Institut für Architektur von Anwendungssystemen (IAAS)

Abteilung Workflow Applications

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit Nr. 3111

Modellierung regelkonformer Prozesse mit Compliance Scopes

Robert Heinz

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Daniel Schleicher
begonnen am:	29.10.2010
beendet am:	30.04.2011
CR-Klassifikation	D.2.4, H.4.1

Inhalt

Verzeichnis der Abbildungen

Verzeichnis der Tabellen

Verzeichnis der Listings

1 Einleitung

Unternehmen stehen immer größeren Herausforderungen gegenüber, wie z.B. internationaler Konkurrenz durch die Globalisierung und immer kürzere Produktlebenszyklen. Einen Wettbewerbsvorteil gegenüber der Konkurrenz können sich die Unternehmen erarbeiten, die schnell und flexibel auf Marktveränderungen reagieren können[1].

Erschwerend kommt hinzu, dass unter anderem aufgrund von Finanzskandalen in der Vergangenheit, eine Fülle von Gesetzen und Regelungen verabschiedet wurden, um weitere Skandale und Betrügereien zu verhindern. Damit diese Gesetze eingehalten werden, führen immer mehr, v.a. international agierende Unternehmen, sogenannte Compliance Abteilungen ein, die sicherstellen sollen, dass die Gesetze des jeweiligen Landes beachtet werden[2].

Mit Compliance ist hauptsächlich das Einhalten von Gesetzen gemeint, aber es werden auch von Unternehmen selbst auferlegte Regeln, wie z.B. das allgemeine Auftreten gegenüber Kunden und auch Kundenerwartungen, wie z.B. Green IT, zu Compliance gezählt[3][4].

Ein gutes und effizientes Geschäftsprozessmanagement ist ein geeignetes Mittel für Unternehmen, sich diesen wachsenden Herausforderungen zu stellen. Daher gibt es auch immer mehr Tools, mit dem sich Geschäftsprozesse erstellen und ändern lassen, um auf diese Weise die Unternehmen beim Geschäftsprozessmanagement zu unterstützen[5].

1.1 Aufgabenstellung

In der Diplomarbeit „Modellierung von regeltreuen Geschäftsprozessen anhand von Compliance Templates“ von Bernd-Simon Dengel wurden Compliance Templates herangezogen, die den Prozessdesigner beim Erstellen von Geschäftsprozessen unterstützen sollen.

Mit Compliance Templates sind Prozesse gemeint, die Aktivitäten enthalten, die im Prinzip als Platzhalter fungieren, so dass diese Prozesse als Vorlagen verwendet werden können. Es ist vergleichbar mit einer Korrespondenzvorlage, in der ein fester Text hinterlegt ist, aber auch Platzhalter für z.B. Adresse, Anrede oder Unterschrift enthält, welche dann separat vom jeweiligen Bearbeiter ausgefüllt werden.

Mit Compliance Templates können entsprechend Prozessvorlagen erstellt werden, die dann im Detail von den einzelnen Unternehmen vervollständigt werden können[4].

In der folgenden Arbeit sollen Möglichkeiten erarbeitet werden, Compliance Templates um sogenannte Compliance Regeln zu erweitern, d.h. in einer Prozessvorlage sollen bestimmte Regeln hinterlegt werden können, die anschließend während der Designzeit vom System überprüft werden. Bei einem Verstoß gegen eine hinterlegte Regel soll optimaler Weise der Prozessentwickler direkt benachrichtigt werden. Das Ziel ist, dass der Prozessentwickler bei seiner Arbeit so unterstützt wird, dass er sich ganz auf die Prozessmodellierung konzentrieren kann, ohne selbst ständig überprüfen zu müssen, ob der Prozess sich regelkonform verhält oder nicht.

Die Compliance Regeln werden in einer formalen Sprache hinterlegt, damit die Regeln automatisiert überprüft werden können. Daher werden in dieser Arbeit verschiedene Sprachen untersucht, die sich für solche Regeln eignen.

Die Überprüfung der Regeln erfolgt über einen Model Checker, deswegen wird in dieser Arbeit auch erläutert, was ein Model Checker ist und wie das Model Checking prinzipiell funktioniert.

Des weiteren soll der Oryx-Editor so erweitert werden, dass Compliance Regeln für einen Geschäftsprozess hinterlegt und diese über einen geeigneten Model Checker validiert werden können.

1.2 Gliederung der Arbeit

Kapitel 1 bietet einen Überblick über die Arbeit und erläutert die Aufgabenstellung.

In Kapitel 2 werden Grundlagen der Themen besprochen, die in der vorliegende Arbeit aufgegriffen werden. Dazu gehören z.B. die BPMN Notation, temporale Logiken oder der Oryx Editor.

In Kapitel 3 wird das Prinzip des Model Checkings erklärt, zudem enthält es einen Überblick über Model Checker, die verwendet werden können.

In Kapitel 4 werden die Sprachen untersucht, mit der Compliance Regeln erstellt werden können. Dabei werden Abbildungen erarbeitet, mit denen man einen in BPMN modellierten Prozess in die untersuchte formale Sprache überführen kann.

In Kapitel 5 wird auf die Implementierung der Erweiterungen des Oryx Editors eingegangen und in Kapitel 6 wird die Arbeit zusammengefasst und es wird ein Ausblick auf weitere Themengebiete gegeben.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen einiger Themengebiete vermittelt, auf die im weiteren Verlauf der Arbeit zurückgegriffen wird.

Im Speziellen gibt es eine Einführung in den mittlerweile weit verbreiteten BPMN Standard, mit dem Geschäftsprozesse modelliert werden und es wird die Kripke Struktur und die darauf basierenden temporalen Logiken vorgestellt, welche in gängigen Model Checkern Verwendung finden.

Des Weiteren gibt es eine kurze Einführung in den Oryx Editor, der im Zuge dieser Arbeit erweitert werden soll.

2.1 Business Process Modeling Notation (BPMN)

BPMN ist eine standardisierte grafische Notation, um Geschäftsprozesse darzustellen und wurde von der Business Process Management Initiative (BPMI) entwickelt. Die Notation wurde speziell so entworfen, um auf eine einfache Weise die Abfolge von Aktivitäten und den Nachrichtenfluss zwischen verschiedenen Prozessteilnehmern eines Geschäftsprozesses beschreiben zu können[6][7].

Das Ziel war, für Personen, die direkt an Geschäftsprozessen oder an deren Entwicklung und Umsetzung beteiligt sind, eine Notation verfügbar zu machen, die leicht lesbar und verständlich ist[8].

Die BPMI ist mittlerweile in der Object Management Group (OMG) aufgegangen, welche die Weiterentwicklung des BPMN Standards fort führt. Aktuell liegt BPMN in der Version 2.0 vor[6].

BPMN besteht im Grunde aus den vier Grundkategorien Flussobjekte, verbindende Objekte, Schwimmbahnen (Swimlanes) und Artefakte[8], auf die nachfolgend kurz eingegangen werden soll.

2.1.1 Flussobjekte

Zu den Flussobjekten gehören die Kernelemente Ereignis, Aktivität und Gateway.

Ereignis

Ereignisse werden als Kreise dargestellt und es wird zwischen Start-, End- und Zwischenereignissen unterschieden. Die grafische Repräsentation kann Abbildung 1 entnommen werden, wobei das linke Ereignis ein Startereignis repräsentiert, das Ereignis in der Mitte ein Zwischenereignis und das rechte Ereignis stellt ein Endereignis dar.

Abb. 1: Ereignisse (vgl. [6])

Aktivität

Aktivitäten werden als Rechtecke mit abgerundeten Ecken dargestellt. Eine Aktivität kann dabei atomar sein, d.h. einen einzelnen Arbeitsschritt darstellen, oder einen Teilprozess, wobei die Aktivität dann mit einem Pluszeichen erweitert wird.

Abb. 2: Aktivität (vgl. [6])

Gateway

Gateways werden als Rauten dargestellt und werden verwendet, um Verzweigungen und das Zusammenlaufen von Sequenzflüssen zu modellieren(vgl. [8]). Die am häufigsten verwendeten Gateways sind in Abbildung 3 zu sehen. Von links nach rechts wären das das exklusive, inklusive und parallele Gateway.

Abb. 3: Gateways (vgl. [6])

Exklusives Gateway:

Verzweigung – der Sequenzfluss wird auf genau einen ausgehenden Pfad geleitet.

Zusammenführung – es wird auf eine der eingehenden Verbindungen gewartet, um den ausgehenden Fluss zu aktivieren.

Inklusives Gateway:

Verzweigung – der Sequenzfluss wird auf einen oder mehrere ausgehende Pfade geleitet.

Zusammenführung – es reicht aus, dass einer der bei der Verzweigung aktivierten Pfade das Gateway aktiviert. Wenn der Sequenzfluss auf mehr als einen Pfad geleitet wurde, wird auf die anderen aktivierten Pfade nicht gewartet.

Paralleles Gateway:

Verzweigung – der Sequenzfluss wird auf alle ausgehenden Pfade geleitet.

Zusammenführung – es wird auf alle eingehenden Verbindungen gewartet, bevor der ausgehende Sequenzfluss aktiviert wird.

2.1.2 Verbindende Objekte

Die Flussobjekte werden über diese verbindenden Objekte miteinander verbunden. Insgesamt gibt es drei verschiedene Typen von verbindenden Objekten: den Sequenzfluss, den Nachrichtenfluss und die Assoziation.

Sequenzfluss

Der Sequenzfluss wird mit einem durchgezogen Pfeil und ausgefüllter Pfeilspitze dargestellt. Mit dem Sequenzfluss wird die Reihenfolge, in der die Flussobjekte abgearbeitet werden, definiert.

Abb. 4: Sequenzfluss (vgl. [6])

Nachrichtenfluss

Der Nachrichtenfluss wird mit einer gestrichelten Linie und einer nicht ausgefüllten Pfeilspitze dargestellt. Er dient dazu, den Nachrichtenfluss zwischen zwei unabhängigen Prozessteilnehmern zu illustrieren.

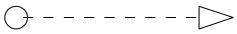


Abb. 5: Nachrichtenfluss (vgl. [6])

Assoziation

Die Assoziation wird mit einer gepunkteten Linie und einer offenen Pfeilspitze dargestellt. Mit Assoziationen werden Daten, Text und andere Artefakte mit den Flussobjekten verbunden. Sie werden dafür verwendet, um die Ein- und Ausgabe von Aktivitäten anzuzeigen.

Abb. 6: Assoziation (vgl. [6])

2.1.3 Schwimmbahn (Swimlane)

Swimlanes verwendet man, wenn Aktivitäten visuell in eigenständigen Kategorien anzeigt werden sollen. Diese Kategorien können verschiedene Unternehmen sein, die am selben Prozess teilhaben, oder auch einfach die unterschiedlichen Abteilungen eines Unternehmens. Die zwei verwendeten Konstrukte werden Pool und Bahn (Lane) genannt.

Pool

Ein Pool stellt einen eigenständigen Teilnehmer eines Prozesses dar. Die in Pools abgebildeten Prozesse sind als eigenständig anzusehen, d.h. es darf keinen Sequenzfluss zwischen einzelnen Pools geben. Man verwendet das Element des Nachrichtenflusses um die Kommunikation zwischen Prozessen in Pools abzubilden.

Abb. 7: Pool (vgl. [6])

Bahn (Lane)

Eine Bahn ist ein Teilbereich eines Pools. Mit einer Bahn werden hauptsächlich unterschiedliche Rollen bzw. Abteilungen eines Unternehmens abgebildet.

Abb. 8: Bahn (Lane) (vgl. [6])

2.1.4 Artefakte

Mit Hilfe von Artefakten kann einem in BPMN modellierten Prozess zusätzlicher Kontext hinzugefügt werden, um den Prozess selber leichter verständlich zu machen. Bisher gibt es die drei Artefakte Datenobjekt, Gruppierung und Anmerkung.

Datenobjekt

Datenobjekte werden durch ein Dokumentsymbol mit einer gefalteten Ecke dargestellt. Dem Datenobjekt kann auch ein Name und der Status, den ein Datenobjekt inne hat, hinzugefügt werden. Mit ihnen können Ein- und Ausgabeinformationen angezeigt werden, die in einem Prozess vorkommen können. Die Datenobjekte haben aber keinerlei Auswirkung auf den Prozessablauf.

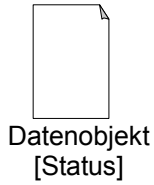


Abb. 9: Datenobjekt (vgl. [6])

Gruppierung

Eine Gruppierung wird durch ein Rechteck dargestellt, dessen Kanten abwechselnd aus Strichen und Punkten bestehen. Gruppierungen können für ein besseres Verständnis des Prozesses eingesetzt werden, indem man z.B. zusammengehörende Aktivitäten entsprechend kennzeichnet. Da Gruppierungen keinen Einfluss auf den Prozessablauf haben, können Gruppierungen z.B. auch über Poolgrenzen hinweg eingesetzt werden.

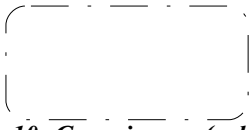


Abb. 10: Gruppierung (vgl. [6])

Anmerkung

Anmerkungen werden durch eine Grafik dargestellt, die einer öffnenden eckigen Klammer ähneln. Sie werden hauptsächlich dazu verwendet, um Zusatzinformationen zum Prozess mitzuteilen.

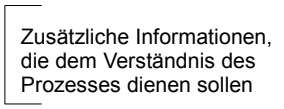


Abb. 11: Anmerkung (vgl. [6])

2.2 Kripke Struktur

Eine Kripke Struktur ist die Basisstruktur für Model Checker[9]. Einem Model Checker wird ein Modell des Systems übergeben, welches überprüft werden soll. Dieses Modell ist in einer Sprache geschrieben, die der Model Checker versteht. Aus diesem Modell generiert der Model Checker eine Kripke Struktur, gegen die die Spezifikation, die in der Regel in einer temporalen Logik (s. Kapitel 2.3) angegeben ist, geprüft wird.

Im Prinzip ist die Kripke Struktur ein Graph mit Knoten und Kanten, wobei die Knoten Zustände darstellen und die Kanten die Transitionen zwischen diesen Zuständen. Dadurch ähnelt die Kripke Struktur einem Automaten.

Im Unterschied zu einem Automaten besitzt die Kripke Struktur kein Eingabealphabet, sondern eine Menge an „Aussagen“, die in den einzelnen Zuständen gültig sind. Diese „Aussagen“ werden nachfolgend Atomic Propositions genannt, da sich dieser Terminus auch in deutschsprachiger Literatur regelmäßig wieder findet.

Ein weiterer Unterschied zu einem Automaten ist, dass es bei einer Kripke Struktur keine Endzustände gibt. Wenn man will, kann man alle Zustände als eine Art Endzustand ansehen.

Zu beachten ist, dass bei Kripke Strukturen keine Deadlock Zustände erlaubt sind, d.h. Zustände, aus denen kein Weg mehr herausführt. Sollten dennoch solche Zustände benötigt werden, dann muss diesem Zustand eine Transition hinzugefügt werden, die wieder auf diesen Zustand zeigt[10].

Die formale Definition einer Kripke Struktur ist wie folgt[10]:

Sei AP eine nicht leere Menge von Atomic Propositions. Eine Kripke Struktur ist ein vier Tupel M = {S, S_0, R, L}, mit:

S – eine Menge von Zuständen,

$S_0 \in S$ – eine Menge von Startzuständen,

$R \subseteq S \times S$ – eine totale Transitionsrelation, d.h. für jeden Zustand $s \in S$ existiert ein Folgezustand $s' \in S$ mit *R(s, s'),*

$L : S \rightarrow 2^{AP}$ – eine Beschriftungsfunktion, die jeden Zustand auf eine Menge AP von atomaren Aussagen abbildet.

Nachfolgend noch ein einfaches Beispiel einer Kripke Struktur[11]:

Sei M = {S, S_0, R, L}, mit:

S = {s_0, s_1}

s_0 ist Startzustand

R = {(s_0, s_1), (s_1, s_0)}

L(s0) = {Licht aus}

L(s1) = {Licht an}

Der zugehörige Graph ist in Abbildung 12 dargestellt.

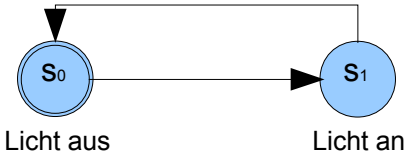


Abb. 12: Beispiel Kripke Struktur (vgl. [11])

2.3 Temporale Logik

Die Spezifikationen, die ein Model Checker gegen ein dem Model Checker übergebenes Modell überprüft, werden in der Regel in einer temporalen Logik formuliert. Die temporale Logik bietet einen Formalismus, mit der Sequenzen von Transitionen zwischen Zuständen einer Kripke Struktur beschrieben werden können.

Aussagen in der temporalen Logik werden mit Hilfe der klassischen Junktoren der Aussagenlogik, wie Konjunktion (∧), Disjunktion (∨) und Negation (¬) gebildet. Zusätzlich kommen noch temporale Quantoren und, abhängig von der verwendeten temporalen Logik, Pfadquantoren hinzu. Insgesamt gibt es fünf temporale Basisoperatoren (X, F, G, U, R) und zwei Pfadquantoren (A, E) (vgl. [12]).

In den Tabellen 1 und 2 werden die Pfadquantoren und temporalen Operatoren sowie deren Bedeutung aufgelistet. Dabei sind die temporalen Operatoren X, F und G unäre Operatoren, U und R sind binäre Operatoren.

Pfadquantor	***Bedeutung***
A	Eigenschaft ist erfüllt für alle Berechnungspfade.
E	Eigenschaft ist erfüllt auf mindestens einem Berechnungspfad.

Tabelle 1: Pfadquantoren (vgl. [12])

Operator	*Bedeutung*
X (next)	Eigenschaft ist im nächsten Zustand des Pfades erfüllt.
F (future , eventually)	Eigenschaft ist irgendwann auf dem Pfad erfüllt.
G (global, always)	Eigenschaft ist auf dem gesamten Pfad erfüllt.
U (until)	Die zweite Eigenschaft ist irgendwann auf dem Pfad erfüllt. In jedem vorherigen Zustand ist die erste Eigenschaft erfüllt.
R (release)	Die zweite Eigenschaft ist auf dem Pfad erfüllt, bis einschließlich dem Zustand, an dem die erste Eigenschaft erfüllt ist. Die erste Eigenschaft muss auf dem Pfad aber nicht unbedingt erfüllt sein, d.h. die zweite Eigenschaft wäre in dem Fall auf dem gesamten Pfad gültig.

Tabelle 2: Temporale Basisoperatoren (vgl. [12])

Nachfolgend werden die Logiken LTL und CTL genauer vorgestellt.

2.3.1 Lineare temporale Logik (LTL)

Bei der linearen temporalen Logik beschreiben die temporalen Operatoren eine Sequenz von Transitionen von einem bestimmten Zustand aus über einen einzigen Pfad, d.h. zu jedem Zeitpunkt gibt es genau einen einzigen Nachfolger (s. Abb. 13)[12][13].

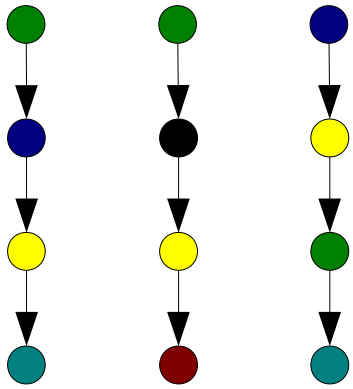


Abb. 13: Symbolischer LTL Pfad (vgl. [14])

Sei M = {S, S_0, R, L} eine Kripke Struktur und π ein Pfad in M. Ein Pfad ist eine unendliche Folge von Zuständen, π = s_0, s_1, …, so dass für jedes i ≥ 0 gilt, $(s_i, s_{i+1}) \in R$ (vgl. [12][14]). Mit π_i ist gemeint, dass der Pfad π ab dem Zustand i betrachtet wird.

- Wenn p eine atomare Aussage ist (s. 2.2 Kripke Struktur), d.h. $p \in AP$, dann ist p eine Pfadformel.
- Wenn p eine Pfadformel ist und $q \in AP$, dann sind ¬p, p ∧ q, p ∨ q, **X**p, **F**p, **G**p, p **U** q und p **R** q ebenfalls Pfadformeln.
- M, π ⊨ p bedeutet, p ist gültig in der Kripke Struktur M entlang dem Pfad π.

Die Relation $\models$ ist wie folgt definiert:

- $M, \pi \models p \quad \Leftrightarrow \quad p \in L(\pi_0)$
- $M, \pi \models \neg p \quad \Leftrightarrow$ nicht $M, \pi \models p$
- $M, \pi \models p \wedge q \quad \Leftrightarrow \quad M, \pi \models p$ und $M, \pi \models q$
- $M, \pi \models p \vee q \quad \Leftrightarrow \quad M, \pi \models p$ oder $M, \pi \models q$
- $M, \pi \models \mathbf{X}p \quad \Leftrightarrow \quad M, \pi_1 \models p$
- $M, \pi \models \mathbf{F}p \quad \Leftrightarrow \quad \exists i \geq 0 : M, \pi_i \models p$
- $M, \pi \models \mathbf{G}p \quad \Leftrightarrow \quad \forall i \geq 0 : M, \pi_i \models p$
- $M, \pi \models p \mathbf{U} q \quad \Leftrightarrow \quad \exists i \geq 0 : M, \pi_i \models q$ und $\forall j < i : M, \pi_j \models p$
- $M, \pi \models p \mathbf{R} q \quad \Leftrightarrow \quad \forall i \geq 0 : M, \pi_i \models q$ oder
  $\exists i \geq 0 : M, \pi_i \models p$ und $\forall j \leq i : M, \pi_j \models q$

Die bisher behandelten Basisoperatoren machen Aussagen über die Zukunft. Es gibt aber für LTL auch Operatoren, die Aussagen über die Vergangenheit machen. Diese Operatoren sind in Tabelle 3 zusammengefasst[15][16].

Operator	***Bedeutung***
Y (yesterday)	Eigenschaft ist im vorhergehenden Zustand des Pfades erfüllt.
O (once)	Eigenschaft war irgendwann in der Vergangenheit auf dem Pfad erfüllt.
H (historically)	Eigenschaft ist auf dem gesamten Pfad in der Vergangenheit erfüllt.
S (since)	Die zweite Eigenschaft war irgendwann in der Vergangenheit auf dem Pfad erfüllt. Seit diesem Zustand ist die erste Eigenschaft erfüllt.
T (triggered)	Die erste Eigenschaft ist irgendwann in der Vergangenheit auf dem Pfad erfüllt, ab diesem Zeitpunkt, an dem die erste Eigenschaft erfüllt ist, ist die zweite Eigenschaft auf dem gesamten Pfad erfüllt. Wenn die erste Eigenschaft in der Vergangenheit niemals erfüllt war, dann ist die zweite Eigenschaft auf dem gesamten Pfad in der Vergangenheit erfüllt.

Tabelle 3: LTL Vergangenheitsoperatoren

Für LTL mit Vergangenheitsoperatoren gelten ebenfalls die weiter oben gemachten Aussagen bzgl. Kripke Struktur, Pfad π und den Pfadformeln p und q. Für eine leichtere Darstellung der Semantik soll der Pfad π jetzt aber als eine unendliche Folge von Zuständen der Form $\pi = \ldots, s_{i-1}, s_i, s_{i+1}, \ldots$ betrachtet werden. Damit lässt sich die Relation $\models$ für die Vergangenheitsoperatoren wie folgt definieren[15][16]:

- $M, \pi \models \mathbf{Y}p \quad \Leftrightarrow \quad M, \pi_{i-1} \models p$
- $M, \pi \models \mathbf{O}p \quad \Leftrightarrow \quad \exists j \leq i : M, \pi_j \models p$
- $M, \pi \models \mathbf{H}p \quad \Leftrightarrow \quad \forall j \leq i : M, \pi_j \models p$

- M, π ⊨ p **S** q ⇔ $\exists k \leq i$: M, π_j ⊨ q und $\forall j,\ k < j \leq i$: M, π_j ⊨ p
- M, π ⊨ p **T** q ⇔ $\forall k \leq i$: M, π_k ⊨ q oder
  $\exists k \leq i$: M, π_k ⊨ p und $\forall j,\ k \leq j \leq i$:M, π_j ⊨ q

2.3.2 Computation Tree Logic (CTL)

Bei der Computation Tree Logic wird jedem temporalen Operator ein Pfadquantor vorangestellt, damit beschreibt die Logik eine Sequenz von Transitionen von einem bestimmten Zustand aus über mehrere Pfade, d.h. zu jedem Zeitpunkt gibt es mehrere mögliche Nachfolger (s. Abb. 14) [12][14].

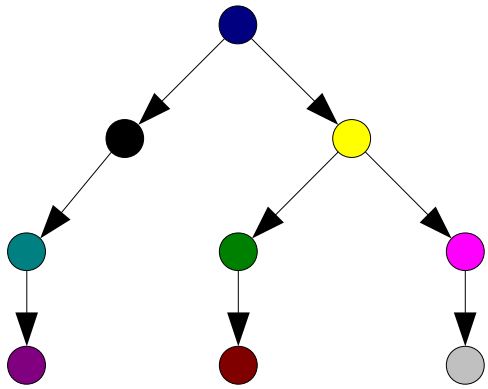


Abb. 14: Symbolischer CTL Pfad (vgl. [14])

Sei M = {S, S_0, R, L} eine Kripke Struktur und π ein Pfad in M. Ein Pfad ist eine unendliche Folge von Zuständen, π = s_0, s_1, …, so dass für jedes i ≥ 0 gilt, $(s_i, s_{i+1}) \in R$ (vgl. [12][14]). Mit π_i ist gemeint, dass der Pfad π ab dem Zustand i betrachtet wird.

- Wenn p eine atomare Aussage ist, d.h. $p \in AP$, dann ist p eine Zustandsformel.
- Wenn p eine Zustandsformel ist und $q \in AP$, dann sind ¬p, p ∧ q und p ∨ q ebenfalls Zustandsformeln.
- Wenn p eine Pfadformel ist, dann sind **E**p und **A**p Zustandsformeln.
- Wenn p eine Zustandsformel ist, dann ist p auch eine Pfadformel.
- Wenn p und q Pfadformeln sind, dann sind auch ¬p, p ∧ q, p ∨ q, **X**p, **F**p, **G**p, p **U** q und p **R** q Pfadformeln.
- M, s ⊨ p bedeutet, p ist gültig in der Kripke Struktur M im Zustand s.
- M, π ⊨ p bedeutet, p ist gültig in der Kripke Struktur M entlang dem Pfad π.

Die Relation ⊨ ist wie folgt definiert:

- M, s ⊨ p ⇔ $p \in L(s)$
- M, s ⊨ ¬p ⇔ nicht M, s ⊨ p

- M, s $\models$ p $\wedge$ q $\Leftrightarrow$ M, s $\models$ p und M, s $\models$ q
- M, s $\models$ p $\vee$ q $\Leftrightarrow$ M, s $\models$ p oder M, s $\models$ q
- M, s $\models$ **A**p $\Leftrightarrow$ $\forall \pi \in \pi(s)$: M, π $\models$ p
- M, s $\models$ **E**p $\Leftrightarrow$ $\exists \pi \in \pi(s)$: M, π $\models$ p
- M, π $\models$ **X**p $\Leftrightarrow$ M, π_1 $\models$ p
- M, π $\models$ **F**p $\Leftrightarrow$ $\exists i \geq 0$: M, π_i $\models$ p
- M, π $\models$ **G**p $\Leftrightarrow$ $\forall i \geq 0$: M, π_i $\models$ p
- M, π $\models$ p **U** q $\Leftrightarrow$ $\exists i \geq 0$: M, π_i $\models$ q und $\forall j < i$: M, π_j $\models$ p
- M, π $\models$ p **R** q $\Leftrightarrow$ $\forall i \geq 0$: M, π_i $\models$ q oder
  $\exists i \geq 0$: M, π_i $\models$ p und $\forall j \leq i$: M, π_j $\models$ q

Umformungen

In CTL gibt es durch die Kombination der Pfadquantoren und temporalen Operatoren die zehn Basisoperatoren **AX**, **EX**, **AF**, **EF**, **AG**, **EG**, **AU**, **EU**, **AR** und **ER**.

Alle diese Operatoren können durch einen Ausdruck der Form **EX**, **EG** und **EU** dargestellt werden. Nachfolgend sind die Umformungen aufgeführt, die zu den **EX**, **EG** und **EU** Ausdrücken führen[12][14]:

- **AX**p $\equiv \neg$**EX**$\neg$p
- **AF**p $\equiv \neg$**EG**$\neg$p
- **AG**p $\equiv \neg$**EF**$\neg$p
- **A**(p **U** q) $\equiv \neg$**E**($\neg$p **R** $\neg$q)
- **A**(p **R** q) $\equiv \neg$**E**($\neg$p **U** $\neg$q)

- **EF**p $\equiv$ **E**(*true* **U** p)
- **E**(p **R** q) $\equiv$ **E**(q **U** (p $\wedge$ q)) $\vee$ **Eg**q

Beispiele CTL Formeln

Nachfolgend sollen zum besseren Verständnis einige CTL Formeln grafisch dargestellt werden (vgl. [17]).

In Abildung 15 sieht man im linken Bild ein System, welches die Formel **AG**p erfüllt, d.h. p muss auf allen Pfaden immer erfüllt sein.

Im rechten Bild sieht man ein System, welches die Formel **EG**p erfüllt, d.h. p muss auf mindestens einem Pfad immer erfüllt sein.

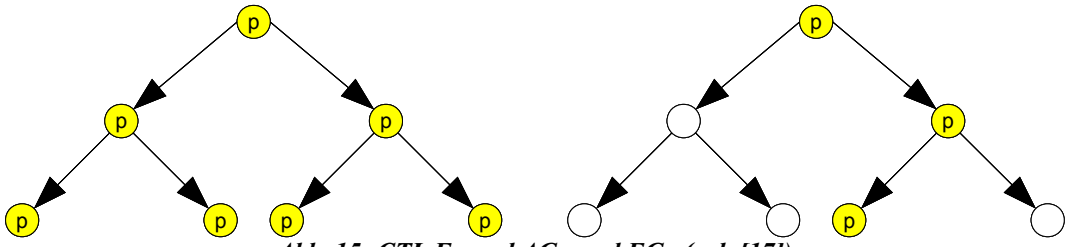


Abb. 15: CTL Formel AGp und EGp (vgl. [17])

In Abbildung 16 sieht man im linken Bild eine Darstellung der CTL Formel **AF**p, d.h. p muss auf allen Pfaden wenigstens einmal erfüllt sein.

Im rechten Bild ist die Formel **EF**p abgebildet, d.h. p muss wenigstens auf einem Pfad einmal erfüllt sein.

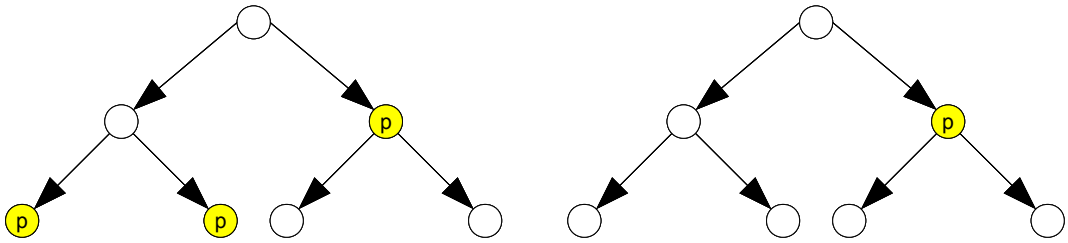


Abb. 16: CTL Formel AFp und EFp (vgl. [17])

2.4 Modallogik

Auch wenn bei Model Checkern die temporale Logik verwendet wird, um zu überprüfende Modelleigenschaften zu beschreiben, soll mit der Modallogik eine weitere Logik vorgestellt werden, die sich, zumindest bis zu einem gewissen Grad, eignet, solche Eigenschaften zu beschreiben.

In der klassischen Aussagenlogik wird einer Aussage ein Wahrheitswert zugeordnet. Die Modallogik unterscheidet sich von der klassischen Aussagenlogik in der Art, dass sie Aussagen eine Möglichkeit oder Notwendigkeit zuordnet. Damit werden Aussagen wie „Es ist möglich, dass es regnet“ oder „notwendigerweise sind alle Kreise rund“ möglich.

Dargestellt wird die Möglichkeit bzw. die Notwendigkeit in der modalen Logik mit den Zeichen ◊ und □[18].

◊p – es ist möglich, dass p

□p – es ist notwendig, dass p

Das Alphabet in der modalen Logik besteht zunächst einmal aus:[19]

- atomaren Aussagen
- der Satzkonstanten ⊥ (Falsum)
- der Implikation →
- Klammern (,)

- und dem einstelligen modalen Operator □

Mit diesem Alphabet werden die Formeln in modaler Logik erstellt, wobei eine Formel wie folgt definiert ist[19]:

- Jede atomare Aussage ist eine Formel.
- ⊥ ist eine Formel.
- Wenn A und B Formeln sind, dann ist auch (A → B) eine Formel .
- Wenn A eine Formel ist, dann ist auch □A eine Formel.

In der modalen Logik können aber auch die bekannten Operatoren der Aussagenlogik wie Negation, Konjunktion oder Disjunktion verwendet werden. Diese und weitere Operatoren wie z.B. die Äquivalenz oder ◊-Operator sind wie folgt definiert[19]:

- $\neg A := A \rightarrow \bot$
- $\top := \neg\bot$
- $A \vee B := \neg A \rightarrow B$
- $A \wedge B := \neg(A \rightarrow \neg B)$
- $A \leftrightarrow B := (A \rightarrow B) \wedge (B \rightarrow A)$
- $\Diamond A := \neg\Box(\neg A)$

Dies sollte für das Verständnis für die modale Logik, soweit sie in dieser Ausarbeitung verwendet wird, genügen. Tiefer gehende Informationen gibt es z.B. bei [20] und [19].

2.5 Oryx Editor

Der Oryx Editor stellt ein Framework zur grafischen Prozessmodellierung dar und wurde vom Hasso Plattner Institut entwickelt. Der Editor läuft komplett in einem Webbrowser, was ihn zum einen unabhängig von der Plattform, auf der er genutzt wird, macht, zum anderen ist keine Installation von zusätzlicher Software nötig. Derzeit gibt es die Möglichkeit Prozesse u.a. mit BPMN, Petri-Netzen oder EPC zu modellieren[21].

Die Modellierung der Prozesse erfolgt über sogenannte Stencils. Wenn man BPMN als Beispiel heranzieht, dann wären solche Stencils die einzelnen Elemente, mit denen man in BPMN einen Prozess modelliert, also z.B. Aktivitäten, Gateways, das Sequenzfluss Verbindungselement, usw..

Für den Anwender präsentiert sich der Editor in vier Bereiche gegliedert. Auf der linken Seite, ist das sogenannte Stencil Set sichtbar, aus welchem man die einzelnen Stencils wählen kann. Auf der rechten Seite sind die Eigenschaften zu dem gerade ausgewählten Stencil sichtbar. In der Mitte, dem größten Bereich, befindet sich die „Zeichenfläche“, auf der die Prozesse modelliert werden können und darüber befinden sich Buttons, mit denen verschiedene Funktionen, die als Plug-Ins realisiert sind, auf-

gerufen werden können, wie z.B. das Speichern eines modellierten Prozesses, Copy und Paste über die Zwischenablage, der Export des Modells in eine PDF Datei und vieles mehr.

In Abbildung 17 ist ein Screenshot des Oryx Editor abgebildet. Hier erkennt man z.B. nicht nur die Stencils auf der linken Seite, sondern auch, dass die Stencils in einzelnen Gruppen gebündelt werden können. In der Abbildung sind die Gruppen Activities und Start Events aufgeklappt.

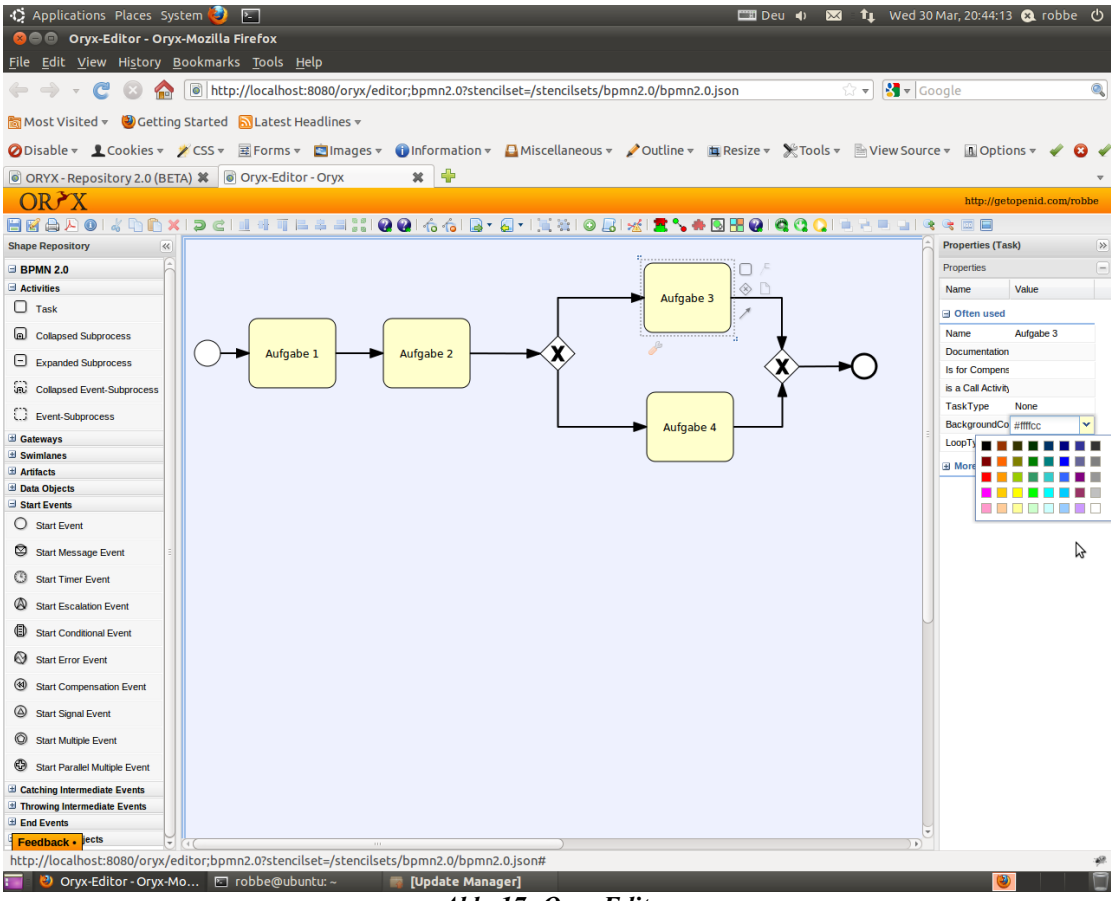


Abb. 17: Oryx Editor

Im Screenshot sind auf der rechten Seite die Eigenschaften des in der Zeichenfläche ausgewählten Elements zu sehen. Über die Eigenschaften können dann detailliertere Einstellungen für die einzelnen Elemente vorgenommen werden, es kann z.B. der Name eingegeben oder geändert werden, es kann aber auch die Hintergrundfarbe geändert werden, um ein bestimmtes Element hervorzuheben.

3 Model Checking

In diesem Kapitel soll dem Leser ein allgemeiner Überblick verschafft werden, was Model Checking ist und weshalb es eingesetzt wird. In diesem Zusammenhang wird auch auf die sogenannte „Zustandsexplosion“ eingegangen, die in Verbindung mit dem Model Checking einhergeht.

Zum Ende des Kapitels werden einige gängige Model Checker vorgestellt, die im akademischen Bereich und auch in der Industrie zum Einsatz kommen.

3.1 Einführung in das Model Checking

Hard- und Softwaresysteme sind mittlerweile ein Teil unseres alltäglichen Lebens geworden. Wir begegnen ihnen in gewöhnlichen Konsumgütern wie Fernbedienungen, Waschmaschinen und Küchengeräten, aber auch in industriellen Anlagen, Kraftfahrzeugen oder medizinischen Geräten.

Ein Ausfall dieser Systeme kann bestenfalls „nur“ ärgerlich sein, schlimmstenfalls, z.B. bei medizinischen Geräten oder in der Luftfahrt, sogar Menschenleben gefährden. Da wir in vielen Dingen inzwischen von solchen Systemen abhängig geworden sind und bei der rasanten technologischen Weiterentwicklung vermutlich sogar noch abhängiger werden, müssen wir uns auf ein korrektes Funktionieren solcher Hard- und Softwaresysteme verlassen können. Daher wird es immer wichtiger Methoden zu entwickeln, die die Korrektheit solcher Systeme überprüfen und damit einen Ausfall dieser Systeme verhindern können[12].

Gängige Praktiken bei der Validierung von Hard- und Softwaresystemen sind Simulations- und Testverfahren. Diese Verfahren sind aber fehleranfällig, zeit- und damit kostenintensiv, und zudem lassen sich auf diese Weise auch nur bestimmte Systemläufe überprüfen. Wenn bei Simulations- und Testverfahren keine Fehler gefunden werden, ist dies trotzdem noch keine Garantie, dass das System fehlerfrei ist[22]. Wie ebenfalls [22] zu entnehmen ist, wurden trotz umfangreicher Tests in zahlreichen Projekten durch das Forschungsinstitut OFFIS[1] immer noch Fehler nachgewiesen, die hauptsächlich durch lange Testsequenzen erreichbar waren.

Im Bereich sicherheitskritischer Systeme werden bereits seit geraumer Zeit formale Nachweise zur Korrektheit gefordert. Im Gegensatz zu herkömmlichen Testverfahren sind formale Korrektheitsnachweise vollständig, da nicht nur ein paar, sondern alle Systemläufe betrachtet werden. Derartige Nachweise mussten bisher durch einen Verifikationsexperten erstellt werden, was zwar den Vorteil der Vollständigkeit mitbringt, aber immer noch sehr zeit- und kostenintensiv ist[22].

Noch besser wäre es, wenn solche Nachweise automatisiert und rechnergestützt erstellt werden können. Ein Ansatz hierfür ist das Model Checking.

Model Checking ist eine automatisierte Methode zur Verifikation von endlichen zustandsbasierten Systemen. Dabei wird in der Regel eine vollständige Suche über den Zustandsraum des Systems durchgeführt, um zu bestimmen, ob eine gegebene Spezifikation von dem Modell erfüllt wird oder nicht[12].

[1] http://www.offis.de/

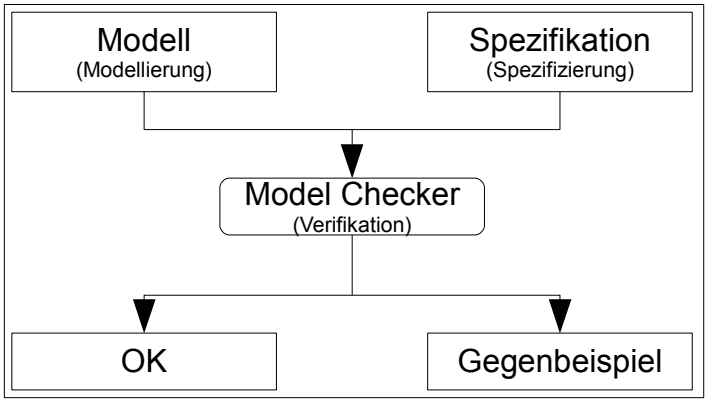


Abb. 18: Struktur eins Model Checking Systems (vgl. [23])

In Abbildung 18 ist die generelle Struktur eines Model Checking Systems dargestellt. Im Prinzip besteht das Model Checking aus den drei Phasen Modellierung, Spezifizierung und Verifikation.

Bei der Modellierung, oder genauer bei der Systemmodellierung, muss das Hard- oder Softwaresystem, welches verifiziert werden soll, in einer formalen Sprache beschrieben werden, welche vom verwendeten Model Checker verstanden wird. Hierfür werden häufig Kripke Strukturen verwendet, da sich diese gut eignen das Modell in eine formale Notation zu überführen[22][24].

Bei der Spezifikation werden Eigenschaften angegebenen, die das zu verifizierende System erfüllen soll. Die Spezifikation wird in der Regel als eine logische Formel angegeben. Bei Hard- und Softwaresystemen wird üblicherweise ein temporale Logik verwendet, da damit zeitliche Zusammenhänge spezifiziert werden können[12][24].

Viele Eigenschaften eines Systems können entweder zu dem Bereich der Sicherheits- oder der Lebendigkeitsbedingungen gezählt werden. Eine Sicherheitsbedingung wäre z.B., dass ein bestimmter Zustand nicht erreicht werden darf. Bei einer Lebendigkeitsbedingung hingegen muss ein bestimmter Zustand auf jeden Fall erreicht werden[24].

Bei der Verifikation werden nun dem Model Checker das Modell und die Spezifikation übergeben, welche dann vom Model Checker überprüft werden. Das Ergebnis ist entweder eine Bestätigung, d.h., dass die Spezifikation vom Modell erfüllt wird, oder es wird ein Gegenbeispiel geliefert. Mit Hilfe dieses Gegenbeispiels kann dann das Modell oder die Spezifikation korrigiert bzw. verbessert werden.

Ein negatives Ergebnis kann aber auch bedeuten, dass ein Fehler bei der Modellierung des Systems selber oder bei der Spezifikation gemacht wurde. Auch hier kann mit Hilfe des Gegenbeispiels der Fehler gefunden und beseitigt werden[12], was ein enormer Vorteil des Model Checking ist.

Ein weiterer Vorteil des Model Checking ist, dass es bei der Überprüfung eines Systems, im Vergleich zu einem formalen Nachweis durch einen Verifikationsexperten, sehr viel schneller ist[12].

Bei allen Vorteilen gibt es aber leider auch Nachteile, bzw. Schwierigkeiten, die dadurch zustande kommen, dass eben der gesamte Zustandsraum des Systems überprüft wird. Gerade bei nebenläufigen Systemen kann der Zustandsraum, also alle möglichen Zustände, die das System annehmen kann, exponentiell zur Modellgröße wachsen.

Dieses exponentielle Wachstum wird auch Zustandsexplosion genannt. Anhand von zwei einfachen Beispielen wird im folgenden Teilkapitel genauer erläutert, was Zustandsexplosion bedeutet.

3.1.1 Zustandsexplosion

Für die Zustandsexplosion gibt es hauptsächlich zwei Gründe:

Zum einen müssen alle möglichen Verhaltensweisen des Systems beachtet werden, d.h., es müssen alle möglichen Eingaben, die für das System möglich sind, untersucht werden, zum anderen muss mit der parallelen Ausführung von mehreren Tasks umgegangen werden.

Die parallele Ausführung von Tasks kann man als ein lineares Ausführen einzelner Aktionen dieser Tasks ansehen, die aber in einer zufälligen Reihenfolge ausgeführt werden können. Daher müssen alle möglichen Reihenfolgen einer solchen Ausführung betrachtet werden[23].

Etwas anschaulicher wird es an folgendem Beispiel, welches größtenteils der Videovorlesung des Hasso Plattner Instituts entnommen ist, die man sich komplett unter [25] anschauen kann.

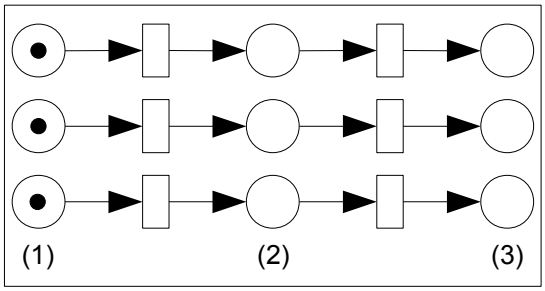


Abb. 19: Petri-Netz (vgl. [25])

Gegeben ist ein Petri-Netz mit drei parallelen Prozessen, wobei jeder Prozess einen einfachen sequentiellen Prozess mit zwei Transitionen darstellt, wie in Abbildung 19 gezeigt. Nehmen wir an, dass an der ersten Stelle der einzelnen Prozesse jeweils eine Marke ist. Dann kann man den Initialzustand, in dem sich das Petri-Netz befindet, mit „111“ bezeichnen. Da die Transitionen der einzelnen Prozesse unabhängig voneinander „feuern“ können, gibt es drei verschiedene mögliche Folgezustände, die das Petri-Netz nach dem ersten Schritt annehmen kann. Die möglichen Zustände werden in Abbildung 20 dargestellt.

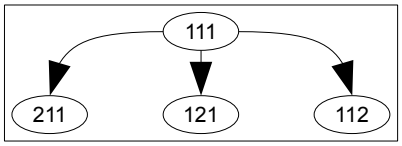


Abb. 20: Zustandsraum nach dem ersten Schritt (vgl. [25])

Da, wie gesagt, die Prozesse unabhängig voneinander sind, muss der Model Checker nun für jeden einzelnen dieser drei Zustände die nun daraus resultierenden Folgezustände berechnen, was zu einem etwas größeren Zustandsraum führt. Der Zustandsraum nach dem zweiten Schritt wird in Abbildung 21 abgebildet.

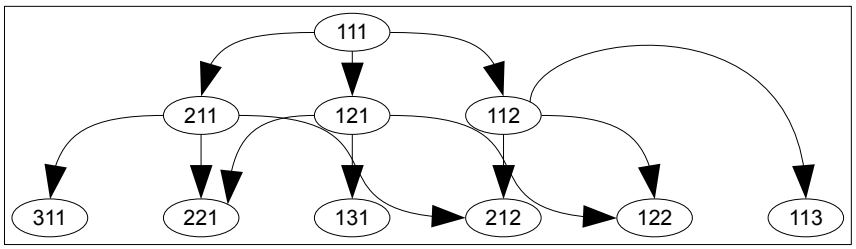


Abb. 21: Zustandsraum nach dem zweiten Schritt (vgl. [25])

Von hier aus muss der Model Checker wiederum für jeden einzelnen Zustand die daraus resultierenden Zustände berechnen und untersuchen.

Das Prinzip sollte klar sein. Daher wird, um dieses Beispiel abzuschließen, noch mit Abbildung 22 der Zustandsraum mit allen möglichen Zuständen gezeigt, die das Petri-Netz annehmen kann. Dadurch kann man auf optische Weise einen kleinen Eindruck erlangen, weshalb man von einer Zustands-“Explosion“ spricht.

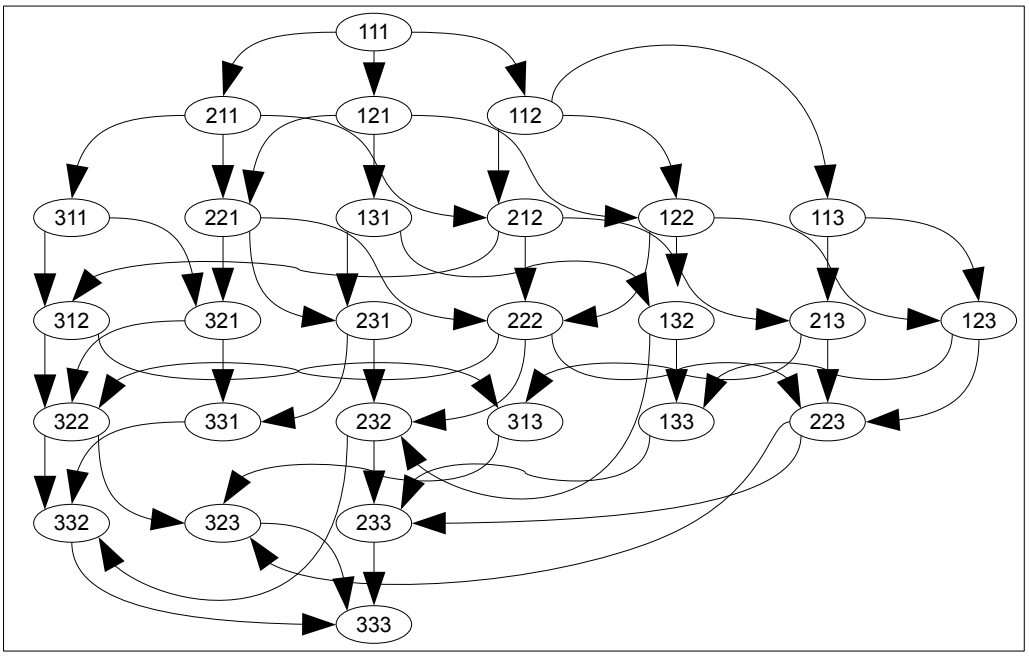


Abb. 22: Kompletter Zustandsraum (vgl. [25])

An einem weiteren, sehr simplen Beispiel, kann man einen Einblick bekommen, mit welchen Größenordnungen man es in der Praxis zu tun bekommen kann.

Angenommen man modelliert vier Warteschlangen, von denen jede Warteschlange 64 Byte speichern kann. Jedes Byte kann 256 verschiedene Werte annehmen. Damit kann

jede Warteschlange 256^{64} unterschiedliche Werte enthalten. Wenn man nun alle vier Warteschlangen betrachtet, wären das insgesamt 256^{256} verschiedene Zustände[23]. Wie groß diese Zahl ist wird richtig deutlich, wenn man mal versucht, diese mit einen Taschenrechner auszugeben:

$$32317006071311007300714876688 67 * 10^{586}$$

Diese Menge an Zuständen können nicht alle in annehmbarer Zeit komplett durchlaufen werden. Daher wurden verschiedene Techniken, wie z.B. das „Binary Decision Diagram“ oder die „Partial Order Reduction“ entwickelt, um der Zustandsexplosion Herr zu werden.

Weiterführende Informationen zur Zustandsexplosion und den genannten wie auch weiteren Techniken, um die Zustandsexplosion in den Griff zu bekommen, können u.a. in [12] nachgelesen werden.

3.1.2 Model Checking am Beispiel

In diesem Teilkapitel sollen anhand eines Beispieles die wichtigsten Schritte beim Model Checking durchgespielt werden. Das hier verwendete Beispiel, wie auch der beschriebene Suchalgorithmus stammen aus [12]. Um das Beispiel einfach zu halten, wurde jedoch eine kleine Änderung an der temporalen Formel vorgenommen, so dass diese nicht identisch mit der in [12] ist.

Im Prinzip lässt sich das Model Checking Problem wie folgt beschreiben. Suche für eine Kripke Struktur $M = \{S, S_0, R, L\}$ und für eine gegebene temporale Formel *f* alle Zustände, in denen *f* wahr ist.

Formal ausgedrückt lautet dies: $\{ s \in S \mid M,s \models f \}$

Der Suchalgorithmus, der in diesem Beispiel verwendet wird, zerlegt die temporale Formel *f* in einzelne Teilformeln und beschriftet die Zustände in denen die Teilformeln *wahr* sind. Diese beschrifteten Zustände sollen in die Menge label(s) aufgenommen werden.

label(s) = {g | g ist Teilformel von f und es gilt $M,s \models g$ } (vgl. [12],[26])

Für den ersten Schritt hieße das, dass die Zustände in denen die atomaren Teilformeln *wahr* sind, in die Menge label(s) aufgenommen werden. Mit atomarer Teilformel sind die Formelbestandteile gemeint, die nicht weiter zerlegbar sind. Bei einer Formel wie z.B. $((a \vee b) \wedge c)$ wären die atomaren Teilformeln *a*, *b* und *c*.

Mit jedem weiteren Schritt wird die nächst größere Formel abgearbeitet und der Beschriftung der Zustände in welcher sie *wahr* ist, hinzugefügt.

Es gilt $M,s \models f$ genau dann, wenn $f \in label(s)$.

In Abbildung 23 ist die Kripke Struktur einer Mikrowelle gegeben. Für das leichtere Verständnis sind nicht nur die atomaren Aussagen angegeben, die wahr sind, sondern auch diejenigen, die nicht wahr sind.

Überprüft werden soll die Formel **AG**(*Start* → **AX** *Heizen*), d.h. auf allen Pfaden soll gelten, dass *Heizen* immer ein direkter Folgezustand von *Start* ist. Diese Formel kann überführt werden in die Formel ¬**EF**(*Start* ∧ **EX**(¬*Heizen*)) (vgl. Kapitel 2). In S(g) sollen nun alle Zustände aufgeführt werden, die die Teilformel g erfüllt.

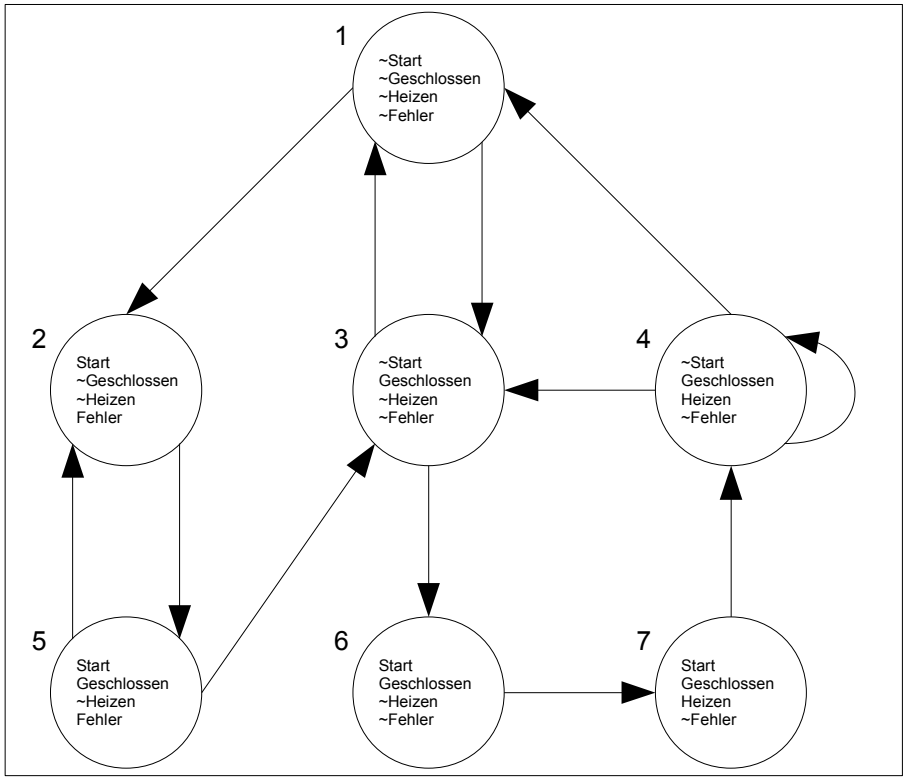


Abb. 23: Kripke Struktur einer Mikrowelle [12]

Zuerst werden die Zustände gesucht, die die atomaren Teilformeln erfüllen. Für die Teilformeln *Start* und *Heizen* wären das

S(*Start*) = {2, 5, 6, 7}.

S(*Heizen*) = {4, 7}.

Für ¬*Heizen* ergibt sich dann

S(¬*Heizen*) = {1, 2, 3, 5, 6}.

Die nächst höhere Teilformel wäre dann **EX**(¬*Heizen*). Um die Zustände zu finden, die diese Teilformel erfüllen, müssen dafür die Zustände gesucht werden, deren Nachfolger die Teilformel ¬*Heizen* erfüllen (vgl. [12]). Das ergibt in diesem Beispiel:

S(**EX**(¬*Heizen*)) = {1, 2, 3, 4, 5}.

Als Nächstes werden die Zustände für die Teilformel *Start* ∧ **EX**(¬*Heizen*) gesucht. Dazu werden die schon gefundenen Zustände der einzelnen Teilformeln herangezogen und man erhält:

S(*Start* ∧ **EX**(¬*Heizen*)) = {2, 5}.

Um nun die Zustände für **EF**(*Start* ∧ **EX**(¬*Heizen*)) zu bekommen, muss eine weitere Umformung durchgeführt werden. Der Ausdruck **EF** *f* kann umgeformt werden in: **E**[*true* **U** *f*][12].

Um die gültigen Zustände für einen Ausdruck wie **E**[f_1 **U** f_2] zu bekommen, müssen erst einmal alle Zustände, in denen f_2 gültig ist, gesucht werden. Anschließend werden mit der invertierten Transitionsrelation alle Zustände gesucht, die über einen Pfad er-

reicht werden, auf dem in jedem Zustand f_1 gültig ist (siehe [12]). Damit bekommt man:

S(**EF**(*Start* ∧ **EX**(¬*Heizen*))) = {1, 2, 3, 4, 5, 6, 7}.

Wenn man jetzt noch die Negation hinzunimmt, dann erhält man für die komplette Formel:

S(¬**EF**(*Start* ∧ **EX**(¬*Heizen*))) = Ø,

d.h. die Spezifikation **AG**(*Start* → **AX** *Heizen*) ist in keinem Zustand gültig und erfüllt daher nicht die gegebene Kripke Struktur.

3.2 Model Checker

Es existieren eine ganze Reihe an freien und kommerziellen Produkten. Model Checker werden hauptsächlich für spezielle Problembereiche entwickelt und sind daher auch nur in den für sie bestimmten Einsatzgebieten verwendbar.

In diesem Teilkapitel wird eine kleine und daher unvollständige Auswahl an Model Checkern kurz vorgestellt. Speziell zu den kommerziellen Produkten gibt es aber nicht viel Informationen, da auf den Herstellerseiten keine weiterführenden Informationen angegeben sind. Diese Model Checker sollen der Vollständigkeit halber hier aufgeführt werden und wurden über die Angaben bei [23] gefunden.

In Tabelle 4 werden die freien hier vorgestellten Model Checker mit ihren jeweiligen Eigenschaften nochmals übersichtlich dargestellt.

3.2.1 Freie Model Checker

Spin

Spin wurde 1980 an den Bell Labs für die formale Verifizierung von verteilten Software Systemen entwickelt. Seit 1991 ist Spin frei als Open Source Software erhältlich. Für die Beschreibung der Systemmodelle verwendet Spin eine höhere Programmiersprache mit dem Namen PROMELA[27].

PROMELA ist die Abkürzung für Process Meta-Language. Die Syntax der Sprache ist an C angelehnt, so dass C Programmierer sich mit PROMELA recht schnell zurechtfinden können. Die Sprache ist aber nicht für die Implementierung von Systemen gedacht, sondern für die Beschreibung von Systemen. Daher liegt das Hauptaugenmerk dieser Sprache auch auf der Modellierung und Synchronisation von nebenläufigen Prozessen und nicht auf der Berechnung von Daten. Hauptbestandteile von solchen Spin Modellen sind asynchrone Prozesse, gepufferte und ungepufferte Nachrichtenkanäle, Synchronisationsaufrufe und strukturierte Daten[28].

Die Spezifikation des Modells wird mit der temporalen Logik LTL vorgenommen. Um mit dem Problem der Zustandsexplosion umzugehen, sind in Spin verschiedene Techniken der Partial Order Reduction implementiert. Optional kann Spin auch so aufgerufen werden, dass die Verifikation mit der Binary Decision Diagram Technik durchgeführt wird[27].

SMV

SMV steht für Symbolic Model Verifier und wurde von Ken McMillan an der Carnegie Mellon University für die Verifikation von synchronen Hardwareschaltkreisen entwickelt[29].

Die Kripke Struktur des Modells wird mit Hilfe von OBDDs (Ordered Binary Decision Diagram) dargestellt. Des Weiteren kann SMV mit vielen anderen Abstraktionstechniken umgehen, wie z.B. Refinement Maps, Symmetrie oder Induktion um mit dem Problem der Zustandsexplosion umzugehen. Als Eingabesprache für die Systemmodellierung wird eine eigene SMV Eingabesprache verwendet. Die Spezifikation, gegen die das Modell geprüft wird, wird in CTL angegeben[30].

Mittlerweile gibt es noch zusätzlich Cadence SMV welche an den Cadence Berkeley Labs entwickelt wurde. Hierbei handelt es sich um eine Erweiterung des SMV Model Checkers. Im Vergleich zum ursprünglichen SMV Model Checker besitzt Cadence SMV eine mächtigere Modellierungssprache und kann zudem nicht nur mit CTL umgehen, sondern auch mit LTL[31].

NuSMV

NuSMV ist ein Open Source Model Checker, der von drei italienischen Forschergruppen und der Carnegie Mellon University entwickelt wurde. Dazu wurde der SMV Model Checker neu implementiert und erweitert.

NuSMV kombiniert BDD basiertes Model Checking mit SAT basiertem Model Checking. Als Modellierungssprache wird eine eigene NuSMV Eingabesprache verwendet, welche eine Erweiterung der SMV Eingabesprache ist. Für die Formalisierung der Spezifikation kann CTL und LTL verwendet werden[32], [33].

Seit Version 2.1 kann NuSMV auch mit temporalen Vergangenheitsoperatoren umgehen[33]. Wenn ein Geschäftsprozess z.B. in BPMN Notation vorliegt, vereinfacht LTL mit Vergangenheitsoperatoren die Formalisierung der Spezifikation, die dem Model Checker übergeben werden muss.

Übersicht über freie Model Checker

Model Checker	*Modellierungssprache*	*Temporale Logik*	*Umgang mit Zustandsexplosionsproblematik*
Spin	PROMELA	LTL	Partial Order Reduction, BDD
SMV	SMV Eingabesprache	CTL	OBDD, Refinement Maps, Symmetrie, Case Analysis, Data Type Reduction, Induktion
NuSMV	NuSMV Eingabesprache	CTL, LTL, LTL mit temporalen Vergangenheitsoperatoren	BDD, SAT

Tabelle 4: Übersicht Model Checker

Dies ist nur eine sehr kleine Auswahl über die wohl verbreitetsten Model Checker. Unter [34] sind noch eine Menge weiterer Model Checker für alle möglichen Problembereiche aufgelistet und bietet einen guten Ausgangspunkt, falls man sich weiter mit dem Thema Model Checker befassen möchte.

3.2.2 Kommerzielle Model Checker

RuleBase

RuleBase wurde von der IBM Research Lab in Haifa entwickelt und dient zur formalen Verifikation von Hardware. Es ist hauptsächlich für die Verifikation der Kontrolllogiken von großen Hardwaresystemen geeignet. Dieses Tool wird schon seit über zehn Jahren von IBM zur formalen Verifikation eingesetzt[35].

RuleBase verwendet eine auf CTL basierende Spezifikationssprache mit dem Namen Sugar[23].

FoCs

FoCs (ausgesprochen wird es wie „Fox“) ist die Abkürzung für Formal Checkers und stammt ebenfalls aus dem IBM Research Lab in Haifa.

Dieses Tool übersetzt in Sugar geschriebene Spezifikationen in eine Hardwarebeschreibungssprache wie Verilog und bindet diese in eine Simulationsumgebung ein[36].

FormalPro

FormalPro ist ein Äquivalenz Checker des Unternehmens Mentor Graphics. Dieses Tool überprüft die Äquivalenz zwischen RTL (Register Transfer Language) Beschreibungen, RTL Design und Gate Level Design[23].

4 Sprachen zur Definition von Compliance Regeln

Im folgenden Kapitel werden anhand eines Beispielprozesses einige Sprachen untersucht, um herauszufinden ob bzw. wie gut sich diese Sprachen zum Definieren von Compliance Regeln eignen.

Der Beispielprozess selber ist in BPMN modelliert und es werden für die einzelnen Sprachen Abbildungen erarbeitet, welche bestimmte grafische Elemente aus BPMN in die untersuchte Sprache überführen. Hierbei werden aber nicht für alle grafischen BPMN Elemente Abbildungen erarbeitet, sondern nur für eine Auswahl der am häufigsten verwendeten Elemente.

Im einzelnen wird eine Abbildung für die Sequenz vorgenommen, sowie Abbildungen für exklusive, inklusive und parallele Gateways.

Zu beachten ist, dass hier nur Abbildungen für den Kontrollfluss erstellt werden und keine Abbildungen für einen Datenfluss.

4.1 Beispielprozess

In Abbildung 24 ist ein fiktiver und an einigen Stellen auch vereinfachter Prozess abgebildet, wie er in einem Unternehmen bei einer eingehenden Bewerbung stattfinden könnte. Dieser Prozess soll zuerst einmal beschrieben werden.

Nach einem Bewerbungseingang im Unternehmen muss dem Bewerber als erstes eine Eingangsbestätigung zugeschickt werden. Im weiteren Entscheidungsprozess muss dann entschieden werden, ob der Bewerber eine Absage erhält, oder ob er zu einem Bewerbungsgespräch eingeladen werden soll. Sollte die Abiturnote des Bewerbers schlechter als 2.5 sein, dann soll ihm gleich abgesagt werden. Mit der Absage des Bewerbers endet der Prozess.

Sofern der Bewerber zu einem Gespräch eingeladen wurde, muss, nachdem das Gespräch stattgefunden hat, wieder eine Entscheidung getroffen werden. Entweder das Gespräch lief erfolgreich und der Bewerber wird zu einem zweiten Gespräch eingeladen oder dem Bewerber wird abgesagt.

Nach einem möglichen zweiten Gespräch gibt es wiederum zwei Möglichkeiten, wie der Prozess anschließend weiter verläuft. Entweder dem Bewerber wird abgesagt, oder es wird ein Vertrag ausgearbeitet.

Wenn ein Vertrag ausgearbeitet wird, muss diesem Vertrag vom Betriebsrat des Unternehmens und dem Personalbereich unabhängig voneinander zugestimmt werden. Eine Einstellung des Bewerbers erfolgt nur, wenn beide, der Betriebsrat und der Personalbereich, dem Vertrag zustimmen, ansonsten wird dem Bewerber abgesagt.

Bei Betrachtung dieses Prozesses ergeben sich aus dem Modell automatisch einige Regeln, die für einen korrekten Ablauf eingehalten werden müssen.

- Es muss z.B. vor allen anderen Aktionen zuerst eine Eingangsbestätigung verschickt werden.
- Ein zweites Vorstellungsgespräch darf es nur geben, wenn es vorher ein erstes Gespräch gab.

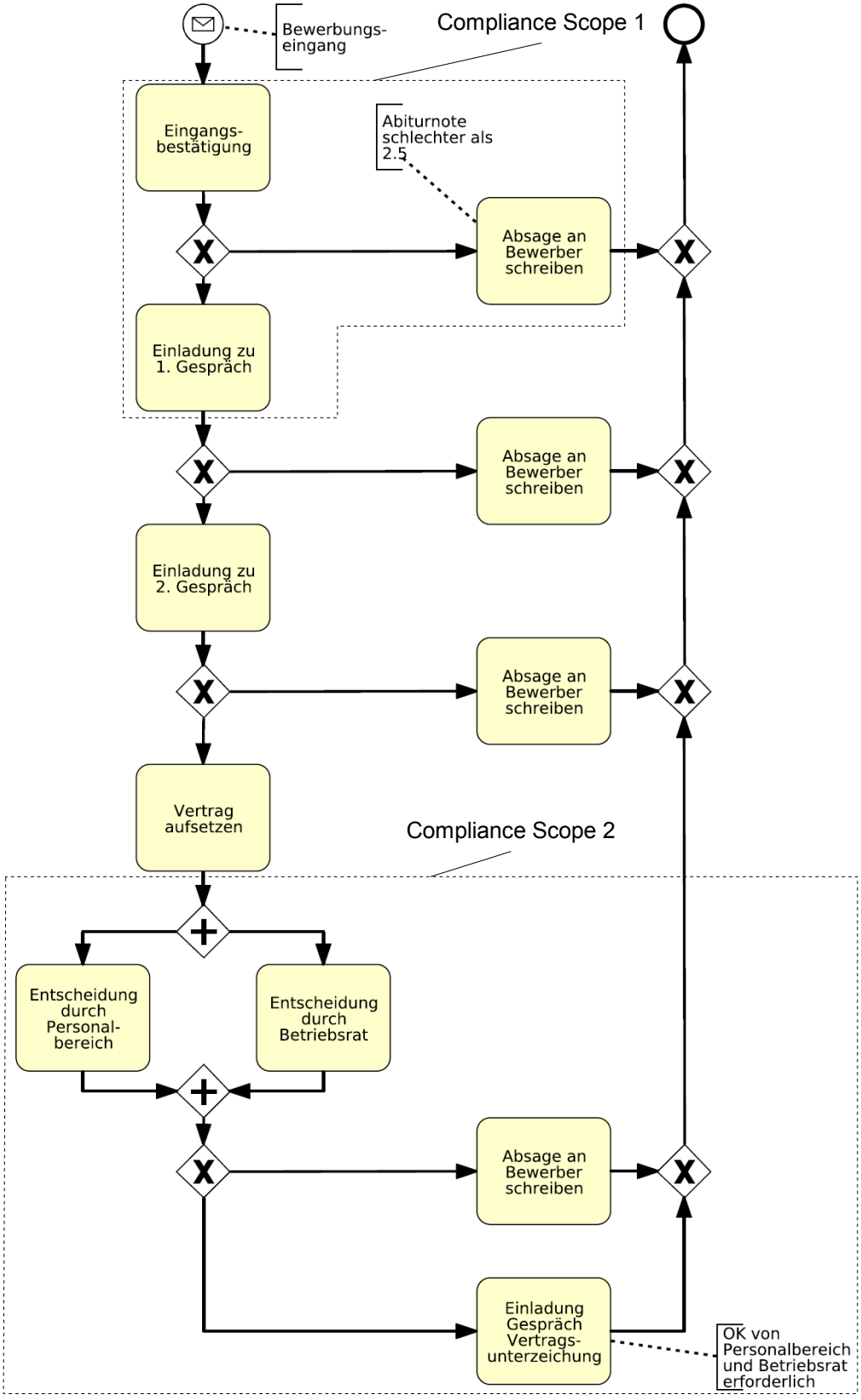


Abb. 24: Bewerbungsprozess

- Ein Vertrag darf nur aufgesetzt werden, wenn der Bewerber an zwei Bewerbungsgesprächen teilgenommen hat.
- Der Bewerber darf nur eingestellt werden, wenn Betriebsrat und Personalbereich dem erstellten Vertrag zugestimmt haben.

In Abbildung 24 sind zwei dieser Regeln als Compliance Scope hervorgehoben.

Compliance Scope 1 wäre die Regel, dass als erste Aktion eine Eingangsbestätigung verschickt werden muss. In Compliance Scope 2 wird die Regel aufgeführt, dass der Bewerber nur eingestellt werden darf, wenn Betriebsrat und Personalbereich dem Vertrag zugestimmt haben.

Nachfolgend sollen diese beiden Regeln in unterschiedlichen logischen Sprachen abgebildet werden. Dazu werden für verschiedene BMPN Elemente Abbildungen in die untersuchte Sprache erarbeitet. BPMN wurde gewählt, da im praktischen Teil der Arbeit mit BPMN modellierte Prozesse um Compliance Regeln erweitert werden. Die hier gewonnenen Erkenntnisse werden dann im praktischen Teil direkt wieder verwendet.

4.2 Lineare temporale Logik (LTL)

Dadurch, dass sich mit LTL zeitliche Abfolgen abbilden lassen, eignet sich diese Sprache gut um Prozesse abzubilden.

Für die Abbildung der grafischen BPMN Elemente auf die LTL Formeln wird auf die Arbeit in [37] zurückgegriffen, in der diese Abbildungen schon ausgearbeitet wurden.

In Tabelle 5 sind die ausgearbeiteten LTL Formeln zusammengefasst dargestellt. Im Vergleich zu [37] wurden lediglich bei den „Split Gateways“ der Vollständigkeit halber noch jeweils eine Aktivität vor das Gateway hinzugefügt und auch in der Formel abgebildet.

Für die im Beispiel des vorherigen Teilkapitel gewählten Compliance Scopes würden sich anhand der Formeln in Tabelle 5 folgende Spezifikationen ergeben, die durch einen geeigneten Model Checker zu prüfen wären.

Compliance Scope 1

LTL Formel:

(1) *(Eingangsbestätigung* **R** ¬*(***F** *Einladung zu 1. Gespräch* ⊕ **F** *Absage an Bewerber schreiben))*

Zur Erinnerung, der **R**elease Operator in der Form (*A* ***R*** *B)* besagt, dass der Ausdruck *B* bis einschließlich dem Zeitpunkt, an dem *A* wahr wird, wahr ist, danach ist der Ausdruck *B* nicht mehr wahr.

Auf die Formel (1) angewandt, bedeutet dies, dass der Ausdruck ¬*(***F** *Einladung zu 1. Gespräch* ⊕ **F** *Absage an Bewerber schreiben)* erst einmal wahr ist, d.h. weder kann

die Aktion Einladung zum ersten Gespräch stattfinden, noch ein Absage an den Bewerber geschrieben werden (beachte die Negation vor dem zweiten Ausdruck).

Nachdem eine Eingangsbestätigung erstellt wurde, wird im nächsten Schritt der zweite Ausdruck „falsch“, d.h. die Negation, die dem zweiten Ausdruck vorangeht, wird aufgehoben und jetzt kann es zu einer einer Einladung oder Absage kommen.

Auf diese Weise kann durch den Kniff mit der Negation des zweiten Ausdrucks versichert werden, dass keine der Aktionen vor einer Eingangsbestätigung durchgeführt werden.

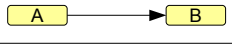
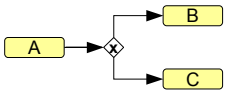
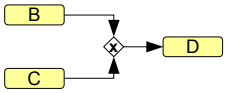
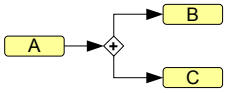
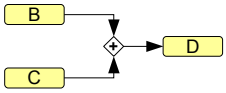
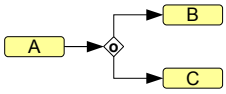
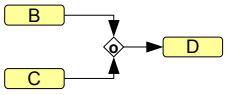
BPMN Konzept	***BPMN Notation***	***LTL Formel***
Sequenz		*(A* **R** *¬B)*
XOR Split Gateway		*(A* **R** *¬(***F** *B ⊕* **F** *C))*
XOR Join Gateway		*((B ⊕ C)* **R** *¬D)*
AND Split Gateway		*(A* **R** *¬(***F** *B ∧* **F** *C))*
AND Join Gateway		*((B ∧ C)* **R** *¬D)*
OR Split Gateway		*(A* **R** *¬(***F** *B ∨* **F** *C))*
OR Join Gateway		*((B ∨ C)* **R** *¬D)*

Tabelle 5: BPMN Elemente als LTL Formel (vgl. [37])

Compliance Scope 2

LTL Formel:

(2) *(***F** *Entscheidung durch Personalbereich ∧* **F** *Entscheidung durch Betriebsrat)* **R** *¬(***F** *Absage an Bewerber schreiben ⊕* **F** *Einladung Gespräch Vertragsunterzeichnung)*

Prinzipiell verhält sich die Formel (2) ebenso wie die Formel (1). Der Unterschied ist lediglich, dass im ersten Ausdruck nun nicht nur eine einzelne Aktion abgebildet wird, sondern die parallele Ausführung von zwei Aktionen.

Durch die logische und-Verknüpfung wird sichergestellt, dass der zweite Ausdruck und damit u.a. die Einstellung des Bewerbers, auch nur dann durchgeführt werden kann, wenn beide parallelen Aktionen aus dem ersten Ausdruck ausgeführt wurden.

4.3 Lineare temporal Logik mit Vergangenheitsoperatoren (LTL-P)

In LTL verfasste Spezifikationen lassen sich mit Hilfe von Vergangenheitsoperatoren oft vereinfachen und auf eine natürlichere Weise darstellen. Dies führt in der Regel auch dazu, dass die Formeln leichter zu verstehen sind[38][13].

Als Beispiel, um diese Aussage zu verdeutlichen, wird oft die Spezifikation „Einem Alarm geht immer ein Fehler voraus“ verwendet. Mit LTL-P kann diese Spezifikation folgendermaßen dargestellt werden[13]:

(3) $\mathbf{G}\ (Alarm \rightarrow \mathbf{O}\ Fehler)$

Diese Spezifikation kann in eine LTL Formel ohne Vergangenheitsoperatoren umgewandelt werden[13]:

(4) $\neg(\neg Fehler\ \mathbf{U}\ (Alarm \wedge \neg Fehler))$

Es ist offensichtlich, dass man den Sinn der Formel (4) im Vergleich zu (3) nicht so ohne Weiteres erkennt.

Wenn man die Formel für eine Sequenz aus dem vorhergehenden Teilkapitel auf dieses Beispiel anwendet, ergibt sich:

(5) $\mathbf{G}\ (Fehler\ \mathbf{R}\ \neg Alarm)$

Aber auch wenn die Formel (5) ähnlich „einfach“ aussieht wie die Formel (3), ist sie dennoch aufgrund des komplexeren **R**elease Operators und auch der Negation in der Formel, schwerer verständlich.

Der Nachteil von LTL mit Vergangenheitsoperatoren ist aber, dass das Model Checking in der Praxis nicht einfach ist. Des Weiteren ist noch unklar, in wie weit sich bestehende Model Checker effizient weiterverwenden lassen[13].

Basierend auf dem Beispiel aus [13] wurden Abbildungen für LTL-P abgeleitet, die in Tabelle 6 zusammengefasst sind. Für die Compliance Scopes aus dem Beispielprozess ergeben sich mit diesen Abbildungen folgende Spezifikationen.

Compliance Scope 1

LTL-P Formel:

(6) $((\text{Absage an Bewerber schreiben} \oplus \text{Einladung zu 1. Gespräch}) \rightarrow \mathbf{O}\ \text{Eingangsbestätigung})$

Diese Formel besagt, dass, bevor es zu einer Absage oder Einladung kommt, vorher eine Eingangsbestätigung erstellt wurde. Wie weiter oben schon beschrieben, ist die in LTL-P verfasste Formel deutlich leichter zu verstehen, als die entsprechende LTL Formel mit dem Release Operator.

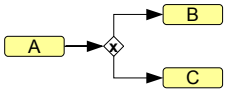
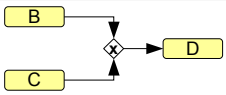
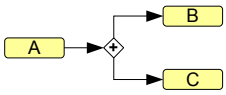
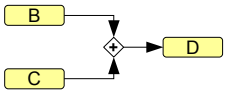
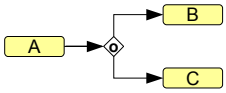
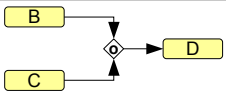
BPMN Konzept	*BPMN Notation*	*LTL-P Formel*
Sequenz		*(B* → **O** *A)*
XOR Split Gateway		*((B* xor *C)* → **O** *A)*
XOR Join Gateway		*D* → **(O** *B* xor **O** *C)*
AND Split Gateway		*((B* ∧ *C)* → **O** *A)*
AND Join Gateway		*D* → **(O** *B* ∧ **O** *C)*
OR Split Gateway		*((B* ∨ *C)* → **O** *A)*
OR Join Gateway		*D* → *(***O** *B* ∨ **O** *C)*

Tabelle 6: BPMN Elemente als LTL-P Formel

Compliance Scope 2

LTL-P Formel:

(7) (*(Absage an Bewerber schreiben* ⊕ *Einladung Gespräch Vertragsunterzeichnung)* → *(***O** *Entscheidung durch Personalbereich* ∧ **O** *Entscheidung durch Betriebsrat))*

Analog zu (6) besagt diese Formel, dass bevor eine Absage oder Einladung zur Vertragsunterzeichnung erstellt wird, eine Entscheidung durch den Personalbereich und den Betriebsrat erfolgen muss.
Indem man den Teil mit der Absage beim Exklusiven Gateway weg lässt, könnte man die Formel sogar noch etwas einfacher formulieren und auf diese Weise näher an der „ursprünglich“ formulierten Regel bleiben, die besagt, dass ein Bewerber nur eingestellt werden kann, wenn der Personalbereich und der Betriebsrat zugestimmt haben.

(8) (*Einladung Gespräch Vertragsunterzeichnung* → *(***O** *Entscheidung durch Personalbereich* ∧ **O** *Entscheidung durch Betriebsrat))*

Man sollte sich nur im Klaren sein, dass auf diese Weise nicht das ganze Exklusiv Gateway abgebildet wird, sondern nur ein einzelner Pfad des Gateways. In dem hier verwendeten Beispiel würde dies für die abzubildende Regel jedoch genügen.

4.4 Computation Tree Logic (CTL)

Da CTL, wie auch LTL, eine Untermenge von CTL* ist, ist die Sprache CTL der Sprache LTL recht ähnlich. Bei CTL muss nun aber jedem Zeitquantor noch einer der Pfadquantoren **A** (Formel gilt für alle Pfade) oder **E** (Formel gilt für mindestens einen Pfad) vorangestellt werden.

Die Abbildung der BPMN Notation nach CTL entspricht im Prinzip der Abbildung wie bei LTL, nur dass den Formeln nun noch ein Pfadquantor hinzugefügt wurde. Da sichergestellt werden soll, dass die Regeln, die sich aus den Compliance Scopes ergeben, immer gelten, wird der Pfadquantor **A** gewählt. Die daraus resultierenden Abbildungen können Tabelle 7 entnommen werden.

Für die beiden Compliance Scopes aus dem Beispielprozess ergeben sich daher die nachfolgenden Formeln.

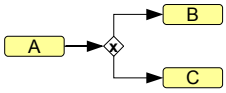
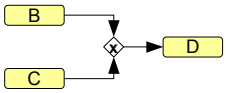
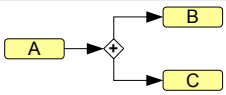
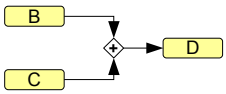
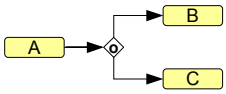
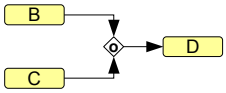
BPMN Konzept	***BPMN Notation***	***CTL Formel***
Sequenz		**AG***(A* **R** ¬*B)*
XOR Split Gateway		**AG***(A* **R** ¬*(***AF** *B* ⊕ **AF** *C))*
XOR Join Gateway		**AG***((B* ⊕ *C)* **R** ¬*D)*
AND Split Gateway		**AG***(A* **R** ¬*(***AF** *B* ∧ **AF** *C))*
AND Join Gateway		**AG***((B* ∧ *C)* **R** ¬*D)*
OR Split Gateway		**AG***(A* ***R*** ¬*(**AF** B* ∨ ***AF*** *C))*
OR Join Gateway		**AG***((B* ∨ *C)* **R** ¬*D*

Tabelle 7: BPMN Elemente als CTL Formel (vgl. [37])

Compliance Scope 1

CTL Formel:

(9) **AG** *(Eingangsbestätigung* **R** ¬*(***AF** *Einladung zu 1. Gespräch* ⊕ **AF** *Absage an Bewerber schreiben))*

Abgesehen von den Pfadquantoren sieht diese Formel wie die LTL Formel (1) aus. Im Grunde besagt diese Formel auch dasselbe wie (1), nur dass jetzt noch eine Aussage über die Pfade gemacht wird. Durch den Pfadquantor A werden alle möglichen Pfade betrachtet, so dass sichergestellt wird, dass es niemals dazu kommt, dass es zu einer Absage oder zu einem Bewerbergespräch kommt, solange keine Eingangsbestätigung verschickt wurde.

Compliance Scope 2

CTL Formel:

(10) **AG** *((***AF** *Entscheidung durch Personalbereich* ∧ **AF** *Entscheidung durch Betriebsrat)* **R** ¬*(***AF** *Absage an Bewerber schreiben* ⊕ **AF** *Einladung Gespräch Vertragsunterzeichnung))*

Analog zu Compliance Scope 1 ist auch die Formel (10), abgesehen von den Pfadquantoren, mit der LTL Formel (2) identisch. Durch die Betrachtung aller möglichen Pfade wird mit dieser Formel sichergestellt, dass die Einstellung des Bewerbers nur erfolgen kann, wenn es eine Entscheidung durch den Personalbereich und den Betriebsrat gab.

Genauer genommen besagt die Formel, dass es nach einer Entscheidung der zugehörigen Stellen entweder zu einer Absage oder eine Einstellung des Bewerbers kommt, dies ändert aber nichts an der Aussage, dass eine Einstellung des Bewerbers nur dann erfolgen kann, wenn es eine Entscheidung durch den Personalbereich und den Betriebsrat gibt. Daher sollte die Formel für die Intention des Compliance Scope 2 ausreichend sein.

4.5 Modallogik

Die temporale Logik, die in den vorhergehenden Teilkapiteln angewendet wurde, ist eine Variante der Modallogik[39]. Auch wenn die Modallogik keine temporalen Operatoren besitzt, lässt sich mit dieser Logik eine gewisse Abfolge von Aktionen durchaus darstellen. Die erarbeiteten Abbildungen von BPMN auf die modallogischen Formeln können Tabelle 8 entnommen werden.

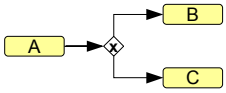
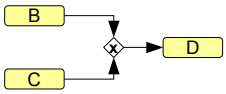
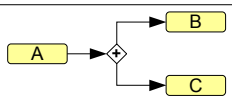
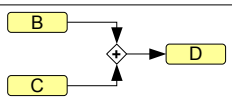
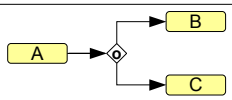
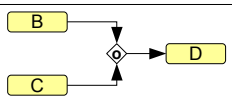
BPMN Konzept	*BPMN Notation*	*LTL Formel*
Sequenz		*(¬A → □¬B)*
XOR Split Gateway		*(¬A → □¬(B ⊕ C))*
XOR Join Gateway		*(¬(B ⊕ C) → □¬D)*
AND Split Gateway		*(¬A → □¬(B ∧ C))*
AND Join Gateway		*(¬(B ∧ C) → □¬D)*
OR Split Gateway		*(¬A → □¬(B ∨ C))*
OR Join Gateway		*(¬(B ∨ C) → □¬D)*

Tabelle 8: BPMN Elemente als Formel in Modallogik

Das, jetzt am Beispiel der Sequenz, die Formel über die Negation ($\neg A \rightarrow \Box\neg B$) aufgebaut wird, anstatt zu sagen „aus A folgt notwendigerweise B“ bzw. $(A \rightarrow \Box B)$, mag auf den ersten Blick etwas unverständlich aussehen, wenn man nur den „normalen“ Sprachgebrauch betrachtet. Oft wird die Implikation auch mit den Worten „wenn A, dann B“ ausgedrückt, wenn man sich aber die Wahrheitstabelle der Implikation (s. Tabelle 9) mal anschaut, wird man verstehen, warum nicht $(A \rightarrow \Box B)$ gewählt wurde, v.a. wenn man sich verdeutlicht, dass die Implikation $(A \rightarrow B)$ in $(\neg A \vee B)$ umgeformt werden kann.

A	B	$A \rightarrow B\ (= \neg A \vee B)$
0	0	1
0	1	1
1	0	0
1	1	1

Tabelle 9: Wahrheitstabelle Implikation

Problematisch bei der Implikation $(A \rightarrow B)$ ist, dass die Aussage wahr ist, auch wenn A nicht ausgeführt wird, aber B. Dies ist etwas, das aus Sicht der Abfolge „B darf nur ausgeführt werden, nachdem A ausgeführt wurde“ nicht sein darf.

Wenn man jetzt die Wahrheitstabelle der Formel $(\neg A \rightarrow \neg B)$ betrachtet (s. Tabelle 10), sieht man, dass die Aussage, wenn A nicht ausgeführt wird, aber B, falsch ist.

A	B	$\neg A \rightarrow \neg B\ (= A \vee \neg B)$
0	0	1
0	1	0
1	0	1
1	1	1

Tabelle 10: Wahrheitstabelle ($\neg A \rightarrow \neg B$)

Mit dieser Formel kann in gewisser Weise daher eine Abfolge realisiert werden, dass B nur ausgeführt wird, nachdem A ausgeführt wurde.

Compliance Scope 1

Modallogik Formel:

(11) $(\neg$*Eingangsbestätigung* $\rightarrow \Box\neg$*(Einladung zu 1. Gespräch* $\oplus$ *Absage an Bewerber schreiben))*

Mit Formel (11) wird einfach ausgedrückt gesagt, dass wenn es keine Eingangsbestätigung gibt, dann darf auch unter keinen Umständen eine Einladung zu einem 1. Gespräch bzw. eine Absage an den Bewerber verschickt werden. Ursprünglich war als

Regel festgelegt, dass die Eingangsbestätigung die erste Aktion im Bewerberprozess sein soll. Da mit der Regel (11) ohne eine Eingangsbestätigung keine nachfolgenden Aktionen erlaubt sind, wird durch diese Formel, auch ohne temporale Operatoren, die ursprüngliche Intention erreicht.

Compliance Scope 2

Modallogik Formel:

(12) $(\neg(\textit{Entscheidung durch Personalbereich} \wedge \textit{Entscheidung durch Betriebsrat}) \rightarrow \Box\neg(\textit{Absage an Bewerber schreiben} \oplus \textit{Einladung Gespräch Vertragsunterzeichnung}))$

Die Formel (12) unterscheidet sich von Formel (11) im Prinzip nur dadurch, dass vor der Implikation diesmal nicht nur auf eine Aktion „gewartet“ wird, sondern, durch das parallele Join im Compliance Scope 2, auf zwei Aktionen. Wird nur eine dieser Aktionen nicht durchgeführt, dann darf auch hier keine Absage bzw. Einladung an den Bewerber geschickt werden.

4.6 Zusammenfassung

Auch wenn es bei der Modallogik keine temporalen Operatoren gibt, lassen sich damit durchaus Abfolgen von Aktionen darstellen, wenn natürlich auch nicht mit so feinen Abstufungen, wie es mit LTL oder CTL möglich ist. Für die Modellierung von Regeln eignet sich meiner Meinung nach LTL mit Vergangenheitsoperatoren noch am besten, einfach weil die Formeln nicht so „aufgebläht“ wirken und am leichtesten verständlich sind. Das Problem bei LTL mit Vergangenheitsoperatoren wird sein, dass das Model Checking mit Vergangenheitsoperatoren in der Praxis nicht einfach ist[13].

Daher wird man letztendlich LTL oder CTL verwenden, um die Regeln in Compliance Scopes zu modellieren. Welche Sprache genutzt wird, wird auch davon abhängen, welcher Model Checker verwendet wird, da nicht jeder Model Checker beide Sprachen unterstützt. Zur Modellierung der Spezifikationen von Prozessen, welche mit BPMN erstellt wurden, sind beide Sprachen geeignet.

5 Implementierung

Teil der vorliegenden Arbeit ist es, den Oryx Editor zu erweitern. Zum einen soll es möglich sein Compliance Scopes zu definieren, zum anderen soll ein Model Checker an Oryx angebunden werden, mit der die Regeln in einem Compliance Scope auf ihre Richtigkeit überprüft werden können.

Zur Erinnerung, Compliance Scopes sind „Bereiche“ in einem Prozess, für den bestimmte Regeln gelten (s.a. Kapitel 4). In dieser Arbeit wird vorausgesetzt, dass ein Prozess in BPMN Notation im Oryx Editor vorliegt. In diesem Prozess können dann die Compliance Scopes definiert werden. Aus den Compliance Scopes wird dann über ein Plug-In die Spezifikation erstellt, die anschließend von einem Model Checker überprüft werden kann.

In den folgenden Teilkapiteln werden die einzelnen Erweiterungen des Oryx Editors näher vorgestellt.

5.1 Architektur

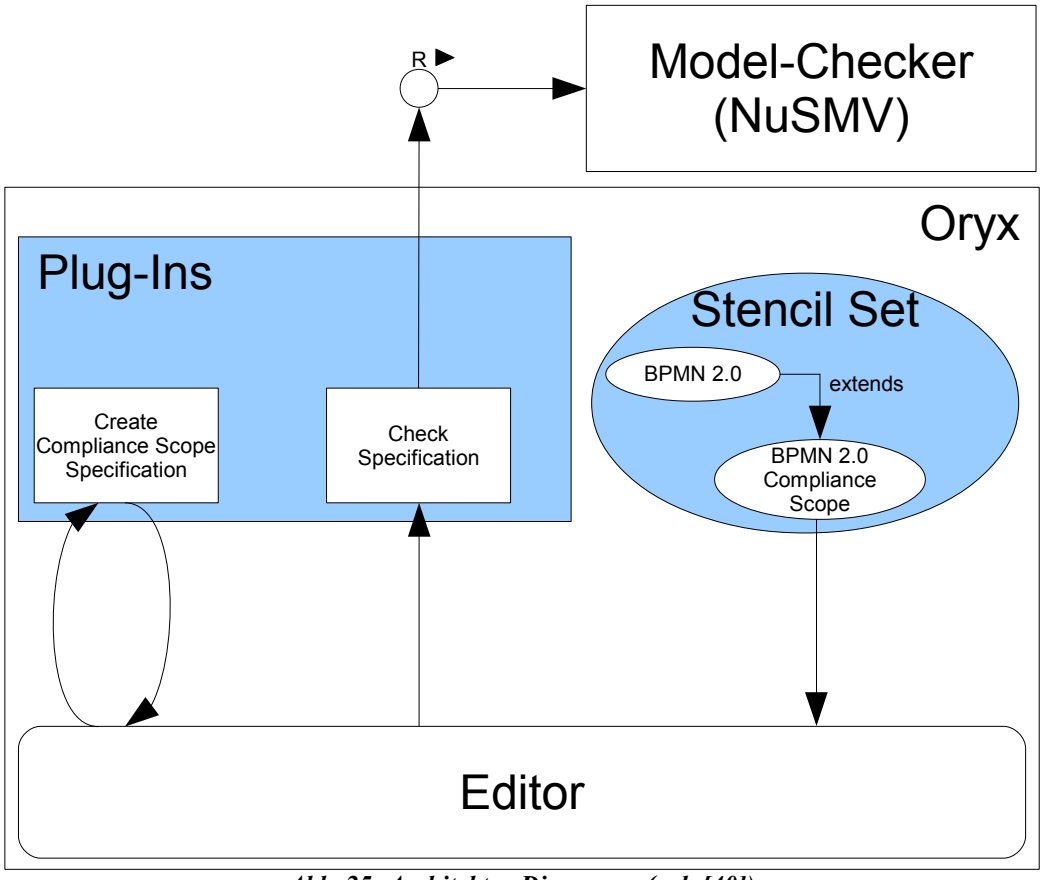


Abb. 25: Architektur Diagramm (vgl. [40])

In Abbildung 25 ist ein Überblick über die erstellten Erweiterungen in einem FMC-Diagramm dargestellt. Das Diagramm ist angelehnt an das FMC-Diagramm in [40], in dem der Aufbau des Oryx Editor im Detail erklärt wird. Nachfolgend werden die Er-

weiterungen kurz vorgestellt. In Kapitel 5.2 wird auf die einzelnen Erweiterungen detailliert eingegangen.

BPMN 2.0 Compliance Scope: dieses Stencil ist eine Erweiterung des BPMN 2.0 Stencil. Aktivitäten enthalten in diesem Stencil das zusätzliche Attribut „Compliance Scope“, mit dem bestimmt werden kann, ob die Aktivität zu einem Compliance Scope gehört oder nicht.

Wenn eine Aktivität einem Compliance Scope angehören soll, dann muss bei dem Attribut Compliance Scope die Bezeichnung des Compliance Scopes angegeben werden, zu welchem die Aktivität gehören soll.

Create Compliance Scope Specification: über dieses Plug-In werden die im Oryx-Editor hinterlegten Compliance Scopes eingelesen.

Für jeden Compliance Scope wird eine Spezifikation in LTL erstellt und im Prozess hinterlegt. In Abbildung 25 ist dies als ein Lese- und Schreibzugriff auf den Editor dargestellt, da die einzelnen Prozesselemente aus dem Editor heraus gelesen und die Spezifikation, oder möglicherweise auch Spezifikationen, je nach dem wie viel Compliance Scopes angelegt wurden, wieder zurück in den „Editor“ geschrieben werden.

Check Specification: über dieses Plug-In wird die im Prozess hinterlegte Spezifikation ausgelesen und ein Modell des Compliance Scopes erstellt. Die Spezifikation wird dem erstellten Modell hinzugefügt.

Wenn mehrere Compliance Scopes im Prozess hinterlegt sind, dann wird für jeden Compliance Scope ein Modell erstellt und mit der entsprechenden Spezifikation ergänzt.

Das erstellte Modell wird dann an den Model Checker übergeben. Das Ergebnis wird dann über eine Ausgabebox an den Anwender zurückgegeben.

Model Checker (NuSMV): der Aufruf des Model Checkers ist als Java Servlet umgesetzt. Das Servlet wird über das Plug-In „Check Specification“ aufgerufen. An dieses Plug-In wird auch die Antwort zurück gesendet, ob die Spezifikation das Modell erfüllt, oder nicht.

5.2 Implementierung der einzelnen Komponenten

In diesem Teilkapitel wird, auch anhand von Codeauszügen, darauf eingegangen, wie die einzelnen Komponenten umgesetzt wurden. Zum besseren Verständnis wird auch bis zu einem gewissen Grad auf den Aufbau von Stencil Sets in Oryx eingegangen und die Spezifikationssprache für den NuSMV Model Checker etwas vorgestellt.

5.2.1 BPMN 2.0 Compliance Scope

Die Komponente BPMN 2.0 Compliance Scope wurde als Erweiterung des BPMN 2.0 Stencil Sets umgesetzt.

Bevor auf diese Erweiterung eingegangen wird, soll zuerst einmal aufgezeigt werden, wie ein Stencil Set und anschließend wie eine Erweiterung eines Stencil Sets aufgebaut ist .

Stencil Set

Ein Stencil Set ist prinzipiell die Beschreibung von grafischen Objekten und Regeln, wie diese grafischen Objekte miteinander in Verbindung stehen. Dabei „besteht“ ein Stencil Set aus verschiedenen Dateien.

Die grafische Repräsentation von Objekten wird über Dateien im SVG Format vorgenommen. Zu jedem grafischen Objekt, d.h. zu jeder SVG Datei gehört auch eine Bilddatei im PNG Format, die ein Icon des grafischen Objektes darstellt[41].

Wenn man sich dazu Abbildung 17 auf Seite 15 dieser Arbeit anschaut, dann sind die Icons, die in der linken Spalte „Shape Repository“ zu sehen sind, die Bilddateien im PNG Format und die grafischen Objekte auf der Zeichenfläche entsprechen den SVG Dateien des Stencil Sets.

Das zentrale Element, in dem die Beziehungen der Objekte zueinander, aber auch Eigenschaften der einzelnen Stencils definiert werden, ist eine Datei im JSON[2] Format.

In dieser Datei werden alle Stencils definiert, im Falle des BPMN 2.0 Stencil Sets wären das z.B. Aktivitäten, Gateways, Events, die Verbindungspfeile usw., es werden aber auch alle Eigenschaften dieser Stencils definiert wie z.B. die Hintergrundfarbe oder der Name einer Aktivität. Zu guter Letzt werden in dieser Datei auch die Regeln definiert, wie die einzelnen Stencils miteinander verbunden werden[41]. Es ist z.B. nicht möglich, zwei Aktivitäten direkt aneinander zu „binden“. Zwischen zwei Aktivitäten befindet sich immer ein verbindendes Element, wie z.B. der Sequenzpfeil.

Damit man eine ungefähre Vorstellung hat, wie so eine Definition des Stencil Sets aussieht, ist in Listing 1 eine kurzer Beispielcode abgebildet (vgl. [42]).

```
 1 {
 2 "title" : "Tree Graph",
 3 "namespace": "http://b3mn.org/stencilset/treeGraph#",
 4 "description" : "This is the specification of a tree graph",
 5 "stencils" : [{
 6     "type" : "node",
 7     "id" : "Node",
 8     "title" : "Node",
 9     "groups" : ["Tree Graph"],
10     "description" : "A node.",
11     "view" : "node.svg",
12     "icon" : "node.png",
13     "roles": ["node"],
14     "properties": […]
15 }],
16 "rules": {…}
17 }
```

Listing 1: Stencil Set – JSON (vgl. [42])

Erweiterung eines Stencil Set

In dieser Arbeit soll Oryx so erweitert werden, dass in BPMN 2.0 modellierte Prozesse um Compliance Scopes erweitert werden können. Hierfür bietet sich das Erweiterungskonzept von Oryx an.

[2] http://www.json.org/

Anstatt, dass die Änderungen in der Original BPMN 2.0 JSON Datei vorgenommen werden, wird eine eigene JSON Datei für die Erweiterung angelegt. Diese Erweiterung kann über ein Plug-In, welches für Oryx schon existiert, geladen werden und steht anschließend dem Anwender zur Verfügung.

Der Aufbau einer Erweiterungsdatei ist im Prinzip identisch zu der Definition des Stencil Sets. Auch in der Erweiterung können grafische Objekte, sowie die Regeln, wie sich die Objekte zueinander verhalten sollen, definiert werden und es muss für jedes grafische Objekt eine SVG und PNG Datei erstellt werden.

Es besteht aber auch die Möglichkeit Stencils und Eigenschaften über eine solche Erweiterung zu entfernen, d.h. wenn die Erweiterung geladen wird, stehen nicht mehr alle Möglichkeiten des Original Stencil Sets zur Verfügung. Damit kann man z.B. Einsteigern ein Basis Stencil Set anbieten, ohne dass sie mit zu vielen oder komplexen Stencils „überfordert“ werden.

In Listing 2 ist der Code der Compliance Scope Erweiterung abgebildet. Hier kann man zum einen den allgemeinen Aufbau einer Stencil Set Erweiterung sehen, zum anderen sieht man hier auch, wie einer Aktivität eine neue Eigenschaft hinzugefügt wird.

Compliance Scope Erweiterung

Um Compliance Scopes abbilden zu können, wurde über das Erweiterungskonzept der Stencil Sets das Stencil „Aktivität“ um die Eigenschaft „Compliance Scope“ erweitert (s. Listing 2).

```
 1 {
 2 "title":"BPMN 2.0 Compliance Scope Extension",
 3 "title_de":"BPMN 2.0 Compliance Scope Erweiterung",
 4 "namespace":"http://oryx-editor.org/stencilsets/extensions/bpmnComplianceScope#",
 5 "description":"Extension to BPMN 2.0 to work with compliance scopes.",
 6 "description_de":"Erweiterung von BPMN 2.0 um Compliance Scopes.",
 7 "extends":"http://b3mn.org/stencilset/bpmn2.0#",
 8 "propertyPackages" :[],
 9 "stencils":[],
10 "properties": [{
11     "roles": ["Task"],
12     "properties": [{
13         "id":"scopeMembership",
14         "type":"String",
15         "title":"Compliance Scope",
16         "title_de":"Compliance Scope",
17         "value":"",
18         "description":"Name/Id of the Compliace Scope the " +
19         "activity belongs to.",
20         "description_de":"Bezeichnung/Id des Compliance Scopes, " +
21         "zu dem die Aktivität gehört",
22         "readonly":false,
23         "optional":true,
24         "refToView":"",
25         "wrapLines":false
26         }]
27     }]
28 }
```

Listing 2: Compliance Scope Erweiterung - JSON

Wie man sieht, ist der Aufbau der Erweiterung dem Code in Listing 1 sehr ähnlich. Dem BPMN Objekt „Aktivität“, im Listing ist dies in Zeile 11 als „Task“ bezeichnet, wird eine neue Eigenschaft hinzugefügt.

Dieser Eigenschaft wird eine „id“ (Zeile 13) gegeben, über die die Eigenschaft später programmatisch angesprochen werden kann. Als Titel (Zeile 15) wurde „Compliance Scope“ gewählt. Unter diesem Namen wird die Eigenschaft dann später im Oryx Editor angezeigt werden (s. Abbildung 26).

Abb. 26: Eigenschaft Compliance Scope

In Abbildung 26 ist ein Teil des Beispielprozesses aus Abbildung 24 sichtbar. In der Abbildung wurde die Aktivität „Entscheidung durch Betriebsrat“ ausgewählt, dadurch werden die Eigenschaften dieser Aktivität auf der rechten Seite angezeigt. Unter den Eigenschaften findet sich nun auch die Eigenschaft „Compliance Scope“, der in diesem Beispiel der Wert „Scope 2“ gegeben wurde. Zur Verdeutlichung wurde die Eigenschaft hier durch einen roten Rahmen hervorgehoben, im Oryx Editor ist dieser Rahmen jedoch nicht zu sehen.

Im Prinzip ist der Code im Listing 2 durch die vorgegebenen und mit verständlichen Namen versehenen Eigenschaften wie „id“, „type“ oder „value“ praktisch selbsterklärend. Daher soll hier nicht auf jede Zeile einzeln, sondern nur noch auf die Möglichkeit der Definition von mehreren Sprachen eingegangen werden.

Der Oryx Editor unterstützt Mehrsprachigkeit. Die Eigenschaften „title“ und „description“ können über Erweiterungen nach dem Schema „_xx_yy“ um verschiedene Sprachen erweitert werden.Wobei „xx“ für den nach ISO definierten Sprachcode steht und „yy“ für den ISO Ländercode. Der Ländercode ist dabei optional[43].

Die Eigenschaften „title“ bzw. „description“ sind im Oryx Editor für den Anwender direkt sichtbar, entweder als Name der Eigenschaft oder im Fall von „description“ als Pop-up Text, wenn man den Mauszeiger über die Eigenschaft hält. Daher bietet es sich auch an, für diese Eigenschaften eine Möglichkeit zu schaffen, mehrere Sprachen zu hinterlegen.

In den Zeilen 16 und 20 sieht man die Eigenschaften „title“ und „description“ mit dem Zusatz „_de“. Wenn jetzt bei der Sprachwahl im Oryx Editor die deutsche Sprache ausgewählt wird (s. Abbildung 27), dann wird für den Titel und die Beschreibung der Text angezeigt, der bei dem „_de“ Zusatz hinterlegt wurde. Wird keine alternative Sprache im Code hinterlegt, dann wird standardmäßig der Text verwendet, der bei „title“ bzw. „description“ hinterlegt wurde.

Abb. 27: Oryx Editor Sprachauswahl

Was in Listing 2 nicht sichtbar ist, ist wie Stencils oder Eigenschaften über eine Erweiterung entfernt werden können.

Hierfür gibt es die Möglichkeit in der JSON Datei der Erweiterung die Eigenschaften „removeStencils“ bzw. „removeProperties“ zu setzen.

In Listing 3 sieht man einen kurzen (und unvollständigen) Beispielcode, der das Prinzip verdeutlichen soll.

```
 1 {
 2 "title":"BPMN 2.0 Compliance Scope Extension",
 3 "title_de":"BPMN 2.0 Compliance Scope Erweiterung",
 4 "namespace":"http://oryx-editor.org/stencilsets/extensions/bpmnComplianceScope#",
 5 "description":"Extension to BPMN 2.0 to work with compliance scopes.",
 6 "description_de":"Erweiterung von BPMN 2.0 um Compliance Scopes.",
 7 "extends":"http://b3mn.org/stencilset/bpmn2.0#",
 8 "removeStencils":[],
 9 "removeProperties":[],
10 "stencils":[],
11 "properties": [{...
```

Listing 3: Extension Beispiel - removeStencils, removeProperties (vgl. [42])

5.2.2 Create Compliance Scope Specification

Die Komponente Create Compliance Scope Specification untersucht den in Oryx modellierten Prozess, ob Compliance Scopes hinterlegt sind. Falls Compliance Scopes hinterlegt sind, wird für jeden einzelnen Compliance Scope eine Spezifikation in LTL erstellt und im Prozess gespeichert.

Die Komponente wurde als Client Plug-In realisiert. Client Plug-Ins werden in JavaScript geschrieben und stellen dem Oryx Editor weitere Funktionalität zur Verfügung. Dabei stellt Oryx dem Entwickler verschiedene „Klassen“ zur Verfügung, um auf einzelne Elemente im Oryx Editor programmatisch zugreifen zu können. Eine zentrale Stellung nimmt hierbei das facade-Objekt ein, über welches man u.a. den Zugriff auf die Zeichenfläche in Oryx erhalten kann. Tiefergehende Informationen zur Entwicklung von Client Plug-Ins können in [40] und [44] nachgelesen werden. Nachfolgend soll, hauptsächlich anhand von Pseudo Code, aufgezeigt werden, wie das Plug-In realisiert wurde.

Implementierung des Create Compliance Scope Specification Plug-Ins

Das facade-Objekt bietet eine „offer“ genannte Methode an. Diese Methode dient sozusagen als Einstieg in das Plug-In. Dieser Methode wird ein Objekt, welches

Name/Wert Paare enthält (vergleichbar mit der Definition der Stencil Sets in der JSON Datei), übergeben. Dieses Objekt enthält dabei z.B. eine kurze Beschreibung des Plug-Ins, welches als Tool Tip im Editor angezeigt wird, oder den Pfad zum Icon, welches in der Toolbar oberhalb der Zeichenfläche erscheint. Viel wichtiger ist, dass hier auch der Funktionsaufruf zu der Funktion hinterlegt werden muss, der letztendlich den Code für das Plug-In enthält.

Um die Spezifikation zu erstellen, die später vom Model Checker geprüft wird, werden erst mal alle Knoten von der Zeichenfläche eingelesen. Anschließend wird jeder Knoten überprüft, ob dieser Knoten zu einem Compliance Scope gehört. Da derzeit nur Aktivitäten einem Compliance Scope hinzugefügt werden können, werden letztendlich auch nur Aktivitäten überprüft, andere Objekte, wie Gateways oder Events, die in Oryx ebenfalls als Knoten angesehen werden, werden dabei übersprungen.

In Listing 4 ist das Vorgehen in an JavaScript angelehnten Pseudocode dokumentiert.

```
 1
 2 canvas          = facade.getCanvas();
 3 all_nodes       = canvas.nodes;
 4
 5 complianceNodes = getComplianceScopeNodes(all_nodes);
 6
 7 for(scope in complianceNodes) {
 8     start_node  = this.getStartNode(scope);
 9     ltl_spec    = createLTLSpec(start_node);
10
11     if (specification = ""))
12         specification = ltl_spec;
13     else
14         specification += ", " + specification;
15 }
16
17 canvas.setProperty("complianceScopeLTLSpecification", specification);
18
```

Listing 4: Pseudocode zur Spezifikationserstellung

Die Funktion „getComplianceScopeNodes“, die in Zeile 5 aufgerufen wird, liefert ein Objekt zurück, dessen Eigenschaften die einzelnen Compliance Scopes sind und der Wert der Eigenschaften enthält ein Array mit allen Aktivitäten, die zu diesem Compliance Scope gehören.

Für jeden Compliance Scope wird anschließend der Startknoten in diesem Compliance Scope gesucht. Ausgehend von diesem Startknoten wird dann die LTL Spezifikation anhand der in Kapitel 4.2 Lineare temporale Logik (LTL) erarbeiteten Regeln erstellt.

Wenn es mehr als einen Compliance Scope gibt, werden die erstellten Spezifikationen mit Komma getrennt in einer Variable gespeichert. Diese Spezifikationen werden dann als Eigenschaft der Zeichenfläche hinzugefügt (Zeile 17). Diese Eigenschaft kann dann später durch das zweite Plug-In (s. Kapitel 5.2.3) ausgelesen werden.

In Listing 5 ist die Erstellung der Spezifikation detaillierter dargestellt. Das Listing ist aber stark vereinfacht, da nur das Prinzip der Erstellung aufgezeigt werden soll, d.h. es werden einige Dinge verkürzt dargestellt, wie z.B. das Zusammensetzen der Strings (siehe z.B. Zeilen 9, 20, 23, usw., es findet keine Abfrage statt, ob die zu befüllende Variable schon gefüllt ist oder nicht) oder die Abfrage, ob es sich bei einem Knoten um eine Aktivität (Zeile 8) oder ein Gateway (Zeilen 11 und 25) handelt.

Um das Listing nicht mit Kontrollabfragen „aufzublähen“ wurden auch Abfragen weggelassen, ob sich ein Knoten z.B. in einem Compliance Scope befindet, oder ob es sich bei dem untersuchten Objekt überhaupt über einen gültigen Knoten handelt. Für das Listing soll angenommen werden, dass als Knoten entweder eine Aktivität oder Gateway zurückgeliefert wird und das sich alle untersuchten Knoten in einem Compliance Scope befinden.

Prinzipiell basiert der Code auf der Untersuchung eines Knotens und dessen Nachfolger. Handelt es sich bei dem Nachfolgeknoten um eine Aktivität, dann handelt es sich um eine Sequenz und die Erstellung der Spezifikation, besser gesagt Teilspezifikation, ist recht einfach. Zur Erinnerung, bei einer Sequenz der Form A → B lautet die Formel (A **R** $\neg B$). In Zeile 9 sieht man, wie diese Formel im Code umgesetzt ist. Da alle Knoten im Compliance Scope untersucht werden, beginnend vom ersten Knoten im Compliance Scope, erhält man auch nur eine Teilspezifikation, die mit jedem Schleifendurchlauf dann bis zur Gesamtspezifikation zusammengesetzt wird. Bei einem Prozess der Art A→B→C wird im ersten Schleifendurchgang die Formel (A **R** $\neg B$) erstellt und im zweiten Durchgang (B **R** $\neg C$), welcher an die erste Formel mit einem logischen und-Operator (&) angehängt wird. Dies führt letztendlich zur Gesamtspezifikation, welche an den Model Checker übergeben werden kann: (A **R** $\neg B$) & (B **R** $\neg C$).

```
 1
 2 function createLTLSpec(node) {
 3
 4     succ = getSuccessor(node);
 5
 6     while (succ != undefined) {
 7
 8         if (succ == Task)
 9             specification += "&" + node + "R !" + succ;
10
11         else if (succ == SplitGateway) {
12             gatewayOperator = getGatewayOperator(succ);
13             outgoing = succ.outgoing;
14
15             for (j = 0; j < outgoing.length; j++) {
16                 specificationArray.push(createLTLSpec(outgoing[j]));
17             }
18
19             forall (elements in specificationArray) {
20                 specificationGateway += gatewayOperator + elements;
21             }
22
23             specification += "&" + node + "R !" + specificationGateway;
24
25         } else if (succ == JoinGateway) {
26             gatewayOperator = getGatewayOperator(succ);
27             incoming = succ.incoming;
28
29             for (j = 0; j < incoming.length; j++) {
30                 startNodeJoinGW = getStartNodeOfJoinGateway(incoming[i]);
31
32                 specificationArray.push(createLTLSpec(startNodeJoinGW);
33             }
34
35             forall (elements in specificationArray) {
36                 specificationGateway += gatewayOperator + elements;
37             }
38
39             specification += "&" + specificationGateway + "R !" + succ;
40         }
41
42         node = succ;
43         succ = getSuccessor(node);
44     }
45 }
46
```

Listing 5: Funktion createLTLSpec in Pseudocode

Handelt es sich beim Nachfolgeknoten nicht um eine Aktivität, sondern um ein Split Gateway, dann wird zuerst für jeden einzelnen Pfad des Gateways eine eigene Spezifikation erstellt. Dies wird dadurch erreicht, dass die Funktion *createLTLSpec* mit dem Nachfolgeknoten des Gateways rekursiv aufgerufen wird.

Man erhält also für jeden ausgehenden Pfad eine eigene Spezifikation. Diese Spezifikationen müssen dann, je nachdem um was für ein Gateway es sich handelt (exklusiv, inklusiv, parallel), mit dem entsprechenden Operator miteinander verbunden werden. Im Listing ist dies in Zeile 19 durch den „Operator“ *gatewayOperator* angedeutet, der vorher in Zeile 12 anhand des Gateway und einer eigenen Funktion bestimmt wird.

Nachdem die Spezifikation des Gateway erstellt wurde, kann diese dann anhand der Formel aus Kapitel 4.2 mit dem Knoten zusammengeführt werden (Zeile 23).

Das Vorgehen bei einem Join Gateway ist im Prinzip gleich dem Vorgehen bei einem Split Gateway. Der Unterschied ist, das bei einem Join Gateway nicht die ausgehenden Pfade untersucht werden, sondern die eingehenden, zudem muss zuerst für jeden eingehenden Pfad der „Startknoten“ des Pfades gesucht werden. Der restliche Ablauf ist identisch zu einem Split Gateway.

Noch eine Bemerkung zum Join Gateway. So wie es in Listing 5 dargestellt ist, funktioniert der Aufruf nur, wenn der Compliance Scope das zum Join Gateway gehörende Split Gateway nicht einschließt. Im Grunde stellt der Compliance Scope 2 aus dem Beispielprozess in Kaptiel 4.1 eine solche Situation dar. In Abbildung 24 ist zwar der Compliance Scope so gezeichnet, dass das Split Gateway im Compliance Scope enthalten ist, da man bisher aber nur Aktivitäten und keine Gateways einem Compliance Scope zuweisen kann, würde das Split Gateway nicht in die zu untersuchenden Knoten mit aufgenommen werden.

Damit Join Gateways generell berücksichtigt werden, müsste man nicht nur den Nachfolgeknoten auf den Typ (Aktivität oder Gateway) abfragen, sondern auch den eigentlichen Knoten. Wenn es sich bei dem eigentlichen Knoten um ein Join Gateway handelt, müsste dies extra behandelt werden, wobei die Spezifikationserstellung für das Join Gateway dem gleichen würde, was im if-Zweig ab Zeile 25 dargestellt ist. Damit das Listing aber nicht unnötig groß wird, wurde auf diesen Teil in Listing 5 verzichtet.

5.2.3 Check Specification

Die Komponente Check Specification erstellt für jeden Compliance Scope im Prozess ein Modell. In dieser Arbeit wird der NuSMV Model Checker verwendet, daher wird ein Modell in der NuSMV Sprache erstellt.

Nachfolgend soll anhand von Beispielen das Prinzip aufgezeigt werden, wie das NuSMV Modell aufgebaut wird, anschließend wird noch anhand von an JavaScript angelehnten Pseudocode aufgezeigt, wie das NuSMV Modell programmatisch erstellt wird.

NuSMV Modell

Der Aufbau eines NuSMV Modells ist recht einfach und soll an einem kurzen Beispiel aus [15] erläutert werden.

Prinzipiell kann man sich ein NuSMV Modell wie einen Automaten mit Zuständen und Transitionen vorstellen. In NuSMV müssen zuerst die verwendeten Variablen deklariert werden, und die Werte, die die Variablen einnehmen können, können als Zustände eines Automaten betrachtet werden. Anschließend an die Variablendeklaration werden die einzelnen Transitionen zwischen den „Zuständen“ definiert.

In Listing 6 ist ein einfaches Beispiel dargestellt, wie ein NuSMV Modell aussieht. In diesem Beispiel werden zwei Variablen deklariert. Die Variable *request*, die boolesche Werte annehmen kann und die Variable *state*, die die Werte *ready* und *busy* annehmen kann.

Der interessante Teil des Beispiels beginnt mit dem *ASSIGN* Abschnitt in Zeile 7. Der Variable state wird zuerst ein Initialwert zugewiesen (*ready*). Über den Befehl *next* werden nur die „Transitionen“ definiert.

Befindet sich die Variable im „Zustand *ready*“ und die Variable *request* enthält den Wert *TRUE*, dann wird *state* der Wert *busy* zugewiesen, man könnte auch sagen, der Automat wechselt in den Zustand *busy*, wenn man das Modell als Automaten auffasst.

```
 1
 2 MODULE main
 3 VAR
 4   request : boolean;
 5   state   : {ready, busy};
 6
 7 ASSIGN
 8   init(state) := ready;
 9   next(state) := case
10                    state = ready & request = TRUE : busy;
11                    TRUE                           : {ready, busy};
12                  esac;
13
```

Listing 6: NuSMV Beispielprozess [15]

In allen anderen Fällen (Zeile 11), kann der Variable *state* entweder der Wert *ready* zugewiesen werden, oder der Wert *busy*. Bei NuSMV wird, nicht wie in gängigen Programmiersprachen das Schlüsselwort „*else*“ verwendet, sondern *TRUE*, was im Endeffekt aber auf dasselbe hinausläuft.

Was man in diesem Beispiel ganz gut sehen kann, ist, dass NuSMV im Prinzip „von Haus aus“ eine einfache Möglichkeit liefert, Sequenzen und Exklusiv Gateways eines in BPMN modellierten Prozesses abzubilden. Im nächsten Abschnitt soll dies etwas detaillierter gezeigt werden.

Beispielprozess mit Sequenz und Exklusiv Gateway

Als Beispiel soll der Prozess aus Abbildung 28 dienen.

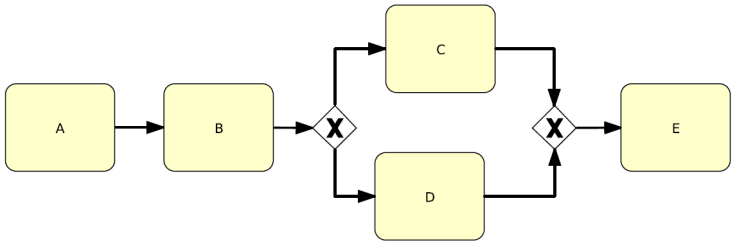


Abb. 28: Beispielprozess mit Sequenzen und Exklusiv Gateway

Der Prozess ist recht einfach aufgebaut. Einer Aktivität A folgt Aktivität B. Der Aktivität B folgt entweder Aktivität C oder Aktivität D und nach den Aktivitäten C oder D folgt Aktivität E.

Für das NuSMV Modell wird eine Variable deklariert, die als „Zustand“ die einzelnen Aktivitäten annehmen kann also A, B, C, D oder E. Die Definition einer Sequenz, wo einer Aktivität einfach eine andere Aktivität folgt, kann man in Listing 7 z.B. in Zeile 9 sehen. Wenn die Variable *state* den Wert A enthält, dann soll ihr im nächsten Schritt der Wert B zugewiesen werden, abgebildet auf den Beispielprozess hieße das, nach Aktivität A folgt Aktivität B.

```
 1
 2 MODULE main
 3 VAR
 4   state : {A, B, C, D, E};
 5
 6 ASSIGN
 7   init(state) := A;
 8   next(state) := case
 9                    state = A : B;
10                    state = B : {C, D};
11                    state = C : E;
12                    state = D : E;
13                    TRUE      : state;
14                  esac;
15
```

Listing 7: NuSMV Modell zum Beispielprozess mit Sequenzen und Exklusiv Gateway

In Zeile 10 kann man die Umsetzung des Exklusiv Gateways sehen. Wenn *state* den Wert B enthält, d.h. der Prozess befindet sich an der Aktivität B, dann soll anschließend entweder Aktivität C oder D aktiviert werden. In NuSMV gibt man dazu die möglichen Werte in geschweiften Klammern an. Es wird genau ein Wert aus den geschweiften Klammern der Variable *state* zugewiesen, was genau der Funktionsweise eines Exklusiv Gateways entspricht.

Die Realisierung eines parallelen oder inklusiven Gateways ist mit NuSMV ein kleines bisschen schwieriger und soll im nächsten Abschnitt erläutert werden.

Beispielprozess mit parallelem Gateway

Der Prozess in Abbildung 29 ist ähnlich einfach, wie im vorherigen Beispiel, nur dass hier ein paralleles Gateway verwendet wird. Nachdem Aktivität A ausgeführt wurde, müssen beide Aktivitäten B und C ausgeführt werden, bevor Aktivität D ausgeführt wird.

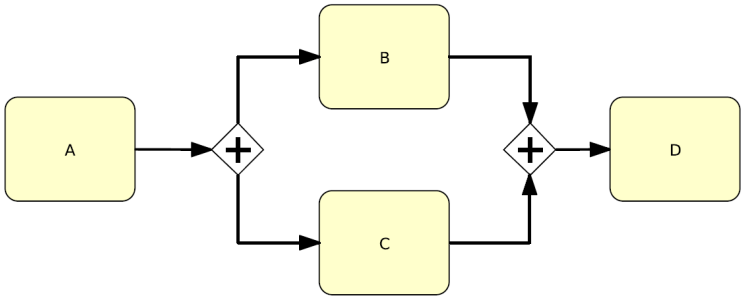


Abb. 29: Beispielprozess mit parallelem Gateway

Das Problem, dass sich für die Modellierung in NuSMV ergibt, ist, dass es nicht möglich ist, einer skalaren Variablen wie state, für die bei der Deklaration alle Werte definiert werden, die die Variable annehmen kann, mehr als einen Wert „gleichzeitig“ zuzuweisen (z.B. über ein Array). Daher wird der parallele Block als ein eigener Zustand bzw. eine eigene Aktivität angesehen. Diese Aktivität soll der Einfachheit halber P1 benannt werden, „P“ für parallel und „1“, weil dies die „Aktivität“ des ersten parallelen Gateway ist. Wären im Prozess zwei parallele Gateways, dann würde man diese als P1 und P2 bezeichnen.

Im Grunde wird das parallele Gateway durch eine Aktivität „ersetzt“ und der Prozess einfach als eine Sequenz angesehen (s. Abbildung 30).

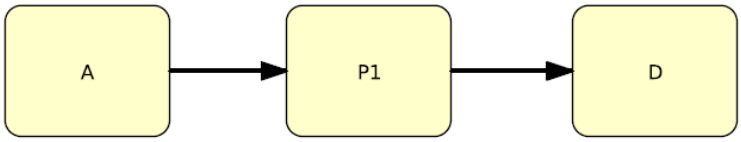


Abb. 30: Beispielprozess mit ersetztem parallelem Gateway

Die Abbildung einer Sequenz wie in Abbildung 30 ist mit NuSMV jetzt wieder einfach, wie man im vorherigen Abschnitt sehen konnte.

Nichtsdestotrotz muss das parallele Gateway selbst natürlich auch abgebildet werden. Dies geschieht über ein eigenes Modul.

In NuSMV muss es ein Modul mit dem Namen *main* geben, vergleichbar zu einem ausführbaren Java Programm, bei dem es ebenfalls eine „main Methode“ geben muss. Zusätzlich zum *main* Modul können in NuSMV weitere Module, die wie Funktionen angesprochen werden können, definiert werden.

Für jedes parallele Gateway wird in NuSMV nun ein eigenes Modul definiert. Das Modul sieht dabei prinzipiell wie das *main* Modul aus Listing 7 aus, nur das es in diesem Modul für jeden Pfad des parallelen Gateways eine eigene *state* Variable gibt, d.h. jeder Pfad des Gateways wird im Grunde als eigener sequentieller Prozess angesehen und dementsprechend umgesetzt. In Listing 8 ist der Prozess aus Abbildung 29 als NuSMV Modell abgebildet.

Wie man sieht, kann man den Modulen, die zusätzlich zum *main* Modul angelegt werden, Parameter übergeben. Dem Modul *procP1*, welches das parallele Gateway darstellt, wird der Inhalt der *state* Variable aus dem aufrufenden Modul übergeben.

Sobald die Variable *state* den Wert P1 annimmt, können auch die Aktivitäten im parallelen Gateway „aktiviert“ werden. Hierfür wurde auch der „Hilfswert“ *start*, mit denen die beiden Variablen *state1* und *state2* initialisiert werden, eingeführt. Würde man die Variablen *state1* mit B, bzw. *state2* mit C, initialisieren, dann hieße das für das Modell, dass die Aktivitäten B und C aktiviert sind, auch wenn der Prozess möglicherweise aufgrund eines Fehlers gar nicht bis zum parallelen Gateway gekommen wäre. Durch das Verwenden des Wertes *start* und der Abfrage, ob der Elternprozess schon am parallelen Gateway angekommen ist, kann gesteuert werden, dass die „Aktivitäten“ B und C erst dann aktiviert werden, wenn der Prozess am Gateway angekommen ist.

```
 1
 2 MODULE procP1(stateParent)
 3 VAR
 4   state1 : {start, B};
 5   state2 : {start, C};
 6
 7 ASSIGN
 8   init(state1) := start;
 9   init(state2) := start;
10
11   next(state1) := case
12                     state1 = start & stateParent = P1 : B;
13                     TRUE                              : state1;
14                   esac;
15
16   next(state2) := case
17                     state2 = start & stateParent = P2 : C;
18                     TRUE                              : state2;
19                   esac;
20
21 MODULE main
22 VAR
23   state : {A, B, P1};
24   stateP1 : procP1(state);
25
26 ASSIGN
27   init(state) := A;
28
29   next(state) := case
30                    state = A : P1;
31                    state = P1 & (stateP1.state1 = B) & (stateP1.state2 = C) : D;
32                    TRUE      : state;
33                  esac;
34
```

Listing 8: NuSMV Modell zum Beispielprozess mit parallelem Gateway

Weiter oben wurde gesagt, dass das parallele Gateway im Prinzip durch eine Aktivität ersetzt wird und das man somit einfach eine Sequenz im *main* Modul abbildet (s.a. Abbildung 30). Für den Schritt von A nach P1 trifft dies auch zu. Für den Schritt von P1 zu D trifft dies nicht ganz zu, da es nicht ausreicht zu sagen, wenn der Prozess im Zustand P1 ist, dann soll danach D folgen. Ob D aufgerufen wird oder nicht hängt davon ab, wie die „Prozesse“ im parallelen Gateway abgelaufen sind. Wie gesagt, jeder Pfad im parallelen Gateway wird im Prinzip als eigener Prozess angesehen. Für die Aktivität D heißt das, das nicht nur im Hauptprozess der Zustand P1 erreicht sein muss, sondern im Prozess procP1, müssen auch die Zustände B und C erreicht sein, erst dann darf im Hauptprozess mit D fortgefahren werden. Diese Abfrage ist in Zeile 31 abgebildet.

Was noch bleibt, ist die Abbildung eines inklusiven Gateways im NuSMV Modell.

Abbildung eines inklusiven Gateways in NuSMV

Das inklusive Gateway wird beinahe identisch wie das parallele Gateway in NuSMV abgebildet. Wie bei parallelen Gateways wird für jedes inklusive Gateway ein eigenes Modul erstellt, welches das inklusive Gateway im Hauptprozess ersetzt, so dass im Hauptprozess wieder einfach „nur“ eine Sequenz abgebildet werden muss. Das Modul für das inklusive Gateway wird dabei genauso aufgebaut wie das des parallelen Gateways.

Der einzige Unterschied würde Zeile 31 im Listing 8 betreffen. Da es bei einem inklusiven Gateway ausreicht, dass nur einer der Pfade des Gateways durchlaufen wird, werden die „einzelnen“ Prozesse des Gateways nicht mit einer logischen und-Verknüpfung verbunden, sondern mit einem logischen „oder“ (s. Listing 9).

```
30 ...
31                  state = P1 & ((stateP1.state1 = B) | (stateP1.state2 = C)) : D;
32 ...
```

Listing 9: Unterschied zu Listing 8 für inklusives Gateway

Erstellung NuSMV Modell

Das NuSMV Modell wird über ein JavaScript Plug-In erstellt. Ähnlich wie bei der Erstellung der Spezifikation in LTL wird auch hier die Kombination eines Knotens mit seinem Nachfolgeknoten untersucht. In Listing 10 ist der grobe Aufbau skizziert, wie das NuSMV Modell erstellt wird.

```
 1
 2 function createNuSMVModel(node) {
 3
 4   while (node != undefined) {
 5
 6     objLeftSide  = getLeftSideTransition(node);
 7     objRightSide = getRightSideTransition(objLeft.Succ);
 8
 9     nuSMVTransition += "\n state = " + objLeftSide.Exp + " : " + objRightSide.Exp;
10
11     if ("procPar" in objRightSide)
12       moduleExtra["P" + parallelCounter] = objRightSide.procPar;
13     else if ("procInc" in objRightSide)
14       moduleExtra["I" + inclusiveCounter] = objRightSide.procInc;
15
16     node     = objRight.Node;
17   }
18
19   nuSMVHead = createNuSMVHead(moduleExtra);
20   ltlSpec   = canvas["ltlSpecification"];
21
22   nuSMVModel = nuSMVHead + nuSMVTransition + ltlSpec;
23
24   new ajax.call(url, nuSMVModel) {
25     onSuccess : alert("Die Spezifikation wird erfüllt."),
26     onFailure : alert("Die Spezifikation wird nicht erfüllt. Bitte überprüfen Sie
       den Prozess!"
27   }
28 }
29
```

Listing 10: Plug-In createNuSMVModel

Wie bei der LTL Spezifikation wird auch dieser Funktion der Startknoten, d.h. der erste Knoten aus dem Compliance Scope, übergeben. Ausgehend von diesem Knoten, werden alle weiteren Knoten im Compliance Scope abgearbeitet.

Der Aufbau dieses Skripts ist relativ einfach. Die Hauptarbeit besteht darin, die Transitionen im NuSMV Modell zu erstellen. Wie man im vorherigen Teilkapitel sehen konnte, besteht eine solche Transition aus einer Abfrage, in welchem „Zustand“ sich das Modell befindet und der Angabe des Folgezustands. Im Skript wird nun die Abfrage, in welchem Zustand sich das Modell befindet, berechnet (Zeile 6, *getLeftSideTransition*) und in Abhängigkeit des Nachfolgeknotens, welcher in der Funktion *getLeftSideTransition* berechnet wird, wird die Angabe des Folgezustands erstellt (Zeile 7, *getRightSideTransition*). Zudem wird in der Funktion *getRightSideTransition* auch der Knoten zurückgegeben, bei dem die Untersuchung der Knoten weitergeht. Durch die teilweise unterschiedliche Behandlung von Gateways muss dies aber nicht unbedingt der direkte Nachfolger des gerade untersuchten Knotens sein, dies wird später aber noch genauer erläutert.

Die Abfrage in den Zeilen 11-14 dienen der Extrabehandlung, wenn es sich um parallele oder inklusive Gateways handelt. Für Gateways diesen Typs werden je eigene

Module erzeugt, die außerhalb des *main* Moduls des NuSMV Modells angegeben werden. Hier werden diese Module zwischengespeichert und in Zeile 19 am Anfang des Modells über eine eigene Prozedur (*createNuSMVHead*) hinzugefügt.

Die Spezifikation in LTL wurde ja über das Plug-in, auf welches in Kapitel 5.2.2 Create Compliance Scope Specification eingegangen wurde, erzeugt und im Prozess gespeichert. In diesem Plug-In wird die erstellte Spezifikation einfach ausgelesen (Zeile 20) und dem NuSMV Modell am Ende hinzugefügt. In der Praxis müsste der Spezifikation noch das Schlüsselwort „LTLSPEC“ vorangestellt werden. Dies wurde in diesem Listing jedoch der Übersichtlichkeit wegen weggelassen.

Die eigentliche Arbeit der Modellerstellung wird in den zwei Funktionen getLeftSideTransition und getRightSideTransition durchgeführt. Bei einer Sequenz wie z.B. A → B würde die Transition in etwa so aussehen: „`state = A : B;`“.

Was über diese beiden Funktionen nun berechnet wird, ist der Teil links und rechts vom Doppelpunkt, in diesem Fall „A“ und „B“.

```
 1
 2 function getLeftSideTransition(node) {
 3
 4   if (node = Task) {
 5     obj.Exp = node.Name;
 6     obj.Succ = getSuccessor(node);
 7     return obj;
 8
 9   } else if (node = ExclusiveSplitGateway) {
10     joinNode = getJoinGateway(node);
11
12     forall outgoing paths {
13
14       nodePath = outgoing[i];
15       while (nodePath != joinNode) {
16
17         objLeft  = getLeftSideTransition(nodePath);
18         objRight = getRightSideTransition(objLeft.Succ);
19
20         transition += "\n state = " + objLeft.Exp + " : " + objRight.Exp;
21
22         nodePath = objRight.Node;
23       }
24     }
25
26     obj.Exp  = transition;
27     obj.Succ = getSuccessor(joinNode);
28
29     return obj;
30
31   } else if (node = ParallelJoinGateway {
32
33     obj.Exp = "P" + parallelCounter + " & (" + moduleExtra["P" +
       parallelCounter].Query + ")";
34     obj.Succ = getSuccessor(node);
35
36     return obj;
37
38   } else if (node = InclusiveJoinGateway {
39       // analog zu ParallelJoinGateway
40 }
41
```

Listing 11: JavaScript Pseudocode zu getLeftSideTransition

In Listing 11 ist grob der Code dargestellt, der den linken Teil vom Doppelpunkt zurück gibt, d.h. den aktuellen Zustand, in dem sich das Modell befindet.

Wenn es sich um eine Aktivität handelt, dann ist das Vorgehen sehr einfach, es wird einfach der Name der Aktivität zurückgegeben, da ja die Aktivitäten im Grunde wie Zustände behandelt werden.

Bei einem exklusiven Gateway wird es schon etwas komplizierter. Zum einen hat ein Gateway mehrere ausgehende Pfade, zum anderen kann in einem solchen Pfad mehr als nur eine Aktivität vorhanden sein, siehe dazu als Beispiel Abbildung 31. Daher werden für jeden ausgehenden Pfad die ganzen Transitionen, also linke und rechte Seite des Doppelpunkts, ähnlich wie im „Hauptskript“ in Listing 10, berechnet. Im Prinzip geht man jeden ausgehenden Pfad einmal komplett durch und erstellt dafür die Transition. Da man auf diese Weise für jeden Knoten, der zu dem exklusiven Gateway gehört, schon die Transition erstellt hat, ist der Nachfolgeknoten des Gateways, beim Beispiel aus Abbildung 31 nicht B oder D, sondern der Nachfolgeknoten, des dazugehörigen „Join-Gateways“, hier wäre das also E.

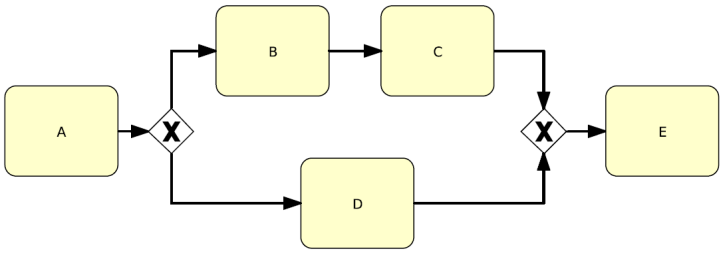


Abb. 31: Sequenz im Pfad eines exklusiv Gateways

Daher wird im Skript auch das zum Split-Gateway zugehörige Join-Gateway gesucht (Zeile 10) und der Nachfolger dieses Gateways (Zeile 27) zurückgegeben.

Die Erstellung der Transition für das parallele und inklusive Gateway ist im Prinzip gleich, daher wird hier nur auf das parallele Gateway eingegangen. Was beim ersten Blick auf den Code auffällt, ist, dass nur das Join-Gateway abgefangen wird. Durch den Aufbau wie das Hauptskript und die Funktionen *getLeftSideTransition* und *getRightSideTransition* zusammenspielen, kommt es nicht vor, dass ein Split-Gateway eines parallelen oder inklusiven Gateways in der node-Variable gespeichert wird.

Dies kommt dadurch zustande, dass der Startknoten immer eine Aktivität ist (zur Erinnerung, Compliance Scopes können derzeit nur Aktivitäten hinzugefügt werden) und damit wird die Funktion *getLeftSideTransition* zu aller erst immer mit einer Aktivität aufgerufen. In der Funktion *getLeftSideTransition* wird aber auch der Nachfolgeknoten berechnet der der Funktion *getRightSideTransition* übergeben wird, da die Knoten in der Ablaufreihenfolge des Prozesses abgearbeitet werden, wird dadurch ein Split-Gateway auch immer zuerst in der Funktion *getRightSideTransition* behandelt. In der Funktion *getRightSideTransition* wird nun berechnet, mit welchem Knoten das Hauptskript weiter arbeitet. Im Falle eines parallelen Split-Gateways wäre das das zugehörige Join-Gateway. Auf diese Weise „überspringt“ man das Split-Gateway als Nachfolgeknoten im Hauptskript, so dass dieses nicht als Parameter mit der Funktion *getLeftSideTransition* aufgerufen wird.

Zum besseren Verständnis betrachte man noch einmal Abbildung 29 auf Seite 45. In diesem Beispiel würde die Aktivität A der Funktion *getLeftSideTransition* übergeben werden und das nachfolgende Split-Gateway der Funktion *getRightSideTransition*. Als

nächster abzuarbeitender Knoten für das Hauptskript würde *getRightSideTransition* nun das zugehörige Join-Gateway zurückgeben (s. Zeile 24, Listing 12).

Daher wird in der Funktion *getLeftSideTransition* auch nur das Join-Gateway betrachtet. Für die „linke Seite“ wird nun eine Überprüfung benötigt, wie sie in Listing 8 in Zeile 31 zu sehen ist, d.h. es wird überprüft, ob sich das Modell im zugehörigen „virtuellen“ Zustand (in Listing 8 wäre das „P1“) befindet und ob die Pfade für das Gateway durchlaufen wurden. Die zugehörige Formel, ob die Pfade durchlaufen wurden, wird in einem Objekt gespeichert, welches zuvor in der Funktion *getRightSideTransition*, erstellt wurde.

Analog zur Funktion *getLeftSideTransition* werden in der Funktion *getRightSideTransition* nun die Ausdrücke berechnet, die rechts vom „Doppelpunkt“ des NuSMV Codes stehen müssen.

```
 1
 2 function getRightSideTransition(node) {
 3
 4   if (node = Task) {
 5     obj.Exp = node.Name;
 6     obj.Node = node;
 7
 8     return obj;
 9
10   } else if (node = ExclusiveSplitGateway) {
11     forall outgoing paths {
12       objTemp = getRightSideTransition(getSuccessor(outgoing[i]));
13       obj.Exp += ", " + objTemp.Exp;
14     }
15
16     obj.Exp = "{" + obj.Exp + "}";
17     obj.Node = node;
18
19     return obj;
20
21   } else if (node = ExclusiveJoinGateway) {
22       objTemp  = getRightSideTransition(getSuccessor(node));
23
24       obj.Exp  = objTemp.Exp;
25       obj.Node = node;
26
27       return obj;
28
29   } else if (node = ParallelSplitGateway) {
30       joinNode  = getJoinGateway(node);
31
32       obj.Exp     = "P" + parallelCounter;
33       obj.ProcPar = createModel(parallel);
34       obj.Node    = joinNode;
35
36   } else if (node = InclusiveSplitGateway) {
37       // analog zu ParallelSplitGateway
38 }
39
```

Listing 12: JavaScript Pseudocode zu getRightSideTransition

Die Funktion *getRightSideTransition* erhält als zu untersuchenden Knoten den Nachfolgeknoten des Knotens, der in der Funktion *getLeftSideTransition* untersucht wurde. Handelt es sich bei einem solchen Nachfolger um eine Aktivität, dann wird, wie auch in der Funktion *getLeftSideTransition* einfach der Name der Aktivität zurückgeben.

Handelt es sich bei dem Nachfolger um ein exklusives Gateway, dann muss unterschieden werden, ob es sich dabei um ein Split- oder Join-Gateway handelt.

Bei einem Split-Gateway wird dafür für jeden Pfad der erste Knoten eines Pfades genommen. Erreicht wird dies durch den rekursiven Aufruf der *getRightSideTransition*

Funktion. Der rekursive Aufruf wird durchgeführt, weil der Nachfolgeknoten des Split-Gateways nicht unbedingt eine Aktivität sein muss, durch den rekursiven Aufruf erhält man auf diese Weise auch einen gültigen Ausdruck, wenn der Nachfolgeknoten erneut ein Split-Gateway ist. Die Ausdrücke für jeden Pfad werden in einer Variable gespeichert und in geschweiften Klammern zurückgegeben, was dem XOR Ausdruck im NuSMV Modell entspricht (s.a. Listing 7, Zeile 10).

Bei einem Join-Gateway wird der Ausdruck der rechten Seite des Nachfolgeknotens des Join-Gateways benötigt. Da dieser Nachfolgeknoten ebenfalls nicht unbedingt eine Aktivität sein muss, wird dieser Ausdruck ebenfalls durch einen erneuten Aufruf der Funktion *getRightSideTransition* erhalten, ähnlich wie im Falle des Split-Gateways.

Im Falle eines parallelen oder inklusiven Gateways müssen nur die Split-Gateways untersucht werden, da die Join-Gateways, wie weiter oben schon erklärt wurde, in der Funktion *getLeftSideTransition* abgearbeitet werden.

Auch hier ist der Ablauf bei parallelen und inklusiven Gateways gleich, daher wird auf das inklusive Gateway nicht noch zusätzlich eingegangen. Wie bei der Vorstellung des NuSMV Codes schon geschrieben, wird ein paralleles Gateway, gemeint ist der ganze Prozess zwischen Split- und Join-Gateway, als eine Aktivität behandelt. Für diese Aktivität wird ein Name benötigt, der sich hier einfach aus einem „P“ (für parallel) und einer Zählvariable zusammensetzt, in der die Anzahl der parallelen Gateways gezählt wird. Die Inkrementierung dieser Variable ist im Listing nicht extra ausgeführt.

Was in der Beschreibung des NuSMV Codes für das parallele Gateway ebenfalls geschrieben wurde, ist, dass für das parallele Gateway ein eigenes Modul erzeugt wird. Dies geschieht über die Funktion createModel. Diese Funktion wurde nicht extra als Listing aufgeführt. Dies liegt daran, das diese Funktion praktisch wie das Hauptskript *createNuSMVModel* funktioniert. Im Prinzip wird, wie im Hauptskript, das Modul aufgebaut, nur dass in diesem Fall über eine for-Schleife für jeden ausgehenden Pfad des Split-Gateways eine eigene Zustandsvariable *state* erstellt wird und damit auch für jede *state* Variable die entsprechenden Transitionen (s. Listing 8). Der Code für das Modell wird als eine Eigenschaft in dem JavaScript Objekt gespeichert, welches an das Hauptskript zurückgeben wird. Im Hauptskript gibt es eine Überprüfung, ob diese Eigenschaft existiert und falls die Eigenschaft existiert, wird der Code in einem extra Objekt abgelegt (Zeilen 11-14 in Listing 10), welches der Funktion übergeben wird, die den „Kopfteil“ des NuSMV Modells erstellt, in dem die zusätzlichen Module letztendlich aufgeführt werden.

Was in den Listings zur Modellerstellung ebenfalls nicht aufgeführt wurde, ist, dass alle Aktivitäten in einer Variable gespeichert werden müssen, damit die *state* Variable, die im Code verwendet wird, korrekt deklariert werden kann. Dies wurde, wie die weiter oben erwähnte Inkrementierung der Zählvariablen für parallele und inklusive Gateways, aus Platzgründen weggelassen, um die Listings nicht unnötig groß zu machen.

5.2.4 Model Checker (NuSMV)

Diese Komponente ist ein Java Servlet und wird durch das Plug-in, welches im vorherigen Teilkapitel beschrieben wurde, aufgerufen.

Das Servlet nimmt das durch das Plug-In erstellte NuSMV Modell als Parameter entgegen und schreibt mit Hilfe der Input- und Outputstreamer Klassen von Java den NuSMV Code in eine Datei, die auf der Festplatte abgelegt wird.

Über die runtime Klasse in Java und dem Pfad zur Datei des NuSMV Modells auf der Festplatte kann nun der Model Checker aufgerufen werden. Voraussetzung dafür ist natürlich, dass der Model Checker auch auf dem System installiert wurde.

Sobald der Model Checker das Modell überprüft hat, gibt er im Erfolgsfall die Meldung „specification *LTLSpecification* is true“ an das Servlet zurück. Wobei „*LTLSpecification“* hier als Platzhalter für die tatsächliche Spezifikation verwendet wird.

Würde z.B. die Sequenz A → B mit der Spezifikation $(A\ \mathbf{R}\ \neg B)$ überprüft werden und das Modell erfüllen, wäre der Rückgabewert des Model Checkers

„specification $(A\ \mathbf{R}\ \neg B)$ is true“ .

Für den Fall, dass die Spezifikation von dem Modell nicht erfüllt wird, wird ein Gegenbeispiel zurückgegeben. Diese Rückgabewerte des Model Checker werden wiederum über die Input- und Outputstreamer Klassen entgegen genommen und ausgewertet, d.h. es wird der String für eine positive Antwort in Java zusammengesetzt, also „specification *LTLSpecification* is true“ und überprüft, ob dieser String in der Antwort des Model Checkers vorkommt.

Je nachdem, ob nun die Spezifikation erfüllt wird oder nicht, wird eine entsprechende Antwort vom Servlet an das aufrufende Plug-In zurückgesendet. Im Plug-In wird dann eine Messagebox generiert, die den Anwender über das entsprechende Ergebnis informiert.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In der vorliegenden Arbeit wurden in Kapitel 4 Abbildungen erarbeitet, mit denen man einen in BPMN modellierten Prozess in eine formale Sprache überführen kann. Untersucht wurden dabei die temporalen Sprachen LTL, LTL mit Vergangenheitsoperatoren und CTL. Etwas außer Konkurrenz wurde dabei auch die Modallogik angeschaut, die sich aufgrund fehlender temporaler Operatoren nicht optimal eignet, einen Prozess abzubilden.

Das Ziel war Möglichkeiten zu erarbeiten, um in einem Prozess Regeln zu hinterlegen, die durch ein Computersystem überprüft werden können. Für die Überprüfung solcher Regeln wurde in dieser Arbeit ein Model Checker verwendet. Dazu wurden in Kapitel 3 einige Model Checker vorgestellt. Letztendlich habe ich mich für den NuSMV Model Checker entschieden, da dieser in der aktuellen Version von Haus aus auch mit LTL-P umgehen kann, also einer temporalen Logik, welche Vergangenheitsoperatoren besitzt. Zusätzlich wurde in Kapitel 3 das Prinzip erklärt, was Model Checking ist, wie es funktioniert und warum man überhaupt Model Checking einsetzt.

In Kapitel 2 wurden die Grundlagen der Themen erläutert, die zum Verständnis dieser Arbeit hilfreich sind. Dazu gehört eine Vorstellung der untersuchten temporalen Logiken CTL, LTL und LTL-P, sowie der Modallogik. Da in Kapitel 4 auf Prozesse, die mit BPMN modelliert wurden, Bezug genommen wurde, wurde auch BPMN selber mit den am häufigsten verwendeten Elementen vorgestellt. Des Weiteren wurde auf die Kripke Struktur eingegangen, da das Verständnis der Kripke Struktur hilfreich für das Model Checking ist.

Da der praktische Teil der Arbeit aus einer Erweiterung des Oryx Editors bestand, wurde im Grundlagenkapitel auch der Oryx Editor und seine Funktionsweise kurz vorgestellt.

Abschließend wurde in Kapitel 5 auf die Art und Weise der Implementierung der einzelnen umgesetzten Komponenten, welche im Rahmen dieser Arbeit entstanden, größtenteils anhand von Pseudocode, eingegangen. Dabei wurde der Oryx Editor mit dem generellen Aufbau von Stencil Sets und einem kleinen Einblick in die Plug-In Programmierung auch ein Stück weit aus Entwicklersicht vorgestellt.

6.2 Ausblick

In dieser Arbeit wurde, aufgrund der Möglichkeit Spezifikationen in LTL-P zu verwenden, der NuSMV Model Checker an Oryx angebunden. Bei der praktischen Umsetzung wurde jedoch erst mal nur die Spezifikation in LTL berücksichtigt, da dadurch die Erstellung der Spezifikation auch für andere Model Checker wie z.B. Spin, verwendet werden kann.

Weitere mögliche Arbeiten könnten daher zum einen die Umsetzung beinhalten, dass Spezifikationen in LTL-P erstellt werden, zum anderen kann das Plug-In, welches das Modell in der NuSMV Sprache erstellt, so erweitert werden, dass z.B. Promela Code für den Spin Model Checker erstellt wird.

Eine weitere, v.a. für den Anwender, sinnvolle und hilfreiche Erweiterung wäre eine differenziertere Rückmeldung auf die Überprüfung einer Spezifikation. Wenn eine Spezifikation das Modell nicht erfüllt, dann wird bisher nur eine Meldung angezeigt, dass die hinterlegte Spezifikation nicht erfüllt wird. Für den Anwender wäre es leichter den Grund für die nicht erfolgreiche Überprüfung der Spezifikation zu finden, wenn das Element im Prozess, welches dazu führt, dass gegen die Spezifikation verstoßen wird, in der Meldung ausgewiesen oder möglicherweise auch direkt in der Zeichenfläche markiert wird. Hierfür ist eine automatische Analyse des Gegenbeispiels nötig, welches der Model Checker im Fehlerfall zurück gibt. Anhand dieser Analyse müsste das Element identifiziert und wieder auf den Oryx Editor abgebildet werden.

Literaturverzeichnis

[1] Allweyer, Thomas: Geschäftsprozessmanagement. Herdecke Bochum: W3L GmbH 2005

[2] Geißler, Cornelia: Was ist...: Compliance Management?. In: Harvard Business Manager (2/2004). Hamburg: manager magazin Verlagsgesellschaft mbH. http://www.harvardbusinessmanager.de/heft/artikel/a-620695.html, 2004

[3] Jürgen, Axel; Rödl, Christian; Nave, José A. Campos: Praxishandbuch Corporate Compliance. Weinheim: WILEY-VCH Verlag GmbH & CoKGaA 2009

[4] Schleicher, Daniel; Anstett, Tobias; Leymann, Frank; Mietzner Ralph: Maintaining Comliance in Customizable Process Models. In: Proceedings of the 17th International Conference on Cooperative Information Systems (CoopIS 2009). Springer Verlag 2009, pp. 1-16

[5] Kuhlang, Peter: Geschäftsprozessmanagement-Tools. Wien: Neuer Wissenschaftlicher Verlag GmbH Nfg KG 2010

[6] Allweyer, Thomas: BPMN 2.0 - Business Process Model and Notation: Einführung in den Standard für die Geschäftsprozessmodellierung. Norderstedt: Books on Demand GmbH 2009

[7] BPMN Information Home. http://www.bpmn.org/

[8] White, Stephen A.: Introduction of BPMN. http://www.bpmn.org/Documents/Introduction_to_BPMN.pdf, 2004

[9] Gumm, H. Peter: Kripke Strukturen und SMV. http://www.mathematik.uni-marburg.de/~gumm/Lehre/SS07/ModelChecking/03_Kripke-Strukturen_und_SMV.pdf, 2007

[10] Latvala, Timo: Reactive Systems: Kripke Structures and Automata. http://www.tcs.hut.fi/Studies/T-79.186/2004/lecture3.pdf, 2004

[11] Neumann, Daniel: Zustandsautomaten/Kripke-Strukturen. http://www2.informatik.hu-berlin.de/top/lehre/WS06-07/se_se/folien/kripkestrukturen.ppt, 2006

[12] Clarke, Edmund M. Jr.; Grumberg, Orna; Peled, Doron A.: Model Checking. Cambridge London: The MIT Press 1999

[13] Pradella, M.; San Pietro, P.; Spoletini, P.; Morzenti, A.: Practical Model Checking of LTL with Past. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.6257&rep=rep1&type=pdf

[14] Tutorial I: An Introduction to Model Checking. http://www.lix.polytechnique.fr/comete/seminar/1-ModelChecking.ppt

[15] Cavada, Roberto; Cimatti, Alessandro; Keighren, Gavin; Olivetti, Emanuele; Pistore, Marco; Roveri, Marco: NuSMV 2.5 Tutorial. http://nusmv.fbk.eu/NuSMV/tutorial/v25/tutorial.pdf

[16] Lange, Dr. Martin: Die Logik LTL. http://www.tcs.ifi.lmu.de/lehre/ws-2009-10/temporallogik/kapitel-4-des-skripts

[17] Weinberg, Daniela: Kripke, Berechnungsbaum und temporale Logik. http://www.informatik.hu-berlin.de/top/lehre/WS08-09/se_mc/folien/Kripke_TempLogik.pdf

[18] Usadel, Arne: Modal Logik. http://www.hs-weingarten.de/~ertel/vorlesungen/thinf/seminar-ws0607/ArneUsadel/modallogik.pdf

[19] Wieckowski, Bartosz: Einführung in die Modallogik. http://www-ls.informatik.uni-tuebingen.de/wieckowski/molo.pdf, 2008

[20] Hughes, G.E.; Cresswell, M.J.: A New Introduction to Modal Logic. London New York: Routledge 1996

[21] Oryx: WebHome. http://bpt.hpi.uni-potsdam.de/Oryx/

[22] Buschermöhle, Rlaf; Brörkens, Mark; Brückner, Ingo; Damm, Werner; Hasselbring, Wilhelm; Josko, Bernhard; Schulte, Christoph; Wolf, Thomas: Model Checking - Grundlagen und Praxiserfahrungen. Informatik Spektrum 27.2, 146-158 (2004)

[23] Barnat, Jirí; Brázdil, Tomás; Krcál, Pavel; Rehák, Vojtech; Safránek, David: Model Checking in IPv6 Hardware Router Design. Technical Report 8/2002, CESNET, 2002

[24] Salnikow, Alex: Kurz & Gut: Model Checking (Grundlagen und Motivation). http://www.se.uni-hannover.de/lehre/kurz-und-gut.php, 2006

[25] Lohmann, Niels: State Space Reduction Techniques to Verify Business Processes. http://tt2.hpi.uni-potsdam.de/de/archive/series/overview/732/, 2010

[26] Sinz, Carsten Dr.: CTL/LTL Model Checking. http://verialg.iti.kit.edu/download/MC-04.pdf, 2009

[27] Spin - Formal Verification. http://spinroot.com

[28] Holzmann, Gerard J.: The Spin Model Checker: Primer and Reference Manual. AT&T Bell Labs Murray Hill New Jersey: Addison-Wesley 2003

[29] Katoen, Joost-Pieter: Concepts, Algorithms, and Tools for Model Checking. Friedrich-Alexander Universität Erlangen-Nürnberg 1998, http://www.scribd.com/doc/49983903/20/The-model-checker-SMV

[30] Der Model Checker SMV. http://swt.cs.tu-berlin.de/lehre/atswt/ss03/SMV_Komplett_new.pdf

[31] Ken McMillan's Home Page at UCB. http://www.kenmcmil.com

[32] Litz, Lothar: Grundlagen der Automatisierungstechnik. München: Oldenbourg Wissenschaftsverlag GmbH 2005

[33] NuSMV Homepage. http://nusmv.fbk.eu/

[34] Liste von Model Checkern. http://www.pst.ifi.lmu.de/~hammer/mc-list.html

[35] RuleBase Parallel Edition. http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/

[36] FoCs. https://www.research.ibm.com/haifa/projects/verification/focs/index.html

[37] Brambilla, Marco; Deutsch, Alin; Sui, Liying; Vianu, Victor: The Role of Visual Tools in a Web Applicaion Design and Verification Framework: A Visual Notation

for LTL Formulae. In: Lecture Notes in Computer Science(3597, pp 233-243). Springer Verlag 2005

[38] Kuthz, Lars: Model Checking Finite Paths and Trees. http://react.cs.uni-sb.de/publications/diss-kuhtz.pdf, 2010

[39] Heinemann, Bernhard; Weihrauch, Klaus: Logik für Informatiker. Stuttgart: B. G. Teubner 1991

[40] Tscheschner, Willi: Bachelorarbeit: Oryx Dokumentation. http://oryx-editor.googlecode.com/files/Bachelor%20thesis%20Willi%20Tscheschner.pdf, 2007

[41] Peters, Nicolas: Oryx Stencil Set Specification. http://code.google.com/p/oryx-editor/downloads/detail?name=OryxSSS.pdf&can=2&q=, 2007

[42] Peters, Nicolas; Tscheschner, Willi: Oryx - Introduction. http://bpt.hpi.uni-potsdam.de/pub/Public/StudentsCorner/oryx-workshop.architecture-stencilsets-plugins.pdf, 2009

[43] I18N - oryx-editor - Internationalization of Oryx. http://code.google.com/p/oryx-editor/wiki/I18N

[44] Wagner-Boysen, Sven: How to develop an editor plugin. http://code.google.com/p/oryx-editor/wiki/HowToDevelopAnEditorPlugin

Erklärung

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

_____

(Robert Heinz)