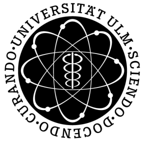


ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Automatisierte Überwachung von Business Process Compliance Regeln mit Daten-, Zeit-, Ressourcen- und Interaktions-Aspekten

Masterarbeit an der Universität Ulm

Vorgelegt von:
Hannes Beck
hannes.beck@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Betreuer:
David Knuplesch

2014

Fassung 26. Mai 2014

© 2014 Hannes Beck

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-LaTeX 2_ε

Kurzfassung

Business Process Compliance (BPC) beschreibt die Ausführung von Geschäftsprozessen im Einklang mit sogenannten Compliance Regeln, welche sich aus Standards, Richtlinien und Gesetzen ergeben können. Ein Ansatz zur Sicherstellung der BPC sieht die separate Modellierung von Compliance Regeln vor. Durch Prüfung dieser Regeln lässt sich die Compliance von Geschäftsprozessen automatisiert feststellen und überwachen.

Eine grafische Notation für Compliance Regeln sind die *Compliance Rule Graphs* (CRG). Mittels dieser lassen sich Compliance Regeln als grafische Modelle definieren, welche sowohl für Menschen als auch Maschinen lesbar und verständlich sind. Eine Erweiterung der CRG sind die *extended Compliance Rule Graphs* (eCRG), sie erweitern die CRG um vier weitere Perspektiven: Neben dem Kontrollfluss ist es mit eCRG auch möglich Compliance Regeln mit Bezug auf Daten-, Zeit-, Ressourcen- und Interaktions-Aspekten vollständig abzubilden.

Im Rahmen dieser Arbeit wird die Implementierung einer Ausführungsengine für eCRG beschrieben, welche zur Laufzeit eine automatisierte Überwachung der Business Process Compliance per eCRG ermöglicht. Die Ablaufprotokolle von Geschäftsprozessen werden hierbei für die Compliance Prüfung herangezogen. Neben der Kontrollflussperspektive werden dabei auch Daten-, Zeit-, Ressourcen- und Interaktions-Aspekte berücksichtigt. Zusätzlich bietet die Implementierung eine grafische Visualisierung des aktuellen Zustands der im Moment ausgeführten eCRG an.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Motivation 1

1.2. Zielsetzung 3

1.3. Gliederung 4

2. Grundlagen **7**

2.1. Compliance Rule Graphs 8

2.1.1. Ansatz 8

2.1.2. Struktur & Modellierungselemente 10

2.1.3. Ausführungssemantik 17

2.2. Extended Compliance Rule Graphs 22

2.2.1. Prozess-Perspektive 23

2.2.2. Interaktions-Perspektive 25

2.2.3. Zeit-Perspektive 26

2.2.4. Daten-Perspektive 27

2.2.5. Ressourcen-Perspektive 28

2.3. Anwendungsbeispiel 29

3. Ausführung & Bewertung von eCRG **31**

3.1. Rahmenbedingungen für die Ausführung von eCRG 32

3.1.1. Markierung der Ausführungszustände 32

3.1.2. Markierung einer eCRG – ExMark 34

3.1.3. Menge von eCRG Markierungen – MarkStructure 34

3.1.4. Knotenhierarchie 37
3.1.5. Vorgänger Voraussetzungen 38
3.2. Ausführunsregeln 39
3.2.1. Task Nodes 40
3.2.2. Message Nodes 47
3.2.3. Data Nodes 48
3.2.4. Resource Nodes 49
3.3. Bewertungsregeln 50
3.4. Veranschaulichung 52

4. Architektur **63**
4.1. Datenmodell 64
4.1.1. Prozessumgebung 64
4.1.2. Bedingungen und Relationen 66
4.1.3. Prozess-Ablaufprotokoll – Schnittstelle 67
4.1.4. eCRG – Schnittstelle 69
4.1.5. Datenstrukturen für die Markierung 71
4.2. Entwurf der grafischen Benutzeroberfläche 72

5. Implementierung **75**
5.1. Technische Grundlagen 76
5.2. Grafische Benutzeroberfläche 81
5.2.1. Ansicht des Programmfensters 82
5.2.2. Funktionen und Dialoge 83
5.2.3. JGraphX betreffende Implementierungen 90
5.3. Monitoring Algorithmen 97
5.3.1. Übersicht über den Gesamtalgorithmus 99
5.3.2. Algorithmen für die Ausführung von Events 100
5.3.3. Bestimmung nicht-abgeschlossener Vorgängerknoten 103
5.3.4. Vorbereitung der Markierungen für die Auswertung 103
5.3.5. Überprüfung der eCRG Forderungen 105
5.3.6. Auswertung der Markierungen 111

6. Related Work **113**

7. Zusammenfassung & Ausblick **117**

A. Abbildungen **119**

B. Quelltexte **125**

Abbildungsverzeichnis **141**

Tabellenverzeichnis **143**

Quelltextverzeichnis **145**

Literaturverzeichnis **147**

1

Einleitung

1.1. Motivation

Prozess-Management-Systeme (PMS) gewinnen in Unternehmen zunehmend an Bedeutung, denn sie ermöglichen unter anderem eine schnelle Reaktionszeit bei Marktveränderungen. Damit sind sie ein wichtiger Erfolgsfaktor im Wettbewerb mit konkurrierenden Unternehmen. Ein weiterer Grund für die fortschreitende Verbreitung von PMS sind die immer komplexer werdenden betrieblichen Abläufe, welche sich nur mehr schwer ohne entsprechende Softwareunterstützung bewältigen lassen [RMLD08].

Neben traditionellen Aspekten, wie der Modellierung, Ausführung und Analyse von Geschäftsprozessen, nimmt Business Process Compliance (BPC) für Unternehmen einen immer wichtigeren Stellenwert im Bereich des Prozessmanagements ein [RMLD08]. BPC bezeichnet die Ausführung von Geschäftsprozessen im Einklang mit den Auflagen,

die sich aus Gesetzen, Branchenstandards und Unternehmensrichtlinien ergeben. Bei Nichteinhaltung dieser Auflagen drohen empfindliche Strafen. Für Unternehmen ist somit ein adäquates Compliance Management unerlässlich [LRMD10]. Dieses beinhaltet unter anderem die Sicherstellung der Compliance für alle im Unternehmen vorhandenen Geschäftsprozesse. Die zuvor genannten Auflagen werden im Folgenden auch als Compliance Regeln bezeichnet.
Im Falle umfassender und komplexer Prozessmodelle oder auch bei zahlreichen Compliance Regeln, wird es immer schwieriger, eine vollständige Compliance Prüfung auf herkömmliche Weise per Hand durchzuführen. Wünschenswert wäre daher eine entsprechende Softwareunterstützung durch ein Programm, das in der Lage ist, automatisiert die Compliance von Geschäftsprozessen zu verifizieren [LKRM$^+$11, KR11]. Dadurch würde ein Großteil des Prüfaufwands vom Programm übernommen und infolgedessen ließe sich eine größere Anzahl von Regeln in verhältnismäßig kurzer Zeit prüfen.
Für die Umsetzung dessen wird jedoch eine maschinenlesbare Beschreibung der Compliance Regeln benötigt. Im Rahmen des Forschungsprojektes $SeaFlows$[1] wurden an der Universität Ulm die sogenannten *Compliance Rule Graphs* (CRG) entwickelt [LRMKD11]. Damit wurde eine grafische Notation zur Definition und Veranschaulichung von Compliance Regeln geschaffen, welche sowohl von Menschen als auch Maschinen interpretiert werden kann. Im Gegensatz dazu sind logischen Ausdrücke zur Definition von Compliance Regeln, beispielsweise Linear Temporal Logik Formeln, nur für Experten auf diesem Gebiet verständlich. Durch Hinzufügen weiterer Modellierungselemente zu den CRG entstanden, innerhalb des C^3Pro Projekts[2], die *extended Compliance Rule Graphs* (eCRG) [KRL$^+$13b]. Dies war notwendig, da mit CRG bis dato nur Modellierungsaspekte bezüglich der Kontrollflussperspektive von Compliance Regeln abgebildet werden konnten. Im Vergleich dazu, können mit eCRG nun auch Aspekte hinsichtlich der Daten-, Zeit-, Ressourcen- und Interaktions-Perspektive vollständig modelliert werden. Die Notwendigkeit dessen wurde anhand einer Fallstudie aufgezeigt [Sem13]. Dabei wurden verschiedene Compliance Regeln aus einem Realwelt-Szenario identifiziert und mittels eCRG definiert. Um alle Regeln komplett modellieren zu können, waren die neu hinzugekommenen Modellierungselemente von eCRG zwingend notwendig.

[1] SeaFlows Projektseite: `http://www.uni-ulm.de/index.php?id=9877`
[2] C^{3}Pro Projektseite: `http://www.uni-ulm.de/index.php?id=36098`

Mit dem von eCRG gewählten Ansatz, lässt sich eine automatisierte Überwachung von Compliance Regeln zur Laufzeit implementieren, welche auf der Evaluation von Prozess-Ablaufprotokollen beruht. Prozess-Ablaufprotokolle entsprechen Snapshots von Prozessinstanzen, sind also Momentaufnahmen des aktuellen Fortschritts von Prozessinstanzen. Befindet sich eine Prozessinstanz nicht mehr in der Ausführung, sondern wurde bereits beendet, dann enthält das dazugehörige Ablaufprotokoll alle Events, die während der gesamten Laufzeit der Prozessinstanz aufgetreten sind. In der Literatur werden Prozess-Ablaufprotokolle oft auch als *execution traces* oder *execution logs* bezeichnet. Aufgrund der Verwendung von Prozess-Ablaufprotokollen, sind per eCRG definierte Compliance Regeln unabhängig von einem speziellen Prozessmetamodell und können somit auf beliebige Prozessinstanzen angewandt werden [KR11].

1.2. Zielsetzung

Ziel der Arbeit ist es ein Programm zu entwickeln, welches in der Lage ist automatisiert und somit weitgehend selbstständig die Compliance von Geschäftsprozessen zur Laufzeit zu überwachen. Die dafür notwendige Compliance Prüfung wird mittels dem Konzept der *extended Compliance Rule Graphs* (eCRG) realisiert. Damit wird zum ersten Mal eine Ausführungsengine für eCRG implementiert, welche im Gegensatz zu bereits existierenden Ansätzen neben dem Kontrollfluss auch Modellierungsaspekte bezüglich der Daten-, Zeit-, Ressourcen- und Interaktions-Perspektive unterstützt. Die Validierung der zu prüfenden Compliance Regeln geschieht dabei mittels sogenannter Prozess-Ablaufprotokolle und damit unabhängig von spezifischen Prozessmodellen.
Durch diese prototypische Implementierung soll zum einen der durch eCRG gewählte Ansatz zur Überprüfung von BPC verifiziert werden und zum anderen soll ein Werkzeug geschaffen werden, welches das Auditieren von Geschäftsprozessen hinsichtlich der Einhaltung von Compliance Vorgaben unterstützt. Denn im Falle einer großen Anzahl von Regeln oder aufgrund umfangreicher und unter Umständen komplexer Regeln ist eine manuelle Prüfung aller Geschäftsprozesse und Compliance Regeln nur mit enormem Aufwand durchführbar. Zur Unterstützung der Nutzer soll eine grafische Darstellung des Zustands der momentan ausgeführten eCRG angeboten werden. Diese visuelle

Rückmeldung soll ein schnelles Identifizieren möglicher Compliance-Verletzungen ermöglichen und damit auch eine effektive Beurteilung der BPC zulassen.
Bei der Entwicklung soll besonders Wert auf eine einfache und intuitive Bedienbarkeit des Programms gelegt werden. Des Weiteren soll das Programm, wo sinnvoll, Schnittstellen zur Verfügung stellen und so gestaltet sein, dass eine spätere Erweiterung beziehungsweise Kopplung mit anderen Programmen möglich ist.

1.3. Gliederung

Der Aufbau dieser Arbeit verfolgt einen inkrementellen Ansatz und untergliedert sich in die nachfolgend beschriebenen Kapitel.
Zu Beginn werden in Kapitel 2 die theoretischen Grundlagen erläutert, welche elementar für das Verständnis dieser Arbeit sind. Dazu werden zunächst in Abschnitt 2.1 die CRG (Compliance Rule Graphs) näher beschrieben. Anschließend befasst sich Abschnitt 2.2 mit den eCRG (extended Compliance Rule Graphs), welche eine Erweiterung der CRG darstellen. Abschnitt 2.3 schließt dieses Kapitel ab, wobei ein Anwendungsbeispiel für eine per eCRG definierte Compliance Regel gegeben wird.
Darauf aufbauend werden in Kapitel 3 die Konzepte, welche für die Ausführung und die darauffolgende Bewertung einer eCRG notwendig sind besprochen. Die Grundlagen und Rahmenbedingungen hierfür werden in Abschnitt 3.1 beschrieben. Abschnitt 3.2 widmet sich den zu beachtenden Regeln, welche für die korrekte Ausführung einer eCRG sorgen. Anschließend wird in Abschnitt 3.3 die Logik für die Bewertung einer ausgeführten eCRG vorgestellt. Die notwendigen Schritte, welche für die Überwachung einer eCRG durchzuführen sind, sind Thema in Abschnitt 3.4. Dabei werden die einzelnen Schritte und deren Abfolge erläutert, um ausgehend von einem Prozess-Ablaufprotokoll eine Validierung einer eCRG vorzunehmen.
Anschließend wird in Kapitel 4 das Architekturkonzept besprochen, dass der späteren Implementierung zugrunde liegt. In Abschnitt 4.1 wird das Datenmodell definiert, auf dem die Implementierung aufbaut und im Anschluss daran wird in Abschnitt 4.2 der Entwurf für die grafische Oberfläche erläutert.
Kapitel 5 widmet sich schließlich der konkreten Implementierung. Zunächst werden

in Abschnitt 5.1 die technischen Grundlagen der Implementierung besprochen. In Abschnitt 5.2 sind verschiedene Aspekte mit Bezug auf die grafische Oberfläche zusammengefasst. Neben der Fensteransicht und der zur Verfügung stehenden Funktionen, wird in einem weiteren Abschnitt die JGraphX Bibliothek näher beschrieben. JGraphX ist eine Java Library, welche die Visualisierung von Graphen unterstützt. Die genaue Implementierung der notwendigen Algorithmen für die Ausführung und Überprüfung der Compliance Regeln ist Thema in Abschnitt 5.3.
Im Anschluss daran befasst sich Kapitel 6 mit verwandten Arbeiten, die inhaltlich mit dieser Arbeit im Zusammenhang stehen. Kapitel 7 fasst abschließend diese Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen und zukünftige Forschungsprojekte mit Themenbezug zu dieser Arbeit.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen erläutert, auf welche im weiteren Verlauf dieser Arbeit Bezug genommen wird. In dem nachfolgenden Abschnitt 2.1 wird zunächst der von CRG verfolgte Ansatz, zur Abbildung und Prüfung von Compliance Regeln, besprochen. Daran anschließend folgt eine Beschreibung der Möglichkeiten und der vorgesehenen Modellierungselemente von CRGs, bevor zuletzt noch näher auf die Ausführungssemantik von CRG eingegangen wird. In Abschnitt 2.2 werden die eCRG, eine Erweiterung der CRG, beschrieben. Die in den nachfolgenden Kapiteln behandelte Implementierung, einer Ausführungsengine für Compliance Regeln zur Überwachung der BPC, wird mittels eCRG realisiert und baut auf deren Konzepte auf. Im Anschluss an diese allgemeine Beschreibung der eCRG folgt eine genauere Darstellung der neu hinzugekommen Modellierungselemente, untergliedert in die einzelnen Perspektiven. Zum Abschluss dieses Kapitels wird in Abschnitt 2.3 ein Anwendungsbeispiel für eine per

eCRG definierte Compliance Regel gegeben, wobei nochmals die wichtigsten Konzepte von eCRG wiederholt werden.

2.1. Compliance Rule Graphs

Compliance Rule Graphs (CRG) sind eine grafische Notation zur Abbildung von Compliance Regeln und ermöglichen eine einfache Regelmodellierung durch Verknüpfung verschiedener Sprachelemente zu einem Graphen. Eingesetzt werden CRGs bei der BPC Überprüfung und stellen einen graph-basierten Ansatz zur Modellierung von Compliance Regeln dar.

2.1.1. Ansatz

Das Ziel bei der Entwicklung von CRGs war einen geeigneten Formalismus zur Definition von Compliance Regeln zu entwerfen, welcher eine automatisierte Compliance Prüfung ermöglicht. Dabei sollte eine ausreichende Ausdrucksstärke erreicht werden und es sollte keine Limitationen bezüglich komplexerer Regeln geben [LKRM+11].

Der von CRGs verfolgte Ansatz zur Modellierung von Compliance Regeln ist unabhängig von den jeweiligen Prozessmodellen. Damit sind per CRG definierte Compliance Regeln universell für verschiedene Prozesse einsetzbar, da sie nicht an ein bestimmtes Prozessmodell gebunden sind [LRMD10]. Diese Unabhängigkeit wird durch die Abbildung von Prozessinstanzen auf sogenannte Prozess-Ablaufprotokolle (*execution traces*) ermöglicht. Diese Ablaufprotokolle enthalten alle relevanten Informationen über eine Prozessinstanz, insbesondere geben sie Auskunft über deren aktuellen Zustand und den Prozessfortschritt. Die BPC von Prozessen lässt sich durch Ausführung der Ablaufprotokolle über die zu prüfenden CRGs und anschließender Prüfung der daraus resultierenden Compliance Zustände feststellen.

Compliance Rule Graphs basieren auf einer Graph-Struktur und bieten eine intuitive und anschauliche Art der Visualisierung [LKRM+11]. Dem Benutzer bleiben somit formale

Details der Regeldefinition (vgl. LTL[1]) verborgen und er kann sich voll und ganz auf die korrekte Abbildung der Compliance Regel konzentrieren. Aufgrund der grafischen Darstellung sind auch Neulinge in der Lage Compliance Regeln zu modellieren, ohne dabei Experte auf diesem Gebiet sein zu müssen und sich zuvor unter Umständen komplexes Fachwissen aneignen zu müssen. Im Vergleich zu logischen Ausdrücken und Formeln, ist es bei dieser Art der Visualisierung auch deutlich einfacher die relevanten Stellen zu identifizieren und einen Vergleich mit anderen Regeln durchzuführen. Ein weiterer Vorteil dieses Ansatzes ist: im Falle einer Verletzung der Compliance, lässt sich detailliert die Ursache dafür feststellen. Damit kann dem Nutzer nützliches Feedback über den Grund der Verletzung, beispielsweise durch Einblendung eines Warnhinweises, gegeben werden.

Da CRGs speziell für Compliance Regeln entwickelt wurden, war es naheliegend sich bei der Konzeption der CRGs an der typischen Struktur von Regeln zu orientieren [LRMKD11]. Jede CRG beschreibt deswegen eine `Wenn` $\rightarrow$ `Dann` Bedingung und besteht aus zwei entsprechende Abschnitten. Zum einen das *Antecedence-Pattern* mit dem sich eine Vorbedingung für eine Compliance Regel formulieren lässt, falls diese nur unter bestimmten Voraussetzungen gelten soll. Wenn es keine solche Vorbedingung für eine Regel gibt, bleibt das Antecedence-Pattern leer. Eine Compliance Regel wird somit nur dann aktiviert, wenn das Antecedence-Pattern erfüllt oder leer ist. In diesem Fall kommt nun das *Consequence-Pattern* zum Tragen. Dieses beschreibt die eigentlichen Forderungen einer Compliance Regel und drückt die zu überprüfende Bedingung aus. Zusammengefasst formuliert eine CRG somit folgendes: Wenn (bei einer Prozessausführung) das Antecedence-Pattern auftritt, dann muss auch das Consequence-Pattern erfüllt sein. Weiter beschreiben sowohl das Antecedence-Pattern, als auch das Consequence-Pattern, Bedingungen betreffs des Auftretens beziehungsweise Nicht-Auftretens bestimmter Events. Außerdem lässt sich eine vorgeschriebene Reihenfolge festlegen, in welcher die Events auftreten müssen. Dazu werden gerichtete Kanten zwischen zwei Event Knoten eingefügt, um Einschränkungen bezüglich des Vorgängers respektive Nachfolgers zum Ausdruck zu bringen. Wobei der von CRGs repräsentierte Graph immer einen azyklischen Graphen darstellt.

[1] Linear Temporal Logic

Des Weiteren unterstützen CRGs eine Mehrfach-Aktivierung. Dies ist notwendig, um alle Aktivierungen einer Compliance Regel während einer Prozessausführung zu registrieren. Dies kann der Fall sein, wenn ein und dasselbe Event mehrmals zu unterschiedlichen Zeiten auftritt.

Zuletzt sei noch erwähnt, das CRGs auch eine formale Spezifikation besitzen. Diese ist im Detail aber nicht weiter relevant für diese Arbeit. Deswegen wird an dieser Stelle darauf verzichtet und für weitere Details auf [Ly13, Kapitel 6] verwiesen.

2.1.2. Struktur & Modellierungselemente

Der Fokus für CRGs liegt bei der Modellierung von Compliance Regeln für Geschäftsprozesse. Bei einer genaueren Untersuchung solcher Regeln wurde eine charakteristische Form für deren Konstruktion identifiziert. Zunächst gibt es meist eine Art Vorbedingung. Damit lässt sich der Geltungsbereich einer Regel einschränken, falls diese nur unter bestimmten Bedingungen gelten soll. Darauf folgt die eigentliche Implikation der Regel, womit ein bestimmter Verlauf für die weitere Prozessausführung gefordert wird. Dieser Konstruktion folgend, besteht eine CRG aus einem *Antecedence-Pattern* (die Vorbedingung) und einem *Consequence-Pattern* (die Forderung). Jedes dieser beiden Pattern entspricht einem Subgraph innerhalb des CRG Gesamtgraphen. Dabei müssen weder die Subgraphen noch der gesamte Graph zwangsläufig zusammenhängend sein. Jedoch muss immer Zyklenfreiheit gewährleistet sein, also ein azyklischer Graph vorliegen.

Wie für Graphen typisch, werden Knoten und Kanten für die Spezifikation einer Compliance Regel verwendet. Damit kann die Ausführung bestimmter Prozessaktivitäten gefordert oder ausgeschlossen werden. Des Weiteren lässt sich eine bestimmte Reihenfolge festlegen, welche für die Prozessausführung gelten muss. Der Schwerpunkt von CRGs liegt dabei auf der Prozessperspektive.

Im Folgenden werden zunächst die Modellierungselemente dargestellt, welche von CRGs unterstützt werden. Im Anschluss daran wird anhand eines Beispiels die Struktur von CRGs und die Verwendung selbiger erläutert.

Knotentypen

Bei der Modellierung von CRGs kommen vier verschiedene Knotentypen zum Einsatz: ANTEOCC, ANTEABS, CONSOCC und CONSABS Knoten (*Nodes*). Abbildung 2.1 zeigt die Symbole, welche für die grafische Darstellung des jeweiligen Knotentyps verwendet werden. Dabei wird zum einen zwischen ANTECEDENCE und CONSEQUENCE Knoten unterschieden und zum anderen wird eine Unterscheidung zwischen OCCURRENCE und ABSENCE Knoten vorgenommen. Während Antecedence Knoten stets durch ein Rechteck repräsentiert werden, besitzen Consequence Knoten abgerundete Ecken. Zur Abhebung der Absence Knoten von den Occurrence Knoten, besitzen ANTEABS und CONSABS Knoten eine zusätzliche horizontale Linie im oberen Bereich.

Abbildung 2.1.: Knotentypen aus [Ly13]

Die Bedeutung der vier verschiedenen Knotentypen ist folgende:

- ein ANTEOCC Knoten fordert die Ausführung einer Aktivität, letztere muss also vorkommen um die Compliance Regel zu aktivieren
- ein ANTEABS Knoten fordert die Nichtausführung einer Aktivität, letztere darf also nicht vorkommen um die Compliance Regel zu aktivieren
- ein CONSOCC Knoten fordert die Ausführung einer Aktivität, letztere muss also vorkommen um die Compliance Regel zu erfüllen
- ein CONSABS Knoten fordert die Nichtausführung einer Aktivität, letztere darf also nicht vorkommen um die Compliance Regel zu erfüllen

Relationen

Mittels Kanten (*Edges*) werden Relationen zwischen zwei Knoten ausgedrückt. Diese sind notwendig um die zuvor beschriebenen Knoten miteinander zu verknüpfen und Bedingungen zwischen ihnen auszudrücken. Für CRGs stehen dazu zwei verschiedene Relationen zur Verfügung. In Abbildung 2.2 sind diese dargestellt.

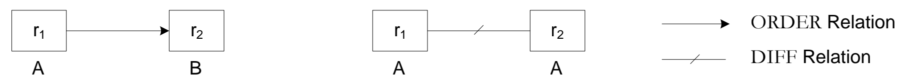


Abbildung 2.2.: Relationen aus [Ly13]

Zunächst die ORDER Relation, welche durch einen Pfeil repräsentiert wird. Sie entspricht einer gerichteten Kante und drückt eine bestimmte Ausführungsreihenfolge für die zwei verbundenen Aktivitäten aus. In dem Beispiel aus Abbildung 2.2 bedeutet dies, dass Aktivität A zwingend vor Aktivität B ausgeführt sein muss um eine Übereinstimmung mit dem Pattern zu erzielen. Anders ausgedrückt: A musst beendet sein bevor B gestartet werden kann, womit A Vorrang hat vor B.

Die DIFF Relation drückt aus, dass die zwei Knoten, welche durch die Kante verbunden sind, nicht mit derselben Aktivitätsausführung belegt werden dürfen. Dies ist notwendig um zum Ausdruck zu bringen, dass es zwei unterschiedliche Ausführungen des gleichen Aktivitätstyps geben soll. Beispielsweise zwei voneinander unabhängige Labortests. Somit sind DIFF Relationen nur zwischen Knoten sinnvoll, die ein identisches oder zumindest überlappendes Profil aufweisen. Denn nur in diesem Fall würde eine Aktivitätsausführung zu beiden Knoten passen.
Würde es keine DIFF Relation zwischen den Knoten r_1 und r_2 in dem Beispiel aus Abbildung 2.2 geben, würde *eine* passende Aktivitätsausführung reichen und beide Knoten wären aufgrund desselben Events erfüllend belegt. Wenn man nun aber eigentlich zum Ausdruck bringen wollte, dass die Knoten für zwei verschiedene Ausführungen stehen, muss eine DIFF Relation eingefügt werden. Aufgrund ihrer Symmetrie entspricht die DIFF Relation einer ungerichteten Kante.

Eigenschaften von Knoten

Jeder CRG Knoten besitzt bestimmte Eigenschaften, wodurch die geforderte Aktivitätsausführung genauer spezifiziert wird. Die möglichen Eigenschaften von CRG Knoten sind in Abbildung 2.3 dargestellt. Bei der Überprüfung von Prozess-Ablaufprotokollen werden diese Eigenschaften als Bedingungen aufgefasst. Anhand der Bedingungen wird nach passenden Ausführungen gefiltert und nur wenn alle Bedingungen zutreffen handelt es sich um ein passendes Event für den jeweiligen Knoten.

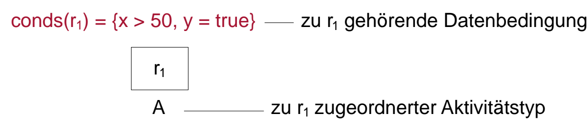


Abbildung 2.3.: Datenbedingungen und Knoteneigenschaften aus [Ly13]

Folgende Eigenschaften können einem CRG Knoten zugewiesen werden:

- Ein *Aktivitätstyp*, womit zum Ausdruck gebracht wird, dass nur Aktivitätsausführungen dieses bestimmten Typs dem jeweiligen Knoten zugeordnet werden können.
- Ein *Bezeichner* für den Knoten, um einen spezifischen Knoten in einem existierenden Prozessmodell zu referenzieren. Dadurch wird aus der CRG eine prozessspezifische Compliance Regel, die sich nahtlos in das existierende Prozessmodell integrieren lässt.
- Eine Menge von *Datenbedingungen* um einen bestimmten Datenwert für die Parameter der zugewiesenen Aktivität zu fordern. Mehrere Datenbedingungen werden als Konjunktion betrachtet, weswegen alle Bedingungen erfüllt sein müssen um eine passende Belegung für den entsprechenden Knoten zu erhalten.

Während der Bezeichner und die Datenbedingung optionale Eigenschaften darstellen, muss jedem CRG Knoten genau eine Aktivität zugeordnet sein.

Aussage und Struktur von CRGs

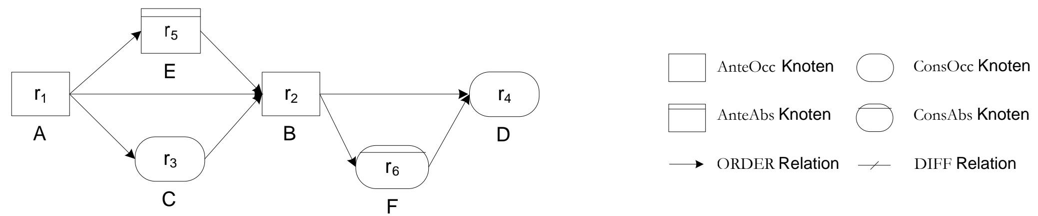
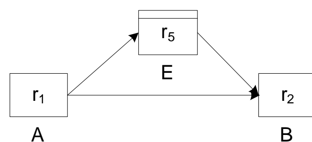


Abbildung 2.4.: Beispiel einer CRG aus [Ly13]

In Abbildung 2.4 ist ein Beispiel für eine CRG abgebildet. Im Folgenden wird die Compliance Regel beschrieben, welche durch die CRG modelliert wird. Abbildung 2.5 zeigt die Aufsplittung der Beispiel CRG in ihr *Antecedence* und *Consequence*-Pattern. Das

(a) Antecedence-Pattern



(b) Consequence-Pattern

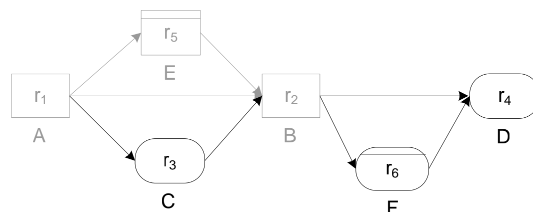


Abbildung 2.5.: Antecedence & Consequence-Pattern einer CRG

Antecedence-Pattern in Abbildung 2.5a stellt die Vorbedingung dar, welche zutreffen muss damit die Regel aktiviert wird. In diesem Fall wird die Ausführung einer Aktivität vom Typ A verlangt. Des Weiteren soll auf A eine Aktivität vom Typ B folgen, wobei zwischen A und B kein E vorkommen darf. Die Forderung der Compliance Regel entspricht dem Consequence-Pattern in Abbildung 2.5b. Wenn die CRG nun aufgrund einer Übereinstimmung des Antecedence-Patterns aktiviert wurde, muss folgendes gelten um die Compliance Regel zu erfüllen. Zum einen muss eine Aktivität vom Typ C zwischen A und B ausgeführt werden und außerdem muss auf die Ausführung von B noch ein D folgen, wobei dazwischen kein F erlaubt ist.

Zusammengefasst modelliert die CRG in Abbildung 2.4 damit folgenden Sachverhalt: Falls A vor B ausgeführt wird und dazwischen kein E vorkommt, dann soll zwischen A und B ein C ausgeführt werden und auf B soll ein D folgen, ohne ein F dazwischen.

In Abbildung 2.6 sind die verschiedenen Zuordnungsmöglichkeiten für die zuvor beschriebene CRG und das Ablaufprotokoll $<A_1, C_1, B_1, E_1, D_1, A_2, B_2, F_1, D_2>$ abgebildet. Sowohl A als auch B sind in dem Ablaufprotokoll zweimal vertreten. Somit reicht es nicht

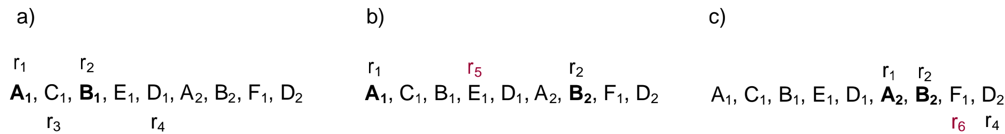


Abbildung 2.6.: Beispielhafte Zuordnung eines Ablaufprotokolls aus [Ly13]

aus, nur eine Ausführung von A gefolgt von B zu untersuchen um eine mögliche Verletzung der Compliance Regel auszuschließen. Es müssen alle möglichen Kombinationen überprüft werden um eine korrekte Validierung der Compliance Regel vorzunehmen. Folgende drei Zuordnungen sind dabei möglich:

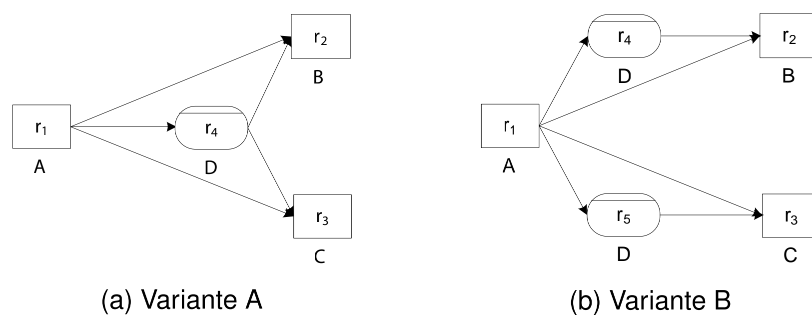
a) Mit der Zuordnung $A_1 \rightarrow r_1$ und $B_1 \rightarrow r_2$ ist das Antecedence-Pattern erfüllt und damit die Regel aktiviert. Denn es wurde auch keine Aktivität vom Typ E zwischen A_1 und B_1 ausgeführt. Des Weiteren finden wir in diesem Fall auch eine passende Belegung für das Consequence-Pattern. C_1 wurde zwischen A_1 und B_1 ausgeführt und D_1 folgt auf B_1, wobei kein F dazwischen vorkommt.
$\longrightarrow$ die Compliance Regel ist bis dato erfüllt

b) Für die Zuordnung $A_1 \rightarrow r_1$ und $B_2 \rightarrow r_2$ bleibt die Regel deaktiviert. Denn aufgrund der Ausführung von E_1 zwischen A_1 und B_2 ist das Antecedence-Pattern verletzt.
$\longrightarrow$ betrifft die Compliance Regel nicht und hat deswegen keine Auswirkungen

c) Zuletzt gibt es noch die Ausführung A_2 gefolgt von B_2. Bei dieser Zuordnung liegt wieder eine Aktivierung der Regel vor, denn das Antecedence-Pattern wird dadurch erfüllt. Jedoch existieren keine passenden Aktivitäten um das Consequence-Pattern zu erfüllen. Denn es gibt zwar die Ausführung D_2, welche wie gefordert auf B_2 folgt, aber dazwischen wurde auch F_1 ausgeführt, was nicht erlaubt ist. Außerdem wurde auch kein C zwischen A_2 und B_2 ausgeführt und damit existiert auch keine passende Belegung für r_3.
$\longrightarrow$ die Compliance Regel ist verletzt für dieses Ablaufprotokoll

Details bei der Modellierung

Nachdem im vorangegangenen Abschnitt ein erstes Beispiel für eine CRG erläutert wurde, werden im Folgenden wichtige Details und dadurch resultierende Unterschiede bei der Modellierung von CRGs beleuchtet. Anhand dieses Beispiels soll außerdem die enorme Ausdrucksstärke von CRGs und die Vielzahl an Möglichkeiten bei der Modellierung von Compliance Regeln verdeutlicht werden.
Abbildung 2.7 zeigt zwei verschiedene CRGs, die sich jedoch beide ähneln und nur geringfügig voneinander abweichen. Bei Variante A, welche in Abbildung 2.7a dargestellt



(a) Variante A (b) Variante B

Abbildung 2.7.: Details & Unterschiede bei der CRG Modellierung aus [Ly13]

ist, wird die Ausführung einer Aktivität vom Typ D durch einen ConsAbs Knoten ausgeschlossen. Für die Ausführungsreihenfolge gilt dabei folgendes: Eine Ausführung von D ist nicht erlaubt, wenn sie auf A folgt und vor B **und** C ausgeführt wird. Durch Variante B, siehe Abbildung 2.7b, wird ein leicht anderer Sachverhalt gefordert. Es ist zwar ebenfalls die Ausführung von D verboten, jedoch wird dies durch zwei verschiedene ConsAbs Knoten spezifiziert. Einer befindet sich zwischen A und B, der andere zwischen A und C. In dieser Variante darf D also **weder** zwischen A und B **noch** zwischen A und C ausgeführt werden. Während in Variante A also beides gleichzeitig verlangt wird, schließt Variante B beide getrennt voneinander aus. Diesem Unterschied sollte man sich bewusst sein, um nicht intuitiv eine CRG zu modellieren, die nicht dem eigentlich geforderten Sachverhalt entspricht.

In der Tabelle 2.1 sind einige verschiedene Ablaufprotokolle und ihre Compliance Zustände für die zwei zuvor besprochenen CRGs abgebildet.

Ablaufprotokolle	**Variante A**		**Variante B**	
$p_1: < A, B, C >$	✓	aktiviert & erfüllt	✓	aktiviert & erfüllt
$p_2: < A, C, B >$	✓	aktiviert & erfüllt	✓	aktiviert & erfüllt
$p_3: < B, C, A >$	(✓)	nicht aktiviert	(✓)	nicht aktiviert
$p_4: < B, A, C >$	(✓)	nicht aktiviert	(✓)	nicht aktiviert
$p_5: < D, A, C, B >$	✓	aktiviert & erfüllt	✓	aktiviert & erfüllt
$p_6: < A, C, B, D >$	✓	aktiviert & erfüllt	✓	aktiviert & erfüllt
$p_7: < A, C, D, B >$	✓	aktiviert & erfüllt	↯	aktiviert & verletzt
$p_8: < A, B, D, C >$	✓	aktiviert & erfüllt	↯	aktiviert & verletzt
$p_9: < A, D, C, B >$	↯	aktiviert & verletzt	↯	aktiviert & verletzt

Tabelle 2.1.: Beispielhafte Ablaufprotokolle und ihre Compliance Zustände für die beiden CRGs aus Abbildung 2.7

Weitere Details bezüglich der syntaktischen Korrektheit von CRGs und möglicher Fehlerquellen bei der Modellierung sind in [Ly13, Kapitel 6] beschrieben. Außerdem werden dort auch die *CRG composites* definiert. Damit können Compliance Regeln mit mehreren disjunkten Consequence-Pattern modelliert werden.

2.1.3. Ausführungssemantik

Dieser Abschnitt beschäftigt sich mit der Ausführungssemantik von CRGs, welche notwendig ist um Compliance Regeln automatisiert zu überprüfen. Im vorangegangen Kapitel wurden CRGs manuell verifiziert, siehe Seite 15. Die Vorgehensweise dabei war, anhand des Ablaufprotokolls eine Übereinstimmung der Graph-Pattern zu erzielen. Dieser Ansatz soll nun automatisiert werden, sodass CRGs auf effiziente Weise maschinell geprüft werden können. Dabei soll das Ablaufprotokoll der Reihenfolge nach abgearbeitet werden und das aktuelle Event, wenn möglich einem passenden CRG Knoten zugeordnet werden. Dieser Ansatz steht damit in direktem Zusammenhang mit der jeweiligen CRG und beruht auf deren Graph-Struktur. Dies stellt sogleich den wesentlichen Unterschied zu anderen Konzepten, die eine automatisierte Überwachung

von Compliance Regeln ermöglichen, dar. Gegenüber Ansätzen, denen ein Automat zu Grunde liegt[2], bietet dies folgende Vorteile [Ly13]:

- Die Compliance Überprüfung findet nicht innerhalb einer Black-Box statt. Stattdessen lässt sich der aktuelle Compliance Zustand direkt anhand der Markierung des CRG Graphen nachvollziehen und ableiten.
- Zwischenergebnisse werden unterstützt. Aufgrund der Ausführungsstrategie von CRGs können die Markierungen zu jeder Zeit betrachtet und ausgewertet werden.
- Des Weiteren ist die Interpretation der Compliance Zustände einfacher, da die Zuordnung der Markierungen sofort erkennbar ist. Bei logischen Formeln ist es hingegen oft schwer, Teilausdrücke den einzelnen Bedingungen der Compliance Regel zuzuordnen.
- Durch Hinzufügen neuer Knoten und Kanten lässt sich eine CRG auf relativ einfache Weise abändern und damit bei Regeländerungen anpassen.
- Außerdem kann bei CRGs die Markierung direkt dazu verwendet werden, den aktuellen Compliance Zustand zu veranschaulichen. Im Falle einer Verletzung der Compliance Regel, können die Ursachen dafür gesondert gekennzeichnet werden.

Der Compliance Zustand von Prozessinstanzen respektive derer Ablaufprotokolle wird durch sogenannte COMPLIANCESTATES repräsentiert. Damit werden verschiedene Markierungen der CRG zusammengefasst, die während der Überprüfung erreicht wurden. Die einzelnen Markierungen dienen dazu, die CRG Knoten mit einem Ausführungszustand zu versehen, um so bereits aufgetretene Ausführungen festzuhalten. Durch Analyse und Auswertung des finalen Compliance Zustands lässt sich schließlich das Resultat der Compliance Prüfung bestimmen. Dazu werden die einzelnen Markierungen betrachtet und es wird überprüft, ob für jede aktivierte Markierung auch eine Erfüllung des Consequence-Patterns existiert. Wenn dies der Fall ist, werden die Forderungen der Compliance Regel eingehalten und die Regel ist damit erfüllt. Andernfalls liegt eine Verletzung der Regel vor und es gibt mindestens eine aktivierte Markierung, für die kein passendes Consequence-Pattern existiert.

[2] beispielsweise Compliance Überwachung mittels LTL Compliance Regeln

Zwei weitere relevante Begriffe bei der Ausführung von CRGs sind MARKSTRUCTURES und EXMARKS. Letztere basieren auf der Graph-Struktur von CRGs und stellen eine Markierung der CRG dar. Damit entsprechen EXMARKS (abgk. `m`) einer möglichen Zuordnung, die aufgrund des Ablaufprotokolls und der CRG möglich ist. MARKSTRUCTURES (abgk. `ms`) fassen mehrere dieser EXMARKS zu einer Gruppe zusammen, wobei alle EXMARKS dieselbe Markierung des Antecedence-Patterns aufweisen müssen. Ein COMPLIANCESTATE beinhaltet wiederum mehrere solche MARKSTRUCTURES.

Beispielhafte Überprüfung einer Compliance Regel

Im Folgenden wird anhand eines Beispiels die Ausführungssemantik von CRGs schematisch veranschaulicht. Auf dieser basiert die später in der Implementierung umgesetzte Ausführungssemantik für eCRG. Abbildung 2.8 zeigt, wie sich die Compliance Zustände für ein Ablaufprotokoll (*execution trace*) ableiten lassen. Dabei wird das Prozess-Ablaufprotokoll, bestehend aus den Events e_1 - e_9, der Reihe nach ausgeführt. Die zu überprüfende Compliance Regel R fordert für die Prozessausführung folgendes: Auf die Ausführung einer Aktivität vom Typ A muss immer eine Aktivität B folgen und wenn B erst einmal ausgeführt ist, darf keine Aktivität vom Typ C mehr folgen. Diese Compliance Regel wird nun Schritt für Schritt über die Events e_1 - e_9 ausgeführt[3]:

$\underline{init}$ Die Ausführung beginnt mit der Initialisierung des Anfangszustands. Der initiale Compliance Zustand entspricht immer einer MARKSTRUCTURE, die eine EXMARK Markierung enthält. Diese Markierung ist zu Beginn noch leer, da auch noch keine Aktivitätsausführungen beobachtet wurden.

$\underline{e_1 = (Start, A, 1)}$ Die CRG R verlangt für den Knoten r_1, dem die Aktivität A zugewiesen ist, keine Vorgänger. Deswegen kann A direkt gestartet werden und der Knoten r_1 wird entsprechend markiert. Damit weitere Ausführungen der Aktivität A ($\rightarrow e_9$) erfasst werden können, wird eine neue MARKSTRUCTURE von ms_1 abgeleitet und die Anfangsmarkierung bleibt unverändert erhalten.

[3]Die genauen Details und die Regeln, die bei der Ausführung der einzelnen Events anzuwenden sind, werden detailliert in Kapitel 3 besprochen. Mittels dieser Regeln werden die Markierungen und damit die Compliance Zustände schrittweise aktualisiert. Für jedes Event, das im Prozess-Ablaufprotokoll enthalten ist, wird eine solche Aktualisierung durchgeführt.

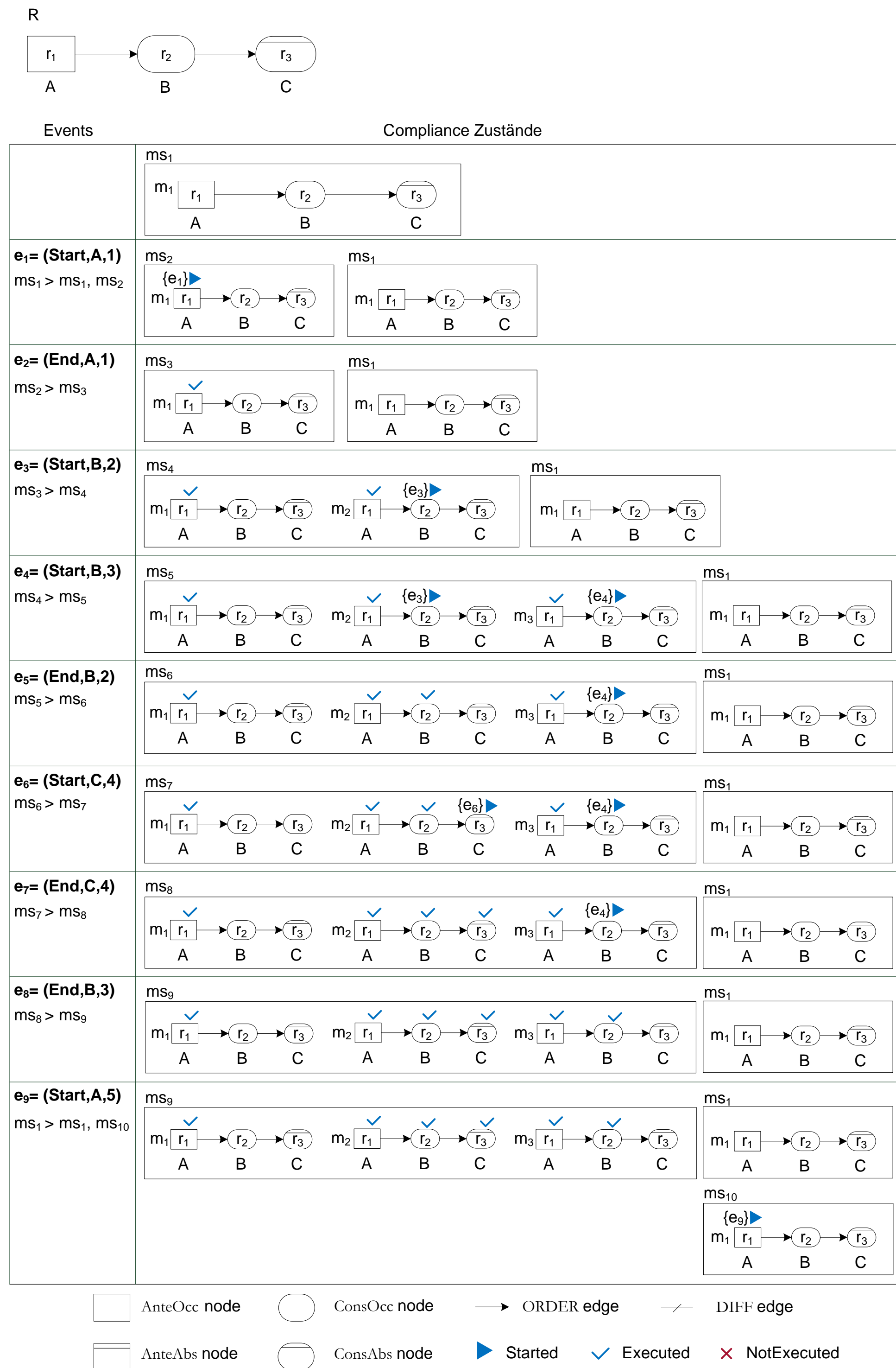


Abbildung 2.8.: Ausführung und Markierung einer CRG aus [Ly13]

$e_2 = (End, A, 1)$ Um das Ende von A_1 entsprechend zu vermerken wird in allen Markierungen, in denen A als gestartet markiert wurde und das zugeordnete Event dieselbe ID aufweist wie das aktuelle Event, die Ausführungsmarkierung des jeweiligen Knoten zu *Executed* aktualisiert. Da A nur in der Markierung $ms_2 \to m_1$ den Ausführungszustand *Started* besitzt, ist dies in diesem Fall die einzige Markierung, die angepasst werden muss. Die daraus resultierende Markierung ms_3 entspricht nun einer Aktivierung der Compliance Regel.

$e_3 = (Start, B, 2)$ Da nur Ausführungen von B relevant sind, die nach A kommen, wird B nur in den Markierungen auf *Started* gesetzt, bei denen eine Ausführung von A festgehalten wurde. Aus demselben Grund wie bereits bei e_1 (Möglichkeit zum Starten weiterer Vorkommen von B) bleibt m_1 unverändert bestehen und es wird stattdessen eine neue Markierung m_2 hinzugefügt um das gestartete B abzubilden.

$e_4 = (Start, B, 3)$ Mit der gleichen Semantik, wie bei dem Event zuvor, wird eine neue Markierung m_3 von m_1 abgeleitet und zu ms_5 hinzugefügt. Damit wird die Ausführung eines weiteren B's (gefolgt auf ein A) festgehalten.

$e_5 = (End, B, 2)$ Bei der Ausführung des Events e_5 wird die Markierung m_2 entsprechend angepasst. Denn im Gegensatz zu m_3, stimmt bei m_2 die ID mit der des verantwortlichen Start Events überein. Damit wurde ein passendes End Event gefunden und r_2 kann somit als *Executed* markiert werden.

$e_6 = (Start, C, 4)$ Die Aktivität C kann nur in der Markierung m_2 gestartet werden, da nur hier die notwendigen Vorgänger bereits ausgeführt wurden. Denn die Compliance Regel besagt, dass *nach* einem B kein C mehr kommen darf. Somit sind nur Markierungen mit ausgeführtem B relevant, was in diesem Fall nur auf m_2 zutrifft. Deswegen wird auch nur dort das Starten einer Aktivität vom Typ C vermerkt[4].

$e_7 = (End, C, 4)$ Dieses Event passt zu der im Schritt zuvor gestarteten Aktivität C_4 und signalisiert deren Ende. Deswegen wird der Ausführungszustand von r_3 in der Markierung m_2 entsprechend aktualisiert.

[4]Hierbei zeigt sich auch bereits ein Unterschied zwischen der Ausführung von Occurrence und Absence Knoten bei CRGs. Während beim Starten von Occurrence Knoten die ursprüngliche Markierung immer unverändert erhalten bleibt, kann bei Absence Knoten die Markierung direkt aktualisiert werden. Denn bei letzteren genügt es, feststellen zu können, ob eine nicht erlaubte Aktivität ausgeführt wurde oder nicht. Verschiedene Ausführungsvarianten sind dabei nicht von Bedeutung.

$\underline{e_8 = (End, B, 3)}$ Mit diesem Event findet auch die, durch das Event e_4 gestartete, Aktivität B_3 ihr Ende. Folglich wird die Markierung des Knoten r_2 in m_3 angepasst und aus *Started* wird *Executed*.

$\underline{e_9 = (Start, A, 5)}$ Zuletzt wurde noch eine weitere Ausführung der Aktivität A beobachtet. Analog zu e_1 wird deswegen eine neue Markierung von der Anfangsmarkierung ms_1 abgeleitet, wobei in dieser die Aktivität A als gestartet vermerkt ist.

$\underline{Ergebnis}$ Die CRG R wird durch das Ablaufprotokoll erfüllt. Für jede Aktivierung der Compliance Regel ($\rightarrow ms_9$) existiert mindestens eine erfüllende Markierung des Consequence-Patterns ($\rightarrow m_3$). Dies deckt sich auch mit einer Analyse des Ablaufprotokolls per Hand. Denn auf jede Ausführung einer Aktivität vom Typ A ($\rightarrow A_1$) folgt eine Aktivität B ($\rightarrow B_3$), ohne das darauf wiederum eine Aktivität C folgen würde.

2.2. Extended Compliance Rule Graphs

Die *extended Compliance Rule Graphs* (eCRG) sind eine grafische Sprache zur Modellierung von Compliance Regeln für Geschäftsprozesse. Die im Abschnitt 2.1 vorgestellten *Compliance Rule Graphs* (CRG) stellen die Grundlage dafür dar. Mit den eCRG werden neue Modellierungselemente dem Konzept der CRG hinzugefügt. Diese Erweiterungen sind notwendig um Regeln modellieren zu können, die über die klassische Kontrollfluss-Perspektive hinausgehen [KRL+13b]. Folgende Perspektiven werden dabei von eCRG unterstützt: Kontrollfluss, Interaktion, Zeit, Daten und Ressourcen. Diese fünf Perspektiven wurden bei einer Fallstudie identifiziert und als essentiell erachtet, um typische Compliance Regeln aus der Realwelt abbilden zu können [Sem13]. Damit wurde gezeigt, dass es nicht ausreicht nur die Kontrollfluss-Perspektive zu unterstützen, sondern auch Zeit-, Daten-, Ressourcen- und Interaktions-Aspekte bei der Modellierung von Compliance Regeln zu beachten sind. Die Kontrollfluss-Perspektive wird im folgenden auch als Prozess-Perspektive bezeichnet. Während sich die CRG hauptsächlich auf den korrekten Kontrollfluss von Prozessen konzentrieren, werden von eCRG alle fünf zuvor genannten Perspektiven abgedeckt.

Zusätzlich bieten eCRG neben der klassischen lokalen Ansicht eine globale Ansicht, um Regeln zwischen verschiedenen Geschäftspartnern abzubilden. Damit werden organisationsübergreifende Prozess-Szenarien, sogenannte *cross-organizational process scenarios*, unterstützt. Diese beiden Ansichten entsprechen dem Kollaborations- und dem Choreographie-Diagramm von BPMN 2.0 und drücken somit zwei unterschiedliche Sichtweisen auf den Prozess aus.

In den folgenden Abschnitten werden die verschiedenen Perspektiven von eCRG besprochen. Neben der grafischen Notation, werden dabei die neu hinzugekommenen Modellierungselemente beschrieben. Dabei ist anzumerken, dass eCRG eine weitere Kategorie von Elementen gegenüber CRG besitzen. Neben Knoten und Kanten, den typischen Graph-Elementen, gibt es auch sogenannte *Attachments*. Damit lassen sich Bedingungen definieren, die an Knoten und Kanten angehängt werden können. Von nun an werden die Begriffe *Node* und *Connector* äquivalent für Knoten und Kante verwendet. Ein weiterer Unterschied zu CRGs sind die *Instance Nodes*. Damit können bestimmte Knoten definiert werden, die unabhängig von der konkreten Regel existieren und bei der Ausführung nicht weiter zu spezifizieren sind. Deswegen sind *Instance Nodes* weder Teil des Antecedence noch des Consequence-Patterns. Ein spezifisches Datum oder beispielsweise ein bestimmter Mitarbeiter können damit abgebildet werden.
Eine formale Spezifikation der eCRG Sprachelemente findet sich in dem technischen Bericht: *On the Formal Semantics of the Extended Compliance Rule Graph* [KRL$^+$13a].

2.2.1. Prozess-Perspektive

Die eCRG Elemente zur Modellierung der *Prozess-Perspektive* sind in Abbildung 2.9 dargestellt. Damit lässt sich der Kontrollfluss festlegen, der für Prozessinstanzen gefordert wird. Zu diesem Zweck gibt es vier verschiedene *Task Nodes*, je zwei für das Antecedence und das Consequence-Pattern. Des Weiteren wird, wie bei den CRG, zwischen Absence und Occurrence Knoten unterschieden. Damit ergeben sich die vier bereits von den CRG bekannten Knotentypen: *Antecedence Occurrence* (AO), *Antecedence Absence* (AA), *Consequence Occurrence* (CO) und *Consequence Absence* (CA). Die Semantik der Knoten stimmt weitgehend mit der von CRG Knoten überein. Jedoch

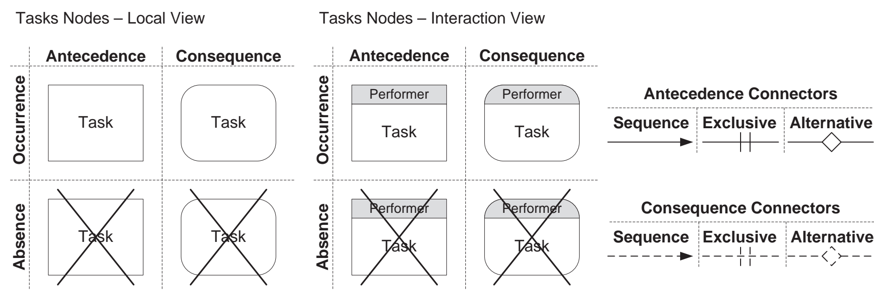


Abbildung 2.9.: eCRG Elemente für die Prozess-Perspektive aus [KRL+13b]

unterscheiden sich die *Task Nodes* leicht in ihrer grafischen Darstellung von den CRG Knoten. Consequence Knoten besitzen zwar ebenfalls eine abgerundete Umrandung, aber bei der Darstellung der Absence Knoten gibt es einen Unterschied: In eCRG Graphen sind sie durch zwei diagonal und überkreuzt verlaufende Linien gekennzeichnet. Außerdem werden die Knoten immer direkt mit dem entsprechenden Tasktyp beschriftet. Wie bereits von den CRG bekannt, drücken die jeweiligen Knoten aus, ob ein bestimmter Task nicht ausgeführt werden darf beziehungsweise muss und ob dies für die Aktivierung der Regel notwendig ist oder eine Konsequenz der Regel darstellt.
Zur Unterstützung einer globalen Ansicht gibt es die vier *Task Nodes* auch noch in einer speziellen Version für die *Interaction View*. Dabei wird zusätzlich noch der *Performer* des Tasks mit angegeben.
Mit den *Sequence Flow Connectoren* wird eine bestimmte Reihenfolge für die Ausführung der Tasks festgelegt. Neu dabei ist, dass zwischen *Antecedence* und *Consequence* Connectoren unterschieden wird. Ohne diese Unterscheidung wäre es beispielsweise nicht möglich, eine Regel zu modellieren, die als Konsequenz eine bestimmte Ausführungsreihenfolge für zwei Antecedence Knoten verlangt. In diesem Zusammenhang sein noch angemerkt, dass eine parallele Ausführung zweier Tasks genau dann vorliegt, wenn die beiden Knoten durch keinen *Sequence Flow Connector* miteinander verbunden sind. Ein weiterer Unterschied zu den CRG besteht darin, dass bei *Task Nodes* zwischen $Start - Start$, $Start - Ende$, $Ende - Start$ und $Ende - Ende$ Beziehungen unterschieden wird. Dadurch können noch differenzierter Bedingungen bezüglich der Ausführungsreihenfolge von Tasks definiert werden. Um sich auf den Startzeitpunkt zu beziehen, wird der Connector mit der linken Seite eines Tasks verbunden. Für den

Endzeitpunkt wählt man entsprechend die rechte Seite.
Neu hinzugekommen sind außerdem zwei weitere Connector-Typen. Ein *Exclusive Connector* fordert die Ausführung von genau einem der beiden miteinander verbundenen Tasks. Mit *Alternative Connectoren* wird hingegen ausgedrückt, dass mindestens einer der Tasks ausgeführt werden muss, die miteinander verbunden sind. Für beide Connectoren gilt jedoch, dass Knoten nicht pattern-übergreifend, sondern nur innerhalb des Antecedence oder Consequence-Patterns miteinander verbunden werden können.

2.2.2. Interaktions-Perspektive

Neu hinzugekommen ist die *Interaktions-Perspektive* für eCRG. Durch diese Perspektive lässt sich der Nachrichtenaustausch bei Prozessen definieren und wenn notwendig entsprechend beschränken. Die Elemente zur Modellierung dieser Perspektive sind

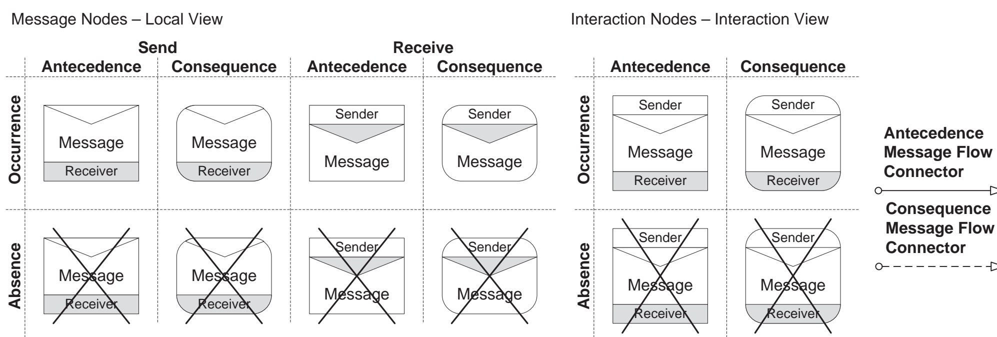


Abbildung 2.10.: eCRG Elemente für die Interaktions-Perspektive aus [KRL$^+$13b]

in Abbildung 2.10 abgebildet. Für die lokale Ansicht stehen zwei verschiedene Klassen von Knoten zur Verfügung. Zum einen *Message Nodes* der Klasse $Send$, um das Versenden einer Nachricht abzubilden. Zum anderen $Receive$ Knoten, mit denen der Empfang einer Nachricht modelliert wird. Unabhängig vom jeweiligen Ereignis, besitzt jeder Messageknoten (*Message Node*) neben einem Nachrichtentyp noch ein Feld für den Kommunikationspartner. Damit wird der Empfänger beziehungsweise Absender der Nachricht genauer spezifiziert.
Mit einem *Message Flow Connector* wird der Nachrichtenfluss einer bestimmten Nachricht von einem $Send$ zu einem $Receive$ Knoten gekennzeichnet.

Um eine Compliance Regel mit mehreren beteiligten Geschäftspartnern abbilden zu können, werden die *Interaction Nodes* benötigt. Diese besitzen sowohl einen Absender, als auch einen Empfänger und dienen der Unterstützung der globalen Ansicht (*Interaction View*) bei der CRG Modellierung.
Jeder Messageknoten ist zudem eindeutig dem Antecedence oder Consequence-Pattern zuzuordnen. Des Weiteren beschreibt jeder Knoten das Auftreten beziehungsweise Nicht-Auftreten eines bestimmten Message-Ereignisses. Damit ergeben sich die bereits von den *Task Nodes* bekannten Knotentypen: AO, AA, CO und CA.

2.2.3. Zeit-Perspektive

Die Elemente aus Abbildung 2.11 dienen der Modellierung der *Zeit-Perspektive*. Zeitpunkte in Compliance Regeln werden durch sogenannte *Point in Time* Knoten definiert. Es wird wieder zwischen *Antecedence* & *Consequence* und zwischen *Occurrence* & *Absence* bei den Knoten unterschieden. Jedoch gibt es bei Point in Time Knoten noch einen weiteren Typ, mit dem sich *bestimmte* Zeitpunkte spezifizieren lassen. Dieser neue Knotentyp gehört der Klasse der *Instance* Knoten an. Das Pattern, dem diese Knoten zugeordnet sind, wird dementsprechend auch als Instance-Pattern bezeichnet.

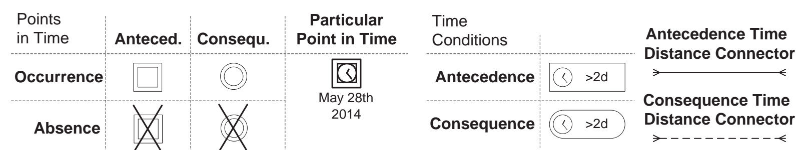


Abbildung 2.11.: eCRG Elemente für die Zeit-Perspektive aus [KRL+13b]

Des Weiteren stehen *Time Condition* Attachments zur Verfügung. Damit lässt sich einerseits die Laufzeit von Tasks beschränken, indem eine Time Condition an eine Task Node angehängt wird. Außerdem kann durch eine Time Condition in Verbindung mit einem Sequence oder Message Flow Connector ein gewisser zeitlicher Abstand zwischen zwei Knoten gefordert werden. Mit der Wahl des Patterns, modelliert die Time Condition entweder eine Voraussetzung für die Aktivierung einer Regel oder stellt eine Bedingung für die Ausführung dar.

Neu sind auch die *Time Distance Connectoren*, denen zwingend eine Time Condition zuzuweisen ist. Durch einen solchen Connector wird ebenfalls zum Ausdruck gebracht, dass zwischen den beiden verbundenen Knoten eine gewisse zeitliche Distanz liegen muss. Im Unterschied zu einem Sequence Flow Connector wird dabei jedoch keine Einschränkung bezüglich der Ausführungsreihenfolge gemacht.

2.2.4. Daten-Perspektive

Abbildung 2.12 zeigt unter anderem die eCRG Elemente für die Daten-Perspektive. Es stehen zwei verschiedene Knoten zur Modellierung von Datencontainern (*Data Container*) und Datenobjekten (*Data Objects*) zur Verfügung. *Data Container* entsprechen einem globalen Datenspeicher und repräsentieren Prozessdatenelemente. *Data Objects* verweisen hingegen auf einen bestimmten Datenwert oder auf eine Objektinstanz. Beide Knotentypen können wiederum Teil des Antecedence oder des Consequence-Patterns sein. Wenn mit ihnen ein bestimmter Datencontainer oder ein bestimmtes Datenobjekt referenziert wird, sind sie dem Instance-Pattern zuzuordnen.
Mittels *Data Flow Connectoren* wird der Datenfluss zu und von den Datenknoten dargestellt. Damit wird festgelegt, welche Tasks lesend oder schreibend auf welches Datenobjekt oder welchen Datencontainer zugreifen.

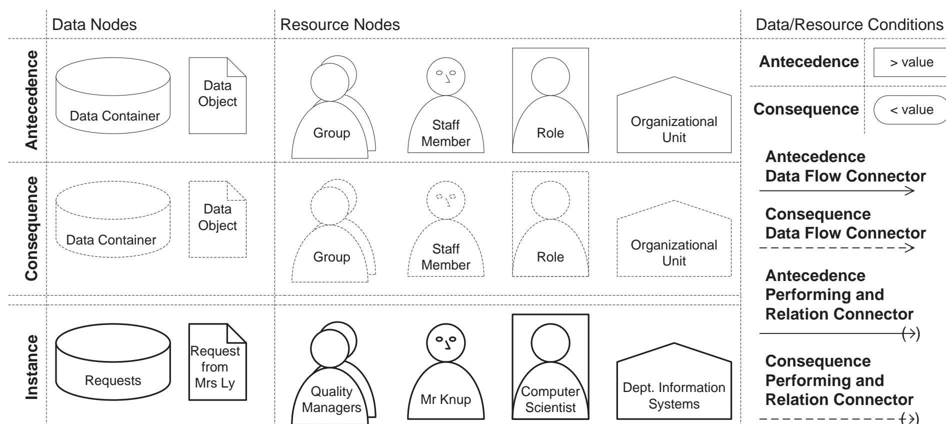


Abbildung 2.12.: eCRG Elemente für die Daten und Ressourcen-Perspektive [KRL+13b]

In Analogie zu den Time Conditions für die Zeit-Perspektive, gibt es an dieser Stelle die *Data Condition* Attachments. Solch ein Attachment kann entweder an einen Datenknoten oder an eine Datenflusskante angehängt werden. Dadurch wird zum Ausdruck gebracht, dass bei der Ausführung nur Werte zulässig sind, welche diese Bedingung erfüllen.
Außer den Datenflusskanten gibt es noch die *Data Relation Connectors*, womit Bedingungen formuliert werden können, die den Wert von *Data Nodes* betreffen. Damit kann zum einen eine Relation zwischen zwei Datenobjekten zum Ausdruck gebracht werden oder zum anderen ein bestimmter Datenwert gefordert werden, den ein Datencontainer zu einem definierten Zeitpunkt aufweisen muss.

2.2.5. Ressourcen-Perspektive

Diese Perspektive dient der Integration des Personalwesens und der Abbildung von Zuständigkeiten. Dafür stehen vier verschiedene *Resource Nodes* zur Verfügung, siehe Abbildung 2.12. Je nach Typ werden durch die Knoten unterschiedliche organisatorische Strukturen abgebildet. Eine *Group* Node repräsentiert eine Gruppe von Mitarbeitern. Mit einem Knoten vom Typ *Staff Member* wird eine einzelner Angestellter modelliert. Zur Abbildung von Stellen und Funktionen dienen die Resource Nodes vom Typ *Role*. Zuletzt gibt es noch die Knoten vom Typ *Organizational Unit*, wodurch verschiedene Organisationseinheiten festgelegt werden können, um beispielsweise Abteilungen zu modellieren. *Resource Nodes* können, wie bereits von den Datenknoten bekannt, entweder Teil des Antecedence oder des Consequence-Patterns sein oder sie repräsentieren eine bestimmte Ressource und gehören somit dem Instance-Pattern an. Zur Unterscheidung werden Instance Knoten fett umrandet gezeichnet und Consequence Knoten setzen sich durch eine gestrichelte Umrandung von den Antecedence Knoten ab.
Performing Relations geben den Mitarbeiter an, der für die Ausführung des jeweiligen Tasks verantwortlich ist. Wenn dafür nur bestimmte Angestellte vorgesehen sind, kann mittels *Resource Conditions* und *Resource Relations* der Kreis erlaubter Mitarbeiter eingeschränkt werden. *Resource Condition* Attachments werden direkt an einen Knoten vom Typ *Resource* angeheftet. Die daraus resultierende Bedingung, muss später bei der Ausführung der Regel überprüft und sichergestellt werden. Mittels *Resource*

Relation Connectors können Beziehungen zwischen verschiedenen *Resource Nodes* ausgedrückt werden.
Ein relativ simples Beispiel für die Verwendung der Modellierungselemente dieser Perspektive ist in Abbildung 2.13 dargestellt. Die dabei modellierte Regel verlangt zunächst einmal, dass vor der Ausführung des Tasks B immer der Task A ausgeführt sein muss. Die Ressourcen-Perspektive der Regel geht dabei noch einen Schritt weiter und verlangt das A und B nur von Mitarbeitern ausgeführt werden darf, die zum einen derselben organisatorischen Einheit U angehören und zum anderen die Rolle X beziehungsweise Y inne haben.

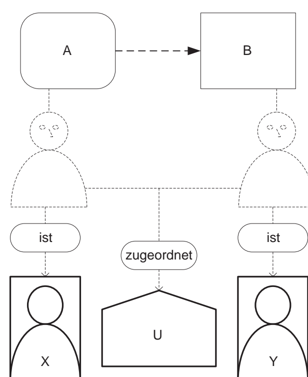


Abbildung 2.13.: Beispiel einer eCRG mit genauerer Spezifizierung der Resource Nodes durch entsprechende Relationen zwischen den Knoten aus [SKR14]

Weitere Details zur Ressourcen-Perspektive und deren Möglichkeiten sind in der Arbeit: *Modeling the Human Perspective of Business Process Compliance Rules with the Extended Compliance Rule Graph* [SKR14] nachzulesen.

2.3. Anwendungsbeispiel

Ein konkretes Beispiel für eine per eCRG definierte Compliance Regel ist in Abbildung 2.14 dargestellt. Diese Regel modelliert die internen Vorgaben einer Versicherung für die Bearbeitung eines Schadensfalls. Die Regel wird aktiviert (Antecedence Pattern), wenn von einem Versicherungskunden eine neue Schadensmeldung eingeht. Anschließend muss die eingegangene Meldung von einem Angestellten bearbeitet werden und

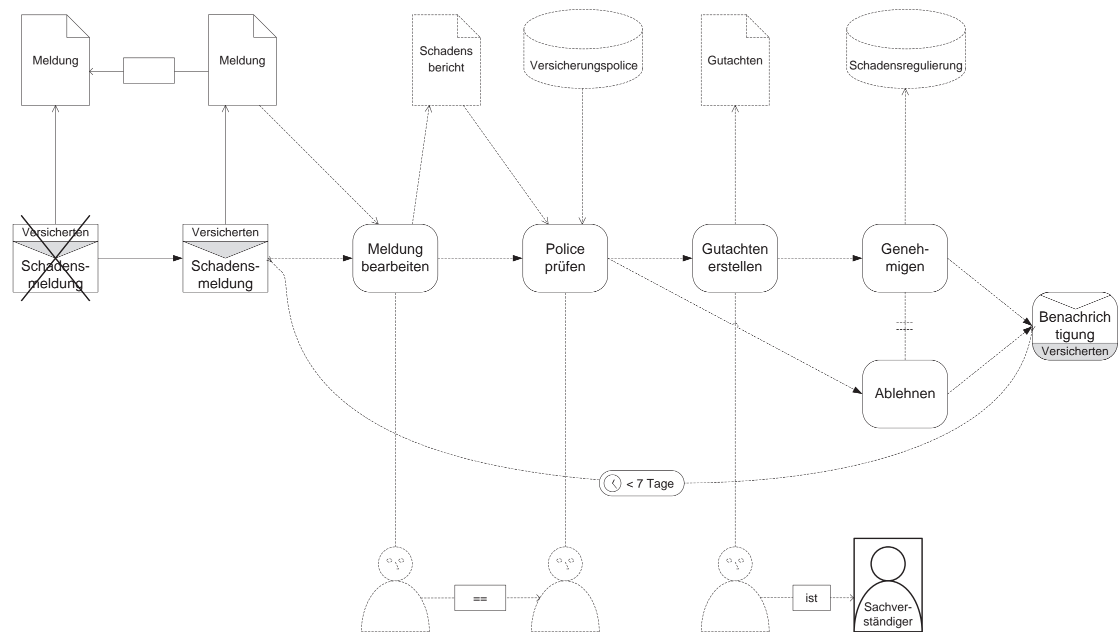


Abbildung 2.14.: eCRG Anwendungsbeispiel

ein Schadensbericht daraus erstellt werden, der für die interne Sachbearbeitung notwendig ist. Derselbe Mitarbeiter, der die Meldung bearbeitet hat, muss nun anhand der abgeschlossenen Versicherungspolice des Kunden überprüfen, ob der gemeldete Schadensfall durch die Police abgedeckt ist. Daraufhin wird die Schadensregulierung entweder abgelehnt oder genehmigt. Falls sie genehmigt wird, muss zuvor noch zwingend ein Gutachten von einem Sachverständigen erstellt werden. Zum Abschluss wird der Versicherte noch über die Entscheidung und das weitere Vorgehen benachrichtigt. Außerdem dürfen höchstens sieben Tage, vom Zeitpunkt des Eingangs der Schadensmeldung bis zum Versand der Benachrichtigung, verstrichen sein.

3 Ausführung & Bewertung von eCRG

Basierend auf den in Kapitel 2 eingeführten eCRG wird im späteren Verlauf dieser Arbeit die Implementierung eines Programms beschrieben, welches die automatisierte Überwachung der Compliance von Geschäftsprozessen ermöglicht. Dieses Kapitel beschäftigt sich im Folgenden nun mit den grundlegenden Konzepten, die für eine automatische Ausführung und Bewertung von eCRG Regeln benötigt werden.

Der Ansatz der hierbei gewählt wurde, basiert auf der Markierung des eCRG Graphen. Die Markierungen spiegeln dabei den Verlauf der ausgeführten Prozessinstanz wieder und werden für die Bewertung der eCRG Regel herangezogen. Denn der Compliance Zustand der Prozessinstanz lässt sich aus den, während der Ausführung getätigten, Markierungen ermitteln.

Im weiteren Verlauf dieses Kapitels, werden im Abschnitt 3.1 zunächst die Grundvoraussetzungen für die Ausführung und Markierung von eCRG Regeln geschaffen. Die

Ausführungsregeln für die Aktualisierung der Markierungen sind Thema im Abschnitt 3.2. Der darauffolgende Abschnitt 3.3 liefert die Regeln für die Auswertung der erzielten Markierungen. Im Abschnitt 3.4 werden die verschiedenen Konzepte, die in den vorangegangenen Unterkapiteln eingeführt wurden, anhand von Beispielen veranschaulicht.

3.1. Rahmenbedingungen für die Ausführung von eCRG

Dieser Abschnitt beschreibt die Grundlagen für die Ausführung von eCRG und schafft die Rahmenbedingungen, die dafür notwendig sind. In den nachfolgenden Unterkapiteln wird zunächst die Markierungsstruktur für die Abbildung der Compliance Zustände definiert und erläutert. Dabei wird zwischen drei verschiedenen Ebenen mit unterschiedlicher Granularität differenziert. Die unterste Ebene betrifft direkt die einzelnen eCRG Elemente. Auf der nächsten Ebene werden mehrere solche Einzelmarkierungen zu einer eCRG Gesamtmarkierung zusammengefasst. Durch die letzte Ebene werden schließlich mehrere eCRG Markierungen zu einer Gruppe zusammengefasst. Im Anschluss daran werden weitere grundlegende Eigenschaften von eCRG beschrieben, die relevant für die Ausführung sind. Dies betrifft zum einen die Hierarchie zwischen den verschiedenen eCRG Knotentypen und zum anderen die sich daraus ergebende Startvoraussetzung, welche sich auf die Vorgängerknoten bezieht. In der Tabelle 3.1 sind die verschiedenen Events samt Parameter dargestellt, die bei der Ausführung von eCRG Regeln unterstützt werden. Eine genauere Beschreibung der Events und ihrer Parameter folgt in den Unterkapiteln des Abschnitts 3.2 bei der Definition der jeweiligen Ausführungsregeln.

Unterstützte Events
START (t, tt, id)
END (t, tt, id)
SEND (t, mt, id, bp)
RECEIVE (t, mt, id, bp)
WRITE (t, id, p, v, d)
READ (t, id, p, v, d)
PERFORM (id, s)

Tabelle 3.1.

3.1.1. Markierung der Ausführungszustände

In Kapitel 2 wurde an einem Beispiel gezeigt, wie sich die Compliance einer CRG überprüfen lässt. Dabei wurden die CRG Knoten mit Ausführungsmarkierungen verse-

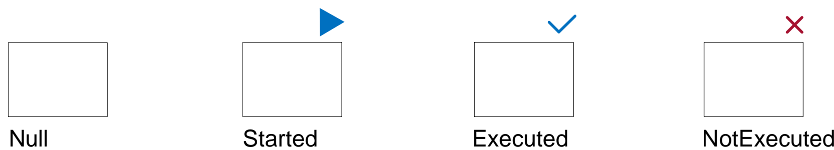


Abbildung 3.1.: Mögliche Ausführungszustände aus [Ly13]

hen, um die beobachteten Events entsprechend abzubilden. Dieses Konzept wird nun auf die Ausführung von eCRG Regeln übertragen, wobei kleinere Anpassungen und Erweiterungen vorgenommen werden.

Anhand der Markierungen (STATEMARKS) lässt sich nicht nur erkennen, welche Events im Ablaufprotokoll bereits aufgetreten sind, sondern auch welche Teile der beiden eCRG Patterns noch nicht erfüllt sind. Der Compliance Zustand eines Prozess-Ablaufprotokolls kann somit aus der aktuellen Markierung der überwachten eCRG ermittelt werden.

Die vier möglichen Ausführungsmarkierungen (*execution state marks*) für eCRG Knoten sind in Abbildung 3.1 dargestellt. Damit wird der Ausführungszustand der einzelnen Aktivitäten abgebildet. Die Bedeutung der Markierungen wird im Folgenden definiert:

NULL Zu Beginn der Auswertung besitzen alle Knoten diese Markierung. Mit NULL wird zum Ausdruck gebracht, dass für den zugehörigen Knoten bislang keine passende Aktivitätsausführung beobachtet wurde.

STARTED Diese Markierung bedeutet, dass für den jeweiligen Knoten ein entsprechendes Start Event im Ablaufprotokoll enthalten ist.

EXECUTED Damit werden vollständig ausgeführte Knoten markiert. Dies ist der Fall, wenn die dazugehörige Aktivität abgeschlossen wurde. Anstelle von EXECUTED wird gelegentlich auch die Bezeichnung COMPLETED verwendet.

NOTEXECUTED Ein mit NOTEXECUTED markierter Knoten bedeutet, dass bis dato kein passendes Event im Ablaufprotokoll enthalten war und auch in der Zukunft keine entsprechende Ausführung mehr folgen kann.

3.1.2. Markierung einer eCRG – ExMark

Als ExMark wird die Ausführungsmarkierung einer eCRG bezeichnet. Die Struktur der jeweiligen eCRG liegt diesen Markierungen direkt zugrunde. In Schaubildern werden die ExMark Markierungen meist durch ein m abgekürzt. In Abbildung 3.2 ist ein Beispiel für solch eine ExMark Markierung dargestellt. ExMarks repräsentieren die möglichen Compliance Zustände, die bei der Ausführung eines Ablaufprotokolls zustande kommen. Dabei enthalten sie für jeden eCRG Knoten eine passende Markierung, die dem Ausführungszustand des Knoten entspricht. Deswegen kann eine ExMark auch als Abbildung aufgefasst werden, die für jeden eCRG Knoten die passende Ausführungsmarkierung liefert. Jede ExMark besteht aus einer AnteExMark und ConsExMark Markierung. Erstere entspricht der Markierung des Antecedence-Patterns und die zweite der des Consequence-Patterns. Abhängig von den enthaltenen Ausführungsmarkierungen wird das jeweilige Pattern entweder verletzt oder erfüllt.

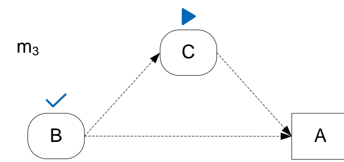


Abbildung 3.2.: ExMark

3.1.3. Menge von eCRG Markierungen – MarkStructure

Noch eine Stufe über den ExMarks befinden sich die MarkStructures (abgk. ms), eine Markierungsstruktur die mehrere ExMarks zu einer Einheit zusammenfasst. Dies ist notwendig, um Compliance Zustände, die durch ExMark Gruppen definiert sind, abbilden zu können. Ein Beispiel für eine solche MarkStructure ist in Abbildung 3.3 dargestellt. Alle ExMarks die in einer MarkStructure enthalten sind, besitzen dieselbe AnteExMark Markierung. So befindet sich in dem Beispiel der Knoten A in beiden Markierungen im Zustand Started. Somit wird eine MarkStructure immer durch genau eine AnteExMark Markierung und einer oder mehreren ConsExMark Markierungen beschrieben. Letztendlich fasst eine MarkStructure mehrere mögliche Konsequenzen für ein und dieselbe Aktivierung der Regel zusammen. Dies vereinfacht die spätere Evaluierung der Markierungen. Der Compliance Zustand einer eCRG wird durch eine Menge solcher MarkStructures repräsentiert.

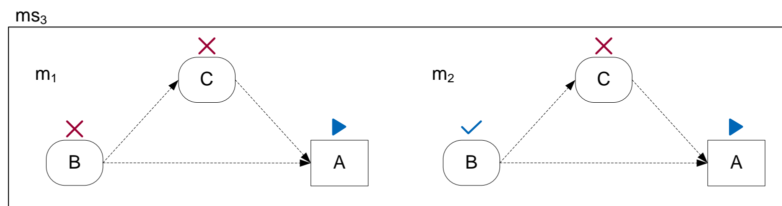


Abbildung 3.3.: Beispiel einer MARKSTRUCTURE

Initialisierung Die Auswertung des Ablaufprotokolls und damit die Ausführung beginnt immer mit der initialen MARKSTRUCTURE. Diese initiale Markierung wird auch als *Nuller* Markierung bezeichnet. Allen eCRG Knoten ist dabei zu Beginn die Ausführungsmarkierung NULL zugewiesen.

Evaluation Jede MARKSTRUCTURE besitzt die folgenden drei Eigenschaften, die bei der Auswertung entsprechend gesetzt werden:

- *Aktivierungszustand* $\longrightarrow$ ACTIVATABLE | ACTIVATED | DEACTIVATED
  Diese Eigenschaft gibt an, ob die jeweilige MARKSTRUCTURE eine Aktivierung der entsprechenden eCRG darstellt.

  Jede MARKSTRUCTURE ist zunächst einmal ACTIVATABLE. Diese bedeutet, dass noch keine Aussage darüber getroffen werden kann, ob das Antecedence-Pattern erfüllt oder verletzt wird. Der weitere Prozessverlauf entscheidet darüber, ob die Regel aktiviert wird oder nicht. Beides ist noch möglich, abhängig davon welche Events bei der Ausführung noch vorkommen beziehungsweise nicht vorkommen.

  Eine aktivierte MARKSTRUCTURE zeigt an, dass die dazugehörige ANTEEXMARK Markierung das Antecedence-Pattern der eCRG erfüllt. Damit liegt eine Aktivierung der entsprechenden Regel vor und es müssen nun die CONSEXMARK Markierungen der MARKSTRUCTURE überprüft werden.

  Der letzte mögliche Wert für diese Eigenschaft ist DEACTIVATED. Damit wird zum Ausdruck gebracht, dass das Antecedence-Pattern bei der Prozessausführung nicht festgestellt wurde und aufgrund einer Verletzung auch in Zukunft nicht mehr erfüllt werden kann. Somit wird die Compliance Regel durch diese Markierung nicht aktiviert und ist damit unerheblich für die weitere Überprüfung der Compliance.

- *Erfüllungsstatus* $\longrightarrow$ SATISFIED | VIOLATED | VIOLABLE | PENDING
  Diese Eigenschaft zeigt an, ob eine aktivierte MARKSTRUCTURE auch mit einer CONSEXMARK Markierung verknüpft ist, die das Consequence-Pattern erfüllt. Ist dies der Fall, lautet der Erfüllungsstatus SATISFIED.

  Sind hingegen alle der MARKSTRUCTURE zugewiesenen CONSEXMARK Markierungen für das Consequence-Pattern verletzt, ist der Wert dieser Eigenschaft VIOLATED. Dies bedeutet, dass aufgrund der bisherigen Prozessausführung eine zukünftige Erfüllung des Consequence-Patterns nicht mehr möglich ist.

  Trifft weder die Bedingung für SATISFIED noch die für VIOLATED zu, besitzt die MARKSTRUCTURE den Status VIOLABLE. Eine Spezialisierung dieses Zustands ist der Status PENDING. Der Unterschied zwischen diesen beiden Zuständen ist, dass für eine MARKSTRUCTURE im Zustand PENDING zwingend weitere Ereignisse im Prozessverlauf notwendig sind, um den Status SATISFIED zu erreichen. Beispielsweise wurde ein CONSOCC Knoten noch nicht vollständig ausgeführt. Im Gegensatz dazu, sind für eine MARKSTRUCTURE im Zustand VIOLABLE keine weiteren Ausführungen notwendig. Mit dem Ende der Prozessausführung wird damit aus VIOLABLE $\rightarrow$ SATISFIED. Während der Ausführung besteht jedoch immer die Möglichkeit, dass durch die entsprechende Markierung eines CONSABS Knoten das Consequence-Pattern noch verletzt wird.

- *Finalität* $\longrightarrow$ FINAL | NON-FINAL
  Bei dieser Eigenschaft wird nur zwischen FINAL und NON-FINAL unterschieden. Eine finale MARKSTRUCTURE enthält nur EXMARKS deren Markierungen sich nicht mehr ändern können. Dies ist genau dann der Fall, wenn alle eCRG Knoten entweder COMPLETED oder NOTEXECUTED sind. Andernfalls ist die MARKSTRUCTURE noch NON-FINAL.

Nach Abschluss der Ausführung des Prozess-Ablaufprotokolls müssen alle MARKSTRUCTURES FINAL und entweder ACTIVATED oder DEACTIVATED sein. Falls ACTIVATED muss außerdem SATISFIED oder VIOLATED gelten.

Das Gesamtresultat der Compliance Überprüfung lässt sich nun aus den einzelnen Markierungen ableiten. Dazu muss überprüft werden, ob alle aktivierten MARKSTRUCTURES

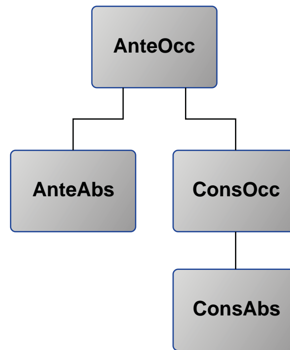


Abbildung 3.4.: Hierarchie der eCRG Knoten

auch erfüllt sind. Wenn ACTIVATED muss also auch SATISFIED gelten, damit die Compliance Regel nicht verletzt wird. Existiert auch nur für eine aktivierte MARKSTRUCTURE keine passende Erfüllung des Consequence-Patterns ist die Compliance Regel verletzt. Wenn alle MARKSTRUCTURES den Aktivierungszustand DEACTIVATED besitzen und somit keine einzige Regelaktivierung vorliegt, spricht man auch von einer *trivial erfüllten* Compliance Regel.

3.1.4. Knotenhierarchie

Die Hierarchie zwischen den vier eCRG Knotentypen ist in Abbildung 3.4 dargestellt. Sie stimmt mit der CRG Knotenhierarchie überein und wurde so für die eCRG Task und Message Nodes übernommen. Die Hierarchie folgt aus der Semantik von eCRGs und ist unter anderem notwendig um eine eindeutige Ausführungsreihenfolge zu gewährleisten. Die ANTEOCC Knoten stehen ganz oben in der Hierarchie, gefolgt von den ANTEABS und den CONSOCC Knoten. Noch eine Stufe tiefer stehen die CONSABS Knoten.

Aus dieser Hierarchie folgen auch die möglichen Relationen zwischen zwei Knoten: Die Position jedes Knoten kann nur durch Kanten zu Knoten spezifiziert werden, die in der Hierarchie höher stehen. Deswegen können ANTEABS Knoten nur ANTEOCC Knoten referenzieren. Für CONSABS Knoten sind hingegen sowohl CONSOCC als auch ANTEOCC Knoten möglich. Im Unterschied zu Absence Knoten können Occurrence Knoten sich auf Knoten desselben Typs beziehen. Sodass sich die Position von CONSOCC Knoten sowohl durch ANTEOCC, als auch durch andere CONSOCC Knoten festlegen lässt.

	ANTEOCC	**ANTEABS**	**CONSOCC**	**CONSABS**
ANTEOCC Vorgänger	COMPLETED	COMPLETED	COMPLETED	COMPLETED
ANTEABS Vorgänger	–	#	#	#
CONSOCC Vorgänger	–	#	COMPLETED	COMPLETED
CONSABS Vorgänger	–	#	–	#

'–' ≙ nicht bei der Überprüfung der Vorgänger zu berücksichtigen

'#' ≙ Kombination nicht möglich, bei korrektem Syntax

Tabelle 3.2.: Voraussetzung für einen Zustandswechsel von NULL nach STARTED in Bezug auf abgeschlossene Vorgängerknoten

3.1.5. Vorgänger Voraussetzungen

Abhängig vom Knotentyp und dem zugeordneten Pattern gibt es unterschiedliche Voraussetzungen bezüglich des Ausführungszustands von Vorgängerknoten, damit ein Knoten gestartet werden kann. So macht es beispielsweise keinen Sinn die Ausführung einer Aktivität X für einen ANTEOCC Knoten durch eine Markierung festzuhalten, wenn ein vorangehender ANTEOCC Knoten noch nicht als vollständig ausgeführt markiert wurde. Denn eine solche Markierung kann niemals zu einer Aktivierung führen, da der Vorgängerknoten aufgrund der zeitlichen Abfolge auch in Zukunft nicht mehr die Markierung EXECUTED erhalten wird.

Aufgrund der Überprüfung der Vorgängerknoten werden unnötige Markierungen vermieden und nur solche Ausführungen vermerkt, die zielführend für die Erfüllung eines Patterns sind. Um einen Knoten starten zu können, müssen deswegen alle relevanten Vorgängerknoten bereits ausgeführt sein. Die relevanten Vorgänger für die vier Knotentypen werden im Nachfolgenden beschrieben und sind in Tabelle 3.2 angegeben, wobei nur die COMPLETED Einträge bei der Überprüfung zu berücksichtigen sind.

ANTEOCC Wird für ein ANTEOCC Knoten ein passendes Start-Event erkannt, muss noch geprüft werden, ob alle Vorgänger desselben Typs die Markierung COMPLETED besitzen. Nur wenn dies auch zutrifft, kann der Knoten gestartet werden. Damit ist das Starten von ANTEOCC Knoten völlig unabhängig vom Ausführungszustand

von Vorgängerknoten mit einem anderen Knotentyp. Dies ist notwendig, um alle Vorkommen des Antecedence-Patterns festzustellen.

ANTEABS Die Position von ANTEABS Knoten kann nur durch ANTEOCC Knoten festgelegt werden. Deswegen können ANTEABS Knoten nur dann gestartet werden, wenn alle vorangehenden ANTEOCC Knoten als COMPLETED markiert wurden.

CONSOCC Die Position von CONSOCC Knoten kann sowohl durch ANTEOCC Knoten als auch durch andere CONSOCC Knoten bestimmt werden. Somit müssen sowohl die ANTEOCC als auch die CONSOCC Vorgänger überprüft werden, um festzustellen ob ein CONSOCC Knoten gestartet werden kann.

CONSABS Um einen CONSABS Knoten starten zu können, müssen alle CONSOCC und ANTEOCC Vorgängerknoten vollständig ausgeführt sein. Denn die Position von CONSABS Knoten lässt sich durch CONSOCC und ANTEOCC Knoten festlegen.

3.2. Ausführunsregeln

Mit den zuvor beschriebenen Markierungsstrukturen lässt sich der Ausführungszustand von eCRG abbilden. Wie diese Markierungen für verschiedene Events anzupassen sind, ist nun Thema dieses Abschnitts. Im Folgenden werden deswegen Ausführungsregeln definiert, die festlegen: Wie eine Markierung aktualisiert werden muss, wenn ein bestimmtes Event auftritt. Außerdem werden diese Regeln benötigt, um später die Compliance Prüfung automatisch durchführen zu können. Bei der Überprüfung wird das jeweilige Ablaufprotokoll schrittweise abgearbeitet und die Markierungen, durch Anwendung der Regeln, entsprechend angepasst. Sodass der aktuelle Prozessfortschritt zu jeder Zeit durch die Markierungen abgebildet wird.

Die grundsätzliche Vorgehensweise, die bei der Aktualisierung von Markierungen angewandt wird, lässt sich folgendermaßen beschreiben: Zunächst wird die Markierung dahingehend überprüft, ob das aktuelle Event zu einem eCRG Knoten passt und damit in die Markierung entsprechend aufzunehmen ist. Dieser Vorgang wird auch als *Matching* bezeichnet. Führt dies zu einem positiven Ergebnis, werden die dabei identifizierten Knoten entsprechend aktualisiert und markiert.

3.2.1. Task Nodes

Mittels START und END Events lässt sich der Verlauf von Prozessinstanzen beschreiben. In Kombination repräsentieren die beiden Events die vollständige Ausführung eines Tasks. Da Tasks üblicherweise nicht einem festen Zeitpunkt zuzuschreiben sind, sondern über eine gewisse Zeitspanne hinweg andauern, wird bei der Ausführung von Tasks zwischen diesen beiden Events differenziert. Für die Prozess-Perspektive bedeutet dies, dass bei der Abarbeitung von Ablaufprotokollen ebenfalls zwischen diesen beiden Events unterschieden werden muss. Die Regeln die bei der Ausführung von Task Nodes zu beachten sind, werden im Nachfolgenden besprochen. Zunächst die Regeln für die Ausführung von START Events, anschließend die Ausführungssemantik für die END Events und zuletzt noch weitere Regeln, die notwendig sind um die Markierungen abzuschließen und für die Auswertung vorzubereiten.

START Events

Ein START Event ist an einen Tasktyp, einen Identifier und an einen Zeitpunkt geknüpft. Damit wird zum Ausdruck gebracht, dass zum Zeitpunkt t ein Task vom Typ tt mit dem Identifier id gestartet wurde. In Beispielen die einen bestimmten Sachverhalt veranschaulichen sollen, wird oft auf die Angabe eines expliziten Zeitpunktes verzichtet. Stattdessen wird anhand der Eventreihenfolge der jeweilige Ausführungszeitpunkt abgeleitet.

Als Erstes muss nun geprüft werden, ob eine Task Node innerhalb der eCRG existiert, die denselben Tasktyp wie das Event aufweist. Ist dies nicht der Fall, kann mit dem nächsten Event fortgefahren werden, denn das Event passt zu keinem Knoten und ist damit nicht relevant für die eCRG. Andernfalls wurde mindestens ein passender Kandidat gefunden. Im nächsten Schritt müssen nun die Ausführungszustände dieser Knoten überprüft werden. Nur Knoten die sich noch im Zustand NULL befinden können gestartet werden[1]. Alle anderen Knoten, denen bereits ein anderer Zustand zugewiesen

[1] Dies gilt bei eCRG auch für ABSENCE Nodes. Im Gegensatz dazu, ist bei CRG auch der Zustand STARTED erlaubt, wenn eine ABSENCE Node gestartet werden soll. Dies kommt daher, das bei eCRG auch ABSENCE Nodes nichtdeterministisch gestartet werden. Bei CRG werden mehrere START Events für einen ABSENCE Knoten hingegen in einer Liste verwaltet.

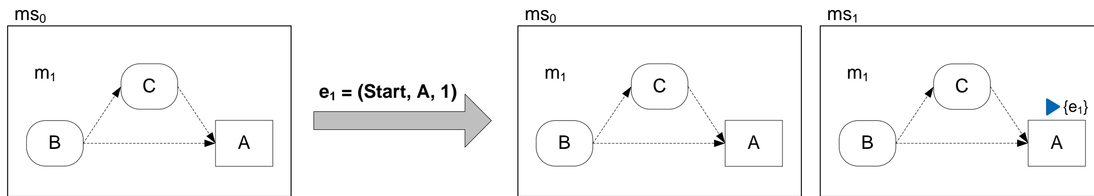


Abbildung 3.5.: Ausführung eines Start Events über eine Antecedence Task Node

wurde, werden aus der Kandidatenliste gestrichen. Bis zu diesem Punkt unterscheidet sich die Ausführung eines START Events nicht für die verschiedenen Task Node Typen. Abhängig vom jeweiligen Pattern und dem Ereignistyp unterscheidet sich jedoch der weitere Ablauf der Ausführung. Im Folgenden werden deswegen die weiteren Schritte getrennt für die vier Knotentypen beschrieben.

ANTEOCC Bevor die identifizierten Knoten gestartet werden können, müssen zuletzt noch die Markierungen der Vorgängerknoten überprüft werden, siehe dazu den Abschnitt *Vorgänger Voraussetzungen* auf Seite 38. Im Falle eines ANTEOCC Knoten muss, der Tabelle 3.2 entsprechend, der Ausführungszustand aller vorausgehenden ANTEOCC Knoten geprüft werden. Nur wenn alle relevanten Vorgänger als COMPLETED markiert wurden, kann der Knoten tatsächlich aufgrund des aktuellen START Events gestartet werden.

Die nach diesem Schritt verbliebene Kandidatenmenge kann nun als gestartet markiert werden. Dazu wird den jeweiligen Knoten der Ausführungszustand STARTED zugewiesen und außerdem das verantwortliche START Event vermerkt. Jedoch geschieht dies nicht in der Original-Markierung, sondern es wird eine Kopie von der bisherigen MARKSTRUCTURE angefertigt, siehe Abbildung 3.5. Nur in dieser Kopie wird das aktuelle START Event entsprechend vermerkt. Die ursprüngliche MARKSTRUCTURE bleibt damit unverändert, insbesondere sind die zuvor identifizierten Knoten weiterhin mit dem Ausführungszustand NULL verknüpft. Diese Vorgehensweise ist notwendig, um weitere START Events desselben Tasktyps vermerken zu können. Andernfalls würden weitere Vorkommen dieses Tasktyps unbemerkt bleiben und daraus resultierende Regelaktivierungen würden nicht registriert werden. Denn es ist durchaus möglich, dass nicht die erste, sondern erst die zweite, dritte, vierte, ... Taskausführung

die Markierung des Antecedence-Patterns vervollständigt und somit eine Regelaktivierung zur Folge hat. Würde man keine Kopien erstellen und stattdessen direkt die Original-Markierung anpassen, würde nur das erste Taskausführung registriert werden und eine mögliche spätere Regelaktivierung bliebe folglich unbemerkt. Diese Vorgehensweise wird auch als *nichtdeterministische* Ausführung bezeichnet.
Besteht die Kandidatenmenge aus mehr als einem Knoten, wird die zuvor erläuterte Aufsplittung in zwei MARKSTRUCTURES nicht nur einmal durchgeführt, sondern für jede Teilmenge einmal. Dadurch werden alle möglichen Ausführungen korrekt erfasst, wobei jede Kombination durch eine separate MARKSTRUCTURE repräsentiert wird.

CONSOCC Bei CONSOCC Knoten genügt es nicht, nur die ANTEOCC Vorgänger zu überprüfen, sondern es müssen auch die CONSOCC Vorgängerknoten betrachtet werden (→ Tabelle 3.2).

Aus demselben Grund wie schon bei ANTEOCC Knoten (Erfassung aller Vorkommen) werden auch CONSOCC Nodes nichtdeterministisch gestartet. Da CONSOCC Nodes dem Consequence-Pattern angehören, muss in diesem Fall jedoch nicht die gesamte MARKSTRUCTURE kopiert werden. Stattdessen reicht es aus, eine Kopie der EXMARK Markierung zu erstellen und dort die notwendigen Anpassungen zur Abbildung des Start Events durchzuführen. Wie schon bei den ANTEOCC Nodes geschieht dies für jede Teilmenge einmal. Alle neu erzeugten EXMARKS sind jedoch derselben MARKSTRUCTURE zuzuordnen, denn sie unterscheiden sich nur in ihrer CONSEXMARK Markierung. Wobei jede CONSEXMARK Markierung einen Versuch darstellt, das Consequence-Pattern der eCRG im Ablaufprotokoll zu identifizieren.

ANTEABS Die relevanten Vorgänger für ANTEABS Knoten entsprechen denen von ANTEOCC Knoten. Damit ist auch die Prüfung der Vorgängerknoten identisch und entspricht somit der von ANTEOCC Knoten.

Im Unterschied zu CRG, werden bei eCRG auch ABSENCE Nodes nichtdeterministisch ausgeführt. Denn bei der Auswertung von eCRG Markierungen muss nicht nur festgestellt werden können, ob ein ABSENCE Knoten ausgeführt wurde oder nicht, sondern es müssen unter Umständen noch weitere Bedingungen erfüllt sein, damit eine Regelverletzung vorliegt. Diese Zusatzbedingungen können jedoch erst zu einem

späteren Zeitpunkt auswertbar sein. Deswegen werden alle Vorkommen erfasst und die möglichen Belegungen für die ABSENCE Nodes in den Markierungen vermerkt[2]. Aufgrund derselben nichtdeterministischen Vorgehensweise bei der Ausführung von ANTEABS Knoten, unterscheidet sich die Ausführungssemantik von ANTEABS Nodes nicht von der von ANTEOCC Nodes.

CONSABS Bei CONSABS Knoten gleicht die Vorgängerüberprüfung der von CONSOCC Knoten. Es müssen ebenfalls alle Vorgänger vom Typ ANTEOCC und CONSOCC überprüft werden.

Wie bereits zuvor erwähnt, werden bei eCRG auch ABSENCE Knoten nichtdeterministisch ausgeführt. Deswegen stimmt die Ausführungssemantik von CONSABS Nodes mit der von CONSOCC Nodes überein.

Somit werden bei eCRG nicht nur die OCCURRENCE Knoten nichtdeterministisch gestartet, sondern auch die ABSENCE Knoten. Dadurch wird gewährleistet, dass alle Vorkommen durch die Markierungen erfasst werden. Dies ist wichtig, wenn nicht die erste Taskausführung, sondern eine spätere zu einer erfüllenden Pattern-Belegung führt.

END Events

Ein END Event ist ebenfalls an einen Tasktyp, einen Identifier und an einen Zeitpunkt geknüpft. Durch solch ein Event wird das Ende einer Taskausführung signalisiert. Auf jedes START Event folgt genau ein END Event, beide mit demselben Tasktyp und dem gleichen Identifier. Zwei so miteinander korrespondierende Events bilden ein Paar und beschreiben zusammen die gesamte Ausführung eines Tasks.

Die Ausführung von END Events gestaltet sich einfacher und läuft für alle Knotentypen gleich ab. Folgende Bedingungen müssen erfüllt sein, damit ein END Event über eine Task Node ausgeführt werden kann:

[2]Beispielsweise ist dies notwendig wenn, für die Regelaktivierung, die Ausführung eines Tasks X durch den Mitarbeiter Z ausgeschlossen wird. Sind in einem Ablaufprotokoll nun zwei START Events vom Typ X enthalten, einmal ausgeführt vom Mitarbeiter Z und das andere mal vom Mitarbeiter Y, würde mit der Semantik von CRG nur die erste Ausführung des ABSENCE Knotens und damit eine Verletzung des Antecedence-Patterns festgestellt werden. Die zweite Ausführung von X würde unterschlagen werden und damit eine Aktivierung der Compliance Regel verhindern. Denn beim zweiten Mal wird der Task von Y ausgeführt und somit liegt keine Verletzung des Antecedence-Patterns vor.

1. Der Tasktyp des Knotens muss mit dem des Events übereinstimmen.
2. Der Knoten muss bereits als STARTED markiert sein.
3. Des START Event das beim Starten des Knoten zusammen mit dem Ausführungszustand vermerkt wurde, muss zu dem END Event passen (gleicher Tasktyp & gleiche ID).

Alle Knoten, bei denen die zuvor aufgeführten Voraussetzungen gegeben sind, werden nun entsprechend aktualisiert. Dazu wird den identifizierten Knoten der Ausführungszustand EXECUTED ($\equiv$ COMPLETED) zugewiesen. Dies wird in der Abbildung 3.6 veranschaulicht.

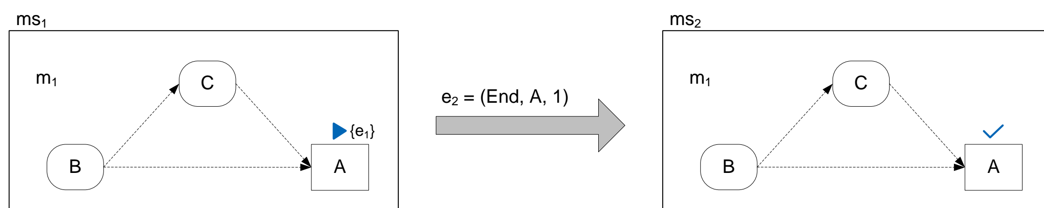


Abbildung 3.6.: Ausführung von End Events bei Task Nodes

Im Unterschied zu START Events, läuft die Ausführung bei END Events somit immer deterministisch ab. Dies bedeutet, dass die Anpassungen an der ursprünglichen Markierung vorgenommen werden und keine neuen Markierungen abgeleitet werden.

Weitere Regeln

Bei der Ausführung von START und END Events sind neben den reinen Markierungsregeln noch die folgenden Regeln anzuwenden, um stets semantisch korrekte Compliance Zustände abzubilden.

Noch nicht abgeschlossene Vorgänger Aufgrund der Hierarchie zwischen den verschiedenen Knotentypen ($\rightarrow$ Abbildung 3.4) können ANTEOCC Nodes unabhängig vom Ausführungszustand ihrer nicht-ANTEOCC Vorgängerknoten gestartet werden. Dasselbe gilt für CONSOCC Nodes in Bezug auf vorausgehende CONSABS Knoten. Somit ist es möglich, dass sich Vorgängerknoten noch immer im Ausführungszustand

NULL oder STARTED befinden und gleichzeitig ein nachfolgender Knoten gestartet wird. Die Markierung der Vorgängernoten muss deswegen an die aktuelle Situation angepasst werden. Denn aufgrund der zeitlichen Abarbeitungsreihenfolge von Events können diese Knoten nicht mehr vollständig ausgeführt werden und müssen deshalb auf NOTEXECUTED gesetzt werden. Andernfalls wäre die Markierung inkonsistent und würde einen widersprüchlichen Compliance Zustand repräsentieren.

Ausführungskonflikte zwischen Knoten Bei der Abarbeitung von START Events kann es zu Ausführungskonflikten zwischen den Knoten kommen. Zum Beispiel tritt solch ein Konflikt auf, wenn mehrere Knoten aufgrund desselben START Events und zur gleichen Zeit ausgeführt werden können. Stehen diese Knoten nun auch noch in Relation zueinander, beispielsweise durch eine Sequence Kante oder einen Exclusive Connector, so kann das gleichzeitige Starten der Knoten zu einer fehlerhaften Markierung führen. Um dies zu verhindern, werden die Knoten entsprechend ihrer Position in der Hierarchie gestartet. Folglich werden zunächst alle ANTEOCC Knoten verarbeitet und gestartet. Anschließend folgen die ANTEABS Knoten, gefolgt von den CONSOCC und CONSABS Knoten. Da beim Starten der Knoten möglicherweise die vorherige Regel zum Einsatz kommt und nicht vollständig abgeschlossene Vorgängerknoten als NOTEXECUTED gekennzeichnet werden, kann sich die Anzahl der ausführbaren Knoten für dieses Event verringern. Denn wenn sich darunter Knoten befinden, die zu Beginn noch als ausführbar identifiziert wurden, sind sie dies nun nicht mehr. Diese Vorgehensweise sorgt dafür, dass zueinander in Relation stehenden Knoten nicht mit ein und demselben Event verknüpft sind.

Absence Markierungen bereinigen Aufgrund der nichtdeterministischen Ausführung von ABSENCE Knoten ist es notwendig, die Markierungen vor der Auswertung zu bereinigen. Dabei werden die vorhandenen MARKSTRUCTURES der Reihe nach überprüft und wenn nötig gesäubert. Denn infolge der nichtdeterministischen Ausführungssemantik bleibt die ursprüngliche nicht gestartete Markierung stets erhalten, um weitere Vorkommen erfassen zu können. Wenn nun aber mindestens eine vollständige Ausführung des ABSENCE Knotens beobachtet wurde, spiegelt die Markierung damit am Ende nicht den tatsächlichen Compliance Zustand wieder. Deswegen werden aus einer MARKSTRUCTURE all diejenigen Markierungen herausgelöscht, welche für den in

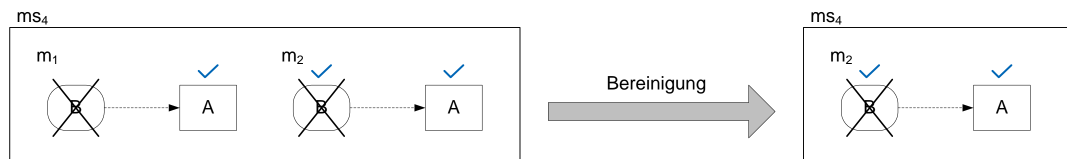


Abbildung 3.7.: Bereinigung nicht ausgeführter Absence Markierungen

Wirklichkeit ausgeführten ABSENCE Knoten noch auf den Ausführungszustand NULL abbilden. In der Abbildung 3.7 wird diese Situation an einem Beispiel veranschaulicht.

Markierungen abschließen Um die Markierungen hinsichtlich ihres Compliance Zustandes auswerten zu können, müssen die MARKSTRUCTURES abgeschlossen werden. Dadurch wird kenntlich gemacht, dass keine weiteren Events mehr folgen und man Ende der Prozessausführung angelangt ist. Nachdem das gesamte Prozess-Ablaufprotokoll abgearbeitet ist, wird den noch nicht vollständig ausgeführten Knoten der Ausführungszustand NOTEXECUTED zugewiesen. Dies betrifft somit alle Knoten, die mit NULL oder STARTED markiert sind. Da keine weiteren Events mehr folgen, können diese Knoten auch nicht mehr den Zustand EXECUTED erreichen. Nach Anwendung dieser Regel, ist damit allen Knoten entweder der Zustand EXECUTED oder NOTEXECUTED zugewiesen. Da die Markierungen nun endgültig sind und sich an ihnen nichts mehr ändert, sind die MARKSTRUCTURES nun auch FINAL.

In Abschnitt 3.4 wird die Ausführung von START und END Events für eCRG Knoten an einem Beispiel veranschaulicht. Die Abbildung 3.9 auf Seite 53 zeigt dabei eine Möglichkeit für die Darstellung der Markierungen und der damit verbundenen Compliance Zustände.

3.2.2. Message Nodes

Neben der Prozess-Perspektive wird von eCRG auch die Modellierung der Interaktions-Perspektive unterstützt. Dazu stehen Message Nodes zur Verfügung, die zum einen das Senden von Nachrichten repräsentieren und zum anderen den Empfang definieren. In Ablaufprotokollen wird der Versand beziehungsweise Empfang einer Nachricht durch SEND und RECEIVE Events protokolliert. Im Gegensatz zu Task Nodes wird bei Message Nodes nicht zwischen Beginn und Ende unterschieden, sondern ein fester Ausführungszeitpunkt angegeben. Die Regeln, die bei der Ausführung von Message Knoten zu beachten sind, werden im nachfolgenden Abschnitt behandelt.

SEND & RECEVIE Events

Da sich SEND und RECEIVE Events nur vom Ereignistyp her unterscheiden, ansonsten aber identisch sind, unterscheidet sich deren Ausführungssemantik auch nicht voneinander. Deswegen werden die beiden Events gemeinsam besprochen.

Ähnlich wie START Events, sind SEND und RECEIVE Events ebenfalls an einen Typ mt, einen Identifier id und an einen Zeitpunkt t geknüpft. Nur das anhand des Typs nicht der Tasktyp, sondern der Messagetyp beschrieben wird. Dieser entspricht jedoch nicht dem Ereignistyp (SEND / RECEIVE). Außerdem wird noch der Geschäftspartner bp, von dem die Nachricht kommt beziehungsweise an den sie geht, mit angegeben.

Nun zur Ausführung von SEND & RECEIVE Events und den dabei durchzuführenden Anpassungen. Stößt man bei der Abarbeitung eines Prozess-Ablaufprotokolls auf eines dieser beiden Events, müssen die Markierungen ebenfalls überprüft und wenn notwendig aktualisiert werden. Da die Ausführungssemantik von Message Nodes der von Task Nodes gleicht, lassen sich die bereits bekannten Regeln hierauf übertragen. Nur dass, wie bereits zuvor erwähnt, bei Message Knoten keine weitere Differenzierung zwischen Anfang und Ende vorgenommen wird, sondern mit Eintritt eines passenden Events, der Knoten sofort als vollständig ausgeführt markiert wird. Die anzuwendenden Regeln für SEND und RECEIVE Events stimmen somit fast komplett mit den Ausführungsregeln von START Events überein. Folgende Anpassungen sind vorzunehmen um die Semantik von

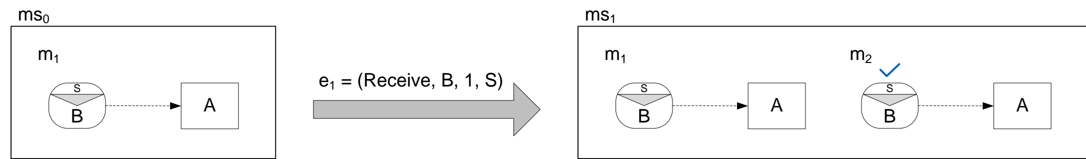


Abbildung 3.8.: Ausführung einer Consequence Message Node

Message Knoten korrekt abzubilden. Zum einen müssen natürlich die Message Nodes anstelle der Task Nodes untersucht werden und außerdem wird dabei überprüft, ob der Messagetyp des Knotens mit dem des Events übereinstimmt. Zum anderen werden die zu einem Event passenden Knoten nicht als gestartet markiert, sondern ihnen wird direkt der Ausführungszustand EXECUTED zugewiesen. Wie in Abbildung 3.8 zu sehen, bleibt die nichtdeterministische Ausführungssemantik dabei jedoch erhalten.

3.2.3. Data Nodes

Durch die Daten-Perspektive wird der lesende und schreibende Datenzugriff auf Datencontainer und Datenobjekte spezifiziert. In Ablaufprotokollen wird der Datenfluss in Form von READ und WRITE Events beschrieben. Dem Namen entsprechend, wird mit diesen beiden Events der Lese- und Schreibzugriff von Tasks und Messages protokolliert und damit im Ablaufprotokoll als Ereignis festgehalten.

WRITE & READ Events

Beide Events können aufgrund ihrer Attribute als Datenflusskante angesehen und entsprechend aufgefasst werden: Zum Zeitpunkt t schreibt (liest) der Task oder die Message mit dem Identifier id den Parameter p mit Wert v in den (aus dem) angegebenen Datencontainer d. Damit unterscheiden sich die beiden Events nur in ihrem Ereignistyp und sind ansonsten identisch. Da auch die Ausführungsregeln übereinstimmen, werden die zu beachtenden Regeln für beide Events gemeinsam besprochen. Genau genommen werden im Folgenden die Regeln für die Ausführung von WRITE Events besprochen. Die dadurch definierte Ausführungssemantik lässt sich jedoch analog bei der Ausführung von READ Events anwenden.

In Unterschied zu den zuvor besprochenen Events, wird bei WRITE Events nicht direkt der Knoten markiert. Anstelle des jeweiligen Knoten, wird die dem Event entsprechende Datenflusskante mit dem geschriebenen Wert markiert. Die Vorgehensweise für die Aktualisierung einer Markierung ist dabei folgende:

1. Zunächst wird überprüft, ob das Event relevant für die Markierung ist. Dies ist der Fall, wenn der mit dem Event angegebene Identifier mit einer der Task Node beziehungsweise Message Node Markierungen übereinstimmt. Anders ausgedrückt, es wird überprüft, ob die aktuelle Markierung den für das WRITE Event verantwortlichen Task beziehungsweise die verantwortlich Message enthält.
2. Führt die Überprüfung zu einem positiven Ergebnis und es wurde eine passende Markierung identifiziert, werden die ausgehenden Datenflusskanten des Knoten ermittelt, der zu der Markierung gehört.
3. Von den ermittelten Kanten werden nun diejenigen markiert, deren Parameter mit dem des Events übereinstimmen.

Die Markierung der Datenknoten findet erst im Anschluss an die Ausführung des Ablaufprotokolls statt. Dabei wird versucht eine passende Belegung für den Knoten zu finden, welche den Markierungen der verbundenen Datenflusskanten entspricht.

3.2.4. Resource Nodes

Für die Modellierung der Ressourcen-Perspektive werden Resource Nodes eingesetzt. Bedingungen zwischen den Knoten werden durch Resource-Relation Kanten ausgedrückt. In Prozess-Ablaufprotokollen wird diese Perspektive von den PERFORM Events abgedeckt. Die Ausführungssemantik dieser Events wird nun im Folgenden erörtert.

PERFORM Events

Durch PERFORM Events wird im Ablaufprotokoll festgehalten, welcher Mitarbeiter s (*Staff Member*) für die Ausführung eines bestimmten Tasks id verantwortlich ist. Wie schon bereits bei den Datenknoten werden auch die Resource Nodes nicht direkt bei

der Ausführung markiert. Stattdessen wird die zum Event passende Kante markiert und dabei der verantwortliche *Performer* des Tasks vermerkt. Die Vorgehensweise ähnelt der von WRITE und READ Events. Zunächst wird überprüft, ob die angegebene Task-ID überhaupt in der Markierung vorhanden ist. Nur wenn dies der Fall ist, handelt es sich um ein relevantes Event und die Markierung muss entsprechend aktualisiert werden. Anhand der ID wird zunächst der dazugehörige Knoten und dann der zum Event passende Performing Connector identifiziert. Diese Kante wird nun mit dem im Event angegebenen Mitarbeiter markiert. Erst nach der Ausführung der im Ablaufprotokoll enthaltenen Events, werden schließlich die Resource Relations überprüft und wenn möglich eine Markierung der Resource Nodes vorgenommen.

3.3. Bewertungsregeln

Nachdem alle Events ausgeführt worden sind und die Markierungen anschließend abgeschlossen wurden, folgt nun die Auswertung der Markierungen.

Zuerst findet eine Überprüfung aller eCRG Elemente und derer Markierungen statt. Werden dabei Verletzungen der eCRG Regel festgestellt, wird dies durch eine entsprechende Markierung für die nachfolgende Auswertung vermerkt. Unter anderem werden die Ausführungszustände der einzelnen eCRG Knoten untersucht. Wird bei dieser Untersuchung festgestellt, dass ein Absence Knoten ausgeführt beziehungsweise ein Occurrence Knoten nicht ausgeführt wurde, liegt eine Verletzung des jeweiligen Patterns vor und die Markierung wird entsprechend gekennzeichnet. Für alle noch nicht markierten Knoten (Data und Resource Nodes) wird außerdem versucht, eine passende Belegung zu finden. Existiert keine solche erfüllende Belegung für einen Knoten, wird dieser ebenfalls als verletzt gekennzeichnet. Zusätzlich zu den Knoten, werden an dieser Stelle auch die Bedingungen und Relationen überprüft, die durch die verschiedenen Kanten definiert wurden. Außer den Sequence Kanten werden dabei zum Beispiel auch die Exclusive, Alternative und Time-Distance Connectoren kontrolliert. Des Weiteren wird überprüft, ob die Resource Relations eingehalten werden und ob der geforderte Datenfluss stattgefunden hat.

Erst im Anschluss an diese abschließende Prüfung aller eCRG Forderungen, werden die Markierungen hinsichtlich ihres Compliance Zustandes evaluiert. Bei der Evaluierung werden zunächst die Compliance Zustände der vorhandenen MARKSTRUCTURES ermittelt und davon wird schließlich die Compliance des gesamten Ablaufprotokolls abgeleitet. Folgende Schritte sind für die Durchführung dessen notwendig:

1. Zunächst müssen die MARKSTRUCTURES der Reihe nach ausgewertet werden. Da durch diese mehrere EXMARK Markierungen zu einer Einheit zusammengefasst werden, müssen zur Bestimmung des Compliance Zustandes die enthaltenen eCRG Markierungen näher untersucht werden.

2. Im nächsten Schritt werden nun die vorhandenen EXMARK Markierungen ausgewertet. Es wird zunächst überprüft, ob die ANTEEXMARK Markierung das Antecedence-Pattern der eCRG erfüllt und damit eine Aktivierung der Regel darstellt. Daraus leitet sich schließlich auch der Aktivierungszustand der MARKSTRUCTURE ab: ACTIVATED wenn das Antecedence-Pattern erfüllt wird und DEACTIVATED falls eine Verletzung vorliegt.

3. Im Anschluss daran müssen im Falle einer Aktivierung die CONSEXMARK Markierungen ausgewertet werden. Stimmt eine solche Markierung mit dem Consequence Pattern überein, erfüllt sie die Regel und ist SATISFIED. Anderenfalls ist das Pattern verletzt und die Markierung somit VIOLATED.

4. Abhängig vom Resultat der vorangegangenen Auswertung der CONSEXMARK Markierungen, bestimmt sich nun der Erfüllungsstatus der jeweiligen MARKSTRUCTURE. Existiert mindestens eine erfüllende Markierung des Consequence-Patterns, so lautet der Erfüllungsstatus der MARKSTRUCTURE ebenfalls SATISFIED. Nur wenn alle CONSEXMARK Markierungen verletzt sind, lautet dieser VIOLATED.

4b. Im Falle von Ablaufprotokollen, die eine noch nicht abgeschlossene Prozessinstanz beschreiben, lässt sich des Weiteren noch zwischen VIOLABLE und PENDING unterscheiden. Voraussetzung für diese zwei Zustände ist, dass die jeweilige MARKSTRUCTURE keine Markierung des Consequence-Patterns enthält, die definitiv erfüllt ist und auch in Zukunft keine Verletzung mehr zulässt. Denn sonst wäre SATISFIED der korrekte Zustand. Eine MARKSTRUCTURE besitzt somit genau dann

den Erfüllungsstatus VIOLABLE, wenn sie eine CONSEXMARK Markierung enthält, die bis dato noch nicht endgültig verletzt ist und nur aufgrund weiterer Events eine Verletzung möglich ist. Den Zustand PENDING besitzen MARKSTRUCTURES, die eine noch nicht vollständige Markierung des Consequence-Patterns enthalten und dies gleichzeitig der einzige Grund ist, der zu einer Verletzung führen würde. Zum Beispiel könnte noch die Ausführung eines CONSOCC Knotens fehlen.

5. Die Compliance des Ablaufprotokolls lässt sich nun aus dem Aktivierungszustand und dem Erfüllungsstatus der ausgewerteten MARKSTRUCTURES ermitteln. Die dabei zu beachtende Regel besagt, dass jede aktivierte MARKSTRUCTURE auch SATISFIED sein muss. Ansonsten existiert eine Aktivierung der Regel ohne eine passende Übereinstimmung der Regelkonsequenz.

3.4. Veranschaulichung

In diesem Abschnitt wird an verschiedenen Beispielen veranschaulicht, wie die in den beiden vorherigen Abschnitten beschriebenen Regeln und Konzepte anzuwenden sind. Gleichzeitig wir dadurch nochmals ein Überblick über die Vorgehensweise gegeben, wie ein Prozess-Ablaufprotokoll über eine eCRG ausgeführt wird und wie sich daraus der Compliance Zustand der beschriebenen Prozessinstanz ermitteln lässt. In den Beispielen wird, wenn nicht notwendig, auf die Angabe der Zeitpunkte verzichtet.

Ausführungsregeln für Task Nodes

Zunächst ein Beispiel für die Ausführung von Task Knoten. In Abbildung 3.9 ist die schrittweise Aktualisierung der Markierungen dargestellt, um die einzelnen Events entsprechend abzubilden. Die dabei zu überprüfende Compliance Regel verlangt die Ausführung von C zwischen jeder Ausführung von A und dem nächsten direkt nachfolgendem B. Die Events e_1 - e_6 entsprechen dem Prozess-Ablaufprotokoll und beschreiben die Ausführung des Tasks A_1, gefolgt von dem Task B_2 und zuletzt noch der Task C_3.

e_1= (Start,A,1)
$ms_0 > ms_1$

e_2= (End,A,1)
$ms_1 > ms_2$

e_3= (Start,B,2)
$ms_2 > ms_3$
$ms_2 > ms_4$

e_4= (End,B,2)
$ms_3 > ms_5$
$ms_4 > ms_6$

e_5= (Start,C,3)
$ms_2 > ms_7$
$ms_6 > ms_8$

e_6= (End,C,3)
$ms_7 > ms_9$
$ms_8 > ms_{10}$

Abbildung 3.9.: Anwendung der Ausführungsregeln für Task Nodes

Die Ausführung der Events beginnt mit der initialen Markierung, dargestellt durch ms_0. Zu Beginn repräsentiert diese Markierung den Ursprungszustand, weswegen sich alle Knoten noch im Ausführungszustand NULL befinden. Im Folgenden wird nun der Aktualisierungsvorgang für die einzelnen Events beschrieben:

$e_1 = (Start, A, 1)$ Zunächst wird überprüft, ob für dieses Event eine passende Task Node existiert. Da der Knoten A vom Typ her übereinstimmt, ist das Event relevant und die vorhandenen Markierungen müssen entsprechend angepasst werden. Zu Beginn ist ms_0 allerdings die einzig vorhandene Markierung. Dem Knoten A ist dabei der Ausführungszustand NULL zugewiesen und kann somit gestartet werden. Die Überprüfung der Vorgängerknoten erübrigt sich, da A keine besitzt. Damit sind alle Voraussetzungen gegeben, um den Knoten A als gestartet zu markieren. Da es sich dabei um einen ANTEOCC Knoten handelt, besagt die Ausführungssemantik, dass ms_0 unverändert erhalten bleibt und eine Kopie davon erstellt wird. In der neu hinzugekommenen MARKSTRUCTURE ms_1 wird A als gestartet markiert.

$e_2 = (End, A, 1)$ Diese Event signalisiert das Ende der im Schritt zuvor gestarteten Taskausführung. Die drei zu überprüfenden Bedingungen werden in ms_1 erfüllt: der Tasktyp des Knotens A stimmt mit dem des Events überein, der Knoten A wurde außerdem als gestartet markiert und zuletzt stimmt die ID des vermerkten Start Events mit der des aktuellen Events überein. Der Ausführungszustand von A wird deswegen von STARTED zu EXECUTED aktualisiert.

$e_3 = (Start, B, 2)$ Bei der Ausführung von e_3 muss nur ms_2 aktualisiert werden. Denn in ms_0 sind die beiden Knoten B nicht ausführbar, da A noch nicht ausgeführt wurde. In ms_2 ist dies hingegen der Fall, sodass hier sowohl der ANTEABS Knoten B, als auch der ANTEOCC Knoten B ausführbar sind. Da in solch einem Fall, die Knoten entsprechend ihrer Hierarchie verarbeitet werden[3], wird zunächst der ANTEOCC Knoten gestartet. Weil der Knoten B vom Typ ANTEOCC ist, wird wie schon bereits bei e_1 eine neue Markierung von ms_2 abgeleitet. In der neuen Markierung ms_3 wird der ANTEOCC Knoten B nun als gestartet markiert und den beiden bis dato nicht vollständig ausgeführten Vorgängerknoten der Ausführungszustand NOTEXECUTED

[3] siehe Abschnitt *Ausführungskonflikte zwischen Knoten* auf Seite 45

zugewiesen. Als nächstes wird nun der weiterhin ausführbare ANTEABS Knoten B in ms_2 gestartet. Dazu wird ebenfalls eine Kopie der MARKSTRUCTURE erstellt und in der daraus resultierenden Markierung ms_4 wird dem ANTEABS Knoten der Ausführungszustand STARTED zugewiesen.

$e_4 = (End, B, 2)$ Analog zu e_2 beschreibt e_4 das Ende der Taskausführung von B. In allen Markierungen, in denen B aufgrund eines passenden Events gestartet wurde, wird der Ausführungszustand der Knoten zu EXECUTED aktualisiert. Dies gilt für die Markierungen ms_3 und ms_4, die deswegen entsprechend angepasst werden.

$e_5 = (Start, C, 3)$ Durch dieses Event wird der Start einer Ausführung von C beschrieben. Nur in ms_2 und ms_6 sind die Voraussetzungen für einen Start von C erfüllt. Denn lediglich dort, ist der Knoten C noch im Zustand NULL und der Knoten A wurde bereits als vollständig ausgeführt markiert. Da es sich bei C nun um einen CONSOCC Knoten handelt, unterscheidet sich die weitere Vorgehensweise nun etwas von den vorherigen Startvorgängen. Statt eine komplette Kopie der MARKSTRUCTURE anzufertigen, wird bei Consequence Knoten die EXMARK Markierung kopiert. Die neue Markierung gehört derselben MARKSTRUCTURE an, wie die ursprüngliche, da sich im Folgenden nichts an der Markierung des Antecedence-Patterns ändert. In der jeweils neu hinzugekommenen Markierung m_2 wird C nun als STARTED markiert.

$e_6 = (End, C, 3)$ Zuletzt wurde noch das Ende von C_3 beobachtet. Deswegen wird der Ausführungszustand des Knotens C in den Markierungen zu EXECUTED aktualisiert, bei denen ein passendes Start Event vermerkt wurde.

Nachdem die Ausführung aller Events komplett ist, werden die Markierungen abgeschlossen. Dabei werden alle Knoten auf NOTEXECUTED gesetzt, die bis dahin noch nicht vollständig ausgeführt wurden. Die Markierungen sind damit FINAL und bereit für die Evaluierung des Compliance Zustandes. Nach der Auswertung ist die Markierung ms_5 die einzige, die aktiviert ist. Die Markierungen ms_0, ms_9 und ms_{10} sind hingegen alle deaktiviert. Bei ms_0 wurde weder A noch B als ausgeführt markiert, bei ms_9 fehlt die Ausführung von B und bei ms_{10} wurde ein Absence Knoten ausgeführt. Da ms_5 aktiviert ist aber das Consequence-Pattern der eCRG nicht erfüllt (C wurde nicht ausgeführt), lautet das Ergebnis der Compliance Prüfung VIOLATED.

Ausführungsregeln für Message und Data Nodes

Nachdem im vorherigen Beispiel die Ausführung von Task Nodes veranschaulicht wurde, zeigt das Beispiel in Abbildung 3.10 die Ausführung von Daten und Message Knoten. Dargestellt ist dort ebenfalls die schrittweise Aktualisierung der Markierungen, wobei die Unterschiede zwischen der Abbildung des Datenflusses und der Vorgehensweise bei Task und Message Knoten deutlich wird. Die zu überprüfende Compliance Regel fordert, dass auf jede eingehende Nachricht vom Typ M, die vom Geschäftspartner P kommt und D beschreibt, die Ausführung eines Tasks T folgt und dabei der Wert aus D gelesen wird. Die Abarbeitung des Prozess-Ablaufprotokolls, repräsentiert durch die Events e_1 - e_5, gestaltet sich wie folgt:

$\underline{e_1 = (Receive, M, 1, P)}$ Mit diesem Event wurde der Empfang einer Nachricht vom Typ M und der ID 1 protokolliert. Versendet wurde die Nachricht vom Geschäftspartner P. Damit passt das Event zu dem einzigen Message Knoten. Des Weiteren ist in der Markierung ms_0 dem Knoten der Ausführungszustand NULL zugewiesen und weil auch keine Vorgängerknoten existieren, kann der Knoten als ausgeführt markiert werden. Da dieser vom Typ ANTEOCC ist und Message Nodes analog zu Task Nodes ausgeführt werden, wird zunächst eine neue MARKSTRUCTURE von ms_0 abgeleitet und in der neu hinzugekommenen Kopie der Message Knoten als EXECUTED markiert.

$\underline{e_2 = (Write, 1, 55)}$ Dieses Event besagt, dass ein Task beziehungsweise eine Message mit der ID 1 den Wert 55 geschrieben hat. Anhand der ID wird nun in den vorhandenen Markierungen nach einer Übereinstimmung gesucht. Für ms_1 verläuft diese Suche positiv: Der Message Knoten wurde aufgrund eines Events als ausgeführt markiert, welches dieselbe ID wie das aktuelle Event aufweist. Folglich wird nun die ausgehende Datenflusskante des Knotens mit dem Wert 55 belegt.

$\underline{e_3 = (Start, T, 3)}$ Ein Task vom Typ T mit der ID 3 wurde gestartet. Da in ms_2 alle Voraussetzungen (Tasktyp, Ausführungszustand und Vorgängerknoten) erfüllt sind, wird den Ausführungsregeln für Consequence Knoten entsprechend eine Kopie von m_1 erstellt und dort der Knoten als gestartet markiert.

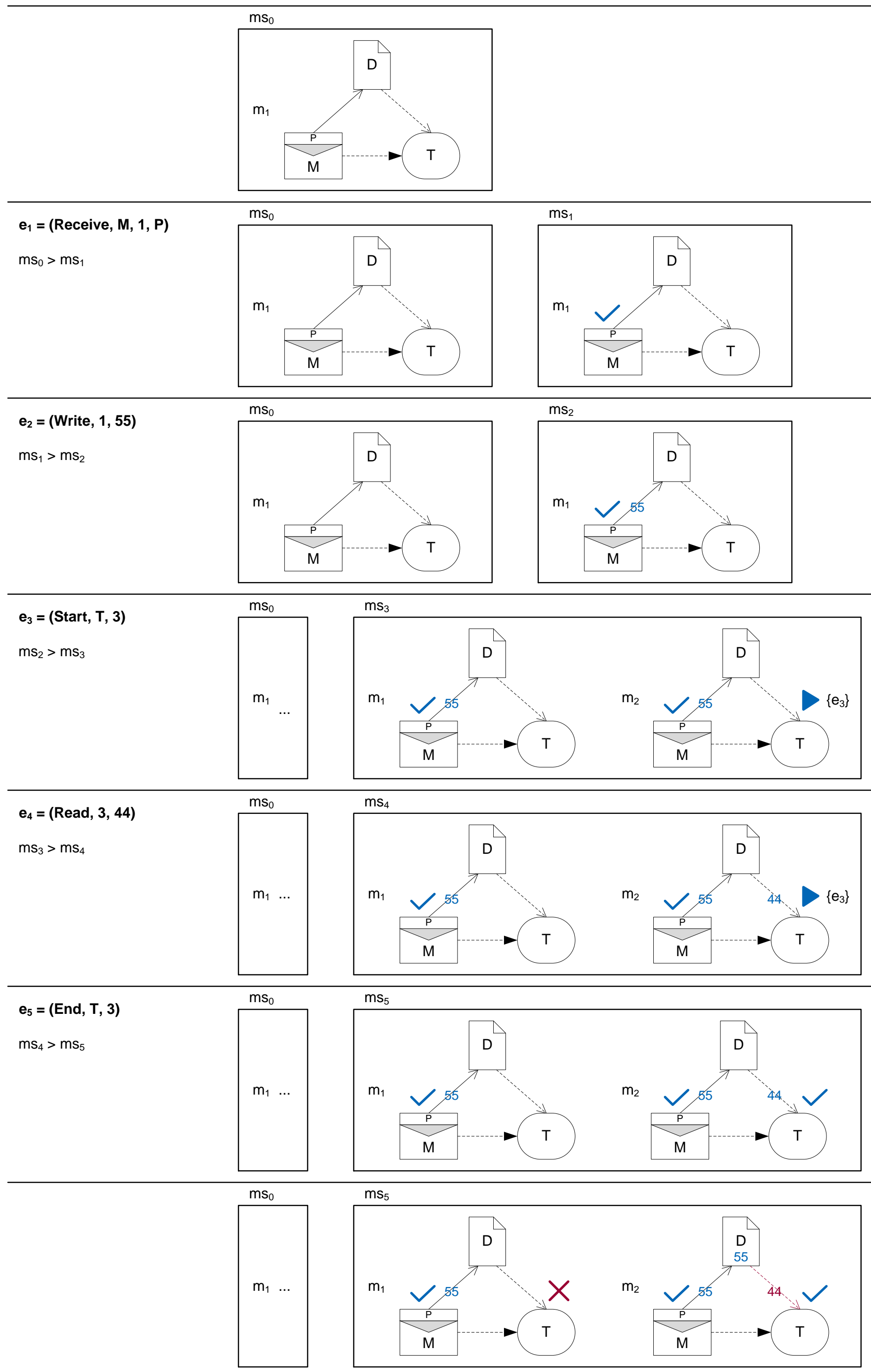


Abbildung 3.10.: Anwendung der Ausführungsregeln für Message & Data Nodes

$\underline{e_4 = (Read, 3, 44)}$ Analog zu e_2 wird hierdurch beschrieben, dass ein Task oder eine Message mit der ID 3 den Wert 44 gelesen hat. Die ID passt zu dem im Schritt zuvor gestarteten Task und somit wird die eingehende Datenflusskante in der Markierung m_2 mit dem gelesenen Wert markiert.

$\underline{e_5 = (End, T, 3)}$ Zuletzt wurde noch das Ende eines Tasks vom Typ T mit der ID 3 protokolliert. Bei der Suche nach einer passenden Startmarkierung wird man ebenfalls in der Markierung $ms_4 \to m_2$ fündig. Entsprechend wird die Markierung zu EXECUTED aktualisiert.

Vor der Auswertung müssen die Markierungen abgeschlossen werden, wobei alle nicht vollständig ausgeführten Knoten als NOTEXECUTED markiert werden. Deswegen wird die Task Node in der Markierung $ms_5 \to m_1$ entsprechend markiert. Dasselbe gilt auch bei der Markierung $ms_0 \to m_1$ für die Task und Message Node. Damit ist ms_0 deaktiviert und ms_5 aktiviert. Folglich müssen nun noch die Markierungen m_1 und m_2 von ms_5 bezüglich des Consequence-Patterns betrachtet werden. Dabei stellt sich heraus, das beide das Consequence-Pattern verletzen und damit ms_5 VIOLATED ist. Denn bei m_1 fehlt die Ausführung von T und in m_2 stimmt der gelesene Wert nicht mit dem überein, der D zugewiesenen wurde. Insgesamt betrachtet ist somit auch die Compliance für die gesamte Prozessinstanz verletzt.

Ausführungsregeln für Resource Nodes

Das in Abbildung 3.11 dargestellte Beispiel dient der Veranschaulichung von PERFORM Events. Gleichzeitig wird dabei die Auswertung einer Ressourcen- und einer Zeit-Bedingung gezeigt. Die zu überprüfende Compliance Regel fordert, dass im Falle einer Ausführung von B, zuvor schon A ausgeführt wurde. Dabei müssen zwischen dem Ende von A und dem Start von B mindestens zwei Stunden vergangen sein. Außerdem darf A nicht von dem Mitarbeiter ausgeführt worden sein, der schon für die Ausführung von B verantwortlich ist. Im Nachfolgenden wird die Ausführung der Events e_1 - e_9 der Reihe nach erläutert. Das durch die Events repräsentierte Ablaufprotokoll lässt sich in drei Gruppen untergliedern, die sich prinzipiell vom Ablauf her ähneln. Deswegen wird auch nur die erste Gruppe im Detail besprochen.

ms0
e1 = (Start, A, 1, 12:00)
ms0 > ms1
ms1
{e1}
e2 = (Perform, 1, John)
ms1 > ms2
ms2
e3 = (End, A, 1, 12:30)
ms2 > ms3
ms3
e4 = (Start, A, 2, 12:45)
e5 = (Perform, 2, Sam)
e6 = (End, A, 2, 13:30)
ms3 > ms4 > ms5 > ms6
ms6
e7 = (Start, B, 3, 14:45)
e8 = (Perform, 3, John)
e9 = (End, B, 3, 15:00)
ms6 > ms7 > ms8 > ms9
ms6
ms9
m1
m2
m3
A
B
>2h
≠
John
Sam

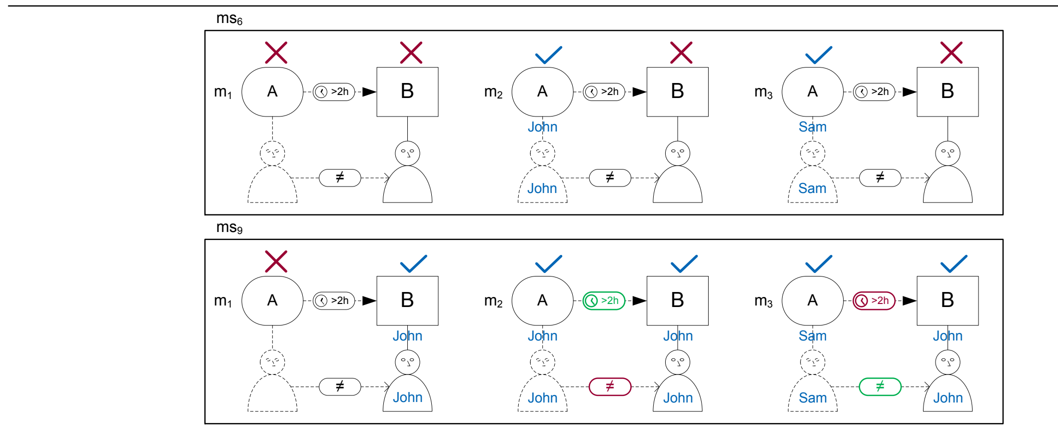


Abbildung 3.11.: Anwendung der Ausführungsregeln für Resource Nodes

$e_1 = (Start, A, 1, 12{:}00)$ Dieses Event passt zu dem Knoten A. Da es sich dabei um einen CONSOCC Knoten handelt, wird eine Kopie der Ausgangsmarkierung erstellt und in dieser neu hinzugekommenen Markierung der Knoten als STARTED markiert.

$e_2 = (Perform, 1, John)$ Durch e_2 wurde der Mitarbeiter $(John)$ protokolliert, der für die Ausführung des Tasks mit der ID 1 verantwortlich ist. Die Ausführungsregel für PERFORM Events besagt, dass die jeweilige *Performkante* mit dem Mitarbeiter markiert wird und die Überprüfung erst nach der Ausführung des Prozess-Ablaufprotokolls stattfindet.

$e_3 = (End, A, 1, 12{:}30)$ Hiermit wurde das Ende der Ausführung des Tasks A mit der ID 1 festgehalten. Bei der Suche nach einer passenden Startmarkierung wird man in der Markierung m_2 fündig. Die ID und der Typ des dort vermerkten Start Events stimmt mit diesem Event überein. Damit wurde eine vollständige Ausführung von A beobachtet, weswegen der entsprechende Knoten die Ausführungsmarkierung EXECUTED erhält.

$e_4 = (Start, A, 2, 12{:}45)$,
$e_5 = (Perform, 2, Sam)$,
$e_6 = (End, A, 2, 13{:}30)$ Die Events e_4-e_6 beschreiben genauso wie die Events e_1-e_3 die Ausführung eines Tasks A. Im Unterschied zu den Events e_1-e_3 ist dieses mal Sam der verantwortliche Mitarbeiter und die Taskinstanz ist durch die ID 2 gekennzeichnet.

Für die Aktualisierung der vorhandenen Markierungen mach dies keinen Unterschied. Der Ausführungssemantik von Consequence Knoten entsprechend, wird somit wie schon bei e_1 eine neue Markierung von m_1 abgeleitet, um dort die erneute Ausführung des Tasks A festzuhalten. Als verantwortlicher Mitarbeiter wird dabei Sam vermerkt.

$\underline{e_7 = (Start, B, 3, 14{:}45)}$,
$\underline{e_8 = (Perform, 3, John)}$,
$\underline{e_9 = (End, B, 3, 15{:}00)}$ Durch die Events e_7-e_9 wurde die Ausführung des Tasks B mit der ID 3 festgehalten. Aufgrund des Tasktyps passen die Events zu dem Knoten B. Hierbei handelt es sich nun aber um einen ANTEOCC Knoten, deswegen sind nur Vorgängerknoten vom selben Typ relevant. Da der Knoten B in diesem Beispiel keine ANTEOCC Vorgänger besitzt, kann B in allen drei Markierungen gestartet werden, auch in der Markierung m_1. Die Ausführungsregeln für Antecedence Knoten befolgend, wird eine Kopie von ms_6 angefertigt. In der daraus resultierenden MARKSTRUCTURE ms_9 wird nun die Ausführung von B vermerkt. Dazu wird in den enthaltenen Markierungen m_1, m_2 und m_3 der Knoten B und die dazugehörige Performkante, den Events e_7-e_9 entsprechend, markiert.

Im Anschluss an die Ausführung des Prozess-Ablaufprotokolls werden die Markierungen wieder abgeschlossen und für die Auswertung vorbereitet. Wie bereits bekannt, wird deswegen allen Knoten, die bis dato nicht ausgeführt wurden, der Ausführungszustand NOTEXECUTED zugewiesen.

Im Folgenden wird nun versucht eine Belegung der Resource Knoten zu finden. In diesem Fall sind alle vorhandenen StaffMember Nodes durch genau eine Performkante spezifiziert. Die Belegung der Resource Knoten folgt somit direkt aus den Markierungen der Performkanten.

Anschließend muss noch die Zeit- und die Ressourcen-Bedingung überprüft werden. Dies ist nur in den Markierungen m_2 und m_3 von ms_9 möglich, denn nur dort wurden beide Seiten der Bedingung markiert. In m_2 liegen zwischen dem Ende von A_1 und dem Start von B_3 zwei Stunden und fünfzehn Minuten, jedoch wurden A und B vom gleichen Mitarbeiter ausgeführt. Somit wird zwar die Zeit Bedingung eingehalten, aber die Resource Relation ist verletzt. In der Markierung m_3 liegt der umgekehrte Fall vor,

die Resource Relation ist erfüllt und die Zeit Bedingung verletzt. Denn A und B wurden hier zwar von verschiedenen Mitarbeitern ausgeführt, jedoch liegen zwischen dem Ende von A_2 und dem Start von B_3 keine zwei Stunden, sondern lediglich eine Stunde und fünfzehn Minuten. Damit lautet der Erfüllungsstatus der CONSEXMARK Markierungen von m_2 und von m_3 VIOLATED.

Nun kommen wir zur abschließenden Bewertung der Markierungen. Die MARKSTRUCTURE ms_6 ist deaktiviert, denn es fehlt die Ausführung von B, welche für eine Aktivierung notwendig ist. Im Gegensatz dazu enthält die ANTEEXMARK Markierung von ms_9 eine Ausführung von B. Damit lautet der Aktivierungszustand von ms_9 ACTIVATED. Das Ergebnis der Compliance Prüfung bestimmt sich somit ausschließlich aus den CONSEXMARK Markierungen von ms_9. Wie bereits zuvor gesehen, ist das Consequence-Pattern bei den Markierungen m_2 und m_3 jedoch verletzt. Bleibt nur noch die Markierung m_1, doch auch diese verletzt das Consequence-Pattern, denn es fehlt die Ausführung von A. Damit existiert für ms_9, obwohl aktiviert, keine erfüllende CONSEXMARK Markierung. Deswegen lautet das Ergebnis der Compliance Prüfung VIOLATED.

4 Architektur

In diesem Kapitel wird die der Implementierung zugrundeliegende Architektur besprochen. Dabei wird zunächst das grundsätzliche Architekturkonzept beschrieben. In Abschnitt 4.1 folgt eine Beschreibung des Datenmodells, bevor in Abschnitt 4.2 der Entwurf der grafischen Oberfläche erläutert wird.

Die Programmarchitektur orientiert sich konzeptionell am MVC[1] Programmierparadigma. Dem entsprechend lässt sich das Programm in drei Komponenten aufgliedern. Durch das *Datenmodell* werden verschiedene Datenobjekte definiert, die beispielsweise für die Abbildung und Speicherung von Prozess-Ablaufprotokollen und Markierungen benötigt werden. Die *grafische Oberfläche* des Programms ist für die Interaktion mit dem Benutzer verantwortlich. So wird das Programm einerseits durch vom Nutzer getätigte Eingaben gesteuert, andererseits wird dem Benutzer darüber zum Beispiel das Ergebnis

[1] Model View Controller

einer Compliance Prüfung präsentiert. Die *Geschäftslogik* für die Überwachung von Compliance Regeln per eCRG ist Bestandteil von Kapitel 5 und stellt die dritte Programmkomponente dar. Anhand dieser Aufteilung wurde auch der Programmcode der Implementierung strukturiert und in entsprechende Pakete untergliedert.

4.1. Datenmodell

Dieser Abschnitt beschreibt das Datenmodell für die Implementierung. Das in den folgenden Unterkapiteln definierte Klassengerüst, stellt die Grundlage für die Abbildung der Datenobjekte dar, die im Kontext der eCRG Überwachung relevant sind.

4.1.1. Prozessumgebung

Die Prozessumgebung (*Process Environment*) entspricht einer globalen Sammlung aller möglichen Datenobjekte, die ein Geschäftsprozess zur Laufzeit annehmen kann. In Abbildung 4.1 ist das Klassendiagramm für die Prozessumgebung dargestellt. Die enthaltenen Klassen sind an die *Definition 1* aus [KRL+13a] angelehnt.
Im Mittelpunkt steht dabei die `ProcessEnvironment` Klasse. Alle existierenden Objekte sind in dieser Klasse registriert. Für die Verwaltung der verschiedenen möglichen Werte hält die Klasse für jeden Datentyp eine eigene Datenstruktur in Form von *Sets* (Mengen). In diesen Mengen sind alle der Umgebung hinzugefügten Datenobjekte enthalten. Die Prozessinstanzen und damit die Ablaufprotokolle können deswegen, genauso wie die eCRG, ausschließlich auf Datenobjekte verweisen, die in der Prozessumgebung vorhanden sind.
Die Klassen `TaskType` und `MessageType` sind beide von der abstrakten Klasse `Type` abgeleitet. Durch sie wird der Typ eines Tasks oder einer Messsage spezifiziert. Die Klassen `TaskIdentifier` und `MessageIdentifier` dienen hingegen der exakten Referenzierung einer bestimmten Task- beziehungsweise Message-Instanz. Anhand der jeweiligen ID lässt sich damit jede Instanz eindeutig identifizieren.
Ein Objekt der `PointInTime` Klasse entspricht einem bestimmten Zeitpunkt, welcher durch den Wert des Objekts festgelegt wird.

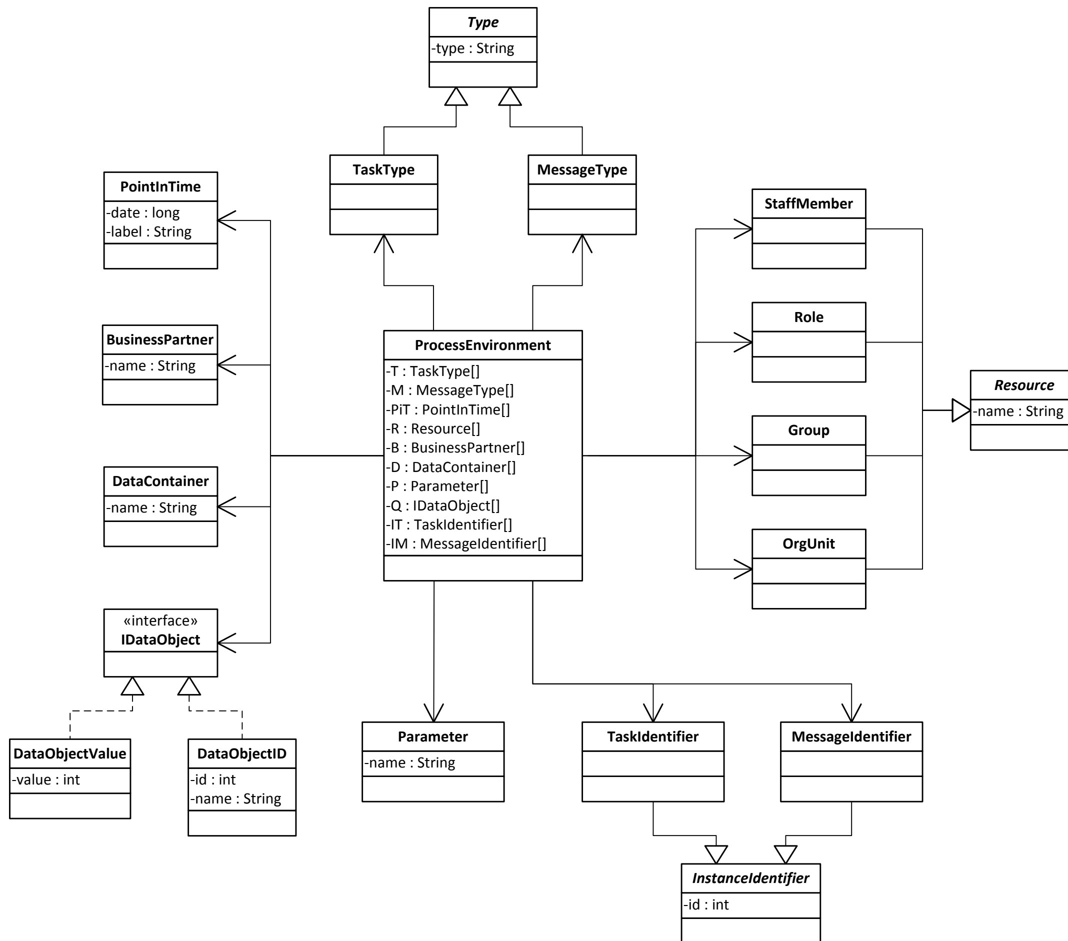


Abbildung 4.1.: Klassendiagramm: Prozessumgebung

Durch die Klasse `BusinessPartner` werden Geschäftspartner beschrieben, welche beim Versand und Empfang einer Nachricht angegeben werden müssen.
Zur Referenzierung globaler Datenspeicher werden Objekte vom Typ `DataContainer` instantiiert. Die Implementierungen des Interface `IDataObject` dienen hingegen der Angabe eines bestimmten Datenobjekts oder Datenwerts. Je nachdem wird entweder ein Objekt vom Typ `DataObjectID` oder `DataObjectValue` erzeugt. Die Klasse `Parameter` definiert dem Namen entsprechend einen Parameternamen. Dadurch lassen sich die ansonsten anonymen Datenobjekte mit einem Namen verknüpfen.
Zuletzt gibt es noch vier verschiedene Realisierungen der abstrakten Klasse `Resource`. Je nach Typ, wird entweder ein einzelner Mitarbeiter, eine bestimmte Rolle oder Funktion, eine Gruppe oder eine Abteilung referenziert.

4.1.2. Bedingungen und Relationen

Basierend auf der vorangegangenen Definition der Prozessumgebung werden in diesem Abschnitt die Klassen für die Abbildung von *Conditions & Relations* definiert, siehe dazu Abbildung 4.2. Damit lassen sich Bedingungen und Relationen bezüglich der Zeit, des Datenwerts und für Ressourcen formulieren. Alle möglichen Bedingungen implementieren das Interface `ICondition`, die Relationen das Interface `IRelation`. Im Weiteren wird sowohl bei Bedingungen als auch bei Relationen, zwischen werte-spezifischen und freidefinierbaren Bedingungen/Relationen unterschieden. Während die werte-spezifischen

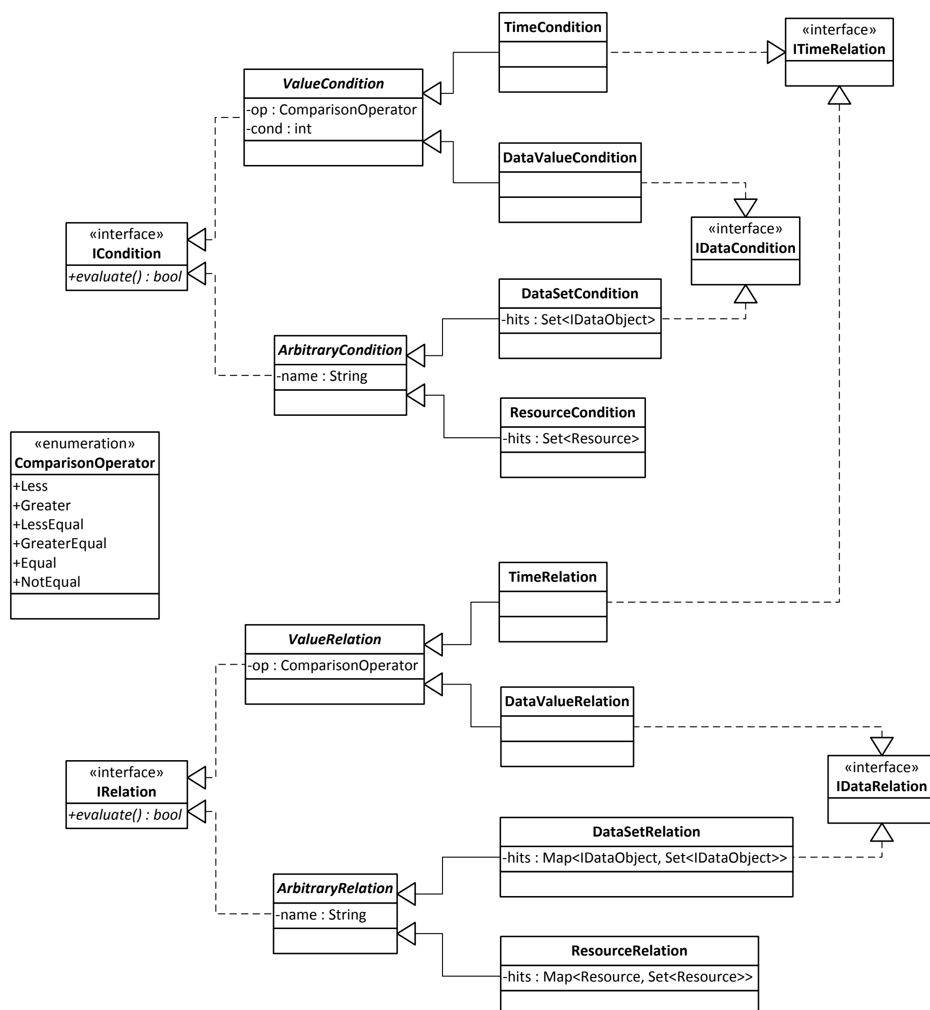


Abbildung 4.2.: Klassendiagramm: Conditions & Relations

Typen `ValueCondition` und `ValueRelation` einen Vergleichsoperator besitzen, werden die frei-definierbaren zunächst nur durch einen Namen, wie beispielsweise *"istChefVon"*, beschrieben. Als Vergleichsoperator (`ComparisonOperator`) stehen folgende sechs Vergleiche zur Verfügung: $Less <$, $Greater >$, $LessEqual \leq$, $GreaterEqual \geq$, $Equal =$ und $NotEqual \neq$.
Mittels `TimeConditions` werden Bedingungen abgebildet, welche die Zeit betreffen. Bei den Datenbedingungen existieren zwei verschiedene Ausprägungen, denn es wird zwischen `DataValueConditions` und `DataSetConditions` unterschieden. Im Gegensatz zu einer `DataValueCondition` wird durch eine `DataSetCondition` keine direkt per Vergleich auswertbare Bedingung formuliert, sondern es wird lediglich eine Treffermenge von gültigen Werten angegeben. Beides sind aber Realisierungen des Interface `IDataCondition`. Bei `ResourceConditions` existiert nur die Variante mit einer fest vorgegebenen Treffermenge, denn ein Wertevergleich macht an dieser Stelle keinen Sinn.
Für jede der bis dato beschriebenen Bedingungen existiert auch eine entsprechende Relation. Der einzige Unterschied dabei ist, dass die Relationen nun zweistellig sind. Ein Sonderfall stellen die `TimeConditions` dar, denn diese können auch als `ITimeRelation` aufgefasst werden. Damit kann eine `TimeCondition` auch mit zwei Parametern ausgewertet werden. In diesem Fall wird anhand des Differenzbetrags der beiden Parameter die Bedingung überprüft. Dies ist notwendig, um eine Bedingung formulieren zu können, die sich auf den zeitlichen Abstand zwischen zwei Knoten bezieht. Zum Beispiel ließe sich dadurch folgendes formulieren: Zwischen dem Ende der Ausführung von T_A und dem Start von T_B sollen maximal drei Tage vergangen sein.

4.1.3. Prozess-Ablaufprotokoll – Schnittstelle

Abbildung 4.3 zeigt das Klassendiagramm für die Prozess-Ablaufprotokolle. Die zentrale Klasse hierbei ist die `ProcessTrace` Klasse. Jede Instanz dieser Klasse entspricht einem Ablaufprotokoll und besitzt neben einem Namen noch eine Menge von Events, welche den Ablauf der Prozessinstanz beschreiben. Die Klasse `Event` ist, ebenso wie die Klassen `TaskEvent`, `MessageEvent` und `IOEvent`, eine abstrakte Klasse.

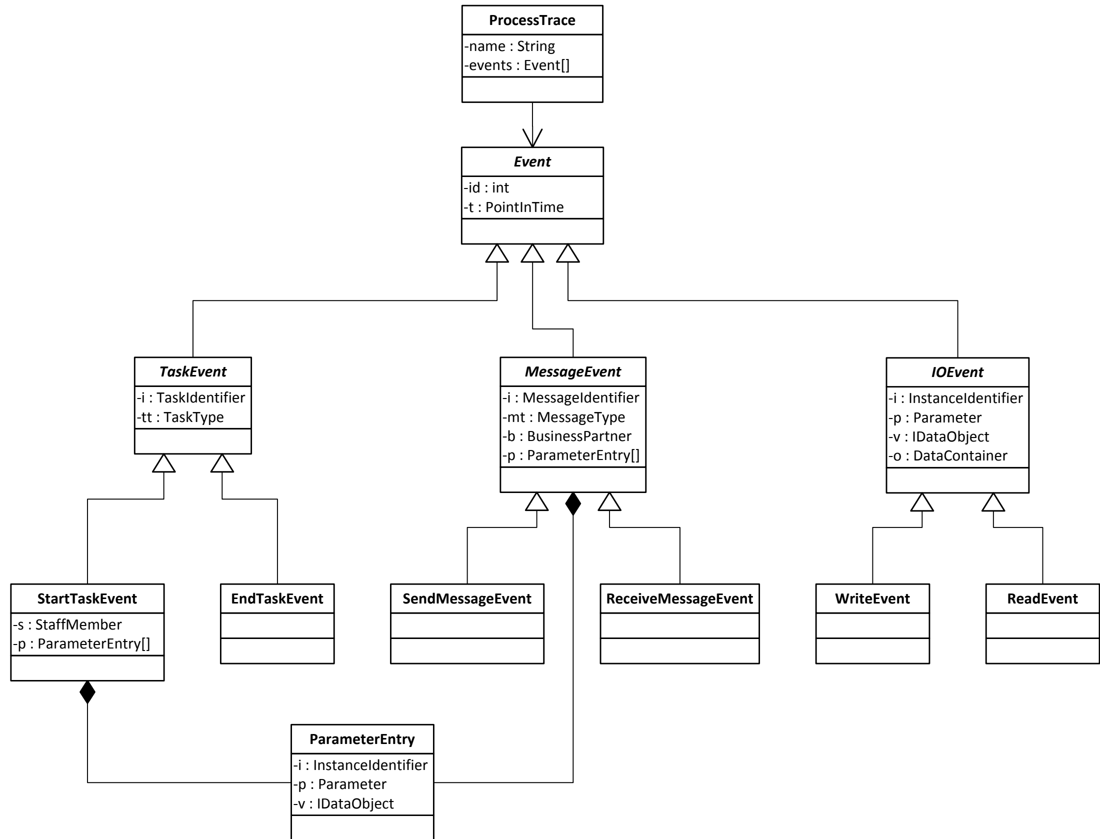


Abbildung 4.3.: Klassendiagramm: Prozess-Ablaufprotokolle

Folglich lassen sich davon keine Instanzen erzeugen. Deswegen können in einem Ablaufprotokoll nur Events enthalten sein, deren Typ mit einem der folgenden übereinstimmt: `StartTaskEvent`, `EndTaskEvent`, `SendMessageEvent`, `ReceiveMessageEvent`, `WriteEvent` oder `ReadEvent`.

Jedes `Event` besitzt eine ID und ist dadurch eindeutig gekennzeichnet. Denn innerhalb ein und desselben `ProcessTraces` darf es nicht mehrere Events mit der gleichen ID geben. Außerdem ist in jedem Event der Zeitpunkt vermerkt, in dem das Event eingetreten ist. Anhand dieser Zeitpunkte bestimmt sich die Anordnung der Events innerhalb des `ProcessTraces`. Wenn sie der Reihe nach entnommen werden, gewährleistet dies die zeitliche Abfolge der Events.

Beiden `TaskEvents` gemeinsam ist Angabe eines $TaskIdentifiers\ i$ und die Spezifizierung des $TaskTypes\ tt$. Des Weiteren besitzt ein `StartTaskEvent` noch zwei weitere Eigenschaften. Zum einen wird der $StaffMember\ s$ mit angegeben, der den

Task gestartet hat. Denn die Angabe des verantwortlichen Mitarbeiters für die Ausführung eines Tasks wird hierbei nicht als eigenständiges Event realisiert. Stattdessen wird das PERFORM Event mit in das START Event hinein integriert. Des Weiteren gibt es noch das Attribut p, indem mehrere $ParameterEntries$ abgespeichert werden können.
Jeder `ParameterEntry` entspricht einer Kombination aus Datenobjekt und Parametername. Da sie nur entweder einem `StartTaskEvent` oder einem `MessageEvent` angehören können, wird ein `ParameterEntry` durch folgende Eigenschaften beschrieben: einem $InstanceIdentifier\ i$, einem $Parameter\ p$ und einem $IDataObject\ v$. Dadurch wird zum einen der Task beziehungsweise die Message referenziert, zu dem der Eintrag gehört und zum anderen wird der Parameter mit einem Datenwert verknüpft.
Bei den `MessageEvents` gibt es keine Unterschiede, beide Events werden durch einen $MessageIdentifier\ i$, einen $MessageType\ mt$ und einen $BusinessPartner\ p$ beschrieben. Soweit stimmt dies auch mit der Definition, der in Kapitel 3 kennengelernten SEND und RECEIVE Events, überein. Zudem besteht nun noch die Möglichkeit, wie schon bei den `StartTaskEvents`, zusätzlich $ParameterEntries$ mit anzugeben.
Zuletzt gibt es noch die beiden `IOEvents`, welche durch die Klassen `WriteEvent` und `ReadEvent` realisiert werden. Charakterisiert sind sie durch einen $InstanceIdentifier\ i$, einen $Parameter\ p$, ein $IDataObject\ v$ und einen $DataContainer\ o$. Damit wird die verantwortliche Aktivität, deren Parameter und der Datenwert, der aus dem (in den) Datencontainer gelesen (geschrieben) wurde, festgelegt.

4.1.4. eCRG – Schnittstelle

Der Aufbau einer eCRG ist in der Abbildung 4.4 dargestellt. Dieser Entwurf entspricht relativ genau der eCRG Definition: Jede eCRG wird durch einen Graphen repräsentiert, wobei der Graph aus verschiedenen Elementen bestehen kann. Dementsprechend erbt die Klasse `eCRG` von der abstrakten Klasse `Graph`. Diese wiederum besteht aus einer Menge von *Nodes*, *Edges* und *Attachments*. Die entsprechenden Klassen `Node`, `Edge` und `Attachment` erben alle von der abstrakten Oberklasse `GraphElement`, um sie als Element des eCRG Graphen zu kennzeichnen. Darüber hinaus ist bei jedem `GraphElement` das `Pattern` vermerkt, dem es zuzuweisen ist.

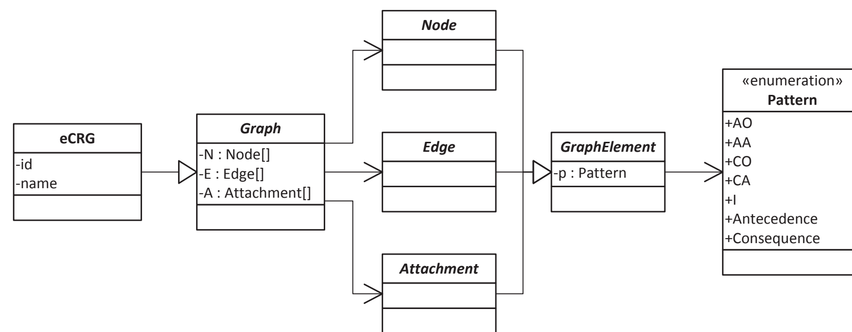


Abbildung 4.4.: Klassendiagramm: eCRG

Um bei den Graph-Elementen nicht nur zwischen *Node*, *Edge* und *Attachment* unterscheiden zu können, sind die entsprechenden Klassen abstrakt gehalten. Somit ist es nicht möglich, von diesen Klassen Instanzen zu erzeugen. Stattdessen werden die verschiedenen Typen von Knoten, Kanten und Attachments durch eine jeweils eigene Klasse implementiert. Dadurch ist es möglich, die unterschiedlichen Semantiken der Elemente abzubilden. Sodass bei der Definition einer Kante beispielsweise, direkt der genaue Knotentyp angegeben werden kann. Andernfalls müsste man sich auf die vergleichsweise unspezifische Klasse `Node` beziehen und es wäre eine explizite Überprüfung des Knotentyps notwendig. In der Tabelle 4.1 sind alle Unterklassen aufgeführt, welche die abstrakten Klassen `Node`, `Edge` und `Attachment` implementieren.

Nodes	**Edges**	**Attachments**
TaskNode	SequenceFlowConnector	DataConditionParameterAttachm.
MessageNode	ExclusiveConnector	DataObjectConditionAttachment
PointInTimeNode	AlternativeConnector	DataFlowConditionAttachment
DataNode	MessageFlowConnector	DataFlowObjectAttachment
ResourceNode	TimeDistanceConnector	TaskDurationAttachment
	WritingDataFlowContainerC.	ResourceConditionAttachment
	ReadingDataFlowContainerC.	
	WritingDataFlowObjectC.	
	ReadingDataFlowObjectC.	
	PerformingRelationConnector	
	ResourceRelationConnector	
	DataRelationConnector	
	DataConditionConnector	
	DataObjectConditionConnector	

Tabelle 4.1.: Die implementierenden Unterklassen von *Node*, *Edge* und *Attachment*

4.1.5. Datenstrukturen für die Markierung

Dieser Abschnitt beschäftigt sich mit den Klassen, die zur Abbildung der Markierungszustände verwendet werden. Abbildung 4.5 zeigt das dazugehörige Klassendiagramm. Die Notation der Klassen orientiert sich dabei an den in Kapitel 3 eingeführten Datenstrukturen zur Markierung von eCRG Graphen.

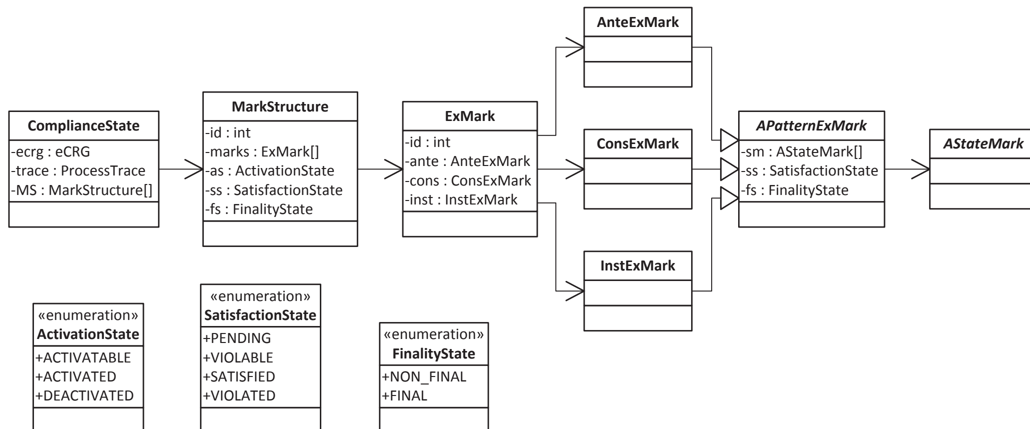


Abbildung 4.5.: Klassendiagramm: Markierungen

Die im Klassendiagramm abgebildete Kette der Klassen von links nach rechts, entspricht der Hierarchie, die zwischen den einzelnen Klassen besteht. Die `ComplianceState` Klasse steht dabei ganz oben in der Hierarchie. Durch diese Klasse werden alle während der Ausführung erreichten Compliance Zustände und deren Markierungen zu einer Einheit zusammengefasst. Jede Compliance Überprüfung beginnt mit der Instantiierung eines solchen Objekts. Deswegen hält jedes `ComplianceState` Objekt auch einen Referenz auf die zu überprüfende `eCRG` und den zu überwachenden `ProcessTrace`. Des Weiteren gibt es noch ein Attribut für die Speicherung der verschiedenen `MarkStructures`.

Auf der nächsten Ebene folgt die Klasse `MarkStructure`. Die Namensübereinstimmung dieser Klasse mit der in Kapitel 3 eingeführten Datenstruktur ist dabei beabsichtigt. Denn sie dient der Abbildung genau dieser Markierungen. Zunächst einmal ist jedes `MarkStructure` Objekt durch eine eindeutige ID gekennzeichnet. Dies dient der Unterscheidung der verschiedenen Instanzen dieser Klasse. Außerdem be-

sitzt diese Klasse noch drei Attribute zur Speicherung des Aktivierungszustands, des Erfüllungsstatus und des Zustands, welcher die Finalität der Markierungen widerspiegelt. Die möglichen Werte für diese drei Attribute sind durch die Enumerationen `ActivationState`, `SatisfactionState` und `FinalityState` festgelegt. Zuletzt beinhaltet jede `MarkStructure` noch eine Menge von `ExMark` Markierungen. Voraussetzung für die enthaltenen `ExMark` Markierungen ist jedoch, dass alle dieselbe Antecedence Markierung repräsentieren und somit den gleichen Aktivierungszustand besitzen.
Auch bei der `ExMark` Klasse ist der Name bewusst gewählt, denn dadurch werden die einzelnen Markierungen einer eCRG abgebildet. Die ID wird auch in diesem Fall zur Unterscheidung der Markierungen benötigt und ist innerhalb einer `MarkStructure` immer eindeutig. In den drei noch verbliebenen Attributen werden die einzelnen Markierungen, für das Antecedence, Consequence und Instance-Pattern einer eCRG, getrennt voneinander gespeichert. Die gemeinsame Oberklasse dieser drei Markierungen ist die `APatternExMark` Klasse. Von dieser abstrakten Klasse erben die Klassen zur Abbildung der Pattern-Markierungen. Außer den einzelnen Markierungen für die eCRG Graph-Elemente beinhaltet diese Klasse noch zwei weitere Attribute für die Speicherung des lokalen Erfüllungs- und Finalitäts-Zustands. Mit lokal ist die Begrenzung auf das jeweilige Pattern gemeint.
Zuletzt bleibt noch die `AStateMark` Klasse. Dies stellt die Basisklasse für alle Markierungen dar, die einem bestimmten Graph-Element zugewiesen werden sollen. Implementierungen dieser Klassen wären zum Beispiel die `TaskStateMark` oder die `DataFlowStateMark` Markierung.

4.2. Entwurf der grafischen Benutzeroberfläche

In diesem Abschnitt wird der Entwurf der grafischen Benutzeroberfläche (GUI) besprochen. Das dabei entwickelte Konzept dient der prototypischen Implementierung einer eCRG Ausführungsengine. Der Entwurf der Programmoberfläche ist in Abbildung 4.6 in Form eines Mockups dargestellt. Im Folgenden wird dieser Entwurf erläutert, wo-

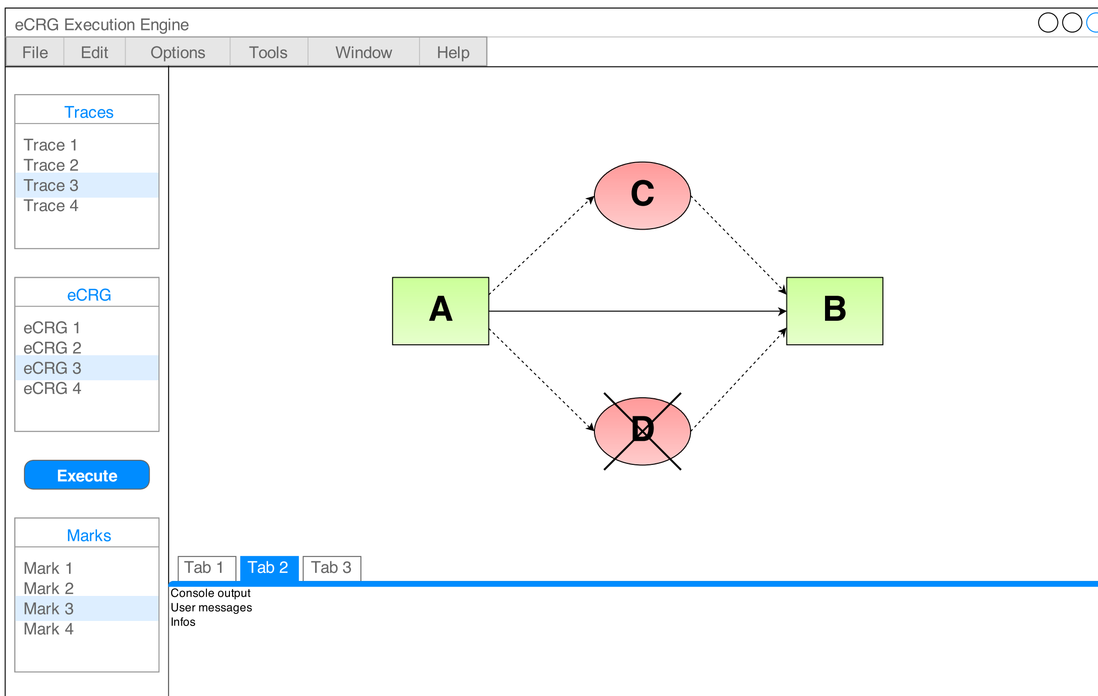


Abbildung 4.6.: Mockup der grafischen Oberfläche

bei auf die Anordnung und den Zweck der verschiedenen Oberflächenkomponenten eingegangen wird.

Im wesentlichen setzt sich die GUI aus den folgenden vier Komponenten zusammen:

- Einer *Menüleiste*, die am oberen Rand des Programmfensters angesiedelt ist und die klassischen Funktionen zur Steuerung des Programms beherbergt.
- Ein *Auswahlbereich* für den die linke Seite der Oberfläche vorgesehen ist. Über die Komponenten in diesem Bereich lässt sich die Ausführung einer eCRG starten. Dazu muss zunächst in der oberen Liste ein Ablaufprotokoll (Trace) ausgewählt werden, bevor im zweiten Schritt die zu überprüfende eCRG angeklickt wird. Durch betätigen des $Execute$ Buttons startet schließlich die Ausführung der gewählten eCRG. Ist diese abgeschlossen, werden in der unteren Liste die verschiedenen Markierungen angezeigt, die währende der Ausführung zustande gekommen sind. Durch einen Mausklick auf einen dieser Listeneinträge, wird der durch die Markierung repräsentierte Compliance Zustand visuell dargestellt.

- Der untere Bereich des Fensters ist für weitere Programmkomponenten freigehalten, die über entsprechende Reiter angewählt werden können. Bestandteil dieser Komponente ist beispielsweise eine Art *Konsole*, für eine textuelle Ausgabe. Darüber kann das Programm dem Benutzer nützliche Informationen und Statusmeldungen präsentieren. Der Nutzer soll dabei aber auch die Möglichkeit haben, die gesamte Komponente ausblenden zu können, um so mehr Platz für die Anzeige der Markierungszustände zu erhalten.
- Der größte Bereich der Benutzeroberfläche, im Zentrum des Fensters, ist für die *Anzeigefläche* bestimmt. Auf dieser Fläche werden die Markierungszustände visualisiert und dem Nutzer zugänglich gemacht. Dazu wird der Graph der überwachten eCRG gezeichnet und die einzelnen Knoten entsprechend ihrer Ausführungsmarkierungen farblich gekennzeichnet. Das im Mockup enthaltene Beispiel zeigt eine aktivierte Markierung, deren Consequence-Pattern verletzt ist. Dies ist durch eine Hervorhebung der beiden, für die Verletzung verantwortlichen, Knoten kenntlich gemacht. Dadurch wird der Nutzer bei der Analyse der erreichten Markierungszustände unterstützt, wodurch er sich schnell einen Überblick verschaffen kann und dabei unter Umständen Compliance-Verletzungen aufdeckt.

5 Implementierung

Thema dieses Kapitels ist die Implementierung des zu entwickelnden Prototyps einer eCRG Ausführungsengine. Dazu werden im Folgenden verschiedene Aspekte der Implementierung näher beschrieben. Das Programm, welches die Realisierung dieses Prototyps darstellt, ermöglicht die Überwachung von, per eCRG definierten, Compliance Regeln. Zusätzlich wird dadurch der von eCRG gewählte Ansatz und das dahinterstehende Konzept verifiziert, um deren Tauglichkeit nachzuweisen.

Im Abschnitt 5.1 werden zunächst die technischen Grundlagen der Implementierung besprochen. Der darauffolgende Teilabschnitt 5.2 beschäftigt sich mit der grafischen Benutzeroberfläche des Programms und der JGraphX Library. Diese Bibliothek unterstützt die Visualisierung von Graphen und vereinfacht so die Ausgabe der erreichten Markierungszustände. Zum Schluss wird im Abschnitt 5.3 die Implementierung des Monitorings besprochen. Dabei werden die verschiedenen Algorithmen erläutert, welche für die Überwachung der eCRG Regeln benötigt werden.

5.1. Technische Grundlagen

Die eCRG Ausführungsengine wurde in der Programmiersprache Java^{TM} unter Verwendung des Java SE Development Kit in der Version 7 Update 55 implementiert.

Für die Programmierung wurde die zum Zeitpunkt der Implementierung aktuelle Version der Entwicklungsumgebung Eclipse eingesetzt. Konkret war dies die Standard Edition von Eclipse in der Version 4.3.2 .

Die grafische Darstellung der, während der Ausführung, erreichten Markierungszustände war eine der Hauptanforderungen für das zu entwickelnde Programm. Bei der Realisierung dessen kam die JGraphX Library zum Einsatz. Mit Hilfe dieser Bibliothek lässt sich der Graph einer eCRG zusammen mit der jeweiligen Markierung auf der Anzeigefläche darstellen. Für die Implementierung wurde die Library in der Version 2.5.0.0 verwendet. Im Nachfolgenden Unterabschnitt wird das hinter JGraphX stehende Konzept und die von der Bibliothek zur Verfügung gestellten Funktionen näher beschrieben.

Für den PDF-Export der, durch JGraphX visualisierten, Markierungszustände auf der Anzeigefläche in das PDF-Format wurde die iText® Library in der Version 5.5.0 genutzt. Die dadurch realisierte Möglichkeit zur Speicherung der Markierungen stellt jedoch keine Kernfunktion des Programms dar, sondern dient lediglich dem Export des Graphen in ein Bildformat zur weiteren Verarbeitung.

JGraphX Bibliothek

Um die Visualisierungs-Komponente des Programms zu realisieren, wurde eine Bibliothek benötigt, welche die folgenden Kriterien erfüllt:

1. ermöglicht auf einfache Weise die Visualisierung von Graphen
2. unterstützt die Programmiersprache Java
3. Open Source beziehungsweise frei-verwendbar durch kostenlose Lizenz
4. die Darstellung der Knoten und Kanten des Graphen kann durch eigene Styles und Shapes (Symbole) angepasst werden

Anhand dieser Liste von Punkten wurden verschiedene, für die Implementierung in Frage kommende, Bibliotheken miteinander verglichen. Die Entscheidung fiel schließlich zugunsten der JGraphX Library, da von dieser alle vier der zuvor genannten Kriterien erfüllt werden.

JGraphX[1] bezeichnet sich selbst als *Java Swing Visualization Library* und entspricht der Version 6 der Vorgängerbibliothek JGraph. Entwickelt wird sie von der Firma JGraph Ltd. Aufgrund einer kompletten Überarbeitung und Neuimplementierung der Bibliothek wurde einen Namensänderung durchgeführt, um kenntlich zu machen, dass es sich bei JGraph 6 nicht einfach nur um eine neue Version der Library handelt. Denn mit JGraphX wurde eine neue API eingeführt, die nicht abwärtskompatibel zu JGraph 5 ist.

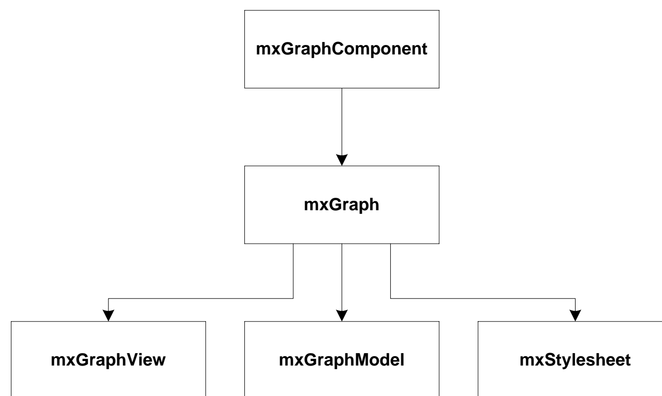
Die mxGraph Produktfamilie beinhaltet neben JGraphX noch weitere Realisierungen dieser Bibliothek für andere Plattformen. Allen ist gemeinsam, dass sie auf derselben Architektur beruhen und soweit möglich die gleiche API besitzen. Die Kernfunktionalität von mxGraph ist die Anzeige interaktiver Diagramme und Graphen. Während JGraphX unter der (modifizierten 3-Klausel-) BSD Lizenz steht, wird von JGraph Ltd auch eine kommerzielle Version für Javascript vertrieben. Dies sei erwähnt, da mit mxGraph in den meisten Fällen diese nicht frei-verfügbare Version bezeichnet wird [UMJ14]. Die Möglichkeiten dieser Bibliothek werden durch die Webapplikation draw.io[2] veranschaulicht. Damit ist es möglich online Diagramme zu erstellen, wobei eine Vielzahl von verschiedenen Anwendungsszenarien unterstützt wird. Zwar basiert diese Webapplikation auf der lizenzpflichtigen Version mxGraph, jedoch gibt sie einen guten Überblick über die Funktionen von JGraphX, da der Funktionsumfang identisch ist. Dieser lässt sich in die folgenden vier Untergruppen einteilen: I. Visualisierung, II. Interaktion (Ändern), III. Layout (Anordnen) und IV. Analyse von Graphen. Im Folgenden wird davon nur die Visualisierungs und die Layout Komponente in Anspruch genommen.

JGraphX wurde in Java programmiert und ist vollständig mit Java Swing kompatibel. Dafür stellt die Bibliothek eine `JComponent` zur Verfügung, die einer per Swing definierten grafischen Benutzeroberfläche hinzuzufügen ist, um eine Anzeigefläche zur Darstellung eines Graphen zu erhalten. Im $HelloWorld$ Beispiel (Quelltext 5.1) entspricht dies den

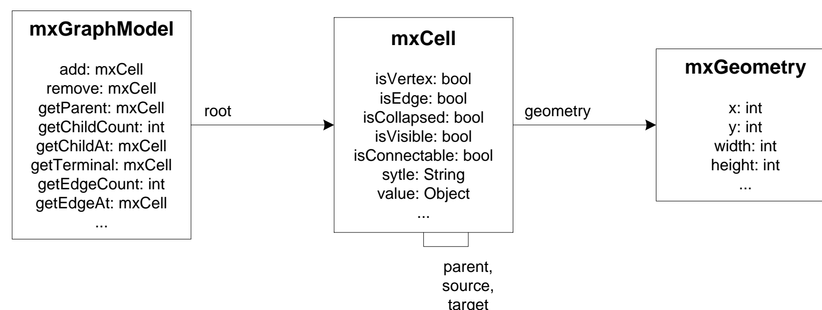
[1] JGraphX Projektwebsite: `https://github.com/jgraph/jgraphx`
[2] Link zur Webapplikation draw.io: `https://www.draw.io/`

Zeilen 17 und 18, wobei eine `mxGraphComponent` dem JFrame hinzugefügt wird. Ein weiterer Punkt der für JGraphX sprach, war die Unterstützung sogenannter Styles. Damit lässt sich die grafische Darstellung der Graph-Elemente relativ einfach an die eCRG Notation anpassen, siehe dazu das Kapitel 5.2.3.

Ein Ausschnitt des Datenmodells von JGraphX ist in Abbildung 5.1 dargestellt. Die Klasse `mxGraph` entspricht der Hauptklasse (*main class*) von JGraphX und steht im Mittelpunkt der Programmierung. In Abbildung 5.1a sind die wichtigsten Klassen abgebildet, die



(a) Mit mxGraph im Zusammenhang stehende Klassen



(b) Das Graph-Modell im Detail

Abbildung 5.1.: Datenmodell von JGraphX aus [UMJ14]

im Zusammenhang mit der Klasse `mxGraph` stehen. Die Klasse `mxGraphComponent` entspricht der Java Swing Komponente für die Darstellung eines `mxGraph`. Die Klasse `mxGraph` verweist wiederum auf die Klassen `mxGraphView`, `mxGraphModel` und `mxStylesheet`. Wobei die Klasse `mxGraphModel` für die Verwaltung der Graphstruktur verantwortlich ist und das Graph-Modell speichert. Die Klasse `mxGraphView` re-

präsentiert hingegen die Ansicht (View) des Graphen und hält die Zustände der verschiedenen Elemente. Darunter fallen beispielsweise die Koordinaten von Knoten oder der Pfad einer Kante. Außerdem ist diese Klasse für die Bestimmung der Grenzen beziehungsweise der Dimension eines Graphen verantwortlich. Die Darstellung und damit das Aussehen der einzelnen Graphelemente wird in einem Objekt der Klasse `mxStylesheet` abgespeichert. Dabei kann für jedes Element individuell eine entsprechende Styledefinition hinterlegt werden. Dies geschieht durch Angabe eine Reihe von $Key \rightarrow Value$ Kombinationen, wie zum Beispiel $fillColor = green$.

Das Graph-Modell zur Speicherung der Struktur des Graphen ist etwas detaillierter in Abbildung 5.1b dargestellt. Was dabei auffällt, ist, dass zunächst einmal nicht zwischen Knoten und Kanten unterschieden wird. Beide werden durch ein `mxCell` Objekt abgebildet. Die Unterscheidung zwischen Knoten oder Kante findet innerhalb dieser Klasse statt. Jedes `mxCell` Objekt verweist zudem auf ein Objekt vom Typ `mxGeometry`, welches die Position und die Größe des Elements beinhaltet. Des Weiteren besitzt die Klasse `mxCell` ein Attribut $value$, um anwendungsspezifische Daten zu speichern.
Die Klasse `mxGraphModel` verwaltet die Graphstruktur und bietet Methoden zur Anpassung dieser an. Außer den Funktionen für das Hinzufügen und Löschen von Elementen, stehen dabei auch noch verschiedene Methoden zur Abfrage bestimmter Aspekte der Graphstruktur zur Verfügung. Denn das Graph-Modell ist hierarchisch konzipiert und in verschiedene Ebenen untergliedert. Den Einstieg in die Graphstruktur bildet ein spezielles `mxCell` Objekt in der Funktion $root$. Die Kinder dieses Wurzelelements befinden sich somit auf der ersten Ebene. Um Konzepte wie *Grouping* und *Folding* realisieren zu können, kann ein `mxCell` Objekt wiederum die Funktion $parent$ für ein anderes Element einnehmen, dass sich dann eine Ebene unterhalb des angegebenen Elternobjekts befindet. Der dadurch aufgebaute Strukturbaum repräsentiert das jeweilige Graph-Modell, dessen Eigenschaften über die Funktionen der Klasse `mxGraphModel` abgefragt und angepasst werden können. Die meisten dieser Methoden sind auch über die Klasse `mxGraph` erreichbar, sodass der Umweg über das Graph-Modell wegfällt. Wie im nachfolgenden Beispiel zu sehen, besitzt die Klasse `mxGraph` beispielsweise Methoden zum Einfügen neuer Knoten und Kanten, wobei diese wiederum auf die Methoden der Klasse `mxGraphModel` abbilden.

Weitere Details bezüglich dem hinter JGraphX stehendem Konzept finden sich im dazugehörigen Manual [UMJ14]. Ein Minimalbeispiel für die Verwendung von JGraphX stellt das $HelloWorld$ Programm aus Quelltext 5.1 dar. Die Zeilen 5 bis 18 betreffen dabei die JGraphX Library. Durch das Programm wird ein kleines Fenster geöffnet, in

```java
public class HelloWorld extends JFrame {
   public HelloWorld() {
      super("Hello, World!");

      mxGraph graph = new mxGraph();
      Object parent = graph.getDefaultParent();

      graph.getModel().beginUpdate();
      try {
         Object v1 = graph.insertVertex(parent, null, "Hello", 0, 0, 80, 30);
         Object v2 = graph.insertVertex(parent, null, "World!", 200, 150,
             80, 30);
         graph.insertEdge(parent, null, "Edge", v1, v2);
      } finally {
         graph.getModel().endUpdate();
      }

      mxGraphComponent graphComponent = new mxGraphComponent(graph);
      getContentPane().add(graphComponent);
   }

   public static void main(String[] args) {
      HelloWorld frame = new HelloWorld();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(400, 320);
      frame.setVisible(true);
   }
}
```

Quelltext 5.1: JGraphX HelloWorld Beispiel aus [JGraphX Examples]

dem ein Graph mit zwei Knoten und einer Kante dargestellt wird. In der Zeile 5 wird dafür zunächst ein Objekt vom Typ `mxGraph` instantiiert. Dieses Objekt repräsentiert zu diesem Zeitpunkt einen leeren Graphen, denn bis dato wurden dem Graphen noch keine Elemente hinzugefügt.

Das Graph-Modell sieht für jedes enthaltene Element eine Eltern-Kind Beziehung vor. Deswegen muss beim Einfügen neuer Elemente immer die jeweilige $parent$ Zelle mit angegeben werden. Der Defaultwert dafür wird in Zeile 6 abgefragt, die im Weiteren hinzugefügten Elemente befinden sich damit auf der obersten Ebene (Layer) direkt unterhalb der Wurzel.

Des Weiteren bietet JGraphX ein sogenanntes Tranaktionsmodell an, um zusammengehörige Updates des Graph-Modells entsprechend zu kennzeichnen. Damit können mehrere aufeinander folgende Änderungen zu einer Transaktion zusammengefasst werden. Folgendes Programmfragment ist dafür verantwortlich:

```
graph.getModel().beginUpdate();
try {
   // ... zusammengehörige Updates durchführen
} finally {
   graph.getModel().endUpdate();
}
```

Diese Vorgehensweise ist unter anderem für Funktionen wie Undo/Redo und bei einem automatischen Layout wichtig. Zusammengehörige Änderungen können dadurch auf einmal zurückgenommen werden und der Graph wird so nicht bei jeder Aktualisierung neu gelayoutet, sondern nur einmal am Ende der Transaktion.
Durch die Zeilen 10 und 11 werden zwei Knoten in den Graphen eingefügt. Anschließend wird in Zeile 12 eine Kante erstellt, die vom Knoten 1 zum Knoten 2 führt.
Wie bereits zuvor erwähnt, wird in den Zeilen 17 und 18 zunächst die Java Swing Komponente für die Visualisierung des Graphen erzeugt, bevor sie im Anschluss daran dem Content-Pane Container des JFrames hinzugefügt wird.

5.2. Grafische Benutzeroberfläche

Dieses Unterkapitel beschäftigt sich mit der grafischen Benutzeroberfläche (GUI) der entwickelten Anwendung. Der in Kapitel 4 besprochene Entwurf der grafischen Oberfläche diente als Grundlage für die Umsetzung der GUI. Der nachfolgende Abschnitt 5.2.1 beschreibt zunächst die Gesamtansicht der Anwendung, wobei die einzelnen Komponenten des Programmfensters erläutert werden. Im Anschluss daran widmet sich der darauffolgende Abschnitt 5.2.2 der Bedienung der Anwendung durch den Nutzer. Dabei werden die verschiedenen Funktionen und Dialoge des Programms beschrieben. Abschließend werden im Abschnitt 5.2.3 verschiedene Aspekte der Implementierung mit Bezug zur JGraphX Library besprochen.

5.2.1. Ansicht des Programmfensters

In Abbildung 5.2 ist die grafische Benutzeroberfläche der zu entwickelnden Anwendung dargestellt. Die vier Komponenten, aus denen die Oberfläche besteht, sind dabei farblich hervorgehoben. Die Komponenten an sich, sowie deren Anordnung, wurde dem Entwurf entsprechend umgesetzt. Die Ansicht des Programmfensters stimmt somit fast vollständig mit dem GUI-Prototyp überein. Kleinere Anpassungen wurden an dem seitlichen Auswahlbereich (Komponente Nr. 2) durchgeführt. Neu hinzugekommen ist dabei die Anzeige des Prüfergebnisses und eine Checkbox. Mittels letzter kann eine erweiterte Ansicht zur gleichzeitigen Betrachtung aller Markierungszustände aktiviert werden. Außerdem beschränkt sich die untere Komponente Nr. 3 auf die Anzeige einer Konsole. Jedoch wurde darauf geachtet, dass das beim Entwurf vorgesehene Konzept weiterhin umsetzbar bleibt. Falls notwendig, können somit weitere Komponenten hinzugefügt werden, wobei der Zugriff über entsprechende Reiter (Tabs) geschieht.

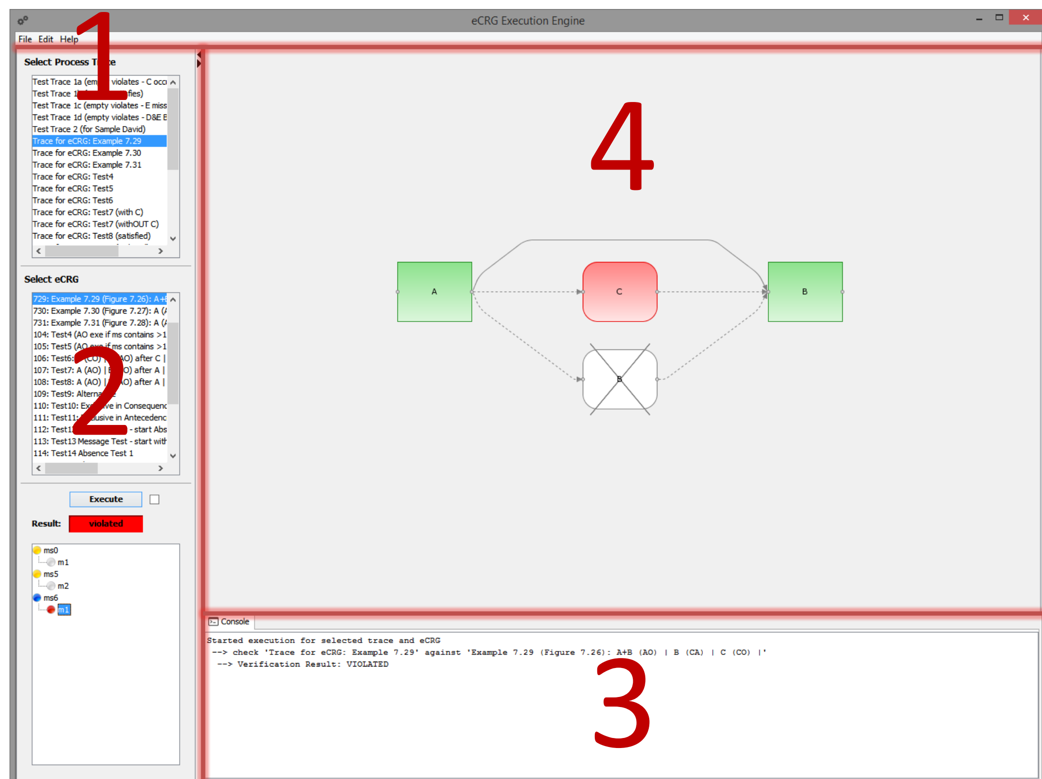


Abbildung 5.2.: Die Komponenten der grafischen Benutzeroberfläche

5.2.2. Funktionen und Dialoge

Diese Teilabschnitt beschreibt die vom Programm zur Verfügung gestellten Funktionen und Dialoge der grafischen Benutzeroberfläche. Die Implementierung der GUI stimmt größtenteils mit dem ursprünglichen Entwurf überein, sodass alle geforderten Grundfunktion umgesetzt wurden. An einigen Stellen wurden dabei Detailverbesserungen vorgenommen. So wurde zum Beispiel ein zusätzliches Feld zur Anzeige des Compliance Ergebnisses implementiert. Außerdem sind die verschiedenen Markierungszustände farblich hervorgehoben. Die genauen Details hierzu, werden nun im Folgenden erläutert.

Ausführung eines Trace über eine eCRG

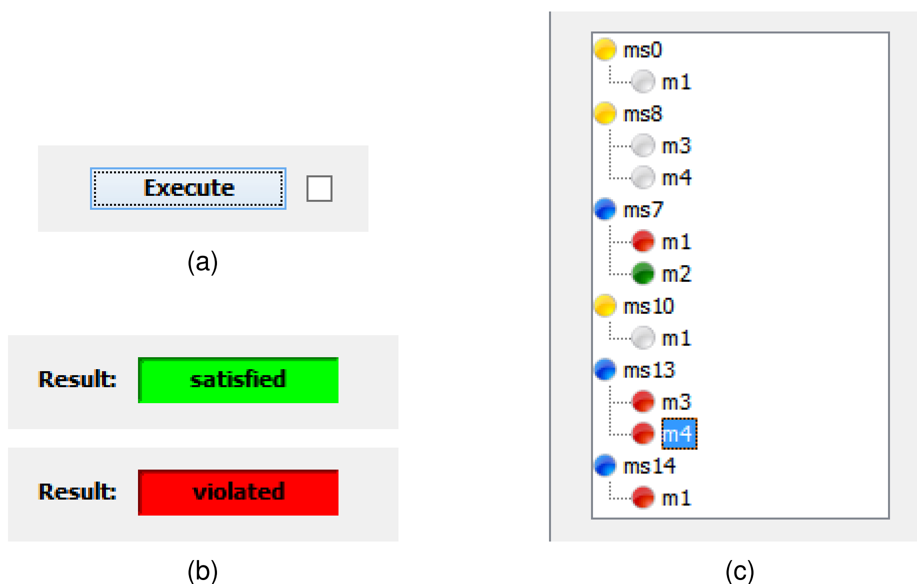


Abbildung 5.3.: Ergebnis der Ausführung eines Trace über eine eCRG

Der Gesamtablauf zur Durchführung einer Compliance Überprüfung wird im Anhang, in den Abbildungen A.1-A.4 veranschaulicht. Dabei sind die einzelnen Schritte dargestellt, die notwendig sind, um die Ausführung der zu überwachenden eCRG zu starten. Zunächst muss ein Trace (Prozess-Ablaufprotokoll) und eine eCRG ausgewählt werden. Der in den jeweiligen Listen selektierte Eintrag, entspricht dabei dem Trace und der eCRG die gegeneinander geprüft werden sollen. Ist dies geschehen, wird durch

Betätigung des $Execute$ Buttons die Ausführung des Traces gestartet. Anschließend werden in der unteren Liste die, während der Ausführung erreichten Markierungen aufgelistet. Des Weiteren wird das Gesamtresultat angezeigt, das sich aus der Auswertung der Markierungen ergibt. In Abbildung 5.3 sind die grafischen Komponenten für das zuvor beschriebene abgebildet: der $Execute$ Button, die Ergebnisanzeige und die Liste der Markierungen. In dieser Liste sind alle Markierungen, aufgegliedert nach MARKSTRUCTURES und EXMARKS, eingetragen, die bei der Ausführung des Traces zustande gekommen sind. Auf der obersten Ebene stehen dabei die MARKSTRUCTURES. Diese wiederum können aufgeklappt werden, wodurch die enthaltenen EXMARK Markierungen zugänglich werden. Die farbliche Codierung der Listeneinträge beschreibt den Zustand

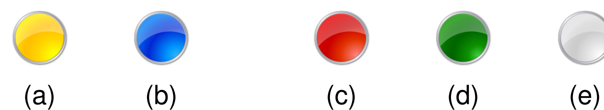


Abbildung 5.4.: Symbole zur Veranschaulichung der Markierungszustände

(a)	(b)	(c)	(d)	(e)
deaktiviert	aktiviert	verletzt	erfüllt	trivial erfüllt

Tabelle 5.1.: Bedeutung der verschiedenen Symbole für die Markierungszustände

der jeweiligen Markierung. In Abbildung 5.4 sind die verschiedenen Symbole abgebildet, die zur Veranschaulichung der Markierungszustände verwendet werden. Die Tabelle 5.1 liefert die dazugehörige Bedeutung für das jeweilige Symbol. Bei MARKSTRUCTURES wird dabei zwischen *aktiviert* und *deaktiviert* unterschieden. Die EXMARK Markierungen sind hingegen entweder als *verletzt*, *erfüllt* oder *trivial erfüllt* gekennzeichnet. Die zuletzt genannten Markierungszustände beziehen sich auf das Consequence-Pattern der eCRG, wohingegen der Status der übergeordneten MARKSTRUCTURE die Markierung des Antecedence-Patterns beschreibt. Der Status *trivial erfüllt* kann nur vorkommen, wenn die Markierung aufgrund eines verletzten Antecedence-Patterns gar nicht erst aktiviert wurde.

Durch diese zusätzliche Information erhält der Nutzer Einblick in die jeweiligen Markierungen und kann sich so schneller einen Überblick verschaffen. Außerdem wird er

dadurch bei der Identifikation möglicher Compliance-Verletzungen unterstützt. In der Abbildung 5.3c sind zum Beispiel die aktivierten MARKSTRUCTURES ms_{13} und ms_{14} dafür verantwortlich, dass das Prüfergebnis *violated* lautet. Denn beide enthalten keine erfüllende Markierung des Consequence-Patterns.

Anzeige der Markierungen

Zur genaueren Untersuchung des Compliance Zustandes lassen sich die erzielten Markierungen grafisch veranschaulichen. Dies geschieht durch Anklicken eines EXMARK Eintrags in der Markierungsliste. Infolgedessen wird die dazugehörige Markierung auf der Anzeigefläche dargestellt. Dazu wird der eCRG Graph gezeichnet und mit entsprechenden Markierungen versehen, die den Ausführungszustand der einzelnen eCRG Elemente beschreiben.

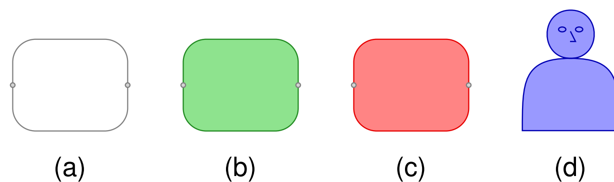


Abbildung 5.5.: Markierungen der eCRG Knoten

(a)	(b)	(c)	(d)
unmarkiert	zulässig	verletzt	belegt

Tabelle 5.2.: Bedeutung der verschiedenen Knotenmarkierungen

Die möglichen Markierungen, die für die Kennzeichnung der eCRG Knoten verwendet werden können, sind in der Abbildung 5.5 dargestellt. Die jeweiligen Bedeutungen sind in der Tabelle 5.2 angegeben. Durch *unmarkiert* wird ein neutraler Zustand repräsentiert, der eCRG Elementen zugewiesen wird, die nicht ausgeführt wurden und weder verletzt noch erfüllt sind. Als *zulässig* wird beispielsweise eine Occurrence Node gekennzeichnet, für die eine passende Ausführung registriert wurde. Auch Bedingungen, die zu wahr evaluieren, erhalten diese Markierung. Knoten und Bedingungen die als *verletzt* gekennzeichnet wurden, signalisieren eine Verletzung des Antecedence oder

des Consequence-Patterns, abhängig vom jeweiligen Patterntyp. Die Markierung *belegt* kann nur eine Data oder Resource Node besitzen. Dies zeigt an, dass eine passende Belegung für den Knoten gefunden wurde, welche die Forderungen erfüllt.

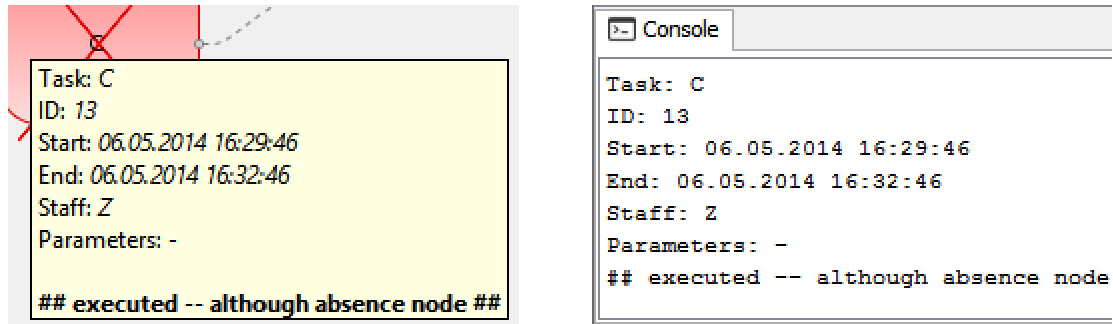


Abbildung 5.6.: Anzeige der Ausführungsmarkierung per Tooltip und Konsole

Um weitere Details zu den Ausführungsmarkierungen zu erhalten, kann entweder mit der Maus über die Markierung gefahren werden, wodurch ein Tooltip mit detaillierteren Informationen zur Markierung erscheint, oder man klickt das dazugehörige Element an, damit die Markierungsdetails auf der Konsole ausgegeben werden. In Abbildung 5.6 ist sowohl die Anzeige des Tooltips, als auch die die Ausgabe auf der Konsole zu sehen. Die Informationen beschränken sich dabei nicht nur auf die Details zur Ausführung, sondern es werden im Falle von Verletzungen auch die Gründe dafür mit angegeben.

Gesamtansicht aller Markierungen

Mit den bisherigen Möglichkeiten können die Markierungen nur einzeln und der Reihe nach begutachtet werden. Statt nun aber jede Markierung separat zu untersuchen, besteht auch die Möglichkeit sich eine Gesamtansicht anzeigen zu lassen. In dieser erweiterten Ansicht werden alle Markierungen auf einmal in textueller Form dargestellt.

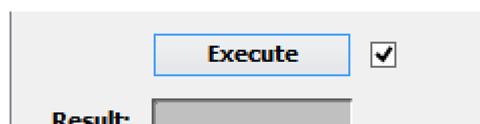


Abbildung 5.7.: Checkbox für Anzeige der Gesamtansicht

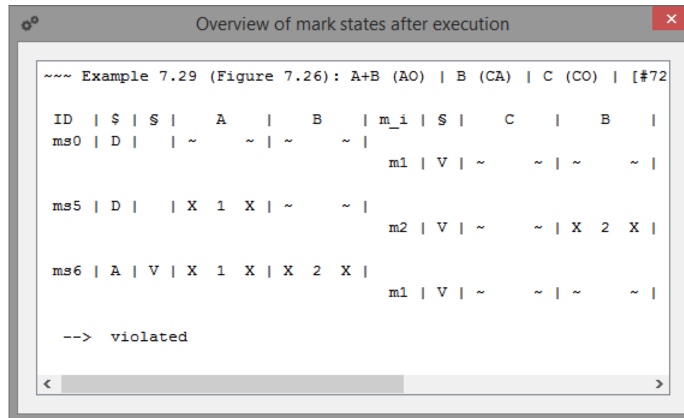
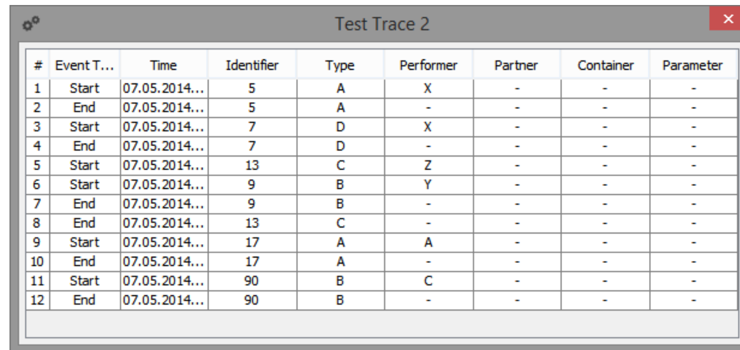


Abbildung 5.8.: Dialog mit einer Gesamtansicht der erreichten Markierungszustände

Zur Aktivierung dieser Ansicht dient die Checkbox neben dem $Execute$ Button, siehe Abbildung 5.7. Ist diese Checkbox aktiviert, öffnet sich im Anschluss an die Ausführung ein Dialog, der eine verkürzte Darstellung der erreichten Markierungszustände enthält. Abbildung 5.8 zeigt einen solchen Dialog, der eine Übersicht über alle Markierungen erlaubt. Das dort dargestellte Beispiel besteht aus drei MARKSTRUCTURES, die wiederum je eine EXMARK Markierung enthalten. Dabei repräsentiert nur die letzte Markierung eine Regelaktivierung, die beiden anderen bleiben, aufgrund fehlender Taskausführungen, deaktiviert. Da jedoch von allen Markierungen das Consequence-Pattern verletzt wird, folgt daraus, dass die durch die eCRG formulierte Compliance Regel von der überwachten Prozessinstanz nicht erfüllt wird.

Trace Anzeige

Zur Ansicht der verschiedenen Prozess-Ablaufprotokolle (*Process Traces*), kann ebenfalls ein entsprechender Dialog geöffnet werden, in dem der jeweilige Trace im Detail dargestellt wird. Dazu genügt ein Doppelklick auf einen der Einträge in der oberen Liste, welche die Traces enthält. Infolgedessen erscheint ein Dialog, der eine Auflistung der im Trace enthaltenen Events zeigt. Zum Schließen des Dialogs, stehen neben der Standard-Schaltfläche hierfür auch die Tasten *Enter* und *Esc* zur Verfügung. Abbildung 5.9 zeigt beispielhaft einen solchen Dialog für einen Trace, der aus zwölf Events besteht und dabei die Details zu den einzelnen Events enthält.



Test Trace 2

#	Event T...	Time	Identifier	Type	Performer	Partner	Container	Parameter
1	Start	07.05.2014...	5	A	X	-	-	-
2	End	07.05.2014...	5	A	-	-	-	-
3	Start	07.05.2014...	7	D	X	-	-	-
4	End	07.05.2014...	7	D	-	-	-	-
5	Start	07.05.2014...	13	C	Z	-	-	-
6	Start	07.05.2014...	9	B	Y	-	-	-
7	End	07.05.2014...	9	B	-	-	-	-
8	End	07.05.2014...	13	C	-	-	-	-
9	Start	07.05.2014...	17	A	A	-	-	-
10	End	07.05.2014...	17	A	-	-	-	-
11	Start	07.05.2014...	90	B	C	-	-	-
12	End	07.05.2014...	90	B	-	-	-	-

Abbildung 5.9.: Dialog für die Anzeige eines Trace

eCRG Anzeige

Auf die gleiche Weise, wie die Traces, können auch die eCRG Regeln betrachtet werden. Durch Doppelklick auf einen der Einträge in der eCRG Liste, wird die Compliance Regel, repräsentiert durch den jeweiligen eCRG Graphen, auf der Anzeigefläche grafisch veranschaulicht. Abbildung 5.10 zeigt beispielhaft die Ausgabe einer solchen eCRG auf der Anzeigefläche. Die einzelnen Elemente der eCRG sind dabei im unmarkierten Zustand dargestellt, da es sich um die reine Anzeige der eCRG handelt.

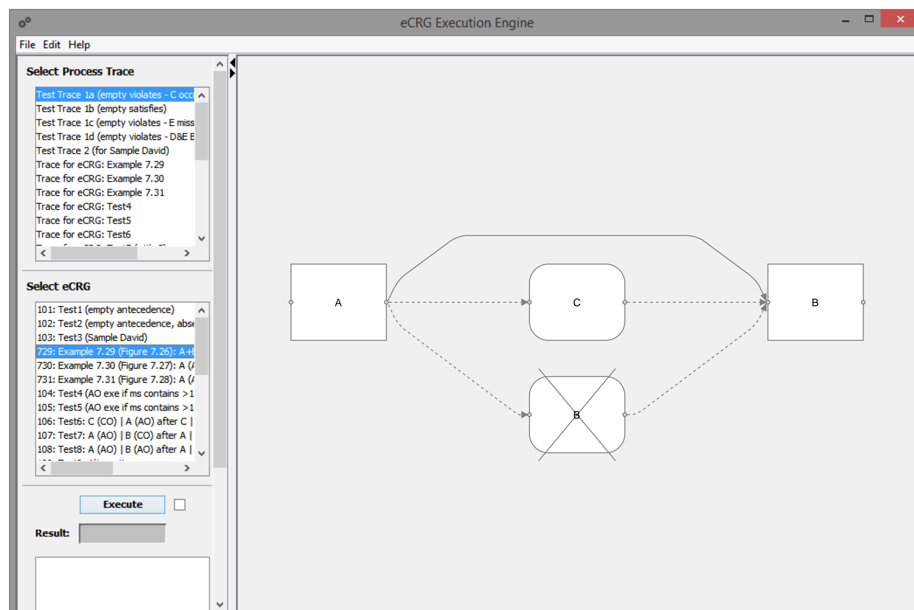
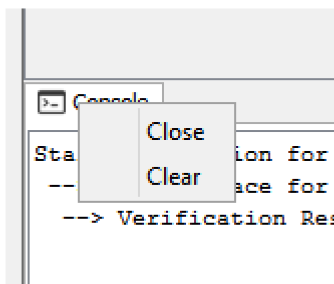


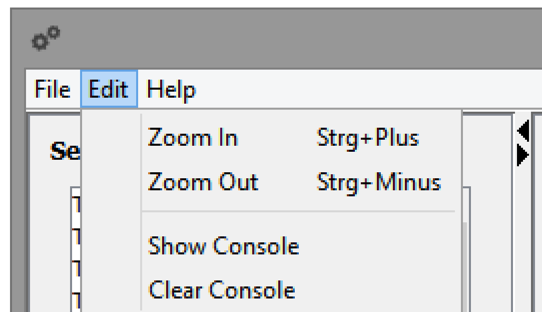
Abbildung 5.10.: Ausgabe einer eCRG auf der Anzeigefläche

Weitere Funktionen

Abschließend sind in diesem Teilabschnitt weitere Funktionen der grafischen Benutzeroberfläche zusammengefasst. In Abbildung 5.11a ist zum Beispiel das Kontextmenü dargestellt, welches erscheint, wenn mit der rechten Maustaste auf den Tab (Reiter) für die Konsole geklickt wird. Die zwei dabei angebotenen Funktionen führen zum einen dazu, dass die Konsole geschlossen wird und nicht länger angezeigt wird, und zum anderen zur Löschung des Konsoleninhalts. Diese Funktionen sind zusätzlich auch über die Menüleiste zu erreichen, was vor allem dann wichtig wird, wenn die Konsole wieder angezeigt werden soll. Zur Vereinfachung der Steuerung, wurden zusätzlich entsprechende Tastenkürzel für die verschiedenen Funktionen der Menüleiste eingerichtet. Beispielsweise wird durch die Tastenkombination $Alt{+}E{+}S$ die Anzeige der Konsole gesteuert. Weitere Funktionen die in der Menüleiste untergebracht sind, sind beispielsweise eine Funktion zur Speicherung der auf der Anzeigefläche dargestellten eCRG (samt Markierung) sowie eine *Zoom* Funktion zur Anpassung der Darstellungsgröße des Graphen auf der Anzeigefläche, siehe Abbildung 5.11b.



(a)



(b)

Abbildung 5.11.: Kontextmenü der Konsole und Funktionen der Menüleiste

5.2.3. JGraphX betreffende Implementierungen

In diesem Unterkapitel werden verschiedene Aspekte der Implementierung vorgestellt, die Bezug zur JGraphX Bibliothek aufweisen. Die Codebeispiele, die dabei verwendet werden, sollen einen Einblick in die Programmierung mit JGraphX geben und stehen repräsentativ für ähnliche weitere Implementierungen im Rahmen der Realisierung des Prototyps.

Übersicht über die eCRG Symbole

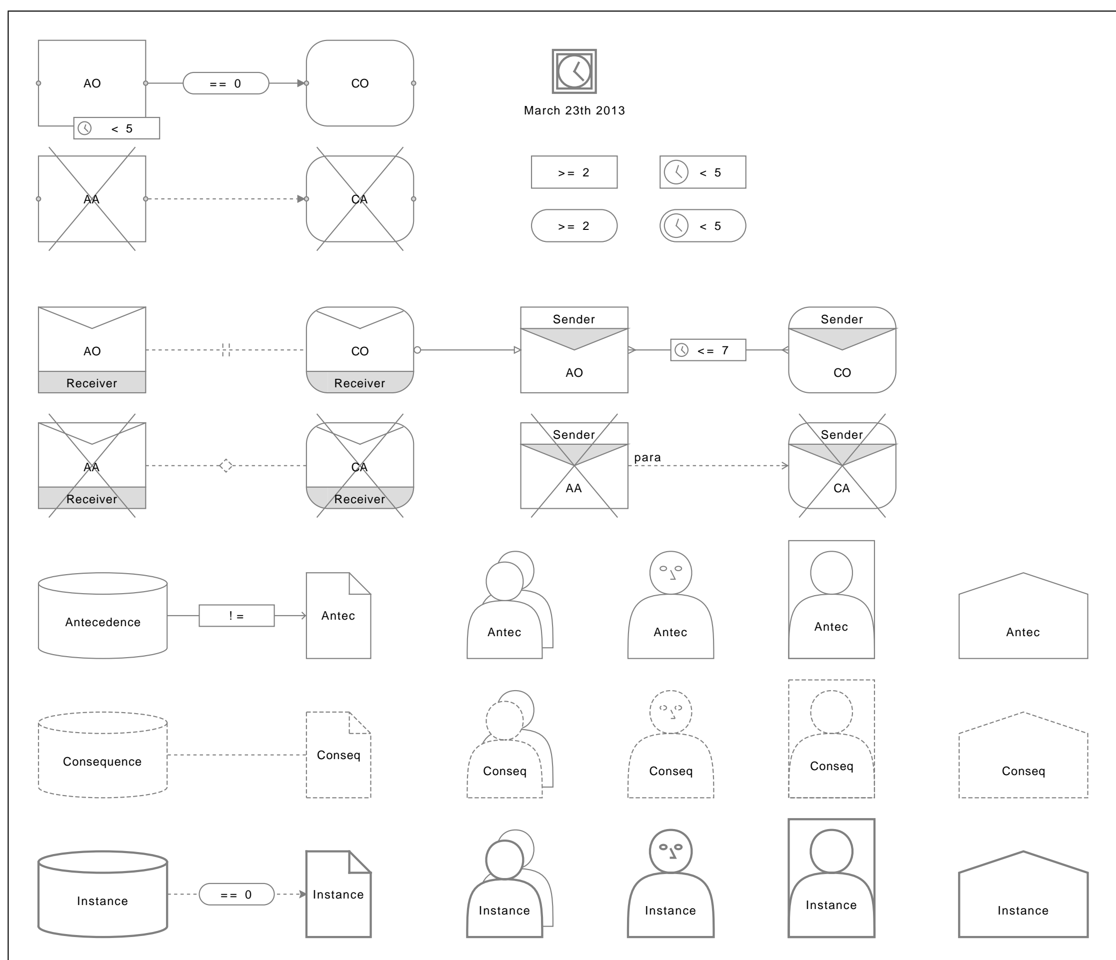


Abbildung 5.12.: Die JGraphX Shapes für die Darstellung der verschiedenen eCRG Symbole entsprechend ihrer Notation

Die Abbildung 5.12 zeigt eine Übersicht über alle implementierten Shapes (Symbole, Templates) für die Darstellung der verschiedenen eCRG Elemente. Die darin abgebildeten Symbole werden für die grafische Veranschaulichung von eCRG Regeln benötigt. Denn die Darstellung der Elemente soll der eCRG Notation entsprechen, sodass die verschiedenen Knoten und Kanten individuell gekennzeichnet sind und somit zwischen den jeweiligen Typen unterschieden werden kann. Die Shapes wurden aufgrund der in [KRL+13b] definierten Notation realisiert, wobei auf eine detailgetreue Abbildung der einzelnen eCRG Elemente geachtet wurde.

In der Übersicht in Abbildung 5.12 sind alle eCRG Elemente vertreten, die von dem Prototyp unterstützt werden, der zusammen mit dieser Arbeit realisiert wurde. Nicht unterstützt werden die frei-belegbaren PointInTime Nodes und die Task und Message Nodes für die Interaction View. Folglich sind dies auch die einzigen Symbole, die in der Übersicht fehlen. Außer den verschiedenen Knotentypen für die fünf unterstützten Perspektiven, sind darin auch die unterschiedlichen eCRG Kantentypen abgebildet. Aus Gründen der Darstellung sind die Knoten an den Enden der Kanten dabei jedoch teilweise frei gewählt, wodurch nicht alle syntaktischen Restriktionen der eCRG Notation eingehalten werden. Ähnliches gilt zum Teil auch für die abgebildeten Attachments, deren Aussage und Verwendung nicht immer gültig ist.

Beispiel einer Shape Definition

Im Anhang ist im Quelltext B.1 der Originalcode der `myDataShape` Klasse abgebildet. Diese Klasse ist für die Darstellung der Data Nodes verantwortlich und zeichnet die durch die Notation festgelegte Form als Grafik.

Die Bezeichner der mit JGraphX im Zusammenhang stehenden Klassen leiten sich aus folgender Namenskonvention ab. Alle aus der JGraphX Library stammenden Klassen beginnen mit den Buchstaben `mx`. Selbst erstellte Klassen, die sich auf eine JGraphX Klasse beziehen und von dieser erben sind durch das Präfix `my` gekennzeichnet.

Die Klasse `myDataShape` erbt von der Klasse `mxBasicShape` und besitzt somit von Grund aus alle Basisfunktionen. Nur die Methode $paintShape$ muss angepasst werden

und wird deswegen überschrieben. Die beiden Parameter der Methode entsprechen zum einen der Zeichenfläche, auf der die Grafik für die Data Nodes gerendert werden soll, und zum anderen dem aktuellen Zellzustand, der unter anderem die Postion und die Styledefinition für die Darstellung enthält.

Zunächst wird in der $paintShape$ Methode die Zelle abgefragt, die im Folgenden visualisiert werden soll. Als nächstes wird der Wert des *User-Objects* gelesen, dass jede `mxCell` in in ihrer $value$ Variable abspeichern kann. Nur wenn es sich dabei um ein Objekt vom Typ `DataNode` handelt, wird das Zeichnen fortgesetzt. In den folgenden Zeilen wird das Symbol für den Datenknoten gezeichnet. Dazu werden zunächst die Koordinaten der Position ermittelt, an welche die Grafik hinkommen soll. Anschließend wird der Hintergrund der Grafik gezeichnet, bevor im Anschluss daran der Vordergrund beziehungsweise die Kontur des jeweiligen Symbols gezeichnet wird. In beiden Schritten wird dabei der Typ des Datenknotens abgefragt, um abhängig davon das korrekte Symbol für den jeweiligen Typ zu zeichnen, siehe dazu Abbildung 5.12. Beim Zeichnen des Vordergrunds kommt zusätzlich noch das Setzen des korrekten Styles für die Kontur hinzu. Abhängig vom Patterntyp wird diese entweder *fett*, *gestrichelt* oder *durchgängig* gezeichnet. Für letzteres bedarf es keiner Anpassung, im Falle der beiden anderen wird der $Stroke$ entsprechend gesetzt.

Nachdem das Symbol gerendert wurde, muss nun noch der jeweilige Knotenbezeichner, auch Label genannt, der Grafik hinzugefügt werden. Dies wurde in die Methode $drawLabel$ ausgelagert. Zu Beginn wird die korrekte Schriftfarbe und die, in der Styledefinition hinterlegte, Schriftart gesetzt. Anschließend wird der zu zeichnende String-Wert ermittelt und die Position festgelegt, an welche das Label hinkommen soll. Zum Schluss, bevor der String schließlich der Grafik hinzugefügt wird, wird aufgrund der Stringlänge der Offset in x-Richtung bestimmt, um den Bezeichner zentriert darzustellen.

Beispiel für die Definition einer Pfeilspitze

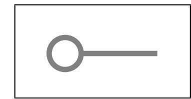
Da JGraphX nicht alle Pfeilspitzen nativ unterstützt, die für die Visualisierung der unterschiedlichen eCRG Connectoren benötigt werden, mussten einige der Pfeilmarkierungen selbst implementiert werden.

```
/**
 * Registration of the self defined markers
 */
public static void register() {
   mxMarkerRegistry.registerMarker("my_oval", new mxIMarker()
   {
      public mxPoint paintMarker(mxGraphics2DCanvas canvas,
            mxCellState state, String type, mxPoint pe, double nx,
            double ny, double size, boolean source)
      {
         double r = size / 2;
         double cx = pe.getX() - nx / 2;
         double cy = pe.getY() - ny / 2;
         Shape shape = new Ellipse2D.Double(cx - r, cy - r, size, size);
         if (mxUtils.isTrue(state.getStyle(), (source) ? "startFill" :
             "endFill", true))
         {
            canvas.fillShape(shape);
         }
         canvas.getGraphics().draw(shape);
         return new mxPoint(-nx, -ny);
      }
   });

   // ... further markers
}
```

Quelltext 5.2: Ausschnitt aus dem Quellcode für die Registrierung und Deklaration der selbst-definierten Pfeilspitzen

Unter anderem musste die ovale Anfangsmarkierung der Message Flow Connectoren neu programmiert werden. Der Quelltext 5.2 enthält den Programmcode hierfür, welcher im Folgenden erläutert wird. Zunächst wird in Zeile 11 die Variable r berechnet, in welcher der Radius für die Markierung gespeichert wird. Die Größe des Radius bestimmt sich aus dem übergebenen Wert des Parameters $size$, der die Größe der Markierung beinhaltet. Anschließend wird ausgehend vom Punkt pe (Start / Ende der dazugehörigen Kante), der Mittelpunkt für die zu zeichnende ovale Markierung ermittelt (Zeile 12+13). Dieser



Punkt dient nun als Grundlage für die Berechnung des linken oberen Ursprungspunktes, an dem die Ellipse in Form eines Kreises gezeichnet werden soll (Zeile 14). Über die Zeilen 15-18 wird geregelt, ob die Markierung *gefüllt* oder *ungefüllt* gezeichnet werden soll. Im Anschluss daran wird schließlich die Kontur der Markierung gezeichnet (Zeile 19). Der Rückgabewert entspricht dem Punkt, an dem die dazugehörige Kante enden soll. In diesem Fall liegt der Punkt auf dem kreisförmigen Rand der Markierung. Die sich aus dem Quelltext 5.2 ergebende Pfeilmarkierung ist in der Abbildung rechts dargestellt.

Implementierung der Ports

Das Konzept der *Ports* wurde für die Darstellung der Task Nodes herangezogen, um entsprechende Ankerpunkte zur Repräsentation des Start- und End-Zeitpunkts zu erhalten. Die Ports selbst werden auch durch Knoten realisiert, jedoch sind sie anders als die bis jetzt vorgekommenen Knoten nicht auf der obersten Hierarchieebene angesiedelt, sondern werden als Kindknoten (child nodes) dem jeweiligen Task Knoten angefügt. Dies ist insbesondere für die Connectoren wichtig, die eine zeitliche Bedingung zum Ausdruck bringen. Denn in diesem Fall muss bei der Formulierung der Bedingung zwischen dem Start- und dem End-Zeitpunkt des Tasks unterschieden werden. Um dies auch grafisch zu veranschaulichen, sind alle Connectoren, die die Zeit-Perspektive betreffen[3], immer mit dem Start oder dem End Port der Task Node verbunden. Der Start Port befindet sich am linken Rand und der End Port am gegenüberliegenden rechten Rand. Kanten wie zum Beispiel der Perform Connector sind hingegen über einen Defaultport mit dem Knoten verbunden. Die Position dieses Ports ist nicht durch einen bestimmten Punkt festgelegt und er wird auch nicht gesondert gekennzeichnet.

Die Klasse `PortFactory`, die im Quelltext B.2 im Anhang abgebildet ist, dient als sogenannte *Factory Class* und vereinfacht die Erstellung neuer Port Knoten. Damit ist die Erstellung der Ports an einer zentralen Stelle geregelt und es genügt ein einfacher Methodenaufruf zur Erzeugung eines neuen Portknotens. Die dafür zu rufende Methode

[3] Sequence, TimeDistance und DataCondition Kanten

lautet $createPort$ und verlangt die Angabe eines Parameters, über den der gewünschte Port spezifiziert wird. Es stehen vier verschiedene Ports für die vier Himmelsrichtungen zur Verfügung. Der einzige Unterschied dabei ist die Position, an der sie gezeichnet werden. Die Festlegung der Positionskoordinaten erfolgt dabei relativ in Abhängigkeit zum jeweiligen Parentknoten. Zulässig sind hierbei alle Werte, die zwischen 0.0 und 1.0 liegen. Die Postion $(0.5, 0.5)$ würde den Punkt genau in der Mitte festlegen. Der Wert 0.0 entspricht dem linken / oberen Rand und 1.0 dem rechten / unteren Rand. Durch die Anpassung des Offsets wird der Punkt schließlich noch in die linke obere Ecke verschoben, die später den Ursprung beim Zeichnen des Ports darstellt. Die `mxCell`, die von der $createPort$ Methode zurückgegeben wird, muss nun nur noch unter Angabe des korrekten Parentknotens dem Graph hinzugefügt werden, siehe dazu die folgenden Programmzeilen.

```
1 // mxGraph graph = ... ; mxCell parent = ... ;
2 mxCell portWest = PortFactory.createPort(PortDirection.West);
3 graph.addCell(portWest, parent);
```

Ausrichtung und Layout des Graphen

Die JGraphX Bibliothek beinhaltet bereits einige Layoutalgorithmen zur Positionierung der im jeweiligen Graph enthaltenen Elemente. Für die Anordnung und Ausrichtung des eCRG Graphen wurde ein hierarchisches Layout als am besten geeignet erachtet, wobei die im Graph enthaltenen Knoten von links nach rechts angeordnet werden. Der Layoutalgorithmus, der in der Klasse `mxHierarchicalLayout` implementiert wurde, unterstützt bereits das gewünschte Layout. Bei der Verwendung dieses Algorithmus gab es zunächst jedoch Probleme, die zu einem nicht erwarteten Layoutergebnis führten. So wurde beispielsweise die Größe mancher Knoten (unerwünschterweise) angepasst und die Port Knoten wurden in den Layoutvorgang miteinbezogen, sodass sich deren Position änderte. Diese Probleme konnten schließlich durch Anpassung der Layoutparameter gelöst werden.

Die im Quelltext 5.3 abgebildete Klasse `myHierarchicalLayout` erbt von der JGraphX Klasse `mxHierarchicalLayout` und enthält die passenden Einstellungen, für die

Durchführung eines korrekten Layouts. In Zeile 13 wird dabei die Richtung für den Layout Algorithmus festgelegt. Anschließend wird in den Zeilen 15 und 16 der Abstand definiert, der bei der Anordnung zwischen den einzelnen Knoten eingehalten werden soll. Die Einstellungen in den Zeilen 18 und 19 sind besonders wichtig, denn dadurch konnten die zuvor beschriebenen Probleme gelöst werden. Erstere verhindert die Neupositionierung der Portknoten und die zweite deaktiviert die Anpassung der Größe bei Parentknoten. Durch die letzte Einstellung in Zeile 20 wird das Zurücksetzen der $EdgeSytles$ verhindert. Die Erstellung einer eigenen Klasse war unter anderem auch deswegen notwendig, da die `mxHierarchicalLayout` Klasse keine Methode zur Anpassung der $traverseAncestors$ Eigenschaft anbietet und so direkt auf die Variable zugegriffen werden musste, was wiederum nur in einer Spezialisierung der Klasse möglich ist.

```
1  /**
2   * Customized Class for performing a hierarchical layout.
3   * Inherits from JGraphX class mxHierarchicalLayout.
4   */
5  public class myHierarchicalLayout extends mxHierarchicalLayout
6  {
7     /**
8      * Constructs a hierarchical layout
9      * @param graph the graph to lay out
10     */
11    public myHierarchicalLayout(mxGraph graph)
12    {
13       super(graph, SwingConstants.WEST);
14
15       setInterHierarchySpacing(100);
16       setInterRankCellSpacing(150);
17
18       traverseAncestors = false;
19       resizeParent = false;
20       disableEdgeStyle = false;
21    }
22 }
```

Quelltext 5.3: Die Klasse `myHierarchicalLayout`

```
public static void layoutGraph(mxGraph graph) {
   myHierarchicalLayout layout = new myHierarchicalLayout(graph);

   graph.getModel().beginUpdate();
   try {
      layout.execute(graph.getDefaultParent());
   }
   finally  {
      graph.getModel().endUpdate();
   }
}
```

Quelltext 5.4: Die Methode zum Layouten des Graphen

Um nun das durch die Klasse `myHierarchicalLayout` definierte Layout anzuwenden, muss die $layoutGraph$ Methode gerufen werden. Dadurch wird das Layout auf den übergebenen Graphen angewandt und die enthaltenen Elemente werden entsprechend angeordnet. Im Quelltext 5.4 ist der Programmcode der Methode abgebildet. Zunächst wird dabei eine neue Instanz der Klasse `myHierarchicalLayout` angelegt. An diesem Objekt wird schließlich die $execute$ Methode aufgerufen, wodurch der Layoutprozess startet. Da der gesamte Graph neu gelayoutet werden soll, wird der Methode der $DefaultParent$ übergeben, sodass der Algorithmus beim Wurzelknoten beginnt. Damit die Darstellung des Graphen auf der Anzeigefläche nur einmal aktualisiert werden muss, wird der Layoutvorgang außerdem zu einer Transaktion zusammengefasst.

5.3. Monitoring Algorithmen

Zum Abschluss von Kapitel 5, wird in diesem Teilabschnitt die Implementierung des *Monitoring* besprochen. Dazu werden im Folgenden einige wichtige und grundlegende Algorithmen beschrieben, welche für die Compliance Überwachung zuständig sind. Dies dient zum einen der Veranschaulichung des Gesamtablaufs des Monitorings und zum anderen soll dadurch Einblick in die Programmierung gegeben werden. Aufgrund des Umfangs der Implementierung, wird im weiteren Verlauf dieses Abschnitts nur auf die

wesentlichen Funktionen (Methoden) eingegangen, um die grundsätzliche Idee zu vermitteln. Im Nachfolgenden wird deswegen auch nur ein kleiner Auszug der Implementierung gezeigt, wobei sich viele Bestandteile vom Ablauf her auch ähneln.

Die folgenden Programmzeilen sind aus dem Quellcode des Programms entnommen und zeigen beispielhaft wie anhand der `Verifier` Klasse ein neuer Monitoring-Vorgang gestartet wird.

```
1  /** Code for starting a new execution and verification */
2  ProcessTrace selectedTrace = listOfTraces.getSelectedValue();
3  eCRG selectedEcrg = listOfEcrgs.getSelectedValue();
4  ComplianceState cs = Verifier.verify(selectedEcrg, selectedTrace);
5  if(cs.isSatisfied()) {
6     resultLabel.setBackground(Color.green);
7     resultLabel.setText("satisfied");
8  } else {
9     resultLabel.setBackground(Color.red);
10    resultLabel.setText("violated");
11 }
```

In den Zeilen 2 und 3 wird zunächst der `ProcessTrace` und die `eCRG` bestimmt, die gegeneinander geprüft werden sollen. Zum Start der Ausführung samt anschließender Auswertung genügt der Aufruf der Methode $verify$, wobei die zuvor ermittelte eCRG und der Trace als Parameter übergeben werden. Anschließend verweist die Variable cs vom Typ `ComplianceState` auf das Ergebnis der Überprüfung samt der dabei erzielten Markierungen. Die Zeilen 5 bis 11 dienen schließlich noch der Anzeige des Resultats auf der Benutzeroberfläche.

```
1  public class Verifier {
2     public static ComplianceState verify(eCRG ecrg, ProcessTrace trace) {
3        Verifier v = new Verifier(ecrg, trace);
4        return v.execute();
5     }
6     // ...
7  }
```

Quelltext 5.5: Die einzige öffentliche Methode der Klasse `Verifier`

Die Klasse `Verifier` ermöglicht die Ausführung eines Trace über eine eCRG und ist außerdem für die Ermittlung des Compliance Zustands verantwortlich. Die einzige nach außen hin sichtbare Methode dieser Klasse ist die $verify$ Methode, welche im Quelltext 5.5 abgebildet ist. Durch sie wird die Verwendung der `Verifier` Klasse etwas vereinfacht. Die im Anhang enthaltenen Methoden fürs Monitoring sind alle Bestandteil dieser Klasse. Im Folgenden werden nun die Methoden aus den Quelltexten B.3 - B.12 näher beschrieben.

5.3.1. Übersicht über den Gesamtalgorithmus

Algorithmus 1 Überprüfung eines Trace für eine eCRG

```
 1: function EXECUTE(Trace, eCRG)
 2:     do Initialisierung der benötigten Datenstrukturen              ▷ Init

 3:     for all Events in Trace do                                ▷ Iteration
 4:         EXECUTE(Event)                                  ▷ Events ausführen
 5:     end for
                                                                    ▷ Cleanup
 6:     CLEANUPMARKS( )              ▷ Erreichte Markierungen bereinigen

 7:     for all MarkStructures ms do                           ▷ Finalization
 8:         FINALIZE(ms)                         ▷ Markierungen abschließen
 9:     end for
                                                                     ▷ Checks
10:     DOCHECKS( )                          ▷ Überprüfe alle eCRG Elemente

11:     for all MarkStructures ms do                             ▷ Evaluation
12:         for all ExMark m in ms do          ▷ Auswertung der Markierungen
13:             EVALUATE(m)
14:         end for
15:         EVALUATE(ms)
16:     end for

17:     return ComplianceState
18: end function
```

In Algorithmus 1 sind die einzelnen Schritte dargestellt, welche der implementierte Monitoring Algorithmus durchläuft. Die dem Algorithmus zugrundeliegende Struktur ist dabei an die $verify$ Methode aus [Ly13] angelehnt und lässt sich in die sechs abgebildeten Teilbereiche untergliedern. Zunächst werden die benötigten Datenstrukturen

initialisiert, dazu gehört zum Beispiel die Erstellung die Anfangsmarkierung ms_0. Anschließend werden die im Ablaufprotokoll (Trace) enthaltenen Events ausgeführt und die Markierungen, entsprechend den in Kapitel 3 eingeführten Regeln, angepasst. Nach der Ausführung, repräsentieren die Markierungen die vorgekommenen Events. Im nächsten Schritt müssen nun die erreichten Markierungen überprüft werden und unter Umständen widersprüchliche Markierungen entfernt werden. Dies ist notwendig, da bei der Ausführung von Absence Knoten die Ursprungsmarkierung immer erhalten bleibt, auch wenn bereits eine passende Ausführung stattgefunden hat. Im Anschluss daran werden die Markierungen abgeschlossen und damit für die Auswertung vorbereitet. Vor der abschließenden Bewertung werden alle Forderungen und Bedingungen, die sich aufgrund der eCRG ergeben, überprüft. Werden dabei Verletzungen der Regel festgestellt, wird dies für die nachfolgende Bewertung entsprechend vermerkt. Zum Schluss werden die Markierungen bezüglich ihres Compliance Zustandes ausgewertet, woraus sich schließlich das Ergebnis der Überprüfung ableitet. Die Methode $execute$ aus dem Quelltext B.3 stellt eine Realisierung dieses Algorithmus dar.

5.3.2. Algorithmen für die Ausführung von Events

Im Nachfolgenden werden die Methoden aus den Quelltexten B.4 bis B.7 näher beschrieben. Allen diesen Methoden ist gemeinsam, dass sie für die Ausführung von Task Events benötigt werden.

Ausführung von Start Events

Die im Quelltext B.4 abgebildete $executeStart$ Methode ist für die Ausführung von `StartTaskEvents` verantwortlich und wird von der $execute$ Methode gerufen, wenn ein Start Event bei der Abarbeitung des Trace auftritt. Die Methode ist so angelegt, dass sie für die Aktualisierung einer MARKSTRUCTURE sorgt. Folglich wird die Methode für jede der bis dato vorhandenen MARKSTRUCTURES einmal aufgerufen. Die Abbildung des Start Events geschieht anhand der in Kapitel 3 hierfür aufgestellten Regeln. Folglich wird das Event, der Knotenhierarchie entsprechend, zunächst über die ANTEOCC Knoten

ausgeführt. Erst im Anschluss daran wird versucht, das Event über die vorhandenen AnteAbs, ConsOcc und ConsAbs Knoten auszuführen. Die inneren for-Schleifen ergeben sich aus der nichtdeterministischen Ausführung aller Knotentypen bei eCRG. Denn infolgedessen können bei jeder dieser Ausführungen neue Markierungen hinzukommen, welche im weiteren Verlauf ebenfalls betrachtet werden müssen. Zum Schluss werden die erzielten ExMark Markierungen zu einer Menge von MarkStructures aggregiert und zurückgegeben.

Ausführung von End Events

Die $executeEnd$ Methode aus Quelltext B.5 stellt den Gegenpart zur im Abschnitt zuvor besprochenen $executeStart$ Methode dar. Tritt im Ablaufprotokoll ein End Event auf, wird diese Methode gerufen und sorgt für die entsprechende Anpassung der vorhandenen Markierungen. Der weitere Ablauf stimmt größtenteils mit der Ausführung von Start Events überein. Die in der MarkStructure enthaltenen ExMark Markierungen werden der Reihe nach aktualisiert, wobei wieder die Knoten entsprechend ihrer Hierarchie ausgeführt werden. Der wesentliche Unterschied hierbei ist jedoch die deterministische Ausführung von End Events. Dies hat zur Folge, dass sich die Markierungen nicht vervielfältigen und somit die inneren for-Schleifen überflüssig sind. Außerdem sollte die Aggregation der verschiedenen ExMark Markierungen zu genau einer MarkStructure führen, andernfalls ging bei der Aktualisierung etwas schief. Die ermittelte MarkStructure wird schließlich als Ergebnis zurückgeliefert.

Hilfsmethoden für die Bestimmung passender Task Nodes

In den Quelltexten B.6 und B.7 sind verschiedene Hilfemethoden enthalten, die bei der Ausführung der beiden Task Events gerufen werden, um nicht relevante Konten auszusortieren. Die Methode $matchStart$ überprüft, ob die übergebene Task Node zu dem angegebenen Start Event passt. Dies ist der Fall, wenn der Tasktyp bei beiden derselbe ist. In der übergeordneten Methode, von der aus diese Methode gerufen wurde, werden anhand des zurückgelieferten Ergebnis nicht zum Event passende

Knotenkandidaten herausgefiltert. Die Methode $matchEnd$ prüft auf analoge Weise, ob die übergebene Task Node zu dem angegebenen ENDTASKEVENT passt. Da bei der Ausführung von End Events nicht nur der Typ übereinstimmen muss, sondern auch die ID des vermerkten Start Events dieselbe sein muss, ist noch eine weitere Prüfung notwendig. Dies übernimmt die $expectedEnd$ Methode. Dabei wird zunächst die aktuelle Ausführungsmarkierung des zu überprüfenden Knotens abgefragt und anschließend das vermerkte Start Event ermittelt. Nun kann die ID des End Events mit der des Start Events verglichen werden und abhängig davon wird entweder *true* oder *false* als Ergebnis zurückgeliefert.

Aufbauend auf diesen Hilfsfunktionen wurden nun weitere Methoden definiert, welche für die Bestimmung relevanter Knoten herangezogen werden. Eine dieser Methoden ist im Quelltext B.7 abgebildet. Die $exAnteNodesStart$ Methode ist für die Antecedence Markierung zuständig und ermittelt, abhängig vom Wert des Parameters occ, alle ANTEOCC beziehungsweise ANTEABS Knoten, die zum angegebenen Start Event passen und zusätzlich die Bedingungen für einen Start erfüllen. Dazu werden alle existierenden Task Nodes der Reihe nach durchgegangen und überprüft. Zunächst wird sichergestellt, dass der Knoten den korrekten Patterntyp besitzt. Trifft dies zu, müssen noch drei weitere Bedingungen überprüft werden. Dies wurde in die $executableAnteStart$ Methode ausgelagert. Dort wird als erstes die $matchStart$ Methode aufgerufen, um sicherzustellen, dass der Knotentyp zum Event passt. Als nächstes wird die Ausführungsmarkierung des Knotens abgefragt, um zu verifizieren, dass sich der Knoten noch im Zustand NULL befindet und somit noch nicht gestartet wurde. Zuletzt wird noch die $checkPredNodes$ Methode aufgerufen, wobei der Ausführungszustand der ANTEOCC Vorgängerknoten überprüft wird und dadurch sichergestellt ist, das alle relevanten Vorgänger bereits vollständig ausgeführt sind. Abhängig vom zurückgelieferten Ergebnis, wird der Knoten entweder in die Kandidatenmenge aufgenommen oder es wird mit der Überprüfung des nächsten Knotens fortgefahren.

5.3.3. Bestimmung nicht-abgeschlossener Vorgängerknoten

In Kapitel 3 wurde im Abschnitt *Noch nicht abgeschlossene Vorgänger* eine weitere Regel beschrieben, die bei jeder Ausführung mit anzuwenden ist. Denn beim Start eines Knoten, müssen nicht zwangsläufig alle Vorgängerknoten bereits vollständig ausgeführt sein ($\rightarrow$ Knotenhierarchie). So ist es beispielsweise möglich, dass ein ANTEOCC Knoten Vorgängerknoten besitzt, die sich noch im Zustand NULL oder STARTED befinden. Diese Knoten müssen nun auf NOTEXECUTED gesetzt werden, um die Markierung konsistent zu halten.

Die im Quelltext B.8 abgebildete Methode $deadConsOcc$ ermittelt alle nicht ausgeführten Vorgängernoten vom Typ CONSOCC für eine Menge von Ausgangsknoten. Der Parameter Q der Methode definiert dabei die Menge der Knoten, deren Vorgängerknoten überprüft werden sollen. Für jeden Knoten q, der in dieser Menge enthalten ist, wird nun eine rückwärts gerichtete Suche im Graphen gestartet, wobei alle vorausgehenden Knoten vom Typ CONSOCC ermittelt werden. Rückwärts gerichtet bedeutet hierbei, dass die eingehenden Sequence Kanten zurückverfolgt werden, um die jeweiligen Vorgängerknoten zu bestimmen. Für die so ermittelten Vorgänger werden nun die Ausführungsmarkierungen abgefragt und überprüft. Die Überprüfung des Ausführungszustands wurde in die Methode $isNotYetCompleted$ ausgelagert. Stellt diese Methode nun fest, dass es sich aufgrund der Markierung um einen noch nicht ausgeführten Knoten handelt, wird dieser in die Ergebnismenge mit aufgenommen, die von der Methode zurückgeliefert wird.

5.3.4. Vorbereitung der Markierungen für die Auswertung

Zu diesem Zeitpunkt ist die Ausführung nun abgeschlossen und die Events werden durch die Markierungen entsprechend abgebildet. Bevor nun aber der Compliance Zustand ausgewertet werden kann, müssen die Markierungen darauf vorbereitet werden. Dazu gehört zum einen die Entfernung widersprüchlicher Markierungen und zum anderen der Abschluss der Markierungen. Die folgenden zwei Teilabschnitte beschreiben die dafür verantwortlichen Methoden.

Absence Markierungen bereinigen

Im Quelltext B.9 ist zunächst die $cleanupCompletedAbsenceMarks$ Methode dargestellt. Genau genommen sind es sogar zwei Methoden mit dem gleichen Namen. Wobei die eine für die `ComplianceState` Datenstruktur definiert wurde, wohingegen die zweite eine `MarkStructure` als Parameter erwartet. Anders als der Methodenname zunächst vermuten lässt, ist die Methode nicht für das Löschen von Markierungen verantwortlich, die eine ausgeführte Absence Node beinhalten. Stattdessen trifft der umgekehrte Fall zu. Falls es zur Ausführung eines Absence Knotens kam, müssen alle Markierungen, in denen dies nicht korrekt vermerkt ist, gelöscht werden. Dies ist zum Beispiel in der unveränderten Ursprungsmarkierung und genauso in allen daraus abgeleiteten Markierungen der Fall.

Die Aufgliederung in zwei separate Methoden ergab sich aus dem unterschiedlichen Löschverhalten, das in Folge der Ausführung einer ANTEABS beziehungsweise einer CONSABS Node anzuwenden ist. Denn im Falle einer ANTEABS Node liegt die Inkonsistenz in der Markierung des Antecedence-Patterns und betrifft somit die gesamte MARKSTRUCTURE. Wenn nun festgestellt wird, dass eine Markierung existiert, in der dem ANTEABS Knoten der Zustand EXECUTED zugewiesen ist, müssen infolgedessen alle MARKSTRUCTURES gelöscht werden, bei denen der Absence Knoten nicht ausgeführt wurde. Handelt es sich hingegen um die Ausführung einer CONSABS Node, müssen die einzelnen CONSEXMARK Markierungen individuell überprüft werden und von Fall zu Fall über die Entfernung entschieden werden. Die Suche nach einer übereinstimmenden Markierung, jedoch mit ausgeführtem Absence Knoten, wurde in die Methode $isNotExecutedAndHasCompletedDescendants$ ausgelagert. Abhängig vom Resultat dieser Methode, wird über die Entfernung der jeweiligen Markierung entschieden.

Abschließen der Markierungen

Nachdem die Markierungen soweit aufbereitet wurden, müssen im nächsten Schritt die einzelnen Ausführungsmarkierungen untersucht werden und wenn notwendig abge-

schlossen werden. Durch diese Finalisierung werden die Markierungen zum Abschluss gebracht und sind so für die anschließende Auswertung vorbereitet.

Die im Quelltext B.10 enthaltene Methode $finalize$ übernimmt diese Aufgabe. Pro Aufruf wird durch diese Methode eine MARKSTRUCTURE zum Abschluss gebracht, wobei die enthaltenen EXMARK Markierungen der Reihe nach abgearbeitet werden. Es folgen schließlich zwei Aufrufe der $markEnd$ Methode, einmal für die Markierung des Antecedence Patterns und das andere mal für die Consequence Markierung. In der $markEnd$ Methode werden nun die einzelnen Ausführungsmarkierungen überprüft und, wenn sie bis dato nicht abgeschlossen wurden, auf NOTEXECUTED gesetzt.

5.3.5. Überprüfung der eCRG Forderungen

Vor der abschließenden Bewertung werden alle bis dato nicht sichergestellten Forderungen, die sich aus den verschiedenen eCRG Elementen ergeben, überprüft und falls verletzt in den jeweiligen Markierungen entsprechend festgehalten. In gewisser Weise, gehört diese Überprüfung damit bereits zur Auswertung, denn hier festgestellte Verletzungen werden bei der Auswertung wieder aufgegriffen und dementsprechend behandelt. In den beiden folgenden Teilabschnitten, wird anhand zweier Beispiele repräsentativ gezeigt, wie die Überprüfung der eCRG Elemente realisiert wurde. Das erste Beispiel dient der Sicherstellung der Sequence Kanten und im zweiten Beispiel wird der Algorithmus zur Überprüfung der Resource Relations veranschaulicht.

```
1 private void doChecks(ComplianceState cs) {
2    for(MarkStructure ms : cs.getMarkStructures()) {
3       for(ExMark m : ms.getAllExMarks()) {
4          checkSequence(m);
5          // ... Überprüfung der anderen eCRG Forderungen/Elemente
6       }
7    }
8 }
```

Die Methode $doChecks$ ist für die Überprüfung der Markierungen zuständig und enthält wiederum mehrere Submethoden für die Verifizierung der verschiedenen eCRG

Elemente. Die abgebildeten Programmzeilen zeigen den Kopf dieser Methode. Wie anhand dieses Ausschnitts zu erkennen ist, sind die einzelnen Prüfroutinen jeweils für eine ExMark Markierung ausgelegt.

Überprüfung der Sequence Kanten

In diesem Teilabschnitt wird die Überprüfung der Sequence Kanten besprochen. Die dafür verantwortliche Methode $checkSequence$ ist im Quelltext B.11 abgebildet. Diese Methode iteriert über alle in der eCRG enthaltenen `SequenceFlowConnectors` und überprüft die Ausführungsreihenfolge der Knoten, die durch die Kante verbunden sind. Für die allermeisten Kanten sollte dies zwar bereits aufgrund der Ausführungssemantik gegeben sein[4], aber es besteht die Möglichkeit, dass eine Sequence Kante zusätzlich noch eine Time Condition beinhaltet. Unter anderem deswegen ist eine abschließende explizite Überprüfung aller Sequence Kanten notwendig. Außerdem kann eine Sequence Kante vom Typ Consequence zwei Antecedence Knoten miteinander verbinden, wodurch die Ausführungsreihenfolge als Forderung formuliert wird. Solch eine Kombination wird bei der Ausführung nicht berücksichtigt, da beim Starten von Antecedence Knoten nur Vorgängerknoten untersucht werden, die über Antecedence Kanten zu erreichen sind. Die $checkSequence$ Methode ermittelt nun zunächst die Zeitstempel für die beiden Enden der jeweiligen Kante und überprüft anhand dieser die korrekte Abfolge der verbundenen Elemente. Wird dabei festgestellt, dass die Abfolge regelwidrig war, erhält die jeweilige Kante eine entsprechende Markierung, wodurch die Verletzung kenntlich gemacht wird. Andernfalls wird überprüft, ob für die Kante zusätzlich noch eine *Time Distance Condition* formuliert wurde. Falls dem so ist, wird die dadurch aufgestellte Relation anhand der zuvor ermittelten Zeitpunkte ausgewertet. Im Fehlerfall erhält die Kante ebenfalls eine entsprechende Markierung. Die Bestimmung der Zeitpunkte wurde in eine separate Methode ausgelagert, um sie an anderer Stelle wiederverwenden zu können. Die $getPointInTime$ Methode berücksichtigt dabei, ob die angegebene Kante mit dem Start oder dem End Port einer Task Node verbunden ist und liefert dementsprechend den dazu passenden Zeitpunkt zurück.

[4]Überprüfung des Zustands von Vorgängerknoten bei der Ausführung und Gewährleistung der zeitlichen Abfolge von Events durch Sortierung dieser

Überprüfung der Resource Relations

Die Überprüfung der Resource Relations ist im Vergleich zur Verifizierung der Sequence Kanten um einiges schwieriger, weswegen der Programmcode für die Implementierung dessen auch deutlich umfangreicher ausfällt. Aufgrund des Umfangs wird an dieser Stelle auf eine detaillierte Beschreibung der konkreten Implementierung verzichtet und stattdessen eine informelle Beschreibung der dafür zuständigen Prüfroutine gegeben. Anhand der einzelnen Schritte von Algorithmus 2 soll die Vorgehensweise zur Überprüfung der Resource Relations veranschaulicht werden, sodass der Ablauf der Routine und damit die Belegung der Resource Knoten nachvollziehbar ist. Algorithmus 2 stellt dabei zwar keine eins zu eins Abbildung der Implementierung dar, vermittelt aber eine Vorstellung darüber, wie die Prüfroutine im Programm realisiert wurde.

Algorithmus 2 Algorithmus zur Überprüfung der Resource Relations in Pseudocode

```
Require: Peform Kanten wurden bei der Ausführung entsprechend markiert
Ensure: Belegung der Resource Nodes unter Einhaltung der Resource Relations
 1: function CHECKRESOURCERELATIONS(ExMark m)
 2:     rrc := GETRESOURCERELATIONCONNECTORS( )
 3:     leafs := { }
 4:     candidates := { }

 5:     ASSIGNSTAFFNODES(m)                                              ▷ I.
 6:     COLLECTSPECLEAFS(m, leafs)                                       ▷ II.
 7:     INITIALIZECANDIDATES(m, candidates)                              ▷ III.

 8:     TRAVERSEANTERESOURCERELATIONS(m, rrc, candidates)      ▷ BFS     ▷ IV.
 9:     candidatesAfterAnte := candidates
10:     TRAVERSECONSRESOURCERELATIONS(m, rrc, candidates)      ▷ BFS     ▷ V.

11:     FINALIZECANDIDATES(candidates)                                   ▷ VI.
12:     ASSIGNCANDIDATESTONODES(candidates, candidatesAfterAnte)         ▷ VII.
13: end function
```

In Abbildung 5.13 ist eine eCRG Regel dargestellt, die im Vergleich zu früheren eCRG Beispielen in dieser Arbeit eine deutlich umfangreichere Ressourcen-Perspektive besitzt. Die Resource Knoten und die dazugehörigen Resource Relation Kanten modellieren dabei einen separaten Subgraph innerhalb des eCRG Gesamtgraph. Die Staff Member Nodes, die durch eine Performkante mit einer Task Node verbunden sind, werden im Folgenden auch als Blattknoten (*Leafs*) des Graphen bezeichnet. Die Besonderheit bei

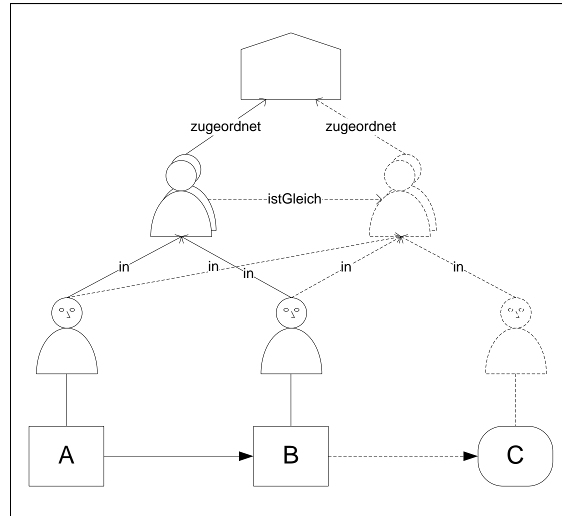


Abbildung 5.13.: Beispiel einer eCRG mit umfangreicher Ressourcen-Perspektive

dieser Art von Knoten ist, dass sie aufgrund des Mitarbeiters, der für die Performkante vermerkt wurde, spezifiziert sind. Voraussetzung für den Start der in Algorithmus 2 dargestellten Routine ist deswegen auch, dass die Performkanten bei der Ausführung der Task Nodes mit dem jeweils verantwortlichen Mitarbeiter markiert werden.

Die Schwierigkeit bei eCRG Regeln mit einer umfangreichen und zugleich vielschichtigen Ressourcen-Perspektive ist, dass es für die Belegung der nicht spezifizierten Resource Knoten möglicherweise eine Vielzahl an Möglichkeiten gibt und sich diese wiederum verschieden kombinieren lassen. Das in Abbildung 5.13 dargestellte Beispiel soll dies veranschaulichen. Denn für die nicht spezifizierten Knoten ist es dort unter Umständen gar nicht so einfach, eine korrekte Belegung zu finden, die alle Resource Relations erfüllt, oder aber deren Nichtexistenz nachzuweisen. Denn ein Knoten kann zu mehreren Knoten in Beziehung stehen und hängt somit auch von verschiedenen anderen Belegungen ab, welche wiederum die eigene Belegung einschränken.

Ein trivialer Ansatz zur Lösung dieses Belegungsproblems wäre eine Tiefensuche mit Backtracking über den Resource Graphen laufen zu lassen. Wird dabei eine Lösung gefunden, hätte man eine erfüllende Belegung und könnte die Knoten entsprechend belegen. Die in dieser Arbeit realisierte Prüfroutine verfolgt einen ähnlichen Ansatz, geht dabei aber den umgekehrten Weg. Statt bei Null zu beginnen, wird zunächst für

jeden freien Knoten eine Kandidatenmenge ermittelt, in der alle möglichen Werte aus der Prozessumgebung enthalten sind. Diese Menge wird nun der Reihe nach für die verschiedenen Resource Relation überprüft, wobei Kandidaten aus der Menge gestrichen werden, wenn durch diese die Relation nicht erfüllt wird. Bleibt am Ende für jeden Knoten mindestens ein Kandidat übrig, wurde ebenfalls eine erfüllende Belegung gefunden. Im Gegensatz zur Backtracking Lösung werden so keine Teillösungen verworfen, sondern es genügen lokale Anpassungen, um die Kandidatenmengen aktuell zu halten. Außerdem ist der für die Implementierung gewählte Ansatz gegenüber dem auf Tiefensuche basierenden Ansatz im Vorteil, wenn für die jeweilige Ausführung keine erfüllende Belegung existiert. Denn statt nun jede mögliche Kombination per Tiefensuche zu durchlaufen, können bei dem auf Kandidatenmengen beruhenden Ansatz, die verschieden Belegungen ausprobiert werden und die gewählt werden, welche die wenigsten Verletzungen zur Folge hat.

Im Folgenden werden nun die einzelnen Schritte des zuvor in Pseudocode angegebenen Algorithmus 2 beschrieben:

I. Als erstes werden die Staff Member Nodes mit dem Mitarbeiter belegt, der bei der jeweiligen Performkante vermerkt wurde. Diese nun belegten Staff Member Nodes repräsentieren die Blätter (Leafs) des Resource Graphen.

II. Im nächsten Schritt werden nun die gerade eben spezifizierten Blattknoten in der Datenstruktur $leafs$ zusammengefasst. Aufgrund der fixen Belegung dieser Knoten, startet die Überprüfung der Resource Relations bei diesen Knoten, um so möglichst schon zu Beginn die Kandidatenmengen einzuschränken und dadurch kleinzubekommen.

III. Im dritten Schritt des Algorithmus werden den verschiedenen bis dato nicht spezifizierten Knoten ihre initialen Kandidatenmengen zugewiesen. In jeder dieser Kandidatenmengen sind zu Beginn alle in der Prozessumgebung verfügbaren Objekte enthalten, die vom selben Ressourcentyp wie der jeweilige Knoten sind.

IV. Mit diesem Schritt beginnt nun die Suche nach einer passenden Belegung, zunächst für das Antecedence-Pattern, wobei der Resource Graph in Form einer Breitensuche (BFS) von unten nach oben durchlaufen wird. Startend bei den An-

tecedence Blattknoten werden nun die Resource Relations der Reihe nach ausgewertet und die Kandidatenmengen dabei entsprechend angepasst. Neu hinzugekommene Knoten werden in eine Queue aufgenommen und sind so für eine spätere Betrachtung vorgemerkt.
Ein wichtiger Aspekt sei an dieser Stelle noch angemerkt, ergibt sich bei der Überprüfung einer Resource Relation eine Anpassung der Kandidatenmenge, müssen die bereits bewerteten Nachbarknoten darüber informiert werden, sodass deren Kandidatenmenge unter Umständen abermals angepasst werden muss. Diese Info wird solange im Graphen weiterverbreitet, bis es zu keiner weiteren Anpassung mehr kommt.

V. Nachdem die Kandidaten für das Antecedence-Pattern gesichert wurden, wird nun die Suche auf das Consequence-Pattern ausgeweitet. Wobei abermals Kandidaten aus den Mengen gestrichen werden, wenn diese nicht zur Erfüllung der jeweiligen Relation beitragen. Am Ende sind in den Mengen somit nur mehr gültige Kandidaten vertreten.

VI. Falls für die verschiedenen Knoten teilweise mehr als ein passender Kandidat ermittelt werden konnte, wird versucht, daraus eine erfüllende Belegung zu konstruieren und sich jeweils auf genau einen Kandidaten festzulegen. Für den Fall, dass keine erfüllende Belegung existiert, wird die Belegung gesucht, welche die wenigsten Verletzungen zur Folge hat. Zusätzlich werden die Verletzungen in den dazugehörigen Markierungen für die Auswertung vermerkt.

VII. Abschließend wird nun aufgrund der ermittelten Kandidatenmengen die Belegung der nicht-spezifizierten Resource Knoten vorgenommen. Dies ist eindeutig, da nach Durchführung des vorherigen Schritts nur noch maximal ein Kandidat verbleibt, wenn nicht schon zuvor alle Kandidaten gestrichen worden sind.

5.3.6. Auswertung der Markierungen

Die Programmzeilen des Abschnitts $Evaluation$ im Quelltext B.3 sorgen am Ende der Ausführung für die Evaluierung der Markierungen. Zunächst werden die verschiedenen EXMARK Markierungen durchgegangen und ausgewertet, bevor im Anschluss daran die MARKSTRUCTURES evaluiert werden. Diese Reihenfolge ist notwendig, da sich das Ergebnis für die MARKSTRUCTURES aus den einzelnen EXMARK Bewertungen ableitet. Das Resultat der ANTEEXMARK Evaluierung wird für die Bestimmung des Aktivierungszustands der MARKSTRUCTURE herangezogen und aus den CONSEXMARK Markierungen leitet sich der Erfüllungsstatus ab.

Die Implementierung dieser abschließenden Bewertung ist im Quelltext B.12 für die Evaluierung der MARKSTRUCTURES veranschaulicht und entspricht der in Kapitel 3 hierfür eingeführten Regel. Die dort abgebildete Methode $evaluate(ms)$ bestimmt den Aktivierungszustand und den Erfüllungsstatus der angegebenen `MarkStructure`. Voraussetzung hierbei ist, dass die enthaltenen EXMARK Markierungen bereits ausgewertet wurden. Ob durch die MARKSTRUCTURE nun eine Aktivierung der Compliance Regel vorliegt oder nicht, folgt aus der jeweiligen ANTEEXMARK Markierung. Je nachdem, ob die Markierung das Antecedence-Pattern verletzt oder eine Erfüllung dessen darstellt, wird der Aktivierungszustand der MARKSTRUCTURE auf DEACTIVATED beziehungsweise ACTIVATED gesetzt. Falls dabei nun eine Aktivierung der Regel festgestellt wurde, wird im Folgenden der Erfüllungsstatus ermittelt. Dazu werden die einzelnen CONSEXMARK Markierungen der Reihe nach durchgegangen und geschaut, ob eine der Markierungen das Consequence Pattern erfüllt. Abhängig davon wird anschließend der Erfüllungsstatus entweder auf SATISFIED oder VIOLATED gesetzt. Um kenntlich zu machen, dass die Bewertung abgeschlossen ist und sich an den Markierungen nichts mehr ändert, wird die MARKSTRUCTURE zum Schluss noch auf FINAL gesetzt.

Wurden alle vorhandenen MARKSTRUCTURES auf diese Weise ausgewertet, ist die Ausführung damit abgeschlossen und durch Aufruf der $isSatisfied$ Methode an dem zurückgelieferten `ComplianceState` Objekt, lässt sich schließlich das Ergebnis der Compliance Prüfung abfragen.

6

Related Work

Thema diese Kapitels sind inhaltlich verwandte Arbeiten, die ebenfalls im Gebiet der Business Process Compliance anzusiedeln sind und damit demselben Forschungsbereich angehören. Dadurch sollen andere Ansätze zur Überwachung der BPC aufgezeigt werden, wobei zugleich die Unterschiede der verschiedenen Ansätze beleuchtet werden.

Die im Nachfolgenden beschrieben Ansätze zur Automatisierung der Compliance Prüfung für Geschäftsprozesse unterscheiden sich einerseits anhand der Methode, die ihnen zugrunde liegt, und andererseits durch den Zeitpunkt der Überprüfung. So lassen sich die verschiedenen Verfahren zur Überprüfung der Compliance in zwei Gruppen aufteilen: Zum einen die Verfahren die vor der Prozessausführung ansetzen und eine Validierung des Prozessmodells zur Modellierungszeit vornehmen [LMX07, FSWS10, TEvHP12]; und zum anderen die Ansätze, die eine Überprüfung zur Laufzeit beziehungsweise nach der Ausführung ermöglichen [NS07, ALS11, RFvA12, KRL+13b]. Abhängig vom

jeweiligen Zeitpunkt wird bei den verschiedenen Verfahren zur Überwachung der Compliance auch von *Design Time* Compliance Checking oder *Run Time* Monitoring gesprochen [KR11]. Außerdem kann das jeweilige Konzept, das den verschiedenen Ansätzen zugrunde liegt, als weiteres Klassifikationsmerkmal herangezogen werden. Typischerweise wird dabei eine Einteilung in drei unterschiedliche Klassen vorgenommen [LRMKD11]. Dies wären die auf logischen Formalismen beruhenden Verfahren [GHSW09, KKdV10], als nächstes die auf Pattern (Mustern) basierenden Methoden [NS07, TEvHP12] und zuletzt noch die Klasse der Ansätze, denen das Konzept eines Automaten zugrunde liegt [FSWS10, AWW11]. Bei vielen dieser Ansätze wird jedoch von den teils komplexen Grundformalismen abstrahiert und stattdessen einen einfachere Verwendung ermöglicht. So wird zum Beispiel eine grafische Notation zur Modellierung der Compliance Regeln angeboten und intern eine Transformation dessen in den eigentlichen Formalismus durchgeführt. Dadurch sind auch Nutzergruppen in der Lage Compliance Regeln zu formulieren, die ansonsten aufgrund fehlendem IT-Fachwissens außen vor blieben.

Der in [LMX07] beschriebene Ansatz ermöglicht die visuelle Darstellung und Modellierung von Compliance Regeln in der Business Property Specification Language (BPSL). Diese Sprache stellt eine grafische Repräsentation der Compliance Regeln dar, wobei die so definierten Regeln für die Überprüfung in die Linear Temporal Logik (LTL) transformiert werden. Damit kann nun die Compliance eines Prozessmodells feststellt werden. Die automatische Überprüfung, die dem zugrunde liegt, basiert auf dem Konzept des Model-Checkings. Dabei wird überprüft, ob sich der Geschäftsprozess an die auferlegten Regelungen hält oder diese verletzt. Zwar liegt der Fokus bei BPSL auf der Transformation in LTL, jedoch ist auch eine Übersetzung in CTL (Computation Tree Logic) möglich, wodurch zeitliche Aspekte unterstützt werden.
In [GHSW09] wird ein Ansatz zur Überprüfung der Compliance von Geschäftsprozessen aufgezeigt, der auf semantischen Annotationen (Anmerkungen) beruht. Das in der Arbeit beschriebene Framework ermöglicht die Sicherstellung der Compliance für Geschäftsprozesse und ist der Lage die Verpflichtungen, die sich aus den Regelannotationen ergeben, zu identifizieren und zu bewerten.
Ein weiterer Vorschlag zur Überprüfung der Compliance von Prozessmodellen beruht auf semantischen Constraints (Einschränkungen) und sogenanntem Mixed-Integer Pro-

gramming. Wobei erstere den Bedingungen entsprechen, gegen die geprüft wird, und mit letzterem die Gültigkeit von Prozessänderungen verifiziert wird. Beschrieben ist dieser Ansatz in [KYC13].
Der in [KKdV10] beschriebene Ansatz, beruht auf Techniken des Model-Checkings. Anhand des in der Arbeit vorgeschlagenen Konzepts lässt sich überprüfen, inwieweit die geforderten Compliance Regeln von Prozessmodellen zum Zeitpunkt der Modellierung eingehalten werden. Außerdem unterstützt dieser Ansatz auch die Überprüfung von Compliance Regeln, welche die Zeit- und Daten-Perspektive betreffen.
Ebenfalls zur Design Time setzt der in [TEvHP12] beschriebene Pattern-basierte Ansatz an. Das darin vorgestellte Framwork dient ebenfalls der Ermittlung der Compliance von Geschäftsprozessen und nutzt dafür LTL und MTL (Metric Temporal Logic) als formale Sprachen. Mittels MTL werden auch Patterns unterstützt, die einen zeitlichen Aspekt modellieren. Diese Patterns gehören in diesem Fall der *Time* Kategorie an. Des Weiteren existiert noch eine *Order*, *Occurrence* und eine *Resource* Kategorie. Für jedes der unterstützen Patterns wurde eine Abbildung in LTL respektive MTL definiert, sodass sich die entsprechenden Formeln für die Überprüfung automatisch generieren lassen.
In [FSWS10] wird ein Ansatz zur grafischen Repräsentation von Compliance Regeln vorgestellt, der zudem die Überprüfung von Prozessmodellen ermöglicht. Dazu werden die grafisch spezifizierten Regeln in eine formale Sprache transformiert und die dabei generierten Formeln schließlich verifiziert. Die hierbei entwickelte grafische Sprache nennt sich G-CTL und basiert, wie der Name bereits andeutet, auf CTL.
Der in [AWW11] beschriebene Ansatz nutzt ebenfalls eine grafische Sprache, um die Compliance Vorgaben zu visualisieren. Die dafür verwendete Sprache basiert auf BPMN und heißt BPMN-Q. Damit wird, genauso wie bei BPMN, ein Graph modelliert, der sich aus verschiedenen Pattern zusammensetzt. Jede in BPMN-Q modellierte Compliance Rule besitzt somit eine entsprechende Darstellung als Graph. Zur Überprüfung lässt sich dieser Graph in einen CTL Temporallogik Ausdruck übersetzen. Neben dem Kontrollfluss werden hierbei auch Datenfluss Aspekte mitberücksichtigt.

In [NS07] wird ein Pattern-basierter Ansatz beschrieben, der zudem eine semantische Ebene (Layer) einführt. Anders als die vorherigen Ansätze, unterstützt dieses Konzept die Überwachung von Prozessinstanzen zur Laufzeit. Die Grundidee hierbei ist die

Einführung einer semantischen Zwischenebene, in welcher die Prozessinstanzen anhand von intern vorliegenden Kontrollmechanismen überprüft werden. Diese internen Kontrollen werden durch Pattern modelliert.
Das in [ALS11] beschriebene Verfahren zur automatisierten Compliance Prüfung von Geschäftsprozessen setzt Petrinetz-Pattern zur Überwachung ein. Die Compliance Regeln werden dabei in Form von Petrinetzen definiert. Die Muster (Pattern) aus denen diese Petrinetze bestehen werden bei der Analyse von Prozessinstanzen, je nach Prozessverlauf, weiter geschaltet. Wird dabei ein erfüllender Zustand erreicht, ist sichergestellt, dass die Forderungen der Compliance Regel eingehalten werden. Andernfalls liegt entweder eine Verletzung vor oder der erfüllende Zustand wurde noch nicht erreicht, da noch nicht alle Forderungen eingetreten sind. Dieser Ansatz unterstützt neben der Kontrollfluss-Perspektive auch Datenfluss-Aspekte.
Der Ansatz aus [RFvA12] nutzt ebenfalls Petrinetze und führt zusätzlich sogenannte Alignments ein. Für die Modellierung von Compliance Regeln werden, wie beim zuvor beschriebenen Ansatz, Petrinetz-Pattern verwendet, welche in diesem Fall auch als Petrinetz-Fragmente bezeichnet werden. Um nun die Compliance einer bestimmten Regel zu ermitteln, wird der Event-Log, der die zu überprüfende Prozessinstanz beschreibt, der Reihe nach abgearbeitet und die einzelnen Events den jeweils passenden Fragmenten zugeordnet. Auf Basis dessen wird nun überprüft, ob es Abweichungen zwischen den Logeinträgen (Events) und den formal spezifizierten Petrinetz-Fragmenten gibt. Anhand der Alignments werden somit die Compliance Verletzungen einer Prozessinstanz aufgedeckt. Zusätzlich zur Kontrollfluss-Perspektive unterstützt dieser Ansatz auch Compliance Regeln mit Bezug zur Daten- und Ressourcen-Perspektive.

7 Zusammenfassung & Ausblick

Diese Arbeit beschreibt die Entwicklung eines Programms, welches auf Basis von eCRG die automatisierte Überwachung der Compliance von Geschäftsprozessen ermöglicht. Im Mittelpunkt der Arbeit steht dabei die erste Realisierung einer Ausführungsengine für eCRG sowie die Beschreibung der zugrundeliegenden Konzepte. Einen weiteren Schwerpunkt stellt die grafische Visualisierung von Compliance Zuständen dar.

Das im Rahmen dieser Arbeit entwickelte Programm dient als *Proof of Concept* Implementierung, wodurch die Eignung und die Tauglichkeit des eCRG Konzepts zur Abbildung und Überprüfung von Compliance Regeln gezeigt wird. Des Weiteren wird dadurch der von eCRG gewählte Ansatz zur Überwachung der BPC verifiziert. Die Ausführungsengine für eCRG, die aus dieser Arbeit hervorgeht, unterstützt dabei alle fünf Perspektiven von eCRG und ermöglicht so die Überwachung von Compliance Regeln mit Bezug zur Prozess-, Interaktions-, Zeit-, Daten- und Ressourcen-Perspektive.

Inhaltlich untergliedern sich die sechs vorausgehenden Kapitel wie folgt. Kapitel 1 beginnt mit der Einleitung und definiert die Ziele dieser Arbeit. Die fachlichen Grundlagen, auf denen diese Arbeit aufbaut, werden anschließend in Kapitel 2 erläutert. Zunächst werden dabei die CRG besprochen, bevor im Anschluss daran, deren Erweiterung, die eCRG beschrieben werden. In Kapitel 3 werden die grundlegenden Konzepte und Regeln eingeführt, die in der Ausführungsengine umgesetzt wurden und für die Ausführung und Bewertung einer eCRG verantwortlich sind. Das der Implementierung zugrundeliegende Architekturkonzept wird in Kapitel 4 beschrieben und beinhaltet unter anderem das Datenmodell für die Abbildung der eCRG. Kapitel 5 widmet sich schließlich der Implementierung der eCRG Ausführungsengine, wobei unter anderem die verschiedenen Funktionen der grafischen Benutzeroberfläche beschrieben werden. Außerdem wird in diesem Kapitel die Realisierung des Monitorings im Programm näher beleuchtet. Im Anschluss daran werden in Kapitel 6 verschiedene andere Arbeiten vorgestellt und miteinander verglichen, die ebenfalls die Überwachung der BPC zum Thema haben.

Aufbauend auf dieser Arbeit sind für die Zukunft folgende Erweiterungen und Optimierungen angedacht. Zum einen sollen die bis dato noch nicht enthaltenen PointInTime Nodes in die Ausführungsengine integriert werden, sodass auch die Zeit-Perspektive von eCRG Regeln vollständig unterstützt wird. Zudem soll der implementierte Monitoring Algorithmus auf mögliche Optimierungen hin untersucht werden und dadurch weiter verbessert werden. Um die Einbindung und Modellierung von eCRG Regeln zu vereinfachen, soll das Programm an einen grafischen Editor angebunden werden beziehungsweise eine eigenständige Applikation hierfür realisiert werden. Des Weiteren wird die Integration der entwickelten Ausführungsengine in eine bestehende Prozessmanagement Systemlandschaft angestrebt, um so direkten Zugriff auf aktuelle Prozess-Ablaufprotokolle und die Organisationsstrukturen von Unternehmen (Ressourcenmodelle) zu erhalten. Außerdem soll der Inhalt dieser Arbeit in eine Publikation einfließen und veröffentlicht werden.

A
Abbildungen

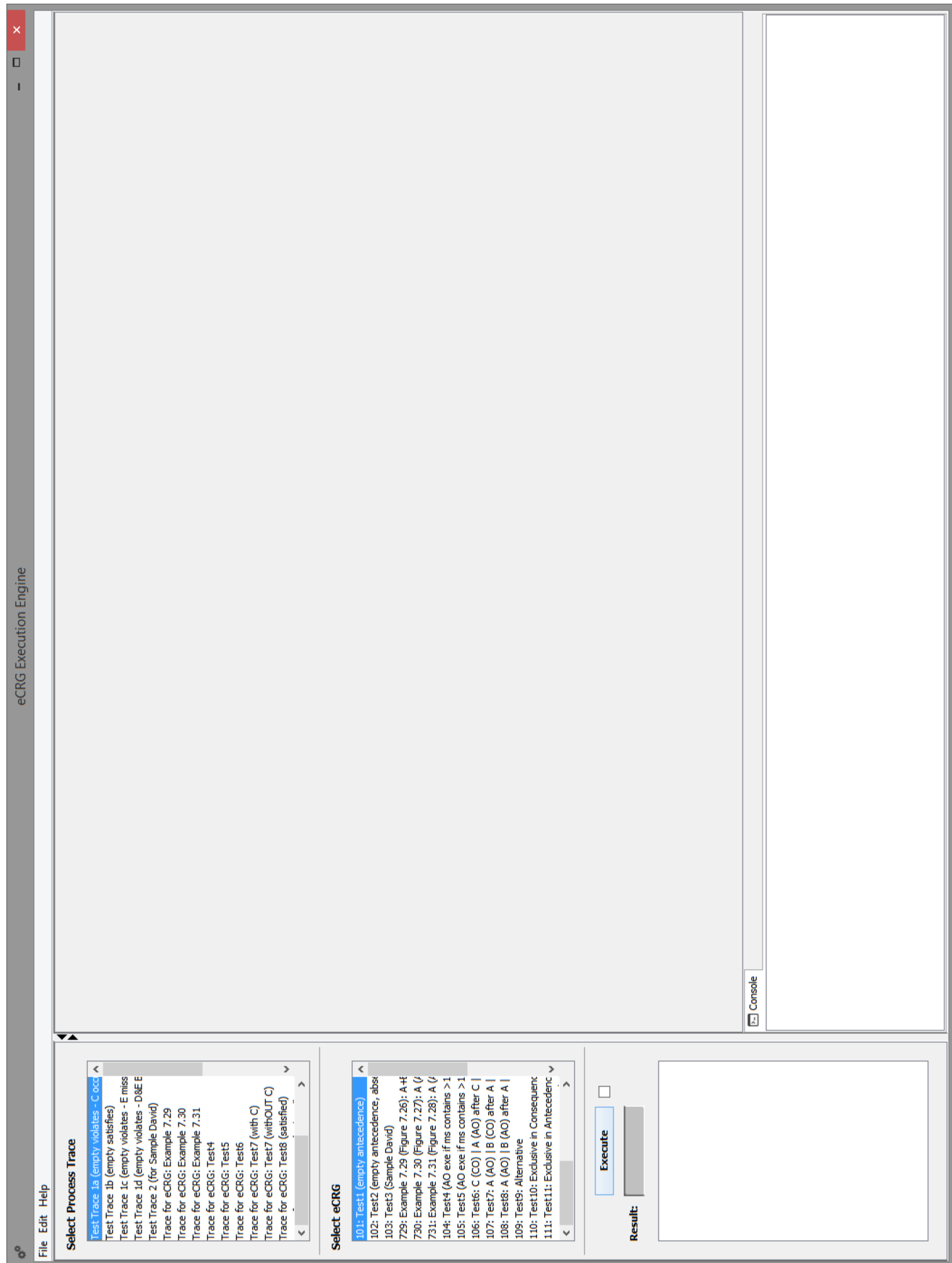


Abbildung A.1.: Startansicht

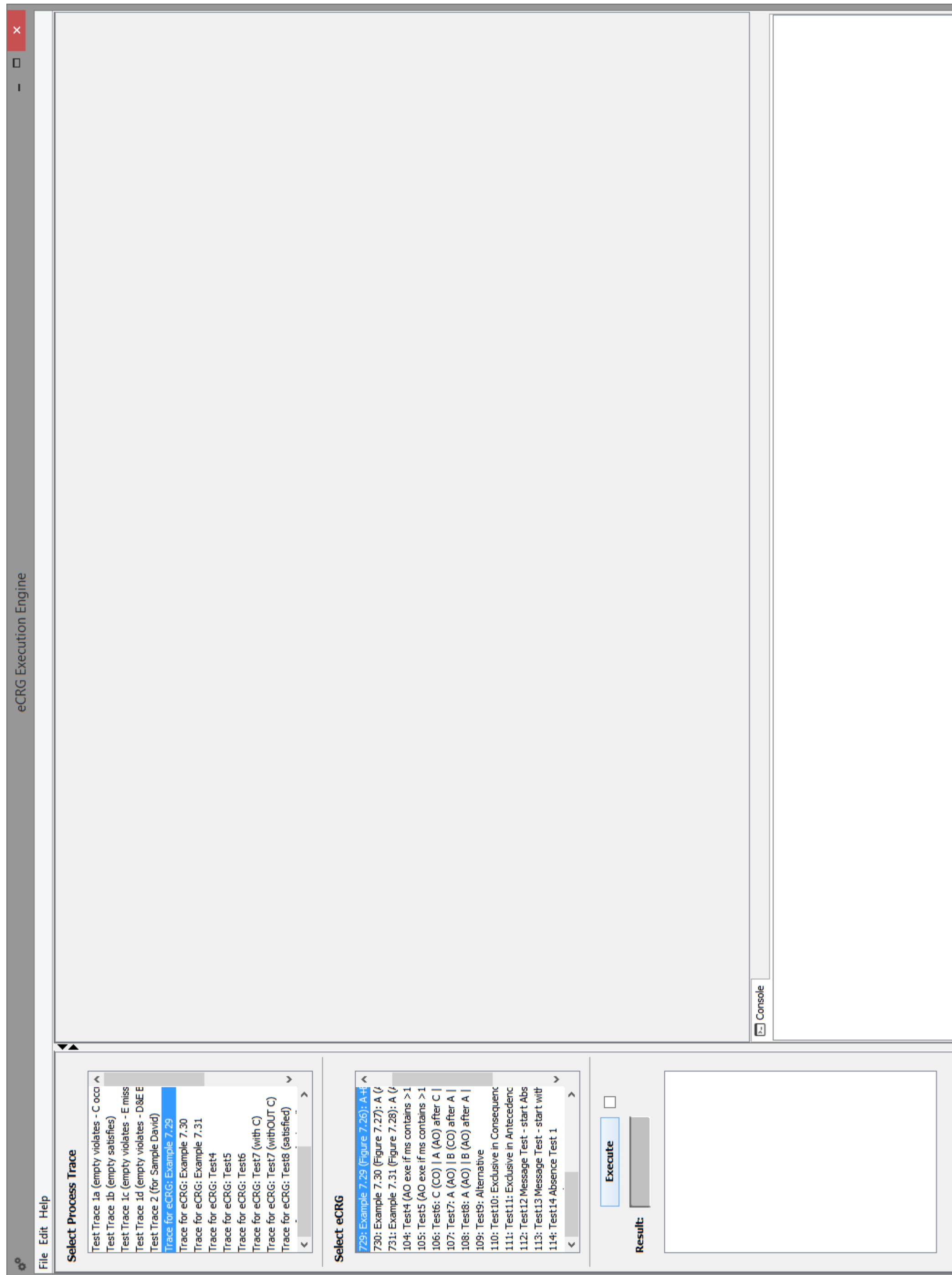


Abbildung A.2.: Schritt 1 – Auswahl des Trace und der eCRG

Abbildung A.3.: Schritt 2 – Start der Ausführung & Bewertung

Abbildung A.4.: Schritt 3 – Ansicht der erreichten Markierungszustände

B
Quelltexte

Die Klasse myDataShape

```
/**
 * This class defines the shape for the visualization of Data Nodes
 */
public class myDataShape extends mxBasicShape {

   @Override
   public void paintShape(mxGraphics2DCanvas canvas, mxCellState state)
   {
      if (state.getCell() instanceof mxCell)
      {
         Object value = ((mxCell) state.getCell()).getValue();
         if(value instanceof DataNode)
         {
            DataNode dataNode = (DataNode) value;
            float strokeWidth = mxUtils.getFloat(state.getStyle(),
                  mxConstants.STYLE_STROKEWIDTH, 1););

            Rectangle rect = state.getRectangle();
            int x = rect.x;
            int y = rect.y;
            int w = rect.width;
            int h = rect.height;
            int h4 = h / 4;
            int h2 = h4 / 2;
            int w23 = 2*w/3;

            // Paints the background
            if (configureGraphics(canvas, state, true))
            {
               Area area = new Area();
               if(dataNode.isDataContainerNode()) {
                  area.add(new Area(new Rectangle(x, y + h4/2, w, h-h4)));
                  area.add(new Area(new Ellipse2D.Float(x, y, w, h4)));
                  area.add(new Area(new Ellipse2D.Float(x, y+h-h4, w, h4)));
               } else if (dataNode.isDataObjectNode()) {
                  area.add(new Area(new Rectangle(x, y, w23, h4)));
```

```
37                 area.add(new Area(new Rectangle(x, y+h4, w, 3*h4)));
38                 int[] xpoints = {x+w23, x+w23, x+w};
39                 int[] ypoints = {y, y+h4, y + h4};
40                 area.add(new Area(new Polygon(xpoints, ypoints, 3)));
41             }
42             canvas.fillShape(area, hasShadow(canvas, state));
43         }
44 
45         // Paints the foreground
46         if (configureGraphics(canvas, state, false))
47         {
48             if(dataNode.getPattern() == Pattern.I)
49                 canvas.getGraphics().setStroke(
50                         new BasicStroke(Math.round(2 * strokeWidth)));
51             else if(dataNode.getPattern() == Pattern.Consequence)
52                 canvas.getGraphics().setStroke(
53                         new BasicStroke(strokeWidth,
54                                 BasicStroke.CAP_BUTT,
55                                 BasicStroke.JOIN_MITER, 1.0f,
56                                 new float[] { 4.0f, 2.0f }, 0.0f));
57 
58             if(dataNode.isDataContainerNode()) {
59                 canvas.getGraphics().drawOval(x, y, w, h4);
60                 canvas.getGraphics().drawLine(x, y+h2, x, y+h-h2);
61                 canvas.getGraphics().drawLine(x+w, y+h2, x+w, y+h-h2);
62                 canvas.getGraphics().drawArc(x, y+h-h4, w, h4, 0, -180);
63             } else if(dataNode.isDataObjectNode()) {
64                 canvas.getGraphics().drawLine(x, y, x, y+h);
65                 canvas.getGraphics().drawLine(x, y+h, x+w, y+h);
66                 canvas.getGraphics().drawLine(x+w, y+h, x+w, y+h4);
67 
68                 canvas.getGraphics().drawLine(x, y, x+w23, y);
69                 canvas.getGraphics().drawLine(x+w23, y, x+w, y+h4);
70                 canvas.getGraphics().drawLine(x+w23, y+h4, x+w23, y);
71                 canvas.getGraphics().drawLine(x+w23, y+h4, x+w, y+h4);
72             }
73         }
74         drawLabel(canvas, state, dataNode);
```

```
75           }
76        }
77     }
78
79     private void drawLabel(mxGraphics2DCanvas canvas, mxCellState state,
80           DataNode dataNode) {
81        Rectangle rect = state.getRectangle();
82        Map<String, Object> style = state.getStyle();
83        Color fontColor = mxUtils.getColor(style,
84              mxConstants.STYLE_FONTCOLOR, Color.black);
85        canvas.getGraphics().setColor(fontColor);
86        Font scaledFont = mxUtils.getFont(style, canvas.getScale());
87        canvas.getGraphics().setFont(scaledFont);
88        FontMetrics fm = canvas.getGraphics().getFontMetrics();
89
90        String l = null;
91        int dy = 0;
92        if(dataNode.isDataContainerNode()) {
93           l = dataNode.getDataContainer().getContainerName();
94           dy = 5*rect.height/8;
95        } else if(dataNode.isDataObjectNode()) {
96           IDataObject o = dataNode.getDataObject();
97           if(o instanceof DataObjectValue) {
98              l = ((DataObjectValue) o).getValue()+"";
99           } else if(o instanceof DataObjectID) {
100             DataObjectID ido = (DataObjectID) o;
101             l = ido.getName();
102          }
103          dy = rect.height/2 + (2*fm.getMaxAscent()-fm.getHeight())/2;
104       }
105       if(l != null) {
106          int sw = fm.stringWidth(l);
107          int dx = (rect.width - sw) / 2;
108          canvas.getGraphics().drawString(l, rect.x + dx, rect.y + dy);
109       }
110    }
111 }
```

Quelltext B.1: Originalcode der `myDataShape` Klasse für die Form der Data Nodes

Die Klasse `PortFactory`

```
/**
 * Handles the creation of the port cells
 */
public class PortFactory {

   public enum PortDirection {
      West, East, North, South
   }
   public static final int PORT_RADIUS = 2;
   public static final int PORT_DIAMETER = PORT_RADIUS * 2;
   private static final String portStyle =
       "shape=ellipse;perimter=ellipsePerimeter;fillColor=lightgray";

   private static final mxGeometry geoWest = new mxGeometry(0, 0.5,
         PORT_DIAMETER, PORT_DIAMETER);
   private static final mxGeometry geoEast = new mxGeometry(1.0, 0.5,
         PORT_DIAMETER, PORT_DIAMETER);
   private static final mxGeometry geoNorth = new mxGeometry(0.5, 0,
         PORT_DIAMETER, PORT_DIAMETER);
   private static final mxGeometry geoSouth = new mxGeometry(0.5, 1.0,
         PORT_DIAMETER, PORT_DIAMETER);

   static {
      geoWest.setOffset(new mxPoint(-PORT_RADIUS, -PORT_RADIUS));
      geoWest.setRelative(true);

      geoEast.setOffset(new mxPoint(-PORT_RADIUS , -PORT_RADIUS));
      geoEast.setRelative(true);

      geoNorth.setOffset(new mxPoint(-PORT_RADIUS, -PORT_RADIUS));
      geoNorth.setRelative(true);

      geoSouth.setOffset(new mxPoint(-PORT_RADIUS , -PORT_RADIUS));
      geoSouth.setRelative(true);
   }

```

```
36    public static mxCell createPort(PortDirection direction) {
37       mxCell port = new mxCell(direction, getGeo(direction), portStyle);
38       port.setVertex(true);
39       return port;
40    }
41
42    private static mxGeometry getGeo(PortDirection direction) {
43       if (direction == PortDirection.West)
44          return geoWest;
45       else if (direction == PortDirection.East)
46          return geoEast;
47       else if (direction == PortDirection.North)
48          return geoNorth;
49       else //if (direction == PortDirection.South)
50          return geoSouth;
51    }
52 }
```

Quelltext B.2: Die Klasse `PortFactory` zur Erstellung von Port-Knoten (Zellen)

Die Methode `execute`

```java
private ComplianceState execute() {
   // INITIALIZATION
   ComplianceState cs = new ComplianceState(ecrg, trace);

   // reset the global data store
   DataStorage.clear();

   // ITERATION
   for (Event e : trace.getAllEvents()) {
      TreeSet<MarkStructure> MSnew = new TreeSet<>();

      if (e instanceof StartTaskEvent) {
         for (MarkStructure ms : cs.getMarkStructures())
            MSnew.addAll(executeStart(ms, (StartTaskEvent) e));
      } else if (e instanceof EndTaskEvent) {
         for (MarkStructure ms : cs.getMarkStructures())
            MSnew.add(executeEnd(ms, (EndTaskEvent) e));
      } else if (e instanceof SendMessageEvent) {
         for (MarkStructure ms : cs.getMarkStructures())
            MSnew.addAll(executeMessage(ms, (SendMessageEvent) e));
      } else if (e instanceof ReceiveMessageEvent) {
         for (MarkStructure ms : cs.getMarkStructures())
            MSnew.addAll(executeMessage(ms, (ReceiveMessageEvent) e));
      } else if (e instanceof WriteEvent) {
         for (MarkStructure ms : cs.getMarkStructures())
            MSnew.add(executeIO(ms, (WriteEvent) e));
      } else if (e instanceof ReadEvent) {
         for (MarkStructure ms : cs.getMarkStructures())
            MSnew.add(executeIO(ms, (ReadEvent) e));
      }
      cs.setMarkStructures(new LinkedHashSet<MarkStructure>(MSnew));
   }

   // CLEANUP
   cleanupCompletedAbsenceMarks(cs);

```

```
37      // FINALIZATION
38      for (MarkStructure ms : cs.getMarkStructures()) {
39         finalize(ms);
40      }
41
42      // CHECKS
43      doChecks(cs);
44
45      // EVALUATION
46      for (MarkStructure ms : cs.getMarkStructures()) {
47         for (ExMark m : ms.getAllExMarks()) {
48            evaluate(m.getAnteExMark());
49            evaluate(m.getConsExMark());
50         }
51         evaluate(ms);
52      }
53
54      return cs;
55   }
```

Quelltext B.3: Ausführung und Bewertung eines Trace über eine eCRG

Die Methode `executeStart`

```
1  private Collection<MarkStructure> executeStart(MarkStructure ms,
       StartTaskEvent e) {
2     LinkedHashSet<ExMark> M_ms = ms.getAllExMarks();
3
4     // INIT
5     LinkedHashSet<ExMark> M_AnteOcc = null;
6     LinkedHashSet<ExMark> M_AnteAbs = null;
7     LinkedHashSet<ExMark> M_ConsOcc = null;
8     LinkedHashSet<ExMark> M_ConsAbs = null;
9     LinkedHashSet<ExMark> M_Res = new LinkedHashSet<>();
10
11    // ITERATION
12    for(ExMark m : M_ms) {
13       M_AnteOcc = executeAnteOccStart(m, e);
14       for(ExMark m2 : M_AnteOcc) {
15          M_AnteAbs = executeAnteAbsStart(m2, e);
16          for(ExMark m3 : M_AnteAbs) {
17             M_ConsOcc = executeConsOccStart(m3, e);
18             for(ExMark m4 : M_ConsOcc) {
19                M_ConsAbs = executeConsAbsStart(m4, e);
20                M_Res.addAll(M_ConsAbs);
21             }
22          }
23       }
24    }
25
26    // AGGREGATION
27    LinkedHashSet<MarkStructure> MS_res = aggregate(M_Res);
28
29    return MS_res;
30 }
```

Quelltext B.4: Ausführung von `StartTaskEvents`

Die Methode executeEnd

```
private MarkStructure executeEnd(MarkStructure ms, EndTaskEvent e) {
   LinkedHashSet<ExMark> M_ms = ms.getAllExMarks();

   // INIT
   LinkedHashSet<ExMark> M_Res = new LinkedHashSet<>();

   // ITERATION
   for(ExMark m: M_ms) {
      ExMark m_AnteOcc = executeAnteOccEnd(m, e);
      ExMark m_AnteAbs = executeAnteAbsEnd(m_AnteOcc, e);
      ExMark m_ConsOcc = executeConsOccEnd(m_AnteAbs, e);
      ExMark m_ConsAbs = executeConsAbsEnd(m_ConsOcc, e);
      M_Res.add(m_ConsAbs);
   }

   // AGGREGATION
   LinkedHashSet<MarkStructure> MS_res = aggregate(M_Res);
   if(MS_res.size() != 1) System.err.println("aggregation should actually
       lead to eaxctly one ms, because end events are executed
       deterministic.");
   MarkStructure ms_res = MS_res.toArray(new MarkStructure[] {})[0];

   return ms_res;
}
```

Quelltext B.5: Ausführung von EndTaskEvents

Hilfsmethoden für die Ausführung von `TaskEvents`

```
private boolean matchStart(TaskNode n, StartTaskEvent startEvent) {
   if( n.getType().equals(startEvent.getType()) )
      return true;
   return false;
}

private boolean matchEnd(TaskNode n, EndTaskEvent endEvent) {
   if( n.getType().equals(endEvent.getType()) )
      return true;
   return false;
}

private boolean expectedEnd(APatternExMark m, TaskNode n, EndTaskEvent
    endEvent) {
   AStateMark<?> sm = m.getStateMarkFor(n);
   if(sm instanceof TaskStateMark) {
      TaskStateMark tsm = (TaskStateMark) sm;
      StartTaskEvent startEvent = tsm.getStartEvent();
      if(startEvent != null) {
         TaskIdentifier sid = startEvent.getTaskIdentifier();
         TaskIdentifier eid = endEvent.getTaskIdentifier();
         if(sid.equals(eid))
            return true;
      }
   }
   return false;
}
```

Quelltext B.6: Hilfsmethoden zur Bestimmung der zum Event passenden Knoten

Die Methode exAnteNodesStart

```
1  private LinkedHashSet<TaskNode> exAnteNodesStart(AnteExMark anteExMark,
2        StartTaskEvent e, boolean occ) {
3     LinkedHashSet<TaskNode> res = new LinkedHashSet<>();
4     Pattern checkPattern = occ ? Pattern.AO : Pattern.AA;
5     for(TaskNode tn : getAllTaskNodes()) {
6        if(tn.getPattern() == checkPattern) {
7           if(executableAnteStart(anteExMark, tn, e))
8              res.add(tn);
9        }
10    }
11    return res;
12 }
13
14 private boolean executableAnteStart(AnteExMark anteExMark, TaskNode n,
15       StartTaskEvent e) {
16    if(matchStart(n, e)) { // i
17       AStateMark<?> sm = anteExMark.getStateMarkFor(n);
18       if(sm instanceof TaskStateMark) {
19          TaskStateMark tsm = (TaskStateMark) sm;
20          if(tsm.isNotStarted()) { // ii
21              // iii -- check state of predecessors
22             if(checkPredNodes(n.getStartPort(), anteExMark, true, true))
23                return true; // i, ii and iii were all true
24          }
25       }
26    }
27    return false;
28 }
```

Quelltext B.7: Bestimmung der relevanten Ante. Task Nodes für ein `StartTaskEvent`

Die Methode `deadConsOcc`

```
1  private <T extends Node> Set<Node> deadConsOcc(ConsExMark consExMark, Set<T>
       Q) {
2     Set<Node> res = new HashSet<Node>();
3     for(T q : Q) {
4        for(Node n : ecrg.predConsOccAll(q)) {
5           AStateMark<?> sm = consExMark.getStateMarkFor(n);
6           if(isNotYetCompleted(sm))
7              res.add(n);
8        }
9     }
10    return res;
11 }
12
13 private boolean isNotYetCompleted(AStateMark<?> sm) {
14    if(sm instanceof TaskStateMark) {
15       TaskStateMark tsm = (TaskStateMark) sm;
16       if(tsm.isNotStarted() || tsm.isStarted()) // NULL or STARTED
17          return true;
18    } else if(sm instanceof MessageStateMark) {
19       MessageStateMark msm = (MessageStateMark) sm;
20       if(msm.isNotStarted())
21          return true;
22    }
23    return false;
24 }
```

Quelltext B.8: Bestimmung nicht-abgeschlossener ConsOcc Vorgängerknoten

Die Methode `cleanupCompletedAbsenceMarks`

```
private void cleanupCompletedAbsenceMarks(ComplianceState cs) {
   Iterator<MarkStructure> it = cs.getMarkStructures().iterator();
   while(it.hasNext()) {
      if(cleanupCompletedAbsenceMarks(it.next()))
         it.remove();
   }
}

private boolean cleanupCompletedAbsenceMarks(MarkStructure ms) {
   Iterator<ExMark> it = ms.getExMarksIterator();
   while(it.hasNext()) {
      ExMark m = it.next();
      AnteExMark aem = m.getAnteExMark();
      for(AStateMark<?> sm : aem.getAllStateMarks()) {
         if(sm instanceof IAbsenceStateMark) {
            IAbsenceStateMark asm = (IAbsenceStateMark) sm;
            if(asm.isNotExecutedAndHasCompletedDescendants()) {
               return true; // remove whole ms
            }
         }
      }
      ConsExMark cem = m.getConsExMark();
      for(AStateMark<?> sm : cem.getAllStateMarks()) {
         if(sm instanceof IAbsenceStateMark) {
            IAbsenceStateMark asm = (IAbsenceStateMark) sm;
            if(asm.isNotExecutedAndHasCompletedDescendants()) {
               it.remove(); break; // remove m
            }
         }
      }
   }
   if(ms.getAllExMarks().size() == 0)
      return true;
   return false;
}
```

Quelltext B.9: Bereinigung inkonsistenter Markierungen für Absence Nodes

Die Methode `finalize`

```
private void finalize(MarkStructure ms) {
   for(ExMark m : ms.getAllExMarks()) {
      AnteExMark aem = m.getAnteExMark();
      markEnd(aem);
      ConsExMark cem = m.getConsExMark();
      markEnd(cem);
   }
}

private void markEnd(APatternExMark pem) {
   for(AStateMark<?> sm : pem.getAllStateMarks())
      if(isNotYetCompleted(sm))
         sm.setNotExecuted();
}
```

Quelltext B.10: Finalisierung nicht-abgeschlossener Ausführungsmarkierungen

Die Methode `checkSequence`

```
private void checkSequence(ExMark m) {
   for(SequenceFlowConnector sfc : getSequenceConnectors()) {
      PointInTime pit1 = getPointInTime(m, sfc, true);
      PointInTime pit2 = getPointInTime(m, sfc, false);
      if(getTime(pit1) <= getTime(pit2)) {
         if(sfc.isConditionAlwaysTrue())  continue;
         else {
            if( !sfc.getTimeRelation().evaluate(pit1, pit2) )
               addEdgeMark(sfc, "time distance condition is violated", m);
         }
      } else {
         addEdgeMark(sfc, "sequence order violated", m);
      }
   }
}
```

Quelltext B.11: Überprüfung aller `SequenceFlowConnectors`

Die Methode `evaluate(ms)`

```
1  private void evaluate(MarkStructure ms) {
2     if(ms.isFinal())
3        if(!ms.isActivatable())
4           return;
5
6     if(ms.isActivatable()) {
7        if(ms.getAnteExMark().isSatisfied())
8           ms.setActivated();
9        else if(ms.getAnteExMark().isViolated())
10          ms.setDeactivated();
11       else
12          System.err.println("evaluate: AnteExMark is not final");
13    }
14
15    if(ms.isActivated()) {
16       boolean match = false;
17       for(ConsExMark cem : ms.getConsExMarks()) {
18          if(cem.isSatisfied()) {
19             match = true;
20             break;
21          }
22       }
23
24       if(match)
25          ms.setSatisfied();
26       else
27          ms.setViolated();
28    }
29
30    ms.setFinal();
31 }
```

Quelltext B.12: Bewertung einer `MarkStructure`

Abbildungsverzeichnis

2.1. CRG Knotentypen 11
2.2. CRG Relationen 12
2.3. CRG Datenbedingungen und Knoteneigenschaften 13
2.4. Beispiel einer CRG 14
2.5. Antecedence & Consequence-Pattern einer CRG 14
2.6. Zuordnung von Ablaufprotokollen 15
2.7. Details & Unterschiede bei der CRG Modellierung 16
2.8. Ausführung und Markierung einer CRG 20
2.9. eCRG Elemente für die Prozess-Perspektive 24
2.10. eCRG Elemente für die Interaktions-Perspektive 25
2.11. eCRG Elemente für die Zeit-Perspektive 26
2.12. eCRG Elemente für die Daten und Ressourcen-Perspektive 27
2.13. Beispiel einer eCRG mit genauerer Spezifizierung der Resource Nodes . 29
2.14. eCRG Anwendungsbeispiel 30

3.1. Ausführungszustände von eCRG Knoten 33
3.2. Beispiel einer ExMark 34
3.3. Beispiel einer MarkStructure 35
3.4. Hierarchie der eCRG Knoten 37
3.5. Ausführung eines Start Events über eine Antecedence Task Node 41
3.6. Ausführung von End Events bei Task Nodes 44
3.7. Bereinigung nicht ausgeführter Absence Markierungen 46
3.8. Ausführung einer Consequence Message Node 48

3.9. Anwendung der Ausführungsregeln für Task Nodes 53
3.10. Anwendung der Ausführungsregeln für Message & Data Nodes 57
3.11. Anwendung der Ausführungsregeln für Resource Nodes 60

4.1. Klassendiagramm: Prozessumgebung 65
4.2. Klassendiagramm: Conditions & Relations 66
4.3. Klassendiagramm: Prozess-Ablaufprotokolle 68
4.4. Klassendiagramm: eCRG 70
4.5. Klassendiagramm: Markierungen 71
4.6. Mockup der grafischen Oberfläche 73

5.1. Datenmodell von JGraphX 78
5.2. Die Komponenten der grafischen Benutzeroberfläche 82
5.3. Ergebnis der Ausführung eines Trace über eine eCRG 83
5.4. Symbole zur Veranschaulichung der Markierungszustände 84
5.5. Markierungen der eCRG Knoten 85
5.6. Anzeige der Ausführungsmarkierung per Tooltip und Konsole 86
5.7. Checkbox für Anzeige der Gesamtansicht 86
5.8. Dialog mit einer Gesamtansicht der erreichten Markierungszustände . . . 87
5.9. Dialog für die Anzeige eines Trace 88
5.10. Ausgabe einer eCRG auf der Anzeigefläche 88
5.11. Kontextmenü der Konsole und Funktionen der Menüleiste 89
5.12. Darstellung der verschiedenen eCRG Symbole 90
5.13. Beispiel einer eCRG mit umfangreicher Ressourcen-Perspektive 108

A.1. Startansicht 120
A.2. Schritt 1 – Auswahl des Trace und der eCRG 121
A.3. Schritt 2 – Start der Ausführung & Bewertung 122
A.4. Schritt 3 – Ansicht der erreichten Markierungszustände 123

Tabellenverzeichnis

2.1. Compliance Zustände von verschiedenen Ablaufprotokollen 17

3.1. Liste der unterstützten Events 32
3.2. Startvoraussetzungen für eCRG Knoten 38

4.1. Die implementierenden Unterklassen von Node, Edge und Attachment . . 70

5.1. Bedeutung der verschiedenen Symbole für die Markierungszustände . . . 84
5.2. Bedeutung der verschiedenen Knotenmarkierungen 85

Quelltextverzeichnis

5.1. JGraphX HelloWorld Beispiel 80
5.2. Ausschnitt aus dem Quellcode für die Registrierung und Deklaration der selbst-definierten Pfeilspitzen 93
5.3. Die Klasse myHierarchicalLayout 96
5.4. Die Methode zum Layouten des Graphen 97
5.5. Die einzige öffentliche Methode der Klasse Verifier 98

B.1. Originalcode der DataShape Klasse 126
B.2. Die Klasse PortFactory 129
B.3. Ausführung und Bewertung eines Trace über eine eCRG 131
B.4. Ausführung von StartTaskEvents 133
B.5. Ausführung von EndTaskEvents 134
B.6. Hilfsmethoden zur Bestimmung der zum Event passenden Knoten 135
B.7. Bestimmung der relevanten Ante. Task Nodes für ein StartTaskEvent . . 136
B.8. Bestimmung nicht-abgeschlossener ConsOcc Vorgängerknoten 137
B.9. Bereinigung inkonsistenter Markierungen für Absence Nodes 138
B.10. Finalisierung nicht-abgeschlossener Ausführungsmarkierungen 139
B.11. Überprüfung aller SequenceFlowConnectors 139
B.12. Bewertung einer MarkStructure 140

Literaturverzeichnis

[ALS11] Accorsi, Rafael ; Lowis, Lutz ; Sato, Yoshinori: Automated Certification for Compliant Cloud-based Business Processes. In: *Business & Information Systems Engineering* 3 (2011), Nr. 3, S. 145–154

[AWW11] Awad, Ahmed ; Weidlich, Matthias ; Weske, Mathias: Visually Specifying Compliance Rules and Explaining their Violations for Business Processes. In: *Journal of Visual Languages & Computing* 22 (2011), Nr. 1, S. 30–55

[FSWS10] Feja, Sven ; Speck, Andreas ; Witt, Sören ; Schulz, Marcel: Checkable Graphical Business Process Representation. In: *Proceedings of the 14th East European Conference on Advances in Databases and Information Systems (ADBIS 2010)*, Springer, Sept. 2010 (LNCS 6295), S. 176–189

[GHSW09] Governatori, Guido ; Hoffmann, Jörg ; Sadiq, Shazia ; Weber, Ingo: Detecting Regulatory Compliance for Business Process Models through Semantic Annotations. In: *Business Process Management Workshops*. Springer, 2009 (LNBIP 17), S. 5–17

[KKdV10] Kokash, Natallia ; Krause, Christian ; de Vink, Erik P.: Time and Data-Aware Analysis of Graphical Service Models in Reo. In: *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)*, IEEE, Sept. 2010, S. 125–134

[KR11] Knuplesch, David ; Reichert, Manfred: Ensuring Business Process Compliance Along the Process Life Cycle / University of Ulm. 2011 (UIB-2011-06). – Technical Report

[KRL+13a] KNUPLESCH, David ; REICHERT, Manfred ; LY, Linh Thao ; KUMAR, Akhil ; RINDERLE-MA, Stefanie: On the Formal Semantics of the Extended Compliance Rule Graph / University of Ulm. 2013 (UIB-2013-05). – Technical Report

[KRL+13b] KNUPLESCH, David ; REICHERT, Manfred ; LY, Linh Thao ; KUMAR, Akhil ; RINDERLE-MA, Stefanie: Visual Modeling of Business Process Compliance Rules with the Support of Multiple Perspectives. In: *Proceedings of the 32th International Conference on Conceptual Modeling (ER 2013)*, Springer, November 2013 (LNCS 8217), S. 106–120

[KYC13] KUMAR, Akhil ; YAO, Wen ; CHU, Chao-Hsien: Flexible Process Compliance with Semantic Constraints Using Mixed-Integer Programming. In: *INFORMS Journal on Computing* 25 (2013), Nr. 3, S. 543–559

[LKRM+11] LY, Linh Thao ; KNUPLESCH, David ; RINDERLE-MA, Stefanie ; GÖSER, Kevin ; PFEIFER, Holger ; REICHERT, Manfred ; DADAM, Peter: SeaFlows Toolset - Compliance Verification Made Easy for Process-Aware Information Systems. In: *Information Systems Evolution*. Springer, 2011 (LNBIP 72), S. 76–91

[LMX07] LIU, Ying ; MULLER, Samuel ; XU, Ke: A static compliance-checking framework for business process models. In: *IBM Systems Journal* 46 (2007), Nr. 2, S. 335–361

[LRMD10] LY, Linh Thao ; RINDERLE-MA, Stefanie ; DADAM, Peter: Design and Verification of Instantiable Compliance Rule Graphs in Process-Aware Information Systems. In: *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE 2010)*, Springer, Juni 2010 (LNCS 6051), S. 9–23

[LRMKD11] LY, Linh Thao ; RINDERLE-MA, Stefanie ; KNUPLESCH, David ; DADAM, Peter: Monitoring Business Process Compliance Using Compliance Rule Graphs. In: *Proceedings of the 19th International Conference on Cooperative Information Systems (CoopIS 2011)*, Springer, Oktober 2011 (LNCS 7044), S. 82–99

[Ly13] LY, Linh Thao: *SeaFlows – A Compliance Checking Framework for Supporting the Process Lifecycle*, University of Ulm, Dissertation, Mai 2013

[NS07] NAMIRI, Kioumars ; STOJANOVIC, Nenad: Pattern-Based Design and Validation of Business Process Compliance. In: *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, Springer, November 2007 (LNCS 4803), S. 59–76

[RFvA12] RAMEZANI, Elham ; FAHLAND, Dirk ; VAN DER AALST, Wil M.P.: Where Did I Misbehave? Diagnostic Information in Compliance Checking. In: *Business Process Management*. Springer, 2012 (LNCS 7481), S. 262–278

[RMLD08] RINDERLE-MA, Stefanie ; LY, Linh Thao ; DADAM, Peter: Business Process Compliance. In: *EMISA Forum* (2008), August, S. 24–29

[Sem13] SEMMELRODT, Franziska: *Modellierung klinischer Prozesse und Compliance Regeln mittels BPMN 2.0 und eCRG*, University of Ulm, Master Thesis, November 2013

[SKR14] SEMMELRODT, Franziska ; KNUPLESCH, David ; REICHERT, Manfred: Modeling the Resource Perspective of Business Process Compliance Rules with the Extended Compliance Rule Graph. In: *Proceedings of the 15th International Working Conference on Business Process Modeling, Development, and Support (BPMDS 2014)*, Springer, Juni 2014 (LNBIP 175). – Accepted for Publication

[TEvHP12] TURETKEN, Oktay ; ELGAMMAL, Amal ; VAN DEN HEUVEL, Willem-Jan ; PAPAZOGLOU, Michael P.: Capturing Compliance Requirements: A Pattern-Based Approach. In: *IEEE Software* 29 (2012), Mai, Nr. 3, S. 28–36

[UMJ14] JGRAPH LTD. (Hrsg.): *JGraphX User Manual*. `http://jgraph.github.io/mxgraph/docs/manual_javavis.html`: JGraph Ltd., Mai 2014

Name: Hannes Beck Matrikelnummer: 665057

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Hannes Beck