Fakultät II – Department für Informatik

Diplomarbeit

Model-Checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten

Vorgelegt von: Roland Meyer
Erstprüfer: Prof. Dr. Ernst-Rüdiger Olderog
Zweitprüfer: Dipl.-Inform. Jochen Hoenicke

Oldenburg, den 9. August 2005

Zusammenfassung

Der Nachweis des korrekten Verhaltens von Software- und Hardwaresystemen ist zu einem integralen Bestandteil des Entwicklungsprozesses geworden. Für große Systeme wird die Methode des Testens exemplarischer Systemabläufe aufgrund der großen Menge möglichen Verhaltens ineffizient. Model-Checking ist der automatisierte Nachweis geforderter Eigenschaften. Dabei wird jedes mögliche Systemverhalten betrachtet. In dieser Arbeit wird ein Model-Checking Verfahren entwickelt, das in der Intervall-basierten Logik Duration Calculus spezifizierte Eigenschaften für Systeme nachweist, die mit Phasen-Event-Automaten modelliert sind. Mit der Implementierung eines Tools und der Anwendung auf eine Fallstudie wird die Praxistauglichkeit der Technik nachgewiesen.

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die es mir ermöglicht haben, diese Diplomarbeit zu schreiben. Mein besonderer Dank gilt dabei Professor Dr. Olderog für die Betreuung während der Diplomarbeit aber auch für die Betreuung im Laufe meines Studiums. Für die Beantwortung meiner Fragen, für die Ratschläge und Hilfen möchte ich mich herzlich bedanken. Ebenfalls gebührt mein Dank Jochen Hoenicke, der mich in den Bereich der Phasen-Event-Automaten eingeweiht und mir das Model-Checking näher gebracht hat. Für die vielen Hilfestellungen, Korrekturen und Kommentare bedanke ich mich. Mein Dank gilt auch Johannes Faber, der mir die Möglichkeit bot, die in dieser Arbeit entwickelten Ansätze im Rahmen meiner AVACS-Tätigkeit an Fallstudien zu überprüfen.
Für die Korrekturen einer ersten Version dieser Diplomarbeit danke ich Jochen Hoenicke, Timo Warns und meiner Schwester Sylvia Meyer.

Insbesondere möchte ich mich bei meiner Freundin Katrin Lambertus und bei meiner Mutter Lydia Meyer bedanken, die mir immer Halt und Kraft für mein Studium gegeben haben.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Model-Checking mit Testautomaten . . . 1
1.2 Überblick . . . 4

2 Grundlagen **7**
2.1 Duration Calculus . . . 7
2.1.1 Symbole . . . 8
2.1.2 Zeitabhängige Zustandsausdrücke . . . 9
2.1.3 Terme und Formeln . . . 11
2.1.4 Gültigkeit, Erfüllbarkeit und Eigenschaften . . . 13
2.1.5 Events . . . 15
2.1.6 Liveness . . . 16
2.2 Phasen-Event-Automaten . . . 18
2.2.1 Definitionen . . . 18
2.2.2 Eigenschaften . . . 21
2.3 Tools . . . 23
2.3.1 Das Paket `pea` . . . 24
2.3.2 Das Tool *Moby/PEA* . . . 26

3 Testformeln **27**
3.1 Die Formelklasse *Testform* . . . 27
3.2 Sync-Events . . . 30
3.3 Normalform . . . 33

4 Testautomaten **43**
4.1 Semantische Anpassung . . . 43
4.2 Testautomaten . . . 45
4.3 Potenzmengenkonstruktion . . . 47
4.4 Testautomatensemantik . . . 55
4.5 Model-Checking von Testformeln mit Testautomaten . . . 66

5 Entwicklung eines Testformel-Compilers **69**
5.1 Anforderungsanalyse . . . 69
5.1.1 Analyse der Rahmenbedingungen . . . 69
5.1.2 Funktionale und nichtfunktionale Anforderungen . . . 70

5.2 Entwurf . . . 73
5.2.1 Datenmodellierung . . . 74
5.2.2 Architektur-Übersicht . . . 80
5.2.3 Statische Systemsicht . . . 82
5.2.4 Dynamische Systemsicht . . . 88
5.3 Implementierung, Dokumentation und Test . . . 89
5.4 Weiterentwicklung . . . 91

6 Fallstudie: Treatment of Emergency Messages 97
6.1 Modell . . . 97
6.1.1 Überblick . . . 97
6.1.2 Komponenten . . . 98
6.2 Spezifikation . . . 102
6.3 Model-Checking mit *Uppaal* . . . 104
6.4 Verifikationsergebnisse . . . 107

7 Zusammenfassung und Ausblick 111

Literaturverzeichnis 115

Index 119

Abbildungsverzeichnis

2.1 Bahnübergang . . . 9
2.2 Interpretation $\mathcal{I}(TP)$ . . . 9
2.3 Interpretation $\mathcal{I}[\![\neg(TP = crs)]\!]$ . . . 11
2.4 Beispiel eines Phasen-Event-Automaten . . . 20
2.5 Paket `pea` . . . 24
2.6 Graphischer Editor und XML-Export von *Moby/PEA* . . . 26

4.1 Interpretation $\mathcal{I}(sndAprMsg)$ . . . 44
4.2 Testautomaten (schematische Darstellung) . . . 55

5.1 Legende der XML-Schema Operatoren . . . 74
5.2 Datentyp TRACE . . . 75
5.3 Datentyp FORMULATREE . . . 76
5.4 Datentyp TESTFORM . . . 77
5.5 Datentyp MCFORM . . . 78
5.6 Datentyp PEA . . . 79
5.7 Paket `pea.modelchecking` . . . 80
5.8 Funktionalität des Compilers und Format der Daten . . . 82
5.9 Normalform-Compiler Klassen . . . 84
5.10 Aktivitätsdiagramm zur Methode `buildNF` . . . 84
5.11 Änderung des Formelbaumes durch `changeNodeSyncChildAnd` . . . 85
5.12 Die Klasse `Trace2PEACompiler` . . . 86
5.13 Converter Klassen . . . 87
5.14 UML 2.0 Sequenzdiagramm: Kompilationsvorgang . . . 94
5.15 UML 2.0 Sequenzdiagramm: Berechnung der Normalform . . . 95
5.16 UML 2.0 Sequenzdiagramm: Berechnung der Testautomatensemantik . . . 96

6.1 Komponenten der Fallstudie, [Fab05] . . . 98
6.2 Phasen-Event-Automat $TrainEM_{CSP}$ . . . 99
6.3 Phasen-Event-Automat $TrainEM_{DC}$ . . . 100
6.4 Phasen-Event-Automat $TrainReact_{CSP}$ . . . 100
6.5 Phasen-Event-Automat $ComNW_{CSP}$ . . . 101
6.6 Phasen-Event-Automat RBC_{CSP} . . . 102
6.7 Phasen-Event-Automat *Environment* . . . 102
6.8 Testautomat $\mathcal{P}(TF_2)$ . . . 104

6.9 Vorgehen beim Model-Checking mit *Uppaal* 105
6.10 In *Uppaal* geladenes, parallel komponiertes System 108
6.11 Model-Checking mit dem Tool *Verifier* 109

1 Einleitung

Diese Einleitung dient als Einführung in das Gebiet des Model-Checkings, insbesondere des Model-Checkings mit Testautomaten. Ein Überblick über die Diplomarbeit schließt sich an.

1.1 Model-Checking mit Testautomaten

In fast jedem technischen Gegenstand des täglichen Lebens befinden sich eingebettete Hardware- und Softwaresysteme und nicht selten hängen Menschenleben von der korrekten Funktionsweise dieser Systeme ab. Es wird daher ein Verfahren benötigt, um nachzuweisen, dass ein System erwartetes Verhalten zeigt.

Die Methode des Testens installierter Systeme beobachtet einen Teil des möglichen Systemverhaltens und prüft, ob dieses Verhalten den Erwartungen entspricht. Das Testen ist ein integraler Bestandteil des Entwicklungsprozesses, dennoch wird die Effizienz angezweifelt:

> "Although provably effective in the very early stages of debugging, when the design is still infested with multiple bugs, their [test and simulation] effectiveness drops quickly as the design becomes cleaner, and they require an alarmingly increasing amount of time to uncover the more subtle bugs."(Pnueli, 1999, [Cla99])

In einem stabilen System gewinnt die Frage nach der Überdeckung des möglichen Verhaltens durch Tests an Relevanz. In sehr großen Systemen mit einer großen Spanne an möglichem Verhalten wird der durch Tests abgedeckte Bereich des Verhaltens im Vergleich zur Menge allen möglichen Verhaltens sehr klein sein.

Der Vorteil des Testens ist die Arbeit am tatsächlich vorliegenden System. Es ist nicht notwendig, das System durch ein Modell nachzubilden, die Resultate bedürfen keiner Interpretation. Andererseits verhindert die Notwendigkeit einer Implementierung, dass das Testen in frühen Phasen des Entwicklungsprozesses eingesetzt werden kann, um die Architektur des Systems zu evaluieren. Diesem Problem kann jedoch mit der Erstellung von Prototypen begegnet werden, die sich ihrerseits überprüfen lassen. Obwohl das Testen großer Systeme als ineffizient beklagt wird, gibt es keine Beschränkung in der Größe der testbaren Systeme.

Der *state of the art* Ansatz in der Verwendung formaler Methoden ist, ein geeignetes Systemmodell zu erstellen und nachzuweisen, dass die geforderte Eigenschaft für das Systemmodell gültig ist. Das Problem mit dieser Technik ist jedoch, dass für Systeme von industriell relevanter Größe keinerlei Methoden verfügbar sind, die das Modell und

das System auf formalem Wege in Beziehung setzen. Das Modell stellt eine Abstraktion des Systems dar und die Eigenschaft ist im System nur im Hinblick auf die Abstraktion gültig .
Mit Model-Checking Verfahren kann der Nachweis von Eigenschaften für Systemmodelle automatisiert werden. Es gibt eine beachtliche Anzahl an Model-Checking Techniken und es ist eine Frage sowohl des Modells als auch der Eigenschaft, welches Verfahren anwendbar ist. Model-Checking mit Testautomaten ist eine Methode, die nicht auf eine Klasse von Systemmodellen beschränkt ist. Ein Testautomat wird verwendet, um das zu untersuchende Systemverhalten auszudrücken. Dieser Automat wird mit dem System parallel komponiert und beobachtet das Systemverhalten. Wenn das System das Verhalten aufweist, das der Testautomat widerspiegelt, geht er in seinen Endzustand über. Der Endzustand ist nicht erreichbar, wenn das Verhalten des Testautomaten vom System vermieden wird.
Im Folgenden werden zunächst die Begriffe des Systemmodells und der Spezifikation näher vorgestellt. Anschließend wird die Idee des Model-Checkings mit Testautomaten verdeutlicht.

Systemmodelle

Die als kritisch dargestellten Computersysteme sind durch ununterbrochene Interaktion mit ihrer Umwelt gekennzeichnet. Die Umgebungswerte werden mittels Sensoren ausgelesen und mittels Aktuatoren beeinflusst. Aufgrund dieses Verhaltens werden Systeme der Klasse als reaktiv bezeichnet.
Model-Checking ist als automatisches Beweisverfahren zu verstehen, das auf ein Modell eines Systems, d.h. eine Darstellung des Systems in einer formalen Sprache, angewandt wird. Die erste Tätigkeit zur Verifikation eines Systems ist daher die Erstellung eines geeigneten Modells des Systems. Automaten mit Zuständen und Zustandsübergängen stellen ein geeignetes Modell für reaktive Systeme dar.
Eigenschaften werden immer nur für das Modell eines Systems nachgewiesen. Es ist daher wichtig, dass das Modell und das installierte System derart im Verhältnis stehen, dass Aussagen über das Modell auf Systemebene Gültigkeit besitzen. Aus diesem Grund sollte das Modell den Zustandsraum des Systems umfassen und das System so feingranular wie möglich repräsentieren. Wenn das Modell Zustände ausließe, so könnten für das Modell Eigenschaften als wahr angenommen werden, die im System durch Abläufe mit nicht im Modell repräsentierten Zuständen verletzt werden.
In der Realität steht die Forderung der feinen Granularität eines Systemmodells in direktem Konflikt mit der handhabbaren Größe des Modells. Es lassen sich zur Zeit Modelle mit einer Größe von ungefär 10^{120} Zuständen automatisch verifizieren. Mit der Nutzung von Variablen und mehreren nahezu unabhängigen Systemkomponenten ist dies eine sehr enge Grenze des Model-Checkings.
Obwohl die Verifikation auf einer Automatendarstellung des Systems vorgenommen wird,

wird das Systemmodell häufig in einer sogenannten Modellierungssprache entwickelt. Für diese Modellierungssprachen ist eine Abbildung, genannt Semantik, anzugeben, die jedem Modell in der Modellierungssprache ein Automatenmodell zuordnet. Das in dieser Diplomarbeit betrachtete Automatenmodell der Phasen-Event-Automaten dient in der Dissertation [Hoe05a] von Jochen Hoenicke als Semantik für die von ihm entwickelte Modellierungssprache CSP-OZ-DC.

Systemspezifikationen

Wie das System sind auch die Eigenschaften, die das System erfüllen soll, in einer wohldefinierten syntaktischen Notation anzugeben, einer sogenannten Spezifikationssprache. Für jede dieser Eigenschaften ist zu definieren, was es bedeutet, dass das System die Eigenschaft erfüllt. Dabei bezieht sich der Begriff der Erfüllbarkeit meist auf die Automatendarstellung des Systems. Amir Pnueli erkannte 1977 als erster, dass für reaktive Systeme sogenannte Verhaltenseigenschaften von besonderem Interesse sind. Diese Eigenschaften, ausgedrückt als logische Formeln, beschreiben die Pfade des möglichen Systemverhaltens. Während in den Anfängen des Model-Checkings die temporalen Logiken CTL oder CTL* untersucht wurden, soll in dieser Arbeit ein Verfahren entwickelt werden, um automatisch durch Formeln der Intervall-basierten Logik Duration Calculus dargestellte Eigenschaften für Systeme nachzuweisen.

Model-Checking mit Testautomaten

Die Entstehung des Model-Checkings wird häufig mit dem Artikel [Cla86] von Clarke, Emerson und Sistla in Verbindung gebracht. In diesem Paper gelang der automatische Nachweis von Eigenschaften, spezifiziert in der temporalen Logik CTL, für endliche Transitionssysteme. Die Grundidee dabei ist, dass, im Falle eines endlichen Modells, die Eigenschaften eines Systems aus dem Systemmodell ableitbar sind. Die Verifikation bedarf so keiner Einwirkung von außen, auch bezeichnet als Push-Button-Verification.
Um zu überprüfen, ob eine Eigenschaft gilt, wird alles mögliche Systemverhalten betrachtet.

> "The questions of adequate coverage or a missed behaviour become irrelevant" (Pnueli, 1999, [Cla99])

Ist die zu untersuchende Eigenschaft verletzt, liefert der Verifikationsalgorithmus, Model-Checker genannt, ein Gegenbeispiel möglichen Verhaltens, das die Spezifikation verletzt.

Die Analyse erreichbarer Zustände ist als Basisfunktionalit eines Model-Checkers zu betrachten. Soll eine Formel F in einer dem Model-Checker unbekannten Logik überprüft werden, so wird ein Automat $\mathcal{P}(\neg F)$, genannt Testautomat, erzeugt. Der Testautomat ist gerade derart gestaltet, dass die Formel F von einem Modell Ph erfüllt wird genau dann, wenn die Parallelkomposition $Ph \parallel \mathcal{P}(\neg F)$ einen gewissen Zustand in $\mathcal{P}(\neg F)$ nicht

erreichen kann. Dabei bedeutet die Parallelkomposition Synchronisation beider Automaten auf das gemeinsame Alphabet.
Ein entscheidender Vorteil in der Benutzung von Testautomaten ist, dass das Produkt $Ph \parallel \mathcal{P}(\neg F)$ von einem Startzustand aus sukzessive aufgebaut werden kann. Wird ein Endzustand des Testautomaten erreicht, so stellt das bis zu diesem Zeitpunkt beobachtete Verhalten ein Gegenbeispiel zu der zu untersuchenden Eigenschaft dar. Das Model-Checking kann an dieser Stelle beendet werden. Wenn das System Ph als Parallelkomposition mehrerer Komponenten gegeben ist, so ist es nicht notwendig, das Produkt Ph vor dem Model-Checking zu berechnen. Es wird die Komposition der Komponenten des Systems Ph mit $\mathcal{P}(\neg F)$ iterativ berechnet. Das Verfahren kann beendet werden, sobald ein Endzustand in $\mathcal{P}(\neg F)$ erreichbar ist. Das Gesamtsystem Ph sowie das Produkt $Ph \parallel \mathcal{P}(\neg F)$ werden nur dann vollständig aufgebaut, falls die Eigenschaft erfüllt ist und kein Gegenbeispiel gefunden werden kann. Diese in der Zeit- und Raumkomplexität sehr effiziente Methode wird nach dem Paper [Cou92] als On-The-Fly Model-Checking bezeichnet.[1]

1.2 Überblick

In dieser Diplomarbeit ist, basierend auf dem Testautomatenansatz, ein Model-Checking Verfahren zu entwickeln, das automatisch als Duration Calculus Formeln spezifizierte Eigenschaften für Systeme nachweist, die mit Phasen-Event-Automaten modelliert sind. Nach der Identifikation einer Klasse der Duration Calculus Formeln, für die Testautomaten konstruiert werden können, ist eine formale Semantik in Form von Testautomaten für Formeln der identifizierten Klasse anzugeben. Abschließend ist die Korrektheit des Model-Checking Verfahrens zu belegen. Nach Erarbeitung des theoretischen Fundaments soll ein Compiler implementiert werden, der die Konstruktion der Testautomaten automatisch vornimmt.

Die Arbeit gliedert sich wie folgt: Nach Definition der grundlegenden Begriffe des Duration Calculus und der Phasen-Event-Automaten in Kapitel 2 wird die Formelklasse *Testform* in Kapitel 3 definiert. Es wird nachgewiesen, dass sich alle Formeln der Klasse in eine gewisse Darstellung überführen lassen, die es erlaubt, Testautomaten für das Model-Checking von Formeln der Klasse zu konstruieren. Kapitel 4 erklärt eine semantischen Anpassung zwischen dem Bereich der Duration Calculus Formeln und dem der Phasen-Event-Automaten. Anschließend werden Testautomaten als Phasen-Event-Automaten mit einem ausgezeichneten Zustand definiert. Nachfolgend wird eine Semantik angegeben, die jeder Testformel in der zum Model-Checking geeigneten Darstellung einen Testautomaten zuordnet. Das Kapitel schließt mit dem Beweis der Korrektheit des

[1] Der einleitende Text erscheint als längeres Paper *Model Checking using Testing* in einem technischen Bericht des Departments für Informatik, Fakultät II, Carl von Ossietzky Universität Oldenburg, [Mey05].

Model-Checking Verfahrens. Die Ergebnisse der Theorie werden in Kapitel 5 durch die Implementierung eines Compilers umgesetzt. Die in Kapitel 6 durchgeführte Fallstudie zeigt die praktische Anwendbarkeit des Verfahrens auf. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf Ansätze für weitere Arbeiten, Kapitel 7.

2 Grundlagen

In der vorliegenden Diplomarbeit wird ein Model-Checking Verfahren erarbeitet und implementiert, welches als Phasen-Event-Automaten (PEAs) modellierte Systeme gegen Formeln der Logik Duration Calculus (DC) verifiziert. Die Formeln des Duration Calculus sind als vom System zu erfüllende Spezifikationen zu betrachten. Nach der Definition dieser Logik wird die formale Definition der PEAs vorgenommen. Das Kapitel schließt mit einem Überblick über die vorhandenen Implementierungen im Rahmen der PEAs: das Java-Paket `pea` und das Werkzeug *Moby/PEA*.

2.1 Duration Calculus

Im Folgenden wird die Intervall-basierte Logik Duration Calculus (DC), dem Vorgehen in [Cha03] folgend, vorgestellt. Der DC wurde entwickelt, um Eigenschaften von Realzeitsystemen *geeignet* spezifizieren zu können. *Geeignet* heißt in diesem Zusammenhang, dass die Zeit als kontinuierliche Größe nicht durch einen abzählbaren Wertebereich dargestellt wird. Die maßgebliche Beobachtung, die zu der Entwicklung des DC geführt hat, ist, dass sich Zustände eines Realzeitsystems durch Funktionen über die Zeit – dargestellt als reelle Zahlen – beschreiben lassen. Die Funktionen besitzen einen diskreten Wertebereich, es gibt jedoch Erweiterungen des DCs zur Spezifikation hybrider Systeme, die kontinuierliche Wertebereiche zulassen. Um das Verhalten eines Realzeitsystems beschreiben zu können, wird der Ablauf des Systems innerhalb eines Zeitintervalls betrachtet. Zur Festlegung korrekten Verhaltens innerhalb des Intervalls dient insbesondere die Dauer eines Zustandes. Ist die Dauer des Zustandes sicherheitskritisch, so ist zu definieren, dass ein unkritischer Folgezustand eingenommen werden muss. Das Hauptaugenmerk dieser Diplomarbeit ist nicht die Spezifikation von Realzeitsystemen im DC, sondern der automatisierte Nachweis von Systemeigenschaften. Es sei auf die Dissertation von Anders P. Ravn [Rav94] verwiesen, in der er typische Eigenschaften von Realzeitsystemen und die zugehörigen Formalisierungen im DC untersucht.
Der DC entstand im Rahmen des ProCoS – Provably Correct Systems – Projektes, welches im Jahre 1989 ins Leben gerufen wurde. Das letzte Treffen der ProCoS Arbeitsgruppe fand im April 1997 statt.

Nachstehend wird die Definition von Syntax und Semantik des DC vorgenommen (vgl. [Cha03]). Dabei sei die *Zeit* kontinuierlich definiert als $\mathbb{T}ime := \mathbb{R}_{\geq 0}$, die *Menge aller Intervalle* sei angegeben als $([b,e],[b,e[,]b,e],]b,e[) \in \mathbb{I}ntv$, $b \leq e, b, e \in \mathbb{T}ime = \mathbb{R}_{\geq 0}$, wobei $[b,e[$ ein links-geschlossenes, rechts-offenes, $]b,e]$ ein links-offenes, rechts-geschlossenes Intervall darstellt.

2.1.1 Symbole

Im DC sind, wie auch in der Prädikatenlogik, globale Variablen, Funktionen und Prädikate als syntaktische Elemente vorgesehen. Ferner gibt es Observablen – Variablen, denen Funktionen als Werte zugewiesen werden.

Definition 2.1.1 (Syntax der Symbole)
Gegeben seien die folgenden Mengen von Symbolen:

GVar: Eine unendliche Menge von *globalen Variablen* $x, y, z, \ldots$. Sie sind unabhängig von Zeitpunkten oder Zeitintervallen, als Bedeutung wird ihnen eine reelle Zahl zugewiesen.

SVar: Eine unendliche Menge von *Observablen* oder *zeitabhängigen Zustandsvariablen* $X, Y, \ldots$ mit endlichen Datenbereichen $D(X) = \{d_1, \ldots, d_m\}$, $m \in \mathbb{N}$. Sie stellen Abbildungen von $\mathbb{T}ime$ in die entsprechenden Datenbereiche dar. So wird für eine Observable angegeben, welchen Wert sie in welchem Zeitpunkt annimmt.

FSymb: Eine unendliche Menge von *globalen Funktionssymbolen* $f^n, g^m, \ldots$, $n, m \in \mathbb{N}_0$. Ein Symbol f^n repräsentiert eine Funktion $\mathbb{R}^n \to \mathbb{R}$. Diese Funktion ist unabhängig von Zeitintervallen oder Zeitpunkten.

PSymb: Eine unendliche Menge von *globalen Prädikatssymbolen* $p^n, q^m, \ldots$, $n, m \in \mathbb{N}_0$. Einem Prädikatssymbol mit Arität n wird eine Abbildung in den booleschen Bereich $\mathbb{R}^n \to \mathbb{B}$ mit $\mathbb{B} := \{tt, ff\}$ zugewiesen. Die Interpretation von Prädikatssymbolen hängt weder von Zeitpunkten noch Zeitintervallen ab.

□

Um die angedeutete Bedeutung der Symbole präziser zu fassen, wird in der folgenden Definition eine formale Semantik angegeben:

Definition 2.1.2 (Semantik der Symbole)
Die Semantik der DC Symbole ist wie folgt definiert:

- Sei mit $Val := GVar \to \mathbb{R}$ die *Menge aller Belegungen*, d.h. Abbildungen von $GVar$ in die reellen Zahlen, gegeben. Die Semantik einer globalen Variablen $x \in GVar$ ist durch eine *Belegung* $\mathcal{V} \in Val$ als $\mathcal{V}(x) \in \mathbb{R}$ bestimmt.
- Sei eine Observable X vom Typ $D(X)$ gegeben. Dann ist ihre Bedeutung durch eine *Interpretation* $\mathcal{I} : SVar \to (\mathbb{T}ime \to D(X))$ gegeben, die der Observablen eine Abbildung $\mathcal{I}(X) : \mathbb{T}ime \to D$ zuordnet, auch notiert als $X_{\mathcal{I}}$.
- Funktions- und Prädikatssymbolen $f^n \in FSymb$, $p^m \in PSymb$, $m, n \in \mathbb{N}_0$, wird als Semantik eine der Arität entsprechende *Abbildung* $\hat{f}^n : \mathbb{R}^n \to \mathbb{R}$ und $\hat{p}^m : \mathbb{R}^m \to \mathbb{B}$ zugewiesen.

□

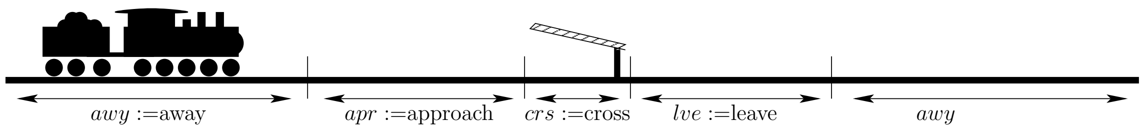

Abbildung 2.1: Bahnübergang

Ein Bahnübergang sei als Beispiel gegeben: Es wird eine Observable namens TP (TrainPosition) eingeführt, welche den Wertebereich $D(TP) := \{apr, crs, lve, awy\}$ besitzt. Die Observable gibt die Position eines Zuges in Bezug auf den Bahnübergang an, dargestellt in Abbildung 2.1. Eine mögliche Interpretation ist die in Abbildung 2.2 angegebene Funktion $\mathcal{I}(TP)$.

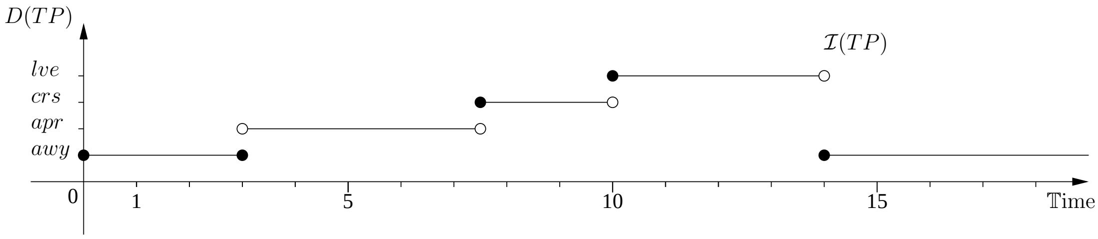

Abbildung 2.2: Interpretation $\mathcal{I}(TP)$

Es sei angemerkt, dass die Wahl der Interpretation willkürlich geschehen ist, eine andere Interpretation $\mathcal{I}'$ könnte eine andere Funktion $\mathcal{I}'(TP)$ vorgeben. Der Vorstellung widerspricht eine Interpretation, die auf *awy* den Funktionswert *crs* folgen lässt, ohne dass ein *apr*, ein Annähern an den Bahnübergang, stattgefunden hat. Dennoch ist eine solche Interpretation der Observablen erlaubt, es bedarf einer Spezifikation des Systems, um derartiges Verhalten zu verbieten.

2.1.2 Zeitabhängige Zustandsausdrücke

Der DC ist als Erweiterung der Intervall-basierten Logik *IL* aufzufassen, [Cha03]. In *IL* gibt es eine Menge *TVar* von temporalen Variablen, welchen als Semantik eine Abbildung von der Menge aller Intervalle in die reellen Zahlen gegeben ist. Im DC sollen diese Variablen genutzt werden, um die Dauer gewisser Zustände bestimmen zu können. Es werden *zeitabhängige Zustandsausdrücke* eingeführt, mit denen der Zustand, dessen Dauer zu messen ist, charakterisiert wird. Über die Zustandsausdrücke wird (Riemann-) integriert. Die syntaktische Kombination aus Integraloperator $\int$ und Zustandsausdruck kann als Menge *TVar* des DC aufgefasst werden. Durch den Integraloperator ist die Semantik mittels der Semantik des Zustandsausdrucks festgelegt.

Definition 2.1.3 (Syntax zeitabhängiger Zustandsausdrücke)
Sei $X \in SVar$ vom Typ $D(X) = \{d_1, \ldots, d_m\}$, $m \in \mathbb{N}$. Die Menge der *zeitabhängigen*

Zustandsausdrücke ist definiert durch

$$\varphi := 0 \mid 1 \mid X = d_i \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2, \tag{2.1}$$

wobei $1 \leq i \leq m$ gilt und φ_1, φ_2 Zustandsausdrücke sind. Die logischen Operatoren $\vee, \Rightarrow, \Leftrightarrow$ werden als Kurzschreibweisen verwendet. □

Zeitabhängige Zustandsausdrücke können intuitiv als Abbildungen aufgefasst werden, die zu einem Zeitpunkt angeben, ob ein gewisser Zustand erreicht ist. Der Idee folgend wird einem Ausdruck φ als Semantik eine Abbildung $\mathcal{I}[\![\varphi]\!] : \mathbb{T}ime \to \{0,1\}$ zugeordnet.

Definition 2.1.4 (Semantik zeitabhängiger Zustandsausdrücke)
Gegeben seien ein zeitabhängiger Zustandsausdruck φ und eine Interpretation $\mathcal{I}$. Dann ist die Semantik von φ als $\mathcal{I}[\![\varphi]\!] : \mathbb{T}ime \to \{0,1\}$ induktiv definiert:

$$\mathcal{I}[\![0]\!](t) := 0 \tag{2.2}$$

$$\mathcal{I}[\![1]\!](t) := 1 \tag{2.3}$$

$$\mathcal{I}[\![X = d_i]\!](t) := \begin{cases} 1, \text{ falls } X_{\mathcal{I}}(t) = d_i \\ 0, \text{ sonst} \end{cases} \tag{2.4}$$

$$\mathcal{I}[\![\neg\varphi_1]\!](t) := 1 - \mathcal{I}[\![\varphi_1]\!](t) \tag{2.5}$$

$$\mathcal{I}[\![\varphi_1 \wedge \varphi_2]\!](t) := \begin{cases} 1, \text{ falls } \mathcal{I}[\![\varphi_1]\!](t) = 1 \text{ und } \mathcal{I}[\![\varphi_2]\!](t) = 1 \\ 0, \text{ sonst,} \end{cases} \tag{2.6}$$

wobei $X \in SVar$ vom Typ $D(X) = \{d_1, \ldots d_m\}$, $m \in \mathbb{N}$, und φ_1, φ_2 zeitabhängige Zustandsausdrücke sind. Anstatt $\mathcal{I}[\![\varphi]\!]$ wird auch die Notation $\varphi_{\mathcal{I}}$ genutzt. □

Durch die Wahl des Zielbereichs von $\varphi_{\mathcal{I}}$ als $\{0,1\}$ ist eine natürliche Definition des Integrals über $\varphi_{\mathcal{I}}$ möglich. Dennoch ist nicht jede Funktion mit Werten in $\{0,1\}$ Riemann-integrierbar, wie mit der Dirichlet-Funktion gezeigt werden kann. Es wird daher für Interpretationen endliche Variabilität angenommen, [Cha03].

Definition 2.1.5 (Endliche Variabilität)
Für alle Observablen $X \in SVar$ und alle Interpretationen $\mathcal{I}$ besitzt $\mathcal{I}(X)$ in allen Intervallen $[b,e] \in \mathbb{I}ntv$ endlich viele Unstetigkeitsstellen. □

Mit der Definition der endlichen Variabilität folgt unmittelbar, dass alle zeitabhängigen Zustandsausdrücke Riemann-integrierbar sind:

Bemerkung 2.1.1
Seien eine Observable X und eine Interpretation $\mathcal{I}$ gegeben. Die Funktion $X_{\mathcal{I}}$, aber insbesondere[1] $(X = d_i)_{\mathcal{I}}$ sowie $0_{\mathcal{I}}$ und $1_{\mathcal{I}}$ sind Treppenfunktionen, die Riemann-integrierbar sind. Die Interpretationen der übrigen Zustandsausdrücke lassen sich als Produkte (Konjunktion) bzw. Differenzen (Negation) dieser Funktionen darstellen und sind wegen

[1] mit geeignetem Datenbereich

der Permanenzeigenschaften der Riemann-integrierbaren Funktionen auch Riemann-integrierbar.

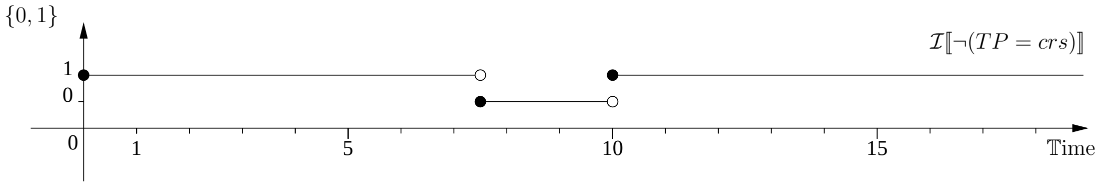

Abbildung 2.3: Interpretation $\mathcal{I}[\![\neg(TP = crs)]\!]$

Als Beispiel sei die Observable TP in der Interpretation $TP_{\mathcal{I}}$ aus Abbildung 2.2 gegeben. Der zeitabhängige Zustandsausdruck $\neg(TP = crs)$ beschreibt den Zeitbereich, in dem sich der Zug nicht auf dem Bahnübergang befindet. Die Semantik $\neg(TP = crs)_{\mathcal{I}}$ des Zustandsausdrucks ist in Abbildung 2.3 angegeben.

2.1.3 Terme und Formeln

Neben globalen Variablen und Funktionen stellen im DC auch Integrale über zeitabhängigen Zustandsausdrücken Terme dar. Damit ist auf syntaktischer Ebene die Möglichkeit gegeben, die Dauer eines Zustandes darzustellen.

Definition 2.1.6 (Syntax der Terme)
Die Klasse der *Terme* $\Theta_1, \Theta_2 \in \mathit{Term}$ ist induktiv definiert:

$$\Theta := x \mid \ell \mid \textstyle\int \varphi \mid f^n(\Theta_1, \ldots, \Theta_n), \tag{2.7}$$

wobei $x \in \mathit{GVar}$, φ ein zeitabhängiger Zustandsausdruck, $f^n \in \mathit{FSymb}$ und $\Theta_1, \ldots, \Theta_n \in \mathit{Term}$ sind. $\square$

Die Bedeutung eines Terms ist eine Abbildung, die einem gegebenen Intervall und einer gegebenen Belegung der globalen Variablen eine reelle Zahl zuordnet. Das Symbol ℓ bezeichnet beispielsweise gerade die Länge des gegebenen Intervalls. Die von *IL* übernommene Auswertung der Terme in Abhängigkeit von gegebenen Intervallen rechtfertigt die Bezeichnung des DC als Intervall-basierte Logik.

Definition 2.1.7 (Semantik der Terme)
Gegeben sei ein Term $\Theta \in \mathit{Term}$ und eine Interpretation der Observablen $\mathcal{I}$. Die Semantik von Termen $\mathcal{I} : \mathit{Term} \to ((\mathit{Val} \times \mathbb{I}\mathit{ntv}) \to \mathbb{R}))$ ordnet jedem Term Θ eine Abbildung $\mathcal{I}[\![\Theta]\!] : \mathit{Val} \times \mathbb{I}\mathit{ntv} \to \mathbb{R}$ zu, welche mit folgenden Gleichungen induktiv definiert ist:

$$\mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) := \mathcal{V}(x) \tag{2.8}$$
$$\mathcal{I}[\![\ell]\!](\mathcal{V}, [b, e]) := e - b \tag{2.9}$$

$$\mathcal{I}[\![\textstyle\int \varphi]\!](\mathcal{V}, [b, e]) := \int_b^e \varphi_{\mathcal{I}}(t)\, dt \tag{2.10}$$

$$\mathcal{I}[\![f^n(\Theta_1, \ldots, \Theta_n)]\!](\mathcal{V}, [b, e]) := \hat{f}^n(\mathcal{I}[\![\Theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\Theta_n]\!](\mathcal{V}, [b, e])), \tag{2.11}$$

wobei $n \in \mathbb{N}_0$, $x \in GVar$, $\mathcal{V} \in Val$, $[b, e] \in \mathbb{I}ntv$, $\Theta_1, \ldots, \Theta_n \in Term$, $f^n \in FSymb$ und φ ein zeitabhängiger Zustandsausdruck sind. □

Die Forderung der endlichen Variabilität sichert die Existenz des Riemann-Integrals und so die Wohldefiniertheit der Semantik.

Prädikate über DC Termen bilden DC Formeln, welche mit booleschen Operatoren und prädikatenlogischen Quantoren verknüpft werden. Aus *IL* ist der Operator Chop übernommen; er fordert, dass zwei Formeln aufeinander folgen: zunächst gilt die erste Formel, anschließend die zweite. Darüber hinaus sind Quantifizierungen über Observablen zugelassen. Diese Erweiterung der in [Cha03] vorgestellten Formelklasse ist in [Tap01] vorgenommen worden. Die Quantifizierung über Observablen dient in Kapitel 3 als elementares Hilfsmittel zum Beweis einer Normalform für eine Teilklasse aller DC Formeln.

Definition 2.1.8 (Syntax der Formeln)
Die Klasse der DC *Formeln* $F, G \in Form$ ist definiert als

$$F := p^m(\Theta_1, \ldots, \Theta_m) \mid \neg F_1 \mid (F_1 \wedge F_2) \mid (F_1 \,;\, F_2) \mid \exists x : F_1 \mid \exists X : F_1, \tag{2.12}$$

wobei $m \in \mathbb{N}_0$, $p^m \in PSymb$, $x \in GVar$, $X \in SVar$, $F_1, F_2 \in Form$ sowie $\Theta_1, \ldots, \Theta_m \in Term$. Die booleschen Operatoren $true, \vee, \Rightarrow, \Leftrightarrow$ sowie der Allquantor sind als Abkürzungen zugelassen. □

Um die Semantik von Formeln mit Existenzquantoren festlegen zu können, wird der Begriff der *Gleichheit* zweier Funktionen *modulo* gewisser Funktionswerte erklärt (vgl. [Tap01]).

Definition 2.1.9 (Gleichheit modulo)
Gegeben seien zwei Abbildung $f, f' : A \to B$. Die beiden Funktionen heißen *gleich modulo* $a \in A$, in Zeichen $f =_{\setminus\{a\}} f'$, falls

$$\forall a' \in A \setminus \{a\} : f(a') = f'(a'). \tag{2.13}$$

□

Da DC Terme in Abhängigkeit von Intervallen und Belegungen ausgewertet werden, ist auch der Wahrheitswert von DC Formeln von Intervallen und Belegungen abhängig. Die Semantik der Observablenquantifizierung ist aus [Tap01] übernommen.

Definition 2.1.10 (Semantik der Formeln)
Gegeben sei eine Interpretation $\mathcal{I}$ der Observablen. Dann ist die Semantik der DC Formeln eine Abbildung $\mathcal{I} : Form \to ((Val \times \mathbb{I}ntv) \to \mathbb{B})$, welche einer Formel F eine

Abbildung $\mathcal{I}[\![F]\!] : Val \times \mathbb{I}ntv \rightarrow \mathbb{B}$ zuordnet, die wie folgt definiert ist:

$$\begin{aligned}
\mathcal{I}[\![p^m(\Theta_1,\ldots,\Theta_m)]\!](\mathcal{V},[b,e]) &:= \hat{p}^m(\mathcal{I}[\![\Theta_1]\!](\mathcal{V},[b,e]),\ldots,\mathcal{I}[\![\Theta_m]\!](\mathcal{V},[b,e])) && (2.14)\\
\mathcal{I}[\![\neg F_1]\!](\mathcal{V},[b,e]) &:= tt \text{ gdw. } \mathcal{I}[\![F_1]\!](\mathcal{V},[b,e]) = \mathit{ff} && (2.15)\\
\mathcal{I}[\![F_1 \wedge F_2]\!](\mathcal{V},[b,e]) &:= tt \text{ gdw. } \mathcal{I}[\![F_1]\!](\mathcal{V},[b,e]) = tt \text{ und}\\
&\quad\ \mathcal{I}[\![F_2]\!](\mathcal{V},[b,e]) = tt && (2.16)\\
\mathcal{I}[\![F_1\,;F_2]\!](\mathcal{V},[b,e]) &:= tt \text{ gdw. es ein } m \in [b,e] \text{ gibt, so dass}\\
&\quad\ \mathcal{I}[\![F_1]\!](\mathcal{V},[b,m]) = tt \text{ und}\\
&\quad\ \mathcal{I}[\![F_2]\!](\mathcal{V},[m,e]) = tt && (2.17)\\
\mathcal{I}[\![\exists x : F_1]\!](\mathcal{V},[b,e]) &:= tt \text{ gdw. es eine Belegung } \mathcal{V}' =_{\setminus\{x\}} \mathcal{V} \text{ gibt, so dass}\\
&\quad\ \mathcal{I}[\![F_1]\!](\mathcal{V}',[b,e]) = tt && (2.18)\\
\mathcal{I}[\![\exists X : F_1]\!](\mathcal{V},[b,e]) &:= tt \text{ gdw. es eine Interpretation } \mathcal{I}' =_{\setminus\{X\}} \mathcal{I} \text{ gibt mit}\\
&\quad\ \mathcal{I}'[\![F_1]\!](\mathcal{V},[b,e]) = tt, && (2.19)
\end{aligned}$$

wobei $m \in \mathbb{N}_0$, $p^m \in PSymb$, $\mathcal{V} \in Val$, $x \in GVar$, $X \in SVar$, $[b,e] \in \mathbb{I}ntv$, $F_1, F_2 \in Form$ und $\Theta_1,\ldots,\Theta_m \in Term$ sind. □

Es haben sich in der Arbeit mit dem DC für häufig benötigte Formeln Abkürzungen etabliert:

Definition 2.1.11 (Abkürzungen)
Seien φ ein zeitabhängiger Zustandsausdruck und F eine Formel.

$$\begin{aligned}
\lceil\rceil &:= \ell = 0 && \text{(Punktintervall)} && (2.20)\\
\lceil\varphi\rceil &:= \textstyle\int\varphi = \ell \wedge \ell > 0 && \text{(fast überall } \varphi\text{)} && (2.21)\\
\Diamond F &:= true\,; F\,; true && \text{(irgendwann } F\text{)} && (2.22)\\
\Box F &:= \neg\Diamond\neg F && \text{(immer } F\text{)} && (2.23)
\end{aligned}$$

□

2.1.4 Gültigkeit, Erfüllbarkeit und Eigenschaften

Duration Calculus Formeln werden genutzt, um das Verhalten von Realzeitsystemen zu spezifizieren. Realzeitsysteme sind reaktive Systeme, die ununterbrochen mit ihrer Umwelt interagieren. Daher ist es wichtig, für diese Systeme nachzuweisen, dass sie sich in jedem Intervall wie gewünscht verhalten. Der Wahrheitswert der Formel soll nur noch von der Interpretation der Observablen, nicht von den Intervallgrenzen abhängen.

Definition 2.1.12 (Erfüllbarkeit und Gültigkeit)
Die Erfüllbarkeits- und Gültigkeitsbegriffe sind wie folgt definiert:

- Eine Formel F *gilt in* $\mathcal{I}, \mathcal{V}, [b,e]$, in Zeichen $\mathcal{I}, \mathcal{V}, [b,e] \models F$, falls $\mathcal{I}[\![F]\!](\mathcal{V},[b,e]) = tt$.

- Eine Interpretation $\mathcal{I}$ *erfüllt* eine Formel F, in Zeichen $\mathcal{I} \models F$, falls $\mathcal{I}, \mathcal{V}, [b, e] \models F$ für alle Belegungen $\mathcal{V}$ und alle Intervalle $[b, e]$ gilt. Wenn es eine Interpretation $\mathcal{I}$ gibt, so dass $\mathcal{I} \models F$ wahr ist, dann ist F *erfüllbar*.

- Eine Interpretation *erfüllt* eine Formel F *von 0 an*, in Zeichen $\mathcal{I} \models_0 F$, falls $\mathcal{I}, \mathcal{V}, [0, t] \models F$ für alle Belegungen $\mathcal{V}$ und alle Zeitpunkte $t \in \mathbb{R}_{\geq 0}$ gilt. Eine Formel heißt *erfüllbar von 0 an*, falls es eine Interpretation gibt, die F von 0 an erfüllt.

- Eine Formel F ist *allgemeingülig*, notiert $\models F$, falls für alle Interpretationen $\mathcal{I}$ gilt: $I \models F$.

□

Nachfolgend wird eine wichtige Eigenschaft der Gültigkeitsrelation nachgewiesen, die in Kapitel 4 benötigt wird. Sie reduziert die Frage, ob eine Formel F in einer gegebenen Interpretation $\mathcal{I}$ im Intervall $[b, e]$ gilt, auf die Frage, ob die Formel F in einer abgewandelten Interpretation $\mathcal{I}_k$ im Intervall $[b - k, e - k]$, $k \in [0, b]$ gilt.

Definition 2.1.13
Gegeben seien eine Interpretation $\mathcal{I}$ und ein Zeitpunkt $k \in \mathbb{T}ime$. Die Interpretation $\mathcal{I}_k$ ist definiert als

$$\mathcal{I}_k(X)(t) := \mathcal{I}(X)(t + k),\ t \in \mathbb{T}ime,$$

für alle $X \in SVar$. □

Das folgenden Lemma verschiebt die Frage der Gültigkeit einer Formeln in einem Intervall in ein anderes Intervall. Es verallgemeinert Lemma 3.1, S.44 in [Cha03].

Lemma 2.1.1
Gegeben seien eine Interpretation $\mathcal{I}$, eine Belegung $\mathcal{V}$, ein Intervall $[b, e]$ und $k \in [0, b]$. Es gilt:

$$\mathcal{I}_k, \mathcal{V}, [b - k, e - k] \models F \text{ gdw. } \mathcal{I}, \mathcal{V}, [b, e] \models F$$

Beweis
Der Beweis folgt analog zu dem Beweis in [Cha03]: Die einzigen interpretationsabhängigen Terme sind Integrale über zeitabhängigen Zustandsausdrücken. Es gilt per Definition von $\mathcal{I}_k$ für alle zeitabhängigen Zustandsausdrücke: $\mathcal{I}[\![\varphi]\!](t + k) = \mathcal{I}_k[\![\varphi]\!](t)$, $t \in \mathbb{T}ime$. Damit lässt sich folgende Gleichungskette herleiten:

$$\int_{b-k}^{e-k} \mathcal{I}_k[\![\varphi]\!](t)dt = \int_{b-k}^{e-k} \mathcal{I}[\![\varphi]\!](t + k)dt = \int_{b}^{e} \mathcal{I}[\![\varphi]\!](t)dt$$

Es sind alle interpretationsabhängigen Terme als gleich nachgewiesen. Damit gilt die Äquivalenz. □

2.1.5 Events

Realzeitsysteme und ihre Umwelt werden im Modell als Automaten mit einer Menge von Zuständen und Transitionen zwischen diesen Zuständen dargestellt. Interaktionen eines Systems mit der Umwelt führen zu Zustandsänderungen, welche auf Automatenebene nachvollzogen werden. Im Modell geschehen Interaktion und Zustandsänderung augenblicklich, ohne Zeitverlust. Die Interaktion wird dabei als Ereignis während der Zustandsänderung, d.h. beim Passieren einer Transition im Automaten, nachgebildet. Die definierten Kurzformen für DC Formeln sind dafür geeignet, Zustände von Systemen zu beschreiben($\lceil\varphi\rceil$, φ Zustandsausdruck). Mit der Definition von Events wird eine Darstellung für Verhalten beim Zustandswechsel eingeführt.
Dem Ansatz in [Cha03] folgend basieren Events auf Transitionsformeln, welche Zustandsübergänge in Realzeitsystemen charakterisieren. Ein Übergang vom Zustand φ zum Zustand $\varphi'(\neq \varphi)$ findet zum Zeitpunkt $t \in \mathbb{R}_{\geq 0}$ statt, falls der Zustand φ eine Zeitspanne bis t stabil war und von t aus eine Zeit lang φ' gilt.

Definition 2.1.14 (Transitionsformeln)
Gegeben seien eine Interpretation $\mathcal{I}$, eine Belegung $\mathcal{V}$, ein Intervall $[b, e]$ und sowie ein zeitabhängiger Zustandsausdruck φ. Die *Transitionsformeln* sind mit folgenden Äquivalenzen definiert:

$$\mathcal{I}, \mathcal{V}, [b, e] \models \Nwarrow\varphi \text{ gdw. } \exists 0 < \delta < b : \mathcal{I}, \mathcal{V}, [b - \delta, b] \models \lceil\varphi\rceil \tag{2.24}$$

$$\mathcal{I}, \mathcal{V}, [b, e] \models \Nearrow\varphi \text{ gdw. } \exists 0 < \delta : \mathcal{I}, \mathcal{V}, [e, e + \delta] \models \lceil\varphi\rceil \tag{2.25}$$

$\square$

Transitionsformeln fordern die Existenz eines gewissen Zustandes vor und nach einem gegebenen Intervall. Um den Zustandswechel in einem Punkt zu beschreiben, muss das betreffende Intervall die Länge Null besitzen.

Definition 2.1.15 (Transitionsformeln)
Gegeben sei ein zeitabhängiger Zustandsausdruck φ.

$$\nwarrow\varphi := \Nwarrow\varphi \wedge \ell = 0 \tag{2.26}$$

$$\nearrow\varphi := \Nearrow\varphi \wedge \ell = 0 \tag{2.27}$$

$\square$

Sind zwei zeitabhängige Zustandsausdrücke φ und φ' gegeben, so lässt sich der Wechsel vom Zustand φ zum Zustand φ' mit der Formel $\nwarrow\varphi \wedge \nearrow\varphi'$ ausdrücken. In Realzeitsystemen ist es oftmals von Bedeutung, auf steigende oder fallende Flanken von Observablen oder Zuständen zu reagieren. Eine steigende Flanke des Zustands φ wird durch $\nwarrow\neg\varphi \wedge \nearrow\varphi$ ausgedrückt.

Definition 2.1.16 (Transitionsformeln)
Gegeben sei ein zeitabhängiger Zustandsausdruck φ.

$$\uparrow\varphi := \nwarrow\neg\varphi \wedge \nearrow\varphi \tag{2.28}$$

$$\downarrow\varphi := \nwarrow\varphi \wedge \nearrow\neg\varphi \tag{2.29}$$

$\square$

Um während eines Zustandsübergangs die Interaktion eines Realzeitsystems mit seiner Umwelt modellieren zu können, werden Events eingeführt. Events werden als boolesche Observablen[2] dargestellt, die den Namen des zu beschreibenden Ereignisses tragen. Das Auftreten eines Events wird als steigende oder fallende Flanke der Observablen definiert.

Definition 2.1.17 (Events)
Gegeben sei eine boolesche Observable $\mathcal{E}$. Das *Event* $\updownarrow\mathcal{E}$ ist definiert als

$$\updownarrow\mathcal{E} := \uparrow(\mathcal{E} = tt) \vee \downarrow(\mathcal{E} = tt) \tag{2.30}$$

$$\not\updownarrow\mathcal{E} := (\nwarrow(\mathcal{E} = tt) \wedge \nearrow(\mathcal{E} = tt)) \vee (\nwarrow(\mathcal{E} = ff) \wedge \nearrow(\mathcal{E} = ff)) \tag{2.31}$$

Das Ausbleiben eines Events wird mit $\boxminus\mathcal{E}$ dargestellt:

$$\boxminus\mathcal{E} := \lceil \mathcal{E} = tt \rceil \vee \lceil \mathcal{E} = ff \rceil \tag{2.32}$$

$\square$

Um spezifizieren zu können, dass bei einem Zustandsübergang ein Event nicht auftreten darf, wurde die Formel $\not\updownarrow\mathcal{E}$ definiert. Sie legt das Nicht-Auftreten eines Events als Ausbleiben der steigenden und fallenden Flanke der Eventobservablen fest. Würde das Ausbleiben des Events als Negation des Events definiert, so entfiele die Forderung des Punktintervalls.

2.1.6 Liveness

Gemäß Skakkebæk in [Ska94] wird die Klasse der DC Formeln in diesem Abschnitt um zwei Chop Operatoren erweitert, die die Spezifikation von Formeln erlauben, die in Intervallen außerhalb des gegebenen Intervalls gelten müssen. Die Erweiterung ist für die Definition von Sync-Events in Kapitel 3, welche zu einer Normalform führen, notwendig.

Definition 2.1.18 (Syntax der Formeln mit Liveness)
Seien F_1 und F_2 Duration Calculus Formeln. Dann sind $F_1 \lhd F_2$ und $F_1 \rhd F_2$ Duration Calculus Formeln. Die Operatoren $\lhd$ und $\rhd$ werden als *Chop* Operatoren bezeichnet. $\square$

Definition 2.1.19 (Semantic der Formeln mit Liveness)
Gegeben seien eine Interpretation $\mathcal{I}$, eine Belegung $\mathcal{V}$ und ein Intervall $[b, e]$. Die Semantik der Chop Operatoren ist durch die folgenden Äquivalenzen definiert:

$$\begin{aligned} \mathcal{I}[\![F_1 \lhd F_2]\!](\mathcal{V}, [b, e]) := tt \text{ gdw. es ein } k \leq b \text{ gibt mit} \\ \mathcal{I}[\![F_1]\!](\mathcal{V}, [k, b]) = tt \text{ und } \mathcal{I}[\![F_2]\!](\mathcal{V}, [k, e]) = tt \end{aligned} \tag{2.33}$$

[2] Observablen mit Datenbereich $\mathbb{B}$

$$\mathcal{I}[\![F_1 \rhd F_2]\!](\mathcal{V}, [b,e]) := tt \text{ gdw. es ein } k \geq e \text{ gibt mit}$$
$$\mathcal{I}[\![F_1]\!](\mathcal{V}, [b,k]) = tt \text{ und } \mathcal{I}[\![F_2]\!](\mathcal{V}, [e,k]) = tt \tag{2.34}$$

□

Der in Definition 2.1.11 vorgestellte irgendwann Operator fordert die Existenz eines Teilintervalls, für welches eine Formel gelten soll, in einem gegebenen Intervall. In der temporalen Logik CTL* sagt der irgendwann Operator aus, dass es einen Punkt in der Zukunft geben soll, zu dem eine Eigenschaft erfüllt ist. Wenn das gegebene Intervall zu kurz ist, liefert obiger irgendwann Operator die Aussage, die Formel sei nicht erfüllt, obwohl es in Zukunft durchaus ein Teilintervall gibt, das kein Teilintervall des gegebenen Intervalls ist, aber die Formel erfüllt. Es wird ein neuer *echter irgendwann* Operator definiert, der die Existenz eines Teilintervalls nicht vom gegebenen Intervall abhängig macht.

Definition 2.1.20 (Echtes irgendwann, echtes immer)
Gegeben sei eine Formel F.

$$\Diamond_L F := (true\,;F\,;true) \rhd true \qquad \text{(echtes irgendwann)} \tag{2.35}$$
$$\Box_L F := \neg \Diamond_L \neg F \qquad \text{(echtes immer)} \tag{2.36}$$

□

Das echte irgendwann ist über Chop Operatoren definiert, welche in Beweisen mühsam zu zerlegen sind. Es wird im nachfolgenden Lemma eine semantische Charakterisierung des Operators bewiesen.

Lemma 2.1.2 (Charakterisierung von irgendwann)
Gegeben seien eine Interpretation $\mathcal{I}$, eine Belegung $\mathcal{V}$, ein Intervall $[b,e]$ und eine Formel F. Es gilt folgende Äquivalenz:

$$\mathcal{I}, \mathcal{V}, [b,e] \models \Diamond_L F \text{ gdw. } \exists m_1 \in [b,\infty[: \exists m_2 \in [m_1,\infty[: \mathcal{I}, \mathcal{V}, [m_1,m_2] \models F$$

Beweis
Der Beweis folgt unmittelbar aus den Definitionen:

$$\begin{aligned}
& \mathcal{I}, \mathcal{V}, [b,e] \models \Diamond_L F \\
\{\text{Definition } \Diamond_L\} \; & \Leftrightarrow \mathcal{I}, \mathcal{V}, [b,e] \models (true\,;F\,;true) \rhd true \\
\{\text{Definition } \rhd\} \; & \Leftrightarrow \exists k \in [e,\infty[: \mathcal{I}, \mathcal{V}, [b,k] \models true\,;F\,;true \text{ und } \mathcal{I}, \mathcal{V}, [e,k] \models true \\
\{\text{Neutr. Elem. Konj}\} \; & \Leftrightarrow \exists k \in [e,\infty[: \mathcal{I}, \mathcal{V}, [b,k] \models true\,;F\,;true \\
\{\text{Definition };\} \; & \Leftrightarrow \exists k \in [e,\infty[: \exists m_1 \in [b,k] : \exists m_2 \in [m_1,k] : \mathcal{I}, \mathcal{V}, [b,m_1] \models true \\
& \quad \text{und } \mathcal{I}, \mathcal{V}, [m_1,m_2] \models F \text{ und } \mathcal{I}, \mathcal{V}, [m_2,k] \models true \\
\{\text{Neutr. Elem. Konj}\} \; & \Leftrightarrow \exists k \in [e,\infty[: \exists m_1 \in [b,k] : \exists m_2 \in [m_1,k] : \mathcal{I}, \mathcal{V}, [m_1,m_2] \models F \\
& \Leftrightarrow \exists m_1 \in [b,\infty[: \exists m_2 \in [m_1,\infty[: \mathcal{I}, \mathcal{V}, [m_1,m_2] \models F
\end{aligned}$$

Die Richtung $\Leftarrow$ der letzten Äquivalenz gilt aufgrund der Unbeschränktheit der reellen Zahlen. □

2.2 Phasen-Event-Automaten

Im Rahmen seiner Dissertation [Hoe05a] hat Jochen Hoenicke in der Abteilung Correct System Design an der Carl von Ossietzky Universität Oldenburg die Spezifikationssprache *CSP-OZ-DC* entwickelt, welche die namhaften Spezifikationssprachen CSP, Objekt Z und Duration Calculus integriert. CSP-OZ-DC Modelle beschreiben sowohl das Verhalten als auch die Daten von Systemen und vermögen es, Realzeitbedingungen für beide Aspekte auszudrücken. Verhalten wird in einem CSP Teil festgehalten und schildert die Interaktion des Systems mit seiner Umwelt. Datentypen und Datenänderungen als Konsequenz von Interaktionen werden mittels eines Objekt Z Teils angegeben. Ein Duration Calculus Part beschreibt Realzeitverhalten, welches das System nicht aufweisen darf. In den Fallstudien des Teilprojektes R des Sonderforschungsbereichs AVACS wird CSP-OZ-DC als Spezifikationssprache eingesetzt, [AVA].
Um Modelle automatisch verifizieren zu können, sind Phasen-Event-Automaten (PEAs) als operationale Semantik für CSP-OZ-DC definiert worden. PEAs sind spezielle Timed Automata, welche sowohl den Datenaspekt des Objekt Z Teils der Spezifikation als auch den Verhaltensaspekt wiederspiegeln. Daten werden durch Bedingungen modelliert, die an Zustände des Automaten geknüpft sind, während Verhaltensaspekte in Form von Events bei der Zustandsänderung wiedergegeben werden.

2.2.1 Definitionen

Die nachstehenden Definitionen für Phasen-Event-Automaten sind aus [Hoe05b] übernommen.

Bemerkung 2.2.1
In PEAs werden Bedingungen als prädikatenlogische Formeln über getypten Variablen ausgedrückt. Es sei eine Grundmenge Var von *Variablen* gegeben. Für eine Variable $x \in Var$ bezeichne $\mathbf{type}(x)$ den *Typ*, $D(\mathbf{type}(x))$ den *Datenbereich* des Typs. Einzig endliche Datentypen[3] sowie die reellen Zahlen sind zugelassen. Über den reellen Zahlen sind nur linear arithmetische Ausdrücke erlaubt. Eine *Belegung* $\beta \in \mathbf{Val}$ ordnet jeder Variablen $x \in Var$ einen Datenwert $\beta(x) \in D(\mathbf{type}(x))$ zu. Dabei bezeichnet **Val** die Menge aller Belegungen von Variablen in Var. Es gilt $\mathbf{Events} \subseteq Var$ mit $\mathbf{type}(\mathcal{E}) = \mathbb{B}$ für alle Events $\mathcal{E} \in \mathbf{Events}$ und $\mathbf{Clocks} \subseteq Var$ mit $\mathbf{type}(c) = \mathbb{T}ime = \mathbb{R}_{\geq 0}$ für alle $c \in \mathbf{Clocks}$.
In der Menge der Variablen ist jede Variable gestrichen und ungestrichen enthalten. Die gestrichene Version $x' \in Var$ einer Variablen $x \in Var$ bezieht sich auf die Belegung nach dem Passieren einer Transition, die ungestrichene Version auf die Belegung vor Beschreiten einer Transition. Mit β' ist eine Belegungen der gestrichenen Variablen $x' \in V'$ gegeben, für die gilt $\beta'(x') = \beta(x)$ für alle $x' \in V'$, $\beta' \in \mathbf{Val}$.
Zu einer Menge von Variablen $V \subseteq Var$ bezeichne $\mathcal{L}(V)$ die Klasse der *prädikatenlogischen Formeln erster Stufe über Variablen in V*. Für eine gegebene Formel ϕ bezeichne

[3] Datentypen mit endlichem Datenbereich

$Var(\phi)$ die enthaltenen freien Variablen. Sei $\beta \in \mathbf{Val}$ eine Belegung der Variablen in $Var(\phi)$. Dass β die Formel erfüllt, wird mit $\beta \models \phi$ notiert.

Definition 2.2.1 (Phasen-Event-Automat)
Ein *Phasen-Event-Automat* ist ein Tupel $Ph = (P, V, A, C, E, s, I, P_0)$, wobei

P eine endliche Menge von *Zuständen* (auch *Phasen* genannt), (2.37)

$V \subseteq Var \backslash (\mathbf{Events} \cup \mathbf{Clocks})$ eine endliche Menge von *Variablen*, (2.38)

$A \subseteq \mathbf{Events}$ eine endliche Menge von *Events*, (2.39)

$C \subseteq \mathbf{Clocks}$ eine endliche Menge von *Uhren* und (2.40)

$E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathbb{P}(C) \times P$ die *Transitionsrelation* ist. (2.41)
Ein Element $(p_1, g, X, p_2) \in E$ stellt eine *Transition* von Phase p_1 zu Phase p_2 dar, wobei der *Guard* g erfüllt ist und alle Uhren in X zurückgesetzt werden.

$s : P \to \mathcal{L}(V)$ ist eine Abbildung, die jeder Phase eine Bedingung bezüglich der Zustandsvariablen zuordnet. Diese *Zustandsinvariante* ist erfüllt, während sich der Automat in der Phase befindet. (2.42)

$I : P \to \mathcal{L}(C)$ ist eine Abbildung, die jeder Phase eine *Zeitbedingung* oder *Uhrenbedingung* (Engl.:clock invariant) zuordnet. (2.43)

$P_0 \subseteq P$ ist eine Menge von *Startzuständen.* (2.44)

Für jeden Zustand $p \in P$ gibt es eine *Stotterkante* $(p, \bigwedge_{i=1}^{n} \neg \mathcal{E}_i \wedge \bigwedge_{j=1}^{m} v_j = v_j', \emptyset, p)$ mit $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\} \subseteq A, \{v_1, \ldots, v_m\} \subseteq V$.

Eine *Konfiguration* eines Phasen-Event-Automaten ist als Tripel (p, β, γ) gegeben. Dabei ist $p \in P$ eine Phase, β eine V-Belegung und γ eine C-Belegung. □

Konfigurationen sind als Momentaufnahmen des Systemzustandes aufzufassen. Dabei setzt sich der Systemzustand zusammen aus dem Zustand im Zustandsraum P (Verhaltensaspekt, bestimmt die möglichen Interaktionen), der Belegung der Variablen in V (Datenaspekt) und der Uhrenbelegung (Realzeitaspekt). Ein Ablauf eines Phasen-Event-Automaten ist eine Folge von Konfigurationen, die beim Passieren der Transitionen entsteht, gepaart mit Informationen über die Dauer der einzelnen Konfigurationen und Events zwischen den Konfigurationen.

Definition 2.2.2
Die Syntax χ_Y bezeichnet die *charakteristische Funktion für* $Y \subseteq \mathbf{Events}$. Es gilt $\chi_Y(\mathcal{E}) = true$ gdw. $\mathcal{E} \in Y$. Sei γ eine Belegung, $t \in \mathbb{T}ime$: der *Timeshift* Operator $\gamma + t$ ist definiert mittels $(\gamma + t)(c) := \gamma(c) + t$. Sei $X \subseteq \mathbf{Clocks}$, dann ist $\gamma[X := 0]$ definiert durch $(\gamma[X := 0])(c) := \gamma(c)$, falls $c \notin X$, $(\gamma[X := 0])(c) := 0$ sonst. □

Definition 2.2.3 (Run)
Ein *Run* (oder *Ablauf*) eines Phasen-Event-Automaten ist eine endliche oder unendliche

Sequenz

$$\langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \tag{2.45}$$

von Konfigurationen, *Dauern* $t_i \in \mathbb{R}_{\geq 0}$ und Events $Y_i \subseteq A$, $i \in \mathbb{N}_0$ derart, dass folgende Bedingungen erfüllt sind:

$$p_0 \in P_0 \tag{2.46}$$
$$\forall c \in C : \gamma_0(c) = 0 \tag{2.47}$$
$$\forall i \in \mathbb{N}_0 : \beta_i \models s(p_i) \tag{2.48}$$
$$\forall i \in \mathbb{N}_0 : \forall 0 \leq \delta \leq t_i : \gamma_i + \delta \models I(p_i) \tag{2.49}$$
$$\forall i \in \mathbb{N}_0 : \exists (p_i, g, X, p_{i+1}) \in E :$$
$$\beta_i \cup \beta'_{i+1} \cup (\gamma_i + t_i) \cup \chi_{Y_i} \models g \land \tag{2.50}$$
$$\gamma_{i+1} = (\gamma_i + t_i)[X := 0] \tag{2.51}$$

Ein Zustand $p \in P$ heißt *erreichbar* in einem Run r, falls es eine Konfiguration (p, β, γ) in r gibt. Die *Menge aller Abläufe* eines PEA Ph wird mit $\mathbf{Run}(Ph)$ bezeichnet. □

Die Definition fordert, dass alle Abläufe in einem Startzustand beginnen, (2.46), und dass die Uhren initial auf Null gesetzt sind, (2.47). In jeder Konfiguration erfüllt die Belegung der Observablen die Zustandsinvariante, (2.48). Uhrenbedingungen müssen zu jedem Zeitpunkt über die gesamte Dauer einer Konfiguration gelten, (2.49). Zeile 2.50 besagt, dass der Guard einer jeden Kante erfüllt werden muss, dabei sind die Belegungen vor dem Beschreiten einer Transition, die Belegungen im Anschluss, die Uhrenbelegungen und die Events während des Übergangs von Bedeutung. Definition 2.51 gibt an, dass sich die Belegung einer Uhr in einer Folgekonfiguration als Summe der Belegung in und der Dauer der vorangegangenen Konfiguration ergibt. Die Belegung ist Null, falls die Uhr in der Menge X der zurückgesetzten Uhren enthalten ist.

Abbildung 2.4 zeigt einen Phasen-Event-Automaten. Die einzig vorhandene Variable

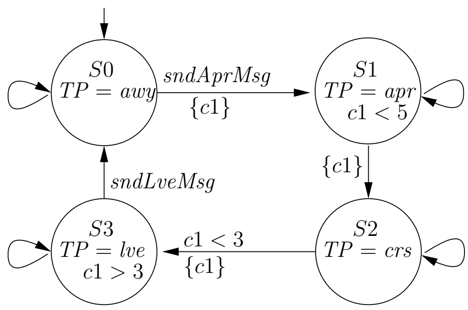

Abbildung 2.4: Beispiel eines Phasen-Event-Automaten

in V ist TP, es treten die Events *sndAprMsg* und *sndLveMsg* auf. Ein möglicher Run könnte mit der Folge

$$\langle (S0, \beta_0(TP) = awy, \gamma_0(c1) = 0), 5, \{sndAprMsg\}, (S1, \beta_1(TP) = apr, \gamma_1(c1) = 0), \ldots \rangle$$

von Konfigurationen, Dauern und Eventmengen beginnen.

Die PEA Semantik von CSP-OZ-DC ist kompositionell definiert, d.h. für jeden Spezifikationsaspekt wird ein PEA als Semantik erstellt. Die Semantik des Gesamtsystems ergibt sich aus der Verknüpfung der einzelnen PEAs für CSP, OZ und DC Teil. Die Verknüpfung ist als Parallelkomposition erklärt, welche die Synchronisation aller Automaten auf gemeinsame Observablen und gemeinsame Events fordert.

Definition 2.2.4 (Parallelkomposition)
Seien zwei PEAs $Ph_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, P_{0,i})$, $i \in \{1, 2\}$, mit $C_1 \cap C_2 = \emptyset$ gegeben. Dann ist die *Parallelkomposition* $Ph_1 \parallel Ph_2 := (P, V, A, C, E, s, I, P_0)$ definiert durch:

$$P := P_1 \times P_2 \qquad \text{(Kartesisches Produkt der Zustände)} \tag{2.52}$$
$$V := V_1 \cup V_2 \qquad \text{(Vereinigung der Variablenmengen)} \tag{2.53}$$
$$A := A_1 \cup A_2 \qquad \text{(Vereinigung der Eventmengen)} \tag{2.54}$$
$$C := C_1 \cup C_2 \qquad \text{(Vereinigung der disjunkten Uhrenmengen)} \tag{2.55}$$
$$s((p_1, p_2)) := s_1(p_1) \wedge s_2(p_2) \qquad \text{(Konjunktion der Zustandsinvarianten)} \tag{2.56}$$
$$I((p_1, p_2)) := I_1(p_1) \wedge I_2(p_2) \qquad \text{(Konjunktion der Uhrenbedingungen)} \tag{2.57}$$
$$P_0 := P_{(0,1)} \times P_{(0,2)} \qquad \text{(Kartesisches Produkt der Startzustände)} \tag{2.58}$$

$$E := \{((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p'_1, p'_2)) \mid (p_1, g_1, X_1, p'_1) \in E_1 \wedge (p_2, g_2, X_2, p'_2) \in E_2\}. \tag{2.59}$$

□

2.2.2 Eigenschaften

In diesem Abschnitt werden Eigenschaften von Phasen-Event-Automaten, die später benötigt werden, aufgezeigt.

Lemma 2.2.1 besagt, dass in einem Lauf eine Konfiguration mit Dauer t in viele Konfigurationen kürzerer Dauern zerlegt werden kann, indem die Stotterkante genutzt wird. Mit dieser Eigenschaft ist es einzelnen Automaten einer Parallelkomposition möglich, Werte von Variablen, die nicht in beiden Variablenmengen enthalten sind, asynchron zu ändern. Der passive Automat benutzt die Stotterkante, während der aktive Automat die Variablenbelegung ändert.

Lemma 2.2.1 (Lemma1, [Hoe05b])
Gegeben seien ein PEA $Ph = (P, V, A, C, E, s, I, P_0)$ sowie der Ablauf

$$r := \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \in \mathbf{Run}(Ph)$$

Für alle $i \geq 0$ und alle $0 < \delta < t_i$ erhält man einen Ablauf $r' \in \mathbf{Run}(Ph)$, indem man die Sequenz $\langle (p_i, \beta_i, \gamma_i), t_i, Y_i \rangle$ durch $\langle (p_i, \beta_i, \gamma_i), \delta, \emptyset, (p_i, \beta_i, \gamma_i + \delta), t_i - \delta, Y_i \rangle$ ersetzt.

Wird aus zwei PEAs die Parallelkomposition gebildet, so ergibt sich aus synchronen Runs der einzelnen PEAs ein Run der Parallelkomposition. Synchron bedeutet, die Runs stimmen in der Belegung der gemeinsamen Variablen und in dem Auftreten der Events überein. Das Lemma fordert dabei, dass die Dauern der Konfigurationen in beiden Runs übereinstimmen. Mit Lemma 2.2.1 stellt dies keine Einschränkung der gegebenen Abläufe dar.

Lemma 2.2.2
Seien mit $Ph_l = (P_l, V_l, A_l, C_l, E_l, s_l, I_l, P_{0,l})$, $l \in \{1, 2\}$, zwei Phasen-Event-Automaten gegeben mit $V_1 \cap V_2 =: V_s$, $A_1 \cap A_2 =: A_s$ und $C_1 \cap C_2 = \emptyset$. Seien die Abläufe

$$r_l = \langle (p_{0,l}, \beta_{0,l}, \gamma_{0,l}), t_0, Y_{0,l}, (p_{1,l}, \beta_{1,l}, \gamma_{1,l}), t_1, Y_{1,l}, \ldots \rangle \in \mathbf{Run}(Ph_l)$$

gegeben und gelte mit $X \in V_s$, $\mathcal{E} \in A_s$:

$$\begin{aligned} &\beta_{i,1}(X) = \beta_{i,2}(X) && \text{für alle } i \geq 0 \\ &\mathcal{E} \in Y_{i,1} \Leftrightarrow \mathcal{E} \in Y_{i,2} && \text{für alle } i \geq 0 \end{aligned}$$

Dann gibt es einen Ablauf $r \in \mathbf{Run}(Ph_1 \parallel Ph_2)$.

Bemerkung 2.2.2
Erreicht Ph_1 in dem Ablauf r_1 einen Zustand $p_{k,1}$, $k \geq 0$, zum Zeitpunkt $\sum_{i=0}^{k-1} t_{i,1}$, dann erreicht auch r einen Zustand $(p_{k,1}, p_{k,2})$ zu dem Zeitpunkt.

Beweis
Betrachte den Ablauf

$$r := \langle ((p_{0,1}, p_{0,2}), \beta_{0,1} \cup \beta_{0,2}, \gamma_{0,1} \cup \gamma_{0,2}), t_0, Y_{0,1} \cup Y_{0,2}, \ldots \rangle$$

von $Ph_1 \parallel Ph_2$. Die Vereinigung $\beta_{k,1} \cup \beta_{k,2}$, $k \geq 0$, ergibt eine Abbildung, d.h. die Rechtseindeutigkeit bleibt erhalten, da $\beta_{k,1}(X) = \beta_{k,2}(X)$, $X \in V_s$. Für $\gamma_{k,1} \cup \gamma_{k,2}$ gilt die Rechtseindeutigkeit, da die Uhrenmengen disjunkt sind.
Dass r ein Ablauf von $Ph_1 \parallel Ph_2$ ist, wird wie folgt nachgewiesen:
$p_{0,l} \in P_{0,l}$ und $l \in \{1, 2\}$, also ist $(p_{0,1}, p_{0,2}) \in P_{0,1} \times P_{0,2} =: P_0$. Für alle $c \in C_l$ gilt $\gamma_{0,l}(c) = 0$, $l \in \{1, 2\}$. Da $\beta_{k,l} \models s(p_{k,l})$ und $s(p_{k,1})$ die Variablenbelegungen für $V_2 \setminus V_1$ bzw. $s(p_{k,2})$ die Belegungen für $V_1 \setminus V_2$ nicht einschränkt, gilt $\beta_{k,1} \cup \beta_{k,2} \models s(p_{k,1}) \wedge s(p_{k,2})$. Analog wird für die Invarianten der Uhren argumentiert.
Betrachte den Übergang von $((p_{k,1}, p_{k,2}), \beta_{k,1} \cup \beta_{k,2}, \gamma_{k,1} \cup \gamma_{k,2})$ zu $((p_{k+1,1}, p_{k+1,2}), \beta_{k+1,1} \cup \beta_{k+1,2}, \gamma_{k+1,1} \cup \gamma_{k+1,2})$ nach Dauer t_k mit Eventmenge $Y_{k,1} \cup Y_{k,2}$, $k \geq 0$. Es gibt eine Kante $(p_{k,1}, g_1, X_1, p_{k,1})$ in Ph_1 und es gilt, da r_1 ein Ablauf von Ph_1 ist: $\beta_{k,1} \cup \beta'_{k+1,1} \cup (\gamma_{k,1} + t_k) \cup \chi_{Y_{k,1}} \models g_1$. Analog gibt es eine solche Kante $(p_{k,2}, g_2, X_2, p_{k,2})$ in Ph_2 und es gilt, da r_2 ein Ablauf von Ph_2 ist: $\beta_{k,2} \cup \beta'_{k+1,2} \cup (\gamma_{k,2} + t_k) \cup \chi_{Y_{k,2}} \models g_2$. Per Definition der Parallelkomposition gibt es eine Kante $((p_{k,1}, p_{k,2}), g_1 \wedge g_2, X_1 \cup X_2, (p_{k+1,1}, p_{k+1,2}))$

in $Ph_1 \parallel Ph_2$. Die Argumentation, dass der Guard erfüllt wird, verläuft analog zu der Argumentation, dass die Zustandsinvariante erfüllt ist. Benötigt wird zusätzlich die Aussage, dass Events in A_s gleichzeitig in beiden Automaten auftreten. Damit dürfen die Eventmengen vereinigt werden, da Events der Menge $Y_{k,1} \cap Y_{k,2}$ sowohl g_1 als auch g_2 erfüllen und Events aus $Y_{k,2} \setminus Y_{k,1}$ nicht in g_1 spezifiziert sind, bzw. Events aus $Y_{k,1} \setminus Y_{k,2}$ nicht in g_2 vorkommen.
Sei ohne Einschränkung $c \in C_1$. Es gilt $(\gamma_{k+1,1} \cup \gamma_{k+1,2})(c) = \gamma_{k+1,1}(c) = \gamma_{k,1}[X_1 := 0](c) = (\gamma_{k,1} \cup \gamma_{k,2})[X_1 := 0](c) = (\gamma_{k,1} \cup \gamma_{k,2})[X_1 \cup X_2 := 0](c)$. Die erste Gleichheit gilt, weil die Uhrenmengen disjunkt sind, die zweite, weil r_1 ein Ablauf von Ph_1 ist. Die übrigen Gleichheiten gelten, weil die Uhrenmengen disjunkt sind.
Es gelten alle Eigenschaften aus Definition 2.2.3 und damit ist r ein Ablauf der Parallelkomposition. Die Bemerkung folgt unmittelbar aus der Definition von r. □

Lemma 2.2.3 stellt das Gegenstück zu 2.2.2 dar: Es zeigt auf, wie aus einem Ablauf der Parallelkomposition zweier Automaten für jeden der einzelnen Automaten ein Ablauf gewonnen wird, indem Belegungen, Uhren und Variablen auf die entsprechenden Mengen eingeschränkt werden.

Lemma 2.2.3
Seien mit $Ph_l = (P_l, V_l, A_l, C_l, E_l, s_l, I_l, P_{0,l})$, $l \in \{1, 2\}$, zwei Phasen-Event-Automaten gegeben. Sei mit

$$r = \langle ((p_{0,1}, p_{0,2}), \beta_0, \gamma_0), t_0, Y_0, ((p_{1,1}, p_{1,2}), \beta_1, \gamma_1), t_1, Y_1, \ldots\rangle \in \mathbf{Run}(Ph_1 \parallel Ph_2)$$

ein Ablauf der Parallelkomposition gegeben. Dann sind

$$r_l := \langle (p_{0,l}, \beta_0|_{V_l}, \gamma_0|_{C_l}), t_0, Y_0 \cap A_l, (p_{1,l}, \beta_1|_{V_l}, \gamma_1|_{C_l}), t_1, Y_1 \cap A_l, \ldots\rangle \in \mathbf{Run}(Ph_l)$$

Abläufe in Ph_1 bzw. Ph_2.

Beweis
Der Beweis folgt unmittelbar mit der Definition der Zustandsinvarianten, Uhrenbedingungen und Guards als Konjunktionen der Prädikate der einzelnen Automaten. □

2.3 Tools

Das Java-Paket `pea` führt die Berechnung von Countertrace-Automaten durch. Es stellt wichtige Datentypen und Algorithmen bereit, die bei der Implementierung der Testautomatensemantik aus Kapitel 4 wiederverwendet werden.
Das Tool *Moby/PEA* dient der Erstellung von Phasen-Event-Automaten über eine graphische Oberfläche. Es ist im Rahmen des AVACS Projektes von Johannes Faber geschrieben worden und wird im Rahmen des Teilprojektes R genutzt. Der zu entwickelnde Compiler soll zu *Moby/PEA* kompatibel sein, d.h. vom Compiler ausgegebene Daten sollen in *Moby/PEA* gelesen werden können.

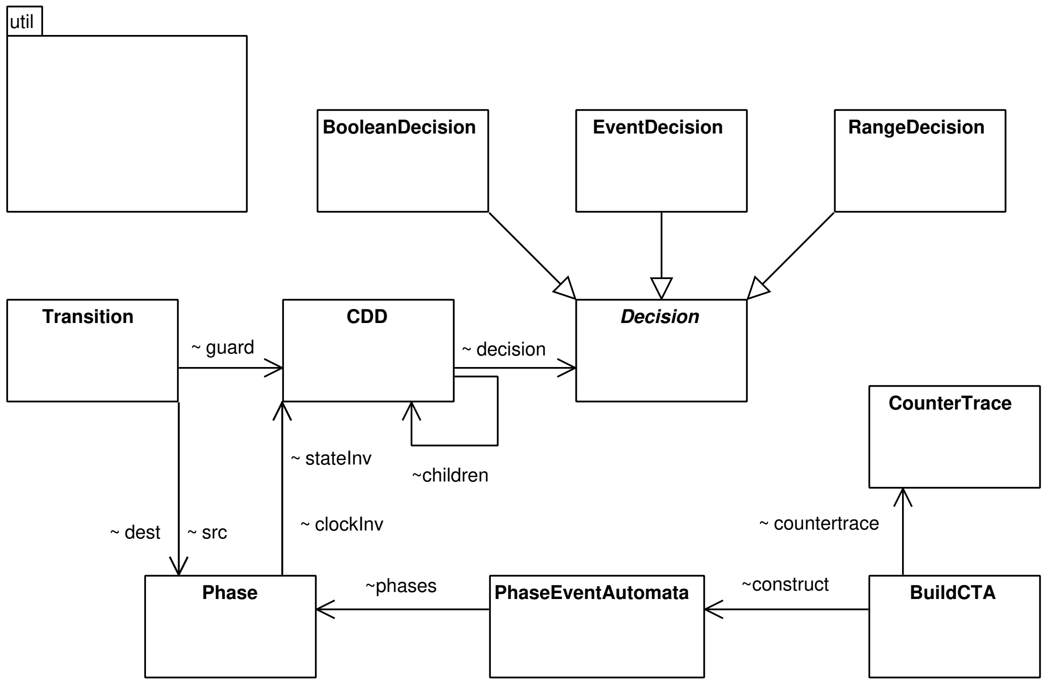

Abbildung 2.5: Paket `pea`

2.3.1 Das Paket `pea`

Das von Jochen Hoenicke entwickelte Java-Paket `pea` bietet Klassen für Countertraces, Formeln und Phasen-Event-Automaten. Ferner besitzt das Paket die Klasse `BuildCTA` zum Aufbau von Countertrace-Automaten. Diese Klasse führt die in Abschnitt 4.3 vorgestellte Potenzmengenkonstruktion fast vollständig durch.
Die Struktur des Paketes `pea` ist in Abbildung 2.5 abgebildet. Phasen-Event-Automaten werden durch die Klasse `PhaseEventAutomata` implementiert. Dabei besteht ein Phasen-Event-Automat aus einer Menge von `Phase` Objekten, die die Phasen des Automaten darstellen. Transitionen sind als Attribute in ihren Quellphasen enthalten. Die Klasse `PhaseEventAutomata` implementiert mit der Methode `parallel` die Parallelkomposition von Phasen-Event-Automaten aus Definition 2.2.4.
Das Paket `pea` dient der Konstruktion von PEAs zu gegebenen Countertraces, die den im Kapitel 3 vorgestellten Traces entsprechen. Ein `CounterTrace`-Objekt setzt sich aus einer Folge von `DCPhase` und `Chop` Objekten zusammen, welche die Datenstrukturen aus Abschnitt 4.3 wiederspiegeln.
Formeln, die als Zustandsinvarianten oder Uhrenbedingungen in Phasen, als Guards in Transitionen oder als Invarianten in DC Phasen enthalten sind werden als `CDD` (Constraint Decision Diagram) Objekte repräsentiert. Diese von BDDs abgeleitete Darstellung erlaubt eine effiziente Durchführung aller boolescher Operationen auf Formeln. Die Klasse stellt entsprechende Methoden `and`, `or` und `negate` bereit. Zur Zeit lassen sich mit CDDs drei Arten von Bedingungen in Formeln ausdrücken: Mit `EventDecision`

Objekten werden Events formuliert, `BooleanDecision` Objekte repräsentieren boolesche Aussagen. Eine `RangeDecision` gibt mittels $\{=, \neq, \leq, <, >, \geq\}$ Bedingungen für eine reellwertige Variable an. Alle drei Klassen erben von der abstrakten Klasse `Decision`, auf die von `CDD` zugegriffen wird.
Es sei angemerkt, dass jede `Decision` nur als ein Objekt repräsentiert wird. Wann immer ein zweites Objekt mit einer schon vorhandenen Aussage erstellt werden soll, wird ein Verweis auf das existente Objekt zurückgeliefert. Damit ist die Darstellung von Formeln Speicherplatz sparend implementiert.
Kern des Paketes `pea` ist die Klasse `BuildCTA`. Sie berechnet den PEA zu einem gegebenen `CounterTrace` Objekt. Im Rahmen dieser Arbeit soll die Klasse zu einem Compiler für Testformeln in model-checkbarer Darstellung erweitert werden. Im Folgenden werden die wichtigsten Methoden der Klasse `BuildCTA` vorgestellt:

`buildAut` Mit `buildAut` wird die Übersetzung eines gegebenen `CounterTrace` Objektes angestoßen. Nach Aufruf von `precomputeCDDs` werden in einer Schleife zu allen Phasen die Nachfolger berechnet. Abschließend werden die initialen Zustände ermittelt und das Ergebnis der Berechnung als `PhaseEventAutomata` Objekt zurückgegeben.

`precomputeCDDs` Die Methode `precomputeCDDs` berechnet zu Beginn der Kompilation Formeln, auf die während des Aufbaus der Countertrace-Automaten zurückgegriffen wird.

`findTrans` Für jede zu bearbeitende Phase werden mit `initTrans` zunächst weitere Formeln vorberechnet. Anschließend werden rekursiv alle Nachfolgephasen mit den notwendigen Transitionen ermittelt.

`recursiveBuildTrans` Die rekursive Berechnung der Nachfolgephasen wird mit der Methode `recursiveBuildTrans` vorgenommen. Es wird zu jeder möglichen Belegung der Uhren, Variablen und Events des Automaten eine Nachfolgephase bestimmt. Die Rekursion bricht ab, falls sich für die Transition notwendige Bedingungen ausschließen, oder wenn alle DC Phasen der gegebenen Countertrace bei der Berechnung der Nachfolgephase berücksichtigt wurden. In diesem Fall wird die neue Phase mit `buildNewTrans` erzeugt. Sollte die gesamte Countertrace in der Nachfolgephase erkannt worden sein, so wird der Nachfolgezustand nicht konstruiert.

`buildNewTrans` Mit `buildNewTrans` werden sowohl die Nachfolgephase als auch die Transition zu dieser Phase erzeugt. Jede erreichte aber noch nicht behandelte Phase wird der Menge aller zu behandelnden Phasen hinzugefügt.

Die Methode `recursiveBuildTrans` überprüft für jede DC Phase einer gegebenen Countertrace, ob sie sich in der Menge der aktiven Phasen des Nachfolgezustandes befindet. Mit der Klasse `BuildCTA.PhaseBits` sind DC Phasen als Bitvektoren dargestellt, die mit den Shifting-Operatoren sehr effizient manipuliert werden können.

2.3.2 Das Tool Moby/PEA

Das Tool *Moby/PEA* ist ein neues Werkzeug in der *Moby*-Reihe. Mit *Moby/PEA* ist es möglich, als Phasen-Event-Automaten modellierte Systeme in Form von Projekten zu verwalten. Die Modelle werden als Netze von Phasen-Event-Automaten, repräsentiert als einfache Knoten, dargestellt. Mit einem Doppelklick lassen sich die einzelnen PEAs be-

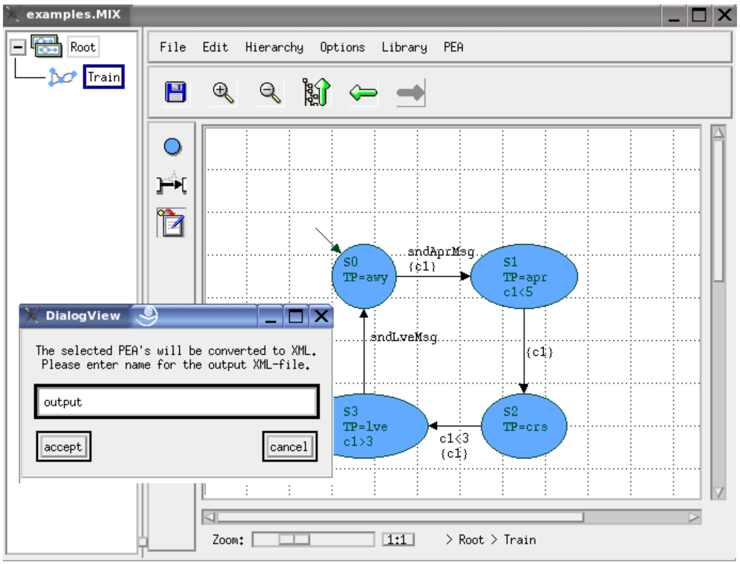

Abbildung 2.6: Graphischer Editor und XML-Export von *Moby/PEA*

arbeiten. Es wird das in Abbildung 2.6 gezeigte Menü zum Einfügen von Zuständen und Kanten benutzt. Ferner erlauben Kontextmenüs die Bearbeitung von Zustandsnamen, Invarianten und Uhrenbedingungen bzw. Guards und Resetmengen bei bei Transitionen. Über den Menüeintrag *PEA* ist es möglich, Uhren und Variablen für den editierten Automaten anzugeben. In Abschnitt 5.4 wird jedoch erklärt, dass sich die Unterstützung von getypten Variablen für Phasen-Event-Automaten zur Zeit noch in der Entwicklung befindet. Unter dem Eintrag *PEA* findet sich außerdem ein Menüpunkt *XML-Export*. Mit einem Klick öffnet sich das in Abbildung 2.6 zu sehende Fenster, das den Namen der Ausgabedatei erwartet. Die vom Programm geschriebene XML-Datei entspricht dem später vorgestellten Schema PEA.XSD. In zukünftigen Versionen von *Moby/PEA* soll es auch möglich sein, die Ausgaben des in dieser Arbeit entwickelten Compilers in *Moby/PEA* zu laden.

3 Testformeln

Um zu prüfen, ob ein gegebener Phasen-Event-Automat Ph eine negierte Duration Calculus Formel erfüllt, wird die Formel in einen PEA mit Endzustand – einen Testautomaten – übersetzt, der mit Ph parallel komponiert wird. Ist in der parallelen Komposition der Endzustand des Testautomaten nicht erreichbar, so ist die negierte Formel erfüllt.
Es ist bereits für die in [Cha03] vorgestellte Klasse $RDC_1(r)$, $r \in \mathbb{R}_{>0}$, nachgewiesen, dass die Erfüllbarkeit von Formeln dieser Klasse unentscheidbar ist. Daher kann nicht angenommen werden, dass beliebige DC Formeln in PEAs übersetzt werden können. Ein Beispiel für eine Formel, zu der kein PEA konstruiert werden kann[1], ist

$$\neg\Diamond(\updownarrow a\,;(\lceil true \rceil \wedge \ell = 1)\,;\updownarrow b).$$

Die Formel verbietet, dass nach Auftreten eines Events a ein Event b eine Zeiteinheit später erscheint. Ein Automat müsste bei jedem Auftreten von a eine Uhr stellen, um sicherzugehen, dass nach einer Zeiteinheit kein b folgt. Da die Menge an auftretenden a's nicht beschränkt ist, würde eine beschränkte Menge an gegebenen Uhren eines PEA nicht ausreichen, um die Eigenschaft zu überprüfen.
Martin Fränzle kritisiert in seiner Dissertation [Frä96], dass viele Unentscheidbarkeitsresultate für Realzeitsysteme von unbeschränktem Verhalten des Modells in beschränkter Zeit herrührten. Das Verhalten tatsächlich existenter Systeme in beschränkter Zeit sei jedoch durch die Geschwindigkeit der Hardware nach oben beschränkt. Also seien die Unentscheidbarkeitsresultate der Theorie nicht auf die Praxis übertragbar.

In dieser Diplomarbeit wird mögliches Verhalten nicht durch eine mininmale Zeitdauer eingeschränkt. Stattdessen wird eine Teilklasse der Duration Calculus Formeln identifiziert, welche sich in Phasen-Event-Automaten übersetzen lassen. Der Definition der Formelklasse schließen sich Beobachtungen über ihre Ausdruckskraft an. Mit der Einführung von Sync-Events ist es möglich, Formeln der identifizierten Klasse in eine Normalform zu überführen, welche die Übersetzung in Phasen-Event-Automaten erleichtert und das Model-Checking effizienter macht.

3.1 Die Formelklasse *Testform*

Nachfolgend wird die Klasse der Testformeln definiert. Der Grundgedanke ist, komplexe Formeln aus Traces, welche den Countertraces in [Hoe05a] entsprechen, zu konstruieren. Für Countertraces ist bekannt, dass PEAs konstruiert werden können. Es werden weder

[1] Eine Definition, wie PEAs und Formeln im Verhähltnis stehen, wird in Kapitel 4 gegeben.

exakte Längen für Phasen, noch *true*-Phasen erlaubt. Mit zwei Beobachtungen wird gezeigt, dass sowohl Formeln mit exakten Längen als auch Formeln, die *true*-Phasen enthalten, in PEAs übersetzt werden können.

Bemerkung 3.1.1
Formeln der Klasse *Testform* enthalten keine globalen Variablen. Aus diesem Grund ist die Belegung von Variablen für die Erfüllbarkeit von Formeln irrelevant und wird der Einfachheit halber weggelassen.

Definition 3.1.1 (*Testform*)
Die Klasse *Testform* der zum Model-Checking geeigneten Formeln ist in vier Schritten induktiv definiert

$$\begin{aligned}
\mathit{Phase} &:= \ell > 0 \land \ell \sim k \mid Ph \land \lceil \varphi \rceil \mid Ph \land \boxminus \mathcal{E} && (3.1)\\
\mathit{Trace} &:= Ph \mid \updownarrow \mathcal{E} \mid \not\updownarrow \mathcal{E} \mid T_1 \,;\, T_2 && (3.2)\\
\mathit{BForm} &:= T \mid \neg F \mid F_1 \land F_2 && (3.3)\\
\mathit{Testform} &:= (\neg \Diamond_{L} T \mid) F \mid TF_1 \,;\, TF_2 \mid TF_1 \land TF_2 \mid TF_1 \lor TF_2, && (3.4)
\end{aligned}$$

wobei $Ph \in \mathit{Phase}$, $T, T_1, T_2 \in \mathit{Trace}$, $F, F_1, F_2 \in \mathit{BForm}$, $TF_1, TF_2 \in \mathit{Testform}$, $k \in \mathbb{R}_{>0}$, φ ein zeitbehafteter Zustandsausdruck, $\mathcal{E}$ eine boolesche Observable und $\sim \in \{\emptyset, \leq, <, >, \geq\}$ sind. Die leere Menge stellt das Auslassen eines Timebounds dar und wird als $\ell > 0$ aufgefasst.

Eine Trace besitzt als erstes Element immer eine Phase. □

Die Klammerung ($\neg \Diamond_{L} T$) bedeutet, dass es durchaus möglich ist, eine Normalform für Formeln anzugeben, die die Liveness-Eigenschaft enthalten. Diese Formeln sind jedoch im nachfolgenden Kapitel 4 nicht in Testautomaten übersetzt, weil sich Liveness nicht durch das Testen der Erreichbarkeit eines einzigen Zustandes überprüfen lässt. Für das LTL Model-Checking beispielsweise muss, um Liveness mit Testautomaten zu checken, eine starke Zusammenhangskomponente gefunden werden, in welcher ein Endzustand erreichbar ist. Ferner müssen Fairness-Annahmen über das System gemacht werden. Daher wird die Liveness-Eigenschaft in dieser Diplomarbeit nur in der Normalform berücksichtigt.

Obwohl in Traces der Testformelklasse Phasen keine exakten Längen besitzen dürfen, sagt Lemma 3.1.1 aus, dass nicht negierte Traces, in denen Phasen mit exakten Längen enthalten sind, in der Klasse der Testformeln liegen. Der Grund ist, dass exakte Längen nur in Verbindung mit der Negation zu Eigenschaften führen, welche nicht mit Automaten überprüfbar sind. Bezugnehmend auf das Beispiel der Einleitung ist es jedoch möglich, zu überprüfen, ob ein Event b in dem Abstand einer Zeiteinheit auf ein Event a folgt. Traces mit exakten Längen dürfen mit allen Operatoren der Klasse *Testform* verknüpft werden, ohne dass die resultierenden Formeln die Klasse *Testform* verlassen.

Lemma 3.1.1 (Exakte Längen in Traces)
Die Klasse der Traces mit exakten Längen ist definiert als:

$$\begin{aligned} \mathit{PhaseEx} &:= \ell = k \mid Ph \wedge \lceil \varphi \rceil \mid Ph \wedge \boxminus \mathcal{E} \\ \mathit{TraceEx} &:= Ph \mid \updownarrow \mathcal{E} \mid \not\updownarrow \mathcal{E} \mid T_1 \,;\, T_2 \\ \mathit{TestformEx} &:= T \mid TF_1 \,;\, TF_2 \mid TF_1 \wedge TF_2 \mid TF_1 \vee TF_2, \end{aligned}$$

wobei $Ph \in \mathit{PhaseEx}$, $T, T_1, T_2 \in \mathit{TraceEx}$, $TF_1, TF_2 \in \mathit{TestformEx}$, $k \in \mathbb{R}_{>0}$, φ ein zeitbehafteter Zustandsausdruck und $\mathcal{E}$ eine boolesche Observable sind, kann in *Testform* ausgedrückt werden, d.h. für eine Formel in *TestFormEx* gibt es eine Testformel die genau dann gilt, wenn die Formel in *TestFormEx* wahr ist.

Beweis
Der Beweis beruht auf der Beobachtung, dass jede Phase eine Testformel darstellt und dass sich Gleichheit durch $\ell \leq k \wedge \ell \geq k$ ausdrücken lässt.
Jede Phase $\ell = k \wedge \bigwedge_{i=1}^{n} \lceil \varphi_i \rceil \wedge \bigwedge_{j=1}^{m} \boxminus \mathcal{E}_j$ in *PhaseEx* lässt sich in der Klasse der Testformeln ausdrücken als

$$\left(\ell > 0 \wedge \ell \leq k \wedge \bigwedge_{i=1}^{n} \lceil \varphi_i \rceil \wedge \bigwedge_{j=1}^{m} \boxminus \mathcal{E}_j \right) \wedge \left(\ell > 0 \wedge \ell \geq k \wedge \bigwedge_{i=1}^{n} \lceil \varphi_i \rceil \wedge \bigwedge_{j=1}^{m} \boxminus \mathcal{E}_j \right).$$

Ferner ist auf der Ebene der Testformeln der Chop-Operator erlaubt, daher ist es mit der genannten Überführung von Phasen aus *PhaseEx* möglich, jede Trace in *TraceEx* und schließlich jede Formel aus *TestformEx* in *TestForm* zu konstruieren. □

Ebenso wie keine exakten Längen in Traces erlaubt sind, sind keine *true*-Phasen in der Klasse *Phase* vorgesehen, eine Phase besitzt immer den Constraint $\ell > 0$. Das nachfolgende Lemma zeigt, dass Formeln mit *true*-Phasen innerhalb von Traces dennoch in der Klasse der Testformeln liegen.

Lemma 3.1.2 (*true*-Phasen in Traces)
Die Klasse der Testformeln aus *TestformTrue* mit *true*-Phasen lässt sich durch *Testform* ausdrücken, d.h. zu einer Formel mit *true*-Phasen gibt es eine Testformel, so dass die Formel mit *true*-Phasen in einer Interpretation und in einem Intervall gilt, gdw. die Testformel gültig ist. Dabei sei *TestformTrue* definiert als:

$$\begin{aligned} \mathit{PhaseTrue} &:= \mathit{true} \mid \ell > 0 \wedge \ell \sim k \mid Ph \wedge \lceil \varphi \rceil \mid Ph \wedge \boxminus \mathcal{E} \\ \mathit{TraceTrue} &:= Ph \mid \updownarrow \mathcal{E} \mid \not\updownarrow \mathcal{E} \mid T_1 \,;\, T_2 \\ \mathit{BFormTrue} &:= T \mid \neg F \mid F_1 \wedge F_2 \\ \mathit{TestformTrue} &:= F \mid TF_1 \,;\, TF_2 \mid TF_1 \wedge TF_2 \mid TF_1 \vee TF_2, \end{aligned}$$

wobei $Ph \in \mathit{PhaseTrue}$, $T, T_1, T_2 \in \mathit{TraceTrue}$, $F, F_1, F_2 \in \mathit{BFormTrue}$, $TF_1, TF_2 \in \mathit{TestformTrue}$, $k \in \mathbb{R}_{>0}$, φ ein zeitbehafteter Zustandsausdruck, $\mathcal{E}$ eine boolesche Observable und $\sim \in \{\emptyset, \leq, <, >, \geq\}$ sind. Ferner darf nur eine Phase ungleich *true* das erste Element einer Trace sein.

Beweis
Der Beweis nutzt die Distributivität zwischen $\vee$ und **;** sowie die Äquivalenz $true \Leftrightarrow \ell = 0 \vee \lceil true \rceil$.
Enthalte eine Trace $T := (\ldots ; true ; \ldots)$ eine *true*-Phase, so gilt:

$$\begin{aligned} T :=& (\ldots ; true ; \ldots) \\ \{\text{Def. } true\} \Leftrightarrow& (\ldots ; (\ell = 0 \vee \lceil true \rceil) ; \ldots) \\ \{\text{Chop-Oder}\} \Leftrightarrow& (\ldots ; \ldots) \vee (\ldots ; \lceil true \rceil ; \ldots) \end{aligned}$$

Gegebenenfalls vor der Trace stehende Negationen können mit dem Gesetz von de Morgan eliminiert werden. Die resultierende Formel liegt in *Testform*, sowohl im negierten als auch im nicht negierten Fall. □

Bemerkung 3.1.2
Die Konstruktion erzeugt in der Anzahl der *true*-Phasen exponentiell viele Formeln. Sie kann vermieden werden, indem Traces mit *true*-Phasen direkt in PEAs überführt werden. Die Übersetzung ist in [Hoe05a] zu finden.

3.2 Sync-Events

In Abschnitt 3.3 wird eine Normalform für Formeln der Klasse *Testform* nachgewiesen. Die Idee ist, eine disjunktive Normalform über Traces zu erzeugen. Da sowohl die Konjunktion als auch der Operator Chop auf der Ebene der Testformeln erlaubt sind, ist es in Formeln wie $(G \wedge H) ; F$ (mit Testformeln G, H, F) notwendig, beide Operatoren zu trennen. Zu diesem Zweck werden Sync-Events definiert.

Definition 3.2.1
Seien F, G DC Formeln, $\mathcal{E} \in SVar$ eine boolesche Observable, die weder in F noch in G vorkommt. Dann ist das *Sync-Event* $F \underset{\mathcal{E}}{\Updownarrow} G$ definiert als:

$$F \underset{\mathcal{E}}{\Updownarrow} G := F ; (\updownarrow \mathcal{E} \wedge \neg (true \lhd (\updownarrow \mathcal{E} ; \ell > 0)) \wedge \neg ((\ell > 0 ; \updownarrow \mathcal{E}) \rhd true)) ; G \qquad (3.5)$$

□

Bemerkung 3.2.1
Sync-Events sind als besondere Events definiert und können daher nicht zum Zeitpunkt Null auftreten. Es gilt die Implikation

$$F \underset{\mathcal{E}}{\Updownarrow} G \Rightarrow F \wedge \ell > 0.$$

Ein Sync-Event entspricht einem Event, das einmalig auftritt, d.h. einer booleschen Observablen, die nur zu einem Zeitpunkt ihren Wert ändert. Bis zu diesem Zeitpunkt muss die Formel F gelten, ab dann ist die Formel G erfüllt. Die Idee der Sync-Events ist es, den Chop-Punkt zwischen zwei Formeln auszudrücken, ohne auf die Länge des Intervalls zuzugreifen. Mit der Nutzung von Sync-Events können Realzeitsysteme Event-triggered modelliert werden, während die Nutzung von ℓ eine Time-triggered Modellierung vorsieht.

Sync-Events verhalten sich untereinander und gegenüber dem Chop-Operator assoziativ, ferner sind die Chop-Oder Distributivitäten auch für Sync-Events gültig. Der Grund für diese Eigenschaften ist die Definition der Sync Events über den Chop-Operator.

Bemerkung 3.2.2 (Assoziativ- und Distributivgesetze)
Seien F, G, H DC Formeln, $\mathcal{E}, \mathcal{E}_1, \mathcal{E}_2 \in SVar$ boolesche Observablen, die nicht in F, G, H vorkommen. Dann sind folgende Äquivalenzen gültig:

$$\left(F \underset{\mathcal{E}_1}{\Updownarrow} G\right) \underset{\mathcal{E}_2}{\Updownarrow} H \Leftrightarrow F \underset{\mathcal{E}_1}{\Updownarrow} \left(G \underset{\mathcal{E}_2}{\Updownarrow} H\right) \qquad \text{(Sync-Ass)} \tag{3.6}$$

$$\left(F \underset{\mathcal{E}}{\Updownarrow} G\right) ; H \Leftrightarrow F \underset{\mathcal{E}}{\Updownarrow} (G ; H) \qquad \text{(Sync-Chop-Ass1)} \tag{3.7}$$

$$(F ; G) \underset{\mathcal{E}}{\Updownarrow} H \Leftrightarrow F ; \left(G \underset{\mathcal{E}}{\Updownarrow} H\right) \qquad \text{(Sync-Chop-Ass2)} \tag{3.8}$$

$$F \underset{\mathcal{E}}{\Updownarrow} (G \vee H) \Leftrightarrow \left(F \underset{\mathcal{E}}{\Updownarrow} G\right) \vee \left(F \underset{\mathcal{E}}{\Updownarrow} H\right) \qquad \text{(Sync-Disj-Dist1)} \tag{3.9}$$

$$(F \vee G) \underset{\mathcal{E}}{\Updownarrow} H \Leftrightarrow \left(F \underset{\mathcal{E}}{\Updownarrow} H\right) \vee \left(G \underset{\mathcal{E}}{\Updownarrow} H\right) \qquad \text{(Sync-Disj-Dist2)} \tag{3.10}$$

Sync-Events sind nicht in Formeln der Klasse *Testform* enthalten, daher muss eine Aussage gefunden werden, die es erlaubt, in eine gegebene Formel Sync-Events einzufügen. Die im nachstehenden Lemma bewiesene Äquivalenz namens *Sync-Event Introduction*, kurz *Sync-Intro*, stellt eine Möglichkeit dar.
Es bestehen keine Distributivgesetz zwischen der Konjunktion und dem Chop-Operator. Dennoch kann gezeigt werden, dass es zwei verschiedene Distributivitäten zwischen Sync-Events und der Konjunktion gibt. Beide sind in 3.2.1 angeführt. Zusammen mit der Sync-Event Einführung ist damit die Frage nach einer Kapselung zwischen **;** und $\wedge$ in Testformeln über Sync-Events beantwortet.
Mit Zeile 3.11 in Lemma 3.2.1 kann die Livess-Eigenschaft immer als Konjunkt der Testformel betrachtet werden. Damit werden Traces und Liveness-Eigenschaften in Testformeln separiert.

Lemma 3.2.1 (Sync-Event Introduction und besondere Distributivität)
Seien $T \in Trace$ und $\mathcal{E} \in SVar$ eine boolesche Observable, die nicht in den DC Formeln F, F_i, G, G_j, H $(1 \leq i \leq m, 1 \leq j \leq n)$, $m, n \in \mathbb{N}$ vorkommt. Es gelten folgende Äquivalenzen:

$$\left(\neg \overset{L}{\Diamond} T\right) ; F \Leftrightarrow \left(\neg \overset{L}{\Diamond} T\right) \wedge ((\lceil true \rceil ; F) \vee F) \tag{3.11}$$

$$(F \wedge \ell > 0)\,;\,G \Leftrightarrow \exists \mathcal{E} : \left(F \underset{\mathcal{E}}{\Updownarrow} G\right) \qquad \text{(Sync-Intro)} \tag{3.12}$$

$$\left(\bigwedge_{i=1}^{m} F_i\right) \underset{\mathcal{E}}{\Updownarrow} \left(\bigwedge_{j=1}^{n} G_j\right) \Leftrightarrow \bigwedge_{i=1}^{m} \left(F_i \underset{\mathcal{E}}{\Updownarrow} \mathit{true}\right) \wedge \bigwedge_{j=1}^{n} \left(\lceil \mathit{true} \rceil \underset{\mathcal{E}}{\Updownarrow} G_j\right) \qquad \text{(Sync-Dist+)} \tag{3.13}$$

$$F \underset{\mathcal{E}}{\Updownarrow} (G \wedge H) \Leftrightarrow \left(F \underset{\mathcal{E}}{\Updownarrow} G\right) \wedge \left(F \underset{\mathcal{E}}{\Updownarrow} H\right) \qquad \text{(Sync-Conj-Dist1)} \tag{3.14}$$

$$(F \wedge G) \underset{\mathcal{E}}{\Updownarrow} H \Leftrightarrow \left(F \underset{\mathcal{E}}{\Updownarrow} H\right) \wedge \left(G \underset{\mathcal{E}}{\Updownarrow} H\right) \qquad \text{(Sync-Conj-Dist2)} \tag{3.15}$$

Beweis
Eigenschaft 3.11 wird mit folgender Äquivalenzkette nachgewiesen:

$$\begin{aligned}
&\mathcal{I}, [b, e] \models (\neg \Diamond T)\,;\,F \\
\Leftrightarrow &\exists m \in [b, e] : (\mathcal{I}, [b, m] \models \neg \Diamond T \text{ und } \mathcal{I}, [m, e] \models F) \\
\Leftrightarrow &\exists m \in [b, e] : ((\text{nicht } \exists m_1 \in [b, \infty[: \exists m_2 \in [m_1, \infty[: \mathcal{I}, [m_1, m_2] \models T) \text{ und} \\
&\qquad\qquad \mathcal{I}, [m, e] \models F) \\
\Leftrightarrow &(\text{nicht } \exists m_1 \in [b, \infty[: \exists m_2 \in [m_1, \infty[: \mathcal{I}, [m_1, m_2] \models T) \text{ und} \\
&\qquad\qquad (\exists m \in [b, e] : \mathcal{I}, [m, e] \models F) \\
\Leftrightarrow &\mathcal{I}, [b, e] \models \neg \Diamond T \wedge (\mathit{true}\,;\,F) \\
\Leftrightarrow &\mathcal{I}, [b, e] \models \neg \Diamond T \wedge ((\lceil \mathit{true} \rceil\,;\,F) \vee F)
\end{aligned}$$

Die erste Äquivalenz gilt per Definition, die zweite nutzt die Charakterisierung des irgendwann-Operators aus Lemma 2.1.2. Anschließend wird gebraucht, dass die Zeitpunkte im Liveness-Ausdruck nicht durch den äußeren Quantor gebunden werden. Die folgende Äquivalenz gilt per Definition. Die letzte Zeile der Äquivalenzkette nutzt $\mathit{true} \Leftrightarrow \lceil \mathit{true} \rceil \vee \ell = 0$.

Eigenschaft 3.12 wird *Sync-Event Introduction* oder *Sync-Event Einführung* genannt. Die Richtung $\Leftarrow$ gilt aufgrund der Definition der Sync-Events, die Richtung $\Rightarrow$ lässt sich wie folgt nachweisen:
Gelte:

$$\begin{aligned}
&\mathcal{I}, [b, e] \models (F \wedge \ell > 0)\,;\,G \\
\{\text{Definition }\mathbf{;}\} \Leftrightarrow &\exists m \in]b, e] : (\mathcal{I}, [b, m] \models F \text{ und } \mathcal{I}, [m, e] \models G)
\end{aligned}$$

Die Menge der Observablen in F und G ist endlich, die Menge aller boolescher Observablen in SVar ist jedoch unendlich. Es gibt also eine Observable, $\mathcal{E}$, die nicht in F und nicht in G vorkommt. Definiere eine Interpretation $\mathcal{I}'$ für die gilt: $\mathcal{I}' = \mathcal{I}$ bis aus die Interpretation von $\mathcal{E}$, welche definiert sei durch:

$$\mathcal{I}'(\mathcal{E})(t) = \begin{cases} \mathit{ff}, & t < m, \\ tt, & t \geq m. \end{cases}$$

Dann gilt folgende Aussage:

$$\begin{aligned} & \exists \mathcal{I}' =_{\backslash\{\mathcal{E}\}} \mathcal{I} : \exists m \in]b,e] : (\mathcal{I}', [b,m] \models F \text{ und } \mathcal{I}', [m,e] \models G \text{ und} \\ & \quad \mathcal{I}', [m,m] \models (\updownarrow \mathcal{E} \wedge \neg (\mathit{true} \lhd (\updownarrow \mathcal{E} \,;\, \ell > 0)) \wedge \neg ((\ell > 0 \,;\, \updownarrow \mathcal{E}) \rhd \mathit{true}))) \\ \Leftrightarrow & \mathcal{I}, [b,e] \models \exists \mathcal{E} : \left(F \underset{\mathcal{E}}{\Updownarrow} G \right) \end{aligned}$$

Lemma 3.13 ist eine Konsequenz aus der Definition der Sync-Events:

$$\begin{aligned} & \mathcal{I}, [b,e] \models \left(\bigwedge_{i=1}^{m} F_i \right) \underset{\mathcal{E}}{\Updownarrow} \left(\bigwedge_{j=1}^{n} G_j \right) \\ \Leftrightarrow & \exists m \in]b,e] : (\mathcal{I}, [b,m] \models F_i \ (1 \leq i \leq m) \text{ und} \\ & \quad \mathcal{I}, [m,m] \models (\updownarrow \mathcal{E} \wedge \neg (\mathit{true} \lhd (\updownarrow \mathcal{E} \,;\, \ell > 0)) \wedge \neg ((\ell > 0 \,;\, \updownarrow \mathcal{E}) \rhd \mathit{true})) \\ & \quad \text{und } \mathcal{I}, [m,e] \models G_j \ (1 \leq j \leq n)) \\ \Leftrightarrow & \mathcal{I}, [b,e] \models \bigwedge_{i=1}^{m} \left(F_i \underset{\mathcal{E}}{\Updownarrow} \mathit{true} \right) \wedge \bigwedge_{j=1}^{n} \left(\lceil \mathit{true} \rceil \underset{\mathcal{E}}{\Updownarrow} G_j \right) \end{aligned}$$

Die übrigen Distributivgesetzte folgen analog unmittelbar aus der Definition der Sync-Events. □

Bemerkung 3.2.3
Seien die Voraussetzungen aus Lemma 3.2.1 erfüllt. Distributivgesetz 3.13 besitzt eine Komplexität von $m+n+1$, d.h. die Anwendung des Distributivgesetztes führt zu $m+n+1$ konjugierten Formeln. Die gewöhnlichen Distributivgesetze 3.14 und 3.15 aus Lemma 3.2.1 besitzen eine Komplexität von $m * n$.

3.3 Normalform

Der Nachweis einer Normalform für eine Formelklasse reduziert die Aufgabe der Generierung von Testautomaten auf Formeln einer bestimmten Gestalt. Darüber hinaus kann die Konstruktion der Normalform zu einer gegebenen Formel als Leitfaden für die Erstellung der Testautomaten verstanden werden. Es wird aufgezeigt, welche Teile der Formel im Automaten repräsentiert werden müssen, in welcher Reihenfolge die Formelteile zu übersetzen sind und wie die resultierenden Automaten zu verbinden sind. Eine Normalform gibt außerdem Aufschluss, welche Formelteile überflüssig sind und welche in äquivalenter Weise ausgedrückt werden können, so dass die Testautomaten eine geringere Komplexität aufweisen.

In diesem Abschnitt wird eine disjunktive Normalform für Testformeln nachgewiesen, welche als kleinstes Element Traces anstelle von Aussagen besitzt. Im nächsten Kapitel werden dann Traces analog zu [Hoe05a] in Testautomaten übersetzt. Die Konjunktion innerhalb der Disjunktionsglieder kann mit der Parallelkomposition auf Ebene der PEAs dargestellt werden. Die Disjunktion wird nicht auf PEA-Ebene überführt, sondern durch das Model-Checking einzelner Disjunktionsglieder ausgedrückt. Das Model-Checking kann beendet werden, wenn das erste erfüllte Disjunktionsglied gefunden ist. Ein Analogon zu diesem Resultat auf Formelebene ist das On-The-Fly Model-Checking auf Automatenebene: Das Model-Checking kann beendet werden, sobald das erste Gegenbeispiel bei der Entfaltung gefunden worden ist.

Theorem 3.3.1 (Normalformtheorem)
Jede Testformel ist äquivalent zu einer Formel der Form

$$\exists \mathcal{S}_{ij*} : \left(\bigvee_i \left(\bigwedge_j \mathcal{T}_{ij} \right) \right), \tag{3.16}$$

wobei

$$\begin{aligned} \mathcal{T}_{ij} :=& Tr_{ij} \mid Tr_{ij} \underset{\mathcal{S}_{ij}}{\Updownarrow} true \mid \lceil true \rceil \underset{\mathcal{S}_{ij}}{\Updownarrow} Tr_{ij} \mid \qquad (3.17)\\ & \lceil true \rceil \underset{\mathcal{S}_{ij1}}{\Updownarrow} \hat{T}r_{ij} \underset{\mathcal{S}_{ij2}}{\Updownarrow} true \mid \neg \Diamond_{L} T_{ij} \mid \lceil true \rceil \underset{\mathcal{S}_{ij}}{\Updownarrow} \neg \Diamond_{L} T_{ij} \\ \hat{T}r_{ij} :=& Tr_{ij} \mid true \qquad (3.18)\\ Tr_{ij} :=& T_{ij} \mid \neg T_{ij}, \qquad (3.19) \end{aligned}$$

mit $T_{ij} \in \mathit{Trace}$, $* \in \{1, 2, \epsilon\}$ und frischen booleschen Observablen S_{ij*}, die nicht in einer der Formeln vorkommen.

Beweis
Der Beweis wird mittels struktureller Induktion über den Aufbau der Testformeln geführt.

Induktionsanfang
Sei $T \in Trace$, dann gilt:

$$\neg \Diamond_{L} T \Leftrightarrow \exists \mathcal{S} : \left(\neg \Diamond_{L} T \right),$$

dabei sei $\mathcal{S}$ eine boolesche Observable, die nicht in T vorkommt. Also ist $\neg \Diamond_{L} T$ äquivalent zu einer Formel in Normalform.
Jede Formel $F \in \mathit{BForm}$ kann als disjunktive Normalform über Traces dargestellt werden:

$$F \Leftrightarrow \bigvee_i \left(\bigwedge_j Tr_{ij} \right)$$

$$\{\text{Definition } \exists\mathcal{S}\} \Leftrightarrow \exists\mathcal{S}_{ij} : \left(\bigvee_i \left(\bigwedge_j Tr_{ij}\right)\right),$$

wobei $Tr_{ij} = T_{ij}$ oder $Tr_{ij} = \neg T_{ij}$ mit $T_{ij} \in \mathit{Trace}$, $\mathcal{S}_{ij} \in \mathit{SVar}$ boolesche Observable, die nicht in T_{ij} vorkommen. Damit ist jede Formel äquivalent zu einer Formel in Normalform.

Induktionsschluss
Angenommen die Aussage gilt für

$$TF^l \Leftrightarrow \exists\mathcal{S}^l_{i^l j^l *} : \left(\bigvee_{i^l} \left(\bigwedge_{j^l} \mathcal{T}^l_{i^l j^l}\right)\right)$$

mit $l \in \{1, 2\}$. Es ist zu zeigen, dass $TF^1 \,;\, TF^2$, $TF^1 \wedge TF^2$ und $TF^1 \vee TF^2$ in Normalformdarstellung überführt werden können.

Sei zunächst der Fall $TF^1 \Rightarrow \ell > 0$ gegeben. Die Transformation von $TF^1 \,;\, TF^2$ geschieht in drei Schritten: Zunächst wird gezeigt, dass die Formel zu einer Formel mit außenstehenden Quantoren äquivalent ist. Dann werden die Disjunktionen aus den Formeln herausgezogen, gefolgt von den Konjunktionen. Abschließend werden Liveness-Teile von den Konjunktionsgliedern abgetrennt. Für den ersten Schritt wird folgende Äquivalenz genutzt:

$$\begin{aligned}
& TF^1 \,;\, TF^2 \\
\Leftrightarrow & \exists\mathcal{S}_{new} : \left(TF^1 \underset{\mathcal{S}_{new}}{\Updownarrow} TF^2\right) \\
\Leftrightarrow & \exists\mathcal{S}_{new} : \left(\left(\exists\mathcal{S}^1_{i^1 j^1 *} : \left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}^1_{i^1 j^1}\right)\right)\right) \underset{\mathcal{S}_{new}}{\Updownarrow} TF^2\right) \\
\Leftrightarrow & \exists\mathcal{S}_{new} \exists\mathcal{S}'^1_{i^1 j^1 *} : \left(\left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}'^1_{i^1 j^1}\right)\right) \underset{\mathcal{S}_{new}}{\Updownarrow} TF^2\right) \\
\Leftrightarrow & \exists\mathcal{S}_{new} \exists\mathcal{S}'^1_{i^1 j^1 *} \exists\mathcal{S}'^2_{i^2 j^2 *} : \left(\left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}'^1_{i^1 j^1}\right)\right) \underset{\mathcal{S}_{new}}{\Updownarrow} \left(\bigvee_{i^2} \left(\bigwedge_{j^2} \mathcal{T}'^2_{i^2 j^2}\right)\right)\right)
\end{aligned}$$

Die erste Äquivalenz benutzt die Sync-Event Introduction, die aufgrund der Voraussetzung $TF^1 \Rightarrow \ell > 0$ angewandt werden werden darf. Die zweite Äquivalenz gilt wegen der Induktionsannahme. Mit gebundener Umbenennung können die bestehenden Quantoren aus den Formeln herausgezogen werden. Das geschieht in der dritten Äquivalenz für TF^1. Abschließend werden die Quantoren für TF^2 herausgezogen.
Die Quantifizierung wird sich im Folgenden nicht mehr ändern, daher wird abkürzend $\exists$ ohne Observablen geschrieben. Mit der Distributivität von Chops und Disjunktionen

ist obige Formel äquivalent zu:

$$
\begin{aligned}
&\exists : \left(\bigvee_{i^1} \bigvee_{i^2} \left(\left(\bigwedge_{j^1} \mathcal{T}'^{1}_{i^1 j^1} \right) \underset{\mathcal{S}_{new}}{\Updownarrow} \left(\bigwedge_{j^2} \mathcal{T}'^{2}_{i^2 j^2} \right) \right) \right) \\
\Leftrightarrow &\exists : \left(\bigvee_{i^1} \bigvee_{i^2} \left(\bigwedge_{j^1} \left(\mathcal{T}'^{1}_{i^1 j^1} \underset{\mathcal{S}_{new}}{\Updownarrow} true \right) \wedge \bigwedge_{j^2} \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \mathcal{T}'^{2}_{i^2 j^2} \right) \right) \right)
\end{aligned}
\tag{3.20}
$$

Die Äquivalenz benutzt die besondere Distributivität zwischen Sync-Events und der Konjunktion.
Es bleibt zu zeigen, dass $\mathcal{T} \underset{\mathcal{S}_{new}}{\Updownarrow} true$ (Fall a) sowie $\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \mathcal{T}$ (Fall b) für jedes $\mathcal{T}$ in eine Konjunktion von Formeln der Form in 3.17 überführt werden können:

Fall 1 ($\mathcal{T} = Tr$): Sowohl $Tr \underset{\mathcal{S}_{new}}{\Updownarrow} true$ als auch $\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} Tr$ sind von gesuchter Form.

Fall 2 ($\mathcal{T} = Tr \underset{\mathcal{S}}{\Updownarrow} true$):

Fall a:

$$
\begin{aligned}
\left(Tr \underset{\mathcal{S}}{\Updownarrow} true \right) \underset{\mathcal{S}_{new}}{\Updownarrow} true \Leftrightarrow & Tr \underset{\mathcal{S}}{\Updownarrow} \left(true \underset{\mathcal{S}_{new}}{\Updownarrow} true \right) \\
\{\text{Sync-Dist+}\} \Leftrightarrow & \left(Tr \underset{\mathcal{S}}{\Updownarrow} true \right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} true \underset{\mathcal{S}_{new}}{\Updownarrow} true \right)
\end{aligned}
$$

Fall b:
Die Formel $\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} Tr \underset{\mathcal{S}}{\Updownarrow} true$ ist von der gesuchten Form.

Fall 3 ($\mathcal{T} = \lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} Tr$):

Fall a:
Die Formel $\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} Tr \underset{\mathcal{S}_{new}}{\Updownarrow} true$ ist von der gesuchten Form.

Fall b:

$$
\begin{aligned}
& \lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} Tr \right) \\
\{\text{Assoziativität}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil \right) \underset{\mathcal{S}}{\Updownarrow} Tr \\
\{\text{Sync-Dist+}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} true \right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} Tr \right)
\end{aligned}
$$

Fall 4 ($\mathcal{T} = \lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} \hat{Tr} \underset{\mathcal{S}_2}{\Updownarrow} true$)**:**

Fall a:

$$
\begin{aligned}
& \left(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} \hat{Tr} \underset{\mathcal{S}_2}{\Updownarrow} true\right) \underset{\mathcal{S}_{new}}{\Updownarrow} true \\
\{\text{Assoziativität}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} \hat{Tr}\right) \underset{\mathcal{S}_2}{\Updownarrow} \left(true \underset{\mathcal{S}_{new}}{\Updownarrow} true\right) \\
\{\text{Sync-Dist+}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} \hat{Tr} \underset{\mathcal{S}_2}{\Updownarrow} true\right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}_2}{\Updownarrow} true \underset{\mathcal{S}_{new}}{\Updownarrow} true\right)
\end{aligned}
$$

Fall b:

$$
\begin{aligned}
& \lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \left(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} \hat{Tr} \underset{\mathcal{S}_2}{\Updownarrow} true\right) \\
\{\text{Assoziativität}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil\right) \underset{\mathcal{S}_1}{\Updownarrow} \left(\hat{Tr} \underset{\mathcal{S}_2}{\Updownarrow} true\right) \\
\{\text{Sync-Dist+}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} true\right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} \hat{Tr} \underset{\mathcal{S}_2}{\Updownarrow} true\right)
\end{aligned}
$$

Fall 5 ($\mathcal{T} = \neg \Diamond_{\ell} T$)**:**

Fall a:

$$
\begin{aligned}
& \left(\neg \Diamond_{\ell} T\right) \underset{\mathcal{S}_{new}}{\Updownarrow} true \\
\Leftrightarrow & \left(\neg \Diamond_{\ell} T\right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} true\right) \\
\Leftrightarrow & \neg \Diamond_{\ell} T \wedge \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} true \,;\, true\right) \\
\Leftrightarrow & \neg \Diamond_{\ell} T \wedge \exists \mathcal{S}_{new_2} : \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} true \underset{\mathcal{S}_{new_2}}{\Updownarrow} true\right)
\end{aligned}
$$

Sync-Events sind über Chops definiert, für Chops gilt Zeile 3.11 in Lemma 3.2.1. Für diesen Fall muss in der Disjunktion $(\lceil true \rceil \,;\, F) \vee F$ das erste Disjunktionsglied erfüllt sein, da ein Sync-Event nicht zum Zeitpunkt Null auftreten kann. So ist die erste Äquivalenz begründet. Die zweite Äquivalenz benutzt die Aussage $true \Leftrightarrow true \,;\, true$. Abschließend wird ein neues Sync-Event eingeführt. Dies kann mit gebundener Umbenennung aus der Formel herausgezogen werden. Damit ist die gesuchte Form erreicht

Fall b:
Es ist $\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)$ von der gesuchten Form.

Fall 6 ($\mathcal{T} = \lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)$):

Fall a:

$$
\begin{aligned}
& \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)\right) \underset{\mathcal{S}_{new}}{\Updownarrow} true \\
\{\text{Assoziativität}\} \Leftrightarrow & \lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\left(\neg \Diamond^{L} T\right) \underset{\mathcal{S}_{new}}{\Updownarrow} true\right) \\
\Leftrightarrow & \lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\left(\neg \Diamond^{L} T\right) \wedge \left(true \underset{\mathcal{S}_{new}}{\Updownarrow} true\right)\right) \\
\{\text{Sync-Dist+}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} true\right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)\right) \wedge \\
& \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} true \underset{\mathcal{S}_{new}}{\Updownarrow} true\right) \\
\Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)\right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} true \underset{\mathcal{S}_{new}}{\Updownarrow} true\right)
\end{aligned}
$$

Die zweite Äquivalenz benutzt wiederum Zeile 3.11 in Lemma 3.2.1. Es kann jedoch nicht gefolgert werden, dass vor $\underset{\mathcal{S}_{new}}{\Updownarrow}$ eine Zeit lang *true* gelten muss, da vor dem ersten Sync-Event bereits eine Zeitspanne größer Null gegeben ist. In der letzten Äquivalenz wird das erste Konjunkt entfernt, da es von dem dritten impliziert wird.

Fall b:

$$
\begin{aligned}
& \lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)\right) \\
\{\text{Assoziativität}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil\right) \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right) \\
\{\text{Sync-Dist+}\} \Leftrightarrow & \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} true\right) \wedge \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} \left(\neg \Diamond^{L} T\right)\right)
\end{aligned}
$$

Alle Ausdrücke sind von der gesuchten Form.

Für den Fall, dass TF^1 nicht notwendigerweise eine Länge des gegebenen Intervalls größer Null erfordert, gilt folgende Äquivalenz:

$$
TF^1 \,;\, TF^2
$$

$$
\begin{aligned}
&\Leftrightarrow \big(TF^1 \wedge (\ell > 0 \vee \ell = 0)\big) \mathbin{;} TF^2 \\
&\Leftrightarrow \big((TF^1 \wedge \ell > 0) \vee (TF^1 \wedge \ell = 0)\big) \mathbin{;} TF^2 \\
&\Leftrightarrow \big((TF^1 \wedge \ell > 0) \mathbin{;} TF^2\big) \vee \big((TF^1 \wedge \ell = 0) \mathbin{;} TF^2\big) \\
&\Leftrightarrow \big((TF^1 \wedge \ell > 0) \mathbin{;} TF^2\big) \vee \big(TF^1 \wedge TF^2\big)
\end{aligned}
$$

Die letzte Äquivalenz erklärt sich wie folgt: Damit $TF^1 \wedge \ell = 0$ erfüllt ist, muss TF^1 die Gestalt $\bigvee \left(\bigwedge \left(\neg \Diamond_L T\right)\right)$ besitzen, alle anderen Terme in Zeile 3.17 implizieren eine Länge des Intervalls größer Null. Für Liveness-Eigenschaften ist aber nur die linke Intervallgrenze von Bedeutung, so dass $(TF^1 \wedge \ell = 0) \mathbin{;} TF^2$ äquivalent ist zu $TF^1 \wedge (\ell = 0 \mathbin{;} TF^2)$.

Der erste Fall ist soeben behandelt worden, der Zweite Fall wird nachfolgend gezeigt. Damit ist der Induktionsschluss für den Fall $TF^1 \mathbin{;} TF^2$ bewiesen.

Die Transformation von $TF^1 \wedge TF^2$ wird wie folgt durchgeführt: Mit gebundener Umbennenung wird die Quantifizierung nach außen gezogen. Die Distributivität zwischen Disjunktion und Konjunktion führt anschließend zu einer Formel in Normalform.
Die erste Äquivalenz gilt aufgrund der Induktionsannahme:

$$
\begin{aligned}
&TF^1 \wedge TF^2 \\
\Leftrightarrow & \left(\exists \mathcal{S}^1_{i^1 j^1 *} : \left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}^1_{i^1 j^1}\right)\right)\right) \wedge \left(\exists \mathcal{S}^2_{i^2 j^2 *} : \left(\bigvee_{i^2} \left(\bigwedge_{j^2} \mathcal{T}^2_{i^2 j^2}\right)\right)\right) \\
\Leftrightarrow & \exists \mathcal{S}'^1_{i^1 j^1 *} : \exists \mathcal{S}'^2_{i^2 j^2 *} : \left(\left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}'^1_{i^1 j^1}\right)\right) \wedge \left(\bigvee_{i^2} \left(\bigwedge_{j^2} \mathcal{T}'^2_{i^2 j^2}\right)\right)\right) \\
\Leftrightarrow & \exists \mathcal{S}'^1_{i^1 j^1 *} : \exists \mathcal{S}'^2_{i^2 j^2 *} : \left(\bigvee_{i^1} \bigvee_{i^2} \left(\left(\bigwedge_{j^1} \mathcal{T}'^1_{i^1 j^1}\right) \wedge \left(\bigwedge_{j^2} \mathcal{T}'^2_{i^2 j^2}\right)\right)\right)
\end{aligned}
$$

Die Formel ist in Normalform.

Für die Umformung von $TF^1 \vee TF^2$ werden nur die Induktionsannahme und gebundene Umbenennung benötigt:

$$
\begin{aligned}
&TF^1 \vee TF^2 \\
\Leftrightarrow & \left(\exists \mathcal{S}^1_{i^1 j^1 *} : \left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}^1_{i^1 j^1}\right)\right)\right) \vee \left(\exists \mathcal{S}^2_{i^2 j^2 *} : \left(\bigvee_{i^2} \left(\bigwedge_{j^2} \mathcal{T}^2_{i^2 j^2}\right)\right)\right) \\
\Leftrightarrow & \exists \mathcal{S}'^1_{i^1 j^1 *} : \exists \mathcal{S}'^2_{i^2 j^2 *} : \left(\left(\bigvee_{i^1} \left(\bigwedge_{j^1} \mathcal{T}'^1_{i^1 j^1}\right)\right) \vee \left(\bigvee_{i^2} \left(\bigwedge_{j^2} \mathcal{T}'^2_{i^2 j^2}\right)\right)\right)
\end{aligned}
$$

Auch diese Formel ist von der gesuchten Form.

Da für jeden der drei Fälle, entsprechend den drei Operatoren in *Testform*, nachgewiesen ist, dass zu einer Formel eine äquivalente Formel in Normalform existiert, ist die Aussage bewiesen. □

Es werden nur Formeln ohne Liveness in Kapitel 4 in Testautomaten übersetzt. Für Testformeln dieser Gestalt gilt ein stärkeres Resultat als obiges Theorem: Jede der *true*-Phasen in 3.17 dauert eine Zeit größer Null. Das nachfolgende Korollar fasst die Aussage formal.

Korollar 3.3.1 (Normalform für Testformeln ohne Liveness)
Jede Testformel ohne Liveness ist äquivalent zu einer Formel der Form

$$\exists \mathcal{S}_{ij*} : \left(\bigvee_i \left(\bigwedge_j \mathcal{T}_{ij} \right) \right), \tag{3.21}$$

wobei

$$\mathcal{T}_{ij} := Tr_{ij} \mid Tr_{ij} \underset{\mathcal{S}_{ij}}{\Updownarrow} \lceil true \rceil \mid \lceil true \rceil \underset{\mathcal{S}_{ij}}{\Updownarrow} Tr_{ij} \mid \lceil true \rceil \underset{\mathcal{S}_{ij1}}{\Updownarrow} Tr_{ij} \underset{\mathcal{S}_{ij2}}{\Updownarrow} \lceil true \rceil \tag{3.22}$$

$$Tr_{ij} := T_{ij} \mid \neg T_{ij}, \tag{3.23}$$

mit $T_{ij} \in \mathit{Trace}$, $* \in \{1, 2, \epsilon\}$ und $\mathcal{S}_{ij*}$ frischen booleschen Observablen, die nicht in einer der Formeln vorkommen.

Beweis
Jede Trace besitzt als erstes Element per Definition eine Phase mit $\ell > 0$. Aus diesem Grund kann im vorangegangenen Beweis in der Zeile 3.20 die Phase *true* durch $\lceil true \rceil$ ausgetauscht werden. Es gilt dann:

$$\begin{aligned}
&\exists : \left(\bigvee_{i^1} \bigvee_{i^2} \left(\bigwedge_{j^1} \left(\mathcal{T'}^1_{i^1 j^1} \underset{\mathcal{S}_{new}}{\Updownarrow} true \right) \wedge \bigwedge_{j^2} \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \mathcal{T'}^2_{i^2 j^2} \right) \right) \right) \\
\Leftrightarrow &\exists : \left(\bigvee_{i^1} \bigvee_{i^2} \left(\bigwedge_{j^1} \left(\mathcal{T'}^1_{i^1 j^1} \underset{\mathcal{S}_{new}}{\Updownarrow} \lceil true \rceil \right) \wedge \bigwedge_{j^2} \left(\lceil true \rceil \underset{\mathcal{S}_{new}}{\Updownarrow} \mathcal{T'}^2_{i^2 j^2} \right) \right) \right).
\end{aligned}$$

Die im obigen Beweis folgenden Äquivalenzen gelten unverändert mit $\lceil true \rceil$-Phasen. Einzig bei der Verwendung des besonderen Distributivgesetzes Sync-Dist+ muss die Beobachtung, dass $\lceil true \rceil$-Phasen und Traces eine Länge größer Null besitzen, gebraucht werden, um das entstehende *true* in ein $\lceil true \rceil$ zu ändern. □

In Zuständen von PEAs vergeht per Definition Zeit. Phasen in DC Formeln sollen auf Zustände in PEAs abgebildet werden. Ohne das gezeigte Korollar ergäbe sich ein Problem mit *true*-Phasen, welche von einem leeren Intervall erfüllt würden. Diese Phasen könnten nicht ohne weiteres von PEAs erkannt werden, es müssten zusätzliche Kanten eingefügt werden. Da obiges Korollar jede *true*-Phase in eine $\lceil true \rceil$-Phase überführt, ist die Hilfskonstruktion nicht notwendig.

Ein weiteres Problem ist, dass die Testautomaten des nachfolgenden Kapitels eine $\lceil true \rceil$-Phase am Ende eines jeden Konjunktionsglieds erwarten. Dies kann mit dem folgenden Korollar erreicht werden:

Korollar 3.3.2 (Model-Checkbare Darstellung)
Gegeben seien eine Testformel ohne Liveness TF und eine Interpretation $\mathcal{I}$. Dann gilt:

$$\exists t \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t] \models TF \Leftrightarrow \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models TF', \tag{3.24}$$

dabei hat TF' die Form

$$\exists \mathcal{S}_{ij*} : \left(\bigvee_i \left(\bigwedge_j \mathcal{T}_{ij} \right) \right), \tag{3.25}$$

wobei

$$\mathcal{T}_{ij} := Tr_{ij} \underset{\mathcal{S}_{ij}}{\Updownarrow} \lceil true \rceil \mid \lceil true \rceil \underset{\mathcal{S}_{ij1}}{\Updownarrow} Tr_{ij} \underset{\mathcal{S}_{ij2}}{\Updownarrow} \lceil true \rceil \tag{3.26}$$

$$Tr_{ij} := T_{ij} \mid \neg T_{ij}, \tag{3.27}$$

mit $T_{ij} \in \mathit{Trace}$, $* \in \{1, 2, \epsilon\}$ und $\mathcal{S}_{ij*}$ frischen booleschen Observablen, die nicht in einer der Formeln vorkommen.

Bemerkung 3.3.1
Gegeben seien eine Interpretation $\mathcal{I}$ und eine DC-Formel F. Es gilt:

$$\begin{aligned} & \exists t \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t] \models F \\ \Leftrightarrow & \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models F \mathbin{\boldsymbol{;}} \lceil true \rceil \\ \{\text{Sync-Intro}\} \Leftrightarrow & \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models \exists \mathcal{E} : F \underset{\mathcal{E}}{\Updownarrow} \lceil true \rceil \end{aligned}$$

Beweis (des Korollars 3.3.2)
Sei TF die gegebene Testformel. Dann gilt mit einer gegebenen Interpretation $\mathcal{I}$:

$$\begin{aligned} & \exists t \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t] \models TF \\ \{\text{Bemerkung}\} \Leftrightarrow & \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models \exists \mathcal{S} : TF \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil \\ \{\text{Korollar 3.3.1}\} \Leftrightarrow & \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models \exists \mathcal{S} : \left(\left(\exists \mathcal{S}_{ij*} : \left(\bigvee_i \left(\bigwedge_j \mathcal{T}_{ij} \right) \right) \right) \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil \right) \\ \Leftrightarrow & \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models \exists \mathcal{S} : \exists \mathcal{S}'_{ij*} : \left(\bigvee_i \left(\bigwedge_j \left(\mathcal{T}'_{ij} \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil \right) \right) \right) \end{aligned}$$

Für die letzte Äquivalenz wird die Argumentation aus dem Beweis des Normalformtheorems 3.3.1 auf die Formeldarstellung aus Zeile 3.22 in Korollar 3.3.1 angewandt: Zunächst werden mit gebundener Umbenennung die Quantoren nach außen gezogen, dann wird die Distributivität von Chop und Disjunktion genutzt. Abschließend wird Sync-Dist+ gebraucht. Die Argumentation $\lceil true \rceil \Leftrightarrow \lceil true \rceil \underset{S}{\Updownarrow} \lceil true \rceil$ führt zu der angegebenen Form.
Mit der Fallunterscheidung (Fall 1 – Fall 4) aus dem Beweis des Normalformtheorems wird die Darstellung der Konjunktionsglieder wie angegeben erreicht. $\square$

In Abschnitt 6.2 wird die Berechnung der model-checkbaren Darstellung zu einer gegebenen Testformel beispielhaft vorgestellt.

4 Testautomaten

4.1 Semantische Anpassung

Um die Frage beantworten zu können, ob ein gegebenes System von Phasen-Event-Automaten Ph eine negierte Testformel erfüllt, ist der Begriff der Erfüllbarkeit einer Formel durch einen Automaten zu präzisieren. Erfüllbarkeit von Formeln im Duration Calculus ist bezüglich Interpretationen und Intervallen definiert. Automaten besitzen als semantischen Bereich Abläufe in $\mathbf{Run}(Ph)$. Daher muss die Beziehung zwischen Interpretationen und Abläufen erklärt werden.

Definition 4.1.1 (Zu einem Run gehörige Interpretation)
Seien $X_1, \ldots, X_m$ Observablen mit Datenbereichen $D(X_1), \ldots, D(X_m)$ und seien $\mathcal{E}_1, \ldots, \mathcal{E}_n$ boolesche Observable. Sei $Ph = (P, V, A, C, E, s, I, P_0)$ ein Phasen-Event-Automat. Die Variablenmenge V bestehe aus den Observablen X_i, $V = \{X_1, \ldots, X_m\}$, die Eventmenge bestehe aus den Observablen $\mathcal{E}_i$, $A = \{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$. Sei mit

$$r := \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \in \mathbf{Run}(Ph)$$

ein Run von Ph gegeben. Eine Interpretation $\mathcal{I}$ heißt *zugehörig zu r*, falls gilt:

$$\mathcal{I}(X_i)(t) = \beta_j(X_i) \qquad \text{für alle } X_i \in V, \text{ für fast alle } t \in \left] \sum_{k=0}^{j-1} t_k, \sum_{k=0}^{j} t_k \right[, \text{für alle } j \in \mathbb{N}_0 \tag{4.1}$$

$$\mathcal{I}(\mathcal{E}_i)(t) = const \qquad \text{für alle } \mathcal{E}_i \in A, \text{ für fast alle } t \in \left] \sum_{k=0}^{j-1} t_k, \sum_{k=0}^{j} t_k \right[, \text{ für alle } j \in \mathbb{N}_0 \tag{4.2}$$

$$\begin{aligned} \mathcal{I}(\mathcal{E}_i)(t) \neq \mathcal{I}(\mathcal{E}_i)(t') \qquad & \text{für alle } \mathcal{E}_i \in Y_j, \text{ für fast alle } t \in \left] \sum_{k=0}^{j-1} t_k, \sum_{k=0}^{j} t_k \right[, \\ & \text{für fast alle } t' \in \left] \sum_{k=0}^{j} t_k, \sum_{k=0}^{j+1} t_k \right[, \text{ für alle } j \in \mathbb{N}_0 \end{aligned} \tag{4.3}$$

$$\mathcal{I}(\mathcal{E}_i)(t) = \mathcal{I}(\mathcal{E}_i)(t') \qquad \text{für alle } \mathcal{E}_i \notin Y_j, \text{ für fast alle } t \in \left] \sum_{k=0}^{j-1} t_k, \sum_{k=0}^{j} t_k \right[,$$

$$\text{für fast alle } t' \in \left]\sum_{k=0}^{j} t_k, \sum_{k=0}^{j+1} t_k\right[, \text{ für alle } j \in \mathbb{N}_0. \tag{4.4}$$

Dabei sei für die Summe $\sum_{k=0}^{-1} t_k := 0$ definiert. □

Die erste Gleichung der Definition sagt aus, dass die Observablen, die der PEA umfasst, dem Run entsprechend ausgewertet werden. Die zweite Gleichung verbietet das Auftreten von Events innerhalb von States im Run. Die dritte Gleichung fordert, dass sich die Interpretation der booleschen Observablen, die Events des Automaten darstellen, ändert, wenn der Automat von einem Zustand in den nächsten übergeht und sich die Observablen in der Eventmenge befinden. Die vierte Gleichung verbietet eine solche Änderung für den Fall, dass die Observablen nicht in der Eventmenge enthalten sind.

Durch die Belegungen in einem Run eines PEA wird eine Interpretation induziert. Damit gibt es zu jedem Run eine zugehörige Interpretation. Diese Aussage ist für den Nachweis der Korrektheit des Model-Checking Verfahrens von Bedeutung.

Lemma 4.1.1
Gegeben seien ein Phasen-Event-Automat $Ph = (P, V, A, C, E, s, I, P_0)$ sowie ein Ablauf $r \in \mathbf{Run}(Ph)$. Dann gibt es eine Interpretation $\mathcal{I}$, so dass $\mathcal{I}$ zu r gehört.

Beweis
Die Interpretation der Observablen $X_i \in V$ ist durch die Gleichung 4.1 in Definition 4.1.1 vorgegeben. Die Interpretation der Observablen, welche Events im Automaten darstellen, wird initial auf falsch gesetzt und dann den Zeilen 4.2 – 4.4 in Definition 4.1.1 entsprechend geändert. Die Interpretation von Observablen $X \notin (V \cup A)$ ist beliebig. Die so gewonnene Interpretation $\mathcal{I}$ ist zu r gehörig, weil sie die Forderungen 4.1 – 4.4 in Definition 4.1.1 erfüllt. □

In Abbildung 2.2 ist eine Interpretation der Observablen TP angegeben. Diese Interpretation $\mathcal{I}$ ordne der Observablen *sndAprMsg* die Funktion $\mathcal{I}(sndAprMsg)$ aus Abbil-

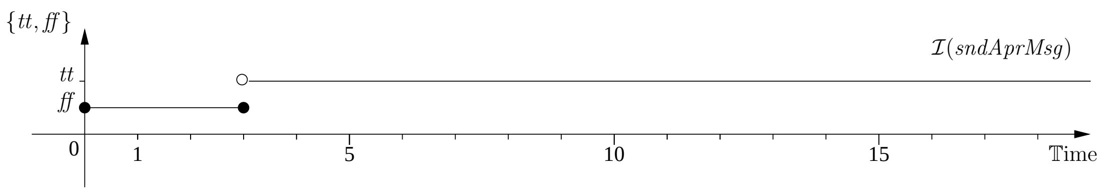

Abbildung 4.1: Interpretation $\mathcal{I}(sndAprMsg)$

dung 4.1 zu. Der Beginn eines Ablaufs des PEA aus Abbildung 2.4, zu dem die Interpretation $\mathcal{I}$ gehört, ist:

$\langle (S0, \beta_0(TP) = awy, \gamma_0(c1) = 0), 3, \{sndAprMsg\}, (S1, \beta_1(TP) = apr, \gamma_1(c1) = 0), \ldots \rangle$.

Die Interpretation $\mathcal{I}$ gehört nicht zu dem in Abschnitt 2.2 angegebenen Ablauf.

Die Erfüllbarkeit einer Formel durch einen Phasen-Event-Automaten ist über die Interpretationen definiert, die zu den Abläufen des Automaten gehören.

Definition 4.1.2 ($Ph \models_0 F$)
Ein Phasen-Event-Automat Ph *erfüllt* eine Formel $F \in Form$, $Ph \models_0 F$, falls alle Interpretationen $\mathcal{I}$, die zu einem Run $r \in \mathbf{Run}(Ph)$ gehören, die Formel von Null an erfüllen:

$$Ph \models_0 F :\Leftrightarrow \forall \mathcal{I} : \forall r \in \mathbf{Run}(Ph) : (\mathcal{I} \text{ gehörig zu } r \Rightarrow \mathcal{I} \models_0 F) \tag{4.5}$$

□

4.2 Testautomaten

Testautomaten sind Phasen-Event-Automaten, die einen ausgezeichneten Zustand, den sogenannten *Bad-State*, besitzen. In Abschnitt 4.4 wird zu jeder Testformel ein Testautomat konstruiert, so dass die Formel von einer Interpretation auf einem Intervall erfüllt wird genau dann, wenn der Testautomat seinen Bad-State in einem Run erreicht, zu dem die Interpretation gehört. Die Formeln in der Formelklasse *Testform* spezifizieren schlechtes Verhalten. Das Model-Checking Verfahren überprüft, ob das schlechte Verhalten vermieden wird, indem es den Testautomaten mit dem System parallel komponiert. Ist das Verhalten im System möglich, so wird der Bad-State des Testautomaten erreicht. Auf diese Weise ist die Namensgebung begründet.

Definition 4.2.1 (Testautomat)
Ein *Testautomat* TA ist definiert als Tupel

$$TA := (P, V, A, C, E, s, I, P_0, p_{Bad}), \tag{4.6}$$

wobei $(P, V, A, C, E, s, I, P_0)$ ein Phasen-Event-Automat ist und $p_{Bad} \in P$ ein ausgezeichneter Zustand, genannt *Bad-State*. □

Die Abläufe eines Testautomaten stimmen mit den Abläufen des zugrunde liegenden PEA überein.

Definition 4.2.2 (Run)
Sei $TA = (Ph, p_{Bad})$ ein Testautomat, dann ist $\mathbf{Run}(TA) := \mathbf{Run}(Ph)$. Ein Run

$$r := \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \in \mathbf{Run}(TA)$$

heißt *Fehlerablauf*, falls es eine Konfiguration $(p_{Bad}, \beta_i, \gamma_i)$, $i \in \mathbb{N}_0$, in r gibt. □

Die Testautomatensemantik für Formeln der Klasse *Testform* benötigt neben der Parallelkomposition von Testautomaten die sequentielle Komposition zweier Testautomaten sowie das Hiding von Events in Testautomaten.
Die Parallelkomposition von Testautomaten entspricht der Parallelkomposition der PEAs. Das Tupel der Bad-States in den einzelnen Testautomaten stellt den Bad-State im parallel-komponierten Automaten dar.

Definition 4.2.3 (Parallele Komposition von Testautomaten)
Seien $TA_i = (Ph_i, p_{Bad,i})$, $i \in \{1, 2\}$ Testautomaten mit disjunkten Uhrenmengen, $C_1 \cap C_2 = \emptyset$. Dann ist die *parallele Komposition* definiert als:

$$TA_1 \parallel TA_2 := (Ph_1 \parallel Ph_2, (p_{Bad,1}, p_{Bad,2}))\,. \tag{4.7}$$

□

Die sequentielle Komposition zweier Testautomaten führt zu einem Testautomaten, welcher der Hintereinanderausführung der beiden gegebenen Automaten entspricht. Der Bad-State des komponierten Automaten wird vom zweiten Testautomaten übernommen. Ferner wird beim Übergang vom ersten zum zweiten Testautomaten ein Event gefordert. Die Idee dabei ist: Der Testautomat stellt die Semantik einer Testformel in model-checkbarer Darstellung dar. Das Event entspricht dann einem Sync-Event in der gegebenen Formel. Auf diese Weise synchronisieren sich parallel-komponierte Testautomaten, die konjugierten Formeln entsprechen. Auch wird beim Übergang zu jedem Zustand ein Guard für die Übergangskante ermittelt. Daher fordert die sequentielle Komposition eine Funktion in den Zuständen als Parameter.

Definition 4.2.4 (Sequentielle Komposition von Testautomaten)
Seien $TA_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, P_{0,i}, p_{Bad,i})$, $i \in \{1, 2\}$ Testautomaten mit disjunkten Uhrenmengen, $C_1 \cap C_2 = \emptyset$, und disjunkten States, $P_1 \cap P_2 = \emptyset$. Seien $\mathcal{S} \notin A_1 \cup A_2$ ein Event und $\gamma : P_1 \to \mathcal{L}(V_1 \cup V_1' \cup A_1 \cup C_1)$ eine Abbildung, die für jede Phase einen Guard liefert. Dann ist die *sequentielle Komposition* definiert als:

$$TA_1 \bullet_{\{\mathcal{S}\},\{\gamma\}} TA_2 := (P, V, A, C, E, s, I, P_0, p_{Bad})\,, \tag{4.8}$$

wobei

$$\begin{aligned}
P :=& P_1 \cup P_2 && \text{(Vereinigung der Zustände)} && (4.9)\\
V :=& V_1 \cup V_2 && \text{(Vereinigung der Variablen)} && (4.10)\\
A :=& A_1 \cup A_2 \cup \{\mathcal{S}\} && \text{(Vereinigung der Eventmengen mit } \{\mathcal{S}\}\text{)} && (4.11)\\
C :=& C_1 \cup C_2 && \text{(Vereinigung der Uhrenmengen)} && (4.12)\\
s :=& s_1 \cup s_2 && \text{(Vereinigung der Zustandsinvarianten)} && (4.13)\\
I :=& I_1 \cup I_2 && \text{(Vereinigung der Uhrenbedingungn)} && (4.14)\\
P_0 :=& P_{0,1} && \text{(Startzustände von } TA_1\text{)} && (4.15)\\
p_{Bad} :=& p_{Bad,2} && \text{(Bad-State von } TA_2\text{)} && (4.16)
\end{aligned}$$

Die Menge der Transitionen ergibt sich aus der Vereinigung der Transitionen von TA_1 und TA_2. Hinzu kommen Transitionen von jedem Zustand aus TA_1 zu jedem Startzustand von TA_2. Der Guard wird von der Funktion γ und dem gegebenen Event $\mathcal{S}$ bestimmt.

$$E := E_1 \cup E_2 \cup \{(p, \mathcal{S} \wedge \gamma(p), C_1 \cup C_2, p_0) \mid p \in P_1, p_0 \in P_{0,2}\} \tag{4.17}$$

□

Sync-Events treten einmalig auf. In PEAs dürfen Events, die in der Eventmenge des Automaten enthalten sind, an jeder Kante auftreten, an der sie nicht explizit verboten sind. Um Sync-Events an allen Kanten außer den Übergangskanten der sequentiellen Komposition zweier Automaten zu verbieten, wird der *Hiding-Operator* definiert.

Definition 4.2.5 (Hiding in Testautomaten)
Seien $TA = (P, V, A, C, E, s, I, P_0, p_{Bad})$ ein Testautomat und $\mathcal{S}$ ein Event. Dann ist $TA \setminus \{\mathcal{S}\} := (P, V, A \cup \{\mathcal{S}\}, C, E', s, I, P_0, p_{bad})$ *der Automat, der aus* TA *hervorgeht, indem* $\mathcal{S}$ *versteckt wird*. Die Sprechweise ist auch TA *hides* $\mathcal{S}$. Dabei ist E' definiert als

$$\begin{aligned} E' := &\{(p, g \wedge \neg\mathcal{S}, X, p') \mid (p, g, X, p') \in E \wedge \mathcal{S} \notin \mathit{Var}(g)\} \cup \\ &\{(p, g, X, p') \mid (p, g, X, p') \in E \wedge \mathcal{S} \in \mathit{Var}(g)\}, \end{aligned} \tag{4.18}$$

d.h. die Kanten werden aus E übernommen. Falls der Guard einer Kanten $\mathcal{S}$ nicht enthält, so wird $\neg\mathcal{S}$ dem Guard hinzugefügt, falls der Guard $\mathcal{S}$ enthält, so bleibt die Kante unverändert. □

4.3 Potenzmengenkonstruktion

Es wird aufgezeigt, wie aus einer gegebenen Trace ein Phasen-Event-Automat konstruiert wird, so dass eine Interpretation und ein Intervall ein Präfix der Trace genau dann erfüllen, wenn für den PEA eine Funktion den Wert *true* liefert. Die hier vorgestellten Definitionen sind aus [Hoe05a] übernommen, die Funktionen *enter* und *complete* sind vereinfacht worden, weil in dieser Arbeit keine Phasen betrachtet werden, die von einem leeren Intervall erfüllt werden. Die angeführten Resultate aus [Hoe05a] dürfen angewandt werden, weil die in [Hoe05a] betrachtete Klasse der Countertraces die hier genutzen Traces abdeckt.

Testformeln setzen sich aus Traces zusammen, welche aus Phasen und Events bestehen. Im vorliegenden Abschnitt werden nur die Traces betrachtet, deren letztes Element eine Phase ist. Da die Konstruktion des Automaten sehr aufwendig zu beschreiben ist, werden zunächst syntaktische Elemente eingeführt, welche die Erklärung vereinfachen.

Definition 4.3.1 (Syntax für Phasen und Traces)
Eine Trace tr setzt sich zusammen aus Phasen $phase = \ell > 0 \wedge \ell \sim k \wedge \bigwedge_{i=1}^{m} \lceil \varphi_i \rceil \wedge \bigwedge_{j=1}^{n} \boxminus \mathcal{E}_j$, Events $\updownarrow \mathcal{E}$ und verbotenen Events $\not\updownarrow \mathcal{E}$, dabei seien $\sim \in \{\emptyset, \leq <, >, \geq\}$, $k \in$

$\mathbb{R}_{>0}$, φ_i zeitbehaftete Zustandsausdrücke und $\mathcal{E}, \mathcal{E}_j \in SVar$ boolesche Observable, $(1 \leq i \leq m, 1 \leq j \leq n)$.

- Es bezeichne

$$phase.timeop := \sim, \tag{4.19}$$

$$phase.bound := k, \tag{4.20}$$

$$phase.inv := \bigwedge_{i=1}^{m} \lceil \varphi_i \rceil, \tag{4.21}$$

$$phase.forbidden := \{\mathcal{E}_1, \ldots, \mathcal{E}_n\}. \tag{4.22}$$

  Für Eventspezifikationen[1] gilt: $\phi(\mathcal{E}_1) ; \ldots ; \phi(\mathcal{E}_r) \Leftrightarrow \bigwedge_{k=1}^{r} \phi(\mathcal{E}_k)$. Weil auf eine Folge von Eventspezifikationen immer eine Phase folgt, kann mittels

$$phase.enterEvents := \bigwedge_{k=1}^{r} \phi(\mathcal{E}_k) \tag{4.23}$$

  auf alle Events unmittelbar, vor einer Phase zugegriffen werden.

- Es genügt daher, eine Trace als Folge von Phasen zu verstehen. Die Phasen sind der Einfachheit halber durchnummeriert, mit $tr(i)$ wird auf die i-te Phase der Trace zugegriffen. Die Menge aller Phasen einer Trace wird mit $tr.set$ bezeichnet.

- Eine Trace besitzt die *Länge* n, $length(tr) = n$, falls die Trace n Phasen hat. Soll nicht die Trace, sondern nur das *Präfix der Länge* $m \leq length(tr)$, d.h. die ersten m Phasen, betrachtet werden, so bezeichnet die Formel $Prefix_m(tr)$ die Folge der ersten m Phasen und Events bis einschließlich der Phase $tr(m)$.

- Es ist oft notwendig zu entscheiden, welchen Operator eine Phase besitzt. Es werden folgende Mengen eingeführt:

$$LB(tr) := \{tr(i) \mid tr(i).timeop \in \{>, \geq\}\}, \qquad \text{(Untere Schranken)} \tag{4.24}$$

$$UB(tr) := \{tr(i) \mid tr(i).timeop \in \{<, \leq\}\}, \qquad \text{(Obere Schranken)} \tag{4.25}$$

$\square$

Der PEA zu einer gegebenen Trace besitzt als Zustände p alle Teilmengen $p.in$ von Phasen der Trace. Die Trace ist bis zu dem maximalen Index in $p.in$ erkannt worden. Phasen in $p.in$ werden auch als *aktiv* bezeichnet. Die Menge wird weiter aufgeteilt in wartende DC Phasen in $p.wait$, die noch nicht vollständig vom Automaten erkannt wurden. Die DC Phasen tragen als Operatoren $\geq$ oder $>$. Die Menge $p.gteq$ repräsentiert die DC Phasen mit einem $\geq$ Operator. Analog werden die DC Phasen in $p.in$ mit $<$ Operator durch $p.less$ dargestellt.

[1] dargestellt als $\phi(\mathcal{E})$ mit $\phi(\mathcal{E}) = \updownarrow \mathcal{E}$ oder $\phi(\mathcal{E}) = \not\updownarrow \mathcal{E}$

Definition 4.3.2 ($PowerSet(tr)$)
Gegeben sei eine Trace tr. Die Zustände p des zu konstruierenden Automaten $\mathcal{P}(tr)$ sind Quadrupel $p = (p.in, p.wait, p.gteq, p.less)$ mit

$$p.in \subseteq tr.dom \tag{4.26}$$
$$p.wait \subseteq p.in \cap LB(tr) \tag{4.27}$$
$$p.gteq \subseteq p.wait \tag{4.28}$$
$$p.less \subseteq p.in \cap UB(tr) \tag{4.29}$$

Die Menge aller möglichen Zustände wird als $PowerSet(tr)$ bezeichnet. □

Der Name $PowerSet(tr)$ ergibt sich aus der Definition der Zustände über alle Teilmengen $p.in$ von $tr.set$. Es sei hierbei angemerkt, dass die Menge der Zustände des PEAs exponentiell in der Anzahl der Phasen der Trace, in der Mächtigkeit von $LB(tr)$ und in der Mächtigkeit von $UB(tr)$ wächst.

Bemerkung 4.3.1 (Komplexität des Zustandsraums)
Gegeben sei eine m elementige Trace. Es gelte $LB(tr) \neq \emptyset$, $UB(tr) \neq \emptyset$. Für eine gegebene Menge mit k Elementen sei $|\, LB(tr) \cap k \,|$ die Mächtigkeit der Menge, die als Schnitt der gegebenen Menge mit $LB(tr)$ entsteht. Es gilt:

$$4^m \geq |\, PowerSet(tr) \,| \geq \Sigma_{k=0}^{m} \binom{m}{k} * 2^{|LB(tr) \cap k|} * 2^{|UB(tr) \cap k|} > \Sigma_{k=0}^{m} \binom{m}{k} = 2^m$$

Besitzen fast alle Phasen einen Bound, so geht der Term $2^{|LB(tr) \cap k|} * 2^{|UB(tr) \cap k|}$ mit nahezu 2^k in die Ungleichung ein.

Die Abschätzung gibt den Worst-Case an, oft ist nur ein Bruchteil der Phasen erreichbar.

Beweis
Die obere Schranke ergibt sich aus dem Worst-Case, dass alle Phasen einen $\geq$-Bound besitzen. Das folgende $\geq$ entsteht durch das Weglassen der Menge $p.gteq$. Das $>$ liegt in $LB(tr) \neq \emptyset$ und $UB(tr) \neq \emptyset$ und damit $2^{|LB(tr) \cap k|} * 2^{|UB(tr) \cap k|} > 1$ begründet. □

Um die Definition von $\mathcal{P}(tr)$ übersichtlicher zu gestalten, werden im Folgenden einige Hilfsfunktionen definiert:

canseep: Ist in der Trace die Folge $\lceil A \rceil ; \lceil B \rceil$ enthalten, so kann auch die Phase $\lceil B \rceil$ beim Betreten der Phasen $\lceil A \rceil$ zu $p.in$ hinzugefügt werden, da gilt $\lceil A \wedge B \rceil \Rightarrow \lceil A \rceil ; \lceil B \rceil$. Es können all die Nachfolgephasen hinzugefügt werden, deren Vorgängerphasen keine untere Schranke in der Dauer besitzen und die selber keine Events zum Betreten erfordern. Die Funktion *canseep* gibt zu einer Trace und einer Phase des Automaten an, in welche weiteren DC Phasen auf obige Weise hineingerutscht wird.

keep: Die Funktion *keep* nimmt als Parameter eine Trace, einen Zustand des Automaten und eine Belegung der Variablen, Events und Uhren des Automaten. Die Funktion liefert die Menge an DC Phasen, in denen sich der Automat zur Zeit befindet und in denen er bleiben kann. Dazu muss eine DC Phase in *p.in* enthalten sein, die Belegung muss die Invariante der Phase erfüllen und es dürfen keine verbotenen Events auftreten. Die Invariante und die Events seien dabei als einfache prädikatenlogische Formeln verstanden, die von der Belegung erfüllt werden müssen. Die Invariante wird durch die Konjunktion der Zustandsausdrücke als Formel dargestellt, die Events durch die Negation der Eventvariablen, $\neg\mathcal{E}$.

enter: Enter bestimmt unter den Parametern Trace, Zustand und Belegung die DC Phasen, welche von p aus betreten werden können. Eine DC Phase kann betreten werden, falls die vorhergehende Phase aktiv, d.h. in *p.in* enthalten ist, falls die *enterEvents* der zu betretenden Phase auftreten und deren Invariante erfüllt ist. Dabei werden *enterEvents* und Invariante wiederum als prädikatenlogische Formeln aufgefasst. Ist die vorangehende Phase in der Menge *p.wait* enthalten, so kann die neue Phase nur betreten werden, falls die vorangehende Phase einen $\geq$ Bound hatte und die Uhrenbelegung diesen Bound erfüllt. Besitzt die vorangegangene Phase keinen $\geq$ Bound, so darf noch nicht in die neue Phase gewechselt werden, weil noch keine Zeitdauer beobachtet worden ist, die den $>$ Bound erfüllt. Ist die vorangehende Phase in *p.less* enthalten, so muss die Belegung der zugehörigen Uhrenvariablen den gegebenen $<$ Bound erfüllen.

seep: Die Funktion *seep* nimmt als Parameter eine Trace, einen Zustand und eine Belegung entgegen. Sie ermittelt alle Phasen, die unmittelbar mit den Phasen in *p.in* betreten werden können. Zur Bestimmung der Menge wird die Funktion *canseep* benutzt. Es werden jedoch nur diejenigen Phasen von *seep* ausgewählt, deren Invariante durch die Belegung erfüllt wird.

Definition 4.3.3 (*canseep*(tr, p), *keep*(tr, p, val), *enter*(tr, p, val), *seep*(tr, p, val))
Seien eine Trace tr, ein Zustand $p \in PowerSet(tr)$ und eine Belegung $val \in Val$ der Variablen in V, der Events in A und der Uhren in C des zu konstruierenden Automaten gegeben.
Die Funktion *canseep* ordnet der Trace und dem Zustand eine Menge an Phasen der Trace zu. Es gilt:

$$\begin{aligned} tr(i) \in canseep(tr, p) :\Leftrightarrow & tr(i-1) \in p.in \setminus p.wait \wedge \\ & tr(i).enterEvents = \bigwedge_{i=1}^{k} \not\Updownarrow \mathcal{E}_i, k \in \mathbb{N} \end{aligned}$$

Die Funktionen *keep*, *enter* und *seep* ordnen Trace, Zustand und Belegung eine Menge von Phasen der Trace zu:

$$tr(i) \in keep(tr, p, val) :\Leftrightarrow tr(i) \in p.in \wedge val \models \bigwedge_{\mathcal{E} \in tr(i).forbidden} \neg\mathcal{E} \wedge tr(i).inv$$

$$
\begin{aligned}
tr(i) \in enter(tr, p, val) :\Leftrightarrow \\
&tr(i-1) \in p.in \wedge \\
&val \models tr(i).enterEvents \wedge \\
&val \models tr(i).inv \wedge \\
&(tr(i-1) \in p.wait \Rightarrow tr(i-1) \in p.gteq \wedge val \models c_{i-1} \geq tr(i-1).bound) \\
&(tr(i-1) \in p.less \Rightarrow val \models c_{i-1} < tr(i-1).bound)
\end{aligned}
$$

$$tr(i) \in seep(tr, p, val) :\Leftrightarrow tr(i) \in canseep(tr, p) \wedge val \models tr(i).inv$$

□

Die Funktion *successor* benutzt selbige Hilfsfunktionen, um unter einer gegebenen Belegung zu einem Zustand $p \in PowerSet(tr)$, tr eine gegebene Trace, den Nachfolgezustand $p' \in PowerSet(tr)$ zu bestimmen. Der Nachfolgezustand besteht aus den folgenden Mengen:

$p'.in$: Die Menge der DC Phasen, die im Nachfolgezustand aktiv sind, setzt sich zusammen aus den Phasen,

a) in denen der Automat bleiben kann,

b) die betreten werden können und

c) in die hineingerutscht wird.

Formal ist dies die Vereinigung der Mengen *keep*(tr, p, val), *enter*(tr, p, val) und *seep*(tr, p', val). Dabei wird *seep* rekursiv für die neu ermittelte Phase aufgerufen. In [Hoe05a] wird bemerkt, dass die Konstruktion deterministisch zu einem Nachfolgezustand führt.

$p'.wait$: Es müssen alle Phasen aus (b) und (c) mit einer unteren Schranke in die Menge $p'.wait$ aufgenommen werden. Aus der vorhergehenden Menge $p.wait$ bleiben die Phasen in der Menge $p'.wait$, deren Uhr noch nicht die Schranke erreicht hat.

$p'.gteq$: Eine Phase ist in der Menge $p'.gteq$, wenn sie in $p'.wait$ vorhanden und der Operator $\geq$ ist. Ferner muss die Phase mit *enter* betreten aber nicht durch *keep* bewahrt worden sein. Darüber hinaus bleiben alle Phasen, die in $p.gteq$ liegen und mit *keep* bewahrt werden, in $p'.gteq$. Eine Phase, die zwar einen Bound $\geq$ besitzt, aber mit Durchrutschen erreicht wird, wird der Menge $p'.gteq$ nicht hinzugefügt. Da die Phase zu früh betreten worden ist, muss für sie der $>$ Bound erfüllt werden.

$p'.less$: Eine Phase ist in der Menge $p'.less$ vorhanden, falls sie eine obere Schranke besitzt und nicht immer wieder durch erneutes Hineinrutschen erreicht werden kann. Ferner muss die Phase den Operator $<$ tragen. Ist die Phase durch Hineinrutschen erreicht worden und kann nicht betreten werden, so muss ein $\leq$ Operator der Phase wie ein $<$ behandelt werden, weil die Phase bereits einen kurzen Moment beobachtet wurde, bevor sie aktiv war. Daher muss auch eine Phase mit $\leq$ Operator, die mit *canseep* erreicht wird, der Menge hinzugefügt werden.

Definition 4.3.4
Gegeben seien eine Trace tr, ein Zustand p und eine Belegung $val \in Val$. Die Abbildung *successor* ermittelt mit diesen Parametern einen Nachfolgezustand $successor(tr, p, val) = (p'.in, p'.wait, p'.gteq, p'.less) \in PowerSet(tr)$ wie folgt:

$$\begin{aligned}
p'.in :=& keep(tr, p, val) \cup enter(tr, p, val) \cup seep(tr, p', val)\\
p'.wait :=& (LB(tr) \cap p'.in) \setminus keep(tr, p, val) \cup\\
& \{tr(i) \in p.wait \mid val \models c_i < tr(i).bound\}\\
tr(i) \in p'.gteq :\Leftrightarrow& tr(i) \in p'.wait \wedge tr(i).timeop = \geq \wedge\\
& tr(i) \in (p.gteq \cap keep(tr, p, val)) \cup (enter(tr, p, val) \setminus keep(tr, p, val))\\
tr(i) \in p'.less :\Leftrightarrow& tr(i) \in (UB(tr) \cap p'.in) \setminus canseep(tr, p') \wedge\\
& (tr(i).timeop = < \vee\, tr(i) \in canseep(tr, p) \setminus enter(tr, p, val))
\end{aligned}$$

□

Der Automat zu einer Trace tr, $\mathcal{P}(tr) = (P, V, A, C, E, s, I, P_0)$, ist über die folgenden Komponenten definiert:

P: Der Phasen-Event-Automat $\mathcal{P}(tr)$, der zu einer Trace konstruiert wird, besitzt als Zustände die Menge $PowerSet(tr)$.

V: Die Variablen in V sind gerade die Observablen in den Zustandsausdrücken von tr, d.h. der Automat und die Trace besitzen die gleichen syntaktischen Elemente, aber mit unterschiedlicher Semantik: Die Variable X im Automaten wird mit einem Wert belegt, während der Observablen X durch eine Interpretation eine Funktion zugeordnet wird.

A: Analog sind die booleschen Observablen, die in Eventspezifikationen der Trace vorkommen, gerade die Events in der Menge A des Automaten, wiederum mit anderer Semantik.

C: Für jede Phase, die einen Bound trägt, wird eine Uhr im Automaten eingeführt.

s: Die Zustandsinvarianten $s(p)$ ergeben sich aus den zeitbehafteten Zustandsausdrücken innerhalb der Phasen, interpretiert als prädikatenlogische Formeln.

I: Für jede Phase $tr(i)$ mit oberer Schranke, die in einem Zustand aktiv ist, aber nicht immer wieder durch *seep* erreicht werden kann, wird die Bedingung $c_i \leq tr(i).bound$ der Uhreninvariante hinzugefügt. Außerdem wird die Forderung für jede Phase $tr(i) \in p.wait$ gestellt.

P_0: Jeder Zustand p_0 in der Menge der Startzustände muss die erste Phase der Trace enthalten, $tr(1) \in p_0.in$. Außerdem werden die Phasen hinzugenommen, die mit *seep* unter einer gewissen Belegung erreichbar sind. Die Phasen, die eine untere Schranke haben, werden der Menge $p_0.wait$ zugefügt, die Menge $p_0.less$ enthält $tr(1)$, falls $tr(1).timeop = <$. Die Menge $p.gteq$ umfasst die erste Phase $tr(1)$, falls der Operator $\geq$ ist.

E: Die Transitionsrelation besitzt Elemente (p, g, X, p'). Der Folgezustand p' ergibt sich aus der Anwendung der *successor* Funktion mit einer Belegung val. Alle Bedingungen, die bei der Anwendung von *successor*, *keep*, *enter* und *seep* gestellt werden, ergeben konjugiert den Guard g. Für eine Wartephase wird die Uhr nur dann zurückgesetzt, wenn die Phase nicht bewahrt werden kann. In einer Phase mit oberer Schranke wird die Uhr so oft wie möglich zurückgesetzt, d.h. sobald die Phase neu betreten werden kann. Für rekursiv durch *seep* betretene Phasen wird die Uhr nicht zurückgesetzt.

Definition 4.3.5 ($\mathcal{P}(tr)$)
Gegeben sei eine Trace tr, dann ist $\mathcal{P}(tr) := (P, V, A, C, E, s, I, P_0)$ definiert mittels:

$$
\begin{aligned}
P :=& PowerSet(tr) \\
V :=& \{X \in SVar \mid X \text{ ist in einem Zustandsausdruck von } tr \text{ enthalten}\} \\
A :=& \{X \in SVar \mid X \text{ ist in einer Eventspezifikation von } tr \text{ enthalten}\} \\
C :=& \{c_i \mid tr(i) \in UB(tr) \cup LB(tr)\} \\
s(p) :=& \bigwedge_{tr(i) \in p.in} tr(i).inv \wedge \bigwedge_{tr(i) \in canseep(tr,p) \setminus p.in} \neg tr(i).inv \\
I(p) :=& \bigwedge_{tr(i) \in p.wait \cup (UB(tr) \cap (p.in \setminus canseep(tr,p)))} c_i \leq tr(i).bounds \\
P_0 :=& \{p \in P \mid \exists val \in Val : \\
& \quad p.in = \{tr(1)\} \cup seep(tr, p, val) \\
& \quad p.wait = p.in \cap LB(tr) \\
& \quad p.less = \begin{cases} \{tr(1)\}, & tr(1).timeop = < \\ \emptyset, & \text{sonst,} \end{cases} \\
& \quad p.gteq = \begin{cases} \{tr(1)\}, & tr(1).timeop = \geq \\ \emptyset, & \text{sonst,} \end{cases}
\end{aligned}
$$

die Transitionsrelation ist definiert durch

$$
\begin{aligned}
E := \{(p, g, X, p') \mid & p \in P \wedge \exists val \in Val : \\
& p' = successor(tr, p, val) \wedge \\
& g = \text{Konjunktion aller Bedingungen, die bei der} \\
& \qquad \text{Berechnung von } successor(tr, p, val) \text{ aufgetreten sind}
\end{aligned}
$$

$$
\begin{aligned}
X = \{c_i \in C \mid (tr(i) \in p'.wait \setminus keep(tr, p, val)) \vee \\
tr(i) \in ((UB(tr) \cap p'.in) \setminus seep(tr, p', val)) \cap \\
(enter(tr, p, val) \cup seep(tr, p, val))\} \qquad \}
\end{aligned}
$$

□

Die Funktion $complete_{tr,i}$ prüft, ob das Präfix $Prefix_i(tr)$ im Zustand p des Automaten $\mathcal{P}(tr)$ unter einer gegebenen Belegung γ erfüllt ist. Der Definition schließt sich ein zentrales Lemma an, das die Äquivalenz von Abläufen in $\mathcal{P}(tr)$ und der Gültigkeit von tr in einem Intervall aufzeigt. Es ist ein zentrales Resultat in [Hoe05a] und wird hier ohne Beweis wiedergegeben.

Definition 4.3.6 ($complete_{tr,i}(p, \gamma)$)
Gegeben seien eine Trace tr, eine Phase der Trace $tr(i)$ mit $i \leq length(tr)$, ein Zustand p des Automaten $\mathcal{P}(tr)$ und eine Uhrenbelegung γ.

$$
\begin{aligned}
complete_{tr,i}(p, \gamma) :\Leftrightarrow & tr(i) \in p.in \wedge \\
& (tr(i) \in p.wait \Rightarrow tr(i) \in p.gteq \wedge \gamma \models c_i \geq tr(i).bound) \wedge \\
& (tr(i) \in p.less \Rightarrow \gamma \models c_i < tr(i).bound)
\end{aligned}
$$

□

Lemma 4.3.1 ([Hoe05a])
Gegeben seien eine Trace tr, ein Index $m \leq length(tr)$, eine Interpretation $\mathcal{I}$, ein Zeitpunkt $t \in \mathbb{R}_{\geq 0}$ und ein Ablauf $r = \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr))$ mit einer Konfiguration (p_k, β_k, γ_k), $k \in \mathbb{N}_0$, so dass $\mathcal{I}$ zu r gehört und $\sum_{j=0}^{k} t_j = t$. Dann gilt:

$$
complete_{tr,m}(p_k, (\gamma_k + t_k)) \Leftrightarrow \mathcal{I}, [0, t] \models Prefix_m(tr)
$$

Beweis
Der Beweis ist in [Hoe05a] zu finden. □

Die Frage, ob eine Interpretation auf einem Intervall $[0, t]$ ein Präfix einer Trace erfüllt, wird reduziert auf die Frage, ob sich ein Lauf, zu dem die Interpretation gehört, zum Zeitpunkt t in einem Zustand befindet, so dass die *complete* Funktion den Wert *true* liefert. Das nachfolgende Lemma sagt aus, dass es zu jeder Interpretation einen Ablauf des Automaten gibt, so dass die Interpretation zu dem Ablauf gehört.

Lemma 4.3.2 ([Hoe05a])
Gegeben seien eine Trace tr, eine Interpretation $\mathcal{I}$ und ein Zeitpunkt $t \in \mathbb{R}_{\geq 0}$. Dann gibt es einen unendlichen Ablauf $r = \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr))$, so dass $\mathcal{I}$ zu r gehört und dass r einen Zustand p_k zum Zeitpunkt $t = \sum_{j=0}^{k-1} t_k$ erreicht, $k \in \mathbb{N}$.

Beweis
Die Aussage ist gültig, da in jedem Zustand zu jeder Belegung der Variablen ein Nachfolgezustand in $\mathcal{P}(tr)$ berechnet worden ist. Dass ein Zustand genau zum geforderten

Zeitpunkt erreicht wird, ist mit der Verwendung der Stotterkanten zu begründen. Eine genauere Argumentation findet sich in [Hoe05a]. □

Im nachfolgenden Kapitel werden die Automaten $\mathcal{P}(tr)$ zu Testautomaten erweitert. Ziel ist ein Theorem über Testformeln ähnlich Lemma 4.3.1 und 4.3.2.

4.4 Testautomatensemantik

Die in Abschnitt 4.3 vorgestellte Potenzmengenkonstruktion dient in [Hoe05a] als Semantik für Countertraces. Sie soll zu einer Semantik für Testformeln in model-checkbarer Darstellung ergänzt werden.
Eine Testformel in model-checkbarer Darstellung besteht gegebenenfalls aus einer führenden $\lceil true \rceil$-Phase, gefolgt von einem Sync-Event und einer Trace oder negierten Trace. Anschließend wird ein weiteres Sync-Event erwartet. Am Ende steht wiederum eine $\lceil true \rceil$-Phase. Abbildung 4.2 verdeutlicht, wie zu diesen Formeln ein Testautomat konstruiert wird. Der Testautomat besitzt einen Startzustand mit Invariante $true$. Von

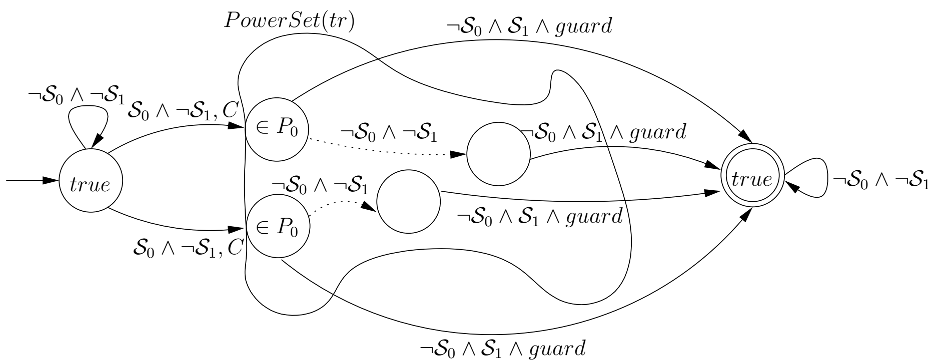

Abbildung 4.2: Testautomaten (schematische Darstellung)

diesem Startzustand führt eine Kante zu jedem Startzustand des Trace-Automaten $\mathcal{P}(tr)$, die mit $\mathcal{S}_0$, dem ersten Sync-Event, beschriftet ist. Der Trace-Automat besitzt alle Zustände und Kanten der Potenzmengenkonstruktion, angedeutet durch die gestrichelten Pfeile und den Zustandsraum $PowerSet(tr)$. Zusätzlich wird dem Zustandsraum des Testautomaten ein Bad-State mit Invariante $true$ hinzugefügt. Von jedem Zustand in $\mathcal{P}(tr)$ wird eine Kante zum Bad-State eingefügt, die das zweite Sync-Event, $\mathcal{S}_1$, erwartet und mit einem Guard beschriftet ist. Mit dem Hiding-Operator werden die Sync-Events an allen Transitionen verboten, an denen sie nicht auftreten. Damit wird die Eigenschaft der Sync-Events, nur einmalig auftreten zu dürfen, nachgeahmt. Die Synchronisation der Formeln wird über die Parallelkomposition auf Automatenebene erreicht. Damit ist die Semantik der Testformeln kompositionell definiert, abgesehen von der Konstruktion der Trace-Automaten.

Die nachstehende Hilfsfunktion nimmt als Parameter einen Zustand und liefert eine prädikatenlogische Formel. Diese Formel dient als Teil des Guards aus obiger Konstruktion.

Definition 4.4.1 ($guard_{tr,i}(p)$)
Gegeben seien eine Trace tr, der Index eines Trace-Elements $i \leq length(tr)$ und der Phasen-Event-Automat $\mathcal{P}(tr) = (P, V, A, C, E, s, I, P_0)$. Die Abbildung

$$guard_{tr,i} : P \rightarrow \mathcal{L}\,(V \cup V' \cup A \cup C)$$

ist definiert als

$$guard_{tr,i}(p) := in_{tr,i}(p) \wedge wait_{tr,i}(p) \wedge less_{tr,i}(p),$$

mit

$$in_{tr,i}(p) := \begin{cases} true, & \text{falls } tr(i) \in p.in \\ false, & \text{sonst} \end{cases}$$

$$wait_{tr,i}(p) := \begin{cases} true, & \text{falls } tr(i) \notin p.wait \\ c_i \geq tr(i).bounds, & \text{falls } tr(i) \in p.wait \wedge tr(i) \in p.gteq \\ false, & \text{sonst} \end{cases}$$

$$less_{tr,i}(p) := \begin{cases} true, & \text{falls } tr(i) \notin p.less \\ c_i < tr(i).bounds, & \text{sonst.} \end{cases}$$

□

Die Funktion *guard* spiegelt die Forderungen der *complete*-Funktion wieder. Der Zusammenhang ist, dass *complete* für einen Zustand p und eine Belegung genau dann *true* liefert, wenn die Belegung die Formel $guard(p)$ erfüllt.

Lemma 4.4.1
Gegeben sei eine Trace tr der Länge m, deren letztes Element eine Phase ist, und ein Ablauf $r = \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, \ldots, \rangle \in \mathbf{Run}(\mathcal{P}(tr))$. Es gilt:

$$complete_{tr,m}(p_k, (\gamma_k + t_k)) \Leftrightarrow (\gamma_k + t_k) \models guard_{tr,m}(p_k)$$

Beweis
Der Beweis folgt unmittelbar aus der Definition von *complete* und *guard*. □

Mit der Transitivität der Äquivalenz gilt, dass ein Präfix einer Trace genau dann erfüllt ist, wenn eine Belegung $guard(p)$ erfüllt.

Die oben skizzierte Konstruktion liefert eine kompositionelle Semantik in Form von Testautomaten für Testformeln in model-checkbarer Darstellung. Die angeführte Beschreibung basiert auf folgender Definition:

Definition 4.4.2 (Testautomatensemantik)
Sei tr eine Trace mit n Elementen, deren letzte Phase das Element $m \leq n$ sei, darauf folgen Event- oder NoEvent-Spezifikationen $\phi(\mathcal{E}_{m+1})$;...;$\phi(\mathcal{E}_n)$. Sei $tr' = Prefix_m(tr)$ die Trace, die aus tr entsteht, wenn man die Elemente $m+1, \ldots, n$ entfernt. Seien $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2 \in SVar$ boolesche Observable. Die Testautomatensemantik ist induktiv definiert:

$$\mathcal{P}(tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil) := \Big(\mathcal{P}(tr') \bullet_{\{\mathcal{S}\},\{\bigwedge_{i=m+1}^{n} \phi(\mathcal{E}_i) \wedge guard_{tr',m}\}} \mathcal{P}(\lceil true \rceil)\Big) \setminus \{\mathcal{S}\} \tag{4.30}$$

$$\mathcal{P}(\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil) := \Big(\mathcal{P}(tr') \bullet_{\{\mathcal{S}\},\{\bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i) \vee \neg guard_{tr',m}\}} \mathcal{P}(\lceil true \rceil)\Big) \setminus \{\mathcal{S}\} \tag{4.31}$$

$$\begin{aligned}&\mathcal{P}(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil) := \\ &\qquad \bigg(\Big(\mathcal{P}(\lceil true \rceil) \bullet_{\{\mathcal{S}_1\},\{true\}} \Big(\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)\Big)\Big) \setminus \{\mathcal{S}_2\}\bigg) \setminus \{\mathcal{S}_1\}\end{aligned} \tag{4.32}$$

$$\mathcal{P}(TF_1 \wedge TF_2) := \mathcal{P}(TF_1) \parallel \mathcal{P}(TF_2), \tag{4.33}$$

wobei $Trace$ eine Trace oder eine negierte Trace ist, TF_1 und TF_2 von der Form in Zeile 3.26 sind. Der Automat $\mathcal{P}(\lceil true \rceil)$ ist definiert als:

$$(\{p_{true}\}, \emptyset, \emptyset, \emptyset, \{(p_{true}, true, \emptyset, p_{true})\}, s(p_{true}) := true, I(p_{true}) := true, \{p_{true}\}, p_{true})$$

□

Der nachfolgende Satz stellt das Analogon zu den Lemmata 4.3.1 und 4.3.2 aus [Hoe05a] dar. Er reduziert die Frage der Gültigkeit einer Formel auf einem Intervall auf die Erreichbarkeit des Bad-State im Testautomaten der Formel zu einem gewissen Zeitpunkt. Der Satz ist zentral im Beweis des Model-Checking Verfahrens in Kapitel 4.5.

Theorem 4.4.1 (Charakterisierung der Erfüllbarkeit mit Testautomaten)
Gegeben seien ein Disjunktionsglied TF einer Testformel in model-checkbarer Darstellung, eine Interpretation $\mathcal{I}$ und ein Zeitpunkt $t \in \mathbb{R}_{\geq 0}$. Dann gilt:

$$\begin{aligned}&\mathcal{I}, [0,t] \models TF \\ \Leftrightarrow &\exists \text{ Fehlerablauf } r = \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(TF)) : \\ &\mathcal{I} \text{ gehört zu } r \text{ und } r \text{ erreicht den Bad-State } p_n \text{ und } \textstyle\sum_{j=0}^{n-1} t_j = t,\end{aligned}$$

wobei $\mathcal{P}(TF)$ der Testautomat aus Definition 4.4.2 ist.

Beweis
Der Beweis wird mittels struktureller Induktion über den Aufbau der Disjunktionsglieder in der model-checkbaren Darstellung von Testformeln geführt.

Induktionsanfang
Erster Fall ($tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil$):
Zeige die Aussage für $tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil$, wobei tr $n \in \mathbb{N}$ Elemente besitze. Sei die letzte Phase in tr gerade das Element $m \leq n$ und seien die Elemente $m+1, \ldots, n$ Event- und NoEvent-Spezifikationen $\phi(\mathcal{E}_{m+1})\,;\ldots;\phi(\mathcal{E}_n)$. Sei $tr' = Prefix_m(tr)$ die Trace, die entsteht, wenn man die ersten m Phasen betrachtet.

⇒: Gegeben seien eine Interpretation $\mathcal{I}$ und ein Zeitpunkt $t \in \mathbb{R}_{>0}$, so dass

$$\mathcal{I}[\![tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil]\!][0,t] = tt$$

$$\Leftrightarrow \exists t' \in]0,t[: \mathcal{I}[\![tr']\!][0,t'] = tt \text{ und } \mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j) \wedge \underset{\mathcal{S}}{\Updownarrow}]\!][t',t'] = tt \text{ und } \mathcal{I}[\![\lceil true \rceil]\!][t',t].$$

Dabei steht $\underset{\mathcal{S}}{\Updownarrow}$ abkürzend für $true \underset{\mathcal{S}}{\Updownarrow} true$. Die Äquivalenz benutzt die Aussage, dass das Trennen zweier Eventspezifikationen mittels Chop äquivalent zu der Konjunktion der Events ist: $\phi(\mathcal{E}_1)\,;\phi(\mathcal{E}_2) \Leftrightarrow \phi(\mathcal{E}_1) \wedge \phi(\mathcal{E}_2)$.

Nach Lemma 4.3.2 gibt es einen unendlichen Ablauf

$$r' := \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr')),$$

zu dem $\mathcal{I}$ gehört und für den gilt $\sum_{l=0}^{k} t_l = t'$, $k \in \mathbb{N}$. Ferner darf ohne Einschränkung angenommen werden, dass für den Zeitpunkt t gilt: $t = \sum_{l=0}^{p} t_l$, $p > k$. Weil $\mathcal{I}[\![tr']\!][0,t'] = tt$, gilt mit der Äquivalenz zwischen *complete* und der Gültigkeit des Präfixes nach Lemma 4.3.1 $complete_{tr',m}(p_k, (\gamma_k + t_k)) = tt$. Dann ist

$$\begin{aligned} r := \langle &(p_0, \beta_0, \gamma_0), t_0, Y_0, \ldots, (p_k, \beta_k, \gamma_k), t_k, Y_k \cup \{\mathcal{S}\}, \\ &(p_{true}, \beta_{k+1}, \hat{\gamma}_{k+1}), t_{k+1}, Y_{k+1}, (p_{true}, \beta_{k+2}, \hat{\gamma}_{k+2}), t_{k+2}, Y_{k+2}, \ldots \rangle, \end{aligned}$$

wobei für alle $c \in C$ gelte

$$\hat{\gamma}_j(c) := \begin{cases} 0, & \text{falls } j = k+1 \\ \hat{\gamma}_{j-1} + t_{j-1}, & \text{falls } j > k+1, \end{cases}$$

ein Ablauf in $\mathbf{Run}(\mathcal{P}(tr') \bullet_{\{\mathcal{S}\}, \{\bigwedge_{i=m+1}^{n} \phi(\mathcal{E}_i) \wedge guard_{tr',m}\}} \mathcal{P}(\lceil true \rceil))$. Der Nachweis, dass es sich um einen Ablauf des Automaten handelt, wird wie folgt geführt:
Es gilt $p_0 \in P_0$, da die Startzustände aus $\mathcal{P}(tr')$ in der sequentiellen Komposition übernommen werden. Die Uhrenbelegungen werden bis zum Zeitpunkt t' aus dem Ablauf r' in den Run r übernommen, daher gilt $\gamma_0(c) = 0$. Für die Zustände p_l gilt, weil r' ein Ablauf von $\mathcal{P}(tr')$ ist, dass $\beta_l \models s(p_l)$, $0 \leq l \leq k$. Für alle β_l, $l \geq k+1$, gilt die Forderung, da $s(p_{true})$ äquivalent zu $true$ ist. Analog lässt sich für die Uhrenbelegungen argumentieren. Für alle $0 \leq l < k$ gibt es eine Kante

(p_l, g, X, p_{l+1}), so dass $\beta_l \cup \beta'_{l+1} \cup (\gamma_l + t_l) \cup \chi_{Y_l} \models g$ und $\gamma_{l+1} = (\gamma_l + t_l)[X := 0]$. Für $l = k$ betrachte die Kante $(p_k, \mathcal{S} \wedge \bigwedge_{i=m+1}^{n} \phi(\mathcal{E}_i) \wedge guard_{tr',m}(p_k), C, p_{true})$. Nach Voraussetzung gilt $complete_{tr',m}(p_k, (\gamma_k + t_k)) = tt$. Mit Lemma 4.4.1 folgt daher $(\gamma_k + t_k) \models guard_{tr',m}(p_k)$ und mit $\chi_{\{\mathcal{S}\}} \models \mathcal{S}$ gilt $(\gamma_k + t_k) \cup \chi_{Y_k \cup \{\mathcal{S}\}} \models \mathcal{S} \wedge guard_{tr',m}(p_k)$. Weil $\mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)]\!][t', t']$ gilt, ändert $\mathcal{I}(\mathcal{E}_j)$ den Wert zum Zeitpunkt t' genau dann, wenn $\phi(\mathcal{E}_j) = \updownarrow \mathcal{E}_j$. Da Y_k genau die Events enthält, deren Interpretation zum Zeitpunkt t' ihren Wert ändert, und keine weiteren, gilt $\chi_{Y_k} \models \bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)$. Es gilt $(\gamma_k + t_k) \cup \chi_{Y_k \cup \{\mathcal{S}\}} \models \bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j) \wedge \mathcal{S} \wedge guard_{tr',m}(p_k)$. Für alle $c \in C$ gilt $\hat{\gamma}_{k+1}(c) = 0 = (\gamma_k(c) + t_k)[C := 0]$. Für alle weiteren Folgen $(p_{true}, \beta_j, \hat{\gamma}_j), t_j, Y_j$ in dem Run gilt $\beta_j \cup \beta'_{j+1} \cup (\hat{\gamma}_j + t_j) \cup \chi_{Y_j} \models \neg \mathcal{S}$ und per Definition $\hat{\gamma}_{j+1} = \hat{\gamma}_j + t_j = (\hat{\gamma}_j + t_j)[\emptyset := 0]$, $j \geq k + 1$. Also ist r ein Lauf in $\mathbf{Run}(\mathcal{P}(tr') \bullet_{\{\mathcal{S}\}, \{\bigwedge_{i=m+1}^{n} \phi(\mathcal{E}_i) \wedge guard_{tr',m}\}} \mathcal{P}(\lceil true \rceil))$.

Es bleibt zu zeigen, dass die Interpretation $\mathcal{I}$ zu dem gesuchten Ablauf gehört. Da $\mathcal{I}$ zu r' gehört und r die Eventmengen bis auf $\{\mathcal{S}\}$, die Dauern der Zustände und die Belegungen der Observablen von r' übernimmt, muss nur für das neue Event $\mathcal{S}$ nachgewiesen werden, dass 4.2 – 4.4 erfüllt sind. Da das Event nur zum Zeitpunkt $\sum_{l=0}^{k} t_l = t'$ auftritt, gilt 4.2. Zu diesem Zeitpunkt wird die Kante von p_k zu p_{true} passiert, es treten die Events $Y_k \cup \{\mathcal{S}\}$ auf. Weil dies zugleich die einzige Transition ist, an der das Event erlaubt ist, gelten 4.3 und 4.4. Damit ist die Interpretation zu r gehörig. Außerdem ist r ein Fehlerablauf. Der Bad-State wird genau zum geforderten Zeitpunkt $t = \sum_{l=0}^{k} t_l + \sum_{l=k+1}^{p} t_l$ erreicht.

$\Leftarrow$: Gegeben seien die Interpretation $\mathcal{I}$ und der Zeitpunkt $t \in \mathbb{R}_{>0}$. Sei

$$\begin{aligned} r := \langle &(p_{0,1}, \beta_{0,1}, \gamma_{0,1}), t_{0,1}, Y_{0,1}, \ldots, (p_{k,1}, \beta_{k,1}, \gamma_{k,1}), t_{k,1}, Y_{k,1} \cup \{\mathcal{S}\}, \\ &(p_{true}, \beta_{0,2}, \gamma_{0,2}), t_{0,2}, Y_{0,2}, (p_{true}, \beta_{1,2}, \gamma_{1,2}), t_{1,2}, Y_{1,2}, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil)) \end{aligned}$$

der Ablauf, so dass $\mathcal{I}$ zu r gehört und r den Bad-State p_{true} zum Zeitpunkt $t = \sum_{l=0}^{k} t_{l,1} + \sum_{l=0}^{j} t_{l,2}$ erreicht, $j \in \mathbb{N}_0$. Die Darstellung von r mit dem Sync-Event beim Übergang zum Zustand p_{true} ist mit der Definition der sequentiellen Komposition gerechtfertigt, die an jeder Kante zum Übergang vom Testautomaten TA_1 zum Testautomaten TA_2 das Event $\mathcal{S}$ fordert. Da r ein Ablauf des Automaten $\mathcal{P}(tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil)$ ist, stellt auch jede kürzere Folge einen Ablauf dieses Automaten dar. Insbesondere ist

$$r_1 := \langle (p_{0,1}, \beta_{0,1}, \gamma_{0,1}), t_{0,1}, Y_{0,1}, \ldots, (p_{k,1}, \beta_{k,1}, \gamma_{k,1}), t_{k,1} \rangle$$

ein Ablauf von $\mathcal{P}(tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil)$. Da dieser Run nach Definition von $\mathcal{P}(tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil)$ nur Zustände und Transitionen des Automaten $\mathcal{P}(tr')$ benutzt, handelt es sich bei r_1 auch um einen Ablauf von $\mathcal{P}(tr')$. Desweiteren gehört $\mathcal{I}$ zu r_1, da sich weder die V-Belegungen $\beta_{j,1}$, $0 \leq j \leq k$, noch die Eventmengen $Y_{j,1}$, $0 \leq j < k$, ändern. Weil r ein Ablauf der sequentiell komponierten Automaten ist, muss der Guard beim Passieren der Kante von $p_{k,1}$ zu p_{true} erfüllt sein, d.h. es gilt $(\gamma_k + t_k) \models$

$guard_{tr',m}(p_{k,1})$. Nach Lemma 4.4.1 gilt daher auch $complete(p_{k,1},(\gamma_{k,1}+t_{k,1})) = tt$. Mit dem Zeitpunkt $t' = \sum_{l=0}^{k} t_{l,1}$ folgt nach Lemma 4.3.1 $\mathcal{I}[\![tr']\!][0,t'] = tt$. Bleibt zu zeigen, dass $\mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j) \wedge \underset{s}{\Updownarrow}]\!][t',t'] = tt$. Betrachte die Observable $\mathcal{E}_j$, $m+1 \leq j \leq n$. Da $\mathcal{I}$ zu r gehört, ändert sich die Interpretation $\mathcal{I}(\mathcal{E}_j)$ zu dem Zeitpunkt t' genau dann, wenn dieses Event in der Eventmenge $Y_{k,1} \cup \{\mathcal{S}\}$ enthalten ist. Da aber $\chi_{Y_{k,1}\cup\{\mathcal{S}\}} \models \bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)$, ändert sich die Interpretation $\mathcal{I}(\mathcal{E}_j)$ genau dann, wenn $\phi(\mathcal{E}_j)$ gerade $\updownarrow \mathcal{E}_j$. Für $\phi(\mathcal{E}_j) = \not\updownarrow \mathcal{E}_j$ bleibt $\mathcal{I}(\mathcal{E}_j)$ zum Zeitpunkt t' konstant. Damit gilt $\mathcal{I}[\![\phi(\mathcal{E}_j)]\!][t',t']$ für alle $m+1 \leq j \leq n$. Analog argumentiert ergibt sich, dass $\mathcal{I}[\![\updownarrow \mathcal{S}]\!][t',t'] = tt$. Ferner ist in keiner anderen Eventmenge des Runs das Event $\mathcal{S}$ enthalten, da jede Kante des Automaten $\mathcal{P}(tr')$ und die Stotterkante im Automaten $\mathcal{P}(\lceil true \rceil)$ das Event $\mathcal{S}$ verbieten und es keine Kanten von $\mathcal{P}(\lceil true \rceil)$ nach $\mathcal{P}(tr')$ gibt. Da der Run r unendlich lang ist und da $\mathcal{I}$ zu r gehört, ändert $\mathcal{I}(\mathcal{S})$ nur ein einziges Mal den Wert und es gilt $\mathcal{I}[\![\underset{\mathcal{S}}{\Updownarrow}]\!][t',t'] = tt$.

Zusammen ergibt sich $\mathcal{I}[\![tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil]\!][0,t]$.

Zweiter Fall ($\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil$):

Zeige die Aussage für $\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil$, wobei die Trace tr die Elementanzahl $n \in \mathbb{N}$ habe. Sei die letzte Phase in tr gerade das Element $m \leq n$ und seien die Elemente $m+1,\ldots,n$ Event- und NoEvent-Spezifikationen $\phi(\mathcal{E}_{m+1})\,\mathbf{;}\ldots\mathbf{;}\,\phi(\mathcal{E}_n)$. Sei $tr' = Prefix_m(tr)$.

⇒: Gegeben seien eine Interpretation $\mathcal{I}$ und ein Zeitpunkt $t \in \mathbb{R}_{>0}$, so dass

$$\begin{aligned}
& \mathcal{I}[\![\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil]\!][0,t] = tt \\
\{\text{Def. } \mathbf{;}\} \Leftrightarrow & \exists t' \in]0,t[: \mathcal{I}[\![\neg tr]\!][0,t'] = tt \text{ und } \mathcal{I}[\![\underset{\mathcal{S}}{\Updownarrow}]\!][t',t'] = tt \text{ und } \mathcal{I}[\![\lceil true \rceil]\!][t',t] \\
\{\text{Def. } tr'\} \Leftrightarrow & \exists t' \in]0,t[: \left(\mathcal{I}[\![tr']\!][0,t'] = f\!f \text{ oder } \mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)]\!][t',t'] = f\!f \right) \\
& \text{und } \mathcal{I}[\![\underset{\mathcal{S}}{\Updownarrow}]\!][t',t'] = tt \text{ und } \mathcal{I}[\![\lceil true \rceil]\!][t',t]
\end{aligned}$$

Betrachte den unendlichen Ablauf nach Lemma 4.3.2

$$r' := \langle (p_0,\beta_0,\gamma_0), t_0, Y_0, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr'))$$

für den gilt, dass $\mathcal{I}$ zu r' gehört und r' die Konfiguration (p_k,β_k,γ_k) zum Zeitpunkt $t' = \sum_{l=0}^{k-1} t_l$ erreicht. Ferner gelte wiederum $\sum_{l=0}^{p} t_l = t$, $p > k$. Die Folge

$$\begin{aligned}
r := \langle & (p_0,\beta_0,\gamma_0), t_0, Y_0, \ldots, (p_k,\beta_k,\gamma_k), t_k, Y_k \cup \{\mathcal{S}\}, \\
& (p_{true},\beta_{k+1},\hat{\gamma}_{k+1}), t_{k+1}, Y_{k+1}, (p_{true},\beta_{k+2},\hat{\gamma}_{k+2}), t_{k+2}, Y_{k+2}, \ldots \rangle,
\end{aligned}$$

konstruiert wie im ersten Fall, ist ein Ablauf in $\mathbf{Run}(\mathcal{P}(\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil))$. Der Nachweis, dass es sich bei r tatsächlich um einen Ablauf der sequentiell komponierten Automaten handelt, verläuft analog zum ersten Fall, einzig für den Übergang

$$(p_k,\beta_k,\gamma_k), t_k, Y_k \cup \{\mathcal{S}\}, (p_{true},\beta_{k+1},\hat{\gamma}_{k+1})$$

muss nachgewiesen werden, dass an der Kante von p_k zu p_{true} der geänderte Guard $\mathcal{S} \wedge \big(\bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i) \vee \neg guard_{tr',m}(p_k)\big)$ erfüllt ist. Dazu werden die Fälle unterschieden, ob unter der gegebenen Interpretation und in dem Intervall $[0, t']$ die Trace tr' nicht erfüllt wurde, oder die folgenden Events $\wedge_{j=m+1}^{n} \phi(\mathcal{E}_j)$ nicht aufgetreten sind. Für beide Fälle wird nachgewiesen, dass der Guard erfüllt ist.

Fall 1: $\mathcal{I}[\![tr']\!][0, t'] = \mathit{ff}$
Nach Lemma 4.3.1 gilt $complete_{tr',m}(p_k, (\gamma_k + t_k)) = \mathit{ff}$. Dann folgt mit Lemma 4.4.1 $(\gamma_k + t_k) \models \neg guard_{tr',m}(p_k)$. Damit gilt bereits $(\gamma_k + t_k) \cup \chi_{Y_k \cup \{\mathcal{S}\}} \models \mathcal{S} \wedge \big(\bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i) \vee \neg guard_{tr',m}(p_k)\big)$.

Fall 2: $\mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)]\!][t', t'] = \mathit{ff}$
Es gibt ein $j \in \{m+1, \ldots, n\}$ für das gilt $\mathcal{I}[\![\phi(\mathcal{E}_j)]\!][t', t'] = \mathit{ff}$. Das bedeutet wiederum $\mathcal{I}[\![\neg\phi(\mathcal{E}_j)]\!][t', t'] = tt$[2]. Die Menge Y_k enthält genau die Events, für welche die Interpretation den Funktionswert zum Zeitpunkt t' ändert. Da die Interpretation die Formel $\neg\phi(\mathcal{E}_j)$ zum Zeitpunkt t' erfüllt, gilt daher auch $\chi_{Y_k} \models \neg\phi(\mathcal{E}_j)$. Damit ist bereits $(\gamma_k + t_k) \cup \chi_{Y_k \cup \{\mathcal{S}\}} \models \mathcal{S} \wedge \big(\bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i) \vee \neg guard_{tr',m}(p_k)\big)$ wahr.

Desweiteren gilt analog zum vorigen Fall, dass $\mathcal{I}$ zu r gehört. Der Bad-State p_{true} wird zum Zeitpunkt t erreicht.

$\Leftarrow$: Gegeben seien die Interpretation $\mathcal{I}$ und der Zeitpunkt $t \in \mathbb{R}_{>0}$. Sei analog zum ersten Fall

$$\begin{aligned} r := \langle &(p_{0,1}, \beta_{0,1}, \gamma_{0,1}), t_{0,1}, Y_{0,1}, \ldots, (p_{k,1}, \beta_{k,1}, \gamma_{k,1}), t_{k,1}, Y_{k,1} \cup \{\mathcal{S}\}, \\ &(p_{true}, \beta_{0,2}, \gamma_{0,2}), t_{0,2}, Y_{0,2}, (p_{true}, \beta_{1,2}, \gamma_{1,2}), t_{1,2}, Y_{1,2}, \ldots\rangle \end{aligned}$$

der Ablauf in $\mathbf{Run}(\mathcal{P}(\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil))$, so dass $\mathcal{I}$ zu r gehört und r den Bad-State p_{true} zum Zeitpunkt $t = \sum_{l=0}^{k} t_{l,1} + \sum_{l=0}^{j} t_{l,2}$ erreicht, $j \in \mathbb{N}_0$. Mit obiger Argumentation kann wiederum gefolgert werden, dass

$$r_1 := \langle (p_{0,1}, \beta_{0,1}, \gamma_{0,1}), t_{0,1}, Y_{0,1}, \ldots, (p_{k,1}, \beta_{k,1}, \gamma_{k,1}), t_{k,1}\rangle$$

ein Ablauf von $\mathcal{P}(tr')$ ist, zu dem $\mathcal{I}$ gehört. Mit der Definition der sequentiellen Komposition gilt nunmehr, dass $(\gamma_{k,1} + t_{k,1}) \cup \chi_{Y_{k,1} \cup \{\mathcal{S}\}} \models \bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i) \vee \neg guard_{tr',m}(p_{k,1})$. Es werden die Fälle $\chi_{Y_{k,1} \cup \{\mathcal{S}\}} \models \bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i)$, und $\gamma_{k,1} + t_{k,1} \models \neg guard_{tr',m}(p_{k,1})$ unterschieden. Für beide Fälle wird gefolgert, dass $(\mathcal{I}[\![tr']\!][0, t'] = \mathit{ff}$ oder $\mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)]\!][t', t'] = \mathit{ff})$.

Fall 1: $\gamma_{k,1} + t_{k,1} \models \neg guard_{tr',m}(p_{k,1})$
Nach Lemma 4.4.1 gilt $complete(p_{k,1}, (\gamma_{k,1} + t_{k,1})) = \mathit{ff}$. Mit dem Zeitpunkt $t' = \sum_{j=0}^{k} t_{j,1}$ gilt nach Lemma 4.3.1 $\mathcal{I}[\![tr']\!][0, t'] = \mathit{ff}$. Damit gilt insbesondere $(\mathcal{I}[\![tr']\!][0, t'] = \mathit{ff}$ oder $\mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)]\!][t', t'] = \mathit{ff})$.

[2] Die Schreibweise ist an dieser Stelle unproblematisch, denn $\mathcal{I}[\![\neg \updownarrow \mathcal{E}]\!][t', t'] \Leftrightarrow \mathcal{I}[\![\not\updownarrow \mathcal{E}]\!][t', t']$ auf einem Punktintervall.

Fall 2: $\chi_{Y_{k,1}\cup\{\mathcal{S}\}} \models \bigvee_{i=m+1}^{n} \neg\phi(\mathcal{E}_i)$
Es gibt einen Index $j \in \{m+1,\ldots,n\}$ für den gilt $\chi_{Y_{k,1}\cup\{\mathcal{S}\}} \models \neg\phi(\mathcal{E}_j)$. Da die Interpretation zu dem Ablauf gehörig ist, ist die Funktion $\mathcal{I}(\mathcal{E}_j)$ unstetig zum Zeitpunkt t' genau dann, wenn $\mathcal{E}_j \in Y_{k,1} \cup \{\mathcal{S}\}$. Weil $\chi_{Y_{k,1}\cup\{\mathcal{S}\}} \models \neg\phi(\mathcal{E}_j)$, gilt so auch $\mathcal{I}[\![\neg\phi\mathcal{E}_j]\!][t',t'] = tt$. Daraus folgt sofort $(\mathcal{I}[\![tr']\!][0,t'] = ff$ oder $\mathcal{I}[\![\bigwedge_{j=m+1}^{n} \phi(\mathcal{E}_j)]\!][t',t'] = ff)$.

Die Argumentation aus dem ersten Fall kann genutzt werden, um zu zeigen, dass $\mathcal{I}[\![\underset{\mathcal{S}}{\Updownarrow}]\!][t',t'] = tt$. Insgesamt gilt $\mathcal{I}[\![\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil]\!][0,t]$.

Induktionsschluss
Erster Fall ($\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil$):
Angenommen die Aussage gilt für $Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil$, wobei $Trace$ eine Trace oder eine negierte Trace ist. Seien eine Interpretation $\mathcal{I}$ und ein Zeitpunkt $t \in \mathbb{R}_{>0}$ gegeben.

⇒: Gelte

$$\begin{aligned}
&\mathcal{I}[\![\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil]\!][0,t] \\
\Leftrightarrow &\exists t' \in]0,t[: \mathcal{I}[\![\lceil true \rceil]\!][0,t'] \text{ und } \mathcal{I}[\![\underset{\mathcal{S}_1}{\Updownarrow}]\!][t',t'] \text{ und } \mathcal{I}[\![Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil]\!][t',t] \\
\Leftrightarrow &\exists t' \in]0,t[: \mathcal{I}[\![\lceil true \rceil]\!][0,t'] \text{ und } \mathcal{I}[\![\underset{\mathcal{S}_1}{\Updownarrow}]\!][t',t'] \text{ und } \mathcal{I}_{t'}[\![Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil]\!][0,t-t']
\end{aligned}$$

Die erste Äquivalenz benutzt die Definition des Chop Operators, die zweite gilt wegen Lemma 2.1.1. Nach Induktionsvoraussetzung gibt es einen Ablauf

$$\begin{aligned}
r_2 := \langle &(p_{0,2}, \beta_{0,2}, \gamma_{0,2}), t_{0,2}, Y_{0,2}, \ldots, (p_{k,2}, \beta_{k,2}, \gamma_{k,2}), t_{k,2}, Y_{k,2} \cup \{\mathcal{S}_2\}, \\
&(p_{true_2}, \beta_{0,3}, \gamma_{0,3}), t_{0,3}, Y_{0,3} \ldots \rangle \in \mathbf{Run}(\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)),
\end{aligned}$$

der den Bad-State p_{true_2} zum Zeitpunkt $\sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3} = t - t'$ erreicht, $j \in \mathbb{N}_0$, und für den gilt, dass $\mathcal{I}_{t'}$ zu r_2 gehört. Betrachte den Ablauf

$$r := \langle (p_{true_1}, \beta_{0,1}, \gamma_{0,1}), t', \{\mathcal{S}_1\} \cup Y(t'), (p_{0,2}, \beta_{0,2}, \gamma_{0,2}), t_{0,2}, Y_{0,2}, \ldots \rangle$$

in $\mathbf{Run}(\mathcal{P}(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil))$, wobei $Y(t')$ die Menge der Events sei, deren Interpretation im Zeitpunkt t' unstetig ist. Für $X \in V$, $c \in C$ seien definiert:

$$\begin{aligned}
\beta_{0,1}(X) &:= \text{beliebig} \\
\gamma_{0,1}(c) &:= 0
\end{aligned}$$

Es handelt sich tatsächlich um einen Ablauf der sequentiell komponierten Automaten, denn $p_{true_1} \in \{p_{true_1}\} = P_0$. Für alle Uhren gilt $\gamma_{0,1}(c) = 0$. Es gilt

$\beta_{0,1} \models s(p_{true_1})$, da $s(p_{true_1})$ äquivalent zu $true$ ist. Die übrigen V-Belegungen erfüllen die Zustandsinvarianten, $\beta_{j,2} \models s(p_{j,2})$, $0 \leq j \leq k$, und $\beta_{j,3} \models s(p_{j,3})$, $0 \leq j$, denn r_2 ist ein Ablauf von $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)$. Analog wird für die Uhrenbelegungen argumentiert. An den Kanten, die aus $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)$ übernommen wurden, ist zusätzlich das Event $\mathcal{S}_1$ verboten. Da das Event $\mathcal{S}_1$ jedoch nicht in der Eventmenge des Automaten $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)$ enthalten ist, kann keine der Mengen $Y_{j,2}$, $0 \leq j \leq k$, und $Y_{j,3}$, $0 \leq j$, den Guard $\neg\mathcal{S}_2$ verletzen. Damit ist die Forderung aus Zeile 2.50 in Definition 2.2.3 auch für die geänderten Guards an den Kanten aus $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)$ erfüllt. Für den Übergang von p_{true_1} zu $p_{0,2}$ ist zu zeigen, dass der Guard $\mathcal{S}_1 \wedge \neg\mathcal{S}_2$ der Kante $(p_{true_1}, \mathcal{S}_1 \wedge \neg\mathcal{S}_2, C, p_{0,2})$ erfüllt ist. Dies gilt, da die Menge $Y(t') \cup \{\mathcal{S}_1\}$ das Event $\mathcal{S}_1$ enthält. Ferner kann die Menge $Y(t')$ nicht $\mathcal{S}_2$ enthalten, da $Y(t')$ alle Events enthält, die zum Zeitpunkt t' auftreten. Das Event $\mathcal{S}_2$ tritt nur einmalig beim Übergang zum Bad-State p_{true_2} auf, nicht zum Zeitpunkt t'. So gilt $\chi_{Y(t') \cup \{\mathcal{S}_1\}} \models \mathcal{S}_1 \wedge \neg\mathcal{S}_2$. Mit $\gamma_{0,2}(c) = 0 = (\gamma_{0,1}(c) + t')[C := 0]$ ist r ein Ablauf der sequentiellen Komposition.
Konstruiere einen Ablauf r', indem die Phase $(p_{0,2}, \beta_{0,2}, \gamma_{0,2})$ an jeder Unstetigkeitsstelle der Interpretation $\mathcal{I}$ im Intervall $[0, t']$ aufgespalten wird. Anschließend definiere einen Ablauf r'', wobei die Belegungen $\beta'_{j,1}$ und Eventmengen $Y_{j,1}$ von der Interpretation bestimmt werden. Es bleibt zu zeigen, dass die Interpretation zu r'' gehörig ist. Per Definition sind die Belegungen und Events in dem Intervall $[0, t']$ korrekt. Der Ablauf r_2 entspricht der Teilfolge des Runs r'', die zum Zeitpunkt t' beginnt. Es gilt per Definition $\mathcal{I}(X)(t + t') = \mathcal{I}_{t'}(X)(t)$, $t \in \mathbb{T}ime$. Weil $\mathcal{I}_{t'}$ zu r_2 gehört, stimmen die Belegungen und Eventmengen des Runs r'' auch ab dem Zeitpunkt t' mit der Interpretation $\mathcal{I}$ überein. Damit ist $\mathcal{I}$ zu r'' gehörig. Der Bad-State p_{true_2} wird zum Zeitpunkt $t' + \sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3} = t' + t - t' = t$ erreicht.

$\Leftarrow$: Gegeben seien die Interpretation $\mathcal{I}$ und der Zeitpunkt $t \in \mathbb{R}_{>0}$. Sei

$$\begin{aligned} r := \langle & (p_{true_1}, \beta_{0,1}, \gamma_{0,1}), t_{0,1}, Y_{0,1}, \ldots, (p_{true_1}, \beta_{n,1}, \gamma_{n,1}), t_{n,1}, Y_{n,1} \cup \{\mathcal{S}_1\}, \\ & (p_{0,2}, \beta_{0,2}, \gamma_{0,2}), t_{0,2}, Y_{0,2}, \ldots, (p_{k,2}, \beta_{k,2}, \gamma_{k,2}), t_{k,2}, Y_{k,2} \cup \{\mathcal{S}_2\}, \\ & (p_{true_2}, \beta_{0,3}, \gamma_{0,3}), t_{0,3}, Y_{0,3}, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)) \end{aligned}$$

ein Ablauf, so dass $\mathcal{I}$ zu r gehört und r den Bad-State p_{true_2} von $\mathcal{P}(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)$ zum Zeitpunkt $t = \sum_{l=0}^{n} t_{l,1} + \sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3}$ erreicht, $j \in \mathbb{N}_0$.
Keine der Event-Mengen außer $Y_{n,1} \cup \{\mathcal{S}_1\}$ enthält $\mathcal{S}_1$, da der Guard einer jeden Kante außer der Transition von $p_{true,1}$ zu $p_{0,2}$ das Event $\mathcal{S}_1$ verbietet, d.h. $\neg\mathcal{S}_1$ enthält. Da $\mathcal{I}$ zu r gehört, ändert $\mathcal{I}$ die Interpretation von $\mathcal{S}_1$ nur dann, wenn dies von einer der Eventmengen vorgesehen wird, d.h. $\mathcal{I}(\mathcal{S}_1)$ ist nur im Punkt $\sum_{j=0}^{n} t_{j,1}$ unstetig. Also gilt $\mathcal{I}[\![\underset{\mathcal{S}_1}{\Updownarrow}]\!][\sum_{j=0}^{n} t_{j,1}, \sum_{j=0}^{n} t_{j,1}] = tt$.

Der Testautomat $\mathcal{P}(\lceil true\rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil)$ entspricht bis auf den Zustand p_{true_1} und bis auf die von dort ausgehenden Transitionen dem Testautomaten $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil)$[3]. Daher ergibt die Teilfolge

$$r_2 := \langle (p_{0,2}, \beta_{0,2}, \gamma_{0,2}), t_{0,2}, Y_{0,2}, \ldots \rangle$$

des Ablaufs r einen Ablauf in $\mathbf{Run}(\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil))$, der den Bad-State p_{true_2} von $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil)$ zum Zeitpunkt $\sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3}$ erreicht. Weil der Ablauf r_2 der Teilfolge des Runs r entspricht, die zum Zeitpunkt $\sum_{l=0}^{n} t_{l,1}$ beginnt, und weil $\mathcal{I}$ zu r gehört, gehört $\mathcal{I}_{\sum_{l=0}^{n} t_{l,1}}$ zu r_2. Da der Satz per Induktionsannahme für den Automaten $\mathcal{P}(Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil)$ gilt, folgt:

$$\begin{aligned} & \mathcal{I}_{\sum_{j=0}^{n} t_{j,1}} [\![Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil]\!] [0, \sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3}] = tt \\ \{\text{Lemma 2.1.1}\} \Leftrightarrow & \mathcal{I} [\![Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil]\!] [\sum_{l=0}^{n} t_{l,1}, \sum_{l=0}^{n} t_{l,1} + \sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3}] = tt \end{aligned}$$

Zusammen mit obiger Argumentation über das erste Sync-Event gilt $\mathcal{I}[\![\lceil true\rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil]\!][0, \sum_{l=0}^{n} t_{l,1} + \sum_{l=0}^{k} t_{l,2} + \sum_{l=0}^{j} t_{l,3}] = tt$.

Zweiter Fall ($TF_1 \wedge TF_2$):
Angenommen die Aussage gilt für TF_1 und für TF_2, dann gilt für $TF_1 \wedge TF_2$:

$\Rightarrow$: Sei t der Zeitpunkt, so dass

$$\mathcal{I}[\![TF_1 \wedge TF_2]\!][0,t] = tt \Leftrightarrow \mathcal{I}[\![TF_1]\!][0,t] = tt \text{ und } \mathcal{I}[\![TF_2]\!][0,t] = tt$$

Per Induktionsannahme gibt es Automaten $\mathcal{P}(TF_1)$ und $\mathcal{P}(TF_2)$ und Fehlerabläufe

$$r_l := \langle (p_{0,l}, \beta_{0,l}, \gamma_{0,l}), t_{0,l}, Y_{0,l}, (p_{1,l}, \beta_{1,l}, \gamma_{1,l}), t_{1,l}, Y_{1,l}, \ldots \rangle \in \mathbf{Run}(\mathcal{P}(TF_l)),$$

$l \in \{1,2\}$ mit Konfigurationen $(p_{Bad,1}, \beta_{i,1}, \gamma_{i,1})$ bzw. $(p_{Bad,2}, \beta_{j,2}, \gamma_{j,2})$, $i, j \in \mathbb{N}_{>0}$, so dass $\mathcal{I}$ zu r_1 bzw. $\mathcal{I}$ zu r_2 gehört und $\sum_{k=0}^{i-1} t_{k,1} = t = \sum_{k=0}^{j-1} t_{k,2}$.
Zeige: Es gibt einen Fehlerablauf $r \in \mathbf{Run}(\mathcal{P}(TF_1) \parallel \mathcal{P}(TF_2))$, so dass $\mathcal{I}$ zu r gehört und r den Bad-State zum Zeitpunkt t erreicht. Sei r_1' der Fehlerablauf, der aus r_1 hervorgeht, indem Stotterkanten genau zu den Zeitpunkten benutzt werden, wenn r_2 aber nicht r_1 eine Transition nimmt. Analog sei r_2' definiert. Die Abläufe haben die Gestalt

$$r_l' = \langle \left(p_{0,l}', \beta_{0,l}', \gamma_{0,l}'\right), t_0, Y_{0,l}', \left(p_{1,l}', \beta_{1,l}', \gamma_{1,l}'\right), t_1, Y_{1,l}', \ldots \rangle,$$

[3]Einzig die verbotenen Sync-Events $\mathcal{S}_1$ an den Transitionen von $\mathcal{P}(\lceil true\rceil \underset{\mathcal{S}_1}{\Updownarrow} Trace \underset{\mathcal{S}_2}{\Updownarrow} \lceil true\rceil)$ unterscheiden die Automaten.

$l \in \{1,2\}$, d.h. die Transitionen werden synchron passiert. Ferner handelt es sich bei beiden Abläufen um Fehlerabläufe der entsprechenden Automaten. Die Interpretation gehört zu beiden Abläufen. Sei $V_s := V_1 \cap V_2$ die Menge der gemeinsamen Variablen der Automaten. Für $X \in V_s$ gilt: $\beta'_{i,1}(X) = \mathcal{I}(X)(t) = \beta'_{i,2}(X)$, $i \in \mathbb{N}_0$, $t \in \left]\sum_{k=0}^{i-1} t_k, \sum_{k=0}^{i} t_k\right[$. Es sei $A_s := A_1 \cap A_2$ die Menge der gemeinsamen Events. Für $\mathcal{E} \in A_s$ gilt analog $\mathcal{E} \in Y'_{i,1} \Leftrightarrow \mathcal{E} \in Y'_{i,2}$. Lemma 2.2.2 ist anwendbar und damit gibt es einen Ablauf $r \in \mathbf{Run}(\mathcal{P}(TF_1) \parallel \mathcal{P}(TF_2))$. Ferner erreicht r nach der Bemerkung zum Lemma zum Zeitpunkt t den Zustand $(p_{Bad,1}, p_{Bad,2})$. Damit ist r ein Fehlerablauf.

Bleibt zu zeigen, dass $\mathcal{I}$ zu r gehörig ist. Sei $X \in V_1$, dann gilt $\mathcal{I}(X)(t) = \beta'_{i,1}(X)$ für alle $t \in \left]\sum_{k=0}^{i-1} t_k, \sum_{k=0}^{i} t_k\right[$, $i \in \mathbb{N}_0$, weil $\mathcal{I}$ zu r_1 gehört. Analog wird für $X \in V_2$ argumentiert. Damit gilt 4.1 in Definition 4.1.1. Weil $\mathcal{I}$ zu r_1 gehört, sind die Werte von $\mathcal{I}(\mathcal{E})$, $\mathcal{E} \in A_1$, innerhalb der Intervalle $\left]\sum_{k=0}^{i-1} t_{k,1}, \sum_{k=0}^{i} t_{k,1}\right[$, $i \in \mathbb{N}_0$, konstant. Bei der Definition von r'_1 werden diese Intervalle weiter aufgespalten, so ist $\mathcal{I}(\mathcal{E})$ insbesondere auf den kleineren Intervallen von r'_1 konstant. Da r die Intervalle von r'_1 übernimmt, gilt 4.2 für $\mathcal{E} \in A_1$. Mit analoger Argumentation für $\mathcal{E} \in A_2$ ist 4.2 aus Definition 4.1.1 erfüllt. Der Ablauf r übernimmt als Eventmengen die Vereinigungen der Mengen $Y'_{i,1}$, $Y'_{i,2}$, $i \in \mathbb{N}_0$. Da $\mathcal{I}$ zu r'_1 und r'_2 gehört, enthält die Menge $Y'_{i,1} \cup Y'_{i,2}$ genau die Events, deren Interpretation sich im Zeitpunkt $\sum_{k=0}^{i} t_k$ ändert. Damit gelten auch 4.3 und 4.4 und $\mathcal{I}$ ist zu r gehörig.

$\Leftarrow$: Gegeben seien eine Interpretation $\mathcal{I}$, ein Zeitpunkt $t \in \mathbb{R}_{>0}$ und ein Fehlerablauf

$$r := \langle((p_{0,1}, p_{0,2}), \beta_0, \gamma_0), t_0, Y_0, ((p_{1,1}, p_{1,2}), \beta_1, \gamma_1), t_1, Y_1, \ldots\rangle$$

in $\mathbf{Run}(\mathcal{P}(TF_1) \parallel \mathcal{P}(TF_2))$ mit Bad-State $(p_{Bad,1}, p_{Bad,2})$, erreicht zum Zeitpunkt $t = \sum_{k=0}^{i} t_k$, $i \in \mathbb{N}_{>0}$. Dann sind nach Lemma 2.2.3

$$r_l := \langle(p_{0,l}, \beta_0|_{V_l}, \gamma_0|_{C_l}), t_0, Y_0 \cap A_l, (p_{1,l}, \beta_1|_{V_l}, \gamma_1|_{C_l}), t_1, Y_1 \cap A_l, \ldots\rangle$$

Fehlerabläufe von $\mathcal{P}(TF_l)$, $l \in \{1,2\}$, mit Bad-States $p_{Bad,l}$ zu den Zeitpunkten $\sum_{k=0}^{i} t_k$. In r_1 bzw. r_2 werden die Belegungen aus r übernommen und auf Variablen aus V_1 bzw. V_2 eingeschränkt. Die Eventmengen werden ebenfalls auf die Events der entsprechenden Automaten eingeschränkt. Ferner sind in r_1 bzw. r_2 die Dauern aus r übernommen. Weil $\mathcal{I}$ zu r gehört, folgt so, dass $\mathcal{I}$ auch zu r_1 und r_2 gehört. Per Induktionsvoraussetzung gilt nun

$$\begin{aligned} &\mathcal{I}[\![TF_1]\!]([0, \sum_{k=0}^{i} t_k]) = tt \text{ und } \mathcal{I}[\![TF_2]\!]([0, \sum_{k=0}^{i} t_k]) = tt \\ \Leftrightarrow &\mathcal{I}[\![TF_1 \wedge TF_2]\!]([0, \sum_{k=0}^{i} t_k]) = tt \end{aligned}$$

□

4.5 Model-Checking von Testformeln mit Testautomaten

Eine Testformel TF spezifiziert falsches Verhalten. Es wird geprüft, ob ein gegebener Phasen-Event-Automat $Ph = (P, V, A, C, E, s, I, P_0)$ die negierte Testformel, $\neg TF$, erfüllt, also das falsche Verhalten vermeidet. Der Automat ist fehlerhaft, wenn dies nicht der Fall ist.

$$
\begin{aligned}
& \neg\,(Ph \models_0 \neg TF) && (4.34)\\
\{\text{Def. } 4.1.2\} \Leftrightarrow & \neg\,(\forall \mathcal{I} : \forall r \in \mathbf{Run}(Ph) : (\mathcal{I} \text{ gehörig zu } r \Rightarrow \mathcal{I} \models_0 \neg TF)) && (4.35)\\
\{\text{Def. } \models_0\} \Leftrightarrow & \neg\,(\forall \mathcal{I} : \forall r \in \mathbf{Run}(Ph) : \\
& \qquad (\mathcal{I} \text{ gehörig zu } r \Rightarrow \forall t \in \mathbb{R}_{\geq 0} : \mathcal{I}[\![\neg TF]\!][0,t] = tt)) && (4.36)\\
\{\text{Def. } \neg\} \Leftrightarrow & \exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \\
& \qquad \mathcal{I} \text{ gehörig zu } r \wedge \mathcal{I}[\![TF]\!][0,t] = tt && (4.37)
\end{aligned}
$$

Die Äquivalenzkette zeigt, dass eine Interpretation und ein Intervall als Gegenbeispiel gefunden werden müssen, in denen das fehlerhafte Verhalten möglich ist.
In Kapitel 3 Korollar 3.3.2 ist gezeigt worden, dass eine Testformel TF durch eine Formel in model-checkbarer Darstellung ausgedrückt werden kann. Diese Darstellung erlaubt eine einfache Konstruktion von Automaten aus gegebenen Testformeln. Sei die model-checkbare Darstellung der Testformel TF durch die Formel $\exists Obs : TF'$, TF' aus Korollar 3.3.2, gegeben. Es wird über eine feste Anzahl von Observablen quantifiziert, wobei ohne Einschränkung angenommen werden darf, dass $Obs \cap A = \emptyset$[4]. Die obige Äquivalenzkette wird fortgesetzt:

$$
\begin{aligned}
& \exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \mathcal{I} \text{ gehörig zu } r \wedge \mathcal{I}[\![TF]\!][0,t] = tt && (4.38)\\
\Leftrightarrow & \exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I} \text{ gehörig zu } r \wedge \mathcal{I}[\![\exists Obs : TF']\!][0,t'] = tt && (4.39)\\
\Leftrightarrow & \exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists \mathcal{I}' =_{\setminus\{Obs\}} \mathcal{I} : \exists t' \in \mathbb{R}_{\geq 0} : \\
& \qquad \mathcal{I} \text{ gehörig zu } r \wedge \mathcal{I}'[\![TF']\!][0,t'] = tt && (4.40)\\
\Leftrightarrow & \exists \mathcal{I}'' : \exists r \in \mathbf{Run}(Ph) : \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}'' \text{ gehörig zu } r \wedge \mathcal{I}''[\![TF']\!][0,t'] = tt && (4.41)
\end{aligned}
$$

Die Übergang von Zeile 4.38 zu Zeile 4.39 ist in Korollar 3.3.2 nachgewiesen. Die Quantifizierung über Observablen in Zeile 4.39 ist mit Zeile 4.40 aufgelöst. Weil $Obs \cap A = \emptyset$, ist $\mathcal{I}'' := \mathcal{I}'$ zu einem Run von Ph gehörig. Aus diesem Grund gilt die Richtung $\Rightarrow$ von Zeile 4.40 zu Zeile 4.41. Die Richtung $\Leftarrow$ gilt mit der Interpretation der Voraussetzung $\mathcal{I}''$, eingesetzt sowohl für $\mathcal{I}$ als auch für $\mathcal{I}'$.
Um zu prüfen, ob ein gegebener Automat eine negierte Testformel, $\neg TF$, verletzt, müssen eine Interpretation und ein Intervall gefunden werden, so dass die unquantifizierte model-checkbare Darstellung der Testformel $TF' := \bigvee_i \left(\bigwedge_j \mathcal{T}_{ij}\right)$, $\mathcal{T}_{ij}$ aus Korollar 3.3.2, unter der Interpretation in dem Intervall gilt. Um das Verfahren zu automatisieren, werden zu der Formel TF' Automaten $\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ konstruiert. Die Frage, ob eine

[4] Dies kann durch gebundene Umbenennung erreicht werden.

Interpretation $\mathcal{I}$ auf einem Intervall $[0,t']$ die Formel $\bigwedge_j \mathcal{T}_{ij}$ erfüllt, ist in Satz 4.4.1 als äquivalent zu der Frage nachgewiesen worden: Gibt es einen Lauf $r \in \mathbf{Run}(\mathcal{P}(\bigwedge_j \mathcal{T}_{ij}))$, so dass die Interpretation $\mathcal{I}$ zu r gehört und r den Bad-State des Automaten zum Zeitpunkt t' erreicht. Mit dieser Äquivalenz ist folgendes Model-Checking Verfahren möglich:

$$
\begin{aligned}
&\neg\,(Ph \models_0 \neg TF) && (4.42)\\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \mathcal{I}[\![TF']\!][0,t] = tt \wedge \mathcal{I} \text{ gehörig zu } r && (4.43)\\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \mathcal{I}[\![\bigvee_i \left(\bigwedge_j \mathcal{T}_{ij}\right)]\!][0,t] = tt \wedge \mathcal{I} \text{ gehörig zu } r && (4.44)\\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \bigvee_i \left(\mathcal{I}[\![\bigwedge_j \mathcal{T}_{ij}]\!][0,t] = tt\right) \wedge \mathcal{I} \text{ gehörig zu } r && (4.45)\\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \exists i :\\
&\qquad \mathcal{I}[\![\bigwedge_j \mathcal{T}_{ij}]\!][0,t] = tt \wedge \mathcal{I} \text{ gehörig zu } r && (4.46)\\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists t \in \mathbb{R}_{\geq 0} : \exists i : \exists r' \in \mathbf{Run}(\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})) :\\
&\qquad r' \text{ erreicht den Bad-State } p_{Bad} = p_k \wedge \sum_{i=0}^{k-1} t_i = t \wedge\\
&\qquad \mathcal{I} \text{ gehörig zu } r' \wedge \mathcal{I} \text{ gehörig zu } r && (4.47)\\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(Ph) : \exists i : \exists r' \in \mathbf{Run}(\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})) :\\
&\qquad r' \text{ erreicht den Bad-State } p_{Bad} \wedge \mathcal{I} \text{ gehörig zu } r' \wedge \mathcal{I} \text{ gehörig zu } r && (4.48)\\
\Leftrightarrow &\exists i : \exists \mathcal{I} : \exists r'' \in \mathbf{Run}(Ph \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})) :\\
&\qquad \mathcal{I} \text{ gehörig zu } r'' \wedge r'' \text{ erreicht einen Zustand } (p, p_{Bad}) && (4.49)\\
\Leftrightarrow &\exists i : \exists r'' \in \mathbf{Run}(Ph \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})) : r'' \text{ erreicht einen Zustand } (p, p_{Bad}) && (4.50)
\end{aligned}
$$

Die erste Äquivalenz gilt aufgrund der vorangegangenen Äquivalenzkette 4.34 – 4.41, in der folgenden Äquivalenz 4.44 ist die model-checkbare Darstellung von TF' als disjunktive Normalform über Traces eingesetzt. Zeile 4.45 und 4.46 nutzen die Definition der Disjunktion. Die folgende Äquivalenz 4.47 ist durch die Charakterisierung der Erfüllbarkeit über Testautomaten gerechtfertigt, Satz 4.4.1. In der Äquivalenz 4.48 wird die Forderung nach einem Zeitpunkt weggelassen. Wenn es einen Zustand $p_k = p_{Bad}$ gibt, so gibt es auch eine Konfiguration (p_k, β_k, γ_k), die Summe $\sum_{i=0}^{k-1} t_k$ ergibt dann den geeigneten Zeitpunkt $t \in \mathbb{R}_{\geq 0}$. Damit gilt die Richtung $\Leftarrow$. Die Richtung $\Rightarrow$ ist per Definition wahr. Die Äquivalenz von 4.48 und 4.49 ist im Beweis von Satz 4.4.1 nachgewiesen worden[5]: Für beide Abläufe gibt es nach Satz 2.2.2 einen Ablauf der parallelen Komposition

[5] Im zweiten Fall des Induktionsschlusses, Richtung $\Rightarrow$.

zu dem die Interpretation gehört. Dabei sei p ein Zustand von Ph. Die letzte Äquivalenz ist wegen des Lemmas 4.1.1 wahr.

Theorem 4.5.1 (Model-Checking Theorem)
Gegeben sei eine Testformel ohne Liveness TF, welche die model-checkbaren Darstellung $\bigvee_i \left(\bigwedge_j \mathcal{T}_{ij}\right)$ besitzt Ferner sei Ph ein Phasen-Event-Automat. Die Erfüllbarkeit der negierten Testformel durch den Phasen-Event-Automaten lässt sich mit folgendem Model-Checking Verfahren entscheiden:

$$\begin{aligned} &\neg\,(Ph \models_0 \neg TF) \\ \Leftrightarrow &\exists i : \exists r'' \in \mathbf{Run}(Ph \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})) : r'' \text{ erreicht einen Zustand } (p, p_{Bad}), \end{aligned}$$

wobei p ein Zustand von Ph und p_{Bad} der Bad-State von $\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ seien.

Beweis
Der Satz ist mit Herleitung 4.34 – 4.50 gültig. □

Bemerkung 4.5.1
Das Verfahren verlangt die Erreichbarkeit des Bad-State im Testautomaten für *ein* Disjunktionsglied. Das Model-Checking kann daher für jedes Disjunktionsglied einzeln durchgeführt werden und wird beendet, sobald in einem Disjunktionsglied der Bad-State erreichbar war. Das Parallelprodukt $Ph \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ muss nur für die tatsächlich überprüften Disjunktionsglieder aufgebaut werden. Stellt sich in einem Disjunktionsglied i der Bad-State als erreichbar heraus, so müssen die Parallelkompositionen $Ph \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{kj})$, $k \in \{i+1, \ldots\}$ nicht ausgerechnet werden. Die Art, wie gezielt Rechenzeit gespart wird, kann mit dem Vorgehen beim On-The-Fly Model-Checking verglichen werden.

5 Entwicklung eines Testformel-Compilers

In Kapitel 4 wird gezeigt, wie Testformeln in Phasen-Event-Automaten mit Endzuständen, sogenannte Testautomaten, überführt werden können. Mit diesen Testautomaten ist es möglich, das Erfüllbarkeitsproblem von Formeln bezüglich Phasen-Event-Automaten auf ein Erreichbarkeitsproblem der Parallelkomposition des Automaten und des Testautomaten zu reduzieren. Dazu wird jede Testformel in eine model-checkbare Darstellung überführt, die als disjunktive Normalform über Traces interpretiert werden kann. Zu jedem Disjunktionsglied wird die parallele Komposition der Automaten berechnet, die die Semantik der entsprechenden konjugierten Traces bilden. Das Model-Checking Verfahren prüft für jedes einzelne Disjunktionsglied, ob der Endzustand aller parallel-komponierter Testautomaten erreicht wird. Ist dies der Fall, so ist das Model-Checking beendet, ansonsten wird das nächste Disjunktionsglied überprüft.
Im Folgenden soll ein Tool entwickelt werden, das die Konstruktion der Phasen-Event-Automaten[1] aus einer gegebenen Testformel automatisiert. Nach einer Analyse der Umgebung, innerhalb derer das Werkzeug entwickelt und genutzt wird, werden die Systemanforderungen ermittelt. Anschließend wird in der Entwurfsphase eine Architektur erarbeitet, die es ermöglicht, die gestellten Anforderungen zu erfüllen. Diese Architektur wird in der Implementierung in ein lauffähiges Java-Programm überführt, das abschließend zu testen ist.

5.1 Anforderungsanalyse

Nach einer Analyse der Rahmenbedingungen zur Entwicklung und Nutzung des Werkzeugs wird konkret auf die funktionalen und nichtfunktionalen Anforderungen des zu entwerfenden Systems eingegangen.

5.1.1 Analyse der Rahmenbedingungen

Im Rahmen der Erforschung von Phasen-Event-Automaten ist das Java-Pake `pea` entwickelt worden. Kern des Paketes ist ein Countertrace-Compiler, der Countertraces, eine Teilklasse aller DC Formeln, in Phasen-Event-Automaten übersetzt. Die Automaten

[1] genauer: der Phasen-Event-Automaten Netze zu jedem Disjunktionsglied

dienen, anders als Testautomaten, der Spezifikation und nicht der Überprüfung eines Systems. Das Paket `pea` bietet eine Repräsentation von Phasen-Event-Automaten sowie Formeln in Java. Ferner ist der Algorithmus zur Durchführung der Potenzmengenkonstruktion fast vollständig im Countertrace-Compiler implementiert. Das zu entwickelnde Tool wird als Unterpaket `pea.modelchecking` in das Paket `pea` integriert. Dadurch ist es möglich, die bereits implementierte und in mehreren Fallstudien (vgl. z.B. [Hoe05b]) getestete Funktionalität des Paketes `pea` im Testautomaten-Compiler zu nutzen.
Im Rahmen des AVACS Projektes [AVA] ist, basierend auf der *Moby* Klassenbibliothek, das Werkzeug *Moby/PEA* entstanden. *Moby/PEA* erlaubt eine graphische Bearbeitung von Phasen-Event-Automaten und die anschließende Überführung in ein XML-Format. Der Testformel-Compiler des Paketes `pea.modelchecking` soll in das Tool *Moby/PEA* eingebunden werden können, der Kompilationsvorgang wird dann von *Moby/PEA* aus angestoßen. Der Compiler übersetzt eine gegebene Testformel und erzeugt geeignet viele Netze von Phasen-Event-Automaten[2] in XML-Darstellung, die dann in *Moby/PEA* geladen werden können.
Das Model-Checking wird mit dem Model-Checker *ARMC* durchgeführt. *ARMC* ist ein Reachability Model-Checker für Transition-Constraint-Systems. Um Erreichbarkeit in Phasen-Event-Automaten mittels *ARMC* prüfen zu können, ist im Rahmen des *AVACS* Projektes eine Übersetzung von PEAs in Transition-Constraint-Systems definiert worden, die es erlaubt, Eigenschaften der PEAs über Transition-Constraint-Systems zu checken. Die Ergebnisse sind in [Hoe05b] veröffentlicht. Die Konstruktion ist in *Moby/PEA* mit einem XSLT-Stylesheet realisiert worden. Nach der Eingabe eines Phasen-Event-Automaten in XML-Darstellung wird ein Transition-Constraint-System ausgegeben, das in *ARMC* geladen werden kann. Der Model-Checker wird remote angesteuert.

5.1.2 Funktionale und nichtfunktionale Anforderungen

Im Folgenden werden die funktionalen und nichtfunktionalen Systemanforderungen an den zu implementierenden Compiler ermittelt. Zu diesem Zweck wird die Funktionsweise des Compilers gemeinsam mit den benötigten Funktionen genau beschrieben. Es wird näher betrachtet, welche Kriterien neben der Funktionalität unerlässlich sind, damit der Compiler nutzbar ist.

Funktionale Anforderungen

Der Compiler übersetzt Testformeln in Phasen-Event-Automaten. Der Benutzer muss daher die Möglichkeit haben, Testformeln in den Compiler einzugeben. Um Fehlern vorzubeugen, werden Testformeln mittels XML-Dateien spezifiziert, die vom Compiler gegen ein XML-Schema verifiziert werden. Dieses Schema muss entworfen werden. Der Vorteil einer XML-Darstellung von Testformeln ist die strenge Typisierung. Beim Einlesen der

[2] für jedes Disjunktionsglied ein Netz, wie oben erläutert

Datei wird geprüft, ob ausschließlich Elemente verwandt worden sind, die in dem entsprechenden Kontext gültig sind. Dabei hängt die Überprüfung einzig vom Schema ab.
Um Testformeln übersetzen zu können, werden sie in model-checkbare Normalform überführt. Es ist ein Algorithmus zu entwickeln, der eine als XML-Datei gegebene Formel in eine Formel in model-checkbarer Form übersetzt. Um Testformeln in model-checkbarer Form direkt angeben zu können, wird ein eigenes Datenformat für Formeln dieser Gestalt eingeführt. Damit kann der erst Compile-Schritt bei Bedarf entfallen. Ferner ist der erste Compile-Schritt vom System entkoppelt und kann gegen eine andere Implementierung getauscht werden. Damit wird die Struktur des Compilers modular gehalten.
Zu Formeln in model-checkbarer Darstellung werden Phasen-Event-Automaten nach der in Kapitel 4 definierten Semantik konstruiert. Ein Algorithmus ist zu erarbeiten, der diese Funktionalität bietet. Die Darstellung der Phasen-Event-Automaten in Java soll genutzt werden. Ferner soll der Algorithmus auf dem Compile-Algorithmus für Countertraces aufsetzen, da dieser bereits nahezu die vollständige Potenzmengenkonstruktion vornimmt.
Um die konstruierten Phasen-Event-Automaten in *Moby/PEA* laden zu können, muss eine XML-Ausgabe für Phasen-Event-Automaten geschaffen werden. Es besteht im Kontext von *Moby/PEA* bereits ein XML-Datenformat[3] zur Darstellung von Phasen-Event-Automaten. Formeln werden in diesem Format nur als Strings repräsentiert. Ferner ist, begründet durch die Nutzung der DTD, kaum Typensicherheit für Daten gegeben. Es wird für Datenwerte also nicht geprüft, ob sie einem Typ genügen. Es soll mit dem Paket `pea.modelchecking` eine neues XML-Datenformat für Phasen-Event-Automaten, basierend auf XML-Schema, geschaffen werden, welches die folgenden Anforderungen erfüllt:

- Die bestehenden XML-Daten, welche bezüglich der DTD valide sind, sollen bezüglich des neuen Schema-Formates gültig sein, zum einen um die Anpassungen der Export-Funktion von Moby/PEA an das neue Format gering zu halten, zum andern, um das Werkzeug, das Phasen-Event-Automaten in Transition-Constraint-Systems überführt, nicht reimplementieren zu müssen.

- Ein Datenformat für Formeln ist zu entwerfen, welches die Darstellung von Formeln als Strings ersetzt. Es ist als Teil des XML-Schemas zu entwickeln und ermöglicht daher, Formeln statisch auf konsistente Daten zu überprüfen.

- Es ist ein Konvertierungsalgorithmus für Phasen-Event-Automaten in Java nach XML zu implementieren. Damit können durch den Compiler in Java erzeugte Phasen-Event-Automaten in XML-Dateien geschrieben werden, die in *Moby/PEA* geladen werden können.

- Das Paket `pea.modelchecking` muss Funktionalität zum Konvertieren von Phasen-Event-Automaten in XML nach Java bieten. Mit dieser Funktion können in *Moby/PEA* erstellte und nach XML exportierte Phasen-Event-Automaten mit Java

[3]spezifiziert durch eine Dokument-Type-Definition (kurz: DTD)

Funktionen des Paketes `pea` parallel komponiert werden. Damit ist es nicht notwendig, die Parallekomposition von Phasen-Event-Automaten auf XML-Ebene zu implementieren.

- Das Datenformat für Phasen-Event-Automaten soll ein hohes Maß an Typsicherheit und Konsistenz in Daten schaffen. Das XML-Schema ist mit notwendigen Referenz- und Typmechanismen zu versehen.

Nichtfunktionale Anforderungen

Unter nichtfunktionalen Anforderungen werden nicht einzelne Funktionen verstanden, die ein System zu leisten hat, sondern funktionsübergreifende Merkmale, die die zu entwickelnde Software erfüllen muss. Nach [Som01] werden nichtfunktionale Anfoderungen unterteilt in Produkt-, Unternehmens- und externe Anforderungen. Da das Paket `pea.modelchecking` nicht innerhalb eines standardisierten Entwicklungsprozesses erarbeitet wird, ergeben sich keine Unternehmensanforderungen im Sinne von [Som01]. Die Produkt- und externen Anforderungen an das Paket werden nachfolgende diskutiert:
Wie bereits angesprochen, werden Testformeln als XML-Dateien in das System eingegeben. Daraus ergeben sich unmittelbar Benutzbarkeitsanforderungen für den Compiler. Es ist notwendig, eine für Menschen leicht lesbare und leicht schreibbare XML-Darstellung der Testformeln zu schaffen. Es sind geeignete Tag-Namen und eine intuitive Darstellung der Daten als Attribute oder Elemente im Schema festzulegen. Um das Ergebnis der Kompilation nachzuprüfen, ist eine lesbare XML-Darstellung der Phasen-Event-Automaten elementar. Dabei sind die Tag-Namen und die Wahl der Elemente sowie Attribute von *Moby/PEA* vorgegeben. Es ist eine übersichtliche Darstellung für Formeln in Invarianten und Guards zu entwerfen.
Schlägt die Übersetzung gegebener Testformeln fehl, muss der Benutzer des Systems darüber unterrichtet werden, wo in der Kompilation ein Fehler auftrat, oder ob bereits die Eingabe fehlerhaft war. Validierungsmechanismen, das Loggen der Funktionalität und Fehlerbehandlung sind Anforderungen, die einzuhalten sind, damit der Compiler nutzbar ist.
Mit dem Testformel-Compiler soll das Paket `pea.modelchecking` entstehen, welches neben der Compile-Funktionalität auch Konvertierungsmethoden für Testautomaten bietet. Es wird als Schnittstelle zwischen der Java-Darstellung von Phasen-Event-Automaten durch das Paket `pea` und der XML-Darstellung von Phasen-Event-Automaten in *Moby/PEA* fungieren. Für das Paket ergibt sich die Benutzbarkeitsanforderung gut dokumentierten und leicht lesbaren Codes, damit die Einbindung in *Moby/PEA* im Rahmen des *AVACS* Projektes stattfinden kann.
Der Testformel-Compiler bewältigt eine hoch komplexe Aufgabe: bereits die Konstruktion der Testautomaten ist vielfach exponentiell, zusätzlich müssen die Normalform berechnet und die Konvertierungen vollzogen werden. Die grundlegende Effizienzanforderung ist daher die Realisierung einer performanten Implementierung zur praktischen Anwendbarkeit des Werkzeugs. Unter Performanz wird dabei eine möglichst geringe Dauer der Kompilation bei einer Eingabe verstanden.

Aus der Analyse des Kontextes[4], in dem das Paket `pea.modelchecking` entwickelt wird, ergeben sich zahlreiche externe Anforderungen:
Das Werkzeug soll als Unterpakt `pea.modelchecking` des Paketes `pea` entworfen werden. Damit ist die Verwendung der Programmiersprache Java vorgegeben. Ferner ist das bestehende Java-Datenformat für DC Phasen, Events, Phasen-Event-Automaten und Formeln zu nutzen, um auf die Implementierung der Potenzmengenkonstruktion und der Parallelkomposition für Phasen-Event-Automaten zurückgreifen zu können.
Das Paket ist derart zu gestalten, dass das Datenformat in Form eines XML-Schemas für Phasen-Event-Automaten kompatibel zur bestehenden DTD in *Moby/PEA* ist. *Moby/PEA* enthält einen Export-Mechanismus für Phasen-Event-Automaten, welcher andernfalls stark überarbeitet werden müsste. Desweiteren existiert ein Werkzeug zur Übersetzung von als XML Dateien gegebenen Phasen-Event-Automaten in Transition-Constraint-Systems in *ARMC* lesbarer Darstellung. Mit Einführung eines inkompatiblen Schemas wäre dieses Tool nicht mehr nutzbar.
Der Compiler, aber auch die Konvertierungsmethoden sind so zu gestalten, dass sie als Plugin in *Moby/PEA* genutzt werden können. Das bedeutet, *Moby/PEA* kann die Funktionalität verwenden, ohne Kenntnis über die Implementierung zu haben. Die Parameter des Tools liefern alle Informationen, die zur Nutzung benötigt werden.

5.2 Entwurf

Das Paket `pea.modelchecking` stellt die Logik bereit, um in XML-Darstellung angegebene Testformeln in Phasen-Event-Automaten zu übersetzen. Das Ergebnis des Kompilationsprozesses sind mehrere XML-Dateien. Eine jede enthält die Phasen-Event-Automaten, die zu einem Disjunktionsglied der model-checkbaren Darstellung der Testformel gehören. Das Parallelprodukt der Automaten in einer Datei wird nur dann im Compiler ausgerechnet, wenn ein entsprechendes Flag übergeben wird. Ferner stellt das Paket `pea.modelchecking` die Funktionalität zur Verfügung, die Berechnung des Produktautomaten auf dem Kompilationsergebnis nachzuholen. Für die Entscheidung, die Berechnung nicht grundsätzlich mit der Kompilation durchzuführen, sprechen mehrere Argumente: Mit gesetzter Compileroption wird der Produktautomat zu jedem Disjunktionsglied bestimmt. In Abschnitt 4.5 ist darauf hingewiesen worden, dass die Produktautomaten für die Disjunktionsglieder nacheinander ausgerechnet werden sollten, da die Model-Checking Prozedur beendet werden kann, sobald ein Disjunktionsglied erfüllt ist. Damit ist es unnötig, die verbleibenden Produktautomaten auszurechnen. Soll nur der Automat zu einer Formel inspiziert werden, so ist der Aufwand, die parallele Komposition zu berechnen, nicht erwünscht. Indem das Berechnen des Parallelproduktes nicht grundsätzlich durchgeführt, die Möglichkeit jedoch bereitgestellt wird, die Berechnung im Nachhinein durchzuführen, erreicht der Compiler eine Kapselung der Funktionalität. Bei Bedarf kann das Produkt berechnet werden, bietet jedoch der zu nutzende Model-Checker eine performantere Implementierung dieser Funktion, so kann die Berechnung

[4] siehe Abschnitt 5.1.1

dort vorgenommen werden.

Zunächst wird die Modellierung der Daten durch XML-Schemata vorgestellt. Nach einem Überblick über die Klassen des Paketes `pea.modelchecking` und ihrer Interaktion wird auf die statische Systemsicht eingegangen. Dabei werden die bedeutenden Klassen des Paketes `pea.modelchecking` sowie die umgestaltete Klasse `Trace2PEACompiler` des Paketes `pea` mit ihren Methoden erläutert. In der folgenden Beschreibung der dynamischen Systemsicht wird die Interaktion der Klassen aufgezeigt.

5.2.1 Datenmodellierung

Der zu entwickelnde Compiler wird, wie in Abschnitt 5.1.1 erörtert, in Verbindung mit *Moby/PEA* und *ARMC* genutzt. Um eine wohldefinierte Schnittstelle im Austausch von Daten zwischen Compiler und *Moby/PEA* bereitzustellen, werden für alle für externe Anwendungen bedeutende Daten XML-Schemata erstellt. Die Definition der Schemata in einem austauschlastigen System entspricht der Erstellung von abstrakten Datentypen in Form von Klassen in einem in sich geschlossenen Programm.
Im Compiler soll ein validierender XML-Parser zum Einsatz kommen, der gegebene XML-Dokumente auf ihre Gültigkeit überprüft. Neben des Checks der geeigneten Struktur des Dokuments durch Prüfung der Tag-Namen werden insbesondere die verwendeten Datenwerte in Elementen oder Attributen auf ihre Typen hin geprüft. Von XML-Schema sind einfache Datentypen vorgegeben, die sich mittels in XML-Schema bereitgestellter Mechanismen verfeinern lassen oder zu abstrakten Datentypen zusammengefasst werden können. Im Folgenden werden die wichtigsten Datentypen vorgestellt, dabei wird die weitverbreitete Darstellung des XML-Schemas als erweitertes Baumdiagramm mit XML-Operatoren genutzt. Eine Legende zu den im Diagramm genutzten Operatoren ist mit Abbildung 5.1 gegeben.

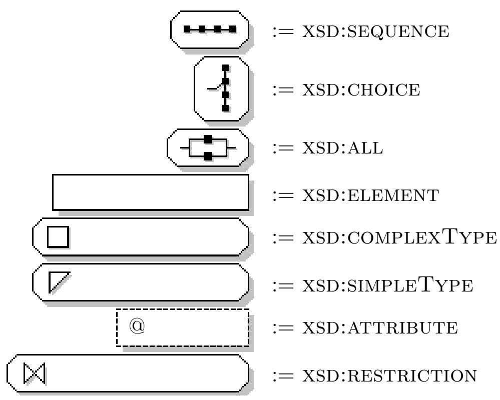

Abbildung 5.1: Legende der XML-Schema Operatoren

Traces

Die Definition der Datentypen für den Compiler ist auf vier XML-Schemata verteilt. Im Schema BasicTypes.xsd sind Typen definiert, welche als Basis zur Deklaration höherer Typen in den anderen Schemata genutzt werden. In dieser Datei findet sich die Deklaration des zusammengesetzen Typs[5] Trace. Abbildung 5.2 stellt die Elemente von Trace schematisch dar. Das boolesche spec Attribut der Trace bestimmt, ob die Trace negiert

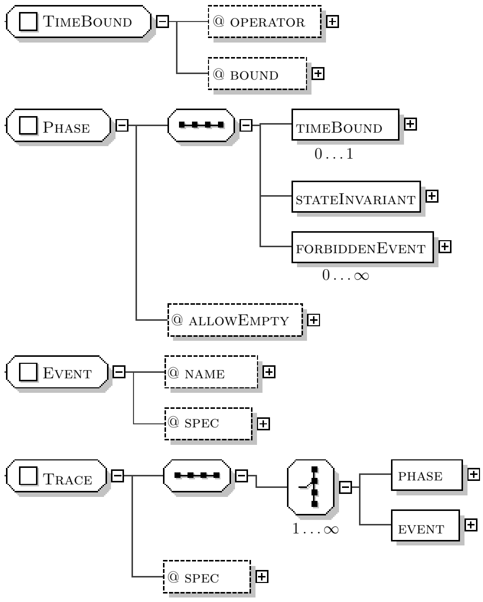

Abbildung 5.2: Datentyp Trace

(spec=false) oder nicht negiert (spec=true) aufzufassen ist. Die Trace besteht aus einer nicht-leeren Folge von Phase und Event Elementen. Die Folge muss mindestens ein Element enthalten, damit die Existenz einer führenden Phase in der Trace gewährleistet ist. Ein Event besitzt die Attribute name und spec. Name ist vom XML-Typ xsd:string und gibt den Namen des Events an. spec ist ein boolesches Attribut und besagt, ob es sich um eine Event- oder eine NoEvent-Spezifikation handelt, wobei true für eine Event-Spezifikation steht. Eine Phase wird angegeben als Sequenz bestehend aus einer optionalen Angabe eines Bounds vom Typ TimeBound, einem Zustandsausdruck gegeben durch das Attribut stateInvariant vom Typ Formula und einer optionalen Menge an verbotenen Events, angegeben als forbiddenEvent Elemente mit Datentyp EventExpression. Eine Phase besitzt das optionale boolesche Attribut allowEmpty. Es soll in zukünftigen Versionen des Compilers für eine Phase angeben, ob sie von einem leeren Intervall erfüllt wird. Der Compiler liest das Attribut aus, ignoriert jedoch den Wert und übergibt zur weiteren Verarbeitung *false*. Soll die Verwendung von allowEmpty implementiert werden, so ist im Compiler sehr geringer Anpassungsaufwand nötig. Ein TimeBound besitzt die Attribute bound vom Typ ContinuousTime, und

[5] xml: complexType

OPERATOR, dem als Wert ein String aus {GREATER, GREATEREQUAL, LESS, LESSEQUAL} zugewiesen wird. Die Datentypen FORMULA und EVENTEXPRESSION werden im nachfolgenden Kapitel erläutert.

Formeln

Der Compiler greift auf drei verschiedene Typen von Formeln zurück: Elemente vom Typ FORMULA dienen als Invarianten in Traces, als Zustandsinvarianten und Uhrenbedingungen in Phasen-Event-Automaten sowie als Guards an Transitionen. Zur Repräsentation von Testformeln wird der Datentyp TESTFORM genutzt. Die model-checkbare Darstellung von Testformeln ist mit MCFORM im Compiler gegeben. Die model-checkbare Darstellung von Testformeln kann mit dem Datentyp TESTFORM ausgedrückt werden, jedoch sorgt ein eigenes, strengeres Format für eine verbesserte Typprüfung. Ferner stellt die model-checkbare Darstellung eine Normalform dar, die eine strikte Struktur aufweist: Innerhalb der Disjunktionen sind alle Elemente durch Konjunktionen verbunden. Durch geeignete Darstellung dieser Struktur kann das Konvertieren von Testformeln vereinfacht und damit beschleunigt werden. Ferner wird Speicherplatz gespart.

Datentyp Formula Der Datentyp FORMULA ist im Schema BASICTYPES.XSD deklariert. Eine Formel besteht entweder aus einem einzigen Element vom Typ BOOLEANEXPRESSION, EVENTEXPRESSION oder RANGEEXPRESSION oder aber aus einem Formelbaum vom Typ FORMULATREE welcher in Abbildung 5.3 dargestellt ist. Ein FORMULATREE besitzt ein Attribut OPERATOR, das den Operator re-

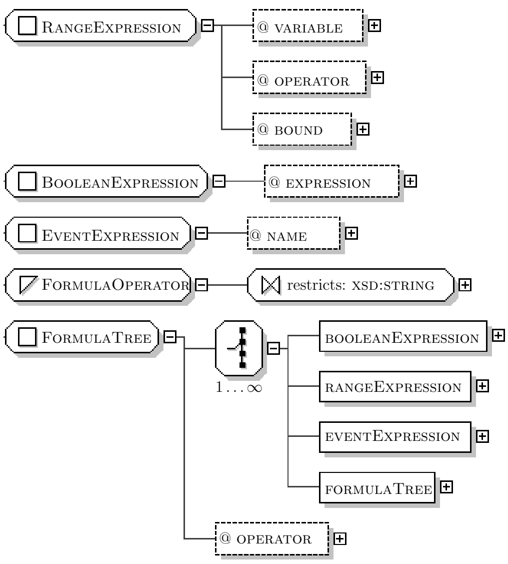

Abbildung 5.3: Datentyp FORMULATREE

präsentiert, mit dem die Kindelemente des FORMULATREE-Elementes verbunden werden. Dabei umfasst der Datentyp von OPERATOR die booleschen Operatoren

AND, OR und NOT. Als Kindelemente sind die für FORMULA genannten Typen zulässig, d.h. insbesondere besteht ein FORMULATREE rekursiv aus FORMULATREE Elementen. Damit ist eine Formel als Baum, also als dynamische Datenstruktur, in XML definiert. Die Anzahl an Kind Elementen eines FORMULATREE Elements ist nicht nach oben beschränkt. So ist es möglich, eine große Anzahl von Elementen zu verknüpfen, ohne dass der Baum in die Tiefe wächst. Ein Baum muss mindestens ein Kind besitzen, da die Negation der einzige unäre Operator ist, die übrigen Operatoren sogar zwei oder mehr Operanden erwarten. Wird die Negation mit mehr als einem oder die anderen Operatoren mit weniger als zwei Operanden versehen, so gibt der Compiler eine Fehlermeldung aus.
Eine EVENTEXPRESSION stellt ein Event in einer Testformel oder ein Element der Menge A eines PEA in XML dar. Dabei nimmt der Paramter NAME vom Typ String den Namen des Events auf. Eine BOOLEANEXPRESSION repräsentiert einen Ausdruck, dem ein boolescher Wert zugewiesen wird. Der Datentyp wird benutzt, um Prädikate, die sich nicht mit RANGEEXPRESSIONS darstellen lassen, zu beschreiben. Syntaktisch unterscheidet eine BOOLEANEXPRESSION nur die Benennung des Parameters von einem Event: Die EXPRESSION spezifiziert die Aussage, die durch die BOOLEANEXPRESSION ausgedrückt wird. Eine RANGEEXPRESSION gibt eine Bedingung für eine getypte Variable, gegeben als Attribut VARIABLE, an. Die aktuelle Version des Compilers ignoriert die Typisierung und nimmt jede Variable als reellwertig an. Als Bedingungen werden in der ersten Compilerversion einzig Vergleiche der Variablen mit Konstanten erlaubt. Aus dem Grund sind als Operatoren nur GREATER, GREATEREQUAL, LESS, LESSEQUAL, EQUAL und NOTEQUAL vorgesehen, welche im Attribut OPERATOR gespeichert werden. Das Attribut BOUND vom Typ XSD:DOUBLE gibt die Konstante des Vergleichs an.

Datentyp TestForm Die schematische Darstellung des Datentyps TESTFORM ist in Abbildung 5.4 gegeben. Eine Testformel ist analog zu Formeln vom Typ FORMULA als Baum angegeben, dessen Elemente jedoch vom Typ TFTREE oder TRACE sind. Als Operatoren sind AND, OR, CHOP und Sync-Events, dargestellt durch

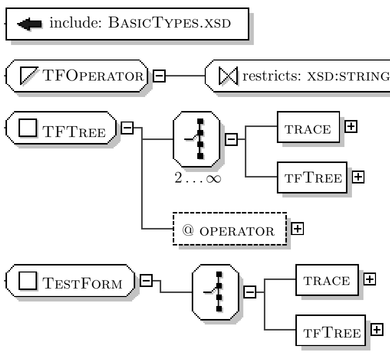

Abbildung 5.4: Datentyp TESTFORM

ein S mit folgenden Zahlen, erlaubt. Ein TFTree verlangt mindestens zwei Kinder, da alle Operatoren mindestens zwei Operanden erwarten. Die Definition von Testformeln ist im Schema TestForm.xsd vorgenommen.

Datentyp MCForm Die model-checkbare Darstellung einer Formel ist gegeben durch eine nicht-leere Folge von MCForm Elementen, die den Disjunktionsgliedern der Formel entsprechen. Ein jedes MCForm Element besitzt eine nicht-leere Folge von MCTrace-Elementen, welche den Formeln in Zeile 3.26 aus Korollar 3.3.2 entsprechen. Vor einem führenden Sync-Event steht immer eine $\lceil true \rceil$-Phase, ebenso folgt auf das ausgehende Sync-Event immer eine $\lceil true \rceil$-Phase. Aus diesem Grund werden allein die Sync-Events als Attribute von MCTrace dargestellt, die Darstellung der $\lceil true \rceil$-Phasen kann unterbleiben. Das erste Sync-Event wird durch das Attribut entrySync von MCForm, das zweite Sync-Event durch das Attribut ExitSync dargestellt. Beide Attribute sind vom Typ xsd:string, da Event-Expression als complexType nicht in Attributen verwendet werden darf. Das entrySync-Event ist optional, das zweite Event muss angegeben werden. Abbildung 5.5 veranschaulicht die gegebene Beschreibung. Die Elemente sind in der Datei ModelCheckForm.xsd definiert.

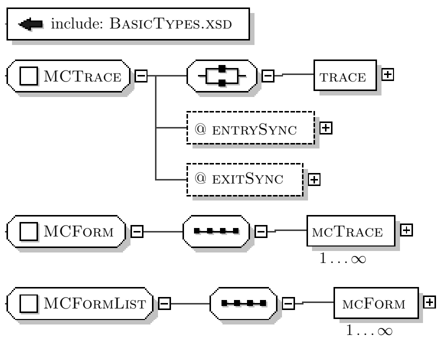

Abbildung 5.5: Datentyp MCForm

Phasen-Event-Automaten

Im XML-Schema PEA.xsd ist der Datentyp PEANet definiert. Ein PEANet Element repräsentiert eine Menge von PEAs, die parallel zu komponieren sind.
Der Datentyp für Phasen-Event-Automaten ist PEA, dargestellt in Abbildung 5.6 und besteht aus einer optionalen Liste aller Variablen des PEA, variables vom Typ VariableList, einer optionalen Liste aller Uhren, clocks vom Typ ClockList, einer optionalen Auflistung aller Events, events vom Typ EventList, der Menge aller Phasen, phases vom Typ PhaseList und der optionalen Menge aller Transitionen, transitions vom Typ TransitionList. Die Menge aller Phasen muss angegeben werden und darf nicht leer sein. Ferner besitzt ein PEA das Attribut name, das seinen Namen angibt. Über Schlüsselmechanismen von XML-Schema wird bereits bei der Erstellung

des XML-Dokuments sichergestellt, dass der Name eines jeden PEAs in einem PEANet eindeutig ist.
Eine Phase in einem PEA ist vom Datentyp PEAPhase, ihre Invariante wird mit dem

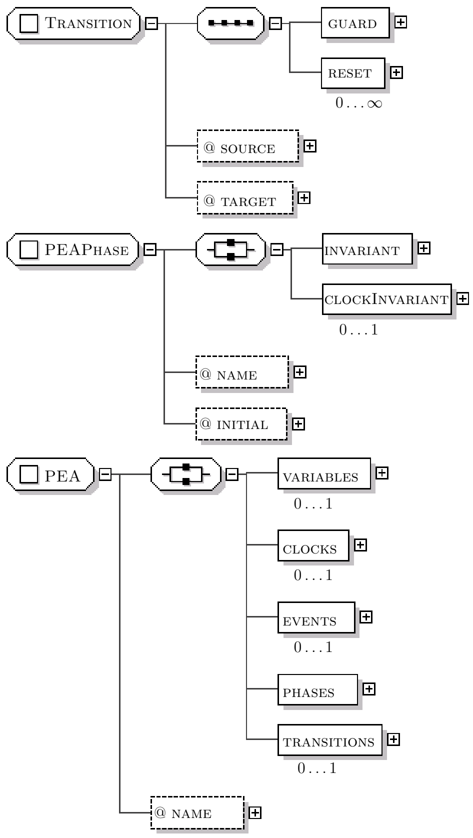

Abbildung 5.6: Datentyp PEA

Element invariant vom Typ Formula angegeben. Eine Uhrenbedingung ist optional und kann mit clockInvariant ebenfalls als Formel angegeben werden. Eine Phase besitzt mit dem Attribut name einen Namen, der innerhalb eines PEAs eindeutig sein muss. Das optionale boolesche Attribut initial gibt an, ob es sich um eine initiale Phase handelt, der default Wert ist *false*. XML-Schema garantiert über Key-Referenzen, dass alle Events, Variablen und Uhren, die in Formeln verwendet werden, in den entsprechenden Listen deklariert sind.
Transitionen vom Typ Transition besitzen einen Guard, guard, vom Typ Formula und eine Menge von Uhren, reset vom Typ Clock, die beim Passieren der Transition zurückgesetzt werden. Das Attribut source gibt die Quelle, target das Ziel der Transition an. Die Werte von source und target sind jeweils Referenzen auf die entsprechenden Phasen.
Die Menge der Events, Variablen und Uhren separat in Listen anzugeben, erhöht den Informationsgehalt eines XML-Dokuments nicht, da alle Angaben in den übrigen XML-Elementen enthalten sind. Dennoch bietet es für *Moby/PEA* komfortableren Zugriff auf

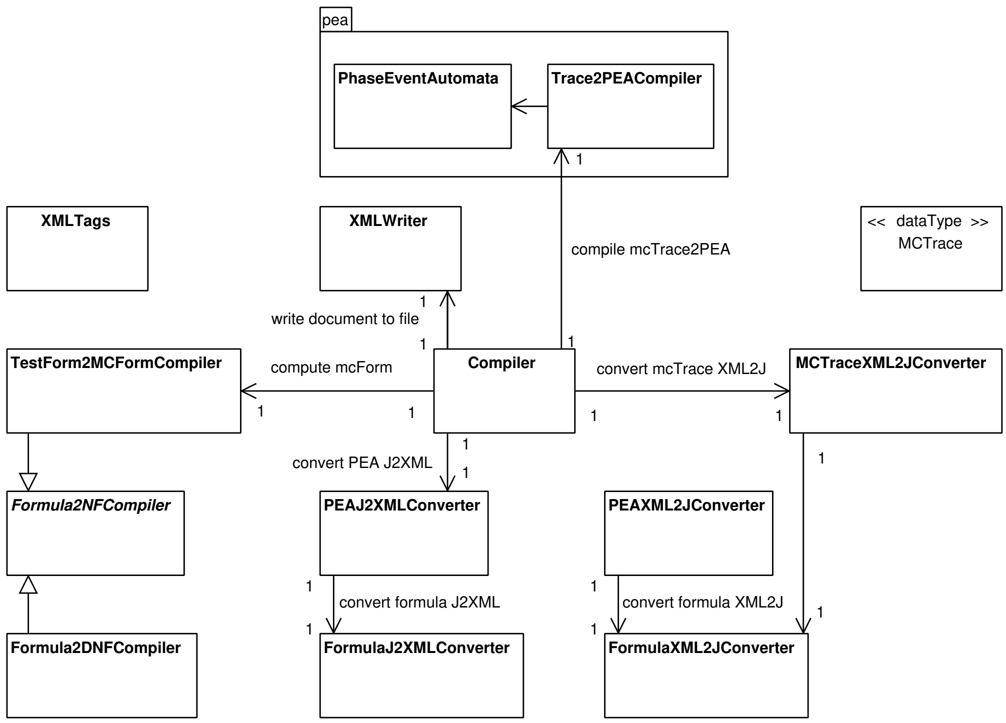

Abbildung 5.7: Paket `pea.modelchecking`

die entsprechenden Werte.

5.2.2 Architektur-Übersicht

Kern des in Abbildung 5.7 dargestellten Paketes `pea.modelchecking` ist die Klasse `Compiler`. Sie wird durch den Benutzer oder durch ein anderes Tool aufgerufen. Dabei wird ihr der Name der XML-Datei mit der zu übersetzenden Testformel übergeben. Die Datei entspricht dem XML-Schema TESTFORM.XSD. Weitere Parameter bestimmen, in welche Dateien Ergibnisse des Kompilationsprozesses geschrieben werden, sowie, ob die resultierenden PEAs parallel zu komponieren sind.
Der Compiler benutzt die Klasse `TestForm2MCFormCompiler`, um die gegebene Testformel in eine Formel der model-checkbaren Darstellung zu überführen. Das Resultat dieses Prozesses ist ein XML-Dokument entsprechend dem Schema MODELCHECKFORM.XSD. Die Klasse `TestForm2MCFormCompiler` erbt Basismethoden von der abstrakten Klasse `Formula2NFCompiler`, um Formeln in eine Normalform zu überführen.
Die model-checkbare Darstellung besteht aus einer Liste von MCFORM Elementen, die den Disjunktionsgliedern entsprechen. Ein jedes MCFORM Element enthält wiederum, den Konjunktionsgliedern entsprechend, eine Sequenz von MCTRACE Elementen. Die Klasse `MCTraceXML2JConverter` überführt jedes MCTRACE Element in ein Java-Objekt vom Typ `MCTrace`. Formeln, die in Zustandsausdrücken der Trace vorhanden sind, werden mittels eines `FormulaXML2JConverters` in Java-Objekte umgewandelt. Auf diese

Weise entsteht eine erhöhte Kapselung der Systemfunktionalität: Die XML-Darstellung von Formeln kann geändert werden und es ist im Paket `pea.modelchecking` nur die Komponente `FormulaXML2JConverter` anzupassen.
Die Klasse `Trace2PEACompiler` des Paketes `pea` wird vom Compiler genutzt, um `MCTrace` Objekte in Objekte vom Typ `PhaseEventAutomata` zu kompilieren.
Nach der eigentlichen Übersetzung durch die Klasse `Trace2PEACompiler` verwendet die `Compiler` Klasse einen `PEAJ2XMLConverter`, um die Java-Objekte[6] in XML-Elemente des Formats PEA umzuwandeln. Die Transformation von Formeln innerhalb der PEAs wird durch die Klasse `FormulaJ2XMLConverter` übernommen, welche das Gegenstück zu `FormulaXML2JConverter` darstellt. Alle PEAs zu einem Disjunktionsglied werden von der Klasse `PEAJ2XMLConverter` in einem, dem Schema PEA.XSD entsprechenden, XML-Dokument zusammengefügt.
Die Klasse `XMLWriter` wird vom Compiler verwendet, um die XML-Dokumente mit den Automaten der Disjunktionsglieder in Dateien zu schreiben.
Das Paket `pea.modelchecking` umfasst neben den angeführten Klassen zur Kompilation die Klassen `PEAXML2JConverter` und `Formula2DNFCompiler`. Objekte vom Typ `PEAXML2JConverter` lesen eine gegebene XML-Datei mit einem Netz von Phasen-Event-Automaten ein und konvertieren dieses Netz in ein Array von `PhaseEventAutomata` Objekten. Für die Konvertierung von Formeln wird ein `FormulaXML2JConverter` genutzt. *Moby/PEA* benutzt die Klasse `PEAXML2JConverter`, um nach XML exportierte PEAs auf Java-Ebene mit dem PEA zu einem Disjunktionsglied der Testformeldarstellung parallel komponieren zu können. Die Klasse `PhaseEventAutomata` stellt eine entsprechende Methode zur Verfügung. Ein `Formula2DNFConverter` nimmt eine Formel in XML-Darstellung entgegen und überführt diese in disjunktive Normalform. Da *ARMC* die Eingabe von Formeln in DNF fordert, ist die Funktionalität für *Moby/PEA* bereitzustellen.
In `XMLTags` werden alle XML-Bezeichner der Schemata vorgehalten. In den Klassen des Paketes wird dann mit den statischen Variablen dieser Klasse gearbeitet. Bei Änderung eines XML-Tags in den Schemata ist so nur eine Klasse im Paket `pea.modelchecking` anzupassen.

Der Compiler arbeitet sowohl auf XML-Elementen als auch auf Java-Objekten. Die Trennung der Funktionalität entlang der Datenrepräsentation ist in Abbildung 5.8 dargestellt: Während die Berechnung der model-checkbaren Darstellung oder der DNF einer Formel auf XML-Ebene erfolgt, wird die Konstruktion der Testautomatensemantik sowie die Parallelkomposition auf Java-Ebene vorgenommen.
Die Berechnung einer Normalform erfordert eine Repräsentation der Formel als Baum. Dieser Baum wird mit rekursiv angewandten Ersetzungsvorschriften manipuliert. Eine mit DOM-geparste XML-Datei wird direkt als Baum im Speicher dargestellt. Der Parser bietet alle Funktionalität, die zur Manipulation notwendig ist. Überdies ist der Aufwand, eine Formel in model-checkbare Normalform zu überführen, gering im Vergleich zu dem Aufwand, Automaten zu der gegebenen Formel zu konstruieren. Mit diesem Argument

[6] vom Typ `PhaseEventAutomata`

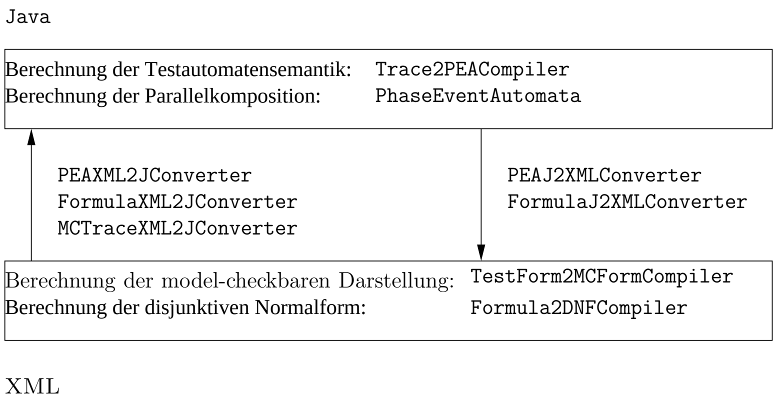

Abbildung 5.8: Funktionalität des Compilers und Format der Daten

wird die Berechnung der model-checkbaren Darstellung für Formeln auf der XML-Ebene vorgenommen. Für die Weiterentwicklung des System ist zu evaluieren, ob eine Berechnung der DNF auf Java-Ebene Performanzvorteile bringt. Bei einer großen Anzahl an Formeln mag die Manipulation auf XML-Basis zu Geschwindigkeitseinbußen führen.
Die Berechnung der Testautomatensemantik erfolgt mit der Klasse `Trace2PEACompiler`, die eine Weiterentwicklung der Klasse `BuildCTA` darstellt. Die Berechnung der Potenzmengenkonstruktion war bereits in `BuildCTA` vorhanden und konnte unter geringen Änderungen übernommen werden. Ferner arbeitet die Klasse mit Bitmasken und CDDs, womit eine performante Implementierung garantiert ist. Unter den Aspekten des geringen Anpassungsbedarfs und der performanten Implementierung wurde die Berechnung der Testautomaten in Java im Paket `pea` durchgeführt.
Die in Abbildung 5.8 an den Pfeilen angegebenen Klassen mit Namen `XML2J` dienen der Konvertierung von PEAs, MCTraces und Formeln von XML nach Java. Die Klassen mit Namen `J2XML` überführen PEA und Formel Objekte in XML-Elemente.

5.2.3 Statische Systemsicht

Die Klasse `Compiler`

Die Kompilation einer gegebenen Testformel in XML-Darstellung wird über die Klasse `Compiler` angestoßen. Zu diesem Zweck beinhaltet die Klasse eine `main`-Methode, der Parameter übergeben werden können:

`-tf` Der Parameter `-tf` gibt die XML-Datei der Testformel an, die übersetzt werden soll. Der Parameter muss verwendet werden, wird er weggelassen, gibt der Compiler die Bedienungsanleitung aus und terminiert.

`-mc` Mit `-mc` wird der Name der Datei angegeben, in welche die model-checkbare Darstellung der Testformel geschrieben werden soll. Der Parameter ist optional, per default wird an die Datei der Testformel das Suffix `_mc.xml` gehängt.

`-pea` Gibt den Namen für Phasen-Event-Automaten innerhalb der Netze an. An den angegebenen Namen wird jeweils die Nummer des Automaten gehängt. Standardmäßig wird der Name `pea` vergeben, die Verwendung des Parameters ist daher nicht verpflichtend.

`-net` Die zu den Disjunktionsgliedern konstruierten Netze von Phasen-Event-Automaten werden in Dateien `peaNet`, gefolgt von der Nummer des Disjunktionsglieds, gespeichert. Sollen andere Dateien verwandt werden, kann der neue Dateiname über den Parameter `-peaNet` angegeben werden.

`-lc` Eine Konfigurationsdatei für die im Compiler enthaltenen Logger kann über `-lc` angegeben werden.

`-help` Mit `-help` kann sich der Benutzer eine Erklärung zur Verwendung des Compilers ausgeben lassen.

`-parallel` Ist der Parameter `-parallel` auf *true* gesetzt, so wird die parallele Komposition aller PEAs in den Netzen der Disjunktionsglieder ausgerechnet. Das Ergebnis wird in eine Datei mit dem Namen der Netzdatei und der Endung `_par.xml` geschrieben.

Die Main-Methode leitet die ausgewerteten Parameter unmittelbar an die `compile` Methode weiter, welche im Kapitel 5.2.4 näher beschrieben ist.

Normalform-Compiler Klassen

In Abbildung 5.9 sind die Klassen zur Berechnung der disjunktiven Normalform einer prädikatenlogischen Formel und der model-checkbaren Darstellung einer Testformel angegeben. Die disjunktive Normalform wird über einen `Formula2DNFCompiler` konstruiert, die Klasse `TestForm2MCFormCompiler` erzeugt die model-checkbare Darstellung. Beide Klassen erben die Basisalgorithmen zur Bestimmung einer Normalform von der abstrakten Klasse `Formula2NFCompiler`.

Die Klasse `Formula2NFCompiler` Im Folgenden werden die Methoden der Klasse vorgestellt:

`isTreeElement` Entscheidet für einen gegebenen Knoten, ob es sich um einen Teilbaum handelt. Die Methode ist von den erbenden Klassen zu implementieren.

`isBasicElement` Entscheidet für einen gegebenen Knoten, ob es sich um ein Blatt des Baumes handelt. Die Methode ist von den erbenden Klassen zu implementieren.

`makeBinary` Überführt einen gegebenen Formelbaum in einen Binärbaum, einzig die mit der Negation beschrifteten Knoten besitzen einen Nachfolger. Die Methode ist in der abstrakten Klasse zu realisieren und muss nicht überschrieben werden.

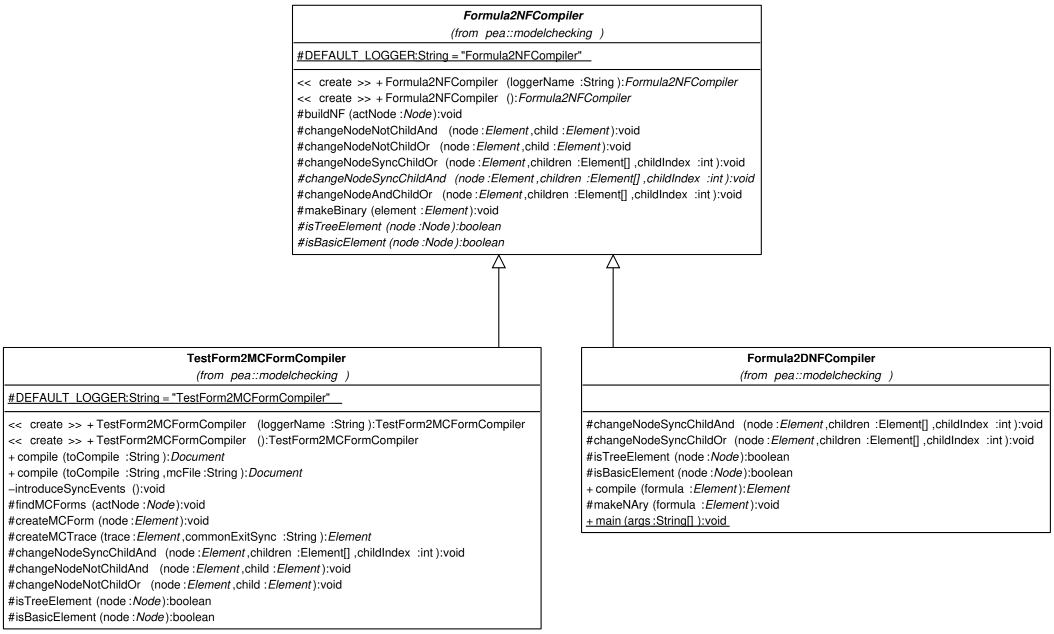

Abbildung 5.9: Normalform-Compiler Klassen

`buildNF` Die Konstruktion der Normalform wird von der Methode `buildNF` rekursiv vorgenommen, sie ist schematisch im Aktivitätsdiagram 5.10 angegeben: Zunächst

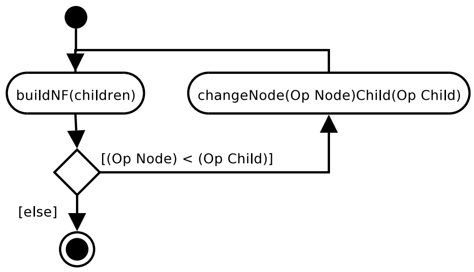

Abbildung 5.10: Aktivitätsdiagramm zur Methode `buildNF`

werden die Kindknoten eines Elternknoten in Normalform gebracht. Anschließend wird geprüft, wie sich der Operator des Elternknotens zu den Operatoren der Kindknoten verhält. Gegebenenfalls muss der Baum verändert werden. Als Hilfsmittel dienen die Methoden `changeNode(Operator Node)Child(Operator Child)`.

`changeNode(Operator Node)Child(Operator Child)` Die Methode gibt an, inwiefern ein Baum zu verändern ist, wenn der Elternknoten den Operator `Operator Node` besitzt und ein Kindknoten den Operator `Operator Child`. Die besondere Distributivität zwischen Sync-Events und der Konjunktion wird mit der Methode `changeNodeSyncChildAnd` implementiert. Sie verändert den Baum wie in Abbildung 5.11 angegeben: B_1, B_2, B_3 stellen die Traces T_1, T_2, T_3 dar. Das Diagramm veranschaulicht den Sachverhalt $(T_1 \wedge T_2) \underset{\mathcal{S}}{\Updownarrow} T_3 \Leftrightarrow (T_1 \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil) \wedge (T_2 \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil) \wedge (\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} T_3)$. Für die Operatoren, die sowohl in Formeln als auch in Test-

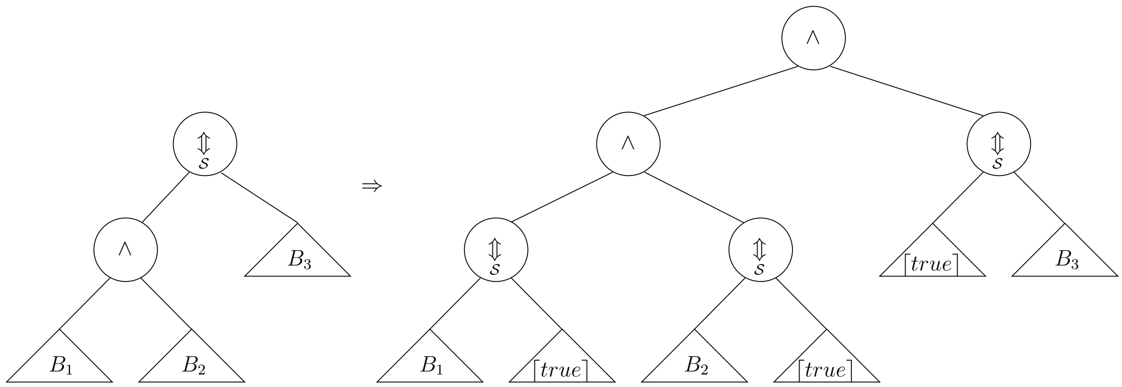

Abbildung 5.11: Änderung des Formelbaumes durch `changeNodeSyncChildAnd`

formeln enthalten sind, sind die Methoden in der abstrakten Klasse implementiert. Die übrigen Methoden müssen in den abgeleiteten Klassen überschrieben werden.

Die Klasse `TestForm2MCFormCompiler` Neben der ererbten Funktionalität bietet die Klasse `TestForm2MCFormCompiler` Methoden an, um den erstellten Formelbaum in eine Darstellung zu überführen, die dem Schema ModelCheckForm.xsd entspricht. Zu diesem Zweck werden weitere Methoden bereitgestellt:

`compile` Mit `compile` wird die Übersetzung angestoßen. Als Parameter kann neben der zu übersetzenden Datei eine weitere Datei angegeben werden, in welche die model-checkbare Darstellung geschrieben wird.

`introduceSyncEvents` Die Methode ersetzt die Chop-Operatoren durch Sync-Events. Die Anwendung entspricht der Äquivalenz im Lemma Sync-Event-Introduction.

`findMCForms` Ein ModelCheckForm.xsd entsprechendes XML-Dokument enthält eine Folge von MCForm Elementen, die den Disjunktionsgliedern entsprechen. Die Methode `findMCForms` sucht die Wurzelelemente aller Disjunktionsglieder einer Formel, die bereits in Normalform gebracht wurde.

`createMCForm`, `createMCTrace` Der Aufbau der MCForm Elemente wird mit der Methode `createMCForm` durchgeführt. Innerhalb eines MCForm Elements befinden sich MCTrace Elemente. Sie werden mit der Methode `createMCTrace` erstellt.

Die Klasse `Trace2PEACompiler`

Die Übersetzung von Formeln in model-checkbarer Darstellung in Testautomaten findet in der Klasse `Trace2PEACompiler` statt, die aus der Klasse `BuildCTA` hervorgegangen ist. Abbildung 5.12 zeigt die Klasse mit ihren Methoden.

Trace2PEACompiler *(from pea)*
<< create >> + Trace2PEACompiler (loggerName :String):Trace2PEACompiler << create >> + Trace2PEACompiler ():Trace2PEACompiler + compile (name :String ,ct :CounterTrace):PhaseEventAutomata + compile (name :String ,mcTrace :MCTrace):PhaseEventAutomata −precomputeCDDs ():void −initTrans (srcBits :PhaseBits):void −buildNewTrans (): −recursiveBuildTrans (): −findTrans (srcBits :PhaseBits,src :Phase):void −buildAut ():PhaseEventAutomata −buildEntrySyncTransitions ():void −buildExitSyncTransitions ():void −addForbiddenEventsToTransitions ():void −buildExitGuard (pBits :PhaseBits):CDD

Abbildung 5.12: Die Klasse `Trace2PEACompiler`

`compile` Die `compile`-Methoden initiieren die Übersetzung. Sie benutzen die schon in `BuildCTA` vorhandene Methode `buildAut`, welche derart angepasst ist, dass sie je nach Parameter Countertrace-Automaten oder Testautomaten konstruiert.

Aus `BuildCTA` übernommene Methoden Die Methoden zur Berechnung der Potenzmengenkonstruktion sind aus `BuildCTA` übernommen und so abgeändert worden, dass sie, wenn ein Testautomat konstruiert werden soll, die vollständige Potenzmengenkonstruktion durchführen, wenn ein Countertrace-Automat erstellt werden soll jedoch die Konstruktion wie in `BuildCTA` abbrechen, bevor ein Zustand erreicht wird, in dem die Formel vollständig erkannt ist.

`buildEntrySyncTransitions` Mit `buildEntrySyncTransitions` wird dem Automaten, der aus der Potenzmengenkonstruktion hervorgegangen ist, ein neuer Startzustand mit Zustandsinvariante und Uhrenbedingung *true* hinzugefügt. Von diesem Zustand wird zu jedem der vorherigen Startzustände eine Transition eingefügt, die mit dem ersten Sync-Event der Testformel und der Negation des zweiten Sync-Events beschriftet ist. Ferner erhält der neue Startzustand eine reflexive Transition, an der beide Sync-Events verboten sind. Die Methode wird nur aufgerufen, falls die Trace zwei Sync-Events besitzt.

`buildExitSyncTransitions, buildExitGuard` Es wird ein Zustand `FINAL_PEAName` mit Invariante und Uhrenbedingung *true* dem Automaten hinzugefügt. Er stellt den

Bad-State dar. Ferner wird von jedem Zustand innerhalb des Potenzmengenautomaten eine Transition zu dem neuen Zustand erstellt. Diese ist mit dem zweiten und, wenn vorhanden, der Negation des ersten Sync-Events beschriftet. Außerdem erhalten die Transitionen den Guard aus Definition 4.4.1, er wird mittels `buildExitGuard` erstellt. Der Bad-State erhält weiterhin eine reflexive Kante mit der Negation beider Sync-Events.

`addForbiddenEventsToTransitions` Die Methode verbietet das Auftreten der Sync-Events an Transitionen des Potenzmengenautomaten. Zusammen mit den vorherigen Methoden wird so gerade der Testautomat aus Abschnitt 4.4 erstellt.

Converter Klassen

Die Klassen zum Konvertieren von Phasen-Event-Automaten und Formeln zwischen Java und XML sind in Abbildung 5.13 aufgezeigt. Im Folgenden wird die Konvertierung von Java nach XML beschrieben, die umgekehrte Richtung folgt analog. Zunächst werden dazu die Methoden der Klasse `PEAJ2XMLConverter` vorgestellt:

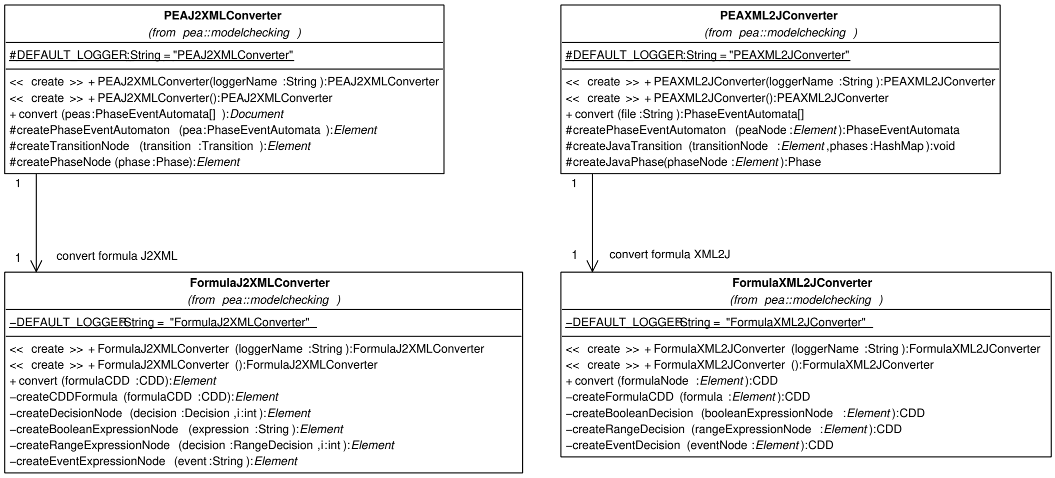

Abbildung 5.13: Converter Klassen

`convert` Mit der Methode `convert`, der ein Array von `PhaseEventAutomata` Objekten übergeben wird, wird die Konvertierung initiiert.

`createPhaseEventAutomata` Die `convert` Methode nutzt `createPhaseEventAutomata`, um einzelne Automaten in XML zu übersetzen. Die Methode deligiert wiederum die Aufgabe, die XML-Elemente für Phasen und Transitionen zu erstellen.

`createPhaseNode, createTransitionNode` Die Methode `createPhaseNode` erzeugt zu einem Java Phase Objekt ein XML PEAPhase Element. Für die Übersetzung von Formeln wird die Klasse `FormulaJ2XMLConverter` verwendet.

Die Klasse `FormulaJ2XMLConverter` erstellt zu einer als `CDD` gegebenen Formel ein XML-Element. Dabei werden folgende Methoden verwendet:

`convert` Die Konvertierung wird über die Methode `convert` begonnen. Die eigentliche Erstellung des Formelelements wird mittels `createCDDFormula` durchgeführt.

`createCDDFormula` Die Methode `createCDDFormula` wird genutzt, um rekursiv zu einer gegebenen Formel den XML-Baum aufzubauen. Die Arbeitsweise der Methode ist an `toString` in `CDD` angelehnt. Um zu einer `Decision` in einer Formel ein XML-Element zu erstellen, wird `createDecisionNode` verwendet.

`createDecisionNode`, `create[...]ExpressionNode` Mit `createDecisionNode` wird die Klasse der `Decision` bestimmt und entsprechend die Erstellung des XML-Elementes in einer der Methoden `create[...]ExpressionNode` vorgenommen.

5.2.4 Dynamische Systemsicht

Nachfolgend wird das Verhalten der wichtigsten Methoden des Compile-Vorgangs beschrieben.

Kompilationsvorgang

Die Kompilation wird von der Methode `compile` der Klasse `Compiler` gesteuert. Sie fügt die Funktionalitäten aller Komponenten des Paketes `pea.modelchecking` zusammen. Abbildung 5.14 zeigt die Übersetzungsschritte entsprechend der Reihenfolge ihres Auftretens: Nachdem die Klasse `TestForm2MCFormCompiler` mit ihrer `compile` Methode eine gegebene Testformel in model-checkbare Darstellung konvertiert hat, werden für alle Disjunktionsglieder weitere Arbeitsschritte durchgeführt: Es werden alle MCTrace-Elemente, die in einem Disjunktionsglied enthalten sind, ausgelesen und von XML nach Java konvertiert. Für jede Trace wird ein unbenutzer Name für einen Phasen-Event-Automaten erzeugt und die Methode `compile` der Klasse `Trace2PEACompiler` aufgerufen. Diese erzeugt die `PhaseEventAutomata` Objekte, die der MCTrace entspechen. Alle PEAs zu Traces eines Disjunktionsglieds werden mittels `convert` von `PEAJ2XMLConverter` von Java nach XML konvertiert. Für jedes Disjunktionsglied wird ein neuer Dateiname geschaffen und das PEANet mittels `write` von `XMLWriter` in die Datei geschrieben.

Normalformberechnung

Die Methode `compile` der Klasse `TestForm2MCFormCompiler` berechnet die model-checkbare Darstellung zu einer Testformel. Ihre Arbeitsweise ist in Abbildung 5.15 dargestellt und soll im Folgenden erläutert werden: Das XML-Dokument wird als Datei übergeben, die mit einem XML-Parser des Pakets org.apache.xerces [Xer] ausgelesen wird. Der `xerces`-Parser ist geeignet zu konfigurieren, damit er eine Validation der XML-Datei vornimmt. Der ausgelesene XML-Baum, der sich, wie in Abschnitt 5.2.1 beschrieben, nicht in Binärdarstellung befindet, wird mit `makeBinary` in einen Binärbaum umgewandelt. Mit `introduceSyncEvents` werden in der Formel auftretende Chop-Operatoren gegen geeignete Sync-Events ausgetauscht. Die Normalform wird mit `buildNF`, beschrieben in 5.2.3, berechnet. Abschließend werden die MCForm-Elemente in der Baumdarstellung der Formel gesucht. Dabei werden die Methoden `createMCForm` und `createMCTrace`, wie in 5.2.3 erläutert, verwandt. Zwischenergebnisse der Berechnung werden in Dateien geschrieben. So können einzelne Schritte nachvollzogen, aber auch die Zwischenergebnisse weiterbenutzt werden.

Testautomatenberechnung

Die Testautomatensemantik zu einer gegebenen Trace der Form in Zeile 3.26, Definition 3.3.2, wird mit der Methode `compile` der Klasse `Trace2PEACompiler` erzeugt. Die `compile` Methode ruft ihrerseits die `buildAut` Methode auf. Diese wurde aus `BuildCTA` übernommen und für die Kompilation von Testformeln abgeändert: Zunächst werden Teilformeln vorberechnet, die während der Potenzmengenkonstruktion benötigt werden. Die Potenzmengenkonstruktion ist durch die `loop`-Schleife angedeutet. Die Methode `recursiveBuildTrans` ist so abgewandelt, dass die Potenzmenge vollständig aufgebaut wird. Mit `addForbiddenEvents` werden im Potenzmengenautomaten alle Sync-Events verboten. Anschließend werden die Transitionen und der Bad-State mit der Methode `buildExitSyncTransitions` eingefügt. Dabei wird auf `buildExitGuard` zurückgegriffen. Falls die Trace ein Entry-Sync-Event erfordert, werden der neue Startzustand zusammen mit den notwendigen Kanten durch `buildEntrySyncTransitions` erzeugt.

5.3 Implementierung, Dokumentation und Test

Arbeitsumgebung

Die Entwicklung des Compilers wurde auf einem *SuSE LINUX 9.2 Professional* System durchgeführt. Der Entwurf entstand mit einer Evaluationslizenz des Tools *Poseidon for UML Professional Edition 3.1* der Firma *Gentleware*. Die Implementierung wurde mit dem *Java Development Kit* (JDK) der Firma *Sun* in der Version 1.4.2 mit der Entwicklungsumgebung *Eclipse* in der Version 3.0.2 durchgeführt. Zum Erstellen der XML-Schemata sowie der XML-Dateien für Tests und für die Fallstudie wurde das Eclipse Plugin <oXygen/> XML als Testlizenz in den Versionen 5.1.0, 6.0, 6.1.0 genutzt. Innerhalb des Compilers wurde das Tool *Log4j* [Log] in der Version 1.2.9 aus dem Paket `org.apache.log4j` eingesetzt. Außerdem wurde das Paket `org.apache.xerces` benutzt, um auf den XML-Parser *Xerces2* der Version 2.6.2 zugreifen zu können, [Xer].

Implementerungsrichtlinien

Das entwickelte Paket `pea.modelchecking` wird im Rahmen des AVACS-Projektes zur Verifikation von Fallstudien eingesetzt. Um die von *Moby/PEA* geforderten Funktionalitäten bereitstellen zu können, wurde das System im Entwurf in mehrere Module zerlegt, die keine, oder eine klar definierte Kopplung mit anderen Systemkomponenten besitzen. In der Implementierung wurde darauf geachtet, dass keinerlei zusätzlich Abhängigkeiten erzeugt oder unerwünschte Seiteneffekte von den Komponenten hervorgerufen wurden. Um Seiteneffekte bei der Nutzung von Komponenten des Pakets durch andere Tools ausschließen zu können, wurden in den Klassen die Attribute und Methoden konsequent mit `private` oder `protected` qualifiziert. Die angeführten Maßnahmen erlauben es Außenstehenden nach einer geringen Einarbeitungszeit, den Compiler zu warten oder weiterzuentwickeln und an neue Aufgaben anzupassen.

Model-Checking birgt neben den Problemen der Spezifikation, gefordertes Systemverhalten durch eine geeignete Formel auszudrücken, das Problem, die Spezifikation in einem für den Model-Checker geeigneten Format auszudrücken. Der Compiler des Paketes `pea.modelchecking` nimmt als Eingabeformat Dateien entgegen, die dem XML-Schema TestForm.xsd entsprechen. Neben den XML-Schema Mechanismen zur Validation, die durch den `xerces`-Parser angewandt werden, werden durch die konsequente Verwendung von Exceptions in allen Klassen des Paketes `pea.modelchecking` Plausibilitätsprüfungen durchgeführt und eine hohe Typsicherheit gewährleistet. Im Falle von Fehleingaben gibt der Compiler umfassende Fehlermeldungen aus, die den Benutzer über die Art des Fehlers aufklären und, wenn möglich, Wege aufzeigen, diesen Fehler zu beheben.

Da die Spezifikation einer Systemanforderung ein durchaus schwieriges Problem darstellt, kommt es vor, dass das Ergebnis der Übersetzung eine Menge von Automaten ist, die nicht der Erwartung des Benutzers entspricht. Weil die Kompilierung in einer Vielzahl von Schritten durchgeführt wird, ist neben der Sichtung der vom Compiler generierten Dokumente ein Dokument hilfreich, welches den Kompilationsprozess nachvollziehbar macht. In dem implementierten Compiler verwendet jede Komponente einen Log4j-Logger [Log], ein Tool, welches das Systemverhalten dokumentiert. Mit einer Konfigurationsdatei, die der Benutzer beim Aufruf des Compilers angeben kann, wird bestimmt, wohin die Informationen des Loggers geschrieben werden sollen. Standardmäßig stehen die Console oder Log-Dateien zur Verfügung. Ferner sind die Nachrichten nach Stufen geordnet und es kann in der Konfigurationsdatei angegeben werden, ab welcher Stufe Nachrichten auszugeben sind. Ist die Stufe `DEBUG` gewählt, so werden viele Informationen über einzelne Arbeitsschritte der Komponenten des Compilers ausgegeben, während `WARN` nur Ausnahmezustände einer Komponente dokumentiert. Für jede Komponente können Ziel und Stufe der Ausgabe separat spezifiziert werden. Die Nachvollziehbarkeit des Systemverhaltens ist insbesondere auch in Weiterentwicklungsphasen sinnvoll, da fehlerhaftes Verhalten entsprechend dokumentiert wird.

Dokumentation

Zu den Paketen `pea.modelchecking` aber auch zur Klasse `pea.Trace2PEACompiler` ist eine API-Dokumentation in HTML mit `javadoc` erstellt worden. Dabei sind nicht nur die von außen zugreifbaren Elemente dokumentiert worden, sondern auch interne Methoden einer Klasse. Auf diese Weise ist es für Außenstehende leichter, die Funktionsweise der Komponenten nachzuvollziehen. Für die genutzen XML-Schemata sind mit dem <oXygen/> Plugin automatisch Schema-Dokumentationen erstellt worden, welche die verwandten Schemata visualisieren. Elemente sind mit der Dokumentation ihres Typs verlinkt.

Tests

Das Vorgehen beim Testen des Compiler ist durch die Beschreibung des Testprozesses in [Som01] inspiriert. Die einzelnen Komponenten des Systems sind – sofern möglich – unabhängig voneinander getestet. Dabei kamen Fehlertests zur Anwendung: Zunächst wurden für die einzelnen Komponenten Testfälle entwickelt und anschließend zu den Testfällen passende Testdaten erstellt. Bei der Erarbeitung möglicher Testfälle ist nach dem Prinzip des "Black-Box-Testens" vorgegangen worden: Die Kenntnis über die Funktionsweise eines Systemteils floß nicht in die Erstellung der Testfälle ein. Allein anhand des geforderten Ein- und Ausgabe-Verhaltens einer Klasse wurden Testdaten erstellt. Mit den Daten ist die entsprechende Systemfunktionalität aufgerufen und das Ergebnis der Ausführung der Komponente mit dem zu erwartenden Ergebnis verglichen worden, um fehlerhaftes Verhalten aufzudecken.
Als hilfreich hat sich die Verwendung der eigenen Dateiformate für jeden Schritt der Kompilierung erwiesen. Zunächst konnte der `TestForm2MCFormCompiler` getestet werden. Durch das eigene Format ModelCheckForm.xsd war es möglich, unabhängig vom Übersetzungsergebnis des `TestForm2MCFormCompiler`s den `MCTraceXML2JConverter` zu testen. Die model-checkbare Darstellung einiger Formeln ließ sich zum Testen von Hand generieren. Die erstellten Dateien wurden dem `MCTraceXML2JConverter` zugeführt. Wie auch der `PEAXML2JConverter` war diese Klasse jedoch von der korrekten Funktionsweise des `FormulaXML2JConverters` abhängig, die wiederum in getrennten Tests, unabhängig von der übrigen Systemfunktionalität, nachgewiesen werden konnte. Mit statisch generierten Systemdaten wurde der `Trace2PEACompiler` aufgerufen und so unabhängig vom übrigen System getestet. Ebenso wurde mit dem `PEAJ2XMLConverter` verfahren.

Nach den Tests erfolgte eine Bottom-up-Integration des Systems: Zunächst wurden `TestForm2MCFormCompiler` und `MCTraceXML2JConverter` gekoppelt und das Ergebnis der Hintereinanderausführung ausgegeben. Durch das Zusammenfügen der Klassen `Trace2PEACompiler`, `PEAJ2XMLConverter` und `XMLWriter` ließ sich auch deren Funktionsweise überprüfen. Abschließend wurden alle Bestandteile integriert, die geforderte Funktionalität konnte problemlos erreicht werden.

5.4 Weiterentwicklung

Für die Weiterentwicklung des Compilers ist die Verwendung interpretierter Terme und Formeln über getypten Variablen sinnvoll. Der Inhalt einer BooleanExpression könnte dazu als Formel aufgefasst werden. Die Werte der Expression werden bisher einzig auf

Gleichheit geprüft, würden jedoch interpretierte Formeln erlaubt, so könnten auch nicht `equals` identische Ausdrücke als äquivalent nachgewiesen werden. Auf diese Weise kann der Zustandsraum der PEAs eingeschränkt werden, indem frühzeitig erkannt wird, dass sich BooleanExpressions ausschließen. Getypte Variablen schaffen Typsicherheit bei der Verwendung von Variablen in Formeln. Zur Compile-Zeit kann eine syntaktische Überprüfung stattfinden, ob Variablen in dem gegebenen Kontext korrekt genutzt wurden. Ferner wird der Zustandsraum weiter eingeschränkt, weil Formeln wie $TP = apr$, wobei TP eine Variable, apr ihr Wert seien, die typisch in Zustandsausdrücken sind, andere Formeln wie $TP = crs$, wobei crs ein Datenwert ungleich apr sei, ausschließen.
Nicht nur im Hinblick auf Formeln, die in Guards und Invarianten genutzt werden, stellt die Verwendung von Variablen eine sinnvolle Erweiterung der Funktionalität dar. Auch bei der Verwendung von Events scheinen Parameter in Form von getypten Variablen wünschenswert. Die Spezifikation eines Events $sendMessage(message)$ erfordert in der aktuellen Version des Compilers die Einführung eines Events für jeden Wert, den die Variable *message* besitzen kann. In Kapitel 6 zeigt sich, wie ohne Parameter zum einen die Anzahl der PEAs in einem System ansteigt und außerdem PEAs entstehen, die Teilgraphen mit bis auf den Parameter von Events identischem Verhalten besitzen.

Formeln sind in PEAs als Bäume repräsentiert. Allerdings ist die Repräsentation einzig innerhalb der Tools *Moby/PEA* und `pea.modelchecking` einheitlich. Transition-Constraint-Systems (TCS) und *ARMC* benutzen zur Zeit nicht die hier vorgestellte Darstellung. Es scheint im Rahmen des *AVACS* Kontextes erstrebenswert, ein einheitliches Format zu entwickeln, welches die oben genannten Funktionalitäten unterstützt. Neben einer einheitlichen Darstellung von Formeln ist eine einheitliche Repräsentation für Phasen-Event-Automaten und Transition-Constraint-Systems anzustreben. Die Formate sollten von Weiterentwicklungen des *ARMC* Model-Checkers gelesen werden können. Auf diese Weise wird die fehleranfällige Konvertierung zwischen PEA und TCS Darstellungen sowie TCS und *ARMC* lesbaren Darstellungen obsolet.

Es ist in Lemma 3.1.2 aufgezeigt worden, wie Testformeln mit DC Phasen, die von einem leeren Intervall erfüllt werden, in Testautomaten übersetzt werden können. Diese geschilderte Übersetzung lässt die Anzahl der Testautomaten jedoch in der Anzahl der $true$-Phasen exponentiell wachsen. In [Hoe05a] ist aufgezeigt, wie sich direkt Automaten zu Countertrace-Formeln erzeugen lassen, die $true$-Phasen besitzen. Die vorgestellte Testformelsemantik sollte um die Ansätze in [Hoe05a] erweitert werden, so dass sich Testformeln mit $true$-Phasen direkt übersetzen lassen. In Weiterentwicklungen des Compilers sollte die erweiterte Testautomatensemantik implementiert werden. Mit der Verwendung des allowEmpty Attributs in der Definition von Phase Elementen sieht das XML-Schema BasicTypes.xsd bereits die Eingabe der erweiterten Testformeln vor, einzig der Compiler `Trace2PEACompiler` ist anzupassen.
Auch sollte die Eingabe von Traces mit exakten Längen, wie sie in Lemma 3.1.1 vorgestellt worden sind, implementiert werden. Es ist zwar prinzipiell möglich, beide Ungleichungen von Hand einzugeben, dennoch ist dieser Vorgang eine potentielle Fehlerquelle und erfordert Arbeit, die dem Benutzer abgenommen werden kann. Daher sollte der

Vorgang automatisiert werden.
Ist ein model-checking Verfahren für Formeln mit Liveness-Spezifikationen entwickelt worden, ist eine entsprechende Übersetzung zu implementieren.

Die Eingabe von Testformeln erfolgt zur Zeit getrennt von *Moby/PEA*, indem XML-Dateien von Hand generiert werden. Mit einem XML-Tool kann diese Arbeit durchaus zügig und komfortabel erledigt werden. Eine erhöhte Benutzerfreundlichkeit würde jedoch erreicht, wenn alle Eingaben allein in *Moby/PEA* durchgeführt werden könnten und *Moby/PEA* die Testformel-Datei automatisch generierte. Mit einer geeigneten graphischen Oberfläche wäre es sogar möglich, die Testformel *zusammenzuklicken* und den Benutzer nur an dedizierten Stellen Werte eingeben zu lassen. So werden die Möglichkeiten für Fehleingaben durch den Benutzer weiter eingeschränkt.

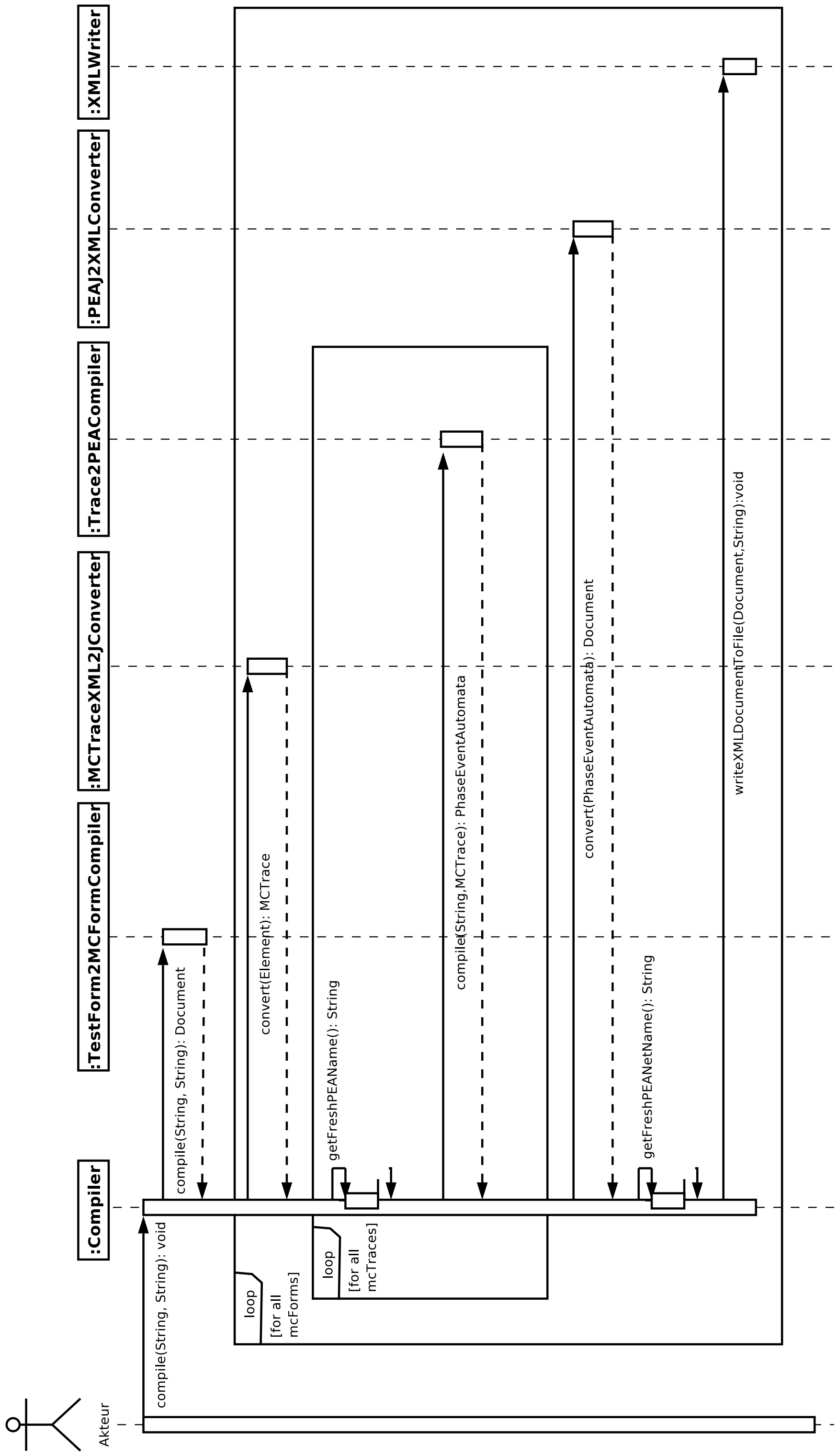

Abbildung 5.14: UML 2.0 Sequenzdiagramm: Kompilationsvorgang

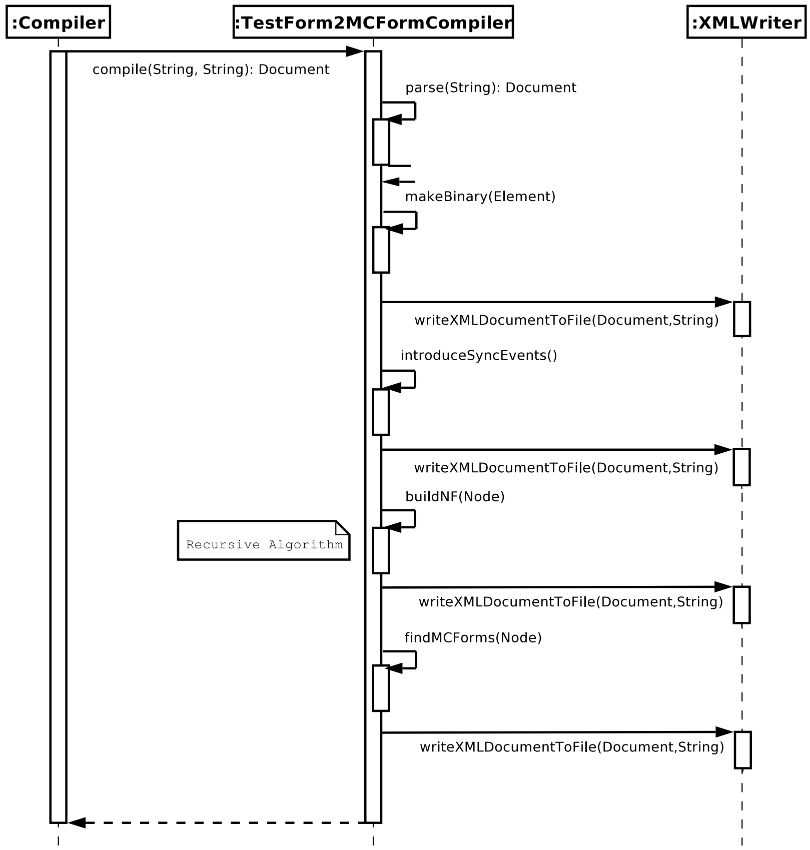

Abbildung 5.15: UML 2.0 Sequenzdiagramm: Berechnung der Normalform

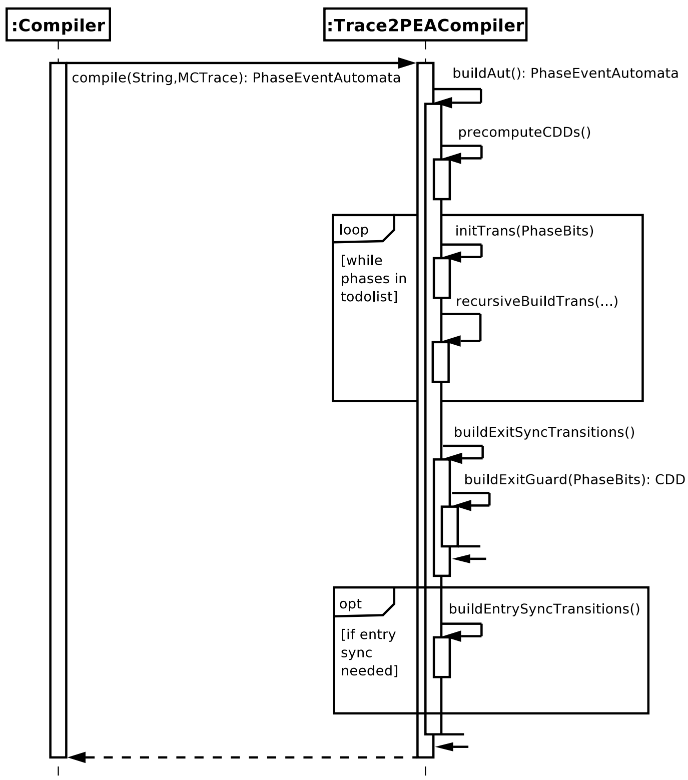

Abbildung 5.16: UML 2.0 Sequenzdiagramm: Berechnung der Testautomatensemantik

6 Fallstudie: Treatment of Emergency Messages

Die Fallstudie *Treatment of Emergency Messages* des *AVACS* Projektes erstellt ein Formales Modell für Teile des *European Train Control System*s (ETCS) zur Steuerung des Schienenverkehrs. Auf einer Schienenstrecke befinden sich mehrere aufeinander folgende Züge. Sie sind über ein Kommunikationsnetzwerk mit einer zentralen Steuerung, dem sogenannten *Radio Block Center* (RBC), verbunden. Der RBC weist den Zügen Positionen auf der Strecke zu, bis zu denen sie fahren dürfen. Idealerweise liegen diese Position unmittelbar hinter dem Ende eines vorausfahrenden Zuges, um eine hohe Auslastung der Strecke zu gewährleisten. Tritt bei einem der führenden Züge ein Notfall ein, so werden über das Netzwerk Warnmeldungen versandt, welche die Züge zum Stehenbleiben veranlassen. Mit Model-Checking Methoden wird nachgewiesen, dass, werden die gegebenen Realzeitbedingungen eingehalten, keine Auffahrunfälle passieren können. In [Fab05] ist ein formales Modell der Fallstudie in CSP-OZ-DC gegeben.
Anhand eines stark vereinfachten Teils des Modells sollen in diesem Kapitel Verifikationsergebnisse mit dem in der Diplomarbeit entwickelten Model-Checking Verfahren erzielt werden. Es wird der Model-Checker *Uppaal* genutzt, da die Anbindung von *Moby/PEA* an *ARMC* noch nicht zuverlässig arbeitet.

6.1 Modell

In der *AVACS* Fallstudie ist die Anzahl der Züge nicht nach oben beschränkt. In dieser Diplomarbeit wird die Verifikation jedoch nur für den Fall zweier aufeinander folgender Züge durchgeführt. Desweiteren werden einzig Übertragungszeiten für Notfallnachrichten und Reaktionszeiten beim Bremsen der Züge berücksichtigt. In der AVACS-Fallstudie sind darüberhinaus Fahrer, Position und Antwortnachrichten modelliert.

Nach einem Überblick über die Situation der Fallstudie werden die einzelnen Komponenten des System als PEA Modelle vorgestellt.

6.1.1 Überblick

Die Situation in der Fallstudie sei durch die Abbildung 6.1, übernommen aus [Fab05], dargestellt: Zwei Hochgeschwindigkeitszüge folgen einander auf einer Schienenstrecke. Im führenden Zug wird eine Notfallsituation entdeckt, was den Zug veranlaßt, ein Alarmsignal über das Kommunikationsnetzwerk an den RBC zu senden. Der RBC generiert

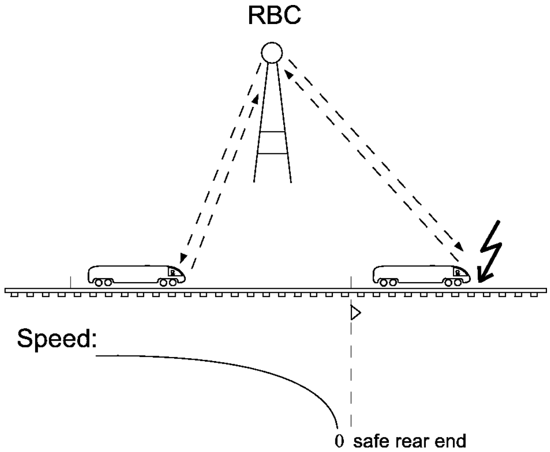

Abbildung 6.1: Komponenten der Fallstudie, [Fab05]

bei Empfang des Alarms Warnungen für beide Züge, die über das Kommunikationsnetzwerk verschickt werden. Bei Empfang der Warnung bremst der entsprechende Zug. Die Bremskurve ist unter dem Zug angedeutet. Der Zug hält, bevor er das Ende des voranfahrenden Zuges erreicht hat. Für das Versenden von Nachrichten über das Netzwerk stehen fünf Zeiteinheiten[1] zur Verfügung. Ebenfalls fünf Zeiteinheiten vergehen maximal bis zum Versand einer Warnung nach Eingang des Alarms. Weitere fünf Zeiteinheiten werden von den Zügen maximal benötigt, um nach Empfang einer Warnung die Bremsen betätigt zu haben.

Initial befindet sich das System in dem Zustand, dass keiner der Züge bremst und der führende Zug keine Notfallsituation entdeckt hat. Mit einem Umgebungsprozess wird in der vorliegenden Version der Fallstudie ausgeschlossen, dass mehr als eine Notfallsituation entdeckt wird.

6.1.2 Komponenten

Die zu den Komponenten des Systems angegebenen Phasen-Event-Automaten stellen die PEA Semantik für CSP-Prozesse und Countertrace-Formeln dar. Aus diesem Grund spezifiziert ein mit CSP indizierter Automat ausschließlich Verhalten, während ein mit DC indizierter Automat nur Realzeitbedingungen ausdrückt. Um die Nutzbarkeit von Datenwerten beim Model-Checking aufzeigen zu können, sind den CSP indizierten Automaten der Züge Variablen hinzugefügt worden.

Führender Zug

In der *AVACS* Fallstudie ist das Verhalten eines Zuges durch zwei parallele CSP-Prozesse ausgedrückt. Der erste Prozess namens *Running* schildert das Verhalten des Zuges im

[1] entsprechend 500ms in der *AVACS* Fallstudie

Betrieb: Entweder werden die Position ausgelesen und das Fahrverhalten angepasst, oder es wird eine Notfallsituation entdeckt. Der zweite Prozess, genannt *HandleEM*, ist bereit, eine Warnung zu empfangen und eine Bremsung einzuleiten.

Das Verhalten des führenden Zuges, modelliert durch den PEA $TrainEM_{CSP}$ in Ab-

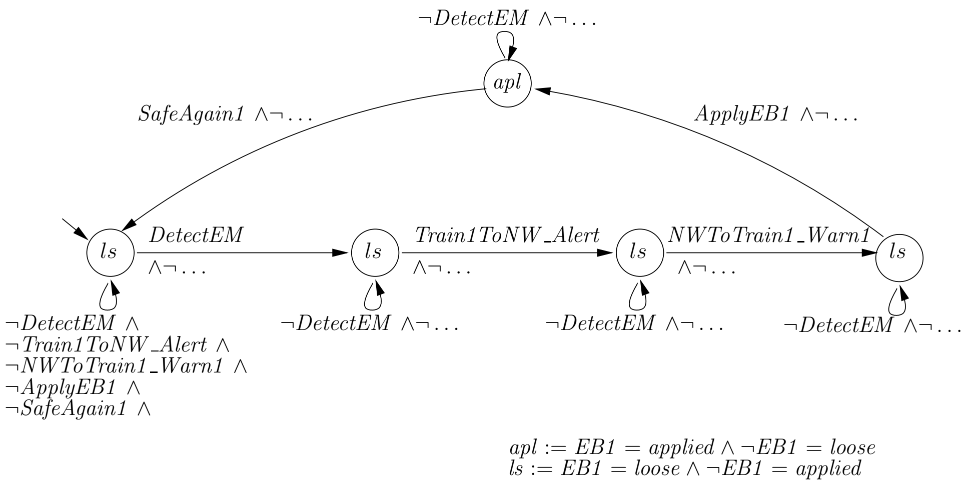

Abbildung 6.2: Phasen-Event-Automat $TrainEM_{CSP}$

bildung 6.2, entspricht dem CSP-Teil zum Entdecken einer Notfallsituation. Zunächst wird das Auftreten einer Notfallsituation über das Event *DetectEM* ausgedrückt. Anschließend sendet der Zug über *TrainToNW_Alert* die Alarmnachricht an das Netzwerk. Der Zug wartet auf die folgende Warnung vom Netzwerk mittels *NWToTrain1_Warn1* und bremst mittels *ApplyEB1*. Nach der Bremsung kann das System mit *SafeAgain1* wieder in den Ausgangszustand übergehen. An den Stotterkanten sind alle Events des Automaten verboten, damit das Verhalten in der spezifizierten Reihenfolge abläuft. Die Zustandsinvarianten besagen, dass die Bremse nach Empfang der Nachricht *ApplyEB1* angezogen, ansonsten gelöst ist.

Die Realzeitanforderung lautet, dass nach einer Warnung vom Netzwerk nicht mehr als fünf Zeiteinheiten vergehen dürfen, bis die Bremsen angezogen werden. Diese Forderung wird in der *AVACS* Fallstudie als Countertrace-Formel $\neg\Diamond(\updownarrow \mathit{NWToTrain1\_Warn1} ; \boxminus \mathit{ApplyEB1} \wedge \ell \geq 5)$ ausgedrückt. In dem hier vorliegenden Modell drückt der PEA $TrainEM_{DC}$ aus Abbildung 6.3 das gewünschte Verhalten aus. Er stellt gerade die PEA Semantik der Formel dar. Der Automat geht aus der Potenzmengenkonstruktion der nicht negierten Formel hervor, indem alle Zustände, in denen die Formel vollständig erkannt wurde, entfernt werden. Solange das Event *NWToTrain1_Warn1* nicht eingetreten ist, verweilt der Automat in seinem Startzustand. Nach Auftreten des Events *NWToTrain1_Warn1* geht der Automat in einen Zustand mit Invariante $c_2 < 5$ über, wobei c_2 die Uhr des Automaten ist. Es darf weniger als fünf Zeiteinheiten kein *Apply_EB1* Event kommuniziert werden, spätestens nach dieser Dauer wird ein solches Event vom Automaten gefordert.

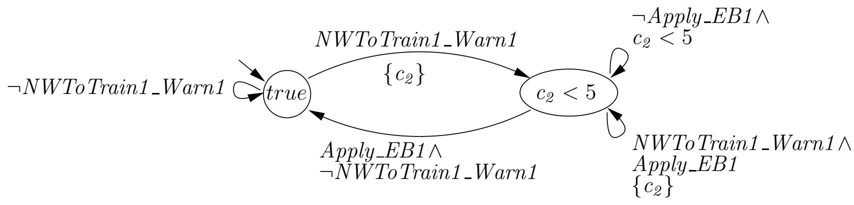

Abbildung 6.3: Phasen-Event-Automat $TrainEM_{DC}$

Folgender Zug

Das Verhalten des folgenen Zuges entspricht dem Prozess zum Entgegennehmen von Warnungen. Es ist durch den PEA $TrainReact_{CSP}$ in Abbildung 6.4 bestimmt. Der Zug

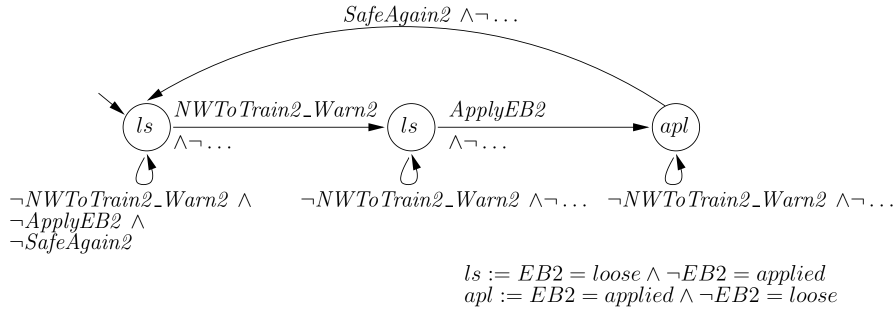

Abbildung 6.4: Phasen-Event-Automat $TrainReact_{CSP}$

erwartet ein *NWToTrain2_Warn2* Event. Tritt ein solches Event ein, bremst der Zug, angedeutet durch *ApplyEB2*, und gibt abschließend mit einem *SafeAgain2* Event an, dass die Notfallsituation überwunden ist.
Neben dem Automaten, der das Verhalten des Zuges spezifiziert, gibt es einen Automaten für die Realzeitanforderung. Der Zug muss spätestens fünf Zeiteinheiten nach Erhalt der Warnmeldung bremsen. Der Automat $TrainReact_{DC}$ entspricht dem Automaten $TrainEM_{DC}$, einzig die Indizes in den Events sind abgeändert.

Kommunikationsnetzwerk

Das Verhalten des Kommunikationsnetzwerks ist durch den Automaten $ComNW_{CSP}$, dargestellt in Abbildung 6.5, gegeben. Der Automat wartet auf eine Kommunikation des Alarm-Events *Train1ToNW_Alert* durch den ersten Zug. Dieses Event wird mittels *NWToRBC_Alert* an den RBC übergeben. Der RBC sendet seinerseits mit *RBCToNW_Warni*, $i \in \{1, 2\}$, Warnungen aus, die mit *NWToTraini_Warni* durch das Netzwerk ausgeliefert werden. Das Verhalten in der Auslieferung der *Warn*-Nachrichten unterscheidet sich nicht, es wird eine Nachricht vom RBC entgegengenommen und an einen Zug weitergeleitet. Mit der Verwendung parametrisierter Events hätte der Automat $ComNW_{CSP}$ um einen Zustand verkleinert werden können. In der *AVACS* Fallstudie

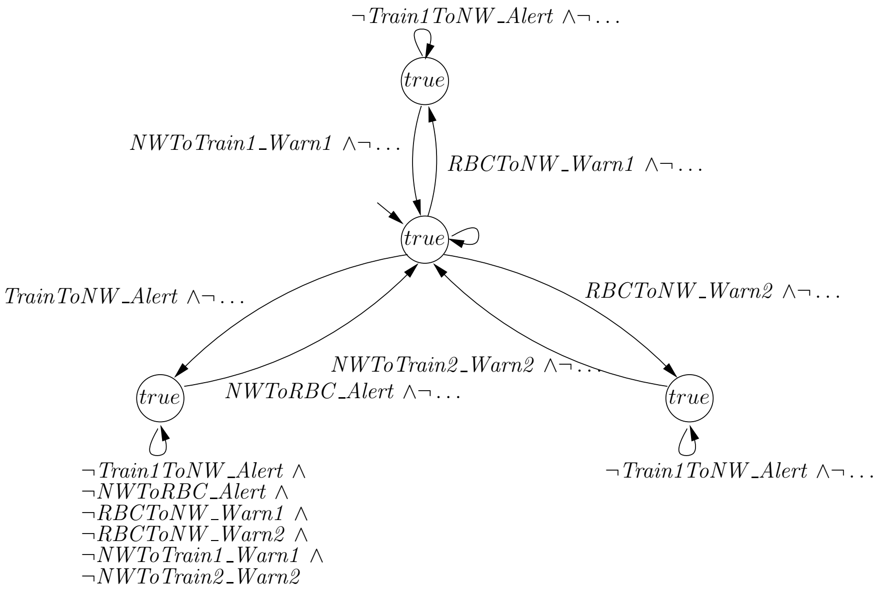

Abbildung 6.5: Phasen-Event-Automat $ComNW_{CSP}$

werden weitere Nachrichten neben *Alert* und *Warn* versandt, so dass parametrisierte Events für die Verständlichkeit des Modells unerlässlich sind.
Neben dem Verhalten sind für das Kommunikationsnetzwerk in der *AVACS* Fallstudie drei Realzeiteigenschaften in Form von Countertrace-Formeln spezifiziert, die in dieser Darstellung der Fallstudie in ihrer Semantik als PEAs gegeben sind. Mit der Formel $\neg\Diamond(\updownarrow \mathit{Train1ToNW\_Alert} ; \boxminus \mathit{NWToRBC\_Alert} \wedge \ell \geq 5)$ wird ausgedrückt, dass das Weiterleiten einer Alarm-Nachricht an den RBC nicht länger als fünf Zeiteinheiten dauert. Der Automat $ComNW_{DC_1}$ entspricht dem Automaten $TrainEM_{DC}$, wenn man die *NWToTrain1_Warn1*-Nachricht gegen *Train1ToNW_Alert* und *ApplyEB1* gegen *NWToRBC_Alert* austauscht. Ebenso gibt es zwei weitere Automaten $ComNW_{DC_2}$ und $ComNW_{DC_3}$, welche das rechtzeitige Ausliefern der anderen beiden Nachrichten sicherstellen.

RBC

Der RBC erwartet ein *NWToRBC_Alert*-Event. Nach Eintreten des Events werden die Warnungen *RBCToNW_Warn1* und *RBCToNW_Warn2*, ausgedrückt als Events, an das Netzwerk übergeben. Anschließend befindet sich der RBC in seinem Ausgangszustand. Das beschriebene Verhalten wird durch den Phasen-Event-Automaten RBC_{CSP} in Abbildung 6.6 erreicht.
Die Realzeiteigenschaft des RBCs lautet: nach Empfang der Alarm-Nachricht vergehen bis zum Senden der zweiten Warnung weniger als fünf Zeiteinheiten. Der Automat RBC_{DC} entspricht bis auf Eventnamen $Train1_{DC}$ aus Abbildung 6.3.

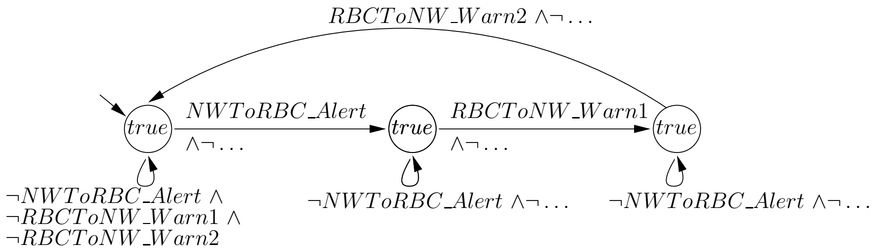

Abbildung 6.6: Phasen-Event-Automat RBC_{CSP}

Umgebung

Der Automat *Environment*, der das Verhalten der Umgebung modelliert, erlaubt ein einmaliges Auftreten einer Notfallsituation und damit ein einmaliges Auftreten des Events *DetectEM*. Er ist in Abbildung 6.7 dargestellt.

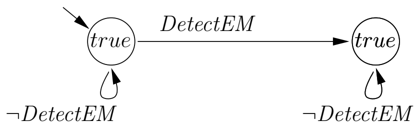

Abbildung 6.7: Phasen-Event-Automat *Environment*

6.2 Spezifikation

Es werden zwei Eigenschaften vorgestellt, die für das System nachgewiesen werden können. Mit der ersten Eigenschaft soll exemplarisch die Berechnung der model-checkbaren Darstellung durch den Compiler aufgezeigt werden. Die zweite Testformel erläutert die Konstruktion eines Testautomaten.
Zur Definition der ersten Testformel seien folgende Formeln gegeben:

$$\begin{aligned}
\mathit{Warn} :=& \lceil \mathit{true} \rceil \,;\, \updownarrow \mathit{Train1ToNW_Alert} \,;\, \lceil \mathit{true} \rceil ; \\
& \updownarrow \mathit{NWToTrain1_Warn1} \,;\, \lceil \mathit{true} \rceil \,;\, \updownarrow \mathit{NWToTrain2_Warn2} \\
\mathit{NoBrake}_1 :=& \boxminus \mathit{ApplyEB1} \wedge \ell > 5 \\
\mathit{NoBrake}_2 :=& \boxminus \mathit{ApplyEB2} \wedge \ell > 5
\end{aligned}$$

Die Testformel TF_1 ist definiert durch

$$\mathit{TF}_1 := \mathit{Warn} \,;\, (\mathit{NoBrake}_1 \wedge \mathit{NoBrake}_2)\,.$$

Die Eigenschaft stellt folgendes Verhalten dar: Zunächst wird ein Alarm vom führenden Zug ausgelöst. Dieser wird korrekt weitergeleitet und es werden Warnungen an beide

Züge ausgegeben. Dennoch dauert es bei beiden Zügen länger als fünf Zeiteinheiten, bis die Bremsen angezogen werden.
Korollar 3.3.2 folgend wird die Formel TF_1 ; $\lceil true \rceil$ überprüft. Dabei wird die $\lceil true \rceil$-Phase an die Traces der Testformel gehängt, die kein Exit-Sync-Event besitzen. Dies sind gerade die letzten Teilformeln einer durch Chop getrennten Formel. Der Compiler nutzt diese Beobachtung, berechnet die Normalform einzig für TF_1 und fügt am Ende $\lceil true \rceil$-Phasen ein. Die Formel TF_1 wird wie folgt in Normalform überführt:

$$\begin{aligned} & Warn \,;\, (NoBrake_1 \wedge NoBrake_2) \\ \{\text{Sync-Intro}\} \Leftrightarrow & \exists \mathcal{S}_0 : Warn \underset{\mathcal{S}_0}{\Updownarrow} (NoBrake_1 \wedge NoBrake_2) \\ \{\text{Sync-Dist+}\} \Leftrightarrow & \exists \mathcal{S}_0 : Warn \underset{\mathcal{S}_0}{\Updownarrow} \lceil true \rceil \wedge \lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_1 \wedge \lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_2 \end{aligned}$$

Es werden $NoBrake_1$ und $NoBrake_2$ $\lceil true \rceil$-Phase mit einem gemeinsamen Sync-Event $\mathcal{S}_1$ angehängt, damit die Formel in model-checkbarer Darstellung vorliegt:

$$\begin{aligned} \exists \mathcal{S}_0 : \exists \mathcal{S}_1 : Warn \underset{\mathcal{S}_0}{\Updownarrow} \lceil true \rceil \wedge \\ \lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_1 \underset{\mathcal{S}_1}{\Updownarrow} \lceil true \rceil \wedge \lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_2 \underset{\mathcal{S}_1}{\Updownarrow} \lceil true \rceil \end{aligned}$$

Zu den drei Formeln werden vom Compiler Testautomaten generiert, die parallel mit dem System komponiert werden. Mit einem Model-Checker ist nachzuweisen, dass das Tupel der drei Endzustände in der Parallelkomposition des Systems mit den Testautomaten nicht erreichbar ist.

Die Konstruktion von Testautomaten soll anhand der Testformel

$$TF_2 := \lceil true \rceil \,;\, \updownarrow Train1ToNW\_Alert \,;\, (\ell > 20 \wedge \lceil EB2 = loose \rceil)$$

vorgestellt werden. Zunächst wird die Potenzmengenkonstruktion für die Formel durchgeführt. Der resultierende Automat entspricht bis auf den Zustand *FINAL_pea0* dem Automaten in Abbildung 6.8. Im Zustand $st0$ wird verweilt, solange das erwartete Event *Train1ToNW_Alert* nicht auftritt. Mit Auftreten dieses Events wird in den Zustand $st01w$ gewechselt, die Uhr $pea0\_X1$ wird zurückgesetzt. Der Name besagt, dass die Menge der aktiven Phasen die DC Phase 1 der Formel enthält. Die untere Schranke ist jedoch noch nicht erreicht. Daher befindet sich die DC Phase 1 in der Menge *wait*, angedeutet durch das w. Hat die Uhr $pea0\_X1$ den Wert 20 erreicht, so wird in den Zustand $st01$ gewechselt, d.h. die erste Phase befindet sich nicht mehr in der Wartemenge. Wird die Bedingung $EB2 = loose$ verletzt, so wird wieder in den Anfangszustand gewechselt.
Die model-checkbare Darstellung fügt der Formel *NoBrake* eine durch einen Sync-Event $\mathcal{S}_0$ getrennte $\lceil true \rceil$-Phase hinzu. Das Erreichen dieser $\lceil true \rceil$-Phase entspricht im Testautomaten dem Erreichen des Bad-State, genannt *FINAL_pea0*. Nach der Testautomatensemantik in Abschnitt 4.4 wird von jedem Zustand zum Bad-State eine Transition eingefügt, die mit einem Guard und dem Sync-Event beschriftet ist. Für $st0$ und $st01w$

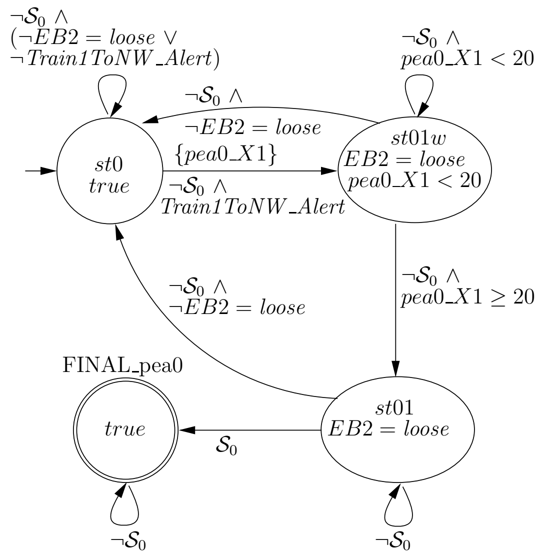

Abbildung 6.8: Testautomat $\mathcal{P}(TF_2)$

ist der Guard jedoch äquivalent zu *false*, die Transition wird daher weggelassen. Im Zustand *st*01*w* beispielsweise liefern die Hilfsfunktionen von *guard in* und *less* den Wert *true*. Die Funktion *wait* ermittelt jedoch, dass die DC Phase 1 in der Menge der wartenden Phasen enthalten ist und keinen $\geq$ Bound besitzt. Daher liefert *wait* den Wert *false*. Für den Zustand *st*01 ist der Guard äquivalent zu *true*. Mit dem Hiding wird das Event $\mathcal{S}_0$ an jeder weiteren Kante verboten.

6.3 Model-Checking mit Uppaal

Die Anbindung von *Moby/PEA* an den Model-Checker *ARMC* ist noch mit Fehlern behaftet. Um dennoch Verifikationsergebnisse für die Fallstudie erzielen zu können, ist das Tool *Uppaal* genutzt worden. Die von *Uppaal* verwendeten Realzeitautomaten unterscheiden sich in mehreren Punkten von Phasen-Event-Automaten:

- *Uppaal* unterstützt boolesche und Integervariablen. In Phasen-Event-Automaten dürfen beliebige endliche Datentypen verwandt werden.

- Phasen-Event-Automaten fordern, dass in Zuständen Zeit vergeht. In *Uppaal* gibt es *Delay Transitions*, die das Vergehen von Zeit in Zuständen simulieren. Es ist jedoch nicht zwingend erforderlich, dass Automaten eine Dauer größer Null in Zuständen verweilen.

- Die in *Uppal* genutzten Realzeitautomaten besitzen nur einen Anfangszustand, in PEAs gibt es eine Menge von Anfangszuständen.

Es sind in den Phasen-Event-Automaten der Fallstudie einzig Events und die Variablen EBi, $i \in \{1, 2\}$, mit dem Wertebereich $\{applied, loose\}$ genutzt worden. Während sich

Events unmittelbar durch *Uppaal*-Events ausdrücken ließen, könnten die Variablen durch boolesche Variablen ersetzt werden. Um Zeit vergehen zu lassen, könnte beim Betreten eines jeden Zustands der Automaten eine Uhr auf Null gesetzt werden, die das Vergehen von Zeit überprüft. Da das Modell aus elf Phasen-Event-Automaten besteht, würden auf diese Weise elf neue Uhren eingeführt, was den Zustandsraum stark vergrößern dürfte. Aus diesem Grund ist eine andere Herangehensweise zum Model-Checking mit *Uppaal* gewählt worden, die nachfolgend erläutert wird. Das Aktivitätsdiagramm in Abbildung

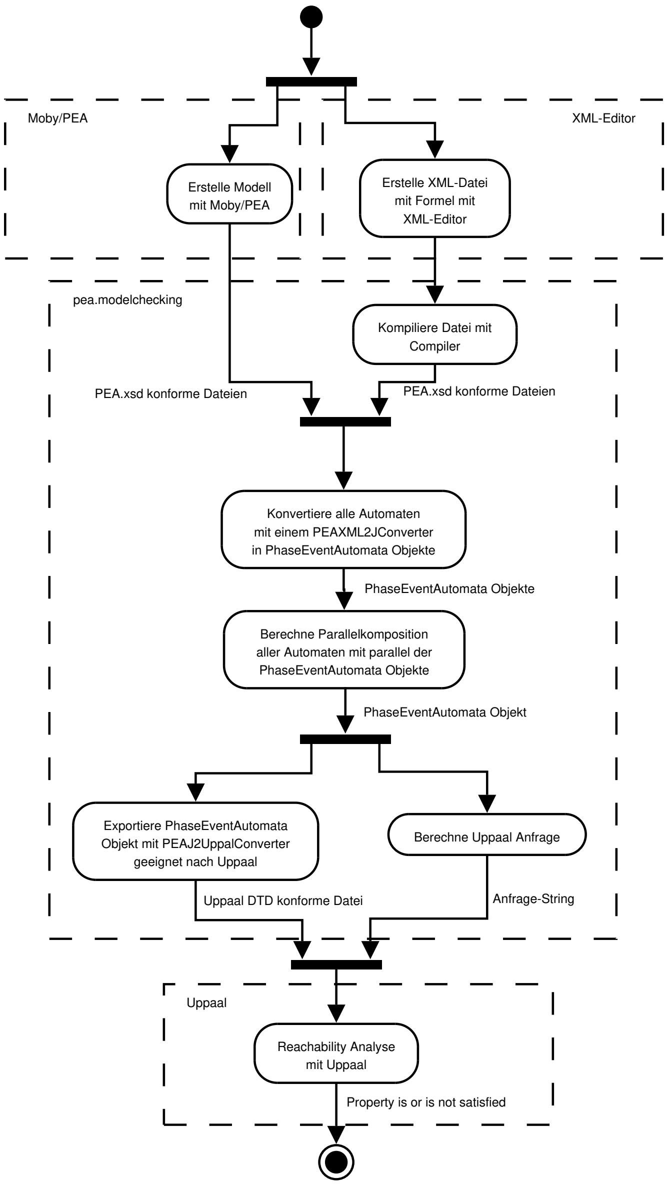

Abbildung 6.9: Vorgehen beim Model-Checking mit *Uppaal*

6.9 veranschaulicht die Erklärung. Die Kästen in der Graphik bezeichnen einzelne Tools, die benutzt wurden. Der Name der Tools ist in den oberen Ecken notiert. Die Pfeile sind mit den Ausgaben der vorangegangenen Aktivitäten beschriftet.

- Es werden alle PEAs des Modells sowie die Testautomaten der zu checkenden Eigenschaft mit einem `PEAXML2JConverter` eingelesen.
- Das Parallelprodukt aller Automaten wird über die `parallel` Methode der Klasse `PhaseEventAutomata` berechnet. Dabei werden nur die erreichbaren Zustände berechnet. Transitionen, deren Guard immer zu *false* ausgewertet wird, weil sie sich ausschließende Eventspezifikationen, Variablenbedingungen oder Uhrenbedingungen besitzen, werden weggelassen. Ebenso sind nur Zustände angeführt, deren Guard durch eine geeignet Variablenbelegung erfüllbar ist. Nach der Berechnung der Parallelkomposition sind alle Automaten auf gemeinsame Events synchronisiert[2], es sind nur die Guards und Zustandsinvarianten im System enthalten, die durch Variablenbelegungen erfüllt werden können.
- Mit einem `PEAJ2UppaalConverter` wird der entstandene Automat in ein Uppaal-lesbares XML-Format geschrieben. Dabei ersetzt *true* alle Events und alle Variablenbedingungen, da alle Automaten synchronisiert und die bestehenden Variablenbedingungen durch geeignete Belegungen erfüllbar sind. Zudem wird jeder Kante eine Bedingung $timer > 0$ sowie die Zuweisung $timer := 0$ hinzugefügt. Der Guard und das Zurücksetzen der Uhr erzwingen, dass in den Zuständen des Automaten Zeit vergehen muss.
- Der `PEAJ2UppaalConverter` benennt jeden der Zustände, die das Tupel der Endzustände der Testautomaten enthalten, in einen Zustand *Finali* um und generiert eine *Uppaal*-Anfrage `E<>(Final0||...||Final(Anzahl-1))`, wobei `Anzahl` der Anzahl der umbenannten Zustände entspricht.
- Das entstandene Modell kann in *Uppaal* geladen werden. Mit dem in *Uppaal* enthaltenen *Verifier* wird die vom `PEAJ2UppaalConverter` generierte Anfrage beantwortet. Die Aussage `Property is satisfied` bedeutet, es kann einer der gegebenen Endzustände des Systems erreicht werden. Die Aussage `Property is not satisfied` besagt, keiner der Endzustände des Systems ist erreichbar. Wenn der Testautomat die Semantik der Testformel *TF* darstellte und es nur einen Startzustand gibt, so bedeutet die Nicht-Erreichbarkeit nach dem Model-Checking Theorem $Ph \models \neg TF$, wobei *Ph* das gegebene Netz von Phasen-Event-Automaten ist.
- Besitzt die Parallelkomposition mehr als einen Startzustand, so ist das Modell mit jedem der möglichen Startzustände in *Uppaal* zu laden. Ist für alle Startzustände nachgewiesen, dass kein Endzustand erreicht werden kann, erfüllt der Automat die negierte Testformel.

Mit der Berechnung des Parallelproduktes vor dem Export der Automaten nach *Uppaal* wird die Anzahl an notwendigen *timer*-Uhren von elf auf eine Uhr reduziert.

[2] Der Grund ist die Konjunktion der Guards in der Parallelkomposition und die Struktur der Automaten: Es sind beim Beschreiten einer Kante alle Events außer dem aktuell geforderten verboten. Stotterkanten verbieten alle Events eines Automaten.

Bemerkung 6.3.1 (Anpassung der Semantik)
Es ist in dieser Arbeit auf einen Nachweis der Äquivalenz der Semantiken für PEAs und *Uppaal*-lesbare Automaten verzichtet worden. Realzeitautomaten in *Uppaal* besitzen als Semantik ein zeitbehaftetes Transitionssystem, *timed transition system* (TTS). Es ist offensichtlich, dass die Menge aller Runs eines PEA Ph, $\mathbf{Run}(Ph)$, genau die Elemente besitzt, die als Pfad in einem der TTS zu Ph enthalten sind. Die Menge aller TTS zu Ph besteht aus den TTS zu jedem der Realzeitautomaten, der Ph gleicht, aber weder Events noch Invarianten besitzt, nur einen der Startzustände hat und die Timerkonstruktion benutzt.

Der Operator `E<>` ist in der *Uppaal* Hilfe wie folgt erklärt:

> "The property E<> p evaluates to true for a timed transition system if and only if there is a sequence of alternating delay transitions and action transitions s0–>s1–>...–>sn, where s0 is the initial state and sn satisfies p."

Die Formel `E<> p` ist also in einem TTS genau dann erfüllt, wenn es einen Pfad in dem TTS gibt, über den ein Zustand erreicht wird, der `p` erfüllt. Mit der Forderung, dass sich die Menge aller Runs und die Menge der TTS entsprechen, wird auf diese Weise gerade die Erreichbarkeit eines Zustandes nach Definition 2.2.3 in einem Phasen-Event-Automaten ausgedrückt.

In Abbildung 6.10 ist der in *Uppaal* geladene Phasen-Event-Automat

$$TrainEM_{CSP} \parallel \ldots \parallel Environment \parallel TF_2$$

abgebildet. Abbildung 6.11 zeigt das in *Uppaal* enthaltene *Verifier*-Tool, mit dem zunächst die von `PEAJ2UppaalConverter` generierte Anfrage und anschließend die Deadlockfreiheit des Systems gecheckt wurden. Ein Deadlock wird in *Uppaal* über das Schlüsselwort `deadlock` spezifiziert.

6.4 Verifikationsergebnisse

Es konnte mittels Model-Checking nachgewiesen werden, dass das System $TrainEM_{CSP} \parallel \ldots \parallel Environment$ die Formeln

$$\neg TF_1 = \neg(Warn\,;(NoBrake_1 \wedge NoBrake_2))$$

sowie

$$\neg TF_2 = \neg(\lceil true \rceil\,; \updownarrow Train1ToNW\_Alert\,; (\ell > 20 \wedge \lceil EB2 = loose \rceil))$$

erfüllt, indem gezeigt wurde, dass keiner der für den ersten Fall neun, im zweiten Fall 18 möglichen Endzustände erreichbar ist. Ebenfalls wurde gezeigt, dass das System die

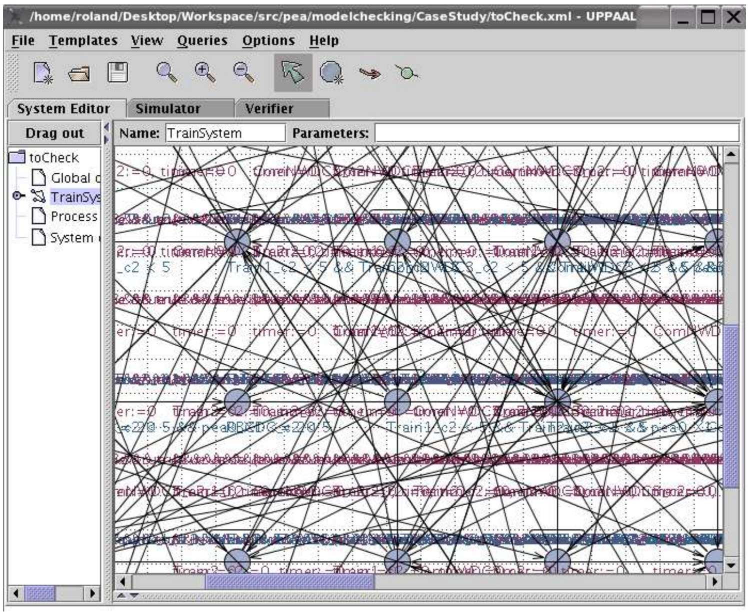

Abbildung 6.10: In *Uppaal* geladenes, parallel komponiertes System

Formel $\neg TF_1'$ nicht erfüllt. Die Formel TF_1' entspricht bis auf die Schrankenwerte TF_1, jedoch ist fünf durch vier ersetzt worden. Auch ist die Formel

$$\neg(\lceil true \rceil \,;\, \updownarrow Train1ToNW\_Alert \,;\, \lceil EB2 = loose \rceil \land \ell > 19)$$

nicht erfüllt. *Uppaal* liefert einen Pfad als Gegenbeispiel, in dem die Bremsen des zweiten Zuges länger als 19 Zeiteinheiten nicht angezogen waren. Mit geeigneter Benennung der Zustände ist es möglich, das Verhalten des parallel komponierten Systems anhand des Gegenbeispiels nachzuvollziehen.
Es wurden weitere Anfragen an das System gestellt. Es konnte nachgewiesen werden, dass das Weiterleiten aller Nachrichten in der geforderten Zeit durchgeführt wird und dass die Nachrichten in der erwarteten Reihenfolge auftreten. Für Testformeln ohne Realzeiteigenschaften, die sich nur auf die Reihenfolge der Nachrichten beziehen, wurde bereits beim Errechnen der Parallelkomposition ermittelt, dass keiner der Endzustände erreichbar ist. In diesen Fällen musste der Model-Checker nicht mehr aufgerufen werden.

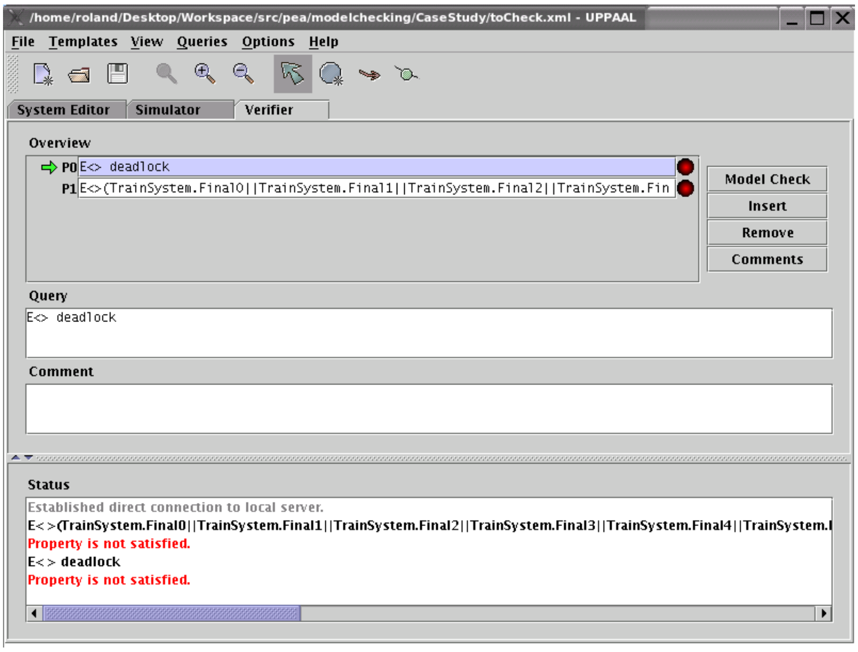

Abbildung 6.11: Model-Checking mit dem Tool *Verifier*

7 Zusammenfassung und Ausblick

Abschließend werden in diesem Kapitel die Ergebnisse der Diplomarbeit zusammengefasst. Es sind während der Erstellung dieser Arbeit weitere Fragen im Rahmen des Model-Checkings von Phasen-Event-Automaten bezüglich Duration Calculus Formeln aufgetreten, die als Ausblick auf mögliche weitere Arbeiten vorgestellt werden sollen.

Zusammenfassung

In dieser Diplomarbeit ist ein Model-Checking Verfahren zum automatisierten Nachweis von Eigenschaften von Phasen-Event-Automaten erarbeitet worden. Phasen-Event-Automaten sind ein neues Modell für Realzeitsysteme, das sowohl Daten- als auch Eventaspekte wiederspiegelt. Eigenschaften dieser Systeme werden in der Intervall-basierten Logik Duration Calculus spezifiziert, in der Begriffe wie Zustände, Dauern oder Events sehr natürlich definiert werden können.
Zunächst wurde die Formelklasse *Testform* der Testformeln identifiziert. Es konnte nachgewiesen werden, dass sich jede Formel dieser Klasse durch eine äquivalente Formel in einer gewissen, sogenannten Normalform darstellen lässt. Zum Nachweis dieser Tatsache wurde eine besondere Sorte von Events, Sync-Events, definiert. Sync-Events zeichnen sich dadurch aus, dass sie einmalig auftreten. Mit dem Begriff der Sync-Events ist es möglich, das Erreichen von Chop-Punkten auszudrücken, ohne auf Zeitdarstellungen zurückzugreifen. Mit dieser Idee konnte gezeigt werden, dass zwischen Sync-Events und der Konjunktion in Duration Calculus Formeln ein Distributivgesetz gilt, das zu einer deutlich kleineren Anzahl an Formeln führt als übliche Distributivität zwischen Operatoren. Nachdem gezeigt war, wie jede Formel in eine Normalform überführt werden kann, konnte darauf aufbauend nachgewiesen werden, dass jede Formel der Klasse *Testform* in gewissem Sinne erfüllbarkeitsäquivalent zu einer Formel in model-checkbarer Darstellung ist.
Im nachfolgenden Kapitel wurde eine semantische Anpassung zwischen Duration Calculus Formeln und Phasen-Event-Automaten vorgenommen. Es wurde erklärt, in welchem Zusammenhang Interpretationen einer Formel und Runs eines PEAs stehen, und, was es bedeutet, dass ein Phasen-Event-Automat eine Formel erfüllt.
Die Idee des Model-Checking Verfahrens ist es, zu einer Formel einen Phasen-Event-Automaten zu konstruieren, der mit dem System parallel komponiert wird und einen ausgezeichneten Zustand erreicht, wenn das System das in der Formel spezifizierte Verhalten zeigt. Zu diesem Zweck wurden Testautomaten, PEAs mit Endzustand, und Operatoren auf Testautomaten definiert, die eine geeignete Semantik für Testformeln in model-checkbarer Darstellung erlauben. Es wurde die Potenzmengenkonstruktion für

Trace-Formeln aus [Hoe05a] vorgestellt und basierend darauf eine Semantik für Testformeln in model-checkbarer Darstellung definiert. Nach einem Theorem über die Äquivalenz der Erfüllbarkeit einer Formel und der Erreichbarkeit des Endzustandes im Testautomaten konnte gezeigt werden, dass ein gegebener Phasen-Event-Automat die negierte Testformel genau dann nicht erfüllt, wenn einer der Endzustände in der parallelen Komposition von PEA und Testautomat erreichbar ist. Damit war die Korrektheit des Model-Checking Verfahrens nachgewiesen.

Im sich anschließenden Kapitel wurde das Java-Paket `pea.modelchecking` erarbeitet, das die Berechnung der model-checkbaren Darstellung für Testformeln und die Konstruktion der Testautomaten automatisiert. Es wurde aufgezeigt, wie das Paket im *AVACS* Projekt innerhalb der existierenden Tools `pea`, *Moby/PEA* und *ARMC* genutzt werden soll. Nach einer Erläuterung der definierten Austauschformate und der Funktionalität des Paketes wurde darauf eingegangen, welche Weiterentwicklungen des Tools im Rahmen des *AVACS* Projektes wünschenswert wären und inwiefern das Paket derart gestaltet worden ist, dass diese Änderungen problemlos einfließen können.

In einem abschließenden Kapitel ist die Anwendbarkeit des model-checking Verfahrens und des Paketes `pea.modelchecking` mit der Durchführung einer Fallstudie aufgezeigt worden. Dabei ist die im *AVACS* Projekt erarbeitete Fallstudie *Treatment of Emergency Messages* gewählt worden, das Modell wurde jedoch vereinfacht. Nachdem Testformeln für das System spezifiziert worden waren, wurde gezeigt, wie mit dem Tool *Uppaal* Model-Checking von Phasen-Event-Automaten vorgenommen werden kann. Zu diesem Zweck war ein prototypisches Tool `PEAJ2Uppaalconverter` geschrieben worden, welches PEAs von Java in die *Uppaal*-lesbare Darstellung umwandelt und die notwendige Anfrage generiert. Die gewünschten Model-Checking Ergebnisse konnten erzielt werden. Es wurde *Uppaal* als Tool gewählt, da der im *AVACS* Projekt genutzte Model-Checker *ARMC* die Überprüfung von PEAs noch nicht zufriedenstellend durchführt.

Ausblick

Zum Nachweis der Existenz einer Normalform für Testformeln sind Sync-Events eingeführt worden. Sync-Events bieten die Möglichkeit, Zeitpunkte zu charakterisieren und erlauben so eine gewisse Distributivität zwischen der Konjunktion und dem Chop-Operator, dargestellt durch ein eingeführtes Sync-Event. Es stellt sich die Frage, welche Formelklassen neben der vorgestellten mit der Einführung von Sync-Events eine Normalform besitzen. Der Nachweis einer Normaform für eine Formelklasse reduziert Fragen über Formeln der Klasse auf Formeln einer gewissen Gestalt. In weiteren Untersuchungen kann die Arbeit auf derartige Formeln fokussiert werden.
Mit der Beobachtung, dass Sync-Events Zeitpunkte wiederspiegeln, tritt die Frage auf, inwiefern sich Formeln mit dem ℓ-Operator durch äquivalente Formeln ausdrücken lassen, die Sync-Events aber nicht den ℓ-Operator nutzen. Eine Motivation für die Untersuchung dieser Frage stellt die Nähe der Sync-Events zu Automatenmodellen dar.

Während Events nahezu natürlich in Automatenmodellen enthalten sind, ist es unklar, inwiefern der ℓ-Operator auf Uhren, Zeitdauern oder ähnliches abgebildet wird. Um für noch größere Formelklassen operationelle Semantiken zu definieren, scheint daher eine weitere Erforschung von Sync-Events sinnvoll.
Es ist in dieser Diplomarbeit aufgezeigt worden, wie Model-Checking von Phasen-Event-Automaten für eine Teilklasse der Duration Calculus Formeln möglich ist. Eine Charakterisierung der Klasse von Duration Calculus Formeln, die model-checkbar sind, scheint ein schwieriges Problem zu sein. Insbesondere ist zunächst zu klären, inwiefern sich Liveness-Eigenschaften automatisch überprüfen lassen. Momentan liegen noch keine Ansätze vor, wie die Erfüllbarkeit von Testformeln mit Liveness-Eigenschaften durch Model-Checking überprüft werden kann.
In dieser Arbeit ist aufgezeigt worden, dass die Klasse der Testformeln mit $true$-Phasen von der Klasse der Testformeln überdeckt wird. Die angegebene Transformation erfordert die Überprüfung von in der Anzahl der $true$-Phasen exponentiell vielen Testformeln. In [Hoe05a] ist es gelungen, zu Countertrace-Formeln mit $true$-Phasen direkt Phasen-Event-Automaten als Semantik anzugeben. Die in [Hoe05a] angegebene Erweiterung der Potenzmengenkonstruktion könnte übernommen werden, um direkt Testformeln mit $true$-Phasen in Testautomaten übersetzen zu können. Dabei ist die in Abschnitt 4.4 vorgestellte Theorie anzupassen.
Neben der Erarbeitung weniger komplexer PEA-Semantiken für Testformeln ist eine Charakterisierung weiterer Formelklassen, die von der Klasse *Testform* überdeckt werden, sinnvoll, um den Rahmen der model-checkbaren Formeln skizzieren zu können.

Es ist bereits in Abschnitt 5.4 weitere Funktionalität aufgezeigt worden, die für die Verwendung des Compilers im *AVACS* Projekt sinnvoll erscheint. In den Fallstudien des Teilprojektes R1 wird mit *CSP-OZ-DC*-Modellen gearbeitet. Für *CSP-OZ-DC*-Modelle ist eine formale Semantik in Form von Phasen-Event-Automaten definiert. Es erscheint im Hinblick auf die Verbreitung der Spezifikationssprache ein Tool erstrebenswert, das die Eingabe von *CSP-OZ-DC*-Modellen über eine graphische Benutzungsoberfläche erlaubt. Diese Modelle werden auf Knopfdruck automatisch in Phasen-Event-Automaten übersetzt. Auf diesen Phasen-Event-Automaten können anschließend Model-Checking Anfragen durchgeführt werden, die komfortabel über eine Oberfläche eingegeben werden.

Literaturverzeichnis

[Ace03] Aceto L., Bouyer P., Burgueño A., Larsen K. G. The Power of Reachability Testing for Timed Automata. *Theoretical Compututer Science*, 300(1-3):411–475, 2003.

[Alu94] Alur R., Dill D. L. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Alu99] Alur R. Timed Automata. In *CAV '99: Proceedings of the 11th International Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22, 1999.

[Asa97] Asarin E., Caspi P., Maler O. A Kleene Theorem for Timed Automata. In *LICS '97: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 160–171. IEEE Computer Society, 1997.

[AVA] Homepage des Sonderforschungsbereichs *AVACS*. http://www.avacs.org/. Letzter Zugriff: 17. Juli 2005.

[Bér01] Bérard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L., Schnoebelen Ph., McKenzie P. *Systems and Software Verification – Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[Bry86] Bryant R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bus04] Buschermöhle R., Brörkens M., Brückner I., Damm W., Hasselbring W., Josko B., Schulte C., Wolf T. Model Checking (Grundlagen und Praxiserfahrungen). *Informatik Spektrum*, 27(2):146–158, 2004.

[Cha03] Chaochen Z., Hansen M. *Duration Calculus – A Formal Approach to Real-Time Systems*. Springer-Verlag, 2003.

[Cla86] Clarke E. M., Emerson E. A., Sistla A. P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cla99] Clarke E. M., Grumberg O., Peled D. A. *Model Checking*. The MIT Press, 1999.

[Cou92] Courcoubetis C., Vardi M., Wolper P., Yannakakis M. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[Die02] Dierks H., Lettrari M. Constructing Test Automata from Graphical Real-Time Requirements. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 433–454. Springer-Verlag, 2002.

[Eic05] Eichner C., Fleischhack H., Meyer R., Schrimpf U., Stehno C. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In *SDL 2005: Model Driven*, volume 3530 of *Lecture Notes in Computer Science*, pages 133–148. Springer-Verlag, 2005.

[Fab05] Faber J., Meyer R. ETCS Case Study: Treatment of Emergency Messages. csd.informatik.uni-oldenburg.de/∼jfaber/avacs/EM_modeldescription.pdf, Juli 2005. Nur für AVACS-Mitglieder zugreifbar, letzter Zugriff: 02. August 2005.

[Frä96] Fränzle M. *Controller Design from Temporal Logic: Undecidability Need Not Matter*. PhD thesis, Christian-Albrechts-Universität Kiel, 1996.

[Gül02] Gülcü C. *Short Introduction to log4j*, 2002. Letzter Zugriff: 08. August 2005.

[Hoe02] Hoenicke J., Olderog E.-R. CSP-OZ-DC: A Combination of Specification Techniques for Processes, Data, and Time. *Nordic Journal of Computing*, 9(4):301–334, 2002.

[Hoe05a] Hoenicke J. *Combination of Processes, Data, and Time*. PhD thesis, C.v.O. University Oldenburg, 2005. Unpublished.

[Hoe05b] Hoenicke J., Maier P. Model-Checking of Specifications Integrating Processes, Data, and Time. In *FM '05: Proceedings of the International Conference on Formal Methods* , volume 3582 of *Lecture Notes in Computer Science*, pages 465–480. Springer-Verlag, 2005.

[Jec04] Jeckle M., Rupp C., Hahn J., Zengler B., Queins S. *UML 2 glasklar*, volume 1. Carl Hanser Verlag München Wien, 2004.

[Let00] Lettrari M. Eine Testautomatensemantik für Constraint Diagrams und ihre Anwendung. Master's thesis, C.v.O. Universität Oldenburg, 2000.

[Log] Homepage des Projektes *Log4j*. http://logging.apache.org/log4j/docs. Letzter Zugriff: 08. August 2005.

[Mar04] Martin Fränzle. Model-Checking Dense-Time Duration Calculus. *Formal Aspects of Computing*, 16(2):121–139, 2004.

[Mer01] Merz S. Model Checking: A Tutorial Overview. In *Modeling and Verification of Parallel Processes (MOVEP 2000)*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001.

[Mey05] Meyer R. Model Checking using Testing. Technical report, C. v. O. University Oldenburg, 2005. Zur Veröffentlichung angenommen, noch nicht publiziert.

[Rav94] Ravn A. P. *Design of Embedded Real-Time Computing Systems* . PhD thesis, Technical University of Denmark, 1994.

[Sif01] Sifakis J. Modeling Real-Time Systems – Challenges and Work Directions. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 373–389. Springer-Verlag, 2001.

[Ska94] Skakkebæk J. U. Liveness and Fairness in Duration Calculus. In Jonsson B., Parrow J., editor, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 283–298. Springer-Verlag, 1994.

[Som01] Sommerville I. *Software Engineering*. Pearson Studium, 6 edition, 2001.

[Tap01] Tapken J. *Model Checking of Duration Calculus Specifications*. PhD thesis, C.v.O. University Oldenburg, 2001.

[Upp] Homepage des Projektes *Uppaal*. http://www.uppaal.com/. Letzter Zugriff: 08. August 2005.

[Xer] Homepage des Projektes *Xerces2-J*. http://xml.apache.org/xerces2-j/. Letzter Zugriff: 08. August 2005.

Symbole

A 19
$\boxed{L}$ 17
$BFormTrue$ 29
$BForm$ 28
$\mathbb{B}$ 8
C 19
χ_Y 19
$\lhd$ 16
$\rhd$ 16
$canseep(tr, p)$ 50
$complete_{tr,i}(p, \gamma)$ 54
E 19
$Ph \models_0 F$ 45
$\diamond L$ 17
$\updownarrow \mathcal{E}$ 16
$\boxminus \mathcal{E}$ 16
$\not\updownarrow \mathcal{E}$ 16
$\models$ 13, 19
$\models_0$ 14
$enter(tr, p, val)$ 50
$FSymb$ 8
$\mathcal{L}(V)$ 18
$Form$ 12
$\hat{f}^n$ 8
f^n 8
$GVar$ 8
$=_{\setminus\{a\}}$ 12
$guard_{tr,i}(p)$ 56
$TA \setminus \{\mathcal{S}\}$ 47
I 19
IL 9
$\mathbb{I}ntv$ 7
$\mathcal{I}$ 8
$\mathcal{I}_k$ 14
(p, β, γ) 19
$keep(tr, p, val)$ 50
$LB(tr)$ 48
$length(tr)$ 48
$PhaseEx$ 29
$PhaseTrue$ 29
$Phase$ 28
P 19
P_0 19
Ph 19
$Ph_1 \parallel Ph_2$ 21
$PowerSet(tr)$ 49
$Prefix_m(tr)$ 48
$TA_1 \parallel TA_2$ 46
$PSymb$ 8
$\mathcal{P}(\lceil true \rceil)$ 57
$\mathcal{P}(tr)$ 53
$\hat{p}^n$ 8
$phase$ 47
  $bound$ 48
  $enterEvents$ 48
  $forbidden$ 48
  inv 48
  $timeop$ 48
p 49
  $gteq$ 49
  in 49
  $less$ 49
  $wait$ 49
p^n 8
$RDC_1(r)$ 27
$\mathbf{Run}(Ph)$ 20
$F \underset{\mathcal{E}}{\Updownarrow} G$ 30

$TA_1 \bullet_{\{s\},\{\gamma\}} TA_2$ 46
$SVar$ 8
s 19
$seep(tr, p, val)$ 50
$successor(tr, p, val)$ 52
$TestformEx$ 29
$TestformTrue$ 29
$Testform$ 28
$TraceEx$ 29
$TraceTrue$ 29
$Trace$ 28
(p_1, g, X, p_2) 19
TA 45
$TVar$ 9
$\mathbb{T}ime$ 7
$\gamma + t$ 19
$Term$ 11
$\mathbf{type}(x)$ 18
$tr(i)$ 48
$UB(tr)$ 48
V 19
Val 8
Val 18
Var 18

Sachverzeichnis

A

Arbeitsumgebung 89

B

Bad-State 45
Besondere Distributivität 31

C

`CounterTrace`
  `Chop` 24
  `DCPhase` 24
Charakterisierung der Erfüllbarkeit mit Testautomaten 57
Charakterisierung von irgendwann 17
Charakteristische Funktion 19
CSP-OZ-DC 18

D

Datenbereich 18
Datentyp 18
Dokumentation 90
  `javadoc` 90
Duration Calculus 7
  Allgemeingültigkeit 14
  Erfüllbarkeit 14
  Erfüllbarkeit durch Automaten 45
  Erfüllbarkeit von 0 an 14
  Events 16
  Gültigkeit 13
  Liveness
    Chop 16
    echtes immer 17
    echtes irgendwann 17
  Symbole(Semantik) 8
    Belegung 8
    Interpretation 8
  Symbole(Syntax) 8
    Globale Funktionssymbole 8
    Globale Prädikatssymbole 8
    Globale Variablen 8
    Observablen 8
  Terme und Formeln 11
    Formeln(Syntax) 12
    Terme(Semantik) 11
    Terme(Syntax) 11
  Transitionsformeln 15
  Zeitabhängige Zustandsausdrücke (Semantik) 10
  Zeitabhängige Zustandsausdrücke (Syntax) 9

E

Endliche Variabilität 10
Event-triggered 31
Exakte Längen in Traces 29

F

Fallstudie
  ApplyEB1 99
  ApplyEB2 100
  $ComNW_{CSP}$ 100
  $ComNW_{DC_1}$ 101

$ComNW_{DC_2}$ 101
$ComNW_{DC_3}$ 101
DetectEM 99
Environment 102
NWToRBC_Alert 100
NWToTrain1_Warn1 99
NWToTrain2_Warn2 100
NWToTrain[...]_Warn[...] 100
RBCToNW_Warn[...] 100
RBC_{CSP} 101
RBC_{DC} 101
SafeAgain1 99
SafeAgain2 100
Train1ToNW_Alert 100
$TrainEM_{CSP}$ 99
$TrainEM_{DC}$ 99
$TrainReact_{CSP}$ 100
$TrainReact_{DC}$ 100
TrainToNW_Alert 99
Fehlerablauf 45

G

Gleichheit modulo 12

H

Hilfsfunktionen
canseep 49
$complete_{tr,i}$ 54
enter 50
guard 56
keep 49
seep 50
successor 52

I

Implementierungsrichtlinien 89

K

Komplexität des Zustandsraums 49
Konfigurationsdatei 90

M

Menge aller Intervalle 7
Model-Checking Theorem 68

N

Normalform
für Testformeln 34
für Testformeln ohne Liveness 40
Model-Checkbare Darstellung 41

P

Phasen-Event-Automaten 19
Ablauf 19
Dauer 20
Definition 19
Erreichbarkeit 20
Events 19
Guard 19
Konfiguration 19
Menge aller Abläufe 20
Parallelkomposition 21
Phasen 19
Run 19
Startzustände 19
Stotterkanten 19
Transition 19
Transitionsrelation 19
Uhren 19
Uhrenbedingung 19
Variablen 19
Zustände 19
Zustandsinvariante 19
Prädikatenlogische Formeln erster Stufe über getypten Variablen 18
ProCos 7

R

RBC 97

S

Sync-Event 30
Sync-Event Introduction 31
Sync-Intro 31

T

true-Phasen in Traces 29
Testautomaten
  Definition 45
  Hiding 47
  Parallelkomposition 46
  Run 45
  Sequentielle Komposition 46
Testautomatensemantik 57
Tests 91
Time-triggered 31
Timeshift 19
Trace
  i-te Phase 48
  Länge 48
  Präfix der Länge m 48

U

Uppaal 97

W

Weiterentwicklung 91

Z

Zeit 7
Zu einem Run gehörige Interpretation43

Paket-, Klassen- und Elementverzeichnis

B

BuildCTA . . . 25, 82, 86, 89
- PhaseBits . . . 25
- buildAut . . . 25
- buildNewTrans . . . 25
- findTrans . . . 25
- precomputeCDDs . . . 25
- recursiveBuildTrans . . . 25

BasicTypes.xsd . . . 75
BooleanExpression . . . 77, 91
- expression . . . 77

C

CDD . . . 24, 88
- and . . . 24
- negate . . . 24
- or . . . 24

Compiler . . . 80, 82
- -help . . . 83
- -lc . . . 83
- -mc . . . 82
- -net . . . 83
- -parallel . . . 83
- -pea . . . 83
- -tf . . . 82
- compile . . . 83, 88
- main . . . 82

ContinuousTime . . . 75
complexType . . . 75

D

Decision . . . 25, 88
- BooleanDecision . . . 25
- EventDecision . . . 24
- RangeDecision . . . 25

E

EventExpression . . . 75, 77
- name . . . 77

Event
- name . . . 75
- spec . . . 75

F

Formula2DNFCompiler . . . 83
Formula2NFCompiler . . . 80, 83
- buildNF . . . 84
- changeNode[...]Child[...] . . . 84
- isBasicElement . . . 83
- isTreeElement . . . 83
- makeBinary . . . 83

FormulaJ2XMLConverter . . . 81, 87
- convert . . . 88
- createCDDFormula . . . 88
- createDecisionNode . . . 88
- create[...]ExpressionNode . . . 88

FormulaXML2JConverter . . . 80, 81, 91
Formulae2DNFCompiler . . . 81
FormulaTree . . . 76

booleanExpression 77
eventExpression 77
formulaTree 77
operator 76
rangeExpression 77
Formula 75, 76
booleanExpression 76
eventExpression 76
formulaTree 76
rangeExpression 76

J

J2XML 82
javadoc 90

L

log4j
DEBUG 90
WARN 90

M

MCTraceXML2JConverter 80, 91
MCTrace 80
MCForm 78, 80, 85, 89
entrySync 78
exitSync 78
mcTrace 78
MCTrace 78, 80, 85, 88
ModelCheckForm.xsd 78, 80, 85, 91

P

PEAJ2UppaalConverter 106
PEAJ2XMLConverter 81, 87
convert 87
createPhaseEventAutomata 87
createPhaseNode 87
createTransitionNode 87
PEAXML2JConverter 81, 91
PhaseEventAutomata 24, 81, 88
Phase 24
pea.modelchecking 73, 74, 80
pea 74, 81
private 90
protected 90
PEA.xsd 26, 78, 81
PEANet 78, 88
PEAPhase
clockInvariant 79
initial 79
invariant 79
name 79
PEA 81
clocks 78
events 78
name 78
phases 78
transitions 78
variables 78
Phase
allowEmpty 92
forbiddenEvent 75
stateInvariant 75
timeBound 75
pea 82

R

RangeExpression 77
bound 77
operator 77
variable 77

T

TestForm2MCFormCompiler . . 80, 83, 85, 88, 91
buildNF 89
compile 85, 88
createMCForm 85, 89
createMCTrace 85, 89
findMCForms 85
introduceSyncEvents 85, 89
makeBinary 89

Trace2PEACompiler . . 74, 81, 82, 86, 91, 92
addForbiddenEventsToTransitions 87
addForbiddenEvents 89
buildAut 89
buildEntrySyncTransitions 86, 89
buildExitGuard 86, 89
buildExitSyncTransitions . 86, 89
compile 86, 88, 89
recursiveBuildTrans 89
TFTree 77
Trace 77
tfTree 77
TestForm.xsd 78, 80, 90
TestForm 77
Trace 77
tfTree 77
TimeBound 75
bound 75
operator 76
Trace 75
event 75
phase 75
spec 75
Transition 79
guard 79
reset 79
source 79
target 79

X

XML2J 82
XMLTags 81
XMLWriter 81, 91
write 88
xerces-Parser 88
xsd:string 75

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt zu haben.

Oldenburg, 9. August 2005

(Roland Meyer)