 Logikbasierte Verifikation . . . 27
4.4.2 Modellbasierte Verifikation . . . 27
4.4.3 Graphbasierte Verifikation . . . 31
4.5 Zusammenfassung und Diskussion . . . 32

5 Integration von Integritätsregeln in ein Prozess-Management-System 34
5.1 Aufbau der zu verifizierenden Integritätsregeln . . . 36
5.1.1 Struktur der Integritätsregeln . . . 37
5.1.2 Form des Bedingungs- und des Folgeteils . . . 38
5.1.3 Reihenfolgebeziehungen . . . 38

6 Verifikation 40
6.1 Lösungsansatz . . . 41
6.2 Herleitung der strukturellen Kriterien . . . 41
6.2.1 Strukturelle Kriterien für Bedingungsteile . . . 42
6.2.2 Strukturelle Kriterien für Folgeteile . . . 45
6.3 Verifikation der strukturellen Kriterien . . . 54
6.3.1 Die Ausführung der Quellaktivität S schließt die Ausführung der Zielaktivität T aus $(S \not< \bullet \not> T)$ . . . 54
6.3.2 Die Ausführung der Quellaktivität S impliziert die Ausführung der Zielaktivität T $(S \bullet <> T)$ . . . 54
6.3.3 Die Ausführung der Quellaktivität S impliziert die Versorgung mit den Zielaktivitäten T_i $(S \bullet <> [T_1/T_2/\dots])$ . . . 59
6.4 Verifikation von Reihenfolgebeziehungen . . . 65
6.5 Zusammenfassung und Diskussion . . . 69

7 Effizienzaspekte 71
7.1 Eingrenzung der zu verifizierenden Integritätsregeln . . . 71
7.2 Optimierungen der Algorithmen für strukturelle Kriterien . . . 77
7.3 Effizienzanalyse von Integritätsregeln . . . 79

8 Zusammenfassung und Ausblick 80

Anhang 89

A Algorithmen 89

Kapitel 1

Einleitung

Aufgrund des hohen Wettbewerbsdrucks und rasch ändernder Umweltbedingungen müssen Geschäftsprozesse oft angepasst und optimiert werden. Deshalb wird versucht, Geschäftsprozesse explizit zu machen und sie computergestützt durchzuführen. Dazu werden Prozess-Management-Systeme (PMS) verwendet. Mit ihnen können Prozesse meistens graphisch in Form von Prozessschemas modelliert werden. Diese Prozessschemas können dann als Prozessinstanzen vom Prozess-Management-System ausgeführt werden. Um eine Prozessinstanz auszuführen, muss das Prozess-Management-System nach der Beendigung einer Aktivität zur nächsten weiterschalten, eventuell Anwendungsprogramme starten und die möglichen Bearbeiter von Aktivitäten bestimmen. Neben der Adaptivität von Prozessen ist auch ihre Korrektheit von entscheidender Bedeutung, damit man zur Laufzeit keine bösen Überraschungen erhält.

1.1 Semantische Prozessverifikation

Bei Prozessen wird zwischen syntaktischer und semantischer Korrektheit unterschieden. Ein Prozessmodell ist syntaktisch korrekt, wenn es gemäß der Syntax des Prozessmetamodells modelliert wurde, so dass z.B. keine Deadlocks auftreten können und eine Transition nur von einem einzigen Knoten abgeht. Je nach Prozess-Meta-Modell, wie z.B. Petri-Netzen oder ADEPT WSM-Netzen [Rei00], gibt es bereits verschiedene Verfahren, um Prozesse auf syntaktische Korrektheit zu untersuchen.

Wenn ein Prozess syntaktisch korrekt modelliert wurde, kann es allerdings trotzdem der Fall sein, dass dieses keinen Sinn ergibt. So können z.B. beliebige Aktivitäten in einem Prozess in beliebiger Reihenfolge kombiniert werden, auch wenn sich strikt an das Prozess-Meta-Modell gehalten wird. Prozesse stellen jedoch meistens ein Modell für einen Ausschnitt der Realität dar, für den gewisse Gesetze bzw. Regeln gelten. Folglich müssen diese ebenfalls für den Prozess gelten. Eine Regel kann z.B. besagen, dass einem Patienten nicht Aspirin und Marcumar zusammen verabreicht werden darf, da die beiden Medikamente miteinander unverträglich sind. In einem Prozess, der die Behandlung eines Patienten modelliert, sollten deshalb nie die beiden Aktivitäten *Verabreiche Aspirin* und *Verabreiche Marcumar* in einer Ausführung vorkommen können. In einem syntaktisch korrekten Pro-

zess können jedoch diese beiden Aktivitäten in einer Ausführung vorkommen. Die Regel, dass einem Patienten nicht Aspirin und Marcumar zur gleichen Zeit verabreicht werden darf, stellt dabei eine Information auf semantischer Ebene dar, zu der der Prozess konsistent sein muss. Syntaktisch lässt sich die semantische Information nicht ausücken. Deshalb muss die semantische Information auf andere Weise im Prozess hinterlegt werden, um dem Prozess ein tieferes Verständnis für den zu modellierenden Sachverhalt zu geben.

Manuell lässt sich oft nur schwer prüfen, ob ein Prozess konsistent zu der für ihn geltenden semantischen Information ist. Bei komplexen Prozessen oder bei Änderungen zur Laufzeit kann leicht der Überblick verloren werden [LRD06b]. Deshalb ist es von Vorteil, wenn das Prozess-Management-System die semantische Verifikation von Prozessen Prozesses übernimmt.

1.2 Aufgabenstellung der Diplomarbeit

Nachdem der Begriff „Semantische Prozessverifikation“ geklärt wurde, wird hier die Aufgabenstellung der vorliegenden Arbeit erläutert.

Dabei ist die Aufgabenstellung der Diplomarbeit zweigeteilt. Der erste Teil besteht aus Literaturarbeit. Es soll dabei der Begriff Geschäftsregeln geklärt werden und aktuelle Ansätze zur Integration von Geschäftsregeln in Prozess-Management-Systeme untersucht werden.

Im zweiten Aufgabenteil werden eigene konzeptuelle Überlegungen zur Integration von Geschäftsregeln angestellt. Ziel ist die semantische Prozessverifikation mit Geschäftsregeln. Dabei soll untersucht werden, inwieweit sich mittels graphbasierten Tests diese Geschäftsregeln effizient verifizieren lassen.

1.3 Aufbau der Diplomarbeit

Die vorliegende Arbeit gliedert sich grob in zwei Teile. Der erste Teil von Kapitel 2 bis 4 schafft die Grundlagen für die konzeptuelle Arbeit im zweiten Teil, der von Kapitel 5 bis 7 geht.

In Kapitel 2 wird in das Prozess-Meta-Modell eingeführt, auf dem die graphbasierte semantische Prozessverifikation des zweiten Teils beruht. Kap.3 stellt eine Einführung in Geschäftsregeln dar. Zu deren Integration in Prozess-Management-Systeme werden in Kapitel 4 aktuelle Ansätze vorgestellt.

Der zweite Teil der Arbeit beginnt mit Kapitel 5, in dem die zu verifizierenden Geschäftsregeln spezifiziert werden. In Kapitel 6 erfolgt schließlich die graphbasierte semantische Prozessverifikation mit Geschäftsregeln. Kapitel 7 behandelt einige Optimierungsmöglichkeiten, z.B. mögliche Erweiterungen von Prozess-Meta-Modellen.

Zum Schluss werden in Kapitel 8 die wichtigsten Erkenntnisse dieser Arbeit zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

Kapitel 2

Grundlagen Prozess-Meta-Modell

Die Konzepte und Prozessbeispiele, die im Rahmen dieser Arbeit vorgestellt werden, basieren auf einem bestimmten Prozess-Meta-Modell, den sog. ADEPT WSM-Netzen [Rei00]. Die für diese Arbeit relevanten Grundlagen von ADEPT WSM-Netzen werden im Folgenden vorgestellt.

2.1 Blockstrukturierung

Die ADEPT WSM-Netze gehören zu den sog. blockstrukturierten Prozess-Meta-Modellen. Das bedeutet, dass alle Kontrollstrukturen, wie z.B. Sequenz, parallele Verzweigung und bedingte Verzweigung einen eigenen Block bilden, der eine eindeutige Start- und Endaktivität besitzt. Diese Blöcke können beliebig ineinander verschachtelt sein. Allerdings dürfen sie sich nicht überschneiden [Rei00]. Abb. 2.1 zeigt ein solches blockstrukturiertes ADEPT WSM-Netz, wobei die einzelnen Blöcke farblich unterlegt sind. So ist z.B. Block *2* eine bedingte Verzweigung, d.h. es kann nur eine der Knoten A oder B ausgeführt werden. Dabei stellt S_2 bzw. J_2 die sog. Start- bzw. Endaktivität dar. Block *4* ist wiederum eine parallele Verzweigung. Folglich werden, wenn der Block ausgeführt wird, immer alle Zweige ausgeführt, d.h. sowohl Knoten E als auch Knoten F.

Blockstrukturierte Prozess-Meta-Modelle haben mehrere Vorteile gegenüber unstrukturierten Prozess-Meta-Modellen, wie z.B. Petri-Netzen. Prozesse eines blockstrukturierten Prozess-Meta-Modells lassen sich im Vergleich zu Prozessen eines unstrukturierten Prozess-Meta-Modells leichter erstellen, analysieren und auch ändern. Die Prozesse lassen sich mit einem blockstrukturierten Prozess-Meta-Modell wegen der Übersichtlichkeit und weniger möglichen Fehlerquellen leichter erstellen. Syntaktische Prüfungen und Prüfungen, die das dynamische Verhalten betreffen, können effizienter durchgeführt werden. Durch die Blockstrukturierung besitzen die Prozesse von blockstrukturierten Prozess-Meta-Modellen auch bereits gewisse Dynamikeigenschaften, die in Prozessen von unstrukturierten Prozess-Meta-Modellen nicht automatisch gegeben sind. Da bei reiner Blockstrukturierung die Ausdrucksmächtigkeit der Prozess-Meta-Modelle geringer ist, besitzt ADEPT einige Erweiterungen, wie z.B. Synchronisationskanten, die weiter unten vorgestellt werden.

Abbildung 2.1: Blockstrukturierung und Datenflussmodellierung in einem ADEPT-Prozess

2.2 Kontrollstrukturen

Nachdem die allgemeinen Eigenschaften von blockstrukturierten Prozess-Meta-Modellen erläutert wurden, sollen jetzt die für die vorliegende Arbeit wichtigen Kontrollstrukturen von ADEPT WSM-Netzen anhand der Abb. 2.1 und 2.2 vorgestellt werden. Dabei gibt es die folgenden vier grundlegenden Kontrollstrukturen:

Sequenz Der Block *3* bildet eine Sequenz. Das bedeutet, dass, wenn diese Sequenz ausgeführt wird, zuerst die Aktivität C und danach die Aktivität D ausgeführt wird.

Verzweigungen Bei den Verzweigungen werden wiederum drei Typen unterschieden:

- Parallele Verzweigung:
  Die Knoten von Block *4* in Abbildung 2.2 bilden eine solche parallele Verzweigung. Alle Aktivitäten in dem sog. AND-Block werden dabei ausgeführt. Dabei ist die Reihenfolge zwischen den Aktivitäten auf verschiedenen Zweigen nicht festgelegt. Es kann also sein, dass einmal die Aktivität E vor der Aktivität F und dass andere Mal F vor E ausgeführt wird. Die Startaktivität S_3 des Blocks

heißt AND-Split und die entsprechende Endaktivität J_3 ein AND-Join.

- Bedingte Verzweigung: Bei einer bedingten Verzweigung wie Block *1* in Abbildung 2.2 wird nur immer ein Zweig des sog. XOR-Blocks ausgeführt. Die Startaktivität S_1 bzw. die Endaktivität J_1 heißen wiederum XOR-Split bzw. XOR-Join.

Synchronisationskante (Sync-Kante) Mit einer Synchronisationskante kann die Ausführungsreihenfolge von Aktivitäten auf verschiedenen Zweigen eines AND-Blocks festgelegt werden. Abbildung 2.2 zeigt eine Synchronisationskante zwischen den Ak-

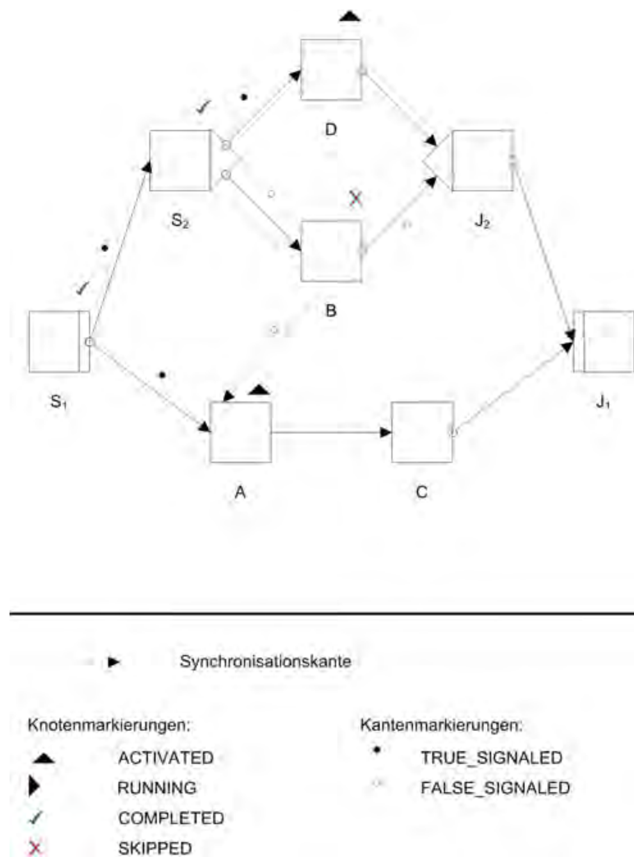


Abbildung 2.2: Prozessinstanz mit einer Synchronisationskante zwischen den Aktivitäten B und C

tivitäten B und A. Diese Synchronisationskante *(B, A)* bedeutet, dass die Aktivität A frühestens ausgeführt werden kann, nachdem die Aktivität B zuvor erfolgreich beendet wurde oder nachdem feststeht, dass B nicht mehr ausgeführt werden kann, weil ein anderer XOR-Zweig ausgewählt wurde als der, auf dem B liegt.

2.3 Dynamisches Verhalten

Nachdem die statischen Elemente des Kontrollflusses eingeführt wurden, wird im Folgenden vorgestellt, wie die Ausführungszustände von Prozessinstanzen gekennzeichnet werden

können. Dazu gibt es Kanten- und Knotenmarkierungen.

Bei den Knotenmarkierungen werden u.a. die Markierungen ACTIVATED, RUNNING, COMPLETED und SKIPPED unterschieden (s. Abbildung 2.2). Ein Knoten wird als ACTIVATED gekennzeichnet, wenn er aktiviert ist und damit ausgeführt werden kann. Sobald er ausgeführt wird, wechselt seine Knotenmarkierung zu RUNNING und nach dessen erfolgreicher Ausführung schließlich zu COMPLETED. Ein Knoten, der als SKIPPED markiert ist, kann schließlich nicht mehr zur Ausführung kommen, weil ein anderer XOR-Zweig, als der auf dem er liegt, zur Ausführung kommt bzw. gekommen ist.

Als Kantenmarkierungen gibt es TRUE_SIGNALED und FALSE_SIGNALED. Mit TRUE_SIGNALED wird eine Kante markiert, die bei der Prozessausführung passiert wird und mit FALSE_SIGNALED wird eine Kante markiert, die abgewählt wurde und nicht passiert wird.

In der Abbildung 2.2 wurden die Aktivitäten S_1 und S_2 bereits erfolgreich ausgeführt. Deswegen sind diese Aktivitäten als COMPLETED markiert und die Kanten zwischen S_1 und S_2, zwischen S_2 und D und zwischen S_1 und A als TRUE_SIGNALED markiert. Knoten D ist deswegen momentan aktiviert und ist damit als ACTIVATED gekennzeichnet. Da D auf einem Zweig eines XOR-Blocks liegt, ist der andere Zweig als abgewählt markiert. Folglich ist die Aktivität B als SKIPPED markiert und die Kanten zwischen S_2 und B und zwischen B und J_2 als FALSE_SIGNALED markiert. Da somit feststeht, dass die Aktivität B nicht mehr zur Ausführung kommt, ist die Synchronisationskante als TRUE_SIGNALED und die Endaktivität der Synchronisationskante als ACTIVATED dargestellt.

Um auszudrücken, welche Aktivitäten eines Prozessschemas in einer bestimmten Instanz ausgeführt werden bzw. ausgeführt wurden, gibt es die sog. Ausführungsspur (*execution trace*). Die Ausführungsspur stellt den sequentiellen Verlauf einer Prozessinstanz dar. Bei der Prozessinstanz in Abbildung 2.2 ist die Ausführungsspur z.B. $<S_1, S_2, D, J_2, J_1>$. Das bedeutet, dass die Aktivitäten S_1, S_2, D, J_2 und J_1 zur Ausführung gekommen sind bzw. zur Ausführung kommen werden.

2.4 Datenflussmodellierung

Nachdem damit die Kontrollflusskonstrukte in ADEPT behandelt wurden, soll noch kurz die Datenflussmodellierung in ADEPT behandelt werden. In ADEPT wird der Datenfluss zwischen zwei Aktivitäten modelliert, indem die Eingabe- bzw. Ausgabeparameter der beiden Aktivitäten über ein globales Datenelement miteinander verbunden werden (s. Abbildung 2.1). Bei diesem Prozess wird jeweils der Ausgabeparameter der Knoten A, B und C über das Datenelement d mit dem Eingabeparameter des Knoten E verbunden. Dabei gibt es mehrere Arten von Eingabeparametern, wobei uns nur die sog. obligaten Eingabeparameter interessieren. Damit ein Knoten ein Datenelement obligat lesen darf, muss in jeder Ausführungsspur einer seiner Vorgängerknoten dieses Datenelement schreiben.

Mit dem sog. WriterExists-Algorithmus [Rei00] kann geprüft werden, ob ein bestimm-

ter Knoten K ein Datenelement obligat lesen darf. Das bedeutet, dass der WriterExists-Algorithmus prüft, ob in jeder Ausführungsspur, in der K vorkommt, ein Vorgängerknoten von K das Datenelement schreibt. Im Folgenden wird der WriterExists-Algorithmus mit einem Beispiel vorgestellt, wobei Synchronisationskanten nicht betrachtet werden.

Die Idee bei dem Algorithmus ist, dass alle Vorgängerknoten von K ausgehend von den Knoten, die das Datenelement schreiben, markiert werden, die das Datenelement obligat lesen dürfen. Der Algorithmus terminiert schließlich, wenn der Knoten K markiert wurde oder wenn kein weiterer Knoten mehr markiert werden kann. Nur im ersten Fall ist sichergestellt, dass in jeder Ausführungsspur, in der K vorkommt, ebenfalls ein Vorgängerknoten von K vorkommt, der das Datenelement schreibt, so dass K das Datenelement lesen darf.

Bei der Markierung wird zwischen den XOR-Join-Knoten und den anderen Knotentypen unterschieden. Ein Nicht-XOR-Join-Knoten wird markiert, wenn einer seiner direkten Vorgängerknoten entweder das Datenelement schreibt oder markiert ist. In diesen beiden Fällen darf ein Nicht-XOR-Join-Knoten markiert werden, da in jeder Ausführungsspur, in der er vorkommt, ebenfalls alle seine direkten Vorgängerknoten vorkommen. Bei einem XOR-Join-Knoten wird allerdings in jeder Ausführungsspur nur ein direkter Vorgängerknoten ausgeführt. Das bedeutet, dass ein XOR-Join-Knoten nur markiert werden kann, wenn alle direkten Vorgängerknoten von ihm entweder markiert sind oder das Datenelement schreiben. Um feststellen zu können, ob alle direkten Vorgängerknoten eines XOR-Join-Knoten markiert sind bzw. das Datenelement schreiben, erhält jeder Knoten einen Zähler mit dem Anfangswert *0*. Jedes Mal, wenn ein direkter Vorgängerknoten eines nicht bereits markierten Knoten markiert wurde, wird der Zähler des noch nicht markierten Knotens erhöht. Wenn der Wert des Zählers eines XOR-Join-Knoten gleich der Anzahl der direkten Vorgängerknoten bzw. gleich der Anzahl der in den XOR-Join-Knoten einmündenden Kontrollkanten ist, wird der XOR-Join-Knoten markiert. Wenn ein neuer Knoten markiert werden konnte, wird in der nächsten Iteration untersucht, ob seine direkten Nachfolgerknoten markiert werden können.

Am Prozess in Abbildung 2.1 soll mittels WriterExists-Algorithmus geprüft werden, ob der Knoten E das Datenelement d obligat lesen darf, wenn die Knoten A, B und C dieses Datenelement schreiben. Dabei werden in der ersten Iteration die direkten Nachfolgerknoten von A, B und C betrachtet. Zuerst wird der Nachfolgerknoten J_1 von A betrachtet. Da J_1 ein XOR-Join-Knoten ist und nicht alle seine direkten Vorgängerknoten das Datenelement schreiben bzw. markiert sind, kann J_1 nicht markiert werden. Der Nachfolgerknoten von B ist wiederum ebenfalls J_1. Jetzt sind allerdings alle direkten Vorgängerknoten von J_1 markiert und somit kann J_1 ebenfalls markiert werden und in der folgenden Iteration dessen Nachfolgerknoten J_2 betrachtet werden. Allerdings muss zuerst in dieser Iteration noch der Nachfolgerknoten D von C untersucht werden. D kann ebenfalls markiert werden, da er ein Nicht-XOR-Join-Knoten ist und sein direkter Vorgängerknoten d schreibt. Der direkte Nachfolgerknoten von D ist J_2 und wurde bereits für die nächste Iteration vorgemerkt, wodurch in der nächsten Iteration nur der Knoten J_2 betrachtet wird. Damit endet die erste Iteration. In der zweiten Iteration wird wie erläutert geprüft, ob J_2

markiert werden kann. J_2 ist zwar ein XOR-Join-Knoten, allerdings sind alle seine direkten Vorgängerknoten D und J_1 markiert, weswegen er ebenfalls markiert werden kann. In der nächsten Iteration wird sein Nachfolgerknoten S_3 betrachtet. Da S_3 kein XOR-Join-Knoten ist und sein direkter Vorgängerknoten markiert ist, wird S_3 ebenfalls markiert und darauffolgend seine beiden Nachfolgerknoten E und F untersucht. Da diese ebenfalls keine XOR-Join-Knoten sind und ihr direkter Vorgängerknoten jeweils markiert ist, können auch sie markiert werden. Damit endet der WriterExists-Algorithmus, da der Knoten, der d obligat lesen möchte, markiert werden konnte und somit d obligat lesen darf.

2.5 API-Funktionen von ADEPT

Bei der semantischen Verifikation im zweiten Teil der Diplomarbeit werden die in der Tabelle 2.1 angegebenen ADEPT-Funktionen verwendet, die in [Rei00] eingeführt werden.

2.6 Adaptivität

Adaptivität ist eine zentrale Eigenschaft von PMS. Darunter versteht man, dass in einem PMS die Prozesse geändert werden können. Adaptivität ist von zentraler Bedeutung, da Unternehmen oft ihre Prozesse aufgrund von sich stetig ändernden Geschäftsbedingungen anpassen müssen. Deshalb sollten die PMS erlauben, die modellierten Prozesse zu ändern. ADEPT unterstützt dies im Gegensatz zu vielen herkömmlichen PMS Prozessänderungen sowohl auf der Ebene des Prozesstyps als auch auf der Ebene der Prozessinstanz. Daneben wird in ADEPT noch die sog. Schema-Evolution unterstützt. Bei der Schema-Evolution werden Änderungen des Prozesstyps auf bereits laufende und eventuell individuell geänderte Prozessinstanzen propagiert. Nach einer Änderung prüft ADEPT auch unmittelbar die syntaktische Korrektheit des geänderten Prozesstyps bzw. der geänderten Prozessinstanz, z.B. dass keine Verklemmung vorliegt. In Zukunft soll nach einer Prozessänderung ebenfalls die semantische Korrektheit mit überprüft werden können [LRD06b], die im zweiten Teil der Diplomarbeit behandelt wird.

Funktion	**Bedeutung**
c_succ(N)	Liefert die Menge aller direkten Nachfolgerknoten von Knoten N bzgl. normaler Kontrollkanten.
c_pred(N)	Liefert die Menge aller direkten Vorgängerknoten von Knoten N bzgl. normaler Kontrollkanten.
c_succ(N)*	Liefert die Menge aller direkten und indirekten Vorgängerknoten von Knoten N bzgl. normaler Kontrollkanten.
c_pred(N)*	Liefert die Menge aller direkten und indirekten Vorgängerknoten von Knoten N bzgl. normaler Kontrollkanten.
succ(N)	Liefert die Menge aller direkten Nachfolgerknoten von Knoten N bzgl. normaler Kontrollkanten und Sync-Kanten.
pred(N)	Liefert die Menge aller direkten Vorgängerknoten von Knoten N bzgl. normaler Kontrollkanten und Sync-Kanten.
succ(N)*	Liefert die Menge aller direkten und indirekten Nachfolgerknoten von Knoten N bzgl. normaler Kontrollkanten und Sync-Kanten.
pred(N)*	Liefert die Menge aller direkten und indirekten Vorgängerknoten von Knoten N bzgl. normaler Kontrollkanten und Sync-Kanten.
join(S)	Liefert den zur Split-Aktivität S gehörenden Join-Knoten.
split(J)	Liefert den zur Join-Aktivität J gehörenden Split-Knoten.
MinBlock(N)*	Liefert den minimalen Kontrollblock, der alle Knoten der Menge N* umschließt.

Tabelle 2.1: ADEPT-Funktionen

Kapitel 3

Geschäftsregeln

Nachdem im vorangegangenen Kapitel eine Einführung in ADEPT gegeben wurde, sollen in diesem Kapitel die Grundlagen im Bereich der Geschäftsregeln gelegt werden. Das Thema Geschäftsregeln gelangt zur Zeit immer mehr an Bedeutung. Man kann sich auch ungefähr vorstellen, was sich dahinter verbirgt, trotzdem soll hier zuerst eine genaue Definition gegeben werden, um eventuelle Unklarheiten zu beseitigen. Darauffolgend werden die möglichen Kategorien von Geschäftsregeln vorgestellt, um einen besseren Überblick über die Ausdrucksmächtigkeit von Geschäftsregeln zu erhalten. Darauffolgend wird der sog. Business Rules-Ansatz kurz vorgestellt, bei dem es darum geht, in welcher Form Geschäftsregeln am besten gehalten werden.

3.1 Begriffsklärung Geschäftsregeln

Geschäftsregeln (*Business Rules*) dienen dazu, einer Organisation zu verhelfen, ihre Ziele zu erreichen, indem sie ihr Verhalten steuern [Wik07a]. Während eine Strategie auf eher abstrakte Weise das Geschäftsverhalten vorgibt, geben Geschäftsregeln es im Detail an. Sie setzen damit die Strategie konkret um. Allerdings kommt es oft vor, dass die Geschäftsregeln nur implizit „in den Köpfen der Mitarbeiter" vorliegen und diese sich ihrer gar nicht wirklich bewußt sind. Dies hat zur Folge, dass Geschäftsregeln nur schwer geändert werden können und auch verloren gehen können, wenn die entsprechenden Leute die Organisation verlassen. Damit befasst sich der Business Rules Ansatz, der später noch vorgestellt wird (s. Kap. 3.3). Hier folgen nun einige Beispiel-Geschäftsregeln. Sie setzen die Strategie „Mehr für die Kundenzufriedenheit tun" um. Wie die weiteren in diesem Kapitel vorkommenden Geschäftsregeln könnten sie in einer Autovermietung eingesetzt werden.

- Ein guter Kunde ist ein Kunde, der in den letzten 12 Monaten Autos im Wert von 1000 Euro ausgeliehen hat.
- Einem guten Kunden muss bei jedem Geschäft ein Rabatt von 5 % gegeben werden.
- Wenn mehr als die Hälfte aller Autos verliehen wurden, müssen von anderen Filialen Autos angefordert werden. Damit immer genug Autos vorhanden sind und jedem

Kunden ein Auto ausgeliehen werden kann.

Mit der folgenden Definition wird versucht, noch mal auf den Punkt zu bringen, was der Begriff Geschäftsregel ausdrückt.

Definition: Eine Geschäftsregel ist eine Richt- oder Leitlinie, um das Geschäftsverhalten einer Organisation zu steuern bzw. zu beeinflussen.
[Wik07a]

3.2 Kategorien von Geschäftsregeln

Es gibt unterschiedliche Kategorisierungsschema für Geschäftsregeln. Die Business Rules Group unterscheidet in [Bus00] drei Kategorien von Geschäftsregeln:

- Strukturaussagen
- Aktionsaussagen
- Ableitungen

Diese drei Kategorien werden im Folgenden näher vorgestellt.

3.2.1 Strukturaussagen

Die Strukturaussagen bilden das Unternehmensvokabular, das in den anderen beiden Kategorien von Geschäftsregeln verwendet wird. Es gibt zwei Kategorien von Strukturaussagen:

- Fachbegriffe
- Fakten

Ein Fachbegriff ist ein Wort mit einer bestimmten Bedeutung für das Geschäft, z.B. Kunde oder Reservierung. Ein Fakt ermöglicht elementare Aussagen, die zwei oder mehr Fachbegriffe in Beziehung setzen, z.B. „Jeder Kunde reserviert ein Auto.“. Jeder Fachbegriff hat in einem Fakt eine ganz bestimmte Rolle inne, die sog. Objektrolle. Jedoch kann ein Fachbegriff in unterschiedlichen Fakten in unterschiedlichen Rollen verwendet werden. Deshalb besteht ein Fakt aus Objektrollen und nicht aus Fachbegriffen, um die Rolle des Fachbegriffs darzustellen.

Kategorisierungsmöglichkeiten von Fachbegriffen

Ein Fachbegriff kann zum einen danach kategorisiert werden, ob er geschäftsspezifisch oder allgemein ist. Ein geschäftsspezifischer Fachbegriff hat in einem bestimmten Kontext eine bestimmte Bedeutung für das Unternehmen. Im Kontext einer Autovermietung ist z.B. die Reservierung oder Buchung ein solcher geschäftsspezifischer Fachbegriff. Ein allgemeiner Fachbegriff ist ein alltäglich verwendeter Begriff, dessen Bedeutung klar ist, z.B. Kunde

oder Auto. Im Gegensatz zu einem allgemeinen Fachbegriff muss ein geschäftsspezifischer Fachbegriff immer explizit mittels einem oder mehreren Fakten definiert werden, z.B. kann der geschäftsspezifische Fachbegriff Reservierung durch den Fakt "Ein Kunde reserviert ein Auto über einen gewissen Zeitraum." definiert werden. Zum anderen lassen sich Fachbegriffe danach kategorisieren, ob sie ein Literal oder ein Typ sind. Ein Literal ist ein bestimmter Wert oder eine Instanz, während ein Typ eine Abstraktion einer Menge von Instanzen oder Werten darstellt.

Kategorisierungsmöglichkeiten von Fakten

Es gibt zwei Möglichkeiten, Fakten zu kategorisieren. Zum einen lassen sich Fakten danach kategorisieren, ob sie Basisfakten oder abgeleitete Fakten sind. Ein Basisfakt ist eine grundlegende Aussage, die einfach angenommen wird, z.B. „ein Fahrzeug gehört zu einer bestimmten Fahrzeugklasse". Ein abgeleiteter Fakt ist eine Aussage, die aus anderen Fakten abgeleitet werden kann. Das Ableiten kann mittels einer mathematischen Berechnung oder mit einer Folgerung geschehen, z.B. kann der abgeleitete Fakt Miethöhe durch die mathematische Berechnung *Miethöhe * Anzahl Tage* abgeleitet werden.

Zum anderen können Fakten danach kategorisiert werden, ob sie Attribute, Teilnahmen oder Generalisierungen sind. Ein Attribut ist ein Fakt mit zwei Fachbegriffen, wobei ein Fachbegriff einen Aspekt des anderen Fachbegriffs beschreibt, z.B. „Farbe ist ein Attribut von Auto.". Eine Generalisierung ist ein Fakt, wobei einer der Fachbegriffe einen Supertyp des anderen Fachbegriffs bzw. der anderen Fachbegriffe darstellt, z.B. „Ein Filialleiter ist ein Angestellter.". Eine Teilnahme ist ein Fakt, der mehrere Fachbegriffe in irgendeine Beziehung - abgesehen von der Attribut- und Generalisierungsbeziehung - setzt, z.B. „Fahrzeuge können von Kunden gemietet werden.". Teilnahmen werden typischerweise in Entity-Relationship-Modellen als Beziehungen modelliert.

3.2.2 Aktionsaussagen

Der Begriff Aktionsaussage wird hier synonym zu dem Begriff Integritätsregel verwendet. Eine Integritätsregel ist eine Aussage, die immer erfüllt sein muss, z.B. „Ein Kunde darf gleichzeitig nicht mehr als fünf Autos ausgeliehen haben." Integritätsregeln schränken also die Ergebnisse von Aktionen ein, indem sie Bedingungen vorschreiben, die immer erfüllt sein müssen und die nicht durch Aktionen verletzt werden dürfen.

Aktionsaussagen können danach kategorisiert werden, ob sie die Einschränkungen zwingend vorschreiben oder nur empfehlen. Im ersten Fall muss die Aktionsaussage befolgt werden, z.B. „Ein Kunde darf gleichzeitig nicht mehr als fünf Autos ausgeliehen haben" und im zweiten Fall kann die Aktionsaussage auch ignoriert werden, z.B. „Ein Kunde soll gleichzeitig nicht mehr als fünf Autos ausgeliehen haben".

Wie in Kap. 1 angesprochen, behandelt der zweite Teil der Diplomarbeit einen eigenen Ansatz zur semantischen Verifikation von Prozessen mit Geschäftsregeln. Allgemein wird bei der semantischen Verifikation mit Geschäftsregeln geprüft, ob ein Prozess konsistent zu

der semantischen Information ist, die für diesen Prozess gilt und die in Form von Geschäftsregeln vorliegt. Speziell bei dem Ansatz hier werden zur Spezifikation der semantischen Information nur Integritätsregeln verwendet. Dabei werden wir uns auf Integritätsregeln beschränken, mit denen komplexe Aussagen über die Ausführung von Aktivitäten gemacht werden können. Eine Integritätsregel, die mit diesem Ansatz verifiziert werden soll, kann z.B. so aussehen: „Ein Kunde, der mit der Zahlung für ein bereits ausgeliehenes Auto im Verzug ist, kann ein neues Auto nur ausleihen, wenn er entweder die bisher angefallene Miete begleicht oder seine Zahlungsfähigkeit als ausreichend eingestuft wird.“.

3.2.3 Ableitungen

Eine Ableitung leitet aus einem Basisfakt einen abgeleiteten Fakt ab. Die zwei Kategorien von Ableitungen sind mathematische Berechnungen und Folgerungen. Ein Beispiel einer mathematischen Berechnung ist „Der abgeleitete Fakt Miethöhe wird abgeleitet durch die mathematische Berechnung Mietrate * Anzahl Tage.“ Eine Folgerung kann wiederum eine logische Induktion, bei der vom Besonderen zum Allgemeinen gefolgert wird, oder eine logische Deduktion, bei der vom Allgemeinen zum Besonderen gefolgert wird, sein. Eine Folgerung prüft eine Bedingung und im positiven Fall wird ein neuer Fakt abgeleitet. Ein Beispiel einer Folgerung ist: „Wenn ein Kunde häufig Autos mietet, dann bekommt er 5 Prozent Rabatt.“

3.3 Der Business Rules-Ansatz

In jeder Organisation existieren Geschäftsregeln. Jedoch sind sich die Mitarbeiter dessen oft gar nicht bewusst, weil sie oft nicht aufgeschrieben oder kommuniziert werden, sondern nur in den Köpfen der Mitarbeiter existieren und nur nach ihnen gehandelt wird. Beim Business Rules-Ansatz [GS06, vH02] geht es darum, diese oft nur informal vorliegenden Geschäftsregeln zu identifizieren und zu externalisieren, indem sie z.B. unabhängig von Software-Systemen gehalten werden. Dadurch können besser Konflikte zwischen Geschäftsregeln erkannt werden und auch besser der Zweck und Nutzen von Geschäftsregeln bestimmt werden. Auch können dadurch Geschäftsregeln leichter angepasst werden. Da sich die Unternehmen immer schneller an eine sich immer schneller ändernde Umwelt anpassen müssen, ist dies von großem Vorteil.

Zudem sollten sie in einer für Fachexperten verständlichen Form ausgedrückt werden, so dass sie nicht nur von IT-Experten, sondern auch direkt von Fachexperten eingegeben und geändert werden können. Dadurch kann das Geschäftswissen direkt von den dafür zuständigen Personen geändert werden.

3.3.1 Ausgangslage

Der Business Rules-Ansatz (*Business Rules Approach*) versucht, zwei Probleme zu lösen:

- In einem Unternehmen ist oft nicht bekannt, aus welchen Gründen manche Geschäftsregeln angewendet werden. So kann es sein, dass sie früher zur Erreichung der Unternehmensziele beigetragen haben, aber jetzt nicht mehr aktuell sind oder sich sogar negativ auf die Erreichung der Unternehmensziele auswirken. Wenn die Motivation der Geschäftsregel unbekannt ist, kann sie folglich auch nicht geändert oder abgeschafft werden. [GS06]
- Unternehmen und ihre IT-Systeme können nicht schnell genug an die sich ständig ändernde Umwelt angepasst werden. [GS06]

3.3.2 Grundkonzepte

Die Grundkonzepte des Business Rules-Ansatzes werden prägnant durch die STEP-Prinzipien [vH02] zusammengefaßt:

***S*eparate** Damit ist gemeint, dass die Geschäftsregeln von den Prozessen und den anderen Aspekten des Systems getrennt werden sollen. Die Geschäftsregeln werden somit als etwas Eigenständiges behandelt. Das hat zur Folge, dass die Geschäftsregeln wiederverwendet und unabhängig vom restlichen System geändert werden können. [vH02]

***T*race** Zu jeder Geschäftsregel wird ihr Zweck festgehalten. Damit kann später bestimmt werden, inwieweit die Geschäftsregel ihren Zweck erfüllt. Daneben wird zu jeder Geschäftsregel noch festgehalten, an welchen Stellen sie ausgeführt wird. So kann später leicht gesehen werden, was alles von einer Regeländerung betroffen ist. [vH02]

***E*xternalize** Da manche Geschäftsregeln nur implizit in den Köpfen der Mitarbeiter vorhanden sind, sollen zum einen die Geschäftsregeln bewußtgemacht und dokumentiert werden und zum anderen sollen sie in einem Format ausgedrückt werden, das für die Fachleute verständlich ist. Damit wissen die Fachleute, welche Geschäftsregeln es gibt und können dann auch selber ihre Geschäftsregeln ändern. Im Gegensatz dazu können die Fachleute die Geschäftsregeln nicht mehr verstehen bzw. ändern, wenn die Geschäftsregeln „hart verdrahtet“ im Programmcode vorliegen. [vH02]

***P*osition** Geschäftsregeln sollen zum Ändern positioniert werden, indem sie z.B. in einer Rule Engine implementiert werden, so dass sie leicht und schnell geändert werden können. [vH02]

3.4 Zusammenfassung

Dieses Kapitel hatte zum Ziel, einen Überblick über Geschäftsregeln zu geben. Zuerst wurde erklärt, dass unter dem Begriff Geschäftsregeln konkrete Aussagen zur Steuerung des Geschäftsverhaltens verstanden werden. Danach wurden die verschiedenen Kategorien von Geschäftsregeln kurz vorgestellt, wobei unser Hauptaugenmerk in der restlichen Arbeit auf den Aktionsaussagen liegen wird. Mit ihnen werden wir nämlich die für die Prozessverifikation notwendige semantische Information hinterlegen. Zum Schluss wurde noch der

Business Rules Ansatz vorgestellt. Dabei geht es im Prinzip darum, die Geschäftsregeln zu externalisieren, damit sich die Mitarbeiter deren bewußt werden und damit diese leichter angepasst werden können.

Kapitel 4

Ansätze zur Integration von Geschäftsregeln in Prozess-Management-Systeme

Nachdem im vorangegangenen Kapitel ein allgemeiner Überblick über Geschäftsregeln gegeben wurde, werden wir in diesem Kapitel speziell den Einsatz von Geschäftsregeln in PMS betrachten. Im ersten Teil geht es darum, wie Geschäftsregeln in PMS eingesetzt werden können, um Prozesse zu modellieren. Danach werden Business Rules Engines vorgestellt, mit denen Geschäftsregeln in PMS integriert werden können. Im Anschluss folgen die aktuellen Forschungsansätze im Bereich der Adaption von Prozessen mit Hilfe von Geschäftsregeln. Dabei kann man zwischen Ansätzen unterscheiden, die die Adaptionsmöglichkeiten von Prozessen zur Laufzeit einschränken und Ansätzen, die Prozesse automatisch adaptieren. Zuletzt werden Ansätze zur semantischen Prozessverifikation mit Geschäftsregeln behandelt, wobei die Ansätze dort in eine der drei Kategorien logikbasierte, modellbasierte und graphbasierte Verifikation fallen.

4.1 Prozessmodellierung mit Geschäftsregeln

Hier geht es weniger um die Integration von Geschäftsregeln in PMS, sondern viel mehr um die Modellierung bzw. Beschreibung von Prozessen mit Geschäftsregeln. Prozesse lassen sich mit unterschiedlichen Formalismen beschreiben, z.B. mit Petri-Netzen, Aktivitätennetzen oder ADEPT WSM-Netzen. Sie lassen sich aber auch mittels Regeln beschreiben. Die beiden vorgestellten Ansätze [KEP00] und [BBKK04] verwenden zur Modellierung des Kontrollflusses Geschäftsregeln in Form von ECA-Regeln. ECA-Regeln bestehen aus den drei Komponenten *E*vent, *C*ondition und *A*ction. Die Event-Komponente spezifiziert das Ereignis, das eintreten muss, damit die Regel überhaupt ausgewertet wird. Bei der Regelauswertung wird die in der Condition-Komponente spezifizierte Bedingung überprüft. Ist die Bedingung erfüllt, wird die in der Action-Komponente spezifizierte Aktion ausgeführt. Die Beendigung dieser Aktion kann wiederum ein oder auch mehrere Ereignisse auslösen.

Für beide Ansätze gilt, dass sie zwar ausdrucksmächtig sind, aber für die Spezifikation von komplexen Prozessen absolut untauglich sind. Das liegt daran, dass regelbasierte Prozessbeschreibungen schlecht visualisierbar und damit schlecht wartbar sind. Auch gibt es im Gegensatz zu Petri-Netzen z.B. nur wenige Möglichkeiten zur formalen Analyse, z.B. des dynamischen Verhaltens. Eine regelbasierte Prozessbeschreibung kann nur formal analysiert werden, indem sie mit einem anderen analysierbareren Formalismus, wie z.B. Petri-Netze, ausgedrückt wird [Dad07].

4.2 Integration von Geschäftsregeln in PMS mit Business Rules Engines

Der Zweck der Integration von Geschäftsregeln in ein PMS mit einer *Business Rules Engine (BRE)* ist, Entscheidungen bzgl. der Prozessausführung oder Entscheidungen innerhalb von Aktivitätenknoten auszulagern. So kann z.B. die Logik eines XOR-Splits mit einer BRE ausgelagert werden (s. Abbildung 4.1). Der Vorteil davon ist, wie in Unterkap. 3.3 erläutert, dass die Regeln einfacher geändert werden können, wenn sie unabhängig vom umgebendem System gehalten werden. Am Beispiel von *IBM Websphere MQ Workflow* [IBM07] und *ILOG JRules* [ILO07] soll gezeigt werden, wie eine BRE an ein PMS gebunden werden kann, wobei die Funktionalität von *ILOG JRules* über eine reine BRE hinausreicht, da mit *ILOG JRules* auch Geschäftsregeln eingegeben und verwaltet werden können.

Für die Anbindung von externen Programmen besitzt *MQ Workflow* u.a. ein Framework namens *User-defined Program Execution Server (UPES)*. Damit können sich beliebige Prozessknoten mit externen Anwendungen verbinden. *ILOG JRules UPES* ist eine Implementierung dieses Frameworks, mit dem sich die Prozessknoten mit *ILOG JRules* verbinden können (s. Abbildung 4.2). Zur Laufzeit kann z.B. ein Prozess auf Basis dessen, ob ein Kunde kreditwürdig ist oder nicht, eine bestimmte Verzweigung ausführen. Falls die Regeln zur Feststellung, ob der Kunde kreditwürdig ist, in *ILOG JRules* gehalten werden, sendet *MQ Workflow* eine sog. Aufrufanfrage an *ILOG JRules UPES* beim Erreichen des entsprechenden Verzweigungsknoten, wobei die Regeln in ILOG JRules als *if-then-else* Regeln vorliegen. In diesem Fall (s. Abbildung 4.1) muss eine Aufrufanfrage die Prozessdaten *debts level*, *income* und *property* als Eingabeparameter enthalten. So wird z.B. in der Instanz I_1, bei der geprüft wird, ob Herr Meier einen Kredit erhalten kann, die Aufrufanfrage mit den konkreten Werten 3100 Euro für das monatliche Einkommen, 11000 Euro für das Eigentum und -3000 Euro für den Schuldenstand übertragen. *ILOG JRules UPES* bestimmt dann aus der Aufrufanfrage die Eingabeparameter und die auszuführenden Regeln, worauf es diese an *ILOG JRules* weiterleitet, das diese Regeln mit diesen Eingabeparametern ausführt. In diesem Fall bestimmt *ILOG JRules UPES* aus der Aufrufanfrage, dass die Regel R_1 mit den übergegebenen Eingabeparametern aufgerufen werden muss. Die Regel R_1 bestimmt, ob einem Kunden ein Kredit verliehen werden darf und liefert genau dann *true* zurück, wenn das monatliche Einkommen größer als 3000 Euro, das Eigentum mehr als 10000 Euro und der Schuldenstand weniger als 1000 Euro beträgt. Die Daten *mo-*

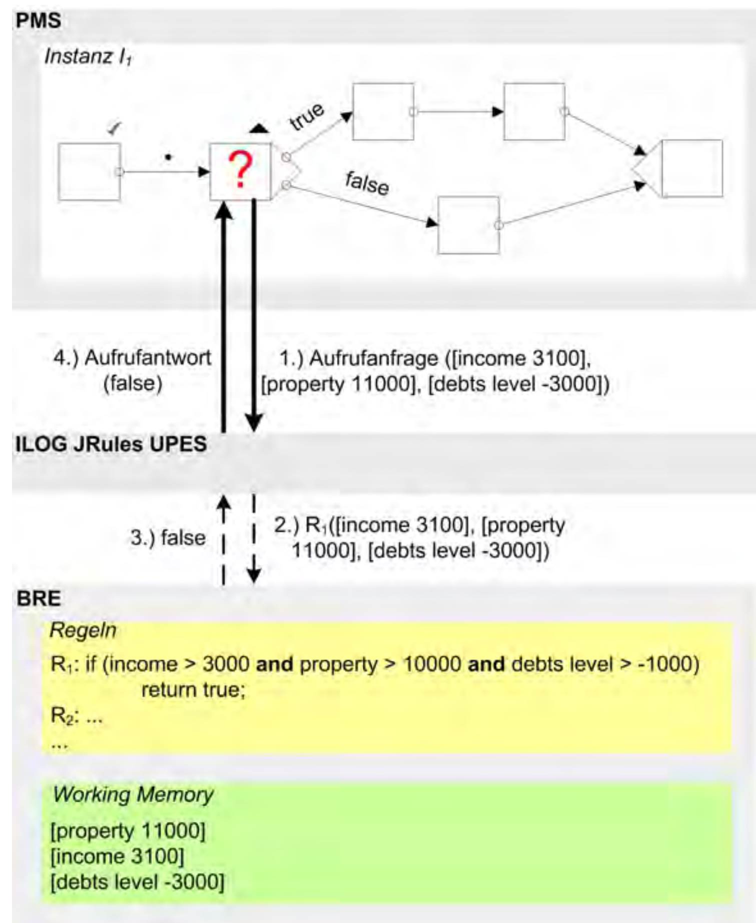


Abbildung 4.1: Kopplung einer BRE an ein PMS am Beispiel von *ILOG JRules* und *MQ Workflow*

natliches Einkommen gleich 3100 Euro, *Eigentum gleich 11000 Euro* und *Schuldenstand gleich -3000 Euro* bilden hier das sog. *Working Memory*, d.h. die Fakten, gegen die die Regeln ausgewertet werden. R_1 liefert wiederum für diese als Ausgabeparameter, ob der Kunde kreditwürdig ist, *false* zurück, da der Schuldenstand mehr als 1000 Euro beträgt. Der Rückgabewert wird an *ILOG JRules UPES* weitergereicht, das diesen in eine Aufrufantwort verpackt und an *MQ Workflow* weiterleitet. Auf Basis dieser Ausgabeparameter kann in dem Prozess z.B. die entsprechende Verzweigung genommen werden [ILO05].

Wenn eine BRE naiv implementiert wäre, müsste bei jeder Änderung des *Working Memory* jeder Bedingungsteil einer Geschäftsregel einzeln gegen den ganzen *Working Memory* überprüft werden. Dies wäre ziemlich ineffizient, denn zum einen betrifft eine Änderung des *Working Memorys* meistens nur wenige Objekte. Damit sind auch nur wenige Bedingungsteile von solchen Änderungen betroffen. Zum anderen sind die einzelnen Bedingungen der Bedingungsteile auch teilweise identisch. Es reicht also, die einzelnen Bedingungen, die in mehreren Bedingungsteilen von Geschäftsregeln vorkommen, nur einmal auszuwerten. Beides berücksichtigt der Rete-Algorithmus, der in den meisten BREs implementiert ist, um effizient die Geschäftsregeln zu bestimmen, deren Bedingungsteile erfüllt sind und die

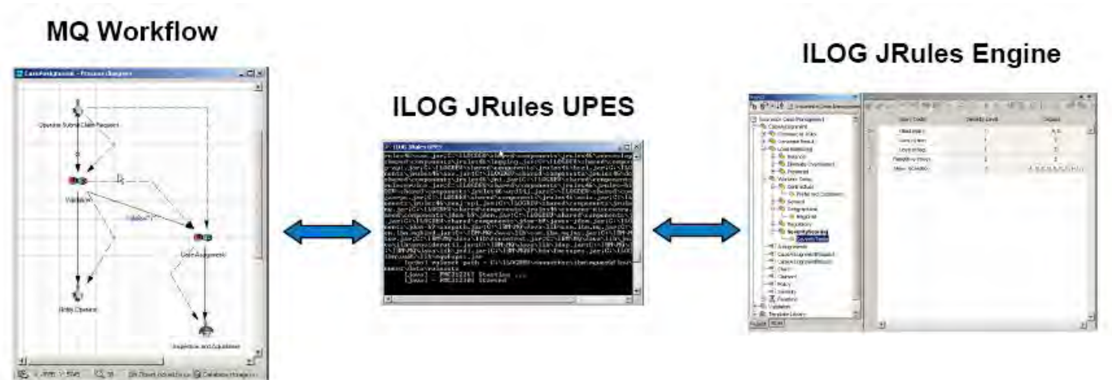


Abbildung 4.2: Anbindung von ILOG JRules an MQ Workflow

damit nur neu ausgewertet werden müssen.

Dazu erzeugt der Rete-Algorithmus für den Bedingungsteil jeder Geschäftsregel ein sog. Rete-Netz. Für den Bedingungsteil der im vorigen Beispiel vorgestellten Geschäftsregel R_1 sieht das entsprechende Rete-Netz wie in Abbildung 4.3 aus. Allgemein wird ein Rete-Netz so aufgebaut, dass jeder atomaren Bedingung des Bedingungsteils ein sog. Alphaknoten entspricht. Hier prüfen die drei Alphaknoten jeweils die Bedingung *Einkommen größer als 3000 Euro (income > 3000)*, *Eigentum größer als 10000 Euro (property > 10000)* bzw. *Schuldenstand weniger als 1000 Euro (debts level > -1000)*. Bei der Bestimmung, ob der Bedingungsteil einer Geschäftsregel zutrifft, werden die Elemente des *Working Memory*, die sog. WMEs, zuerst sukzessive den Alphaknoten präsentiert und geprüft, ob deren Bedingung durch ein WME erfüllt wird. Wenn ein WME die Bedingung eines Alphaknoten erfüllt, wird es in den zum Alphaknoten gehörenden Alpha-Memory-Knoten gespeichert. Da hier die Bedingungen *Einkommen größer als 3000 Euro (income > 3000)* und *Eigentum größer als 10000 Euro (property > 10000)* der Alphaknoten erfüllt sind, wird das entsprechende WME *[income 3100]* und *[property 11000]* im jeweiligen Alpha-Memory-Knoten gespeichert. Da die Bedingung *Schuldenstand weniger als 1000 Euro (debts level > -1000)* nicht erfüllt ist, bleibt der Alpha-Memory-Knoten des entsprechenden Alphaknotens leer.

Die einzelnen Alpha-Memory-Knoten sind wiederum mit sog. Betaknoten verbunden. Diese entsprechen den Junktoren im Bedingungsteil, die die atomaren Bedingungen miteinander verknüpfen und stellen damit zusammengesetzte Bedingungen dar. Der Betaknoten in unserem Rete-Netz entspricht dabei der Bedingung *Einkommen größer als 3000 Euro und Eigentum größer als 10000 Euro und Schuldenstand weniger als 1000 Euro (income > 3000 $\wedge$ property > 10000 $\wedge$ debts level > -1000)*, also dem kompletten Bedingungsteil. Um speichern zu können, welche Tupel von WMEs die Bedingung des Betaknoten erfüllen, wird der Betaknoten wiederum von einem sog. Beta-Memory-Knoten gefolgt. In unserem Fall entspricht der Betaknoten dem Konjunktionsoperator. Das bedeutet, dass ein Tupel von WMEs hier nur in den zugehörigen Beta-Memory-Knoten abgespeichert wird, wenn alle Bedingungen der Alphaknoten erfüllt sind und damit alle Alpha-Memory-Knoten mindestens ein WME enthalten. In unserem Fall ist der Alpha-Memory-Knoten des Alphaknoten

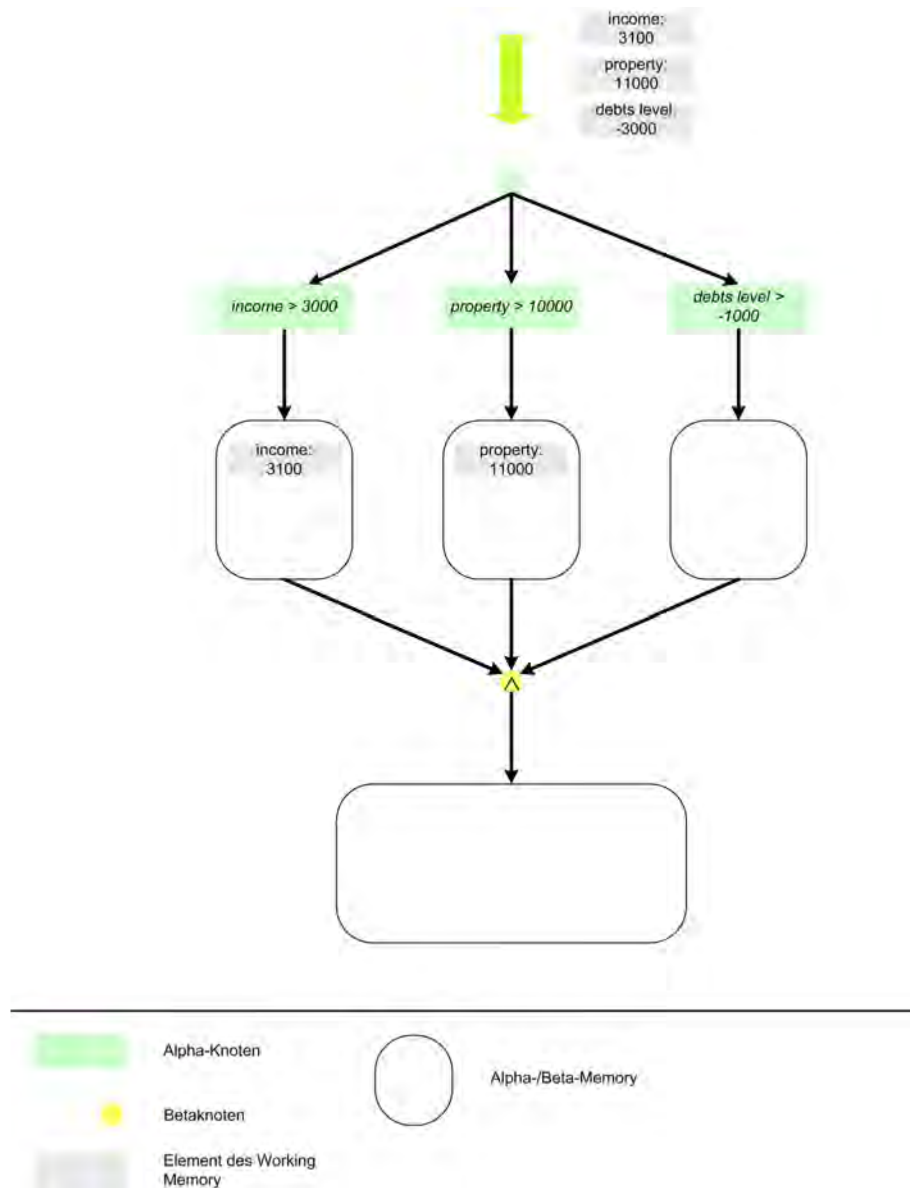


Abbildung 4.3: Rete-Netz zur Geschäfsregel R_1

mit der Bedingung *Schuldenstand weniger als 1000 Euro (debts level > -1000)* leer. Folglich bleibt auch der Beta-Memory-Knoten leer.

In unserem Rete-Netz ist der Beta-Memory-Knoten auch gleichzeitig der sog. Terminalknoten. Der Terminalknoten ist immer der letzte Memory-Knoten im Rete-Netz. Dieser enthält alle WMEs bzw. Tupel von WMEs, die den gesamten Bedingungsteil erfüllen. Damit ist der Bedingungsteil einer Geschäftsregel dann erfüllt, wenn der Terminalknoten des zugehörigen Rete-Netzes mindestens ein Element bzw. ein Tupel enthält. Folglich muss auch die Geschäftsregel neu ausgewertet werden. In unserem Beispiel ist der Terminalknoten der Beta-Memory-Knoten. Da dieser kein Tupel enthält, ist der Bedingungsteil der Regel R_1 nicht erfüllt und R_1 wird nicht neu ausgewertet und somit nicht true zurückgegeben.

Da bei einem Durchlauf des Rete-Algorithmus die entsprechenden Fakten in den jeweiligen Alpha-Memory- bzw. Beta-Memory-Knoten gespeichert werden, müssen ab dem ersten Durchlauf nur noch die Änderungen im *Working Memory* in das Rete-Netz eingegeben werden. Wenn Herr Meier beispielsweise seine Schulden komplett tilgt und sich an seinem Einkommen und Eigentum nichts ändert, müssen nicht mehr sämtliche Bedingungen des Bedingungsteils überprüft werden, da sich das Rete-Netz die erfüllten Bedingungen „merkt". Folglich muss nur die Änderung des Working Memory, das neu hinzugekommene Element *[debts level 0]* dem Rete-Netz präsentiert werden (s. Abbildung 4.4). Dadurch ist

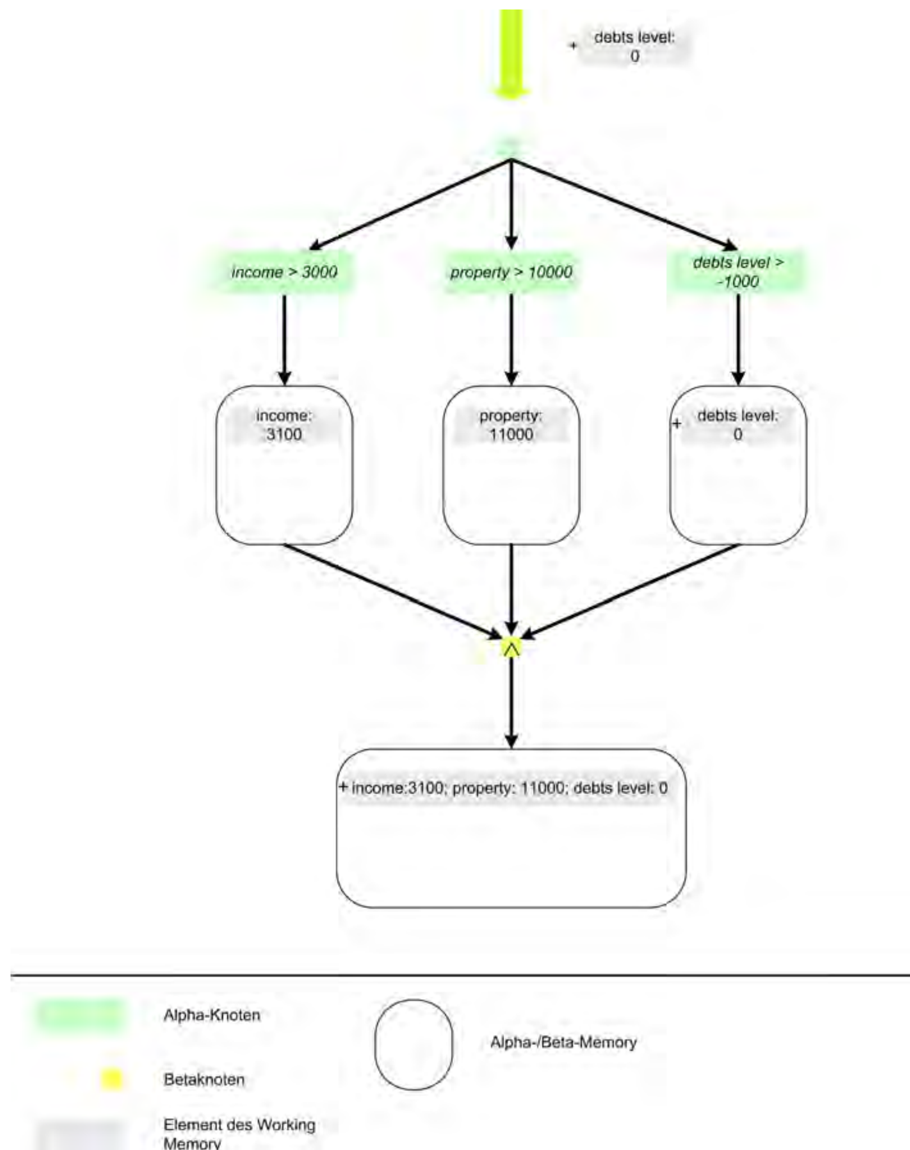


Abbildung 4.4: Rete-Netz für die Geschäfsregel R_1 nach einer Änderung des *Working Memory*

jetzt die Bedingung *Schuldenstand weniger als 1000 Euro (debts level > -1000)* des rechten Alphaknoten erfüllt. Folglich wird dieses WME *[debts level 0]* in den zugehörigen Alpha Memory-Knoten gespeichert. Dadurch ist jetzt ebenfalls die Bedingung des Betaknoten er-

füllt, da mittlerweile jeder Alpha-Memory-Knoten ein WME enthält und somit alle Bedingungen der Alphaknoten erfüllt sind. Der Beta-Memory-Knoten, der in diesem Rete-Netz den Terminalknoten bildet, ist somit auch erfüllt. Der Bedingungsteil der Geschäfsregel R_1 ist damit erfüllt und wird neu ausgewertet. Folglich wird *true* zurückgegeben.

Da eine BRE meistens mehrere Geschäftsregeln auswertet, wäre es ungeeignet, für jede Geschäftsregel ein einzelnes Rete-Netz zu halten. Deswegen setzt der Rete-Algorithmus die einzelnen Rete-Netze zu einem Rete-Netzwerk zusammen, indem die gleichen Anfangsteile der Netze miteinander verschmolzen werden. Dadurch müssen auch Bedingungen, die in mehreren Bedingungsteilen vorkommen, nur einmal ausgewertet werden [FH 04].

4.3 Einschränkung und Automation der Prozessadaption

Die Ansätze in diesem Abschnitt behandeln die Steuerung bzw. Beeinflussung der Prozessadaption mit Hilfe von Geschäftsregeln. Dabei werden zwei verschiedene Arten von Ansätzen unterschieden:

- Ansätze zur Einschränkung der Prozessadaption mit Geschäftsregeln
- Ansätze zur automatischen Prozessadaption mit Geschäftsregeln

4.3.1 Einschränkung der Adaptions- und Kompositionsmöglichkeiten von Prozessen

Der Ausgangspunkt bei den Ansätzen von Sadiq et al. [SOS05] und von Wainer et al. [WBB04] ist, dass ein Prozessschema zur Modellierungszeit nicht komplett spezifiziert werden kann. Dafür kann es mehrere Gründe geben: Zum einen kann es sein, dass zur Modellierungszeit nicht der gesamte Prozess bekannt ist und zum anderen, dass der Prozess von Mal zu Mal angepasst werden muss. So kann z.B. ein Prozess, der einen Testpan darstellt, um die Ursache für einen Defekt bei einem Elektrogerät zu finden, nicht komplett zur Modellierungszeit spezifiziert werden. Es müssen nämlich je nach Defekt unterschiedliche Einzeltests durchgeführt werden. Zudem wird angenommen, dass die Einzeltests, die den Aktivitäten T1 bis T5 entsprechen, beliebig kombiniert werden können mit den zwei Einschränkungen, dass T2 immer vor T3 ausgeführt werden muss bzw. T4 und T5 immer parallel ausgeführt werden müssen.

Wegen der Vielzahl der möglichen Alternativen können alle Alternativen nicht zur Modellierungszeit im Kontrollflussgraph modelliert werden, da dieser sonst unübersichtlich wird. Stattdessen wird dem Benutzer die Möglichkeit gegeben, die Prozessinstanz zur Laufzeit zu adaptieren. Dabei soll die fertig spezifizierte Prozessinstanz aber gewissen Regeln entsprechen. Deshalb entwickelten Sadiq et al. [1] das Konzept von zur Modellierungszeit nicht vollständig vordefinierten Subprozessen, den sog. *pockets of flexibility*. Ihr Zweck liegt

[1]Da die Verfahren von Sadiq et al. und Wainer et al. nach dem gleichen Schema ablaufen, das Verfahren von Sadiq et al. aber mächtiger ist, wird nur das Verfahren von Sadiq et al. vorgestellt.

darin, einen Kompromiss zwischen Flexibilität und Kontrolle zu schaffen. Auf der einen Seite bieten sie mehr Flexibilität als ein komplett definiertes Prozessmodell, aber auf der anderen Seite geben sie mehr Regeln vor als ein komplett unspezifiziertes Prozessmodell. Dazu werden die als *pockets of flexibility* modellierten Subprozesse erst zur Laufzeit vom Benutzer komplett spezifiziert. Dabei muss sich der Benutzer an gewisse Einschränkungen halten. So darf er nur die in der *pocket of flexibility* angegebenen Prozessfragmente unter Einhaltung der ebenfalls dort angegebenen semantischen Integritätsregeln verwenden. Das erwähnte Beispiel wird dann wie in Abbildung 4.5 gezeigt mit einer *pocket of flexibility* modelliert.

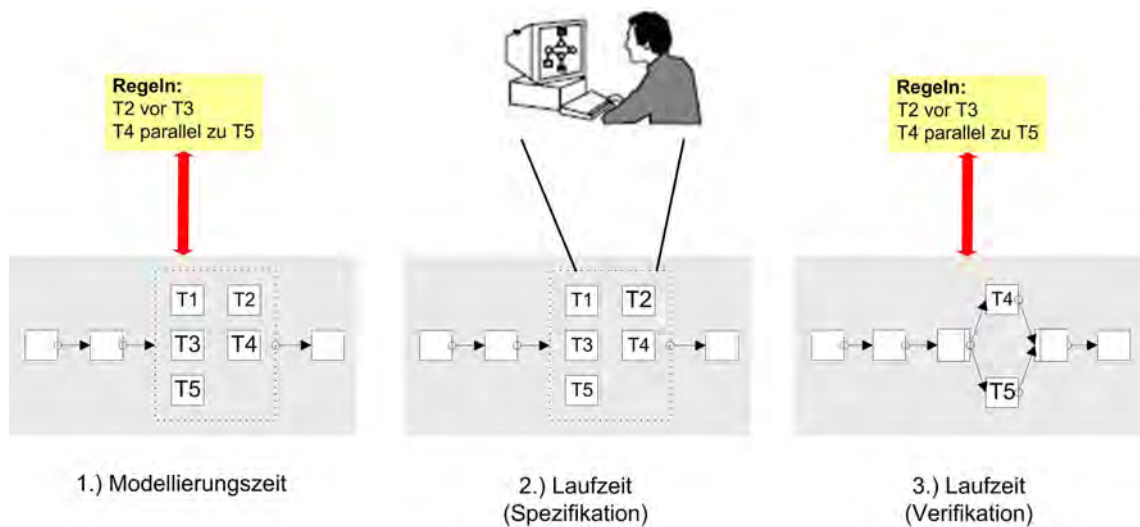


Abbildung 4.5: Einschränkung der Adaptionsmöglichkeiten mit einer *pocket of flexibility*

Allgemein können für die *pockets of flexibility* folgende Arten von Integritätsregeln angegeben werden:

- Integritätsregeln zur Ablaufplanung, um z.B. die Ausführungsreihenfolge bestimmter Prozessfragmente festzulegen
- Strukturelle Integritätsregeln, die angeben, wie bestimmte Prozessfragmente zusammengesetzt werden können, z.B. ob zwei bestimmte Prozessfragmente nur parallel oder nur seriell ausgeführt werden können.
- Abhängigkeits- und Exklusions-Integritätsregeln, um z.B. anzugeben, dass wenn ein bestimmtes Prozessfragment in dem fertig definierten Subprozess vorkommt, ebenfalls ein anderes Prozessfragment vorkommen muss bzw. nicht vorkommen darf

Zur Laufzeit wird dann die Prozessinstanz ausgeführt, bis ein solches *pocket of flexibility* erreicht wird. In dem Fall muss der Benutzer dieses *pocket of flexibility* fertig spezifizie-

ren, worauf diese Adaption dann systemseitig mit Hilfe der hinterlegten Integritätsregeln verifiziert wird.

Leider hat dieser Ansatz einige Nachteile. Ein Nachteil ist, dass bereits zur Modellierungszeit bekannt sein muss, wo häufig Änderungen vorkommen, damit dieser Subprozess als *pocket of flexibility* spezifiziert wird. Manche Integritätsbedingungen lassen sich mit diesem Ansatz auch nur kompliziert ausdrücken, da die Integritätsregeln immer nur innerhalb eines *pocket of flexibility* gelten. Um z.B. die Existenzabhängigkeit einer Aktivität am Prozessanfang von einer anderen Aktivität am Prozessende darzustellen, müsste der gesamte Prozess als *pocket of flexibility* modelliert werden.

4.3.2 Automatische Adaption von Prozessen

Bei der automatischen Adaption von Prozessen ist die Idee, den Prozess durch Adaptionen über die Zeit konsistent zu halten, da bei vielen Prozessinstanzen zur Laufzeit verschiedene Situationen bzw. Ausnahmen vorkommen können, bei denen die Prozessinstanzen jeweils auf eine bestimmte Weise abgeändert werden müssen. Beispielsweise kann bei der klinischen Behandlung eines Patienten die Ausnahme auftreten, dass die Leukozytenzahl des Patienten unter einen bestimmten kritischen Wert absinkt. In diesem Fall sollte ein bestimmtes Medikament X zwei Tage lang abgesetzt werden. In dem entsprechenden klinischen Behandlungsprozess sollte also die Aktivität *Verabreiche Medikament X* im Zeitraum von zwei Tagen nach dem Absinken der Leukozytenzahl gelöscht werden. Allerdings lassen sich diese Ausnahmen, die jeweils immer auf die gleiche Weise behandelt werden müssen, mit der entsprechenden Prozessadaption nicht im Kontrollflussgraph modellieren, da nicht bekannt ist, ob und wann die Ausnahmen während der Prozessausführung auftreten. Stattdessen wird bei dem Ansatz von Müller et al. [MGR04, GRH+04] ein PMS mit einer Regelbasis gekoppelt. Die Regeln liegen dabei als erweiterte ECA-Regeln vor, wobei eine erweiterte ECA-Regel jeweils angibt, wie der Prozess bei einer bestimmten Ausnahme adaptiert werden muss. Die erweiterten ECA-Regeln besitzen zusätzlich zu der *Event*-, *Condition*- und *Action*-Komponente noch eine optionale *Time*-Komponente, in der die Gültigkeitsdauer der Prozessadaption spezifiziert werden kann. Die *Event*-Komponente enspricht dabei der speziellen Änderung der Prozessdaten. Bei einer entsprechenden Änderung wird die in der *Condition*-Komponente spezifizierte Bedingung ausgewertet. Ist diese erfüllt, wird die Prozessinstanz entsprechend der *Action*-Komponente automatisch ad-hoc modifiziert. Die im Beispiel genannte Ausnahmebehandlung wird entsprechend durch folgende ECA-Regel realisiert:

Event Neues Untersuchungsergebnis von Patient P

Condition Leukozytenzahl < 1000 #/mm^3

Action Lösche Aktivität „Verabreiche Medikament X“

Time 2 Tage

Das Besondere an diesem Ansatz ist, dass bei diesem im Grunde ein Prozess mit Hilfe eines Kontrollflussgraphen und ECA-Regeln modelliert wird. Dadurch lassen sich zum einen mehr Alternativen modellieren, ohne dass der Kontrollflussgraph unübersichtlich wird und zum anderen lassen sich auch Alternativen modellieren, bei denen zur Modellierungszeit nicht bekannt ist, ob sie überhaupt und wann sie auftreten und die sich nicht mit einem Kontrollflussgraphen allein modellieren lassen könnten.

4.4 Semantische Verifikation von Prozessen mit Geschäftsregeln

Bei Prozessen wird syntaktische und semantische Korrektheit unterschieden [Nah05, LRD06b]. Ein Prozessmodell ist syntaktisch korrekt, wenn es gemäß der Syntax des Prozessmetamodells konstruiert wurde, so dass z.B. keine Deadlocks auftreten können und eine Transition nur von einem einzigen Knoten abgehen kann. Je nach Prozessmetamodell, wie z.B. Petri-Netzen oder ADEPT WSM-Netzen, gibt es bereits verschiedene Verfahren, um Prozesse auf syntaktische Korrektheit zu untersuchen. Ein Prozess modelliert meistens einen bestimmten Ausschnitt der realen Welt. Wenn der Prozess nicht korrekt den zu modellierenden Ausschnitt wieder gibt, kann es sein, dass der Prozess zwar syntaktisch korrekt ist, aber keinen Sinn macht. In diesem Fall ist der Prozess semantisch nicht korrekt. Deswegen wird versucht, dem Prozess mit Hilfe von semantischer Information ein tieferes Verständnis zu geben. Ein Prozess gilt dann als semantisch korrekt, wenn er konsistent zu der für ihn geltenden semantischen Information ist. Bei den in dieser Arbeit vorgestellten Ansätzen zur semantischen Verifikation mit Geschäftsregeln liegt dabei die semantische Information in Form von Integritätsregeln bzw. Aktionsaussagen vor (s. Abbildung 4.6), einer speziellen Unterkategorie der Geschäftsregeln (s. Abschnitt 3.2.2). Wie in Abschnitt 3.2.2 auch beschrieben, sind Integritätsregeln bzw. Aktionsaussagen Aussagen, die immer erfüllt sein müssen. Bei einem klinischen Behandlungsprozess kann es z.B. eine Integritätsbedingung geben, dass niemals Aspirin und Marcumar zusammen verabreicht werden dürfen, da die beiden Medikamente unverträglich sind. Damit dieser Behandlungsprozess semantisch korrekt bzgl. dieser Integritätsregel ist, dürfen demnach in keiner Ausführungsspur die Aktivitäten *Verabreiche Aspirin* und *Verabreiche Marcumar* vorkommen. Zur semantischen Prozessverifikation werden nachfolgend logikbasierte, modellbasierte und graphbasierte Verfahren vorgestellt. Die logikbasierte Verifikation besteht darin, das Prozessschema und die zu verifizierenden Integritätsregeln jeweils als logische Formel darzustellen und diese logisch miteinander zu verknüpfen. Bei der modellbasierten Verifikation werden die Zustände und Zustandsübergänge des als Zustandsübergangssystems modellierten Prozesses analysiert, um den Prozess gegen eine Integritätsregel zu verifizieren. Bei der graphbasierten Verifikation schließlich wird der als Graph dargestellte Prozess mit Hilfe von Graphalgorithmen gegen die Integritätsregeln verifiziert. Im Folgenden gehen wir jeweils genauer auf die einzelnen Verfahren ein.

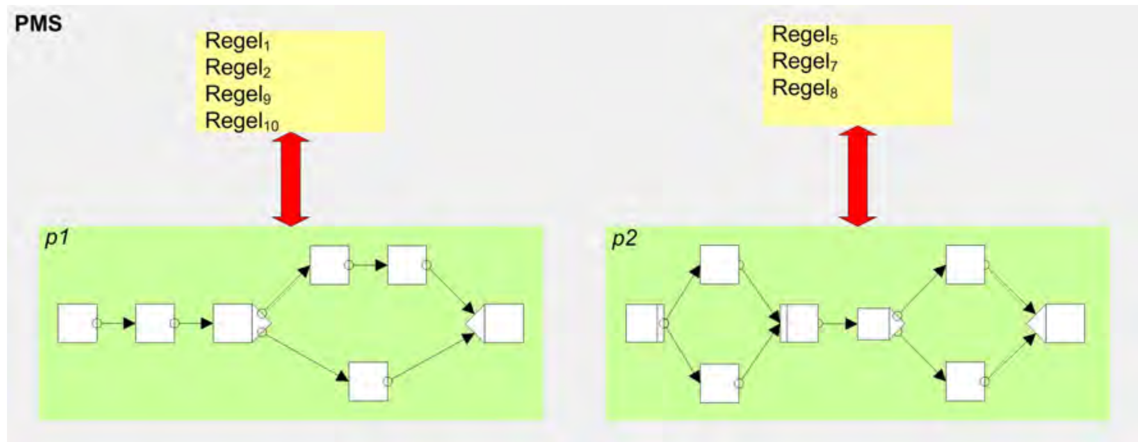


Abbildung 4.6: Schematische Darstellung der semantischen Verifikation mit Geschäftsregeln

4.4.1 Logikbasierte Verifikation

In [DKRR98] wird ein Verfahren zur Verifikation eines Prozessschemas gegen eine einfache Algebra von Integritätsregeln, mit der komplexe Aussagen über die Ausführung von Aktivitäten gemacht werden können, vorgestellt. Dabei werden das Prozessschema und alle Integritätsregeln als logische Formeln in *Concurrent Transaction Logic* (CTR) dargestellt. CTR erweitert die Prädikatenlogik um die Möglichkeit, die zeitliche Abfolge von Formeln auszudrücken. Mittels logischen Umformungen wird versucht, alle Integritätsregeln in das Prozessschema „hineinzukompilieren“ [DKRR98]. Gelingt dies nicht aufgrund eines logischen Widerspruchs, ist das Prozessschema zu den Integritätsregeln nicht konsistent. Ansonsten ist das Prozessschema semantisch korrekt. Ein Vorteil dieses Ansatzes ist, dass er auf Logik basiert, die bekanntlich sehr ausdrucksmächtig ist. Folglich kann der Ansatz leicht erweitert werden, um weitere Integritätsregeln zu verifizieren. Ein Nachteil hingegen ist, dass das Verfahren nicht für die Adaptivität von Prozessen optimiert ist. Das bedeutet, es wird nach jeder Prozessänderung jeweils der Prozess gegen sämtliche Integritätsregeln verifiziert. Allerdings genügt es nur die Integritätsregeln bei der Verifikation zu betrachten, die durch die Prozessänderung verletzt sein könnten, da vor der Änderung der Prozess bereits semantisch korrekt gewesen ist.

4.4.2 Modellbasierte Verifikation

Zur modellbasierten Verifikation von Prozessen kann das sog. Model Checking eingesetzt werden. Model Checking ist ein Verfahren zur automatischen Verifikation einer Systembeschreibung. Dabei prüft der sog. Model Checker, ob die Systembeschreibung, das sog. Modell, eine bestimmte Anforderung, die sog. Spezifikation, erfüllt. Der Model Checker gibt dann entweder aus, dass das Modell die Spezifikation erfüllt oder falls das Modell die Spezifikation nicht erfüllt ein Gegenbeispiel, das zeigt, warum das Modell die Spezifikation nicht erfüllt. Meistens wird das Modell als ein Zustandsübergangssystem und die Spezifikation mit einer temporallogischen Formel dargestellt. Mit der Temporallogik lassen

sich bestimmte Anforderungen an einzelne Zustände oder an Zustandsfolgen des Zustandsübergangssystems definieren. So lassen sich z.B. Anforderungen stellen, dass immer ein Endzustand erreicht werden soll oder dass das System immer wieder auf den Anfangszustand zurückgehen soll. Beim Model Checking wird dann das Zustandsübergangssystem nach Zuständen durchsucht, die diese Anforderungen erfüllen [Pal04].

Ein Problem ist allerdings, dass die Anzahl der Zustände des Zustandsübergangssystems ziemlich groß werden kann. Diese Tatsache bezeichnet man auch als Zustandsexplosion. Allerdings gibt es verschiedene Ansätze, um dem Problem der Zustandsexplosion zu begegnen. Ein Ansatz ist, symbolische Algorithmen zu verwenden. Diese stellen das System statt als eine Art Zustandsübergangssystem als aussagenlogische Formel dar, die sich effizienter bearbeiten lässt. Abstraktion ist ein anderer Ansatz. Dabei geht es darum, das System zu vereinfachen und damit den Zustandsraum zu reduzieren. Um z.B. ein Softwaresystem auf gegenseitigen Ausschluss zu verifizieren, können als Abstraktion des ganzen Programmcodes nur die booleschen Variablen und die Kontrollflusskonstrukte genommen werden. Eine Abstraktion sollte zwar immer korrekt sein, d.h. dass die Eigenschaften, die für die Abstraktion erfüllt sind, auch für das Originalsystem erfüllt sind. Allerdings ist die Abstraktion oft nicht vollständig, d.h. dass nicht alle Eigenschaften, die für das Originalsystem erfüllt sind, auch für die Abstraktion gelten [Wik07b].

Im Kontext von PMS stellt das zu verifizierende Prozessschema das Modell und die Integritätsbedingung, gegen die das Prozessschema verifiziert werden soll, die Spezifikation dar. Bei den aktuellen Ansätzen erfolgt die Verifikation zur Modellierungszeit. Im Falle, dass das Modell nicht der Spezifikation genügt, bietet es sich an, als Gegenbeispiel die bzw. eine Ausführungsspur anzugeben, die die Spezifikation verletzt. Zur Behebung des Problems der Zustandsexplosion verwenden die hier vorgestellten Ansätze das Mittel der Abstraktion.

Mit dem von Gruhn et al. [GL05] vorgeschlagenen Model Checking-Verfahren kann die strukturelle Korrektheit von Prozess-Schemas verifiziert werden. Dazu wird geprüft, ob jede mögliche Ausführung des Prozesschema den Endeknoten erreicht und ob alle gestarteten Aktivitäten bis dahin beendet werden. Mit diesem Model Checking-Verfahren kann zusätzlich auch verifiziert werden, ob das Prozess-Schema gewisse Deadlines einhält und ob Ressourcen-Konflikte auftreten können. Es gibt nämlich Ressourcen, die nur exklusiv genutzt werden können, d.h. sie können zu einem bestimmten Zeitpunkt nur von einer Aktivtät genutzt werden. Wenn mehrere Aktivitäten aber gleichzeitig auf eine Ressource zugreifen wollen, kommt es zu Resourcenkonflikten und dadurch im schlimmsten Fall zum Überschreiten von Deadlines bzw. zu Deadlocks. Ob eine Ressource zu einem bestimmten Zeitpunkt von mehr als einer Aktivität gebraucht wird, kann mit Hilfe der für jede Aktivität angegebenen minimalen und maximalen Ausführungszeit bestimmt werden. Desweiteren lassen sich Spezifikationen angeben, in denen eines der folgenden Patterns oder mehrere Patterns verknüpft durch $\wedge$ und $\vee$ angegeben werden:

- Absence: Eine bestimmte Aktivität kommt in keiner Spur vor.
- Existence: Eine bestimmte Aktivität kommt in jeder Spur vor.

- Time-bounded Existence: Eine bestimmte Aktivität kommt innerhalb einer bestimmten Zeitspanne vor.
- Precedence: In jeder Spur folgt auf eine bestimmte Aktivität A eine andere Aktivität B.
- Time-bounded Precedence: In jeder Spur kann eine bestimmte Aktivität B nur nach einer bestimmten Zeitspanne auf eine andere Aktivität A folgen.

  In jeder Spur kann nur innerhalb einer bestimmten Zeitspanne auf eine bestimmte Aktivität A eine andere Aktivität B folgen.
- Response: In jeder Spur folgt auf eine bestimmte Aktivität A eine andere Aktivität B.
- Time-bounded Response: In jeder Spur folgt auf eine bestimmte Aktivität A eine andere Aktivität B innerhalb einer bestimmten Zeitspanne.

Zusätzlich kann definiert werden, für welchen Bereich des Prozessmodells die angegebenen Pattern gelten sollen, z.B. global für das ganze Prozessmodell, vor bzw. nach einer bestimmten Aktivität oder zwischen zwei Aktivitäten.

Bei diesem Verfahren wird das Modell als sog. UPPAAL-Modell und die Spezifikation in TCTL (Timed Computational Tree Logic), einer besonderen temporalen Logik, ausgedrückt. Die Grundelemente eines UPPAAL-Modells sind Prozesse, Zähler und Kanäle. Die Prozesse werden durch Zustandsübergangssysteme dargestellt. Ihre Synchronisierung erfolgt mit Hilfe von Kanälen. Wenn z.B. eine Transition in einem Prozess X passiert wird, kann in einen bestimmten Kanal *channel1* geschrieben werden. In einem anderen Prozess Y kann die Leseoperation des Kanals channel1 als sog. *Guard* einer Transition dienen. Das bedeutet, dass diese Transition nur genommen werden kann, wenn aus *channel1* gelesen werden kann und damit zuvor *channel1* geschrieben wurde. Beim Überqueren einer Transition kann ein Zähler verändert werden. Eine Bedingung bzgl. eines Zählers kann auch als Guard einer Transition verwendet werden.

Auf folgendem Weg wird ein Prozessmodell in ein UPPAAL-Modell transformiert: Jeder Knoten wird als eigenständiger Prozess modelliert. Jede Kontrollflusskante wird als ein Kanal modelliert. Damit wird ein Knoten mit nur einer eingehenden Kontrollflusskante aktiviert, wenn der der Kontrollflusskante entsprechende Kanal gelesen werden kann. Analog verhält es sich mit AND- und OR-Join-Knoten. Diese werden aktiviert, wenn alle den Kontrollfusskanten ensprechende Kanäle beschrieben wurden bzw. einer der den Kontrollfusskanten ensprechender Kanal beschrieben wurde. Um den bzw. die nachfolgenden Knoten bei einem AND-Split zu aktivieren, muss in den bzw. die Kanäle, die den ausgehenden Kontrollflusskanten entsprechen, geschrieben werden. Bei diesem Ansatz enthalten die Prozessmodelle zusätzlich noch für jede Aktivität die von ihr benötigten nur exklusiv nutzbaren Ressourcen sowie die minimale und maximale Ausführungszeit, um etwaige Ressourcenkonflikte und Nicht-Einhaltung von Deadlines überprüfen zu können. Zur Aufnahme dieser Informationen besitzt jeder Prozess eines UPPAAL-Modells, der einem Ak-

tivitätenknoten entspricht, passende Variablen. Zeitbezogene Spezifikationen werden dann mit diesen Variablen und Zählern ausgewertet. Bei diesem Ansatz wird das Problem der Zustandsexplosion mittels Abstraktion angegangen, indem nicht alle Eigenschaften des Prozessmodells in das UPPAAL-Modell transformiert werden. Um z.B. ein Prozessmodell gegen die Spezifikation „Es darf bzgl. der Ressource r10 kein Ressourcenkonflikt auftreten." zu verifizieren, können Informationen über die anderen Ressourcen ignoriert werden.

Andere Ansätze wie z.B. der von Karamanolis et al. [KGMW99] und der von Wong [WG06, Won06] stellen das Modell und die Spezifikation jeweils in einer Prozessalgebra dar. Eine Prozessalgebra ist ein Verfahren zur formalen Modellierung von nebenläufigen Systemen. Dabei erfolgen die Interaktionen und Synchronisierungen von bzw. zwischen den verschiedenen unabhängigen Prozessen durch den Austausch von Nachrichten. Mit Hilfe von algebraischen Gesetzen können diese Modelle von nebenläufigen Systemen manipuliert und analysiert werden [Wik07c].

Stellvertretend für diese Ansätze, die Prozessalgebra verwenden, soll hier der Ansatz von Wong näher dargestellt werden. Der Ansatz von Wong kann Vorkommen von Aktivitäten allgemein und speziell Vorkommen von Aktivitäten in einer bestimmten Reihenfolge verifizieren. Daneben kann auch ein Prozessmodell strukturell verifiziert werden, so z.B. ob in jeder Spur der Endknoten erreicht wird. Bei diesem Ansatz wird das Modell und die Spezifikation jeweils in der Prozessalgebra CSP (Communicating Sequential Processes) dargestellt. Folgendermaßen werden die zu verifizierenden Prozessmodelle in CSP dargestellt: Jeder Knoten wird als eigenständiger Prozess und jede Kontrollflusskante wird durch das Senden bzw. Empfangen von Ereignissen modelliert. Damit ist bei einem Knoten mit einer eingehenden Kontrollflusskante das Empfangen eines Ereignisses der Trigger, der den Knoten aktiviert. Analog verhält es sich mit AND-Join- und OR-Join-Knoten. Bei einem Knoten entspricht das Senden eines bzw. mehrerer Ereignisse bei einem AND-Split wiederum dem Beenden dieses Knoten.

Bei Karamanolis et al. [KGMW99] können Sicherheits- und Lebendigkeitseigenschaften, wie z. B. ob Deadlocks auftreten können und ob jede Aktivität ausführbar ist, aber auch andere Eigenschaften, wie z.B. ob Aktivitäten in einer speziellen Reihenfolge ausgeführt werden, verifiziert werden. Hier wird dem Problem der Zustandsexplosion auch durch Abstraktion begegnet. Der dem Verfahren zugrunde liegende Model Checker nutzt nämlich die hierarchische Struktur von Systemen aus, indem zuerst jedes Subsystem einzeln verifiziert wird und nachher die einzelnen Subsysteme zusammen verifiziert werden. Dabei werden dann die internen Details der Subsysteme ausgeblendet, wodurch weniger Zustände zustande kommen. Um diesen Vorteil des Model Checkers auszunutzen, wird bei dem Verfahren auch ein Prozessmetamodell verwendet, mit dem zusammengesetzte Aktivitäten modelliert werden können.

Bei den hier vorgestellten Model Checking-Ansätzen mussten die Prozessmodelle immer in einem bestimmten Prozessmetamodell vorliegen und es konnte nur mit einem bestimmten Model Checker das Prozessmodell verifiziert werden. In dem Ansatz von Kloos et al. [FFK05] wird ein Framework vorgestellt, um verschiedene Prozessmetamodelle und Mo-

del Checker verwenden zu können. Um dies zu erreichen, werden die Prozessmodelle nicht direkt in die Eingabesprache des Model Checkers transformiert, sondern in eine Zwischensprache das sog. Common Formal Model und danach erst in die Eingabesprache des Model Checkers. Damit das Framework ein neues Prozessmetamodell oder einen neuen Model Checker unterstützen kann, muss also jeweils nur ein Compiler, der eine Spezifikation des Prozessmetamodells in eine Spezifikation des Common Formal Model transformiert bzw. ein Compiler, der eine Spezifikation des Common Formal Model in eine Spezifikation der Eingabesprache des Model Checkers transformiert, in das Framework integriert werden.

Der Vorteil bei den Model Checking-Verfahren ist, dass eventuelle Fehler mit der Angabe des Gegenbeispiels gleich lokalisiert werden. Dem stehen zwei Nachteile gegenüber. Zum einen gibt es wie oben angesprochen das Problem der Zustandsexplosion. Allerdings gibt es zu dessen Vermeidung bereits Mittel, wie z.B. Abstraktion. Zum anderen gibt es wie bei den logikbasierten Verfahren keinen Ansatz, der für Adaptivität optimiert ist. Das bedeutet, dass auch bei einer kleinen Prozessänderung stets das gesamte Prozessmodell mit allen Integritätsregeln verifiziert werden muss.

4.4.3 Graphbasierte Verifikation

Bei der graphbasierten Verifikation werden die Integritätsregeln mit Graphalgorithmen verifiziert. Bei dem Ansatz [LRD06a] können bis jetzt einfache Abhängigkeits-Integritätsregeln und Exklusions-Integritätsregeln verifiziert werden. Bei den Abhängigkeits-Integritätsregeln hängt eine Aktivität *source* von einer Aktivität *target* ab und bei den Exklusions-Integritätsregeln dürfen die Aktivität *source* und *target* nicht zusammen im Prozess vorkommen. Der Ansatz beruht darauf, dass der Kontrollflußgraph des Prozesses traversiert wird [Zho06] und dabei sämtliche Aktivitäten, die in den Integritätsregeln vorkommen, gesucht werden. Nachdem der Prozess vollständig traversiert wurde, werden die Integritätsregeln entsprechend dem Vorkommen bzw. Nichtvorkommen ihrer Aktivitäten ausgewertet. Das Besondere an dem Ansatz ist, dass er für die Adaptivität von Prozessen optimiert ist. Dabei wird angenommen, dass der Prozess vor der Änderung semantisch korrekt war. Nach der Änderung müssen deswegen nicht mehr alle für diesen Prozess geltenden Integritätsregeln verifiziert werden, sondern nur die Integritätsregeln, die überhaupt durch die Änderung verletzt sein können. Generell werden bei dem Ansatz drei unterschiedliche Verifikationsszenarien unterschieden:

Verifikation eines neu erstelltes Prozessschema Bei einem neu erstellten Prozess-Schema müssen nur die Integritätsregeln verifiziert werden, bei denen die *source*-Aktivität im Schema vorkommt. Eine Abhängigkeits-Integritätsregel kann nämlich nur verletzt sein, wenn die *source*-Aktivität auch im Schema vorhanden ist. Eine Exklusions-Integritätsregel kann auch nur verletzt sein, wenn sowohl die *source*- als auch die *target*-Aktivität im Schema auftauchen.

Verifikation einer ad-hoc-modifizierten Prozessinstanz Wie oben beschrieben wird bei jeder Prozessänderung angenommen, dass der Prozess zuvor semantisch korrekt

war. Deshalb müssen bei einer Ad-hoc-Änderung auch nur die Integritätsregeln verifiziert werden, die durch diese Ad-hoc-Änderung verletzt sein können. Dabei werden drei mögliche Ad-hoc-Änderungsoperationen betrachtet, bei denen jeweils nur bestimmte Integritätsregeln verifiziert werden müssen:

- Einfügen einer Aktivität
- Löschen einer Aktivität
- Verschieben einer Aktivität

Verifikation der Migration eines geänderten Prozessschemas Die Migration eines geänderten Prozess-Schema auf noch nicht ad-hoc-modifizierte Prozess-Instanzen muss natürlich nicht verifiziert werden, da die Schema-Änderung bereits semantisch verifiziert wurde und damit die Instanzen, die dieses Schema referenzieren, ebenfalls korrekt sind. Bei ad-hoc-modifizierten Prozess-Instanzen muss die Migration auch nicht verifiziert werden, wenn die Instanz- und Schema-Änderungen gleich sind, denn dabei wird die als semantisch korrekt angenommene Prozess-Instanz nicht verändert. Wenn die Instanz-Änderungen eine Teilmenge der Schema-Änderungen darstellen, muss die Migration ebenfalls nicht semantisch verifiziert werden. Dieser Fall entspricht der Migration eines geänderten Prozess-Schema auf eine noch nicht ad-hoc-modifizierte Prozess-Instanz und deshalb ist die Migration auch semantisch korrekt. Auch wenn die Schema-Änderungen eine Teilmenge der Instanz-Änderungen darstellen, muss die Migration ebenfalls nicht semantisch verifiziert werden, da im Grunde an der Prozess-Instanz keine Änderung durchgeführt wird und diese damit semantisch korrekt bleibt. Die Migration muss nur semantisch verifiziert werden, wenn die Schema-Änderungen und die Instanz-Änderungen teilweise oder komplett disjunkt sind. In diesem Fall gibt es entweder keine gemeinsamen Änderungen auf Schema- und Instanz-Ebene, Änderungen auf Schema-Ebene, die nicht auf Instanz-Ebene vorkommen oder Änderungen auf Instanz-Ebene, die auf Schema-Ebene nicht vorkommen. Allerdings müssen die zu migrierenden Instanzen dabei auch nur gegen die Integritätsregeln verifiziert werden, die durch das Zusammenspiel der Schema- und Instanz-Änderungen überhaupt verletzt sein können.

4.5 Zusammenfassung und Diskussion

Der erste Teil des Kapitels befasste sich mit der regelbasierten Prozessbeschreibung. Dazu werden in den hier erwähnten Ansätzen Geschäftsregeln in ECA-Form verwendet. Die Ansätze unterscheiden sich nur darin, wie ein Prozessschema mit Geschäftsregeln modelliert wird. Allerdings sind beide Verfahren praktisch nicht einsetzbar. Zum einen sind die durch beide Verfahren erzeugten Prozessmodelle unübersichtlich und können damit nur schwer geändert werden. Zum anderen gibt es keine formalen Methoden zur Analyse dieser Prozessmodelle.

Im zweiten Teil des Kapitels wurde kurz erläutert, wie Geschäftsregeln mit Business Rules Engines in PMS verwendet werden können. Es kann z.B. für einen bestimmten XOR-Split in der BRE eine Geschäftsregel hinterlegt werden. Wenn der Kontrollfluß dann diesen XOR-Split-Knoten erreicht, entscheidet das BRE aufgrund der vom PMS übertragenen Daten, welcher Zweig auszuführten ist. Hier wurde auch darauf eingegangen, wie die meisten Business Rules Engines implementiert sind. Mit dem Rete-Algorithmus werden auf effiziente Weise die Geschäftsregeln bestimmt, deren Bedingungsteile erfüllt sind und die damit neu ausgewertet werden müssen. Der Rete-Algorithmus beruht darauf, dass zum einen bei einer Änderung des Working Memory nur die davon betroffenen Bedingungsteile neu ausgewertet werden und zum anderen die Bedingungen, die in mehreren Bedingungsteilen vorkommen, nur einmal ausgewertet werden.

Im dritten Teil des Kapitels ging es um die Prozessadaption mit Geschäftsregeln. Bei den hier vorgestellten Ansätzen zur Einschränkung der Adaptionsmöglichkeiten von Prozessen ist man davon ausgegangen, dass zur Modellierungszeit der Prozess nicht fertig spezifiziert werden kann. Deshalb wird dem Benutzer zur Laufzeit die Möglichkeit gegeben, die Prozessinstanz zu adaptieren. Damit die geänderten Prozesse auch noch semantisch korrekt sind, werden die Adaptionsmöglichkeiten durch Integritätsregeln eingeschränkt. Der Ansatz von Müller et al. befasste sich zwar auch mit der Adaption von laufenden Prozessinstanzen, allerdings erfolgte hier die Adaption automatisch mit Hilfe von Geschäftsregeln in ECA-Form. Wenn ein bestimmtes Ereignis eintritt und gleichzeitig eine bestimmte Bedingung erfüllt ist, wird die Prozessinstanz entsprechend dem Aktionsteil der ECA-Regel geändert.

Der letzte Teil des Kapitels behandelte die semantische Verifikation von Prozessen. Hier wurden drei verschiedene Arten von Verifikationsansätzen vorgestellt. Bei den logikbasierten Verfahren wird der Prozess und die Eigenschaft, gegen die der Prozess verifiziert werden soll, jeweils als logische Formel dargestellt. Wenn sich die beiden Formeln verknüpfen lassen, ohne dass ein logischer Widerspruch auftritt, folgt daraus, dass der Prozess die getestete Eigenschaft erfüllt. Bei den modellbasierten Verfahren zur semantischen Prozessverifikation wird der Prozess meistens als Zustandsübergangssystem und die zu verifizierende Eigenschaft als temporallogische Formel, mit der Pfade und Zustände des Zustandsübergangssystems beschrieben werden können, dargestellt. Beim Model Checking wird dann geprüft, ob das Modell die Spezifikation erfüllt, indem das Zustandsübergangssystem nach Zuständen durchsucht wird, die die Anforderungen erfüllen. Im Fehlerfall wird dann ein Gegenbeispiel ausgegeben. Der Nachteil dieser logik- und modellbasierten Verfahren ist, dass sie nicht für die Adaptivität von Prozessen optimiert sind. Das graphbasierte Verfahren von Ly et al. hingegen ist bereits für Adaptivität optimiert. Nach einer Prozessänderung werden dabei nicht mehr alle definierten Integritätsregeln verifiziert, sondern nur die, die verletzt sein könnten.

Kapitel 5

Integration von Integritätsregeln in ein Prozess-Management-System

Nachdem in Kap. 2 und 3 die Grundlagen zum verwendeten Prozess-Meta-Modell bzw. zu Geschäftsregeln gelegt wurden und in Kap. 4 die Literatur zur Integration von Geschäftsregeln in Prozess-Management-Systemen vorgestellt wurde, beginnt mit diesem Kapitel der zweite Teil der Diplomarbeit. Hier wird die eigene konzeptuelle Arbeit zur semantischen Prozessverifikation in ADEPT WSM-Netzen vorgestellt, wobei die semantische Information in Form von Integritätsregeln vorliegt, einer bestimmten Kategorie von Geschäftsregeln (s. Unterkap 3.2.2).

ADEPT WSM-Netze können bereits syntaktisch verifiziert werden [Rei00,Rin04]. Eine syntaktische Verifikation beinhaltet u.a., dass eine Überprüfung auf mögliche Verklemmungen erfolgt und eine Überprüfung, dass die obligaten Eingabeparameter versorgt sind. Allerdings stellt ein Prozess meistens einen Ausschnitt der Realität dar, der gewissen Regeln unterliegt. Folglich muss ein Prozess zusätzlich zu den syntaktischen Regeln noch zu den Regeln auf semantischer Ebene konsistent sein. Deshalb muss neben der syntaktischen Verifikation noch eine semantische Verifikation der Prozesse erfolgen. Hao Zhou hat in seiner Diplomarbeit „Effiziente Überprüfung semantischer Korrektheit in adaptiven Prozess-Management-Systemen“ [Zho06] bereits einen ersten Schritt bei der semantischen Verifikation getan, indem er Prozesse auf einfache semantische Integritätsregeln verifiziert hat. Seine Arbeit soll in der vorliegenden Diplomarbeit fortgesetzt werden, indem Regeln über erlaubte Kombinationen von Aktivitäten verifiziert werden. Bei einem klinischen Behandlungsprozess soll z.B. die Integritätsregel verifiziert werden können, dass, wenn einem Patienten Aspirin und nicht Marcumar verabreicht wird, ebenfalls eine Blutwert- oder eine Leberuntersuchung durchgeführt werden muss und kein Spalt verabreicht werden darf, wobei die Blutwertuntersuchung vor der Verabreichung von Aspirin und die Leberuntersuchung nach der Verabreichung von Aspirin erfolgen muss (s. Abbildung 5.1). Um solche Regeln auf Prozesse anwenden zu können, wird in diesem Kapitel der Aufbau und die Semantik von Integritätsregeln eingeführt, deren Verifikation im Anschluss erfolgt. Die Verifikation sollte hierbei möglichst effizient ablaufen.

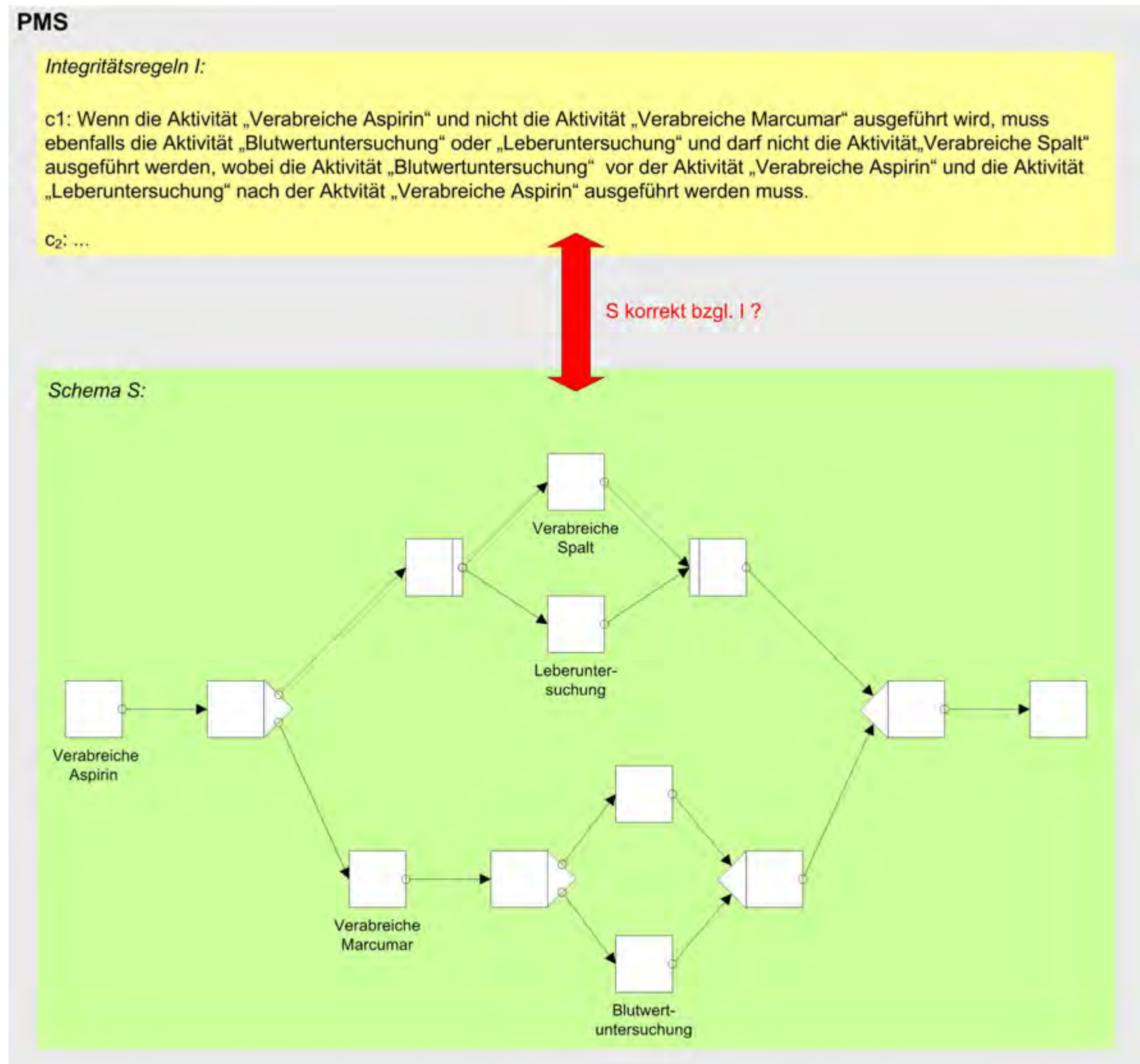


Abbildung 5.1: Semantische Prozessverifikation mit einer Beispiel-Integritätsregel

Dieser zweite Teil der vorliegenden Arbeit ist in folgende Kapitel gegliedert:

In diesem Kapitel wird ein Überblick über die zu verifizierenden Integritätsregeln gegeben und motiviert, warum wir uns im Rahmen der Diplomarbeit für die Verifikation dieser Integritätsregeln entschieden haben. Kapitel 6 behandelt die Verifikation dieser Integritätsregeln für ADEPT-Graphen. Dabei werden zuerst strukturelle Kriterien hergeleitet, auf die die komplexen Integritätsregeln abgebildet werden sollen. Für diese strukturellen Kriterien werden im Anschluss Verifikationsalgorithmen bestimmt. Kap. 7 behandelt Effizienzaspekte der Verifikation, z.B. wie die Menge der zu verifizierenden Integritätsregeln eingeschränkt werden kann oder wie die Verifikationsalgorithmen für die strukturellen Kriterien durch Erweiterungen des Prozess-Meta-Modells optimiert werden können.

5.1 Aufbau der zu verifizierenden Integritätsregeln

In diesem Unterkapitel wollen wir die Integritätsregeln vorstellen, für die im Folgenden Verifikationsalgorithmen entwickelt werden sollen. Dabei handelt es sich um Integritätsregeln, die grundlegende Aussagen über die Ausführung von Aktivitäten zulassen, wie im eingangs erwähnten Beispiel. Mit den Regeln, die im Rahmen dieser Arbeit betrachtet werden, wird ein Grundstein für die semantische Verifikation gelegt, da die Regeln die Basis für mögliche Erweiterungen der Ausdrucksmächtigkeit (z.B. hinsichtlich der Gültigkeit von Integritätsregeln) bilden. So lassen sich leicht Reihenfolgen zwischen den Literalen einer Integritätsregel definieren. Die Gültigkeit der Integritätsregeln kann auch vom Vorliegen bestimmter Kontextdaten abhängig gemacht werden. Es kann z.B. bei einem medizinischen Behandlungsprozess Integritätsregeln definiert werden, die nur für Patienten ab einem Alter von 65 Jahren gelten sollen. Ebenso kann die Gültigkeit einer Integritätsregel von einer bestimmten Zeitspanne abhängig gemacht werden. So könnte z.B. eine auf eine Woche begrenzte Integritätsregel definiert werden, dass nach dem Sinken des Blutdrucks ein bestimmtes Medikament abzusetzen ist. Eine andere mögliche Erweiterung dieses Kerns von Integritätsregeln wäre anzugeben, wie strikt die Integritätsregeln befolgt werden müssen. So müssen manche Integritätsregeln strikt eingehalten werden, wie z.B. folgende, die besagt, dass nach einem medizinischen Notfall dem Patienten ein bestimmtes Medikament verabreicht werden muss. Andere Integritätsregeln hingegen haben eher den Charakter von Empfehlungen [LRD06b]. Die Integritätsregeln sollen zwar eingehalten werden, müssen aber nicht. Diese Kern-Integritätsregeln können ggf. auch um eine Priorisierungsmöglichkeit erweitert werden. Damit könnten leicht Konflikte (widersprüchliche Regeln) zwischen Integritätsregeln aufgelöst werden oder Integritätsregeln überschrieben werden, so dass für Spezialfälle besondere Integritätsregeln definiert werden können. Abbildung 5.2 veran-

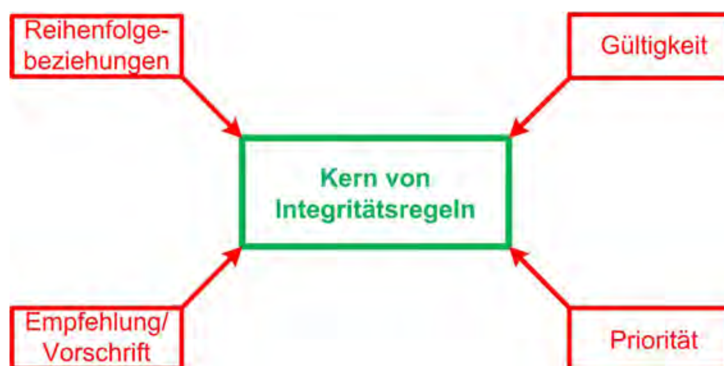


Abbildung 5.2: Erweiterungsmöglichkeiten des hier vorgestellten Kerns an Integritätsregeln

schaulicht den beschriebenen Sachverhalt, dass dem Benutzer mit den hier behandelten Integritätsregeln ein Grundgerüst zum Definieren von Integritätsregeln gegeben wird, das leicht um zusätzliche Features wie z.B. Gültigkeit oder Reihenfolgebeziehungen erweitert werden kann.

5.1.1 Struktur der Integritätsregeln

Die von uns betrachteten Integritätsregeln setzen sich aus einem Bedingungs-, einem Folgeteil und optionalen Reihenfolgebeziehungen zusammen, wobei der Bedingungsteil auch leer sein kann.

Bedingungs- und Folgeteil bestehen jeweils wieder aus Literalen und Operatoren, die die Literale miteinander verknüpfen. Zusätzlich können für die Literale im Bedingungs- und im Folgeteil Reihenfolgebeziehungen definiert werden.

Literale

Ein Literal A drückt aus, dass eine Aktivität A in einer Ausführungsspur vorkommt. Dementsprechend drückt $\neg A$ aus, dass A nicht in der Ausführungsspur vorkommt. Formaler ausgedrückt: Sei $\sigma = <a_1, a_2, \ldots, a_n>$ eine Ausführungsspur, wobei die Menge der a_i die ausgeführten Aktivitäten darstellt. A evaluiert zu true, wenn es ein a_i in σ gibt $(1 \leq i \leq n)$ mit $a_i = A$. Analog evaluiert $\neg A$ dann zu true, wenn es kein a_i in σ gibt $(1 \leq i \leq n)$ mit $a_i = A$.

Operatoren

Die Literale können mit dem Konjunktionsoperator $\wedge$ *(und)* und dem Disjunktionsoperator $\vee$ *(oder)* miteinander verbunden werden. $A \wedge B$ drückt aus, dass die Aktivitäten A und B zusammen in einer Ausführungsspur vorkommen und $A \vee B$, dass A oder B in einer Ausführungsspur vorkommen. Formal ausgedrückt: Sei $\sigma = <a_1, a_2, \ldots, a_n>$ eine Ausführungsspur, wobei die Menge der a_i die ausgeführten Aktivitäten darstellt. $A \wedge B$ evaluiert dann zu true, wenn es ein a_i und ein a_j in σ gibt $(1 \leq i,j \leq n)$ mit $a_i = A$ und $a_j = B$. $A \vee B$ evaluiert dann zu true, wenn es ein a_i in σ gibt $(1 \leq i \leq n)$ mit $a_i = A$ oder $a_i = B$.

Neben den Aussagen über das Vorhandensein bzw. Fehlen von Aktivitäten kann wie bereits erwähnt auch eine bestimmte Reihenfolge zwischen den Aktivitäten gefordert werden. Dazu werden jeweils zwei Aktivitäten mit den Vergleichsoperatoren $<$ und $>$ miteinander verglichen. $A < B$ bedeutet dabei, dass die Aktivität A vor B ausgeführt werden muss und $A > B$ bedeutet, dass die Aktivität A nach B ausgeführt werden muss. Formal ausgedrückt: Sei $\sigma = <a_1, a_2, \ldots, a_n>$ eine Ausführungsspur, wobei die Menge der a_i die ausgeführten Aktivitäten darstellt. $A < B$ evaluiert dann zu true, wenn es ein a_i und ein a_j in σ gibt $(1 \leq i < j \leq n)$ mit $a_i = A$ und $a_j = B$. $A > B$ evaluiert dann zu true, wenn es ein a_i und ein a_j in σ gibt $(1 \leq i < j \leq n)$ mit $a_i = B$ und $a_j = A$.

Eine Integritätsregel c ist über einer Ausführungsspur erfüllt, falls der Bedingungsteil mit den optionalen Reihenfolgebeziehungen für dessen Literale nicht auf die Ausführungsspur zutrifft oder der Bedingungsteil und ebenfalls der Folgeteil jeweils mit den optionalen Reihenfolgebeziehungen für die jeweiligen Literale für die Ausführungsspur zutreffen. Wenn der Bedingungsteil leer ist, gilt er für jede beliebige Ausführungsspur als erfüllt.

Dementsprechend ist eine Integritätsregel c über einem Prozess p erfüllt, wenn sie für alle Ausführungsspuren von p erfüllt ist.

5.1.2 Form des Bedingungs- und des Folgeteils

Um die Zahl der zu verifizierenden Fälle einzuschränken und damit die Verifikation der Integritätsregeln effizienter zu gestalten, wurde die Form der Kombinationen von Aktivitätensymbole im Bedingungs- und im Folgeteil eingeschränkt. So darf der Bedingungsteil nur als Konjunktion von (negierten) Aktivitätensymbolen und der Folgeteil nur als Konjunktive Normalform (KNF) von (negierten) Aktivitätensymbolen angegeben werden. Die Ausdrucksmächtigkeit des Folgeteils wird dadurch nicht eingeschränkt, da sich jede aussagenlogische Formel auch als KNF darstellen lässt. Die Ausdrucksmächtigkeit des Bedingungsteils wird damit ebenfalls nicht eingeschränkt, da sich jede beliebige Kombination von Aktivitätensymbolen im Bedingungsteil über die Disjunktive Normalform (DNF) in eine Konjunktion umwandeln lässt. In der folgenden Integritätsregel $(A \vee B) \wedge (C \vee D) \rightarrow conclusion$ lässt sich der Bedingungsteil zuerst durch Ausmultiplizieren in eine DNF umwandeln: $(A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D) \rightarrow conclusion$. Die Integritätsregel bedeutet, dass wenn A und C oder B und C oder ... ausgeführt werden, jeweils auch die Kombination im Folgeteil ausgeführt werden muss. Die Integritätsregel lässt sich also in mehrere Teilregeln aufspalten, indem jeder Disjunktionsterm den Bedingungsteil einer neuen Integritätsregel ergibt und der Folgeteil der alten Integritätsregel den Folgeteil der neuen Integritätsregel ergibt. Somit ergeben sich in unserem Beispiel folgende Integritätsregeln:

- $A \wedge C \rightarrow conclusion$
- $A \wedge D \rightarrow conclusion$
- $B \wedge C \rightarrow conclusion$
- $B \wedge D \rightarrow conclusion$

5.1.3 Reihenfolgebeziehungen

Reihenfolgebeziehungen dürfen nicht zwischen beliebigen Literalen definiert werden. Wenn z.B. Reihenfolgebeziehungen zwischen Literalen definiert werden, die zur selben Disjunktion gehören, würde sich folgender Widerspruch ergeben. Die Disjunktion könnte nur erfüllt sein, wenn alle Aktivitäten der Disjunktion, zwischen denen Reihenfolgebeziehungen existieren, im Prozess in der korrekten Reihenfolge vorkommen würden. Allerdings widerspricht dies der Definition einer Disjunktion, wonach nur eine Aktivität im Prozess vorkommen muss, damit die Disjunktion erfüllt ist. Deshalb dürfen nur Reihenfolgebeziehungen zwischen positiven Literalen einer Integritätsregel definiert werden, die nicht zur selben Disjunktion gehören.

Dabei gibt es zwei verschiedene Fälle, in denen eine Reihenfolgebeziehung zwischen zwei Literalen erfüllt sein kann. Zum einen ist eine Reihenfolgebeziehung erfüllt, wenn die

beiden Literale zusammen ausgeführt werden können und in der angegebenen Reihenfolge vorkommen. Das bedeutet wiederum, dass eine Reihenfolgebeziehung als verletzt gilt, wenn die Reihenfolge zwischen zwei Literalen unklar ist, weil die entsprechenden Aktivitäten auf verschiedenen Zweigen eines AND-Blocks liegen und keine Sync-Kante die Reihenfolge zwischen ihnen festlegt. Zum anderen ist eine Reihenfolgebeziehung erfüllt, wenn die Literale zwischen denen die Reihenfolgebeziehung besteht, nicht zusammen ausgeführt werden können. Das kann zwei Gründe haben. Ein Grund ist, dass eine der beiden Aktivitäten nicht im Prozess vorkommt. Der andere Grund ist, dass diese auf verschiedenen Zweigen eines XOR-Blocks liegen.

Die eingangs erwähnte Integritätsregel, die besagt, dass in einem klinischen Behandlungsprozess, wenn einem Patienten Aspirin und kein Marcumar verabreicht wird, eine Blutwert- oder eine Leberuntersuchung durchgeführt werden muss und kein Spalt verabreicht werden darf, wobei Aspirin nach der Blutwertuntersuchung und vor der Leberuntersuchung verabreicht werden muss, kann wie folgt ausgedrückt werden:
$A \wedge \neg M \rightarrow (B \vee L) \wedge \neg S,\ [B < A,\ L > A]$

Nachdem in diesem Kapitel die Integritätsregeln eingeführt wurden, die verifiziert werden sollen, wird im Folgenden deren effiziente graphbasierte Verifikation in ADEPT WSM-Netzen betrachtet.

Kapitel 6

Verifikation

Wie in Kapitel 5 erläutert, ist das Thema der konzeptuellen Arbeit die semantische graphbasierte Prozessverifikation mit den in Unterkapitel 5.1 definierten Integritätsregeln. Mit diesen können komplexe Aussagen darüber gemacht werden, in welchen Kombinationen Aktivitäten ausgeführt werden dürfen. Die Verifikation dieser Integritätsregeln in einem ADEPT-Graphen ist Gegenstand dieses Kapitels (s. Abbildung 6.1). Um diese Integri-

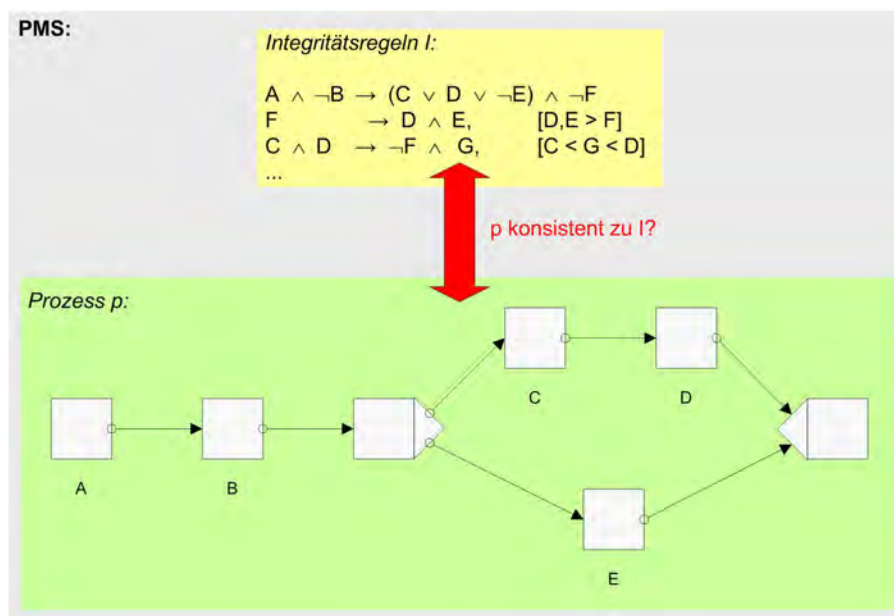


Abbildung 6.1: Semantischen Prozessverifikation mit Integritätsregeln der Kategorie, wie in Kap. 5 definiert

tätsregeln zu verifizieren wird die Idee verfolgt, zunächst die Integritätsregeln auf sog. strukturelle Kriterien abzubilden. Diese definieren, wie ein ADEPT-Graph aussehen muss, damit er eine bestimmte elementare Beziehung zwischen Aktivitäten einhält. Ziel ist es, eine automatische Abbildungsvorschrift zu finden, mit der automatisch jede beliebige Integritätsregel auf die entsprechenden strukturellen Kriterien abgebildet und damit verifiziert werden kann. Der hierfür entwickelte Verifikationsalgorithmus sollte möglichst effizient ab-

laufen. Das hat zur Folge, dass die Verifikation einer einzelnen Integritätsregel so effizient wie möglich sein soll, zum anderen sollten nur solche Integritätsregeln verifiziert werden, die auch verletzt sein können.

6.1 Lösungsansatz

Die Verifikation der Integritätsregeln soll dadurch erfolgen, dass diese zuerst auf strukturelle Kriterien abgebildet werden und diese strukturellen Kriterien dann verifiziert werden.

Bei der Überprüfung, ob eine Integritätsregel für einen Prozess p erfüllt ist, müssen zwei Fälle unterschieden werden: Der eine Fall liegt vor, wenn in keiner Ausführungsspur von p der Bedingungsteil zutrifft. Der andere Fall liegt vor, wenn für mindestens eine Ausführungsspur der Bedingungsteil zutrifft. Deshalb sollen zuerst strukturelle Kriterien hergeleitet werden, um anhand eines ADEPT-Graphen zu verifizieren, ob der Bedingungsteil in keiner Ausführungsspur zutrifft. Wenn dies der Fall ist, gilt die Integritätsregel als erfüllt. Ansonsten muss geprüft werden, ob für jede Ausführungsspur, in der der Bedingungsteil zutrifft auch der Folgeteil zutrifft. Dieser Fall soll ebenfalls mit strukturellen Kriterien überprüft werden, was zur Folge hat, dass dafür ebenfalls strukturelle Kriterien hergeleitet werden. Am Ende soll automatisch mit Hilfe einer automatischen Abbildungsvorschrift jede beliebige Integritätsregel auf die entsprechenden strukturellen Kriterien abgebildet werden können, um mit ihnen verifizieren zu können, ob die Integritätsregel erfüllt ist.

Vor der Verifikation beider Fälle werden jeweils zuerst die Reihenfolgebeziehungen der Aktivitäten in den strukturellen Kriterien verifiziert. Falls Reihenfolgebeziehungen von Aktivitäten dabei nicht erfüllt sind, werden daraufhin die strukturellen Kriterien entsprechend geändert oder wenn feststeht, dass die strukturellen Kriterien nicht mehr erfüllt sein können, werden diese gar nicht mehr ausgewertet.

6.2 Herleitung der strukturellen Kriterien

Im Folgenden werden zunächst sämtliche strukturellen Kriterien hergeleitet, um beliebige der in Kapitel 5 definierten Integritätsregeln auf diese abzubilden und mit ihnen zu verifizieren. Dabei werden Reihenfolgebeziehungen vorerst nicht betrachtet. Wie in Unterkapitel 6.1 erläutert wurde, werden bei der Verifikation zwei verschiedene Fälle unterschieden, denn eine Integritätsregel kann zum einen dadurch erfüllt sein, dass ihr Bedingungsteil in keiner Ausführungsspur erfüllt ist und zum anderen dadurch, dass ihr Folgeteil in jeder Ausführungsspur erfüllt ist, in der ihr Bedingungsteil erfüllt ist. Deshalb werden zuerst anhand von Beispielen strukturelle Kriterien hergeleitet, um verifizieren zu können, ob der Bedingungsteil in keiner Ausführungsspur erfüllt ist. Im Anschluss daran werden strukturelle Kriterien gesucht, um zu verifizieren, ob der Folgeteil in jeder Ausführungsspur erfüllt ist, in der der Bedingungsteil erfüllt ist. Die Verifikation der einzelnen strukturellen Kriterien wird in Kapitel 6.3 behandelt.

6.2.1 Strukturelle Kriterien für Bedingungsteile

Hier wird das Ziel verfolgt, strukturelle Kriterien zu finden, anhand derer ausgeschlossen werden kann, dass der Bedingungsteil in einer Ausführungsspur erfüllt ist. In diesem Fall gilt die Integritätsregel als erfüllt und es muss nicht mehr geprüft werden, ob der Folgeteil in jeder Ausführungsspur erfüllt ist, in der der Bedingungsteil erfüllt ist. Damit automatisch bei jeder beliebigen Integritätsregel geprüft werden kann, ob der Bedingungsteil in keiner Ausführungsspur erfüllt ist, wird eine automatische Abbildungsvorschrift angegeben, mit der Integritätsregeln auf strukturelle Kriterien abgebildet werden können.

Wann gilt in keiner Ausführungsspur der Bedingungsteil der Integritätsregel $c_1 = A \wedge B \rightarrow C$?

Der Bedingungsteil $A \wedge B$ der Integritätsregel ist in einer Ausführungsspur t erfüllt, wenn in t die beiden Aktivitäten A und B zusammen ausgeführt werden. Folglich ist der Bedingungsteil in keiner Ausführungsspur erfüllt, wenn in jeder Ausführungsspur A und B nicht zusammen vorkommen. Das ist der Fall, wenn mindestens eine der Aktivitäten nicht im Prozess vorkommt oder wenn beide Aktivitäten im Prozess vorkommen, jedoch auf verschiedenen Zweigen eines XOR-Blocks liegen. Demnach ist bei dem Prozess in Abbildung 6.2 der Bedingungsteil von c_1 in keiner Ausführungsspur erfüllt, da in keiner Ausführungsspur A und B zusammen vorkommen.

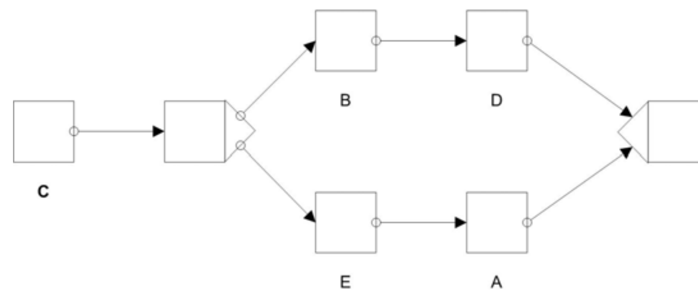


Abbildung 6.2: Die Aktivitäten A, B und C können nicht alle zusammen ausgeführt werden, da A und B auf verschiedenen Zweigen eines XOR-Blocks liegen.

Allgemein lässt sich folgendes schließen: Damit in einem ADEPT-Graphen zwei Aktivitäten S und T nicht zusammen ausgeführt werden können, darf entweder eine der Aktivitäten S oder T nicht im ADEPT-Graphen vorkommen oder wenn beide Aktivitäten vorkommen, müssen beide auf verschiedenen Zweigen eines XOR-Blocks liegen. Für ein solches Aussehen eines ADEPT-Graphen definieren wir das strukturelle Kriterium *S schließt T aus* ($S \not< \bullet \not> T$).

Demnach ist hier der Bedingungsteil $A \wedge B$ in keiner Ausführungsspur erfüllt, wenn das strukturelle Kriterium $A \not< \bullet \not> B$ erfüllt ist.

Wann gilt in keiner Ausführungsspur der Bedingungsteil der Integritätsregel $\boldsymbol{c_2 = A \wedge \neg B \rightarrow C}$?

Bevor die Integritätsregeln auf strukturelle Kriterien abgebildet werden, werden die negierten Literale der Integritätsregeln zuerst vom Bedingungs- in den Folgeteil bzw. vom Folge- in den Bedingungsteil gebracht. So bestehen die Integritätsregeln nur aus positiven Literalen und lassen sich dadurch einheitlich behandeln und auf strukturelle Kriterien abbilden. In diesem Fall wird das negierte Literal $\neg B$ als positives Literal B auf die Seite des Folgeteils gebracht und mit $\vee$ verknüpft. Nach dieser Umformung sieht die Integritätsregel folgendermaßen aus: $c_2' = A \rightarrow B \vee C$. Damit die ursprüngliche Integritätsregel c_2 erfüllt ist, muss in jeder Ausführungsspur, in der A vorkommt und B nicht vorkommt, C vorkommen. Die umgeformte Integritätsregel c_2' ist bereits in einer Ausführungsspur erfüllt, wenn in dieser A vorkommt. Allerdings muss C nur in einer Ausführungsspur vorkommen, wenn in dieser B nicht vorkommt. Demnach ist die umgeformte Integritätsregel c_2' zur ursprünglichen Integritätsregel c_2 logisch und strukturell äquivalent und die Umformung korrekt. Nach dieser Umformung ist trivialerweise der Bedingungsteil in keiner Ausführungsspur erfüllt, wenn A nicht im Prozess vorkommt.

Wann gilt in keiner Ausführungsspur der Bedingungsteil der Integritätsregel $\boldsymbol{c_3 = \neg A \wedge \neg B \rightarrow C}$?

Analog zur Umformung der Integritätsregel c_2 lassen sich bei c_3 die negierten Literale $\neg A$ und $\neg B$ auf die andere Seite bringen. Damit befindet sich in diesem Fall kein Literal mehr im Bedingungsteil und dieser ist somit leer. Allerdings kann laut 5.1.2 der Bedingungsteil auch leer sein, was bedeutet, dass der Bedingungsteil in jeder Ausführungsspur als erfüllt gilt. Folglich muss bei einer Integritätsregel, deren Bedingungsteil nur aus negierten Literalen besteht, geprüft werden, ob der Folgeteil der umgeformten Integritätsregel in jeder Ausführungsspur erfüllt ist. Nur in diesem Fall ist die umgeformte und damit auch die ursprüngliche Integritätsregel erfüllt.

Wann gilt in keiner Ausführungsspur der Bedingungsteil der Integritätsregel $\boldsymbol{c_4 = A \wedge B \wedge C \wedge \neg D \wedge \neg E \rightarrow F}$?

Wie bei c_2 und c_3 lässt sich die Integritätsregel auch hier wieder umformen, indem die negierten Literale des Bedingungsteils in den Folgeteil gebracht werden. Nach dieser Umformung sieht die Integritätsregel folgendermaßen aus: $c_4' = A \wedge B \wedge C \rightarrow F \vee D \vee E$. Der Bedingungsteil dieser umgeformten Integritätsregel ist in keiner Ausführungsspur erfüllt, wenn in jeder Ausführungsspur mindestens eine der Aktivitäten A, B und C nicht vorkommt. Dies ist der Fall, wenn sich zwei der Aktivitäten gegenseitig ausschließen. Für den Prozess in Abbildung 6.3 gilt, dass sich weder A und B noch, dass sich B und C gegenseitig ausschließen, d.h. die strukturellen Kriterien $A \not< \bullet \not> B$ und $B \not< \bullet \not> C$ sind nicht erfüllt. Trotzdem lassen sich nicht alle drei Aktivitäten gemeinsam ausführen, da sich A und C gegenseitig ausschließen $(A \not< \bullet \not> C)$. Somit ist das strukturelle Kriterium $S \not< \bullet \not> T$

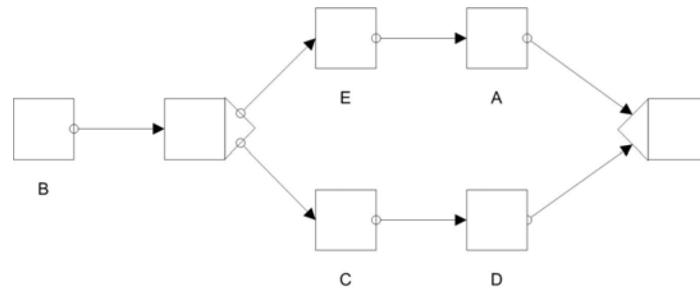


Abbildung 6.3: $A \not< \bullet \not> C$ ist erfüllt, jedoch $A \not< \bullet \not> B$ und $B \not< \bullet \not> C$ jeweils nicht.

nicht transitiv. Um demnach herauszufinden, ob sich zwei Aktivitäten aus einer Menge von Aktivitäten gegenseitig ausschließen, muss dies jeweils für zwei beliebige Aktivitäten aus der Menge geprüft werden. Damit der Bedingungsteil von c_4 in keiner Ausführungsspur erfüllt ist, muss also nur eines der strukturellen Kriterien $A \not< \bullet \not> B$, $B \not< \bullet \not> C$ oder $A \not< \bullet \not> C$ erfüllt sein. Ist allerdings der Bedingungsteil der umgeformten Integritätsregel c_4*'* in mindestens einer Ausführungsspur erfüllt, muss für jede Ausführungsspur, in der dieser erfüllt ist, geprüft werden, ob der Folgeteil von c_4*'* ebenfalls erfüllt ist. Nur wenn das gilt, ist auch die ursprüngliche Integritätsregel c_4 erfüllt.

Automatische Abbildungsvorschrift für die Verifikation, ob der Bedingungsteil in keiner Ausführungsspur erfüllt ist

Sei c eine Integritätsregel mit beliebig vielen positiven und negierten Literalen. Dadurch, dass die negierten Literale des Bedingungsteils in den Folgeteil gebracht werden, enthält der Bedingungsteil nach der Umformung höchstens noch positive Literale oder ist leer. Der Bedingungsteil kann auch leer sein, wenn er ursprünglich bereits leer gewesen ist oder nur negierte Literale enthalten hat. Der leere Bedingungsteil ist per Definition (s. Unterkapitel 5.1.2) in jeder Ausführungsspur erfüllt. Damit die zugehörige Integritätsregel erfüllt ist, muss folglich der Folgeteil ebenfalls in jeder Ausführungsspur erfüllt sein.

Sei p die Anzahl der positiven Literale P_i im Bedingungsteil. Dann gilt:

p = 0 Der Bedingungsteil ist in jeder Ausführungsspur erfüllt (s. Unterkapitel 5.1.2).

p = 1 Der Bedingungsteil ist in keiner Ausführungsspur erfüllt, wenn die entsprechende Aktivität P im Prozess nicht vorkommt.

p > 1 Der Bedingungsteil ist in keiner Ausführungsspur erfüllt, wenn es ein beliebiges Paar *(i,j)* $(1 \leq i,j \leq p)$ gibt, so dass das strukturelle Kriterium $P_i \not< \bullet \not> P_j$ erfüllt ist.

Mit dieser Überlegung lassen sich Bedingungsteile auf strukturelle Kriterien abbilden, um zu verifizieren, ob der Bedingungsteil in keiner Ausführungsspur erfüllt ist.

6.2.2 Strukturelle Kriterien für Folgeteile

Wie in Unterkapitel 6.1 erläutert, muss der Folgeteil einer Integritätsregel mit betrachtet werden, wenn in mindestens einer Ausführungsspur der Bedingungsteil erfüllt ist. In diesem Fall gilt eine Integritätsregel als erfüllt, wenn für jede Ausführungsspur, in der der Bedingungsteil zutrifft, ebenfalls der Folgeteil zutrifft. Dafür werden im Folgenden strukturelle Kriterien hergeleitet.

Wann ist die Integritätsregel $c_5 = A \wedge B \rightarrow C$ erfüllt, wenn in mindestens einer Ausführungsspur der Bedingungsteil erfüllt ist?

Wenn in mindestens einer Ausführungsspur A und B zusammen ausgeführt werden können, ist die Integritätsregel erfüllt, wenn in jeder Ausführungsspur, in der A und B vorkommen ebenfalls C und D vorkommen. Es gilt allerdings, dass sich die Ausführung einer Aktivität immer auf die Ausführung einer anderen Aktivität zurückführen lässt. Das bedeutet, wenn die beiden Aktivitäten A und B die Aktivität C implizieren, lässt sich die Ausführung von C entweder auf die Ausführung von A oder von B zurückführen. Es besteht demnach zwischen den Literalen A und B im Bedingungsteil der Integritätsregel $c_5 = A \wedge B \rightarrow C$ kein Zusammenhang und damit lässt sich dieser zerlegen. Die Integritätsregel c_5 ist demnach erfüllt, wenn entweder in jeder Ausführungsspur, in der A vorkommt, auch C vorkommt oder wenn in in jeder Ausführungsspur, in der B vorkommt, auch C vorkommt. Wenn beim

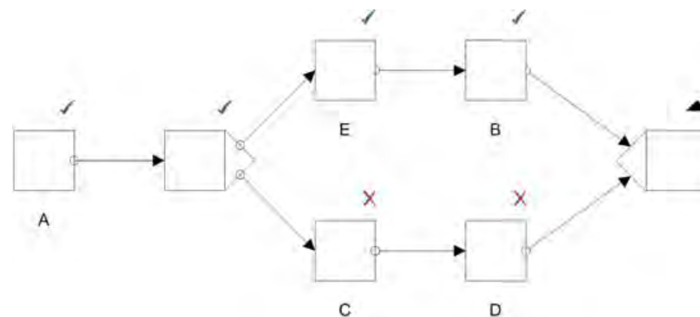


Abbildung 6.4: Wenn der obere Zweig des XOR-Blocks ausgewählt wird, wird zwar A, aber nicht C ausgeführt.

Prozess in Abbildung 6.4 der obere Zweig des XOR-Blocks gewählt wird, wird zwar A aber nicht C ausgeführt. Folglich kommt nicht in jeder Ausführungsspur, in der A vorkommt, ebenfalls C vor. Hingegen wird beim Prozess in Abbildung 6.5 immer, wenn A ausgeführt wird, ebenfalls C ausgeführt. Wenn der obere Zweig des XOR-Blocks gewählt wird, wird A und auch C ausgeführt. Wenn der untere Zweig des XOR-Blocks ausgewählt wird, wird zwar C nicht, aber auch A nicht ausgeführt und somit kommt in jeder Ausführungsspur, in der A vorkommt, ebenfalls C vor.

Allgemein gilt also, dass in jeder Ausführungsspur, in der eine Aktivität S vorkommt, ebenfalls eine Aktivität T vorkommt, wenn sich die beiden Aktivitäten zusammen ausführen lassen. Das bedeutet, dass sie nicht auf verschiedenen Zweigen eines XOR-Blocks liegen

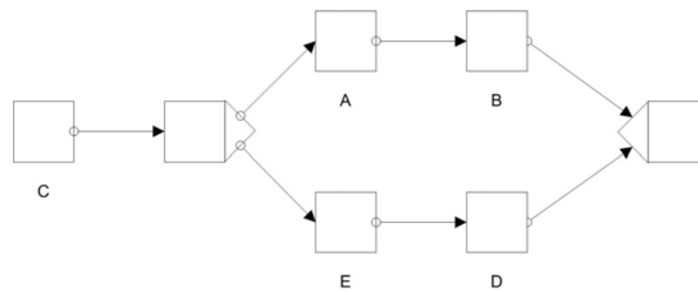


Abbildung 6.5: Unabhängig davon, welcher Zweig des XOR-Blocks ausgewählt wird, kommt in jeder Ausführungsspur, in der A vorkommt, auch C vor.

dürfen. Zusätzlich darf T nicht in einem XOR-Block liegen, in dem S nicht liegt. Anders ausgedrückt, S und T müssen entweder zueinander in einer Vorgänger- bzw. Nachfolgerbeziehung stehen oder auf verschiedenen Zweigen eines AND-Blocks liegen und T darf nicht in einem XOR-Block liegen, in dem S nicht liegt. Für die Verifikation einer solchen Konstellation zweier Aktivitäten S und T definieren wir ein neues strukturelles Kriterium *S impliziert die Ausführung von T ($S \bullet\!<> T$)*, wobei $<>$ hier bedeutet, dass die Reihenfolge, in der die Aktivitäten S und T ausgeführt werden, keine Rolle spielt. Demnach kann S vor oder nach T ausgeführt werden. $S \bullet\!<> T$ evaluiert außerdem zu *true*, wenn S nicht im Prozess vorkommt.

Wann ist die Integritätsregel $c_6 = A \wedge B \rightarrow C \wedge D$ erfüllt, wenn in mindestens einer Ausführungsspur der Bedingungsteil erfüllt ist?

Um das leichter verifizieren zu können, spalten wir die Integritätsregel $c_6 = A \wedge B \rightarrow C \wedge D$ in die beiden Integritätsregeln $c_{61} = A \wedge B \rightarrow C$ und $c_{62} = A \wedge B \rightarrow D$ auf, da folgendes gilt: $A \wedge B \rightarrow C \wedge D \Longleftrightarrow A \wedge B \rightarrow C$ *und* $A \wedge B \rightarrow D$. Die Hinrichtung ("$\Rightarrow$") ist offensichtlich. $c_6 = A \wedge B \rightarrow C \wedge D$ bedeutet, dass in jeder Ausführungsspur, in der A und B vorkommt, auch C und D vorkommen müssen. Die aufgespaltenen Integritätsregeln $c_{61} = A \wedge B \rightarrow C$ bzw. $c_{62} = A \wedge B \rightarrow D$ bedeuten wiederum, dass in jeder Ausführungsspur, in der A und B vorkommt, nur eine Teilmenge der Aktivitäten von c_6, C bzw. D, vorkommen muss. Wenn demnach c_6 erfüllt ist, ist zwangsläufig auch c_{61} und c_{62} erfüllt. Die Rückrichtung ("$\Leftarrow$") wäre nicht korrekt, wenn es zwischen C und D einen Zusammenhang geben würde, so dass C und D in mindestens einer Ausführungsspur nicht zusammen vorkommen könnten. Jedoch kann aufgrund der Integritätsregel-Definition in 5.1 aus c_{61}, dass in jeder Ausführungsspur, in der A und B zusammen vorkommen, auch C vorkommen muss ($c_{61} = A \wedge B \rightarrow C$) und aus c_{62}, dass in jeder Ausführungsspur, in der A und B zusammen vorkommen, auch D vorkommen muss ($c_6 = A \wedge B \rightarrow D$), logisch gefolgert werden, dass in jeder Ausführungsspur, in der A und B zusammen vorkommen, auch C und D vorkommen müssen ($\mathrm{c_6 = A \wedge B \rightarrow C \wedge D}$). Auf gleiche Weise lässt sich eine Integritätsregel aufspalten, wenn ein Literal bzw. mehrere Literale im Folgeteil negiert ist bzw. sind. Die Integritätsregeln c_{61} bzw. c_{62} lassen sich noch weiter zerlegen. Analog zur Integritätsregel c_5 lassen sich c_{61} und c_{62} entlang der Konjunktion zerlegen. c_{61} ist damit

erfüllt, wenn eines der beiden strukturellen Kriterien $A \bullet\!<\!> C$ oder $B \bullet\!<\!> C$ erfüllt ist. c_{62} wiederum ist erfüllt, wenn entweder $A \bullet\!<\!> D$ oder $B \bullet\!<\!> D$ erfüllt ist.

Daraus ergibt sich, dass die Integritätsregel $c_6 = A \wedge B \rightarrow C \wedge D$ folgendermaßen auf strukturelle Kriterien abgebildet wird:

$$\begin{aligned} &\mathrm{c}_6 = \mathrm{A} \wedge \mathrm{B} \rightarrow \mathrm{C} \wedge \mathrm{D} \\ \Longleftrightarrow\ &\mathrm{c}_{61} \equiv \Big(\mathrm{A} \bullet\!<\!> \mathrm{C} \vee \mathrm{B} \bullet\!<\!> \mathrm{C}\Big) \wedge \\ &\mathrm{c}_{62} \equiv \Big(\mathrm{A} \bullet\!<\!> \mathrm{D} \vee \mathrm{B} \bullet\!<\!> \mathrm{D}\Big) \end{aligned}$$

Wann ist die Integritätsregel $c_7 = \neg A \wedge B \rightarrow C$ erfüllt, wenn in mindestens einer Ausführungsspur der Bedingungsteil erfüllt ist?

Angenommen, der Bedingungsteil $\neg A \wedge B$ ist in einer Ausführungsspur erfüllt, dann ist die Integritätsregel c_7 erfüllt, wenn in jeder Ausführungsspur, in der A nicht vorkommt und B vorkommt, C vorkommt. Wie man an c_5 gesehen hat, lässt sich eine Integritätsregel entlang einer Konjunktion im Bedingungsteil zerlegen, wenn auf beiden Seiten dieser Konjunktion positive Literale stehen. Allerdings kann hier so nicht verfahren werden, was im Folgenden erklärt werden soll. $c_7 = \neg A \wedge B \rightarrow C$ bedeutet, dass sich die Ausführung von C auf die Ausführung von B oder auf die Nicht-Ausführung von A zurückführen lässt. Die Nicht-Ausführung von A kann jedoch für die Ausführung verschiedener alternativer Aktivitäten stehen, wobei nicht alle die Ausführung von C implizieren müssen. Deshalb muss die Nicht-Ausführung von A allein nicht C implizieren. Es kann auch sein, dass B allein nicht C impliziert, sondern dass durch die Ausführung von B unter den verschiedenen alternativen Aktivitäten zu A gerade die Aktivität bestimmt wird, auf die sich die Ausführung von C zurückführen lässt. In diesem Fall besteht zwischen den Literalen $\neg A$ und B des Bedingungsteils ein Zusammenhang und damit dieser nicht verloren geht, darf der Bedingungsteil nicht zerlegt werden.

Das soll an dem Prozess in Abbildung 6.6 veranschaulicht werden. Wenn man c_7 wie c_5 zerlegt, erhält man die Integritätsregeln $c' = \neg A \rightarrow C$ und $c'' = B \rightarrow C$. Für diesen Prozess ist zwar c_7 erfüllt, denn in jeder Ausführungsspur, in der A nicht vorkommt und B vorkommt, kommt C vor. Sowohl c' als auch c'' sind jedoch nicht erfüllt. c' ist nicht erfüllt, denn wenn der untere Zweig des äußeren XOR-Blocks ausgewählt wird, wird C nicht ausgeführt, obwohl A nicht ausgeführt wird. c'' ist ebenfalls nicht erfüllt, denn, wenn der untere Zweig des inneren XOR-Blocks ausgewählt wird, wird auch C nicht ausgeführt, obwohl B ausgeführt wird. Das liegt wie oben erklärt daran, dass hier der Zusammenhang zwischen den Literalen verloren geht, wenn der Bedingungsteil zerlegt wird. Zusammenfassend lässt sich sagen, dass aus diesem Grund der Bedingungsteil entlang einer Konjunktion nicht zerlegt werden darf, wenn mindestens eines der umgebenden Literale negiert ist.

c_7 lässt sich jedoch wie bei der Umformung von c_2 oder c_3 umformen, indem das negierte Literal $\neg A$ als positives Literal A auf die Seite des Folgeteils gebracht und mit $\vee$ verknüpft wird. Danach sieht die Integritätsregel folgendermaßen aus: $c_7' = B \rightarrow A \vee C$.

Wenn B im Prozess vorkommt, ist c_7' erfüllt, wenn in jeder Ausführungsspur, in der B vorkommt, ebenfalls A oder C vorkommt.

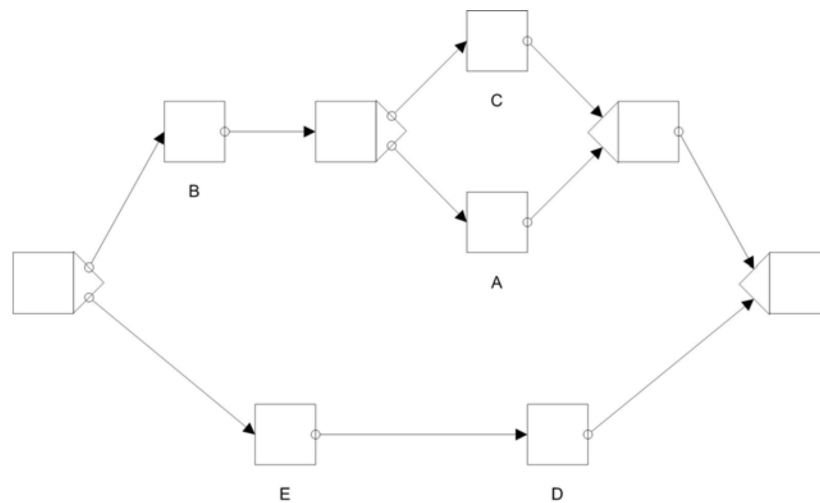


Abbildung 6.6: Die Integritätsregel $c_2 = = \neg A \wedge B \rightarrow C$ ist erfüllt, die beiden Integritätsregeln $c' = \neg A \rightarrow C$ und $c'' = B \rightarrow C$ jedoch jeweils nicht.

Man könnte annehmen, dass sich die Aktivitäten in dieser Konstellation befinden, wenn eines der strukturellen Kriterien $B \bullet\!<> A$ oder $B \bullet\!<> C$ erfüllt ist. Dies soll am Prozess

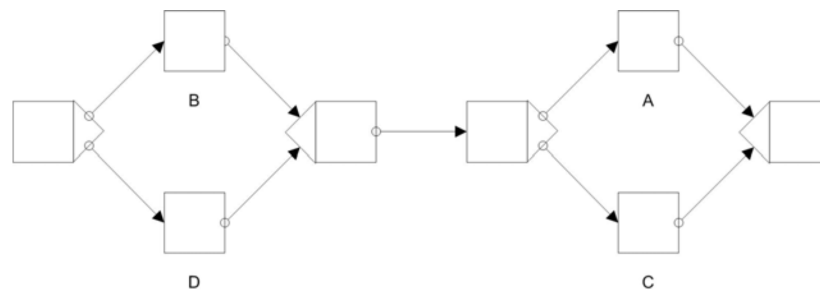


Abbildung 6.7: In jeder Ausführungsspur, in der B vorkommt, kommt entweder A oder C vor. Allerdings ist weder $B \bullet\!<> A$ noch $B \bullet\!<> C$ erfüllt.

in Abbildung 6.7 untersucht werden. Allerdings befinden sich hier A und C jeweils im hinteren XOR-Block, in dem B nicht liegt. Demnach sind die beiden strukturellen Kriterien $B \bullet\!<> A$ und $B \bullet\!<> C$ nicht erfüllt, obwohl in jeder Ausführungsspur entweder A oder C vorkommt. Daraus folgt, dass mit Hilfe der strukturellen Kriterien $B \bullet\!<> A$ oder $B \bullet\!<> C$ nicht verifiziert werden kann, ob in jeder Ausführungsspur A oder C vorkommt.

Um allgemein zu überprüfen, ob in einem ADEPT-Graphen in jeder Ausführungsspur, in der eine Aktivität S vorkommt, ebenfalls mindestens eine Aktivität aus einer bestimmten Menge von Aktivitäten $\{T_1, T_2, \ldots\}$ vorkommt, führen wir deshalb ein neues strukturelles Kriterium, *S impliziert die Versorgung mit den Aktivitäten* $T_1, T_2, \ldots$ $(S \bullet\!<> [\ T_1 \ |\ T_2 \ |\ \ldots])$, ein. Diese Überprüfung entspricht in etwa dem Problem der Versorgung von obligaten Eingabeparametern (s. Unterkapitel 2.4), wobei bekanntlich ein obligater Eingabeparameter als versorgt gilt, wenn in jeder Ausführungsspur, in der die Leseraktivität vorkommt, ein Vorgängerknoten vorkommt, der das Datenelement schreibt.

Der Unterschied zum strukturellen Kriterium $S \bullet<> [\ T_1\ /\ T_2\ /\ \ldots]$ ist, dass die Aktivitäten $T_1,\ T_2, \ldots$ beim Problem der obligaten Datenversorgung vorher ausgeführt werden müssen, während beim strukturellen Kriterium die Reihenfolge unbedeutend ist. Für die Verifikation des strukturellen Kriteriums $A\ \bullet<>\ [B\ /\ C\ /\ \ldots]$ muss deshalb der WriterExists-Algorithmus (s. Unterkapitel 2.4) entsprechend angepasst werden.

Wann ist die Integritätsregel $c_8 = A \rightarrow \neg B \vee \neg C$ erfüllt, wenn in mindestens einer Ausführungsspur der Bedingungsteil erfüllt ist?

Analog zur Umformung von c_2 lässt sich hier das negierte Literal $\neg B$ vom Folgeteil in den Bedingungsteil bringen. c_8 ist nämlich erfüllt, wenn in jeder Ausführungsspur, in der A vorkommt, ebenfalls entweder B nicht oder C nicht vorkommt. Das bedeutet, dass in jeder Ausführungsspur, in der A und B vorkommen, C nicht vorkommen darf, damit c_3 erfüllt ist. Folglich lässt sich c_3 nach $c_3' = A \wedge B \rightarrow \neg C$ umformen. Jedoch lässt sich $\neg C$ nicht auch noch auf die Seite des Bedingungsteils bringen, denn laut Unterkapitel 5.1.2 darf der Folgeteil nicht leer sein. $c_3' = A \wedge B \rightarrow \neg C$ lässt sich allerdings genau wie c_2 bzw. c_3 entlang der Konjunktion im Bedingungsteil zerlegen, da die beiden Literale A und B des Bedingungsteils positiv sind. Der Grund dafür ist, dass die Nicht-Ausführung einer Aktivität sich ebenfalls auf die Ausführung einer einzelnen Aktivität zurückführen lässt. Dies soll mit einem Widerspruchsbeweis gezeigt werden. Angenommen, dies wäre nicht der Fall und $c_3' = A \wedge B \rightarrow \neg C$ ließe sich nicht zerlegen. Dann müsste dadurch, dass die Aktivität A ausgeführt wird, die Aktivität B erst die Aktivität C ausschließen. Damit allerdings B ohne C ausgeführt werden könnte und B nicht C ausschließen würde, müsste C in einem XOR-Block X liegen, in dem B nicht liegt. Da jedoch immer ein Zweig Z von X genommen werden müsste, in dem B nicht liegt, müsste die Ausführung von A eine Aktivität auf Z implizieren. Das wäre jedoch nur möglich, wenn A ebenfalls auf Z liegt. Die Ausführung von A würde folglich die Ausführung von C ausschließen. Dadurch ist die Annahme widerlegt, dass nur durch die Ausführung einer beliebigen Aktivität X eine andere Aktivität Y eine Aktivität Z ausschließt.

Wie bei der Überprüfung von c_5 gezeigt, besteht zwischen den positiven Literalen eines Bedingungsteils kein Zusammenhang und so lässt sich der Bedingungsteil in die beiden Integritätsregeln $c_{81}' = A \rightarrow \neg C$ und $c_{82}' = B \rightarrow \neg C$ zerlegen. Damit ist c_8' (und damit c_8) erfüllt, wenn $c_{81}' = A \rightarrow \neg C$ oder $c_{82}' = B \rightarrow \neg C$ erfüllt sind, wobei c_{81}' erfüllt ist, wenn in jeder Ausführungsspur, in der A vorkommt, nicht C vorkommt. A und C dürfen demnach nicht zusammen in einer Ausführungsspur vorkommen. Dieser Sachverhalt kann mit dem bereits bekannten strukturellen Kriterium *A schließt C aus* ($A \not\prec\bullet\not\succ C$) ausgedrückt werden. Analog kann auch c_{82}' auf das strukturelle Kriterium *B schließt C aus* ($B \not\prec\bullet\not\succ C$) abgebildet werden.

Automatische Abbildungsvorschrift für die Verifikation, ob eine Integritätsregel, die keine Reihenfolgebeziehungen enthält, erfüllt ist

Basierend auf den anhand der Beispiele vorgestellten Überlegungen wurde folgende automatische Abbildungsvorschrift entwickelt, um beliebige Integritätsregeln auf strukturelle Kriterien abzubilden. Dabei wurden Reihenfolgebeziehungen außen vorgelassen. Die Verifikation von Integritätsregeln, die Reihenfolgebeziehungen enthalten, wird in Unterkapitel 6.4 behandelt.

Nach Kapitel 5 ist jede zu verifizierende Integritätsregel von folgender Form:
$c = (\neg)B_1 \wedge \ldots \wedge (\neg)B_b \rightarrow \Big((\neg)F_{11} \vee \ldots \vee (\neg)F_{1n}\Big) \wedge \ldots \wedge \Big((\neg)F_{f1} \vee \ldots \vee (\neg)F_{fn}\Big)$,
(b, in $\geq$ 1).

Schritt 1:
if (Folgeteil von c enthält mindestens eine Konjunktion) **then**

Die Integritätsregel c wird entlang der Konjunktion(en) im Folgeteil zerlegt.

end if
Nach Schritt 1 haben alle Integritätsregeln c_i die folgende Form:
$c_i = (\neg)B_1 \wedge \ldots \wedge (\neg)B_b \rightarrow (\neg)F_1 \vee \ldots \vee (\neg)F_f$, $(b, f \geq 1)$, wobei c erfüllt ist, wenn alle c_i $(1 \leq i \leq n)$, in die c zerlegt wurde, erfüllt sind.

Schritt 2:
if (Bedingungsteil von c_i enthält negierte Literale $\neg B_i$) **then**

Alle negierten Literale $\neg B_i$ werden aus dem Bedingungsteil mit $\vee$ als positive Literale F_i in den Folgeteil gebracht.

end if
Nach Schritt 2 haben alle Integritätsregeln c_i' die folgende Form:
$c_i' = [\ B_1 \wedge \ldots \wedge B_b\ /\ (leer)\ ] \rightarrow (\neg)F_1 \vee \ldots \vee (\neg)F_f$, $(b, f \geq 1)$, wobei c erfüllt ist, wenn alle c_i' erfüllt sind.

Schritt 3:
if (Folgeteil von c_i' enthält negierte Literale $\neg F_i$) **then**
if (Folgeteil enthält mind. ein positives Literal F_i) **then**

Fall a: Alle negierten Literale $\neg F_i$ aus dem Folgeteil werden mit $\wedge$ als positive Literale B_i in den Bedingungsteil gebracht.

else

Fall b: Alle negierten Literale $\neg F_i$ aus dem Folgeteil werden bis auf $\neg F_1$ mit $\wedge$ als positive Literale B_i in den Bedingungsteil gebracht.

end if
end if
Nach Schritt 3 haben alle Integritätsregeln c_i" die folgende Form:

c_i" = [$B_1 \wedge \ldots \wedge B_b$ / (leer)] → [$F_1 \vee \ldots \vee F_f$ / $\neg F_1$] , ($b, f \geq 1$), wobei c erfüllt ist, wenn alle c_i" erfüllt sind.

Schritt 4: Abbildung auf strukturelle Kriterien für die Prüfung des Bedingungsteils

if (Bedingungsteil von c_i" enthält kein positives Literal B_i) **then**

Die Integritätsregel c_i" ist in jeder Ausführungsspur erfüllt.

else if (Bedingungsteil von c_i" enthält genau ein positives Literal B_1) **then**

Die Integritätsregel c_i" ist in keiner Ausführungsspur erfüllt, wenn die Aktivität B_1 nicht im Prozess enthalten ist.

else if (Bedingungsteil von c_i" enthält mehr als ein positives Literal B_i, wobei p die Anzahl der positiven Literale B_i ist) **then**

Die Integritätsregel c_i" ist in keiner Ausführungsspur erfüllt, wenn es mindestens ein beliebiges Paar (i,j) $(1 \leq i,j \leq p)$ gibt, so dass das strukturelle Kriterium $B_i \not< \bullet \not> B_j$ erfüllt ist.

end if

Falls der Bedingungsteil einer Integritätsregel c_i" in keiner Ausführungsspur erfüllt ist, gilt die Integritätsregel als erfüllt. Damit endet die Verifikation für c_i" hier. Ansonsten wird für jede Integritätsregel c_i" normal mit Schritt 5 fortgefahren.

Schritt 5:

if (Bedingungsteil von c_i" enthält mehr als ein positives Literal B_i) **then**

Die Integritätsregeln c_i" werden entlang der Konjunktion(en) im Bedingungsteil in die Integritätsregeln $c_{i,j}$" zerlegt.

end if

Nach Schritt 5 haben alle Integritätsregeln $c_{i,j}$" die folgende Form:
$c_{i,j}$" = [B_1 / (leer)] → [$F_1 \vee \ldots \vee F_f$ / $\neg F_1$] , ($f \geq 1$), wobei Die Integritätsregel c ist erfüllt, wenn von allen c_i" jeweils ein $c_{i,j}$" erfüllt ist.

Schritt 6: Abbildung auf strukturelle Kriterien für die Prüfung des Folgeteils

Die Integritätsregeln können jetzt direkt auf die entsprechenden strukturellen Kriterien abgebildet werden:

if (Folgeteil von $c_{i,j}$" enthält nur das negierte Literal $\neg F_1$) **then**

if (Bedingungsteil = B_1) **then**

Die Integritätsregel $c_{i,j}$" wird auf das strukturelle Kriterium $B_1 \not< \bullet \not> F_1$ abgebildet.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel wird auf das strukturelle Kriterium $START \not< \bullet \not> F_1$ abgebildet.

end if

else if (Folgeteil von $c_{i,j}$" enthält nur das positive Literal F_1) **then**

if (Bedingungsteil = B_1) **then**

Die Integritätsregel wird auf das strukturelle Kriterium $B_1 \bullet <> F_1$ abgebildet.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel wird auf das strukturelle Kriterium $START \bullet <> F_1$ abgebildet.

end if

else if (Folgeteil von $c_{i,j}$" enthält f $(f > 1)$ positive Literale F_i) **then**

if (Bedingungsteil = B_1) **then**

Die Integritätsregel wird auf das strukturelle Kriterium $B_1 \bullet <> [\ F_1 \ / \ldots / \ F_f\ ]$ abgebildet.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel wird auf das strukturelle Kriterium $START \bullet <> [\ F_1 \ / \ldots / \ F_f\ ]$ abgebildet.

end if

end if

Damit ist die ursprüngliche Integritätsregel c erfüllt, wenn für jede Integritätsregel c_i" entweder der Bedingungsteil in keiner Ausführungsspur erfüllt ist oder der Folgeteil in jeder Ausführungsspur erfüllt ist, in der der Bedingungsteil erfüllt ist. Das bedeutet, dass mindestens eines der strukturellen Kriterien, auf das c_i" abgebildet wurde, erfüllt sein muss.

Im Folgenden wird die automatische Abbildungsvorschrift anhand eines Beispiels angewendet.

Beispiel einer Verifikation mit Hilfe der automatischen Abbildungsvorschrift

Anhand der komplexen Integritätsregel $c = A \land \neg B \land \neg C \rightarrow (D \lor \neg E \lor \neg F) \land (G \lor H)$ und dem Prozess in Abbildung 6.8 soll gezeigt werden, wie eine Integritätsregel verifiziert werden kann, indem sie mit Hilfe der automatischen Abbildungsvorschriften auf strukturelle Kriterien abgebildet wird. Bevor allerdings mit der Abbildung auf strukturelle Kriterien begonnen werden kann, muss die Integritätsregel passend umgeformt werden. Da unsere Integritätsregel c einen Konjunktionsoperator im Folgeteil enthält, wird sie in *Schritt 1* in die beiden Integritätsregeln $c_1 = A \land \neg B \land \neg C \rightarrow D \lor \neg E \lor \neg F$ und $c_2 = A \land \neg B \land \neg C \rightarrow G \lor H$ zerlegt.

Im nächsten Schritt werden bei beiden Integritätsregeln c_1 und c_2 jeweils die negierten Literale aus dem Bedingungsteil in den Folgeteil gebracht. Somit sehen nach *Schritt 2* die umgeformten Integritätsregeln c_1' und c_2' folgendermaßen aus:

$c_1' = A \rightarrow D \lor \neg E \lor \neg F \lor B \lor C$

$c_2' = A \rightarrow G \lor H \lor B \lor C$

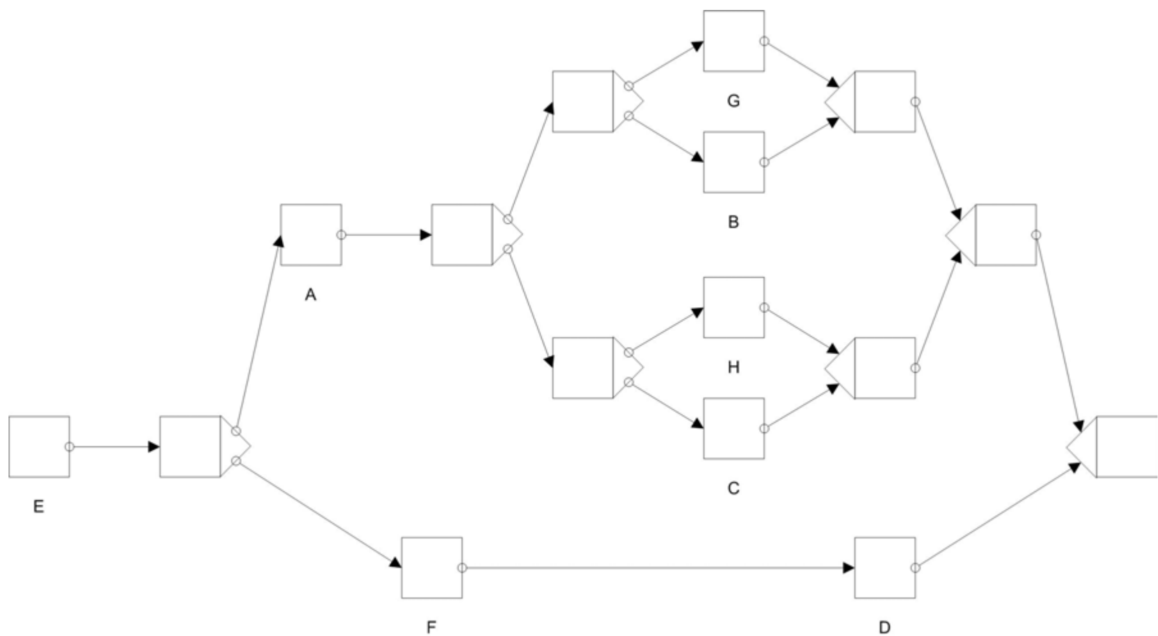


Abbildung 6.8: Die Integritätsregel $c = A \wedge \neg B \wedge \neg C \rightarrow (D \vee \neg E \vee \neg F) \wedge (G \vee H)$ ist erfüllt.

Da der Folgeteil von c_1’ positive und negierte Literale enthält, werden alle negierten Literale in den Bedingungsteil gebracht. Nach dieser Umformung von *Schritt 3 (Fall a)* ist c_1” $= A \wedge E \wedge F \rightarrow D \vee \vee B \vee C$. Da c_2’ kein negiertes Literal enthält, ist c_2” unverändert zu c_2’.

In *Schritt 4* können jeweils die umgeformten Integritätsregeln auf strukturelle Kriterien abgebildet werden. Dabei wird c_1” auf $A \not< \bullet \not> E$, $A \not< \bullet \not> F$ und $E \not< \bullet \not> F$ abgebildet. Da das strukturelle Kriterium $A \not< \bullet \not> F$ erfüllt ist, ist der Bedingungsteil von c_1” in keiner Ausführungsspur erfüllt. Damit gilt c_1” als erfüllt. c ist allerdings erst erfüllt, wenn sowohl c_1” als auch c_2” erfüllt sind. Deshalb muss ungeachtet der Tatsache, dass c_1” bereits als erfüllt verifiziert werden konnte, noch c_2” überprüft werden. Der Bedingungsteil von c_2” enthält nur das Literal A. Folglich ist der Bedingungsteil in mindestens einer Ausführungsspur erfüllt, da A im Prozess vorkommt. Es muss demnach verifiziert werden, ob in jeder Ausführungsspur, in der der Bedingungsteil von c_2” erfüllt ist, ebenfalls dessen Folgeteil erfüllt ist.

Da der Bedingungsteil von c_2” nur ein Literal enthält, muss c_2” nicht weiter umgeformt werden (*Schritt 5*), sondern kann direkt in *Schritt 6* auf das strukturelle Kriterium $A \bullet\!<\!> [G \mid H \mid B \mid C]$ abgebildet werden. $A \bullet\!<\!> [G \mid H \mid B \mid C]$ ist erfüllt. Daraus folgt, dass in jeder Ausführungsspur, in der der Bedingungsteil erfüllt ist, ebenso der Folgeteil von c_2” und dadurch auch c_2” erfüllt ist. Somit sind sowohl c_1” als auch c_2” erfüllt und deshalb auch die ursprüngliche Integritätsregel c.

Strukturelles Kriterium	Bedeutung
A schließt B aus: $A \not< \bullet \not> B$	In jeder Ausführungsspur, in der A vorkommt, kommt nicht B vor.
A impliziert B: $A \bullet <> B$	In jeder Ausführungsspur, in der A vorkommt, kommt auch B vor.
A impliziert die Versorgung mit {B, C ...}: $A \bullet <> [B \mid C \mid \ldots]$	In jeder Ausführungsspur, in der A vorkommt, kommt mind. eine der Aktivitäten aus der Menge $\{B, C, \ldots\}$ vor.

Tabelle 6.1: Überblick über neu eingeführte strukturelle Kriterien

6.3 Verifikation der strukturellen Kriterien

In diesem Unterkapitel wird im Detail vorgestellt, wie die Verifikation der einzelnen strukturellen Kriterien über entsprechende Verifikationsalgorithmen abläuft.

6.3.1 Die Ausführung der Quellaktivität *S* schließt die Ausführung der Zielaktivität *T* aus ($S \not< \bullet \not> T$)

Damit das strukturelle Kriterium $S \not< \bullet \not> T$ erfüllt ist, dürfen in keiner Ausführungsspur S und T zusammen vorkommen. Dies prüft der *Exclusion*-Algorithmus (s. Anhang A.1, Algorithmus 1). Zuerst wird hier geprüft, ob eine der beiden Aktivitäten nicht im Prozess vorkommt. In diesem Fall ist $S \not< \bullet \not> T$ erfüllt. Das erfolgt durch eine definierte *index()*-Funktion, die den Index einer Aktivität zurückliefert oder -1, falls die Aktivität nicht im Prozess vorkommt. Selbst wenn beide Aktivitäten im Prozess vorkommen, können sie in keiner Ausführungsspur zusammen vorkommen, für den Fall, dass sie auf verschiedenen Zweigen eines XOR-Blocks liegen. Dazu muss der minimale Block, der S und T umschließt, der XOR-Block sein. Falls eine der beiden Aktivitäten allerdings der XOR-Split oder XOR-Join ist, kommen die beiden Aktivitäten dennoch in einer Ausführungsspur vor. Damit S und T sich wirklich auf verschiedenen Zweigen eines XOR-Blocks befinden, muss zusätzlich zu der Bedingung, dass beide in einem XOR-Block liegen müssen, die Bedingung gelten, dass S weder Vorgänger noch Nachfolger von T ist.

6.3.2 Die Ausführung der Quellaktivität *S* impliziert die Ausführung der Zielaktivität *T* ($S \bullet <> T$)

Das strukturelle Kriterium $S \bullet <> T$ ist erfüllt, wenn entweder in jeder Ausführungsspur, in der die Quellaktivität S vorkommt, ebenfalls die Zielaktivität T vorkommt oder wenn die Quellaktivität S nicht im Prozess vorkommt. Zur Verifikation dieses strukturellen Kriteriums wurden hier zwei verschiedene Ansätze entwickelt. Bei dem ersten Ansatz, dem TargetExists1-Algorithmus, wurde der WriterExists-Algorithmus (s. Unterkapitel 2.4), der die Versorgung obligater Eingabeparameter überprüft, angepasst. Dieser

beruht darauf, den Prozessgraphen in Breitensuche zu traversieren. Der andere Ansatz, der LabelledTargetExists1-Algorithmus, beruht darauf, die Positionen der Quell- und der Zielaktivität relativ zu den XOR-Blöcken im Prozessgraphen zu bestimmen. Im Folgenden wird zuerst der TargetExists1-Algorithmus und im Anschluss der LabelledTargetExists1-Algorithmus vorgestellt.

Verifikationsalgorithmus TargetExists1

Der TargetExists1-Algorithmus (s. Anhang A.1, Algorithmus 2) basiert auf dem WriterExists-Algorithmus, wobei die Zielaktivität T bzw. die Quellaktivität S dem Schreiberknoten bzw. dem Leserknoten entspricht. Zwischen dem TargetExists1-Algorithmus und dem WriterExists-Algorithmus gibt es nur einen wesentlichen Unterschied.

Während die Versorgung des obligaten Eingabeparameters eines Leserknoten nur Vorgängerknoten sicherstellen können, kommen für die „Versorgung" der Quellaktivität S sämtliche Knoten im Prozess in Frage. Das bedeutet, dass die Reihenfolge zwischen der Quellaktivität S und der Zielaktivität T unbedeutend ist. Deshalb muss beim TargetExists1-Algorithmus in der Markierungsphase die Konstellation zwischen der Quellaktivität und der Zielaktivität bestimmt werden. Dabei gibt es grundsätzlich vier Konstellationen, in denen die Quell- und die Zielaktivität im Prozessgraphen zueinander liegen können. Sie können zueinander in einer Vorgänger- bzw. Nachfolgerbeziehung stehen oder auf verschiedenen Zweigen eines AND- bzw. XOR-Blocks liegen, wobei die letzte Konstellation nicht betrachtet werden muss, da in diesem Fall die Zielaktivität nicht zusammen mit der Quellaktivität ausgeführt werden kann (s. Unterkapitel 6.3.2).

Der Prozessgraph wird ausgehend von der Zielaktivität zur Quellaktivität traversiert. Folglich wird der Prozessgraph in Vorwärts- bzw. Rückwärtsrichtung traversiert, wenn die Quellaktivität Nachfolger bzw. Vorgänger der Zielaktivität ist. Wenn die Quell- und die Zielaktivität jedoch auf verschiedenen Zweigen eines AND-Blocks liegen, wird der Prozessgraph ausgehend von der Zielaktivität in Vorwärtsrichtung bis zum AND-Join-Knoten durchlaufen. Der Grund dafür ist, dass in jeder Ausführungsspur, in der der AND-Join-Knoten vorkommt, ebenfalls der Zweig, auf dem die Quellaktivität liegt, vorkommt. Falls Quell- und Zielaktivität auf verschiedenen Zweigen eines AND-Blocks liegen, wird versucht, die „Versorgung" des AND-Join-Knotens sicherzustellen. In diesem Fall wird immer dann, wenn die Quellaktivität ausgeführt wird, eine der Zielaktivitäten ausgeführt. Das ist trivialerweise auch der Fall, wenn die Quellaktivität auf dem Zweig des AND-Blocks in einem weiteren XOR-Block liegen sollte.

Da bei dem TargetExists1-Algorithmus die Reihenfolge, in der die Quellaktivität und die Zielaktivität ausgeführt werden, keine Rolle spielt, müssen hier im Gegensatz zum WriterExists-Algorithmus Sync-Kanten (s. Unterkapitel 2.2) nicht betrachtet werden.

Ein weiterer Unterschied zum WriterExists-Algorithmus besteht darin, dass der TargetExists1-Algorithmus *true* zurückliefert, wenn die Quellaktivität S nicht im Prozess vorkommt.

Anhand eines Beispiels wird im Folgenden der TargetExists1-Algorithmus vorgestellt.

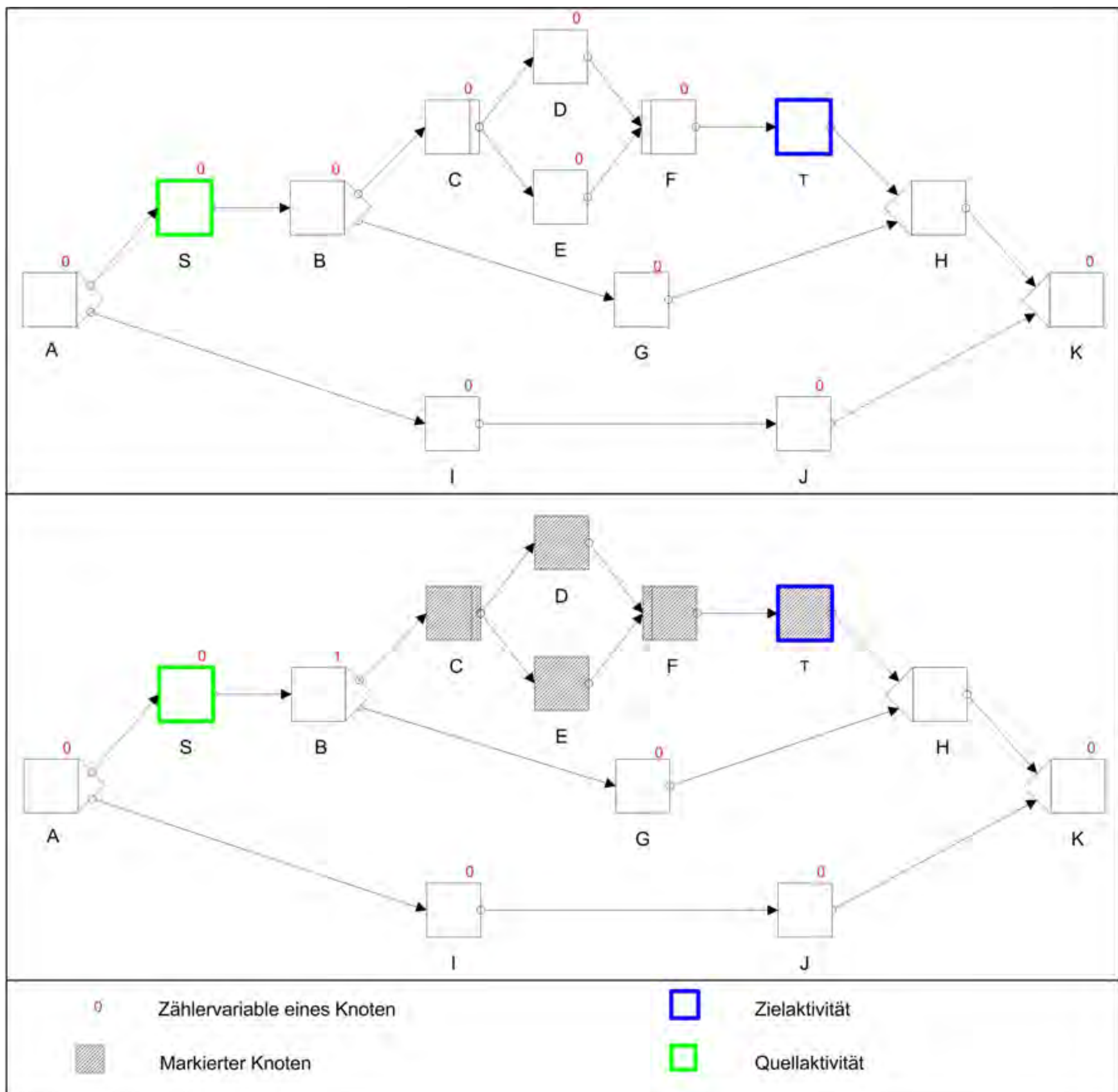


Abbildung 6.9: Verifikation des strukturellen Kriteriums $S \bullet <> T$ mit dem TargetExists1-Algorithmus

Nachdem alle Knoten des Prozesses initialisiert wurden, indem jeweils die Zählervariable auf *0* gesetzt wurde und die eventuelle Markierung gelöscht wurde, wird die Konstellation zwischen der Quellaktivität S und der Zielaktivität T im Prozessgraphen bestimmt. S ist ein Vorgänger von T. Somit wird der Prozess in Rückwärtsrichtung traversiert. Abgesehen davon läuft der TargetExists1-Algorithmus analog zum WriterExists-Algorithmus ab. Zuerst wird *T1* als versorgt markiert und daraufhin die Zählervariable des einzigen Nachfolgerknoten F inkrementiert. Da F kein XOR-Split-Knoten ist, kann er auch markiert werden, denn er kommt in jeder Ausführungsspur vor, in der auch T vorkommt. Als nächstes werden von F die beiden Nachfolgerknoten D und E betrachtet. Das bedeutet, dass ihre Zähler ebenfalls inkrementiert werden. Da D und E keine XOR-Split-Knoten sind, können sie markiert werden. Im Anschluss wird C betrachtet. Da für C dasselbe gilt, wird C markiert.

Der Nachfolgerknoten von C ist ein XOR-Split-Knoten. Daraus folgt, dass dessen Zählervariable gleich der Anzahl der ausgehenden Kontrollkanten sein muss, damit dieser markiert werden kann. Nur in diesem Fall ist sichergestellt, dass bei Ausführung jedes XOR-Zweigs eine Zielaktivität vorkommt. Allerdings ist dies hier nicht der Fall, somit kann B nicht markiert werden. Da kein weiterer Knoten mehr markiert werden kann, endet der Algorithmus, ohne dass die Quellaktivität S markiert werden konnte. Das strukturelle Kriterium $S \bullet <> T$ ist demnach auf diesem Prozess nicht erfüllt.

Verifikationsalgorithmus LabelledTargetExists1 basierend auf Positionsmarkierungen

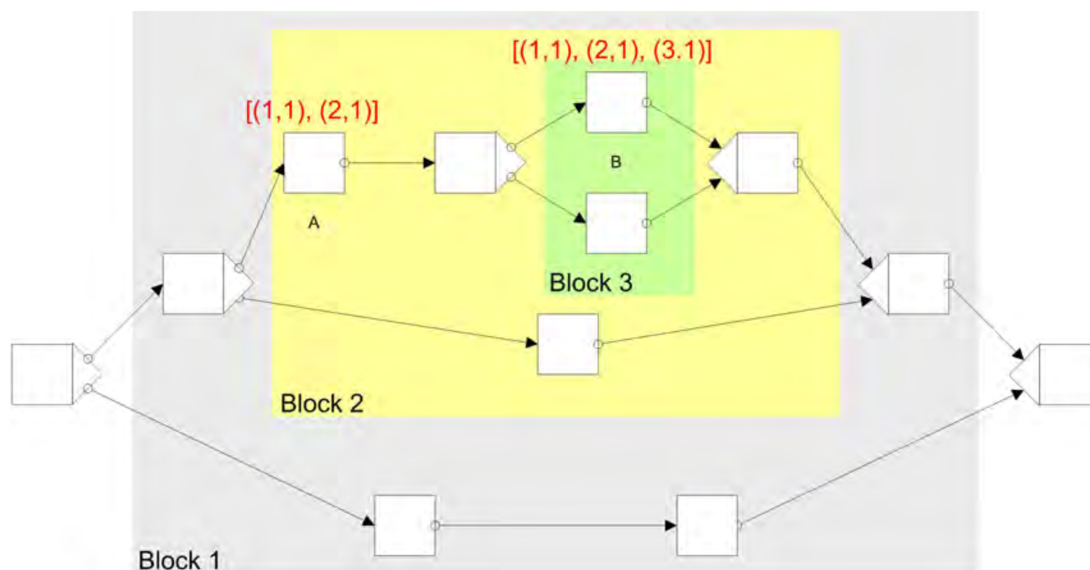
Der Prinzip des Algorithmus (s. Anhang A.1, Algorithmus 3) besteht darin, die Konstellation der beiden Aktivitäten S und T in Bezug auf die XOR-Blöcke herauszufinden, denn bei der Verifikation des strukturellen Kriteriums $S \bullet <> T$ spielt nur eine Rolle, wie S und T in Bezug auf die XOR-Blöcke im Prozess liegen. Zum einen müssen nämlich S und T zusammen ausgeführt werden können, d.h. sie dürfen nicht auf verschiedenen Zweigen eines XOR-Blocks liegen und zum anderen darf T nicht in einem XOR-Block liegen, in dem S nicht liegt. Das bedeutet, es muss nur geprüft werden, ob S und T auf verschiedenen Zweigen eines XOR-Blocks liegen oder T in einem XOR-Block liegt, in dem T nicht liegt. In diesen Fällen ist das strukturelle Kriterium $S \bullet <> T$ nicht erfüllt, in allen anderen Fällen ist es erfüllt.

Um die Konstellation der beiden Aktivitäten bzgl. der XOR-Blöcke herauszufinden, verwendet der Algorithmus Positionsmarkierungen. Die Positionsmarkierung für eine Aktivität gibt dabei an, in welchen XOR-Blöcken und jeweils auf welchem Zweig in diesen XOR-Blöcken diese Aktivität liegt. Sie ist als eine *FIFO*-Liste *(First In, First Out)* von Tupeln der Form *(Blocknummer, Zweignummer)* realisiert, wobei ein Tupel *(Blocknummer, Zweignummer)* für einen XOR-Block steht, in dem die Aktivität liegt. Die Blocknummer kennzeichnet dabei eindeutig den XOR-Block und die Zweignummer den Zweig in diesem XOR-Block. Die dafür notwendige Nummerierung der Blöcke und Zweige erfolgt in der Reihenfolge, in der die Aktivitäten beim Durchlaufen des Prozesses besucht werden. Dabei werden die Aktivitäten blockweise besucht. Im Falle einer Verzweigung werden die Zweige der Reihe nach von oben nach unten ausgewählt.

Bei diesem Algorithmus wird zuerst geprüft, ob Quell- und Zielaktivität überhaupt im Prozess vorkommen. Wenn die Quellaktivität nicht vorkommt, gilt das strukturelle Kriterium als erfüllt. Wenn die Quellaktivität vorhanden ist, die Zielaktivität jedoch nicht vorkommt, ist der Trivialfall gegeben und das strukturelle Kriterium somit nicht erfüllt. Sind sowohl Quell- als auch Zielaktivität vorhanden, werden die Positionsmarkierungen der Aktivitäten vorgenommen.

Enthält die Positionsmarkierung von T keine Tupel, dann befindet sich T in keinem XOR-Block und kommt demnach in jeder Ausführungsspur vor. Damit kommt T auch in jeder Ausführungsspur vor, in der S vorkommt und das strukturelle Kriterium $S \bullet <> T$ ist erfüllt. Enthält die Positionsmarkierung von T hingegen Tupel, müssen sämtliche Tupel

dieser Positionsmarkierung ebenfalls in der Positionsmarkierung von S vorkommen. Denn wenn T in einem XOR-Block liegt, in dem S nicht liegt, fehlt in der Positionsmarkierung von S ein Tupel mit der entsprechenden Blocknummer. Auch wenn zwar S und T im selben XOR-Block aber auf verschiedenen Zweigen liegen, enthält die Positionsmarkierung von S zwar ein Tupel mit gleicher Blocknummer aber mit unterschiedlicher Zweignummer. In diesem Fall fehlt ebenfalls in der Positionsmarkierung von S das entsprechende Tupel. Somit liegt S nur in jedem XOR-Block, in dem T liegt, wenn jedes Tupel der Positionsmarkierung von T ebenfalls in der Positionsmarkierung von S vorkommt. In diesem Fall ist das strukturelle Kriterium $S \bullet <> T$ erfüllt.



[(1,2), (2,1)] Positionsmarkierung einer Aktivität

Abbildung 6.10: Verifikation von $A \bullet <> B$ mit dem LabelledTargetExists1-Algorithmus

Bei dem Prozess in Abbildung 6.10 liegt B in einem XOR-Block, in dem A nicht liegt. Das strukturelle Kriterium $A \bullet <> B$ ist demnach auf diesem Prozess nicht erfüllt. Das soll im Folgenden mit dem LabelledTargetExists1-Algorithmus nachgewiesen werden.

Da A und B beide im Prozess vorkommen, werden zuerst die Positionsmarkierungen der Aktivitäten A und B vorgenommen. Die Positionsmarkierung von A ist *[(1,1), (2,1)]*, da A auf Zweig *1* im XOR-Block *1* bzw. auf Zweig *1* im XOR-Block *2* liegt und die Positionsmarkierung von B ist *[(1,1), (2,1), (3,1)]*, da B ebenfalls in den gleichen XOR-Blöcken und jeweils auf den gleichen Zweigen wie A liegt und zusätzlich noch im Block *3* auf dem Zweig *1*.

Da die Positionsmarkierung von B Tupel enthält, wird in einer Schleife geprüft, ob alle Tupel der Positionsmarkierung von B ebenfalls in der Positionsmarkierung von A enthalten sind. Damit wird geprüft, ob A in jedem XOR-Block liegt, in dem auch B liegt. Im ersten Schleifendurchlauf wird das Tupel *(1,1)* aus der Positionsmarkierung von B entnommen.

Das Tupel *(1,1)* kommt ebenfalls in der Positionsmarkierung von A vor, was zur Folge hat, dass das nächste Tupel *(2,1)* aus der Positionsmarkierung von A entnommen wird, das wiederum auch in der Positionsmarkierung von A vorkommt. Da das letzte Tupel der Positionsmarkierung von B *(3,1)* nicht in der Positionsmarkierung von A vorkommt, liegt B in dem XOR-Block Nr. *3* auf Zweig *1*, in dem A nicht liegt. Somit ist das strukturelle Kriterium $A \bullet <> B$ auf dem Prozess in Abbildung 6.10 verletzt.

Der LabelledTargetExists1-Algorithmus ist im Normalfall aufwendiger als der WriterExists-Algorithmus, da jedes Mal zuerst der Prozess durchlaufen werden muss, um die Positionsmarkierungen der Quell- und der Zielaktivität zu bestimmen. Wenn allerdings das Prozessmetamodell bereits diese Positionsmarkierungen der Knoten, z.B. als Knoten-IDs, bereitstellt, ist der LabelledTargetExists1-Algorithmus wesentlicher effizienter und dem WriterExists-Algorithmus vorzuziehen.

6.3.3 Die Ausführung der Quellaktivität S impliziert die Versorgung mit den Zielaktivitäten T_i ($S \bullet <> [T_1/T_2/\ldots]$)

Das strukturelle Kriterium $S \bullet <> [T_1/T_2/\ldots]$ ist für einen Prozess erfüllt, wenn entweder in jeder Ausführungsspur, in der die Quellaktivität S vorkommt, mindestens eine der Zielaktivitäten T_i vorkommt oder wenn die Quellaktivität S im Prozess nicht vorkommt. Dieses strukturelle Kriterium ähnelt dem strukturellen Kriterium $S \bullet <> T$ mit dem Unterschied, dass bei $S \bullet <> T$ in jeder Ausführungsspur, in der die Quellaktivität S vorkommt, die gleiche Zielaktivität T vorkommt. Wie bei der Verifikation des strukturellen Kriteriums $S \bullet <> T$ wurden wieder dieselben beiden Ansätze verfolgt. Zum einen wurde der WriterExists-Algorithmus für dieses Verifikationsproblem angepasst. Zum anderen wurde ein Ansatz basierend auf den Positionsmarkierungen entwickelt, die bei der Verifikation des strukturellen Kriteriums $S \bullet <> T$ eingeführt wurden.

Verifikationsalgorithmus TargetExists2

Der TargetExists1-Algorithmus zur Überprüfung des strukturellen Kriteriums $S \bullet <> T$ wird hier angepasst. Der einzige Unterschied zwischen beiden Verifikationsszenarien ist, dass bei $S \bullet <> [T_1/T_2/\ldots]$ mehrere Zielaktivitäten die „Versorgung" der Quellaktivität sicherstellen können. Deswegen läuft beim TargetExists2-Algorithmus die Markierungsphase in mehreren Iterationen ab, wobei in jeder Iteration ausgehend von der gerade betrachteten Zielaktivität T_i der Prozessgraph in Richtung der Quellaktivität S traversiert wird. Jede Iteration entspricht der Ausführung des TargetExists1-Algorithmus.

Im Folgenden soll mit Hilfe des TargetExists2-Algorithmus das strukturelle Kriterium $S \bullet <> [T1\ /\ T2\ /\ \ldots]$ für den Prozess in Abbildung 6.11 verifiziert werden. Nachdem geprüft wurde, dass die Quellaktivität S im Prozess vorkommt, wird zuerst in der Initialisierungsphase bei allen Knoten der Zähler auf *0* gesetzt und eine eventuelle Markierung zurückgesetzt. Darauf folgt die Markierungsphase.

In der ersten Iteration wird dabei die Zielaktivität *T1* betrachtet. Da *T1* und S auf verschiedenen Zweigen eines AND-Blocks mit dem AND-Join-Knoten L liegen, wird L in

dieser Iteration als Quellaktivität betrachtet und der Prozess muss ausgehend von $T1$ in Richtung L traversiert werden. Zuerst wird $T1$ markiert, anschließend wird der Zähler aller nachfolgenden Knoten erhöht. Das bedeutet, dass der Zähler von K den Wert 1 hat. Allerdings ist K ein XOR-Join-Knoten. Damit dieser markiert werden kann muss sichergestellt sein, dass bei jeder Ausführung eines XOR-Blocks eine Zielaktivität vorkommt. Deshalb muss der Wert der Zählervariablen gleich der Anzahl der eingehenden Kontrollflusskanten sein, damit ein XOR-Join-Knoten markiert werden kann. Das ist hier nicht der Fall und folglich kann K nicht markiert werden. Da $T1$ sonst keine weiteren Nachfolgerknoten hat, bricht hier die Iteration ab.

In der zweiten Iteration ist $T2$ die aktuelle Zielaktivität. Da die Quellaktivität S ein Vorgänger von $T2$ ist, wird der Prozess in Rückwärtsrichtung traversiert. Zuerst wird wieder $T2$ markiert. Danach wird die Zählervariable des einzigen Nachfolgers C um 1 erhöht. Da C ein XOR-Split-Knoten ist, muss zur Sicherstellung, dass bei der Ausführung jedes XOR-Zweigs die Versorgung mit einer Zielaktivität sichergestellt ist, der Wert der Zählervariablen gleich der Anzahl der ausgehenden Kontrollflusskanten von C sein. Folglich kann C nicht als „versorgt“ markiert werden. Da sonst keine weiteren Knoten mehr betrachtet werden können, endet hier die zweite Iteration.

In der nächsten Iteration stellt $T3$ die aktuelle Zielaktivität dar. Nachdem $T3$ markiert wurde, wird der Zähler des Vorgängers C um 1 erhöht. Der Zähler des XOR-Split-Knoten C hat jetzt den Wert 2, der gleich der Anzahl der ausgehenden Kontrollflusskanten ist. Folglich ist jede Ausführung eines XOR-Zweigs durch die Ausführung einer Zielaktivität „versorgt“ und C kann ebenfalls als „versorgt“ markiert werden. Daraufhin wird der Zähler der Vorgängeraktivität S auf 1 erhöht. Da die Quellaktivität S ein „normaler“ Knoten ist, wird sie ebenfalls markiert. Somit konnte die Quellaktivität markiert werden und der TargetExists2-Algorithmus liefert zurück, dass das strukturelle Kriterium $S \bullet\!<\!> [T1 \mid T2 \mid \ldots]$ erfüllt ist.

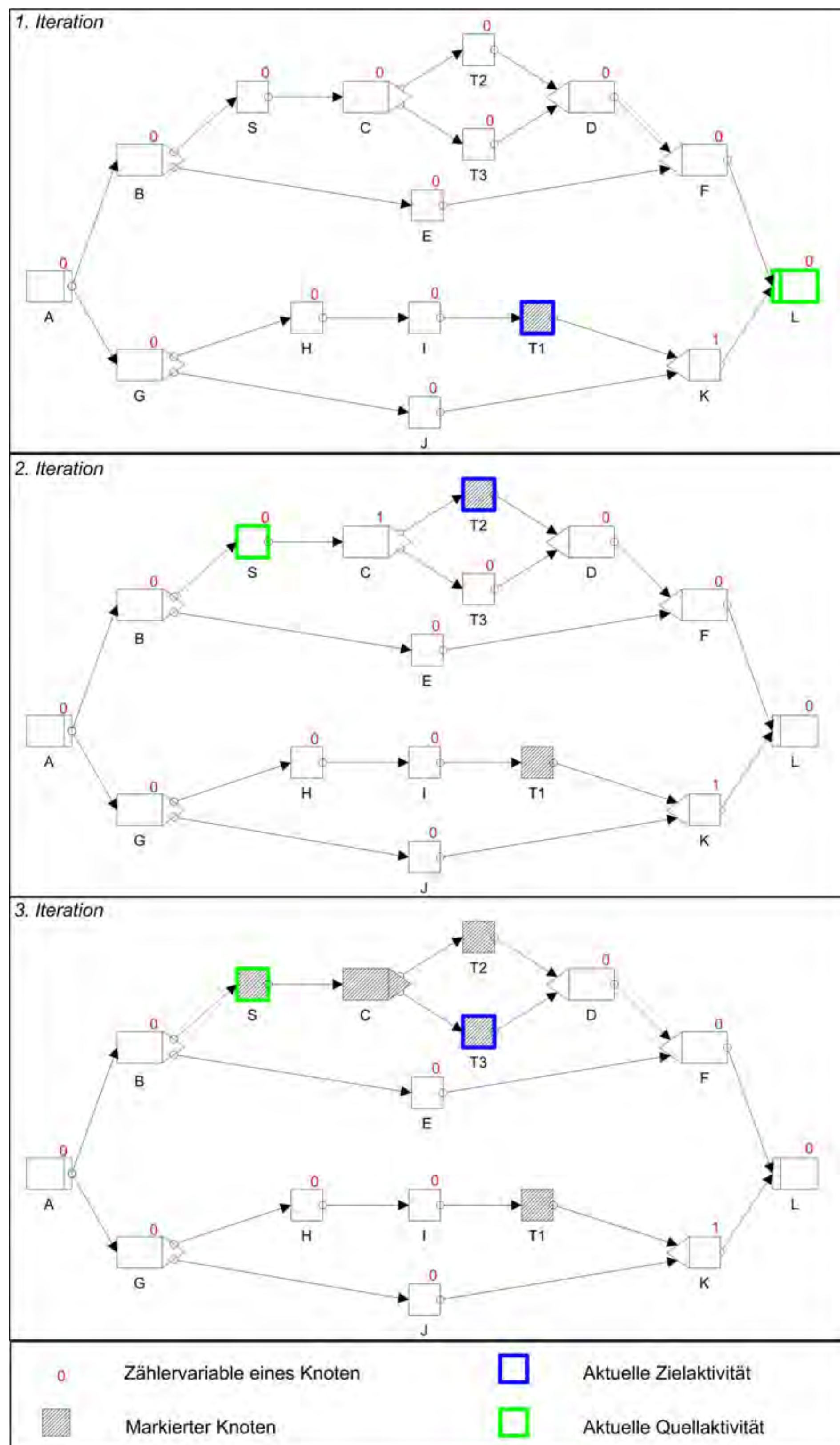


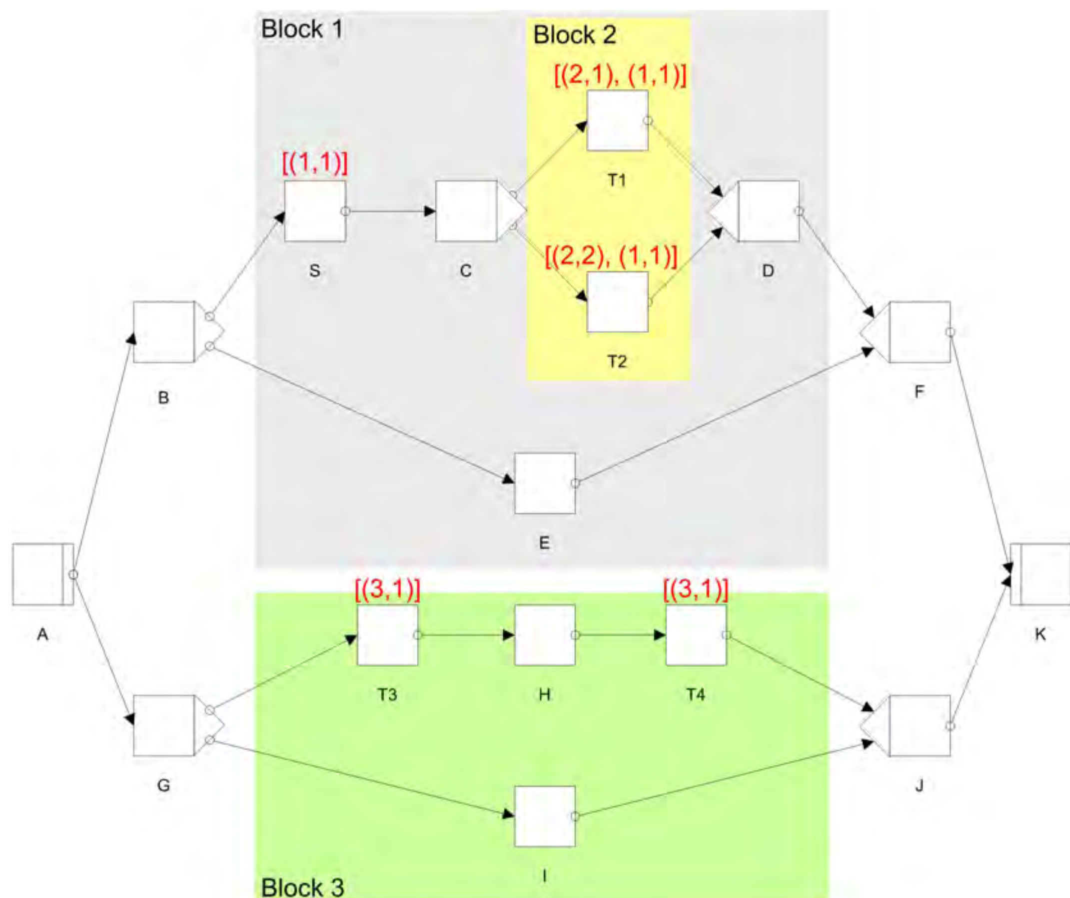
Abbildung 6.11: Verifikation des strukturellen Kriteriums *S •<> [T1 | T2 | T3]* mit dem TargetExists2-Algorithmus

Zeitkomplexität des Verifikationsalgorithmus Es wird angenommen, dass jede Operation einen konstanten Aufwand hat und jeder Knoten höchstens *n-1* Nachfolgerknoten hat. Bei der Aufwandsschätzung ist die Größe der Eingabe n gleich der Anzahl der Knoten im Prozess. Wenn man nur die Schleifen betrachtet, hat der Algorithmus eine Komplexität von $O(n^3)$. Aber im Normalfall ist die Komplexität viel geringer. AND- und Aktivitätenknoten müssen nämlich nur einmal ausgewertet werden und nur XOR-Join-Knoten, wenn die Quellaktiviät hinter der Zielaktivität liegt, bzw. XOR-Split-Knoten, wenn die Quellaktiviät vor der Zielaktivität liegt, müssen eventuell mehrfach ausgewertet werden.

Verifikationsalgorithmus LabelledTargetExists2 basierend auf Positionsmarkierungen

Bei diesem Verfahren wird genau wie bei der Verifikation des strukturellen Kriteriums $S \bullet <> T$ mit dem LabelledTargetExists1-Algorithmus die Konstellation der Quell- und der Zielaktivitäten bezüglich der XOR-Blöcke geprüft. Alle XOR-Blöcke, in denen die Quellaktivität liegt, müssen nicht betrachtet werden, da nur in jeder Ausführungsspur, in der die Quellaktivität vorkommt, die Versorgung mit den Zielaktivitäten sichergestellt sein muss. Deshalb werden die den XOR-Blöcken entsprechenden Tupel aus den Positionsmarkierungen aller Zielaktivitäten entfernt.

Als erstes wird wiederum geprüft, ob die Quellaktivität S im Prozess vorkommt. Kommt sie nicht vor, liefert der Algorithmus *true* zurück. Andernfalls wird für jede Zielaktivität T_i geprüft, ob die Quellaktivität S die Zielaktivität T_i impliziert, was dem strukturellen Kriterium $S \bullet <> T_i$ entspricht. $S \bullet <> T_i$ ist erfüllt, wenn die Zielaktivität T_i in keinem XOR-Block liegt, in dem die Quellaktivität nicht liegt. Das bedeutet, dass alle Tupel der Positionsmarkierung von T_i ebenfalls in der Positionsmarkierung von S vorkommen müssen. Nachdem also aus der Positionsmarkierung der Zielaktivität T_i alle Tupel von S gelöscht wurden, ist $S \bullet <> T_i$ erfüllt, wenn die Positionsmarkierung von T_i keine Tupel mehr enthält. Wenn die Positionsmarkierung einer Zielaktivität nicht leer ist, wird geprüft, ob die Ziel- und die Quellaktivität auf verschiedenen Zweigen eines XOR-Blocks liegen. Anhand der Positionsmarkierung der Quell- und der Zielaktivität kann das leicht festgestellt werden, denn in dem Fall enthalten die Positionsmarkierungen der beiden Aktivitäten jeweils ein Tupel mit gleicher Block- und unterschiedlicher Zweignummer. Wurden zwei solche Tupel gefunden, gilt das strukturelle Kriterium $S \not< \bullet \not> T$ als erfüllt und diese Zielaktivität muss nicht weiter für die Sicherstellung der Versorgung mit den Zielaktivitäten betrachtet werden. Wurde das strukturelle Kriterium $S \bullet <> [T_1/T_2/\ldots]$ noch nicht als erfüllt verifiziert und wurden noch nicht alle Zielaktivitäten von der Verifikation ausgeschlossen, da $S \not< \bullet \not> T_i$ erfüllt ist, so werden die verbliebenen Zielaktivitäten solange in einer Schleife untersucht, bis herausgefunden wurde, ob die Ausführung der Quellaktivität die Versorgung mit den Zielaktivitäten sicherstellt oder bis keine weiteren Knoten mehr untersucht werden können. Die verbliebenen Zielaktivitäten werden zuerst absteigend nach der Anzahl der Tupel ihrer Positionsmarkierungen sortiert. Das bedeutet, dass die erste Zielaktivität in den meisten XOR-Blöcken enthalten ist. Diese Zielaktivität wird zuerst betrachtet und



[(1,2), (2,1)] Positionsmarkierung einer Aktivität

Abbildung 6.12: Verifikation des strukturellen Kriteriums $S \bullet <> [T1 \,/\, T2 \,/\, T3 \,/\, T4]$ mit dem LabelledTargetExists2-Algorithmus

das Tupel, das dem innersten XOR-Block entspricht, ausgelesen. Dann wird für diesen XOR-Block untersucht, ob für jeden Zweig die Versorgung mit den Zielaktivitäten sichergestellt ist. Ein Zweig eines XOR-Blocks gilt dabei als versorgt, wenn das oberste Tupel der Positionsmarkierung als Blocknummer die Blocknummer dieses XOR-Blocks und als Zweignummer die Zweignummer des gerade untersuchten Zweigs enthält. Wenn für mindestens einen Zweig des XOR-Blocks die Versorgung mit den Zielaktivitäten nicht sichergestellt werden kann, dann kann für die Ausführung dieses XOR-Blocks die Versorgung mit den Zielaktivitäten nicht sichergestellt werden und damit müssen alle Zielaktivitäten in diesem XOR-Block nicht weiter betrachtet werden. Hat jedoch die Überprüfung des XOR-Blocks ergeben, dass bei dessen Ausführung die Versorgung mit den Zielaktivitäten sichergestellt ist, muss sich um diesen XOR-Block folglich nicht mehr gekümmert werden und alle Tupel, die zu diesem XOR-Block gehören, werden aus den Positionsmarkierungen von allen Zielaktivitäten gelöscht. Wenn danach die Positionsmarkierung von Zielaktivitäten keine Tupel

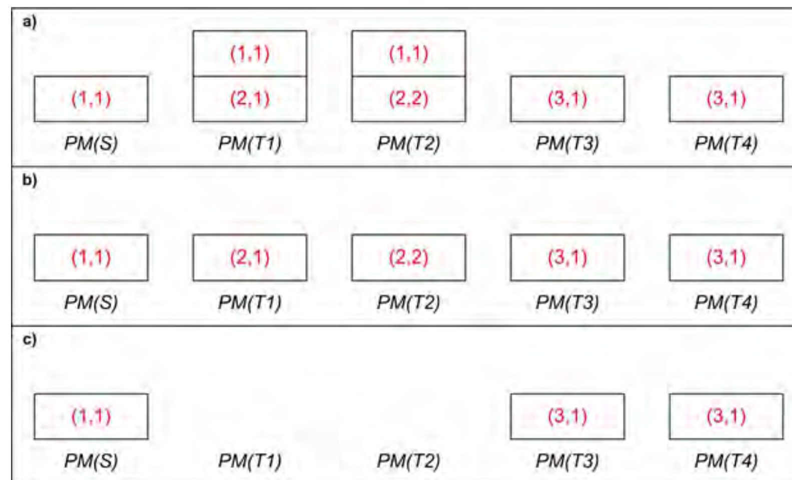


Abbildung 6.13: Positionsmarkierungen bei der Verifikation in Abbildung 6.12

mehr enthalten, bedeutet das, dass in jeder Ausführungsspur, in der die Quellaktivität vorkommt mindestens eine von diesen Zielaktivitäten mit jetzt leerer Positionsmarkierung vorkommt und das strukturelle Kriterium $S \bullet\!<\!> [T_1 \mid T_2 \mid \ldots]$ erfüllt ist. Gibt es keine der Zielaktivitäten mit leerer Positionsmarkierung, wird die nächste Iteration der Schleife ausgeführt.

Das strukturelle Kriterium $S \bullet\!<\!> [T1 \mid T2 \mid T3 \mid T4]$ soll für den Prozess in Abbildung 6.12 verifiziert werden. Dazu wird zuerst geprüft, ob die Quellaktivität überhaupt im Prozess vorkommt. Da diese vorkommt, werden als nächstes für die Quellaktivität und die Zielaktivitäten die Positionsmarkierungen bestimmt (s. Abbildung 6.13, a)).

Daraufhin werden alle Tupel aus den Positionsmarkierungen der Zielaktivitäten entfernt, die ebenfalls in der Positionsmarkierung der Quellaktivität vorkommen. Danach haben die Zielaktivitäten folgende Positionsmarkierungen wie in Abbildung 6.13, b). Das bedeutet, dass die Quellaktivität nicht die Ausführung einer bestimmten Zielaktivität impliziert, denn die Positionsmarkierungen aller Zielaktivitäten enthalten noch Tupel. Die Ausführung der Quellaktivität *S* schließt die Ausführung einer der Zielaktivitäten auch nicht aus, denn keine der Zielaktivitäten besitzt ein Tupel mit der gleichen Blocknummer und unterschiedlicher Zweignummer zu *S*. Es müssen also alle Zielaktivitäten weiter betrachtet werden.

Da alle Zielaktivitäten die gleiche Tupelanzahl besitzen, liegen sie, nachdem sie alle absteigend nach der Tupelanzahl ihrer Positionsmarkierungen sortiert wurden, in der Reihenfolge *T1, T2, T3, T4* vor. Zuerst wird deswegen der innerste XOR-Block *(C,D)* von *T1* untersucht. *T1* und *T2* versorgen die beiden Zweige des XOR-Blocks *(C,D)*, da die Block- und Zweignummer des jeweils verbliebenen und damit gleichzeitig obersten Tupels der Positionsmarkierung einen Zweig des betrachteten XOR-Blocks kennzeichnet. Damit der gesamte XOR-Block die Versorgung mit den Zielaktivitäten sicherstellt, muss er nicht weiter betrachtet werden. Deswegen können die zu dem XOR-Block gehörigen Tupel aus den Positionsmarkierungen der Zielaktivitäten gelöscht werden. Die Zielaktivitäten

besitzen dann die in Abbildung 6.13, Fall c) gezeigten Positionsmarkierungen. Die Positionsmarkierungen von *T1* und *T2* sind also leer, d.h., dass in jeder Ausführungsspur, in der *S* vorkommt ebenfalls entweder *T1* oder *T2* vorkommt. Folglich ist damit das strukturelle Kriterium $S \bullet <> [T1 \ / \ T2 \ / \ T3 \ / \ T4]$ erfüllt.

6.4 Verifikation von Reihenfolgebeziehungen

In den vorherigen Unterkapiteln 6.2 und 6.3 wurden bei der Verifikation der Integritätsregeln Reihenfolgebeziehungen nicht berücksichtigt. Das ist Gegenstand dieses Kapitels. Dabei läuft die Verifikation von Integritätsregeln, die Reihenfolgebeziehungen enthalten, ebenfalls in zwei Schritten ab.

Bei der Verifikation, ob der Bedingungsteil einer Integritätsregel mit Reihenfolgebeziehungen in keiner Ausführungsspur erfüllt ist, muss zusätzlich geprüft werden, ob eine der Reihenfolgebeziehungen eines Literals des Bedingungsteils zu einem anderen Literal des Bedingungsteils nicht erfüllt ist. Die Reihenfolgebeziehungen zwischen zwei Literalen werden dabei mit den ADEPT-Funktionen succ*(N) bzw. pred*(N) verifiziert (s. Unterkapitel 2.5). Ein Bedingungsteil gilt dabei in keiner Ausführungsspur als erfüllt, wenn eine Reihenfolgebeziehung zwischen zwei seiner Literale nicht erfüllt ist. In diesem Fall gilt die Integritätsregel als erfüllt.

Sind jedoch die Reihenfolgebeziehungen zwischen allen Literalen des Bedingungsteils erfüllt, muss die Integritätsregel auf strukturelle Kriterien abgebildet werden, um zu prüfen, ob der Bedingungsteil in keiner Ausführungsspur erfüllt ist. In dem Fall ist die Verifikation beendet und die Integritätsregel gilt ebenfalls als erfüllt.

Andernfalls werden die Integritätsregeln – wie im Verifikationsfall, wenn keine Reihenfolgebeziehungen definiert sind – auf strukturelle Kriterien abgebildet, um zu prüfen, ob der Folgeteil in jeder Ausführungsspur erfüllt ist, in der der Bedingungsteil erfüllt ist. Bevor allerdings die strukturellen Kriterien verifiziert werden, wird erst geprüft, für welche positiven Literale des Folgeteils mindestens eine Reihenfolgebeziehung verletzt ist, da die Literale des Bedingungsteils bereits verifiziert wurden. Die Literale, bei denen mindestens eine Reihenfolgebeziehung verletzt ist, können nicht zur Erfüllung des jeweiligen strukturellen Kriteriums beitragen. Deshalb muss in diesem Fall das strukturelle Kriterium, in dem dieses Literal vorkommt, entsprechend angepasst werden. Die positiven Literale des Folgeteils kommen dabei nur als Zielaktivität F bzw. F_i in den strukturellen Kriterien $S \bullet <> F$ bzw. $S \bullet <> [F_1 \ / \ F_2 \ / \ \ldots]$ vor. Das bedeutet, dass, wenn eine Reihenfolgebeziehung eines Literals F des Folgeteils verletzt ist, das in dem strukturellen Kriterium $S \bullet <> F$ vorkommt, $S \bullet <> F$ dadurch nicht erfüllt ist. Entsprechend gilt, wenn eine Reihenfolgebeziehung eines Literals F_1 des Folgeteils nicht erfüllt ist, das in dem strukturellen Kriterium $S \bullet <> [F_1 \ / \ F_2 \ / \ \ldots]$ vorkommt, F_1 nicht zur „Versorgung" von S verwendet werden kann. Damit muss das strukturelle Kriterium $S \bullet <> [F_1 \ / \ F_2 \ / \ \ldots]$ in $S \bullet <> [F_2 \ / \ \ldots]$ abgeändert werden.

Die so modifizierten strukturellen Kriterien werden im Anschluss wie im Fall der Verifi-

kation ohne Reihenfolgebeziehungen verifiziert. Damit sieht die automatische Abbildungsvorschrift folgendermaßen aus:

Automatische Abbildungsvorschrift für die Verifikation, ob eine Integritätsregel, die Reihenfolgebeziehungen enthalten kann, erfüllt ist

Nach Kapitel 5 ist jede zu verifizierende Integritätsregel von folgender Form:
$c = (\neg)B_1 \wedge \ldots \wedge (\neg)B_b \rightarrow \Big((\neg)F_{11} \vee \ldots \vee (\neg)F_{1n}\Big) \wedge \ldots \wedge \Big((\neg)F_{f1} \vee \ldots \vee (\neg)F_{fn}\Big),$
(b, in ≥ 1) mit beliebigen Reihenfolgebeziehungen zwischen positiven Literalen

Die Schritte 1 bis 3 sind identisch zum Fall der Verifikation von Integritätsregeln ohne Reihenfolgebeziehungen.

Schritt 4: Abbildung auf strukturelle Kriterien für die Prüfung des Bedingungsteils

if (Reihenfolgebeziehung zwischen zwei Literalen B_i von c_i" des Bedingungsteils ist nicht erfüllt) **then**

Der Bedingungsteil ist in keiner Ausführungsspur erfüllt. Die Integritätsregel ist damit erfüllt.

else

if (Bedingungsteil von c_i" enthält kein positives Literal B_i) **then**

Die Integritätsregel c_i" ist in jeder Ausführungsspur erfüllt.

else if (Bedingungsteil von c_i" enthält genau ein positives Literal B_1) **then**

Die Integritätsregel c_i" ist in keiner Ausführungsspur erfüllt, wenn die Aktivität B_1 nicht im Prozess enthalten ist.

else if (Bedingungsteil von c_i" enthält mehr als ein positives Literal B_i, wobei p die Anzahl der positiven Literale B_i ist) **then**

Die Integritätsregel c_i" ist in keiner Ausführungsspur erfüllt, wenn es mindestens ein beliebiges Paar *(i,j)* $(1 \leq i,j \leq p)$ gibt, so dass das strukturelle Kriterium $B_i \not< \bullet \not> B_j$ erfüllt ist.

end if

end if

Falls die Bedingung einer Integritätsregel c_i" in keiner Ausführungsspur erfüllt ist, gilt die Integritätsregel als erfüllt. Damit endet die Verifikation für c_i" hier. Ansonsten wird für jede Integritätsregel c_i" normal mit dem nächsten Schritt fortgefahren.

Der Schritt 5 ist wieder identisch zum Fall der Verifikation von Integritätsregeln ohne Reihenfolgebeziehungen.

Schritt 6: Abbildung auf strukturelle Kriterien für die Prüfung des Folgeteils

Die Integritätsregeln können jetzt direkt auf die entsprechenden strukturellen Kriterien abgebildet werden:

if (Folgeteil von $c_{i,j}$” enthält nur das negierte Literal $\neg F_1$) **then**

if (Bedingungsteil = B_1) **then**

Die Integritätsregel $c_{i,j}$” wird auf das strukturelle Kriterium $B_1 \not<\bullet\not> F_1$ abgebildet.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel $c_{i,j}$” wird auf das strukturelle Kriterium $START \not<\bullet\not> F_1$ abgebildet.

end if

else if (Folgeteil von $c_{i,j}$” enthält nur das positive Literal F_1) **then**

if (mindestens eine Reihenfolgebeziehung von F_1 ist nicht erfüllt) **then**

Die Integritätsregel $c_{i,j}$” wird auf false abgebildet.

else

if (Bedingungsteil = B_1) **then**

Die Integritätsregel $c_{i,j}$” wird auf das strukturelle Kriterium $B_1 \bullet<> F_1$ abgebildet.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel $c_{i,j}$”wird auf das strukturelle Kriterium $START \bullet<> F_1$ abgebildet.

end if

end if

else if (Folgeteil von $c_{i,j}$” enthält *f (f > 1)* positive Literale F_i) **then**

if (alle Reihenfolgebeziehungen von mehr als einem F_i sind erfüllt) **then**

if (Bedingungsteil = B_1) **then**

Die Integritätsregel $c_{i,j}$” wird auf das strukturelle Kriterium $B_1 \bullet<>$ *[* F_1 */ ... /* F_e *]*, (e <= f) abgebildet, wobei für alle F_i alle Reihenfolgebeziehungen erfüllt sind.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel $c_{i,j}$” wird auf das strukturelle Kriterium $START \bullet<>$ *[* F_1 */ ... /* F_e *]*, (e <= f) abgebildet, wobei für alle F_i alle Reihenfolgebeziehungen erfüllt sind.

end if

else if (alle Reihenfolgebeziehungen von genau einem F_i sind erfüllt) **then**

if (Bedingungsteil = B_1) **then**

Die Integritätsregel $c_{i,j}$” wird auf das strukturelle Kriterium $B_1 \bullet<> F_i$ abgebildet.

else if (Bedingungsteil = leer) **then**

Die Integritätsregel $c_{i,j}$" wird auf das strukturelle Kriterium $START \bullet\!<> F_i$ abgebildet.

end if

else if (alle Reihenfolgebeziehungen von keinem F_i sind erfüllt) **then**

Die Integritätsregel $c_{i,j}$" wird auf *false* abgebildet.

end if

end if

Damit ist die ursprüngliche Integritätsregel c erfüllt, wenn für jede Integritätsregel c_i" entweder der Bedingungsteil in keiner Ausführungsspur erfüllt ist oder der Folgeteil in jeder Ausführungsspur erfüllt ist, in der der Bedingungsteil erfüllt ist. Das bedeutet, dass mindestens eines der strukturellen Kriterien, auf das c_i" abgebildet wurde, erfüllt sein muss.

Nachfolgend wird gezeigt, wie mit der automatischen Abbildungsvorschrift Integritätsregeln verifiziert werden können, die Reihenfolgebeziehungen enthalten.

Beispiel einer Verifikation mit Hilfe der automatischen Abbildungsvorschrift

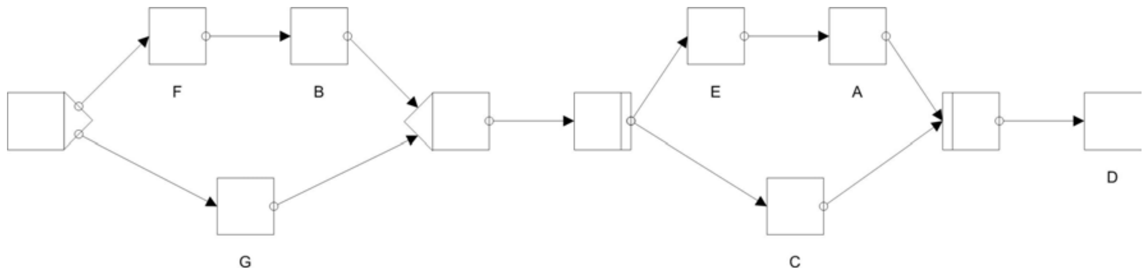


Abbildung 6.14: Integritätsregel $c = A \wedge B \rightarrow (C \vee D) \wedge (E \vee F)$, $[A < B,\ A > C,\ A < D,\ B < F]$ ist erfüllt.

An dem Prozess in Abbildung 6.14 und der Integritätsregel $c = A \wedge B \rightarrow (C \vee D) \wedge (E \vee F)$, $[A > B,\ A > C,\ A < D,\ B < F]$ soll die Verifikation mit Reihenfolgebeziehungen demonstriert werden.

Da c im Folgeteil einen Konjunktionsoperator enthält, wird c in *Schritt 1* entlang diesem in die beiden Integritätsregeln $c_1 = A \wedge B \rightarrow C \vee D$ und $c_2 = A \wedge B \rightarrow E \vee F$ zerlegt.

Schritt 2 bzw. *Schritt 3* fallen hier weg, da der Bedingungs- bzw. Folgeteil beider Integritätsregeln c_1 und c_2 jeweils keine negierten Literale enthält.

Deshalb kann direkt in *Schritt 4* geprüft werden, ob der Bedingungsteil von c_1 bzw. c_2 in keiner Ausführungsspur erfüllt ist. Dazu werden zuerst die Reihenfolgebeziehungen zwischen den Literalen des Bedingungsteils verifiziert. Die einzige Reihenfolgebeziehung dieser Art ist hier erfüllt, da A Nachfolger von B ist. Deshalb muss das strukturelle Kriterium $A \not<\!\bullet\!\not> B$ noch geprüft werden. Dieses ist ebenfalls erfüllt, da A und B auf verschiedenen

Zweigen eines XOR-Blocks liegen. Somit ist der Bedingungsteil in mindestens einer Ausführungsspur erfüllt und es muss im Anschluss geprüft werden, ob der Folgeteil in jeder Ausführungsspur erfüllt ist, in der der Bedingungsteil erfüllt ist.

Dazu werden in *Schritt 5* jeweils c_1 und $c_2$entlang dem Konjunktionsoperator aufgespalten, so dass die zu verifizierende Integritätsregel c erfüllt ist, wenn $c_{1,1} = A \rightarrow C \vee D$ oder $c_{1,2} = B \rightarrow C \vee D)$ erfüllt ist und wenn $c_{2,1} = A \rightarrow E \vee F$ oder $c_{2,1} = B \rightarrow E \vee F$ erfüllt ist.

In *Schritt 6* werden die Integritätsregeln $c_{i,j}$ auf strukturelle Kriterien abgebildet, nachdem alle Reihenfolgebeziehungen der Literale des Folgeteils verifiziert wurden. Da die Reihenfolgebeziehungen $B < F$ und $A > C$ verletzt sind, können die Aktivitäten F und C nicht zur Versorgung der Aktivitäten A bzw. B beitragen. Deshalb wird die Integritätsregel $c_{1,1}$ auf das strukturelle Kriterium $A \bullet\!<> D$, $c_{1,2}$ auf das strukturelle Kriterium $B \bullet\!<> D$, $c_{2,1}$ auf $A \bullet\!<> E$ und $c_{2,2}$ auf $B \bullet\!<> E$ abgebildet. Die strukturellen Kriterien $A \bullet\!<> D$ und $A \bullet\!<> E$ sind jeweils erfüllt. Deshalb ist auch die zu verifizierende Integritätsregel c auf diesem Prozess erfüllt.

6.5 Zusammenfassung und Diskussion

Mit den hier verifizierten Integritätsregeln wurde eine Basis geschaffen für die graphbasierte semantische Verifikation von Prozessen. Dabei wurde die Idee verfolgt, die Integritätsregeln auf strukturelle Kriterien abzubilden und diese im Anschluss zu verifizieren.

Da es zwei verschiedene Fälle gibt, in denen eine Integritätsregel erfüllt sein kann, wurde die Verifikation entsprechend in zwei Schritten durchgeführt. Zuerst wurde geprüft, ob der Bedingungsteil der Integritätsregel in keiner Ausführungsspur erfüllt ist. Falls dieser nämlich in keiner Ausführungsspur erfüllt ist, gilt die Integritätsregel als erfüllt. Andernfalls muss geprüft werden, ob in jeder Ausführungsspur, in der der Bedingungsteil erfüllt ist, ebenfalls der Folgeteil erfüllt ist.

Neben den Basis-Integritätsregeln wurden auch Integritätsregeln verifiziert, die zusätzlich Reihenfolgebeziehungen zwischen Aktivitäten enthalten haben. Dabei wurde der gleiche Verifikationsansatz wie bei den Basis-Integritätsregeln verfolgt mit dem Unterschied, dass vor der Verifikation der strukturellen Kriterien geprüft wurde, von welchen Aktivitäten Reihenfolgebeziehungen verletzt sind und entsprechend die strukturellen Kriterien angepasst wurden.

Für die Verifikation der strukturellen Kriterien wurden – vom strukturellen Kriterium $S \nless\bullet\ngtr T$ abgesehen – zwei verschiedene Ansätze verfolgt. Zum einen wurde der WriterExists-Algorithmus angepasst und zum anderen wurden Positionsmarkierungen verwendet, wobei das strukturelle Kriterium $S \nless\bullet\ngtr T$ ebenfalls auf Basis von Positionsmarkierungen verifiziert werden könnte. Es müsste nur geprüft werden, ob die Positionsmarkierungen von S und T jeweils ein Tupel mit gleicher Block- und verschiedener Zweignummer enthalten. Die beiden auf Positionsmarkierungen basierenden Algorithmen sind dabei aufwendiger, wenn die Positionsmarkierungen bei der Verifikation jeder Integritätsregel neu

bestimmt werden müssen. Deshalb wäre es besser, wenn entweder alle Integritätsregeln auf einmal verifiziert werden könnten oder die Positionsmarkierungen vom Prozess-Meta-Modell, z.B. in Form von Knoten-IDs, bereitgestellt werden könnten. Falls die Positionsmarkierungen von diesem bereitgestellt werden, sind die darauf basierenden Algorithmen LabelledTargetExists1 und LabelledTargetExists2 auch effizienter als die beiden auf dem WriterExists-Algorithmus basierenden TargetExists1- und TargetExists2-Algorithmen. In dem Fall müßte das Prozess-Meta-Modell bei Prozessänderungen allerdings auch die Positionsmarkierungen pflegen.

Kapitel 7

Effizienzaspekte

Das vorliegende Kapitel behandelt im Wesentlichen Optimierungierungsmöglichkeiten für die Verifikation der Integritätsregeln. Dabei werden zwei verschiedene Wege beschritten.

Zum einen soll die Menge der zu verifizierenden Integritätsregeln eingeschränkt werden, zum anderen soll die Verifikation der einzelnen strukturellen Kriterien optimiert werden.

Die Idee bei der Eingrenzung der zu verifizierenden Integritätsregeln besteht darin, dass nach einer Prozessänderung meistens nur wenige Integritätsregeln verletzt sein können, da von einer Prozessänderung meistens nur ein kleiner Prozessausschnitt betroffen ist [LRD06b]. Wenn z.B. im Prozess eine Aktivität C gelöscht wird, die in der Integritätsregel $A \wedge \neg B \rightarrow C \vee D$ vorkommt, kann diese Integritätsregel dadurch verletzt sein, muss jedoch nicht. Das hängt davon ab, ob der Bedingungsteil in mindestens einer Ausführungsspur erfüllt ist und falls dieser in mindestens einer Ausführungsspur erfüllt ist, ob D in jeder Ausführungsspur vorkommt, in der der Bedingungsteil erfüllt ist. Es muss also zuerst der Bedingungsteil geprüft werden. Dafür stellt der Algorithmus im Unterkapitel 7.1 eine Optimierung dar.

In Unterkapitel 7.2 wird versucht, die Verifikation der strukturellen Kriterien effizienter zu gestalten, indem das Prozess-Meta-Modell dafür optimiert wird.

Im Anschluss behandelt 7.3, wie sich die verschiedenen Bestandteile von Integritätsregeln auf ihren Verifikationsaufwand auswirken. Daraus soll abgeleitet werden, wie Integritätsregeln am besten formuliert werden, damit ihre Verifikation so effizient wie möglich abläuft.

7.1 Eingrenzung der zu verifizierenden Integritätsregeln

Das Prinzip besteht darin, mit einem Rete-Netzwerk die Menge der Integritätsregeln anzunähern, deren Bedingungsteile in mindestens einer Ausführungsspur potentiell erfüllt sein können (s. Anhang A.1, Algorithmus 6). Dazu wird geprüft, ob jedes Literal des Bedingungsteils einer Integritätsregel im Prozess vorkommt. Das bedeutet, es wird nur geprüft, ob jedes Literal des Bedingungsteils in einer beliebigen Ausführungsspur vorkommt, aber nicht, ob alle Literale des Bedingungsteils zusammen in einer Ausführungsspur vorkommen. So gilt z.B. der Bedingungsteil $A \wedge B$ einer Integritätsregel in einer Spur eines

Prozesses p als erfüllt, wenn in p A und B auf verschiedenen Zweigen eines XOR-Blocks liegen. Allerdings ist der Bedingungsteil tatsächlich in keiner Ausführungsspur erfüllt, da in keiner Ausführungsspur A und B zusammen vorkommen können. Falls deswegen das Rete-Netzwerk liefert, dass der Bedingungsteil einer Integritätsregel erfüllt ist, muss noch mit dem strukturellen Kriterium $S \not\prec\bullet\not\succ T$ geprüft werden, ob die Ausführung einer Aktivität des Bedingungsteils die Ausführung einer anderen Aktivität des Bedingungsteils ausschließt. Um nicht bei jeder Überprüfung eines Bedingungsteils die strukturellen Kriterien neu bestimmen zu müssen, könnten diese in einem Cache abgelegt werden. Bevor ein strukturelles Kriterium bestimmt wird, könnte dann dort zuerst nachgeschaut werden, ob dieses bereits bestimmt wurde.

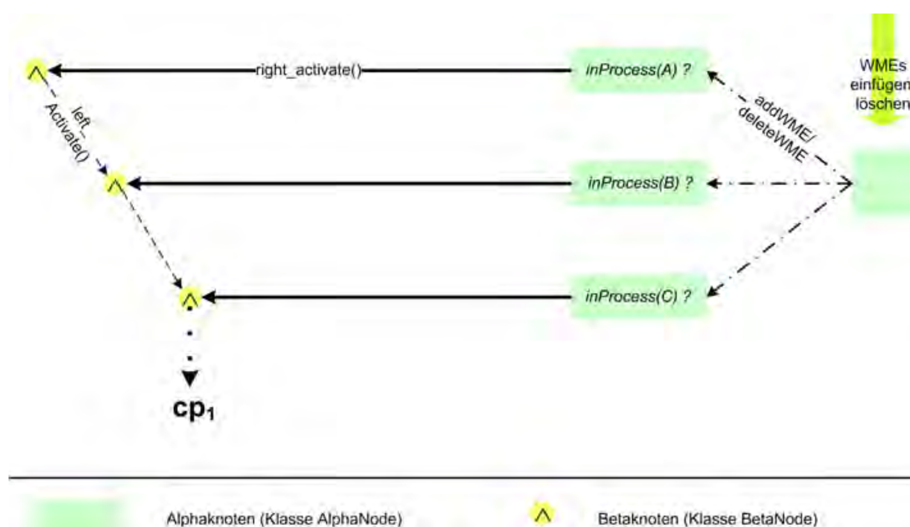


Abbildung 7.1: Rete-Netz für den Bedingungsteil $\mathrm{cp}_1 = \mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}$

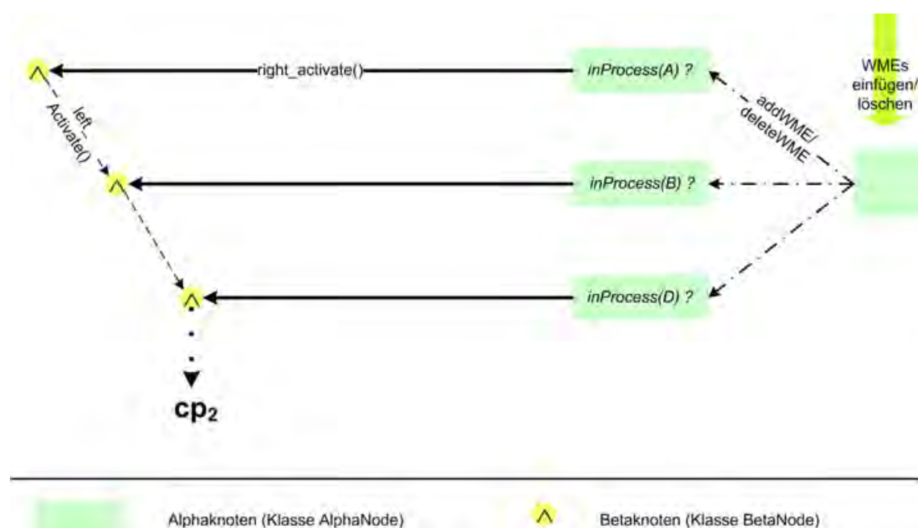


Abbildung 7.2: Rete-Netz für den Bedingungsteil $cp_2 = A \wedge B \wedge D$

Zur Annäherung, ob der Bedingungsteil einer Integritätsregel in mindestens einer Ausführungsspur erfüllt ist, wird ein normales Rete-Netz verwendet. Dieses besitzt allerdings

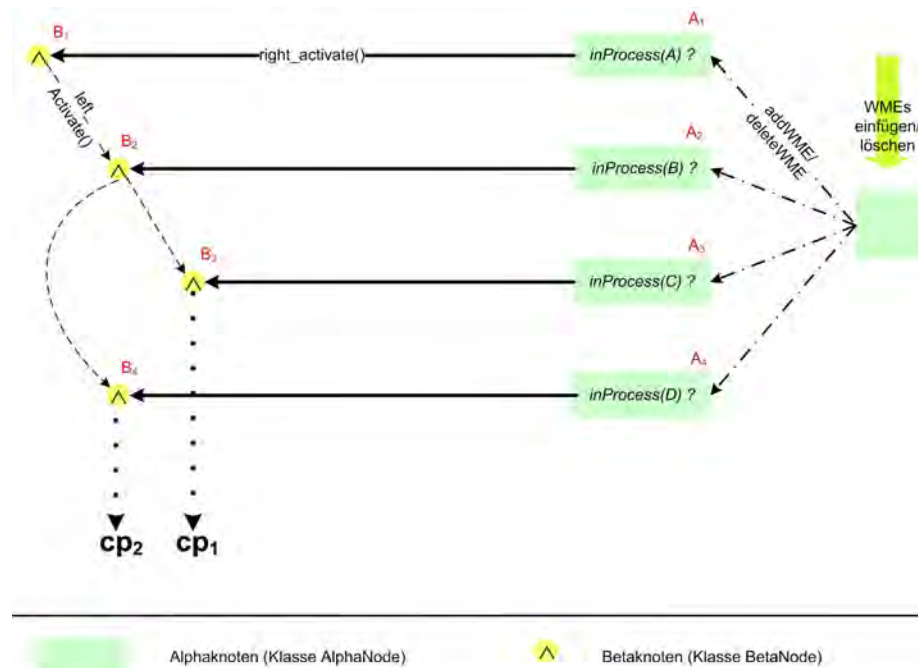


Abbildung 7.3: Rete-Netzwerk für die Bedingungsteile $\mathrm{cp}_1 = \mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}$ und $\mathrm{cp}_2 = \mathrm{A} \wedge \mathrm{B} \wedge \mathrm{D}$

keine Memory-Knoten, da die einzelnen Bedingungen der Alpha- und Betaknoten keine Variablen enthalten, an die Elemente bzw. Tupel des Working Memory gebunden werden und auf die in anderen Bedingungen zugegriffen werden muss. Damit sind die einzelnen Bedingungen unabhängig voneinander und es muss nicht gespeichert werden, für welche Elemente bzw. Tupel des Working Memory eine Bedingung erfüllt ist. Wenn eine Bedingung eines Alpha- bzw. Betaknoten erfüllt ist, wird deshalb nur der Alpha- bzw. Betaknoten als erfüllt markiert, indem dessen Flag *fulfilled* gesetzt wird.

Die Alphaknoten stellen dabei die Bedingung *inProcess(A)* dar, die besagt, dass eine bestimmte Aktivität A im Prozess vorkommen muss. Die Bedingung eines Betaknoten ist erfüllt, wenn die Bedingungen der beiden Vorgängerknoten erfüllt sind. Das *Working Memory* enthält für jede Aktivität A des Prozesses ein Element *inProcess(A)*.

Das Rete-Netzwerk ist so aufgebaut, dass ein Alpha- und ein Betaknoten immer der rechte bzw. linke Vorgängerknoten des nachfolgenden Betaknoten darstellen. Der erste Betaknoten hat allerdings nur einen Alphaknoten und keinen Betaknoten als Vorgänger. Der letzte Betaknoten wiederum stellt den Terminalknoten dar und besitzt deswegen keinen Nachfolger. Das Rete-Netz in Abb. 7.1 bzw. 7.2 stellt dabei den Bedingungsteil $cp_1 = A \wedge B \wedge C$ bzw. $cp_2 = A \wedge B \wedge D$ einer Integritätsregel dar. Wie man sieht, ist der Anfangsteil der Rete-Netze für cp_1 und cp_2 gleich. Wenn die beiden Rete-Netze zu einem Rete-Netzwerk verbunden werden, wird deshalb dieses Anfangsnetz nur einmal übernommen. Der letzte Knoten dieses Netzes hat dann zwei Nachfolger, da die jeweils letzte Bedingung der beiden Bedingungsteile unterschiedlich ist (s. Abb. 7.3). Wenn das Rete-Netzwerk mit Fakten gefüllt wird, werden die nachfolgenden Betaknoten eines Kno-

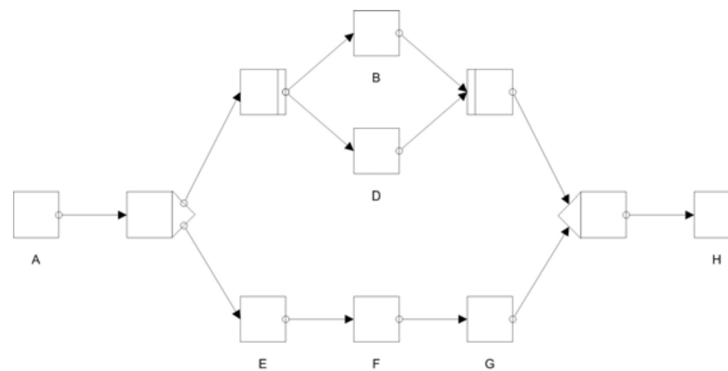


Abbildung 7.4: Prozess, auf dem das *Working Memory* für das Rete-Netzwerk in Abb. 7.3 basiert

ten K aktiviert, im Falle, dass die Bedingung von K erfüllt ist und dieser somit als erfüllt markiert ist. Nachdem ein Betaknoten aktiviert wurde, prüft dieser, ob seine eigene Bedingung erfüllt ist. Dazu müssen die Bedingungen seiner beiden Vorgängerknoten erfüllt sein. Damit ein Betaknoten unterscheiden kann, von welchem der Elternknoten die Aktivierung ausging, rufen die Alphaknoten bei ihren nachfolgenden Betaknoten die Methode *right_ activate()* auf, während die Betaknoten bei ihren nachfolgenden Betaknoten die Methode *left_ activate()* aufrufen. Dadurch weiß der nachfolgende Betaknoten, von welchem seiner beiden Vorgängerknoten die Bedingung erfüllt ist und muss nur abfragen, ob auch die Bedingung des jeweils anderen Elternkoten erfüllt ist.

Um das zu veranschaulichen soll mit dem Rete-Netzwerk in Abbildung 7.3 geprüft werden, ob die Bedingungsteile $cp_1 = A \wedge B \wedge C$ und $cp_2 = A \wedge B \wedge D$ auf dem Prozess in Abbildung 7.4 in mindestens einer Ausführungsspur erfüllt sind. Zuerst werden den Alphaknoten sukzessive alle Aktivitäten des Prozesses präsentiert, indem ihre Methode *addWME()* aufgerufen wird. Durch das Präsentieren des WME *inProcess(A)* wird die Bedingung des Alphaknoten A_1 erfüllt. Dieser wird somit als erfüllt markiert (*fulfilled* = *true*) und aktiviert seinen nachfolgenden Betaknoten B_1, indem er dessen Methode *right_ activate()* aufruft. Da dieser nur A_1 als Vorgängerknoten hat, wird auch er als erfüllt markiert. Somit ruft B_1 wiederum seinen nachfolgenden Betaknoten B_2 mit der Methode *left_ activate()* auf. Dieser prüft dann, ob die Bedingung seines rechten Vorgängerknoten A_2 ebenfalls erfüllt ist. Allerdings ist A_2 nicht als erfüllt markiert, somit ist die Bedingung von B_2 nicht erfüllt und dieser aktiviert nicht seinen Nachfolgerknoten. Wenn das Element *inProcess(B)* dem Rete-Netzwerk präsentiert wird, wird die Bedingung von A_2 erfüllt. A_2 wird als erfüllt markiert und ruft wiederum die Methode *right_ activate()* von B_2 auf. Jetzt sind die Bedingungen der beiden Vorgängerknoten von B_2 erfüllt. Folglich ist dieser ebenfalls erfüllt und aktiviert seine nachfolgenden Betaknoten B_3 und B_4. Diese können allerdings nicht markiert werden, da jeweils die Bedingung *inProcess(C)* bzw. *inProcess(D)* ihrer rechten Vorgängerknoten A_3 bzw. A_4 nicht erfüllt ist. Beim Präsentieren der WMEs von Aktivitäten, die zu keiner Bedingung eines Alphaknoten passen, bleibt das Rete-Netzwerk unverändert. Da D ebenfalls im Prozess vorhanden ist, wird bei der

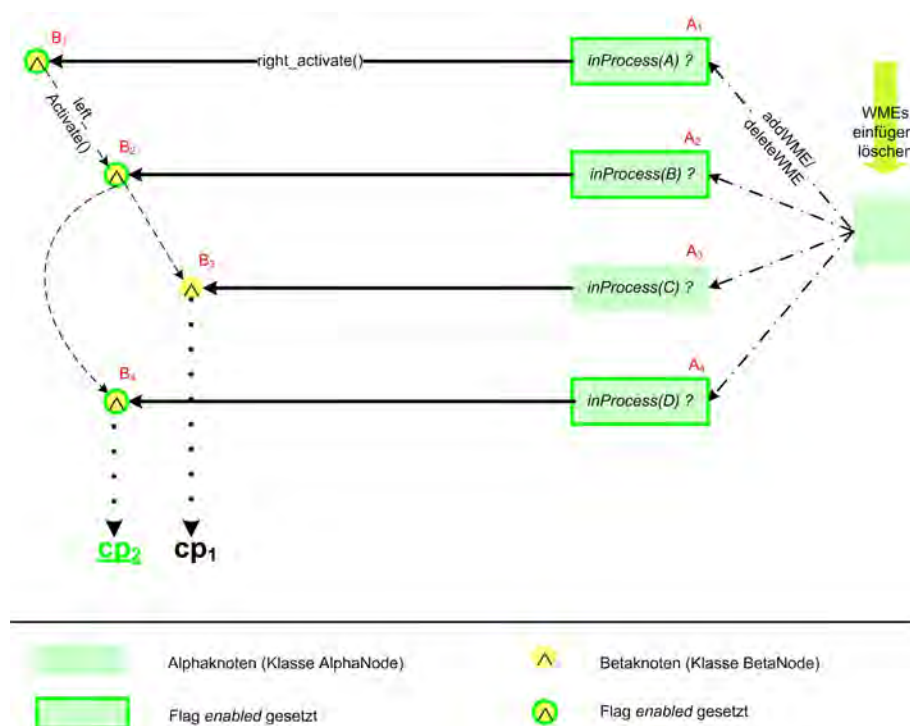


Abbildung 7.5: Bestimmung der Integritätsregeln, deren Bedingungsteil in mindestens einer Spur erfüllt ist, wenn der Prozess in Abb. 7.4 als *Working Memory* dient

Präsentation des entsprechenden WME *inProcess(D)* die Bedingung des Alphaknoten A_4 erfüllt und die Methode *right_activate()* des Betaknoten B_4 aufgerufen. Beide Vorgänger sind dadurch erfüllt und somit kann B_4 auch als erfüllt markiert werden. B_4 bildet für den Bedingungsteil cp_2 den Terminalknoten. Daraus folgt, dass alle drei Aktivitäten A, B und D des Bedingungsteils cp_2 jeweils in mindestens einer Ausführungsspur vorkommen (s. Abbildung 7.5).

A, B und D können zwar in einer Ausführungsspur zusammen vorkommen, müssen jedoch nicht. Deshalb muss noch mit den strukturellen Kriterien $A \not\prec\bullet\not\succ B$, $A \not\prec\bullet\not\succ D$ und $B \not\prec\bullet\not\succ D$ geprüft werden, ob zwei der Aktivitäten des Bedingungsteils sich gegenseitig ausschließen. Wie am Prozess in Abbildung 7.4 zu sehen ist, liegen B und D auf verschiedenen Zweigen eines XOR-Blocks. Das strukturelle Kriterium $B \not\prec\bullet\not\succ D$ ist also erfüllt und der Bedingungsteil cp_2 in keiner Ausführungsspur erfüllt.

Ein Vorteil des Rete-Algorithmus ist, dass bei einer Änderung des *Working Memory* nicht mehr alle Fakten ausgewertet werden müssen, da die Fakten, die die Bedingung eines Knoten erfüllen, bei diesem gespeichert werden. Das bedeutet für diesen Ansatz, dass beim Einfügen der Aktivität C in den Prozess (s. Abbildung 7.6), nur das WME *inProcess(C)* dem Rete-Netzwerk präsentiert werden muss. Dadurch wird die Bedingung *inProcess(C)* des Alphaknoten A_3 erfüllt. A_3 wiederum aktiviert den Betaknoten B_3. Folglich sind die Bedingungen beider Vorgängerknoten von B_3 erfüllt und B_3 kann ebenfalls als erfüllt markiert werden. Da B_3 den Terminalknoten für den Bedingungsteil $cp_1 = A \wedge B \wedge C$ darstellt, liefert der Rete-Algorithmus zurück, dass beide Bedingungsteile $cp_1 = A \wedge B \wedge C$ und

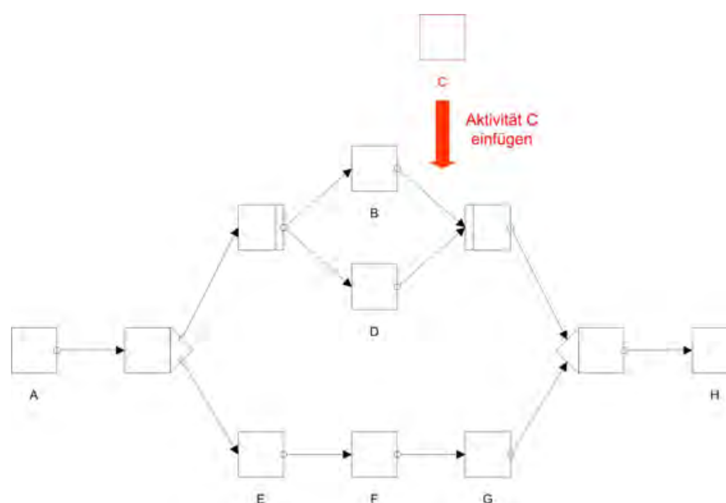


Abbildung 7.6: Änderung des *Working Memory* durch Einfügen der Aktivität C in den Prozess

$cp_2 = A \wedge B \wedge D$ potentiell erfüllt sind (s. Abbildung 7.7).

In diesem Fall muss wieder nachgeprüft werden, ob die Aktivitäten der Bedingungsteile cp_1 und cp_2 in einer Ausführungsspur zusammen vorkommen. Für cp_1 müssen deswegen die strukturellen Kriterien $A \not\prec\bullet\not\succ B$, $A \not\prec\bullet\not\succ C$ und $B \not\prec\bullet\not\succ C$ ausgewertet werden. Da $A \not\prec\bullet\not\succ B$ bereits für cp_2 im letzten Durchgang des Algorithmus berechnet wurde, liegt es deshalb im Cache und muss nicht mehr neu bestimmt werden. $A \not\prec\bullet\not\succ C$ und $B \not\prec\bullet\not\succ C$ liegen nicht im Cache und müssen deswegen neu bestimmt werden. Alle drei strukturellen Kriterien liefern *true* zurück. Demnach ist der Bedingungsteil cp_1 in mindestens einer Ausführungsspur erfüllt.

Ob sich zwei Aktivitäten gegenseitig ausschließen, muss ebenfalls noch für cp_2 geprüft werden. In diesem Durchgang liegen alle zu bestimmenden strukturellen Kriterien bereits im Cache, so dass schneller zum Ergebnis gelangt wird, dass der Bedingungsteil cp_2 in keiner Ausführungsspur erfüllt ist.

Ein anderer Ansatz basierend auf dem Rete-Algorithmus besteht darin, dem Rete-Netzwerk alle strukturellen Kriterien der Form $S \not\prec\bullet\not\succ T$ zu präsentieren, die zwischen allen Aktivitäten des Prozesses gelten. Dabei muss nicht für jedes beliebige Paar von Aktivitäten geprüft werden, ob sie sich gegenseitig ausschließen, da alle Aktivitäten auf einem Zweig eines XOR-Blocks die gleichen strukturellen Kriterien zu den anderen Aktivitäten des Prozesses haben. Das Rete-Netzwerk ist dabei gleich wie bei dem ersten Ansatz aufgebaut. Die Knoten haben nur unterschiedliche Bedingungen. Die Alphaknoten testen auf das strukturelle Kriterium $S \not\prec\bullet\not\succ T$, während die Bedingung eines Betaknoten erfüllt ist, wenn die Bedingung von einem der beiden Vorgängerknoten erfüllt ist. Folglich wird im Gegensatz zum ersten Ansatz genau geprüft, ob der Bedingungsteil einer Integritätsregel in keiner Ausführungsspur erfüllt ist. Der Nachteil ist allerdings, dass der Ansatz nicht besonders effizient ist, da er sämtliche strukturelle Kriterien $S \not\prec\bullet\not\succ T$ jeder Aktivität zu jeder anderen Aktivität im Prozess als Eingabe bekommt.

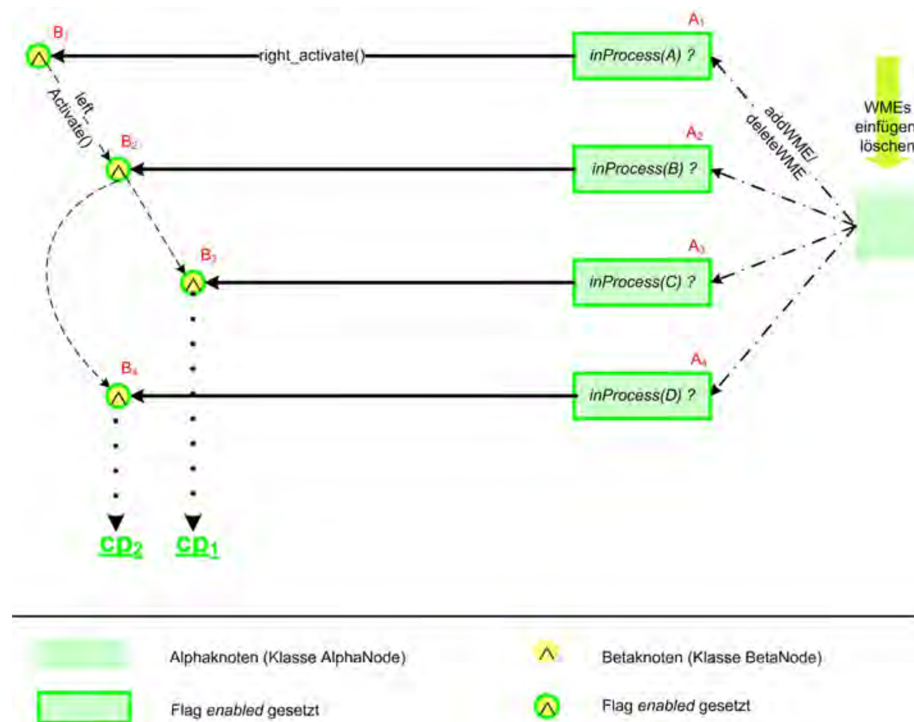


Abbildung 7.7: Rete-Netzwerk nach der Änderung des *Working Memory*

7.2 Optimierungen der Algorithmen für strukturelle Kriterien

Nachdem erläutert wurde, wie die Verifikation von Integritätsregeln dadurch optimiert werden kann, dass der Prüfaufwand zur Bestimmung der zu verifizierenden Integritätsregeln verringert wird, wird hier die Frage behandelt, wie die Verifikation von Integritätsregeln sich dadurch optimieren lässt, dass die Verifikation eines strukturellen Kriteriums effizienter gemacht wird. Dabei wird versucht, die Optimierung der Verifikationsalgorithmen dadurch zu erzielen, dass die Infrastruktur des Prozess-Meta-Modells erweitert wird.

Die beiden auf den WriterExists-Algorithmus basierenden Algorithmen TargetExists1 und TargetExists2 lassen sich auf diese Weise optimieren, indem der Prozess mit Weginformationen ausgestattet wird. So lassen sich bestimmte Blöcke beim Traversieren des Prozess komplett überspringen, falls die Quellaktivität nicht in ihnen liegt. Damit ist die Prozesstraversierung nicht mehr eine uninformierte Breitensuche nach der Quellaktivität. Mit den bestehenden Prozess-Meta-Modellen lassen sich allerdings diese Routinginformationen nicht festhalten. Eine mögliche Erweiterung eines Prozess-Meta-Modells besteht darin, in der ID jedes Knotens alle Blöcke, in denen er sich befindet, wie in Abbildung 7.8 zu speichern. Die Knoten-ID könnte so aufgebaut sein, dass die ID jedes Blocks von innen nach außen durch einen Punkt getrennt angegeben wird und danach durch einen Doppelpunkt getrennt die bisherige Knoten-ID folgt. Nach dieser Konvention hätte z.B. die Aktivität T die Knoten-ID *1:x*, B die Knoten-ID *3.2:y* und S die Knoten-ID *5:z*, wobei x, y und z jeweils für die alten IDs der Knoten stehen. Die Verifikation des strukturellen Kriteriums

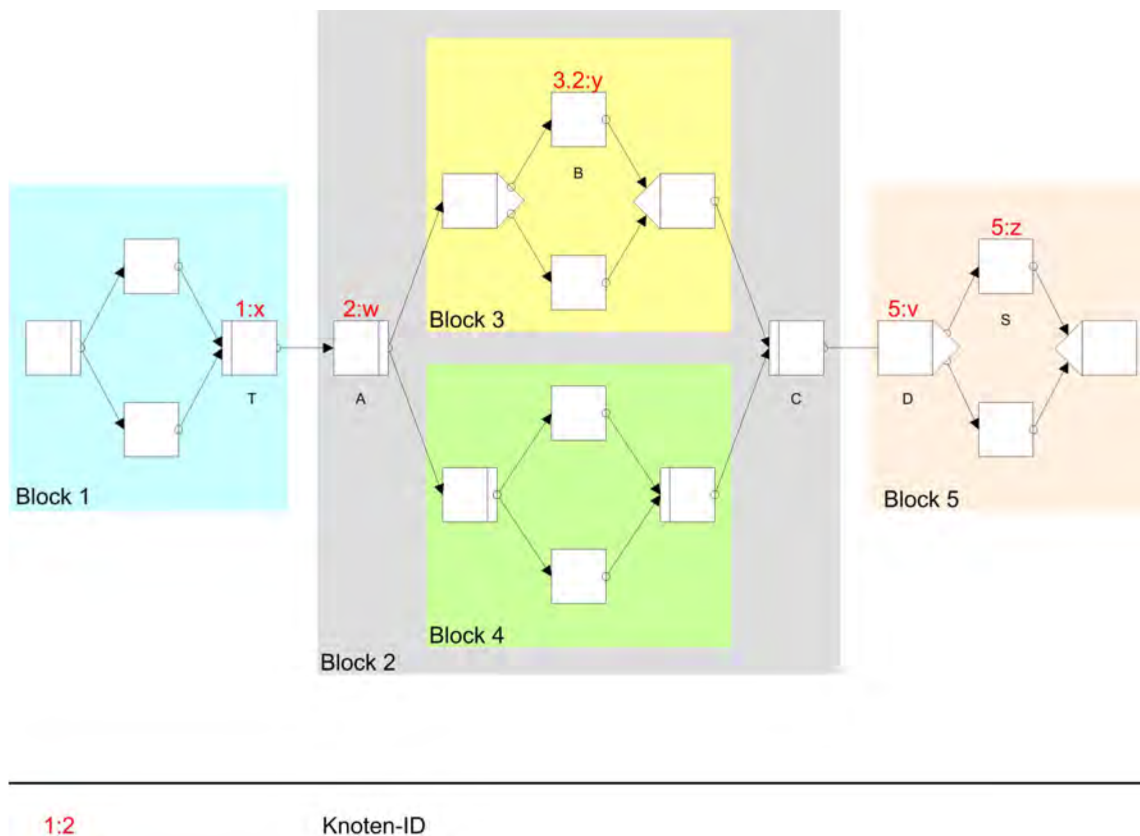


Abbildung 7.8: Optimierung des Prozessmetamodells durch besondere Knoten-IDs

$S \bullet <> T$ mit dem TargetExists1-Algorithmus läuft mit dieser Optimierung des Prozess-Meta-Modells folgendermaßen ab: Zuerst wird die Knoten-ID von S bestimmt. Aus dieser folgt, dass S nur in dem Block mit der ID *5* liegen muss. Anschließend wird der Nachfolgerknoten A von T besucht. Der AND-Split-Knoten A besitzt die ID *2:w*, so kann mit der ADEPT-Funktion *join()* zum Join-Knoten C des Blocks gesprungen werden, da bekannt ist, dass S nur im Block *5* liegt. Danach wird der Nachfolger D von C besucht. Dieser hat die Knoten-ID *5:v* und liegt somit im richtigen Block. Der Block muss dann normal Aktivität für Aktivität durchlaufen werden. Allerdings ist einer der beiden Nachfolger von D bereits die Quellaktivität und kein XOR-Join-Knoten, demnach kann die Quellaktivität markiert werden und der TargetExists1-Algorithmus liefert *true* zurück.

Um bei diesem Optimierungsansatz mit dem TargetExists2-Algorithmus die Aktivitäten in übersprungenen Blöcken nicht mehr auswerten zu müssen, da für die Ausführung dieser Blöcke die Versorgung mit den Zielaktivitäten bereits sichergestellt ist, müssen die Knoten dieser Blöcke als versorgt markiert werden. Dafür wird eine Funktion gebraucht, die alle Knoten eines Blocks anhand der Block-ID zurückliefert.

Eine Optimierung für die auf Positionsmarkierungen basierenden Algorithmen LabelledTargetExists1 und LabelledTargetExists2 besteht darin, die Positionsmarkierungen vom Prozess-Meta-Modell bereitstellen zu lassen. So muss nicht vor der Überprüfung jeder Integritätsregel der Prozess durchlaufen werden, um die Positionsmarkierungen zu bestimmen.

In diesem Fall sind die LabelledTargetExists1 und LabelledTargetExists2-Algorithmen wesentlich effizienter als die auf dem WriterExists-Algorithmus basierenden TargetExists1- und TargetExists2-Algorithmen. Falls das Prozess-Meta-Modell diese Positionsmarkierungen bereitstellen würde, müsste es allerdings diese auch bei Prozessänderungen mitpflegen.

7.3 Effizienzanalyse von Integritätsregeln

Hier soll untersucht werden, inwieweit sich die Integritätsregeln in verschiedene Klassen einteilen lassen, was den Aufwand ihrer Verifikation betrifft. Falls dies möglich ist, können daraus nützliche Rückschlüsse gezogen werden, wie Integritätsregeln so formuliert werden können, dass ihr Verifikationsaufwand möglichst gering wird. Dazu muss zum einen der Verifikationsaufwand für die einzelnen strukturellen Kriterien bestimmt werden und zum anderen, auf welche strukturellen Kriterien sich die verschiedene Klassen von Integritätsregeln abbilden lassen. Die Verifikation von $S \bullet <> [T_1 / \ldots / T_n]$ ist dabei am aufwendigsten, da hier der Prozessgraph im Normalfall am weitesten durchlaufen wird.

Im Folgenden wird untersucht, auf welche strukturelle Kriterien sich die verschiedenen Klassen von Integritätsregeln abbilden lassen, um ihren Verifikationsaufwand zu bestimmen. Wie in Unterkapitel 6.1 erläutert, kann eine Integritätsregel auf zwei verschiedene Möglichkeiten erfüllt sein. Zum einen kann die Integritätsregel dadurch erfüllt sein, dass ihr Bedingungsteil in keiner Ausführungsspur erfüllt ist und zum anderen kann eine Integritätsregel dadurch erfüllt sein, dass in jeder Ausführungsspur, in der ihr Bedingungsteil erfüllt ist ebenfalls ihr Folgeteil erfüllt ist.

Zuerst wird bestimmt, wie der Verifikationsaufwand zur Prüfung des Bedingungsteils von den einzelnen Elementen einer Integritätsregel abhängt. Den meisten Verifikationsaufwand verursachen dabei die negierten Literale des Bedingungsteils, da sie auf die Seite des Folgeteils gebracht werden müssen und dort Element einer Disjunktion sind. Dadurch werden sie auf das strukturelle Kriterium $S \bullet <> [T_1 / \ldots / T_n]$ abgebildet, was den größten Verifikationsaufwand verursacht. Allgemein nimmt der Verifikationsaufwand mit der Anzahl der Literale im Bedingungsteil zu. Bei mehreren positiven Literalen wird die Integritätsregel auf mehrere strukturelle Kriterien abgebildet und bei mehreren negierten Literalen enthält das strukturelle Kriterium $S \bullet <> [T_1 / \ldots / T_n]$, auf das die negierten Literale nach den Umformungen abgebildet werden, mehr Zielaktivitäten.

Der Aufwand der Verifikation, ob in jeder Ausführungsspur, in der der Bedingungsteil erfüllt ist, ebenfalls der Folgeteil erfüllt ist, ist bei Integritätsregeln mit mehreren Disjunktionen im Folgeteil am größten, da sie auf strukturelle Kriterien der Form $B_i \bullet <> [F_1 / \ldots / F_f]$ bzw. $START \bullet <> [F_1 / \ldots / F_f]$ abgebildet werden, je nachdem ob der Bedingungsteil Literale B_i enthält oder leer ist.

Zusammenfassend lässt sich sagen, dass der Verifikationsaufwand für Integritätsregeln, die mehrere negierte Literale im Bedingungsteil sowie mehrere Disjunktionen im Folgeteil enthalten, am größten ist.

Kapitel 8

Zusammenfassung und Ausblick

Im ersten Teil der Arbeit wurde das Prozess-Meta-Modell eingeführt, auf dem die graphbasierte semantische Prozessverifikation des zweiten Teils basiert. Ebenso wurde der Begriff Geschäftsregeln eingeführt und aktuelle Ansätze zur Integration von Geschäftsregeln in Prozess-Management-Systeme vorgestellt.

Im zweiten Teil, der sich mit der graphbasierten semantischen Prozessverifikation befasste, wurde zuerst die Art der zu verifizierenden Integritätsregeln vorgestellt. Dabei handelte es sich um Integritätsregeln, mit denen sich grundlegende Aussagen über erlaubte Ausführungskombinationen von Aktivitäten machen lassen. Diese lassen sich auch leicht zu komplexeren Integritätsregeln erweitern. Dabei wurde für eine Erweiterung, der Angabe von Reihenfolgebeziehungen zwischen Aktivitäten, bereits ein Verifikationsansatz vorgestellt. Eine andere Erweiterung dieses Kerns von Integritätsregeln besteht darin, ihre Gültigkeit einzuschränken, indem diese z.B. von Kontextdaten abhängig gemacht wird. Mit dieser Erweiterung lassen sich Integritätsregeln definieren, die z.B. nur für Personen ab einem Alter von 60 Jahren gelten sollen. Die Integritätsregeln können auch dadurch erweitert werden, dass angegeben wird, wie strikt diese einzuhalten sind. So können Integritätsregeln als Vorschriften oder Empfehlungen formuliert werden. Im Falle von Vorschriften müssen die Integritätsregeln strikt eingehalten werden und im Falle von Empfehlungen sollen die Integritätsregeln zwar eingehalten werden, müssen aber nicht. Ein weiteres mögliches Feature besteht darin, den Integritätsregeln Prioritäten zu vergeben. Damit könnten leicht Konflikte zwischen Integritätsregeln aufgelöst werden.

Nachdem die Art der Integritätsregeln festgelegt wurde, folgte die semantische Prozessverifikation mit Graph-Algorithmen. Dabei wurde die Idee verfolgt, die Integritätsregeln zuerst auf strukturelle Kriterien abzubilden und daraufhin diese strukturellen Kriterien zu verifizieren. Für diese Verifikation wurden im Anschluss verschiedene Ansätze zur Optimierung vorgeschlagen. Ein auf dem Rete-Algorithmus basierender Optimierungsansatz hatte zum Ziel, die Menge der zu prüfenden Integritätsregeln einzuschränken. Zwei andere Ansätze versuchten die Verifikation durch Anpassung des Prozess-Meta-Modells zu optimieren. So bestand ein Ansatz darin, dass das Prozess-Meta-Modell die für die LabelledTargetExists1- und LabelledTargetExists2-Algorithmen verwendeten Positionsmarkierungen bereitstellt. In dem Fall müssen für die Verifikation einer Integritätsregel

nicht jedes Mal die Positionsmarkierungen neu bestimmt werden, was ziemlich aufwendig ist, da der Prozess so weit zu durchlaufen ist, bis alle relevanten Aktivitäten markiert sind. Wenn allerdings das Prozess-Meta-Modell diese bereitstellt, sind die auf Positionsmarkierungen basierenden LabelledTargetExists1- und LabelledTargetExists2-Algorithmen wesentlich effizienter als die auf den WriterExists-Algorithmus basierenden TargetExists1- und TargetExists2-Algorithmen. Die TargetExists1- und TargetExists2-Algorithmen (und damit auch der WriterExists-Algorithmus) können optimiert werden, indem der Prozess mit Weginformationen ausgestattet wird, so dass beim Durchlaufen des Prozesses nicht relevante Blöcke übersprungen werden können. Am Schluss wurde noch der Verifikationsaufwand von verschiedenen Klassen von Integritätsregeln bestimmt.

Diese Arbeit könnte weitergeführt werden, indem die Integritätsregeln mit anderen Ansätzen, wie z.B. dem Model Checking, verifiziert werden und die Effizienz mit dem graphbasierten Ansatz verglichen wird. Da bis jetzt nur Prozessschemas verifiziert werden, wäre eine andere Erweiterung dieser Arbeit, ebenfalls Instanzen mit den hier vorgestellten Integritätsregeln zu verifizieren. Bei der Verifikation unterscheiden sich Instanzen von Schemas im Wesentlichen nur hinsichtlich der XOR-Blöcke. So ist z.B. in Abbildung 8.1 auf dem Prozessschema S die Integritätsregel $A \rightarrow B \vee C$ nicht erfüllt. Hingegen ist die Integritätsregel auf der Instanz I erfüllt, da der Zweig, auf dem D liegt, abgewählt wurde. Ein möglicher Ansatz zur Verifikation von Instanzen wäre, die abgewählten Zweige von XOR-Blöcken auszublenden.

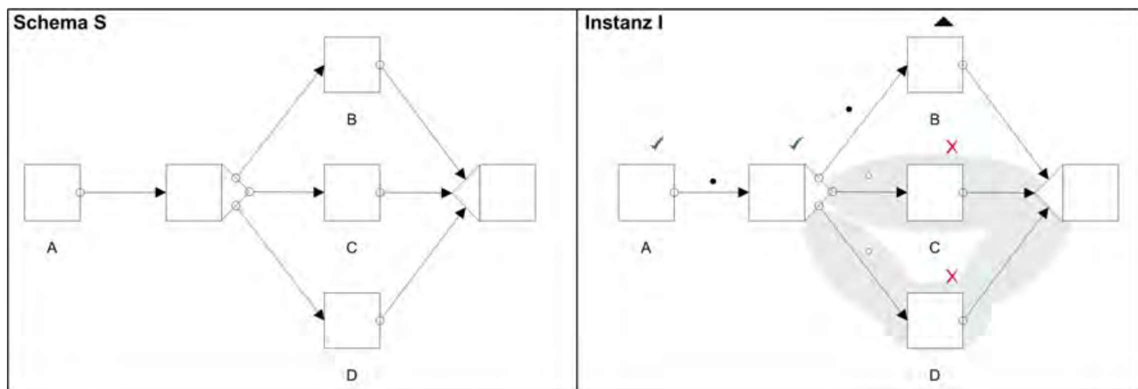


Abbildung 8.1: Integritätsregel $A \rightarrow B \vee C$ ist auf dem Prozessschema S nicht erfüllt, jedoch auf der Prozessinstanz I.

Literaturverzeichnis

[BBKK04] Bae, J., H. Bae, S. Kang und Y. Kim: *Automatic Control of Workflow Processes Using ECA Rules.* IEEE Transactions on Knowledge and Data Engineering, 16(8):1010–1023, 2004.

[Bus00] Business rules group: *Defining business rules - What are they really?*, Juli 2000.

[Dad07] Dadam, P.: *Vorlesung: Workflow-Management-Systeme*, 2006/07.

[DKRR98] Davulcu, H., M. Kifer, C. Ramakrishnan und I. Ramakrishnan: *Logic based modeling and analysis of workflows.* In: *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, Seiten 25–33, New York, NY, USA, 1998. ACM Press.

[FFK05] Fisteus, J., L. Fernandez und C. Kloos: *Applying model checking to BPEL4WS business collaborations.* In: *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, Seiten 826–830, New York, NY, USA, 2005. ACM Press.

[FH 04] FH Bonn-Rhein-Sieg: *Wissensbasierte Systeme II*, 2003/04.

[GL05] Gruhn, V. und R. Laue: *Using timed model checking for verifying workflows*, 2005.

[GRH+04] Greiner, U., J. Ramsch, B. Heller, M. Löffler, R. Müller und E. Rahm: *Adaptive Guideline-based Treatment Workflows with AdaptFlow.* In: *Proc. Symposium on Computerized Guidelines and Protocols (CGP 2004)*, 2004.

[GS06] Grässle, M. und P. Schacher: *Agile Unternehmen durch Business Rules.* Springer, 2006.

[IBM07] IBM: *www.ibm.com/software/integration/wmqwf*, Juli 2007.

[ILO05] ILOG: *ILOG JRules and IBM MQWF - White Paper*, 2005.

[ILO07] ILOG: *www.ilog.com/products/jrules*, Juli 2007.

[KEP00] Knolmayer, Gerhard, Rainer Endl und Marcel Pfahrer: *Modeling Processes and Workflows by Business Rules.* In: *Business Process Management, Models, Techniques, and Empirical Studies*, Seiten 16–29, London, UK, 2000. Springer-Verlag.

[KGMW99] Karamanolis, C., D. Giannakopoulou, J. Magee und S. Wheater: *Modeling and analysis of workflow processes.* Technischer Bericht, Department of Computing, Imperial College, 1999.

[LRD06a] Ly, T., S. Rinderle und P. Dadam: *Integration and Verification of Semantic Constraints in Adaptive Process Management Systems*, 2006.

[LRD06b] Ly, T., S. Rinderle und P. Dadam: *Semantic Correctness in adaptive Process Management Systems.* In: *Proc. Int'l. Conf. on Business Process Management, BPM 2006*, 2006.

[MGR04] Müller, R., U. Greiner und E. Rahm: *AGENT WORK: a workflow system supporting rule-based workflow adaptation.* Data Knowl. Eng., 51(2):223–256, 2004.

[Nah05] Nahler, M.: *Semantische Konflikte in adaptiven Prozess-Management-Systemen.* Diplomarbeit, Universität Ulm, 2005.

[Pal04] Palshikar, G.: *An introduction to model checking*, 2004.

[Rei00] Reichert, M.: *Dynamische Ablaufänderungen iin Workflow-Management-Systemen.* Doktorarbeit, Universität Ulm, 2000.

[Rin04] Rinderle, S.: *Schema Evolution in Process Management Systems.* Doktorarbeit, Universität Ulm, Fakultät für Informatik, 2004.

[SOS05] Sadiq, S., M. Orlowska und W. Sadiq: *Specification and validation of process constraints for flexible workflows.* Inf. Syst., 30(5):349–378, 2005.

[vH02] Halle, B. von: *Business rules applied - Building better systems using the business rules approach.* Wiley Computer Publishing, 2002.

[WBB04] Wainer, J., F. Bezerra und P. Barthelmess: *Tucupi: a flexible workflow system based on overridable constraints.* In: *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, Seiten 498–502, New York, NY, USA, 2004. ACM Press.

[WG06] Wong, P. und J. Gibbons: *A process algebraic approach to workflow verification*, 2006.

[Wik07a] Wikipedia (EN): *Business rule - Wikipedia*, Mai 2007.

[Wik07b] Wikipedia (EN): *Model Checking - Wikipedia*, Mai 2007.

[Wik07c] WIKIPEDIA (EN): *Process Algebra - Wikipedia*, Mai 2007.

[Won06] WONG, P.: *Towards a unified model for workflow processes*, 2006.

[Zho06] ZHOU, H.: *Effiziente Überprüfung semantischer Korrektheit in adaptivien Prozess-Management-Systemen.* Diplomarbeit, Universität Ulm, 2006.

Abbildungsverzeichnis

2.1 Blockstrukturierung und Datenflussmodellierung in einem ADEPT-Prozess . 5

2.2 Prozessinstanz mit einer Synchronisationskante zwischen den Aktivitäten B und C 6

4.1 Kopplung einer BRE an ein PMS am Beispiel von *ILOG JRules* und *MQ Workflow* 19

4.2 Anbindung von ILOG JRules an MQ Workflow 20

4.3 Rete-Netz zur Geschäfsregel R_1 21

4.4 Rete-Netz für die Geschäfsregel R_1 nach einer Änderung des *Working Memory* 22

4.5 Einschränkung der Adaptionsmöglichkeiten mit einer *pocket of flexibility* . . 24

4.6 Schematische Darstellung der semantischen Verifikation mit Geschäftsregeln 27

5.1 Semantische Prozessverifikation mit einer Beispiel-Integritätsregel 35

5.2 Erweiterungsmöglichkeiten des hier vorgestellten Kerns an Integritätsregeln 36

6.1 Semantischen Prozessverifikation mit Integritätsregeln der Kategorie, wie in Kap. 5 definiert 40

6.2 Die Aktivitäten A, B und C können nicht alle zusammen ausgeführt werden, da A und B auf verschiedenen Zweigen eines XOR-Blocks liegen. 42

6.3 $A \not\prec\bullet\not\succ C$ ist erfüllt, jedoch $A \not\prec\bullet\not\succ B$ und $B \not\prec\bullet\not\succ C$ jeweils nicht. 44

6.4 Wenn der obere Zweig des XOR-Blocks ausgewählt wird, wird zwar A, aber nicht C ausgeführt. 45

6.5 Unabhängig davon, welcher Zweig des XOR-Blocks ausgewählt wird, kommt in jeder Ausführungsspur, in der A vorkommt, auch C vor. 46

6.6 Die Integritätsregel $c_2 = = \neg A \wedge B \rightarrow C$ ist erfüllt, die beiden Integritätsregeln $c' = \neg A \rightarrow C$ und $c'' = B \rightarrow C$ jedoch jeweils nicht. 48

6.7 In jeder Ausführungsspur, in der B vorkommt, kommt entweder A oder C vor. Allerdings ist weder $B \bullet<> A$ noch $B \bullet<> C$ erfüllt. 48

6.8 Die Integritätsregel $c = A \wedge \neg B \wedge \neg C \rightarrow (D \vee \neg E \vee \neg F) \wedge (G \vee H)$ ist erfüllt. 53

6.9 Verifikation des strukturellen Kriteriums $S \bullet<> T$ mit dem TargetExists1-Algorithmus 56

6.10 Verifikation von $A \bullet<> B$ mit dem LabelledTargetExists1-Algorithmus . . 58

6.11 Verifikation des strukturellen Kriteriums $S \bullet<> [T1 \mid T2 \mid T3]$ mit dem TargetExists2-Algorithmus 61
6.12 Verifikation des strukturellen Kriteriums $S \bullet<> [T1 \mid T2 \mid T3 \mid T4]$ mit dem LabelledTargetExists2-Algorithmus 63
6.13 Positionsmarkierungen bei der Verifikation in Abbildung 6.12 64
6.14 Integritätsregel $c = A \wedge B \rightarrow (C \vee D) \wedge (E \vee F), [A < B, A > C, A < D, B < F]$ ist erfüllt. 68

7.1 Rete-Netz für den Bedingungsteil $\mathrm{cp}_1 = \mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}$ 72
7.2 Rete-Netz für den Bedingungsteil $cp_2 = A \wedge B \wedge D$ 72
7.3 Rete-Netzwerk für die Bedingungsteile $\mathrm{cp}_1 = \mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}$ und $\mathrm{cp}_2 = \mathrm{A} \wedge \mathrm{B} \wedge \mathrm{D}$ 73
7.4 Prozess, auf dem das *Working Memory* für das Rete-Netzwerk in Abb. 7.3 basiert 74
7.5 Bestimmung der Integritätsregeln, deren Bedingungsteil in mindestens einer Spur erfüllt ist, wenn der Prozess in Abb. 7.4 als *Working Memory* dient . . 75
7.6 Änderung des *Working Memory* durch Einfügen der Aktivität C in den Prozess 76
7.7 Rete-Netzwerk nach der Änderung des *Working Memory* 77
7.8 Optimierung des Prozessmetamodells durch besondere Knoten-IDs 78

8.1 Integritätsregel $A \rightarrow B \vee C$ ist auf dem Prozessschema S nicht erfüllt, jedoch auf der Prozessinstanz I. 81

Tabellenverzeichnis

2.1 ADEPT-Funktionen 10

6.1 Überblick über neu eingeführte strukturelle Kriterien 54

Algorithmenverzeichnis

1 Exclusion-Algorithmus 89
2 TargetExists1-Algorithmus 90
3 LabelledTargetExists1-Algorithmus 94
4 TargetExists2-Algorithmus 95
5 LabelledTargetExists2-Algorithmus 99
6 ConditionpartSatisfiedController-Algorithmus 103

Anhang A

Algorithmen

Algorithmus 1 Exclusion-Algorithmus

function S $\not<\bullet\not>$ T(CFS, S, T) $\rightarrow$ (result)

input

CFS :	Korrekter KF-Graph (N, E, ...)
S $\in$ N:	Quellaktivität
T $\in$ N:	Zielaktivität

output

result :	Bool'scher Wert, der anzeigt, ob die Quellaktivität die Ausführung der Zielaktivität ausschließt

if (S $\notin$ N) **then**
  //S kommt nicht im Prozess vor.
  return true;
else if (T $\notin$ N) **then**
  //T kommt nicht im Prozess vor.
  return true;
else if (minBlock(S,T) = XOR **and** S $\notin$ c_pred*(T) **and**
  S $\notin$ c_succ*(T)) **then**
  //S und T liegen auf verschiedenen Zweigen eines XOR-Blocks.
  return true;
else
  return false
end if
end function

Algorithmus 2 TargetExists1-Algorithmus

function S $\bullet <>$ T(CFS, S, T) $\rightarrow$ (result)

input

CFS :	Korrekter KF-Graph (N, E, ...)
S $\in$ N:	Quellaktivität
T $\in$ N:	Zielaktivität

output

result :	Bool'scher Wert, der anzeigt, ob die Quellaktivität die Ausführung der Zielaktivität impliziert

if (S $\notin$ N) **then**
    //S kommt nicht im Prozess vor.
    return true;
end if
//Initialisierung: Jeden Knoten mit den Attributen Supplied und Counter inititali-
//sieren
for all (n $\in$ N) **do**
    Supplied(n) := false;
    Counter(n) := 0;
end for
//Markierungsphase
if (S $\in$ c_succ*(T)) **then**
    //S liegt hinter der Zielaktivität T. $\rightarrow$ Der Prozess muss vorwärts traversiert
    //werden. Für direkte Nachfolger n der Zielaktivität T: Erhoehe Counter(n*)*
    //um 1 und fuege n der Menge NodeList der noch zu betrachtenden Knoten*
    //hinzu.
    for all (n* $\in$ c_succ(T)) **do**
        Counter(n*) := Counter(n*) + 1;
        NodeList := NodeList $\cup$ {n*};
    end for
    //Solange neue Knoten untersuchen, bis S gefunden wurde oder kein weiterer
    //Knoten mehr dazukommt.
    while (NodeList $\neq \emptyset$ **and** Supplied(S) = false) **do**
        NewNodeList := $\emptyset$;
        for all (n $\in$ NodeList) **do**
            if (V_{in}^{n} $\in$ {ONE_OF_ONE, ALL_OF_ALL} **or** Counter(n) =
                #Eingangskontrollkanten von n) **then**
                Supplied(n) := true;

```
            for all (n* ∈ c_succ(n)) do
                if (Supplied(n*) = false) then
                    Counter(n*) := Counter(n*) + 1;
                    NewNodeList := NewNodeList ∪ {n*};
                end if
            end for
        end if
    end for
    NodeList := NewNodeList;
  end while
else if (S ∈ c_pred*(T)) then
  //Analog zum Fall S ∈ c_succ*(t): Der Unterschied ist, dass der Prozess in
  //diesem Fall rückwärts traversiert wird. Für direkte Vorgänger n* der Zielak-
  //tivität T: Erhoehe Counter(n*) um 1 und fuege n* der Menge NodeList der
  //noch zu betrachtenden Knoten hinzu.
  for all (n* ∈ c_pred(T)) do
      Counter(n*) := Counter(n*) + 1;
      NodeList := NodeList ∪ {n*};
  end for

  //Solange neue Knoten untersuchen, bis S markiert wurde oder bis kein weiterer
  //Knoten mehr dazukommt.
  while (NodeList ≠ ∅ and Supplied(S) = false) do
      NewNodeList := ∅;
      for all (n ∈ NodeList) do
          if (V^n_out ∈ {ONE_OF_ONE, ALL_OF_ALL} or Counter(n) =
              #Ausgangskontrollkanten von n) then
              Supplied(n) := true;
              for all (n* ∈ c_pred(n)) do
                  if (Supplied(n*) = false) then
                      Counter(n*) := Counter(n*) + 1;
                      NewNodeList := NewNodeList ∪ {n*};
                  end if
              end for
          end if
      end for
      NodeList := NewNodeList;
  end while
```

```
    else if (MinBlock({S, T}) = AND-Block) then
        //Wenn S und T auf zwei unterschiedlichen Zweigen eines AND-Blocks liegen,
        //dann ist S versorgt, wenn der Join-Knoten des AND-Blocks versorgt ist. Des-
        //halb reicht es zu prüfen, ob ein Weg von T zu dem Join-Knoten gefunden
        //werden kann. In was für einem Block S liegt, spielt keine Rolle. Selbst wenn
        //die S nicht immer ausgeführt wird, wenn der AND-Join-Knoten ausgeführt
        //wird (die S also in einem XOR-Block liegt), muss die Versorgung mit den
        //Zielaktivitäten im AND-Block sichergestellt sein. Es ist nämlich nicht bekannt,
        //wann die S ausgeführt wird bzw. nicht. Deswegen muss angenommen werden,
        //dass A immer ausgeführt wird, wenn der AND-Join-Knoten zur Ausführung
        //kommt. Analog zum Fall S ∈ c_succ*(t): Der Unterschied ist, dass jetzt
        //S := AND-Join gilt.
        OldS-A :=SA;
        S := Endknoten des MinBlock({S, T});
        //Für direkte Nachfolger n* der Zielaktivität T: Erhoehe Counter(n*) um 1 und
        //fuege n* der Menge NodeList der noch zu betrachtenden Knoten hinzu.
        for all (n* ∈ c_succ(t)) do
            Counter(n*) := Counter(n*) + 1;
            NodeList := NodeList ∪ {n*};
        end for
        //Solange neue Knoten untersuchen, bis S markiert wurde oder bis kein weiterer
        //Knoten mehr dazukommt.
        while (NodeList ≠ ∅ and Supplied(S) = false) do
            NewNodeList := ∅;
            for all (n ∈ NodeList) do
                if (V_in^n ∈ {ONE_OF_ONE, ALL_OF_ALL} or Counter(n) =
                    #Eingangskontrollkanten von n) then
                    Supplied(n) := true;
                    for all (n* ∈ c_succ(n)) do
                        if (Supplied(n*) = false) then
                            Counter(n*) := Counter(n*) + 1;
                            NewNodeList := NewNodeList ∪ {n*};
                        end if
                    end for
                end if
            end for
            NodeList := NewNodeList;
        end while
        //S wieder zurücksetzen
        S := OldS-A;
```

```
        else
            //S und T befinden sich auf zwei unterschiedlichen Zweigen eines XOR-
            //Blocks. Damit können sie nie zusammen ausgeführt werden. Deshalb
            //kann T nichts dazu beitragen, dass wenn S ausgeführt wird, die Ver-
            //sorgung mit den Zielaktivitäten sichergestellt ist.
        end if
        //Wurde die Quellaktivität S markiert?
        if (Supplied(S) = true) then
            return true;
        end if
    end function
```

Algorithmus 3 LabelledTargetExists1-Algorithmus

```
function S •<> T(CFS, S, T) → (result)
input
    CFS :        Korrekter KF-Graph (N, E, ...)
    S ∈ N:       Quellaktivität
    T ∈ N:       Zielaktivität
output
    result :     Bool'scher Wert, der anzeigt, ob die Quellaktivität die Aus-
                 führung der Zielaktivität impliziert
functions
    getPositionLabel(A) → (positionLabel):
    gibt die Positionsmarkierung positionLabel für die Aktivität A zurück

    containsTupel(positionLabel, tupel) → {True, False}:
    gibt zurück, ob die Positionsmarkierung positionLabel ein Tupel tupel enthält

  if (S ∉ N) then
     return true;
  else if (T ∉ N) then
     return false;
  end if
  //Initialisierung: Erstelle die Positionsmarkierungen von S und von T.
  //(Die Positionsmarkierung für eine Aktivität gibt an, in welchen XOR-Blöcken und
  //jeweils auf welchem Zweig in diesen XOR-Blöcken diese Aktivität liegt. Sie ist als
  //eine FIFO-Liste (First In, First Out) von Tupeln der Form (Blocknummer, Zweig-
  //nummer) realisiert, wobei ein Tupel (Blocknummer, Zweignummer) der Positions-
  //markierung für einen XOR-Block steht, in dem die Aktivität liegt. Die Blocknum-
  //mer kennzeichnet eindeutig den XOR-Block und die Zweignummer den Zweig in
  //diesem XOR-Block.)
  if (getPositionLabel(T) = ∅) then
     //T liegt in keinem XOR-Block und wird damit immer ausgeführt.
     return true;
  else
     for all (tupel ∈ getPositionLabel(T)) do
        if (! containsTupel(getPositionLabel(S), tupel)) then
           return false;
        end if
     end for
     return true;
  end if
end function
```

Algorithmus 4 TargetExists2-Algorithmus

```
function S•<>[T1|T2|...](CFS, S, {T1, T2, ...}) → (result)

input
    CFS :                      Korrekter KF-Graph (N, E, ...)
    S ∈ N:                     Quellaktivität
    {T1, T2, ...} ⊆ N:         Menge der Ziel-Aktivitäten

output
    result :                   Bool'scher Wert, der anzeigt, ob die Quellaktivität
                               die Versorgung mit den Ziel-Aktivitäten impliziert

    //Initialisierung: Jeden Knoten mit den Attributen Supplied und Counter inititali-
    //sieren
    if (S ∉ N) then
        //S kommt nicht im Prozess vor.
        return true;
    end if
    for all (n ∈ N) do
        Supplied(n) := false;
        Counter(n) := 0;
    end for

    for all (t ∈ {T1, T2, ...}) do
        if (Supplied(t) = false) then
            Supplied(t) := true;
            NodeList := ∅;
            if (S ∈ c_succ*(t)) then
                //S liegt hinter der momentan betrachteten Zielaktivität t. → Der Prozess
                //muss vorwärts traversiert werden. Für direkte Nachfolger n* der Zielak-
                //tivität t: Erhoehe Counter(n*) um 1 und fuege n* der Menge NodeList
                //der noch zu betrachtenden Knoten hinzu.
                for all (n* ∈ c_succ(t)) do
                    if (Supplied(n*) = FALSE) then
                        Counter(n*) := Counter(n*) + 1;
                        NodeList := NodeList ∪ {n*};
                    end if
                end for
```

```
        //Menge der als versorgt markierten Knoten solange erweitern, bis sie
        //entweder S enthält oder kein weiterer Knoten mehr dazukommt.
        while (NodeList ≠ ∅ and Supplied(S) = FALSE) do
            NewNodeList := ∅;
            for all (n ∈ NodeList) do
                if (V^n_in ∈ {ONE_OF_ONE, ALL_OF_ALL} or Counter(n) =
                    #Eingangskontrollkanten von n) then
                    //Den Knoten als versorgt markieren
                    Supplied(n) := TRUE;
                    for all (n* ∈ c_succ(n)) do
                        if (Supplied(n*)= FALSE) then
                            Counter(n*) := Counter(n*) + 1;
                            NewNodeList := NewNodeList ∪ {n*};
                        end if
                    end for
                end if
            end for
            NodeList := NewNodeList;
        end while
    else if (S ∈ c_pred*(t)) then
        //Analog zum Fall S ∈ c_succ*(t): Der Unterschied ist, dass der Prozess
        //in diesem Fall rückwärts traversiert wird. S liegt hinter der momentan
        //betrachteten Zielaktivität t. → Der Prozess muss vorwärts traversiert
        //werden. Für direkte Vorgänger n* der Zielaktivität t: Erhoehe Coun-
        //ter(n*) um 1 und fuege n* der Menge NodeList der noch zu betrach-
        //tenden Knoten hinzu.
        for all (n* ∈ c_pred(t)) do
            if (Supplied(n*) = FALSE) then
                Counter(n*) := Counter(n*) + 1;
                NodeList := NodeList ∪ {n*};
            end if
        end for
        //Menge der als versorgt markierten Knoten solange erweitern, bis sie
        //entweder S enthält oder kein weiterer Knoten mehr dazukommt.
        while (NodeList ≠ ∅ and Supplied(S) = FALSE) do
            NewNodeList := ∅;
            for all (n ∈ NodeList) do
                if (V^n_out ∈ {ONE_OF_ONE, ALL_OF_ALL} or Counter(n) =
                    #Ausgangskontrollkanten von n) then
                    //Den Knoten als versorgt markieren
                    Supplied(n) := TRUE;
```

```
                for all (n* ∈ c_pred(n)) do
                    if (Supplied(n*)= FALSE) then
                        Counter(n*) := Counter(n*) + 1;
                        NewNodeList := NewNodeList ∪ {n*};
                    end if
                end for
            end if
        end for
        NodeList := NewNodeList;
    end while
else if (MinBlock({SourceActivity, t}) = AND-Block) then
    //Wenn S und t auf zwei unterschiedlichen Zweigen eines AND-Blocks
    //liegen, dann ist S versorgt, wenn der Join-Knoten des AND-Blocks
    //versorgt ist. Deshalb reicht es zu prüfen, ob ein Weg von t zu dem
    //Join-Knoten gefunden werden kann. In was für einem Block S liegt,
    //spielt keine Rolle. Selbst wenn die S nicht immer ausgeführt wird,
    //wenn der AND-Join-Knoten ausgeführt wird (die S also in einem
    //XOR-Block liegt), muss die Versorgung mit den Zielaktivitäten im
    //AND-Block sichergestellt sein. Es ist nämlich nicht bekannt, wann die
    //S ausgeführt wird bzw. nicht. Deswegen muss angenommen werden,
    //dass A immer ausgeführt wird, wenn der AND-Join-Knoten zur Aus-
    //führung kommt.
    //Analog zum Fall S ∈ c_succ*(t): Der Unterschied ist, dass jetzt
    //S := AND-Join gilt.
    OldS-A :=SA;
    S := Endknoten des minBlock(S, t);
    //S liegt hinter der momentan betrachteten Zielaktivität t. → Der
    //Prozess muss vorwärts traversiert werden. Für direkte Nachfolger n*
    //der Zielaktivität t: Erhoehe Counter(n*) um 1 und fuege n* der Menge
    //NodeList der noch zu betrachtenden Knoten hinzu.
    for all (n* ∈ c_succ(t)) do
        if (Supplied(n*) = FALSE) then
            Counter(n*) := Counter(n*) + 1;
            NodeList := NodeList ∪ {n*};
        end if
    end for

    //Menge der als versorgt markierten Knoten solange erweitern, bis sie
    //entweder S enthält oder kein weiterer Knoten mehr dazukommt.
    while (NodeList ≠ ∅ and Supplied(S) = FALSE) do
        NewNodeList := ∅;
```

```
                    for all (n ∈ NodeList) do
                        if (V_in^n ∈ {ONE_OF_ONE, ALL_OF_ALL} or Counter(n) =
                            #Eingangskontrollkanten von n) then
                            //Den Knoten als versorgt markieren
                            Supplied(n) := TRUE;
                            for all (n* ∈ c_succ(n)) do
                                if (Supplied(n*)= FALSE) then
                                    Counter(n*) := Counter(n*) + 1;
                                    NewNodeList := NewNodeList ∪ {n*};
                                end if
                            end for
                        end if
                    end for
                    NodeList := NewNodeList;
                end while
                //S wieder zurücksetzen
                S := OldS-A;
            else
                //S und t befinden sich auf zwei unterschiedlichen Zweigen eines XOR-
                //Blocks. Damit können sie nie zusammen ausgeführt werden. Deshalb
                //kann t nichts dazu beitragen, dass wenn S ausgeführt wird, die Ver-
                //sorgung mit den Zielaktivitäten sichergestellt ist.
            end if
        end if

        if (Supplied(S) = TRUE) then
            return TRUE;
        end if
    end for
end function
```

Algorithmus 5 LabelledTargetExists2-Algorithmus

function S•<>[T_1|T_2|...](CFS, S, targetActivities) → (result)

input

CFS :	Korrekter KF-Graph (N, E, ...)
S ∈ N:	Quellaktivität
targetActivities:	Menge der Ziel-Aktivitäten (={T_1, T_2, ...} ⊆ N)

output

result :	Bool'scher Wert, der anzeigt, ob die Quellaktivität die Versorgung mit den Zielaktivitäten impliziert

functions

sort(liste) → (void)
sortiert die Aktivitätenliste liste absteigend nach der Anzahl der Tupel, die die Positionsmarkierungen der einzelnen Aktivitäten enthalten

readActivity(liste) → (A)
liest die erste Aktivität A aus der Liste von Aktivitäten liste aus (aber entnimmt die Aktivität nicht)

readTupel(positionLabel) → (tupel)
liest das erste Tupel tupel aus der Positionsmarkierung positionLabel aus (aber entnimmt dieses nicht)

getNumberOfBranches(blockNr) → (number)
gibt zurück, wie viele Zweige der XOR-Block blockNr besitzt

getPositionLabel(A) → (positionLabel)
gibt die Positionsmarkierung positionLabel der Aktivität A zurück

containsTupel(positionLabel, tupel) → {True, False}
gibt zurück, ob die Positionsmarkierung positionLabel das Tupel tupel besitzt

removeActivity(liste, A) → (void)
entfernt die Aktivität A aus der Liste von Aktivitäten liste

removeTupel(A, tupel) → (void)
entfernt das Tupel tupel aus der Positionsmarkierung von Aktivität A

```
if (S ∉ N) then
    //S kommt nicht im Prozess vor.
    return true;
end if

//Alle XOR-Blöcke, in denen S liegt, müssen nicht versorgt werden, da nur die
//Ausführung von S die Versorgung mit den Zielaktivitäten impliziert.
for all (T ∈ targetActivities) do
    for all (tupel ∈ getPositionLabel(S)) do
        removeTupel(T, tupel);
    end for
    //Impliziert die Ausführung von S die Ausführung der momentan betrachteten
    //Zielaktivität T?
    //Zielaktivität darf nicht in einem XOR-Block liegen, in dem S nicht liegt. →
    //Positionsmarkierung muss leer sein.
    if (getPositionLabel(T) = ∅) then
        return true;
    else
        //Liegen S und die momentan betrachtete Zielaktivität T auf verschiedenen
        //Zweigen eines XOR-Blocks?
        //→ S und können nie zusammen ausgeführt werden.
        //→ T trägt nicht zur Versorgung mit den Zielaktivitäten bei.
        //Dazu: Prüfen, ob es in S und T jeweils ein Tupel mit gleicher BlockNr
        //und unterschiedlicher ZweigNr gibt.
        for all (tupel1 ∈ getPositionLabel(S)) do
            for all (tupel2 ∈ getPositionLabel(T)) do
                if (tupel1.BlockNr = tupel2.BlockNr and
                    tupel1.ZweigNr ≠ tupel2.ZweigNr) then
                    removeActivity(targetActivities, T);
                end if
            end for
        end for
    end if
end for
while (targetActivities ≠ ∅) do
    sort(targetActivities);
    currentT-A := readActivity(targetActivities);
    tupel := readTupel(getPositionLabel(currentT-A));
    currentBlockNr := tupel.BlockNr;
    nr_of_branches := getNumberOfBranches(blockNr);
```

```
for (i:=1 to number_of_branches) do
    //Gibt es eine T, die auf diesem Zweig und nicht in einem weiteren XOR-
    //Block liegt? → Die Positionsmarkierung muss (currentBlockNr,i) als er-
    //stes Tupel besitzen.
    supplied := false;
    for all (T ∈ targetActivities) do
        tupel := readTupel(T);
        if (currentBlockNr = tupel.BlockNr and i = tupel.ZweigNr) then
            supplied := true;
        end if
    end for
    if (!supplied) then
        //Alle Zielaktivitäten, die in diesem Block liegen, helfen nicht bei der
        //Versorgung mit den Zielaktivitäten. → Diese Zielaktivitäten sollen
        //nicht weiter betrachtet werden.
        for all (T ∈ targetActivities) do
            for (i:=1 to number_of_branches) do
                if (containsTupel(getPositionLabel(T),
                    (currentBlockNr,i))) then
                    removeActivity(targetActivities, T);
                end if
            end for
        end for
        break;
    end if
end for
//Jeder Zweig ist versorgt.
//→ Dieser XOR-Block muss nicht mehr betrachtet werden.
//→ Aus den Positionsmarkierungen der Aktivitäten wird das zu diesem XOR-
//Block gehörende Tupel gelöscht.
for all (T ∈ targetActivities) do
    for (i:=1 to number_of_branches) do
        if (containsTupel(getPositionLabel(T),
            (currentBlockNr,i))) then
            removeTupel(T, (currentBlockNr,i));
        end if
    end for
end for
```

```
        //Ist die Versorgung mit den Zielaktivitäten sichergestellt?
        for all (T ∈ targetActivities) do
            if (getPositionLabel(T) = ∅) then
                return true;
            end if
        end for
    end while
    return false;
end function
```

Algorithmus 6 ConditionpartSatisfiedController-Algorithmus

```
abstract class Reteknoten {
    boolean fulfilled;
    JoinNodeList children;
}

class Alphaknoten extends Reteknoten {
    String attribute;
    String value;
    public Alphaknoten(String attribute, String value, JoinNodeList children) {
        this.attribute = attribute;
        this.value = value;
        this.fulfilled = false;
        this.children = children;
}

    void addWME(WME wme) {
        if (!this.fulfilled) {
            if (this.attribute == wme.attribute and this.value == wme.value) {
                this.fulfilled = true;
                foreach child : children
                    child.right_activate();
            }
        }
    }


    void deleteWME(WME wme) {
        if (this.fulfilled) {
            if (this.attribute == wme.attribute and this.value == wme.value) {
                this.fulfilled = false;
                foreach child : children
                    child.right_deactivate();
            }
        }
    }
}
```

```
class Betaknoten extends Reteknoten {
    Reteknoten leftParent;
    Reteknoten rightParent;
    public Betaknoten(Reteknoten leftParent, Reteknoten rightParent) {
        this.fulfilled = false;
        this.leftParent = leftParent;
        this.rightParent = rightParent;
    }

    void right_activate() {
        if (!this.fulfilled) {
            if (this.leftParent == NULL) {
                //oberster Betaknoten => keinen linken Vorgängerknoten
                this.fulfilled = true;
                foreach child : children
                    child.left_activate();
            }else if (this.leftParent.fulfilled) {
                this.fulfilled = true;
                foreach child : children
                    child.left_activate();
            }
        }
    }
    void right_deactivate() {
        if (this.fulfilled) {
            this.fulfilled = false;
            foreach child : children
                child.left_deactivate();
        }
    }
    void left_activate() {
        if (!this.fulfilled) {
            if (this.rightParent.fulfilled) {
                this.fulfilled = true;
                foreach child : children
                    child.left_activate();
            }
        }
    }
```

```
        void left_deactivate() {
                if (this.fulfilled) {
                        this.fulfilled = false;
                        foreach child : children
                                child.left_deactivate();
                }
        }
}
```

```
class WME {
        String attribute;
        String value;
}
```