Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis Nr. 3386

LTL- Erfüllbarkeitsprüfung für inkrementelle Entwicklung von Geschäftsprozessen

Alexej Burkow

Studiengang:	Wirtschaftsinformatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Daniel Schleicher
begonnen am:	01.03.2012
beendet am:	31.08.2012
CR-Klassifikation:	D.2.4, D.2.5, D.3.2, H.4.1, H.5.3

Abstract

Heutige Unternehmen stehen einer immer größer werdenden Menge an internen und externen Regelwerken, den Compliance-Regeln, gegenüber. Ihre Konsistenz muss bei der Entwicklung von organisationsübergreifenden Geschäftsprozessen sichergestellt werden. Die vorliegende Arbeit beschäftigt sich mit der automatischen Durchsetzung von Compliance in Geschäftsprozessmodellen.

In einer vorhergehenden Arbeit wurde der webbasierte BPMN-Editor Oryx um die Überprüfung der Einhaltung von Compliance-Regeln in Prozessmodellen mittels Model-Checking erweitert. Die Prozessmodelle werden in sogenannte Compliance-Scopes aufgeteilt, die mit Compliance-Regeln in der linearen temporalen Logik (LTL) annotiert sind und die selbst weitere Compliance-Scopes enthalten können. In dieser Arbeit wird der Prototyp so weiterentwickelt, dass die verschachtelten Compliance-Regeln automatisch auf Konsistenz geprüft werden.

Dabei basiert die Lösung auf einem vorhandenen Konzept der Konsistenzprüfung verschachtelter Compliance-Regeln, in dem die Compliance-Regeln als aussagenlogische Formeln formuliert werden. Diese Regeln werden an die enthaltenen Prozessbereiche rekursiv weitergegeben und mit ihren Compliance-Regeln auf Erfüllbarkeit geprüft. Neben der Übertragung dieses Konzeptes auf die LTL und den Prototyp wird die Gültigkeitsprüfung von Compliance-Regeln integriert. Es wird ein Ansatz zur Erkennung von den zur Weitergabe relevanten Teilregeln entwickelt. Dieser Ansatz basiert auf der Analyse der den LTL-Regeln entsprechenden Büchi-Automaten. Des Weiteren baut die Entscheidung zur Weitergabe auf den Teilergebnissen des Model-Checking auf. Daher werden das Model-Checking und die Konsistenzprüfung zu einer gemeinsamen Compliance-Prüfung kombiniert. Im Ausblick wird auf die Weiterentwicklungsmöglichkeiten der Lösung eingegangen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

1 Einleitung

Aufgrund des technologischen Fortschritts und der globalen Tätigkeit heutiger Unternehmen können neue Marktchancen in neuen Geschäftsfeldern entdeckt werden [Bur04]. Doch gleichzeitig werden auch die Wettbewerbsregeln immer komplexer. Die Unternehmen stehen einer immer größer werdenden Menge an externen und internen Regelwerken, wie z. B. den internationalen Rechnungslegungsvorschriften IFRS oder der Qualitätsmanagement-Normenreihe ISO 9000, gegenüber. Dabei sind die Regularien einem ständigem Wandel unterzogen. So wurde z. B. Anfang 2012 im Zuge des technischen Fortschritts eine neue EU-Datenschutzrichtlinie [www12c] vorgeschlagen und ab 2013 werden aufgrund der Finanzkrise die neuen Eigenkapitalregeln für Banken nach Basel III in Kraft treten [www12b]. Insbesondere im Finanzsektor ist die stetige Zunahme an Regularien deutlich erkennbar [PR10]. Zudem wird z. B. in [www12d] eine „erhebliche Überregulierung“, die zu schlechteren Kundenbeziehungen und höheren Personalkosten führt, diskutiert.

Die Einhaltung von Regeln und Gesetzen wird als *Compliance* bezeichnet. Die Nichterfüllung von *Compliance-Regeln* kann von wirtschaftlichen Einbußen über Imageverluste bis hin zu Geld- und Freiheitsstrafen führen [MMW07]. Daher wird im Rahmen des *Compliance Managements* die sichere und effiziente Erfüllung der internen und externen Regeln angestrebt.

Erschwerend kommt hinzu, dass sich verschiedene Regeln oft auf gleiche Geschäftsprozesse beziehen und dabei nicht frei von *Widersprüchen* sind [KSMP07]. Die Entdeckung und Beseitigung solcher Widersprüche ist einer der Hauptaspekte bei der Umsetzung von Compliance [JR10]. Beispielsweise können die Inkonsistenzen länderspezifisch sein. So müssen die Unternehmensniederlassungen in verschiedenen Ländern bei der Speicherung personenbezogener Daten unterschiedliche Datenschutzbestimmungen befolgen [JR10].

Aufgrund ständiger Veränderungen in den Regularien sind manuelle Compliance-Prüfungen, z. B. durch Audits, sowie die automatisierte Erkennung von Compliance-Verletzungen, z. B. anhand von Log-Dateien, oft unzureichend [Sac08]. Denn zum Zeitpunkt der Fehlerentdeckung ist oft bereits ein Schaden verursacht worden. Dagegen lassen sich mit dem Compliance „*by design*“-Ansatz [SGN07] viele Regelverletzungen bereits während der Entwicklung von Geschäftsprozessmodellen automatisch vermeiden. Dazu können die Regeln in einer logischen Sprache spezifiziert und den Prozessmodellen zugewiesen werden. Die Verifikation der Modelle gegen ihre Spezifikationen kann anschließend mittels des seit den 1980-er Jahren in der Hard- und Softwareentwicklung erforschten *Model-Checking* [CES86] vollautomatisch erfolgen.

Dieses Verfahren wird erst seit einigen Jahren in der Welt der Geschäftsprozesse eingesetzt [FPR06, RMLD08]. Bei der Entwicklung von Geschäftsprozessen sind oft mehrere Unternehmen beteiligt [SALS10], die weltweit verteilt sein können. Beispielsweise entstehen durch das Outsourcing unternehmensübergreifende Geschäftsprozesse. Die Unternehmens-bereiche und Abteilungen der beteiligten Unternehmen müssen dabei sowohl die Vorgaben ihrer übergeordneten Bereiche als auch ihre internen Geschäftsregeln sowie branchenspezifische und standortabhängige Regelungen beachten. Insgesamt ergibt sich dadurch ein hoher Kommunikationsaufwand [KFB04], der aufgrund der Entfernung sowie der zeitlichen und sprachlichen Unterschiede zu Missverständnissen und damit zu *Inkonsistenzen zwischen Teil- und Hauptprozessen* führen kann.

Eine Inkonsistenz liegt dabei z. B. vor, wenn in einem Teilprozess eine Aktivität spezifiziert wird, die laut der Regel des übergeordneten Prozessmodells nicht erlaubt ist. Diesem Problem wird in [SALS10] mit dem Konzept des *inkrementellen Entwicklungsprozesses* von Geschäftsprozessen begegnet. Das Konzept ermöglicht es Prozessdesignern aus verschiedenen Organisationen einen Gesamtprozess mit konsistenten Compliance-Regeln zu entwickeln. Die Modellierung wird durch eine Aufteilung des Prozessmodells so unterstützt, dass die zu beachtenden Regeln automatisch auf Widerspruchsfreiheit geprüft werden. Der inkrementelle Entwicklungsprozess stellt sicher, dass die Compliance-Regeln der Unterprozesse nicht die Regeln der Prozesse verletzen, in die sie eingebettet sind.

In dieser Arbeit wird das oben genannte Konzept des *inkrementellen Entwicklungsprozesses* umgesetzt. Im Gegensatz zu der in [SALS10] verwendeten Aussagenlogik erfolgt die Spezifikation der Compliance-Regeln in der *linearen temporalen Logik* (LTL). Die LTL ermöglicht zeitliche Aspekte, wie z. B. Reihenfolgen oder wiederkehrende Aktivitäten auszudrücken. Daher werden in dieser Arbeit erweiterte Problemstellungen diskutiert sowie ein entsprechender Lösungsansatz entwickelt und implementiert.

1.1 Aufgabenstellung

Das Ziel dieser Arbeit ist die Umsetzung des *inkrementellen Entwicklungsprozesses* aus [SALS10], wobei statt der dort betrachteten Aussagenlogik als Spezifikationssprache für Compliance-Regeln die *lineare temporale Logik* verwendet wird. Dazu soll auf einem Prototyp aufgebaut werden, der in [Gro10] um sogenannte Compliance-Scopes und das Model-Checking erweitert wurde. Die Compliance-Scopes stellen dabei abgegrenzte Prozessbereiche mit zugewiesen Compliance-Regeln dar.

Die bestehende prototypische Implementierung soll dahingehend erweitert werden, dass die Compliance-Regeln von verschachtelten Compliance-Scopes (siehe Abbildung 1.1) automatisch auf Konsistenz geprüft werden können. Dazu sollen die Compliance-Regeln von verschachtelten Compliance-Scopes in geeigneter Art verknüpft und auf Erfüllbarkeit geprüft werden. Zu diesem Zweck ist ein geeigneter SAT-Solver einzubinden. Der Prozessdesigner soll informiert werden, falls die Compliance-Regeln im Widerspruch zu den Compliance-Regeln der äußeren Compliance-Scopes stehen und es damit nicht möglich wird, einen regelkonformen Prozess zu modellieren. Des Weiteren sollen Optimierungsmöglichkeiten untersucht und ggf. implementiert werden.

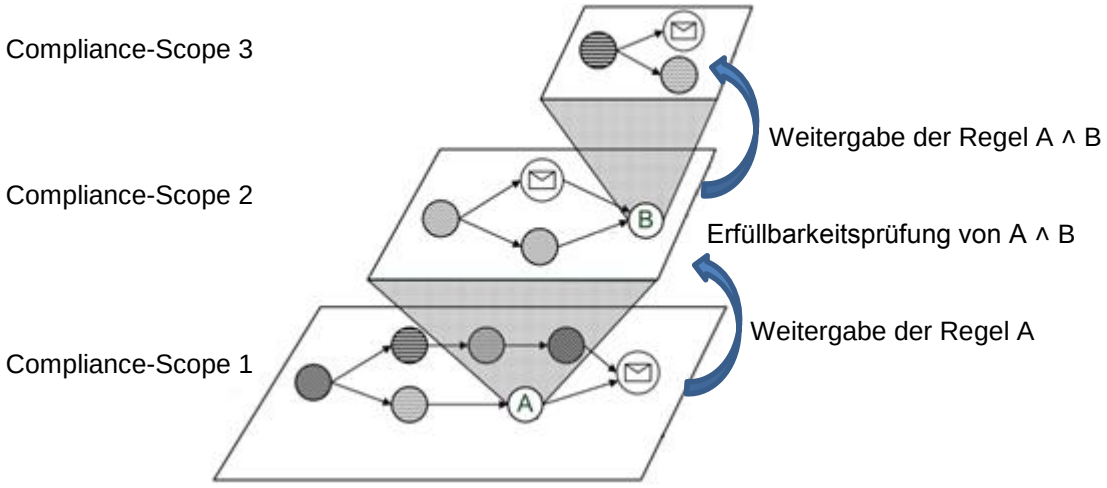


Abbildung 1.1: Verschachtelte Compliance-Scopes

1.2 Aufbau der Arbeit

Kapitel 1 enthält die Beschreibung der Aufgabenstellung und des Aufbaus der Arbeit sowie eine Einführung in die Problemstellung.

Im Kapitel 2 erfolgt die Einordnung dieser Arbeit im Geschäftsprozessmanagement (GPM). Dazu wird die Notwendigkeit der frühzeitigen Durchsetzung von Compliance-konformen Geschäftsprozessmodellen erläutert und im GPM-Lebenszyklus eingeordnet. Des Weiteren werden der webbasierten Editor Oryx und die verwendete Modellierungssprache BPMN vorgestellt. In nachfolgenden Unterkapiteln werden notwenige Grundlagen behandelt, auf die im weiteren Verlauf der Arbeit zurückgegriffen wird. Dies sind die lineare temporale Logik, das Model-Checking und die Erfüllbarkeitsprüfung.

Im Kapitel 3 werden die Konzepte und Vorarbeiten vorgestellt, auf denen diese Arbeit aufbaut. Dazu gehören die Plausibilitätsprüfungen für Prozessspezifikationen, der inkrementelle Entwicklungsprozess sowie der Oryx-Prototyp, der in der vorhergehenden Arbeit um Compliance-Scopes und das Model-Checking erweitert wurde.

Im Kapitel 4 wird das Konzept der inkrementellen Entwicklung von Compliance-konformen Geschäftsprozessen auf die LTL übertragen. Dazu wird eine Definition von positiven und negativen Teilregeln im Rahmen der LTL eingeführt und die temporalen Gültigkeitsbereiche von LTL-Formeln diskutiert. Anschließend wird die zu implementierte Konsistenzprüfung von verschachtelten Compliance-Regeln beschrieben.

Im Kapitel 5 wird die Umsetzung des Konzeptes beschrieben. Dabei wird zunächst ein architektonischer Überblick über die veränderten und hinzugefügten Komponenten des Prototyps gegeben. Anschließend werden die Details der Implementierung im Front- und Backend erläutert. Des Weiteren wird die Erweiterung des Model-Checking beschrieben, die das Model-Checking mit LTL-Formeln mit den Operatoren *Globally* und *Until* ermöglicht.

Im Kapitel 6 ist eine Zusammenfassung dieser Arbeit zu finden. Das Kapitel 7 bietet einen Ausblick zur Erweiterung des Konzeptes und des Prototyps.

2 Grundlagen

In diesem Kapitel wird das Thema der vorliegenden Arbeit im Geschäftsprozessmanagement eingeordnet sowie notwenige Grundlagen behandelt, auf die im weiteren Verlauf der Arbeit zurückgegriffen wird. Dies sind die lineare temporale Logik, das Model-Checking und die Erfüllbarkeitsprüfung.

2.1 Geschäftsprozessmanagement

Unter Geschäftsprozessmanagement (GPM) wird ein Führungskonzept zur zielgerichteten Steuerung der Geschäftsprozesse eines Unternehmens [SS08] verstanden. Ein Geschäftsprozess ist eine Verknüpfung wertschöpfender Aktivitäten, die zusammen zur Erfüllung eines wirtschaftlichen Ziels führen. Ein solches Ziel ist in der Regel die Erfüllung eines Kundenbedürfnisses. Es können primäre und sekundäre Geschäftsprozesse unterschieden werden. Die primären Geschäftsprozesse, wie z. B. Produktions- und Marketingprozesse, haben einen direkten Einfluss auf die Wertschöpfung und die Wettbewerbsfähigkeit. Die sekundären Geschäftsprozesse, wie z. B. Personalbeschaffung und IT-Support haben eine unterstützende Funktion [SS08].

Die primären Aufgaben des GPM haben einen strategischen Charakter. Auf der strategischen Ebene werden die wettbewerbsrelevanten Geschäftsprozesse auf die strategischen Unternehmensziele ausgerichtet und durch Kennzahlensysteme kontrolliert. Das GPM hat dabei einen maßgeblichen Einfluss auf die Organisationsstruktur eines Unternehmens [SS08]. Auf der operativen Ebene werden die Geschäftsprozesse strukturiert, ausgeführt und laufend optimiert. Zur Automatisierung von Geschäftsprozessen werden Workflow Management Systeme (WMS) eingesetzt. Mit *Workflow* wird ein „vollständig oder teilweise automatisierter Geschäftsprozess“ bezeichnet [HN09].

Der GPM-Lebenszyklus

Die mit einem Geschäftsprozess oder Workflow verbunden Tätigkeiten können im sogenannten GPM-Lebenszyklus zusammengefasst werden [Joc10] (siehe Abbildung 2.1). Ausgehend von den aus der Geschäftsstrategie abgeleiteten Geschäftsanforderungen wird ein neues Prozessmodell, z. B. in der Sprache *Business Process Model and Notation* (BPMN), erstellt. Dabei stellen die sogenannten *Key Performance Indicators* (KPIs) nicht-funktionale Leistungsparameter bezüglich Zeit, Kosten, Qualität und Flexibilität dar. In der Implementierungs-Phase wird das Modell in eine ausführbare Sprache, wie die *Business Process Execution Language* (BPEL), übersetzt. Danach erfolgt die Bereitstellung in der Produktivumgebung und anschließend die Ausführung, wobei in der Regel mehrere Instanzen des Prozesses entstehen. Während der Ausführung werden die KPIs laufend gemessen und protokolliert. Auf Grundlage der im nächsten Schritt erfolgenden Analyse der Messdaten und des Abgleichs mit den strategischen Zielen wird das Prozessmodell verbessert.

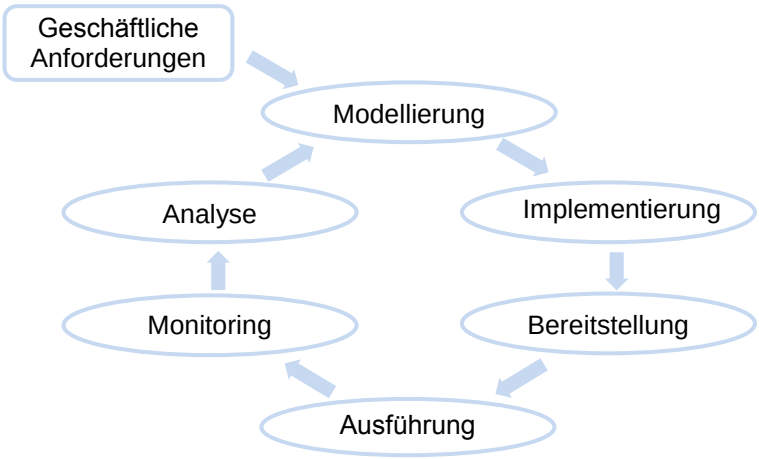


Abbildung 2.1: GPM-Lebenszyklus, in Anlehnung an [Joc10]

2.1.1 Compliance

Der englische Begriff „*compliance*“ bedeutet so viel wie Erfüllung oder Einhaltung. Unter Compliance wurde ursprünglich die Einhaltung gesetzlicher Regelungen zum Anlegerschutz auf dem Kapitalmarkt verstanden. Heute umfasst dieser Begriff die Konformität zu allen für ein Unternehmen relevanten internen und externen Regularien, die im Folgenden *Compliance-Regeln* oder *Regeln* bezeichnet werden. Externe Compliance-Regeln sind beispielsweise Gesetzte zur Rechnungslegung und zum Datenschutz. Intern handelt es sich neben Unternehmensrichtlinien auch um die Einhaltung gesellschaftlicher Werte [Que11]. Insbesondere die Compliance zu aktuellen Nachhaltigkeitstrends, die ökonomische, ökologische und soziale Aspekte umfassen, wird zunehmend auf freiwilliger Basis integriert [www10a]. Dadurch gewinnen Unternehmen Vertrauen von Seiten der Kunden, Geschäftspartner und Mitarbeiter [Kal12].

Inkonsistenzen zwischen Compliance-Regeln

Damit Gesetze und Normen von unterschiedlichen Unternehmen angewendet werden können, werden sie nicht genau formalisiert, sondern abstrakt gehalten [KSMP07]. Wenn verschiedene Regelwerke sich auf gleiche Geschäftsprozesse beziehen, entstehen oft Widersprüche die entdeckt und gelöst werden müssen [KSMP07, JR10]. Das Spannungsfeld zwischen internen und externen Regeln ist in Abbildung 2.2 dargestellt. Beispielsweise gilt nach dem Datenschutzgesetz das Selbstbestimmungsrecht über das persönliche E-Mail-Postfach. Andererseits schreiben andere Gesetzte die Archivierungs- und Aufbewahrungspflicht steuerrelevanter Daten vor. Das heißt, der Zugriff zu den Postfächern muss in bestimmten Situationen auch für andere Personen gewährleistet werden [Rös09].

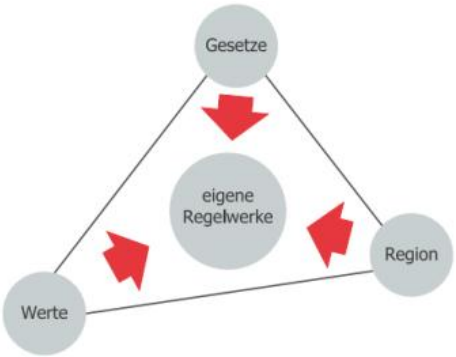


Abbildung 2.2: Interne und externe Compliance-Regeln [JR10]

Des Weiteren müssen eventuelle Abhängigkeiten zwischen den Regelwerken beachtet werden. So können Inkonsistenzen aufgrund von Änderungen in den Regelwerken entstehen. Bei Änderung der internen Regeln müssen die externen Regeln überprüft werden und umgekehrt [JR10].

Bedeutung und Umsetzung der Compliance

Die Bedeutung der Compliance ist in den letzten Jahren stark gestiegen [PR10] und wird zunehmend als fester Bestandteil in der Organisationsstruktur großer Unternehmen integriert [EKA11]. Die Einhaltung interner und externer Regeln kann erfolgskritisch für ein Unternehmen sein, da ihre Missachtung von Imageverlusten bis hin zu Geld- und Freiheitsstrafen für die verantwortlichen Geschäftsführer und Vorstände zur Folge haben [MMW07].

Die Umsetzung der Compliance wird im Wesentlichen von der Informationstechnik (IT) unterstützt. Beispielsweise helfen sogenannte Compliance Management-Systeme (CMS) „Compliance in allen relevanten Geschäftsprozessen des Unternehmens sicherzustellen“ [HE10]. Da sich die internen und externen Anforderungen auf Geschäftsprozesse beziehen, besteht ein enger Zusammenhang mit dem Geschäftsprozessmanagement [WK06]. Dabei ist eine hohe Flexibilität der IT-Infrastruktur eine Voraussetzung um die Geschäftsprozesse an Umfeldveränderungen anpassen zu können.

2.1.1.1 Compliance „by detection“

Grundsätzlich können zwei Ansätze zur Umsetzung von Compliance unterschieden werden: *Compliance „by detection*“ und *Compliance „by design*“. Bei dem *Compliance „by detection*“-Ansatz [Sac08] werden die Regel-Verletzungen durch eine ex-post Analyse aufgedeckt. Das können z. B. Audits oder die Analyse von Log-Dateien ausgeführter Workflows (siehe Abschnitt 2.1 und Phase des Monitoring und der Analyse in Abbildung 2.4) sein. Bei diesem Ansatz besteht das Problem, dass die Regel-Verletzungen erst entdeckt werden, nachdem sie Schaden verursacht haben. Außerdem wird eine vollständige konsistente Aufzeichnung aller tatsächlichen Aktivitäten vorausgesetzt, was nicht immer möglich ist.

2.1.1.2 Compliance „by design“

Aus dem Software Engineering ist bekannt, dass Fehler aus einer der Entwicklungs-Phasen (links in Abbildung 2.3]) typischerweise auf derselben Ebene in einer Umsetzungs-Phase (rechts) entdeckt werden. Das bedeutet, dass die Fehlerbehebungskosten, umso höher sind, je früher die Fehler begangen werden. Dabei steigt der Schaden durch unentdeckte Fehler mit der Zeit exponentiell an. Daher sollten Fehler möglichst früh entdeckt werden [LL10].

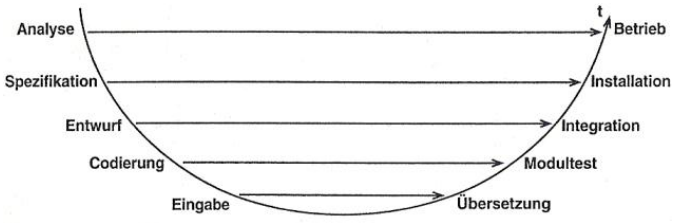


Abbildung 2.3: Ebenen der Fehlerentdeckung [LL10]

Der *Compliance „by design“*-Ansatz [SGN07] vermeidet Fehler indem die Prozessmodelle während der Modellierung auf Einhaltung der Compliance-Regeln verifiziert werden. Der Nachteil dieses Ansatzes ist jedoch, dass nicht mit Sicherheit entschieden werden kann, dass das verifizierte Geschäftsprozessmodell den Regeln entspricht. Der Grund dafür ist, dass nicht immer alle möglichen speziellen Geschäftsvorfälle ex-ante bekannt sind. Im Gegensatz zum „*by detection*“-Ansatz ist jedoch eine Durchsetzung von gewünschtem und eine Verhinderung von ungewünschtem Verhalten möglich. Allerdings ist eine flexible Anpassung an Umfeldveränderungen unmöglich. Daher wird in [Sac08] eine Kombination aus beiden Ansätzen vorgeschlagen.

2.1.2 Einordnung dieser Arbeit im GPM-Lebenszyklus

In dieser Arbeit liegt der Fokus auf dem Compliance „by design“-Ansatz. Die relevanten Phasen des GPM-Lebenszyklus sind in der Abbildung 2.4 hervorgehoben. Bei der Modellierung werden nun zusätzlich die Compliance-Regeln (siehe Abschnitt 2.1.1) berücksichtigt. Während beim „by detection“-Ansatz die Regel-Verletzungen nach der Prozessausführung analysiert werden, wird hier das Prozessmodell vor seiner Implementierung auf mögliche Compliance-Verletzungen untersucht.

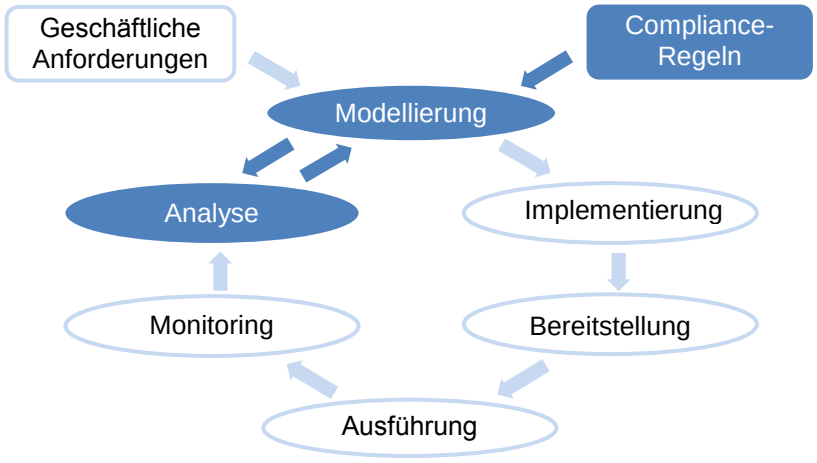


Abbildung 2.4: Einordnung dieser Arbeit im GPM-Lebenszyklus, in Anlehnung an [Joc10]

2.1.3 Business Process Model and Notation

Die Business Process Model and Notation (BPMN) ist eine standardisierte grafische Notation zur Beschreibung von Geschäftsprozessen [wwwf]. Die einheitliche Darstellung von Geschäftsprozessmodellen ermöglicht insbesondere die Zusammenarbeit von Geschäfts-prozessentwicklern. Ursprünglich wurde die Sprache aus der Sicht der Fachverantwortlichen entwickelt. Seit der Version BPMN 2.0 hat die Sprache eine definierte Ausführungssemantik, die es ermöglicht die Prozessmodelle in den Business Prozess Management Systemen (BPMS) auszuführen oder auf andere ausführbare Sprachen wie Business Process Execution Language (BPEL) abzubilden [www11b].

Die grundlegenden grafischen Elemente der BPMN sind Ereignisse, Aktivitäten und Gateways. Einige ihrer Variationen werden anhand des Beispielprozesses in Abbildung 2.5 erläutert. Die Ereignisse werden als Kreise dargestellt. Dies können Start-, Zwischen- oder Endereignisse sein. Ein Zwischenereignis (doppelt umrandet) kann z. B. den Eingang oder Ausgang einer Nachricht bedeuten. Die abgerundeten Rechtecke stellen Aktivitäten dar. Sie können Aufgaben, die auch Tasks genannt werden, oder mit einem Klick auf ein Pluszeichen aufklappbare Teilprozesse darstellen. Rauten mit einem Pluszeichen sind parallele Gateways, die bei Verzweigungen alle ausgehenden Kanten aktivieren. Bei Zusammenführungen warten sie auf alle eingehenden Kanten, bevor sie den ausgehenden Sequenzfluss aktivieren. Rauten mit einem X-Zeichen sind exklusive Gateways. Bei Verzweigungen aktivieren sie genau eine ausgehende Kante. Bei Zusammenführungen warten sie nur auf eine eingehende Kante. Ein Überblick über weitere grafische Elemente der BPMN kann z. B. in [www11a] in Form eines Posters heruntergeladen werden.

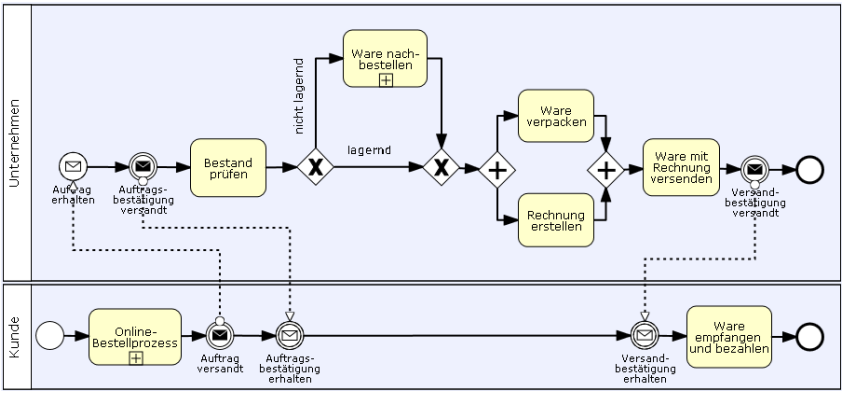


Abbildung 2.5: Beispiel für einen BPMN-Prozess

2.1.4 Der Editor Oryx

Oryx ist ein am Hasso-Plattner-Institut der Universität Potsdam entwickelter webbasierter BPMN-Editor [wwwl]. Der Editor steht unter einer Open-Source-Lizenz zur Verfügung [wwwg] und wurde ursprünglich vor allem für Forschungszwecke entwickelt [DOW08]. Eine kommerzielle Version von Oryx wird von dem Unternehmen Signavio vertreiben [wwwj].

Der Editor ist in einen client- und serverseitigen Bereich aufgeteilt. Die Clientseite wird auch als das Frontend bezeichnet und enthält die in JavaScript programmierte Benutzeroberfläche (siehe Abbildung 2.6). Die Benutzeroberfläche wird im Webbrowser aufgerufen und ist in vier Hauptbereiche unterteilt. Im linken Bereich befinden sich grafische Elemente, die per Drag&Drop in die Zeichenfläche im mittleren Bereich gezogen werden können. Nach dem Markieren eines grafischen Elements können nachfolgende Elemente ausgewählt werden oder im rechten Bereich die Eigenschaften bearbeitet werden. Im oberen Bereich befindet sich die horizontale Toolbar, in der Zusatzfunktionen aufrufbar und durch in Plugin-Konzept integrierbar sind [www10b, Tsc07]. Neben JavaScript werden externe JavaScript-Bibliotheken eingesetzt, wie z.B. Prototype [wwwi], welche die objektorientierte Programmierung und den Datenaustausch zwischen Client und Server erleichtert. Der Datenaustausch erfolgt dabei unter der Nutzung des textbasierten JSON-Formats [wwwb] mittels AJAX [wwwa].

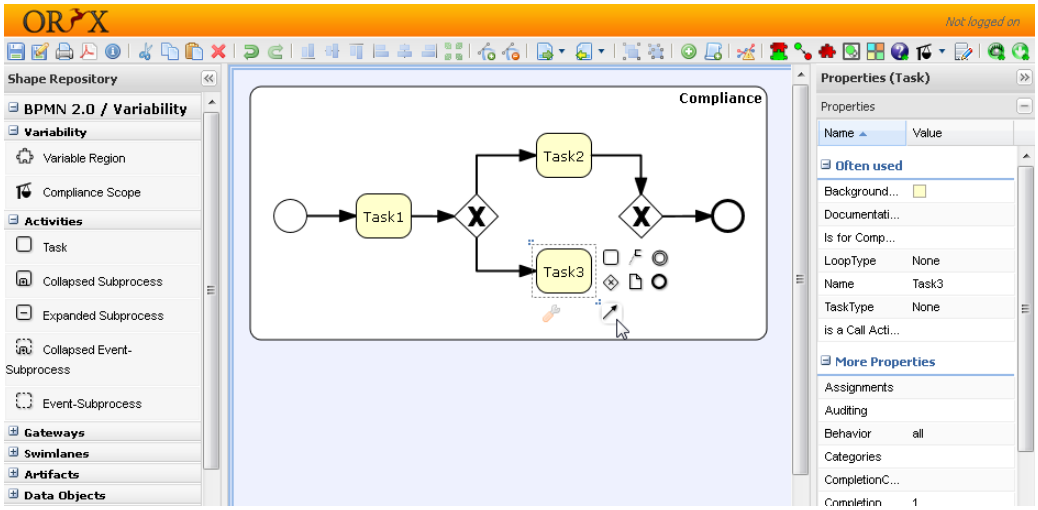


Abbildung 2.6: Benutzeroberfläche des Editors Oryx

Die Serverseite ist das sogenannte Backend, welches unter anderem die Datenhaltung und Anbindung zu anderen Systemen übernimmt [DOW08]. Das Backend ist in Java programmiert. Dabei werden die Anfragen des Frontends mittels der speziellen Java-Klassen, den sogenannten Java Servlets [wwwc], entgegengenommen.

2.2 Temporale Logik

Als Vater der modernen temporalen Logik gilt Arthur Norman Prior, der in den 1950-er Jahren die Grundlagen gelegt hat [www12g, www12h]. Amir Pnueli führte die temporale Logik in den späten 1970-er Jahren in die Informatik ein und erhielt dafür in 1996 den Turing Award [www12f]. Die temporale Logik ermöglicht zeitliche Zusammenhänge zwischen Ereignissen in reaktiven, das heißt mit ihrer Umwelt interagierenden, Systemen wie Kommunikationsprotokollen und Betriebssystemen zu beschreiben [Pnu77, Pnu86]. Heute werden verschiedene Arten temporaler Logik zur Spezifikation funktionaler Eigenschaften von Hardware- und Softwaresystemen, in denen zeitliche Aspekte eine Rolle spielen, eingesetzt [HT10]. Ein neueres Einsatzgebiet, wie in dieser und ähnlichen Arbeiten [FPR06, RMLD08, Gro11] beschrieben, ist die Spezifikation von Geschäftsprozessen.

Im Gegensatz zu aussagenlogischen Formeln, die konstante Werte repräsentieren, beschreiben temporallogische Formeln Sequenzen von Werten. In [DAC98] werden neben einem Muster-System für ausdrückbare Systemeigenschaften auch die in Abbildung 2.7 dargestellten zeitlichen Gültigkeitsbereiche einer temporallogischen Formel analysiert.

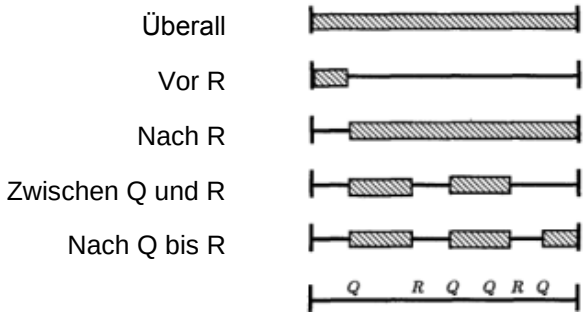


Abbildung 2.7: Gültigkeitsbereiche temporallogischer Formeln, nach [DAC98]

Ein Modell für eine temporallogische Formel ist eine *temporale Struktur* [HT10], die auch *Kripke-Struktur* [Kri71] genannt wird. Eine Kripke-Struktur kann als ein gerichteter Graf visualisiert werden. Die Knoten dieses Grafen repräsentieren Systemzustände, in denen bestimmte aussagenlogische Variablen gelten [HT10]. Die Kanten des Grafen bilden die möglichen Zustandsübergänge.

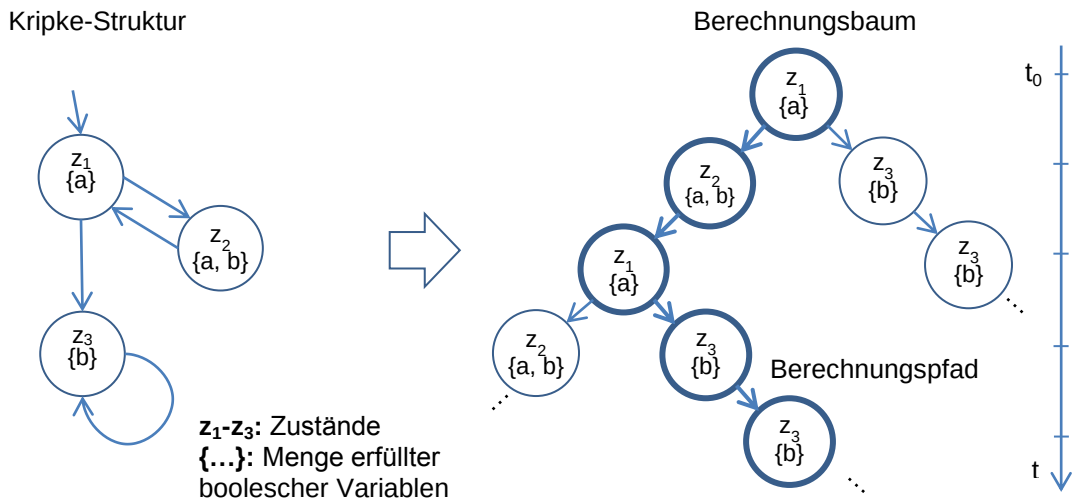


Abbildung 2.8: Kripke-Struktur, Berechnungsbaum, Berechnungspfad, nach [HT10]

Im linken Teil der Abbildung 2.8 ist eine Kripke-Struktur mit drei Zuständen dargestellt, in denen jeweils eine Menge boolescher Variablen angegeben ist, die *wahr* sind. Von dem Startzustand z_1 aus können alle möglichen Zustandssequenzen verfolgt werden, wodurch der im rechten Teil der Abbildung dargestellte *Berechnungsbaum* [HT10] aufgezeichnet werden kann. Die unendlichen Pfade des Berechnungsbaums stellen die möglichen ***Berechnungspfade***, das heißt Sequenzen aktivierter Zustände in einem ausgeführten System, dar. Die Eigenschaften dieser Pfade können durch temporallogische Formeln beschrieben werden. Statt temporalen oder Kripke-Strukturen beschreiben manche Autoren die modellierten Systeme auch als sogenannte *Transitionssysteme* [HR04], die ebenfalls als gerichtete Grafen visualisiert werden können.

2.2.1 Lineare temporale Logik

Im Rahmen dieser Arbeit wird die *lineare temporale Logik* (LTL) verwendet. In der Literatur wird diese Art der temporalen Logik auch als PLTL (*propositional linear temporal logic)* [Eme95] oder PTL (*propositional temporal logic*) [CPP93] bezeichnet. Die LTL ist eine Erweiterung der Aussagenlogik um temporale Operatoren. Mit ihnen ist es möglich die Veränderung der Variablenbelegung im zeitlichen Verlauf zu beschreiben. Diese Logik wird *linear* bezeichnet, weil sie Sequenzen von Systemzuständen beschreibt. Die Zeit ist dabei diskret, was bedeutet, dass jeder Zustandsübergang dem Fortschritt der Zeit um eine Zeiteinheit entspricht [Fis11]. Die formalen Grundlagen der LTL wurden aus der Modallogik übernommen [Fis11]. In der Modallogik ist es ausdrückbar, dass etwas *möglicherweise* (◇-Operator) oder *notwendigerweise* (□-Operator) stattfindet. In der temporalen Logik werden diese Operatoren zeitlich interpretiert.

Definition 2.1 (LTL-Syntax): Wenn a eine atomare Aussage ist, können LTL-Formeln mit Hilfe der Metasprache Backus-Naur-Form (BNF) wie folgt induktiv definiert werden [HRT05, HR04]:

$\varphi ::=$ *true* | *false* | a | $\neg\varphi$ | $(\varphi \rightarrow \varphi)$ | $(\varphi \leftrightarrow \varphi)$ | $(\varphi \wedge \varphi)$ | $(\varphi \vee \varphi)$ | $\bigcirc\ \varphi$ | $\Diamond\ \varphi$ | $\Box\ \varphi$ | $(\varphi\ U\ \varphi)$ | $(\varphi\ W\ \varphi)$ | $(\varphi\ R\ \varphi)$

Aus der obigen Definition ist erkennbar, dass eine LTL-Formel wie eine aussagenlogische Formel entweder zu *true (wahr)* oder *false* (falsch) ausgewertet werden kann und im einfachsten Fall nur aus einem Literal (a) besteht. Die Operatoren ¬, →, ↔, ∧ und ∨ entsprechen den aussagenlogischen Operatoren *Negation, Implikation, Äquivalenz, Konjunktion* und *Disjunktion*. In der folgenden Tabelle wird die Bedeutung der Zeitoperatoren erklärt sowie ihre textuelle und symbolische Schreibweise angegeben [HT10, HR04, Hol03].

Textuell	**Symbolisch**	**Interpretation**
X φ („*Next* φ“)	○φ	Im nächsten Zustand gilt φ (φ=*true*).
F φ („*Finally* φ“)	◇φ	Im Betrachteten oder mindestens in einem der folgenden Zustände gilt φ (Garantie).
G φ („*Globally* φ“)	□φ	In allen Zuständen inklusive dem Betrachteten gilt φ (Invarianz).
φ *U* ψ („φ *Until* ψ“)		Entweder im betrachteten Zustand gilt ψ oder in mindestens einem der folgenden Zustände gilt ψ und davor gilt φ ab dem betrachteten Zustand.
φ *W* ψ („φ *Weak Until* ψ“)		Es gilt entweder φ *U* ψ oder □φ.
φ *R* ψ („φ *Release* ψ“)		Entweder gilt □ψ oder es gibt einen Zustand, in dem φ und ψ gelten und davor gilt ψ ab dem betrachteten Zustand.

Tabelle 2.1: Die Zeitoperatoren von LTL

Eine LTL-Formel wird auf einem unendlichen Berechnungspfad π ausgewertet. Im rechten Teil der Abbildung 2.8 ist ein Berechnungspfad markiert, auf dem es beispielsweise einen Zustand gibt, ab dem b=*true* immer erfüllt ist. Das heißt, die LTL-Formel ◇□b ist auf diesem Pfad erfüllt.

Die Berechnungsbäume von Systemmodellen, die mit temporaler Logik spezifiziert werden, enthalten typischerweise mehrere Berechnungspfade. Eine LTL-Formel wird von einem Systemmodell genau dann erfüllt, wenn sie auf allen Berechnungspfaden erfüllt wird [RV01]. In der Kripke-Struktur in Abbildung 2.8 ist die LTL-Formel ◇□b nicht erfüllt, weil es einen Berechnungspfad gibt (z_1, z_2, z_1, z_2, ...), auf dem es keinen Zustand gibt, ab dem b=*true* immer erfüllt ist. Dagegen ist die Formel ◇b auf dieser Kripke-Struktur erfüllt. Auch ◇a ist erfüllt, jedoch ist ○a nicht erfüllt, weil im Zeitpunkt t_1 der Zustand z_3 möglich ist, in dem a nicht erfüllt ist.

Definition 2.2 (Modell einer LTL-Formel): Es seien M eine temporale Struktur, π ein unendlicher Pfad von M und φ eine LTL-Formel. Wenn π die Formel φ erfüllt, dann bedeutet die Schreibweise $\pi \vDash \varphi$, dass π ein *Modell* für φ ist. $M \vDash \varphi$ bedeutet, dass die Struktur ein Modell für φ ist (vgl. [HR04]).

Definition 2.3 (Semantik der LTL-Zeitoperatoren): Es sei M eine temporale Struktur, π^0 ein Pfad des Berechnungsbaums von M beginnend mit dem Initialzustand z_0, π^i ein Suffix eines Pfades beginnend mit dem Zustand z_i und φ eine LTL-Formel. Dann gilt (vgl. [HR04]):

$\pi^0 \vDash a \quad \Leftrightarrow \quad a \in z_0$, a ∈ {atomare Aussagen}

$\pi^0 \vDash \bigcirc\varphi \quad \Leftrightarrow \quad \pi^1 \vDash \varphi$

$\pi^0 \vDash \Diamond\varphi \quad \Leftrightarrow \quad \exists i \geq 0, \pi^i \vDash \varphi$

$\pi^0 \vDash \Box\varphi \quad \Leftrightarrow \quad \forall i \geq 0, \pi^i \vDash \varphi$

$\pi^0 \vDash \varphi\, U\, \psi \quad \Leftrightarrow \quad \exists i \geq 0, \pi^i \vDash \psi$ und $\forall k, k = 0 \leq k < i,\ \pi^k \vDash \varphi$

$\pi^0 \vDash \varphi\, W\, \psi \quad \Leftrightarrow \quad \varphi\, U\, \psi$ oder $\Box\varphi$

$\pi^0 \vDash \varphi\, R\, \psi \quad \Leftrightarrow \quad \exists i \geq 0, \pi^i \vDash \varphi$ und $\forall k, k = 0 \leq k \leq i,\ \pi^k \vDash \psi$ oder $\Box\psi$

Beispiele:

Eine andere Darstellung für Berechnungspfade, wie z. B. in [Fis11] verwendet, ist in Abbildung 2.9 zu sehen. Auf dem abgebildeten Berechnungspfad gelten folgende LTL-Formeln: □a, ◇y, ◇□z, ◇(a ∧ y ∧ c), □(z→a), □(y→○○□z), v *U* y, y *R* c, a *W* x, a ∧ v

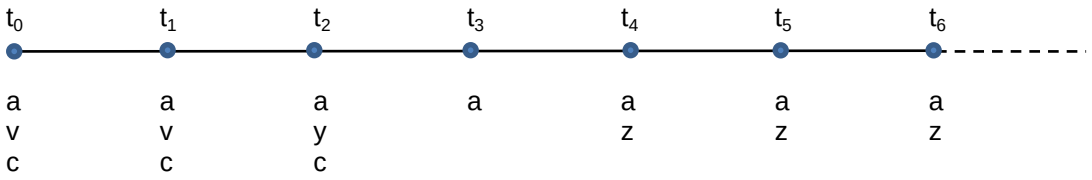


Abbildung 2.9: Zustandssequenz für Beispiele von LTL-Formeln

Es ist wichtig zu beachten, dass eine LTL-Formel, die keine Zeitoperatoren enthält nur dann auf einem Pfad erfüllt ist, wenn sie im Initialzustand erfüllt ist. Beispielsweise gilt auf dem Pfad in Abbildung 2.9 die Formel a→v, wohingegen a→z nicht gilt. Auch die Vorbedingung p in einer Formel der Art p→◇q oder die Nachbedingung q in ◇p→q müssen im Initialzustand erfüllt sein.

2.2.1.1 Büchi-Automaten als Modelle von LTL-Formeln

In der Informatik werden unter Automaten mathematische Konstrukte verstanden, die unter anderem aus Zuständen und Zustandsübergängen bestehen. Sie werden dazu genutzt ein Systemverhalten, das heißt die Transformation einer Eingabe in eine Ausgabe, zu beschreiben oder eine bestimmte Art von Eingaben zu erkennen [SS11], Sch08]. Dabei werden die Eingaben als Wörter einer formal definierten Sprache bezeichnet.

Büchi-Automaten sind eine spezielle Art von Automaten, die aus endlich vielen Zuständen, einem Startzustand, einer Menge von Endzuständen und einer Transitionsfunktion bestehen und als Eingabe unendliche Wörter erkennen, das heißt *akzeptieren* [HL11]. Ein Büchi-Automat akzeptiert ein Wort, wenn ein akzeptierender Zustand unendlich oft besucht wird.

In [VW94] wurde gezeigt, dass zu jeder LTL-Formel ein Büchi-Automat konstruierbar ist, sodass seine Sprache genau den Modellen (siehe Definition 2.2) der LTL-Formel entspricht. Die Wörter

der Sprache entsprechen dabei den Berechnungspfaden (siehe Abbildung 2.8) des Modells. Das heißt, ein Büchi-Automat, welcher das Modell einer LTL-Formel darstellt, akzeptiert genau die Eingaben, die den Berechnungspfaden des Systemmodells entsprechen. In Abbildung 2.10 sind zu drei LTL-Formeln beispielhaft die möglichen Berechnungspfade sowie die entsprechenden Büchi-Automaten abgebildet.

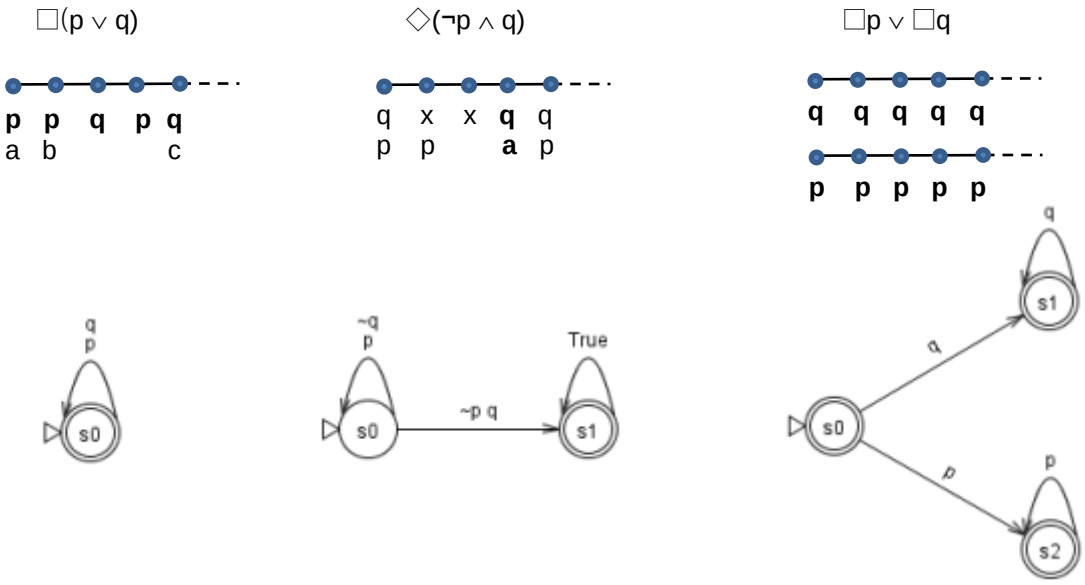


Abbildung 2.10: Beispiele für Büchi-Automaten (generiert mit GOAL [YKTH12])

In der grafischen Darstellung eines Büchi-Automaten werden die akzeptierenden Zustände doppelt umkreist. Die Pfeile symbolisieren Zustandsübergänge. Ein Zustandsübergang im Büchi-Automat findet nach einer Auswertung eines Zustands im Berechnungspfad statt, wenn alle nacheinander stehenden Variablen auf dem Pfeil *im nächsten Zustand des Berechnungspfades* erfüllt sind. Von den untereinander stehenden Variablen ist für einen Zustandsübergang die Erfüllung einer der Variablen ausreichend. Ein Pfeil mit *True* bedeutet dass jede Variablenbelegung zu einem Zustandsübergang führt. Beispielsweise erreicht der mittlere Automat den akzeptierenden Zustand s1 und verbleibt dort unabhängig vom weiteren Verlauf, sobald im Berechnungspfad der Zustand erreicht wird, in dem p=*false* und q=*true* gilt.

Es gibt zahlreiche Tools zur Generierung von Büchi-Automaten. Beispielsweise kann auf der Webseite [Gas] durch Eingabe der LTL-Formel in ein Formular ein Büchi-Automat generiert werden. Des Weiteren sind im Programm GOAL [YKTH12] neben anderen Tools zu temporalen Logiken auch Generatoren für Büchi-Automaten enthalten. Und schließlich bietet der Model-Checker SPIN (siehe Abschnitt 2.3.2) diese Möglichkeit mit dem Befehl „spin -f <Formel>“.

2.2.1.2 Äquivalenzen

Im Folgenden werden einige der für die Arbeit mit LTL wichtigen Äquivalenzen aufgeführt [HR04]. Zwei LTL-Formeln ψ und φ sind semantisch äquivalent, symbolisch als φ≡ψ bezeichnet, wenn alle Modelle für φ auch Modelle für ψ sind und umgekehrt [HR04]. Die Operatoren □ und ◇ sowie *U* und *R* sind dual zueinander. Beispielsweise lässt sich □ mit ◇ ausdrücken und umgekehrt. Der ○-Operator ist dual zu sich selbst.

$\neg\Box\varphi \equiv \Diamond\neg\varphi$ $\neg\Diamond\varphi \equiv \Box\neg\varphi$ $\neg\bigcirc\varphi \equiv \bigcirc\neg\varphi$

$\neg(\varphi\ U\ \psi) \equiv \neg\varphi\ R\ \neg\psi$ $\neg(\varphi\ R\ \psi) \equiv \neg\varphi\ U\ \neg\psi$ $\varphi\ U\ \psi \equiv \varphi\ W\ \psi \wedge \Diamond\psi$

$\Diamond\varphi \equiv true\ U\ \varphi$ $\Box\varphi \equiv false\ R\ \varphi$

Es gelten folgende Distributiv-Gesetze bezüglich $\Box$ und $\Diamond$:

$\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$ $\Box(\varphi \vee \psi) \not\equiv \Box\varphi \vee \Box\psi$

Die nicht äquivalenten Formeln $\Box(\varphi \vee \psi)$ und $\Box\varphi \vee \Box\psi$ könnten intuitiv so interpretiert werden, dass zu jedem Zustand entweder φ oder ψ gelten muss. $\Box(\varphi \vee \psi)$ bedeutet aber, dass auf einem erfüllenden Pfad in jedem Zeitpunkt entweder φ oder ψ gelten kann. Dagegen drückt $\Box\varphi \vee \Box\psi$ aus, dass es zwei Möglichkeiten für einen erfüllenden Pfad gibt, sodass entweder zu jedem Zeitpunkt φ oder zu jedem Zeitpunkt ψ gilt.

$\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$ $\Diamond(\varphi \wedge \psi) \not\equiv \Diamond\varphi \wedge \Diamond\psi$

Die Formel $\Diamond(\varphi \wedge \psi)$ bedeutet, dass es einen Zustand gibt, in dem φ und ψ gleichzeitig erfüllt sind. Die Formel $\Diamond\varphi \wedge \Diamond\psi$ drückt dagegen aus, dass φ und ψ auch zu unterschiedlichen Zeitpunkten erfüllt sein können.

Weiterhin gelten die aus der Aussagenlogik bekannten Äquivalenzen, wie z. B. die De Morgan‘schen Regeln und die Implikation [Sch00]:

$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$

2.2.1.3 Beispiele für ausdrückbare Systemeigenschaften in LTL

Die durch temporallogische Formeln ausdrückbaren Systemeigenschaften können unterschiedlich klassifiziert werden [MP92], wie z. B. in Lebendigkeits-, Sicherheits- und Fairnesseigenschaften. Einige LTL-Formeln haben eigene Namen, weil sie oft verwendet werden [Hol03]. In der Tabelle 2.2 werden einige einfache oft verwendete LTL-Formeln aufgeführt.

Formel	**Typ**	**Interpretation**
$\Box(p \rightarrow (p\ U\ q))$		Jeder Zustand in dem p gilt, führt zu einem Zustand in dem q gilt und dazwischen bleibt p gültig.
$p \rightarrow \Diamond q$	Antwort	Wenn p gilt, dann wird irgendwann q garantiert gelten.
$\Box\Diamond p$	Wiederholung („immer wieder“)	Falls p in einem Zustand nicht gilt, wird garantiert, dass p im weiteren Verlauf wieder gelten wird.
$\Diamond\Box z$	Stabilität	Ab einem garantierten Zustand gilt für immer z.
$\Diamond p \rightarrow \Diamond q$	Korrelation	Fall p auftritt, wird garantiert q auftreten.

Tabelle 2.2: Beispiele für häufig verwendete LTL-Formeln nach [Hol03]

Obwohl LTL im Vergleich zu anderen temporalen Logiken als eine intuitive Spezifikationssprache gilt [RV01], entstehen für scheinbar einfache Sachverhalte oft komplexe Ausdrücke, die schwer herzuleiten oder zu merken sind. Daher wurde in [www12e] eine Sammlung von Vorlagen sowohl für LTL als auch für andere temporale Logiken veröffentlicht. Mit diesen Vorlagen können solche Sachverhalte wie Abwesenheit, Existenz, Vorrang und Antwort für jeden der in Abbildung 2.7 dargestellten Gültigkeitsbereiche ausgedrückt werden. In der folgenden Tabelle 2.3 werden einige Beispiele aus dieser Vorlagen-Sammlung vorgestellt.

Formel	Typ	Interpretation
$\Box(\neg q \vee \Diamond(q \wedge \Diamond p))$	Existenz	p tritt nach q auf.
$\Diamond q \rightarrow (\neg p \; U \; q)$	Abwesenheit	p tritt vor q nicht auf.
$\Box(q \rightarrow \Box\neg p)$	Abwesenheit	p tritt nach q nicht auf.

Tabelle 2.3: LTL-Vorlagen nach [www12e]

2.2.1.3.1 Sicherheitseigenschaften

Informell ausgedrückt, garantiert eine *Sicherheitseigenschaft*, dass ein unerwünschter Zustand *niemals* eintritt [MP92] Im Allgemeinen sind Sicherheitseigenschaften von der Form:

$$\Box a.$$

Dabei ist a eine aussagenlogische Formel. Eine Sicherheitseigenschaft muss auf allen Berechnungspfaden erfüllt sein [Pnu86]. Die zu der informellen Beschreibung passende Form $\Box\neg(a \wedge b \wedge \ldots \wedge z)$ drückt den *gegenseitigen Ausschluss* von Variablen aus [Pnu86]. Beispielsweise kann damit ausgedrückt werden, dass die Netzwerkteilnehmer a-z niemals eine Ressource gleichzeitig nutzen sollen.

Weitere Beispiele [SSL10]: $\Box(a \rightarrow \Box b)$, $\Diamond a \rightarrow \Box b$

Für Sicherheitseigenschaften ist charakteristisch, dass ihre Gegenbeispiele endliche Pfade sind [HT10, Kin94, KYV01]. Wenn beispielsweise die Eigenschaft $\Box\neg(a \wedge b)$ verifiziert werden soll, wird nach einem Zustand gesucht, der das Gegenteil, das heißt $\neg\Box\neg(a \wedge b) \equiv \Diamond(a \wedge b)$, erfüllt. Sobald ein solcher Zustand gefunden wird, wird der Pfad vom Startzustand bis zu diesem Zustand als Gegenbeispiel ausgegeben.

2.2.1.3.2 Lebendigkeitseigenschaften

Informell ausgedrückt, garantiert eine *Lebendigkeitseigenschaft*, dass ein erwünschter Zustand eintritt [MP92]. Die grundlegenden Lebendigkeitseigenschaften sind von der Form [Pnu86]:

$$\Diamond a, \quad \Diamond\Box a \quad \text{oder} \quad \Box\Diamond a.$$

Dabei ist a eine aussagenlogische Formel. Ein weiteres Beispiel ist $\Box(p \rightarrow \Diamond q)$ welches die *Erreichbarkeit* ausdrückt. Beispielsweise kann damit ausgedrückt werden, dass immer wenn eine Anfrage (p) gestellt wird, erfolgt garantiert eine Antwort (q) [Pnu86].

Weitere Beispiele [SSL10]: $\Box a \rightarrow \Diamond b$

Gegenbeispiele für Lebendigkeitseigenschaften sind unendliche Pfade [HT10, Kin94, KYV01]. Das heißt, zum Nachweis der Nichterfüllung einer Lebendigkeitseigenschaft muss im Modell eine endlose Schleife gefunden werden, in der die Lebendigkeitseigenschaft nie erfüllt wird. Wenn das Modell keine Endlosschleifen enthält, ist ein Gegenbeispiel ein Pfad von Anfangs- bis zum Endzustand.

2.2.1.3.3 Co-Safety Eigenschaften

Eine Lebendigkeitseigenschaft deren Negation eine Sicherheitseigenschaft ist und umgekehrt, wird *Co-Safety* bezeichnet, z. B. $\Diamond a$ oder $\Box a \rightarrow \Diamond b$ [SSL10]. Nicht Co-Safety (siehe auch. Anhang A.3). sind z. B. $\neg a \; U \; b$, $\Box(a \rightarrow \Diamond b$, $\Diamond\Box a$

2.2.1.3.4 Fairness-Eigenschaften

Fairness-Eigenschaften drücken aus, dass etwas kontinuierlich passiert [Fis11]. Beispielsweise ist ein Planungsprozess fair, wenn er andere um eine Ressource konkurrierenden Prozesse derart verzahnt, dass sie gleich oft die Ressource nutzen können [Eme95]. Mit dem “unendlich oft”-Konstrukt □◇a können unterschiedlich starke *Fairness-Eigenschaften* ausgedrückt werden. Die folgenden vier Fairness-Eigenschaften [Fis11] sind von stark zu schwach sortiert:

□◇Anfrage → □◇Antwort	„Unendlich viele Anfragen → Unendlich viele Antworten“
□◇Anfrage → ◇Antwort	„Unendlich viele Anfragen → Mindestens eine Antwort“
□Anfrage → □◇Antwort	„Ununterbrochene Anfragen → Unendlich viele Antworten“
□Anfrage → ◇Antwort	„Ununterbrochene Anfragen → Mindestens eine Antwort

2.2.1.4 Normalformen

Komplexe logische Formeln können sowohl für den Benutzer schwer verständlich als auch algorithmisch schwer bearbeitbar sein. Daher werden komplexe Formeln oft in einfachere, aber zur den ursprünglichen Formeln semantisch äquivalente, Formeln umgeformt [Fis11]. Beispielsweise benötigen viele Algorithmen zur Erfüllbarkeitsprüfung aussagenlogischer Formeln eine Eingabe in *konjunktiver Normalform* [GNTV01, HR04].

Definition 2.4 (**Konjunktive Normalform, KNF):** Eine aussagenlogische Formel ist in *konjunktiver Normalform,* falls sie eine Konjunktion von Disjunktionen von Literalen ist und die Negationszeichen nur vor den Literalen stehen [Sch00].

Beispiel für eine KNF: (a ∨ b) ∧ (¬c) ∧ (¬a ∨ d ∨ e)

Hier sind a, b, c, d und e *Literale*, das heißt atomare Aussagen. Die Ausdrücke in den Klammern werden *Klauseln* bezeichnet. Laut Definition dürfen in den Klauseln einer KNF nur durch den ODER-Operator verbundene positive oder negative Literale vorkommen. Dabei kann jede aussagenlogische Formel in eine äquivalente KNF umgeformt werden [Sch00].

Definition 2.5 (Separated Normal Form, SNF): Sei Start eine Variable, die nur im Startzustand erfüllt ist und k_a, l_b und l positive oder negative Literale. Dann ist eine LTL-Formel ihrer *Separated Normal Form*, wenn sie von der Form $\square(\bigwedge_i K_i)$ ist, mit KlauselnK_i, die nur die Operatoren →, ○ und ◇ enthalten und von der folgenden Form sind (vgl. [Fis97, Fis11]):

start → $\bigvee l_b$	(Startzustand)
$\bigwedge k_a$ → ○ $\bigvee l_b$	(Nächster Zustand)
$\bigwedge k_a$ → ◇l	(Zukunft)

Beliebige LTL-Formeln können durch eine Reihe von Transformationsregeln [Fis11] in eine SNF umgeformt werden. Für das folgende Beispiel werden in Abbildung 2.11 zwei Modelle angegeben.

Beispiel für eine SNF: □((**start**→a) ∧ (**start**→b) ∧ ((a ∧ b)→○(c ∨ d)) ∧ (c → ◇c) ∧ (d →◇e))

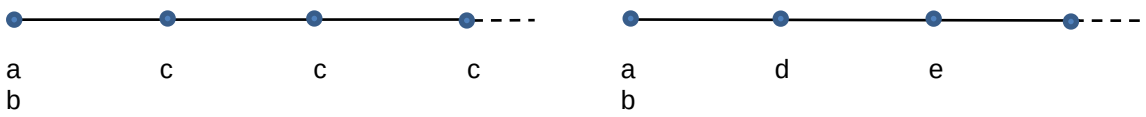


Abbildung 2.11: Modelle für das SNF-Beispiel

2.2.2 Weitere Arten temporaler Logik

Neben der LTL gibt es auch andere Arten temporaler Logik, die sich in der Wahl der Operatoren unterscheiden und dementsprechend unterschiedliche Ausdrucksmächtigkeit und Komplexität aufweisen. So können z. B. Vergangenheitsoperatoren eingeführt [LPZ85] oder der Bezug zur Realzeit [AH92] integriert werden.

Insgesamt können drei Arten temporaler Logiken unterschieden werden. Neben LTL ist die zweite Art die *verzweigende temporale Logik* CTL (engl. c*omputational tree logic* oder *branching-time logic)*, der ein verzweigendes Zeitmodell zugrunde liegt. In CTL gibt es die gleichen Temporaloperatoren (◇, □, *U*, ○) wie in der LTL. Jedoch muss *jedem* Temporaloperator unmittelbar ein Pfadquantor (*E, A*) vorangestellt werden [RV01]. Während eine LTL-Formel auf allen Berechnungspfaden eines Systems erfüllt sein muss, kann in CTL mit dem Pfadquantor *E* ausgedrückt werden, dass ein Ausführungspad existiert, auf dem eine bestimmte Systemeigenschaft erfüllt ist. Beispielsweise könnte dies die in einem Hardwaresystem wichtige *Reset*-Eigenschaft sein, die sicherstellt, dass das System in den Ausgangszustand zurück versetzt werden kann [Hol03]. Der Pfadquantor *A* drückt aus, dass die geforderte Systemeigenschaft auf allen Berechnungspfaden erfüllt ist. In den LTL-Formeln ist zwar implizit auch der Pfadquantor A enthalten, jedoch vor der gesamten Formel. Umgekehrt gibt es LTL-Formeln, wie z. B. ◇□a, die in der CTL nicht ausdrückbar sind. Insbesondere Fairness-Eigenschaften sind in CTL nicht ausdrückbar. Beide Logikarten sind Spezialfälle der CTL*, in der die Beschränkung, dass jedem Temporaloperator ein Pfadquantor vorangestellt werden muss, entfällt [RV01].

Weitere Details zu CTL* sowie den Unterschieden von LTL und CTL können in [EH84, EL87, Eme95, RV01, Hol03] gefunden werden. Die Tabelle 2.4 fasst die wichtigsten Unterschiede zusammen.

LTL	CTL
Auswertung über linearen Zustandsstrukturen	Auswertung über baumartigen Zustandsstrukturen
Keine Pfadquantoren	Pfadquantoren *E* und *A* vor jedem Temporaloperator
Intuitiv	Nicht intuitiv
Model-Checking in exponentieller Zeit in Abhängigkeit von der Größe der Spezifikation	Model-Checking in linearer Zeit in Abhängigkeit von der Größe der Spezifikation
	Fairness-Eigenschaften nicht direkt ausdrückbar (nur im Verifikations-algorithmus)

Tabelle 2.4: Unterschiede zwischen LTL und CTL, nach [RV01]

Aufgrund effizienterer Verifikationsmöglichkeiten wurde die CTL in der Industrie bevorzugt [RV01] verwendet. Heutzutage existieren sowohl für CTL als auch für LTL effiziente Model-Checking-Verfahren und beide Logikarten haben sich in bestimmten Einsatzgebieten etabliert. Die CTL wird beispielsweise bevorzugt in Hardware- und die LTL in Softwareverifikation eingesetzt [Hol03].

2.3 Model-Checking

Model-Checking ist eine Methode zur frühzeitigen Entdeckung von Fehlern in Systemen zur Entwurfszeit. Genauer gesagt, werden unter Model-Checking Verfahren zur vollautomatischen Verifikation von Modellen reaktiver, das heißt mit ihrer Umwelt interagierender, Systeme mit einer endlichen Zahl von Zuständen verstanden [CGL96]. Dieses Verfahren wurde in den 1980-er Jahren formal eingeführt [CES86, EL87]. Die Spezifikation des zu überprüfenden Systems muss in einer formalen Sprache wie z. B. der LTL oder CTL (siehe Abschnitt 2.2) vorliegen. Durch eine systematische Verfolgung aller Ausführungsmöglichkeiten des Modells, wird es dahingehend überprüft, ob es die in seiner Spezifikation geforderten Eigenschaften erfüllt.

Das Model-Checking wird in vielen Bereichen industriell eingesetzt, insbesondere in der Entwicklung von Hardware- und Softwaresystemen [HT10]. Somit können systematische Fehler z. B. in der Chip-Herstellung schon vor der kostspieligen Produktion ausgeschlossen werden. Wie für die temporale Logik, ist auch für das Model-Checking der Einsatz in der Geschäftsprozessmodellierung ein relativ neues Einsatzgebiet [FPR06, RMLD08].

Die Model-Checking-Verfahren können im Allgemeinen in Explizite und Symbolische klassifiziert werden. Im Gegensatz zum expliziten Model-Checking wird bei symbolischem Model-Checking die Berechnung des vollen Zustandsraums vermieden, indem die Zustände und ihre Beziehungen durch Formeln und binäre Entscheidungsbäume beschrieben werden [CGL96]. Dies ist vorteilhaft bei sehr großen Zustandsräumen, da eine Formel viele Zustände gleichzeitig beschreiben kann [HT10]. In dem in dieser Arbeit verwendeten Prototyp wird der explizite Model-Checker SPIN verwendet. Daher wird im Folgenden auf das explizite Model-Checking und SPIN näher eingegangen.

2.3.1 Explizites Model-Checking

Die expliziten Model-Checker bauen den Berechnungsbaum des zu verifizierenden Systems explizit im Speicher auf [RV07]. Sie wenden Tiefen- und Breitensuche an um einen Berechnungspfad zu finden, welcher der Spezifikation widerspricht [Hol03]. Falls ein solcher Pfad gefunden wird, wird dieser dem Benutzer als sogenanntes Gegenbeispiel ausgegeben. Ein Gegenbeispiel ist eine Zustandssequenz vom Startzustand bis zu dem Zustand, in dem die geforderte Eigenschaft nicht gilt.

Dieser Ansatz wird auch *automatentheoretischer Ansatz* bezeichnet. Sowohl das Modell als auch die Spezifikation sind Beschreibungen von Ausführungsmöglichkeiten, die als akzeptierte Wörter von Büchi-Automaten (siehe Abschnitt 2.2.1.1) betrachtet werden können. Daher wird die Beziehung zwischen Modellen und Spezifikationen auf die Beziehung zwischen Sprachen und Automaten zurückgeführt [Var99]. Dabei wird aus den Büchi-Automaten des zu verifizierenden Modells und seiner negierten Spezifikation ein Kreuzprodukt erstellt und die Sprache L dieses dritten Automaten auf Leerheit geprüft [DLP04, Deh04]. Wenn L nicht leer ist, bedeutet das, dass die negierte Spezifikation im modellierten System erfüllt und die Spezifikation damit verletzt wird. Als Beweis wird als Gegenbeispiel das akzeptierte Wort des Automaten, das heißt die Zustandssequenz, die im verifizierten Modell die Spezifikation verletzt, ausgegeben. Die Abbildung 2.12 gibt einen schematischen Überblick zum expliziten Model-Checking.

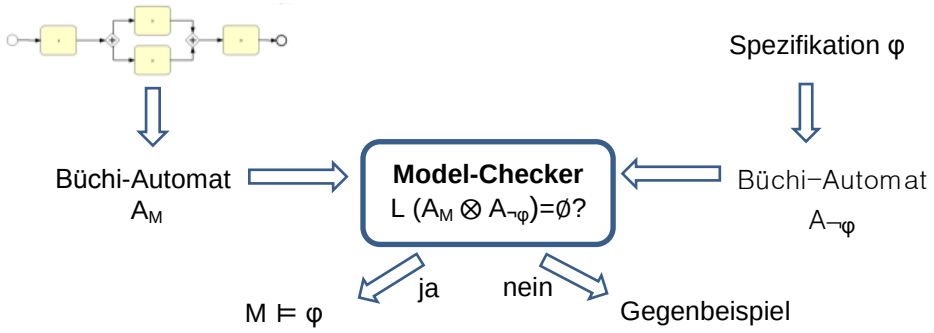


Abbildung 2.12: Explizites Model-Checking schematisch, nach [DLP04]

2.3.2 Model-Checker SPIN

SPIN ist ein in den 1980-er Jahren in den Bell Laboratories entwickelter und seit 1991 frei verfügbarer expliziter Model-Checker. Seinem Entwickler, Gerard Holzmann, wurde im Jahr 2002 der „Software System Award“ verliehen [www12a]. Das Akronym SPIN steht für „Simple Promela Interpreter“ [Hol03]. Die in SPIN zu verifizierenden Modelle müssen in der Sprache Promela vorliegen und die Spezifikationen in der LTL. SPIN enthält einen eigenen LTL-Übersetzer, mit dem Büchi-Automaten aus LTL-Formeln generiert werden können.

2.3.2.1 PROMELA

Das Akronym PROMELA steht für „Process Meta Language“ und bezeichnet eine Beschreibungssprache für Systeme, die mit anderen Systemen interagieren. Ihr Schwerpunkt liegt daher in der Beschreibung von Synchronisation und Koordination von asynchronen Prozessen. Der Promela-Code ist zwar für Simulations- und Verifikationszwecke ausführbar, stellt jedoch ein Systemmodell auf hoher Abstraktionsebene dar. Der ausführbare Promela -Code wird Promela-Programm oder *Promela-Modell* bezeichnet. Die Sprachkonstrukte von Promela sind auf Prozessinteraktion spezialisiert und ermöglichen insbesondere die Unterscheidung zwischen deterministischen und nichtdeterministischen Abläufen [Hol03]. Beispielsweise ist es in realen, auf unterschiedliche Standorte verteilten Systemen zu einem Zeitpunkt unbekannt, welcher der parallelen Prozesse den nächsten Schritt ausführen wird. Solche Prozesse können dadurch simuliert werden, dass der weitere Verlauf in einem Modell nicht-deterministisch ausgewählt wird, falls es dafür mehrere Möglichkeiten gibt [Hol03].

2.3.2.2 Never Claims

Ein Never Claim ist ein in Promela beschriebenes Systemverhalten welches *niemals* eintreten soll. Genau genommen, handelt es sich dabei um einen Büchi-Automaten (siehe Abschnitt 2.2.1.1), der aus der negierten Form der Spezifikation erstellt wird. Mit Never Claims ermöglicht es SPIN zu prüfen, ob und in welchem Schritt in einem Prozessmodell ein unerwünschter Zustand eintritt (vgl. Abschnitt 2.3.1). Da die mit Spin generierten Büchi-Automaten zur Lösung der Aufgabenstellung in dieser Arbeit (siehe Abschnitt 4.2.3.3) verwendet werden und einige Erweiterungen der Prototyps (siehe Abschnitt 5.4) auch die Never Claims betreffen, werden nachfolgend zwei Beispiele erläutert. Die Ausrufezeichen stehen dabei für Negation und jeder Zustand, dessen Bezeichnung mit „accept“ beginnt, ist ein möglicher Endzustand [Hol03] des Automaten.

2.3.2.2.1 Never Claim für die Lebendigkeitseigenschaft ◇a

Der Never Claim im Listing 2.1 stellt einen Büchi-Automaten für die negierte Lebendigkeitseigenschaft ◇a dar, oder anders ausgedrückt, einen Büchi-Automaten für die Sicherheitseigenschaft ¬◇a (≡□¬a). Der Büchi-Automat wird schrittweise abwechselnd zu dem Büchi-Automat des zu verifizierenden Modells ausgeführt. Ein Gegenbeispiel für ◇a wird gefunden, wenn alle Zustände des Modells durchlaufen werden und dabei kein Zustand mit a=*true* gefunden wird. Das heißt, es muss eine Ausführungsschleife gefunden werden, in der ¬a gilt. Eine solche Schleife wird in

Listing 2.1 durch die if-Schleife im „accept_init"-Zustand repräsentiert.

```
never  {    /* !(<>a) */
accept_init:
T0_init:
        if
        :: (! ((a))) -> goto T0_init
        fi;
}
```

Listing 2.1: Never Claim für ◇a oder Büchi-Automat für □¬a

Dieser Büchi-Automat terminiert, falls die if-Abfrage unendlich oft durchlaufen wird. Das passiert nur, wenn a=*false* nach jedem Ausführungsschritt im Modell gilt und somit die Spezifikation ◇a verletzt wird. In diesem Fall ist das Gegenbeispiel ein unendlicher Pfad oder, wenn das verifizierte Modell keine Endlosschleifen enthält, ein Pfad vom Start- bis zum Endzustand. Solche Gegenbeispiele sind typisch für Lebendigkeitseigenschaften (sieh Abschnitt 2.2.1.3.2).

Anderenfalls, wenn in einem Zustand des Modells a=*true* gilt, wird der Never Claim blockiert, weil es in der if-Schleife keine Alternative zu a=*false* gibt. Damit wird kein Gegenbeispiel gefunden und die Spezifikation ist in diesem Fall erfüllt.

2.3.2.2.2 Never Claim für die Sicherheitseigenschaft □a

Das
Listing 2.2 zeigt den Never Claim für die Sicherheitseigenschaft □a. Anders ausgedrückt, ist dies ein Büchi-Automat für ¬□a oder die äquivalente Formel ◇¬a (siehe Abbildung 2.13). Er wird schrittweise abwechselnd mit dem Modell ausgeführt, welches □a erfüllen soll. Dadurch wird nach einem Gegenbeispiel gesucht, welches zu einem Zustand führt, in dem a=*false* gilt.

```
never  {    /* !([]a) */
T0_init:
        if
        :: (! ((a))) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

Listing 2.2: Never Claim für □a oder Büchi-Automat für ◇¬a

Im Gegensatz zum vorherigen Beispiel gibt es hier zwei Alternativen in der if-Schleife. Falls immer a=*true* gilt, verleibt der Automat in der if-Schleife und terminiert nicht. In diesem Fall haben die Sprachen der Büchi-Automaten des Modells und der negierten Spezifikation keine gemeinsamen Wörter (vgl. Abbildung 2.12), das heißt die Spezifikation ist erfüllt.

2.3.2.2.3 Der „accept_all“-Zustand

Falls ein Zustand mit a=*false* gefunden wird, springt der Automat zu dem Zustand *„accept_all“* und akzeptiert alle weiteren zustände. Der akzeptierende Zustand „accept_all“ in Listing 2.2 entspricht dem Zustand s1 in der unteren Abbildung 2.13. Im Gegensatz zum vorherigen Beispiel wird nach dem Halten des Never Claims ein Gegenbeispiel mit endlicher Länge ausgegeben, was typisch für Sicherheitseigenschaften (siehe Abschnitt 2.2.1.3.1) ist.

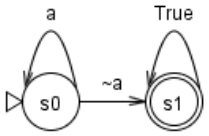


Abbildung 2.13: Der „accept_all“-Zustand im Büchi-Automat für ◇¬a

2.3.2.3 Zusammenfassung

Beim Model-Checking mit SPIN wird aus einer in LTL vorliegenden Spezifikation ein Never Claim generiert und mit dem in Promela beschriebenen Modell zu einem ausführbaren Programm verknüpft [Hol03]. Dabei greifen das Modell und der Never Claim auf dieselben globalen Variablen zu und nur das Modell kann die Variablen verändern.

Der Zustandsraum des Modells wird schrittweise nach einer Verletzung der Spezifikation durchsucht. Dazu werden das zu prüfende Modell und der Never Claim schrittweise abwechselnd ausgeführt. Nach jedem Schritt im Modell wird abhängig von dem neuen Zustand ein Schritt im Never Claim ausgeführt. Falls der Never Claim terminiert, bedeutet das, dass die Spezifikation verletzt wurde, weil ein zu ihr widersprüchlicher Zustand gefunden wurde. In diesem Fall wird als Gegenbeispiel ein endlicher oder unendlicher Pfad ausgegeben (vgl. Abschnitte 2.3.2.2.1 und 2.3.2.2.2).

2.4 LTL-Erfüllbarkeitsprüfung

2.4.1 Grundlagen der Erfüllbarkeitsprüfung

Im Folgenden werden die zum Verständnis von Erfüllbarkeits- und Gültigkeitsprüfungen notwendigen Grundlagen der Aussagelogik vorgestellt.

Definition 2.6 (Modell): Ein *Modell* für eine aussagenlogische Formel ist eine Belegung atomarer Formeln mit *wahr* oder *falsch*, sodass die Formel *wahr* wird [Sch00].

Die Belegungen *wahr* und *falsch* werden oft auch mit 1 und 0 oder mit *true* und *false* bezeichnet. Beispielsweise ist die Formel φ = a ∧ b unter der Belegung (a=1, b=1) *wahr.* Das heißt, diese Belegung ist ein Modell für φ.

Definition 2.7 (erfüllbar, unerfüllbar): Eine Formel φ ist *erfüllbar*, falls für sie mindestens ein Modell existiert, anderenfalls ist sie *unerfüllbar* [Sch00].

Definition 2.8 (gültig): Eine Formel φ ist *gültig*, falls jede Belegung ein Modell für φ ist [Sch00].

Anhand der Wahrheitstafel in Tabelle 2.5 werden einige Beispiele erläutert. Da in der zu φ gehörenden Spalte sowohl Nullen als auch Einsen vorkommen, ist φ *erfüllbar*. Da für jede

Belegung von a und b in der zu ψ gehörenden Spalte nur Einsen stehen, ist ψ *gültig.* Dagegen ist ξ *ungültig*.

a	b	φ=a→b	¬φ	ψ=a∨(a→b)	ξ=(a ∧ ¬a)
0	0	1	0	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	1	0	1	0

Tabelle 2.5: Wahrheitstafel für erfüllbare (φ, ¬φ), gültige (ψ) und unerfüllbare (ξ) Formel

Eine logische Formel kann entweder *gültig*, *unerfüllbar* oder *erfüllbar aber nicht gültig* sein. Insbesondere ist eine gültige Formel auch eine erfüllbare Formel. Die Abbildung 2.14 veranschaulicht, dass eine gültige Formel durch Negation zu einer unerfüllbaren Formel wird. Dagegen bleiben *erfüllbare aber ungültige* Formeln nach Negation *erfüllbar und ungültig* [Sch00], wie beispielsweise φ in Tabelle 2.5.

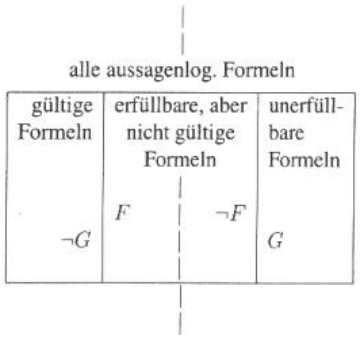


Abbildung 2.14: Zusammenhang zwischen Gültigkeit und Erfüllbarkeit [Sch00]

Satz 2.1 (gültig): Eine Formel φ ist **gültig** genau dann, wenn ¬φ unerfüllbar ist [Sch00].

Aus dem Satz 2.1 folgt, dass φ ungültig ist, wenn ¬φ erfüllbar ist. Z. B. sind φ und ξ in der Tabelle 2.5 ungültig, weil ihre Negationen erfüllbar sind.

Satz 2.2 (gültige Teilformeln): In einer erfüllbaren aber ungültigen Formel, die eine Konjunktion von Teilformeln darstellt, können *gültige Teilformeln* vorkommen.

Der Satz 2.2 lässt sich anhand eines einfachen Beispiels beweisen: Seien die Teilformeln A und B gültig und C erfüllbar. Dann ist die Formel F = A ∧ B ∧ C aufgrund der möglichen Belegung (A=1, B=1, C=1) erfüllbar aber aufgrund der möglichen Belegung (A=1, B=1, C=0) ungültig.

Satz 2.3 (erfüllbare Konjunktion): Eine Konjunktion von Teilformeln ist *erfüllbar* genau dann, wenn alle Teilformeln erfüllbar sind [Sch00].

Satz 2.4 (gültige Konjunktion): Eine Konjunktion von Teilformeln ist *gültig* genau dann, wenn alle Teilformeln gültig sind.

Der Satz 2.4 folgt aus der Tautologieregel: F ∧ G ≡ G, falls F gültig ist [Sch00].

Denn falls F die Konjunktion aller gültigen Teilformeln repräsentiert und G die einzige ungültige Teilformel ist, dann ist die gesamte Formel ungültig. Nur wenn G auch gültig ist, ist die Gesamtformel gültig.

2.4.2 SAT-Solver

Mit SAT (engl. satisfiability) wird das Erfüllbarkeitsproblem für logische Formeln bezeichnet. SAT-Solver sind entsprechende Systeme zur Erfüllbarkeitsprüfung logischer Formeln. Das SAT-Problem gehört zu der Komplexitätsklasse der nicht in polynomialer Zeit bezüglich der Eingabelänge lösbaren Probleme [Sch08]. Die Beziehung zwischen der Laufzeit und der Anzahl der Variablen in einer logischen Formel ist exponentiell. Es ist ein bedeutendes Forschungsthema in der Informatik, weil viele Probleme aus derselben Komplexitätsklasse sich auf das SAT-Problem zurückführen lassen [FM09]. Damit würde ein effizienter Algorithmus für das SAT-Problem auch eine effiziente Lösung für viele andere Probleme bedeuten.

Die LTL-Erfüllbarkeitsprüfung kann wie das Model-Checking in explizite und symbolische Verfahren klassifiziert werden [RV07]. Das heißt, die Übersetzung der Modelle und der LTL-Formeln erfolgt jeweils explizit oder symbolisch. Des Weiteren wurde in [RV07] experimentell festgestellt, dass symbolische Methoden zur LTL-Erfüllbarkeitsprüfung schneller als explizite Methoden sind. Im Folgenden werden einige Möglichkeiten zu Erfüllbarkeitsprüfung von LTL-Formeln kurz vorgestellt. sowie beispielhaft entsprechende SAT-Solver vorgestellt.

2.4.2.1 LTL-Erfüllbarkeitsprüfung durch Zurückführung auf Model-Checking

In [RV07] wird ein Ansatz zur LTL-Erfüllbarkeitsprüfung durch Zurückführung auf Model-Checking vorgestellt. Der Unterschied zum normalen Model-Checking besteht darin, dass das zu prüfende Modell anhand der LTL-Formel generiert wird. Es wird ein *universelles Modell* M generiert, welches alle möglichen Berechnungspfade des mit der LTL-Formel spezifizierten Systems enthält. Eine Formel φ ist genau dann erfüllbar, wenn das universelle Modell M die Negation ¬φ nicht erfüllt. In diesem Fall wird der Model-Checker ein Gegenbeispiel ausgeben, welches bedeutet, dass das unerwünschte Verhalten ¬φ im universellen Modell nicht möglich ist.

2.4.2.2 LTL-Erfüllbarkeitsprüfung durch alternierende Automaten

In [WDMR08] wird ein weiterer automaten-theoretischer Ansatz beschrieben, bei dem sowohl der explizite Aufbau eines nichtdeterministischen Büchi-Automaten vermieden als auch nicht auf reine boolesche Ableitung zurückgegriffen wird. Dabei werden LTL-Formeln in sogenannte alternierende Büchi-Automaten übersetzt, die kompakter als Büchi-Automaten sind. Während Büchi-Automaten im schlimmsten Fall exponentiell viele Zustände enthalten, enthalten alternierende Automaten nur linear viele Zustände. Anschließend wird der sogenannte Antichain-Algorithmus verwendet, der in [WDHR06] vorgestellt wird.

Die Autoren von [WDMR08] haben ein Tool namens ALASKA [Wul12] veröffentlicht, welches sowohl als Model-Checker als auch als SAT-Solver benutzt werden kann. Die Software wird über die Konsole bedient. Dabei wird die zu prüfende LTL-Formel in Textform als Parameter übergeben und als Ergebnis „formula is SATISFIABLE“, „formula is NONSATISFIABLE“, „formula is NONVALID“ oder „formula is VALID“ ausgegeben. Die Ausgaben sind einfach zu parsen und das Tool ist damit in Verifikationssoftware einfach integrierbar.

2.4.2.3 Maude

Als ein Beispiel für einen symbolischen SAT-Solver wird das unter der GNU-Lizenz frei verfügbare Logik-Rahmenwerk Maude vorgestellt. Maude ist eine am Forschungsinstitut SRI International (Stanford Research Institute) entwickelte ausführbare Spezifikationssprache für Gleichungs- und Termersetzungssysteme. Dabei kann Maude als Ausführungsumgebung für verschiedene logische Sprachen dienen [wwwe]. Ein Termersetzungssystem stellt in der theoretischen Informatik ein Berechnungsmodell dar, welches aus Mengen von Ersetzungsregeln besteht. Eine Ersetzungsregel liegt in der Form *Linker_Term* → *Rechter_Term* vor und bedeutet, dass der linke Term durch den rechten Term substituiert werden kann [wwwk].

In Maude ist ein Modul für Model-Checking und Erfüllbarkeitsprüfung integriert [wwwd]. Im Folgenden wird ein Einblick in das System an einem praktischen Beispiel für LTL-Erfüllbarkeitsprüfung gegeben.

Die Literale der zu prüfenden Formel werden in einem neuen Modul, welches die Module SAT-SOLVER und LTL erweitert, in der Datei “SAT-SOLVER-TEST“ gespeichert.

```
fmod SAT-SOLVER-TEST is
   extending SAT-SOLVER .
   extending LTL .
   ops a b c  : -> Formula .
endfm
```

Listing 2.3: LTL-Erfüllbarkeitsprüfung in Maude (Modul)

Anschließend wird Maude auf der Kommandozeile mit “maude.linux64“ gestartet, wobei sich die Maude-Konsole öffnet. In der Maude-Konsole werden zunächst die Module model-checker.maude und SAT-SOLVER-TEST geladen und mit dem Befehl “red satSolve(<formel>) .“ die Erfüllbarkeitsprüfung einer LTL-Formel aufgerufen (Siehe Listing 2.4).

Die Ausgabe “result SatSolveResult: model(a ; b, (~ c) ; c)“ bedeutet, dass es zu der angegebenen Formel das in Abbildung 2.15 angegebene Modell existiert. Im Falle einer unerfüllbaren Formel wird „result Bool: false“ ausgegeben.

```
Maude> load model-checker.maude
Maude> load SAT-SOLVER-TEST
Maude> red satSolve(a /\ (O b) /\ (O O ((~ c) /\ [](c \/ (O c))))) .
reduce in SAT-SOLVER-TEST : satSolve(O O (~ c /\ [](c \/ O c)) /\ (a /\ O b)) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result SatSolveResult: model(a ; b, (~ c) ; c)
```

Listing 2.4: LTL-Erfüllbarkeitsprüfung in Maude (Konsole)

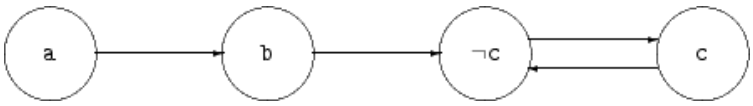


Abbildung 2.15: LTL-Erfüllbarkeitsprüfung in Maude (gefundenes Modells) [wwwd]

3 Verwandte Arbeiten

In diesem Kapitel werden die auf Compliance „by design“ (siehe Abschnitt 2.1.1) ausgerichteten Konzepte und Vorarbeiten vorgestellt, auf denen diese Arbeit aufbaut. Dazu gehören die Plausibilitätsprüfungen für Systemspezifikationen, der inkrementelle Entwicklungsprozess sowie der Oryx-Prototyp, der in der vorhergehenden Arbeit um Compliance-Scopes und das Model-Checking erweitert wurde.

3.1 Plausibilitätsprüfungen von Spezifikationen

Wie im Abschnitt 2.1.1.2 gezeigt, ist es aufgrund des exponentiellen Anstiegs der Fehlerbehebungskosten wichtig, Fehler möglichst früh zu erkennen. Dazu werden z. B. Plausibilitätsprüfungen sowohl am Systemmodell als auch an seiner Spezifikation vorgenommen [Kup06]. Im letzteren Fall ist es das Ziel zu erkennen, ob das System die Spezifikation auf eine triviale ungewollte Art erfüllt. Eine häufige Fehlerquelle dieser Art ist eine immer unerfüllte Vorbedingung einer Implikation [BB94]. Das folgende Beispiel demonstriert die Notwendigkeit von Plausibilitätsprüfungen.

$$\Box(\text{Anfrage} \rightarrow \Diamond \text{Antwort})$$

„Auf jede Anfrage folgt schließlich eine Antwort.“

Diese Lebendigkeits-Eigenschaft (siehe Abschnitt 2.2.1.3.2) ist in jedem Modell erfüllt, in dem es keine Anfragen gibt, was nicht der Absicht des Autors dieser Spezifikation entsprechen sollte. Viele industrielle Verifikationsprogramme berücksichtigen bereits diese Fehlerquelle, indem sie nach Teilformeln suchen, welche die Erfüllung der Spezifikation nicht beeinflussen. Im obigen Beispiel beeinflusst die Teilformel „Antwort“ das Ergebnis nicht, wenn im Modell keine „Anfrage“ vorkommt. Die Spezifikation wird erst dann verletzt, wenn es eine „Anfrage“ gibt und dann niemals eine „Antwort“ folgt. Um solche Teilformeln zu finden, wird das Model-Checking typischerweise mit modifizierten Teilformeln in der Spezifikation wiederholt [BBDER01, Kup06].

Eine weitere typische Fehlerquelle sind fehlende Lebendigkeits-Eigenschaften im Zusammenhang mit Sicherheits-Eigenschaften in ihrer typischen Form:

$$\Box\neg\ (\text{Client1\_druckt} \wedge \text{Client2\_druckt})$$

„Zu keinem Zeitpunkt können Client1 und Client2 gleichzeitig drucken.“

In einem Modell, in dem weder Client1 noch Client2 jemals drucken, ist diese Spezifikation zwar erfüllt, jedoch entspricht dieses Modell nicht der Absicht der Spezifikation. Es ist daher ratsam die Sicherheits-Eigenschaften mit entsprechenden Lebendigkeits-Eigenschaften zu kombinieren [Pnu86].

3.2 Notwendigkeit von Erfüllbarkeits- und Gültigkeitsprüfungen

Ein positives Ergebnis des Model-Checking garantiert keine Fehlerfreiheit des modellierten Systems. Ein Grund dafür können mögliche Fehler in der Spezifikation sein, wie im Abschnitt 3.1 gezeigt. Als weitere Plausibilitätsprüfungen muss ein Verifikationssystem laut [Var97, RV07] neben der Erfüllbarkeit auch die Ungültigkeit einer Spezifikation überprüfen können. Denn in den folgenden drei Fällen sind die gewöhnlichen Plausibilitätsprüfungen nicht ausreichend [RV07]:

Fall 1: Die Spezifikation ist *unerfüllbar*. Es ist unmöglich zu einer unerfüllbaren Spezifikation ein Modell zu entwickeln und der Model-Checker wird immer ein negatives Resultat ausgeben. Es liegt also ein Fehler in der Spezifikation vor.

Beispiel: $(a \; U \; b) \wedge \neg(\Diamond b)$

Per Definition verlangt der Until-Operator, dass b schließlich auftreten muss. Daher liegt ein Widerspruch vor.

Fall 2: Die Spezifikation ist *gültig*, das heißt erfüllbar in allen Modellen (vgl. Definition 2.8). Hier liegt ebenfalls ein Fehler in der Spezifikation vor, da es keinen Sinn ergibt absichtlich eine Spezifikation zu erstellen, die von beliebigen Modellen erfüllt wird.

Beispiel: $\Box(a \rightarrow \Diamond b)$

Diese Formel ist *gültig*, falls a und b äquivalent sind. Falls statt a und b komplexere Formeln verwendet werden, ist die Gültigkeit nicht so offensichtlich, wie in diesem Beispiel.

Fall 3: Auch wenn die Teilspezifikationen ***erfüllbar*** sind, kann die *Gesamtspezifikation* aufgrund widersprüchlicher Teilspezifikation *unerfüllbar* sein.

Beispiel: Teilspezifikation 1: $\Diamond b \wedge \ldots$
Teilspezifikation 2: $\Box\neg b \wedge \ldots$

Insbesondere ist der dritte Fall für diese Arbeit von Interesse. Er kann in Situationen auftreten, in denen für ein Prozessmodell mehrere Spezifikationen gelten müssen, z. B. wenn es sich dabei um eine Spezifikation eines Teilprozesses handelt, der die übergeordnete Spezifikation des Gesamtprozesses ebenfalls erfüllen muss [SALS10].

3.3 Inkrementelle Entwicklung Compliance-konformer Geschäftsprozessmodelle

In [SALS10] wurde das Konzept der Compliance-Templates [SALM09] um den ***inkrementellen Entwicklungsprozess*** erweitert. Dieser stellt sicher, dass die Compliance-Regeln der Unterprozesse nicht die Regeln der Prozesse verletzen, in die sie eingebettet sind. Dazu werden Füllbereiche definiert, denen Compliance-Regeln als aussagenlogische Formeln zugeordnet werden können. Die Compliance-Regeln werden dabei an Unterprozesse, die diese Füllbereiche verfeinern, weitergegeben. In Abbildung 3.1 sind drei Schichten des *inkrementellen Entwicklungsprozesses* beispielhaft dargestellt. Im Prozess-Template auf der untersten Schicht ist im Füllbereich A spezifiziert, dass die Aktivität A ausgeführt werden muss. Dieser Füllberiech wird durch einen Designer des Unterprozesses 1 auf der höheren Schicht verfeinert. Dieser enthält die Compliance-Regel B im Füllbereich B. Die Compliance-Regeln A und B werden durch den

logischen UND-Operator „^“ zu einer Compliance-Regel verknüpft, so dass der Unterprozess 1 sie beide erfüllen muss. Die verknüpften Compliance-Regeln werden dabei auf Erfüllbarkeit geprüft. Der Füllberiech B wird von dem Unterprozess 2 verfeinert und erbt die verknüpften Compliance-Regeln vom Unterprozess 1. Somit werden die Compliance-Regeln der ersten zwei Schichten erfüllt, wenn die Aktivitäten A und B im Unterprozess 2 modelliert werden. Die Compliance-Regeln werden in der Normalform KNF (siehe Abschnitt 2.2.1.4) angegeben. Somit stellen die Klauseln der KNF die Teilregeln dar, die an Unterprozesse weitergegeben werden können.

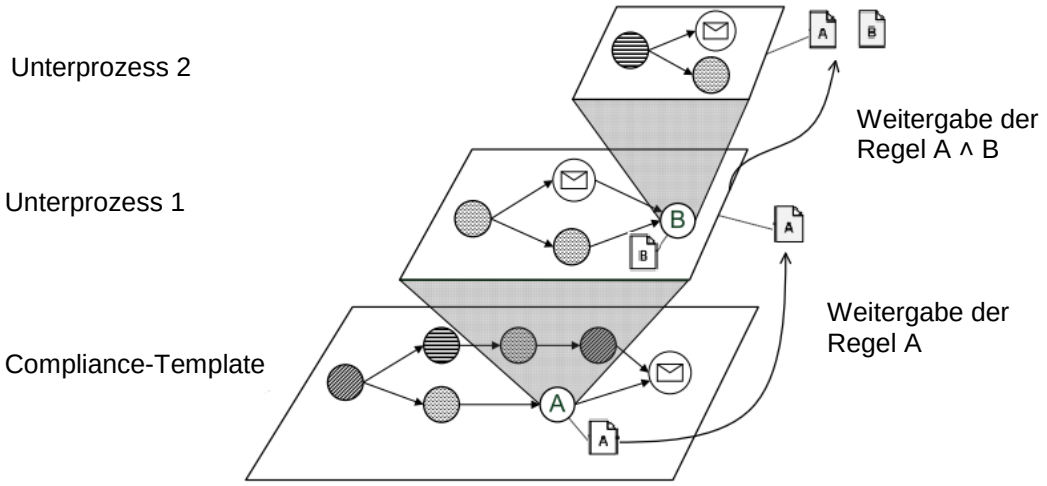


Abbildung 3.1: Weitergabe von Compliance-Regeln, nach [SALS10]

3.3.1 Konflikte

Bei der Verknüpfung von Compliance-Regeln können Konflikte zwischen geerbten und vorhandenen Compliance-Regeln entstehen, so dass die verknüpften Compliance-Regeln unerfüllbar sind. In [SALS10] wurden die folgenden *direkten* und *indirekten Konflikte* eingeführt:

Definition 3.1 (Indirekter Konflikt): Ein *indirekter Konflikt* tritt bei der Verknüpfung zweier Compliance-Regeln auf, wenn ein **positives Literal auf der tieferen Schicht** mit der negativen Form des gleichen Literals auf der höheren Schicht verknüpft wird.

Beispielsweise entsteht ein indirekter Konflikt, wenn in der Abbildung 3.1 im Unterprozess 1 statt B die Regel ¬A spezifiziert wird.

Definition 3.2 (Direkter Konflikt): Ein *direkter Konflikt* tritt bei der Verknüpfung zweier Compliance-Regeln auf, wenn ein **negatives Literal auf der tieferen Schicht** mit der positiven Form des gleichen Literals auf der höheren Schicht verknüpft wird.

So bedeutet beispielsweise die Entdeckung eines direkten Konflikts, dass die Spezifikation eines Unterprozesses korrigiert werden muss, weil sie die übergeordnete Compliance-Regel verletzt. Die Abbildung 3.2 zeigt ein Beispiel für einen direkten Konflikt.

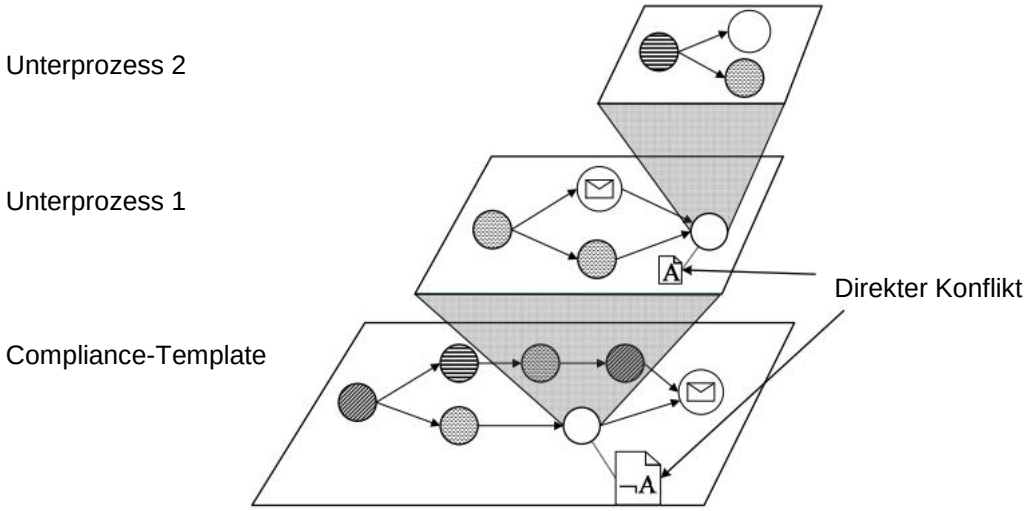


Abbildung 3.2: Direkter Konflikt, nach [SALS10]

3.3.2 Positive erfüllte Literale

Die Regeln dürfen im Falle positiver erfüllter Literale nicht weitergegeben werden. Wie in Abbildung 3.3 zu erkennen, sind dem Unterprozess 1 die Regeln X und Y zugeordnet. Da die Regel X erfüllt ist, wird sie nicht an den Unterprozess 2 weitergegeben.

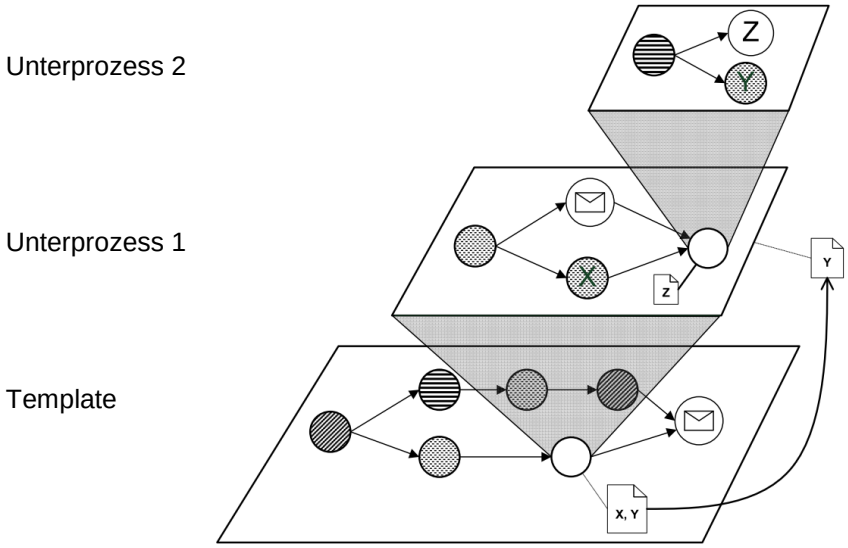


Abbildung 3.3: Positive erfüllte Literale, nach [SALS10]

3.4 Der Prototyp

In [SWLS10] wurden sogenannte ***Compliance-Scopes*** als eine Erweiterung der BPMN 2.0 eingeführt. Sie stellen Bereiche von Geschäftsprozessmodellen dar, in denen ***Compliance-Regeln*** (siehe Abschnitt 2.1.1) erfüllt sein müssen. Diese Bereiche werden in existierenden Modellen oder Prozess-Templates definiert und können selbst beliebig viele Compliance-Scopes enthalten. Durch die Zuordnung von Compliance-Regeln zu Compliance-Scopes wird die Erkennung von Compliance-Verletzungen während der Modellierung von Geschäftsprozessen ermöglicht.

Die Compliance-Scopes (siehe Abbildung 3.5) und ihre automatische Verifikation durch Model-Checking (siehe Abschnitt 2.3) wurden in der Diplomarbeit [Gro11] in dem webbasierten Editor Oryx (siehe Abschnitt 2.1.4) prototypisch umgesetzt. Die Compliance-Regeln werden dabei als LTL-Formeln (siehe Abschnitt 2.2.1) grafisch modelliert und den Compliance-Scopes zugeordnet. Als Model-Checker wird SPIN (siehe Abschnitt 2.3.2) verwendet. Dazu wird das in einem Compliance-Scope enthaltene BPMN-Modell in der Systembeschreibungssprache Promela (siehe Abschnitt 2.3.2.1) zusammen mit der assoziierten LTL-Formel in Form eines Never Claims (siehe Abschnitt 2.3.2.2) an SPIN übergeben. Die Compliance-Scopes, in denen die Compliance-Regeln verletzt werden, werden farbig hervorgehoben und als Gegenbeispiel wird eine Abfolge von Aktivitäten ausgegeben, die zu der Regel-Verletzung führt. Im Folgenden wird der für diese Arbeit relevante Ablauf der Modellierung und des Model-Checking im Prototyp beschrieben. Für weitere Details sei auf [Gro11] verwiesen.

3.4.1 LTL-Editor

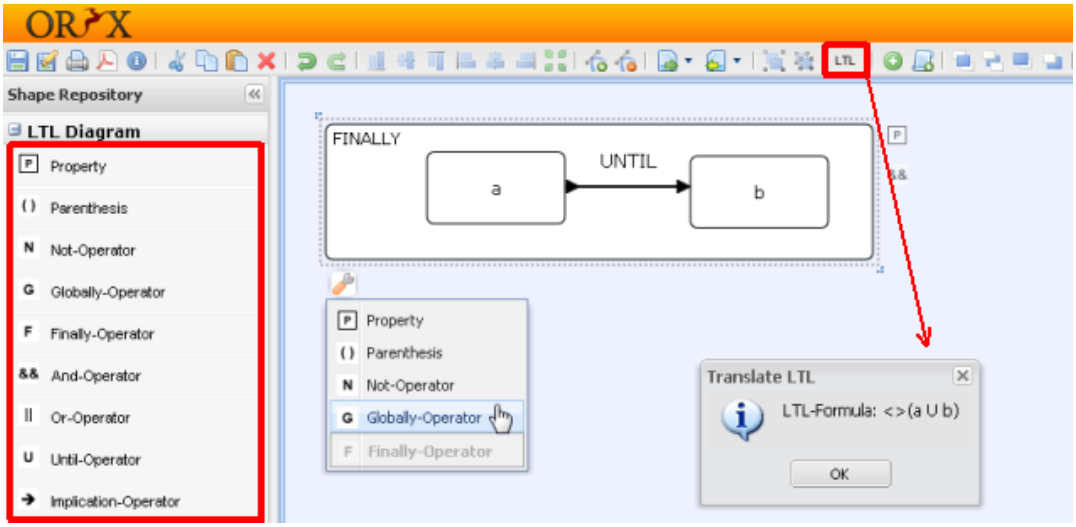


Abbildung 3.4: Grafischer LTL-Editor

Eine Compliance-Regel wird aus einzelnen LTL-Formeln zusammengesetzt, die in einem grafischen Editor modelliert werden. In der Abbildung 3.4 sind alle per Drag&Drop ins Diagramm einfügbaren Operatoren (¬,∨,∧,→,◇,□,U) sichtbar.[1] In der horizontalen Toolbar befindet sich der LTL-Button, mit dem sich eine modellierte Formel in ihre Textdarstellung, in diesem Fall „<>(a U b)“, umwandeln lässt.

3.4.2 Compliance-Regel-Editor

Nachdem ein BPMN-Diagramm mit mindestens einem Compliance-Scope modelliert wurde, kann für einen markierten Compliance-Scope eine Compliance-Regel erstellt werden. Dazu wird ein Compliance-Scope mit der Maus ausgewählt und über ein Drop-Down-Menü in der Oryx-Toolbar der *Compliance Wizard* aufgerufen (siehe in Abbildung 3.5).

[1] Der Next-Operator wird im Prototyp nicht unterstützt, weil das Model-Checking mit SPIN aus Performancegründen standardmäßig unter sogenannter partial order reduction ausgeführt wird [Gro11]. Dadurch ist der Berechnungsbaum für LTL-Formeln ohne den Next-Operator mit einer geringeren Anzahl von Verzweigungen möglich. Somit sind relative Ausführungsreihenfolgen in parallelen Prozesspfaden nicht ausdrückbar [Hol03]. Diese sind jedoch auch in realen Geschäftsprozessen in der Regel unbestimmt [For02]. Für das Model-Checking unter Berücksichtigung des Next-Operators muss SPIN mit einem speziellen Parameter aufgerufen werden [Hol03].

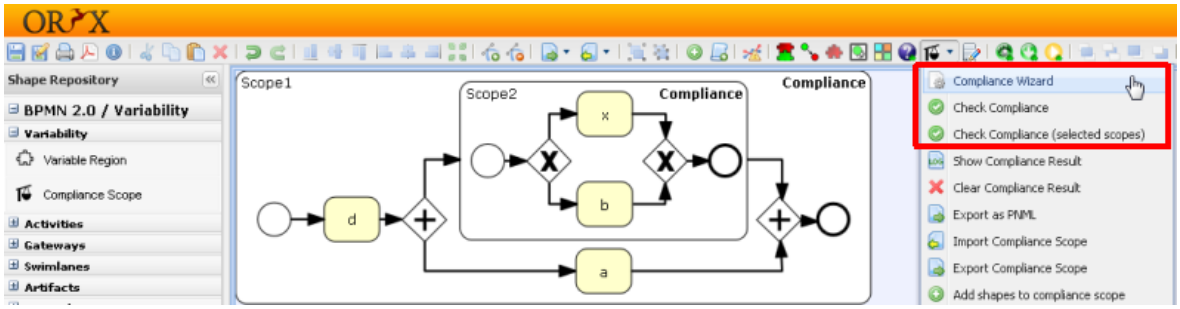


Abbildung 3.5: Aufruf der Compliance-Prüfung

In dem in Abbildung 3.6 dargestellten *Compliance Wizard* erfolgt die Zusammensetzung der Compliance-Regel aus einzelnen LTL-Formeln mit Hilfe der logischen Operatoren NOT, AND und OR. Die Operatoren können als Operanden entweder weitere Operatoren oder LTL-Regeln enthalten. Außerdem können Datentransfer-Regeln eingefügt werden, die in dieser Arbeit jedoch nicht betrachtet werden. Somit wird die Compliance-Regel als ein ***Regelbaum***, auch *Operatorenbaum* genannt, dargestellt. Der Regelbaum enthält die logischen Operatoren in den Knoten und die Namen und IDs der LTL- oder Datentransfer-Regeln in den Blättern (vgl. [Gro11].

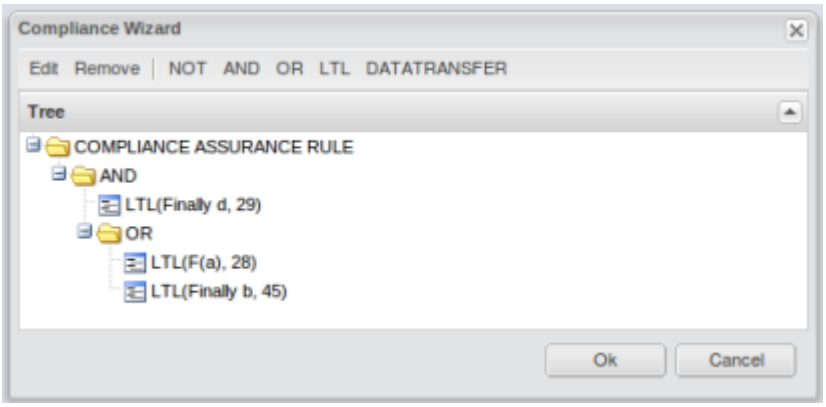


Abbildung 3.6: Der Regelbaum im Compliance Wizard

3.4.3 Compliance-Prüfung

Nachdem die Modellierung des BPMN-Prozesses und die Erstellung der Compliance-Regel abgeschlossen sind, kann die Compliance-Prüfung, das heißt Model-Checking aller Compliance-Scopes oder nur der Ausgewählten, über das in Abbildung 3.5 dargestellte Dropdown-Menü aufgerufen werden („Check Compliance“).

Auswertung des Regelbaums

Statt für die lange Compliance-Regel wird das Model-Checking nur für die einzelnen LTL-Regeln von den Blättern des Regelbaums ausgeführt. An jedem Knoten wird der boolesche Wert entsprechend der logischen Semantik des jeweiligen Operators berechnet. Das Gesamtergebnis in der Wurzel gibt an, ob die Compliance-Regel im Modell erfüllt wird (vgl. [Gro11]). Während für den UND-Operator alle Operanden das Model-Checking erfolgreich bestehen müssen, reicht es für den OR-Operator, wenn nur ein Operand vom Modell erfüllt wird. Auch im Falle des UND-Operators werden nicht immer alle LTL-Regeln geprüft, denn sobald ein nicht erfüllter Operand festgestellt wird, werden alle anderen übersprungen. Andererseits ist es beim ODER-Operator möglich, dass alle Operanden geprüft werden müssen, bis ein erfüllender Operand gefunden wird.

Ergebnis

Nach dem erfolgten Model-Checking werden die Compliance-Scopes entsprechend ihren Ergebnissen farblich gekennzeichnet. Beispielsweise bedeutet die grüne Färbung in Abbildung 3.7, dass die Compliance-Regel im Compliance-Scope erfüllt wird, während rot das Gegenteil bedeutet.

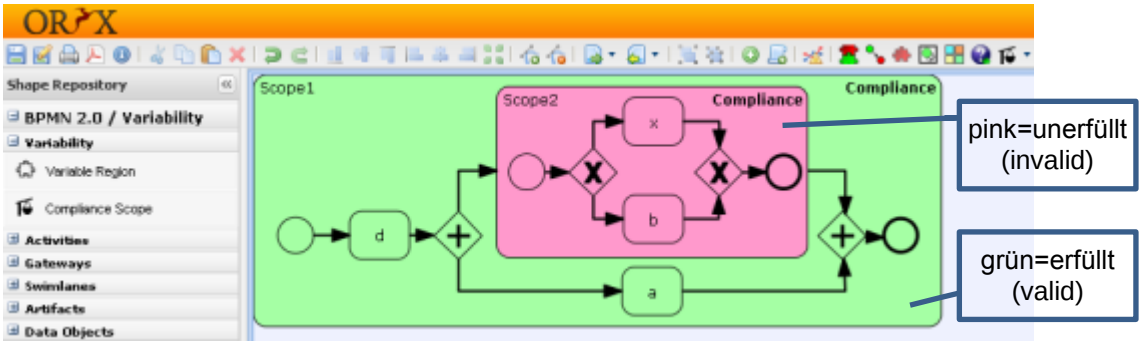


Abbildung 3.7: Farbliche Kennzeichnung der Compliance-Scopes

Gleichzeitig wird das Ergebnisfenster eingeblendet. Im ersten Reiter wird die Zusammenfassung (siehe Abbildung 3.8) aller Ergebnisse und in den weiteren Reitern die Ergebnisse und Logs einzelner Compliance-Scopes angezeigt. Falls das Modell seine Spezifikation nicht erfüllt, wird ein Gegenbeispiel angegeben. Beispielsweise wird in Abbildung 3.9 ein Ausführungspfad angegeben, der die Compliance-Regel ◇c verletzt.

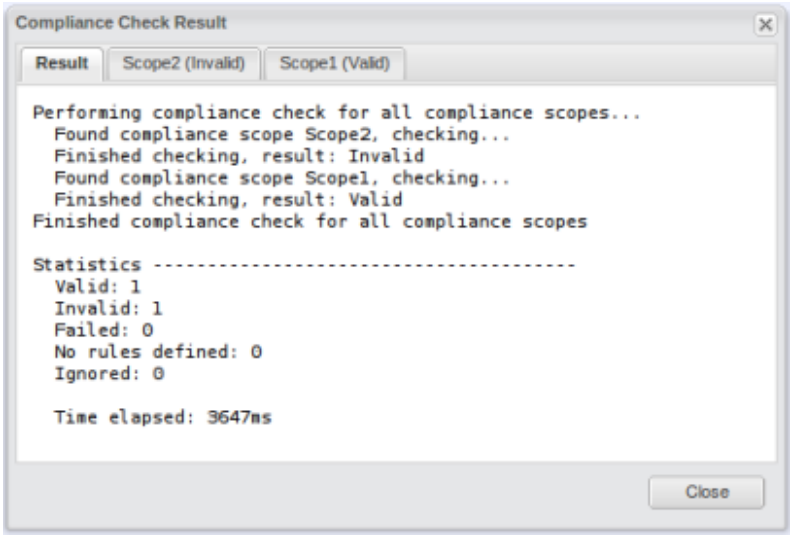


Abbildung 3.8: Ergebnisfenster einer Compliance-Prüfung (Gesamtergebnis)

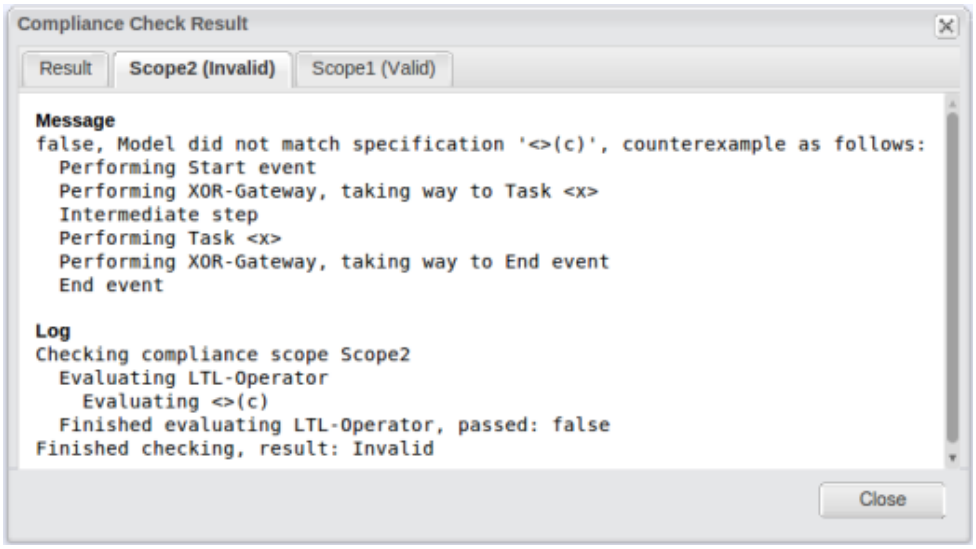


Abbildung 3.9: Ein Gegenbeispiel nach dem Model-Checking

3.4.4 Variable Regionen

In [SALM09] wurden sogenannte *Compliance-Templates* eingeführt und in diesem Prototyp durch [Köt10] implementiert. Dabei enthalten die *Compliance-Templates* sogenannte ***variable Regionen***, die mit Aktivitäten gefüllt werden müssen, damit ein ausführbarer Prozess entsteht. Wie die nachfolgende Abbildung zeigt, werden variable Regionen mit einem Puzzle-Symbol in der rechten unteren Ecke einer Aktivität kenntlich gemacht. Diese Art von Aktivitäten wird in einigen Abbildungen der folgenden Kapitel verwendet um noch nicht vollständig modellierte Prozessmodelle darzustellen.

Abbildung 3.10: Variable Region

4 Konzept

Aufbauend auf den beschriebenen Grundlagen und Vorarbeiten wird in diesem Kapitel ein Konzept zur Konsistenzprüfung der Compliance-Regeln verschachtelter Compliance-Scopes erarbeitet. In erster Linie geht es um die Übertragung des in Abschnitt 3.3 beschriebenen *inkrementellen Entwicklungsprozesses* aus [SALS10] auf die Compliance-Scopes und die LTL (siehe Abschnitt 2.2.1). Es wird eine Definition für positive und negative Regeln eingeführt, welche die Weitergabe von Regeln an Compliance-Scopes bestimmt. Des Weiteren wird die Integration des Konzeptes in den Oryx-Prototyp (siehe Abschnitt 3.4) erläutert.

4.1 Allgemein

In diesem Kapitel werden wichtige Begriffe definiert sowie einige Beispiele für Compliance-Regeln in LTL angegeben. Das Anführen von Beispielen dient dabei zum einen dazu, die praktische Relevanz des Themas zu verdeutlichen und zum anderen dazu, in den darauf folgenden Abschnitten aus technischen Gründen auf praxisnahe Prozessdiagramme weitgehend verzichten zu können.

4.1.1 Konventionen

Um begriffliche Verwechselungen zu vermeiden wird im Folgenden die Bedeutung wichtiger Begriffe definiert. In Abbildung 4.1 werden die wichtigsten Begriffe aus Sicht des mittleren Compliance-Scopes zusammengefasst.

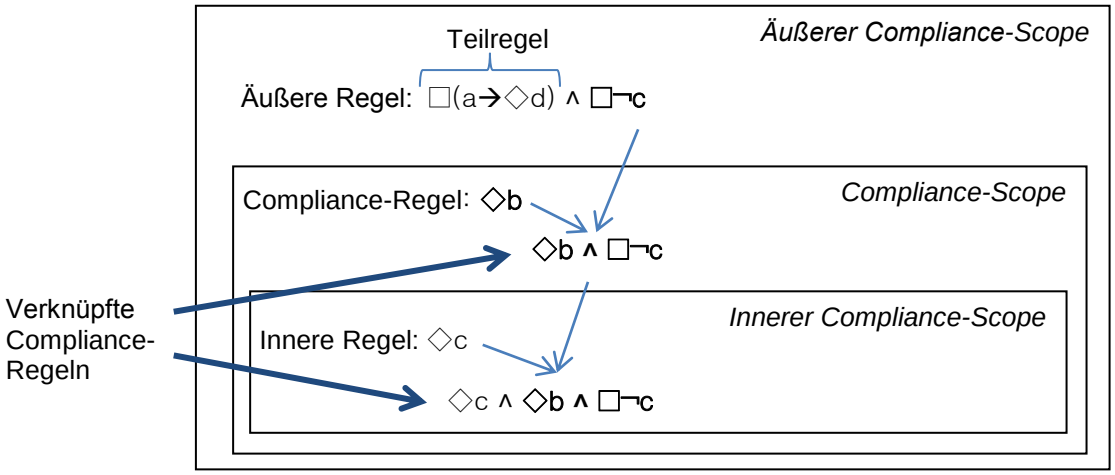


Abbildung 4.1: Schematischer Begriffsüberblick

Compliance-Scope oder ***Scope***: Ein *Compliance-Scope* ist ein abgegrenzter Prozessbereich in einem BPMN-Diagramm, dem eine Compliance-Regel zugeordnet ist. Eine formale Definition, die im Rahmen des Prototyps gilt, kann in [SWLS10] gefunden werden.

Teilregel: Eine *Teilregel* ist eine der durch den logischen UND-Operator auf der höchsten Hierarchieebene des Regelbaums (siehe Abbildung 3.6) verbundenen Teilformeln einer Compliance-Regel.

Compliance-Regel oder ***Regel***: Eine *Compliance-Regel* ist eine Beschreibung der Eigenschaften eines Prozessmodells in der LTL. Eine *Compliance-Regel* kann aus mehreren durch den logischen UND-Operator verknüpften Teilregeln bestehen.

Innerer Compliance-Scope: Die in einem Compliance-Scope - auf derselben Ebene wie die Aktivitäten - enthaltenen Compliance-Scopes werden *innere Compliance-Scopes* oder *innere Scopes* bezeichnet. Die Regeln eines inneren Compliance-Scopes müssen mit den Regeln seiner äußeren Compliance-Scopes konsistent sein.

Äußerer Compliance-Scope: Der Compliance-Scope, in dem sich der betrachtete Compliance-Scope befindet, wird als *äußerer Compliance-Scope* oder *äußerer Scope* bezeichnet.

Innere Compliance-Regel: Die Compliance-Regel des betrachteten Compliance-Scopes wird *innere Compliance-Regel* oder *innere Regel* bezeichnet.

Äußere Compliance-Regel: Die Compliance-Regel des äußeren Compliance-Scopes wird *äußere Compliance-Regel* oder *äußere Regel* bezeichnet.

Verknüpfte Compliance-Regel: Eine *verknüpfte Compliance-Regel* enthält neben der Compliance-Regel des geprüften Compliance-Scopes alle relevanten Teilregeln der äußeren Compliance-Regeln, die durch den logischen UND-Operator verknüpft sind.

Konsistenz: Unter *Konsistenz* wird die logische Erfüllbarkeit einer verknüpften Compliance-Regel verstanden.

Compliance-Prüfung: Unter *Compliance-Prüfung* wird in dieser Arbeit das Model-Checking in Kombination mit der Weitergabe relevanter Teilregeln an innere Compliance-Scopes und der Konsistenzprüfung verstanden.

4.1.2 Anwendungsbeispiele

Als Anwendungsbeispiel wird ein Konzern mit mehreren Geschäftsfeldern betrachtet. Während auf der Konzernebene strategische Vorgaben für alle Geschäftsfelder gelten, haben die Geschäftsfelder und die Abteilungen ihre eigenen branchenspezifischen oder internen Regelungen sowie gesetzliche Bestimmungen zu erfüllen.

Beispielsweise führt der Konzern in Abbildung 4.2 im Zuge der Restrukturierung die folgende konzernweite Compliance-Regel für den Mitarbeitereinstellungs-Prozess ein um Verstößen gegen das Bundesdatenschutzgesetz und Know-How-Verlusten an Mitbewerber vorzubeugen: „*Jeder neue Mitarbeiter muss nach seiner Einstellung über Informations- und Datenschutz im Unternehmen unterwiesen werden*“.

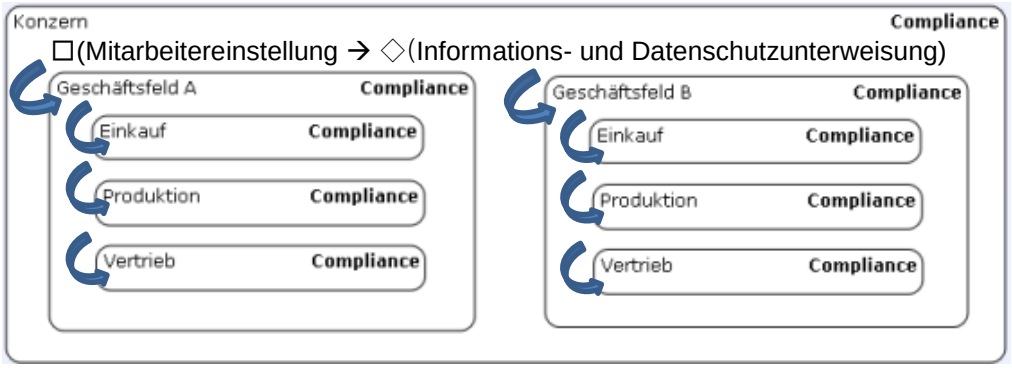


Abbildung 4.2: Anwendungsbeispiel Compliance-Regeln

Wenn keine der Abteilungen diese Regel selbst spezifiziert, kann ihre Verletzung zwar durch Model-Checking aller Geschäftsprozessesse erkannt werden. Die Modellierung des Geschäftsprozesses einer Abteilung kann jedoch zum Zeitpunkt der Fehlerentdeckung weit fortgeschritten sein. Daher ist eine frühe Fehlervermeidung wünschenswert (vgl. Abschnitt 2.1.1.2). Es kann ein Prozess-Template vorgestellt werden, welches wiederum Prozess-Templates für geschäftsfeldspezifische Mitarbeitereinstellungs-Prozesse enthält. Diese sind zwar unabhängig voneinander, müssen jedoch alle die oben genannte Regel erfüllen. Dazu wird die Regel automatisch an sie weitergegen.

Des Weiteren ist es möglich, dass eine Abteilung etwas Gegensätzliches zur konzernweiten Regel spezifiziert. Beispielsweise kann eine weitere konzernweite Compliance Regel aus dem Bereich Datenschutz lauten [Rös09]: „Wenn ein Mitarbeiter ausritt, dann muss sein E-Mail-Postfach gelöscht werden.“ In LTL:

$$\Box(\text{Mitarbeiter tritt aus} \rightarrow \Diamond \text{E-Mail-Postfach löschen})$$

Eine dazu widersprüchliche Compliance-Regel könnte sein: „Wenn ein Mitarbeiter ausritt, dann muss sein E-Mail-Postfach archiviert werden.“ In LTL:

$$\Box(\text{Mitarbeiter tritt aus} \rightarrow \neg\Diamond \text{E-Mails-Postfach archivieren})$$

Durch die automatische Weitergabe der Geschäftsfeldregel an die Abteilungen und anschließende Erfüllbarkeitsprüfung mit den jeweiligen Abteilungsregeln können frühzeitig Inkonsistenzen entdeckt werden.

Eine interne Compliance-Regel in einem Dienstleistungsunternehmen, welches für seine Kunden große Projekte organisiert, könnte wie folgt formuliert sein: „In allen Projektabwicklungsprozessen muss gelten, dass, wenn eine Angebotsanfrage empfangen wird, kein Angebot verschickt wird bevor es nicht von allen Verantwortlichen geprüft wurde.“ In LTL:

$$\Box(\text{Angebotsanfrage empfangen} \rightarrow (\neg\text{Angebot schicken } U \text{ Angebot prüfen}))$$

4.2 Weitergabe von Compliance-Regeln

Die Weitergabe von Compliance-Regeln an Unterprozesse im Rahmen des inkrementellen Entwicklungsprozesses in [SALS10] wurde auf der Grundlage von Aussagenlogik eingeführt. Um die dort definierten direkten und indirekten Konflikte (siehe Definition 3.2 und Definition 3.1) auf die LTL (siehe Abschnitt 2.2.1) übertragen zu können, müssen zunächst für die positiven und negativen Literale Entsprechungen in der LTL gefunden werden. Statt Literalen werden Teilregeln von Compliance-Regeln betrachtet.

4.2.1 Teilregeln

In [SALS10] liegen die Compliance-Regeln in der KNF (siehe Abschnitt 2.2.1.4) vor, so dass die Klauseln die Teilregeln darstellen, die an Unterprozesse weitergegeben werden können. Die Klauseln der Normalform SNF (siehe Definition 2.5) können auch als solche Teilregeln in der LTL aufgefasst werden. In diesem Fall müssen alle Compliance-Regeln vom Benutzer entweder bereits in der SNF eingegeben werden oder automatisch in SNF umgeformt werden. Die Umformung hat jedoch den Nachteil, dass im Falle eines Konflikts (siehe Abschnitt 3.3) zwischen Compliance-Regeln in SNF die zu korrigierende Teilregel in ihrer ursprünglich vom Benutzer eingegebenen Form schwer erkennbar sein kann. Beispielsweise besteht die einfache Formel □a in SNF aus drei Klauseln: □((Start→x) ∧ (x→a) ∧ (x→○a)). Außerdem müssen in diesem Fall die entstehenden Klauseln als eine Einheit betrachtet werden. Obwohl das obige Beispiel als □(Start→x) ∧ □(x→a) ∧ □(x→○a) geschrieben werden kann, dürfen die Klauseln nicht getrennt an innere Compliance-Scopes weitergegeben werden, weil sie nur zusammen äquivalent zu der ursprünglichen Formel □a sind. Außerdem scheidet die Verwendung der SNF aus, weil der verwendete Prototyp den ○-Operator (Next-Operator, siehe Abschnitt 2.2.1) nicht unterstützt.

Alternativ kann eine Art KNF auch für die LTL verwendet werden, in der Teilregeln durch den logischen UND-Operator verknüpft sind. Der zu erweiternde Prototyp bietet bereits die Möglichkeit eine solche Regel im Compliance Wizard (siehe Abbildung 3.6) zu erstellen, indem auf der obersten Ebene des Regelbaums (siehe Abbildung 3.6) ein UND-Operator verwendet wird. Die Operanden dieses UND-Operators bilden somit die Teilregeln, die einzeln an innere Compliance-Scopes weitergegeben werden können. Es ist jedoch zu beachten, dass im Gegensatz zu einer echten KNF in den „Klauseln“ weitere UND-Operatoren und in den Blättern beliebig komplexe LTL-Formeln verwendet werden können. Damit können sehr komplexe Teilregeln entstehen.

4.2.2 Erfüllte Teilregeln

Die erfüllten Teilregeln können durch das im verwendeten Prototyp bereits implementierte Model-Checking erkannt werden. Das Model-Checking im Prototyp erfolgt nur für die LTL-Formeln an den Blättern des Regelbaums (siehe Abbildung 3.6) und das Gesamtergebnis wird entsprechend der logischen Semantik der Operatoren an den Knoten in der Wurzel berechnet (siehe Abschnitt 3.4.3). Daher kann die vorhandene Routine des Model-Checking genutzt werden und die Model-Checking-Ergebnisse für einzelne Teilregeln abgegriffen werden.

4.2.3 Positive und negative Teilregeln

Es stellt sich die Frage, wie positive und negative Literale aus [SALS10] (siehe Abschnitt 3.3) auf die LTL übertragen werden können. Sind beispielsweise (a *U* b) und □(a→◇b) positive und ihre Negationen entsprechend negative Eigenschaften? Die Unterscheidung in positive und negative Eigenschaften ist notwendig, weil positive erfüllte Eigenschaften nicht an Unterprozesse weitergegeben werden müssen (siehe Abbildung 3.3). Im Folgenden wird zunächst die grundlegende Problematik diskutiert und anschließend die in dieser Arbeit geltenden Definitionen für positive und negative Teilregeln angegeben.

Das positive Literal A in der Compliance-Regel in Abbildung 3.1 bedeutet, dass irgendwann garantiert A vorkommen muss. Dies entspricht in LTL der Lebendigkeitseigenschaft (siehe Abschnitt 2.2.1.3.2) ◇a. Entsprechend kann das negative Literal ¬B in LTL als die Sicherheitseigenschaft (siehe Abschnitt 2.2.1.3.1) ¬◇b verstanden werden, was äquivalent zu □¬b ist. Beispielsweise gelten diese einfachen Beispielformeln in dem Prozess in Abbildung 4.3.

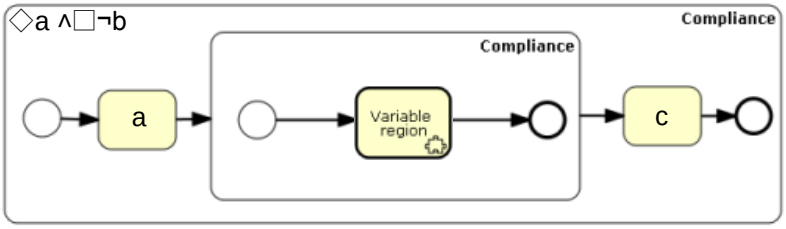


Abbildung 4.3: Einfache positive und negative Eigenschaften

4.2.3.1 Scope-übergreifende Erfüllung

Der grundlegende Unterschied der LTL zur Aussagenlogik besteht darin, dass sich eine LTL-Formel aufgrund der Zeitoperatoren nicht nur auf einen Zustand sondern auf eine Folge von Zuständen oder Zeitpunkten beziehen kann (siehe Abbildung 2.7), wie z. B. das obige Beispiel □¬b. Im Folgenden werden weitere Beispiele erläutert.

So bedeutet die Lebendigkeitseigenschaft ◇□c, dass ab einem zukünftigen Zustand bis zum Endzustand c=*true* gelten muss. Das bedeutet, dass sich ein Teil der Zustandssequenz mit c=*true* in einem inneren Compliance-Scope befinden kann, sodass die Regel Scope-übergreifend erfüllt wird. Die Formel ◇(d *U* c) bedeutet, dass in einem der zukünftigen Zustände c=*true* gilt und es unmittelbar davor eine Zustandssequenz mit d=*true* gibt. Beispielsweise kann ◇(d *U* c) in Abbildung 4.3 dadurch erfüllt werden, dass sich die Zustandssequenz mit a=*true* im inneren Compliance-Scope befindet. Ähnliches gilt für die Formel a *U* d, die sich auf eine Zustandsfolge vom Startzustand bis zu einem Zustand mit d=*true* bezieht. Beispielsweise muss in Abbildung 4.3 im inneren Compliance-Scope die Aktivität d als erste Aktivität vorkommen damit a *U* d erfüllt wird.

4.2.3.2 Grundlegende Gültigkeitsbereiche

Ausgehend von eigenen Beobachtungen und den temporalen Gültigkeitsbereichen aus [DAC98] (siehe Abbildung 2.7) werden folgende grundlegende Gültigkeitsbereiche von Formeln unterscheiden: (1) Formeln, die sich auf alle Zeitpunkte beziehen, insbesondere Sicherheitseigenschaften; (2) Formeln die sich auf diskrete Zeitpunkte beziehen. Das können z. B. Lebendigkeitseigenschaften sein; Und Formeln, die sich auf Zustandsfolgen (3) inklusive des Startzustands oder (4) inklusive des Endzustands oder (5) exklusive des Start- und Endzustands beziehen. Diese Gültigkeitsbereiche sind in Abbildung 4.4 mit Beispielen dargestellt.

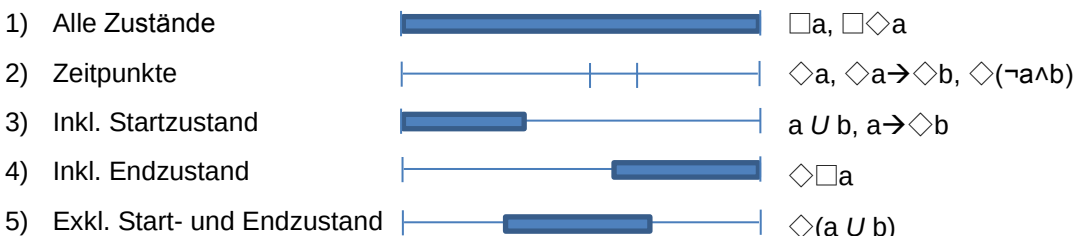


Abbildung 4.4: Grundlegende Gültigkeitsbereiche von LTL-Formeln, in Anlehnung an [DAC98]

Bei den letzten drei Gültigkeitsbereichen stellt sich die Frage, wie Beginn und Ende ihrer Gültigkeitsbereiche erkannt werden können. Beispielsweise kann es bei dem vierten Gültigkeitsbereich am Anfang des Prozesses einen Compliance-Scope geben, an den die Compliance-Regel ◇□a nicht weitergegeben werden darf. Denn ◇□a bezieht sich nur auf ihren Compliance-Scope und soll nicht am Ende von jedem inneren Compliance-Scope gelten. Es sei denn, es wird explizit gewünscht, dass ◇□a in jedem inneren Compliance-Scope gelten soll.

Beispielsweise ist die Regel ◇□a in Abbildung 4.3 bereits erfüllt und es ist unklar, ob sie auch in dem inneren Compliance-Scope erfüllt sein muss.

4.2.3.3 Ansatz zur Erkennung positiver und negativer Compliance-Regeln

Für ein positives Literal in [SALS10] ist charakteristisch, dass es an die Unterprozesse nicht mehr weitergegeben werden muss, wenn es einmal erfüllt ist, da es in den Unterprozessen nicht mehr verletzt werden kann. Dies gilt beispielsweise auch für ◇a in Abbildung 4.3. Ein negatives Literal muss dagegen immer weitergegeben werden, weil es überall im Prozess verletzt werden kann. Dies gilt auch für die Regel □¬b in Abbildung 4.3. Um solche Eigenschaften zu unterscheiden können die ihnen entsprechenden Büchi-Automaten (siehe Abschnitt 2.2.1.1) untersucht werden. Dabei lässt sich feststellen, dass die Büchi-Automaten für Lebendigkeitseigenschaften (siehe Abschnitt 2.2.1.3.2) wie ◇¬a den „accept_all"-Zustand enthalten (siehe
Listing 2.2). Die Sicherheitseigenschaften enthalten diesen Zustand dagegen nicht (siehe
Listing 2.1). Der „accept_all"-Zustand (vgl. 2.3.2.2.2) bedeutet, dass es einen Zustand gibt, ab dem die Eigenschaft unabhängig vom weiteren Verlauf erfüllt ist. Das Fehlen des „accept_all"-Zustands bedeutet, dass die geforderte Eigenschaft in allen Zuständen erfüllt sein muss. Dies entspricht dem Verständnis von positiven und negativen Literalen. Daher basiert die Erkennung positiver und negativer Teilregeln in dieser Arbeit auf den folgenden Definitionen:

Definition 4.1 (Positive Teilregel): Eine *positive Teilregel* ist eine LTL-Formel deren mit SPIN generierter Büchi-Automat einen „accept_all"-Zustand enthält.

Definition 4.2 (Negative Teilregel): Eine *negative Teilregel* ist eine LTL-Formel deren mit SPIN generierter Büchi-Automat keinen „accept_all"-Zustand enthält.

Im Allgemeinen sind diese Definitionen nicht an SPIN gebunden, weil die Büchi-Automaten auch mit anderen Tools generiert werden können (siehe Abschnitt 2.2.1.1). Beispiele für Büchi-Automaten mit und ohne des „accept_all"-Zustands sind in Tabelle A. 1 und Tabelle A. 2 zu finden.

Beispiel (Negative Teilregel, Gültigkeitsbereich 1): Sicherheitseigenschaft □¬(a ∧ b). Es gibt keinen „accept_all"-Zustand im Büchi-Automat, weil in allen Zuständen ¬(a ∧ b) erfüllt sein muss.

Beispiel (Negative Teilregel, Gültigkeitsbereich 1): Die Lebendigkeitseigenschaft □◇a. Es gibt keinen „accept_all"-Zustand im Büchi-Automat, weil in allen Zuständen ◇a erfüllt sein muss.

Beispiel (Negative Teilregel, Gültigkeitsbereich 4): Zum Erkennen der Lebendigkeitseigenschaft ◇□a muss der Büchi-Automat alle Zustände vom Start- bis zum Endzustand prüfen. Daher gibt es keinen „accept_all"-Zustand im Büchi-Automat.

Beispiel (Positive Teilregel, Gültigkeitsbereich 2): Lebendigkeitseigenschaft ◇¬(a ∧ b). Ab einem Zustand im Modell, in dem a und b nicht gleichzeitig erfüllt sind, werden alle weiteren Modellzustände durch den „accept_all"-Zustand des Büchi-Automaten akzeptiert.

Beispiel (Positive Teilregel, Gültigkeitsbereich 2): Lebendigkeitseigenschaft ◇(a →◇b) gibt an, dass, falls irgendwann a eintritt, muss in demselben oder einem der darauffolgenden Zustände b eintreten. Hier sind es ein oder zwei Zustände, die zur Erfüllung beitragen. Nachdem b erfüllt ist, werden alle weiteren Modellzustände durch den „accept_all"-Zustand im Büchi-Automat akzeptiert.

Beispiel (Positive Teilregel, Gültigkeitsbereich 3): a *U* b. Wenn in dem Startzustand a=*true* gilt, akzeptiert der „accept_all"-Zustand des Büchi-Automaten, alle weiteren Modellzustände, wenn in allen vorhergehenden Zuständen a erfüllt ist.

Beispiel (Positive Teilregel, Gültigkeitsbereich 5): ◇(a *U* b) ist in einer Zustandssequenz erfüllt, in der a *U* b gilt. Nach Erkennung dieser Zustandssequenz akzeptiert der „accept_all"-Zustand des Büchi-Automaten alle weiteren Modellzustände.

4.2.3.4 Beobachtungen

Anhand obiger Beispiele lässt sich feststellen, dass die Teilregel in Einklang mit den vorangegangenen Definitionen genau dann positiv ist, wenn es nach dem Gültigkeitsbereich der Teilregel weitere Zustände gibt. Anderenfalls ist sie negativ. Dies wird in Abbildung 4.5 zusammengefasst.

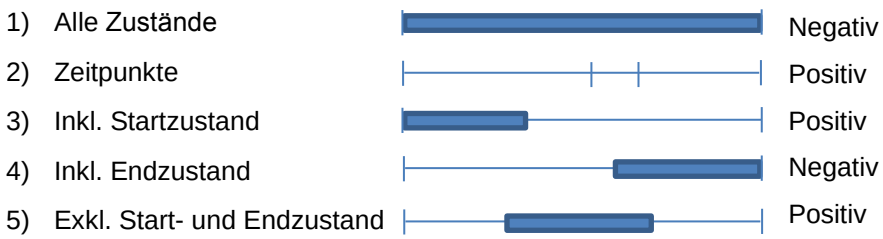


Abbildung 4.5: Positive und negative Teilregeln nach Gültigkeitsbereichen, nach [DAC98]

Eine weitere Beobachtung ist, dass es für LTL-Formeln mit den Gültigkeitsbereichen 2-5 sinnvoll sein kann anzugeben, ob sie nur in ihrem Compliance-Scope, in allen Compliance-Scopes oder nur in ausgewählten Compliance-Scopes gelten sollen.

Die Weitergabe von Compliance-Regeln wird im Folgenden anhand von Beispielen erläutert.

4.2.4 Direkte Konflikte

Gemäß [SALS10] liegt ein direkter Konflikt liegt vor, wenn eine negative Teilregel (siehe Definition 4.2) eines Compliance-Scopes zu ihrer negierten Form aus einem inneren Compliance-Scope im Widerspruch steht. Eine negative Teilregel wird dabei immer an die inneren Compliance-Scopes weitergegeben, weil sie in jedem Zustand innerhalb des Compliance-Scope erfüllt sein muss.

Beispielsweise steht in der Abbildung 4.6 die Lebendigkeitseigenschaft ◇b des inneren Compliance-Scopes zu der Sicherheitseigenschaft □¬b (äquivalent zu ¬◇b) des äußeren Compliance-Scopes im Widerspruch. In diesem Fall wird durch die Erfüllbarkeitsprüfung von □¬b ∧ ◇b die Inkonsistenz festgestellt.

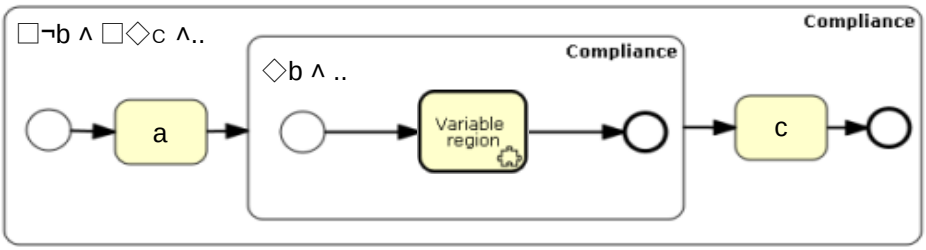


Abbildung 4.6: Direkter Konflikt

Die Compliance-Regel □◇c wird nach der Definition 4.2 als eine negative Teilregel erkannt und daher an den inneren Compliance-Scope weitergegeben. Jedoch bleibt sie auch dann erfüllt, wenn keine Aktivität c im inneren Compliance-Scope vorkommt.

4.2.5 Indirekte Konflikte

Entsprechend [SALS10] liegt ein indirekter Konflikt vor, wenn eine positive Teilregel eines Compliance-Scope zu ihrer negierten Form aus einem inneren Compliance-Scope im Widerspruch steht. Dabei werden positive Eigenschaften nicht weitergegeben, wenn sie bereits erfüllt ist.

4.2.5.1 Erfüllte positive Teilregeln

Das Beispiel in Abbildung 4.7 zeigt drei Teilregeln (a *U* b), ◇c und ◇(d *U* e) aus allen drei Gültigkeitsbereichen für positive Eigenschaften (siehe Abbildung 4.5). Die innere Compliance-Regel (¬◇c ∧¬◇b ∧¬◇e) steht dabei zu allen Teilregeln im Widerspruch. Da jedoch alle drei Regeln erfüllt und aufgrund des „accept_all"-Zustand in ihren Büchi-Automaten als positiv erkannt werden, werden sie nicht an den inneren Compliance-Scope weitergegeben. Somit sind abweichende Regeln in dem inneren Compliance-Scope erlaubt.

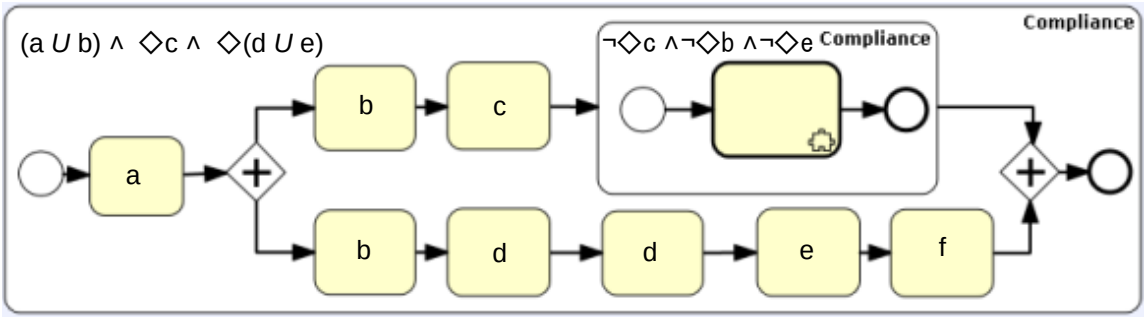


Abbildung 4.7: Indirekte Konflikte (erfüllte positive Teilregeln)

4.2.5.2 Nicht erfüllte positive Teilregeln

Der äußere Compliance-Scope in Abbildung 4.8 enthält drei unerfüllte Teilregeln mit den drei möglichen Gültigkeitsbereichen für positive Eigenschaften (siehe Abbildung 4.5). Beispielsweise wird die Teilregel ◇c an die inneren Compliance-Scopes weitergegeben. Die Erfüllbarkeitsprüfung der inneren Compliance-Regel (¬◇c ∧¬◇b ∧¬◇e) in Verknüpfung mit ◇c ergibt eine Inkonsistenz, sodass die innere Regel unter Umständen korrigiert werden muss. Ein indirekter Konflikt bedeutet jedoch nicht immer, dass eine innere Compliance-Regel korrigiert werden muss. Denn es ist möglich, dass in dem inneren Compliance-Scope die zu den äußeren Regeln widersprüchliche Teilregel tatsächlich gelten muss. In diesem Fall kann auch der zweite innere Compliance-Scope ◇c erfüllen.

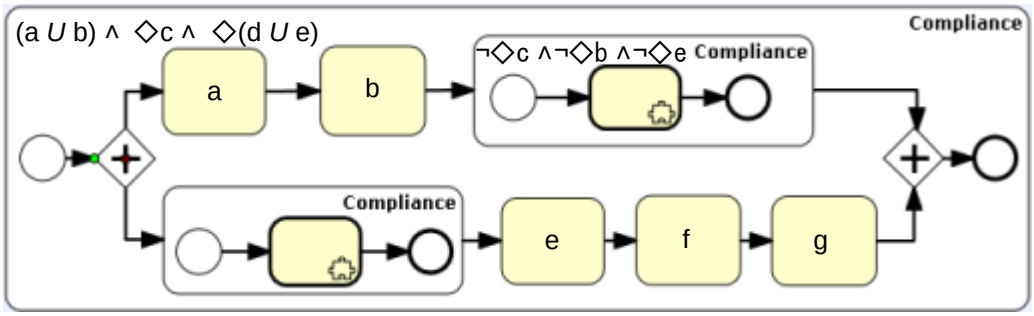


Abbildung 4.8: Indirekte Konflikte (unerfüllte positive Teilregeln)

Die Teilregel ◇(d *U* e) ist nicht erfüllt, weil vor dem Task e noch keine Zustandssequenz mit d=*true* existiert. Auch diese Teilformel kann an alle inneren Compliance-Scopes weitergegeben werden und im weiteren inkrementellen Entwicklungsverlauf in einem von ihnen erfüllt werden.

Die Teilregel (a *U* b) ist nicht erfüllt, weil der untere parallele Pfad weder mit a noch mit b beginnt (vgl. Definition 2.3). Diese Regel hat den Gültigkeitsbereich, der sich auf den Startzustand bezieht (siehe Abbildung 4.4). Der Startzustand wird in diesem Fall von einer der ersten Aktivitäten von den beiden parallelen Pfaden bestimmt. Da die relative Ausführungsreihenfolge paralleler Aktivitäten in realen Geschäftsprozessen in der Regel unbestimmt [For02] ist, muss der untere Pfad diese Teilregel auch erfüllen. Das heißt, sie muss an den unteren Compliance-Scope weitergegeben werden. Es ist jedoch fragwürdig, ob sie auch in dem oberen und in allen anderen möglichen Compliance-Scopes berücksichtigt werden muss, die den Startzustand nicht einbeziehen.

4.2.6 Potentielle Konflikte

Aufgrund von ODER-Operatoren in inneren oder äußeren Compliance-Scopes können einige Konflikte durch Erfüllbarkeitsprüfung von verknüpften Compliance-Regeln nicht entdeckt werden.

4.2.6.1 Disjunktion in einer inneren Compliance-Regel

Beispielsweise bleibt in Abbildung 4.9 der Konflikt zwischen ¬◇e und ◇e durch Erfüllbarkeitsprüfung unentdeckt, weil ¬◇e ∧ (◇b ∨ ◇e) erfüllbar ist. Jedoch wird das Model-Checking den Berechnungspfad über die Aktivität e entdecken und als Gegenbeispiel ausgeben.

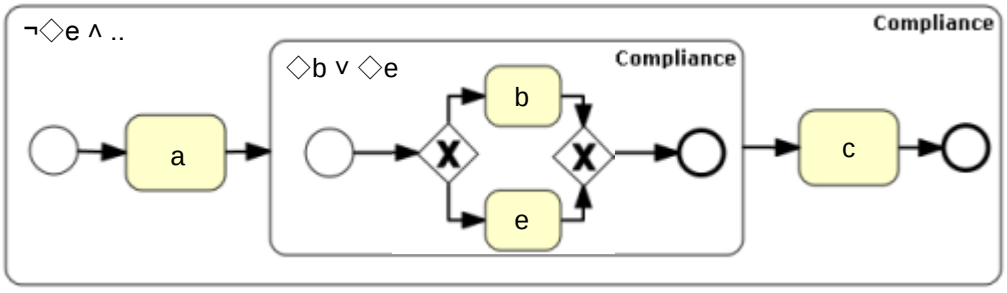


Abbildung 4.9: Disjunktion in einer inneren Compliance-Regel

Die Inkonsistenz im oberen Beispiel kann in der realen Prozessmodellierung unter Umständen sehr spät erkannt werden. Wenn der Modellierer des inneren Compliance-Scopes von der Konsistenz seiner Compliance-Regel mit der äußeren Regel überzeugt ist, wird er seinen Prozess entsprechend seiner Compliance-Regel weiter modellieren. Es kommt hinzu, dass zwischen den

dargestellten Compliance-Scopes weitere Schichten von Compliance-Scopes möglich sind, sodass der Fehler umso später entdeckt wird, je mehr zwischenliegende Compliance-Scopes existieren. Denn während die Compliance-Regeln in realen Entwicklungsprozessen den Designern z. B. zugeschickt werden können, müssen für das Model-Checking erst alle Modelle fertig werden.

4.2.6.2 Unerfüllte Disjunktion von positiven Teilformeln

Der äußere Compliance-Scope in Abbildung 4.10 enthält die positive Teilregel (◇e ∨ ◇b) als Disjunktion zweier positiver Teilformeln. Da diese Teilregel nicht erfüllt ist, wird sie an den inneren Scope weitergegeben. Die Inkonsistenz von ◇e und ¬◇e wird durch die Erfüllbarkeitsprüfung von (◇e ∨ ◇b) ∧ ◇e nicht entdeckt, weil diese Formel erfüllbar ist. Daher wird der Modellierer des inneren Scopes ein aus Sicht des äußeren Scopes nicht Compliance-konformes Modell erstellen. Durch das Model-Checking wird die Nichterfüllung der äußeren Teilregel zwar erkannt. Bei vielen verschachtelten Scopes kann es jedoch nicht direkt ersichtlich sein, in welcher Compliance-Regel der Fehler liegt. Der Model-Checker wird zwar einen Pfad als Gegenbeispiel ausgeben, doch dies ist ein Schleife oder ein Pfad vom Startereignis bis zum Endereignis, weil es sich hier um eine Lebendigkeitseigenschaft handelt (siehe Abschnitte 2.2.1.3.2 und 2.3.2.2.1). Des Weiteren kann es mehrere solcher Pfade geben, auf denen die Teilregel nicht erfüllt ist.

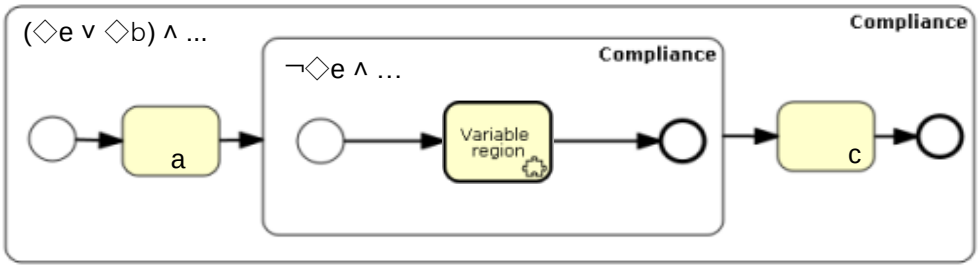


Abbildung 4.10: Unerfüllte Disjunktion von positiven Teilformeln

4.2.6.3 Unerfüllte Disjunktion von negativen Teilformeln

Da die Teilregel (¬◇e ∨ ¬◇b) in Abbildung 4.11 negativ ist (siehe Büchi-Automat in Listing A. 1), wird sie an den inneren Compliance-Scope weitergegeben. Der Konflikt zwischen ¬◇e und ◇e wird durch Erfüllbarkeitsprüfung von (¬◇e ∨ ¬b) ∧ ◇e nicht entdeckt. Diese Teilregel ist äquivalent zu ¬(◇e ∧ ◇b). Das heißt, sie wird verletzt, wenn im Prozess sowohl e also auch b vorkommen.

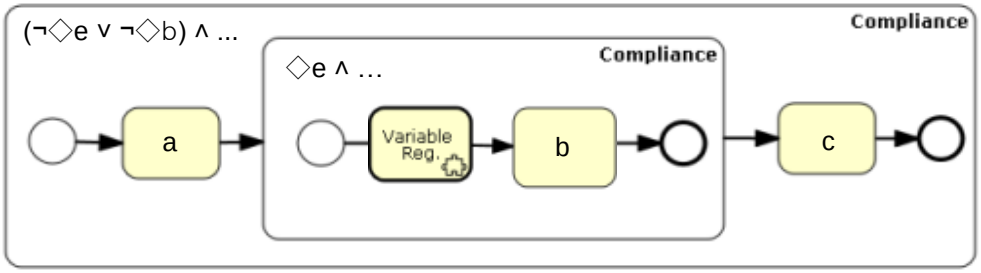


Abbildung 4.11: Unerfüllte Disjunktion von negativen Teilformeln

Da in dem Prozess die Aktivität b vorkommt und e nicht vorkommt, ist die Compliance-Regel zwar erfüllt, aber aufgrund der inneren Regeln ◇e wird der Modellierer die Aktivität e einfügen. Daher wird erst beim nächsten Durchlauf des Model-Checking dieser Fehler entdeckt. Da es sich in diesem Fall um eine Sicherheitseigenschaft handelt, ist das Gegenbeispiel ein endlicher Pfad (siehe Abschnitte 2.2.1.3.1 und 2.3.2.2.2), der genau zu der verletzenden Aktivität führt. Es kann

jedoch mehrere innere Compliance-Scopes geben, in denen jeweils die Aktivität e oder b vorkommt. Um solche Schwierigkeiten zu vermeiden, ist es denkbar, die Operanden einer Disjunktion in einer Teilregel einzeln auf Erfüllbarkeit mit den inneren Compliance-Regeln zu prüfen.

4.2.6.4 Unerfüllte Disjunktion von negativen und positiven Teilformeln

Im dem Modell in Abbildung 4.12 wird der Konflikt zwischen ¬◇e und ◇e durch die Erfüllbarkeitsprüfung von (¬◇e ∨ ◇b) ∧ ◇e nicht entdeckt, weil diese Formel erfüllbar ist. Dadurch können ähnliche Schwierigkeiten wie in den Abschnitten 4.2.6.2 und 4.2.6.3 entstehen.

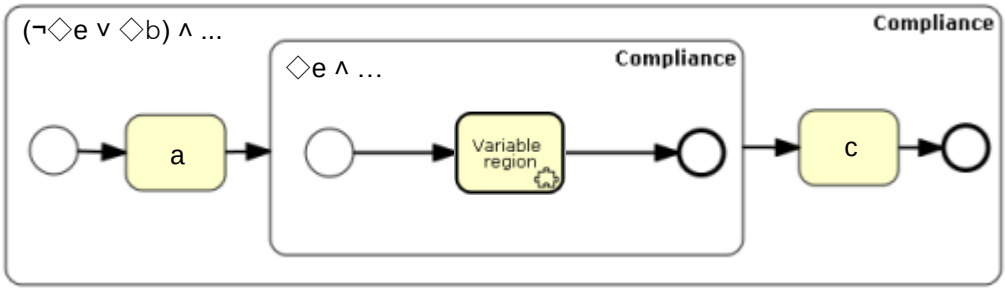


Abbildung 4.12: Unerfüllte Disjunktion von negativen und positiven Teilformeln

Aufgrund der positiven Teilformel ◇b enthält der Büchi-Automat der Teilregel (¬◇e ∨ ◇b) einen „accept_all"-Zustand (siehe Büchi-Automat in Listing A. 2). Daher ist diese Teilregel laut Definition 4.1 positiv. In Abbildung 4.12 ist erkennbar, dass diese Teilregel erfüllt ist, weil keine Aktivität e im Prozessmodell vorkommt. Damit die Teilregel im gesamten Prozess erfüllt wird, muss die Teilformel ¬◇e (≡□¬e) in allen Compliance-Scopes erfüllt sein. Die Teilregel (¬◇e ∨ ◇b) muss daher weitergegeben werden, obwohl sie insgesamt positiv und erfüllt ist. Dies kann dadurch gelöst werden, dass aus Disjunktionen bestehende Teilregeln nicht weitergegeben werden, wenn sie eine positive erfüllte Teilformel enthalten.

4.2.6.5 Umgang mit unerfüllten Disjunktionen

Aus den obigen Beispielen folgt: Wenn die Erkennung positiver und negativer Teilformeln, die Disjunktionen sind, nur anhand des Gesamtergebnisses erfolgt, können falsche Entscheidungen getroffen werden. Vielmehr muss jeder Operand untersucht werden. Nur wenn eine positive erfüllte Teilformel entdeckt wird, muss die gesamte Teilregel nicht weitergegeben werden. Das heißt, aufgrund eine einzigen positiven Teilformel sollte die gesamte Disjunktion als eine positive Teilformel behandelt werden.

4.2.7 Grenzen der Methode und Lösungsansätze

Mit der oben beschriebenen Methode zur Erkennung von Regeln und Teilregeln, die an innere Compliance-Scopes nicht weitergegen werden dürfen, kann nicht immer die richtige Entscheidung getroffen werden. Genauer formuliert, liegt die Schwierigkeit in der Erkennung von Formeln die nicht weitergegeben werden müssen, wenn sie bereits erfüllt sind.

4.2.7.1 Global erfüllte negative Regeln und lokale Inkonsistenz

Eine innere Compliance-Regel muss nicht immer mit der des äußeren konsistent sein. Beispielsweise bedeutet die Fairnesseigenschaft (siehe Abschnitt 2.2.1.3.4) □◇a → □◇b in Abbildung 4.13, dass unendlich oft a und unendlich oft b vorkommen müssen, wenn unendlich oft a vorkommt. Nach der Definition 4.2 wird diese Compliance-Regel als eine *negative Regel* erkannt

und daher an den inneren Compliance-Scope in Abbildung 4.13 weitergegeben. Im inneren Compliance-Scope muss □◇a ∧ □¬b gelten, was im Widerspruch zu der Fairnesseigenschaft steht. Trotz der lokalen Inkonsistenz bleibt die äußere Compliance-Regel durch die Aktivitäten a und b erfüllt.

Das bedeutet, dass es Compliance-Regel mit einem globalen Charakter gibt, die nicht von den Aktivitäten innerer Compliance-Scopes verletzt werden können. Im Vergleich dazu kann die Regel □(c→d) in jedem Compliance-Scope verletzt werden.

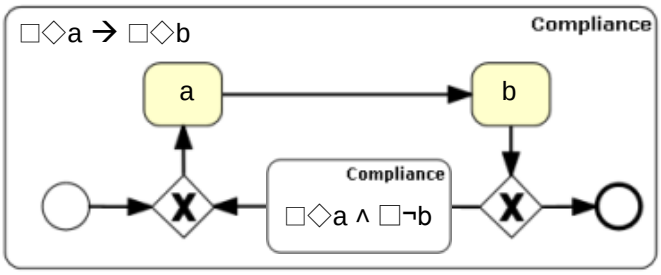


Abbildung 4.13: Global erfüllte Compliance-Regel und lokale Inkonsistenz

4.2.7.2 Gültigkeitsbereich inklusive des Endzustands

Die äußere Regel in Abbildung 4.14 wird als negativ erkannt (Definition 4.2), weil der Büchi-Automat für ◇□a keinen „accept_all"-Zustand enthält (siehe Tabelle A. 1). Daher wird sie an den inneren Compliance-Scope weitergegeben, in dem die zu ihr widersprüchliche Spezifikation ¬◇a angegeben ist. Damit wird durch die Erfüllbarkeitsprüfung die Inkonsistenz der inneren Regel zur Äußeren bemängelt. Dabei kann es sein, dass in dem dargestellten inneren Compliance-Scope tatsächlich keine Aktivität a vorkommen darf.

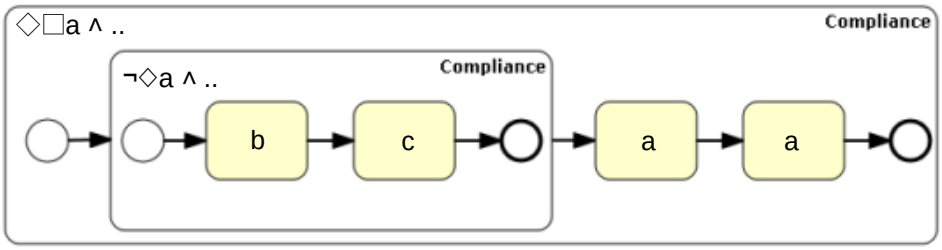


Abbildung 4.14: Scope-übergreifende Erfüllung

Falls die beiden letzten Aktivitäten in Abbildung 4.14 in einem inneren Compliance-Scope eingeschlossen wären, müsste die Regel ◇□a in jedem Fall an diesen Compliance-Scope weitergegeben werden. Solche Formeln die unmittelbar am Ende oder am Anfang eines Prozesses gelten müssen, können z. B. durch Mustererkennung mittels regulärer Ausdrücke erkannt werden und automatisch nur an die entsprechenden Compliance-Scopes weitergegeben werden.

4.2.7.3 Lösungsansätze

Markierung lokal geltender Teilregeln

Eine mögliche Lösung für global erfüllte negative Regeln (Abschnitt 4.2.7.1) ist die Markierung von Teilregeln, die niemals an innere Compliance-Scopes weitergegeben werden müssen, was mit manuellem Aufwand verbunden ist. Wenn jedoch eine Teilregel nicht automatisch weitergegeben wird, wird ihre Erfüllung in den relevanten Compliance-Scopes nicht gefördert. Daher können

solche Regeln zusätzlich in Teilregeln aufgeteilt und an die relevanten inneren Compliance-Scopes verteilt werden. Beispielsweise können in Abbildung 4.13 statt der Aktivitäten a und b zwei Compliance-Scopes modelliert und ihnen jeweils die Regeln ◇a und ◇b zugeordnet werden.

Eingrenzung des Gültigkeitsbereichs

Es bietet sich an die Scope-übergreifend erfüllbaren Compliance-Regeln nur solchen Compliance-Scopes zuzuordnen, die genau ihrem Gültigkeitsbereich entsprechen. Dies betrifft Formeln mit den letzten drei Gültigkeitsbereichen in Abbildung 4.4. Dazu kann eine LTL-Formel dieser Gültigkeitsbereiche in eine LTL-Formel mit dem ersten Gültigkeitsbereich umgeformt werden. Aus ◇□a wird beispielsweise □a.

4.3 Erweiterung der Compliance-Prüfung

Die bisher aus dem unabhängigen Model-Checking einzelner Compliance-Scopes bestehende Compliance-Prüfung (siehe Abschnitt 3.4.3), wird laut dem im Abschnitt 3.3 vorgestellten *inkrementellen* Entwicklungsprozess um die Konsistenzprüfung von Compliance-Regeln erweitert. Dabei wird die Erfüllbarkeit von Compliance-Regeln in Verknüpfung mit ihren äußeren Compliance-Regeln sichergestellt. Wie im Abschnitt 3.2 beschrieben, muss auch für einzelne Compliance-Regeln sichergestellt werden, dass sie nicht gültig sind. Dazu muss nach Satz 2.1 die Erfüllbarkeit der negierten Compliance-Regel gezeigt werden.

Alle nötigen Erfüllbarkeits- und Gültigkeitsprüfung können zwar während einem Durchlauf der Compliance-Prüfung erfolgen. Im Folgenden wird in Bezug auf den vorhandenen Prototyp erläutert, wann und welche Erfüllbarkeits- und Gültigkeitsprüfungen im Hinblick auf eine möglichst effiziente Compliance-Prüfung stattfinden sollen.

4.3.1 Erfüllbarkeits- und Gültigkeitsprüfung von LTL-Formeln

Im Falle einer unerfüllbaren Compliance-Regel ist der Grund dafür nicht direkt ersichtlich. Ein Grund könnte z. B. eine unerfüllbare Teilregel oder Teilformel sein. Um die Fehlersuche im Falle einer unerfüllbaren Compliance-Regel zu erleichtern, soll die Erfüllbarkeits- und Gültigkeitsprüfung (siehe Abschnitt 2.4) einzelner LTL-Formeln während ihrer Erstellung im grafischen LTL-Editor erfolgen. Damit wird sichergestellt, dass die Unerfüllbarkeit einer Compliance-Regel nur auf die Kombination verwendeter LTL-Formeln und logischer Operatoren in den Knoten des Regelbaums zurückzuführen ist.

4.3.2 Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln

Um die Fehlersuche bei unerfüllbaren verknüpften Compliance-Regeln zu erleichtern, soll die Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln während oder nach ihrer Erstellung im Compliance Wizard stattfinden. Es soll nicht möglich sein unerfüllbare oder gültige Compliance-Regeln zu speichern. Damit wird sichergestellt, dass die Unerfüllbarkeit einer verknüpften Compliance-Regel nur auf die Und-Verknüpfung der Compliance-Regeln zurückzuführen ist.

4.3.3 Gültigkeitsprüfung von Teilregeln

Im Falle von positiven erfüllten Teilregeln werden die restlichen Teilregeln an innere Compliance-Scopes weitergegeben (siehe Abschnitt 4.2.5.1). Obwohl die Ungültigkeit der Compliance-Regeln

sichergestellt ist (siehe Abschnitt 4.3.2), können ihre Teilregeln laut Satz 2.2 gültig sein. Daher muss auch die Ungültigkeit einzelner Teilregeln geprüft werden. Die Abbildung 4.15 zeigt den Regelbaum für die erfüllbare und ungültige Compliance-Regel ◇b ∧ (◇a ∨ ¬◇a). Dabei ist die Teilregel (◇a ∨ ¬◇a) das einfachste Beispiel für eine gültige Formel.

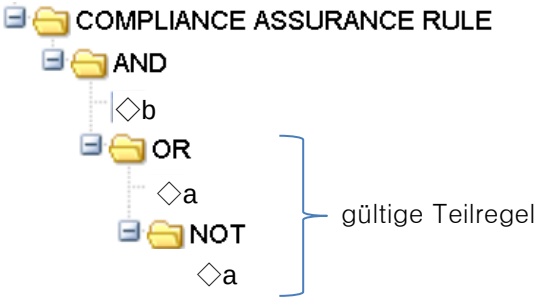


Abbildung 4.15: Gültigkeit von Teilformeln (Beispiel)

Da die Erfüllbarkeit von Compliance-Regeln sichergestellt ist (siehe Abschnitt 4.3.2), ist die Erfüllbarkeitsprüfung von Teilregeln laut Satz 2.3 nicht erforderlich.

4.3.4 Konsistenzprüfung verknüpfter Compliance-Regeln

Wenn die Erfüllbarkeit einzelner Compliance-Regeln sichergestellt ist, sind auch ihre an innere Compliance-Scopes weitergegebenen Teilformeln erfüllbar. Die Unerfüllbarkeit von verknüpften Compliance-Regeln kann daher nur aus ihrem gegenseitigen Ausschluss oder dem gegenseitigen Ausschluss von Teilformeln aus verschiedenen Compliance-Regeln resultieren.

Nichtnotwendigkeit der Gültigkeitsprüfung

Da die Ungültigkeit einzelner Compliance-Regeln sichergestellt ist (siehe Abschnitt 4.3.2), ist eine Gültigkeitsprüfung verknüpfter Compliance-Regeln nach Satz 2.4 nicht notwendig. Denn die verknüpften Compliance-Regeln können nur dann gültig sein, wenn alle durch den logischen UND-Operator (symbolisch: „∧“) verknüpften Compliance-Regeln gültig sind. Die Abbildung 4.16 veranschaulicht, dass die verknüpfte Compliance-Regel aufgrund der Ungültigkeit der inneren Compliance-Regel ungültig ist.

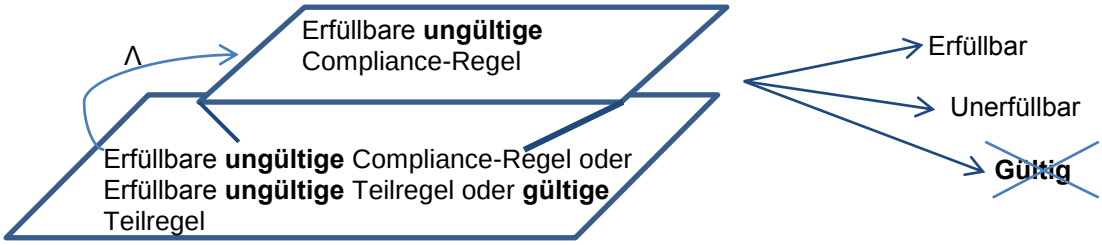


Abbildung 4.16: Ungültigkeit verknüpfter Compliance-Regeln

Vorhandene rekursive Routine des Model-Checking

Aufgrund des Top-Down-Ansatzes des inkrementellen Entwicklungsprozesses muss die Konsistenzprüfung bei dem äußersten Compliance-Scopes beginnen und rekursiv alle inneren Compliance-Scopes einschließen. Dazu kann auf der vorhandenen rekursiven Routine des Model-Checking (siehe Abschnitt 3.4.3) aller Compliance-Scopes aufgebaut werden. In der vorhandenen Routine werden die Compliance-Scopes unabhängig voneinander betrachtet.

Konsistenzprüfung vor Model-Checking und Abbruch der Compliance-Prüfung

Wenn das Model-Checking (siehe Abschnitt 2.3) eines Compliance-Scopes ergibt, dass das BPMN-Modell seine Compliance-Regel erfüllt, ist es immer noch möglich, dass diese Compliance-Regel mit den äußeren Compliance-Regeln inkonsistent oder *gültig* (siehe Abschnitt 3.2) ist. Bei einem negativen Ergebnis des Model-Checking kann nicht davon ausgegangen werden, dass das BPMN-Modell nach einer entsprechenden Korrektur die Compliance-Regel erfüllen wird, weil die Compliance-Regel *unerfüllbar* sein kann.

Im Falle einer Inkonsistenz mit den äußeren Compliance-Regeln oder der Ungültigkeit einer verknüpften Compliance-Regel wird das Model-Checking eines Compliance-Scopes überflüssig. Denn seine Compliance-Regel muss in diesem Fall korrigiert werden. Es liegt also nahe, die Konsistenzprüfung jeweils vor dem Model-Checking eines Compliance-Scopes durchzuführen und bei einer Inkonsistenz die Compliance-Prüfung aller weiteren inneren Compliance-Scopes abzubrechen.

Weitergabe von Compliance-Regeln

Zur Konsistenzprüfung einer Compliance-Regel mit ihren äußeren Compliance-Regeln müssen die Compliance-Regeln aller äußeren Compliance-Scopes zur Verfügung stehen. Dies kann dadurch erreicht werden, dass während des rekursiven Durchlaufs aller Compliance-Scopes die Compliance-Regel eines bereits geprüften Compliance-Scopes als Parameter an die Überprüfung der inneren Compliance-Scopes weitergegeben wird (vgl. Abschnitt 3.3). Eine Compliance-Regel des aktuell geprüften Compliance-Scopes wird dabei mit den äußeren Compliance-Regeln durch den logischen UND-Operator verknüpft und auf Erfüllbarkeit (siehe Abschnitt 2.4) geprüft. Die *verknüpfte Compliance-Regel* wird wiederum an weitere innere Compliance-Scopes solange weitergegeben, verknüpft und auf Erfüllbarkeit geprüft; bis es keine inneren Compliance-Scopes mehr gibt oder eine Inkonsistenz festgestellt wird. Wie in [SALS10] gezeigt, dürfen im Falle von indirekten Konflikten (siehe Abschnitt 3.3) nicht immer die vollständigen Compliance-Regeln, sondern nur die positiven erfüllten Teilregeln weitergegeben werden.

Die Abbildung 4.17 fasst die Abfolge der wichtigsten Schritte der Compliance-Prüfung zusammen. Da es im äußeren Compliance-Scope keine weitergegebenen äußeren Regeln gibt, wird keine Konsistenzprüfung durchgeführt und mit dem Model-Checking begonnen. Während des Model-Checking werden die weiterzugebenden Teilformeln bestimmt, das heißt Teilformeln, die nicht positiv und erfüllt sind. In diesem Beispiel wird ◇c weitergegeben weil a *U* b positiv und erfüllt ist. Es wird eine Inkonsistenz der inneren Compliance-Regeln mit der weitergegebenen Teilregel festgestellt und das Model-Checking sowie weitere Prüfung innerer Compliance-Scopes abgebrochen.

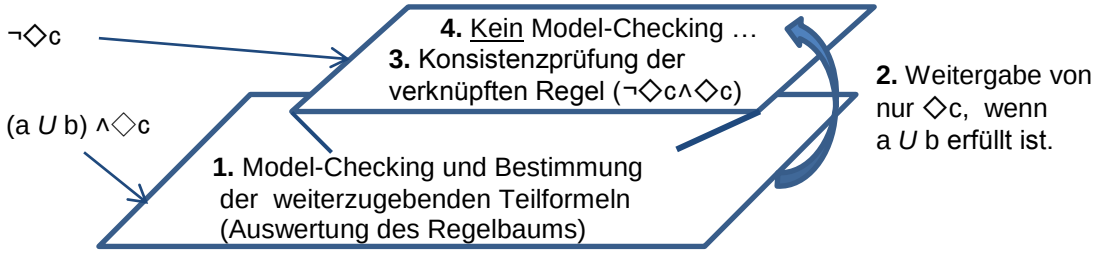


Abbildung 4.17: Schematischer Überblick zur Weitergabe von Compliance-Regeln

4.4 Wahl des SAT-Solvers

Ein SAT-Solver ist ein notwendiges Hilfsmittel zur Lösung der Aufgabenstellung im Rahmen dieser Arbeit. Es wurde früh begonnen nach einem passenden Tool zu suchen. Die Software sollte als Eingabe eine LTL-Formel akzeptieren und als Ergebnis *erfüllbar* oder *unerfüllbar* ausgeben.

So wurde z. B. das Tool ALASKA (siehe Abschnitt 2.4.2.2) getestet. Dieses ist zwar einfach zu bedienen und zu integrieren. Jedoch gab es nur eine 32-bit Version, die auf dem 64-bit Webserver, auf dem der Prototyp installiert ist, nicht lauffähig ist.

Als nächstes wurde das in [RV07] beschriebene Verfahren der Erfüllbarkeitsprüfung durch Zurückführung auf Model-Checking untersucht (siehe Abschnitt 2.4.2.1). Durch diesen Ansatz kann ein eigener SAT-Solver mit Hilfe des vorhandenen Model-Checkers gebaut werden. Dazu müsste ein universelles Modell generiert und gegen die negierte Formel mit SPIN verifiziert werden. Doch in Anbetracht der Zeit wurde darauf verzichtet einen eigenen SAT-Solver zu implementieren.

Die nächste untersuchte Möglichkeit war das Termersetzungssystem Maude (siehe Abschnitt 2.4.2.3) welches unter anderem auch zur LTL-Erfüllbarkeitsprüfung eingesetzt werden kann und in der 64-bit Version zur Verfügung steht. Da es erfolgreich getestet und eingebunden werden konnte, wurden keine weiteren SAT-Solver untersucht.

4.5 Erweiterung des Model-Checking

Im Rahmen der Arbeit mit dem Prototyp wurde festgestellt, dass beim Model-Checking nur der LTL-Operator *Finally* (◇) unterstützt wird. Mit dem Finally-Operator sind jedoch nur solche Regeln ausdrückbar, die bereits in [SALS10] im Rahmen der Aussagenlogik betrachtetet wurden (vgl. Abschnitte 3.2 und 4.2.3). Um Compliance-Regeln aus allen temporalen Gültigkeitsbereichen nach Abbildung 4.4 verifizieren und auf Konsistenz prüfen zu können, müssen die LTL-Operatoren (siehe Abschnitt 2.2.1) *Globally* (□) und *Until (U)* unterstützt werden. Die dafür vorgenommenen Anpassungen werden im Kapitel 5.4 beschrieben.

5 Implementierung

Im diesem Kapitel wird die prototypische Umsetzung des Konzepts aus dem vorhergehenden Kapitel beschrieben. Als Grundlage wird der in Abschnitt 2.1.4 vorgestellte Editor Oryx verwendet. Dabei wird im Wesentlichen auf den Erweiterungen des Editors in [Gro11] aufgebaut. Ausgehend von einem architektonischen Überblick über geänderte und hinzugefügte Komponenten werden die Details der Erweiterungen im Front- und im Backend beschrieben. Anschließend wird eine Erweiterung des Model-Checking beschrieben, die das Model-Checking mit LTL-Formeln mit den Operatoren *Globally* und *Until* ermöglicht.

5.1 Architektur

In Abbildung 5.1 wird die Architektur der Oryx-Erweiterung als vereinfachtes UML-Komponentendiagramm dargestellt. In der auf [Gro11] und [Köt10] basierenden Darstellung sind nur die zum Verständnis der Erweiterungen relevanten Komponenten berücksichtigt. Auf der linken Seite wird das Frontend bestehend aus den Hauptkomponenten Editor, der in dieser Arbeit nicht geändert wurde, und den Plugins dargestellt. Auf der rechten Seite werden die wichtigsten Komponenten des Backends und die eingebundenen Kommandozeilenprogramme dargestellt.

Im Frontend wurde durch das LTLSat-Plugin in der Oryx-Toolbar ein neuer Button hinzugefügt. Mit diesem lassen sich die im Editor modellierten LTL-Formeln auf Erfüllbarkeit und Gültigkeit prüfen. Der Compliance Wizard wurde um die automatische Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln erweitert.

Zur Ausführung der Erfüllbarkeits- und Gültigkeitsprüfung wurde das Kommandozeilenprogramm Maude (siehe Abschnitt 2.4.2.3) mit Hilfe der Komponente MaudeAdapter integriert. Da Maude die Eingabe in einer anderen Syntax erwartet, wurde der LTLTranslator entsprechend erweitert. Der LTLTranslator übersetzt ein im LTL-Editor erstelltes LTL-Modell in eine LTL-Formel in textueller Darstellung.

Wie im Abschnitt 4.2 beschrieben, erfolgt der Aufruf des SAT-Solvers in drei Fällen. Im ersten Fall wird eine LTL-Formel von dem LTLSat-Plugin über das LTL-Servlet an Maude übermittelt. Beim zweiten Fall wird von dem Compliance Wizard (siehe Abbildung 3.6) eine Compliance-Regel über das ComplianceServlet übermittelt. In diesem Fall liest der Compliance Wizard auch die im Regelbaum enthalten LTL-Formeln aus dem Repository und übermittelt sie ebenfalls an das ComplianceServlet. Im letzten Fall wird Maude zur Konsistenzprüfung, das heißt Erfüllbarkeitsprüfung, von verknüpften Compliance-Regeln genutzt.

Die Komponente LTLOperator ist eine der geänderten Klassen aus dem Packet *operators,* die den im Compliance Wizard erstellen Regelbaum auswerten. Diese Klassen wurden aus zwei Gründen angepasst. Zum einen werden sie verwendet um aus dem Regelbaum die LTL-Formel zur Erfüllbarkeits- und Gültigkeitsprüfung zu extrahieren (zweiter Fall im vorherigen Absatz). Zum anderen erfolgt dort die Erkennung der positiven und negativen Teilformeln während der Compliance-Prüfung.

In der Komponente PromelaExport wird die interne Petri-Netz-Darstellung des BPMN-Diagramms in die Sprache Promela (siehe Abschnitt 2.3.2.1) transformiert. Durch den Spin-Adapter erfolgt der Aufruf des Model-Checkers SPIN (siehe Abschnitt 2.3.2). Diese beiden Komponenten wurden für die Unterstützung der temporalen Operatoren *Until* und *Globally* (siehe Abschnitt 2.2.1) erweitert. Des Weiteren wurde der Spin-Adapter erweitert, um zu einer LTL-Formel einen Büchi-Automaten zu generieren und somit die positiven und negativen Teilformeln während der Compliance-Prüfung zu erkennen.

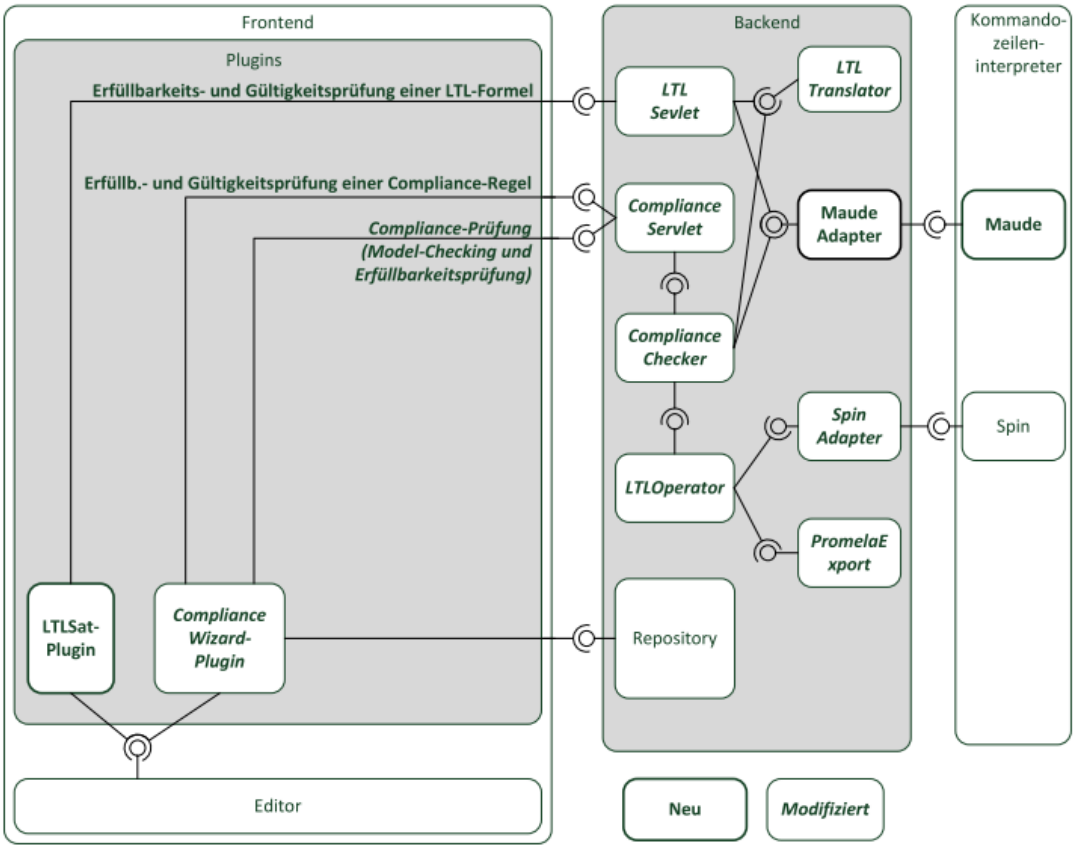


Abbildung 5.1: Architektur der Oryx-Erweiterung, nach [Gro11] und [Köt10]

5.2 Frontend

Im Vergleich zum Backend gab es im Frontend nur wenige Änderungen. Mittels JavaScript wurde ein neues Plugin implementiert und ein Bestehendes erweitert.

5.2.1 LTLSat-Plugin

Mit dem LTLSat-Plugin wurde die Prüfung modellierter LTL-Formeln laut Abschnitt 4.3.1 umgesetzt. Dazu wurde im LTL-Editor die Toolbar um den in Abbildung 5.2 hervorgehobenen SAT-Button zur manuellen Erfüllbarkeits- und Gültigkeitsprüfung erweitert. Das Plugin greift mittels einer AJAX-Anfrage auf ein Java Servlet im Backend zu. Das erstellte Diagramm wird im JSON-Format als ein Parameter an den Server geschickt. In einem weiteren Parameter wird der Typ der Anfrage angegeben, damit das aufgerufene Java Servlet die Anfrage von den anderen möglichen Anfragen unterscheiden kann. Nachdem das Plugin von dem Java Servlet ein Ergebnis erhalten hat, wird dem Benutzer das Ergebnis in einer Meldung ausgegeben, die entweder die Erfüllbarkeit,

Gültigkeit oder Unerfüllbarkeit der modellierten Formel angezeigt. Im abgebildeten Beispiel sind eine unerfüllbare Formel und die entsprechende Meldung zu sehen.

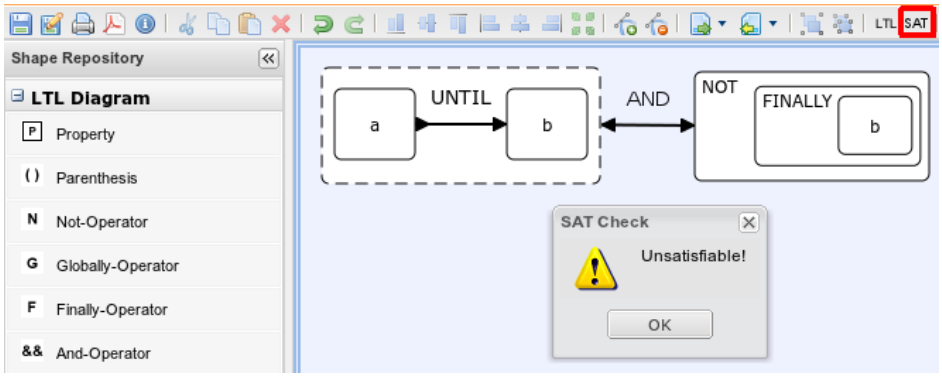


Abbildung 5.2: Erweiterung des LTL-Editors

5.2.2 Erweiterung des Compliance Wizard-Plugins

Im Compliance Wizard wurde die automatische Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln entsprechend Abschnitt 4.3.2 implementiert. Die neue Funktionalität wurde dem vorhandenen Ok-Button hinzugefügt, der die erstellte Compliance-Regel dem vorher ausgewählten Compliance-Scope zuweist (siehe Abbildung 5.3). Die Überprüfung erfolgt vor dieser Zuweisung. Auch hier wird mittels einer AJAX-Anfrage ein Java Servlet angesprochen. Dieses erhält als Parameter den Regelbaum in Form eines Operatorenbaums, an dessen Blättern die Modell-IDs der verwendeten LTL-Regeln eingetragen sind. Des Weiteren werden alle enthaltenen LTL-Modelle im Repository anhand ihrer Modell-ID nachgeschlagen und ebenfalls als Parameter im JSON-Format an den Server geschickt.

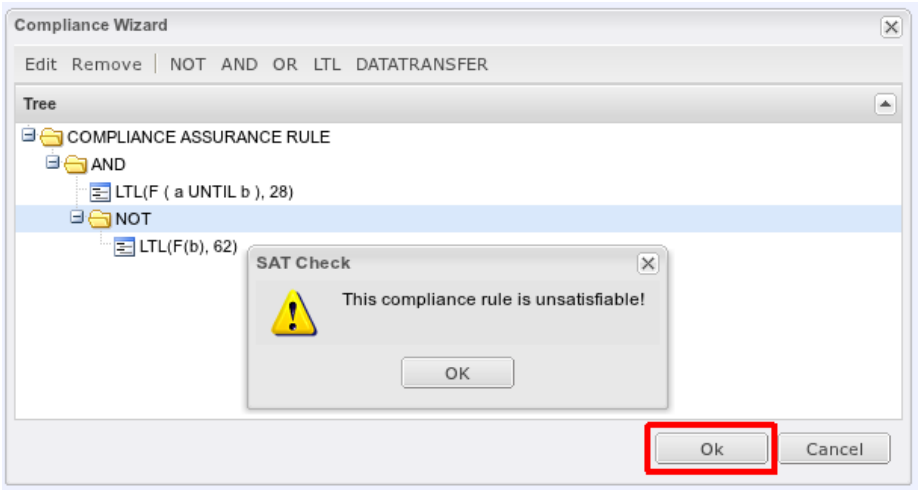


Abbildung 5.3: Erweiterung des Compliance Wizard

Wenn die erstellte Compliance-Regel erfüllbar ist, wird sie dem vorher ausgewählten Compliance-Scope zugewiesen und der Editor geschlossen. Im Falle der Unerfüllbarkeit oder Gültigkeit wird eine entsprechende Warnung ausgegeben. Nach dem Bestätigen der Warnung hat der Benutzer in dem noch geöffneten Editor die Möglichkeit die Compliance-Regel zu überarbeiten. Dadurch ist es nicht möglich eine unerfüllbare oder gültige Compliance-Regel einem Compliance-Scope zuzuweisen.

5.2.3 Ergebnisse einer Compliance-Prüfung

Die im Backend implementierte Konsistenzprüfung von Compliance-Regeln (siehe Abschnitt 4.3.4) erzeugt neue Ausgaben in den Ergebnissen der Compliance-Prüfung (siehe Abschnitt 3.4.3). Daher wird hier anhand eines einfachen Beispiels die Compliance-Prüfung aus der Benutzerperspektive erläutert. Dabei wird die unerfüllte positive Teilregel ◇d an den inneren Compliance-Scope weitergegeben. Der innere Compliance-Scope enthält eine noch nicht ausgefüllte variable Region, in der diese Teilregel im weiteren Verlauf der inkrementellen Entwicklung noch erfüllt werden kann.

5.2.3.1 Ergebnisübersicht

Zusätzlich zu den in Abbildung 3.8 dargestellten möglichen Ausgängen einer Compliance-Prüfung wurde das Ergebnis „***Unsatisfiable***“ definiert. Das Ergebnis *Unsatisfiable* gibt an, dass die Compliance-Regel eines Compliance-Scopes in Verknüpfung mit den weitergegebenen äußeren Compliance-Regeln oder Teilregeln unerfüllbar ist (vgl. Abschnitt 4.3.4). In der Ergebnisübersicht nach „Unsatisfiable“ wird die Anzahl der mit äußeren Compliance-Regeln inkonsistenten Compliance-Scopes angegeben. Diese Compliance-Scopes werden mit einem roten Hintergrund („Scope2“ in Abbildung 5.4) hervorgehoben.

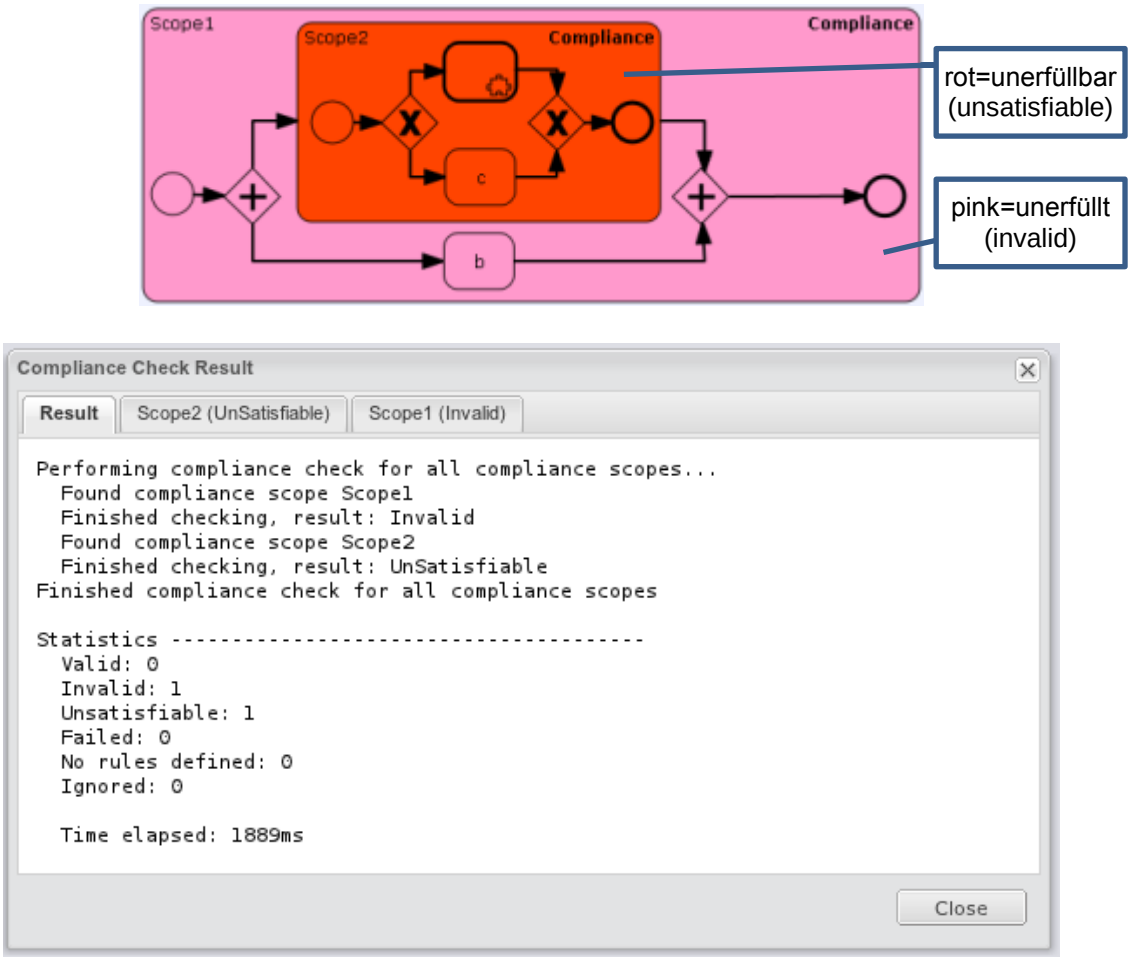


Abbildung 5.4: Das neue Ergebnis „Unsatisfiable“ bei der Compliance-Prüfung

5.2.3.2 Ergebnisse einzelner Compliance-Scopes

Die Abbildung 5.5 zeigt die Details zu dem Compliance-Scope mit dem Ergebnis „Invalid“, das heißt mit nicht erfüllter Compliance-Regel, dar. Im dem oberen, unsichtbaren Teil des abgebildeten Protokolls wird die aus dem Regelbaum ausgelesene Compliance-Regel in Textdarstellung (◇d ∧ (◇a ∨ ◇b)) ausgegeben. Da die Teilregel (◇a ∨ ◇b) durch die Aktivität b erfüllt (siehe Abbildung 5.4, oben) ist und als positiv (siehe Definition 4.1) erkannt wurde, wird sie nicht weitergegeben. Die positive Teilregel ◇d ist dagegen nicht erfüllt, und wird daher an den inneren Compliance-Scope weitergegeben.

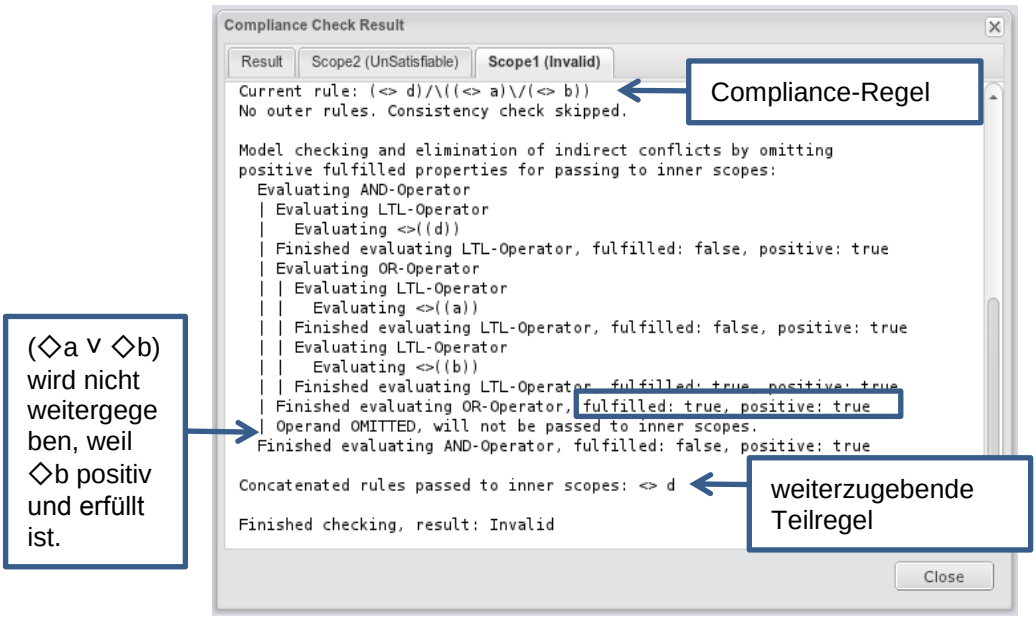


Abbildung 5.5: Weitergabe von Teilformeln an innere Compliance-Scopes (1)

In Abbildung 5.6 sind die Details des Ergebnisses des inneren Compliance-Scopes sichtbar. Es wird zunächst die innere Compliance-Regel ausgelesen und mit der Äußeren durch den UND-Operator verknüpft. Die verknüpfte Compliance-Regel wird anschließend auf Erfüllbarkeit geprüft (vgl. Abschnitt 4.3.4). In diesem Fall ist das Ergebnis „Unsatisfiable“.

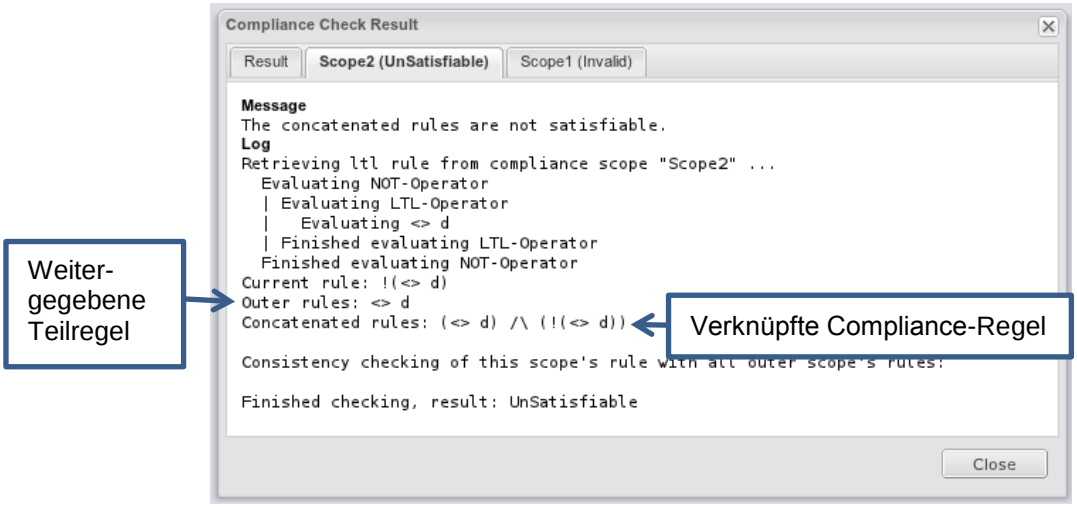


Abbildung 5.6: Weitergabe von Teilformeln an innere Compliance-Scopes (2)

5.3 Backend

Der Großteil der Programmierung fand im Backend statt, wo die Erfüllbarkeits- und Gültigkeitsprüfung sowie die Compliance-Prüfung stattfinden. In diesem Abschnitt werden die neuen und geänderten Komponenten im Backend beschrieben. Die größten Änderungen betreffen die Klassen ComplianceChecker, in der sich die Hauptprozedur der Compliance-Prüfung befindet, und die Klassen aus dem Packet *operators*, die den Regelbaum verarbeiten.

5.3.1 Erfüllbarkeitsprüfung

LTLTranslator

Der LTLTranslator übersetzt das LTL-Modell aus dem JSON-Format in die Textdarstellung [Gro11]. Zur Erfüllbarkeits- und Gültigkeitsprüfung wurde das Kommandozeilenprogramm Maude über die Schnittstelle MaudeAdapter integriert. Der SAT-Solver Maude erwartet als Eingabe eine LTL-Formel in einer anderen Syntax als SPIN. Beispielsweise muss die Konjunktion statt „&&“ mit „^“ und die Negation statt „!“ mit „~“ kodiert werden. Daher wurde der LTLTranslator so erweitert, dass die Maude-Syntax wählbar ist. Außerdem werden während der Übersetzung alle Literale in Klammern eingeschlossen, damit sie im MaudeAdapter durch einen regulären Ausdruck erkannt werden können. Dies ist notwendig, weil in dem Quelltext, welchen Maude ausführt (siehe Abschnitt 2.4.2.3), die atomaren Formeln angegeben werden müssen.

MaudeAdapter

Der MaudeAdapter liest die Literale aus der empfangenen LTL-Formel mit Hilfe eines regulären Ausdrucks aus. Die Literale und die LTL-Formel werden in eine Vorlage zum Aufruf des Maude-SAT-Solvers (siehe Abschnitt 2.4.2.3) eingefügt (siehe Listing 5.1). Diese Vorlage wird in einer temporären Datei gespeichert. Anschließend wird Maude auf der Kommandozeile mit dem Pfad zu dieser Datei als Parameter aufgerufen. Die Ausgabe der Kommandozeile wird ausgewertet und als Ergebnis ein boolescher Wert ausgegeben.

```
String maudeSourceFileContent =
                                "load model-checker.maude \n" +
                                "fmod SAT-SOLVER-TEST is \n" +
                                        "extending SAT-SOLVER .\n" +
                                        "extending LTL .\n" +
                                        "ops " + atomicPredicates(formula) + ": -> Formula .\n"+
                                "endfm \n" +
                                "red satSolve("+ formula +") .\n" +
                                "quit";
```

Listing 5.1: Vorlage zur LTL-Erfüllbarkeitsprüfung mit Maude

5.3.2 Das LTLServlet

Das LTLServlet wurde so erweitert, dass es Anfragen von dem LTLSat-Plugin (siehe Abschnitt 5.2.1) verarbeiten kann. In Abbildung 5.7 ist die grundlegende Interaktion der relevanten Komponenten für die Erfüllbarkeits- und Gültigkeitsprüfung einer im LTL-Editor modellierten LTL-Formel dargestellt.

Das LTLServlet erhält von dem LTLSat-Plugin mittels einer AJAX-Anfrage das LTL-Modell im JSON-Format sowie den Anfragetyp. Mittels des Anfragetyps erkennt das Servlet, dass die Formel auf Erfüllbarkeit und Gültigkeit geprüft werden muss. Dazu wird die Java-Klasse LTLTranslator mit

der Maude-Syntax instanziiert und seine Methode translate() aufgerufen. Anschließend wird der MaudeAdapter mir der Methode isSatisfiable() aufgerufen, die das Ergebnis der Erfüllbarkeitsprüfung als einen booleschen Wert zurückliefert (siehe Abschnitt 5.3.1).

Wenn die LTL-Formel erfüllbar ist, negiert das LTLServlet die Formel und schickt sie erneut an den MaudeAdapter zur Erfüllbarkeitsprüfung, anderenfalls wird „Unerfüllbar“ ausgegeben. Wenn auch die Negation erfüllbar ist, wird als Ergebnis „Erfüllbar und ungültig“, anderenfalls „Gültig“ ausgegeben.

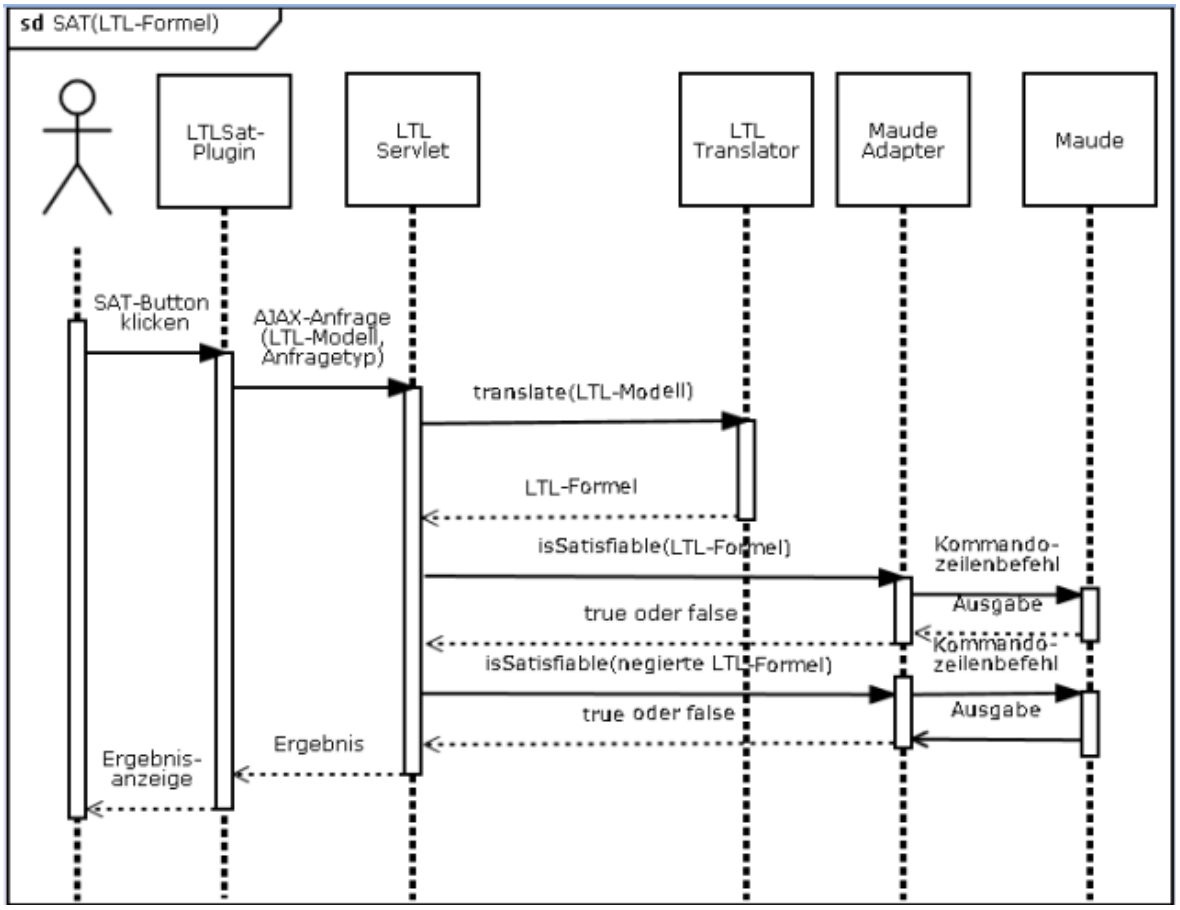


Abbildung 5.7: Sequenzdiagramm zur Erfüllbarkeitsprüfung einer LTL-Formel

5.3.3 Automatische Prüfung von Compliance-Regeln

Das ComplianceServlet wurde so erweitert, dass es Anfragen von dem Compliance Wizard-Plugin (siehe Abschnitt 5.2.2) zur Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln verarbeiten kann. Im Gegensatz zum LTLServlet erhält das ComplianceServlet kein LTL-Modell, welches vom LTLTranslator unmittelbar in eine Textdarstellung übersetzbar ist. Stattdessen erhält es den Operatorenbaum im JSON-Format, an dessen Blättern die IDs der verwendeten LTL-Modelle (siehe Abbildung 3.4) eingetragen sind. Zusätzlich erhält das ComplianceServlet die verwendeten LTL-Modelle im JSON-Format als einzelne Parameter. Zur Verdeutlichung zeigt das Listing 5.2 den JSON-Code des Operatorenbaums aus Abbildung 3.6. Die entsprechende Compliance-Regel im Textdarstellung lautet: ◇d ∧ (◇a ∨ ◇b).

```
{       "type": "andOperator",
        "operands":
                [{"type": "ltlOperator",
                "modelId": "29",
                "modelName": "Finally d"},

                {"type": "orOperator",
                "operands":
                        [{"type": "ltlOperator",
                        "modelId": "28",
                        "modelName": "F(a)"},

                        {"type": "ltlOperator",
                        "modelId": "45",
                        "modelName": "Finally b"}] }]  }
```

Listing 5.2: Beispiel für eine Compliance-Regel im JSON-Format

Um eine Textdarstellung der Compliance-Regel zu erhalten, wurden die in Abbildung 5.8 dargestellten Klassen aus dem Packet *operators* erweitert, weil sie den Operatorenbaum auswerten können (siehe Abschnitt 3.4.3). Die enthaltenen LTL-Regeln werden anhand logischer Operatoren in den Knoten des Baums zu einer Gesamtformel rekursiv zusammengefügt.

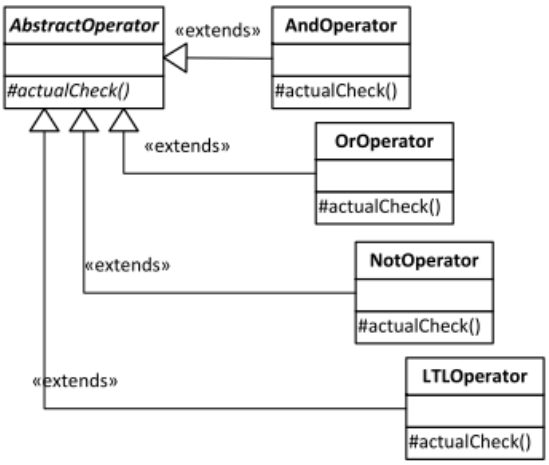


Abbildung 5.8: UML-Klassendiagramm der Operatoren zur Verarbeitung der Regelbaums

Funktionsweise der Operatoren-Klassen zum Auslesen der Gesamtformel

Die Methode ***actualCheck()*** wird mit dem boolesche Parameter ***modelcheck=false*** aufgerufen. Anhand dieses Parameters erfolgt die Unterscheidung zwischen dem Auslesen der Gesamtformel und dem Model-Checking inklusive der Regelweitergabe (siehe Abschnitt 5.3.4). In diesem Fall wird jeder Operator des Regelbaums auf seine Operanden angewendet und als Ergebnis die Gesamtformel in Textdarstellung ausgegeben.

Zunächst wird der Typ des Operators an der Wurzel des Operatorenbaums bestimmt und die entsprechende Operator-Klasse (And-, Or-, Not- oder LTLOperator) instanziiert. Die Klassen And- und OrOperator enthalten in der Methode actualCheck() eine Schleife, welche die Operanden durchläuft. Falls ein Operand ein LTLOperator ist, das heißt ein LTL-Modell enthält, wird das LTL-Modell mit Hilfe des LTLTranslators in die Textdarstellung übersetzt. Anderenfalls wird ein neuer Operator instanziiert. Beispielsweise erstellt der AndOperator im Beispiel aus Listing 5.2 für seinen zweiten Operanden eine Instanz des OrOperators und ruft seine Methode actualCheck() auf. Sobald alle Operanden in der Textdarstellung vorliegen, werden sie von dem OrOperator mit „v“ und von dem AndOperator mit „^“ verknüpft. Der NotOperator enthält keine Schleife, sondern stellt einem in Textdarstellung vorliegenden Operanden das Negationszeichen „~“ voran.

Der Gesamtablauf

Der Gesamtablauf der automatischen Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln ist als UML-Sequenzdiagramm in Abbildung 5.9 dargestellt. Der wesentliche Unterschied zum Sequenzdiagramm zur Erfüllbarkeits- und Gültigkeitsprüfung einer LTL-Formel (siehe Abbildung 5.7) liegt in der Verwendung der Operatoren-Klassen zur Auswertung des Regelbaums. Daher wird der LTLTranslator mehrmals aufgerufen. Zur Vereinfachung der Darstellung repräsentiert der AbstactOperator alle vier Operatoren-Klassen. Außerdem wurde die Hauptprozedur nicht im ComplianceServlet sondern in der Klasse ComplianceChecker implementiert, weil dort ähnliche Funktionalitäten gekapselt sind. Die Gültigkeitsprüfung einer Compliance-Regel erfolgt analog zur Gültigkeitsprüfung einer LTL-Formel (siehe Abschnitt 5.3.2).

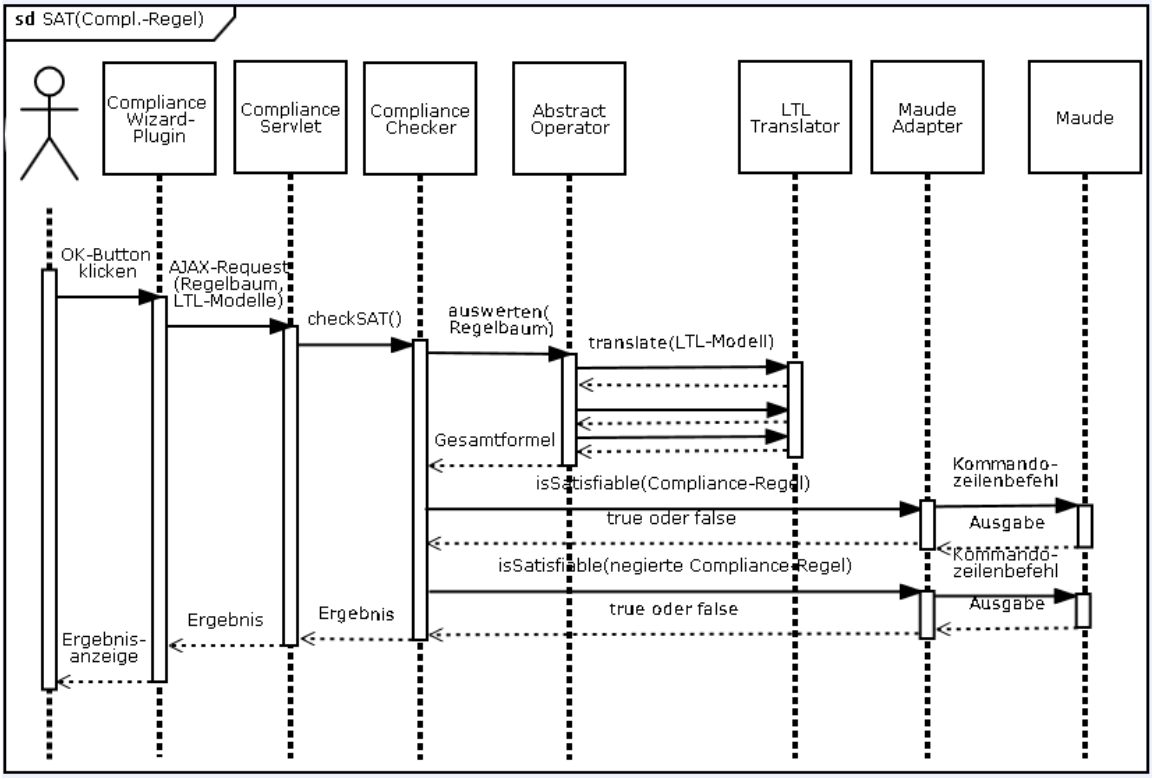


Abbildung 5.9: Sequenzdiagramm zur Erfüllbarkeitsprüfung einer Compliance-Regel

5.3.4 Konsistenzprüfung von Compliance-Regeln

Wie bei der Erfüllbarkeits- und Gültigkeitsprüfung von Compliance-Regeln empfängt das ComplianceServlet eine AJAX-Anfrage vom Compliance Wizard-Plugin und leitet diese an den ComplianceChecker weiter (siehe Abbildung 5.9). Zusätzlich zum Regelbaum und den

verwendeten LTL-Modellen enthält die AJAX-Anfrage das BPMN-Modell sowie die ausgewählten Compliance-Scopes. Zur Konsistenzprüfung von Compliance-Regeln wurde die rekursive Hauptprozedur des Model-Checking in der Klasse ComplianceChecker überarbeitet. Des Weiteren wurden die Operatoren-Klassen (siehe Abbildung 5.8) so erweitert, dass während des Model-Checking die positiven erfüllten Teilregeln (siehe Abschnitt 4.2) erkannt werden. Das Model-Checking der inneren Compliance-Scopes erfolgt erst nachdem die Konsistenz einer Compliance-Regel mit ihren äußeren Compliance-Regeln gewährleistet ist (vgl. Abschnitt 4.3.4). Der neue Ablauf der Compliance-Prüfung wird anhand des Aktivitätsdiagramms in Abbildung 5.10 erläutert. Das Diagramm enthält nur die zum Verständnis wichtigsten Details der Implementierung.

5.3.4.1 Hauptprozedur der Compliance-Prüfung

Wie in der vorherigen Version werden alle grafischen Elemente des BPMN-Diagramms rekursiv durchlaufen. Wenn es sich bei einem Element um keinen Compliance-Scope handelt, kann es auch ein Teilprozess sein, der Compliance-Scopes enthält. Daher werden seine Kind-Elemente durch eine for-Schleife und einen rekursiven Aufruf der gleichen Funktion abgearbeitet. Wenn nur selektierte Compliance-Scopes geprüft werden sollen und der aktuelle Scope nicht in der Auswahl ist, endet die Prüfung dieses Compliance-Scopes mit dem Ergebnis „Ignored“ (vgl. Abbildung 5.4). Dabei werden seine inneren Compliance-Scopes weiterhin geprüft.

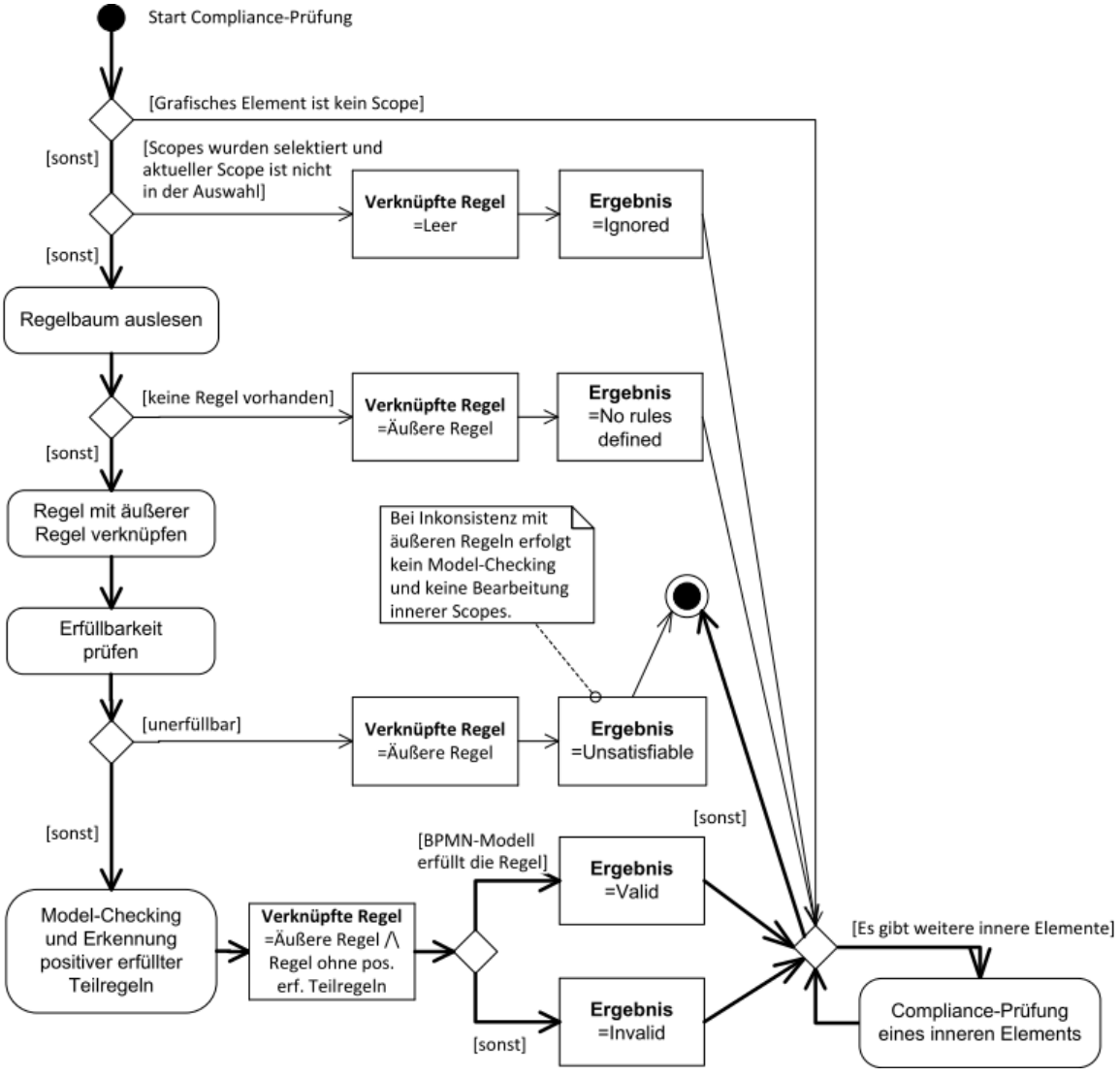


Abbildung 5.10: Aktivitätsdiagramm der Compliance-Prüfung

Vererbung verknüpfter Compliance-Regeln

Bei jedem rekursiven Aufruf der Prozedur in Abbildung 5.10 wird die verknüpfte Compliance-Regel (siehe Abschnitt 4.1.1) als Parameter weitergegeben. Nach jeder abgeschlossenen Prüfung eines Compliance-Scopes wird daher eine neue verknüpfte Compliance-Regel zur Weitergabe bestimmt. In der vorherigen Version wurden im Falle einer Selektion genau die selektierten Compliance-Scopes unabhängig voneinander geprüft. In dieser Version werden automatisch neben dem selektierten Compliance-Scope auch seine inneren Compliance-Scopes geprüft. Die äußeren Compliance-Scopes werden dagegen nicht geprüft. Daher ist im Falle eines ignorierten Compliance-Scopes die verknüpfte Compliance-Regel leer.

Konsistenzprüfung

Da beim Model-Checking nur die LTL-Formeln an den Blättern des Regelbaums benötigt werden, wurde in der vorherigen Version der Regelbaum nicht in eine Textdarstellung übersetzt. Im Gegensatz dazu wird die Textdarstellung nun zur Konsistenzprüfung benötig und analog zu der Beschreibung im Abschnitt 5.3.3 bestimmt. Wenn keine Compliance-Regel gefunden wird und der aktuelle Compliance-Scope eine äußere Compliance-Regel geerbt hat, wird die äußere Compliance-Regel an die inneren Compliance-Scopes vererbt und das Ergebnis auf „No rules defined“ gesetzt.

Die aus dem Regelbaum ausgelesene Compliance-Regel in Textdarstellung wird mit den geerbten äußeren Compliance-Regeln oder Teilregeln durch „^“ verknüpft und auf Erfüllbarkeit geprüft. Dies erfolgt analog zur automatischen Erfüllbarkeitsprüfung von Compliance-Regeln (siehe Abschnitt 5.3.3), jedoch mit dem Unterschied, dass entsprechend Abschnitt 4.3.4 keine Gültigkeitsprüfung erfolgt. Im Falle der Unerfüllbarkeit, wird die Prüfung des Compliance-Scopes abgebrochen und das Ergebnis „Unsatisfiable“ ausgegeben.

Verknüpfung und Weitergabe von Compliance-Regeln

Zur Entscheidung, welche Teilformeln an innere Compliance-Scopes weitergegeben werden, muss der Regelbaum durch die Operatoren-Klassen (siehe Abbildung 5.8) erneut ausgewertet. Im Gegensatz zum Auslesen der Gesamtformel des Regelbaums (siehe Abschnitt 5.3.3) wird der Parameter ***modelcheck*** auf ***true*** gesetzt. Somit erfolgt das Model-Checking der LTL-Formeln an den Blättern des Regelbaums und das boolesche Gesamtergebnis des Model-Checking wird durch Weitergabe der Teilergebnisse zur Wurzel hin berechnet [Gro11] (siehe Abschnitt 3.4.3). Dadurch kann für jede Teilregel (siehe Abschnitt 4.1.1) festgestellt werden, ob sie **erfüllt** oder **unerfüllt** ist.

Wie in den Abschnitten 3.3 und 4.2.5 beschrieben, dürfen erfüllte positive Teilregeln nicht an innere Compliance-Scopes weitergegeben werden. Zur Unterscheidung **positiver** und **negativer** Teilregeln (siehe Definition 4.1 und Definition 4.2) werden die LTL-Formeln an den Blättern des Regelbaums in Büchi-Automaten umgewandelt und auf Vorkommen des „accept_all“-Zustands untersucht (siehe Abschnitt 2.3.2.2.3). Wie beim Model-Checking wird das Gesamtergebnis für eine Teilregel durch Weitergabe der Teilergebnisse zur Wurzel hin berechnet.

5.3.4.2 Funktionsweise der Operatoren-Klassen

Im Folgenden wird anhand des Regelbaums in Abbildung 5.11 die Funktionsweise der Operatoren-Klassen zur Verknüpfung und Weitergabe von Compliance-Regeln erläutert. Die dem Regelbaum entsprechende Compliance-Regel ◇b ∧ (◇a ∨ ¬◇□b) ist beispielsweise in dem Compliance-Scope „Scope1“ in Abbildung 5.4 erfüllt.

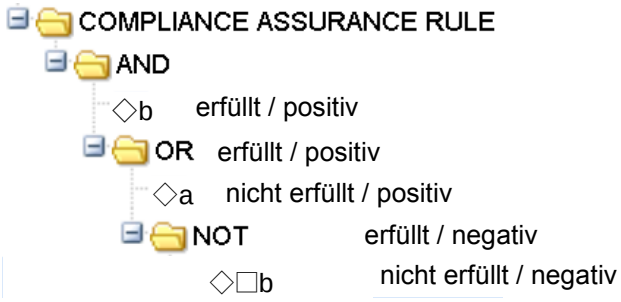


Abbildung 5.11: Regelbaum für ◇d ∧ (◇a ∨ ◇□b)

Im Anhang A.4 sind die Pseudoodes zu den vier Operatoren-Klassen zu finden. Diese enthalten nur den Fall der Compliance-Prüfung und nicht den in Abschnitt 5.3.3 beschrieben Fall mit dem Parameter *modelcheck=false*. Die folgende Beschreibung der Operatoren kann anhand der Pseudocodes verfolgt werden. In alle Operatoren wird ein Objekt *Ergebnis* vom Typ *ComplianceOperatorResult* [Gro11] instanziiert, welches bearbeitet und ausgegeben wird.

5.3.4.2.1 LTLOperator

Durch das Model-Checking im LTLOperator (siehe Listing A. 8) wird die Nichterfüllung von ◇□b festgestellt und im Objekts *Ergebnis* gespeichert. Anschließend wird der Büchi-Automat von der LTL-Formel bestimmt und anhand des Fehlens des „accept_all“-Zustand als *negativ* (nach Definition 4.2) erkannt. Da der LTLOperator in diesem Fall nicht auf dem obersten Level im Operatorenbaum ist, wird die Formel im Ergebnis-Objekt gespeichert und das Ergebnis ausgegeben.

5.3.4.2.2 NotOperator

Der NotOperator hat zuvor den LTLOperator aufgerufen (TeilOperator.actualCheck() in Listing A. 7) und negiert die im Objekt *TeilErgebnis* gespeicherte Formel. Die Negation wird in dem *Ergebnis*-objekt des NotOperators gespeichert. Auch das Model-Checking-Ergebnis wird negiert und gespeichert. Das heißt, die Teilformel ¬◇□b ist erfüllt.

Da es Co-Safety LTL-Formeln (siehe Abschnitt 2.2.1.3.3) gibt, darf eine als negativ erkannte Teilformel durch den logischen NOT-Operator im Regelbaum nicht als eine positive Teilformel interpretiert werden. Vielmehr muss ein neuer Büchi-Automat erstellt und in ihm nach dem „accept_all“-Zustand gesucht werden.

Daher wird der Büchi-Automat für ¬◇□b erzeugt und geprüft. Es handelt sich um eine Co-Safety Formel, weil ihre Negation ebenfalls *negativ* ist. Da der LTLOperator in diesem Fall nicht auf dem obersten Level im Operatorenbaum ist, wird nur das Ergebnis ausgegeben.

5.3.4.2.3 OrOperator

Der OrOperator verarbeitet in einer Schleife alle Operanden, den LTLOperator und den NotOperator. Von dem LTLOperator erhält er *TeilErgebnis.Erfuellt=false.* Daher bleibt *Ergebnis.Erfuellt* des OrOperators auf *false*. Da das *TeilErgebnis* zu ◇a positiv ist, wird

Ergebnis.Positiv des OrOperators auf *true* gesetzt. Das heißt, analog zum Model-Checking [Gro11], wird und bleibt das Ergebnis des OrOperators positiv, sobald ein positiver Operand entdeckt wird. Dies entspricht den Überlegungen im Abschnitt 4.2.6.5. Da der OrOperator in diesem Fall nicht auf dem obersten Level im Operatorenbaum ist, wird nur das *Ergebnis* ausgegeben.

Wenn ◇a erfüllt und der OrOperator auf der obersten Regelbaumebene wäre (z. B. Compliance-Regel ◇a v □b), dann würde die for-Schleife verlassen (exit for) und im *Ergebnis* eine leere Formel ausgegeben werden.

Im nächsten Durchlauf der for-Schleife wird das *TeilErgebnis* des NotOperators ausgewertet. Da der NotOperator erfüllt ist, wird *Ergebnis.Erfuellt* auf *true* gesetzt. *Ergebnis.Positiv=true* kann sich aufgrund ◇a nicht mehr ändern. Das heißt, der OrOperator ist erfüllt und positiv (vgl. Abbildung 5.11).

5.3.4.2.4 AndOperator

Das Ergebnis des AndOperators wird mit *erfüllt* und *positiv* auf *true* initialisiert. Falls ein *negativer* Operand gefunden wird, wird *Ergebnis*.*Positiv* des AndOperators auf *false* gesetzt. Bei einem nicht erfüllten Operanden, wird *Ergebnis*.*Erfuellt* auf false gesetzt.

Da der AndOperator in diesem Fall auf der obersten Regelbaumebene ist, wird entschieden, ob die Teilregeln weitergegeben werden. Da ◇a erfüllt und positiv ist, wird diese Teilformel nicht in *Ergebnis.Formel* gespeichert. Dasgleiche gilt für (◇a v ◇□b). Damit werden beide Formeln nicht an den inneren Compliance-Scopes weitergegeben.

5.4 Erweiterung des Model-Checking

In diesem Abschnitt werden einige Erweiterungen des Model-Checking zur Unterstützung der LTL-Operatoren *Globally* (□) und *Until (U)* beschrieben. Es wurden Anpassungen zur Generierung des ausführbaren Kreuzproduktes (vgl. Abbildung 2.12) des Modells und der negierten LTL-Regel (siehe Abbildung 2.10) vorgenommen. Das heißt, es wurden die generierten Never Claims und die Transformation der Petrinetze in Promela-Modelle angepasst.

5.4.1 Never Claims

Das Promela-Modell und der Never Claim (siehe Abschnitt 2.3.2.2) werden schrittweise nacheinander ausgeführt. Dabei wird der Definitionsteil (sogenannte Macros) als ein zusammenhängender Schritt behandelt. In den Macros werden die im Never Claim verwendeten Variablen auf die Plätze des Petrinetzes abgebildet sowie die Transitionen definiert [Gro11]. Nach den Macros wird ein erster Schritt in dem Never Claim ausgeführt. Das Listing 5.3 zeigt den ursprünglichen Code des Kreuzproduktes eines Modells, welches nur aus einem Task *Test* besteht (siehe Abbildung 5.12), und seiner negierten Spezifikation □Test.

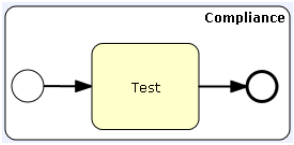


Abbildung 5.12: Compliance-Scope mit einem Task

```
byte p[5];

#define Test p[3]
#define rd_6_transition0 p[1] && !p[2]
#define fire_6_transition0 p[1] = 0; p[2] = 1;
#define rd_7_transition1 p[2] && !p[3]
#define fire_7_transition1 p[2] = 0; p[3] = 1;
#define rd_8_transition2 p[3] && !p[4]
#define fire_8_transition2 p[3] = 0; p[4] = 1;                    /* Ende erster Schritt (Macros) */
active proctype test()
{
  d_step { p[0] = 0; p[1] = 1; p[2] = 0; p[3] = 0; p[4] = 0; }    /* Ende zweiter Schritt */
  do
    :: rd_6_transition0 -> d_step{printf("PROCESSED_6_transition0"); fire_6_transition0}
    :: rd_7_transition1 -> d_step{printf("PROCESSED_7_transition1"); fire_7_transition1}
    :: rd_8_transition2 -> d_step{printf("PROCESSED_8_transition2"); fire_8_transition2}
    :: p[4] -> goto accept
  od;
  accept: printf("Accepted");
}

never  {    /*  !([](Test))  */
T0_init:
        if
        :: (! ((Test))) -> goto accept_all                /* akzeptierender Zustand erreicht */
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

Listing 5.3: Never Claim mit vorzeitigem Abbruch

Die boolesche Variable *Test* entspricht dem Task *Test*. Diese wird auf den Wert des Arrays p an der Stelle 3 abgebildet. In diesem Fall akzeptiert („goto accept_all“) der Never Claim den ersten Zustand, der sich nach dem Ausführen der Macros ergibt, weil nach dem ersten Programmschritt p[3]=0 gilt. Das Ergebnis des Model-Checking ist somit negativ, obwohl das Modell seiner Spezifikation entspricht. Der erste Schritt im Promela-Modell muss daher vom Never Claim übersprungen werden. Außerdem muss auch der zweite Schritt übersprungen werden, in dem die Plätze des Petrinetzes vorbelegt werden (siehe Kommentar in Listing 5.3).

Dies kann mit dem Befehlt *true* erreicht werden. Im Büchi-Automaten entspricht dies einem Zustandsübergang, der unabhängig von der Eingabe stattfindet. Das Listing 5.4 zeigt den entsprechenden Never Claim und die Abbildung 5.13 die entsprechende visuelle Darstellung (vgl. Abbildung 2.13).

```
never  {    /*  !([](Test))  */
true;                                          /* nach den Makros */
true;                                          /* nach der Initialisierung der Plätze */
T0_init:
        if                                     /* nach der ersten Transition im Petrinetz */
        :: (! ((Test))) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
```

Listing 5.4: Erweiterter Never Claim für □Test

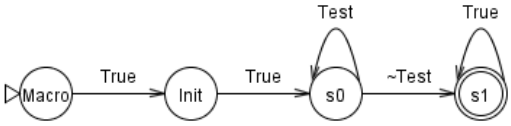


Abbildung 5.13: Erweiterter Never Claim für □Test

5.4.2 Promela-Modell

Im Folgenden werden die Anpassungen der Abbildung des Mapping des internen Petrinetzes auf Promela.

5.4.2.1 Vorbedingungen der Transitionen

In der do-Schleife [wwwh] im Listing 5.3 werden Transitionen nichtdeterministisch auswählt. Durch das Konstrukt d_step{…} werden alle Programmschritte innerhalb der Klammern {…} wie ein atomarer Schritt ausgeführt [wwwh]. Das heißt, der nächste Schritt im Never Claim erfolgt erst nach der vollständigen Ausführung von d_step{…}.

Durch die oben beschriebene Anpassung des Never Claims ergibt das Model-Checking von □Testen immer noch einen Fehler. Der Grund dafür ist, dass der erste Verifikationsschritt im Never Claim bereits nach dem Auswählen und Prüfen der Vorbedingungen einer Transition erfolgt. Daher wurden die Prüfungen der Vorbedingungen mit dem Schalten der Transitionen zu atomaren Schritten vereint (siehe Listing 5.5).

```
do
  :: d_step{rd_6_transition0 -> printf("PROCESSED_6_transition0"); fire_6_transition0}
  :: d_step{rd_7_transition1 -> printf("PROCESSED_7_transition1"); fire_7_transition1}
  :: d_step{rd_8_transition2 -> printf("PROCESSED_8_transition2"); fire_8_transition2}
  :: d_step{p[4] -> goto accept}
od;
```

Listing 5.5: Vorbedingungen der Transitionen

5.4.2.2 Die letzte Transition

Bei der letzten Transition durch *fire_8_transition2* in Listing 5.3 wird der boolesche Wert der Variable *Testen* durch p[3] = 0; auf *false* gesetzt. Die Verifikation dieses letzten Zustands gegen die Spezifikation □*Testen* ergibt wieder einen Fehler. Das heißt der Never Claim terminiert. Aus diesem Grund wird die letzte Transition des Petrinetzes im Promela-Modell nicht mehr berücksichtigt. Dadurch wird die do-Schleife geblockt, weil es keine ausführbaren Anweisungen gibt. Dies wird jedoch in Promela nicht als ein Fehler interpretiert [wwwh]. Alternativ kann die letzte Transition so gestaltet werden, dass die im vorletzten Zustand gültigen Variablen nicht verändert werden.

5.4.2.3 Definition der Tasks

Ein Task, der in einem BPMN-Diagramm mehrmals vorkommt, wird auf verschiedene Plätze des Petrinetzes abgebildet (siehe Listing 5.3). Dabei wird jedoch mit jeder neuen Definition des Tasks die vorhergehende Definition überschrieben, so dass es im Promela-Modell nur einen Platz gibt, der diesem Task entspricht.

```
byte p[8];
…
#define Testen p[4]
#define Testen p[6]
…
```

Listing 5.6: Beispiel für eine überschriebene Task-Definition

Während dies das Model-Checking eines Modells ohne Verzweigungen mit der LTL-Regel ◇ Testen nicht beeinträchtigt, können gleich benannte Tasks auf verzweigten Pfaden nicht gefunden werden. Auch in Modellen ohne Verzweigungen können LTL-Regeln wie □Testen, ◇□Testen, □ ◇Testen oder (Anfrage *U* Antwort), in denen ein Task mehrmals vorkommen kann, das Model-Checking nicht bestehen. Denn nach der Aktivierung eines Platzes des Petrinetzes, dem *kein* Task zugeordnet ist, akzeptiert der Never Claim den neuen Zustand. Die Erweiterung der Abbildung des Petrinetzes auf Promela wird anhand des Diagramms in Abbildung 5.14 und des entsprechenden Promela-Modells in Listing 5.7 erläutert.

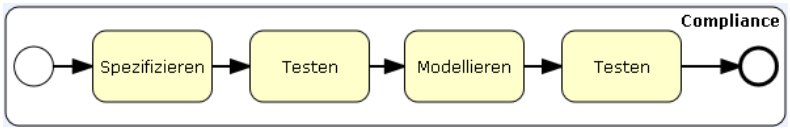


Abbildung 5.14: BPMN-Modell zum erweiterten Promela-Modell

Zustands-Array mit Task-IDs

Für n verschiedene Tasknamen wird das n-stellige *Zustands-Array* state[n] definiert. Das Zustandsarray gibt an, welche Tasks aktiv sind, wobei seine Indizes die *Task-IDs* sind. Jeder Taskname wird dabei mit einer Task-ID aus diesem Array assoziiert, beispielsweise:

```
#define Testen state[1]
```

Dabei bedeutet beispielsweise state[1]=2, dass es zwei aktive Tasks mit der Task-ID=0 gibt. Der Name des Tasks mit der Task-ID=0 ist *Testen,* wie oben definiert.

Task-IDs auf Plätze verteilen

Durch ein weiteres Array wird jedem Platz, welches einen ausgeführten Task repräsentieren soll, eine Task-ID und damit indirekt ein Taskname zugeordnet. Beispielsweise werden der zwei Mal vorkommende Task *Testen* (Task-ID=1) den Plätzen vier und sechs mit taskID_p[4]=1 und taskID_p[6]= 1 zugeordnet.

Schalten des Petrinetzes

Für m Plätze wird weiterhin das Array p[m] definiert. Das Schalten des Petrinetzes und die Vorbedingungen sind unverändert. Beim Schalten wird jedoch zusätzlich das Zustands-Array bearbeitet. Durch state[taskID_p[4]]++; wird der Token des Petrinetzes auf Platz p[4] verschoben. Da taskID_p[4] = 1 gilt, wird state[1] erhöht. Das heißt, der Task *Testen* wird aktiviert. Das Zustands-Array an der Stelle, die dem vorherigen Task entspricht, wird dabei erniedrigt.

```
byte p[8];
byte taskID_p[8];
byte state[3];

#define Spezifizieren    state[0]
#define Testen           state[1]
#define Modellieren      state[2]

#define rd_197_transition0 p[1] && !p[2]
#define fire_197_transition0 p[1] = 0; p[2] = 1;

#define rd_198_transition1 p[2] && !p[3]
#define fire_198_transition1 p[2] = 0; p[3] = 1; state[taskID_p[3]]++;

#define rd_199_transition2 p[3] && !p[4]          /* Task Testen wird aktiviert */
#define fire_199_transition2 p[3] = 0; state[taskID_p[3]]--; p[4] = 1; state[taskID_p[4]]++;

#define rd_200_transition3 p[4] && !p[5]
#define fire_200_transition3 p[4] = 0; state[taskID_p[4]]--; p[5] = 1; state[taskID_p[5]]++;

#define rd_201_transition4 p[5] && !p[6]
#define fire_201_transition4 p[5] = 0; state[taskID_p[5]]--; p[6] = 1; state[taskID_p[6]]++;

#define rd_202_transition5 p[6] && !p[7]
#define fire_202_transition5 p[6] = 0; state[taskID_p[6]]--; p[7] = 1;

#define start p[2]
active proctype test() {
  d_step   {

          taskID_p[3] = 0;
          taskID_p[4] = 1;        /* TaskID des Platzes 4 ist 1 */
          taskID_p[5] = 2;
          taskID_p[6] = 1;        /* Dem Platz 6 ist ein Task mit der Task-ID=1 zugeordnet. */

p[0] = 0; p[1] = 0; p[2] = 1; p[3] = 0; p[4] = 0; p[5] = 0; p[6] = 0; p[7] = 0;   }
  do
    :: d_step{ rd_197_transition0 -> {printf("PROCESSED_197_transition0"); fire_197_transition0}}
    :: d_step{ rd_198_transition1 -> {printf("PROCESSED_198_transition1"); fire_198_transition1}}
    :: d_step{ rd_199_transition2 -> {printf("PROCESSED_199_transition2"); fire_199_transition2}}
    :: d_step{ rd_200_transition3 -> {printf("PROCESSED_200_transition3"); fire_200_transition3}}
    :: d_step{ rd_201_transition4 -> {printf("PROCESSED_201_transition4"); fire_201_transition4}}
    :: p[7] -> goto accept
  od;
  accept: printf("Accepted");
}
```

Listing 5.7: Erweitertes Promela-Modell

5.4.2.4 Aktive Startereignisse in inneren Compliance-Scopes

Da die den Startereignissen entsprechenden Plätze der Petrinetze von inneren Compliance-Scopes markiert sind, werden die Tasks entlang eines Pfades nicht in der erwarteten Reihenfolge aktiviert. Dadurch wird beim Model-Checking beispielsweise bemängelt, dass die Compliance-Regel ◇(¬b *U* a) im Diagramm in Abbildung 5.15 nicht erfüllt ist. In dem weiter unten in der Abbildung angegebenen Gegenbeispiel ist erkennbar, dass Task b vor dem Task a ausgeführt wird.

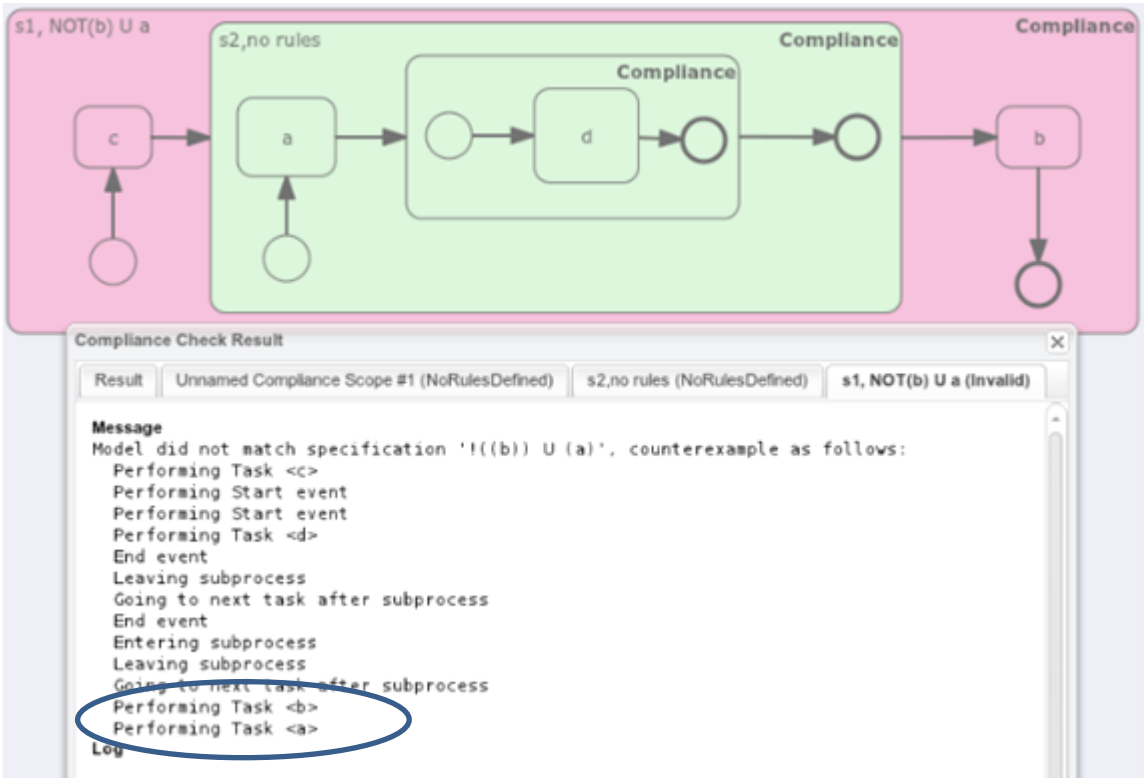


Abbildung 5.15: Aktive Startplätze bei inneren Compliance-Scopes

Alle Plätze des Petrinetzes besitzen einen Index beginnend bei null, wobei ein Startplatz abhängig von dem BPMN-Modell auf einem Platz mit Index kleiner vier liegen kann. Der Startplatz eines inneren Compliance-Scopes hat jedoch relativ zu seinem äußeren Compliance-Scope den Startplatz-Index größer drei. Daher wurde in der Java-Klasse PromelaExport die Schleife, die alle Plätze des Petri-Netzes abarbeitet so geändert, dass die Plätze mit einem Index größer drei nicht markiert werden (vgl. „Ende zweiter Schritt“ in Listing 5.3).

6 Zusammenfassung

In dieser Arbeit wird das Konzept der automatischen Konsistenzprüfung [SALS10] von verschachtelten Geschäftsprozessmodellen umgesetzt. Die Konsistenzprüfung erfolgt dabei mittels logischer Erfüllbarkeitsprüfung von verknüpften Compliance-Regeln. Das auf der Aussagenlogik basierende Konzept wird auf die lineare temporale Logik (LTL) übertragen. Die Umsetzung erfolgt in einem Prototyp [Gro10], dem webbasierten BPMN-Editor Oryx, der in einer vorhergehenden Arbeit um die Compliance-Prüfung von Geschäftsprozessmodellen mittels Model-Checking erweitert wurde.

Nach einer Einleitung wird im zweiten Kapitel diese Arbeit im Geschäftsprozessmanagement eingeordnet und die Relevanz des Ansatzes zur Compliance-Durchsetzung verdeutlicht. Anschließend werden die notwendigen Grundlagen der LTL, des Model-Checking und der Erfüllbarkeitsprüfung vorgestellt. Im nächsten Kapitel werden die verwandten Arbeiten beschrieben. Dabei werden sowohl das oben genannte Konzept der Konsistenzprüfung als auch der Prototyp genauer erläutert.

Ausgehend von begrifflichen Konventionen werden im Kapitel 4 praktische Beispiele für den Einsatz der automatischen Konsistenzprüfung von Compliance-Regeln gegeben. Um die in [SALS10] definierten direkten und indirekten Konflikte zwischen Compliance-Regeln in LTL anzuwenden, wird untersucht, was unter positiven und negativen Literalen im Rahmen der LTL zu verstehen ist. Die Unterschiede zur Aussagenlogik werden beleuchtet und dabei die temporalen Gültigkeitsbereiche von LTL-Formeln diskutiert. Anschließend wird die Definition der positiven und negativen Regeln eingeführt, die auf der Untersuchung der, den LTL-Formeln entsprechenden, Büchi-Automaten basiert. Anhand dieser Definition werden einige direkte und indirekte Konflikte zwischen Compliance-Regeln beispielhaft diskutiert und Schlussfolgerungen für die praktische Umsetzung gezogen. Außerdem werden sogenannte potentielle Konflikte aufgrund von Disjunktionen eingeführt. Des Weiteren wird auf die Erweiterung der Compliance-Prüfung in Bezug auf den Prototyp eingegangen. Um die Fehlersuche bei inkonsistenten Compliance-Regeln zu erleichtern, werden dabei einzelne LTL-Formeln und die aus ihnen zusammengesetzten Compliance-Regeln vor der Compliance-Prüfung auf Erfüllbarkeit und Gültigkeit geprüft. Darüber hinaus wird ein Überblick über die zu implementierende Erweiterung der Compliance-Prüfung gegeben. Dabei erfolgt die Konsistenzprüfung verknüpfter Compliance-Regeln vor dem Model-Checking. Im Falle einer Inkonsistenz wird die Prüfung aller inneren Compliance-Scopes des aktuellen Scopes abgebrochen. Anderenfalls erfolgt mit dem Model-Checking die Zusammensetzung der an die inneren Compliance-Scopes weiterzugebenden Teilregeln. Diese werden anschließend an innere Compliance-Scopes weitergegeben und mit ihren Compliance-Regeln verknüpft.

Im Kapitel 5 werden die bearbeiteten Komponenten des Prototyps anhand eines Komponentendiagramms erläutert und anschließend die Details der Implementierung im Front- und Backend beschrieben. Während zu Erfüllbarkeits- und Gültigkeitsprüfungen die entsprechenden Sequenzdiagramme erläutert werden, erfolgt die Beschreibung der Weitergabe von Compliance-Regeln anhand von Pseudocodes, die im Anhang zu finden sind. Des Weiteren wird die Erweiterung des Model-Checking beschrieben, die das Model-Checking mit den LTL-Operatoren *Globally* und *Until* ermöglicht.

7 Ausblick

Während der Umsetzung wurde Konzept um neue Erkenntnisse erweitert. Im Folgenden wird auf die Bestandteile des Konzeptes, die aus Zeitgründen nicht implementiert werden konnten, sowie weiterführende Ideen eingegangen. Des Weiteren wird ein Hinweis bezüglich der Weiterentwicklung der Abbildung des BPMN-Modells auf Promela gegeben.

Automatische Erfüllbarkeits- und Gültigkeitsprüfung im LTL-Editor

In der aktuellen Version erfolgt die Erfüllbarkeits- und Gültigkeitsprüfung modellierter LTL-Formeln manuell über einen Toolbar-Button im LTL-Editor. Wenn der Benutzer vergisst eine LTL-Formel zu prüfen, können unerfüllbare oder gültige Compliance-Regeln im Compliance-Wizard zusammengesetzt werden. Wie im Abschnitt 4.3.1 des Konzeptes erwähnt, könnten die LTL-Formeln bereits beim Speichern geprüft werden. Dabei sollte es nicht möglich sein eine unerfüllbare oder gültige LTL-Formel abzuspeichern. In dieser Arbeit wurde zunächst nur die manuelle Überprüfung von LTL-Formeln implementiert, weil für einige Tests unerfüllbare und gültige LTL-Formeln benötigt wurden.

Die manuelle Prüfung der LTL-Formeln hat jedoch keine negativen Folgen auf die Compliance-Prüfung, weil Compliance-Regeln im Compliance-Wizard, dem Compliance-Regel-Editor, beim Speichern automatisch auf Erfüllbarkeit und Ungültigkeit geprüft werden. Es ist somit nicht möglich eine unerfüllbare oder ungültige Regel einem Compliance-Scope zuzuweisen. Jedoch kann im Falle einer unerfüllbaren Compliance-Regel nicht sofort auf die dazu führenden LTL-Formeln geschlossen werden.

Automatische Erkennung von Formeln mit bestimmten Gültigkeitsbereichen

Die Beispiele in Abschnitt 4.2.7 zeigen, dass es mit dem aktuellen Ansatz zur Erkennung positiver Teilregeln (siehe Definition 4.1) nicht für alle Formeln sicher entschieden werden kann, ob sie weitergegeben werden sollen. Beispielsweise sollte (a *U* b) nur an die Compliance-Scopes weitergegeben werden, die sich auf den Startzustand beziehen (siehe Abbildung 4.8). Die Formel ◇□a wird als negativ erkannt und wird an alle Compliance-Scopes weitergegeben, obwohl sie in dem Compliance-Scopes gelten muss, der den Endzustand des ursprünglichen Compliance-Scopes enthält. Daher müssen solche Formeln anders erkannt werden, beispielsweise mit Hilfe von regulären Ausdrücken, das heißt der Suche nach Textmustern in den textuellen Darstellungen der Formeln. Auch eine Weiterentwicklung der Untersuchung von entsprechenden Büchi-Automaten ist denkbar. Um die Compliance-Prüfung dadurch nicht zu belasten, kann die Erkennung bei der Speicherung der LTL-Modelle stattfinden. Dabei kann zu jedem LTL-Modell in einem Eigenschaftsfeld seine Zugehörigkeit zu einem Gültigkeitsbereich gespeichert werden.

Disjunktionen in LTL-Formeln

Im Abschnitt 4.2.6 werden potentielle Konflikte aufgrund von Disjunktionen in den Teilregeln diskutiert. Dabei beschränkt sich die Implementierung des Lösungsansatzes (siehe Abschnitt 4.2.6.5.) nur auf Teilregeln, die nicht LTL-Modelle sind. Das heißt, es werden nur die Oder-Operatoren verarbeitet, die als Knoten des Regelbaums im Compliance-Wizard erstellt wurden. Während der Auswertung des Regelbaums (siehe Abschnitt 5.3.4.2.3) sollten daher auch die Oder-Operatoren innerhalb von LTL-Modellen verarbeitet werden.

Gültigkeitsprüfungsprüfung von Teilregeln

In dieser Arbeit wurde der Compliance-Regel-Editor (Compliance-Wizard) um die automatische Erfüllbarkeits- und Gültigkeitsprüfung von zusammengestellten Compliance-Regeln erweitert (siehe Abschnitt 5.2.2). In einem späten Entwicklungsstadium wurde jedoch die Notwendigkeit der Gültigkeitsprüfung einzelner Teilregeln festgestellt und im Konzept aufgenommen (siehe Beispiel in Abschnitt 4.3.3). Wenn die automatische Erfüllbarkeits- und Gültigkeitsprüfung im LTL-Editor implementiert wird, ist nur die Gültigkeitsprüfung von den Teilformeln erforderlich, die nicht nur aus einem LTL-Modell bestehen.

Markierung lokaler Teilregeln

Durch das Nichtweitergeben von positiven erfüllten Teilregeln (siehe Abschnitt 4.2.5.1) werden in den inneren Compliance-Scopes implizit Inkonsistenzen zu ihren äußeren Compliance-Regeln zugelassen. Diese Zulassung könnte auch explizit durch Markierung der Teilregeln erfolgen, die niemals an innere Compliance-Scopes weitergegeben werden sollen. Das heißt, es könnten analog zu einer Programmiersprache lokale und globale Variablen in Form von Teilregeln definiert werden (vgl. Abschnitt 4.2.7.3). Zwei Anwendungsbeispiele sind dazu im Abschnitt 4.2.7 zu finden. Es handelt sich bei diesen Beispielen um negative Compliance-Regeln, die nicht weitergegeben werden sollen.

Weitergabe von Regeln auch an variable Regionen

In der aktuellen Implementierung der Compliance-Prüfung (siehe Abbildung 5.10) werden die Compliance-Regeln nur an Compliance-Scopes weitergegeben. Der Prototyp wurde in einer parallel laufenden Arbeit unter anderem so weiterentwickelt, dass auch den variablen Regionen (siehe Abbildung 3.10) Compliance-Regeln zugewiesen werden können. Die Compliance-Prüfung kann daher um die Weitergabe von Compliance-Regeln an variablen Regionen erweitert werden. Dabei muss das Model-Checking nach der Erfüllbarkeitsprüfung unterbunden werden, weil es keine Aktivitäten in der variablen Region gibt.

Regel-Vorlagen

Für die fehlerfreie Eingabe von LTL-Formeln sind umfangreiche Kenntnisse dieser Spezifikationssprache erforderlich. Es ist nicht einfach LTL-Formeln richtig zu interpretieren und ist fehleranfällig gewünschte Systemeigenschaften in LTL auszudrücken (siehe Abschnitt 3.1). Die syntaktischen Fehler werden in dem Prototyp bereits dadurch eliminiert, dass die Formeln grafisch in einem LTL-Editor erstellt werden [Gro11]. Die semantischen Fehler können beispielsweise durch Regel-Vorlagen und Regel-Assistenten reduziert werden. Eine Sammlung von Vorlagen für LTL-Formeln kann zum Beispiel in [www12e] gefunden werden.

Verbesserung der Abbildung der BPMN-Modelle auf Promela

Trotz der in dieser Arbeit durchgeführten Erweiterung der Abbildung des Petrinetzes auf Promela um das Model-Checking für LTL-Formeln mit den Operatoren *Globally* und *Until* zu ermöglichen (siehe Abschnitt 5.4), werden noch nicht alle BPMN-Modelle korrekt auf Promela abgebildet. Zur Korrektur kann eine Anpassung der BPMN-zu-Petrinetz-Abbildung notwendig sein, weil in der aktuellen Version nur einmal in einem BPMN-Diagramm vorkommende Tasks mehrmals im Petrinetz gespeichert sind. Dies wird anhand eines Beispiels im Anhang A.5 verdeutlicht.

A. Anhang

A.1. Beispiele für Büchi-Automaten

Folgende Tabellen enthält Büchi-Automaten mit und ohne den „accept_all“-Zustand. Die Textdarstellung wurde mit dem Kommandozeilenbefehl von SPIN „spin -f “<Formel>“ und die grafische Darstellung mit dem Programm GOAL [YKTH12] generiert.

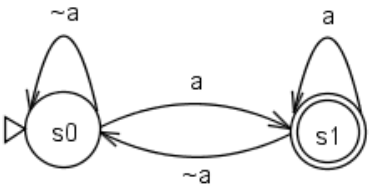
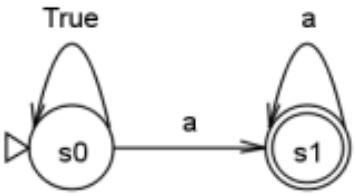
Textdarstellung	Grafische Darstellung
never { /* **[]!(a && b)** */ accept_init: T0_init: if :: (! ((a && b))) -> goto T0_init fi; }	 (Sicherheitseigenschaft)
never { /* **[]<>a** */ T0_init: if :: ((a)) -> goto accept_S9 :: (1) -> goto T0_init fi; accept_S9: if :: (1) -> goto T0_init fi; }	 (Wiederholung)
never { /* **<>[]a** */ T0_init: if :: ((a)) -> goto accept_S4 :: (1) -> goto T0_init fi; accept_S4: if :: ((a)) -> goto accept_S4 fi; }	 (Stabilität)

Tabelle A. 1: Beispiele für Büchi-Automaten ohne des „accept_all“-Zustands

Beispiele für positive Eigenschaften im Sinne der Definition 4.1:

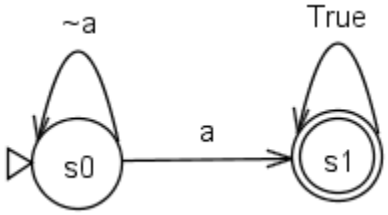
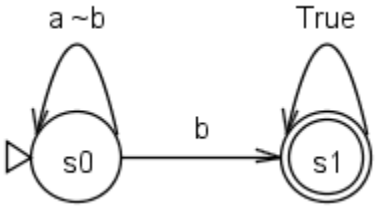
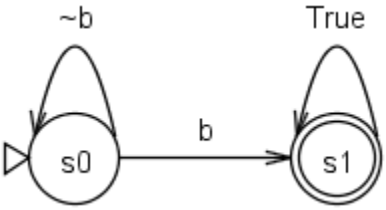
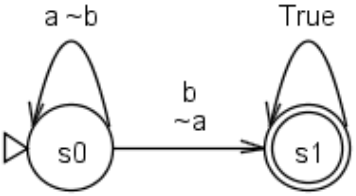
Textdarstellung	Grafische Darstellung
never { /* **<>a** */ T0_init: if :: ((a)) -> goto accept_all :: (1) -> goto T0_init fi; accept_all: skip }	
never { /* **a U b** */ T0_init: if :: ((b)) -> goto accept_all :: ((a)) -> goto T0_init fi; accept_all: skip }	
never { /* **<>(a U b)** */ T0_init: if :: ((b)) -> goto accept_all :: (1) -> goto T0_init fi; accept_all: skip }	
never { /* **[]a -> <>b** */ T0_init: if :: (((! ((a))) || ((b)))) -> goto accept_all :: (1) -> goto T0_init fi; accept_all: skip }	 (Schwache Fairness)

Tabelle A. 2: Beispiele für Büchi-Automaten mit dem „accept_all“-Zustand

A.2. Disjunktionen

```
never  {    /* (!(<>e) || !(<>b)) */
accept_init:
T0_init:
        if
        :: (! ((e))) -> goto accept_S2
        :: (! ((b))) -> goto accept_S5
        fi;
accept_S2:
T0_S2:
        if
        :: (! ((e))) -> goto accept_S2
        fi;
accept_S5:
T0_S5:
        if
        :: (! ((b))) -> goto accept_S5
        fi;
}
```

Listing A. 1: Negative Disjunktion (¬◇e ∨ ¬◇b)

```
never  {    /* (!(<>e) || (<>b)) */
T0_init:
        if
        :: (! ((e))) -> goto accept_S2
        :: ((b)) -> goto accept_all
        :: (1) -> goto T0_S5
        fi;
accept_S2:
        if
        :: (! ((e))) -> goto accept_S2
        fi;
T0_S5:
        if
        :: ((b)) -> goto accept_all
        :: (1) -> goto T0_S5
        fi;
accept_all:
        skip
}
```

Listing A. 2: Positive Disjunktion (¬◇e ∨ ◇b)

A.3. Nicht Co-Safety Eigenschaften

Die Formel (¬a U b) ist positiv im Sinne der Definition 4.1. Ihre Negation ist ebenfalls positiv, weil auch sie den „accept_all“-Zustand enthält. Das heißt, sie ist nicht Co-Safety (siehe Abschnitt 2.2.1.3.3).

```
never  {    /* !(a) U b */
T0_init:
        if
        :: ((b)) -> goto accept_all
        :: (! ((a))) -> goto T0_init
        fi;
accept_all:
        skip
}

never  {    /* !(!(a) U b) */
accept_init:
T0_init:
        If
        :: (! ((b))) -> goto T0_init
        :: (! ((b)) && (a)) -> goto accept_all
        fi;
accept_all:
        skip
}
```

Listing A. 3: Nicht Co-Safety Eigenschaft (positiv)

Sowohl die Eigenschaft □(a→◇b) als auch ihre Negation sind negativ im Sinne der Definition 4.2. Ein weiteres Beispiel ist ◇□b:

```
never  {    /* <>[]b */
T0_init:
        if
        :: ((b)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: ((b)) -> goto accept_S4
        fi;
}

never  {    /* !(<>[]b) */
T0_init:
        if
        :: (! ((b))) -> goto accept_S9
        :: (1) -> goto T0_init
        fi;
accept_S9:
        if
        :: (1) -> goto T0_init
        fi;
}
```

Listing A. 4: Nicht Co-Safety Eigenschaft (negativ)

A.4. Pseudocodes der Operatoren-Klassen

Die folgenden Pseudocodes stellen Ausschnitte aus den Methoden actualCheck() der Operatoren-Klassen (siehe Abbildung 5.8) dar. Sie enthalten nur den Fall ihres Aufrufs mit dem Parameter ***modelcheck=true***. Das heißt, die im Java-Code enthaltene Fallunterscheidung zwischen dem Auslesen der LTL-Formel (siehe Abschnitt 5.3.3) und dem Model-Checking mit Bestimmung der weiterzugebenden Teilregeln (siehe Abschnitt 5.3.4.2) ist in den Pseudocodes nicht enthalten. Das Auslesens und Weitergeben der LTL-Formeln zur Wurzel hin ist jedoch auch hier erkennbar.

```
function AndOperator.actualCheck
                input:  BpmnModel (JSON)
                        ComplianceScope (JSON)
                        Operator (JSON)
                        OperatorLevel (Integer)
                        modelcheck(Boolean)
                output: Ergebnis (ComplianceOperatorResult)
begin
        Ergebnis.Erfuellt = true
        Ergebnis.Positiv = true
        Ergebnis.Formel  = ""
        TeilErgebnis: ComplianceOperatorResult

        for all Operand in Operator.Operanden do

                // Typ des Operands bestimmen und auswerten
                TeilOperator = bestimmeTyp(Operand)
                TeilErgebnis = TeilOperator.actualCheck(…, OperatorLevel+1)

                // Anhand des Ergebnisses eines Operanden wird das Gesamtergebnis bestimmt
                if  not TeilErgebnis.Erfuellt then
                        Ergebnis.Erfuellt = false
                fi

                if not TeilErgebnis.Positiv then
                        Ergebnis.Positiv = false;
                fi

                if OperatorLevel = 1 then

                        if Ergebnis.Positiv and Ergebnis.Erfuellt then
                              Ausgabe = "Operand OMITTED, will not be passed to inner scopes."
                        else
                                Ergebnis.Formel = verknuepfe(Formel, Teilergebnis.Formel)
                        fi

                else
                        Ergebnis.Formel = verknuepfe(Formel, Teilergebnis.Formel)
                fi
        od

        return Ergebnis
end
```

Listing A. 5: Pseudocode der Auswertungsmethode des AndOperators (*modelcheck=true*)

```
function OrOperator.actualCheck
                input:  BpmnModel (JSON)
                        ComplianceScope (JSON)
                        Operator (JSON)
                        OperatorLevel (Integer)
                        modelcheck(Boolean)
                output: Ergebnis (ComplianceOperatorResult)
begin
        Ergebnis.Erfuellt = false
        Ergebnis.Positiv = false
        Ergebnis.Formel  = ""
        TeilErgebnis: ComplianceOperatorResult

        for all Operand in Operator.Operanden do

                // Typ des Operands bestimmen und auswerten
                TeilOperator = bestimmeTyp(Operand)
                TeilErgebnis = TeilOperator.actualCheck(…, OperatorLevel+1)

                // Anhand des Ergebnisses eines Operanden wird das Gesamtergebnis bestimmt
                if  TeilErgebnis.Erfuellt then
                        Ergebnis.Erfuellt = true
                fi

                if TeilErgebnis.Positiv then
                        Ergebnis.Positiv = true
                fi

                if OperatorLevel = 1 then

                        if Ergebnis.Positiv and Ergebnis.Erfuellt then
                                Ausgabe = "… All operands OMITTED."
                                Ergebnis.Formel = ““
                                // Beendigung der Auswertung
                                exit for
                        else
                                Ergebnis.Formel = verknuepfe(Formel, Teilergebnis.Formel)
                        fi

                else
                        Ergebnis.Formel = verknuepfe(Formel, Teilergebnis.Formel)
                fi
        od

        return Ergebnis
end
```

Listing A. 6: Pseudocode der Auswertungsmethode des OrOperators (*modelcheck=true*)

```
function NotOperator.actualCheck
                input:  BpmnModel (JSON)
                        ComplianceScope (JSON)
                        Operator (JSON)
                        OperatorLevel (Integer)
                        modelcheck(Boolean)
                output: Ergebnis (ComplianceOperatorResult)
begin

        Ergebnis.Erfuellt = false
        Ergebnis.Positiv = false
        Ergebnis.Formel  = ""

        // Typ des Operands bestimmen und auswerten
        TeilOperator = bestimmeTyp(Operand)
        TeilErgebnis = TeilOperator.actualCheck(…, OperatorLevel+1)

        // Negation der Formel
        NegierteFormel  = negieren(TeilErgebnis.Formel)
        Ergebnis.Formel  = NegierteFormel

        // Negation des Model-Checking-Ergebnisses
        if not Teilergebnis.Erfuellt then
                Ergebnis.Erfuellt = true
        fi

        BuechiAutomat erzeugen(NegierteFormel  )

        If BuechiAutomat.enthaelt(„accept_all“) then
                Ergebnis.Positiv = true
        else
                Ergebnis.Positiv = false
        fi

        if OperatorLevel = 1 then
                if Ergebnis.Positiv and Ergebnis.Erfuellt then
                        Ausgabe = "Operand OMITTED, will not be passed to inner scopes.”
                        Ergebnis.Formel  = ““
                fi
        fi

        return  Ergebnis
end
```

Listing A. 7: Pseudocode der Auswertungsmethode des NotOperators (*modelcheck=true*)

```
function LTLOperator.actualCheck
                input:  BpmnModel (JSON)
                        ComplianceScope (JSON)
                        Operator (JSON)
                        OperatorLevel (Integer)
                        modelcheck(Boolean)
                output: Ergebnis (ComplianceOperatorResult)
begin

        Ergebnis.Erfuellt = false
        Ergebnis.Positiv = false
        Ergebnis.Formel  = ""

        Formel = LTLTranslator.translate(LTLModell)

        Transformation von BPMN nach Petrinetz
        Transformation von Petrinetz nach Promela-Modell
        Model-Checking(Promela-Modell, Formel)
        if Model erfuellt Formel then
                Ergebnis.Erfuellt = true
        else
                Ergebnis.Erfuellt = false
        fi

        BuechiAutomat erzeugen(Formel)

        If BuechiAutomat.enthaelt(„accept_all“) then
                Ergebnis.Positiv = true
        fi

        if OperatorLevel = 1 then

                if Ergebnis.Positiv and Ergebnis.Erfuellt then
                        Ausgabe = "Operand OMITTED, will not be passed to inner scopes.”
                        Ergebnis.Formel  = ““
                else
                        Ergebnis.Formel  = Formel
                fi

        else
                Ergebnis.Formel  = Formel
        fi

        return  Ergebnis
end
```

Listing A. 8: Pseudocode der Auswertungsmethode des LTLOperators (*modelcheck=true*)

A.5. Das UND-Gateway

Ein Task zu viel

Im Allgemeinen ist es in echten Prozessen nicht vorhersehbar, ob bestimmte Aktivitäten zur gleichen Zeit ausgeführt werden [Hol03, For02]. Im Diagramm in der Abbildung A. 1 sollte daher die Formel und ◇(a ∧ ¬b) verletzt sein. Denn ◇(a ∧ ¬b) drückt aus, dass es garantiert einen Zustand gibt, in dem a=true und b=false gilt. Dies ist nur möglich, wenn immer zuerst a ausgeführt wird.

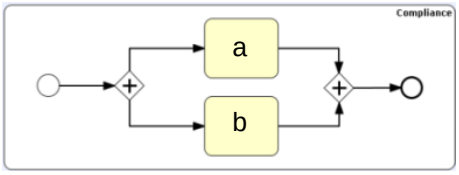


Abbildung A. 1: BPMN-Diagramm für ◇(a ∧ b)

Das Model-Checking des Modells in Abbildung A. 1. ergibt jedoch ein positives Ergebnis, das heißt ◇(a ∧ ¬b) wird fälschlicherweise erfüllt. Das passiert zum einen aufgrund der Erweiterung der Petrinetz-zu-Promela-Abbildung (siehe Abschnitt 5.4.2.3) und zum anderen auf der in dieser Arbeit nicht veränderten BPMN-zu-Petrinetz-Abbildung.

Obwohl es in Abbildung A. 1. nur einen Task a gibt, wird dieser im Promela-Modell zwei Mal definiert (siehe Listing A. 9), jedoch beim zweiten Mal überschrieben. Die oben genannte Erweiterung führt dazu, dass die in Promela definierten Tasks nicht überschrieben werden. Daher bleibt die erste Zuordnung des Tasks a zum Platz p[3] in der neuen Version bestehen (siehe Abbildung A. 2). Die Formel ◇(a ∧ ¬b) wird erfüllt, weil der Task a neben Platz [9] auch dem Platz p[3] zugewiesen ist. Somit gilt nach dem Schalten des UND-Gateways immer a=*true*.

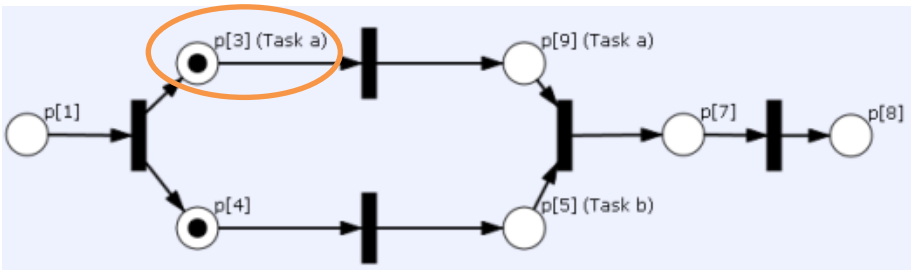


Abbildung A. 2: Petri-Netz für ◇(a ∧ b)

```
byte p[10];

#define a false
#define b p[5]
#define a p[3]
#define b p[5]
#define a p[9]
#define rd_79_transition0 p[1] && !p[3] && !p[4]
#define fire_79_transition0 p[1] = 0; p[3] = 1; p[4] = 1;
#define rd_80_transition1 p[4] && !p[5]
#define fire_80_transition1 p[4] = 0; p[5] = 1;
#define rd_81_transition2 p[5] && p[9] && !p[7]
#define fire_81_transition2 p[5] = 0; p[9] = 0; p[7] = 1;
#define rd_82_transition3 p[7] && !p[8]
#define fire_82_transition3 p[7] = 0; p[8] = 1;
#define rd_83_transition4 p[3] && !p[9]
#define fire_83_transition4 p[3] = 0; p[9] = 1;
active proctype test()
{
  d_step { p[0] = 0; p[1] = 1; p[2] = 0; p[3] = 0; p[4] = 0; p[5] = 0; p[6] = 0; p[7] = 0; p[8] = 0; p[9] = 0;
}
  do
    :: rd_79_transition0 -> d_step{printf("PROCESSED_79_transition0"); fire_79_transition0}
    :: rd_80_transition1 -> d_step{printf("PROCESSED_80_transition1"); fire_80_transition1}
    :: rd_81_transition2 -> d_step{printf("PROCESSED_81_transition2"); fire_81_transition2}
    :: rd_82_transition3 -> d_step{printf("PROCESSED_82_transition3"); fire_82_transition3}
    :: rd_83_transition4 -> d_step{printf("PROCESSED_83_transition4"); fire_83_transition4}
    :: p[8] -> goto accept
  od;
  accept: printf("Accepted");
}


never  {    /*  !(<>(a))  */
accept_init:
T0_init:
            if
            :: (! ((a))) -> goto T0_init
            fi;
}
```

Listing A. 9: Promela-Modell für zwei parallele Tasks (vor der Erweiterung)

Literaturverzeichnis

[AH92] Rajeev Alur and Thomas Henzinger. Logics and models of real time: A survey. In J. de Bakker, C. Huizing, W. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0031988.

[BB94] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st annual Design Automation Conference*, DAC '94, pages 596–602, New York, NY, USA, 1994. ACM.

[BBDER01] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Form. Methods Syst. Des.*, 18(2):141–163, March 2001.

[Bur04] W. Burr. *Innovationen in Organisationen*. Organisation und Führung. Kohlhammer, 2004.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CGL96] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[CPP93] Joelle Cohen, Dominique Perrin, and Jean-Eric Pin. On the expressive power of temporal logic. *J. COMPUT. SYSTEM SCI*, 46:271–294, 1993.

[DAC98] Matthew Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[Deh04] Deharbe, David. Techniques for Temporal Logic Model Checking. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 315–367. Springer, 2004.

[DLP04] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT: an Extensible Model Checking Library Using Transition-Based Generalized Buchi Automata. In *IN PROC. OF MASCOTS'04*, pages 76–83. IEEE Computer Society, 2004.

[DOW08] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx – an open modeling platform for the bpm community. In *BPM*, volume 5240 of *LNCS*. Springer, 2008.

[EH84] E. A Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time. Technical report, Austin, TX, USA, 1984.

[EKA11] O. Engels, J. Kunz, and Bretschneider A. Compliance - Modeerscheinung oder Chefsache? http://www.kpmg.de/Presse/26678.htm, 2011.

[EL87] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[Eme95] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1995.

[Fis97] Michael Fisher. A normal form for temporal logic and its application in theorem-proving and execution. *Journal of Logic and Computation*, 7:429–456, 1997.

[Fis11] M. Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons, 2011.

[FM09] John Franco and John Martin. A history of satisfiability. In *Handbook of Satisfiability*, pages 3–74. 2009.

[For02] M. Forte. *Unschärfen in Geschäftsprozessen*. Weißensee-Verl., 2002.

[FPR06] Daniel Fötsch, Elke Pulvermüller, and Wilhelm Rossak. Modeling and verifying workflow-based regulations. In *ReMo2V*, 2006.

[Gas] Paul Gastin. Ltl 2 ba : fast translation from ltl formulae to bьchi automata. http://www.lsv.ens-cachan.fr/ gastin/ltl2ba/index.php. Zulentzt abgerufen am 04.08.2012.

[GNTV01] Enrico Giunchiglia, Massimo Narizzano, Armando Tacchella, and Moshe Y. Vardi. Towards an efficient library for sat: a manifesto. *Electronic Notes in Discrete Mathematics*, 9:290–310, 2001.

[Gro11] Stefan Grohe. Visualisierung und Implementierung von Compliance Scopes. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011.

[HE10] F. Hülsberg and O. Engels. Compliance Management-Systeme. http://www.kpmg.de/Themen/21705.htm, 2010.

[HL11] M. Hofmann and M. Lange. *Automatentheorie und Logik*. Springer, 2011.

[HN09] H.R. Hansen and G. Neumann. *Wirtschaftsinformatik 1: Grundlagen und Anwendungen*. Uni-Taschenbücher M. Lucius & Lucius, 2009.

[Hol03] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[HRT05] M.S. Hacid, Z.W. Ras, and S. Tsumoto. *Foundations of Intelligent Systems: 15th International Symposium ISMIS 2005, Saratoga Springs, NY, USA, May 25-28, 2005, Proceedings*. Lecture Notes in Computer Science. Springer, 2005.

[HT10] Christian Haubelt and Jürgen Teich. *Digitale Hardware/Software-Systeme: Spezifikation und Verifikation*. eXamen.press. Springer, 2010.

[Joc10] R. Jochem. *Prozessmanagement: Strategien, Methoden, Umsetzung*. Symposion Publishing GmbH, 2010.

[JR10] L. Jansen and P. Roesch. Grundsätze ganzheitlicher Compliance: (GgC). http://www.ganzheitliche-compliance.de/compliance.pdf, 2010.

[Kal12] Patrick Kalbhenn. Kontrolle schafft vertrauen. http://www.handelsblatt.com/unternehmen/buero-special/compliance-kontrolle-schafft-vertrauen/6640652.html, Mai 2012. Zuletzt abgerufen am 20.07.2012.

[KFB04] W. Köhler-Frost and U. Bergweiler. *Outsourcing. Schlüsselfaktoren der Kundenzufriedenheit*. Erich Schmidt Verlag, 2004.

[Kin94] Ekkart Kindler. Safety and Liveness Properties: A Survey. In *EATCS Bulletin*, number 53, pages 268–272. June 1994.

[Köt10] Falko Kötter. Prozessvarianten in unternehmensübergreifenden Servicenetzwerken. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2010. http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3046&engl=0.

[Kri71] S.A. Kripke. *Semantical Considerations on Modal Logic ; Naming and Necessity*. Oxford University Press, 1971.

[KSMP07] Marwane El Kharbili, Sebastian Stein, Ivan Markovic, and Elke Pulvermüller. Towards a Framework for Semantic Business Process Compliance Management, 2007.

[Kup06] Orna Kupferman. Sanity checks in formal verification. In *Proc. of CONCUR'06, LNCS*, pages 37–51. Springer, 2006.

[KYV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001.

[LL10] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Dpunkt.Verlag GmbH, 2010.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Berlin / Heidelberg, 1985. 10.1007/3-540-15648-8_16.

[MMW07] K. Metzlaff, A. Möhlenkamp, and K. Westermann. Leitfaden Kartellrecht. http://www.vilf.de/BDI_Leitfaden-Kartellrecht.pdf, 2007. Zuletzt abgerufen am 03.08.2012.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. The Temporal Logic of Reactive and Concurrent Systems. Springer, 1992.

[Pnu77] Amir Pnueli. The temporal logic of programs. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:46–57, 1977.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer Berlin / Heidelberg, 1986.

[PR10] T. Peek and M. Rode. Compliance im Wandel. http://www.deloitte.com/assets/Dcom-Germany/Local%20Assets/Documents/09_Finanzdienstleister/2010/de_FS_R_Compliance_im_Wandel_150410.pdf, 2010. Zuletzt abgerufen am 30. April 2012.

[Que11] H. Quentmeier. *Praxishandbuch Compliance: Grundlagen, Ziele und Praxistipps für Nicht-Juristen*. SpringerLink : Bücher. Gabler Verlag, 2011.

[RMLD08] Stefanie Rinderle-Ma, Linh Thao Ly, and Peter Dadam. Business process compliance (aktuelles schlagwort). *EMISA Forum*, pages 24–29, August 2008.

[Rös09] P. Rösch. Compliance - oder sagen Sie einfach Regelkonformitдä. http://www.roesch-unternehmensberatung.de/compliance.htm, 2009. Zuletzt abgerufen am 15.07.2012.

[RV01] Moshe Vardi Rice and Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001 (LNCS Volume 2031*, pages 1–22. Springer-Verlag, 2001.

[RV07] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In *14th Workshop on Model Checking Software (SPIN '07)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, pages 149–167. Springer-Verlag, 2007.

[Sac08] Stefan Sackmann. Automatisierung von Compliance. *HMD - Praxis Wirtschaftsinform.*, 263, 2008.

[SALM09] Daniel Schleicher, Tobias Anstett, Frank Leymann, and Ralph Mietzner. Maintaining Compliance in Customizable Process Models. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pages 60–75. Springer Berlin / Heidelberg, 2009.

[SALS10] Daniel Schleicher, Tobias Anstett, Frank Leymann, and David Schumm. Compliant business process design using refinement layers. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems - Volume Part I*, OTM'10, pages 114–131, Berlin, Heidelberg, 2010. Springer Verlag.

[Sch00] U. Schöning. *Logik für Informatiker*. Spektrum Akademischer Verlag, 2000.

[Sch08] U. Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag, 2008.

[SGN07] Shazia Wasim Sadiq, Guido Governatori, and Kioumars Namiri. Modeling control objectives for business process compliance. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM 2007*, volume 4714 of *Lecture Notes in Computer Science*, pages 149–164, Berlin, 2007. Springer.

[SS08] Hermann J. Schmelzer and W. Sesselmann. *Geschäftsprozessmanagement in der Praxis*. Hanser, 2008.

[SS11] Sigrid Schubert and Andreas Schwill. Sprachen, Automaten und Netze. In *Didaktik der Informatik*, pages 253–274. Spektrum Akademischer Verlag, 2011. 10.1007/978-3-8274-2653-6_11.

[SSL10] Jun Sun, Songzheng Song, and Yang Liu. Model checking hierarchical probabilistic systems. In *ICFEM*, pages 388–403, 2010.

[SWLS10] Daniel Schleicher, Monika Weidmann, Frank Leymann, and David Schumm. Compliance Scopes: Extending the BPMN 2.0 Meta Model to Specify Compliance Requirements. In *Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference on*, Dezember 2010.

[Tsc07] Willi Tscheschner. Oryx Dokumentation. Bachelorarbeit, Universität Potsdam, 2007.

[Var97] Moshe Y. Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.

[Var99] Moshe Y. Vardi. Automata-Theoretic Approach to Automated Verification. http://www.cs.rice.edu/\textasciitildevardi/av.html, 1999. Zuletzt abgerufen am 06.07.2012.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[WDHR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *In Proc. of CAV 2006, LNCS 4144*, pages 17–30. Springer-Verlag, 2006.

[WDMR08] M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. Antichains: alternative algorithms for LTL satisfiability and model-checking. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 63–77, Berlin, Heidelberg, 2008. Springer-Verlag.

[WK06] Karl Wagner and Jörg Klückmann. Prozessdesign als Grundlage von Compliance Management, Enterprise Architecture und Business Rules. In August-Wilhelm Scheer, Helmut Kruppke, Wolfram Jost, and Herbert Kindermann, editors, *AGILITÄT durch ARIS Geschftsprozessmanagement*, pages 125–136. Springer Berlin Heidelberg, 2006.

[Wul12] M. De Wulf. ALASKA. http://lit2.ulb.ac.be/alaska, 2012. Zuletzt abgerufen am 10.05.2012.

[wwwa] AJAX Tutorial. http://www.w3schools.com/ajax/default.asp. Zuletzt abgerufen am 12.03.2012.

[wwwb] Introducing JSON. http://www.json.org/. Zuletzt abgerufen am 19.07.2012.

[wwwc] Java Servlet Technology Overview. http://www.oracle.com/technetwork/java/javaee/servlet/index.html. Zuletzt aufgerufen am 12.03.2012.

[wwwd] Ltl model checking. http://maude.cs.uiuc.edu/maude2-manual/html/maude-manualch10.html. Zuletzt abgerufen am 30.07.2012.

[wwwe] Maude overview. http://maude.cs.uiuc.edu/overview.html. Zuletzt aufgerufen am 30.07.2012.

[wwwf] Object management group business process model and notation. http://www.bpmn.org/. Zuletzt abgerufen am 11.03.2012.

[wwwg] Oryx Editor. http://code.google.com/p/oryx-editor. Zuletzt abgerufen am 19.07.2012.

[wwwh] Promela Manual Pages. http://spinroot.com/spin/Man/promela.html. Zuletzt abgerufen am 26.07.2012.

[wwwi] Prototype JavaScript Framework. http://www.prototypejs.org/. Zuletzt abgerufen am 12.03.2012.

[wwwj] Signavio Process Editor - Software as a Service. http://www.signavio.com/de/produkte/process-editor-as-a-service.html. Zuletzt abgerufen am 19.07.2012.

[wwwk] Termersetzungssystem. http://www.de.wikipedia.org/wiki/Termersetzungssystem. Zuletzt abgerufen am 30.07.2012.

[wwwl] The Oryx Project. http://bpt.hpi.uni-potsdam.de/Oryx. Zuletzt abgerufen am 19.07.2012.

[www10a] Compliance und Nachhaltigkeit. http://nachhaltigkeit.daimler.com/reports/daimler/annual/2010/nb/German/20202030/compliance-und-nachhaltigkeit.html, 2010. Zuletzt abgerufen am 20.07.2012.

[www10b] How to develop an editor plugin. http://code.google.com/p/oryx-editor/wiki/HowToDevelopAnEditorPlugin, Februar 2010. Zuletzt aufgerufen am 12.03.2012.

[www11a] BPMN 2.0 - Business Process Model and Notation. http://www.bpmb.de/images/BPMN2_0_Poster_DE.pdf, 2011. Zuletzt abgerufen am 10.03.2012.

[www11b] Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0/PDF, Januar 2011. Zuletzt abgerufen am 10.03.2012.

[www12a] Awards. 2001 – Gerard Holzmann. http://awards.acm.org/citation.cfm?id=0750084&srt=alpha&alpha=H&aw=149&ao=SOFTWSYS&yr=2001, 2012. Zuletzt abgerufen am 05.07.2012.

[www12b] Basel III. Aufseher wollen Umsetzung von Banken-Regeln überwachen. http://www.spiegel.de/wirtschaft/soziales/basel-iii-aufseher-wollen-umsetzung-von-banken-regeln-ueberwachen-a-807884.html, 2012. Zuletzt abgerufen am 20.07.2012.

[www12c] Commission proposes a comprehensive reform of data protection rules to increase users' control of their data and to cut costs for businesses. http://europa.eu/rapid/pressReleasesAction.do?reference=IP/12/46&format=HTML&aged=1&language=DE&guiLanguage=en, Januar 2012. Zuletzt abgerufen am 20.07.2012.

[www12d] Überregulierung im Finanzsektor abbauen. http://www.fpmi.de/de/themen/Ueberregulierung.html, 2012. Zuletzt aufgerufen am 29.07.2012.

[www12e] Property Pattern Mappings for LTL. http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml, Zuletzt abgerufen am 04.07.2012.

[www12f] Amir Pnueli. http://amturing.acm.org/award_winners/pnueli_4725172.cfm, Zuletzt aufgerufen am 03.07.2012.

[www12g] Temporal logic. http://en.wikipedia.org/wiki/Temporal_logic, Zuletzt aufgerufen am 03.07.2012.

[www12h] The Founding Father of Temporal Logic. http://www.prior.aau.dk, Zuletzt aufgerufen am 03.07.2012.

[YKTH12] Chi-Shiang Liu Yih-Kuen Tsay, Ming-Hsien Tsai and Yu-Shiang Hwang. What Is GOAL. http://goal.im.ntu.edu.tw/wiki/doku.php, April 2012. Zuletzt abgerufen am 06.07.2012.

Erklärung gemäß § 14 Abs. 5 und Abs. 6 der Prüfungsordnung der Universitäten Hohenheim und Stuttgart für den Masterstudiengang Wirtschaftsinformatik

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche einzeln kenntlich gemacht.

Die Masterarbeit habe ich noch nicht in einem anderen Studiengang als Prüfungsleistung verwendet.

Des Weiteren erkläre ich, dass mir weder an den Universitäten Hohenheim und Stuttgart noch an einer anderen wissenschaftlichen Hochschule bereits ein Thema zur Bearbeitung als Masterarbeit oder als vergleichbare Arbeit in einem gleichwertigen Studiengang vergeben worden ist.

Stuttgart-Hohenheim, den _____

Unterschrift: _____
(als Originalunterschrift in beiden Exemplaren der Masterarbeit; **nicht** als Kopie)