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

Diplomarbeit Nr. 3112

Visualisierung und Implementierung von Compliance Scopes

Stefan Grohe

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Daniel Schleicher
begonnen am:	15. November 2010
beendet am:	17. Mai 2011
CR-Klassifikation:	D.2.4, H.4.1, H.5.3

Zusammenfassung

Die Einhaltung sowohl gesetzlicher auch als unternehmensinterner Regulierungen beim Design von Geschäftsprozessen wird ein immer wichtigeres Thema.

Basierend auf vorhergehenden Arbeiten, die das Konzept der Variabilität in BPMN 2.0-Diagrammen durch die Definition von *Referenzprozessen* und *Prozessfragmenten* einführten, wird ein Konzept für Compliance Scopes entwickelt. Dabei werden die bestehenden Diagrammtypen um ein neues BPMN-Element *Compliance Scope* erweitert, welches einen Bereich im Diagramm definiert, in welchem zuvor festgelegte Regeln eingehalten werden müssen. Während der Prozessvariantenbildung wird die Ableitung von Prozessvarianten verhindert, die diese Regeln verletzen. Die Regeln werden dabei in einem Regelbaum zusammen gefasst, der die logische Verknüpfung einzelner Regeln erlaubt. Zwei Operatoren zur Kontroll- und Datenflussanalyse werden definiert. Kontrollflussregeln werden dabei in linearer temporaler Logik definiert und mittels Model Checking überprüft.

Das entwickelte Konzept wird in dem webbasierten Prozesseditor Oryx umgesetzt. Der Ausblick behandelt weiterführende Fragestellungen wie Optimierungen bezüglich Performance und graphischer Darstellung, sowie einen Lösungsansatz, ad-hoc Compliance Checks umzusetzen.

Abstract

Compliance to governmental as well as entrepreneurial regulations is becoming an important factor in the design of business processes.

Based on previous work which introduced the concept of variability to BPMN 2.0 diagrams by defining Reference Processes and Process Fragments, a concept for Compliance Scopes is built. This is realized by extending the existing diagram types by a new BPMN element called Compliance Scope which defines an area in a process diagram in which previously specified requirements have to be matched. During the creation of process variants those compliance rules are checked and thereby prevent the creation of process variants violating those rules. Rules are combined using logical operators to an operator tree. Two rule operators for sequence flow as well as data flow verification are defined. Sequence flow verification is performed using model checking.

The presented concepts are implemented using the web-based editor Oryx. The outlook covers additional issues concerning performance optimizitations, improvements of the graphical interface and an option to perform ad-hoc compliance checks.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einführung

Mit der zunehmenden Vernetzung von Geschäftsprozessen über das Internet ergeben sich neue Herausforderungen für die Prozessmodellierung. Mehrere, weltweit verteilte Unternehmensbereiche, aber auch voneinander unabhängige Unternehmen arbeiten in gemeinsamen Geschäftsprozessen zusammen.

Hierbei können sich Standardgeschäftsprozesse, wie beispielsweise die Abwicklung eines Unfallschadens an einem Auto durch ein Versicherungsunternehmen, bei unterschiedlichen Zusammenarbeitskonstellationen in Details unterscheiden. So arbeitet ein deutschlandweit vertretenes Versicherungsunternehmen an unterschiedlichen Standorten mit anderen Gutachtern und Autowerkstätten zusammen, die jeweils ihre eigenen Prozesse nutzen. Damit entsteht ein Bedarf an Prozessvorlagen, die für die konkrete Instanziierung individuell angepasst werden können. Eine Umsetzungsmöglichkeit besteht in abstrakten Prozessen mit Platzhaltern, die nach dem Baukastenprinzip durch Einsetzen von Teilprozessen konkretisiert werden.

Gleichzeitig müssen beim Prozessdesign immer mehr Regelungen und Auflagen beachtet werden. Diese können dabei aus ganz unterschiedlichen Quellen stammen. Zum einen kann der Gesetzgeber die Nichteinhaltung gesetzlicher Regelungen mit Strafen ahnden. Die weltweite Finanzkrise in den vergangenen Jahren haben neue Gesetze für Banken zur Folge gehabt, die berücksichtig werden müssen [Han10]. Unternehmen können aber auch eigene Richtlinien formulieren, die Einfluss auf die Geschäftsprozesse nehmen. So kann zum Beispiel, je nach Höhe der Schadenssumme, die Durchführung eines zweiten Gutachtens gefordert werden. Aber auch die gewünschte Außenwirkung eines Unternehmens kann sich auf die Unternehmensziele und damit die Prozesse auswirken. Greenpeace führt regelmäßig ein Ranking von IT-Unternehmen anhand der Einhaltung von Umweltrichtlinien durch [Gre10]. Hierbei werden auch die Unternehmensprozesse berücksichtigt.

Die dabei erforderlichen Regeln können unterschiedlicher Natur sein. Neben Regeln, die den Kontrollfluss betreffen, etwa durch Forderung von Vorhandensein bestimmter Aktivitäten oder Einhaltung gewisser Reihenfolgen von Tasks, können Regeln auch Anforderungen an den Datenfluss stellen. Gerade bei der Zusammenarbeit unterschiedlicher Unternehmen spielt hier die Datensicherheit eine wichtige Rolle. Patienten-, Kunden- und Kreditkartendaten müssen besonders sorgfältig geschützt werden. So führte das Bekanntwerden eines Hackerangriffs auf das von Sony betriebene Playstation Network Ende April 2011 zur Verunsicherung der Kunden [Son11].

Damit sieht sich der Nutzer beim Modellieren eines Prozesses mit einer großen Menge von zusätzlichen Anforderungen konfrontiert. Deshalb müssen Konzepte entwickelt werden, die dem Prozessdesigner Hilfsmittel an die Hand geben, um die Einhaltung der Anforderungen

sicherzustellen. Verletzungen von Regelungen müssen in einem frühen Entwicklungsstadium erkannt werden, da eine späte Erkennung kostenintensiv sein oder im schlimmsten Fall die Zahlung von Strafen nach sich ziehen kann. Zusätzlich ist eine Rollenverteilung anzustreben, bei der Compliance Experten für die Umsetzung der Regelungen und Auflagen verantwortlich sind und diese in Prozessvorlagen integrieren.

1.1. Aufgabenstellung

Zielsetzung dieser Arbeit ist die Erarbeitung eines Bedienkonzepts für die graphische Modellierung von Prozessen mit Compliance Scopes. Dazu sollen geeignete Formulierungsmöglichkeiten für die Definition von Regeln sowohl für den Kontroll- als auch den Datenfluss in einem BPMN-Modell entwickelt werden.

Während der Prozessvariantenbildung soll dem Prozessdesigner die Funktion angeboten werden, die aktuelle Prozessvariante auf Einhaltung der zuvor definierten Regeln zu überprüfen. Hierbei ist eine Darstellung anzustreben, die auch für Prozessdesigner verständlich ist, die nicht mit dem dahinter liegenden Verifikationsprozess vertraut sind.

Das erarbeitete Konzept soll als Erweiterung des webbasierten Prozessmodelleditors Oryx umgesetzt werden. Dabei soll Oryx um die Möglichkeit der graphischen Modellierung von Compliance Scopes und Regelbausteinen erweitert werden. Zur Verifikation von den Kontrollfluss betreffenden Regeln soll ein geeigneter Model Checker in Oryx eingebunden werden.

1.2. Gliederung der Arbeit

Kapitel 1 gibt eine Einführung in das Thema der Arbeit und beschreibt die Aufgabenstellung sowie den weiteren Aufbau der vorliegenden Arbeit. In Kapitel 2 werden die funktionalen und nicht-funktionalen Anforderungen an die zu entwickelnde Lösung detaillierter erläutert.

Kapitel 3 beschreibt die notwendigen Grundlagen für die Arbeit und die entsprechende Umsetzung. Themen sind Model Checking sowie die bei der Implementierung von Oryx eingesetzten Techniken. Vorhandene Ansätze zur Überprüfung von Compliance sowie die in einer vorangegangenen Arbeit geleisteten Vorarbeiten finden sich in Kapitel 4.

Auf den im zweiten Kapitel dargestellten Anforderungen basierend wird die entwickelte Lösung in Kapitel 5 beschrieben. Unter anderem wird hier der Regelbaum 5.2 und das Arbeiten mit Compliance Scopes 5.5 erläutert. Kapitel 6 geht näher auf einzelne Aspekte der Implementierung ein. Neben der Grundarchitektur des entwickelten Prototypen wird hier auch auf das Front- und Backend sowie die Erweiterbarkeit eingegangen.

Eine Zusammenfassung der Arbeit und einen Ausblick auf Themen im Kontext der Arbeit bietet Kapitel 7.

2. Anforderungen

An die erarbeitete Lösung wurden eine Reihe von funktionalen und nicht-funktionalen Anforderungen gestellt, die in diesem Kapitel genauer betrachtet und erläutert werden.

Erweiterung der Vorarbeiten aus der Diplomarbeit zum Variabilitätskonzept

In der vorhergehenden Diplomarbeit *Prozessvarianten in unternehmensübergreifenden Servicenetzwerken* [Köt10] wurde Oryx bereits um die Beschreibungsmöglichkeit von Variabilität in BPMN-Modellen erweitert. Damit ist es möglich, Prozesstemplates zu erstellen, die dann entsprechend der aktuellen Anforderungen zu konkreten Prozessvarianten abgeleitet werden können.

Da die Variabilität eng mit dem Konzept der Compliance Scopes verbunden ist, ist es sinnvoll, die neu entwickelte Lösung auf dieser bestehenden Erweiterung aufzubauen.

Unterstützung unterschiedlicher Beschreibungssprachen für Compliance Rules

Je nach Art der Regeln, die in einem Compliance Scope erfüllt sein sollen, ist die Formulierung in unterschiedlichen Beschreibungssprachen sinnvoll.

Zur Beschreibung zeitlicher Anforderungen, wie der Reihenfolge auszuführender Tasks, kann die Linear Temporal Logic (kurz LTL) zum Einsatz kommen. Die Definition von Regeln zum Datenaustausch kann dagegen mit anderen Ausdrucksformen besser und leichter ausgedrückt werden.

Die zu erstellende Lösung sollte daher auch nachträglich leicht um weitere Beschreibungssprachen erweiterbar sein. In der Diplomarbeit ist ein Model Checker für die Auswertung von in LTL formulierter Regeln einzubinden.

Außerdem soll die logische Verknüpfung von Regeln in unterschiedlichen Beschreibungssprachen in einer gemeinsamen Regel für den Compliance Scope ermöglicht werden, sodass in einem einzigen Compliance Scope Regeln mit unterschiedlichen Beschreibungssprachen überprüft werden können.

Einbettung auf Client-Seite Je nach einzubettender Sprache kann eine graphische Modellierung der Regeln sinnvoll erscheinen. Die zu erarbeitende Lösung sollte deshalb die Einbindung graphischer Regeleditoren ermöglichen.

Einbettung auf Server-Seite Hier können für verschiedene Regelbeschreibungssprachen Algorithmen angegeben werden, die für die Verarbeitung der jeweiligen Regel zuständig sind.

Unterstützung von Patterns

Da der Prozesstemplatedesigner eventuell nicht mit der LTL und anderen Beschreibungssprachen vertraut ist, sollten Patterns ihn dabei unterstützen, dennoch zum gewünschten Ziel zu kommen.

Beispiele für konkrete Instanzen von Patterns sind

- Stelle sicher, dass in jedem Fall ein Task mit Namen „durch Geschäftsleitung prüfen“ ausgeführt wird.
- Stelle sicher, dass falls ein Task mit Namen „durch Praktikant bearbeiten“ ausgeführt wird, im nächsten Schritt ein Task mit Namen „durch Praktikantenbetreuer prüfen“ ausgeführt wird.

Um dem Benutzer zu ermöglichen, aus einfacheren Grundregeln komplexere Regeln zu erschaffen, soll die Kombination von Patterns zu einer Compliance-Regel unterstützt werden.

Execution Semantics in BPMN 2.0

Die im Januar erschienene neue Major-Version 2.0 [Obj11] der Business Process Model and Notation (kurz BPMN) definiert im Gegensatz zur vorigen Version 1.2 [Obj09] eine Ausführungssemantik (Execution Semantics). In Ansätzen, die auf der ersten Version der BPMN basieren, wurde deshalb eine eigene Interpretation der Ausführungssemantik verwendet.

Hintergrund der Einführung der Execution Semantics ist der Wunsch BPMN direkt ausführbar zu machen. Gleichzeitig wird aber darauf geachtet die Execution Semantics ähnlich zu der BPEL Semantik zu gestalten, um Interoperabilität zu gewährleisten [Lit08].

Da in dieser Arbeit die zweite Version der BPMN verwendet wird, müssen bei der Übernahme von bestehenden Ansätzen die Unterschiede zur Execution Semantics von BPMN 2.0 berücksichtigt werden.

Performance

In der Usability sind die Reaktionszeiten des Systems eine wichtige Anforderungen [Eur]. Die Informationen müssen in einem Format und Tempo angezeigt werden, die dem Benutzer angepasst sind. Deshalb soll die erarbeitete Lösung so implementiert werden, dass die Antwortzeiten möglichst gering ausfallen.

Hierbei ist zu prüfen, ob die Durchführung von Compliance Checks sofort bei Änderung eines BPMN-Modells möglich ist oder zu langen Wartezeiten führt, sodass nur die Prüfung auf Wunsch durchgeführt werden kann. Im ersteren Falle erhält der Benutzer zwar ein unmittelbares Feedback, ob die von ihm erstellte Prozessvariante alle Regeln erfüllt, dauert der Verifikationsprozess allerdings zu lange, verliert das unmittelbare Feedback an Attraktivität.

Integration in Oryx

Die zu realisierende Lösung ist in Oryx zu integrieren. Dabei sollen vorhandene und dokumentierte Erweiterungsmechanismen wie Plugins und Stencilsets nach Möglichkeit vorrangig genutzt werden, anstatt nicht offiziell vorgesehene. Das erleichtert in der späteren Wartung auch die Migration auf neuere Oryx-Versionen.

Bei der Integration des Model Checkers ist darauf zu achten, dass dieser sowohl unter einer Windows- als auch einer Linuxserverumgebung lauffähig ist. Außerdem soll der Model Checker gekapselt werden, sodass ein späterer Austausch des Model Checkers möglich ist.

Nachvollziehbarkeit von LTL-Regel-Verletzungen

Der Model Checker liefert im Falle der Verletzung einer Regel durch das BPMN-Diagramm ein Gegenbeispiel. Dieses soll nach Möglichkeit entsprechend aufbereitet und dem Endbenutzer zur Verfügung gestellt werden, damit dieser die Regelverletzung nachvollziehen und seine eigene Regel verbessern kann.

3. Grundlagen

Die für die Erarbeitung der Lösung benötigten Grundlagen werden in diesem Kapitel vorgestellt. Dies ist zum einen die Business Process Model and Notation, zum anderen das Thema Model Checking. Der verwendete Model Checker Spin sowie die darin verwendete Programmiersprache Promela wird vorgestellt. Da die Implementierung als Erweiterung von Oryx erfolgt, werden außerdem Oryx und die dort verwendeten Techniken und Technologien vorgestellt.

3.1. Business Process Model and Notation

Die Business Process Model and Notation (BPMN) ist eine graphische Notation zur Beschreibung von Geschäftsprozessen, deren Entwicklung um 2002 begann und die später von der Object Management Group (OMG) zur weiteren Pflege übernommen wurde [Silo9]. Für die Verwendung der BPMN fallen keine Lizenzkosten an, eine Reihe von BPMN-Werkzeugen ist zudem kostenlos verfügbar.

Durch die Ähnlichkeit zu Flowcharts [ISO] sind BPMN-Modelle auch von Businessanwendern einfach zu verstehen. Die BPMN definiert aber im Gegensatz zu Flowcharts eine feste Bedeutung für alle Elemente, was die Kommunikation über in BPMN beschriebene Prozesse vereinfacht. Anders als die Unified Modeling Language (UML) [Obj10] ist die BPMN weniger IT-zentriert und kann damit sowohl von den Geschäftsanwendern als auch von denjenigen, die die technische Umsetzung und Pflege der Geschäftsprozesse vornehmen, verstanden werden [Obj11]. Zusätzlich zu Flowcharts gibt es Event-getriggertes Verhalten [Silo9].

Die graphischen Elemente der BPMN in fünf Grundkategorien eingeteilt [Obj11]:

Flussobjekte Diese definieren als wichtigste graphische Elemente das Verhalten eines Geschäftsprozesses. Insgesamt gibt es drei verschiedene Flussobjekte: Ereignisse, Aktivitäten und Gateways.

Daten Hierzu gehören Datenobjekte, Datenein- und -ausgänge sowie Data Stores.

Verbindende Objekte Diese verbinden Flussobjekte miteinander oder Flussobjekte mit anderen Informationen. Hierbei werden Kontrollfluss, Nachrichtenfluss, allgemeine Verknüpfungen und Datenverknüpfungen unterschieden.

Pools und Swimlanes Mit Swimlanes und Pools werden verschiedene Teilnehmer an einem Geschäftsprozess repräsentiert. Die Kommunikation zwischen Pools findet über Nachrichten statt.

Artefakte Hiermit können Prozesse um zusätzliche Informationen angereichert werden. Die gängigsten Artefakte sind die Gruppierung und die Text Annotation.

Ein Beispielprozess findet sich in Abbildung 3.1.

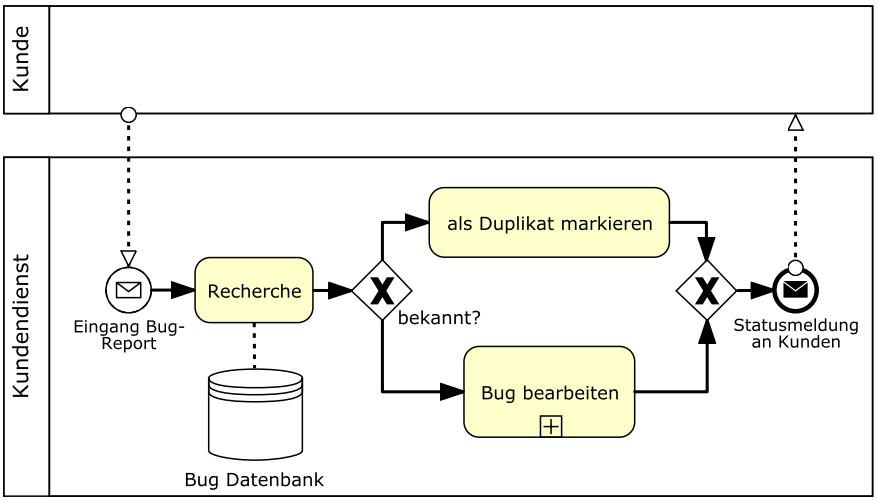


Abbildung 3.1.: Beispiel BPMN-Prozess

Eine Übersicht über die Elemente der BPMN findet sich in [BPM10], die vollständige Spezifikation in [Obj11].

3.2. Model Checking

Mit der zunehmenden Komplexität von Software wird es schwieriger, deren Fehlerfreiheit zu gewährleisten. In besonders kritischen Anwendungen werden deshalb formale Methoden gewöhnlichen Tests vorgezogen. Formale Methoden beschreiben Systeme als mathematische Modelle, um dann die Korrektheit der Systeme zu beweisen [Now09].

In einigen Anwendungsgebieten wird Model Checking bereits intensiv genutzt [CGP01]. So zum Beispiel bei der Verifikation von großen integrierten Schaltungen.

Ein Model Checker benötigt zwei Eingaben, eine Beschreibung des zu überprüfenden Systems und die Spezifikation, die das System erfüllen muss. Anschließend läuft die Verifikation vollautomatisch und liefert im Falle einer Verletzung der Spezifikation ein Gegenbeispiel.

3.2.1. Systembeschreibung

Im ersten Schritt wird ein formales Modell des zu verifizierenden Systems erstellt. Dabei müssen in dem erstellten Modell alle Eigenschaften vertreten sein, die für die Verifikation

der Korrektheit notwendig sind. Eigenschaften, die für die Verifikation dagegen irrelevant sind, werden weggelassen, um das Modell nicht unnötig komplex zu gestalten und beim späteren Verifikationsprozess keine Laufzeiteinbußen zu erzeugen. So werden bei digitalen Schaltungen die anliegenden Spannungen wegabstrahiert und durch boolsche Werte repräsentiert [CGP01].

Die Beschreibung des zu testenden Systems kann dabei unter anderem als Kripkestruktur [CGP01] erfolgen. Eine Kripkestruktur ist ein gerichteter Graph. Die Knoten dieses Graphen beschreiben dabei Zustände des Systems. Jeder Zustand ist mit Bedingungen annotiert, die in diesem Zustand erfüllt sind. Die Kanten des Graphen beschreiben Zustandsübergänge. Diese Zustandsübergänge sind dabei an keine Bedingungen geknüpft. Eine Teilmenge der Zustände wird als Startzustände ausgezeichnet.

Eine Kripkestruktur ist in Abbildung 3.2 gegeben. z_1 ist als Startzustand gekennzeichnet. In diesem Zustand gilt Bedingung a, im Zustand z_7 gelten die Bedingungen b und c. Von Zustand z_1 kann das System entweder in z_2 oder z_4 wechseln, die Wahl des Nachfolgezustands erfolgt nichtdeterministisch.

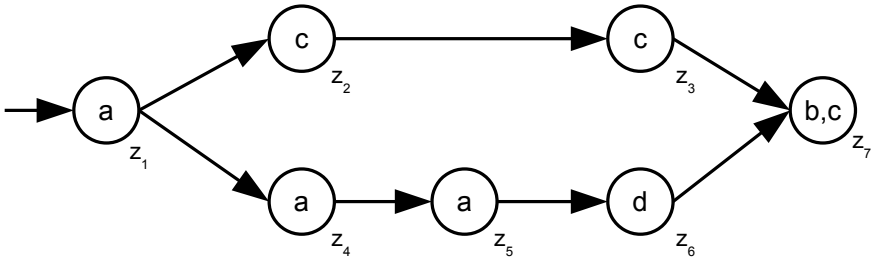


Abbildung 3.2.: Beispiel Kripkestruktur

Auf dem erstellten Modell werden nun zu erfüllende Eigenschaften (die Spezifikation) beschrieben.

3.2.2. LTL

Eine Beschreibungsmöglichkeit für die einzuhaltende Spezifikation ist die lineare temporale Logik (kurz LTL). Die LTL erweitert die Aussagenlogik um temporale, das heißt zeitbezogene, Operatoren. Die Auswertung der Regel erfolgt dabei auf einzelnen Ausführungen des Models. Folgende Operatoren werden dabei unterstützt:

Next ϕ Im nächsten Zustand gilt Bedingung ϕ.

Finally ϕ Irgendwann im weiteren Verlauf der Ausführung gilt Bedingung ϕ.

Globally ϕ Im weiteren Verlauf gilt in jedem Zustand Bedingung ϕ.

ϕ **Until** ψ Bis Bedingung ψ eintritt, gilt Bedingung ϕ und Bedingung ψ tritt irgendwann im weiteren Verlauf ein.

Tabelle 3.1 zeigt eine Auswertung der Operatoren auf der in Abbildung 3.2 gegebenen Kripkestruktur.

Formel / Zustand	z_4	z_2	z_7	z_5
Next d	falsch	falsch	falsch	wahr
Finally d	wahr	falsch	falsch	wahr
Globally c	falsch	wahr	wahr	falsch
a *Until* d	wahr	falsch	falsch	wahr

Tabelle 3.1.: Beispielauswertung LTL-Operatoren

Daneben existiert noch eine Reihe weiterer temporaler Operatoren [Now09], die sich aber, ebenso wie `Finally` und `Globally`, aus den Operatoren der Aussagenlogik sowie `Next` und `Until` darstellen lassen, da diese bereits eine Junktorenbasis bilden:

true $\equiv \phi \vee \neg\phi$

Finally $\phi \equiv$ true Until ϕ

Globally $\phi \equiv \neg$ (Finally $\neg\ \phi$)

ϕ **Release** $\psi \equiv \neg\ (\neg\ \phi$ Until $\neg\ \psi)$

Neben der LTL gibt es noch weitere Beschreibungsmöglichkeiten für temporale Anforderungen. Die Computation Tree Logic (CTL) betrachtet im Gegensatz zur LTL nicht nur eine mögliche Zukunft, sondern berücksichtigt alle möglichen Ausführungen [Now09]. Damit lässt sich zum Beispiel beschreiben, dass mindestens eine mögliche Ausführung des Systems existiert, in der *Task1* ausgeführt wird. Beide Logiken können jeweils Bedingungen formulieren, die sich mit der anderen Logik nicht ausdrücken lassen. Es existiert aber eine Vereinigung von beidem in der CTL*.

3.3. Spin

Spin [Spia] ist ein bewährtes und weit verbreitetes Werkzeug zur Modellprüfung. Die Entwicklung begann bereits 1980 in den Bell Labs. Seit 1991 ist die Software frei verfügbar.

Sowohl die Beschreibung des Systems als auch die Spezifikation der Anforderung erfolgt dabei in Promela. Beide werden anschließend konkateniert und in C-Code umgewandelt. Der C-Code wird mit einem C-Compiler wie dem GCC [GT] kompiliert. Das dabei erzeugte Programm simuliert das spezifizierte System und prüft dabei die Spezifikation.

Neben dem Model Checking ermöglicht Spin auch das Simulieren von Modellen durch deren schrittweises Ausführen und Nachverfolgen des Ablaufs.

3.3.1. Promela

Spin verwendet zur Formulierung der Systembeschreibung eine eigene Programmiersprache die *Process or Protocol Meta Language* (kurz Promela) [Ger97]. Promela ist an C angelehnt und besteht aus drei Typen von Objekten: Prozessen, Nachrichtenkanälen und Variablen. Prozesse definieren dabei das Verhalten des Modells und können über globale Variablen und Nachrichtenkanäle miteinander kommunizieren.

Für das Verständnis der betrachteten Lösungsansätze der Systembeschreibung in Promela sind einige Besonderheiten von Promela relevant, die im Folgenden kurz erläutert werden.

Kommunikation über Channels

Nachrichten werden mit eigenen Operatoren in Channels geschrieben und aus diesen gelesen (siehe Listing 3.1).

```
channel ! token; // Token in Channel schreiben
channel ? token; // Token aus Channel lesen (blockiert)
```

Listing 3.1: Beispiel Channels

Nichtdeterminismus

Auch Nichtdeterminismus lässt sich mit Promela umsetzen. Dies findet zum Beispiel Anwendung bei der zufälligen Wahl des Folgezustandes beim Umformen einer Kripkestruktur in ein Promela-Programm.

```
if
:: condition1 -> statement1
:: condition2 -> statement2
fi;

do
:: condition1 -> statement1
:: condition2 -> statement2
od;
```

Listing 3.2: Beispiel Nichtdeterminismus

Listing 3.2 zeigt das `if`-Konstrukt und die `do`-Schleife in Promela. Beim `if`-Konstrukt werden bei der Ausführung zunächst alle Bedingungen ausgewertet. Sind mehrere Bedingungen erfüllt, so wird anschließend zufällig einer der erfüllten Zweige ausgeführt. Die Interpretation bei der `do`-Schleife erfolgt analog.

Never Claims

Never Claims sind spezielle Prozessarten, die beim Start der Verifikation einmal initialisiert werden und anschließend parallel zu den Prozessen des Modells laufen. Sie dienen dazu, unerwünschtes Verhalten des Modells zu erkennen. Die Prozesse des Modells und der Prozess mit dem Never Claim werden schrittweise abwechselnd ausgeführt. Damit erzeugen die Prozesse des Modells den nächsten Zustand, der im nachfolgenden Schritt dann umgehend durch den Never Claims auf Korrektheit überprüft wird. Wird ein Never Claim beendet, so ist das durch den Never Claim beschriebene unerwünschte Verhalten eingetreten und die Spezifikation verletzt.

Als Beispiel sei die Spezifikation gegeben, dass stets die Bedingung abc gelten muss, als LTL-Formel: `[]abc`. Spin erlaubt nun die Generierung von passenden Promela-Programmen aus LTL-Formeln. Dazu wird Spin mit dem Befehlszeilenparameter `-f` und der darzustellenden LTL-Formel in negierter Form aufgerufen. So wird die LTL-Formel `[]abc` in einen Never Claim wie in Listing 3.3 dargestellt umgeformt.

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

Listing 3.3: Beispiel Never Claim

Sobald ein Prozess des Modells die globale Variable `abc` auf `false` setzt, wird der Never Claim beendet und damit die Spezifikation verletzt.

Damit mehrere Anweisungen in einem Block ausgeführt werden können, bevor der Wechsel zum Never Claim stattfindet, können die Anweisungen in eine `d_step`-Umgebung eingeschlossen werden.

Seit Version 6 können LTL-Formeln auch direkt in der Systembeschreibung angegeben werden, ohne den Umweg über den Never-Claim [ltl].

3.4. Oryx

Oryx [ory] ist ein Open Source Framework zur graphischen Prozessmodellierung, das am Hasso-Plattner-Institut in Potsdam entwickelt wurde und inzwischen sowohl als Open Source Anwendung als auch als kommerzielles Produkt der Firma Signavio verfügbar ist [sig].

Der Client-Teil läuft dabei im Webbrowser und ist somit plattformübergreifend nutzbar. Der Serverteil ist als JavaServlets realisiert. Durch seine Erweiterbarkeit bietet sich Oryx

für Erweiterungen wie die im Rahmen dieser Arbeit entwickelte Lösung an. Im Oryx-Coderepository [ory11] finden sich bereits eine Reihe von Erweiterungen wie Modellierungsmöglichkeiten für Petrinetze, BPMN-Q und UML-Diagramme.

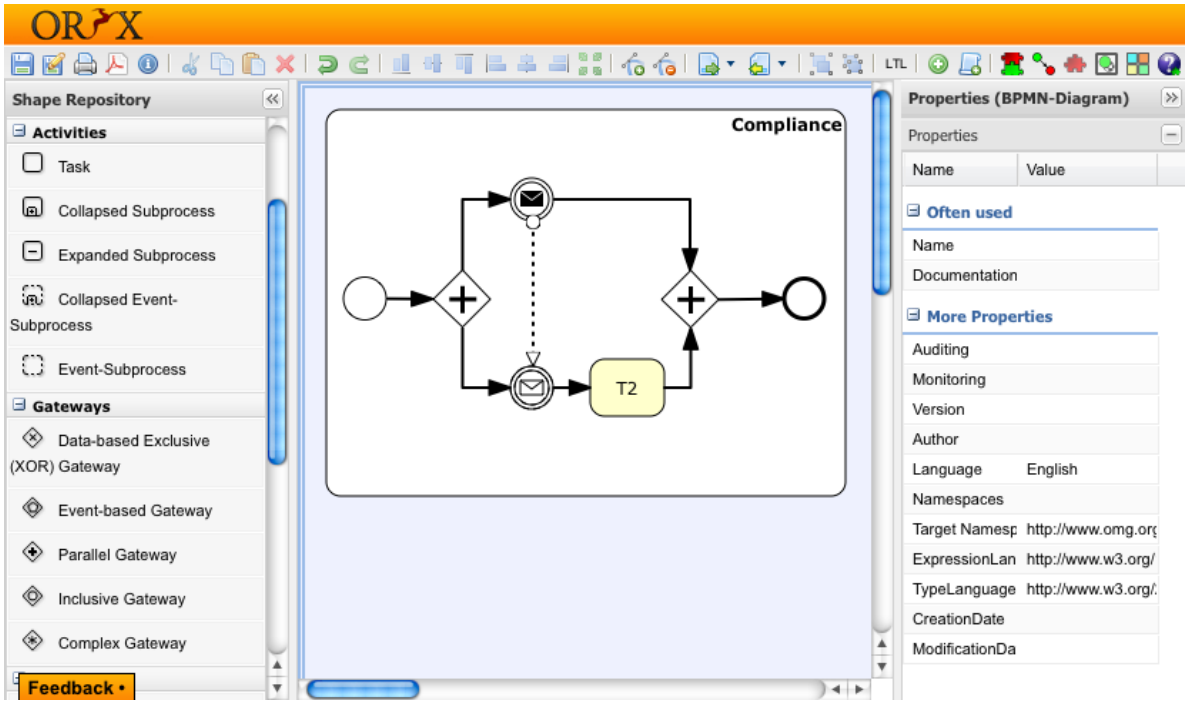


Abbildung 3.3.: GUI von Oryx

3.4.1. Frontend

Die Benutzeroberfläche wird vom Benutzer über den Webbrowser aufgerufen und genutzt. Bei der Entwicklung wurde Wert auf ein *Desktop Feeling* gelegt, das heißt der Editor wurde wie eine Desktop-Anwendung mit den dort gewohnten Konzepten, beispielsweise Drag and Drop, konzipiert [Tsc07]. Wichtige Bestandteile der Architektur des Frontends sind Plugins und Stencilsets, die nachfolgend genauer erläutert werden.

Eine Darstellung der GUI von Oryx findet sich in Abbildung 3.3. Auf der linken Seite befindet sich die Toolbox mit allen für den Diagrammtypen verfügbaren graphischen Primitiven. Auf der rechten Seite befindet sich ein Editor, mit dem Eigenschaften des aktuell markierten Elements modifiziert werden können. Oben befindet sich die Toolbar, in der die verfügbaren Plugins Zusatzfunktionen zur Verfügung stellen.

Stencilsets

Ein Stencilset fasst die graphischen Primitive (Stencils) eines Diagrammtyps zusammen und kann zusätzlich einen Satz von Regeln enthalten, der definiert, wie sich die graphischen Elemente untereinander verhalten [Pet07].

Die graphischen Elemente eines Stencilsets unterteilen sich in Knoten und Kanten und können neben der graphischen Darstellung weitere Eigenschaften, wie einen Namen, aufweisen, die sich über den Eigenschaftseditor im Oryx-Frontend manipulieren lassen.

Ähnlich zur Vererbung in der Objektorientierung bieten Stencilsets die Erweiterung um weitere Stencilset Extensions. Dies wurde zur Definition des Stencilsets zur Variabilität genutzt, welches das BPMN 2.0-Stencilset um Variability Points und Variability Attributes erweitert.

Rules definieren die Regeln, die in einem Diagrammtyp eingehalten werden müssen. Die Dokumentation spezifiziert drei verschiedene Regeltypen:

Connection Rules Hiermit wird definiert, welche graphischen Primitive miteinander verbunden werden können. So kann ein Task mit einer Kontrollflusskante verbunden werden, aber nicht (direkt) mit einem Gateway.

Cardinality Rules Dieser Regeltyp beschränkt unter anderem die Anzahl ausgehender Kanten eines Knotens oder die Anzahl möglicher Instanzen eines graphischen Primitivs in einem Elternelement.

Containment Rules Mit diesem Regeltyp wird angegeben, welche graphischen Primitive andere enthalten können.

Um ähnliches Verhalten und ähnliche Eigenschaften zusammenzufassen, können Stencils Rollen zugeordnet werden [how]. So können weder Tasks noch Gateways weitere BPMN-Elemente enthalten. Datenobjekte und Data Stores können beide nicht mit Kontrollflusskanten verbunden werden.

Neben dem hier erweiterten Stencilset zur BPMN [Pol07] wurden bereits auch Stencilsets für Petrinetze und UML-Diagramme erstellt und in Oryx eingebunden.

Plugins

Die Benutzeroberfläche kann auf Frontend-Seite durch Plugins erweitert werden [Tsc07]. Diese werden beim Aufruf der Benutzeroberfläche geladen und erhalten Zugriff auf eine spezielle Fassade, über die sie auf den Datenbestand und die Funktionen des Frontends zugreifen können.

Die Kommunikation zwischen Plugins erfolgt über Events. Plugins können dabei eigene Events definieren und auslösen, sowie Events abonnieren. Events werden unter anderem dann ausgelöst, wenn Stencils dem Graphen hinzugefügt werden oder sich Eigenschaften der Stencils ändern.

3.4.2. Backend

Das Backend ist in Java realisiert. Erweiterungen um neue Funktionalitäten erfolgen als JavaServlets [Oraa]. Insbesondere aufwändigere Operationen lassen sich einfacher im Backend umsetzen. So ist zum Beispiel der Aufruf von nativen Anwendungen auch nur hier möglich.

3.5. Scalable Vector Graphics

Das in XML formulierte Scalable Vector Graphics Format [svg] erlaubt die Definition von zweidimensionalen Vektorgraphiken. Nachdem SVG inzwischen von den gängigen Browsern relativ vollständig unterstützt wird [Sch11], hat sich SVG zu einem im Web weit verbreiteten Standard entwickelt. In Oryx wird SVG zur Beschreibung der graphischen Primitive in Stencilsets verwendet.

Das Design der SVG-Graphiken muss dabei nicht als SVG von Hand erstellt werden, sondern kann durch eine Reihe graphischer Editoren erfolgen, zum Beispiel mit Inkscape [Ink].

Oryx erweitert die SVG-Syntax um zusätzliche Attribute und Knoten [Pet07]. So erlauben Anchors die Definition von Layoutverhalten im Falle einer Größenänderung eines Elementes durch den Benutzer, Eigenschaftswerte können über das Text-Attribut mit der graphischen Darstellung etwa zur Anzeige des Namens eines Tasks verwendet werden. Mittels Magnets und Dockers wird die Definition von Ankerpunkten beim Versehen von Knoten mit Kanten beschrieben.

3.6. JavaScript Object Notation

Die JavaScript Object Notation (JSON) [jso, Cro08] ist ein leichtgewichtiges Datenaustauschformat. JSON ist für Menschen leicht lesbar und auf Grund seines einfachen Aufbaus leicht zu parsen. Im Gegensatz zu XML ist JSON deutlich kompakter. Damit bietet sich JSON überall dort an, wo die Möglichkeiten von JSON ausreichen und die Bandbreite zum Datentransfer beschränkt ist. JSON basiert auf zwei grundlegenden Datentypen:

- Sammlungen von Name/Wertepaaren zur Beschreibung eines Objektes,
- Arrays zur Definition von Sammlungen von Objekten.

Listing 3.4 stellt einen Datensatz zu einem Auto in JSON dar. Die Ausstattungsmerkmale werden als Array definiert, die Inhaberinformationen als Unterobjekt umgesetzt. Als Datentypen für Eigenschaftswerte stehen Zeichenketten, Dezimalzahlen, `true` und `false`, `null` sowie Arrays und Objekte zur Verfügung.

JSON kann direkt in JavaScript geparst werden [eva], für Java ist ein entsprechender Parser [Cro] verfügbar. In Oryx findet JSON umfangreiche Anwendung. So unter anderem bei

der Definition von Stencilsets, beim Datenaustausch mit dem Backend und als Datenhaltungsformat für erstellte Modelle.

```
{
    "Kennzeichen": "S-AB-1233",
    "Kilometerstand": 18000,
    "TUEV": true,
    "Ausstattung": ["Radio", "Klima"],
    "Inhaber": {
        "Name": "Max Mustermann",
        "Geburtstag": "1980-01-01"
    }
}
```

Listing 3.4: Beispiel JSON

4. Vorhandene Ansätze und Vorarbeiten

In diesem Kapitel werden bestehende Ansätze zur Regelüberprüfung in BPMN-Diagrammen genauer betrachtet. Zunächst werden Compliance Scopes und Compliance Templates vorgestellt und verglichen. Anschließend wird nacheinander auf Ansätze zur Kontroll- und Datenflussverifikation eingegangen. Außerdem wird die bereits in der Diplomarbeit *Prozessvarianten in unternehmensübergreifenden Servicenetzwerken* erarbeitete Lösung betrachtet, auf der in der vorliegenden Arbeit aufgebaut wird.

4.1. Definition des Compliance Scopes

In [SWLS10] werden Compliance Scopes formal definiert. Als Grundlage hierfür dient die Definition des Hypergraphen [Ber89]. Sei $X = x_1, x_2, ..., x_n$ eine Menge von BPMN 2.0-Tasks und E_i eine Hyperkante, dann ist ein Hypergraph über X eine Familie $H = E_1, E_2, ..., E_m$ von Untermengen von X, sodass gilt:

$$E_i \neq \emptyset \text{ für } (i = 1, 2, ...m) \text{ und } \bigcup_{i=1}^{m} E_i = X$$

Das heißt jede Kante muss mindestens einen Task verbinden und jeder Task von X muss von mindestens einer Kante erfasst werden. Sei C die (endliche) Menge aller Compliance-Regeln. Darauf aufbauend wird nun die Menge CS der Compliance Scopes, die auf einen Business Prozess angewendet werden, definiert:

$$CS \subseteq H \times (2^C \setminus \emptyset)$$

Damit ist ein Compliance Scope ein Element aus der Menge CS, das heißt er verbindet eine Hyperkante mit einem Satz Regeln. Dabei hat jeder Compliance Scope mindestens eine verknüpfte Regel, Compliance Scopes ohne Regeln sind damit nicht zugelassen.

4.2. Definition des Compliance Templates

[SALM09] definiert den Begriff des Compliance Templates und beschreibt den Umgang mit diesen.

In Abbildung 4.1 wird ein Compliance Template dargestellt. Dieses besteht aus drei Komponenten: einem abstrakten Geschäftsprozess, sowie Variabilitätsbeschreibungen und

Compliance-Deskriptoren. In dem abstrakten Geschäftsprozess sind Platzhalter enthalten (in der Abbildung mit *Opaque* bezeichnet), die ersetzt werden müssen, bevor der Prozess ausgeführt werden kann. Dazu definieren die Variabilitätsbeschreibungen Alternativen, die für die Platzhalter eingesetzt werden können. Alternativen werden durch verschiedene Typen kategorisiert. So ist die Angabe von expliziten, leeren, freien oder aus Expressions generierten Alternativen möglich.

Zusätzliche Compliance-Regeln können mit den Compliance-Deskriptoren ausgedrückt werden. Nicht mit *Opaque* bezeichnete Aktivitäten sind Compliance-Aktivitäten und dürfen weder entfernt noch anderweitig verändert werden. Compliance-Deskriptoren bestehen aus Compliance-Punkten, von denen jeder eine Compliance-Anforderung an den Geschäftsprozess beschreibt. Jeder dieser Compliance-Punkte besteht wiederum aus einer Menge von Compliance-Links, die auf Aktivitäten im abstrakten Prozess zeigen, die nicht verändert werden dürfen, sowie eine Reihe von Assurance-Regeln, die für einzelne Platzhalter angeben, welche Regeln gelten müssen.

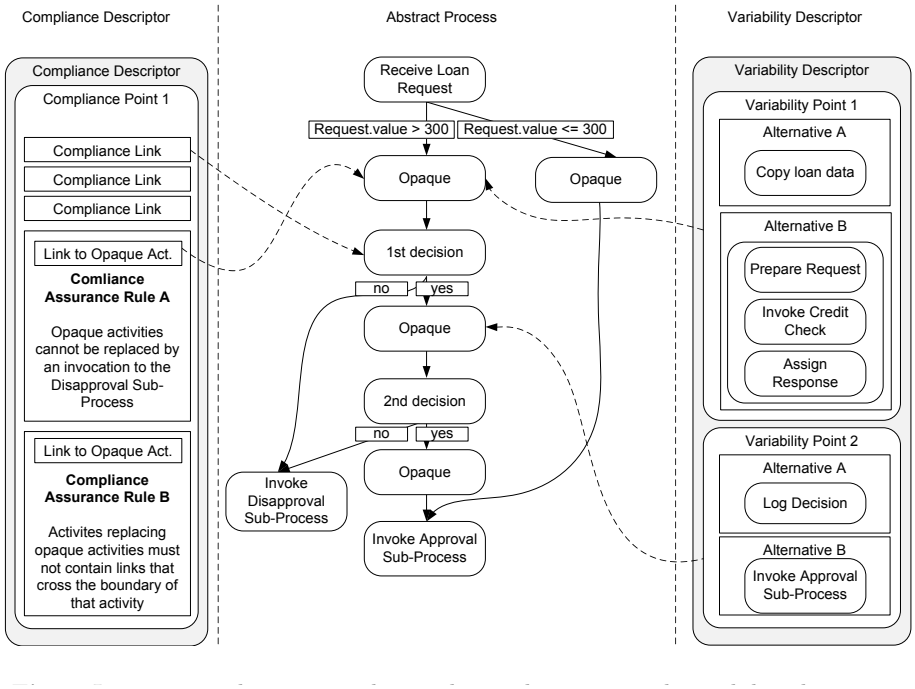


Abbildung 4.1.: Compliance Template [SALM09]

Variabilitätsbeschreibungen und Compliance-Deskriptoren werden dabei unabhängig vom abstrakten Prozess definiert, was eine Wiederverwendung ermöglicht.

Im Gegensatz zur Definition des Compliance Scope lassen sich die Compliance-Regeln hier nicht nachträglich in bestehende Prozesse integrieren, denn *Opaque*-Aktivitäten fehlen in bestehenden, ausführbaren Prozessen.

4.3. Ansätze zur Kontrollflussverifikation

Eine Reihe von Arbeiten beschäftigen sich mit der Verifikation des Kontrollflusses. Zweck der Kontrollflussanalyse ist es unter anderem, sicherzustellen, dass Tasks in der korrekten Reihenfolge ausgeführt oder Tasks im Prozess in jedem Fall durchgeführt werden. Aber auch komplexere Regeln wie *Falls die Aktivität „Gutachten erhalten" ausgeführt wird, dann muss anschließend irgendwann die Aktivität „Gegengutachten einholen" ausgeführt werden.* lassen sich damit überprüfen.

4.3.1. PROPOLS

In [YMH+06] wird die Spezifikationssprache Property Specification Pattern Ontology Language for Service Composition (kurz PROPOLS) vorgestellt. Diese basiert auf den in [DAC98] vorgestellten Property Patterns, welche wiederum Verallgemeinerungen oft genutzter temporaler Formeln sind.

So definiert PROPOLS unter anderem die Patterns *Absence*, *Existence* und *Precedence* über definierten Geltungsbereichen wie *Globally* oder zwischen zwei Ereignissen (*Between ... And*). Ein mögliches kombiniertes Pattern ist in Listing 4.1 dargestellt.

```
Customer.GetOrderFulfilled Precedes Bank.Transfer Globally
And
Customer.GetOrderFulfilled LeadsTo Bank.Transfer Globally
```

Listing 4.1: Pattern in PROPOLS [YMH+06]

Die definierten Properties werden zum Überprüfen von BPEL Service Composition Schemas genutzt. Der Verifikationsprozess läuft in diesem Ansatz dann wie folgt ab:

Schritt 1 Zunächst werden die durch Pattern ausgedrückten Eigenschaften als totale, deterministische, endliche Automaten umgesetzt. Dabei ist auch definiert, wie der resultierende Automat eines kombinierten Patterns aus den Automaten der einzelnen Patterns erstellt wird.

Schritt 2 Anschließend wird das BPEL-Schema in ein endliches, deterministisches, beschriftetes Transitionssystem umgeformt, woraus dann wieder ein totaler, deterministischer und endlicher Automat erstellt wird.

Schritt 3 Die Konformität des BPEL-Schema zu den in PROPOLS definierten Eigenschaften wird ermittelt, indem nachgeprüft wird, ob alle vom BPEL-Automaten akzeptierten Sequenzen auch vom Automaten, der die Eigenschaften repräsentiert, akzeptiert werden.

Dies wiederum erfolgt durch Schnitt des BPEL-Automaten mit dem Komplement des Eigenschafts-Automaten. Ist der Schnitt leer, so ist die definierte Eigenschaft stets erfüllt.

Der Verifikationsprozess ähnelt damit dem des automaten-basierten Model Checkings. Dort werden sowohl das zu prüfende System als auch die zu prüfende Eigenschaft in einen Büchi-Automaten (dieser arbeitet auf unendlichen Wörtern und akzeptiert, wenn ein Endzustand unendlich oft durchlaufen wird) überführt. Um zu prüfen, ob das System der definierten Eigenschaft genügt, wird anschließend der Büchi-Automat des Systems mit dem Komplement des Büchi-Automaten der Spezifikation geschnitten und ermittelt, ob der Schnittautomat nur die leere Sprache akzeptiert. Ist dies der Fall, so ist die Spezifikation erfüllt [CGP01].

4.3.2. Ansätze zur Verifikation mittels Model Checking

In einigen Arbeiten wurde Model Checking bereits zur Überprüfung von BPMN-Modellen eingesetzt. Hier finden sich zwei Ansätze, um ein BPMN-Modell auf Promela, die Eingabesprache von Spin, umzuformen: Das Mapping mittels mehrerer Prozesse und Channels zur Kontrollflussbehandlung und der Zwischenschritt über ein Petrinetz, welches in einem Promela-Programm schrittweise simuliert wird.

Mapping der BPMN auf Prozesse/Channels

In [VF07] wird die Verifikation von Web Services und Geschäftsprozessen mit Spin behandelt. Dazu werden die in [RAAM06] präsentierten Workflow-Pattern-Spezifikationen (Sequenz, Auswahl, Parallelismus und Synchronisierung) genutzt, die sich auf den Kontrollfluss beschränken. Für jedes dieser Workflow-Patterns wird eine entsprechende Promela-Übersetzung erstellt. Die wichtigsten (Sequenz, Paralleler und Exklusiver Split, Synchronisierung und einfacher Merge) werden vorgestellt.

```
active proctype XORGateway() {
  end:
    rendezvous_channels[1] ? _;

    if
    :: rendezvous_channels[2] ! 1
    :: rendezvous_channels[3] ! 1
    fi
}
```

Listing 4.2: Umsetzung eines XOR-Gateways

Prozesse, Unterprozesse und Aktivitäten werden dabei als Promela-Prozesse umgesetzt, der Kontrollfluss erfolgt über Channels. Am Beispiel einer Reiseagentur wird die Umsetzung eines BPMN-Modells in Promela, sowie die Verifikation von in LTL formulierter Spezifikationen illustriert.

Listing 4.2 zeigt die Umsetzung eines XOR-Gateways. Der Prozess wird bei Programmbeginn gestartet und wartet auf dem eingehenden Channel (entspricht der eingehenden Kontrollflusskante) auf das Signal, mit der Ausführung zu beginnen. Anschließend wird eine

der beiden nachfolgenden Kanten zufällig gewählt und auf dem entsprechenden Channel signalisiert.

Der in [VF07] vorgestellte Ansatz wird unter anderem in [Mü10] weiter verfolgt. Zusätzlich wird hier ein Algorithmus zur Verifikation der temporalen Bedingungen *Direktnachfolger*, *Direktvorgänger*, *Direktabfolge* und *negierte Direktabfolge* vorgestellt. Dieser verwendet allerdings kein Model Checking, sondern prüft die Struktur des BPMN-Modells durch eigenen Code direkt.

Mapping eines BPMN-Modells auf ein Petrinetz

Der in [DDO07, WMM09, Wol10] vorgestellte Ansatz verfolgt dagegen einen anderen Weg der Umsetzung. Hier wird das BPMN-Modell nicht direkt in ein Promela-Programm umgewandelt, sondern zunächst auf ein Petrinetz abgebildet, welches dann als Promela-Programm umgesetzt wird.

Jedem BPMN-Element ist dabei ein Petrinetz-Fragment zugeordnet. Das Mapping des gesamten BPMN-Modells erfolgt durch Umsetzung der einzelnen BPMN-Elemente in ihre Petrinetzentsprechungen.

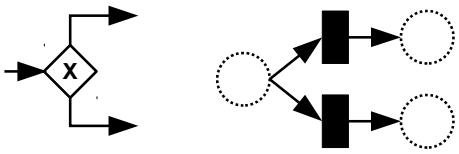


Abbildung 4.2.: XOR-Gateway im Petrinetz, nach [DDO07]

Das Petrinetzfragment für das XOR-Gateway ist in Abbildung 4.2 dargestellt. Das XOR-Gateway wird aktiv, sobald ein Token auf dem eingehenden Platz eintrifft. Nichtdeterministisch wird dann eine der beiden Transitionen getätigt und damit einer der nachfolgenden Zweige gewählt.

4.3.3. Visuelle Notationen für LTL-Formeln

Da die Formulierung vor allem komplexerer LTL-Formeln für unerfahrene Anwender schwierig ist, existieren Ansätze zur graphischen Notation. In [BDSV05] werden zwei mögliche Notationen untersucht. Neben der Verwendung der BPMN wird auch eine eigene Notation vorgeschlagen, die im Gegensatz zur BPMN die volle Ausdrucksmächtigkeit der LTL abbilden kann.

Abbildung 4.1 zeigt die graphischen Darstellungen einiger Operatoren.

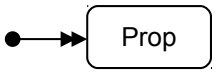
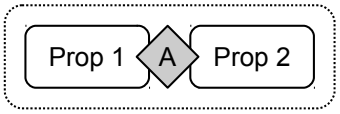
Operator	Beispielformel	Repräsentation
Finally	*Finally* Prop	
Until	Prop1 *Until* Prop2	
Klammerung	(Prop1 *And* Prop2)	

Tabelle 4.1.: Auszug Operatorendarstellungen LTL (nach [BDSV05])

4.4. Ansätze zur Verifikation des Datenflusses

Bestehende Ansätze [AWW09, KLRM+10], die sich mit den Daten innerhalb eines Prozesses beschäftigten, behandeln vor allem die Erweiterung von Kontrollflussregeln um zusätzliche Datenaspekte. [AWW09] erweitert die BPMN-Q [AS] dabei um zusätzliche Elemente.

Die BPMN-Q ist eine visuelle Notation, die die BPMN erweitert, um Suchanfragen auf Prozessmodellen beschreiben zu können. Abbildung 4.3 zeigt eine einfache Anfrage, bei der im BPMN-Modell nach einem Untergraphen gesucht wird, in die Durchführung von *Task1* letztendlich stets zur Durchführung von *Task2* führt.

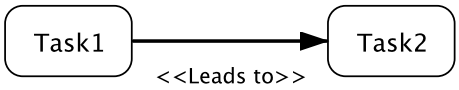


Abbildung 4.3.: BPMN-Q Beispielabfrage [AWW09]

Um bei diesen Regeln zusätzlich noch Daten berücksichtigen zu können, wird die Notation erweitert. Eine entsprechende Regel findet sich in Abbildung 4.4. Hier muss eine negativ ausgefallene Kreditprüfung in jedem Fall zur Ablehnung führen.

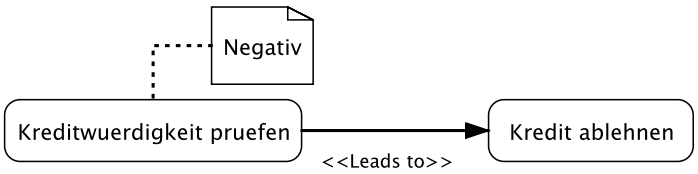


Abbildung 4.4.: BPMN-Q Beispielabfrage mit Datenberücksichtigung [AWW09]

Für die erweiterte Syntax wird dann ein Mapping auf die Past Linear Temporal Logic (kurz PLTL) angegeben. Die PLTL erweitert die LTL zusätzlich um Operatoren, die Aussagen über die Vergangenheit ermöglichen. Diese Operatoren sind unter anderem *Previous*, *Once*, *Always Been* und *Since* (diese ähneln den LTL-Operatoren *Next*, *Finally*, *Globally* bzw. *Until* in ihrer

Definition). Nach dem Mapping der Regeln auf PLTL-Formeln lassen sich diese mittels eines Temporal Logic Query Checkers überprüfen.

4.5. Vorarbeiten der Diplomarbeit zur Variabilität

In der Diplomarbeit *Prozessvarianten in unternehmensübergreifenden Servicenetzwerken* [Köt10] wurde die BPMN bereits um zusätzliche Konstrukte zur Formulierung von Variabilität in BPMN-Modellen erweitert.

Dazu wurde die BPMN 2.0 um folgende Konzepte ergänzt [Köt10]:

Variabler Referenzprozess Ein BPMN-Prozess, der Leerstellen für einzusetzende Prozessfragmente enthält. Außerdem können seine Elemente variable Attribute enthalten.

Prozessfragment Ein separat modellierter Teilprozess, der in einen Referenzprozess eingesetzt werden kann.

Variable Attribute Attribute, die mittels einer eigenen Beschreibungssprache definiert werden, die auf Variabilitätsbeschreibungen aufbaut.

Variantenableitung Auswahl der einzusetzenden Prozessfragmente in die Leerstellen im Referenzprozess durch den Nutzer nach dem Baukastenprinzip. Auswahl konkreter Werte für die variablen Attribute. Als Resultat der Ableitung entsteht eine Prozessvariante.

Prozessvariante Ein abgeleiteter Referenzprozess, der keine Variabilität mehr enthält. Er ist ein gültiger BPMN-Prozess.

Für variable Referenzprozesse und Prozessfragmente wurde dazu jeweils eine eigene Erweiterung des BPMN 2.0-Stencilsets erstellt. Mit dem Diagrammtyp Fragments können Prozessfragmente erstellt werden, zur Definition von Referenzprozessen steht der Diagrammtyp Variability zur Verfügung.

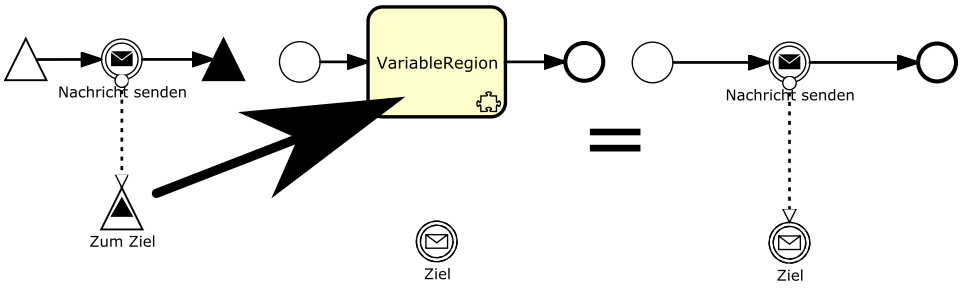


Abbildung 4.5.: Einsetzung eines Fragments mit einem Fragment-Link [Köt10]

Abbildung 4.5 zeigt den Prozess der Prozessvariantenbildung. Ein Prozessfragment wird in einen Referenzprozess eingesetzt. Prozessfragmente besitzen genau einen Fragment-Start (weißes Dreieck) und ein Fragment-Ende (schwarzes Dreieck). Diese werden bei der Prozessvariantenbildung dazu genutzt, das Prozessfragment in den Kontrollfluss des

umgebenden Referenzprozesses einzubinden. Fragment-Links (ineinanderliegende Dreiecke) dienen der Definition von Nachrichtenfluss über die Fragment-/Referenzprozessgrenze hinaus.

Variable Attribute können bei der Prozessvariantenbildung dabei entweder aus einer vordefinierten Liste ausgewählt oder mit Freitext versehen werden, wobei bei Freitext hier zusätzliche Constraints, wie zum Beispiel Wertebereichsbeschränkungen, angegeben werden können.

Mit sogenannten *Dependencies* kann außerdem festgelegt werden, in welcher Reihenfolge die Ableitung vorgenommen werden muss. *Enabling Conditions* schränken die Alternativen für einen Variabilitätspunkt ein und werden in XPath formuliert.

5. Konzept

Unter Berücksichtigung der im zweiten Kapitel vorgestellten Anforderungen wird in diesem Kapitel nun ein Konzept erarbeitet. Dieses baut auf den im vorigen Kapitel erörterten Ansätzen auf.

Hierzu werden der Compliance Scope sowie der Regelbaum zur Beschreibung von Compliance-Regeln inklusive der unterstützten Operatoren definiert. Anschließend werden die LTL-Diagramme zur Definition von temporalen Anforderungen beschrieben, sowie das Arbeiten mit Compliance Scopes.

5.1. Der Compliance Scope

Ein Compliance Scope definiert eine Umgebung in einem BPMN-Diagramm, in der zuvor festgelegte Regeln eingehalten werden müssen. Dies kann zum Beispiel die Anforderung sein, dass in jedem Fall ein Task mit Namen *Kreditwürdigkeit prüfen* ausgeführt wird. In dieser Umgebung sind BPMN-Elemente enthalten.

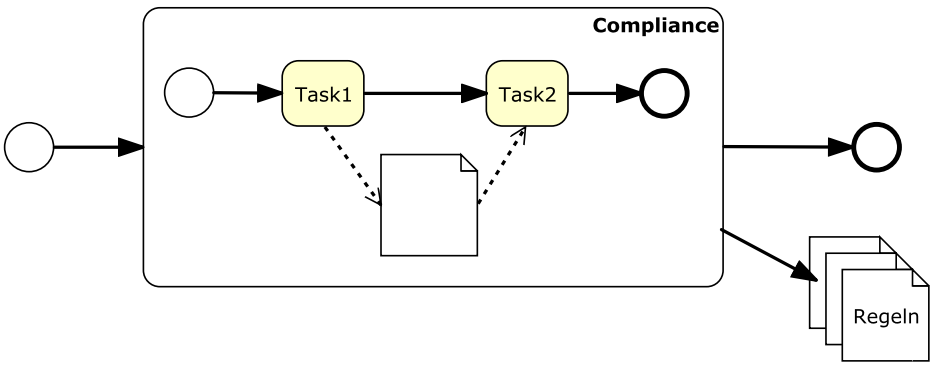


Abbildung 5.1.: Definition Compliance Scope

Abbildung 5.1 zeigt einen Compliance Scope in einem mit BPMN modellierten Prozess. Dieser beinhaltet zwei Tasks, ein Start- und ein Endereignis sowie ein Datenobjekt. Dem vom Compliance Scope umschlossenen Teil des BPMN-Diagramms können nun Regeln zugewiesen werden, zum Beispiel, dass *Task2* nach *Task1* ausgeführt werden muss. Tauscht der Prozessdesigner später die Reihenfolge der beiden Tasks im Compliance Scope, so ist die Regel verletzt.

Aufbauend auf der Definition des Compliance Scopes in [SWLS10] wird die im Rahmen dieser Arbeit verwendete Definition entwickelt, wobei zwei unterschiedliche Ansätze verglichen werden.

Compliance Scopes als loser Verbund von frei gewählten Tasks

In [SWLS10] wird ein Compliance Scope als Hyperkante in einem Hypergraphen definiert, das heißt es werden Verknüpfungen zu jedem einzelnen BPMN-Element abgespeichert, das zum Compliance Scope gehören soll.

Dieser Ansatz ist sehr flexibel, da hier sehr feingranular festgelegt werden kann, welche Elemente zum Compliance Scope gehören sollen und welche nicht. Außerdem können BPMN-Elemente zu mehreren Compliance Scopes gehören. Bei der direkten Umsetzung dieses Ansatzes ist aber mit Problemen in der Handhabbarkeit durch den Anwender zu rechnen, denn für den Prozessdesigner muss die Darstellung der Compliance Scopes und ihrer zugehöriger Elemente nachvollziehbar sein.

Da die zu einem Compliance Scope gehörenden BPMN-Elemente über das komplette Diagramm verteilt sein können, muss für jedes Element deutlich gemacht werden, zu welchem Compliance Scope es gehört. Dies kann durch die Angabe des Namens des Compliance Scopes oder die Verwendung eines eigenen Icons oder einer eigenen Farbe pro Compliance Scope erfolgen. In allen Fällen wird die Darstellung durch die zusätzlichen Compliance Scope-Kennzeichnungen aber deutlich komplexer und damit für den Prozessdesigner schwieriger zu überblicken.

Compliance Scopes als konvexe Hülle

Im Rahmen dieser Arbeit wird deshalb ein modifizierter Ansatz gewählt. Hier bildet der Compliance Scope einen Teil der Hierarchie des BPMN-Modells, das heißt der Compliance Scope beinhaltet, wie zum Beispiel auch ein Subprozess, weitere BPMN-Elemente.

Dieser Ansatz ist zwar weniger flexibel als der zuerst vorgestellte Ansatz, ist aber besser für den Prozessdesigner zu verstehen, da alle zum Compliance Scope gehörenden Elemente an einer Stelle des BPMN-Modells konzentriert sind.

Zusätzlich können zu einem Compliance Scope nicht nur Tasks, sondern beliebige BPMN-Elemente wie Datenobjekte und Gateways gehören. Damit wird die Modellierung von Datenflussregeln ermöglicht, die zum Beispiel für die zu einem Compliance Scope gehörenden Datenobjekte festlegen, welche Daten gelesen und geschrieben werden dürfen.

Beschreibung der Compliance-Regeln

Wie in [SWLS10] auch werden die Compliance-Regeln eines Compliance Scopes als Element der Potenzmenge aller möglichen Regeln definiert. Diese Menge der darstellbaren Compliance-Regeln wird aber weiter eingeschränkt. Dazu werden Compliance-Regeln als Baum aus Operatoren, dem Regelbaum, definiert. Dieser verknüpft einzelne Compliance-Regeln in unterschiedlichen Beschreibungsmöglichkeiten wie der LTL mittels logischer Operatoren zu einer einzigen Compliance-Regel.

5.1.1. Finale Definition

Zusammenfassend werden Compliance Scopes nun wie folgt definiert: Sei $X = x_1, x_2, ..., x_n$ eine Menge von BPMN 2.0-Elementen und $H = E_1, E_2, ..., E_m$ eine Familie von Untermengen, den Hyperkanten über X, sodass gilt:

$$E_i \neq \emptyset \text{ für } (i = 1, 2, ...m) \text{ und } \bigcup_{i=1}^{m} E_i = X$$

Sei weiterhin C die (endliche) Menge aller Compliance-Regeln und $R \subseteq 2^C$ die Menge aller als Regelbaum dargestellten Regelkombinationen aus 2^C, dann ist ein Compliance Scope ein Element aus dem kartesischen Produkt $H \times R$. Das heißt, ein Compliance Scope verknüpft eine Menge von BPMN-Elementen mit einer als Regelbaum dargestellten Menge von Regeln. Zusätzlich müssen folgende Bedingungen gelten:

Zusammenhängend Sind zwei Tasks in dem selben Compliance Scope enthalten und existiert ein Kontrollflusspfad zwischen beiden Tasks, dann sind auch alle Tasks auf dem Kontrollflusspfad zwischen den beiden Tasks im Compliance Scope enthalten.

Kontrollfluss Analog zum Subprozess beginnt die Ausführung in einem Compliance Scope mit einem Startereignis und endet mit einem Endereignis. Ein Compliance Scope enthält damit mindestens ein Start- und mindestens ein Endereignis.

Compliance-Regel Ein Compliance Scope kann mit Compliance-Regeln annotiert sein. Diese definieren unter anderem Kontroll- und Datenflussregeln, die in dem Compliance Scope eingehalten müssen. Die Compliance-Regeln werden mittels logischer Verknüpfungen als Operatorenbaum realisiert.

5.1.2. Vererbung von Regeln

Werden Compliance Scopes ineinander geschachtelt, so muss festgelegt werden, wie im inneren Compliance Scope mit den Regeln des äußeren Compliance Scopes verfahren wird. In der hier erarbeiteten Lösungen werden Compliance-Regeln nicht an enthaltene Compliance Scopes weiter vererbt (das heißt zum Regelsatz des inneren Compliance Scopes hinzugefügt), da bei der Verifikation der Compliance-Regeln eines Compliance Scopes alle inneren Compliance Scopes als normale Subprozesse betrachtet werden.

Als Beispiel sei das in Abbildung 5.2 dargestellte Prozessmodell gegeben. Hier wird in einem Compliance Scope *Unternehmen* ein weiterer Compliance Scope *Abteilung* definiert. Bei der Verifikation jedes der beiden Compliance Scopes wird nur der Regelsatz des jeweiligen Compliance Scopes überprüft. Enthält der Compliance Scope *Unternehmen*, die Regel, dass irgendwann im Prozess *Task1* ausgeführt werden muss, so ist diese im äußeren Compliance Scope erfüllt. Diese Regel wird aber nicht dem Regelsatz des Compliance Scopes *Abteilung* hinzugefügt, sodass also kein weiterer *Task1* im Compliance Scope *Abteilung* enthalten sein muss, damit der innere Compliance Scope erfüllt ist.

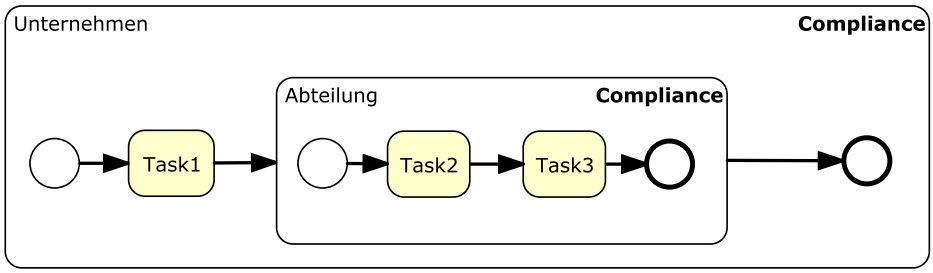


Abbildung 5.2.: Beispiel zur Vererbung von Regeln zwischen Compliance Scopes

Damit unterscheidet sich die hier gewählte Lösung zu [SALS10]. Dort werden sogenannte *Refinement Layers* definiert, über die schrittweise ein Prozesstemplate präzisiert wird. Beim Einsetzen eines Prozesstemplates in ein Prozesstemplate werden dabei bestehende Compliance-Regeln mit denen des neu eingesetzten Prozesstemplates kombiniert. So können Regeln, die bereits im umgebenden Prozesstemplate erfüllt sind und im eingefügten Prozesstemplate nicht mehr verletzt werden können, bei der Betrachtung des inneren Templates ignoriert werden. Damit wird es zum einen ermöglicht, den äußeren Compliance Scope während der Bearbeitung des inneren auszublenden und somit die Konzentration des Nutzers auf diesen zu richten, zum anderen müssen bereits erfüllte Regeln nicht erneut geprüft werden, was den Compliance Check beschleunigt.

Da aber in dem hier erarbeiteten Konzept stets das komplette Prozessmodell weiter bearbeitet wird, werden Compliance-Regeln nicht an bei der Prozessvariantenbildung eingefügte Compliance Scopes weiter gereicht.

5.2. Der Regelbaum

Der Regelsatz eines jeden Compliance Scopes ist als ein Baum aus Operatoren repräsentiert. Operatoren sind dabei logische Operatoren zur Verknüpfung und Operatoren, die einzelne Anforderungen beschreiben. In dieser Arbeit sind dies LTL- und DataTransfer-Operatoren, in späteren Erweiterungen sind auch weitere Operatoren möglich, die Anforderungen zum Beispiel auf Ontologien basierend beschreiben. Blätter dieses Regelbaums sind LTL- oder DataTransfer-Operatoren, innere Knoten logische Operatoren.

Beide Operatorenarten liefern als Ergebnis entweder wahr oder falsch. Das Ergebnis logischer Operatoren wird dabei aus den als Operanden verwendeten Operatoren ermittelt, das Ergebnis der LTL- und DataTransfer-Operatoren über dem Compliance Scope anhand der Definition des jeweiligen Operators.

Abbildung 5.3 zeigt einen Regelbaum, der zwei LTL- und eine DataTransfer-Regel kombiniert.

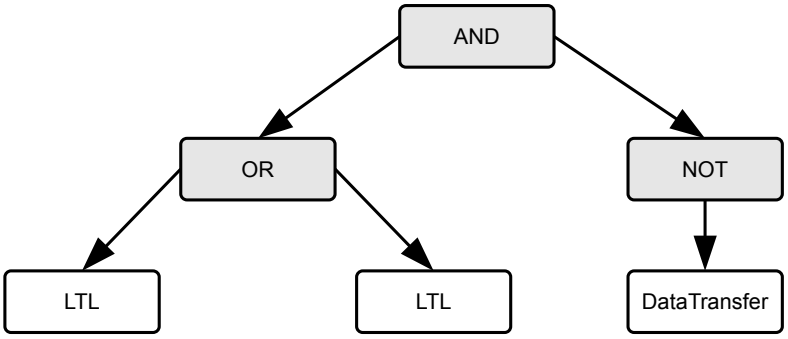


Abbildung 5.3.: Regelbaum

5.2.1. Logische Operatoren

Unterstützt werden die logischen Operatoren AND, NOT und OR. Da bereits einer der beiden binären Operatoren und NOT eine Junktorenbasis bilden, lassen sich weitere Operatoren wie Implikation, Äquivalenz und XOR als Kombination dieser darstellen.

Im Gegensatz zur klassischen Definition dürfen die Operatoren AND und OR auch nur einen oder auch mehr als zwei Operanden besitzen. OR und AND mit jeweils genau einem Operanden sind genau dann erfüllt, wenn ihr Operand erfüllt ist. Bei mehr als zwei Operanden wird die Interpretation von OR und AND analog fortgesetzt, OR ist wahr, wenn mindestens ein Operand erfüllt ist, bei AND müssen alle Operanden erfüllt sein.

5.2.2. LTL-Operator

Der LTL-Operator dient der Kontrollflussanalyse. Die Regel eines LTL-Operators ist dabei durch eine LTL-Formel beschrieben (vgl. Abschnitt 3.2.2). In der bestehenden Implementierung erfolgt die Analyse alleinig auf den Namen der Tasks. Dies lässt sich aber erweitern um weitere Prüfbedingungen, wie zum Beispiel den Typ eines Tasks (WebService, manuell...).

Fortlaufendes Beispiel

Abbildung 5.4 zeigt ein Prozessmodell, welches im weiteren Verlauf dieser Ausarbeitung zur Beschreibung der Verarbeitungsschritte beim Auswerten einer in LTL formulierten Compliance-Regel verwendet wird.

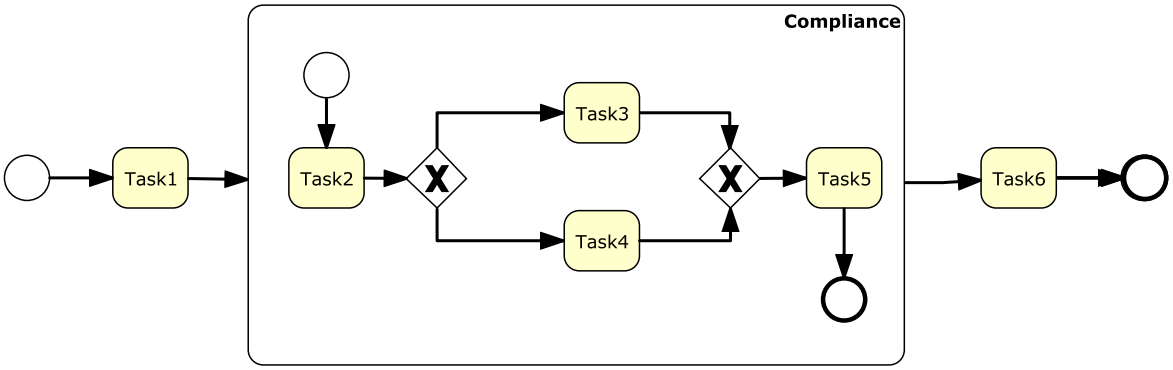


Abbildung 5.4.: Beispielgraph LTL-Operator

Hier werden nacheinander die Tasks *Task1* und *Task2* ausgeführt. Abhängig von einer hier nicht näher spezifizierten Bedingung wird anschließend entweder *Task3* oder *Task4* ausgeführt, bevor der Prozess nach Durchführung von *Task5* und *Task6* beendet wird. Um *Task2* und *Task5* wurde ein Compliance Scope definiert, dem folgende Compliance-Regel zugewiesen wurde:

```
Finally Task2 ∧ Finally Task3
```

`Finally Task2` und `Finally Task3` wurden dabei jeweils als einzelne LTL-Formeln modelliert, die mit einem logischen AND-Operator verknüpft werden. Der Regelbaum des Compliance Scope besteht damit aus drei Operatoren.

Der Next-Operator in LTL-Formeln

Die LTL unterstützt wie in den Grundlagen (3.2.2) vorgestellt auch einen `Next`-Operator, der den Zustand unmittelbar nach dem aktuellen beschreibt. Dieser wird allerdings von der erarbeiteten Lösung nicht unterstützt.

Dies hat zwei Gründe. Zum einen ist der Einsatz des Next-Operators fehleranfällig. Als Beispiel sei die LTL-Formel `Task1 -> Next Task2` und die in Abbildung 5.5 gezeigte Prozessvariante gegeben. Auf den ersten Blick erfüllt diese die angegebene LTL-Formel. Allerdings wird diese verletzt, wenn zuerst *Task1* ausgeführt wird und während dieser ausgeführt wird, parallel mit *Task3* begonnen wird. Das ganze System befindet sich damit durch den Start von *Task3* in einem neuen Zustand, in dem aber noch *Task1* und noch nicht *Task2* ausgeführt wird.

Zum anderen wird der Next-Operator in der Binärdistribution von Spin nicht unterstützt, da dieser nicht kompatibel mit der verwendeten Partial Order Reduction ist [ltl].

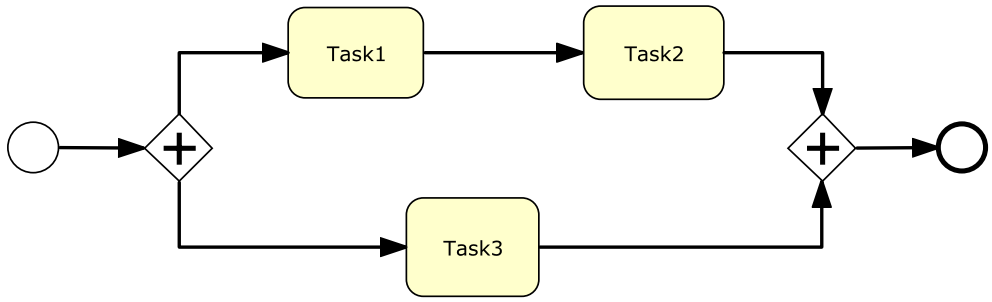


Abbildung 5.5.: Illustration Next-Operator

5.2.3. DataTransfer-Operator

Der DataTransfer-Operator dient der Analyse des Datenflusses in Datenobjekte und aus Datenobjekten heraus. Ein wichtiges Anwendungsszenario dieses Regeltyps ist das Arbeiten mit sensiblen Daten. So muss zum Beispiel sichergestellt werden, dass geheime Kreditkartendaten nicht an externe Dienstleister weitergegeben werden.

In Abbildung 5.6 ist ein Beispielprozess gegeben. Aus *Task1* werden dabei Daten über eine DataAssociation in das Datenobject *DataObject1* übertragen. *DataObject1* liefert anschließend Eingabedaten für *Task2*. Welche Daten über einer Datenflusskante übertragen werden, wird dabei in einer Assignment-Regel hinterlegt [Obj11].

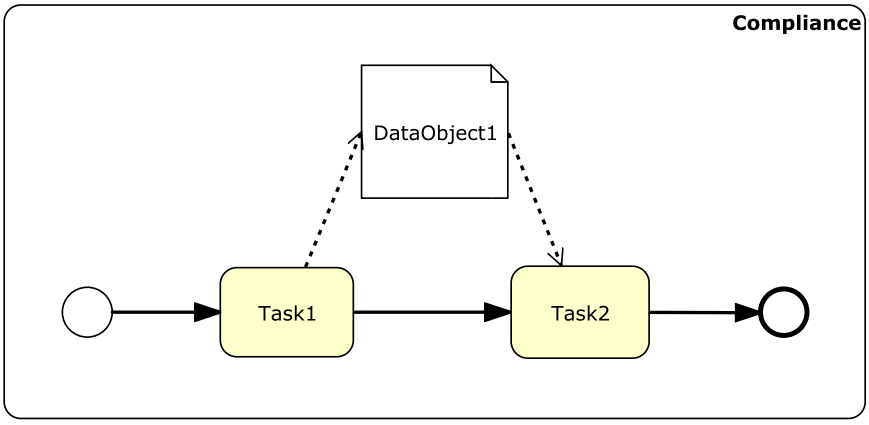


Abbildung 5.6.: Beispielgraph DataTransfer-Operator

Die grundlegenden Eigenschaften des DataTransfer-Operators enthalten folgende Informationen:

Name Datenobjekt Der Name des Datenobjektes, auf den sich die Regel bezieht. Ein leerer Name ist dabei nicht zugelassen.

Pfad Beschreibt die Felder im Datenobjekt, die betrachtet werden.

Sprache Pfad Gibt die Formulierungssprache des Pfades an. Die Spezifikation der BPMN 2.0 [Obj11] lässt hier prinzipiell beliebige Sprachen zu, in der Diplomarbeit wird allerdings nur XPath [xpa] unterstützt.

Einschränkung, wann die Regel geprüft wird

Wann eine DataTransfer-Regel geprüft wird, kann durch mehrere Bedingungen festgelegt werden. Zum einen kann die Richtung des Datenflusses angegeben werden, das heißt ob die Daten in das Datenobjekt geschrieben oder von diesem gelesen werden.

Eine zweite Einschränkungsmöglichkeit bieten die potentiellen Partner eines Datenaustauschs. Hier kann eine Liste der Partner angegeben werden. Die Überprüfung der Regel findet nur dann statt, wenn ein Datenaustausch mit einem der zuvor definierten Partner stattfindet.

Die dritte und letzte Bedingung ist, ob die Regel nur beim Überqueren der Grenze des Compliance Scopes geprüft werden soll. Hierbei muss das Datenobjekt innerhalb des Compliance Scopes liegen. Das stellt sicher, dass das Datenobjekt betreffende Datenflussregeln nahe am Datenobjekt definiert werden.

Regeltyp

Über den Regeltyp lässt sich festlegen, in welcher Beziehung die durch den XPath-Ausdruck in der Regel selektierten Knoten und die durch die Assignment-Regeldefinition bestimmte Knotenmenge stehen sollen. Insgesamt werden vier verschiedene Regeltypen unterstützt.

Regeltyp 1: Nicht diese Die durch den XPath-Ausdruck in der Compliance-Regel beschriebenen Knoten dürfen nicht gelesen oder geschrieben werden. Damit kann zum Beispiel sichergestellt werden, dass sensible Informationen nicht übertragen werden.

Regeltyp 2: Nur diese Nur die durch den XPath-Ausdruck in der Regel beschriebenen Knoten dürfen gelesen und geschrieben werden. Hiermit können Informationen ähnlich einer Whitelist explizit zur Datenübertragung freigegeben werden.

Regeltyp 3: Genau diese Exakt die durch den XPath-Ausdruck in der Regel beschriebenen Knoten müssen gelesen oder geschrieben werden.

Regeltyp 4: Mindestens diese Mindestens die durch den XPath-Ausdruck der Regel beschriebenen Knoten müssen gesetzt werden. Damit lässt sich sicherstellen, dass anschließend die gewünschten Informationen auch im Zielobjekt enthalten sind.

Ausgewertet werden diese Regeln durch Mengenoperationen auf den Knotenmengen, die durch den XPath-Ausdruck der Regel und durch den XPath-Ausdruck der Assignments der DataAssociations beschrieben werden.

Sei A die Knotenmenge, die durch die Assignment-Regel in der DataAssociation beschrieben wird und B die Menge der Knoten, die durch den XPath-Ausdruck im DataTransfer-Operator beschrieben wird, dann werden die einzelnen Regeltypen wie folgt umgesetzt:

Regeltyp 1: Nicht diese $A \cap B \stackrel{!}{=} \emptyset$

Regeltyp 2: Nur diese $A \setminus B \stackrel{!}{=} \emptyset$

Regeltyp 3: Genau diese $A \triangle B = (A \cup B) \setminus (A \cap B) \stackrel{!}{=} \emptyset$

Regeltyp 4: Mindestens diese $B \cap A \stackrel{!}{=} \emptyset$

Der Operator zur Vereinigung ist bereits in XPath 1.0 enthalten, die Operatoren für Schnitt und Mengendifferenz sind in XPath 2.0 neu hinzugekommen [exc, xpa].

Beispiel Regeltyp

Um die einzelnen Regeltypen zu veranschaulichen, sei das in Listing 5.1 gegebene Schemafragment für das Datenobjekt *DataObject1* in Abbildung 5.6 gegeben.

```
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Listing 5.1: Beispielschema eines Datenobjektes

Für den in der Assignment-Regel der eingehenden DataAssociation enthaltenen XPath-Ausdruck werden nun folgende drei Alternativen betrachtet:

Alternative 1

```
xs:element/xs:complexType/xs:sequence/*
```

Alternative 2

```
xs:element/xs:complexType/xs:sequence/*[2]
```

Alternative 3

```
xs:element/xs:complexType/xs:sequence/*[3]
```

In der Compliance-Regel des umgebenden Compliance Scope sei nun folgender XPath-Ausdruck angegeben:

```
xs:element/xs:complexType/xs:sequence/xs:element[@name="from"]
```

Alternative 1 wählt damit alle Unterknoten des *sequence*-Knoten aus, Alternative 2 nur den zweiten und Alternative 3 den dritten. Der XPath-Ausdruck in der Compliance-Regel wählt im gegebenen Schema nur den zweiten Unterknoten des *sequence*-Knoten aus.

Eine Auswertung der einzelnen Regeltypen mit den einzelnen Alternativen ist in Tabelle 5.1 angegeben.

Regeltyp	Alternative 1	Alternative 2	Alternative 3
1 (Nicht diese)	nicht erfüllt	nicht erfüllt	erfüllt
2 (Nur diese)	nicht erfüllt	erfüllt	nicht erfüllt
3 (Genau diese)	nicht erfüllt	erfüllt	nicht erfüllt
4 (Mindestens diese)	erfüllt	erfüllt	nicht erfüllt

Tabelle 5.1.: Beispielauswertung Regeltypen

5.3. LTL-Diagramme

Für die Definition von LTL-Regeln für den LTL-Operator wird ein eigener Diagrammtyp in Oryx geschaffen. Durch die graphische Modellierung erhält der Benutzer einen einfacheren Zugang. Die Übersicht, besonders bei LTL-Formeln mit vielen Klammern, wird durch die Verwendung von Containern erleichtert. Durch die separate Definition in eigenständigen Diagrammen können LTL-Diagramme außerdem in mehreren Prozesstemplates und Prozessen wiederverwendet werden.

Eine Übersicht der graphischen Primitive für die LTL-Operatoren findet sich in Tabelle 5.2. Hierbei werden unäre Operatoren als Container umgesetzt, binäre Operatoren als Kanten. Mehrere Operatorenketten in einem Container werden per AND verknüpft.

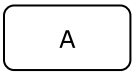
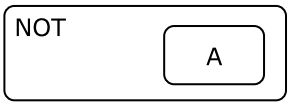
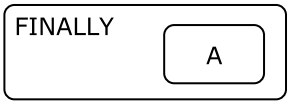
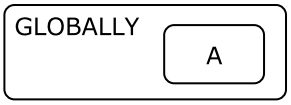
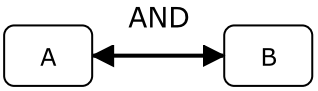
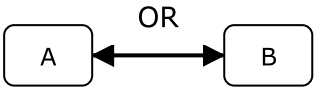
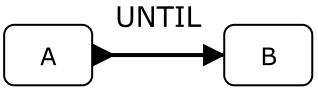
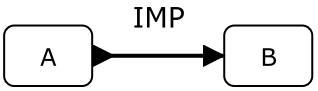
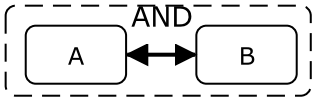
Operator	Promela-Notation	Beispielformel	Repräsentation
Property	-	A	
Not	!	$!A$	
Finally	<>	$<>A$	
Globally	[]	$[]A$	
And	&&	A && B	
Or	||	A || B	
Until	U	A U B	
Implikation	->	A -> B	
Klammerung	( )	(A && B)	

Tabelle 5.2.: Übersicht Operatorendarstellungen LTL (angelehnt an [BDSV05])

5.4. Das Ergebnis eines Compliance Checks

Das Ergebnis eines Compliance Checks ist hierarchisch aufgebaut und setzt sich aus den Ergebnissen der einzelnen Compliance Checks für die einzelnen Compliance Scopes zusammen.

Gesamtergebnis

Das Gesamtergebnis ist eine Zusammenfassung der Einzelergebnisse. Der Aufbau ist wie folgt:

Message Zusammenfassung des Gesamtergebnisses in einem Satz.

Log Ausführliches Log, das beschreibt, welche Compliance Scopes gefunden und mit welchem Ergebnis geprüft wurden.

Einzelergebnisse Liste der Ergebnisse der einzelnen Compliance Scopes.

Ergebnis für einen einzelnen Compliance Scope

Jedes dieser Einzelergebnisse ist wiederum wie folgt aufgebaut:

ID des Compliance Scopes Diese dient der eindeutigen Identifikation des Compliance Scopes zur Zuordnung des Ergebnisses zum zugehörigen Compliance Scope.

Message Kurze Zusammenfassung des Ergebnisses. Bei einem Nichterfüllen einer LTL-Regel wird hier die Herleitung des Gegenbeispiels angegeben.

Log Hier findet sich in aller Regel ein hierarchisches Log der Auswertung der mit dem Compliance Scope verknüpften Regel. Damit lässt sich Nachvollziehen, wie die einzelnen Operatoren ausgewertet wurden.

Compliance Check Resultat Ergebnis der Auswertung wie im nächsten Absatz beschrieben.

Mögliche Ausgänge eines Compliance Checks eines einzelnen Compliance Scopes

Die möglichen Ausgänge eines Compliance Checks sind die folgenden:

Fail Der Compliance Check konnte nicht ausgeführt werden. Dies tritt zum Beispiel ein, wenn der Model Checker Spin nicht gefunden wurde.

Invalid Der Compliance Check wurde erfolgreich durchgeführt, die definierte Regel wird aber verletzt.

Valid Der Compliance Check wurde erfolgreich durchgeführt und die definierte Regel wird nicht verletzt.

NoRulesDefined Es wurde keine Regel definiert. Dieser Fall wird getrennt behandelt, da hier in aller Regel das Prozesstemplate nicht vollendet wurde.

Ignored Der Compliance Scope wurde nicht geprüft. Dies kann eintreten, wenn der Benutzer in der Oryx-Erweiterung nur einen Teil der Compliance Scopes prüfen lassen will.

Alle Ergebnisse schließen die jeweils anderen Ergebnisse aus, das heißt das Resultat eines Compliance Checks für einen Compliance Scope kann niemals aus mehreren der oben genannten Ergebnisse bestehen.

5.5. Arbeiten mit Compliance Scopes

In diesem Abschnitt wird das Arbeiten mit Compliance Scopes genauer erläutert. Dabei wird verdeutlicht, wie die Kombination von Variabilität und Compliance Scopes in der Praxis erfolgt.

Graphische Modellierung von LTL-Formeln

In einem ersten Schritt werden Anforderungen an Prozesse als LTL-Formeln formuliert und anschließend als graphische Diagramme in Oryx erstellt. Mit der Zeit entsteht dabei eine ganze Sammlung von als LTL-Formeln definierten Regeln, die entsprechend weiterverwendet werden können. Ändert sich eine Regel, so muss lediglich das entsprechende LTL-Diagramm angepasst werden und wird ab dann bei allen folgenden Prozessvariantenbildungen berücksichtigt.

Später wird dieser Schritt auch parallel zum nachfolgenden erfolgen, wenn sich die Anforderungen an ein Prozessmodell ändern.

Definition von Prozesstemplates mit Compliance Scopes

Vorlagen für Prozesse werden in Oryx mit dem BPMN 2.0-Variablity-Stencilset erstellt und im Oryx-Repository abgelegt. Dies erfolgt wie in [Köt10] beschrieben. Zusätzlich können Compliance Scopes erstellt werden.

Die Compliance Scopes werden mit einem entsprechenden Editor, dem *Compliance Wizard*, mit Regeln versehen. LTL-Operatoren verknüpfen zuvor erstellte LTL-Diagramme. Durch die separate Speicherung der LTL-Diagramme können diese auch noch später angepasst werden.

Erstellen von Prozessvarianten unter Beachtung der Compliance-Regeln

Nach dem Laden eines Prozesstemplates wird dieses Schritt für Schritt abgeleitet [Köt10]. Dabei werden Variability Points durch die gewünschte Alternative ersetzt und variable Attribute mit konkreten Werten belegt.

Von Zeit zu Zeit kann der Prozessdesigner per Knopfdruck einen Compliance Check durchführen lassen. Noch nicht ersetzte Variability Regions werden dabei wie (namenlose) Tasks

behandelt und verhindern damit nicht die Durchführung von Compliance Checks solange das Prozesstemplate noch nicht fertig abgeleitet wurde.

5.6. Wahl des Model Checkers

Bei der Wahl des Model Checkers fiel die Wahl auf Spin [Spia], da dieser weit verbreitet ist und die Anforderungen abdeckt. Von Spin stehen hier im wesentlichen zwei Implementierungen zur Verfügung. Die originale, in C geschriebene Variante und eine im Rahmen einer Masterarbeit durchgeführte Neuimplementierung in Java: SpinJa [spib].

Die ursprüngliche Implementierung von Spin in C wird bereits seit 1980 entwickelt. Entsprechend oft wurde die Implementierung bereits genutzt, sodass viele Fehler beseitigt wurden. Außerdem wird diese Version beständig gepflegt und weiterentwickelt. Nachteilig ist hier die Tatsache, dass Spin kein Java-Interface bietet und über die Konsole aufgerufen werden muss. Die Ausgabe von Spin muss daher umgeleitet und geparst werden.

SpinJa ist eine Neuimplementierung von Spin in Java, bei der besonderes Augenmerk auf Erweiterbarkeit und Objektorientierung gelegt wurde. Die weitere Entwicklung von SpinJa ist ungewiss, die Implementierung weniger fehlererprobt. Auf Grund der Implementierung in Java ist SpinJa in den meisten Anwendungssituationen geringfügig langsamer als die ursprüngliche Implementierung, das stellt aber keinen Hinderungsgrund für den Einsatz dar.

Da der Status von SpinJa ungewiss ist und das Ausführen der ursprünglichen Spin Version im Webcontainer keine Probleme bereitet, wird die original Spin-Implementierung verwendet. Bei der Implementierung wird die Einbindung des Model Checkers entsprechend gekapselt, sodass später auch ein anderer Model Checker verwendet werden kann.

5.7. Mapping des BPMN-Modells auf die Systembeschreibung

Für das Mapping des BPMN-Modells auf Promela, die Eingabesprache für die Modellbeschreibung in Spin, wurden zwei Ansätze genauer untersucht (siehe auch 4.3.2).

In [VF07] werden die Elemente eines BPMN-Modells direkt in Promela mittels eigenständiger Prozesse umgesetzt. Hierbei wird jeder einzelne Task und jedes Gateway als eigenständiger Prozess definiert. Die Realisierung des Kontrollflusses erfolgt über die Kommunikation zwischen Tasks und Gateways mittels Channels. Über die Channels wird dabei das Signal zum Anfangen der Durchführung des jeweiligen Prozesses vermittelt. Um zu protokollieren, dass bestimmte Tasks ausgeführt wurden, werden in den entsprechenden Prozessen globale Variablen gesetzt.

[DDO07, WMM09, Wol10] dagegen formen das BPMN-Modell zunächst in ein Petrinetz um. Die Plätze des 1-sicheren Petrinetzes werden als Array verwaltet. Die Transitionen jeweils als

zwei Makros: zum einen die Startbedingung für die Transition und zum anderen die Veränderung der Platzbelegungen durch die Transition. Die Beschreibung des zu überprüfenden Modells in Promela besteht anschließend nur aus einem einzigen Prozess, der das Petrinetz simuliert.

In der hier erarbeiteten Lösung wird der Ansatz mit Petrinetzen umgesetzt, da die globalen Variablen im ersten Ansatz bei einer wiederholten Ausführung eines Tasks in einer Schleife wieder zurückgesetzt werden müssen. Durch den dafür zusätzlich nötigen Schritt ist die Umsetzung fehleranfällig. Bei Petrinetzen dagegen erfolgt die Protokollierung der durchgeführten Tasks durch Makros, die prüfen, ob die zugehörigen Plätze der Petrinetzfragmente mit einem Token belegt sind. Nach der Ausführung des Petrinetzfragments wird der Token von dem zugehörigen Platz entfernt, womit das Zurücksetzen des Ausgeführtheitsstatus des Tasks automatisch erfolgt.

Ein weiterer Vorteil des zweiten Ansatzes ist, dass das Mapping mittels Petrinetzen nahe an der Spezifikation der BPMN Version 2.0 ist, da hier die Beschreibung der Execution Semantic auch an Petrinetze angelehnt ist [Obj11]. Außerdem ist das Petrinetz-Mapping leichter nachvollziehbar, bei der Fehlersuche in Mappings der BPMN auf Promela, da sich das generierte Petrinetz mittels geeigneter Werkzeuge graphisch darstellen lässt [pnma, gra].

Das Mapping des BPMN-Modells über ein Petrinetz auf Promela wird in den nächsten beiden Unterabschnitten beschrieben.

5.7.1. Mapping des BPMN-Modells auf ein Petrinetz

Abbildung 5.7 zeigt eine Übersicht des Mappings einiger BPMN-Elemente auf Petrinetzelemente. Plätze mit gestrichelter Umrandung werden dabei wiederverwendet, das heißt die direkte Verbindung eines Startereignisses mit einem Endereignis benötigt insgesamt nur drei Plätze, denn der Endplatz des Startereignisses und der Startplatz des Endereignisses werden zusammengefasst.

Das Petrinetz des BPMN-Modells wird durch Zusammensetzen der Petrinetzfragmententsprechungen der einzelnen BPMN-Elemente aufgebaut. Die Startereignisse der obersten Hierarchieebene, das heißt die Startereignisse des Compliance Scopes nicht aber Startereignisse in enthaltenen Subprozessen, werden mit Tokens vorbelegt.

Petrinetz-Entsprechung des laufenden Beispiels

Das entsprechende Petrinetz für das laufende Beispiel aus Abbildung 5.4 findet sich in Abbildung 5.8.

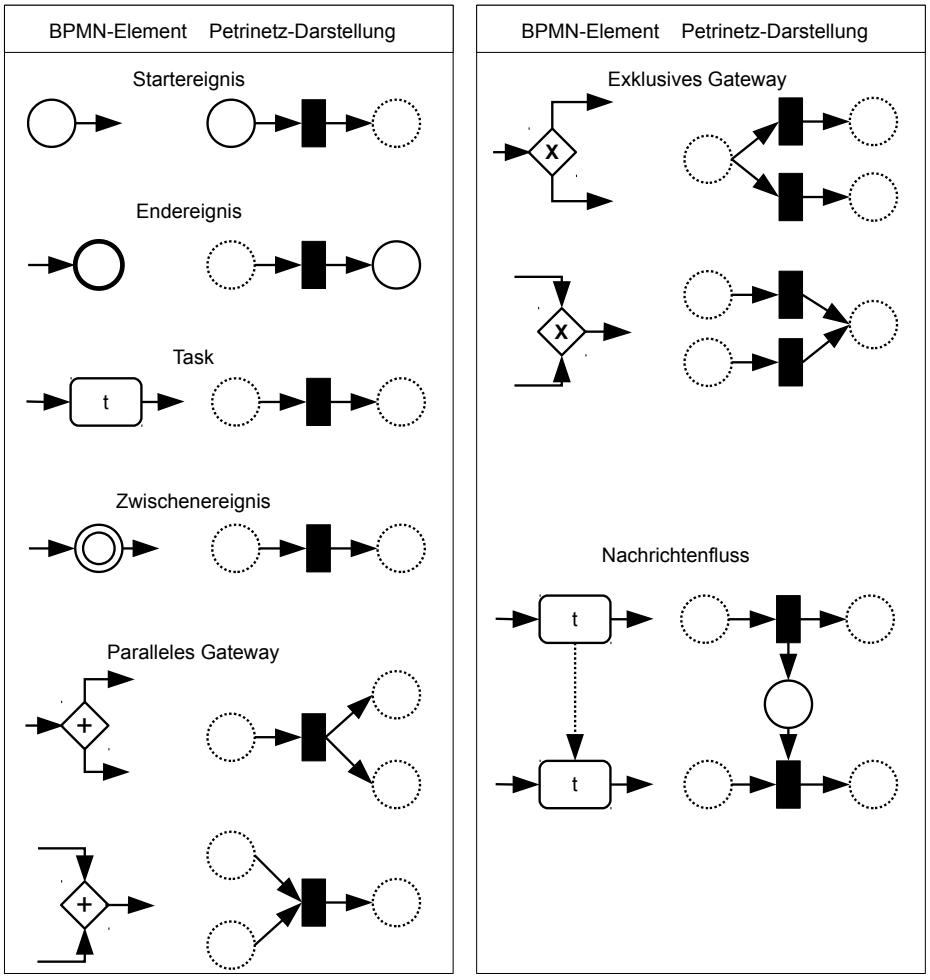


Abbildung 5.7.: Mapping BPMN-Elemente Petrinetz, nach [DDO07]

Minimierung des Petrinetzes und Sonderfälle

Da sich die Größe des zu überprüfenden Petrinetzes direkt auf die Ausführungszeit des Model Checkers auswirkt, wird bei der Generierung des Petrinetzes sparsam mit Stellen und Transitionen umgegangen.

In einigen Situationen müssen aber zusätzliche Plätze und Transitionen eingefügt werden. Dies ist zum Beispiel der Fall bei Gateways mit mehreren eingehenden Kontrollflusskanten. Abbildung 5.9 illustriert dies anhand der Umsetzung eines XOR-Joins. Hier wird für jede eingehende Kante ein separater Platz benötigt. Würden die Plätze 1 und 2 entfernt, wäre bei einer späteren Ausführung nicht nachvollziehbar, ob vor dem Join *Task1* oder *Task2* ausgeführt worden ist. Außerdem wurde zusätzlich Transition t_5 hinzugefügt für die Nach-

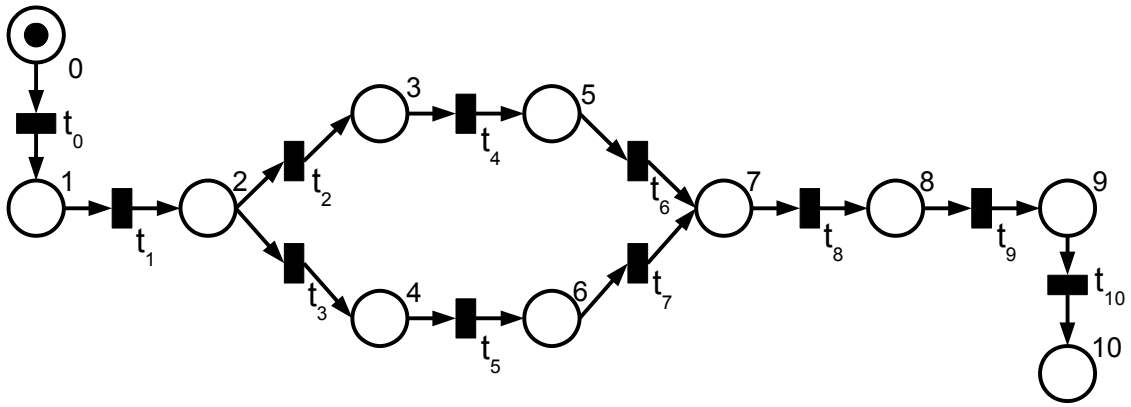


Abbildung 5.8.: Mapping des laufenden Beispiels auf die Petrinetzdarstellung

vollziehbarkeit des Gegenbeispiels. Das Ausführen dieser Transition wird mit 'Gateway geschaltet' protokolliert.

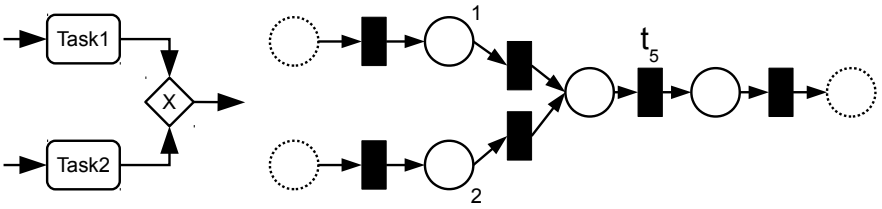


Abbildung 5.9.: Mapping Gateway mit mehreren eingehenden Kanten auf Petrinetz

Zwei Sonderfälle beim Mapping der BPMN auf ein Petrinetz sind in Abbildung 5.10 dargestellt. Zum einen das Mapping von Subprozessen, hier werden die zusätzlichen Transitionen t_1 und t_5 erstellt, für das Eintreten in den beziehungsweise das Verlassen des Subprozesses. Zum anderen wird die Umsetzung eines sich wiederholenden Tasks dargestellt.

5.7.2. Mapping des Petrinetzes auf Promela

Bei der Umwandlung des generierten Petrinetzes in Promela werden die Plätze durch ein Byte-Array repräsentiert, dessen Elemente die Werte Eins und Null annehmen können. Die Konstruktion des Petrinetzes im Abschnitt 5.7.1 stellt dabei sicher, dass das generierte Petrinetz 1-sicher ist, indem Transitionen in der Startbedingung prüfen, ob Plätze, die beim Ausführen der Transition mit Tokens belegt werden, vorher leer sind.

In einem ersten Schritt wird die initiale Arraybelegung hergestellt, das heißt alle als initial markierten Plätze werden auf Eins gesetzt, die zugehörigen Arrayelemente der anderen Plätze auf Null.

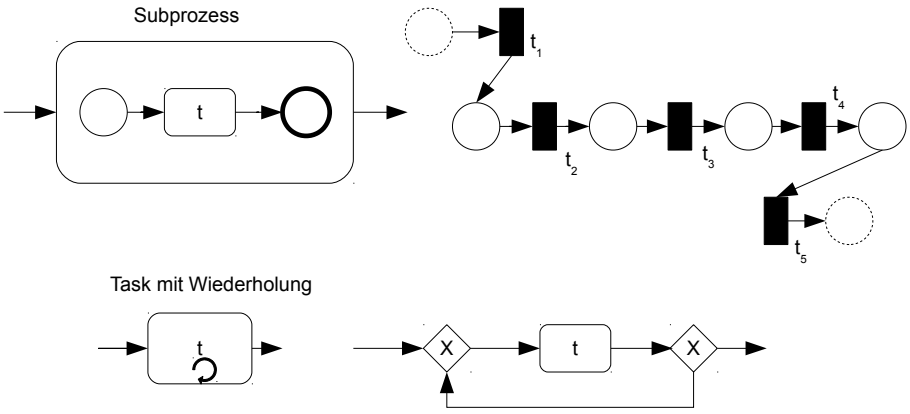


Abbildung 5.10.: Mapping BPMN-Elemente Petrinetz Spezialfälle, [DDO07]

Anschließend wird in einer Endlosschleife in jeder Iteration geprüft, welche Transitionen aktuell möglich sind. Aus den möglichen Transitionen wird dann eine einzige zufällig ausgewählt und ausgeführt. Realisiert wird dies durch ein `if`-Konstrukt, welches in Promela entsprechend definiert ist und Nichtdeterminismus zulässt (siehe Abschnitt 3.3.1).

Einige Plätze sind als Endplätze markiert. Wird ein solcher Platz erreicht, wird die Schleife verlassen und das Programm gilt als korrekt ausgeführt.

Nachverfolgung des Gegenbeispiels

Damit der Benutzer nachvollziehen kann, wie die definierte LTL-Regel durch die aktuell erstellte Prozessvariante verletzt werden kann, wird dem Benutzer ein Gegenbeispiel geliefert.

Spin schreibt bei einer Regelverletzung die Herleitung des Gegenbeispiels in eine sogenannte Trail-Datei [bas]. Dabei werden alle Variablenbelegungen nach jedem Schritt festgehalten (und damit auch die Belegung des Arrays mit den Platzbelegungen). Prinzipiell lässt sich aus dieser Trail-Datei somit ohne weitere Hilfsmittel ermitteln, welche Transitionen ausgeführt wurden, um das Gegenbeispiel zu erzeugen, da keine zwei Transitionen dieselben Änderungen am Platzarray vornehmen.

Ein anderer Ansatz [Wol10] dagegen gibt bei jeder durchgeführten Transition eine entsprechende Logging-Ausgabe mit dem *printf*-Statement aus C aus. Dieses lässt sich verwenden, da Promela von Spin zunächst in C überführt wird. Eingebetteter C-Code wird dabei übernommen. Die Ausgabe dieser Logging-Statements findet sich anschließend ebenfalls in der Trail-Datei.

In der hier erarbeiteten Lösung wurde der Ansatz mit den speziellen Logging-Ausgaben gewählt. Der Hauptgrund für diese Wahl ist ein deutlich weniger fehleranfälliges Parsen, denn abhängig von der Anzahl der gesetzten Tokens einer Transition unterscheidet sich

die Anzahl der Variablenausgaben für eine Transition. Wird nur ein Token von einem Platz entfernt und auf einen anderen Platz gesetzt, werden zwei Komplettbelegungen ausgegeben. Bei der Umsetzung eines exklusiven Gateways kann aber auch die Situation auftreten, dass von einem Platz ein Token entfernt, dafür aber auf zwei andere Plätze jeweils ein Token gesetzt wird. In diesem Fall würden drei komplette Variablenbelegungen in der Logdatei ausgegeben werden. Das heißt abhängig von der Transition muss eine unterschiedliche Anzahl an Variablenbelegungen ausgelesen werden. Da beim Ansatz mit den speziellen Logging-Ausgaben pro Transition nur eine Ausgabe erfolgt, ist das Parsen der Trail-Datei einfacher und damit weniger fehleranfällig.

Bei der Ausgabe der Trail-Datei kann dann darüber hinaus auf die Ausgabe der kompletten Variablenbelegung nach jedem einzelnen Schritt verzichtet werden, wodurch ein schnelleres Parsen möglich ist, da die zu parsende Ausgabe kleiner ist.

Promela-Entsprechung des laufenden Beispiels

Die Promela-Entsprechung des laufenden Beispiels findet sich in Listing 5.2. Nach der Deklaration des Arrays für die Platzbelegungen folgen die Makros zur Erkennung, welche Tasks ausgeführt wurden.

Im Makroblock für die Transitionen wird jede Transition in zwei Makros umgesetzt, die Startbedingung und die Fire-Regel. In jeder Startbedingung wird explizit für jeden Zielplatz geprüft, dass dieser auch leer ist. Damit wird sichergestellt, dass das Petrinetz 1-sicher ist. Im *test*-Prozess wird nach der Herstellung der Initialbelegung so lange zufällig eine der möglichen Transitionen durchgeführt, bis der Endplatz 10 erreicht wird.

Jede durchgeführte Transition wird mit einem `printf`-Statement dokumentiert (siehe auch Nachverfolgung des Gegenbeispiels 5.7.2).

```
byte p[11];

#define Task2 p[2]
#define Task3 p[5]
#define Task4 p[6]
#define Task5 p[9]

#define rd_t0 p[0] && !p[1]
#define fire_t0 p[0] = 0; p[1] = 1;
#define rd_t1 p[1] && !p[2]
#define fire_t1 p[1] = 0; p[2] = 1;
#define rd_t2 p[2] && !p[3]
#define fire_t2 p[2] = 0; p[3] = 1;
#define rd_t3 p[2] && !p[4]
#define fire_t3 p[2] = 0; p[4] = 1;
#define rd_t4 p[3] && !p[5]
#define fire_t4 p[3] = 0; p[5] = 1;
#define rd_t5 p[4] && !p[6]
#define fire_t5 p[4] = 0; p[6] = 1;
#define rd_t6 p[5] && !p[7]
```

```
#define fire_t6 p[5] = 0; p[7] = 1;
#define rd_t7 p[6] && !p[7]
#define fire_t7 p[6] = 0; p[7] = 1;
#define rd_t8 p[7] && !p[8]
#define fire_t8 p[7] = 0; p[8] = 1;
#define rd_t9 p[8] && !p[9]
#define fire_t9 p[8] = 0; p[9] = 1;
#define rd_t10 p[9] && !p[10]
#define fire_t10 p[9] = 0; p[10] = 1;

active proctype test()
{
  d_step { p[0] = 1; p[1] = 0; p[2] = 0; p[3] = 0; p[4] = 0; p[5] = 0; p[6] = 0; p[7] = 0;
      p[8] = 0; p[9] = 0; p[10] = 0; }
  do
    :: rd_t0 -> d_step{printf("PROCESSED_t0"); fire_t0}
    :: rd_t1 -> d_step{printf("PROCESSED_t1"); fire_t1}
    :: rd_t2 -> d_step{printf("PROCESSED_t2"); fire_t2}
    :: rd_t3 -> d_step{printf("PROCESSED_t3"); fire_t3}
    :: rd_t4 -> d_step{printf("PROCESSED_t4"); fire_t4}
    :: rd_t5 -> d_step{printf("PROCESSED_t5"); fire_t5}
    :: rd_t6 -> d_step{printf("PROCESSED_t6"); fire_t6}
    :: rd_t7 -> d_step{printf("PROCESSED_t7"); fire_t7}
    :: rd_t8 -> d_step{printf("PROCESSED_t8"); fire_t8}
    :: rd_t9 -> d_step{printf("PROCESSED_t9"); fire_t9}
    :: rd_t10 -> d_step{printf("PROCESSED_t10"); fire_t10}
    :: p[10] -> goto accept
  od;
  accept: printf("Accepted");
}
```

Listing 5.2: Beispiel Promela aus Petrinetz

6. Implementierung

Das im vorhergehenden Kapitel beschriebene Konzept wird nun exemplarisch im Oryx-Editor umgesetzt.

Bei der Beschreibung der Implementierung wird nach einem kurzen Überblick über die Gesamtarchitektur zunächst auf die Erweiterung des Backends eingegangen. Im Anschluss an die Betrachtung des Frontends wird die Erweiterbarkeit näher erläutert. Den Abschluss bilden einige aus der Implementierung gewonnene Erkenntnisse.

6.1. Übersicht

Die in Abbildung 6.1 dargestellte Architektur der hier entwickelten Lösung baut auf den Vorarbeiten in [Köt10] auf. Zusätzlich zu den dort eingebrachten Erweiterungen zur Variabilität werden Back- und Frontend um folgende Komponenten erweitert:

Im Backend werden zwei neue Servlets erstellt. Das ComplianceServlet ist für die Compliance Checks und die Exportfunktion von Petrinetzen zuständig, das LTLServlet stellt die Funktionalität zur Ad-hoc-Ermittlung der textuellen Darstellungen eines LTL-Modells bereit.

Das Frontend wird wie folgt erweitert: Für das Design von LTL-Modellen wird ein neues Stencilset erstellt. Das Stencilset aus der vorhergehenden Diplomarbeit zur Variabilität wird um einen neuen Stencil *ComplianceScope* erweitert. Dieser Stencil basiert auf dem Stencil *Subprocess*, das heißt er kann weitere BPMN-Elemente enthalten und *SequenceFlows* als eingehende und ausgehende Kanten erhalten. Zur Repräsentation der Compliance Rule erhält der Stencil eine neue Property *Rule*, die den Regelbaum enthält.

Hinzu kommen zwei neue Plugins. Das Plugin *Compliance Wizard* erweitert die Oryx-GUI um die neuen Funktionalitäten Compliance Wizard zur Bearbeitung des Regelbaums, Compliance-Check durchführen, Compliance-Ergebnis anzeigen, Compliance-Ergebnis zurücksetzen, den Export des markierten Compliance Scope als PNML-Diagramm und den Ex- und Import von Compliance Scopes.

Das Plugin *LTLPlugin* fügt der Toolbar von Oryx einen Button hinzu, mit dem sich der Benutzer das aktuelle LTL-Diagramm als textuelle Darstellung ausgeben lassen kann.

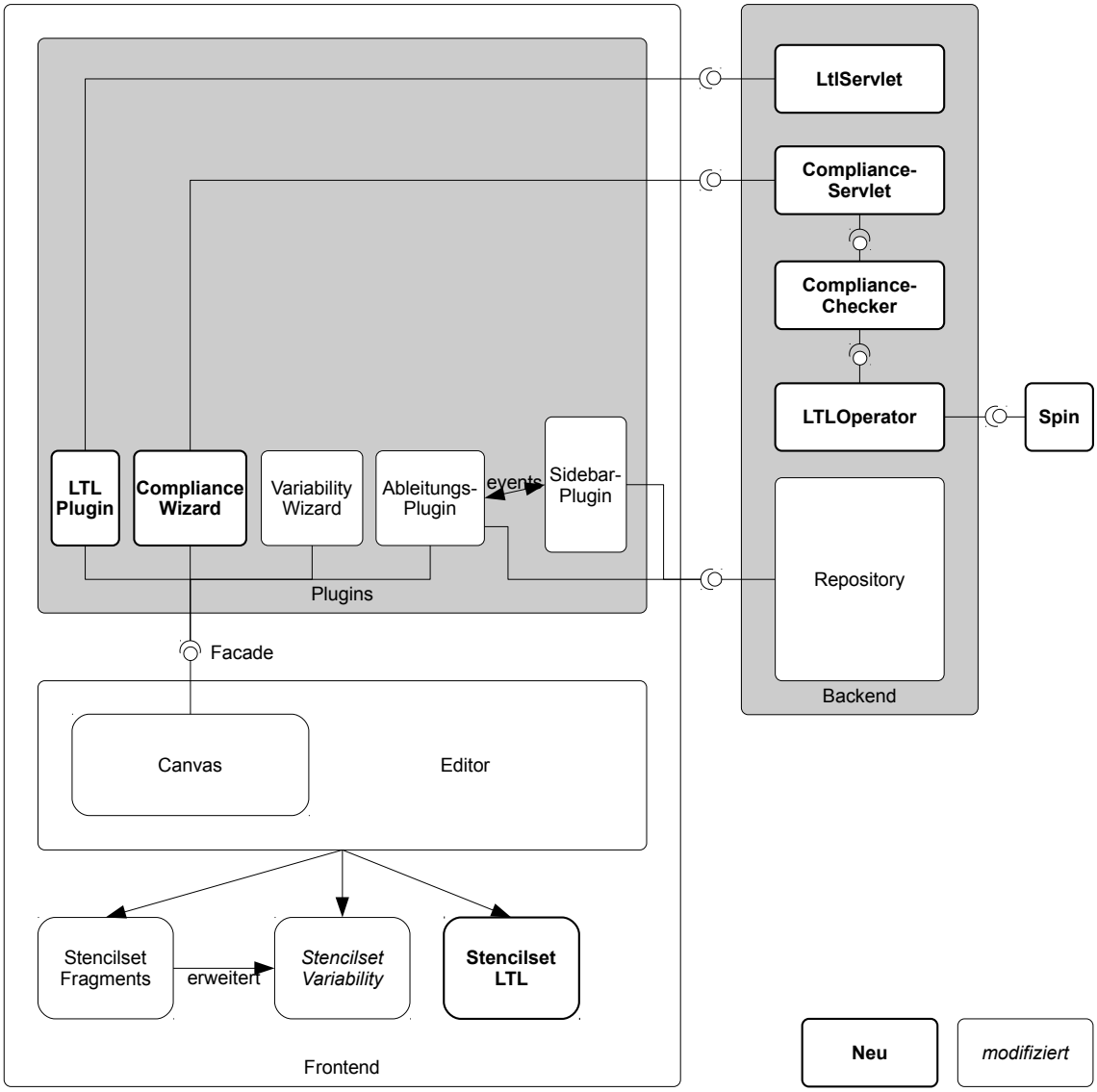


Abbildung 6.1.: Architektur der Oryx-Erweiterung (baut auf [Köt10] auf)

6.2. Backend

Der größte Teil der Implementierung wird im Backend umgesetzt. Das ist zum einen dadurch begründet, dass der Model Checker hier ausgeführt wird und die zugehörigen Mappings deshalb auch hier durchgeführt werden. Zum anderen ist die Implementierung hier leichter zu testen, da mit JUnit und Eclipse mit einem integrierten Debugger ausgereifte Werkzeuge zur Verfügung stehen. Die wichtigsten Aspekte der Implementierung des Backends werden in den folgenden Abschnitten beschrieben.

6.2.1. Testbarkeit

Um eine gute Testbarkeit der Implementierung zu gewährleisten, wurde weitestgehend auf statische Klassen verzichtet und das Prinzip der Dependency Injection [Fow] angewendet. Dieses wurde in Form von Kontexten umgesetzt. Sie vereinen die Informationen und zu nutzenden Interface-Implementierungen, die zur Bearbeitung einer Aufgabe benötigt werden. So erhält die Klasse, die den Regelbaum eines Compliance Scopes auswertet, Zugriff auf den Adapter, der den Zugriff auf das Modell-Repository zum Nachladen von benötigten LTL-Modellen ermöglicht, und den Adapter zum Zugriff auf den Model Checker Spin.

Beim Test können die Standardimplementierungen der Adapter durch eigene Implementierungen ersetzt werden. Das ermöglicht das Testen der Implementierung ohne Zugriff auf das Oryx-Modelrepository, indem entsprechend vorbereitete Daten zurückgegeben werden.

Der Einsatz der Dependency Injection erlaubt außerdem die Unabhängigkeit von Spin und dem C-Compiler während der Entwicklung, da hier entsprechende Beispielausgaben hinterlegt und damit auch entsprechende Fehler generiert werden können.

6.2.2. OryxGraph

An mehreren Stellen im Backend wird auf die von Oryx genutzte JSON-Darstellung von BPMN- und LTL-Modellen zugegriffen. Dies ist unter anderem der Fall bei den Mappings von LTL-Diagrammen und BPMN-Modellen auf die jeweiligen Eingabeformate für den Model Checker sowie der Auswertung des DataTransfer-Operators. Um den Zugriff zu vereinfachen, wird eine Hilfsklasse zum Auswerten dieser Struktur eingeführt, der *OryxGraph*. Dieser stellt einige oft benötigte Methoden zur Verfügung.

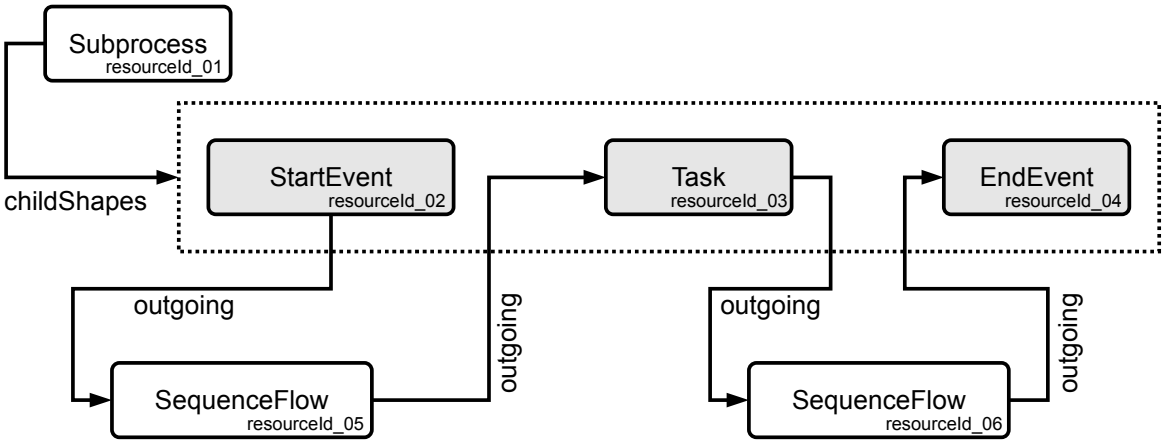


Abbildung 6.2.: Datenstruktur Oryx-JSON-Format

In Abbildung 6.2 ist der Aufbau der JSON-Darstellung eines BPMN-Modells exemplarisch veranschaulicht. Der Subprozess enthält die beiden Ereignisse und den Task als Kindelemente,

die Umsetzung der mit *outgoing* bezeichneten Kanten erfolgen über IDs. Auf Grund des hierarchischen Aufbaus und der ausschließlich vorhandenen Vorwärtsnavigation ergeben sich mehrere Schwierigkeiten beim Zugriff. Zum einen ist keine Rückwärtsnavigation direkt in der Datenstruktur möglich. Diese wird aber zum Beispiel beim Umsetzen von Gateways beim Mapping eines BPMN-Modells auf die Petrinetzdarstellung benötigt. Außerdem ist kein direkter Zugriff über die ResourceId vorgesehen, das heißt beim Entlangwandern einer Kette von Elementen, wie vom Startereignis über die outgoing-Kanten zum Endereignis, muss für jede ResourceId das zugehörige Element in der JSON-Darstellung herausgesucht werden, deshalb wird der OryxGraph eingesetzt.

Beim Mapping eines LTL-Diagramms in die textuelle Darstellung parst der OryxGraph einmal die komplette JSON-Darstellung. Anschließend stehen mehrere Informationen zur Verfügung. Neben dem Mapping aller vorhandenen ResourceIds auf die jeweiligen JSON-Objekte, können für jedes Element die eingehenden und ausgehenden Kanten abgefragt werden. So kann zum Beispiel für den Task in der Beispielabbildung direkt ermittelt werden, welche beiden Elemente sich an seinen beiden Enden befinden. Bei der Abfrage der ausgehenden oder eingehenden Elemente lässt sich außerdem bestimmen, dass nur die Elemente ausgegeben werden, die einem gewünschten Typ entsprechen. So können bei dem Task direkt die ausgehenden Nachrichten- oder Kontrollflusskanten abgefragt werden.

Außerdem leistet der OryxGraph die Unterscheidung in Knoten- und Kantenstenciltypen. Bei der Initialisierung des OryxGraph werden dafür die Knoten- und Kantenstenciltypen angeben. Zu einem gegebenen Stencil kann dann ermittelt werden, ob es sich um einen Knoten- oder Kantentyp handelt.

Die letzte Funktion schließlich erlaubt das Auflisten aller in einem definierten Element enthaltenen Elemente. Hierbei kann angegeben werden welche Containerelemente als Hierarchiegrenzen angesehen werden. Zum Beispiel lässt sich damit festlegen, dass Lanes nicht als Hierarchiegrenze betrachtet werden.

6.2.3. Umwandeln der graphischen Darstellung der LTL-Formeln in die textuelle Repräsentation

Die textuelle Repräsentation einer LTL-Formel aus einem in JSON abgelegten LTL-Modell wird rekursiv durchgeführt. Container-Elemente sind hier die unären Operatoren Klammerung, Negation, Finally und Globally. Diese lösen einen neuen Schritt in der Rekursion aus. Abbildung 6.3 zeigt ein LTL-Diagramm.

In jedem Container können sich Ketten von Operatoren befinden, in der Abbildung enthält der Globally-Operator zwei Ketten. Eine beginnt mit dem *Task2*-Propertyoperator, die andere mit dem NOT-Operator. Von jeder Kette wird der Startknoten ermittelt, welcher dadurch definiert ist, dass er keine eingehenden Kanten hat. Anschließend wird entlang der Kette gewandert und die Operatoren werden auf dem Weg übersetzt. Bei einem unären Operator findet dabei ein neuer Rekursionsschritt statt.

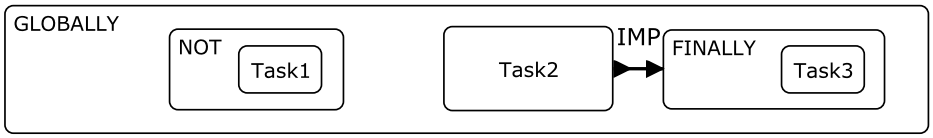


Abbildung 6.3.: Beispiel LTL-Modell

Die einzelnen Operatoren-Ketten in einem Container werden anschließend per AND verknüpft. Die textuelle Repräsentation der in Abbildung 6.3 dargestellten LTL-Formel lautet damit `Globally (Not(Task1) And (Task2 -> Task3))`.

Bei der Abarbeitung des LTL-Diagrammes werden auch eine Reihe von Fehlern in LTL-Modellen erkannt, die nicht durch entsprechende Constraints im LTL-Stencilset vermieden werden können:

- Ein binärer Operator darf keine zwei Operatoren mit unterschiedlichen Elternoperatoren verbinden.
- Binäre Operatoren dürfen keine losen Enden besitzen, das heißt an jedem ihrer Enden muss sich ein weiterer Operator befinden.

Im Falle, dass ein Containerelement keine Operatoren enthält, wird ein `true`-Operator eingefügt.

Während des Erstellens einer LTL-Formel im graphischen Editor kann sich der Benutzer über einen Button in der Toolbar die textuelle Darstellung der LTL-Formel anzeigen lassen. Dabei wird er auch über die oben angegebenen Fehler informiert.

6.2.4. Umwandeln des BPMN-Modells in ein Petrinetz

Das Umformen des BPMN-Modells wird durch das Interface BPMNTranslator bzw. seine Standardimplementierung BPMNTranslatorImpl durchgeführt. Abbildung 6.4 gibt einen Überblick über die für das Mapping zuständigen Klassen.

Die Übersetzung findet dabei in einem TranslatorContext statt, der den Zugriff auf die verwendete OryxGraph-Instanz und die TranslatorFactory bereitstellt. Für jedes unterstützte BPMN-Element ist ein sogenannter Translator zuständig, der die Übersetzung dieses BPMN-Elementtyps in eine Petrinetzkomponente übernimmt. Die TranslatorFactory verwaltet die Zuordnung von BPMN-Elementen auf den entsprechenden Translator. Translators werden wiederverwendet, das heißt nicht für jedes BPMN-Element neu erzeugt. Zusätzlich kann ein Translator für mehrere BPMN-Elementtypen zuständig sein.

Bei Prüfbedingungen für die Ausführbarkeit einer Transition muss gewährleistet sein, dass alle Zielplätze leer sind, da sonst nicht gewährleistet werden kann, dass das erstellte Petrinetz 1-sicher ist.

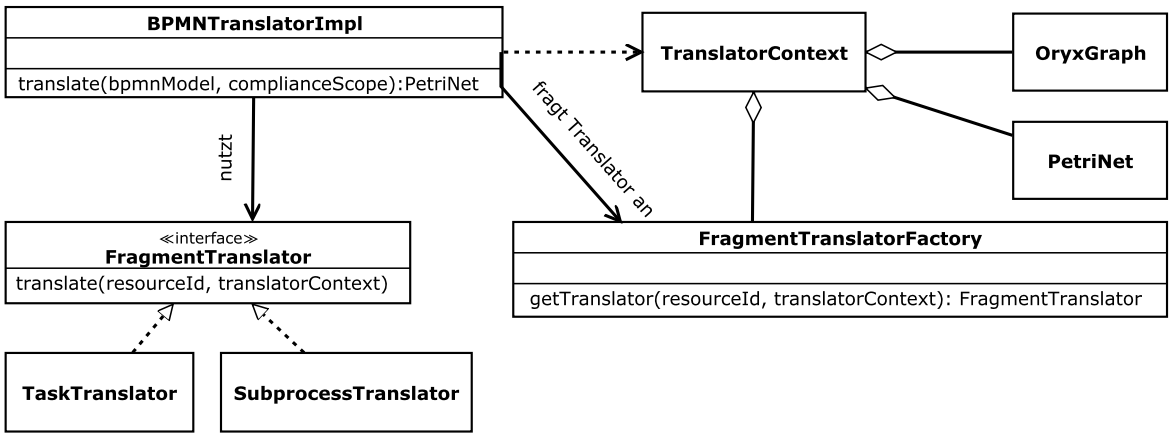


Abbildung 6.4.: BPMNTranslator

Makros für Tasks

Bei der Übersetzung werden Makros für den späteren Promela-Code erzeugt, mit denen bei der späteren Ausführung überprüft werden kann, ob ein Task gerade ausgeführt wird. Dies wird für die Auswertung der LTL-Formeln benötigt. Repräsentiert Platz Nummer 3, dass *Task2* ausgeführt wird, so wird ein Makro `#define Task2 p[3]` erzeugt.

Da in den LTL-Formeln der Compliance-Regeln Tasknamen verwendet werden können, die nicht im Prozessmodell enthalten sein müssen, werden die in den LTL-Formeln enthaltenen Tasknamen in einem weiteren Schritt vor dem Compliance Check extrahiert und mit `false` vorbelegt definiert. Findet sich im Prozessmodell ein Task mit einem Namen, der in einer LTL-Formel verwendet wird, werden die vordefinierten Makros mit den echten Prüfbedingungen überschrieben.

Ergebnisstruktur PetriNet

Das Ergebnis des BPMNTranslators wird in der Klasse PetriNet abgelegt, die wie in Abbildung 6.5 dargestellt aufgebaut ist. Hier sind die Anzahl der Plätze, die Transitionen, die anfänglich belegten Plätze und die Endplätze abgelegt.

Eine Transition besteht aus einer Menge eingehender und ausgehender Plätze. Ein eindeutiger Identifier wird beim Schreiben des Gegenbeispiels durch das Promela-Testprogramm ausgegeben und dient anschließend der Zuordnung der getätigten Transitionen des Gegenbeispiels. Der zusätzlich enthaltene Kommentar dient der Darstellung für den Endbenutzer.

Caching

Dem Prozesstemplatedesigner fällt die Erstellung kleinerer LTL-Formeln leichter, sodass mit hoher Wahrscheinlichkeit mehrere LTL-Operatoren über demselben Compliance Scope ausge-

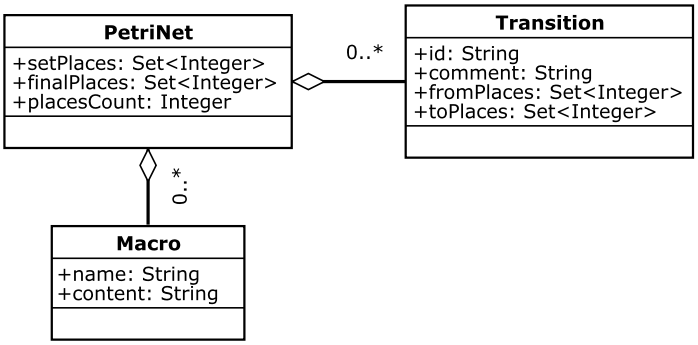


Abbildung 6.5.: Struktur PetriNet

wertet werden. So lässt sich die LTL-Formel aus Abbildung 6.3 `Globally (Not(Task1) And (Task2 -> Task3))` auch als Konjunktion der beiden Formeln `Globally (Not(Task1))` und `Globally ((Task2 -> Task3))` darstellen. Damit werden statt einer nun zwei LTL-Formeln über demselben Compliance Scope überprüft, dabei ändert sich die Petrinetzdarstellung des BPMN-Modells nicht. Deshalb werden generierte Petrinetzdarstellungen gespeichert und wiederverwendet.

Export des generierten Petrinetzes

Das generierte Petrinetz lässt sich in eine Reihe von Formaten exportieren. Neben der für die Auswertung des LTL-Operators benötigte Promela-Darstellung ist auch eine Ausgabe als PNML und eine Ausgabe als vereinfachte Textdarstellung möglich.

Die Petri Net Markup Language (kurz PNML) [pnmb] ist eine in XML formulierte Beschreibungssprache für Petrinetze, die von einer Reihe von Petrinetz-Editoren unterstützt wird. Aus der PNML-Darstellung lässt sich mittels des Hilfsprogramms PNML 2 dot [pnma] eine Zwischendarstellung im DOT-Format [DOT] erzeugen. Das DOT-Format kann dann mit dem Werkzeug GraphViz [gra] in eine graphische Darstellung überführt werden. Die Funktionalität zum Export als PNML wird auch im Frontend für den Endbenutzer angeboten.

Die vereinfachte Textdarstellung dient dem Debugging bei der Erstellung von Mappings von BPMN-Elementen auf Petrinetzelemente. Hier werden die enthaltenen Transitionen und Makros, sowie die anfangs gesetzten Plätze und finalen Plätze aufgelistet.

6.2.5. Auswertung der Compliance-Regeln

Zur Auswertung der Compliance-Regeln wird im BPMN-Modell nach den Compliance Scopes gesucht. Die angehängten, als Regelbaum dargestellten, Compliance-Regeln werden ausgewertet und aus dem Ergebnis dieser Einzelergebnisse wird das Gesamtergebnis aufgebaut.

Die Auswertung logischer Operatoren erfolgt als Auswertung auf den Operanden, die von DataTransfer- und LTL-Operatoren anhand der im Konzept beschriebenen Semantik.

Suche nach den Compliance Scopes und ihre Auswertung

Die Klasse ComplianceChecker erhält als Eingabe ein BPMN-Modell in der von Oryx verwendeten JSON-Darstellung übergeben und gibt als Ergebnis ein CompleteComplianceResult zurück. Dies entspricht dem Ergebnis aus Kapitel 5.4.

Zur Ermittlung des Ergebnisses sucht der ComplianceChecker rekursiv die JSON-Darstellung nach Compliance Scopes ab. Ist für den aktuellen Compliance Scope keine Regel definiert, steht das Ergebnis als *NoRulesDefined* bereits fest. Andernfalls fragt der ComplianceChecker bei der ComplianceOperatorFactory die Implementierung für den als Wurzelknoten verwendeten Operator nach. Auf dieser wird dann die Methode *evaluate* aufgerufen, die als Parameter eine Referenz auf das BPMN-Modell, den Compliance Scope und den auszuwertenden Operator erhält.

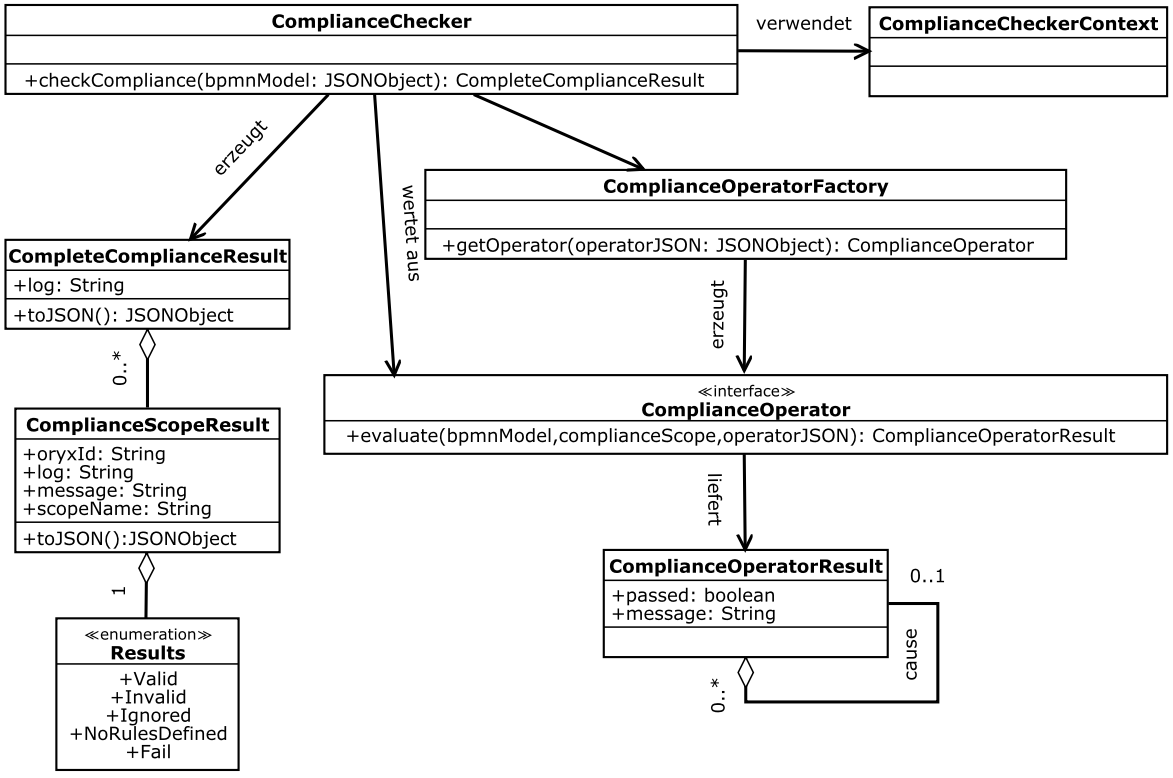


Abbildung 6.6.: ComplianceChecker

Die ComplianceOperatorFactory erzeugt von jedem Operator nur eine Instanz, damit das Caching innerhalb eines Operators ermöglicht wird. Generierte Zwischendarstellungen können somit in einer Operatorenimplementierung weiterverwendet werden.

Der ComplianceCheckerContext

Der ComplianceCheckerContext fasst alle während eines Compliance Checks benötigten Informationen und Interface-Implementierungen zusammen. Dazu enthält er einen Verweis auf einen RepositoryConnector, darüber kann dann auf die Modelle des Oryx-Repositories zugegriffen werden. Dies wird benötigt, um beim Auswerten eines LTL-Operators die referenzierten LTL-Modelle nachzuladen.

Außerdem sind hier Referenzen auf die zu nutzenden Implementierungen des BPMNTranslator und LTLTranslator hinterlegt. Der SpinAdapter wird zum Zugriff auf Spin benötigt und kapselt die Aufrufe von Spin und dem GCC, außerdem parst er die entsprechenden Ergebnisse.

Die CmdExecution wird vom SpinAdapter genutzt und kapselt den Aufruf nativer Anwendungen wie der Model Checkers und des GCC, fängt die Ergebnisse ab und gibt diese an den Aufrufer zurück.

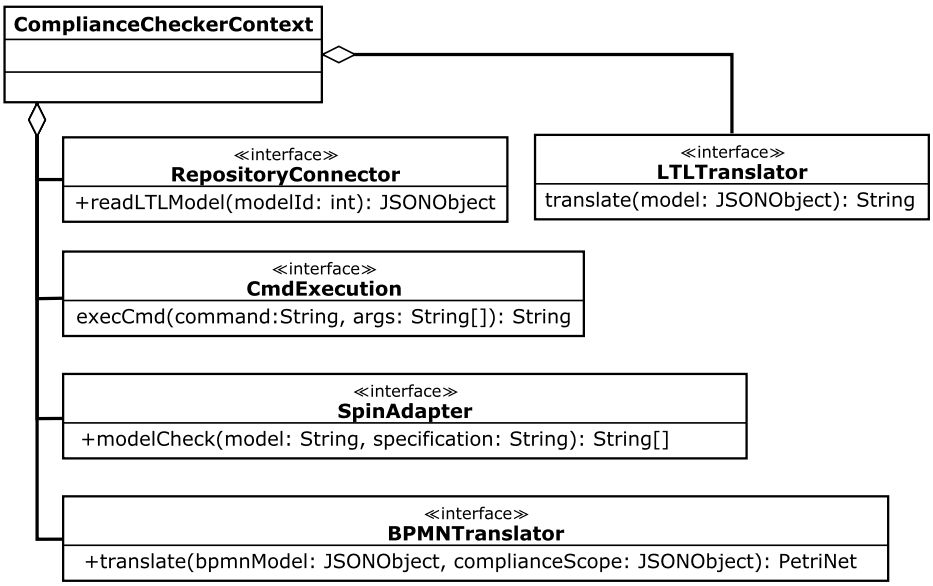


Abbildung 6.7.: ComplianceCheckerContext

Logische Operatoren

Der NOT-Operator fragt bei der ComplianceOperatorFactory die Implementierung für den enthaltenen Operanden an und wertet den Operator mit der erhaltenen Implementierung aus. Anschließend wird das Ergebnis negiert.

Die Implementierung des OR-Operators arbeitet in einer Schleife die enthaltenen Operatoren nacheinander ab. Jeder Operator wird mit der aus der ComplianceOperatorFactory erhalte-

nen Implementierung ausgewertet. Sobald der erste Operator erfüllt ist, wird das Ergebnis des OR-Operators auf erfüllt gesetzt und die Schleife abgebrochen.

Die Implementierung des AND-Operators erfolgt entsprechend der des OR-Operator. Hier wird das Ergebnis des AND-Operators zunächst als erfüllt angenommen und beim ersten Auftreten eines nicht erfüllten Operanden auf nicht erfüllt gesetzt und die Schleife abgebrochen.

LTL-Regeln

Das Auswerten einer LTL-Regel über einem Compliance Scope erfolgt in mehreren Schritten. Zunächst wird das LTL-Model, welches die zu prüfende LTL-Formel repräsentiert, über den RepositoryConnector ausgelesen und anschließend mittels des LTLTranslators in die textuelle Darstellung überführt.

Im nächsten Schritt wird das BPMN-Modell, beziehungsweise der davon benötigte Teil des Compliance Scopes, in ein Petrinetz übersetzt. Wurde der Compliance Scope bereits einmal in ein Petrinetz übersetzt, so wird die alte, zwischengespeichert Version, wiederverwendet. Anschließend wird das Petrinetz als Promela-Quellcode, also in der Eingabesprache für die Modellbeschreibung für Spin, exportiert.

Die Modellspezifikation in der LTL und die Systembeschreibung als Promela-Quellcode werden anschließend dem SpinAdapter übergeben, der zunächst in einem Zwischenschritt Spin die LTL-Spezifikation in eine Promela-Never-Clause übersetzen lässt. Beide Promela-Darstellungen werden anschließend konkateniert in eine temporäre Datei geschrieben und von Spin in C-Quellcode übersetzt. Dieser wird mittels dem installierten GCC in ein ausführbares Programm übersetzt.

Das generierte Testprogramm wird ausgeführt und die Ausgabe des Programms geparst. Im Falle einer Verletzung der LTL-Spezifikation durch das Modell wird Spin ein weiteres Mal mit der Trace-Datei ausgeführt, um die Herleitung des Gegenbeispiels zu erhalten. In dieser Herleitung sind die IDs der getätigten Transitionen im Petrinetz enthalten, von welchen dann auf die durchgeführten Transitionen geschlossen werden kann. Aus der Transitionenabfolge wird anschließen die textuelle Darstellung des Gegenbeispiels für den Endbenutzer erstellt.

DataTransfer-Regeln

Die DataTransfer-Regeln werden in mehreren Schritten ausgewertet. Zunächst wird nach dem Datenobjekt anhand des Namens, dessen Angabe verpflichtend ist, gesucht. Die Suche berücksichtigt dabei nur die in dem Compliance Scope enthaltenen BPMN-Elemente, da sich die DataTransfer-Regeln eines Compliance Scopes nur auf die im Compliance Scope enthaltenen Datenobjekte beziehen.

Anschließend werden die relevanten Kanten (DataAssociations) anhand der Richtung, der Datenaustauschpartner und ob die Grenzen des Compliance Scopes überschritten werden,

herausgesucht. Der OryxGraph kann beim Aufruf anhand der Richtung und des Kantentyps, direkt irrelevante Kanten ausfiltern. Die Information, ob die Grenze des Compliance Scope überschritten wird, lässt sich auch aus dem OryxGraph ermitteln.

Das im Datenobjekt hinterlegte XML-Schema wird ausgelesen. Aus den XPath-Ausdrücken aus der Compliance-Regel und aus den Assignmentregeln der DataAssociation wird mit den im Konzept beschriebenen Umsetzungen ein gemeinsamer XPath-Ausdruck anhand des Regeltyps gebildet. Mittels Saxon [sax] wird dieser gemeinsame XPath-Ausdruck anschließend über dem XML-Schema ausgewertet und die Ergebnismenge auf das Vorhandensein von Knoten geprüft. Sind Knoten in der Ergebnismenge enthalten, so ist die Regel verletzt.

6.2.6. Logging

Um die Nachvollziehbarkeit der Regelauswertungen sowohl für den Implementierer als auch für den späteren Nutzer der Oryx-Erweiterung zu erhöhen, werden in der hier erarbeiteten Lösung während der Compliance-Regel-Auswertung für die Auswertung relevante Informationen in Logs geschrieben.

Zum einen wird dazu der Logging-Mechanismus über Log4J [The] genutzt. Hier finden sich vor allem implementierungsspezifische Informationen wie zum Beispiel die Rückgaben von Spin und des GCC. Die entsprechenden Loggingausgaben lassen sich in den Logdateien von Apache Tomcat nachlesen.

Zum anderen muss der Endbenutzer aber auch die Ergebnisse der Compliance Checks nachvollziehen können. Dafür wird ein eigener Loggingmechanismus eingesetzt, der speziell auf den Endbenutzer zugeschnittene Informationen enthält. Für jeden Compliance Scope und für das Endergebnis werden dabei separate Logs erstellt. Die Logs können nach einem Compliance Check über die Oberfläche angezeigt werden.

Prinzipiell ist es möglich, den zweiten Loggingtyp durch Abfangen und Aufbereiten des Loggings über Log4J zu realisieren. Eine entsprechende Umsetzung ist aber sehr von Log4J abhängig und erschwert damit den Umstieg auf andere Logging-Lösung, wie das in neueren Java-Versionen integrierte Logging.

6.2.7. Ergebnisformat

Die Darstellung des Gesamtergebnisses wird in JSON realisiert, da dieses auf Frontend-Seite geparst wird und das Parsen von JSON in JavaScript über die eval-Funktion bereits enthalten ist.

In Logs und Messages müssen Sonderzeichen wie beispielsweise Zeilenumbrüche escaped werden, da sonst die JSON-Struktur verletzt werden würde. Auf der Frontendseite werden diese mittels String-Funktionen [Koc09] in JavaScript wieder zurückgewandelt.

Ein Gesamtergebnis kann wie in Listing 6.1 gezeigt aussehen.

```
{
    "log": "Performing compliance check for all compliance scopes... ",
    "scopeResults": [
        {
            "log": "Checking compliance scope Unnamed Compliance Scope #1",
            "message": "Model did not match specification '<>(Task3)', counterexample as
                follows...",
            "oryxId": "oryx_F2BFE1E5-EC21-46AE-93BF-5A7940F7BE5F",
            "result": "Invalid",
            "scopeName": "Unnamed Compliance Scope #1"
        },
        {
            "log": "",
            "message": "No rules defined.",
            "oryxId": "oryx_8586F9E9-769B-4C23-90B8-2D86945F57D0",
            "result": "NoRulesDefined",
            "scopeName": "Unnamed Compliance Scope #2"
        }
    ]
}
```

Listing 6.1: Resultat JSON

6.3. Frontend

Die Implementierung des Frontends findet in JavaScript statt. Als Zusatzbibliotheken kommen vor allem Prototype [pro] und ExtJS [ext] zum Einsatz. Prototype erweitert JavaScript um objektorientierte Konzepte und AJAX-Funktionalitäten, ExtJS erweitert JavaScript um ein GUI-Framework.

Modale Dialoge

Alle Dialoge im Frontend werden mittels ExtJS realisiert, welches diese in HTML-Primitive und damit als Teil der HTML-Seite umsetzt. Dadurch wird das Blockieren des Browsers während der Anzeige des Dialogs verhindert, wie dieses bei Verwendung der durch JavaScript bereitgestellten *alert*-Funktion eintritt. Damit kann der Benutzer während der Anzeige eines Dialogs in einen anderen Tab wechseln und so zum Beispiel prüfen, ob das richtige LTL-Modell für den LTL-Operator gewählt wurde.

6.3.1. Erweiterung der Toolbar

Alle Funktionen der Compliance Erweiterung sind in einem Dropdown-Menü in der Toolbar von Oryx zusammengefasst. Dieses ist in Abbildung 6.8 dargestellt.

Hierüber lassen sich die Regeln des markierten Compliance Scopes bearbeiten (*Compliance Wizard*), alle beziehungsweise nur die markierten Compliance Scopes überprüfen, das

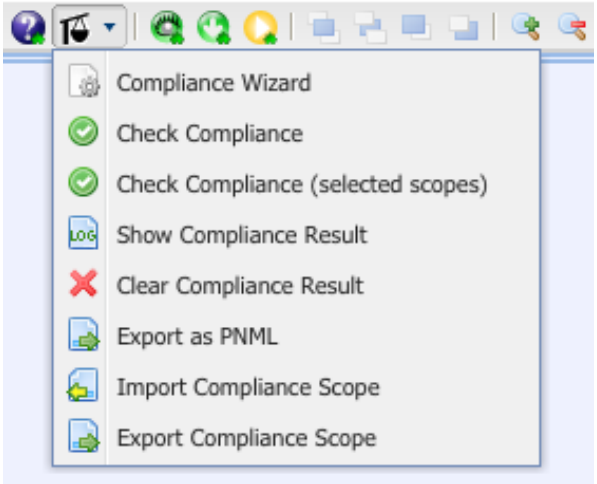


Abbildung 6.8.: Toolbar-Button des Compliance Plugins

Ergebnis des letzten Compliance Checks anzeigen und zurücksetzen, Compliance Scopes exportieren und importieren, sowie die Petrinetzdarstellung eines Compliance Scopes als PNML-Datei herunterladen.

6.3.2. Der Compliance Wizard

Der Compliance Wizard dient der Bearbeitung der mit dem Compliance Scope verknüpften Regel. Hier können Operatoren zum Operatorenbaum hinzugefügt werden und die Eigenschaften der LTL- und DataTransfer-Operatoren bearbeitet werden.

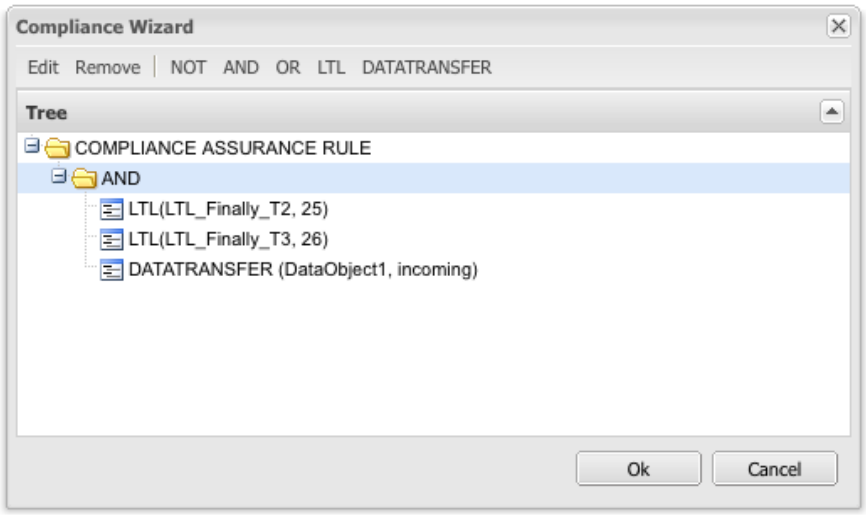


Abbildung 6.9.: Compliance Wizard

Bevor der Regelbaum beim Klick auf OK abgespeichert wird, findet eine Prüfung der Korrektheit des Regelbaums statt. So müssen alle logischen Operatoren mindestens einen Operanden besitzen. Enthält ein NOT-Operator also zum Beispiel keinen Kindknoten, wird der Benutzer darauf hingewiesen und kann den Baum entsprechend korrigieren.

Nach dem Setzen der neuen Regel wird das Ereignis EVENT_EXECUTE_COMMANDS ausgelöst, damit das File-Plugin über die Änderung informiert wird und der Benutzer beim Schließen des Browserfensters eine Nachfrage erhält, ob er die Änderungen am BPMN-Modell speichern möchte.

Editoren für DataTransfer- und LTL-Operatoren

Der in Abbildung 6.10 dargestellte Editor für DataTransfer-Operatoren erlaubt das Festlegen der im Konzept unter 5.2.3 beschriebenen Eigenschaften.

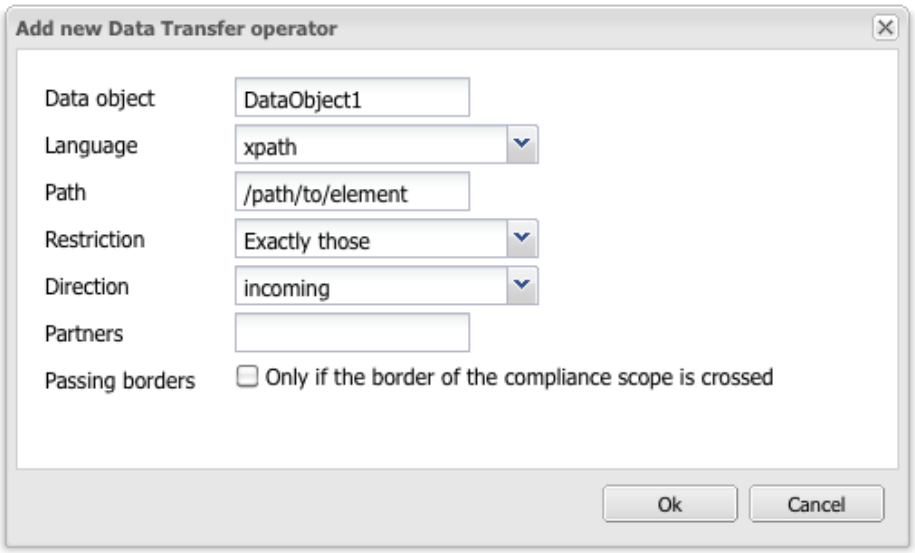


Abbildung 6.10.: Editor für DataTransfer-Regel

Im Editor für LTL-Operatoren kann der Benutzer aus einer Liste der im Oryx-Repository hinterlegten LTL-Modelle wählen.

6.3.3. Nachladen von LTL-Modellen

Neben dem eigentlichen BPMN-Modell mit dem Compliance Scope können zur Verifikation eines Compliance Scopes weitere Diagramme aus dem Oryx-Repository, wie etwa die mittels des LTL-Stencilsets als Diagramme modellierten LTL-Formeln, benötigt werden.

Das Nachladen dieser Diagramme erfolgt auf Client-Seite, da im Gegensatz zur Frontend-Seite [dat] auf Server-Seite für Erweiterungen kein Zugriff auf Repository mit den Modellen vorgesehen ist.

Für das Nachladen von Diagrammen wird die JSON-Darstellung des BPMN-Modells rekursiv nach Compliance Scopes durchsucht. Dabei werden bei einem selektiven Compliance Check, bei dem nur ausgewählte Compliance Scopes überprüft werden sollen, die nicht markierten Compliance Scopes ignoriert. Die Regelbäume der Compliance Scopes werden dann nach LTL-Operatoren durchsucht. Die LTL-Modelle werden anschließend per Ajax-Requests abgefragt. Im ComplianceCheck-Request an den Server werden die geladenen LTL-Modelle in ihrer JSON-Darstellung als Parameter übergeben.

Auf Serverseite wird die Implementierung des RepositoryConnectors verwendet, die ein Mapping von Modellnummer auf die entsprechende JSON-Darstellung enthält.

6.3.4. Darstellung des Ergebnisses

Nach der Durchführung eines Compliance Checks werden die überprüften Compliance Scopes farbig markiert. Die Farbe richtet sich dabei nach dem Ergebnis des Compliance Checks (siehe 5.4) des jeweiligen Compliance Scopes:

grün Die Überprüfung wurde erfolgreich durchgeführt und im Compliance Scope sind alle Regeln erfüllt.

rot Die Überprüfung wurde erfolgreich durchgeführt und im Compliance Scope wird mindestens eine Regel verletzt.

orange Für den Compliance Scope wurde keine Regel definiert. Dies wird separat gekennzeichnet, da hier der Prozesstemplatedesigner mit hoher Wahrscheinlichkeit vergessen hat, die Regeln hinzuzufügen.

grau Die Überprüfung ist fehlgeschlagen.

weiß Der Compliance Scope wurde nicht überprüft.

Im Ergebnisfenster eines Compliance Checks kann anschließend für jeden Compliance Scope detailliert abgelesen werden, wie die Bewertung zustande kam. Bei der Verletzung einer LTL-Regel wird das entsprechende Gegenbeispiel dargestellt.

Die farbigen Overlays und das Ergebnis des Compliance Checks werden beim Speichern des BPMN-Modells nicht mit abgespeichert.

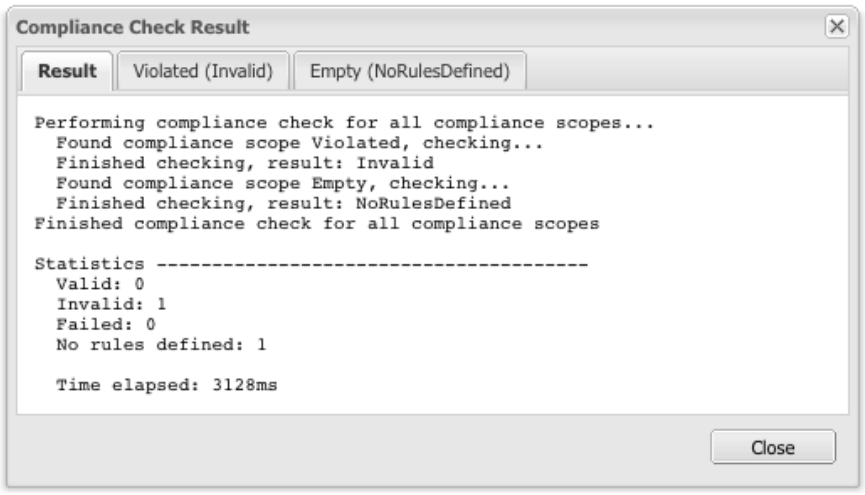


Abbildung 6.11.: Ergebnisfenster

6.3.5. Export und Import von Compliance Scopes

Compliance Scopes können sowohl exportiert als auch importiert werden. Hierzu wurde ein XML-Schema definiert. Exportiert wird dabei nur der Compliance Scope samt Regelbaum, die enthaltenen BPMN-Elemente nicht. Somit lassen sich Compliance Scopes auch in bestehende Prozesse einbinden und einmal definierte Compliance Scopes können mit ihren Regeln wiederverwendet werden.

Wird ein Compliance Scope in einen bestehenden BPMN-Referenzprozess importiert, wird der Compliance Scope als neues Element in das BPMN-Modell eingefügt. Die BPMN-Elemente, die im Compliance Scope enthalten sein sollen, können anschließend per Drag&Drop in den Compliance Scope eingefügt werden. In einem letzten Schritt müssen die Kontrollflusskanten der verschobenen Elemente angepasst werden.

6.4. Erweiterbarkeit

Da die Erweiterbarkeit eine zentrale Anforderungen an die entwickelte Lösung ist, wird in diesem Abschnitt genauer auf die möglichen und vorgesehenen Erweiterungen sowie die dazu notwendigen Anpassungen eingegangen.

6.4.1. Neue Regelbeschreibungssprachen

Bei der Einbindung von neuen Beschreibungssprachen für Regeln müssen sowohl das Backend als auch das Frontend angepasst werden.

Operator im Backend

Für einen neuen Operator ist eine neue Implementierung des ComplianceOperator-Interfaces zu erstellen. Benötigt der Operator dabei weitere Modelle aus dem Oryx-Repository, so lassen sich diese über den RepositoryConnector beziehen.

Anschließend muss der neue Operator in der OperatorFactory registriert werden, damit die Implementierung des Operators bei der Abarbeitung des Regelbaums berücksichtigt wird.

Compliance Rule Editor im Frontend anpassen

Damit der neue Operator auch als neuer Regeltyp vom Benutzer in Oryx verwendet werden kann, ist der Compliance Rule Editor wie folgt anzupassen:

operatorText Diese Funktion erstellt für Operator-Knoten im Baum eine aussagefähige Beschriftung.

rule2Tree Hier wird ein Knoten aus dem Regelbaum in einen Knoten des visuellen Baums übersetzt.

tree2Rule Hier wird ein Knoten aus dem visuellen Baum in einen Knoten des Regelbaums übersetzt.

addOperatorNode Diese Funktion behandelt den Klick auf einen der Toolbarbuttons zum Hinzufügen eines neuen Operators.

editSelectedNode Aus dieser Funktion wird der zum Operatortyp gehörige Operatoreneigenschaftseditor geöffnet.

editComplianceScope Hier wird der Button zum Erstellen eines neuen Knotens mit dem neuen Operatortyp in der ToolBar erzeugt.

findModelsOperator (optional) Referenziert der neue Operatortyp andere Modelle im Repository (wie der LTL-Operator LTL-Modelle), so müssen diese beim Vorladen berücksichtigt werden.

Import und Export anpassen

Damit der neue Operator korrekt importiert und exportiert wird, sind folgende Funktionen im ComplianceWizardPlugIn zu erweitern:

operatorToXml Übersetzt die JSON-Darstellung eines Operators in XML beim Export eines Compliance Scopes.

operatorFromXml Übersetzt die XML-Darstellung eines Operators in die JSON-Darstellung beim Import eines Compliance Scopes.

Außerdem muss das XML-Schema entsprechend ergänzt werden.

6.4.2. Neue Operatoren im LTL-Stencilset

Sind weitere Operatoren wie der Äquivalenz-Operator dem LTL-Stencilset hinzuzufügen, muss die Stencilset-Beschreibung erweitert werden. Dies erfolgt in der JSON-Datei, die das Stencilset definiert. Außerdem müssen sowohl eine Icon-Datei für die Toolbox als auch eine Beschreibung der Darstellung als SVG erstellt werden.

6.4.3. Weitere Umsetzungen von BPMN-Elementen beim Petrinetz-Mapping

Um weitere BPMN-Elemente in dem Mapping auf das Petrinetz zu berücksichtigen, sind Änderungen an der Implementierung im Backend erforderlich. Dazu kann, je nach Ähnlichkeit des umzusetzenden BPMN-Elements zu bestehenden Umsetzungen, entweder ein aktueller Translator erweitert oder ein neuer Translator erstellt werden.

In beiden Fällen muss für das neue BPMN-Element der entsprechende Translator in der TranslatorFactory registriert werden.

6.5. Komplexitätsbetrachtungen

Da die Ausführungsdauer der Compliance Checks entscheidenden Einfluss auf die Akzeptanz der neuen Funktionen durch den Benutzer hat, wird diese im Folgenden genauer analysiert. Hierbei wird vor allem untersucht, wie sich die einzelnen Ausführungsphasen bei der Durchführung auf die Gesamtlaufzeit auswirken. Zu den Phasen gehören unter anderem das Kompilieren und das Ausführen des von Spin generierten Testprogrammes. Hierbei wird nur die Auswertung von LTL-Regeln betrachtet, da diese bereits bei kleinen Beispielen schon länger als eine Sekunde benötigen, während die Ausführungszeit der DataTransfer-Regeln auch bei größeren Modellen in der Testphase der Implementierung unter einer Sekunde betrug. Ein Großteil der Laufzeit wird durch die Kommunikation zwischen Webbrowser und Servlet-Container verursacht.

Die einzelnen Phasen der LTL-Verifikation

Da die Verteilung der Laufzeit über die einzelnen Phasen Gegenstand der nachfolgenden Messungen ist, hier noch einmal die Phasen im Überblick:

Mapping BPMN auf Petrinetz Aus dem im Compliance Scope enthaltenen Ausschnitt des Prozessmodells wird ein Petrinetz generiert.

Petrinetz auf Promela Das erstellte Petrinetz wird in ein Promela-Programm umgeformt.

Übersetzung der LTL-Formel nach Promela Die als LTL-Formeln formulierten Regeln werden mit Spin in Promela übersetzt.

Promela auf C Spin erzeugt aus dem Promela-Programm C-Code.

C kompilieren Der C-Code wird kompiliert, hieraus entsteht das Testprogramm, welches die eigentliche Verifikation durchführt.

Testprogramm ausführen Das Testprogramm simuliert das System und überprüft dabei die Einhaltung der Spezifikation.

Da die Anzahl der Zustände des in Promela beschriebenen Systemmodells maßgeblich für die Ausführungszeit des von Spin generierten Testprogramms ist, wurde als Testszenario die Ausführung paralleler Kontrollflusszweige gewählt.

Parallele Gateways

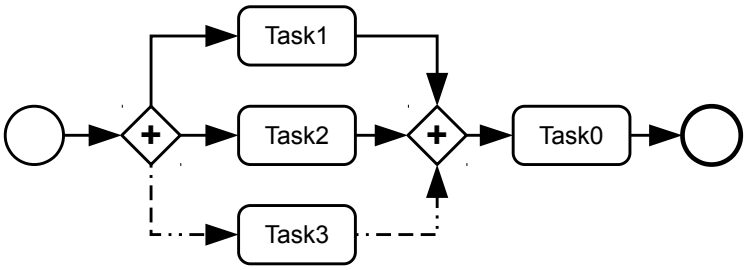


Abbildung 6.12.: Profiling paralleler Gateways - BPMN

Das hierfür verwendete Beispiel ist in Abbildung 6.12 dargestellt. Auf dem Prozessmodell wird die Regel *Finally Tasko* ausgewertet. Zwischen den beiden parallelen Gateways werden nun schrittweise weitere Kontrollflusskanten mit jeweils einem Task eingefügt. Abbildung 6.13 zeigt die entsprechende Darstellung als Petrinetz.

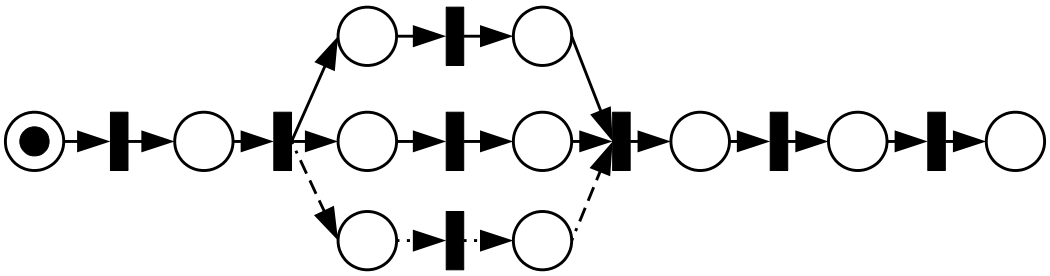


Abbildung 6.13.: Profiling paralleler Gateways - Petrinetz

Sei k die Anzahl der Zweige zwischen den beiden parallelen Gateways, dann berechnet sich die Anzahl der möglichen Zustände des Petrinetzes nach der Berechnungsvorschrift:

$$\text{Anzahl Zustände} = 5 + 2^k$$

Der zweite Summand ergibt sich aus der Tatsache, dass beim Ausführen des ersten parallelen Gateways auf jeden der parallelen Zweige ein Token gelegt wird. Sind alle Token an den

Enden des Zweigs angekommen, wird die Anzahl der im Petrinetz verfügbaren Token beim Ausführen des zweiten parallelen Gateways wieder auf eins reduziert. Während der Ausführung der parallelen Zweige kann jedes Token auf einem von zwei Plätzen liegen.

Abhängig von der Anzahl der Ausführungszweige zwischen den beiden parallelen Gateways wird die Laufzeit gemessen. Tabelle 6.1 listet die Messwerte auf.

# Zweige	1	8	14	16	18	19	20
# Zustände Petrinetz	7	216	16.389	65.541	262.149	524.293	1.048.581
# Zustände intern	16	2.574	262.158	1.179.662	$5 * 10^6$	$11 * 10^6$	$23 * 10^6$
BPMN $\rightarrow$ Petrinetz	0	0	0	0	0	0	0
Petrinetz $\rightarrow$ Promela	0	0	0	0	10	0	10
LTL $\rightarrow$ Promela	50	50	40	60	40	50	221
Promela $\rightarrow$ C	90	100	100	100	100	110	250
Kompilieren	1.021	1.042	1.111	1.162	1.192	1.292	1.582
Testprogramm	80	90	1.042	4.907	24.635	52.716	123.017
Gesamtlaufzeit	1.251	1.362	2.343	6.259	26.337	54.188	124.849

Tabelle 6.1.: Auswertung Profiling paralleler Gateways, Laufzeiten in Millisekunden

Testaufbau und Testsystem

Um möglichst praxisnahe Werte zu erhalten wird der Test auf Frontendseite durchgeführt. Dazu wird das Compliance Plugin um die Profiling-Funktionalität erweitert. Hierbei wird ein vorbereitetes Grundmodell verwendet, das dann per JavaScript um die gewünschte Anzahl von weiteren Tasks und Sequenzflusskanten erweitert wird.

Die weitere Verarbeitung erfolgt dann genau wie bei einem tatsächlichen Compliance Check. Das LTL-Modell für die Regel *Finally Tasko* wird nachgeladen und mitsamt dem BPMN-Model an den Server gesendet. Die Ausführungszeiten lassen sich aus dem erstellten Log in Tomcat ablesen. Sowohl Beginn und Ende der Abarbeitung des Requests als auch die Startzeitpunkte der Phasen werden auf Millisekundenbasis festgehalten.

Die Messungen werden auf einer mit Oracle VirtualBox [Orab] virtualisierten Windows XP-Installation unter Ubuntu 10.04 durchgeführt. Diese läuft auf einem Pentium M 735 mit 1,7 GHz und 1 GB der virtuellen Maschine zugewiesenem Hauptspeicher. Damit sich das System auf den Verifikationsprozess einstellen konnte, werden zunächst alle Tests einmal durchgeführt und deren Ergebnisse verworfen. Anschließend wird jeder Testfall dreimal durchgeführt und der Median ermittelt.

Zusammenfassung

Maßgeblich für die Laufzeit beim Überprüfen von LTL-Regeln ist die Anzahl der Zustände, die das Petrinetz annehmen kann, welches aus dem BPMN-Diagramm erzeugt wird. Weniger relevant dagegen ist die Anzahl der Plätze im Petrinetz.

Eine hohe Anzahl von Zuständen entsteht bei parallel abgearbeiteten Ausführungspfaden. Ein Anwender des Prozesseditors, der mit dem dahinter liegenden Verifikationsprozess nicht vertraut ist, kann nicht nachvollziehen, wie sich verschiedene Prozessmodellvarianten auf die Laufzeit beim Compliance Check auswirken. Ihm kann aber zumindest auf den Weg gegeben werden, Parallelität nicht unbegründet einzusetzen.

7. Zusammenfassung und Ausblick

Im letzten Kapitel dieser Arbeit wird eine Zusammenfassung der Arbeit gegeben. Anschließend folgt ein Ausblick auf weiterführende Themen im Zusammenhang mit der erarbeiteten Lösung.

7.1. Zusammenfassung

In einer vorhergehenden Arbeit [Köt10] wurde der webbasierte Prozesseditor Oryx um die Möglichkeit der Definition von Prozesstemplates erweitert. Prozesstemplates erlauben die Definition abstrakter Prozesse, die dann zu konkreten, auf die aktuellen Anforderungen angepassten, Prozessvarianten abgeleitet werden.

Die vorliegende Arbeit erweitert nun das Konzept der Prozesstemplates um ein Konzept zur Durchsetzung von Compliance in Geschäftsprozessen. Dazu werden zunächst die notwendigen Grundlagen gelegt und bestehende Ansätze zur Compliance Überprüfung präsentiert. Außerdem wird die Arbeit, auf der die Erweiterung aufgebaut wird, erläutert.

Das Konzept berücksichtigt sowohl die Prüfungen des Kontrollflusses als auch des Datenflusses. Anforderungen an den Kontrollfluss werden mittels temporaler Logik formuliert, zum Einsatz kommt hier die LTL. Für die Formulierung von temporalen Anforderungen wurde eine eigene graphische Notation geschaffen. Die Einhaltung dieser Anforderungen wird mittels Model Checking überprüft. Dazu wird erläutert, wie die BPMN-Diagramme und die LTL-Formeln in der Eingabesprache des Model Checkers, Promela, umgesetzt werden.

Bei den Datenflussregeln wird ausgehend von Datenobjekten beschrieben, welche Daten in Datenobjekte geschrieben und welche Daten aus ihnen gelesen werden dürfen. Die Beschreibung sowohl der transferierten Daten als auch der durch die Regeln erfassten Daten erfolgt dabei durch XPath-Ausdrücke.

Die Compliance-Regeln eines Compliance Scopes werden in einem Regelbaum zusammengefasst, der die logische Verknüpfung von Daten- und Kontrollflussregeln erlaubt.

Das erarbeitete Konzept wird exemplarisch im webbasierten Prozesseditor Oryx umgesetzt. Zur Sequenzanalyse wird der Model Checker Spin eingebunden, die Verifikation der Datenflussregeln erfolgt mittels Saxon. Die Verifikation der Regeln findet dabei grundsätzlich im Backend statt, das Frontend stellt Editoren für die Regeltypen und den Regelbaum bereit und bereitet außerdem das beim Verifikationsprozess ermittelte Ergebnis für den Anwender auf.

7.2. Ausblick

Während der Konzeption und Implementierung wurden einige Teilaspekte nicht oder nicht vollständig berücksichtigt. Diese befassen sich mit Performanceoptimierungen, der Erweiterung der Ausdrucksmächtigkeit von Compliance-Regeln und Verbesserungen der Usability. Auf sie wird im Folgenden genauer eingegangen.

7.2.1. Caching und Performance

Um die Ausführungszeit der Compliance Checks weiter zu verkürzen, kann das Caching noch an einigen Stellen erweitert beziehungsweise verbessert werden.

Hat der Benutzer einen Compliance Scope seit dem letzten Compliance Check nicht geändert, so kann durch das Vorhalten von Compliance Ergebnissen Zeit eingespart werden, denn diese können beim nächsten Compliance Check weiter genutzt werden. Das Einsparergebnis hängt hierbei aber sehr vom Benutzer ab. Ein Benutzer, der zwischen zwei Compliance Checks nur wenige Änderungen durchführt, profitiert hier deutlicher, als ein Benutzer, der viele Änderungen durchführt und damit wahrscheinlicher Änderungen am Compliance Scope durchgeführt hat, bevor er den nächsten Compliance Check anstößt.

Werden mehrere LTL-Formeln auf demselben Compliance Scope geprüft, ändert sich die Beschreibung des Modells nicht. In Spin werden aber die LTL-Formel und die Systembeschreibung zusammen in C-Code übersetzt, bevor dieser kompiliert wird. Da das Kompilieren des C-Codes gerade bei kleineren Modellen einen Großteil der Laufzeit des Compliance Checks einnimmt, wäre zu prüfen, ob sich nicht mehrere LTL-Formeln mit einer Modellbeschreibung zusammen in ein einziges Verifikationsprogramm kompilieren lassen und die jeweils zu prüfende LTL-Formel per Befehlszeilenparameter ausgewählt werden kann.

In diesem Zug wäre dann weiter interessant, getrennte LTL-Formeln zusammenzufassen. Das heißt besteht der Regelbaum eines Compliance Scopes aus der Konjunktion zweier LTL-Formeln, ließen sich diese beiden auch zu einer gemeinsamen LTL-Formel zusammenfassen. Damit würde der Model Checker nur noch einmal statt mehrfach aufgerufen werden.

Einen weiteren Performancevorteil bringt das Laden der LTL-Modelle auf der Backendseite direkt aus dem Repository. Pro nachzuladendem System werden etwa 100ms benötigt. Gerade bei größeren Prozesstemplates mit vielen Compliance Scopes können sich hier Zeiteinsparungen ergeben.

7.2.2. Graphische Optimierungen

Um dem Endbenutzer das Nachvollziehen der Gegenbeispiele weiter zu erleichtern, wäre eine graphische Aufbereitung des Gegenbeispiels sinnvoll. So ließe sich die Abfolge der gewählten BPMN-Diagramm-Kanten vom Servlet zurückgeben und diese dann entsprechend in die Darstellung auf Clientseite durch Overlays, wie sie momentan schon zur farblichen

Kennzeichnung der Compliance Scopes nach einem Compliance Check eingesetzt werden, einbauen.

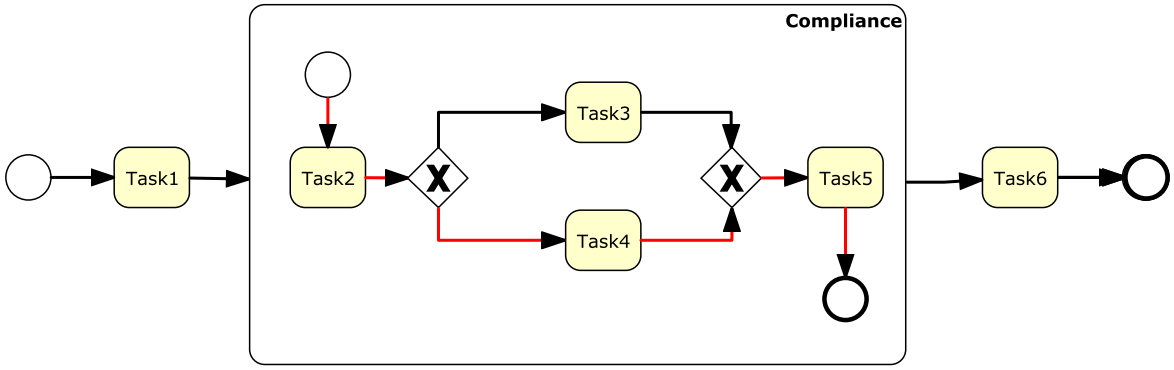


Abbildung 7.1.: Beispiel graphische Darstellung Gegenbeispiel

Eine mögliche Darstellung eines Gegenbeispiels findet sich in Abbildung 7.1. Hierbei handelt es sich wieder um das laufende Beispiel. Dem Compliance Scope wurde die Regel zugewiesen, dass sowohl *Task2* als auch *Task3* stets einmal ausgeführt werden müssen. Diese Regel wird durch den dargestellten Prozess verletzt. Der Ausführungspfad, der zur Verletzung der Regel führt, wird rot dargestellt.

7.2.3. Verbesserungen am Petrinetz-Mapping

Im Mapping des BPMN-Modells auf ein Petrinetz werden unter anderem eventbasierte Subprozesse sowie Exception Handling nicht berücksichtigt. Eine Berücksichtung von beidem führt dazu, dass das zu erstellende Petrinetz und damit das Konzept komplexer wird.

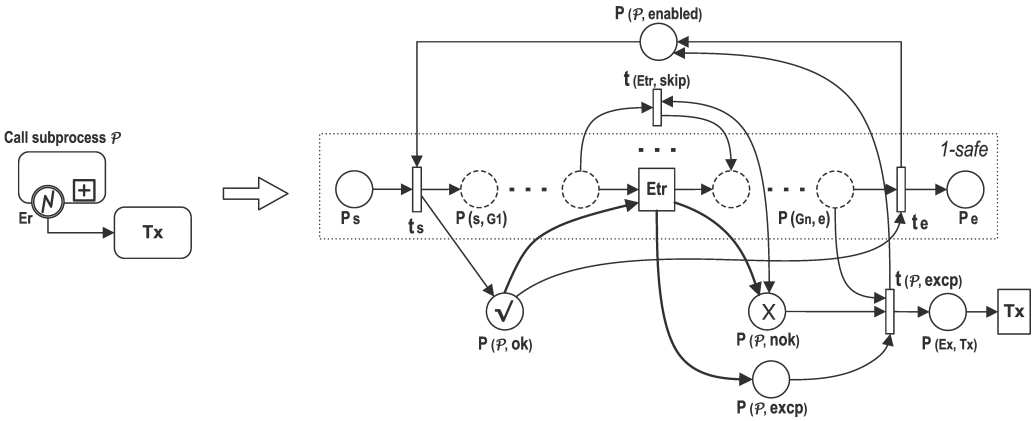


Abbildung 7.2.: Beispiel Umsetzung Exception Handling [DDO07]

Ansätze für eine Umsetzung im Petrinetz finden sich in [DDO07]. Abbildung 7.2 stellt die Petrinetzentsprechung der Fehlerbehandlung in einem Subprozess dar.

7.2.4. Erweiterung der Ausdrucksmächtigkeit der LTL-Operatoren

In der bestehenden Implementierung werden in den LTL-Formeln lediglich die Tasknamen berücksichtigt. Je nach Anwendung könnten aber auch weitere Informationen relevant sein, wie zum Beispiel, ob ein Task manuell ausgeführt wird oder als Webservice realisiert ist. Eine erweiterte LTL-Formel könnte dann wie folgt aussehen:

Finally (Task.Name=Task1 $\wedge$ Task.Implementation=Webservice).

Bei der Ausführung eines Tasks kann dann der Implementierungstyp des Tasks in einer globalen Variable abgelegt und somit in einer LTL-Formel verwendet werden.

7.2.5. Past Linear Temporal Logic

In 4.4 wurde bereits die Past Linear Temporal Logic (kurz PLTL) genannt. Diese ermöglicht durch zusätzliche Operatoren, die sich auf die Vergangenheit beziehen, die Beschreibung weiterer Regeln, die sich nicht mit der LTL ausdrücken lassen.

Ein Beispiel dafür ist die Regel, dass wenn *Task2* durchgeführt wird, in jedem Fall vorher *Task1* durchgeführt worden sein muss. Mit der LTL lässt sich zwar formulieren, dass *Task2* zwingend auf *Task1* folgen muss (hinreichend), die Rückrichtung (notwendig) dagegen nicht. Eine Formulierung in PLTL ist zum Beispiel `Task2 -> Once Task1`.

Der im Rahmen dieser Arbeit verwendete Model Checker Spin unterstützt nur die Linear Temporal Logic. In der Arbeit [PPSM03] wurde eine entsprechende Erweiterung von Spin vorgestellt.

7.2.6. Ad-hoc Compliance Checks

Grundsätzlich ist eine direkte Überprüfung der Einhaltung der definierten Regeln beim Prozessdesign wünschenswert, da der Endbenutzer hierbei zeitnah Rückmeldung bekommt, falls eine Regel verletzt wird. Da die Überprüfung einer LTL-Regel durch den Aufruf von Spin und vor allem die Kompilierung des C-Quellcodes aber schon bei einfachen Regeln mehrere Sekunden benötigt, ist die sofortige Überprüfung zumindest bei temporalen Regeln nicht praktikabel.

Bei DataTransfer-Regeln dagegen ist die Verzögerungszeit deutlich kürzer, sodass hier die zeitnahe Prüfung und Rückmeldung an den Nutzer möglich ist.

Durch den in Oryx verwendeten Event-Mechanismus kann das Compliance Plugin die relevanten Ereignisse abonnieren und sich bei deren Auftreten informieren lassen. Das heißt, Änderungen, die eine erneute Prüfung des Compliance Scopes erforderlich machen, bekommt das Plugin durch das Abonnement dieser Ereignisse mit. Da der Eventhandler an den Kontext des Plugins gebunden ist, bekommt dieses darüber auch Zugriff auf die dem Plugin bei der Initialisierung angebotene Facade und damit auf die JSON-Darstellung des

Diagramms. Außerdem lässt sich in den meisten Fällen aus den Event-Daten (event und shape) auf den Compliance Scope schließen, der nach der gerade durchgeführten Änderung erneut überprüft werden muss.

Bei einer Umsetzung müssen aber einige Punkte beachtet werden. Beim Einfügen von Datenobjekten und Datenverbindungen in das Modell werden mehrere Ereignisse ausgelöst, das heißt, neben dem Ereignis, welches den neuen Compliance Check auslöst, treten weitere Ereignisse auf. Diese müssen zunächst abgewartet werden, bevor mit dem Compliance Check begonnen werden kann. Damit muss für jede Aktion, die einen erneuten Compliance Check auslösen soll, eine geeignete Bedingung gefunden werden, die den Compliance Check anstößt.

Außerdem ist zu prüfen, wie das Plugin erkennt, ob sich der betroffene Compliance Scope prüfen lässt, ohne die Weiterarbeit am Modell zu lange zu verzögern. Ein Ansatz ist, Compliance Scopes mit LTL-Regeln nicht zu prüfen. In diesem Fall muss dem Benutzer kenntlich gemacht werden, welche Compliance Scopes in ihrem aktuellen Zustand geprüft sind und welche nicht. Dies kann mit den bekannten Overlayfarben erfolgen, die bereits zur Ergebnisdarstellung verwendet werden.

A. Anhang

A.1. Inhalt und Aufbau des beigelegten Datenträgers

Der beiliegende Datenträger ist wie folgt aufgebaut:

Ausarbeitung Der Ordner `ausarbeitung` enthält dieses Dokument im PDF-Format.

Projektverzeichnis Das Eclipse-Projekt von Oryx mit den vorgenommenen Erweiterungen findet sich im Ordner `projekt`.

Prototyp Der fertig kompilierte Prototyp findet sich im Verzeichnis `distribution`.

SVN-Patch Der Ordner `patch` enthält einen Patch, mit dem sich die Erweiterungen in einen aktuellen SVN-Checkout von Oryx einpflegen lassen.

Profiling-Patch Die Erweiterung des Compliance Plugins für die Profiling-Tests findet sich im Ordner `profiling` als Patch.

Installationsdateien Im Ordner `install` finden sich Installationsdateien für alle Software, die für den Betrieb der Lösung notwendig ist. Es sind nur Installationsdateien für Windows enthalten.

Quellen Alle als PDF-Dateien verfügbaren, verwendeten Quellen finden sich im Verzeichnis `quellen`. Sie sind nach dem im Literaturverzeichnis verwendeten Kürzel benannt.

A.2. Aufsetzen der Entwicklungsumgebung

Da die hier erarbeitete Lösung auf [Köt10] aufbaut, unterscheidet sich das Aufsetzen der Entwicklungsumgebung nur durch zusätzliche Schritte am Ende des Installationsprozesses. Zunächst sind deshalb die in [Köt10] im Anhang im Abschnitt *Installation* genannten Anweisungen zu befolgen.

Die Einrichtung der für den hier entwickelten Prototyp zusätzlich benötigten Programme, dem GCC und Spin, wird im folgenden beschrieben.

A.2.1. Installation des C-Compilers

Da der Model Checker Spin zum Kompilieren der erzeugten Testprogramme einen C-Compiler benötigt, muss dieser zunächst eingerichtet werden. Unter Windows kann dazu MinGW[1] verwendet werden.

Die gängigen Linuxdistributionen bieten in ihrer Paketverwaltungssoftware vorbereitete Pakete mit dem GCC an. Ubuntu stellt dazu das Paket `build-essential` bereit.

Anschließend muss der Pfad des Compilers unter Windows in die Systemvariable `PATH` eingetragen werden. Dazu wird in der Systemsteuerung unter System→Advanced→Environment Variables im Bereich System Variables an die vorhandene Path-Variable, bei Verwendung der Standardeinstellungen bei der Installation von MingW, der Pfad `C:\mingw\bin` angehängt werden. Unter Linux findet sich der Compiler nach der Installation unter `/usr/bin` wieder und ist damit bereits in einem Verzeichnis aus der Path-Variable.

A.2.2. Installation von Spin

Für Spin stehen vorkompilierte Dateien für Windows und Linux auf der Spin-Homepage zur Verfügung.[2]

Wie der C-Compiler auch, muss Spin unter Windows in die Path-Variable eingebunden werden. Unter Linux kann Spin entweder in ein Verzeichnis aus der Path-Variable kopiert werden oder das Verzeichnis von Spin muss zur Path-Variablen hinzugefügt werden. Da der Webbrowser unter einem eigenen Benutzeraccount läuft, muss dazu die Path-Variable des entsprechenden Benutzeraccounts angepasst werden. Informationen dazu sind der Dokumentation der entsprechenden Linux-Distribution zu entnehmen.

A.2.3. Logs

Neben den für den Benutzer bestimmten Logs (siehe Abschnitt 6.2.6), erstellt die entwickelte Lösung ausführlichere Loggingausgaben, für die Log4J verwendet wird. Unter anderem wird hier zusätzlich geloggt, welche nativen Programme mit welchen Befehlszeilenparametern ausgeführt wurden und welche Rückgaben diese geliefert haben. Damit lassen sich zum Beispiel anhand der Compilermeldungen des GCC Fehler im C-Code nachvollziehen.

Die für den Entwickler bestimmten Meldungen lassen sich in den Logs von Tomcat nachlesen. Unter Windows liegen diese normalerweise im Verzeichnis `C:\Program Files\Apache Software Foundation\Tomcat 7.0\logs`. Standardmäßig loggt Tomcat nur ab dem Info-Level. Da ein Teil der Ausgaben aber vorrangig zur Fehlersuche bestimmt ist und somit

[1] http://www.mingw.org/
[2] http://spinroot.com/spin/Bin/index.html

mit Debug-Level ausgegeben wird, kann das Logging-Level in Tomcat herabgesetzt werden. Dazu ist über Start→Programme→Apache Tomcat 7.0→Configure Tomcat die Konfigurationskonsole von Tomcat zu öffnen. Im Reiter *Logging* kann für die Einstellung *Level* dann das gewünschte Level gewählt werden. Anschließend ist die Änderung mit *OK* zu bestätigen.

A.3. Anleitung

Anhand eines Beispiels werden die wichtigsten Funktionen der Oryx-Erweiterung erläutert. Die Vorgehensweise richtet sich nach der Beschreibung unter 5.5 und setzt Grundkenntnisse im Umgang mit Oryx voraus.

Graphische Modellierung von LTL-Formeln

Eine neue Richtlinie im Unternehmen legt fest, dass Bugs nun vorevaluiert werden müssen. Dies lässt sich als LTL-Formel formulieren: `Finally Bug_vorevaluieren`. Um diese Regel zu modellieren, wird ein neues LTL-Diagramm erstellt. Dazu wird wie in Abbildung A.1 dargestellt auf der Startseite von Oryx im Dropdownmenü *Create New Model* der LTL-Diagrammtyp gewählt.

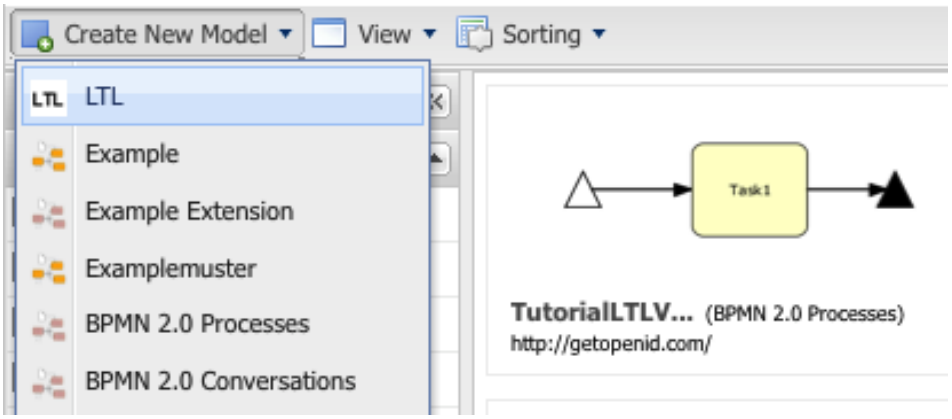


Abbildung A.1.: Auswahl des LTL-Diagrammtyps

Daraufhin öffnet sich ein neuer Diagrammeditor. Links in der Toolbox finden sich die verfügbaren Operatoren. Im ersten Schritt wird nun zunächst ein *Finally*-Operator per Drag&Drop in das Diagramm eingefügt. Anschließend wird ein *Property*-Operator, ebenfalls per Drag&Drop, in den *Finally*-Operator eingefügt. Per Doppelklick auf den *Property*-Operator lässt sich die zu prüfende Eigenschaft, in diesem Fall der Taskname *Bug_evaluieren* setzen. Das erstellte LTL-Diagramm sollte nun wie in Abbildung A.2 dargestellt aussehen. Anschließend wird das Diagramm gespeichert, zum Beispiel als *Finally_BugEvaluieren*.

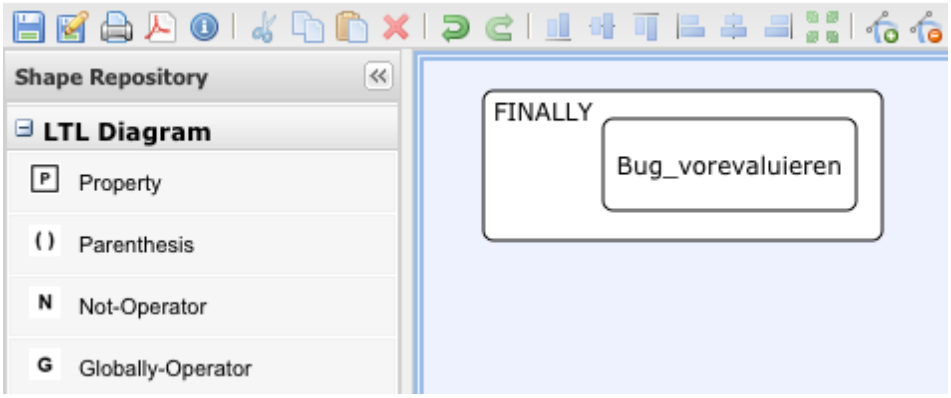


Abbildung A.2.: LTL-Diagramm der Anforderung

Definition von Prozesstemplates mit Compliance Scopes

Im nächsten Schritt erfolgt die Modellierung des Prozesstemplates für die (stark vereinfachte) Bearbeitung eines Bugs. Dazu ist wieder über die Startseite von Oryx ein neues Diagramm zu erstellen, diesmal vom Typ *BPMN 2.0 Variability*. Das Prozesstemplate ist in Abbildung A.3 dargestellt. Dieses enthält eine variable Region. An dieser Stelle können sich Varianten des Prozesses voneinander unterscheiden.

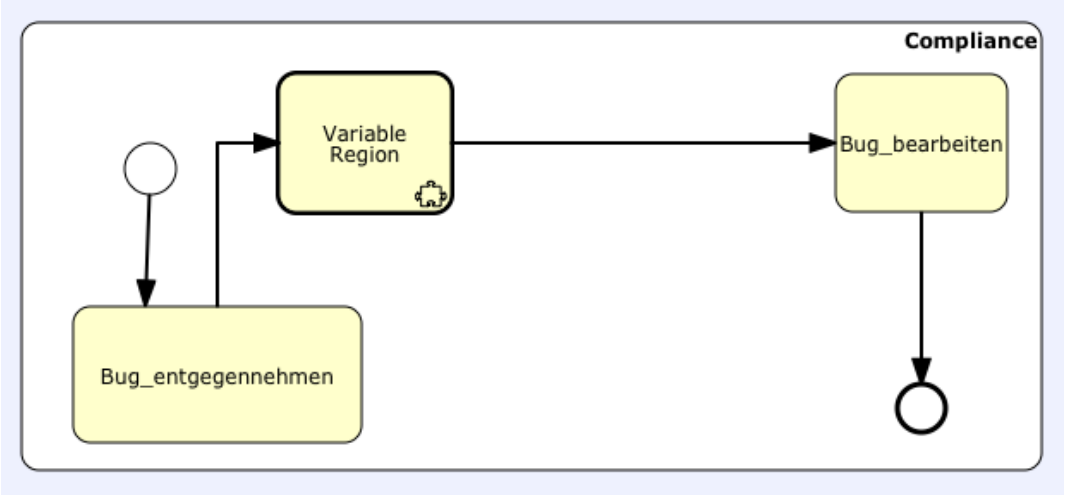


Abbildung A.3.: Prozesstemplate zur Bugbearbeitung

Die Compliance-Regel des Compliance Scopes wird mit dem Compliance Wizard festgelegt, der sich wie in Abbildung 6.8 dargestellt über die Toolbar von Oryx aufrufen lässt. Hier wird jeweils ein LTL- und ein DataTransfer-Operator angelegt, die per AND verknüpft werden (siehe Abbildung A.4). Für den LTL-Operator wird das vorhin erstellte LTL-Diagramm ausgewählt, die Eigenschaften des DataTransfer-

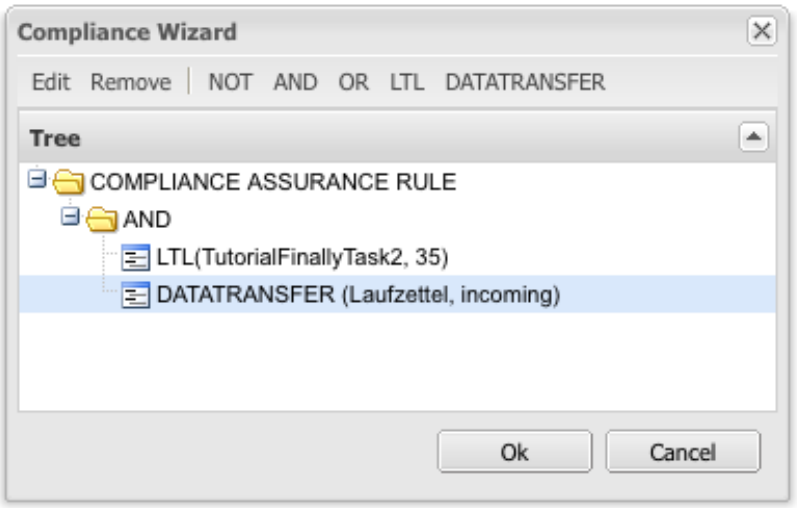


Abbildung A.4.: LTL-Diagramm der Anforderung

Abbildung A.5.: LTL-Diagramm der Anforderung

Operators werden wie in Abbildung A.5 dargestellt gesetzt, die *Path*-Eigenschaft auf `xs:element/xs:complexType/xs:sequence/xs:element[@name='from']` gesetzt.

Das erstellte Prozesstemplate wird anschließend gespeichert.

Erstellen der Prozessfragmente

Für die im Prozesstemplate definierte variable Region werden nun zwei Alternativen erstellt. Dazu werden wieder über die Startseite von Oryx zwei Diagramme vom Typ *BPMN 2.0 Fragments* angelegt. Das eine Prozessfragment sieht vor, dass der Bugreport ausgedruckt wird (siehe Abbildung A.6). Im alternativen Prozessfragment (siehe Abbildung A.7) dagegen wird der Bugreport zunächst einmal vorevaluiert, wobei das Ergebnis des Evaluierungsschrittes in einem Laufzettel festgehalten wird. Die Struktur des Laufzettels wird über den Eigenschaftseditor im rechten Teil des Oryxfensters festgelegt. Dort öffnet sich beim Klick auf den Eintrag *Properties* ein weiterer Editor. Der dortigen Liste lässt sich per *Add* ein neuer Eintrag hinzufügen. Als Name wird `schema` verwendet, im Feld *Structure* wird das Schemafragment aus Listing 5.1 hinterlegt. Anschließend ist noch festzulegen, welche Daten in den Laufzettel geschrieben werden. Dies erfolgt über die Bearbeitung der Eigenschaft *Assignments* der Datenverbindung über den Eigenschaftseditor. Mit einem Klick auf *Add* wird ein neue Zuweisungsregel erstellt, das Feld *To* wird auf `xs:element/xs:complexType/xs:sequence/*` gesetzt, *From* auf `/source/path` und *Language* auf `xpath`.

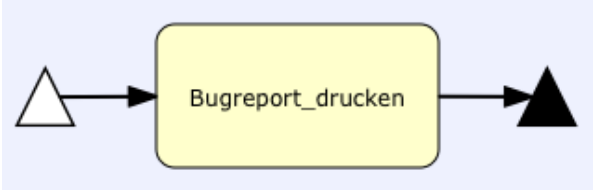


Abbildung A.6.: Erste Alternative

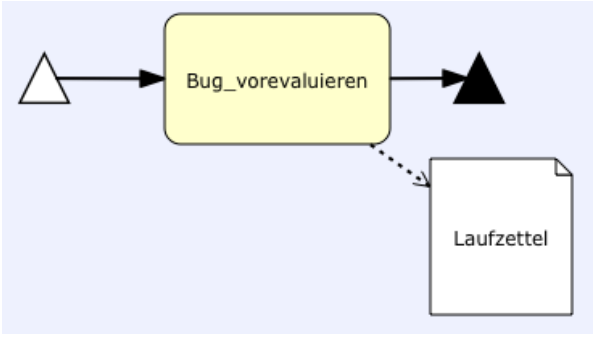


Abbildung A.7.: Zweite Alternative

Das erste Prozessfragment wird als *BugEditFragment1* abgespeichert, das zweite als *BugEditFragment2*.

Erstellen von Prozessvarianten unter Beachtung der Compliance-Regeln

Aus dem Prozesstemplate werden nun konkrete Prozessvarianten erstellt. Dazu wird das zuvor gespeicherte Prozesstemplate wieder geöffnet und mit einem Klick auf den Button

Create variant in der Toolbar mit der Prozessvariantenerstellung begonnen. Die variable Region wird nun grün dargestellt, da für diese noch keine Alternative ausgewählt wurde. Zur Auswahl der Alternative wird die variable Region durch einen Klick markiert und anschließend rechts im Eigenschaftseditor per Doppelklick die erste Alternative *BugEdit-Fragment1* ausgewählt. Da im Prozesstemplate nur eine variable Region definiert wurde, ist damit die Prozessvariantenerstellung bereits abgeschlossen.

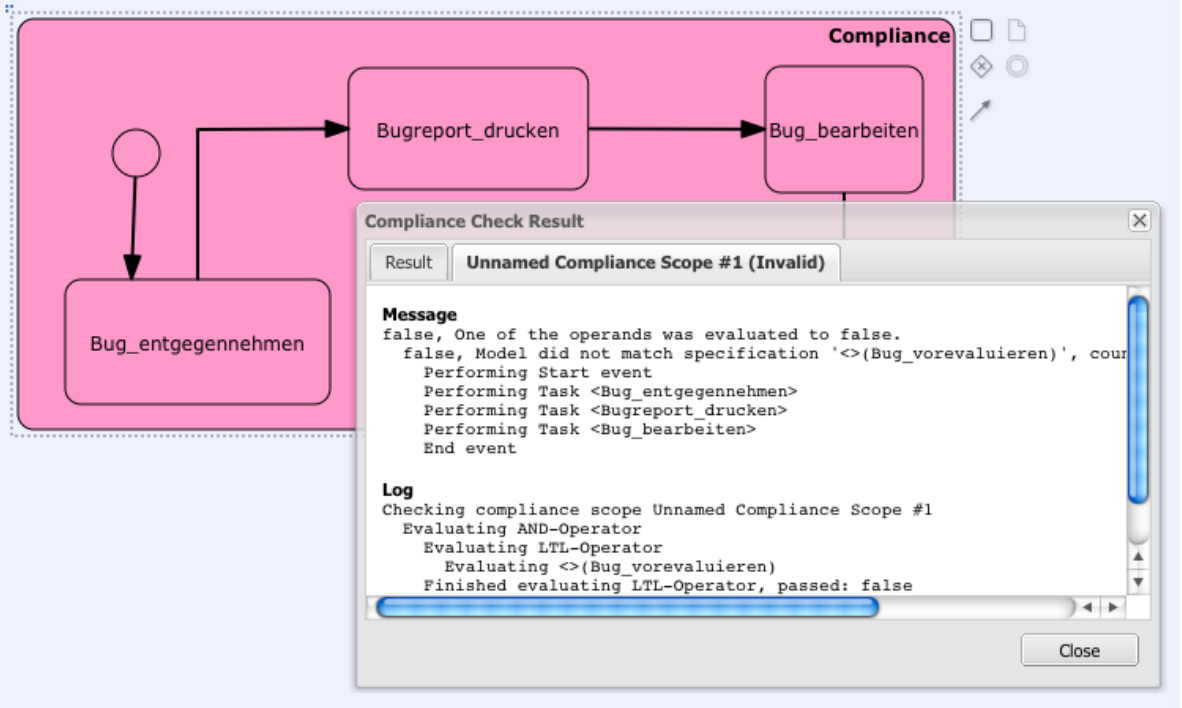


Abbildung A.8.: Verletzung der LTL-Regel beim Einsetzen des ersten Prozessfragments

Nun kann die erstellte Variante auf die Einhaltung der Compliance-Regeln überprüft werden. Dazu wird über das von der Erweiterung in der Toolbar erstellte Dropdownmenü (siehe Abbildung 6.8) ein Compliance-Check angestoßen. Nach dem durchgeführten Compliance-Check sieht das Ergebnis wie in Abbildung A.8 dargestellt aus. Der Compliance Scope wird rot hinterlegt, da die Compliance-Regel verletzt ist, denn der Task *Bug_vorevaluieren* wird im Prozess nicht ausgeführt. Im Ergebnisfenster wird neben dem Gegenbeispiel unter *Message* auch die Auswertung des Operatorenbaums unter *Log* angezeigt.

In einem weiteren Versuch wird nun das zweite Prozessfragment eingesetzt, welches die Compliance-Regel aber ebenfalls verletzt, da hier Felder des Laufzettels gesetzt werden, die von der DataTransfer-Regel nicht zugelassen sind.

Bei beiden Beispielen kann der Benutzer nachvollziehen, warum das eingesetzte Prozessfragment zur Verletzung der Compliance-Regel führte, und dann entweder ein neues Prozessfragment erstellen, welches die Regeln nicht verletzt, oder die bestehenden Prozessfragmente korrigieren.

A.4. Graphische Darstellung der generierten Petrinetze

Um die Umformung eines in JSON vorliegenden BPMN-Diagramms in ein Petrinetz zu überprüfen, lässt sich dieses in eine graphische Darstellung umwandeln. Dies erfolgt über den Export des Petrinetzes als Datei in der Petri Net Markup Language (kurz PNML) und die anschließende Konvertierung in das DOT-Format [DOT] bewerkstelligen.

In einem ersten Schritt wird statt in eine Promela-Darstellung in das PNML-Format exportiert. Eine entsprechende Export-Funktionalität ist bereits im Backend implementiert und wird auch im Frontend angeboten. Dort lässt sich zu dem markierten Compliance Scope die entsprechende PNML-Darstellung als Datei herunterladen.

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
  <net id="n1" type="http://www.pnml.org/version-2009/grammar/ptnet">
    <page id="top-level">
      <name>
        <text>An example P/T-net</text>
      </name>
      <place id="p1">
        <name>
          <text>cond1</text>
        </name>
        <initialMarking>
          <text>1</text>
        </initialMarking>
      </place>
      <place id="p2">
        <name>
          <text>cond2</text>
        </name>
        <initialMarking>
          <text>1</text>
        </initialMarking>
      </place>
      <place id="p3">
        <name>
          <text>done</text>
        </name>
      </place>
      <transition id="t1"/>
      <arc id="a1" source="p1" target="t1"/>
      <arc id="a2" source="p2" target="t1"/>
      <arc id="a3" source="t1" target="p3"/>
    </page>
  </net>
</pnml>
```

Listing A.1: Petrinetz in PNML-Notation

Listing A.1 zeigt eine Beispieldatei im PNML-Format. Darin wird eine Transition definiert, die als Eingänge zwei, jeweils mit einem Token belegte, Plätze, und als Ausgang einen

leeren Platz hat. Plätze und Transitionen werden dabei getrennt voneinander definiert und anschließend mit Kanten verbunden.

Mit dem PNML 2 dot converter[3] wird das in Listing A.1 definierte Petrinetz in das DOT-Format in Listing A.2 überführt.

```
strict digraph "n1" {
overlap=scale;
splines=true;
node[fixedsize=true];
edge[];
    subgraph "clustertop-level" {
        ordering=out;
        comment="An example P/T-net";
        color=blue;
        "p1" [shape=circle,height=.25,width=.25,comment="ready",label="1"];

        "p2" [shape=circle,height=.25,width=.25,comment="ready",label="1"];

        "p3" [shape=circle,height=.25,width=.25,comment="ready",label=""];

        "t1" [shape=box,height=.08,width=.27,comment="t1",label=""];

        "p1" -> "t1"[label=""];
        "p2" -> "t1"[label=""];
        "t1" -> "p3"[label=""];
    }
}
```

Listing A.2: Resultat im DOT-Format

Das DOT-Format ist eine Beschreibungssprache für Graphen. Dazu werden die Knoten und Kanten aufgelistet, Positionen für die einzelnen Objekte müssen dabei nicht angegeben werden. GraphViz[4] berechnet aus den gegebenen Kanten und Knoten ein Layout für den Graphen und gibt diesen als Bilddatei, beispielsweise im PNG-Format, aus.

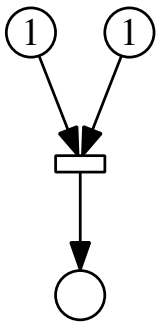


Abbildung A.9.: Darstellung des generierten Petrinetzes

[3] http://pnml.lip6.fr/pnml2dot/introduction.html
[4] http://www.graphviz.org/

A.4.1. Empfohlene Modifikationen am PNML 2 dot converter

Um die Übersichtlichkeit der Petrinetze zu erhöhen wird empfohlen, den Quellcode des PNML 2 dot converters zu modifizieren. In der Klasse `fr.lip6.move.pnml.todot.processors.PTProcessor` sind dabei die in Listing A.3 im diff-Format dargestellten Änderungen vorzunehmen.

```
-sb.append("\"" + "," + "label=\"");
+sb.append("\"" + "," + "label=\"" + workOnName(nhp));

-print("node[fixedsize=true];");
+print("node[fixedsize=false];");
```

Listing A.3: Änderungen an PBML 2 dot

Die erste Modifikation bewirkt, dass Transitionen mit den beim Mapping generierten IDs versehen werden, die zweite, dass Knoten im erzeugten Graph entsprechend ihres Inhaltes in der Größe angepasst werden.

A.5. Mapping von BPMN auf Prozesse/Channels

In dieser Arbeit wurde das Mapping von BPMN-Diagrammen auf die von Spin benötigte Promela-Darstellung mit Petrinetzen realisiert. Der zweite Ansatz mittels Prozessen und Channels wird in diesem Abschnitt für das fortlaufende Beispiel aus Abschnitt 5.2.2 illustriert.

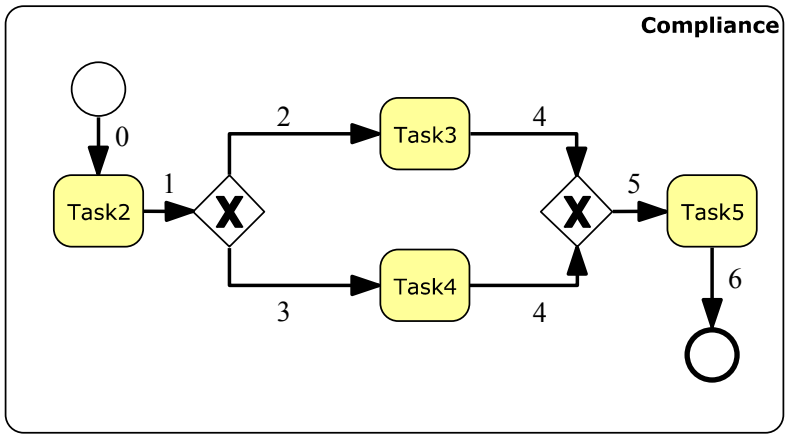


Abbildung A.10.: Laufendes Beispiel mit Channelnummern

Abbildung A.10 zeigt dazu noch einmal das laufende Beispiel. Zusätzlich sind hier die verwendeten Channelnummern eingetragen. Zunächst werden in Listing A.4 die globalen Variablen definiert. Dies sind die verwendeten Channels als Array und die Indikatoren,

welche Tasks aktuell ausgeführt werden. Außerdem werden zwei zur besseren Lesbarkeit des folgenden Codes eingeführte Makros definiert.

```
/* globale Variablen ---------------------------------------------------- */
chan rendezvous_channels[7] = [0] of { bit }

bool taskTwo = false
bool taskThree = false
bool taskFour = false
bool taskFive = false

/* Makros --------------------------------------------------------------- */
inline send(channel, token) {
  channel ! token
}

inline receive(channel, token) {
  channel ? token
}
```

Listing A.4: Umsetzung des laufenden Beispiels mittels Prozesse/Channels

Listing A.5 zeigt die Umsetzungen des Start- und des Endereignisses. Das Startereignis beginnt sofort mit der Ausführung und gibt das Ausführungssignal an den ausgehenden Channel weiter. Das Endereignis wartet am eingehenden Channel auf das Signal.

```
active proctype procStartEvent() { /* Startereignis */
end:
  send(rendezvous_channels[0], 1);
}

active proctype procEndEvent() { /* Endereignis */
end:
  receive(rendezvous_channels[6], _);
}
```

Listing A.5: Umsetzung des laufenden Beispiels mittels Prozesse/Channels

Die Implementierung der beiden exklusiven Gateways ist in Listing A.6 dargestellt. Das erste exklusive Gateway wählt abhängig von einer im Mapping vernachlässigten Bedingung einen der beiden Nachfolgepfade aus. In Promela wird dies mit dem `if`-Konstrukt umgesetzt. Da beide Bedingungen leer und damit erfüllt sind, wird bei der Ausführung nichtdeterministisch einer der beiden Nachrichtenchannels gewählt und mit dem Signal belegt.

Das zweite exklusive Gateway vereinigt die beiden Ausführungspfade wieder. Für die eingehenden Sequenzflusskanten wird derselbe Channel verwendet, über welchen entweder von *Task3* oder *Task4* das Signal zum Ausführen übermittelt wird.

```
active proctype procXorSplit() { /* XOR-Gateway */
end:
  receive(rendezvous_channels[1], _);

  if
  :: send(rendezvous_channels[2], 1)
  :: send(rendezvous_channels[3], 1)
  fi
}

active proctype procXorJoin() { /* XOR-Join */
end_loop:
  receive(rendezvous_channels[4], _);
  send(rendezvous_channels[5], 1);

  goto end_loop
}
```

Listing A.6: Umsetzung des laufenden Beispiels mittels Prozesse/Channels

Die Umsetzung eines BPMN-Tasks findet sich in Listing A.7. Der BPMN-Task wartet am eingehenden Channel auf das Signal, um mit der Ausführung zu beginnen. Anschließend werden zunächst die globalen Indikatorvariablen aktualisiert, bevor am ausgehenden Channel das Ausführungssignal weitergegeben wird.

Das Aktualisieren der Indikatorvariablen erfolgt dabei in einem einzigen Schritt. Das heißt, während der atomic-Block ausgeführt wird, findet kein Wechsel zu dem Prozess statt, der den aktuellen Zustand auf die Einhaltung der Spezifikation überprüft.

```
active proctype procTaskTwo() { /* Task 2 */
end:
  receive(rendezvous_channels[0], _);

  atomic {
    taskTwo = true;
    taskThree = false;
    taskFour = false;
    taskFive = false;
  }

  send(rendezvous_channels[1], 1);
}
```

Listing A.7: Umsetzung des laufenden Beispiels mittels Prozesse/Channels

Die Implementierung der Tasks drei bis fünf erfolgt entsprechend.

Literaturverzeichnis

[AS] A. Awad, S. Sakr. QBP - A Framework for Querying Graph-Based Business Process Models. http://bpmnq.sourceforge.net/. Zuletzt abgerufen am 23. April 2011. (Zitiert auf Seite 30)

[AWW09] A. Awad, M. Weidlich, M. Weske. Specification, Verification and Explanation of Violation for Data Aware Compliance Rules. In *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*, ICSOC-ServiceWave '09, pp. 500–515. Springer-Verlag, Berlin / Heidelberg, 2009. URL http://dx.doi.org/10.1007/978-3-642-10383-4_37. (Zitiert auf den Seiten 7 und 30)

[bas] Basic Spin Manual. http://spinroot.com/spin/Man/Manual.html. Zuletzt abgerufen am 02. Mai 2011. (Zitiert auf Seite 50)

[BDSV05] M. Brambilla, A. Deutsch, L. Sui, V. Vianu. The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae. In D. Lowe, M. Gaedke, editors, *Web Engineering*, volume 3579 of *Lecture Notes in Computer Science*, pp. 233–243. Springer Berlin / Heidelberg, 2005. URL http://dx.doi.org/10.1007/11531371_70. (Zitiert auf den Seiten 8, 29, 30 und 43)

[Ber89] C. Berge. *Hypergraphs : combinatorics of finite sets*. Hypergraphs : combinatorics of finite sets, 1989. (Zitiert auf Seite 25)

[BPM10] BPMN.de. BPMNPoster. http://www.bpmb.de/index.php/BPMNPoster, 2010. Zuletzt abgerufen am 18. April 2011. (Zitiert auf Seite 16)

[CGP01] E. M. Clarke, O. Grumberg, D. Peled. *Model checking*. MIT Press, Cambridge, Mass., 2001. (Zitiert auf den Seiten 16, 17 und 28)

[Cro] D. Crockford. JSON in Java. http://json.org/java/. Zuletzt abgerufen am 01. April 2011. (Zitiert auf Seite 23)

[Cro08] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., Sebastopol, CA, 2008. (Zitiert auf Seite 23)

[DAC98] M. B. Dwyer, G. S. Avrunin, J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, FMSP '98, pp. 7–15. ACM, New York, NY, USA, 1998. URL http://doi.acm.org/10.1145/298595.298598. (Zitiert auf Seite 27)

[dat] How Oryx data management and querying works. http://code.google.com/p/oryx-editor/wiki/DataManagementImplementation. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 67)

[DDO07] R. M. Dijkman, M. Dumas, C. Ouyang. Formal Semantics and Analysis of BPMN Process Models, 2007. URL http://eprints.qut.edu.au/7115/. (Zitiert auf den Seiten 7, 29, 46, 48, 50 und 77)

[DOT] The DOT Language. http://www.graphviz.org/doc/info/lang.html. Zuletzt abgerufen am 08. April 2011. (Zitiert auf den Seiten 59 und 88)

[Eur] Europäischer Rat. Richtlinie 90/270/EWG - Arbeit an Bildschirmgeräten. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31990L0270:DE:HTML. Zuletzt abgerufen am 09. April 2011. (Zitiert auf Seite 12)

[eva] eval Core Function. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Eval. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 23)

[exc] XPath 2.0 Expression Syntax. http://saxon.sourceforge.net/saxon7.9.1/expressions.html#except. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 41)

[ext] Ext JS - Cross-Browser Rich Internet Application Framework. http://www.sencha.com/products/extjs/. Zuletzt abgerufen am 07. April 2011. (Zitiert auf Seite 64)

[Fow] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 55)

[Ger97] R. Gerth. Concise Promela Reference. http://spinroot.com/spin/Man/Quick.html, 1997. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 19)

[gra] Graphviz - Graph Visualization Software. http://www.graphviz.org/. Zuletzt abgerufen am 08. April 2011. (Zitiert auf den Seiten 47 und 59)

[Gre10] Greenpeace International. Guide to Greener Electronics. http://www.greenpeace.org/international/en/campaigns/toxics/electronics/Guide-to-Greener-Electronics/, 2010. Zuletzt abgerufen am 9. Mai 2011. (Zitiert auf Seite 9)

[GT] GCC-Team. GCC, the GNU Compiler Collection. http://gcc.gnu.org/. Zuletzt abgerufen am 01. April 2011. (Zitiert auf Seite 18)

[Han10] Handelsblatt. EU installiert Finanzmarktpolizei. http://www.handelsblatt.com/politik/international/eu-installiert-finanzmarktpolizei/3545336.html, 2010. Zuletzt abgerufen am 9. Mai 2011. (Zitiert auf Seite 9)

[how] How to create a stencil set for oryx by the example of Let's Dance. http://code.google.com/p/oryx-editor/wiki/HowToCreateStencilSet. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 22)

[Ink] Inkscape Community. Inkscape Website. http://inkscape.org/. Zuletzt abgerufen am 18. April 2011. (Zitiert auf Seite 23)

[ISO] ISO 5807:1985. *Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*. ISO, Geneva, Switzerland. (Zitiert auf Seite 15)

[jso] Introducing JSON. http://json.org/index.html. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 23)

[KLRM+10] D. Knuplesch, L. T. Ly, S. Rinderle-Ma, H. Pfeifer, P. Dadam. On enabling data-aware compliance checking of business process models. In *Proceedings of the 29th international conference on Conceptual modeling*, ER'10, pp. 332–346. Springer-Verlag, Berlin / Heidelberg, 2010. URL http://portal.acm.org/citation.cfm?id=1929757.1929789. (Zitiert auf Seite 30)

[Koc09] S. Koch. *JavaScript - Einführung, Programmierung und Referenz - inklusive Ajax*. dpunkt.verlag GmbH, Heidelberg, 2009. (Zitiert auf Seite 63)

[Köt10] F. Kötter. *Prozessvarianten in unternehmensübergreifenden Servicenetzwerken*. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3046&engl=1. (Zitiert auf den Seiten 7, 11, 31, 45, 53, 54, 75 und 81)

[Lit08] M. Little. BPMN 2.0 Virtual Roundtable Interview. http://www.infoq.com/articles/bpmn-2, 2008. Zuletzt abgerufen am 23. April 2011. (Zitiert auf Seite 12)

[ltl] Promela Reference - LTL. http://www.spinroot.com/spin/Man/ltl.html. Zuletzt abgerufen am 08. April 2011. (Zitiert auf den Seiten 20 und 39)

[Mü10] J. Müller. *Modellierung und Auswertung musterbasierter Bedingungen an Geschäftsprozessmodelle*. Ph.D. thesis, Fakultät für Informations- und Kognitionswissenschaften, Eberhard-Karls-Universität Tübingen, 2010. URL http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/DissMueller.pdf. (Zitiert auf Seite 29)

[Now09] D. Nowotka. Networks and Processes. http://www.fmi.uni-stuttgart.de/szs/teaching/ws0809/nets/, 2009. Vorlesungsfolien WS 2008/09. (Zitiert auf den Seiten 16 und 18)

[Obj09] Object Management Group. Business Process Model and Notation (BPMN) - Version 1.2. http://www.omg.org/spec/BPMN/1.2/, 2009. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 12)

[Obj10] Object Management Group. OMG Unified Modeling Language (OMG UML) Infrastructure - Version 2.3. http://www.omg.org/spec/UML/2.3/, 2010. Zuletzt abgerufen am 18. April 2011. (Zitiert auf Seite 15)

[Obj11] Object Management Group. Business Process Model and Notation (BMN) - Version 2.0. http://www.omg.org/spec/BPMN/2.0/, 2011. Zuletzt abgerufen am 08. April 2011. (Zitiert auf den Seiten 12, 15, 16, 39, 40 und 47)

[Oraa] Oracle Corporation. Java Servlet Technology. http://www.oracle.com/technetwork/java/javaee/servlet/index.html. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 23)

[Orab] Oracle Corporation. Oracle VirtualBox. http://www.virtualbox.org/. Zuletzt abgerufen am 07. Mai 2011. (Zitiert auf Seite 72)

[ory] The Oryx Project. http://bpt.hpi.uni-potsdam.de/Oryx/WebHome. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 20)

[ory11] Oryx SVN Repository. http://code.google.com/p/oryx-editor/source/browse/trunk, 2011. Zuletzt abgerufen am 18. April 2011. (Zitiert auf Seite 21)

[Pet07] N. Peters. Oryx - Stencil Set Specification, 2007. Bachelor's Thesis, Universität Potsdam. (Zitiert auf den Seiten 22 und 23)

[pnma] PNML 2 dot converter. http://pnml.lip6.fr/pnml2dot/introduction.html. Zuletzt abgerufen am 02. April 2011. (Zitiert auf den Seiten 47 und 59)

[pnmb] Pnml.org - PNML reference site. http://www.pnml.org/. Zuletzt abgerufen am 03. Mai 2011. (Zitiert auf Seite 59)

[Pol07] D. Polak. Oryx - BPMN Stencil Set Implementation, 2007. Bachelor's Thesis, Universität Potsdam. (Zitiert auf Seite 22)

[PPSM03] M. Pradella, P. S. Pietro, P. Spoletini, A. Morzenti1. Practical Model Checking of LTL with Past. In *ATVA03*. 2003. (Zitiert auf Seite 78)

[pro] Prototype JavaScript framework. http://www.prototypejs.org/. Zuletzt abgerufen am 07. April 2011. (Zitiert auf Seite 64)

[RAAM06] N. Russell, Arthur, W. M. P. van der Aalst, N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, BPMcenter.org, 2006. (Zitiert auf Seite 28)

[SALM09] D. Schleicher, T. Anstett, F. Leymann, R. Mietzner. Maintaining Compliance in Customizable Process Models. In R. Meersman, T. Dillon, P. Herrero, editors, *Proceedings of the 17th International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS 2009)*, volume 5870 of *Lecture Notes in Computer Science*, pp. 60–75. Springer Verlag, Heidelberg, 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-70&engl=0. (Zitiert auf den Seiten 7, 25 und 26)

[SALS10] D. Schleicher, T. Anstett, F. Leymann, D. Schumm. Compliant Business Process Design Using Refinement Layers. In T. D. et al. R. Meersman, editor, *accepted for publication in OTM 2010 Conferences*. Springer Verlag, Berlin / Heidelberg, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-76&engl=0. (Zitiert auf Seite 36)

[sax] The SAXON XSLT and XQuery Processor. http://saxon.sourceforge.net/. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 63)

[Sch11] J. Schiller. SVG Support. http://www.codedread.com/svg-support.php, 2011. Zuletzt abgerufen am 18. April 2011. (Zitiert auf Seite 23)

[sig] Signavio Products Overview. http://www.signavio.com/en/products/overview.html. Zuletzt abgerufen am 18. April 2011. (Zitiert auf Seite 20)

[Sil09] B. Silver. *BPMN Method and Style*. Cody-Cassidy Press, Aptos, CA, 2009. (Zitiert auf Seite 15)

[Son11] Sony Computer Entertainment Europe Limited. PSN/Qriocity Service Update. http://blog.de.playstation.com/2011/04/26/psnqriocity-service-update/, 2011. Zuletzt abgerufen am 9. Mai 2011. (Zitiert auf Seite 9)

[Spia] Spin Homepage. http://spinroot.com/spin/whatispin.html. Zuletzt abgerufen am 08. April 2011. (Zitiert auf den Seiten 18 und 46)

[spib] SpinJa - a model checker for Promela, written in Java. http://code.google.com/p/spinja/. Zuletzt abgerufen am 02. Mai 2011. (Zitiert auf Seite 46)

[svg] Scalable Vector Graphics (SVG) 1.1 (Second Edition). http://www.w3.org/TR/SVG11/. Zuletzt abgerufen am 27. März 2011. (Zitiert auf Seite 23)

[SWLS10] D. Schleicher, M. Weidmann, F. Leymann, D. Schumm. Compliance Scopes: Extending the BPMN 2.0 Meta Model to Specify Compliance Requirements. In *Proceedings of SOCA 2010*, pp. 1–18. IEEE Computer Society, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-93&engl=1. (Zitiert auf den Seiten 25, 34 und 35)

[The] The Apache Software Foundation. Apache log4j. http://logging.apache.org/log4j/. Zuletzt abgerufen am 08. April 2011. (Zitiert auf Seite 63)

[Tsc07] W. Tscheschner. Oryx - Dokumentation, 2007. Bachelor's Thesis, Universität Potsdam. (Zitiert auf den Seiten 21 und 22)

[VF07] C. Vaz, C. Ferreira. Towards Automated Verification of Web Services. In *Proceedings of the IADIS International Conference on WWW/Internet*. Vila Real, Portugal, 2007. (Zitiert auf den Seiten 28, 29 und 46)

[WMM09] C. Wolter, P. Miseldine, C. Meinel. Verification of Business Process Entailment Constraints Using SPIN. In F. Massacci, S. Redwine, N. Zannone, editors, *Engineering Secure Software and Systems*, volume 5429 of *Lecture Notes in Computer Science*, pp. 1–15. Springer Berlin / Heidelberg, 2009. URL http://dx.doi.org/10.1007/978-3-642-00199-4_1. (Zitiert auf den Seiten 29 und 46)

[Wol10] C. Wolter. *A Methodology for Model-Driven Process Security*. Ph.D. thesis, Hasso-Plattner Institute for IT Systems Engineering, 2010. URL http://www.hpi.uni-potsdam.de/forschung/publikationen/dissertationen/dissertation_christian_wolter.html. (Zitiert auf den Seiten 29, 46 und 50)

[xpa] XML Path Language (XPath) 2.0 (Second Edition). http://www.w3.org/TR/xpath20/. Zuletzt abgerufen am 08. April 2011. (Zitiert auf den Seiten 40 und 41)

[YMH+06] J. Yu, T. Manh, J. Han, Y. Jin, Y. Han, J. Wang. Pattern Based Property Specification and Verification for Service Composition. In K. Aberer, Z. Peng, E. Rundensteiner, Y. Zhang, X. Li, editors, *Web Information Systems – WISE 2006*, volume 4255 of *Lecture Notes in Computer Science*, pp. 156–168. Springer Berlin / Heidelberg, 2006. (Zitiert auf den Seiten 8 und 27)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Stefan Grohe)