

ulm university universität
uulm

Computation Sequence Chart
Ein Konzept zur anforderungsbezogenen Testfallgenerierung

DISSERTATION

zur Erlangung des akademischen Grades eines

DOKTOR DER NATURWISSENSCHAFTEN
(DR. RER. NAT.)

der Fakultät für Ingenieurwissenschaften
und Informatik der Universität Ulm

von

VANESSA GROSCH
aus Essen

2011

Gutachter	1.	Prof. Dr. Franz Schweiggert
	2.	Prof. Dr. Helmuth Partsch
Amtierender Dekan:		Prof. Dr.-Ing. Klaus Dietmayer
Tag der Promotion:		08.12.2011

Kurzfassung Anforderungen an das Verhalten eines software-gesteuerten Systems wie auch hinsichtlich weiterer qualitativer Eigenschaften werden im industriellen Kontext nach wie vor in nicht formaler Form, häufig in freiem Text, beschrieben und sind somit semantisch nicht oder nicht hinreichend präzise definiert.

Eine Beschreibung in formaler Form, in einer hinsichtlich Syntax und Semantik klar definierten Sprache, beruht i.a. auf einem Axiomensystem, aus dessen Axiomen und Schlussfolgerungen wohldefinierte Systeme gebildet werden, die vorgegebenen Gesetzen folgen [HB01].

Zwischen der nicht-formalen Anforderung an ein System und der formalen Spezifikation des Systems besteht ein mathematisch nicht fassbarer Unterschied [Sch99]. Er begründet sich auf den unterschiedlichen Vorgehensweisen der Anforderungserstellung.

Der mathematisch nicht vollständig erfassbare Bezug zwischen nicht-formaler Anforderung und formaler Spezifikation betrifft den modellbasierten Test. Der modellbasierte Test zeigt durch die Generierung von Testfällen aus einem Testmodell, dass das ausführbare (reale) System seiner formalen Spezifikation in Form eines Testmodells entspricht. Er zeigt nicht, dass das ausführbare (reale) System den nicht-formalen Anforderungen entspricht, die an das System gestellt wurden. Beim modellbasierten Test muss deshalb zwischen den beiden Aussagen „Das System erfüllt die Anforderungen“ und „Das System erfüllt die formale Spezifikation“ wohl unterschieden werden [Sch99].

Derzeit gibt es keinen praktikablen und gleichzeitig formal korrekten Ansatz, um einen Bezug zwischen nicht-formaler Anforderung und den aus der formalen Spezifikation generierten Testfällen zu ermöglichen. Dieser Bezug würde es erlauben, eine Aussage über die korrekte Umsetzung der nicht-formalen Anforderung in dem getesteten System zu treffen. Die existierenden Verfahren sind entweder zu aufwändig oder nicht formal genug, um durch sie anforderungsbezogene Testfälle generieren zu können.

Die vorliegende Arbeit präsentiert ein Konzept zur anforderungsbezogenen Testfallgenerierung. Eine formale Notation, genannt *Computation Sequence Chart* (CSC), stellt einen Bezug zwischen nicht-formaler und formaler Anforderung her. CSCs können bei der Testfallgenerierung berücksichtigt werden. Um dies zu erreichen, werden die CSCs in eine Logik übersetzt und beim Model-Checking zur Testfallgenerierung in den Prozess integriert.

Das theoretisch entwickelte und praktisch validierte Konzept umfasst

- die Definition einer graphischen Notation (CSC) zur Verlinkung nicht-formaler Anforderungen und formaler Testmodelle,
- die Integration der Notation in das CASE-Tool Enterprise Architect,
- Definitionen, Sätze, Regeln und Softwaremodule zur Transformation der Notation in Prädikatenlogik oder Timed Branching Time Logic und
- die Einbindung von CSCs in die Generierung von Testfällen am Beispiel zweier Model-Checking Verfahren auf Basis von Timed Statecharts und Timed Automata.

Für meinen Vater

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 6
- 1.2 Beitrag der Arbeit . . . 9
- 1.3 Aufbau der Arbeit . . . 10

2 Begriffliche und methodische Grundlagen **11**
- 2.1 System, Realität und Modell . . . 11
- 2.2 Der Begriff der Anforderung . . . 15
- 2.3 Verifikation, Validation und Test . . . 16
- 2.4 Modellbasiertes Testen . . . 17
- 2.5 Von der Anforderung zum Testmodell . . . 23

3 Stand der Wissenschaft **31**
- 3.1 Modellierung von Zeitsystemen . . . 31
  - 3.1.1 Grundlagen . . . 32
  - 3.1.2 Timed Automata und Timed Statecharts . . . 37
  - 3.1.3 Temporallogik . . . 42
- 3.2 Testfallgenerierung aus Modellen . . . 48
  - 3.2.1 Model-Checking . . . 48
  - 3.2.2 Explizites Model-Checking . . . 49
  - 3.2.3 Model-Checking auf Binären Entscheidungsdiagrammen . . . 52
  - 3.2.4 Bounded Model-Checking . . . 54
- 3.3 Requirements Tracing . . . 56
  - 3.3.1 Bestehende Ansätze allgemein . . . 56
  - 3.3.2 Visuelle Notationen . . . 58
  - 3.3.3 Logik-Patterns . . . 64
  - 3.3.4 Observerautomaten . . . 64

4 Computation Sequence Chart **65**
- 4.1 Grundlagen . . . 65
  - 4.1.1 Graphische Notation eines CSCs . . . 69
  - 4.1.2 Transformation in TCTL . . . 72
  - 4.1.3 Beweise durch Prädikatenlogik . . . 73
- 4.2 Sequentielle Ordnungsrelationen . . . 75
- 4.3 Parallele Ordnungsrelationen . . . 86
- 4.4 Gemischte Ordnungsrelationen . . . 93
- 4.5 Komposition mehrerer CSC-Relationen . . . 93
- 4.6 Abgrenzung des Verfahrens . . . 105

5 CSC Testprozess **109**
5.1 Anpassungen des modellbasierten Testprozesses 109
5.2 CSC Korrektheitsprüfung 109
5.3 CSC Testfallgenerierung 111
5.4 CSC Abgleich mit generierten Testfällen 117
5.5 Von der Anforderung zum CSC 119

6 Zusammenfassung und Ausblick **125**
6.1 Ausblick 127

A Algorithmus **129**

Abbildungsverzeichnis **136**

Tabellenverzeichnis **139**

Abkürzungsverzeichnis **140**

Literaturverzeichnis **141**

Kapitel 1

Einleitung

In den 80er Jahren rückten die negativen Auswirkungen des technischen Fortschritts und der Industrialisierung mehr und mehr in das öffentliche Bewusstsein. Die Erkenntnis, dass natürliche Ressourcen nur begrenzt zur Verfügung stehen und jeder Eingriff in die Natur auch indirekte Auswirkungen hat, bewirkte eine Veränderung wissenschaftlichen und technischen Denkens. Wissenschaftler begannen die Frage zu stellen, wie die Welt als Ganzes funktioniert und wie ihre Teile zusammenwirken. Der Wunsch nach einer ganzheitlichen Denkweise, die Mensch, Technik und Natur als Ganzes betrachtet, führte zur Gründung wissenschaftlicher Disziplinen. Die Systemtheorie, die Ökologie und die Ökonomie stellten das ganzheitliche Denken als Paradigma in den Mittelpunkt ihrer wissenschaftlichen Überlegungen [UP91, Ves88].

Eine wichtige These dieser Wissenschaften lautet, dass die Vielfalt, Variabilität und Wandlungsfähigkeit der Prozesse und Wechselwirkungen nicht zu sehr vereinfacht werden dürfen. Albert Einstein empfahl dazu „die Dinge so einfach wie möglich zu machen, aber nicht einfacher“. Die Welt und ihre Zusammenhänge sind komplex und „jeder Versuch, die Komplexität zu reduzieren, erhöht die Komplexität an einem anderen Ort“ [Lot10]. Komplexität erklärt Birger Priddat, Professor für Politische Ökonomie, wie folgt: „Komplexität ist das Spiel mit dem Unbekannten, das immer eine Dimension mehr bietet, als wir uns vorstellen können, als wir wissen. Komplexität bedeutet eigentlich nichts anderes als: Es gibt immer eine Überraschung“ [Lot10].

Die Automobilindustrie lernte aus den Erkenntnissen über die Zusammenhänge von Mensch, Technik und Natur, als sie erkannte, dass eine isolierte Betrachtung des Automobils keinen weiteren Fortschritt im Schutz von Mensch und Natur brachte. Dadurch entstanden Innovationen, die heute in Summe weit sicherere und umweltverträglichere Automobile als noch vor 30 Jahren ermöglichen. Heutige Innovationen konzentrieren sich gezielt auf das Zusammenspiel von Automobil und Umwelt. Drei Beispiele sollen das verdeutlichen. Lane Departure Warning berücksichtigt die Fahrbahnmarkierungen und warnt den Fahrer beim Verlassen der Fahrspur. Hybrid-Fahrzeuge können elektrisch und mit Benzin fahren, um natürliche Ressourcen zu schonen. Notbremsassistenten bei LKWs agieren von der Warnung des Fahrers bis hin zur automatischen Vollbremsung autark. Diese Innovationen beziehen die Umwelt ein und passen Funktionen im Automobil so an, dass sie auf Veränderungen der Umwelt reagieren können und wiederum ihre eigene Auswirkung auf die Umwelt berücksichtigen.

Als Konsequenz aus der erweiterten Reaktivität, vergrößert sich allerdings auch das Beziehungsgeflecht aus Fahrzeugkomponenten und Umweltfaktoren, je mehr Einflüsse in Betracht gezogen werden. Durch jeden weiteren Teilnehmer in diesem Netzwerk entstehen neue Wechselbeziehungen, die es erschweren, alle Zusammenhänge im Blick zu behalten.

In Abbildung 1.1 sind das Automobil und eine Übersicht seiner Wechselwirkungen mit der Umwelt dargestellt. Foto 1 der Abbildung steht exemplarisch für die Wechselwirkungen zwischen Fahrer und Automobil. Der Fahrer steuert anhand des Lenkrads, Gaspedals und anderer Bedienmöglichkeiten das Auto. Über Anzeigen, Airbags, fahrzeuginterne Reaktionen und weitere Funktionen wird der Fahrer beeinflusst. Die Fotos 2 bis 4 beschreiben die Einwirkungen auf den Fahrer und das Automobil. Während der Fahrt können andere Verkehrsteilnehmer oder schlechte Wit-

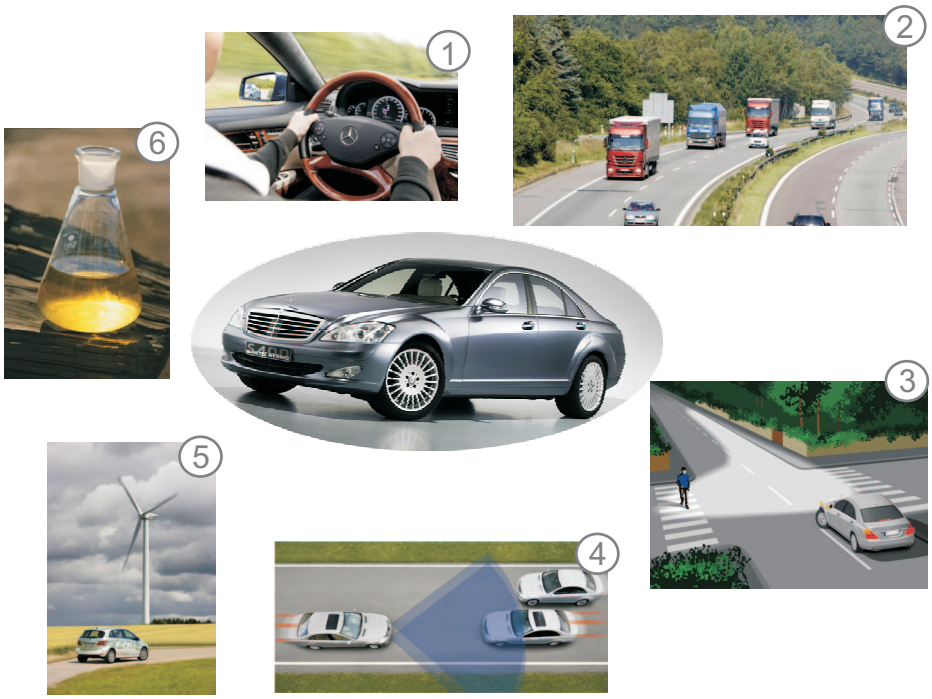


Abbildung 1.1: *Das Automobil und Wechselwirkungen mit seiner Umwelt*

terungsverhältnisse die Fahrzeuginsassen gefährden. Ebenso sind andere Verkehrsteilnehmer potentiell durch ein Automobil gefährdet. Die letzten Fotos 5 und 6 stehen für die Beeinflussung der Natur durch die Herstellung und den Gebrauch eines Fahrzeugs. Bei der Fahrzeugproduktion werden Energie und andere natürliche Ressourcen verbraucht. Abgase und Müll werden an die Natur abgegeben.

Die Abbildung ist nicht vollständig. Um die Graphik zu vervollständigen, müssten alle Einflüsse und Wechselwirkungen identifiziert und beschrieben werden. Diese Aufgabe ist aufgrund der ökologischen, ökonomischen, politischen, sozialen und technischen Zusammenhänge sehr komplex und mit dem heutigen Wissensstand nicht vollständig lösbar.

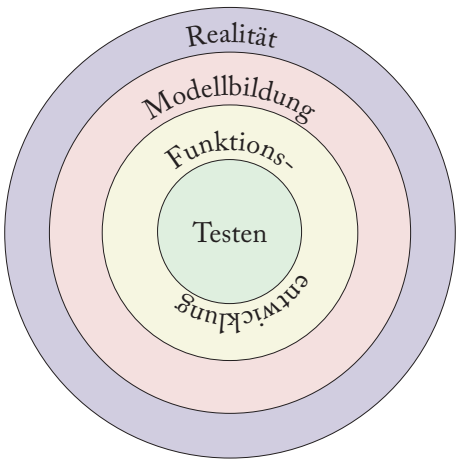


Abbildung 1.2: *Realität - Modell - Funktionsentwicklung*

Eine Fahrzeuginnovation wie Lane Departure Warning ist eine neue (verbesserte) Eigenschaft oder ein neues (verbessertes) Verhalten eines Automobils. Erreicht wird diese Neuerung

durch Funktionen. Eine Funktion ist dabei die Fähigkeit des Systems, Daten auf eine spezifizierte Weise umzuformen oder sein Verhalten aufgrund geänderter Stimuli an die neuen Gegebenheiten anzupassen [Par10]. Funktionen sind zu großen Teilen durch Software realisiert, die sich auf eine oder mehrere Hardwarekomponenten verteilt. Um flexibel auf gewünschte und störende Einflüsse reagieren zu können, müssen diese Komponenten im Fahrzeug dynamisch zusammenwirken. Die Entwicklung der Funktionen erfordert einen Softwareentwicklungsprozess, der das ganzheitliche Denken berücksichtigt. Dafür muss es gelingen, die Realität, in der die Software fehlerfrei funktionieren soll, optimal abzubilden. Da der ganzheitliche Ansatz mehr Wechselwirkungen berücksichtigt, als bisher bei der Funktionsentwicklung üblich, muss der Softwareentwicklungsprozess es erlauben, in der Phase der Spezifikation, der Validation und der Verifikation die Realität so vollständig wie möglich abzubilden.

Wie die Realität in die Funktionsentwicklung aufgenommen werden kann, ist in Abbildung 1.2 und Abbildung 1.3 dargestellt. Der äußere Kreis der Abbildung 1.2 steht stellvertretend für die Realität, in der eine neue Funktion eingebettet sein soll. In der Realität sind alle Zusammenhänge und Wechselwirkungen von Natur, Technik und Mensch enthalten. Die Abbildung der Realität in einem Modell kann nur einen Ausschnitt der Realität wiedergeben. Anhand des Modells wird unter den technischen Gegebenheiten eine Funktion entwickelt, die getestet wird.

Abbildung 1.3 zeigt die einzelnen Schritte von der Realität bis zum Test. Eine Fahrzeuginnovation startet mit dem Übertrag der Realität, in der das Fahrzeug sich bewegt in ein mentales Modell dieser Realität [Par10]. Dieser Vorgang wird als *Erkenntnisprozess* bezeichnet. Das mentale Modell des Funktionsentwicklers enthält allein die ihm bekannten Naturgesetze und Wechselwirkungen. Die Ganzheit des mentalen Modells eines Funktionsentwicklers ist somit ein subjektives Konstrukt.

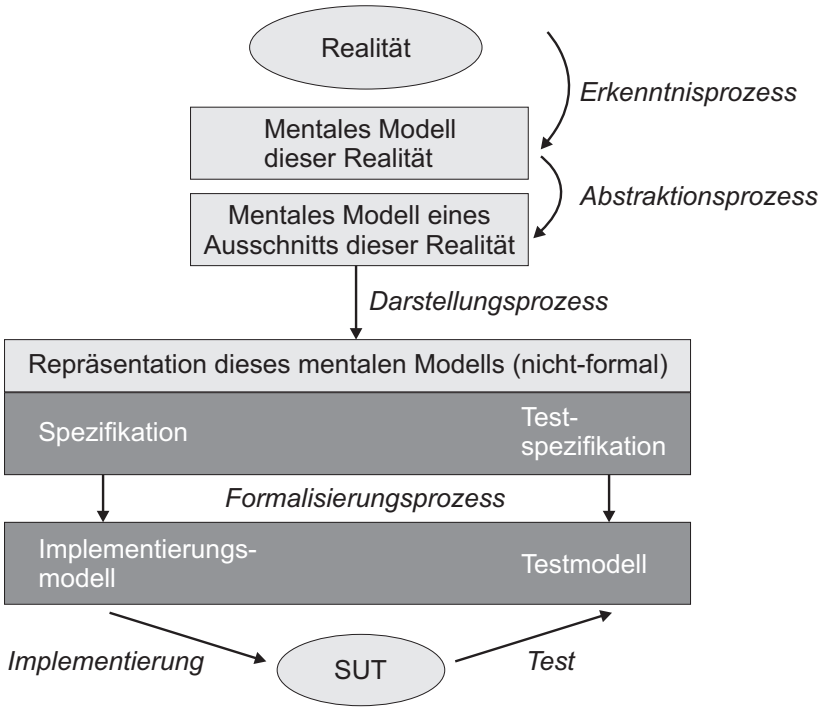


Abbildung 1.3: *(In Anlehnung an [Par10]) Von der Realität zum System under Test (SUT)*

Da nicht alle Bestandteile des mentalen Modells Einfluss auf die neue Funktionalität haben und die Auswirkungen der Funktion begrenzt sind, wählt der Funktionsentwickler im nächsten Schritt, dem *Abstraktionsprozess*, den Ausschnitt des mentalen Modells, der direkt oder indirekt Einfluss auf die Innovation hat. Je nach späterem Nutzen und Einsatz können die Ausschnitte des mentalen Modells variieren. Als mentale Modelle finden die ersten Stufen der Modellbildung keine

Repräsentationen, die anderen Teilnehmern des Entwicklungsprozesses visualisiert werden.

Der Schritt zu einer nicht-formalen Beschreibung des neuen Systems findet im *Darstellungsprozess* statt. Die resultierenden beschreibenden Modelle werden in dem Softwareentwicklungsprozess als Systementwurf zusammengefasst und sind auf unterschiedliche Weise für die verschiedenen Interessengruppen visualisiert.

Die nächste Phase konkretisiert den Systementwurf zu einer Sammlung von Anforderungen, die eine Systemspezifikation bilden. Sie kann nicht-formal oder bereits formal beschrieben sein. Gleiches gilt für die Komponentenspezifikation, die die Aufteilung der Software auf die Hardwarekomponenten enthält. Die letzte Phase vor der Softwareimplementierung, das Softwaredesign, enthält formal spezifiziert die Architektur der Software. Das Testen ist ein wichtiger Bestandteil der Funktionsentwicklung. Je nach eingesetztem Softwareentwicklungsprozess sind die einzelnen Testschritte in die Entwicklung eingebunden. Der Test benötigt ebenfalls eine Spezifikation, in der Vorgaben zu Umfang, Inhalt und Testumgebung gemacht werden.

In welcher Phase des Softwareentwicklungsprozesses eine Formalisierung der nicht-formalen Beschreibungsmodelle stattfindet, wird in der Industrie unterschiedlich gehandhabt. Spätestens die Softwareimplementierung des Systems stellt eine formale Repräsentation der Software dar. Es ist allerdings auch möglich, bereits die Systemspezifikation zu formalisieren. Testspezifikationen können ebenfalls nicht-formal oder formal ausgeführt sein.

Abbildung 1.3 zeigt den modellbasierten Entwicklungsprozess. Bei diesem Prozess wird im *Formalisierungsprozess* ein formales Modell der Systemspezifikation erstellt, das Implementierungsmodell. Ein durchgängig modellbasierter Entwicklungsprozess erstellt ebenfalls ein formales Testmodell auf Basis der Testspezifikation. Das implementierte System (system under test) wird aus dem Implementierungsmodell gewonnen und gegen das Testmodell verifiziert.

Der Prozess der softwarebasierten Funktionsentwicklung leistet einen wichtigen Beitrag, um die Komplexität der gesamten Fahrzeugfunktionalität zu beherrschen. Abbildung 1.4 beinhaltet das V-Modell [BR05] als eine Vorgehensweise im Entwicklungsprozess. Angefangen beim Systementwurf, zeigt sie auf dem linken Ast des Vs, wie der Entwickler einer Funktion stufenweise die Softwareimplementierung über eine stetige Verfeinerung und Formalisierung der Spezifikation erreicht. Dem gegenüber sind auf dem rechten Ast des Vs alle notwendigen Prüfungen und Tests der Software eingetragen. Von unten aufsteigend nimmt der Abstraktionsgrad der Information über die Software immer stärker zu, bis am Ende des Asts der Abnahmetest des Gesamtsystems steht. Aus dem Systementwurf werden Testfälle abgeleitet, die gegen das Gesamtfahrzeug ausgeführt werden. Sie werden in einem Dokument für den Abnahmetest festgehalten. Aus der Systemspezifikation werden Testfälle für den Systemintegrationstest abgeleitet, die gegen die integrierten Hardwarekomponenten ausgeführt werden. Sie werden in der Systemtestspezifikation beschrieben. Das Vorgehen wiederholt sich bis zum Softwaretest auf unterster Ebene. Der Nachteil des V-Modells liegt in der strikten zeitlichen Trennung der einzelnen Entwicklungsphasen, die so in der Realität meist nicht vorgefunden wird [Par10].

Wie zu Beginn der Einleitung erläutert, liegt die Schwierigkeit der Entwicklung von Innovationen in den Wechselwirkungen der Systeme und ihrer Umwelt begründet. Im Sinne des ganzheitlichen Denkens ist die Antwort darauf, soviel Information wie möglich in Modellen darzustellen, um unerwünschte Nebenwirkungen bei der Herstellung und Nutzung technischer Systeme zu vermeiden. Mit dem wachsenden Umfang an Information wächst auch die Komplexität der Funktionen und Modelle. Der Anstieg der Komplexität ist durchaus erlaubt und, entgegen des Trends der letzten Jahrzehnte, heute wieder erwünscht [Lot10]. Die Wissenschaft hat beobachtet, dass technische Systeme als Teil einer komplexen Umwelt nicht entwickelt werden können, indem diese Umwelt stark vereinfacht dargestellt wird. Es muss versucht werden, die Realität so gut es geht abzubilden. Als Konsequenz erfordert die Komplexität der Funktionalität und seiner Umgebungsmodelle neue Wege der Absicherung dieser Funktionen, da bisherige Verfahren des klassischen Testens auf weniger komplexe Systeme ausgelegt waren. Ein Weg, der vermehrt zum Einsatz kommt, ist der *modellbasierte Test* [GNRS09]. Er benötigt ein formales Testmodell, wie in Abbildung 1.3 dargestellt. Auf Basis dieses Testmodells, das für die Beschreibung des system under test (SUT) und seiner Testumgebung erstellt wurde, generiert ein Algorithmus Testfälle, die gegen das SUT ausgeführt werden. Der Vorteil der Erstellung eines formalen Testmodells liegt in der

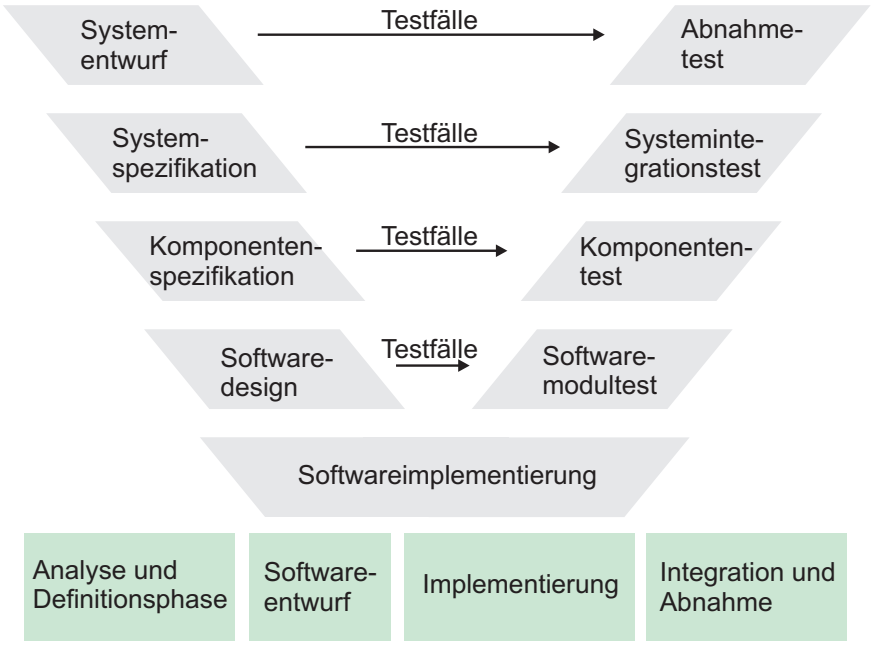


Abbildung 1.4: *Der Entwicklungsprozess nach dem V-Modell*

Möglichkeit, die Realität in beschreibenden Modellen der Testumgebung und des SUTs abzubilden und dieses Wissen bei der Testfallermittlung direkt einsetzen zu können. Der Modellgedanke, der bereits bei der Funktionsentwicklung gelebt wurde, wird beim Testen fortgeführt. Zudem wird durch den Einsatz mathematischer Verfahren die Wiederholbarkeit der Testfallermittlung möglich und Metriken der Modellüberdeckung erlauben Prognosen der Systemabdeckung.

Im Prozess des modellbasierten Tests existiert eine Lücke. Eine wichtige Aussage, die bei der Verifikation eines Systems getroffen werden soll, betrifft die nicht-formalen Anforderungen. Es gilt zu zeigen, dass das implementierte System die nach dem Darstellungsprozess gestellten Systemanforderungen erfüllt. Zwischen der nicht-formalen Anforderung an ein System und der formalen Spezifikation des Systems besteht ein mathematisch nicht fassbarer Unterschied [Sch99]. Die nicht-formale Anforderung steht im direkten Bezug zur Realität. Sie hat zum Zweck, ein *Vor*- oder *Nach*bild eines Ausschnitts der Realität zu beschreiben, der in Software übertragen werden soll. Die formale Spezifikation hingegen beruht auf einem Axiomensystem, aus dessen Axiomen und Schlussfolgerungen wohldefinierte Systeme gebildet werden können, die vorgegebenen Gesetzen folgen [HB01]. Der Bezug des Systems zur Realität bleibt dem Ersteller überlassen. Beim modellbasierten Test muss deshalb zwischen den beiden Aussagen „Das System erfüllt die Anforderungen“ und „Das System erfüllt die formale Spezifikation“ wohl unterschieden werden [Sch99]. In welcher Beziehung die formale Spezifikation zur nicht-formalen Anforderung steht, ist im Testmodell allein nicht enthalten. Der Testmodellerstellung liegen die nicht-formalen Anforderungen zwar als Basis vor, der Bezug zwischen nicht-formaler Anforderung und formalem Testmodell geht allerdings ohne ein bewusstes Festhalten während des Formalisierungsprozesses verloren. Das sogenannte *Requirements Tracing* beugt diesem Verlust vor. Allerdings sind bestehende Verfahren des Requirements Tracing im modellbasierten Test entweder zu aufwändig oder nicht formal genug, um durch sie anforderungsbezogene Testfälle generieren zu können.

Vorliegende Arbeit stellt ein Konzept vor, wie anhand einer Notation zur Anforderungsverfolgung, genannt Computation Sequence Chart (CSC), eine Brücke zwischen nicht-formaler Anforderung und formaler Spezifikation geschlagen wird, die auch eine Generierung von anforderungsbezogenen Testfällen erlaubt (Abbildung 1.5). Das Konzept ist ein Schritt in Richtung der Aussage „Das System erfüllt die Anforderungen“.

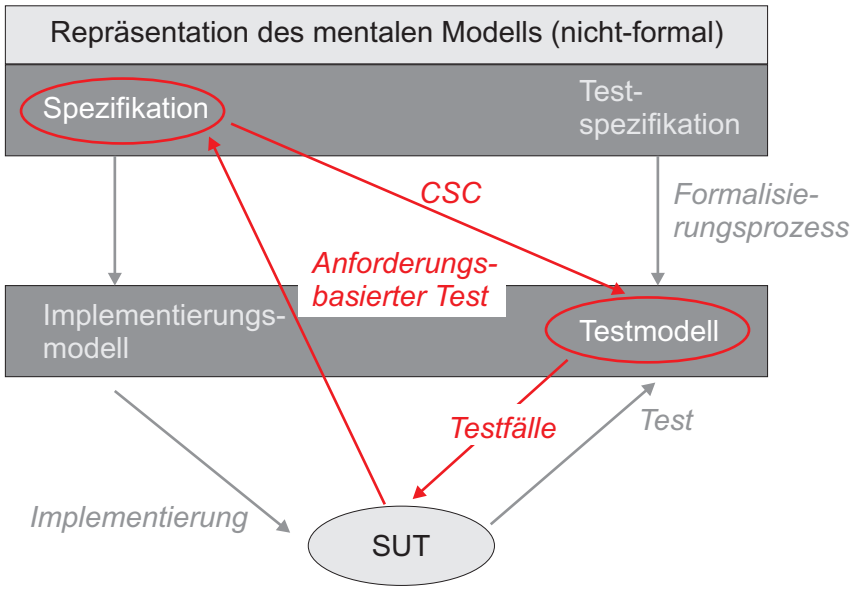


Abbildung 1.5: *Die Spezifikation kombiniert mit dem Testmodell ermöglicht den anforderungsbezogenen Test*

1.1 Motivation

Der Einsatz modellbasierter Testfallermittlungsmethoden birgt mehrere Schwachstellen. Zwei Argumente, die herangezogen werden, sind die Motivation für vorliegende Arbeit. Das erste Argument bemängelt die unzureichende *anforderungsbezogene Verifikation* im modellbasierten Test. Das zweite Argument bezieht sich auf die schwierige *Nachvollziehbarkeit* der Testfallgenerierung und ihrer Ergebnisse. Differenzierter betrachtet ergeben sich daraus folgende Überlegungen.

Ein Kernpunkt der Testfallermittlung ist der Nachweis, dass die an das System gestellten Anforderungen erfüllt wurden. Dies wird nicht allein von Normen gefordert, sondern vor allem vom Kunden, der zum Zeitpunkt der Abnahme wissen muss, ob seine Anforderungen an das System korrekt umgesetzt wurden.

Beim modellbasierten Test wird vorrangig gegen die formale Testspezifikation getestet. Das Testmodell liefert den Input des Testprozesses. Der modellbasierte Test zeigt durch die Generierung von Testfällen aus einem Testmodell, dass das ausführbare (reale) System seiner formalen Spezifikation in Form eines Testmodells entspricht. Er zeigt nicht, dass das ausführbare (reale) System den nicht-formalen Anforderungen, die an das System gestellt wurden, entspricht. Der Bezug zwischen nicht-formaler Systemanforderung und formaler Testspezifikation in Form eines Testmodells ist notwendig, um eine Aussage über die Korrektheit der nicht-formalen Anforderungen anhand des modellbasierten Tests treffen zu können. Es ist sehr aufwändig, im Nachhinein einen generierten Testfall einer nicht-formalen Anforderung zuzuordnen. Leichter ist es, den Bezug des Testmodells zur nicht-formalen Anforderung bereits bei der Modellerstellung zu beachten und festzuhalten. Wird dieser Bezug maschinenverarbeitbar niedergeschrieben, kann ein Verweis von generierten Testfällen auf nicht-formale Anforderungen bereits bei der Generierung erstellt werden.

Ein maschinenverarbeitbarer Anforderungsverweis erzwingt eine Formalisierung der Anforderungen. Auf Basis dieser formalisierten Darstellung können anforderungsbezogene Testfälle generiert, beziehungsweise automatisch ein Abgleich generierter Testfälle mit den Anforderungen vorgenommen werden. Der Aufwand, die Formalisierung zu erhalten, darf den dadurch erreichten Nutzen nicht übersteigen. Es entsteht ein Trade-Off zwischen der Zeit, die in die Formalisierung investiert werden muss und dem Gewinn, der damit erzielt werden kann. Deshalb wird ein Formalismus benötigt, der einfach und schnell erzeugt werden kann und von einem Testfallgenerator

interpretierbar ist.

Die Dokumentation und Nachvollziehbarkeit der Testfallgenerierung ist neben der anforderungsbezogenen Verifikation ein weiterer Schwachpunkt des modellbasierten Tests. Um eine Zuordnung von Testfällen und sonstigen, während des Testprozesses anfallenden Artefakten zu Anforderungen sicherzustellen, ist eine Anforderungsverfolgung notwendig. Die Verfolgung von Anforderungen im Testprozess ist Teil des *Requirements Tracing* und kann auf unterschiedliche Weise erreicht werden. Eine detaillierte Diskussion dazu ist Inhalt von Abschnitt 3.3. Allerdings resultieren aus allen bekannten Verfahren Nachteile bezogen auf ihren Einsatz im modellbasierten Test. Die Nachteile werden in Abschnitt 4.6 dargelegt.

Die Akzeptanz einer neuen Methode zur automatischen Testfallermittlung hängt für den Testingenieur stark von der Nachvollziehbarkeit der Prozessschritte und seiner Ergebnisse ab. Vorrangig von Interesse sind dabei nicht die im Testfallgenerator implementierten Algorithmen der Testfallauswahl. Es geht vielmehr darum, die Motivation von Testfällen nachvollziehen zu können. Bei der Testfallauswahl bilden Anforderungen, bekannte Testfallermittlungsverfahren, Wissen über Fehlerquellen des zu testenden Systems und allgemeine Erfahrungen bei der Fehlerfindung die Grundlage für den Testingenieur. Im Gegensatz dazu sind strukturelle Überdeckungsmaße und Testauswahlkriterien, die maximiert werden sollen die Begründung der Testfallauswahl für einen Testfallgenerator. Beide Motivationen sind notwendig und müssen als zwei Sichtweisen auf denselben Testfall unterstützt und visualisiert werden.

Zusätzlich zur gewünschten Nachvollziehbarkeit der Testfallgenerierung durch Testingenieure müssen Test-Verantwortliche, Zertifizierer und Auditoren auch nachträglich noch nachvollziehen können, welche Testfälle ausgewählt wurden und an welchen Kriterien sich die Auswahl orientierte, um in der Vergangenheit getroffene Entscheidungen und Testendekriterien bewerten zu können. Die Dokumentation der Testfallermittlung sowohl als auch der Testfälle ist ein geforderter Teilschritt der Qualitätsabsicherung. Der Bezug zwischen Testfällen und Anforderungen muss Bestandteil der Testdokumentation sein.

Die Notwendigkeit, sowohl anforderungsbezogene Testfälle generieren zu können als auch die Nachvollziehbarkeit der Testfallgenerierung zu verbessern führte zu dem Ziel, ein formales Requirements Tracing für den Modellbasierten Test zu entwickeln. Eine einzelne Anforderung, meist natürlichsprachlich spezifiziert, soll einem Testfall zugeordnet werden und vice versa. Um die Zuordnung bereits bei der Testfallgenerierung zu erhalten, muss der Testfallgenerator die Zuordnung verarbeiten können. Daraus resultierte die Motivation, eine formale Zuordnung zu erstellen. Diese Zuordnung kann leicht erreicht werden, indem alle an einer Anforderung beteiligten Testmodellelemente mit der Anforderung annotiert werden. Die einfache Annotation hat allerdings drei Schwächen. Erstens kann nicht festgehalten werden, in welcher zeitlichen Ordnung Zustandswechsel passieren sollen. Zweitens kann nicht vorgegeben werden, dass zwei Zustände gleichzeitig besucht werden sollen. Und drittens kann nicht gefordert werden, dass Zustände mehrfach besucht werden, wie es zum Beispiel zur Beschreibung von Zyklen wichtig ist.

Warum die drei Aspekte, zeitliche Ordnung von Zuständen, parallele und sequentielle Synchronisation von Zuständen und Zyklen in Pfaden wichtig sind, wird im Folgenden exemplarisch begründet.

Beispiel 1. *Die zeitliche Ordnung von Zuständen ist notwendig für die folgende Blinkanforderung eines Automobils. Sie ist dem Systemlastenheft der Blinkfunktionalität entnommen.*

> `SLH-870` : Die maximale zeitliche Differenz zwischen der Betätigung des Blinkerhebels und dem Blinken der Lampen beträgt 600ms.

Um einen Testfall zu generieren, der den Inhalt der Anforderung als eine Zustandsfolge am Testobjekt stimuliert, muss das Testobjekt in den Zustand `Blinkerhebel = Links` *(bzw.* `Rechts`*) gebracht werden und spätestens 600ms später muss der Zustand* `Lampe = An` *auftreten. Innerhalb des Testmodells liegen zwischen diesen beiden Zuständen viele Zwischenzustände, die an unterschiedlichste zeitliche Bedingungen geknüpft sind. Ob die Restriktion von maximal 600ms immer eingehalten wird, ist für den Testingenieur nur schwer zu zeigen. Model-Checking ist ein Verfahren um zu beweisen, dass innerhalb des Testmodells die 600ms Grenze*

nie verletzt wird. Bei großen Systemen wie dem Aussenlicht eines Automobils lässt sich dieser Beweis mit den zum heutigen Stand der Technik bekannten Verfahren nicht führen. Es können allerdings Testfälle generiert werden, die zumindest ein paar Situationen im Testobjekt auslösen, durch die getestet werden kann, ob die 600ms Toleranz verletzt wird.

□

Beispiel 2. *Die Blinkanforderung* `SLH-866` *aus dem Systemlastenheft sei gegeben als*

> `SLH-866` : Ein durch Warnblinken unterbrochenes Richtungsblinken wird bei Wegfall der Warnblinkanforderung wieder aufgenommen.

Um eine testbare Systemsituation herzustellen, die Anforderung `SLH-866` *widerspiegelt, muss zuerst* `Richtungsblinken = An` *erzeugt und danach* `Richtungsblinken = An` *gleichzeitig mit* `Warnblinken = An` *am Testobjekt stimuliert sein. Im Anschluss muss der Zustand* `Warnblinken = An` *verlassen werden, während der Zustand* `Richtungsblinken = An` *noch anliegt.*

Als Konsequenz aus dieser Anforderung muss es möglich sein, parallele (zeitliche) Abhängigkeiten zu spezifizieren, um eine Beziehung zwischen Anforderung und Testfall zu erhalten.

□

Beispiel 3. *Die Blinkanforderung* `SLH-868` *aus dem Systemlastenheft sei gegeben als*

> `SLH-868` : Wiederholte Tippblinkanforderungen lösen die letzten 3 Blinkzyklen aus (kein Aufsummieren der Anforderungen).

Tippblinkanforderungen resultieren in 3 Blinkzyklen (Blinkzyklus = Hell- + Dunkelphase der Blinklampe). Eine weitere Tippblinkanforderung während einer bereits angefangen Anforderung darf nicht dazu führen, dass sich die Blinkzyklen jedesmal um 3 erhöhen. Ist zum Beispiel der erste Blinkzyklus einer Anforderung ausgeführt worden, führt die zweite Anforderungen dazu, dass insgesamt 4 Mal geblinkt wird. Die Anzahl darf sich nicht zu 6 Blinkzyklen aufsummieren. Sobald eine neue Tippblinkanforderung anliegt, muss der Zähler der Blinkzyklen somit wieder auf 3 gesetzt werden. Um diese Situation beobachten zu können, muss also mindestens zweimal ein Tippblinken ausgelöst werden. Ein solches zweimaliges Besuchen von Zuständen im Testmodell wird von Testfallgeneratoren für gewöhnlich vermieden, da hinter einem Zyklus ein identisches Systemverhalten vermutet wird. Kann es aber von Bedeutung sein, einen Zyklus im Testobjekt zu beobachten, wie es in Anforderung `SLH-868` *der Fall ist, muss dieser Zyklus erzwungen werden.*

□

Um diese drei Aspekte einem Testfallgenerator als Bedingung vorgeben zu können, braucht man eine Logik, die von ihm interpretiert werden kann. Diese Logik sollte am besten der bereits eingesetzten Logik beim Model-Checking entsprechen. Hierfür bieten sich die Temporallogik und die Prädikatenlogik an. Die Beschreibung von Anforderungen in Temporallogik oder Prädikatenlogik ist aufwändig und schwer und dadurch fehleranfällig. Eine graphische Repräsentation der Logik erleichtert den Umgang damit, da Syntax und Semantik mit der graphischen Notation verknüpft sind, und dem Testingenieur nicht bekannt sein müssen. Bestehende graphische Notationen, die eine logische Repräsentation besitzen, fordern einen, vom Testmodell losgelösten, parallelen Zusatzaufwand. Eine Anforderung muss in diesen Fällen neu formalisiert und in einer anderen Notation als der des Testmodells spezifiziert werden (vgl. Abschnitt 3.3 und Abschnitt 4.6). Der Zusatzaufwand, der dadurch betrieben werden muss, verhindert ihren effizienten Einsatz in der industriellen Praxis. Dies motivierte in der vorliegenden Arbeit dazu, eine einfache graphische Notation zu erstellen, die sich direkt aus den Elementen des Testmodells bedient.

1.2 Beitrag der Arbeit

Der mathematisch nicht vollständig erfassbare Bezug zwischen nicht-formaler Anforderung und formaler Spezifikation begründet den Ansatz einer neuen Methode zur Anforderungsverfolgung und -generierung im modellbasierten Test. Die vorliegende Arbeit stellt ein Konzept vor, wie die Lücke werkzeuggestützt geschlossen werden kann. Der Beitrag vorliegender Arbeit gibt dem Softwareentwickler ein maschinengestütztes Werkzeug zur Hand, das eine graphische Repräsentation seines Verständnisses vom Bezug der formalen Systemspezifikation „Testmodell“ zur nicht-formalen Anforderung an das System speichern und verarbeiten kann. Er wählt dazu die nicht-formale, aus der Realität abgeleitete Anforderung als Beschreibungsmittel des Systems und bildet diese auf das Testmodell ab. Die Abbildung erfolgt über eine Sequenz von Zustandswechseln des Testmodells. Die Eingabeparameter der Pfade stimulieren das ausführbare System so, dass die nicht-formale Anforderung beobachtbar und bewertbar wird. Der Softwareentwickler kann auf Basis der graphischen Repräsentation seines Verständnisses mit anderen Beteiligten die Korrektheit des Testmodells diskutieren, sein Wissen dokumentieren und mithilfe eines Testfallgenerators anforderungsbezogene Testfälle generieren. Zur Generierung von Testfällen wird die graphische Repräsentation in eine Logik übersetzt.

Die im vorangegangenen Abschnitt 1.1 diskutierten Argumente motivierten die Definitionen einer leicht zu erstellenden graphischen Beschreibung einer einzelnen Anforderung, die automatisch formalisiert und bei der Testfallgenerierung eingesetzt wird. Sie stellt den Link zwischen Anforderung, Testmodell und Testfall her. Da heutige Ansätze der Verknüpfung von Anforderungen und ihrer formalen Spezifikation entweder zu wenig Logik enthalten und somit eine ungenügende Aussagekraft besitzen, oder zu viel Logik erzwingen und damit ineffizient sind, wird eine neue Vorgehensweise der Anforderungsverfolgung benötigt.

Die Nachvollziehbarkeit der natürlichsprachlichen Anforderung als auch der anforderungsbezogene Test wird durch eine graphische Beschreibung der Anforderung erreicht. Die Definition der graphischen Notation, genannt *Computation Sequence Chart*, ist der erste Beitrag der Arbeit. Um die Anforderungsbeschreibung bei der modellbasierten Testfallgenerierung einsetzen zu können, wird im zweiten Beitrag der Arbeit eine Transformation der Anforderungsbeschreibung sowohl in die Temporallogik *Timed Branching-time Logic* als auch in *Prädikatenlogik* erläutert.

Die Einführung von Computation Sequence Charts in den modellbasierten Testprozess adressiert die Herausforderung

- des anforderungsbezogenen Tests auf komplexen Anforderungen,
- der Beziehung zwischen Anforderungen und Testfällen,
- der fehlenden Nachvollziehbarkeit der Testfallgenerierung und
- der Wartung und Wiederverwendung von Testmodellen.

Das vorgestellte Konzept beinhaltet

- die Definition einer graphischen Notation (CSC) zur Verlinkung nicht-formaler Anforderungen und formaler Testmodelle,
- die Integration der Notation in das CASE-Tool Enterprise Architect,
- Regeln und Softwaremodule zur Transformation der Notation in Prädikatenlogik oder Timed Branching Time Logic und
- die Einbindung von CSCs in die Generierung von Testfällen am Beispiel zweier Model-Checking Verfahren auf Basis von Timed Statecharts und Timed Automata.

1.3 Aufbau der Arbeit

Die wichtigsten Begriffsdefinitionen werden in Kapitel 2 vorgenommen. Dazu zählen die Begriffe: Realität, System, Modell, Anforderung, Verifikation und Validation sowie modellbasierter Test.

Eine genauere Betrachtung der Modellierung von Zeitsystemen und der Generierung von Testfällen aus den resultierenden Modellen anhand des Model-Checking ist Teil von Kapitel 3. Die wichtigsten Verfahren zur formalen Anforderungsbeschreibung aus Sicht der Zielsetzung vorliegender Arbeit sind in diesem Kapitel ebenfalls beschrieben.

Die Definition und Anwendung einer graphischen Notation zur Anforderungsverfolgung und zum anforderungsbezogenen Test ist Inhalt von Kapitel 4 und der Beitrag der Dissertation. Die Notation heißt *Computation Sequence Chart* (CSC). Ein CSC wird in die Temporallogik Timed Branching-time Logic (TCTL) übersetzt. Die Validation des Ansatzes erfolgt über die Transformation der TCTL-Formeln in Prädikatenlogik.

Computation Sequence Charts sind Teil des modellbasierten Testprozesses. Ihre Erstellung, ihr Einsatz bei der Testfallgenerierung und der Abgleich von CSCs mit generierten Testfällen wird in Kapitel 5 vorgestellt. Zudem wird ihr Einsatz in der Industrie am Beispiel der Außenlichfunktionalität eines Automobils beschrieben.

Die Zusammenfassung (Kapitel 6) zeigt die wichtigsten Eckpunkte der Festlegung einer graphischen Notation im Rahmen des anforderungsbezogenen, modellbasierten Tests. Die Ziele und Motivation der Computation Sequence Charts wird den erreichten Vorteilen gegenübergestellt. Zudem werden im Ausblick offene Punkte und Verbesserungspotential aufgezeigt.

Anhang A enthält den Algorithmus zur Generierung von TCTL-Formeln aus Computation Sequence Charts und das Klassendiagramm der TCTL-Library.

Kapitel 2

Begriffliche und methodische Grundlagen

Der Beitrag vorliegender Arbeit ist eingebettet in umfassendere Fragestellungen der Softwaretechnik. Dazu zählen die Fragen: Was ist ein reales System und was ein Modell des Systems? Welche Rolle spielt die Mathematik bei der Übertragung von Systemmodellen in Software und bei der Verifikation dieser Software? Der modellbasierte Test setzt das Testmodell als eine formale Spezifikation des ausführbaren (realen) Systems ein. Eine Diskussion dieses Verfahrens benötigt die Begriffe *Realität*, *System* und *Modell*. Mathematische Verfahren zur Testfallermittlung dienen der Verifikation von Software. Software wird auf Basis von Anforderungen erstellt. In diesem Rahmen spielen mathematische Modelle, *Anforderungen* und der Begriff *Verifikation* eine Rolle.

Je nach Einsatzgebiet werden diese Begriffe unterschiedlich definiert. Sie sind sowohl in den Geisteswissenschaften, wie der Psychologie und der Philosophie, als auch in den Naturwissenschaften, zum Beispiel in der Physik, Mathematik und Informatik wissenschaftliche Begriffsgrundlage. Der Ansatz vorliegender Arbeit bedient sich vorrangig der Begriffsdefinitionen aus der Systemtheorie, Informatik und Mathematik. Es ist dennoch wichtig, Motivation und Verständnis der Begriffe aus philosophischer Sicht zu berücksichtigen. Speziell die Frage der Erkenntnis und Abbildung realer Systemen und Objekte ist seit jeher eine philosophische Diskussion.

2.1 System, Realität und Modell

System Der Begriff System wird laut Ulrich und Pohl [UP91] wie folgt definiert: „Ein System ist ein aus *Teilen* bestehendes *Ganzes*“. Diese Definition ist sehr allgemein gehalten und sagt über die Eigenschaften und das Verhalten der Teile und des Ganzen nichts aus. Einen Schritt weiter geht die Definition von Dangelmaier [Dan03]: „Ein *System* ist ein aus mehreren *Objekten* zusammengesetztes *geordnetes* Ganzes, das ein *beschreibbares Verhalten* aufweist. Der *Zustand eines Systems* ist die aktuelle Zusammensetzung aller Eigenschaften seiner Objekte. Das *Verhalten des Systems* ist die Veränderung seiner Objekteigenschaften.“ Die Schwierigkeit bei dieser Definition ist der Begriff „Objekt“. Was ist ein Objekt und woher kommt es? Teilweise wird diese Frage im folgenden Paragraph „Realität“ beantwortet werden.

Zur Begriffsdefinition vorliegender Arbeit wird eine dritte Variante aus Ulrich und Pohl [UP91] herangezogen, die sich allein auf dynamische Ganzheiten bezieht und den Begriff des Objekts vermeidet. Die Definition schränkt die allgemeine Definition zudem auf Systeme ein, die sich über die Zeit verändern.

Definition 2.1.1 (System [UP91])**.** *Ein System ist ein dynamisches Ganzes, das als solches bestimmte Eigenschaften und Verhaltensweisen besitzt. Es besteht aus Teilen, die so miteinander*

verknüpft sind, dass kein Teil unabhängig ist von andern Teilen und das Verhalten des Ganzen beeinflusst wird vom Zusammenwirken aller Teile.

□

Der Begriff *Ganzes* drückt aus, dass das System in Teile zerlegt werden kann. Die einzelnen Teile verleihen dem Ganzen durch ihr Zusammenwirken Eigenschaften und Verhaltensweisen, jedoch sind die Eigenschaften des Ganzen mehr als die Summe der Teileigenschaften. Der Begriff *Teil* weist daraufhin, dass dieser nicht als Ganzes verstanden wird, sondern zur Bildung des Ganzen beiträgt. Er ist nicht losgelöst von allen anderen Teilen des Ganzen und wirkt sich somit direkt oder indirekt auf das Ganze aus. Das System ist durch eine Systemgrenze (geschlossen oder offen) von seinem Umfeld abgegrenzt.

Die unterschiedlichen Sichtweisen auf ein System führen zu drei Systemkonzepten [SM02]. Das *funktionale* Konzept stellt die Funktionsbeschreibung des Systems anhand der Änderung von Eingabeoperanden zu Ausgabeoperanden in den Mittelpunkt der Betrachtung. Das *strukturelle* Konzept beschreibt das System über Elemente, ihr Verhalten und sie verknüpfende Relationen. Die Struktur des Systems wird von seiner Umwelt über die Systemgrenze abgegrenzt. Das *hierarchische* Konzept zerlegt das System sukzessive in Teilsysteme.

Realität Jedes System ist auch ein Teilsystem. In jedem Fall ist es ein Teil eines umfassenderen Systems, zusammengesetzt aus seiner Umwelt und ihm selbst. Aus dieser Argumentation wird deutlich, dass sich die Welt aus Teilsystemen zusammensetzt und noch weiter gedacht ist die Welt ein Teilsystem des größeren Systems Universum. Um Systeme und Teilsysteme bestimmen und beschreiben zu können, sollte man also bei der Frage beginnen, *was* das Universum *ist*. Die Frage nach dem Verständnis des Universums (alternativ: der Welt, der Realität oder der Wirklichkeit) ist seit jeher eine philosophische Frage und sollte auch in einer naturwissenschaftlichen Arbeit nicht ohne Berücksichtigung der bereits jahrhundertelang diskutierten Ansätze gestellt und beantwortet werden.

Die Definition des Begriffs Realität wird sich für die vorliegende Arbeit an dem Ziel orientieren, technische Systeme spezifizieren zu können. Sie soll helfen zu verstehen, *woher* ein System kommt, *wer* bestimmt und erkennt, *was* ein System ist und *wie* sich dieses System von der Umwelt abgrenzt. Die Auseinandersetzung mit den Themen Herkunft, Erkennbarkeit und Abgrenzung nennt man den *ontologisch-erkenntnistheoretischen Zugang* [HB01]. Ontologie ist die Lehre von der Gegebenheit des Seins ([Bro79], Band 8). Epistemologie die Lehre vom Wissen und von den Prozessen und Grenzen der Erkenntnisgewinnung ([Bro79], Band 3). Innerhalb dieser Teilgebiete der Philosophie haben sich unterschiedliche Ansichten und Ansätze entwickelt, den Begriff Realität beziehungsweise Wirklichkeit zu interpretieren. Fünf Ansätze sollen im Folgenden kurz erläutert werden.

Der *(externe) Realismus* geht davon aus, „dass es eine - außerhalb unseres Bewusstseins existierende und daher prinzipiell objektiv erfassbare - Welt gibt“ [HB01]. Wir, als Betrachter dieser Welt, sind in der Lage, diese von außen zu beobachten und zu beschreiben. Wir sind somit bei der Festlegung von Systemen nicht Teil des Systems und sind in unserer Wahrnehmung der Dinge und Prozesse objektiv. Systeme und Objekte der Welt müssen noch nicht existieren, sie können auch erst Betrachter-spezifisch definiert, abgegrenzt und spezifiziert werden. Gewisse Eigenschaften sind jedoch vom Betrachter unabhängig gegeben, wie zum Beispiel physikalische Gegebenheiten materieller Objekte (Lage, Größe,...).

Der *naive Realismus* sieht nicht allein das Universum als gegeben und objektiv betrachtbar an, sondern auch alle darin befindlichen Systeme. Die Systeme existieren bereits und müssen nur noch gefunden werden. Die Realität ist so gut erfassbar, wie es die dem Betrachter zur Verfügung stehenden Beschreibungsmittel erlauben. Die Realität umfasst alles, was vom Betrachter erkannt, abgegrenzt und beschrieben werden kann. Dabei ist der Betrachter der Realität im Moment des Erkenntnisprozesses ein Außenstehender.

Der *Idealismus* wird gemeinhin als Gegenpart zum Realismus gesehen. Nach Immanuel Kant (zit.n. [Mor02], S. 54) lautet eine seiner Kernthesen „[...] dass der Mensch sich nicht nach den

Gegenständen richten soll, wie die Planeten sich nach der Sonne richten. Vielmehr richten sich die Gegenstände nach dem menschlichen Erkenntnisvermögen.“ Der Unterschied zum Realismus liegt somit in dem Verständnis, dass Systeme nur soweit existieren, wie wir sie begreifen. Die Möglichkeit, Systeme selber zu bilden und nicht nach ihnen suchen zu müssen, verlangt einen tiefer gehenden Analyse- und Erkenntnisprozess während der Systembildung.

Sowohl der *interne Realismus* als auch der *Konstruktivismus* messen dem Betrachter der Welt eine zentrale Rolle bei [HB01]. Er steht nicht außerhalb des Geschehens, sondern gestaltet durch sein Handeln und Wissen die Welt mit. Interner Realismus und Konstruktivismus propagieren, dass das System immer im Kontext weiterer Systeme, Beteiligter, seiner Verwendung etc. beschrieben und definiert werden sollte. Indem der Spezifikateur eines Systems sein Verständnis der Welt in den Erkenntnisprozess einbringt, subjektiviert er die Systembildung. Laut Konstruktivismus ist es dennoch möglich, ein System objektiv zu spezifizieren. Allerdings ist dies ein aufwändiger Prozess, der erst abgeschlossen ist, wenn eine Gruppe Beteiligter einen Konsens bezüglich der Systemspezifikation erreicht hat.

Modell Der Modellbegriff ist ein weit verbreiteter, unterschiedlich eingesetzter Begriff. Aufgrund der vielfältigen Verwendung und der häufig daraus resultierenden Missverständnisse, propagieren manche Wissenschaftler, in der Softwaretechnik auf den Modellbegriff gänzlich zu verzichten, „es sei denn, die spezielle Bedeutung [...] geht aus dem Kontext eindeutig hervor“ ([Sch99], S. 134). Der Empfehlung folgend, wird an dieser Stelle eine Modelldefinition vorgenommen, die übliche Missverständnisse auszuräumen versucht.

Definition 2.1.2 (Modell)**.** *Ein Modell ist ein System, das ein anderes (reales) System abbildet.*

□

Die Begriffe *System* und *Realität* wurden bereits in den zwei vorangegangen Paragraphen definiert, es bleibt noch der Begriff der *Abbildung*.

Die Erkenntnistheorie als ein Fachgebiet der Philosophie beschäftigt sich unter anderem mit der Frage, inwieweit die menschliche Erkenntnis die Realität abbilden kann. Der menschliche Erkenntnisprozess betrachtet die Wirklichkeit als Urbild, das vom Mensch in Subjekte und Objekte abgebildet wird. Wie im Paragraphen „Realität“ bereits beschrieben, existieren unterschiedliche Thesen zur Erfassung und Abbildbarkeit von Objekten oder Systemen. Hierbei zu beachtende Aspekte sind zum Beispiel, zu welchem Zweck eine Abbildung der Realität erstellt wird, wie ähnlich die Abbildung der Realitätsstruktur ist und wie gut die Beschreibungsmittel zur Abbildung eingesetzt werden können.

Ein Abbild der Realität kann dazu dienen, das *Nach*bild messbarer Naturphänomene zu sein, um so, anhand der Abbildung, die Phänomene zu erklären oder Verhalten zu prognostizieren [Sch99]. Es kann auch *Vor*bild für noch zu entwickelnde Systeme sein. Dies ist zum Beispiel in der Softwaretechnik der Fall. Es wird dabei versucht, ein System zu beschreiben, das erst noch realisiert werden muss. Sinn des *Vor*bilds ist es, Klarheit über die Eigenschaften und das Verhalten des Systems zu gewinnen. Eine Unterscheidung nach der Ziel- und Zweckorientierung nennt man in der Modellbildung auch das **pragmatische Merkmal** [Sch10b]. Es gibt *Beschreibungsmodelle*, die Eigenschaften und Verhalten eines anderen (realen) Systems (Modells) beschreiben. Weitere *Vor*bilder sind die *Bezugsmodelle*. Sie bilden gewünschte Eigenschaften und Verhalten so ab, das auf sie bei der Entwicklung eines Systems/Modells Bezug genommen werden kann (zum Beispiel Prototypen). Es gibt *Erklärungsmodelle* als *Nach*bilder existierender Systeme/Modelle. Ein Erklärungsmodell des realen Systems „Regenwetter“ wäre zum Beispiel ein Regen-Simulationsmodell. Die letzte Gruppe der *Nach*bilder enthält die *Entscheidungsmodelle*. Sie helfen bei der Prognose von Eigenschaften und Verhalten von Systemen/Modellen. Ein Beispiel sind mathematische Modelle zur Prognose der Staugefahr an Feiertagen.

In der Mathematik ist der Begriff Abbildung wie folgt definiert.

Definition 2.1.3 (Abbildung [Heu09])**.** *X und Y seien zwei nicht-leere, aber ansonsten völlig beliebige Mengen. Unter einer* Abbildung *f von X nach (oder in) Y versteht man eine Vorschrift,*

die jedem $x \in X$ in vollkommen eindeutiger Weise genau ein $y \in Y$ zuordnet. Dieses dem Element x zugeordnete Element y bezeichnen wir auch mit $f(x)$ und nennen es den Wert der Abbildung f an der Stelle x oder das Bild *von x unter f, während x ein* Urbild *von $f(x)$ heißt.*

□

Verglichen mit den Bildbegriffen in der Philosophie ist das Urbild ein Vorbild und das Bild ein Nachbild. Die Philosophie versteht unter Urbild meist die Realität. Die Mathematik erlaubt jegliche Menge X aus Elementen, die nicht auf Zahlen eingeschränkt, aber wohldefiniert und unterscheidbar sein müssen. Die Definition 2.1.2 des Begriffs Modell erlaubt sowohl das „reale“ System als auch jedes andere System als Urbild der Modellbildung.

Neben dem pragmatischen Merkmal der Modellbildung wird das **Abbildungsmerkmal** [Sch10b] betrachtet, das sowohl in der Philosophie als auch der Mathematik eine Rolle spielt. Das Abbildungsmerkmal trennt Modelle in isomorphe (strukturgleiche) und homomorphe (strukturähnliche) Modelle. Die Definition von Homomorphismus hat seine Ursprünge in der Mathematik. Ob eine Abbildung ein Homomorphismus ist, wird in unterschiedlichen mathematischen Disziplinen für verschiedene mathematische Strukturen definiert. Als ein Beispiel wird im Folgenden die Definition für algebraische Gruppen vorgestellt. Die Eigenschaft einer Menge, eine Gruppe zu sein, muss an dieser Stelle nicht weiter diskutiert werden. Die Gruppe steht stellvertretend für weitere mathematische Strukturen wie Ringe, Körper etc. Weitere mathematische Strukturen finden sich beispielsweise in der Algebra, der Topologie oder der Geometrie. Jede Struktur erfordert gewisse Eigenschaften der zugrunde gelegten Mengen.

Definition 2.1.4 (Homomorphismus bei Gruppen [Fis10])**.** *Seien G und H Gruppen mit Verknüpfungen $\cdot$ und $\odot$, so heißt eine Abbildung $\phi : G \to H$* Homomorphismus *(von Gruppen), wenn*

$$\phi(a \cdot b) = \phi(a) \odot \phi(b) \text{ für alle } a, b \in G$$

□

Homomorphismen nennt man auch strukturverträgliche/strukturähnliche Abbildungen, da es egal ist, ob zuerst zwei Elemente in G verknüpft und dann abgebildet werden oder die Elemente zuerst nach H abgebildet und dann die Bilder verknüpft werden. Eine Abbildung zweier Strukturen ist ein Isomorphismus, falls sie Teile einer Struktur auf „bedeutungsgleiche“ Teile der anderen Struktur umkehrbar eindeutig abbildet. Dafür muss die Abbildung bijektiv und ein Homomorphismus sein und es muss eine inverse Abbildung existieren, die ebenfalls ein Homomorphismus ist. Die inverse Abbildung bildet die Elemente der Bildmenge auf ihre eindeutig bestimmten Urbildelemente ab [Heu09].

Überträgt man die Definition von Homomorphismus auf die Modellbildung, so sind die Gruppen mit Systemen zu ersetzen. Existiert ein Homomorphismus $\phi : M \to M'$, der das System/Modell M auf das System/Modell M' abbildet, so nennt man das Modell M' homomorph (isomorph entsprechend).

Die Isomorphie, die Strukturgleichheit fordert, liegt in der Philosophie so nicht vor. Ein strukturgleiches Abbild der Realität ist aus Sicht der Erkenntnistheorie nicht erreichbar, da es voraussetzen würde, dass die Menschheit die „absolute“ Wahrheit über die Realität kennt. Lassen sich in der Mathematik noch Aussagen wie „wahr auf einer Menge X“ oder „wahr in dem Modell M“ treffen, so ist eine Aussage „wahr in der Welt“ aus Sicht vieler philosophischer Denkschulen nicht möglich. „*Die Wahrheit auf die Welt bezogener Sätze hängt an empirischer Adäquatheit*“ [Hra93] und ist demnach keine absolute Aussage. Der Begriff der homomorphen Abbildung wird jedoch verwendet, wenn ein Abbild der Realität angefertigt wird, dass den Eigenschaften und dem Verhalten realer Objekte/Systeme zumindest ähneln soll.

Ein drittes Merkmal, anhand dessen Modelle klassifiziert werden können, ist das **Verkürzungsmerkmal** [Sch10b]. Es bezeichnet die abstrakte Modellierung eigentlich Detail treueren Wissens über das System. Damit ist nicht die unvollständige Erkenntnis der Eigenschaften und des Verhaltens eines Systems gemeint, die der Mensch bei der Abbildung der Realität immer

zu berücksichtigen hat. Es geht vielmehr um eine bewusste, zweckgebundene Verkürzung/ Abstraktion der abgebildeten Modellinformation. Abstraktion erlaubt ein besseres Verständnis der Abhängigkeiten des Systems von anderen Einflüssen wie weiteren Systemen, Beteiligten, seiner Verwendung etc. Sie erschwert allerdings das Studium interner Systemprozesse. Die Detaillierung der abgebildeten Information erlaubt ein besseres Verständnis interner Systemabläufe, verringert jedoch die Übersichtlichkeit aufgrund der steigenden Komplexität des Modells [UP91].

Die Gruppe der mathematischen Modelle spielt für vorliegende Arbeit eine wichtige Rolle, da sie sowohl das theoretische Fundament der Testfallermittlung, als auch der Computation Sequence Charts als Beitrag der Arbeit ist.

Die *mathematische Logik* nähert sich der Modellbildung nicht, wie in der Philosophie üblich, ausgehend von der Realität, die in einem Erkenntnisprozess in ein Modell übertragen wird, sondern gibt dem Modellierer Beschreibungsmittel zur Hand, die ihm bei der Modellierung einen klaren Rahmen vorgeben. Zuerst werden die Elemente einer „Beschreibungssprache“ festgelegt, die dann um Regeln und Axiome ergänzt werden. Die Sprache wird mit einer Syntax und Semantik untermauert, die bei der Modellbildung zu beachtende Regeln vorgibt. Kann das System anhand dieser Vorgaben nicht in ein Modell abgebildet werden, muss die Sprache, soweit das möglich ist, angepasst werden [HB01].

Die Vorgabe von Regeln zu Anfang der Modellbildung schränkt den Modellierer ein und verhindert dadurch eine „wahrheitsgetreue“ Abbildung der Realität. Dies beruht nicht allein auf den Grenzen der Beschreibungssprache, sondern auch auf den vorgegebenen Denkstrukturen. Häufig wird der Fehler begangen, die Systemmodellierung solange anzupassen, bis das System zur Modellierungsmethode passt und nicht die Methode zum System. Ohne Regeln können jedoch keine mathematischen Strukturen festgelegt werden, die dann auf einer Maschine umsetzbar sind. Besitzt man ein *mathematisches Modell* eines Systems, kann nachgewiesen werden, ob das Modell „gültig“ ist. Dafür wird gezeigt, ob es Aussagen über die Eigenschaften und Verhalten des Systems „erfüllt“. Die Frage der Gültigkeit und Erfüllbarkeit wird in Abschnitt 3.1.3 wieder aufgegriffen. Die formale Modellierung eines Systems bildet die Grundlage für den modellbasierten Test und auch für den Beitrag vorliegender Arbeit.

2.2 Der Begriff der Anforderung

Ist die Idee eines neuen technischen Systems entstanden, folgt die Systementwicklung. Die Systementwicklung hat zum Ziel, Benutzerwünsche über Software und/oder Hardware zu realisieren. Um die Benutzerwünsche kommunizieren zu können, werden daraus Anforderungen an das System abgeleitet. Anforderungen dienen dazu, folgende Fragen zu beantworten [RPWTU84, Sch10a].

- Warum soll das System entwickelt werden? (Zielsetzung)
- Was soll das System leisten können? (Funktions- und Leistungsumfang)
- Unter welchen Randbedingungen wird es funktionieren müssen? (Nebenbedingungen)

Nach IEEE 610.12 [Ass90], dem Standardglossar für Softwareentwicklungsterminologie, definiert sich der Begriff *Anforderung* wie folgt.

1. Eine Bedingung oder eine Fähigkeit, die von einer Person zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.
2. Eine Bedingung oder Fähigkeit, die eine Software erfüllen oder besitzen muss, um einen Vertrag, eine Norm oder ein anderes, formell bestimmtes Dokument zu erfüllen.
3. Eine dokumentierte Repräsentation einer Bedingung oder Fähigkeit wie in 1. oder 2. genannt.

Anforderungen können in zwei Gruppen aufgeteilt werden [Par10], die funktionalen und die nicht-funktionalen Anforderungen.

Funktionale Anforderungen beschreiben den Funktions- und Leistungsumfang. Funktionen werden dabei über ihren extern sichtbaren Effekt beschrieben, so wie Systemnutzer oder -umgebung das System wahrnehmen. Jede Funktion bestimmt den Wert einer oder mehrerer Ausgabedaten und jede Ausgabe wird durch genau eine Funktion mit Werten belegt [Hen80]. Zu den Funktionen und ihren Ausgaben zählen noch Systemeingaben und ihre Einschränkungen sowie Systemausgaben zur funktionalen Anforderungssammlung.

Die *nicht-funktionalen Anforderungen* können laut Partsch in vier Kategorien gegliedert werden ([Par10], S. 23ff):

- Qualitätsattribute der gewünschten Funktionen (z.B. Zuverlässigkeit, Wartbarkeit, Beschreibung unerwünschter Ereignisse [Hen80])
- Anforderungen an das implementierte System als Ganzes (z.B. Realisierung in Software und/oder Hardware, Topologie, Benutzerfreundlichkeit)
- Vorgaben für die Durchführung der Systemerstellung (z.B. Softwareentwicklungsprozess, Ressourcenplanung, ökologische, ökonomische, politische und soziale Randbedingungen)
- Anforderungen an Prüfung, Einführung, Betreuung und Betrieb (z.B. physikalische Einschränkungen, Abnahmebedingungen, Schulungen, Kundendienst)

Niedergeschrieben werden Anforderungen entweder formal oder nicht-formal in Spezifikationen. Eine *Spezifikationen* wird in IEEE 610.12 [Ass90] definiert als „A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component, and, often, the procedures for determining whether these provisions have been satisfied.“ Die ebenfalls im Standard enthaltene Definition der *Anforderungsspezifikation* legt die Betonung auf die Trennung von funktionalen und nicht-funktionalen Anforderungen. „Requirements specification - A document that specifies the requirements for a system or component. Typically included are functional requirements, performance requirements, interface requirements, design requirements, and development standards“ [Ass90].

2.3 Verifikation, Validation und Test

Wie im vorherigen Abschnitt erläutert, hat die Systementwicklung zum Ziel, Benutzerwünsche, die ein neues technisches System erfüllen soll, über Software und/oder Hardware zu realisieren. Dafür werden die Wünsche anhand von Anforderungen abgebildet und an die Systementwickler kommuniziert.

In dem Prozess der Anforderungserstellung können aufgrund der unterschiedlichen Sichtweisen auf das zu entwickelnde System Anforderungen entstehen, die den Benutzerwünschen nicht mehr entsprechen. Die Validation eines Systems oder einer Komponente soll zeigen, dass „das richtige System beziehungsweise die richtige Komponente“ entwickelt und allen Benutzerwünschen Rechnung getragen wird. Das IEEE Softwareglossar definiert *Validation* als „The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements“ [Ass90]. Eine Möglichkeit der Validation stellt das Review dar. Ein *Review* ist „ein Prozess oder eine Sitzung, während dessen ein Arbeitsergebnis oder eine Gruppe von Arbeitsergebnissen Projektmitgliedern, Managern, Anwendern, Kunden oder anderen interessierten Parteien zum Zweck der Kommentierung, Abstimmung oder Zustimmung vorgelegt wird“ [Sch10c].

Bei der Verifikation eines Systems oder einer Komponente wird davon ausgegangen, dass die Anforderungen bereits validiert wurden. Die Korrektheit der Anforderungen bezüglich der Frage „Entwickeln wir das richtige System?“ wird dabei angenommen. Die Verifikation beantwortet die Frage „Entwickeln wir das System richtig?“. Sie hat zum Ziel, zu prüfen, ob das entwickelte System (bzw. die Komponente) die Anforderungen erfüllt. IEEE 610.12 definiert *verification* als

„The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase“ [Ass90].

Um zu zeigen, dass ein System die am Anfang der Entwicklung an sie gestellte Anforderungen erfüllt, kann entweder ein Beweis der Korrektheit geführt oder das System kann einem Test unterzogen werden.

Bewiesen werden kann, ob ein Softwareprogramm seine formale Spezifikation erfüllt. Die Definition für *Korrektheitsbeweis* lautet „Proof of correctness - (1) A formal technique used to prove mathematically that a computer program satisfies its specified requirements. (2) A proof that results from applying the technique in (1)“ [Ass90]. Die Beweisführung benötigt ein mathematisches Verfahren, das als solches eine formale Datenbasis benötigt. Häufig werden Korrektheitsbeweise auf Implementierungsmodellen (oder Teilen davon) durchgeführt.

Abhängig von der Größe und Spezifikation des Systems kann nicht immer ein Korrektheitsbeweis geführt werden, da die mathematischen Verfahren entweder zu viel Rechenzeit und/oder Speicherplatz benötigen oder eine formale Spezifikation des Softwareprogramms nicht vorliegt. Das Testen ist ein weiteres Verfahren, um ein System zu verifizieren. Im Gegensatz zum Beweis bietet das Testen allerdings keine hundertprozentige Aussage über die Korrektheit des Systems. Die Definition für *Testen* als ein Prozess der Verifikation lautet „An activity in which a system is activated under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system“ ([Eng10], S. 19). Zum Zeitpunkt des Testens nennt man das zu prüfende System *system under test* (SUT). Die Umgebung, in der das SUT getestet wird, heißt *Testumgebung*. Beim Testen werden Testfälle gegen das SUT ausgeführt. Ein *Testfall* ist „(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. (2) (IEEE Std 829-1983 151) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.“ Der Testfall enthält Bedingungen an das SUT und seine Testumgebung, die zum Zeitpunkt des Tests erfüllt sein müssen, die Eingabeparameter und das erwartete Verhalten des SUT in Form von Ausgabeparametern. Entspricht das beobachtete Verhalten dem erwarteten Verhalten, ist der Testfall erfolgreich, ansonsten fehlgeschlagen und eine Untersuchung und Bewertung des abweichenden Verhaltens wird durchgeführt.

2.4 Modellbasiertes Testen

Unter dem Begriff Modellbasiertes Testen sind viele Methoden und Technologien zusammengefasst. Folgende Abschnitte erläutern zuerst die begrifflichen Grundlagen und stellen dann eine Klassifikation der unterschiedlichen Methoden auf.

Testmodell Ein Modell, das entwickelt wird, um es beim Testen einzusetzen, wird *Testmodell* genannt. Ein Testmodell ist zusammengesetzt aus einem *Vorbild* (Beschreibungsmodell) des zu testenden Systems und einem *Nachbild* (Erklärungsmodell) der Testumgebung. Für die vorliegende Arbeit sind die Testmodelle auf abstrakte Beschreibungen der zu testenden Eigenschaften des Testobjekts und der Testumgebung festgelegt. Ein Testmodell wird somit als ein abstrahiertes Verhaltensmodell verstanden, das die für den Test notwendigen funktionalen Aspekte beinhaltet. Die Modellierung von Anwendungsszenarien oder Testsequenzen soll ausgeklammert sein. Die Entscheidung über den Modellierungsformalismus des Testmodells hängt stark vom Charakter des zu modellierenden Systems ab.

Modelltest vs. Modellbasierter Test Der Modelltest hat zum Ziel, ein formales Modell zu verifizieren. Dabei werden Anforderungen an das Modell bzw. Eigenschaften (engl. properties) des Modells formal beschrieben und gegen das Modell geprüft. Ob ein Modell eine Eigenschaft erfüllt, kann anhand eines Model-Checkers entschieden werden [CGP00, CW96]. Verschiedene Model-Checking Verfahren, wie das Bounded Model-Checking und das explizite Model-Checking werden in Abschnitt 3.2 genauer vorgestellt.

Der Modellbasierte Test leitet auf Basis eines Modells Testfälle für ein, sich von dem Modell unterscheidendes Testobjekt ab. In diesem Fall ist das Modell Informationslieferant für zu testende Eigenschaften des Testobjekts, oder für Aktivitäten, durch die zu testende Eigenschaften beobachtbar werden. In welcher Form diese Eigenschaften oder Aktivitäten als Modell abgelegt sind, ist vorerst nicht festgelegt. Die allgemeine Definition von *Modell* erlaubt jegliche abstrakte Beschreibung des Testobjekts, auch natürlichsprachliche Spezifikationen. Um sich klarer von den klassischen Vorgehensweisen der Testfallermittlung abzugrenzen, seien an dieser Stelle allein formale und semi-formale Notationen betrachtet. In der Regel werden im Modellbasierten Test die Testfälle von einem *Testfallgenerator* automatisch abgeleitet.

Modellbasierter Testprozess Der modellbasierte Testprozess ist in Abbildung 2.1 zu sehen.

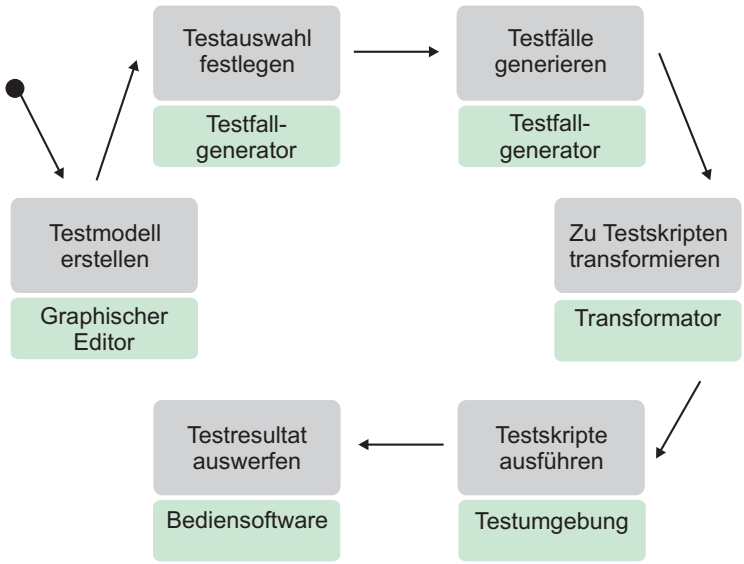


Abbildung 2.1: *Testprozess beim Modellbasierten Testen*

Im ersten Schritt des modellbasierten Testprozesses erstellt der Testingenieur ein Testmodell. Als Informationsgrundlage dienen ihm System-, Komponenten- und Testspezifikationen sowie gegebenenfalls Entwicklungsmodelle.

Im zweiten Schritt legt der Tester Strategien für die Testauswahl fest. Teststrategien können unterschiedlich motiviert sein. Die Notwendigkeit, innerhalb einer Stunde die Grundfunktionalität eines Teilsystems zu prüfen oder spezielle Sicherheitsanforderungen separat zu testen, können Teststrategien sein. Diese Strategien werden formalisiert dem Testfallgenerator zur Verfügung gestellt (bekannte Testauswahlkriterien werden in Abschnitt 2.4 vorgestellt).

Basierend auf diesen Testauswahlstrategien erzeugt der Testfallgenerator automatisch die gewünschten Testfälle. Ein Testfall entspricht aus Sicht des Testfallgenerators einer Sequenz von Zustandswechseln. Die Zustände und Bedingungen an Zustandswechsel stammen dabei aus dem Testmodell. Ein Testfall enthält sowohl Systembedienungen als auch erwartetes Systemverhalten und ist somit Systemstimulation und Vorhersage der Systemreaktion in einem. Welche unterschiedlichen Verfahren bei der Testfallgenerierung eingesetzt werden können, wird in Abschnitt 3.2 genauer betrachtet.

Die Wissenschaft kennt viele Varianten des modellbasierten Tests. Folgende Klassifikation unterteilt die unterschiedlichen Verfahren aufgrund ihrer Testauswahlkriterien, Notationen, Testfallgenerierungsalgorithmen und allgemeiner Unterschiede. Tabelle 2.1 lehnt sich an den Klassifikationsvorschlag von Pretschner, Legeard und Utting [ULP06, UL07] an. Eingeflossen in den Überblick sind Notationen aus der UML. Die Tabelle erhebt keinen Anspruch auf Vollständigkeit.

Klassifikation modellbasierter Testverfahren

a) Allgemeine Unterscheidungen	Umgebungsmodell ↔ SUT-Modell
	Entwicklungsmodell ↔ Testmodell
	Online ↔ Offline Auswertung
	Deterministisch ↔ Nicht-deterministisch Endlich ↔ Unendlich Mit Zeit ↔ Ohne Zeit Diskret ↔ Hybrid ↔ Kontinuierlich
b) Testauswahlkriterien	Anforderungsüberdeckung Strukturelle Modellüberdeckung Äquivalenzklassentests Grenzwerttests Datenüberdeckungskriterien Negativtests (Stabilitätstests) Annehmbarkeitstests Robustheitstests Stresstests Zeitgetriebene Kriterien Zielorientierte Kriterien Fehlerbasierendes Kriterium Benutzerdefinierte Testauswahl Explizite Testfallspezifikation
c) Notationen	*Strukturdiagramme*
	Architektur / Topologie *Verhaltensdiagramme*
	Aktivitätsdiagramm Zustandsdiagramm Use Case Diagramm Kommunikationsdiagramm Sequenzdiagramm Statistische Notation
d) Testfallgenerierung	Manuell Zufällig Graph-Such-Algorithmen (Bounded) Model-Checking Symbolische Ausführung Formale Verifikation

Tabelle 2.1: *Klassifikation modellbasierter Testverfahren*

a) Allgemeine Unterscheidungen

Beschreibt ein Testmodell die Struktur und das Verhalten des zu testenden Systems, spricht man von einem *system under test (SUT)* Modell. Das Testmodell kann in diesem Fall auch als Vorbild für die Testauswertung dienen. Testfälle des SUT Modells werden dabei immer unter der Annahme generiert, dass die Modellierung des Verhaltens des SUT korrekt ist. Eine Herausforderung der SUT Ansätze liegt in der Auswahl einer Folge von Systemstimuli, die es ermöglicht, modellierte Systemeigenschaften beobachten zu können. Eine zweite Herausforderung liegt in der Festlegung einer Notation, die alle zu testenden Systemeigenschaften auf einem adäquaten Abstraktionslevel beschreiben kann.

Testmodelle können auch die Bedienung oder *Umgebung des SUT* beschreiben. Aus der Modellierung der Bedienung des SUT lassen sich mögliche Stimulifolgen generieren, die eine maximale Bedienüberdeckung zum Ziel haben. Fehlbedienung können dabei ebenfalls in Betracht gezogen werden. Eine Teilmenge dieser Verfahren modelliert vorgegebene abstrakte Testabläufe und konkretisiert diese bei der Testdatengenerierung. Modelliert man allein die Umgebung des SUT, muss zudem eine Bewertung des beobachteten Systemverhaltens durchgeführt werden.

Das Testmodell ist mit einem Implementierungsmodell vergleichbar. Der Abstraktionsgrad von Testmodell und Implementierungsmodell unterscheidet sich allerdings je nach Teststufe. Die beobachtbaren Systemeigenschaften während des Systemintegrationstests sind zu gering, um lauffähige Steuergerätesoftware anhand ihrer Modellierung zu generieren. Umgekehrt sind Implementierungsmodelle zu detailliert, um ohne zu großen Aufwand Testfälle für den Systemintegrationstest abzuleiten. Der Testingenieur müsste zuerst eine Abstraktion des Implementierungsmodells vornehmen. Allerdings ist diese Vorgehensweise aus Sicht der Testfallermittlung kritisch, da dadurch hauptsächlich der Codegenerator getestet wird. Fehlerhafte Modellierungen werden dabei übersehen.

In vorliegender Arbeit wird die Erstellung eines separaten Testmodells durch den Testingenieur propagiert, um Testerfahrung einfließen zu lassen, sowie Spezifikations- und Implementierungsfehler aufdecken zu können.

Ein weiterer Aspekt unterscheidet die modellbasierten Testverfahren nach ihrer Auswertung des System-Soll-Verhaltens. Das Testergebnis kann entweder *online*, das heißt während der Testausführung, auf Korrektheit hin überprüft werden oder *offline*, anhand aufgezeichneter Informationen, nachträglich gegen die Spezifikation verglichen werden.

Wenn der Folgezustand eines Systems bestimmt durch Eingabe und Vorzustand immer eindeutig gegeben ist (Vorzustand + Eingabe $\rightarrow$ Folgezustand), spricht man von einem *deterministischen* Modell, ansonsten von einem *nicht-deterministischen* Modell.

Ein Modell heißt *endlich*, wenn es nur endlich viele Systemzustände beschreibt, ansonsten heißt es *unendlich*.

Der Werteverlauf von Daten charakterisiert modellbasierte Testmethoden hinsichtlich der Berechenbarkeit der Testdatenbelegungen. Sind die Wertemengen der Daten endlich oder zumindest abzählbar unendlich spricht man von *diskreten* Modellen. Können Werte auch kontinuierlichen Funktionen oder Kurven folgen, werden die Modelle als *kontinuierlich* bezeichnet und erlauben die Einführung der Stetigkeit. Eine Mischung von diskreten und kontinuierlichen Wertveränderungen heißt *hybrid*. Nähere Informationen finden sich unter [BFPT06].

Spielt die Zeit eine Rolle bei der Modellierung eines Systems, stehen zwei Möglichkeiten der Ausdrucksstärke zur Verfügung. Die Zeit kann als *diskretisiert* oder *kontinuierlich* verstanden werden. Kontinuierliche Zeitbegriffe werden auch als stetig bezeichnet.

In endlichen Automaten mit einem *diskreten Zeitbegriff* wird eine globale Uhr eingeführt, die alle beteiligten Komponenten *synchronisiert*. Die Uhr kann nur positive, ganzzahlige Werte annehmen. Die eigentlich kontinuierliche Zeit wird diskretisiert, indem sich das System nur in ganzzahligen Zeitschritten fortbewegen kann [CGP00].

Um kausal unabhängige Ereignisse in einem *asynchronen* System beschreiben zu können, braucht der Testingenieur einen kontinuierlichen Zeitbegriff, wie er zum Beispiel bei Timed Automata oder Timed Statecharts (siehe dazu Abschnitt 3.1.2) verwendet wird. Endlich viele unabhängige Uhren laufen dabei in parallelen Automaten. Die Uhren synchronisieren sich nicht

und können beliebig auf Null zurückgesetzt werden. Mit Hilfe so genannter *regions* oder *zone graphs* lässt sich eine Schachtelung der Zeitverläufe definieren, der auch einen kontinuierlichen Zeitbegriff und voneinander unabhängig fortschreitende Uhren bei der Testfallermittlung erlaubt [Pel09].

b) Testauswahlkriterien

Eine der größten Herausforderungen der Modellbasierten Testfallermittlung ist das Steuern des Testfallgenerators bei der Testauswahl. Angelehnt an Testziele und Teststrategien des Unternehmens und die Möglichkeiten der jeweiligen Teststufe, soll eine Testfallmenge gewissen vorgegebenen Kriterien genügen.

Aus der Gesamtheit aller funktionsorientierter und strukturorientierter Testmethoden sind folgende Testauswahlkriterien zum Beispiel für den modellbasierten Systemintegrationstest relevant:

- Anforderungsüberdeckung
- Strukturelle Modellüberdeckung
  - Eingabeüberdeckung
  - Ausgabeüberdeckung
  - Zustandsüberdeckung
  - Transitionsüberdeckung
  - Modifizierte Zustands-/Transitionsüberdeckung (Modified Condition/Decision Coverage) [RH03]
  - Entscheidungstabellenüberdeckung (Look up table Coverage)
- Annehmbarkeitstests (Mindestfunktionalität)
- Robustheitstests, Stresstests, Grenzwerttests, Stabilitätstests
- Benutzerdefinierte Testauswahlstrategien
  - Priorisierbarkeit (z.B. Sicherheitsrelevanz)
  - Prozentaussagen (100% Anforderungsüberdeckung)
- Explizite Testfallspezifikation
- Zeitgetriebene Kriterien (z.B. max. Dauer auszuführender Testfälle = 1h)
- Zielorientierte Kriterien (z.B. Akzeptanztestsuite)
- Fehlerbasierendes Kriterium
- Statistische Testfallgenerierungsmethode
- Datenüberdeckungskriterien

Die Auswahl der Testfälle orientiert sich bei der klassischen Testfallermittlung an den durch Normen, wie der ISO 26262 [fS09] geforderten und bewährten Verfahren. Zu den gängigsten Vorgehensweisen zählen die Äquivalenzklassenanalyse, die Grenzwertanalyse und die erfahrungsbasierte Testauswahl.

Die Äquivalenzklassenanalyse teilt den Raum der Systemeingaben (Zeit- und Wertebereiche) in Klassen auf, für deren Elemente gelten muss, dass sie ein identisches Systemverhalten auslösen [WJ91]. Die Klassen dürfen sich überschneiden, es wird jedoch meist versucht, dies zu vermeiden, um eine redundante Datenbelegung zu verhindern. Untersützung bei der Testauswahl

aufgrund von Äquivalenzklassenanalyse bietet die Klassifikationsbaum-Methode. Sie ermöglicht es, anhand einer n-fachen Kombination der einzelnen Repräsentanten einer jeden Äquivalenzklasse, Testfälle zu bestimmen [GG93]. Die Software CTE erlaubt eine übersichtliche, und teilweise automatisierte Arbeitsweise bei dem Einsatz der Klassifikationsbaum-Methode [PW07].

Grenzwertanalysen sind motiviert durch die Häufung von Softwarefehlverhalten an den Grenzen der Äquivalenzklassen und Wertebereiche. Meist werden bei der Grenzwertbildung sehr große und sehr kleine Datenbelegungen gewählt sowie Werte an den Grenzen der Äquivalenzklassen. Existieren keine Äquivalenzklassen für ungültige Datenbelegungen, fallen diese ebenfalls in den Bereich der Grenzwertanalyse. Sie hat ebenfalls die Identifikation und Auswahl ungültiger Eingaben zum Ziel.

Bei der erfahrungsbasierten Testauswahl vermischen sich theoretisch fundierte Erfahrungen der Fehlerhäufung in Software mit der Erfahrung der Tester bei der Fehlerfindung. Die als intuitive und unsystematisch klassifizierte Vorgehensweise garantiert den Einfluss häufig auftretender Fehlerklassen bei der Testfallauswahl.

Weitere Übersichten und Definitionen der einzelnen Kriterien können Utting und Legeard [UL07], Abdurazik et al. [AADO00], Fraser und Wotawa [FW08a], Friske et al. [FSW08], Gaston und Seifert [GS05] oder Offutt et al. [OXL99] entnommen werden.

c) Notationen

Folgende Auswahl ist hauptsächlich Rupp et al. [RHQ+05] entnommen.

Architektur- und Topologiemodelle beschreiben den Aufbau und die Vernetzung sowohl von Hardware- als auch von Softwarekomponenten.

Anwendungsfallmodelle aus der UML umfassen Use Case-, Sequenz- und Kommunikationsdiagramme.

In *Use Case Diagrammen* steht die Interaktion des Nutzers (Mensch oder Maschine) und des Systems im Vordergrund. Die Beschreibung kann durch Pre- und Post-Conditions, Bedingungen, die während der Ausführung des Use Cases wahr bleiben müssen und Verweise auf Anforderungen angereichert werden.

Sequenzdiagramme beschreiben die Interaktionen von Objekten über die Zeit hinweg. Sie basieren auf dem Begriff der Szenarien. Forbrig [For07] definiert das Szenario als „eine spezifische Folge von Aktionen, die zur Verdeutlichung des Verhaltens eines Systems dient“. Üblicherweise zeigen sie den Nutzer und die Objekte, mit denen er während eines Use Cases interagiert. Das Sequenzdiagramm erlaubt somit die Fokusierung auf zeitliche Abläufe, die auch Parallelitäten enthalten können. Ähnlich zu den Sequenzdiagrammen der UML enthalten Message Sequence Charts (MSC), definiert in einem internationalen Standard (ITU-T Z.120 [Uni99]), die Kommunikation von Prozessen mit Hilfe von Nachrichten und Ereignissen über die Zeit hinweg [MP05]. Sequenzdiagramme und MSCs werden in Abschnitt 3.3.2 genauer betrachtet.

Sowohl *Kommunikationsdiagramme* als auch Sequenzdiagramme können herangezogen werden, um den die Interaktion und den Verlauf der Kommunikation von Objekten abzubilden. Prinzipiell beschreiben beide Diagrammtypen die selbe, wenn auch nicht die identische, Information. Bei einem Kommunikationsdiagramm steht die strukturelle Organisation der agierenden Elemente im Vordergrund während bei einem Sequenzdiagramme die zeitliche Sequenz hervorgehoben wird [Gom11].

Aktivitätsdiagramme stellen den Wechsel der Aktivitäten dar und beschreiben, wie die Anwendungsfallmodelle auch, dynamisches Verhalten. Ablaufbeschreibungen innerhalb eines Systems legen den Fokus dabei auf Aktivitäten, und nicht auf Zustände, wie es bei *Zustandsdiagrammen* der Fall ist. Zustandsdiagramme beschreiben den Lebenszyklus von Objekten und ihre Reaktion auf das Umfeld. Aktivitätsdiagramme beschreiben den Ablauf von Algorithmen und Verarbeitungsschritten.

Zustands-Transitions-Diagramme werden in Abschnitt 3.1.2 vorgestellt.

Sind in einem nicht-deterministischen System manche Zustandswechsel wahrscheinlicher als andere, können die Zustandsübergänge mit Wahrscheinlichkeiten versehen werden. Testmodelle, die Wahrscheinlichkeiten beinhalten, werden unter dem Begriff der *statistischen Notationen*

zusammengefasst. Beschreibt der Testingenieur zum Beispiel die Bedienung eines Systems mit Hilfe eines Zustands-Transitions-Diagramms und werden alle Bedienvarianten durch den Nutzer gleich wahrscheinlich durchgeführt, kann die Traversion des Automaten anhand von Wahrscheinlichkeiten an Transitionen beeinflusst werden. Zustandsautomaten mit Übergangswahrscheinlichkeiten an den Transitionen werden als *Markovketten* bezeichnet. Soll ein nichtdeterministischer Automat modelliert werden, kann an Zustandsübergängen die Wahrscheinlichkeit annotiert werden, mit der das System die einzelnen Transitionen überschreitet [Lun06]. Man unterscheidet diskrete und kontinuierliche Markovketten, je nachdem, ob bekannt ist, wann eine Transition ausgeführt wird (kontinuierlich) oder nicht (diskret) [Lig09].

d) Testfallgenerierung

Die manuelle Auswahl von Testfällen ist eine weit verbreite Vorgehensweisen. Dabei orientiert sich der Tester meist an Testauswahlkriterien wie der Anforderungsüberdeckung, der Überdeckung von Äquivalenzklassen oder der Überdeckung von Grenzwerten. Die manuelle Auswahl kann aufwändig und fehleranfällig sein und ist je nach Formalisierungsgrad der Testbasis und der Komplexität der zu lösenden Aufgabe mehr oder weniger automatisierbar. Eine Auswahl zum größten Teil automatischer Testfallermittlungsverfahren wird im Folgenden vorgestellt.

Testfallgenerierungsverfahren unterscheiden sich dahingehend, auf welche Art die Stimulifolgen zur Überdeckung von Testauswahlkriterien (Abschnitt 2.4) berechnet werden. Eine Unterscheidung der Verfahren trennt sie in *statistische* Vorgehensweisen und *zielgerichtete* Vorgehensweisen. Eine bekannte statistische Vorgehensweise wählt die Testfälle mit Hilfe von Markovketten aus. Den Zufall mit in die Testauswahl aufzunehmen, trägt dem meist nicht-deterministischen Einsatz von Systemen Rechnung. Eine zufällige Testauswahl (engl.: random testing) wählt aus den möglichen Inputwerten zufällig Eingangsdaten aus. Statistische Verfahren legen der Veränderung von Eingangsdaten ein statistisches Modell zugrunde, das bei der Testfallgenerierung berücksichtigt wird [DGG04].

Zielgerichtete Testfallgenerierungsverfahren definieren im vorhinein Ziele und überlassen die Auswahl der Testfälle zum Erreichen dieses Ziels nicht dem Zufall. In der Klasse der zielgerichteten Verfahren kann man zwischen *statischen* und *dynamischen* Verfahren unterscheiden. Die statischen Verfahren berechnen aus einer statischen Repräsentation des Testmodells oder Systems eine Testdatenbelegung. Zu den statischen Verfahren zählt die statische Codeanalyse [Lig09].

Dynamische Verfahren führen das Testmodell/System aus oder simulieren es und entscheiden während oder nach einem Systemlauf, wie die nächsten Testdaten gewählt werden sollen, um dem Ziel näher zu kommen. Das evolutionäre Testen zählt als ein Beispiel in die Kategorie der zielgerichteten, dynamischen Testfallgenerierungsverfahren. Beim evolutionären Test wird die Suche nach Eingangsdaten zusammen mit einer zu zeigenden Systemeigenschaft als Optimierungsproblem aufgefasst, das mit evolutionären Algorithmen gelöst wird. Dabei wird solange eine geeignete Auswahl an Eingangsdaten ausgewählt und mutiert, bis ein Verstoß gegen die anfangs festgelegte Systemeigenschaft festgestellt werden kann. Auf diese Art wird versucht, dem System ein Fehlverhalten nachzuweisen [Weg01]. *Model-Checking* ist ebenfalls ein Verfahren, das zur zielgerichteten Testfallgenerierung eingesetzt werden kann. In Abschnitt 3.2 werden die Varianten Expliztes Model-Checking, Model-Checking auf binären Entscheidungsbäumen und Bounded Model-Checking genauer vorgestellt.

2.5 Von der Anforderung zum Testmodell

Um das Konzept der Arbeit im Rahmen einer Anwendungsdomäne anschaulich vermitteln zu können, wird im vorliegenden Abschnitt ein spezielles Sofwareentwicklungsvorgehen ausgewählt. Da diese Arbeit in Zusammenarbeit der Universität Ulm und der DAIMLER AG entstand, fiel die Entscheidung auf die Entwicklung von Elektrik/Elektronik (E/E)-Systemen in der Automobilindustrie.

Es sei an dieser Stelle der Kreativität des Lesers überlassen, den Übertrag der vorgestellten Ideen auch in andere Software/Hardware Entwicklungstechniken und Domänen zu leisten.

Das zeitliche Verhalten von verteilten Systemen in einem Automobil ist ein maßgeblicher Bestandteil der Systemmodellierung. Ein Testmodell muss es ermöglichen, die Zeit adäquat abzubilden. Häufig existieren harte Echtzeitanforderungen an ein System. Die im folgenden vorgestellte Modellierung des Systems „Blinken“ erlaubt den Einsatz von sogenannten *Uhren*, die als Variablen $t \in \mathbb{R}^+$ auf den positiven, reellen Zahlen eingesetzt werden. Die Möglichkeit, diese Uhren zur Modellierung von Echtzeitanforderungen zu verwenden, ist für den Ansatz vorliegender Arbeit nur zweitrangig. Allein die Möglichkeit zur Modellierung zeitlicher Anforderungen ist Teil weiterer Betrachtungen. Die Bedeutung der Echtzeit in automobilen Systemen kann in [KPK+10] nachgelesen werden.

Tabelle 2.2 zeigt einen Auszug aus einem Systemlastenheft der Blinkfunktionalität eines Automobils. Es enthält natürlichsprachliche Systemanforderungen. Jede Einzelanforderung ist an dem vorangestellten SLH-xxx zu erkennen.

1.	**Blinkfunktionalität**
1.1	**Richtungsblinken**
1.1.1	**Funktionsvoraussetzungen**
SLH-859:	Voraussetzung für die Aktivierung von Richtungsblinken ist Schlüssel auf `Radiostellung` oder `Motor an`.
1.1.2	**Funktionsauslöser**
SLH-860:	Die Blinkhebelstellung für Richtungsblinken `Links` oder `Rechts` wird eingelesen und entprellt.
1.1.3	**Funktionsbeschreibung**
SLH-865:	Richtungsblinken kann durch Warnblinken unterbrochen werden.
SLH-866:	Ein durch Warnblinken unterbrochenes Richtungsblinken wird bei Wegfall der Warnblinkanforderung wieder aufgenommen.
SLH-867:	Ein durch Warnblinken unterbrochenes Richtungsblinken wird bei Wegfall der Richtungsblinkanforderung zurückgenommen.
1.1.4	**Abbruchbedingung**
SLH-862:	Wegfall der Funktionsvoraussetzung.
SLH-863:	Eine neue Blinkanforderung deaktiviert die aktuelle Blinkanforderung.

Tabelle 2.2: *Auszug Systemlastenheft Blinkfunktionalität*

Anforderungen werden meist *positiv* definiert, d.h. es wird beschrieben, wann und wodurch eine Funktion aktiviert wird. Wie sich das System verhalten soll, falls die Aktivierung nicht eintritt, wird implizit angenommen. Die Anforderung beschreibt, wann das Fahrzeug blinken soll. Dies geschieht, sobald der Blinkhebel die Stellung `Blinken links` oder `Blinken rechts` annimmt. Es wird nicht beschrieben, wie sich das Fahrzeug verhalten soll, wenn der Blinkhebel weder nach links noch nach rechts zeigt. In diesem Fall wird implizit davon ausgegangen, dass das Fahrzeug nicht blinkt. Diese *negative* Sicht auf das System wird bei einer formalen Systemspezifikation, im Gegensatz zur natürlichsprachlichen Spezifikation, automatisch erzwungen.

In einem Automobil charakterisiert sich das zu testende Zusammenspiel der E/E Komponenten als ein reaktives System, das parallele, meist nicht terminierende Abläufe in Echtzeit enthält. Beim Systemintegrationstest liegt der Fokus sowohl auf dem Systemzustand zu einem definierten Zeitpunkt als auch auf dem Systemverhalten über die Zeit hinweg, das heißt, auf der Veränderung des Systems bei einem Zustandswechsel. Ein Zustandswechsel lässt sich anhand des Vorzustands und des Folgezustands eines Systems beschreiben [CGP00]. Da reaktive Systeme ihre Umgebung beeinflussen oder von ihr beeinflusst werden, ist das Überwachen der Zustandswechsel von großer Bedeutung. Nicht allein der Input und erwartete Output zählen, sondern auch die Wirkkette, sprich die beobachtbaren Zustandswechsel während der Funktionsausführung.

Betrachtet man die Charakteristika zusammenfassend, bieten sich *Statecharts* [Har87], *Ti-*

med Automata [Alu98] und *UML Zustandsautomaten* [RHQ+05] zur Modellierung des Systems an. Eine tiefergehende Betrachtung der einzelnen Modellierungsnotationen findet in Abschnitt 3.1 statt. Ergänzt wird das Testmodell als Spezifikation des Systemverhaltens um Testinformationen, wie zum Beispiel im Test erlaubte Toleranzen der Signalübertragung und um Komponenten der Testumgebung, die speziell beim Testen benötigt werden.

Folgende Abbildungen 2.2 bis 2.6 zeigen eine grobe Übersicht über die Modellierung der Blinkfunktionalität aus Tabelle 2.2 als Statecharts. Die Vorgehensweise der Testmodellerstellung ist nur beispielhaft und kann als Anhaltspunkt für andere Systemmodellierungen dienen.

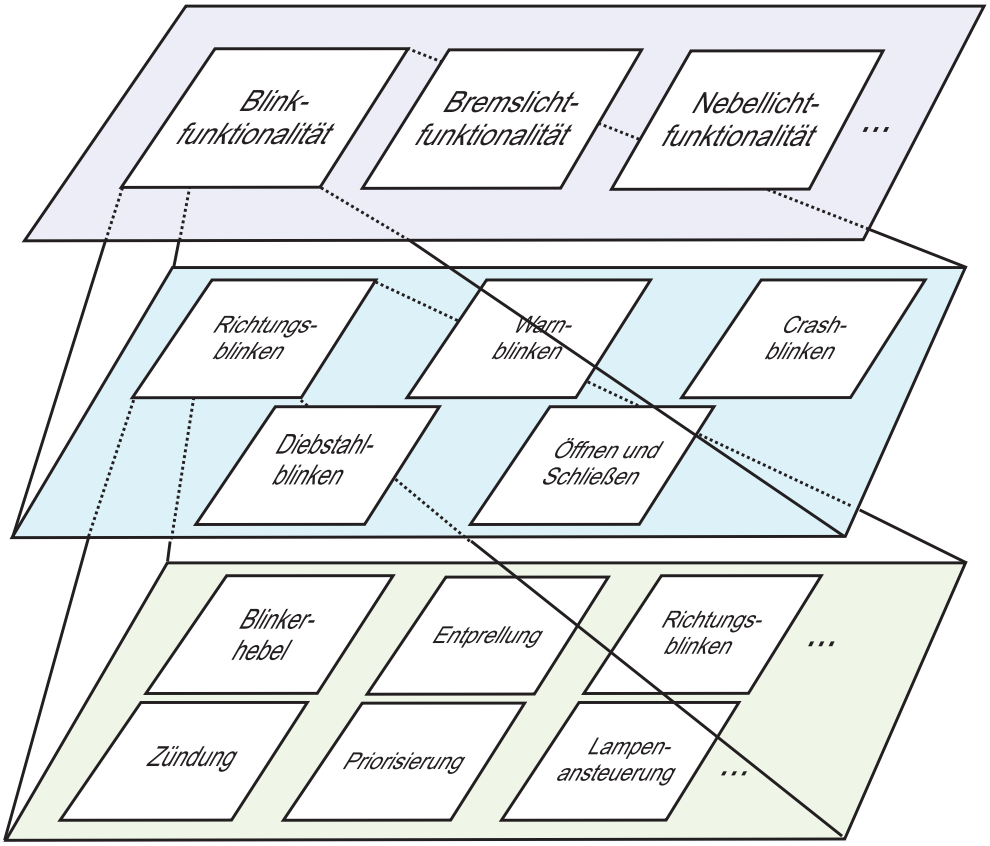


Abbildung 2.2: *Hierarchieebenen innerhalb der Außenlichtfunktionalität*

Abbildung 2.2 bildet die innerhalb des Systemlastenhefts angelegten Hierachieebenen im Testmodell ab. Um die Diskussion mit anderen am Entwicklungsprozess beteiligten Parteien über die Inhalte des Testmodells zu erleichtern, empfiehlt es sich, im Lastenheft vorgegebene Strukturen bei der Modellierung zu berücksichtigen. Auf unterster Ebene der Hierarchie befinden sich die Statecharts des Testmodells. Bei der späteren Testfallgenerierung werden allein die Statecharts eingelesen und verarbeitet, die Hierarchie dient der Übersicht, dem besseren Verständnis des Testmodells und der strukturierten Deklaration von Variablen.

Alle Statecharts laufen parallel und kommunizieren über geteilte, globale Variablen, wobei der Grundsatz für Variablendeklarationen „so lokal wie möglich, so global wie nötig“ gilt. Um den Sichtbereich von Variablen leichter zu strukturieren, helfen die unterschiedlichen Hierarchieebenen. Sie erlauben es, Variablen nur lokal für ein Statechart zu deklarieren. Die meisten verwendeten *Uhr*-Variablen werden auf Statechartebene deklariert. In Abbildung 2.3 und Abbildung 2.4 sind Uhr-Variablen mit *t* bezeichnet. In den abgebildeten Zustandsautomaten steht in den eckigen Klammern die Bedingung an den Übergang von einem Zustand zu einem anderen. Hinter dem Schrägstrich an einer Transition stehen Aktionen, die ausgeführt werden, wenn der Übergang genommen wird. Sollen beim Eintritt in einen Zustand Aktionen ausgeführt werden, wird

dies hinter dem Wort *entry/* vermerkt. Sollen Aktionen wiederholt durchgeführt werden, wird der Begriff *do/* vorangestellt.

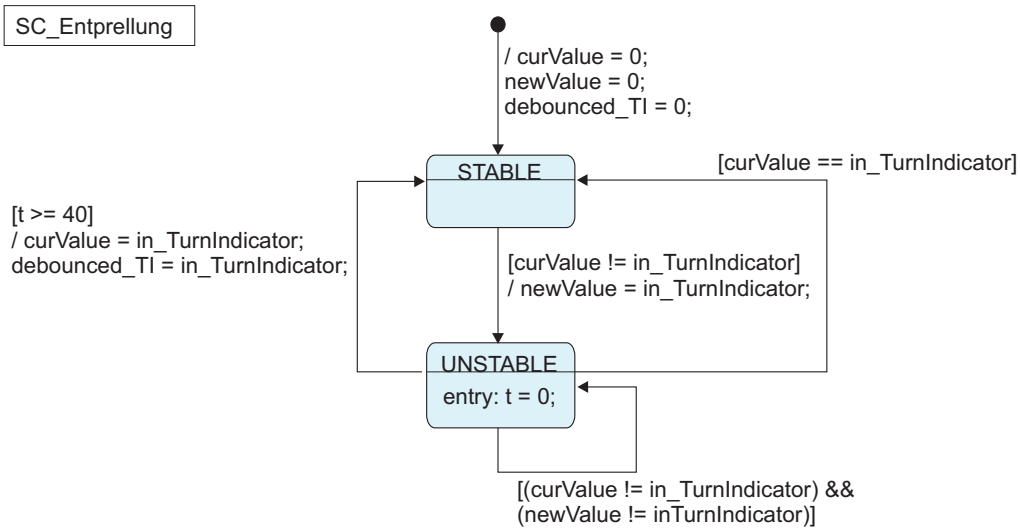


Abbildung 2.3: *Zustandsautomat Entprellung*

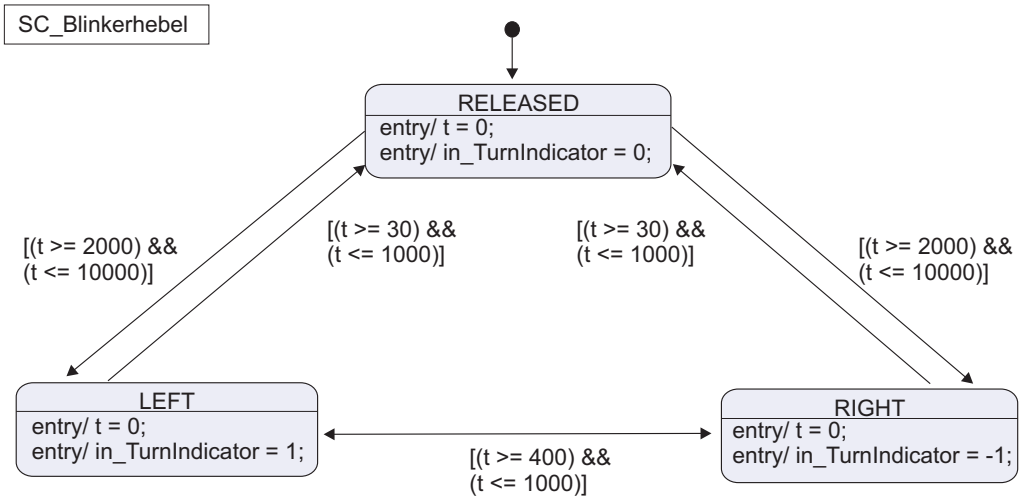


Abbildung 2.4: *Zustandsautomat der Blinkerhebelbedienung*

Der Automat `SC_Blinkerhebel` zählt zu den so genannten *Umgebungsautomaten* des Testmodells. Ein Trennen von Automaten, die die Systemumgebung modellieren von den Modellen des zu testenden Verhaltens spiegelt die Trennung von Stimuli und Soll-Verhalten wider. Automaten der Umgebung beschreiben sowohl akzeptable Systemstimulation als auch Fehlbedienungen. Meist sind diese Automaten nicht-vorhersagbar, da die Systembedienung nicht vorhersagbar ist. Dies ist allerdings nicht immer der Fall. Werden während des Tests Simulationsmodelle eingesetzt, wie zum Beispiel ein Modell zur Simulation von Regen, wird das Verhalten des Regenmodells innerhalb der Testsystemumgebung soweit modelliert, dass eine Soll-Verhaltensberechnung möglich wird. Der Kreis zwischen Umgebung und System muss auch innerhalb eines Testmodells geschlossen bleiben. Diese Modelle sind vorhersagbar, da zu jedem Systemzustand ein eindeutiger Folgezustand berechnet werden kann. Neben diesen Modellen sind auch alle Automaten des Systemverhaltens vorhersagbar. Systeme in der Automobil- aber auch der Flug- und Bahnbranche müssen jederzeit in ihrem Verhalten vorhersagbar sein.

Die Umgebungsautomaten grenzen bei der Testfallgenerierung die erlaubten Stimulivariationen ein. Der Blinkerhebel in Abbildung 2.4 darf nur innerhalb des an den Transitionen vorgegebenen Zeitintervalls bewegt werden. Durch das Bewegen des Hebels und der Freiheit, die Dauer der Bewegung und Wertebelegung der Variable weitestgehend beliebig zu wählen, kann ein Testfallgenerator einen Zustandswechsel im System stimulieren und so gewünschte strukturelle Modellüberdeckungen, wie das Überdecken aller Transitionen erreichen. Die Kunst liegt nun darin, die Wertebelegung der Stimulivariablen und deren zeitlichen Verlauf so zu wählen, dass eine maximale Modellüberdeckung erreicht wird. Da der Testfallgenerator durch die Umgebungsautomaten nicht vollkommen frei ist, erschwert sich diese Aufgabe zusätzlich. Eine vollkommen freie Datenbelegung würde allerdings zu Testfällen führen, die einerseits unrealistisch und andererseits auch falsch sein könnten (durch zu schnelle Bewegungen des Blinkerhebels zum Beispiel, die der mechanisch möglichen Bewegungsgeschwindigkeit widersprechen). Umgebungsmodelle müssen in ihrem Verhalten nachgebildet werden. Wie dies effizient gemacht werden kann, ist zur Zeit noch Teil der Forschung. Bei Umgebungsmodellen wie dem Motormodell darf das zu testende System sich erst weiter bewegen, wenn eine eingestellte Zielgeschwindigkeit auch wirklich durch das Motormodell erreicht wurde. Die Zielgeschwindigkeit kann dadurch nicht beliebig schnell variiert werden. Andere Umgebungsmodelle sind flexibler. Solange ein Regenmodell Regen simuliert, kann zwar, sollte aber nicht immer eine weitere Systemstimulation erfolgen. Die Vorgabe der Systemstimulation und ihrer Grenzen ist somit ein wichtiger Aspekt bei der Testfallgenerierung.

Abbildung 2.4 enthält im `SC_Blinkerhebel` Statechart Transitionen zwischen den Zuständen LEFT und RIGHT. Diese Transitionen beschreiben den direkten Übergang des Blinkerhebels von links nach rechts und umgekehrt. In einem funktionierenden System ist das physikalisch nicht möglich. Der Hebel muss immer über die RELEASED Stellung gehen. Eine Fehlbedienung ist aufgrund mechanischer Probleme allerdings denkbar. Transitionen, die „unrealistisches“ oder „fehlerhaftes“ System- und Umgebungsverhalten beschreiben, werden als spezielle *Robustheitstransitionen* markiert, um Sonderfälle auch getrennt von „normalem“ Verhalten generieren zu können.

Zustandsautomat `SC_Richtungsblinken` in Abbildung 2.5 beinhaltet zwei *Uhr*-Variablen *t* und *s*. Beide Uhren können unabhängig voneinander auf Null gesetzt und ausgewertet werden. Mit Hilfe unabhängiger Uhren lässt sich asynchrones Verhalten modellieren. Es gibt keine globale Uhr, auf die sich die einzelnen Uhren synchronisieren können. Die an der Blinkfunktionalität beteiligten Komponenten in einem Fahrzeug synchronisieren sich ebenfalls nicht. Es gibt einen zentralen Blinkmaster, der dafür sorgt, dass alle dieselben Informationen enthalten, die Ausführungen und Ausführungsdauer der einzelnen Komponenten wird jedoch nicht von ihm synchronisiert. Dass zum Beispiel alle am Blinken beteiligten Lampen synchron blinken, muss deshalb durch einen Synchronisationstest überprüft werden.

Zustandsautomat `SC_Warnblinken` in Abbildung 2.6 beschreibt des Verhalten des Systems Außenlicht, wenn Warnblinken über den Warnblinkschalter ausgelöst wird.

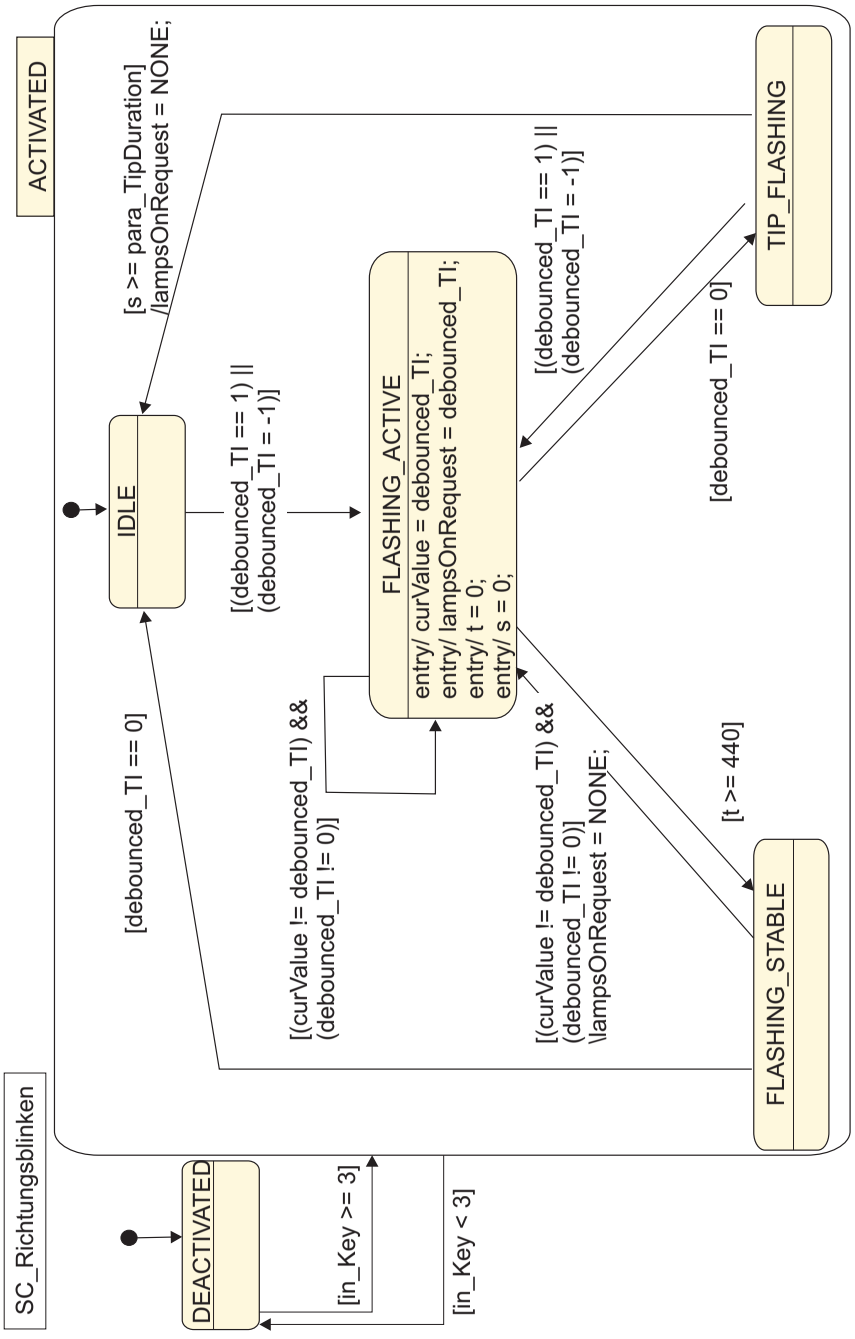


Abbildung 2.5: *Zustandsautomat Richtungsblinken*

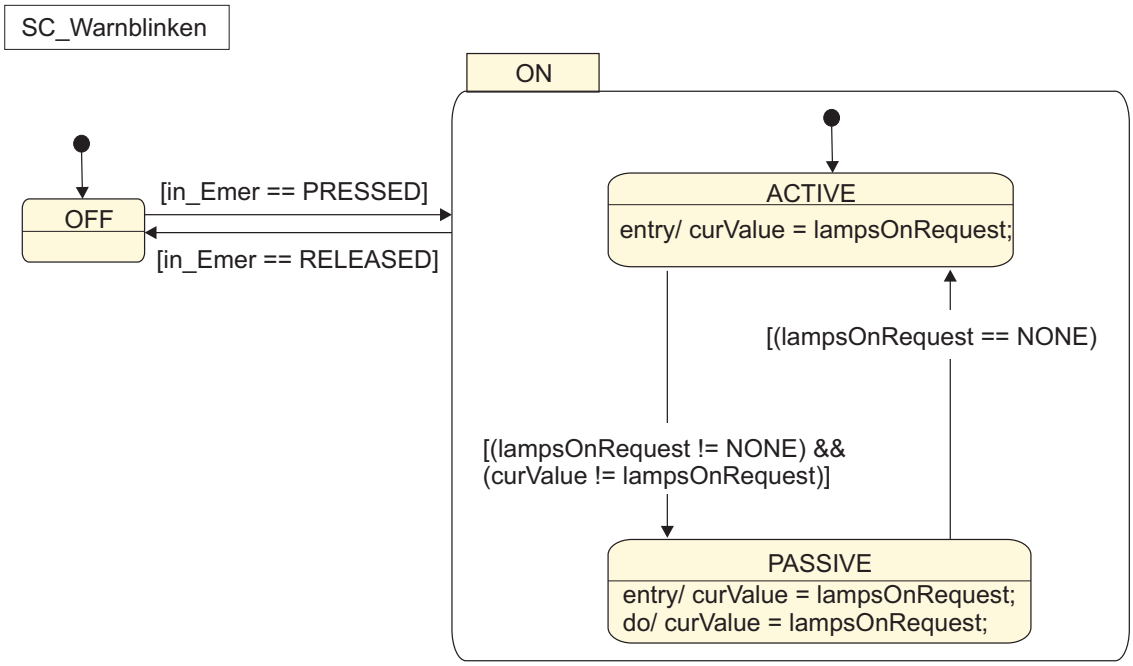


Abbildung 2.6: *Zustandsautomat Warnblinken*

Kapitel 3

Stand der Wissenschaft

Für die vorliegende Arbeit ist speziell der modellbasierte Test von Systemen mit zeitlichen Restriktionen, im Folgenden Zeitsysteme genannt, von Bedeutung (siehe dazu auch Abschnitt 5.5). Das Kapitel „Stand der Wissenschaft“ stellt die wichtigsten Forschungsergebnisse aus den Bereichen modellbasiertes Testen von Zeitsystemen (Abschnitt 3.1), Testfallgenerierung aus Modellen von Zeitsystemen (Abschnitt 3.2) und Requirements Tracing (Abschnitt 3.3) zusammen. Das Kapitel dient der späteren Abgrenzung des Dissertationsthemas von bestehenden Ansätzen (Abschnitt 4.6) und bietet einen Überblick der relevanten Aspekte in den einzelnen Bereichen.

Die wichtigsten Merkmale verteilter Zeitsysteme sind ihr reaktives, nebenläufiges und asynchrones Verhalten. Systeme, die keine globale Uhr zur Synchronisation besitzen und deren Zeitverständnis nicht diskret, sondern kontinuierlich am besten beschrieben werden kann, führt meist zum Einsatz der so genannten *Timed Automata* oder *Timed Statecharts* (Abschnitt 3.1.2). Ein Timed Automaton sowohl als auch ein Timed Statechart ist eine Struktur zur Beschreibung und Modellierung von Zustands- Transitionssystemen. Abschnitt 3.1.1 fasst die Grundlagen dieser Strukturen zusammen. Mit dem Verständnis der zugrundegelegten Struktur kann dann ein Blick auf den allgemeinen Begriff der Systemspezifikation anhand von Logiken (Abschnitt 3.1.3) geworfen werden. Die hier vorgestellte Definition der *Spezifikationssprache TCTL* ist ein grundlegendes Werkzeug für die Lösung zur Anforderungsbeschreibung als Thema vorliegender Arbeit.

Abschnitt 3.2 gibt einen Überblick der wichtigsten Testfallgenerierungsmethoden im Rahmen des modellbasierten Tests von Zeitsystemen. Die CTL-Testfallgenerierung auf Kripke Strukturen und die TCTL-Testfallgenerierung auf Timed Automata ist Inhalt von Abschnitt 3.2.2. Eine weitere Methode, das Model-Checking auf binären Entscheidungsdiagramme, ist in Abschnitt 3.2.3 enthalten. Für große Systeme wird als eine Möglichkeit das Bounded Model-Checking in Abschnitt 3.2.4 vorgestellt.

Requirements Tracing (Abschnitt 3.3) ist Teil des Requirements Engineering und soll eine lückenlose Verknüpfung aller Artefakte im Entwicklungsprozess sicherstellen. In der Literatur finden sich zahlreiche Publikationen, die sich mit der Frage beschäftigen, wie Informationen, die während des Anforderungslebenszyklus entstehen und alle Artefakte, die aufgrund einer Anforderung erzeugt wurden, sinnvoll, schnell und leicht wartbar verwaltet werden können [SAM00, Wie95, GF94, RJ01]. Für den modellbasierten Test sind spezielle visuelle Notationen (Abschnitt 3.3.2), Pattern (Abschnitt 3.3.3) und Observer (Abschnitt 3.3.4) von Interesse.

3.1 Modellierung von Zeitsystemen

Ein System ist ein dynamisches Ganzes, das als solches bestimmte Eigenschaften und Verhaltensweisen besitzt (siehe Abschnitt 2.1). Es besteht aus Teilen, die so miteinander verknüpft sind, dass kein Teil unabhängig ist von andern Teilen und das Verhalten des Ganzen beeinflusst wird vom Zusammenwirken aller Teile [UP91]. Ein System kann entweder operational oder deklarativ spezifiziert werden. Dabei wird zwischen den Begriffen Systemeigenschaften (Aussagen über die

Gesamtheit aller Eigenschaften seiner Teile; meist *deklarativ* spezifiziert) und Systemverhalten (Aussagen über die Veränderung der Eigenschaften seiner Teile; meist *operational* spezifiziert) unterschieden.

Eine Systemeigenschaft beschreibt, was ein System tun soll und was nicht. Ein Beispiel wäre die Anforderung an das System, immer, nach spätestens 5 Sekunden, in einen stabilen Notlauf wechseln zu können. Eine solche Eigenschaft wird im Lastenheft als Anforderung meist nicht-formal formuliert. Um sie automatisiert implementieren oder testen zu können, muss sie formalisiert werden. Im Bereich des Testens setzt der Testingenieur die Systemeigenschaft in eine deklarative Spezifikation um. Einige Systemeigenschaften, die getestet werden müssen, stehen nicht im Lastenheft. Es gibt allerdings Erfahrungen und bekannte Empfehlungen, welche Systemeigenschaften beachtet werden müssen. Wichtige Begriffe sind in diesem Bereich *safety properties* oder *lifeness properties* [ADX01]. Das deklarative Spezifizieren eines Systems ist eine mächtige aber gleichzeitig auch fehleranfällige Vorgehensweise. Sowohl alle Systemeigenschaften, als auch das vollständige Systemverhalten deklarativ zu spezifizieren, ist im Hinblick auf die Einsetzbarkeit in der Industrie nicht umsetzbar. Deshalb wird meist eine Kombination aus deklarativer und operationaler Spezifikation vorgenommen.

Das operational erstellte Systemmodell beschreibt, wie sich das System verhält. Es modelliert oder programmiert das Verhalten in der zum System passenden Notation. Die operationale Spezifikation ist weitaus eingängiger und leichter erlernbar, allerdings nicht so ausdrucksstark wie die deklarative Spezifikation. Hierzu zählen Programmcode wie Embedded C, Petri-Netze oder Statecharts. Systemeigenschaften können häufig aufgrund von Aussagen wie „es muss immer gelten, dass...“ oder „es darf nie passieren, dass...“ nicht operational spezifiziert werden. Die Begriffe *immer*, *nie*, *müssen* und *können* sind anhand von Zustandswechseln nicht beschreibbar. Sie müssen über eine deklarative Spezifikation vorgegeben werden.

3.1.1 Grundlagen

Um die Eigenschaften eines realen Systems über ein Modell dieses Systems verifizieren zu können, müssen sowohl das System als auch die Eigenschaften formal spezifiziert werden. Welche Abstraktion des realen Systems gewählt werden sollte, hängt von den zu verifizierenden Eigenschaften und dem System ab. Für folgende Erläuterungen sei beispielhaft das System *Licht* herangezogen, dass sich aus den Teilsystemen *Lichtschalter* und *Lampe* zusammensetzt, die nicht weiter zerlegt werden können. Das Verhalten der Teilsysteme soll über die Systemvariablen *switch* und *light* modelliert werden.

Sei $V = \{v_1, \ldots, v_n\}$ die endliche Menge an Systemvariablen. Sei für das Beispiel $V = \{switch, light\}$. Die Werte der Variablen seien in der Menge D zusammengefasst. D sei endlich und im Falle des Beispiels gelte $D = \{ON, OFF\}$. Ein *Zustand* s eines Systems ist eine Auswertung der Systemvariablen, also $s : V \rightarrow D$. Dadurch ergibt sich als Menge aller Zustände $S = \{s_1, s_2, s_3, s_4\}$ im System Lichtschalter-Lampe mit $s_1(switch, light) = (OFF, OFF)$, $s_2(switch, light) = (OFF, ON)$, $s_3(switch, light) = (ON, OFF)$ und $s_4(switch, light) = (ON, ON)$. Ist die Zustandsmenge S eines Systems endlich, so heißt das Modell des Systems *endlich*, ansonsten *unendlich*.

Unter der Annahme, dass das System Lichtschalter-Lampe die Systemeigenschaft „die Lampe darf nur an sein, wenn der Lichtschalter auf *ON* steht und nur aus, wenn der Schalter auf *OFF* steht“ erfüllt, sind die Zustände s_2 und s_3 fehlerhafte Systemzustände. Erlaubt ist somit nur ein Wechsel von s_1 nach s_4 und von s_4 nach s_1. Der Wechsel von Zuständen wird auch als *Transition* bezeichnet. Welche Transitionen ein System enthält, wird über die *Transitionsrelation* T festgelegt. Sie kann als Menge geordneter Zustandspaare $T \subseteq S \times S$ oder als Funktion $T : S \rightarrow S$ definiert werden. Die Transitionsrelation als Teilmenge von $S \times S$ enthält nur die Zustandspaare, die im System erlaubt sein sollen, somit gilt für das Beispiel $T = \{(s_1, s_4), (s_4, s_1)\}$. Denkbar ist auch eine Modellierung der Fehlerzustände und somit die Aufnahme der Fehlerzustände in die Transitionsrelation. Die Testfallgenerierung kann dann speziell Testfälle für ein fehlerhaftes Systemverhalten generieren. Meist wird jedoch die sogenannte positive Spezifikation gewählt, die

fehlerhaftes Systemverhalten nicht enthält.

Eine Transitionsrelation heißt *vollständig*, wenn gilt $\forall s \in S \; \exists \; s' \in S$ so dass gilt $(s, s') \in T$. In einem Zustandspaar (s, s') wird s als Vorzustand und s' als Folgezustand bezeichnet. Endet eine System in einem Zustand, das heißt, es verlässt diesen Zustand nicht mehr, so heißt das System *terminierend*, ansonsten *nicht-terminierend*.

In einem Automobil ist das Blinksystem *zustandsbasiert*, das System Motor hingegen *regelungsbasiert*. Die kontinuierlichen Abläufe in einem Motor (Drehzahlberechnungen, Drehmomentberechnungen,...) lassen sich nicht anhand einer endlichen Menge von Zuständen abbilden. Ist das Verhalten eines Programms, Systems oder einer Maschine zustandsbasiert, kann es mithilfe von Zustands-Transitions-Graphen beschrieben werden.

Ein Zustands-Transitions-Graph, der die Basis für weiterführende Varianten darstellt, ist die *Kripke Struktur* [CGP00]. Kripke Strukturen sind eine einfache Art, reaktives Verhalten zu beschreiben. Sie sind die Grundlage temporallogischer Ausdrücke und bieten eine guten Einstieg in die Definition der Timed Automata und Timed Statecharts, die in vorliegendem Kapitel noch genauer erläutert werden. Eine Kripke Struktur ist ein nichtdeterministischer, endlicher Automat. Er besteht aus Zuständen und Transitionen. In einer Kripke Struktur wird der Zustand durch eine Funktion, die ihm eine Menge von wahren *Aussagen* (engl: propositions) zuordnet, charakterisiert. Der Übergang von einem in den nächsten Zustand hängt also vom Wahrheitsgehalt vorgegebener Aussagen ab. Sei im Folgenden AP eine abzählbare Menge atomarer, das heißt nicht weiter zerlegbarer Aussagen. Die Aussage dient der Beschreibung einer Systemeigenschaft. Genau genommen steht *Aussage* für *aussagenlogische Formel* und ist somit kein *Wert* wahr oder falsch, sondern eine Formel, die zu wahr oder falsch auswertet. Aussagenlogische Formeln setzen sich aus einfachen Aussagen, wie zum Beispiel „$x < 3$“ oder boolschen Variablen zusammen (Definition 3.1.2). Anhand von logischen Operatoren wie `AND`, `NOT` oder `OR` werden Aussagen zu einer aussagenlogischen Formel verknüpft, die zu wahr oder falsch auswertet [Sch00b].

Definition 3.1.1 (Kripke Struktur [CGP00])**.**
Eine Kripke Struktur ist ein Tupel $\mathcal{K} = (S, S_0, T, L)$, mit

- S *ist eine endliche Menge von* Zuständen
- $S_0 \subseteq S$ *ist die Menge der* Initialzustände
- $T \subseteq S \times S$ *ist die vollständige* Transitionsrelationen
- $L : S \to 2^{AP}$ *ist die* Zuordnungsfunktion*, die jedem Zustand eine Menge von Aussagen zuordnet, die in diesem Zustand wahr sind. 2^{AP} ist die Potenzmenge von AP.*

□

Beispiel 4. *Der Graph einer Kripke Struktur ist in Abbildung 3.1 zu sehen.*



Abbildung 3.1: *Kripke Graph des Systems Licht*

Die Menge der Zustände sowohl als auch der Initialzustände der Kripke Struktur lautet $S = S_0 = \{ON, OFF\}$. Um die Zustände leichter unterscheiden zu können, wurde ihnen ein Name gegeben. Die Menge der atomaren Propositionen lautet $AP = \{light = true, \neg light = true, power = true, \neg power = true\}$. Die Zuordnungsfunktion gibt $L(OFF) = \{\neg light = true, \neg power = true\}$ und $L(ON) = \{light = true, power = true\}$ zurück.

Die Transitionsrelation $T = \{(ON, OFF), (OFF, ON)\}$ ist vollständig, da $\forall s \in \{OFF, ON\}$ $\exists s' \in \{OFF, ON\}$, so dass gilt $(s, s') \in \{(ON, OFF), (OFF, ON)\}$.

□

Ein *Pfad* durch eine Kripke Struktur $\mathcal{K}$ ist eine unendliche Folge von Zuständen $\pi = s_0, s_1, ...$ für die gilt $(s_i, s_{i+1}) \in T,\ i \geq 0$.

Der Begriff des *Berechnungslaufs* und *Berechnungsbaums* (engl. computation tree) wird im Folgenden anhand der Kripke Struktur definiert. Berechnungsbäume sind aufgefaltete Kripke Strukturen. Abbildung 3.2 zeigt den Zusammenhang von Zustands-Transitions-Graphen und Berechnungsbäumen.

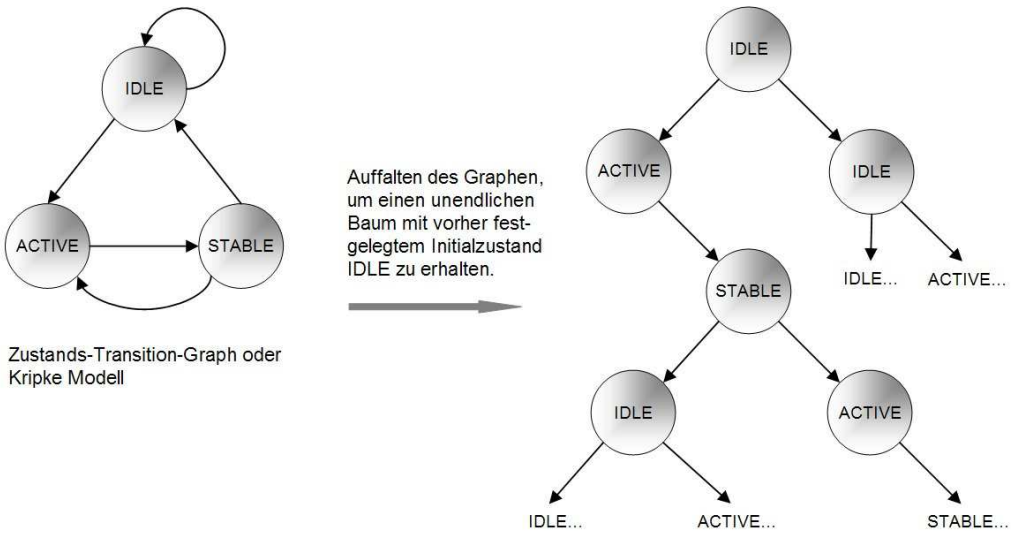


Abbildung 3.2: *Berechnungsbaum ([CGP00], S. 28)*

Angefangen bei einem Initialzustand wird mithilfe der Transitionsrelation die Menge aller Zustandswechsel abgerollt. Der Berechnungsbaum läuft bei nicht-terminierenden Strukturen wie der Kripke Struktur unendlich weiter.

Zur deklarativen Spezifikation der Systemeigenschaften bieten sich Logiken an. Im Folgenden werden die beiden klassischen Logiken Aussagenlogik und Prädikatenlogik vorgestellt.

Aussagenlogik

Die Aussagenlogik (engl. *propositional logic*), auch Boolsche Logik genannt, eignet sich gut, um Systeme zu spezifizieren, deren Systemvariablen und ihre Verknüpfung allein zu den Wahrheitswerten `true` und `false` auswerten [DO03].

Die kleinste Einheit, die noch zu wahr oder falsch auswertet, nennt man eine *atomare Aussage*. Atomare Aussagen sind zum Beispiel

A = Der Zustandsautomat befindet sich im Zustand „IDLE“.

B = Die Variable „sigTurnIndLever“ hat den Wert „LEFT“.

Die Verknüpfung atomarer Aussagen anhand der drei Möglichkeiten `UND`, `ODER` und `NICHT` führt zu folgenden Gebilden [Sch00b].

A *und* B $\Leftrightarrow$ Der Automat befindet sich in „IDLE“ *und* (sigTurnIndLever = LEFT).

A *oder* B $\Leftrightarrow$ (Automat.Zustand = IDLE) *oder* (sigTurnIndLever = LEFT) *oder* beides.

nicht A $\Leftrightarrow$ Der Automat befindet sich *nicht* in „IDLE“.

Symbolisch schreibt man für *und* ’$\wedge$’, für *oder* ’$\vee$’ und für *nicht* ’$\neg$’.
Atomare Aussagen lassen sich zu atomaren Formeln anhand der Syntax und Semantik der Aussagenlogik verknüpfen.

Definition 3.1.2 (Syntax der Aussagenlogik)**.**
Eine atomare Formel φ_i *mit* $(i = 1, 2, 3, \ldots)$ *wird induktiv wie folgt definiert:*

1. *Alle atomaren Formeln sind Formeln*
2. *Für alle Formeln* F *und* G *sind* $(F \wedge G)$ *(*Konjunktion*) und* $(F \vee G)$ *(*Disjunktion*) Formeln*
3. *Für jede Formel* F *ist* $\neg F$ *(*Negation*) eine Formel.*

□

Eine Formel F, die ein Teil einer Formel G darstellt, heißt *Teilformel* von G.
Weitere abkürzende Schreibweisen sind:

$F \rightarrow G$ statt $(\neg F \vee G)$

$F \leftrightarrow G$ statt $((F \wedge G) \vee (\neg F \wedge \neg G))$

$(\bigvee_{i=1}^{n} F_i)$ statt $(\ldots((F_1 \vee F_2) \vee F_3) \vee \ldots \vee F_n)$

$(\bigwedge_{i=1}^{n} F_i)$ statt $(\ldots((F_1 \wedge F_2) \wedge F_3) \wedge \ldots \wedge F_n)$

Die Bedeutung der Verknüpfung atomarer Aussagen zu Formeln wird in der Semantik der Aussagenlogik [Sch00a] festgelegt.

Definition 3.1.3 (Semantik der Aussagenlogik)**.**
Sei ***D*** *eine Teilmenge atomarer Formeln.* ***E*** $\supseteq$ ***D*** *ist die Menge aller atomarer Formeln, die sich aus den Elementen von* ***D*** *aufbauen.* $\mathcal{A} : \mathbf{D} \rightarrow \{0, 1\}$ *bzw.* $\hat{\mathcal{A}} : \mathbf{E} \rightarrow \{0, 1\}$ *bilden die Mengen aus* ***D*** *und* ***E*** *auf die Wahrheitswerte* 0 *(„falsch“) und* 1 *(„wahr“) ab.* $F, G \in \hat{\mathcal{A}}$*.*

1. *Für jede atomare Formel* $A \in \mathbf{D}$ *ist* $\hat{\mathcal{A}}(A) = \mathcal{A}(A)$
2. $\hat{\mathcal{A}}((F \wedge G)) = \begin{cases} 1, & \text{falls } \hat{\mathcal{A}}(F) = 1 \text{ und } \hat{\mathcal{A}}(G) = 1 \\ 0, & \text{sonst} \end{cases}$
3. $\hat{\mathcal{A}}((F \vee G)) = \begin{cases} 1, & \text{falls } \hat{\mathcal{A}}(F) = 1 \text{ oder } \hat{\mathcal{A}}(G) = 1 \\ 0, & \text{sonst} \end{cases}$
4. $\hat{\mathcal{A}}(\neg F) = \begin{cases} 1, & \text{falls } \hat{\mathcal{A}}(F) = 0 \\ 0, & \text{sonst} \end{cases}$

□

Für eine formal korrekte Definition der Semantik musste zwischen $\mathcal{A}$ und $\hat{\mathcal{A}}$ unterschieden werden. Da $\mathcal{A}$ und $\hat{\mathcal{A}}$ auf **D** übereinstimmen, wird im folgenden auf die Unterscheidung verzichtet.

Definition 3.1.4 (Modell, gültig, erfüllbar)**.**
Sei φ *eine Formel und* $\mathcal{A}$ *eine Belegung.*

(i) Falls $\mathcal{A}$ *für alle in* φ *vorkommenden Formeln* φ_i*,* $i = 1 \ldots n,\ n \in \mathbb{N}$ *definiert ist, so heißt* $\mathcal{A}$ *zu* φ passend*.*

(ii) Falls $\mathcal{A}$ *zu* φ *passend ist und* $\mathcal{A}(\varphi) = 1$ *gilt, so schreibt man* $\mathcal{A} \models \varphi$ *(gesprochen:* $\mathcal{A}$ ist ein Modell *für* φ *oder* φ gilt unter der Belegung $\mathcal{A}$*). Falls* $\mathcal{A}(\varphi) = 0$*, so schreibt man* $\mathcal{A} \not\models \varphi$ *(gesprochen:* $\mathcal{A}$ *ist kein Modell für* φ *oder unter* $\mathcal{A}$ *gilt* φ *nicht).*

(iii) Eine Formel φ heißt erfüllbar*, falls φ mindestens ein Modell besitzt, andernfalls heißt φ* unerfüllbar*.*

(iv) Eine Formel φ heißt gültig *(oder* Tautologie*), falls jede zu φ passende Belegung ein Modell für φ ist. Man schreibt $\models \varphi$, falls φ eine Tautologie ist und $\not\models \varphi$ sonst.*

□

Prädikatenlogik

Die Prädikatenlogik (engl. *first order logic*) ist eine Erweiterung der Aussagenlogik. In der Prädikatenlogik ist es möglich, den Bezug von Objekten zueinander über die Quantoren Allquantifizierung $(\forall)$ und Existenzquantifizierung $(\exists)$ herzustellen. Somit kann ausgedrückt werden, ob sich eine Eigenschaft auf alle Objekte bezieht, oder ob mindestens ein Objekt, dass die Eigenschaft erfüllt, existiert. Neben den Quantoren können auch Funktionen und Relationen (d.h. Prädikate) verwendet werden. Zusammen bilden Variablen, Funktionen und Relationen die syntaktischen Bestandteile der Prädikatenlogik [Sch00a].

Definition 3.1.5 (Syntax der Prädikatenlogik: Term)**.**
Die Menge der prädikatenlogischen Terme *wird induktiv wie folgt definiert:*

1. *Jede Variable ist ein Term,*
2. *Falls F ein k-stelliges Funktionssymbol ist und falls $t_1, \ldots, t_k$ Terme sind, so ist auch $f(t_1, \ldots, t_k)$ ein Term.*

□

Eine *Funktionen* f ist ein 3-stelliges Funktionssymbol, *Relationen* wie $<, \geq, >$ sind 2-stellige Funktionssymbole und 0-stellige Funktionssymbole werden auch *Konstanten(symbole)* genannt, meist $a, b, c, \ldots$. Diese Angaben nennt man auch *Signatur*.

Definition 3.1.6 (Syntax der Prädikatenlogik: Formel)**.**
Die Menge der prädikatenlogischen Formeln *wird wie folgt definiert:*

1. *Falls P ein k-stelliges Prädikatensymbol ist, und falls $t_1, \ldots, t_k$ Terme sind, dann ist $P(t_1, \ldots, t_k)$ eine Formel (genannt* atomare Formel*),*
2. *Für jede Formel F ist auch $\neg F$ eine Formel,*
3. *Für alle Formeln F und G sind auch $(F \wedge G)$ und $(F \vee G)$ Formeln,*
4. *Falls x eine Variable ist und F eine Formel, so sind auch $\exists x F$ und $\forall x F$ Formeln.*

□

Sei $\mathcal{A} = (M, f_1, \ldots, f_k, P_1, \ldots, P_m)$ eine Struktur zu einer Signatur. Sie enthält eine nicht-leere Grundmenge M sowie eine Angabe zu Funktionen und Relationen auf dieser Grundmenge.

Definition 3.1.7 (Semantik der Prädikatenlogik)**.**
Sei F eine Formel und $\mathcal{A}$ eine Struktur, die zur selben Signatur gehören. Dann heißt $\mathcal{A}$ zu F passend *und die Relation $\models$ wird wie folgt festgelegt:*

1. *Falls F die Form $F = \neg G$ hat, so ist $\mathcal{A} \models F \Leftrightarrow \mathcal{A} \not\models G$,*
2. *Falls F die Form $F = (G \wedge H)$ hat, so ist $\mathcal{A} \models F \Leftrightarrow \mathcal{A} \models G$ und $\mathcal{A} \models H$,*
3. *Falls F die Form $F = (G \vee H)$ hat, so ist $\mathcal{A} \models F \Leftrightarrow \mathcal{A} \models G$ oder $\mathcal{A} \models H$,*

4. *Falls F die Form $F = \forall x$ hat, so ist $\mathcal{A} \models F \Leftrightarrow \forall m \in M : \mathcal{A}[x/m] \models G$. $\mathcal{A}[x/m]$ bedeutet, dass für die Variable x der Wert m des Grundbereichs einzusetzen ist, sobald die Auswertung der Formel G erfolgt.*

5. *Falls F die Form $F = \exists x$ hat, so ist $\mathcal{A} \models F \Leftrightarrow \exists m \in M : \mathcal{A}[x/m] \models G$.*

6. *Falls F die Form einer atomaren Formel $F = P(t_1, \ldots, t_k)$ hat, so ist $\mathcal{A}[\bar{x}/\bar{m}] \models F \Leftrightarrow P^{\mathcal{A}}(t_1^{\mathcal{A}[\bar{x}/\bar{m}]}, \ldots, t_k^{\mathcal{A}[\bar{x}/\bar{m}]})$. $\mathcal{A}[\bar{x}/\bar{m}]$ beschreibt die Variablenbindung der in allen Termen vorkommenden Variablen $\bar{x}$ an Werte des Grundbereichs.*

□

3.1.2 Timed Automata und Timed Statecharts

Zustandsautomaten zählen zu den operationalen Systemspezifikationen. Ein Zustandsautomat setzt sich aus Zuständen S, Transitionen T und Aktionen a zusammen. Eine Aktion ist eine Veränderung der Wertebelegung von Systemvariablen. Sie wird in Aussagenlogik beschrieben. Ob eine Transition in einem Zustandsautomat schalten darf, hängt von der Bedingung ab, die an die Transition gestellt wird, der so genannten *Transitionsbedingung* (engl: guard condition). Die Transitionsbedingung kann aus mehreren Aktionen zusammengesetzt sein.

Die Zustandsautomaten setzen zur Modellierung von zeitlichen Verläufen sogenannte *Uhren* (engl: clocks, timer) als Variablen $t \in \mathbb{R}^+$ auf den positiven, reellen Zahlen ein.

Sei V eine Menge aus Symbolen, genannt *Variablen* und T eine Menge aus Zeitsymbolen, genannt *Uhren*. Die Funktion

$$\text{elapsed} : T \times \mathbb{N} \to \{true, false\}$$

gibt an, ob eine Uhr $t \in T$ eine natürliche Schranke c überschritten hat oder nicht. Sei $\nu : T \to \mathbb{R}^+$ eine Abbildung der Uhren auf eine nicht-negative reelle Zahl. Dann gilt [PZck]

$$\text{elapsed}(t, c) = \begin{cases} true, & \text{wenn } \nu(t) \geq c \\ false, & \text{wenn } \nu(t) < c \end{cases}$$

Timed Automata ermöglichen es, nebenläufige, nicht-terminierende, reaktive, asynchrone und verteilte Systeme mit Zeitbedingungen abzubilden. Ein Timed Automaton ist ein endlicher Automat, der eine endliche Menge reellwertiger *Uhren* erlaubt. Mithilfe der Uhren kann zeitliches Verhalten abgebildet werden. Folgende Definitionen wurden größtenteils Clarke [CGP00] und Alur [Alu99] entnommen.

Stellt man sich den Timed Automaton als Graph vor, so werden die Knoten des Graphen als *Positionen* (engl: locations) und die Kanten des Graphen als *Schalter* (engl: switches) bezeichnet. Während in einem Knoten die Zeit fortschreiten darf, muss eine Kante immer sofort, das heißt zeitlos, ausgeführt werden. Mit jeder Kante können Uhren wieder auf Null zurückgesetzt werden. Mit jeder Position verbindet man eine Bedingung an Uhren, eine sogenannte *Invariante*. Der Automat darf nur so lange in einer Position bleiben, bis die Invariante falsch wird. An Transitionen, die in der graphischen Repräsentation den Schaltern entsprechen, können ebenfalls Bedingungen an den Wert von Uhren gestellt werden. Andere erlaubte *Transitionsbedingungen* entsprechen den üblichen aussagenlogischen Formeln in Automaten.

Definition 3.1.8 (Timed Automata)**.**
Ein Timed Automaton *ist ein 6-Tupel $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$, mit*

- Σ *ist das endliche Menge der* Aktionen
- L *ist die endliche Menge der* Positionen
- $L_0 \subseteq L$ *ist die Menge der* Startpositionen

- X *ist die Menge der* Uhren
- $I : L \to C(X)$ *ist eine Abbildung von Positionen auf Bedingungen an Uhren, die* Positionsinvariante*.* $C(X)$ *gibt an, wann eine Transition genommen werden kann.*
- $T \subseteq L \times \Sigma \times C(X) \times 2^X \times L$ *ist eine Menge von Transitionen. Das 5-Tupel* $\langle l, a, \varphi, \lambda, l' \rangle$ *entspricht einer Transition von der Position* l *zur Position* l' *markiert mit der Aktion* a*, einer zeitlichen Bedingung* φ*, die angibt, wann die Transition genommen werden kann und einer Menge an Uhren* $\lambda \subseteq X$*, die auf Null gesetzt werden, sobald die Transition schaltet.*

□

Die Zustandsmenge und Transitionsrelation eines Timed Automaton bilden einen endlichen Graph $\mathcal{T}(\mathcal{A}) = (\Sigma, \mathcal{Q}, \mathcal{Q}_0, R)$. Jeder Zustand in $\mathcal{Q}$ ist ein Paar $\langle l, \nu \rangle$, wobei $l \in L$ eine Position und $\nu : X \to \mathbb{R}^+$ eine Abbildung der Uhren auf eine nicht-negative reelle Zahl ist.
Es gibt in einem Timed Automaton zwei Arten von Transitionen.

- *Verzögerungstransitionen* (engl: delay transitions) entsprechen dem Fortschreiten der Zeit während des Aufenthalts in einer Position. Wir schreiben $\langle l, \nu \rangle \xrightarrow{d} \langle l, \nu + d \rangle$ mit $d \in \mathbb{R}^+$ und unter der Voraussetzung, dass für jedes $0 \leq e \leq d$ die Invariante $I(l)$ für $\nu + e$ wahr bleibt.
- *Aktionstransitionen* entsprechen der Ausführung einer Transition aus T. Wir schreiben $\langle l, \nu \rangle \xrightarrow{a} \langle l', \nu' \rangle$ wobei $a \in \Sigma$, unter der Voraussetzung, dass eine Transition $\langle l, a, \varphi, \lambda, l' \rangle$ existiert, so dass gilt: (1) ν erfüllt φ (2) $\nu' = \nu$ für alle Uhren aus $X \backslash \{\lambda\}$ und alle restlichen Uhren aus λ werden auf Null zurückgesetzt.

Die *Transitionsrelation* R erhält man, indem Verzögerungs- und Aktionstransitionen verknüpft werden. Wir schreiben $\langle l, \nu \rangle \stackrel{a}{\Rightarrow} \langle l', \nu' \rangle$ oder kurz $\langle l, \nu \rangle \, R \, \langle l', \nu' \rangle$, falls s'' und ν'' existieren, so dass $\langle l, \nu \rangle \xrightarrow{d} \langle l'', \nu'' \rangle \xrightarrow{a} \langle l', \nu' \rangle$ für $d \in \mathbb{R}^+$.

Beispiel 5. *Der Graph eines Timed Automaton* $\mathcal{A}$ *ist in Abbildung 3.3 zu sehen.*
s *und* t *sind Uhren,* $X = \{t, s\}$*. Die Aktionen lauten* $\Sigma = \{$*switch_on*, *switch_off*$\}$*.* $L = \{OFF, ON\}$*.*

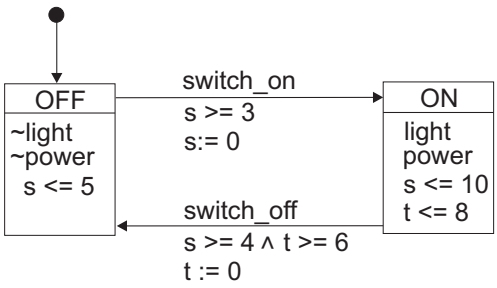


Abbildung 3.3: *Graph des Timed Automaten eines zeitlich eingeschränkten Lichtschalters*

$I(OFF) = \{s \leq 5\}$*,* $I(ON) = \{s \leq 10, t \leq 8, (s \leq 10) \wedge (t \leq 8)\}$*.*
$T = \{\langle OFF,$ *switch_on*$, s \geq 3, s, ON\rangle, \langle ON,$ *switch_off*$, (s \geq 4) \wedge (t \geq 6), t, OFF\rangle\}$*.*
Der Automat darf solange in OFF verweilen, bis die Uhr s *5 Zeiteinheiten überschritten hat. Er darf die Transition mit „switch_on = true“ aber frühestens bei 3 Zeiteinheiten schalten. In diesem Fall wird die Uhr* s *auf 0 zurückgesetzt. In ON darf die Uhr* s *bis zu 10 Zeiteinheiten laufen, die Uhr* t *bis zu 8 Zeiteinheiten. Die Transition mit „switch_off = true“ kann frühestens genommen werden, wenn sowohl* s *4 Zeiteinheiten als auch* t *6 Zeiteinheiten erreicht hat. Dabei wird die Uhr* t *auf 0 gesetzt. Das Verweilen in Positionen als auch das Schalten von Transitionen ist somit an zeitliche Bedingungen geknüpft. Der Lichtschalter in diesem Beispiel kann frühestens nach 3 Zeiteinheiten ein und wiederum frühestens nach 3 Zeiteinheiten ausgeschaltet werden.*

Allgemein sei ein Zustand des Zustands-Transitions-Graphen in Abbildung 3.3 gegeben als $\langle l, \nu(s), \nu(t)\rangle$*. Eine Aktionstransition lautet zum Beispiel* $\langle OFF, 4, 4\rangle \overset{switch_on}{\longrightarrow} \langle ON, 0, 4\rangle$*. Eine Verzögerungstransition* $\langle ON, 0, 4\rangle \xrightarrow{3} \langle ON, 3, 7\rangle$*. Die passende Transition aus der Transitionsrelation* R *lautet* $\langle OFF, 4, 4\rangle \overset{switch_on}{\Longrightarrow} \langle ON, 3, 7\rangle$*.*

□

Ein *Testfall* ist eine (un)endliche Folge von Zuständen $\langle l_0, \nu_0\rangle, \langle l_1, \nu_1\rangle, \langle l_2, \nu_2\rangle \ldots [, \langle l_n, \nu_n\rangle]$, für die gilt:
$\langle l_i, \nu_i\rangle \, R \, \langle l_{i+1}, \nu_{i+1}\rangle$, $i \geq 0$.

Die Folge $\langle l, \nu\rangle \xrightarrow{d} \langle l'', \nu''\rangle \xrightarrow{a} \langle l', \nu'\rangle \ldots$ nennt man auch *(unendliches) Ausführungsfragment*. Beginnt ein unendliches Ausführungsfragment in einem Initialzustand, so nennt man es auch kurz *Ausführung* (engl. execution, run).

Parallele Komposition Laufen zwei Berechnungen eines Systems parallel, können sie einzeln in getrennten Timed Automata modelliert werden, um dann über die parallele Komposition zu einem Gesamtsystem verknüpft zu werden. Die parallele Komposition zweier Timed Automata $\mathcal{A}_1 = (\Sigma_1, L_1, L_0^1, X_1, I_1, T_1)$ und $\mathcal{A}_2 = (\Sigma_2, L_2, L_0^2, X_2, I_2, T_2)$ erzeugt den so genannten *Produktautomaten* $\mathcal{A}_1 \parallel \mathcal{A}_2$.

Eine Annahme, die dabei getroffen werden sollte, ist die Unabhängigkeit der in den Timed Automata verwendeten Uhren, also $X_1 \cap X_2 = \emptyset$. Sonstige Variablen können von beiden Automaten geteilt werden. Diese geteilten Variablen beschreiben die Kommunikation zwischen den Automaten. Es gilt

$$\mathcal{A}_1 \parallel \mathcal{A}_2 = \left\langle \Sigma_1 \cup \Sigma_2, L_1 \times L_2, L_0^1 \times L_0^2, X_1 \cup X_2, I, T \right\rangle$$

wobei $I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ ist. Die Transitionsrelation T erfüllt folgende Regeln:

- Sei a aus der Schnittmenge $\Sigma_1 \cap \Sigma_2$ und gelten die Transitionsrelationen $\langle l_1, a, \varphi_1, \lambda_1, l_1'\rangle \in T_1$ und $\langle l_2, a, \varphi_2, \lambda_2, l_2'\rangle \in T_2$, dann enthält T die Transition $\langle (l_1, l_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (l_1', l_2')\rangle$.
- Sei a aus der Differenz $\Sigma_1 - \Sigma_2$ und gelte $\langle l_1, a, \varphi, \lambda, l_1'\rangle \in T_1$ und $l_2 \in L_2$, dann enthält T die Transition $\langle (l_1, l_2), a, \varphi, \lambda, (l_1', l_2)\rangle$.
- Sei a aus der Differenz $\Sigma_2 - \Sigma_1$ und gelte $\langle l_2, a, \varphi, \lambda, l_2'\rangle \in T_2$ und $l_1 \in L_1$, dann enthält T die Transition $\langle (l_1, l_2), a, \varphi, \lambda, (l_1, l_2')\rangle$.

Für ausführliche Beschreibungen der parallelen Komposition mehrerer Timed Automata sei auf Clarke [CGP00], S. 268 ff verwiesen. Es gibt noch weitere Möglichkeiten, Parallelität in Transitionssystemen zu modellieren. Nähere Informationen hierzu finden sich bei Baier und Katoen [BK08], S. 36 ff. Weitergehende Definitionen und Anwendungen der Timed Automata finden sich ebenfalls bei Bengtsson [BY04] und Alur [ACD90].

Neben den Timed Automata bieten *Timed Statecharts* eine gute Alternative zur Modellierung reaktiver Systeme. Statecharts wurden erstmals 1987 von Harel vorgestellt [Har87]. Mittlerweile ist ihr Einsatz weit verbreitet und viele Varianten und Methoden haben sich auf ihrer Grundlage etabliert. Speziell die Frage, welche Semantik den Statecharts zugrunde gelegt werden soll, beschäftigt die Wissenschaft seit langem. Für vorliegende Arbeit wird die von Peleska und Zahlten [PZck] definierte Semantik betrachtet, die stark an die 1996 von Harel und Naamad [HN96] vorgestellte Semantik STATEMATE anlehnt.

Definition 3.1.9 (Timed Statechart)**.**
Ein Timed Statechart *ist ein 8-Tupel* $S = (V, T, L, L_0, P, G, A, \Lambda)$ *mit*

- V *ist eine* Variablenmenge*. Expr*(V) *ist die Menge der prädikatenlogischen Aussagen auf den Variablen aus* V *und Bexpr* (V) *die Menge Boolscher Aussagen auf* V*,*
- T *ist eine Menge von* Uhren*,*
- L *ist die endliche Menge der* Positionen
- $L_0 \subseteq L$ *ist die Menge der* Startpositionen
- P *ist die Menge der* Transitionsprioritäten
- G *ist die Menge der* Transitionsbedingungen*.*
- $A = \mathbb{P}(V \times$ *Expr*$(V))$ *ist die Menge der* Aktionen*.*
- $\Lambda \subseteq L \times P \times G \times A \times L$ *ist die Menge der* Transitionen*. Das 5-Tupel* $\langle l, p, g, \alpha, l' \rangle$ *entspricht einer Transition von der Position* l *zur Position* l' *markiert mit* α*, einer Bedingung* $g \in$ *Bexpr* (V)*, die angibt, wann die Transition genommen werden kann und einer Priorität* p*.*

□

Ein Statechart setzt sich somit zusammen aus *Positionen* und *Transitionen*. Die Menge der Positionen eines Statecharts s sei mit $L(s)$ bezeichnet. Eine Transition $\tau = \langle l, p, g, \alpha, l' \rangle$ verbindet Ursprungsposition l und Zielposition l'.

$p \in \mathbb{N}$ bezeichnet eine Priorität an der Transition, um nicht-deterministische Automaten zu vermeiden, falls das benötigt wird.

g enthält die Transitionsbedingung. G ist die Vereinigung der Uhr- Bedingungen und der Nicht-Uhr-Bedingungen. Eine Uhr $t \in T$ darf nur in einer Bedingung elapsed$(t, c) \in$ Bexpr (V) verwendet werden. Für eine Nicht-Uhr-Bedingung φ auf Variablen aus V gilt ebenfalls $\varphi \in$ Bexpr (V).

A enthält Aktionen, die ausgeführt werden, sobald die Transition genommen wird. Eine Aktion $\alpha \in A$ kann Variablenzuweisungen bezüglich der Variablen aus V und der erlaubten Aussagen aus Expr(V) beschreiben oder Uhren aus T auf Null zurücksetzen.

Ein Test-Modell ist eine Struktur, die auf Komponenten und parallelen Timed Statecharts aufbaut. Im Folgenden die Definition.

Definition 3.1.10 (Test-Modell aus Statecharts)**.**
Ein (Test-)Modell aus Statecharts *ist ein 4-Tupel* $\mathcal{M} = (C, SC, p_C, sc)$*, mit*

- C *ist die endliche Menge der hierarchisch geordneten* Komponenten
- SC *ist die endliche Menge der* Statecharts*,* $SC = s_1, \ldots, s_n$
- $p_C : C \to \mathbb{P}(C)$ *ist die* Ordnungsfunktion *der Hierarchie auf den Komponenten* C
- sc *bildet Komponentenblätter auf zugehörige Timed Statecharts ab.*

□

Die Domäne der Funktion p_C lautet dom $p_C = C \backslash \{c_r\},\ c_r$ ist die Wurzelkomponente.

Die Funktion $d_C : C \to \{c' \in C \mid p_C(c') = c\}$ bildet Komponenten auf ihre direkten Nachfolger ab.

Die Domäne der Funktion sc lautet dom $sc = \{c \in C \mid d_C(c) = \emptyset\}$. Eine Blattkomponente c erfüllt die Eigenschaft $d_C(c) = \emptyset$.

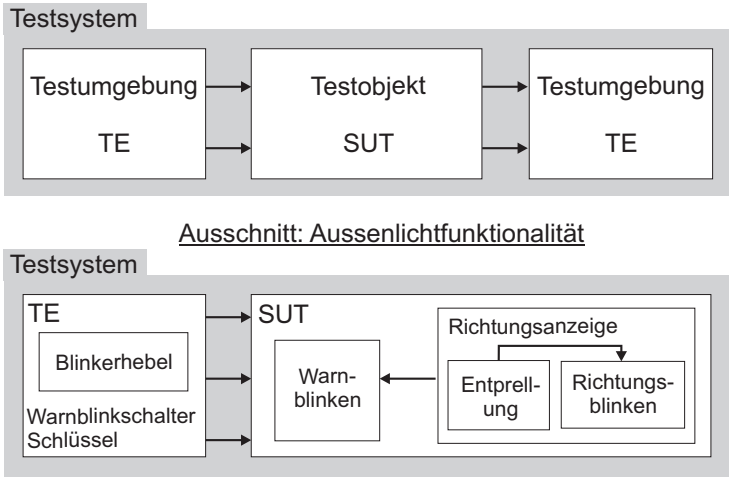


Abbildung 3.4: *Aufbau eines Testsystems im Rahmen der modellbasierten Testfallgenerierung*

Beispiel 6. *Abbildung 3.4 strukturiert den Aufbau eines Testmodells in Testumgebung und Testobjekt. Die untere Hälfte der Graphik gibt den Ausschnitt der Blinkfunktionalität wieder, dessen Automaten in Abschnitt 2.5 durch die Graphiken Abbildung 2.3, Abbildung 2.5 und Abbildung 2.6 beschrieben sind.*

Die Pfeile zwischen Komponenten im Testsystem weisen auf Datenflüsse hin. Zwischen TE und SUT fließt zum Beispiel Information über die Variablen `in_Turnindicator`, `in_Emer` *und* `in_Key`. *Zwischen den Komponenten* `Entprellung` *und* `Richtungsblinken` *wird Information anhand von* `debounced_TI` *weitergegeben und zwischen* `Richtungsanzeige` *und* `Warnblinken` *durch die Variable* `lampsOnRequest`.

Die Elemente des Testmodells $\mathcal{M}$ sehen wie folgt aus:

$$
\begin{aligned}
C &= \{\texttt{Testsystem, TE, SUT, Blinkerhebel, Richtungsanzeige,}\\
&\quad \texttt{Entprellung, Richtungsblinken, Warnblinken}\}\\
c_r &= \texttt{Testsystem}\\
p_c &= \{\texttt{TE} \mapsto \texttt{Testsystem, SUT} \mapsto \texttt{Testsystem,}\\
&\quad \texttt{Blinkerhebel} \mapsto \texttt{TE, Richtungsanzeige} \mapsto \texttt{SUT,}\\
&\quad \texttt{Entprellung} \mapsto \texttt{Richtungsanzeige,}\\
&\quad \texttt{Richtungsblinken} \mapsto \texttt{Richtungsanzeige,}\\
&\quad \texttt{Warnblinken} \mapsto \texttt{SUT}\}\\
d_c &= \{\texttt{Testsystem} \mapsto \texttt{\{TE,SUT\}, TE} \mapsto \texttt{Blinkerhebel},\\
&\quad \texttt{SUT} \mapsto \texttt{\{Richtungsanzeige, Warnblinken\}},\\
&\quad \texttt{Richtungsanzeige} \mapsto \texttt{\{Entprellung, Richtungsblinken\}},\\
&\quad \texttt{Entprellung} \mapsto \emptyset, \texttt{ Richtungsblinken} \mapsto \emptyset,\\
&\quad \texttt{Warnblinken} \mapsto \emptyset\}\\
SC &= \{\texttt{SC_Blinkerhebel, SC_Entprellung,}\\
&\quad \texttt{SC_Richtungsblinken, SC_Warnblinken}\}\\
sc &= \{\texttt{Richtungsblinken} \mapsto \texttt{\{SC_Richtungsblinken,}\\
&\quad \texttt{SC_Entprellung\}, Warnblinken} \mapsto \texttt{SC_Warnblinken}\}
\end{aligned}
$$

Beispielhaft lauten die Positionen für Statechart `SC_Warnblinken`*:*

$$
\begin{aligned}
L(\texttt{SC_Warnblinken}) &= \{\texttt{SC_Warnblinken, OFF, ON, ACTIVE,}\\
&\quad \texttt{PASSIVE, start(OFF), start(ON)}\}
\end{aligned}
$$

Eine Aktion im Statechart `SC_Warnblinken` *ist:*

$do(\texttt{PASSIVE}) = (\texttt{curValue, lampsOnRequest})$

Transitionen im Statechart `SC_Richtungsblinken` *sind zum Beispiel:*

$$\begin{aligned}\tau_1 &= (\texttt{DEACTIVATED, 0, in_Key} \geq 3, \emptyset, \texttt{ACTIVATED})\\ \tau_2 &= (\texttt{TIP_FLASHING, 0, elapsed(s,para_TipDuration),}\\ &\quad \texttt{(lampsOnRequest, NONE), IDLE})\end{aligned}$$

□

Die Transitionsrelation bildet das im Timed Statechart modellierte Systemverhalten ab. Ausgehend vom aktuellen Systemzustand bestimmt die Transitionsrelation über die Positionen, Variablen und den aktuellen Zeitpunkt den Folgezustand. Wie auch bei Timed Automata unterscheidet man bei Timed Statecharts zwischen Aktionstransitionen und Verzögerungstransitionen. Allgemein lautet die Form der Transitionsrelation

$$R \equiv_{def} \big((aktiv_A \wedge R_A) \vee (\neg aktiv_A \wedge R_T)\big) \wedge \mathsf{Inv}' \ ,$$

wobei die Bedeutung von $aktiv_A$, R_A, R_T und Inv′ wie folgt gegeben ist.

$aktiv_A$ ist `true`, wenn in mindestens einem Statechart eine Transition freigegeben ist oder eine do-Aktion ausgeführt werden muss, ansonsten `false`. Eine do-Aktion führt nur dann zu einer Aktionstransition, wenn die linke Seite einer Zuweisung ungleich der rechten ist. Würde diese Bedingung fehlen, könnte bei einem Statechart mit do-Aktion niemals eine Verzögerungstransition ausgeführt werden, weil immer ein diskreter Schritt mit do-Aktion möglich ist.

Eine Aktionstransition R_A kann nur ausgeführt werden, wenn $aktiv_A = \texttt{true}$ gilt. Eine Verzögerungstransition R_T darf nur ausgeführt werden, wenn $aktiv_A = \texttt{false}$ gilt.

Beispiel 7 (Fortsetzung Beispiel 6)**.** *Die Auslöser und die Auswirkung der Transitionen τ_1 und τ_2 aus Beispiel 6 sehen wie folgt aus.*

$$\begin{aligned}\textit{auslöser}(\tau_1) &\equiv \texttt{DEACTIVATED} \wedge (\texttt{in_Key} \geq 3)\\ \textit{auslöser}(\tau_2) &\equiv \texttt{TIP_FLASHING} \wedge\ (\hat{t} - t \geq 3) \wedge \neg(\texttt{in_Key} < 3)\\ \textit{auswirkung}(\tau_1) &\equiv (\hat{t}' = \hat{t}) \wedge \texttt{IDLE}'\\ \textit{auswirkung}(\tau_2) &\equiv (\hat{t}' = \hat{t}) \wedge (\texttt{lampsOnRequest}' = \texttt{NONE}) \wedge \texttt{IDLE}'\end{aligned}$$

Auslöser und Auswirkung bilden die Grundlage der Transitionsrelation R.

□

Zusammenhänge von Timed Automata und Timed Statecharts können in [GB03] nachgelesen werden.

3.1.3 Temporallogik

Die Aussagenlogik und die Prädikatenlogik gehören zu den so genannten *klassischen* Logiken, das heißt, ihre Aussagen werten entweder zu *wahr* oder *falsch* aus. Es gibt auch die *nichtklassischen* Logiken, bei denen eine Aussage „*möglicherweise wahr*“ (Modallogik) oder „*irgendwann mal wahr*“ (Temporallogik) sein kann [Sch00a]. Letztere eignet sich gut, um Eigenschaften zu spezifizieren, die das Verhalten eines Systems über die Zeit hinweg beschreiben. Dies ist bei Systemen mit reaktivem Verhalten der Fall. Eine reaktives System interagiert mit seiner Umwelt und diese Interaktion hängt von zeitlichen Zusammenhängen ab. Aufgrund dessen hat Amir Pnueli 1977 [Pnu77] die Eigenschaften von Computerprogrammen unter dem Blickwinkel zeitlicher Verläufe untersucht und *Temporallogiken* (engl. *temporal logic*) zur Spezifikation dieser Abläufe vorgestellt.

Temporallogiken beschreiben Systemeigenschaften, die Berechnungsläufe von Zustands-Transitionssystemen charakterisieren. Das Basis Zustands-Transitionssystem, auf das die Temporallogiken angewandt werden, ist die Kripke Struktur.

CTL* (computation tree logic) ist eine allgemeine temporale Logik, die zeitliche Abhängigkeiten abbilden kann, ohne einen expliziten Zeitbegriff zu verwenden. Damit kann zum Beispiel spezifiziert werden, dass eine Schranke *immer* geschlossen sein muss, *bevor* ein Zug die Schranke erreicht. Eine solche Eigenschaft wird als ein Pfad oder eine Menge von Pfaden durch Berechnungsbäume abgebildet.

Im Folgenden eine Übersicht der CTL* Operatoren, wobei p und q, mit $p, q \in AP$ Prädikate, d.h. logische Ausdrücke über Symbole, beschreiben.

- **X** p (sprich: next p): im nächsten Pfadzustand muss die Eigenschaft p wahr sein
- **F** p (sprich: eventually p, finally p): in irgendeinem Pfadzustand muss die Eigenschaft p wahr werden
- **G** p (sprich: always p, globally p): Die Eigenschaft p muss immer wahr sein, für alle Pfadzustände
- p **U** q (sprich: p until q): Die Eigenschaft p muss wahr bleiben, solange bis die Eigenschaft q wahr wird. Sobald q wahr wird, muss p nicht mehr wahr bleiben. Wird q nie wahr, muss p immer wahr bleiben.
- p **R** q (sprich: p release q): q muss solange wahr bleiben, bis p wahr wird. Sobald p wahr wird, muss q nicht mehr wahr bleiben. Wird p nie wahr, muss q immer wahr bleiben.

U und **R** verhalten sich dual. Es reicht also, sich auf einen Operator zu beschränken.
Für eine kürzere Schreibweise werden folgende Zeichen eingeführt.

$\bigcirc$ entspricht NEXT

$\mathcal{U}$ entspricht UNTIL

$\Box$ entspricht ALWAYS/GLOBALLY

$\Diamond$ entspricht FINALLY

In CTL* gibt es sowohl Eigenschaften, die für einen kompletten Pfad wahr sein müssen und als *Pfadformeln* bezeichnet werden, als auch *Zustandsformeln*, die nur in einem speziellen Zustand wahr sein müssen. Seien im Folgenden Pfadformeln mit griechischen Kleinbuchstaben gekennzeichnet (ϕ, φ) und Zustandsformeln mit griechischen Großbuchstaben (Φ, Ψ).

CTL* hat zwei Quantoren. **A** („for all computation paths“), ist die so genannte Allquantifizierung $(\forall)$. Eine Formel muss in diesem Fall für alle Pfade gelten. **E** („for some computation path“), wird auch Existenzquantor $(\exists)$ genannt. In diesem Fall muss eine Formel nur für mindestens einen Pfad gelten. Quantoren beziehen sich immer nur auf Pfade, *nicht* auf Zustände. Mithilfe eines Quantors wird eine Pfadformel in eine Zustandsformel gewandelt.

LTL LTL (linear-time logic) ist ebenso wie CTL (branching-time logic) eine Verfeinerung der CTL*. Die Charakteristik von LTL liegt darin, dass die Zeit als linearer Verlauf verstanden wird, in dem, im Gegensatz zur CTL, keine alternativen Entwicklungen über die Zeit hinweg erlaubt sind. Sie bezieht sich also immer nur auf einen Pfad durch den Berechnungsbaum einer Kripke Struktur oder, anders gesagt, betrachtet jeweils nur einen Systemlauf. Als Konsequenz hat LTL nur einen Quantor und zwar **A** („for all computation paths“).

CTL CTL-Formeln werden über Zuständen und Pfaden eines Berechnungsbaums interpretiert.

Die Semantik der CTL-Formeln wird für endliche Pfade einer Kripke Struktur $\mathcal{K} = (S, S_0, T, L)$ definiert. Sei s ein Zustand in der Kripke Struktur und π ein Pfad durch $\mathcal{K}$, dann besagt $\mathcal{K}, s \models \phi$ oder kurz $s \models \phi$, dass ϕ unter s gilt. $\mathcal{K}, \pi \models \phi$ oder kurz $\pi \models \phi$ bedeutet, dass ϕ für alle Zustände s entlang des Pfades π gilt. π_i ist der Startpunkt des Teilpfades von π beginnend mit dem i-ten Zustand. $\pi(i)$ ist der i-te Zustand von π mit $i \in \mathbb{N}_0$. Der Initialzustand ist $\pi(0)$.

Definition 3.1.11 (Syntax CTL)**.**
CTL-Zustandsformeln *lassen sich induktiv wie folgt definieren:*

$$\Phi ::= true \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\, \varphi \mid \forall\, \varphi$$

wobei $p \in AP$ und φ eine Pfadformel ist.
CTL-Pfadformeln *werden durch folgende Grammatik definiert:*

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathcal{U} \Phi_2$$

□

Die Semantik der CTL wird anhand von zwei Erfüllbarkeitsrelationen definiert, einer für die Zustandsformeln und einer für die Pfadformeln.

Definition 3.1.12 (Semantik CTL)**.**
Sei $\mathcal{K} = (S, S_0, T, L)$ eine Kripke Struktur, π ein Pfad durch $\mathcal{K}$ und s ein Zustand aus S. Φ, Φ_1 und Φ_2 seien Zustandsformeln. φ ist eine Pfadformel. Die Relation $\models$ wird für CTL-Zustandsformeln *induktiv wie folgt definiert:*

$$\begin{aligned}
s &\models true &&\Leftrightarrow \quad \textit{für alle } s \\
s &\not\models false &&\Leftrightarrow \quad \textit{für alle } s \\
s &\models p &&\Leftrightarrow \quad p \in L(s) \\
s &\models \neg\Phi &&\Leftrightarrow \quad s \not\models \Phi \\
s &\models \Phi_1 \wedge \Phi_2 &&\Leftrightarrow \quad s \models \Phi_1 \wedge s \models \Phi_2 \\
s &\models \exists\, \varphi &&\Leftrightarrow \quad \pi \models \varphi \textit{ für mindestens ein } \pi \in \textit{Pfade}(s) \\
s &\models \forall\, \varphi &&\Leftrightarrow \quad \pi \models \varphi \textit{ für alle } \pi \in \textit{Pfade}(s)
\end{aligned}$$

Die Relation $\models$ wird für CTL-Pfadformeln *induktiv wie folgt definiert:*

$$\begin{aligned}
\pi &\models \bigcirc\Phi &&\Leftrightarrow \quad \pi(1) \models \Phi \\
\pi &\models \Phi_1 \mathcal{U} \Phi_2 &&\Leftrightarrow \quad \exists i \in \mathbb{N}_0 . \pi(i) \models \Phi_2 \wedge \forall\, 0 \leq j < i . \pi(j) \models \Phi_1
\end{aligned}$$

□

Um weitere Analysen der CTL Formeln leichter durchführen zu können, werden sie, je nach Anwendung, in eine Normalform umgewandelt. An dieser Stelle soll die existentielle Normalform als ein Vertreter der Normalformen erwähnt werden.

Definition 3.1.13 (Existentielle Normalform der CTL)**.**
Für ein $a \in AP$ ist die Menge der CTL-Zustandsformeln in existentieller Normalform *(kurz:* ENF*) gegeben durch*

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists \bigcirc \Phi \mid \exists(\Phi_1 \mathcal{U} \Phi_2) \mid \exists\Box\Phi.$$

□

Satz 3.1.1 (Existenz der ENF für CTL Formeln)**.**
Für jede CTL Formel existiert eine äquivalente CTL Formel in ENF.

□

Zum Umformen in ENF helfen folgende Äquivalenzen

$$\begin{aligned}
\forall \bigcirc \Phi &\equiv \neg\exists \bigcirc \neg\Phi \\
\forall(\Phi \,\mathcal{U}\, \Psi) &\equiv \neg\exists(\neg\Psi \,\mathcal{U}\, (\neg\Phi \wedge \neg\Psi)) \wedge \neg\exists\Box\neg\Psi \\
\forall\Diamond\Phi &\equiv \neg\exists\Box\neg\Phi \\
\forall\Box\Phi &\equiv \neg\exists\Diamond\neg\Phi = \neg\exists(true \,\mathcal{U}\, \Phi).
\end{aligned}$$

Beispiel 8. *Die CTL-Formel* $\Phi = \forall\Box(\texttt{light} \rightarrow \forall\Diamond\neg\texttt{power})$ *lautet in existentieller Normalform*

$$\begin{aligned}
&\forall\Box(\texttt{light} \rightarrow \forall\Diamond\neg\texttt{power}) \\
&\Leftrightarrow \quad \neg\exists(true \,\mathcal{U}\, (\texttt{light} \rightarrow \forall\Diamond\neg\texttt{power})) \\
&\Leftrightarrow \quad \neg\exists(true \,\mathcal{U}\, (\neg\texttt{light} \vee \forall\Diamond\neg\texttt{power})) \\
&\Leftrightarrow \quad \neg\exists(true \,\mathcal{U}\, (\texttt{light} \wedge \neg\forall\Diamond\neg\texttt{power})) \\
&\Leftrightarrow \quad \neg\exists(true \,\mathcal{U}\, (\texttt{light} \wedge \neg(\neg\exists\Box\texttt{power}))) \\
&\Leftrightarrow \quad \neg\exists(true \,\mathcal{U}\, (\texttt{light} \wedge \exists\Box\texttt{power})).
\end{aligned}$$

□

Timed CTL TCTL ist eine 1990 von Alur et al. [ACD90] eingeführte Erweiterung der branching-time Logik CTL mit beschränkten Operatoren und einer punkt-basierten, streng monotonen Echtzeitsemantik [AH92], um feste zeitliche Vorgaben an Zustandswechsel beschreiben zu können. Eine feste zeitliche Vorgabe wäre zum Beispiel „die Bahnschranke muss immer bereits 10 Sekunden vor dem Eintreffen eines Zuges geschlossen sein“.

Vorliegend wird anstatt einer Diskretisierung der Zeit die positive reelle Zeitachse $\mathbb{R}^+$ gewählt. S sei eine Menge von Zuständen mit $s = \langle l, \nu\rangle \in S$. Die Semantik von TCTL wird auf unendlich langen Pfaden durch ein Transitionssystem mit überabzählbar vielen Zuständen und Transitionen definiert. Nicht alle dieser möglichen Pfade sind „realistisch“ [BK08]. Dieses „unrealistische“ Verhalten kann mithilfe von drei Aspekten begründet werden: *Zeitdivergenz*, *Timelock* und *Zenoness* [BK08].

Zeitdivergenz Theoretisch ist es möglich, dass eine Folge von zeitlichen Verzögerungen innerhalb eines Pfades gegen einen Grenzwert konvergiert. Gegeben sei der Pfad

$$\langle l, \nu\rangle \xrightarrow{d_1} \langle l, \nu + d_1\rangle \xrightarrow{d_2} \langle l, \nu + d_1 + d_2\rangle \xrightarrow{d_3} \langle l, \nu + d_1 + d_2 + d_3\rangle \xrightarrow{d_4} \ldots$$

Konvergiert die Folge $d_1, d_2, d_3, d_4, \ldots$ gegen einen Grenzwert d, so entspricht ihr Verhalten nicht mehr dem „realistischen“ Fortschreiten der Zeit bis ins Unendliche. Eine Konvergenz von Verzögerungen innerhalb eines Pfades soll demnach auch für die Definition von Syntax und Semantik der TCTL nicht erlaubt sein. Deshalb wird der Begriff der divergierenden Pfade $Pfade_{div}(s)$ definiert. Dafür wird eine formale Beschreibung der Dauer eines Pfades benötigt.

Definition 3.1.14 (Dauer eines Pfades)**.**
Sei $\mathcal{A}$ ein Timed Automaton oder Timed Statechart mit einer Menge von Aktionen $Akt \subset \Sigma$. Die Funktion Dauer : $Akt \cup \mathbb{R}_{>0} \rightarrow \mathbb{R}_{\geq 0}$ *ist definiert als*

$$Dauer(\tau) = \begin{cases} 0 & \textit{wenn } \tau \in Akt \\ d & \textit{wenn } \tau = d \in \mathbb{R}_{>0} \end{cases}$$

Für unendliche Ausführungsfragmente $\rho = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \ldots$ durch das Modell $\mathcal{T}(\mathcal{A})$, wobei $\tau_i \in Akt \cup \mathbb{R}_{>0}$ sei, gilt

$$Dauer(\rho) = \sum_{i=0}^{\infty} Dauer(\tau_i).$$

Die Dauer eines endlichen Ausführungsfragments wird entsprechend definiert.

□

Zeitdivergenz und Zeitkonvergenz werden damit wie folgt definiert.

Definition 3.1.15 (Zeitdivergenz und Zeitkonvergenz)**.**
Das unendliche Pfadfragment π heißt zeitdivergent*, wenn $Dauer(\pi) = \infty$. Ansonsten heißt π* zeitkonvergent*.*

□

Um die Semantik von TCTL festlegen zu können, wird die Menge aller zeitdivergenter Pfade benötigt.

Definition 3.1.16 (Zeitdivergente Pfadmengen)**.**
Für eine Zustand $s \in \mathcal{T}(\mathcal{A})$ gelte:

$\textit{Pfade}_{div}(s) = \{\pi \in \textit{Pfade}(s) \mid \pi \textit{ ist zeitdivergent}\}$.

□

Timelock Existieren, ausgehend von einem Zustand s, keine zeitdivergenten Pfade, so enthält der Zustand einen *Timelock*. Timelock bedeutet, dass in diesem Zustand Zeit nicht mehr voranschreitet, so wie es der Realität entspräche.

Definition 3.1.17 (Timelock)**.**
Sei $\mathcal{A}$ ein Timed Automaton oder Timed Statechart. Ein Zustand s aus dem Modell $\mathcal{T}(\mathcal{A})$ enthält ein Timelock*, falls gilt: $\textit{Pfade}_{div}(s) = \emptyset$. $\mathcal{A}$ heißt* Timelock-frei*, wenn kein Zustand aus der Menge der erreichbaren Zustände von $\mathcal{A}$ einen Timelock enthält.*

□

Timelocks sollten bei der Modellierung vermieden werden.

Zenoness Neben der Möglichkeit von Timelocks können auch so genannte *Zeno Pfade* Probleme verursachen. Ein Zeno Pfad entsteht, wenn in einem festen Zeitintervall unendlich viele Aktionen ausgeführt werden sollen. Dieses Phänomen wird ebenfalls als unrealistisch eingestuft, da keine unendlich schnellen Prozessoren existieren.

Definition 3.1.18 (Zeno Pfad)**.**
Sei $\mathcal{A}$ ein Timed Automaton oder Timed Statechart. Das unendliche Pfadfragment π aus dem Modell $\mathcal{T}(\mathcal{A})$ ist zeno *(oder ein* Zeno Pfad*), wenn es zeitkonvergent ist und unendlich viele Aktionen $a \in Akt$ entlang des Pfades ausgeführt werden sollen.*

□

Definition 3.1.19 (Nonzenoness)**.**
Ein Timed Automaton oder Timed Statechart $\mathcal{A}$ heißt non-zeno*, wenn kein initialer Zeno Pfad im Modell $\mathcal{T}(\mathcal{A})$ existiert.*

□

Unter der Hypothese, dass nur divergente Pfade durch einen non-zeno Automaten oder eine non-zeno Struktur betrachtet werden und es keine Timelocks gibt, können Syntax und Semantik der TCTL wie folgt festgelegt werden.

Syntax und Semantik der TCTL Sei X eine Uhrenmenge und $AC(X)$ eine Menge von Bedingungen an die Uhren. Die Bedingungen an die Uhren aus X müssen entweder in einer Position oder einer Transition eines Timed Automaton/ Timed Statechart wahr sein.

Definition 3.1.20 (Syntax TCTL)**.**
Die Grammatik einer TCTL-Zustandsformel *ist definiert als:*

$$\Phi ::= true \mid a \mid g \mid \Phi_1 \wedge \Phi_2 \mid \exists\, \varphi \mid \forall\, \varphi$$

mit $a \in Akt$, $g \in AC(X)$, einer Pfadformel φ und Zustandsformeln Φ, Φ_1 und Φ_2.
Die Grammatik der TCTL-Pfadformeln *definiert sich wie folgt:*

$$\varphi ::= \Phi_1 \mathcal{U}^J \Phi_2$$

wobei $J \subset \mathbb{R}_{\geq 0}$ ein Zeitintervall ist, dessen Grenzen natürliche Zahlen sind.

□

Folgende Abkürzungen sind üblich.

$\exists\Diamond^J\Phi \Leftrightarrow \exists$ *true* $\mathcal{U}^J\Phi$, $\forall\Diamond^J\Phi \Leftrightarrow \forall$ *true* $\mathcal{U}^J\Phi$

$\exists\Box^J\Phi \Leftrightarrow \neg\forall\Diamond^J\neg\Phi$, $\forall\Box^J\Phi \Leftrightarrow \neg\exists\Diamond^J\neg\Phi$

$\Diamond\Phi \Leftrightarrow \Diamond^{[0,\infty)}\Phi$, $\Box\Phi \Leftrightarrow \Box^{[0,\infty)}\Phi$, $\Phi\ \mathcal{U}\ \Psi \Leftrightarrow \Phi\ \mathcal{U}^{[0,\infty)}\Psi$

$\exists\Diamond_{(a,b)}\phi \Leftrightarrow \exists\Diamond_{=a}\exists\Diamond_{<(b-a)}\phi$

Definition 3.1.21 (Semantik TCTL, [BK08])**.**
Sei $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$ ein Timed Automaton (alternativ ein Timed Statechart), $a \in AP$ eine atomare Proposition, $g \in AC(X)$ und $J \subset \mathbb{R}_{\geq 0}$ ein Zeitintervall, dessen Grenzen natürliche Zahlen sind. Für einen Zustand $s = \langle l, \nu\rangle$ im Zustandsautomaten $\mathcal{T}(\mathcal{A})$ seien Φ, Ψ TCTL Zustandsformeln und φ eine TCTL Pfadformel. Die Erfüllbarkeitsrelation $\models$ wird induktiv wie folgt definiert.

$$
\begin{aligned}
s \models true &\Leftrightarrow \text{für alle } s\\
s \models a &\Leftrightarrow a \in L(l)\\
s \models g &\Leftrightarrow \nu \in g\\
s \models \neg\Phi &\Leftrightarrow \neg s \models \Phi\\
s \models \phi_1 \wedge \phi_2 &\Leftrightarrow (s \models \Phi) \text{ und } (s \models \Psi)\\
s \models \exists\, \varphi &\Leftrightarrow \pi \models \varphi \text{ für mindestens ein } \pi \in \text{Pfade}_{div}(s)\\
s \models \forall\, \varphi &\Leftrightarrow \pi \models \varphi \text{ für alle } \pi \in \text{Pfade}_{div}(s)\\
\pi \models \Phi\mathcal{U}^J\Psi &\Leftrightarrow \exists i \geq 0.\, s_i + d \models \Psi \text{ für mindestens ein } d \in [0, d_i]\\
&\qquad \text{mit } \sum_{k=0}^{i-1} d_k + d \in J \text{ und}\\
&\qquad \forall j \leq i.\, s_j + d' \models \Phi \vee \Psi \text{ für alle } d' \in [0, d_j]\\
&\qquad \text{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{i-1} d_k + d\\
\pi \models \Diamond^J\Phi &\Leftrightarrow \exists i \geq 0.\, s_i + d \models \Phi \text{ für mindestens ein } d \in [0, d_i]\\
&\qquad \text{mit } \sum_{k=0}^{i-1} d_k + d \in J\\
\pi \models \Box^J\Phi &\Leftrightarrow \forall i \geq 0.\, s_i + d \models \Phi \text{ für alle } d \in [0, d_i]\\
&\qquad \text{mit } \sum_{k=0}^{i-1} d_k + d \in J
\end{aligned}
$$

□

3.2 Testfallgenerierung aus Modellen

Bei der modellbasierten Testfallgenerierung wird eine Stimulifolge bestimmt, die zur Überdeckung eines Pfades im Testmodell führt. Die Testdaten sollen Zielkriterien nachweisen. Diese Zielkriterien werden als Eigenschaften an das System definiert, zum Beispiel strukturelle Modellüberdeckungen oder temporallogische Formeln. Die Testfallermittlung steht dann der Frage der Erfüllbarkeit gegenüber. Mit Erfüllbarkeit ist gemeint, ob das System oder Testmodell die gestellte Eigenschaft erfüllt oder nicht. Die Frage wird in den folgenden Kapiteln genauer betrachtet.

3.2.1 Model-Checking

Beim Model-Checking werden Eigenschaften des zu testenden Systems formal abgelegt und gegen eine Systemverhaltensbeschreibung geprüft.

Die Aufgabe eines Model-Checkers ist die *Verifikation des Systemmodells* gegen eine formalisierte Eigenschaft. Es gilt zu zeigen, ob das System die Eigenschaft erfüllt oder nicht. Ähnlich zum Model-Checking werden vom Tester Systemeigenschaften festgelegt und formalisiert. Der Unterschied besteht darin, dass der Model-Checker eine verneinte Eigenschaft erhält, die durch das Modell erfüllt sein soll. Eine solche Eigenschaft nennt man im Kontext der Testfallgenerierung ein *trap property*. Findet der Model-Checker ein Gegenbeispiel, das heißt ein Systemverhalten, welches der zu testenden Eigenschaft widerspricht, liefert er das Gegenbeispiel zurück. Das Gegenbeispiel kann dann als Testfall für die zu prüfende Eigenschaft eingesetzt werden, um zum Beispiel eine Systemimplementierung gegen die Eigenschaft zu testen. Dieses Gegenbeispiel wird auch als *witness* bezeichnet. Es ist ein Zeuge für die *Eigenschaft*, die nachgewiesen werden soll. Gezeigt werden kann anhand des Gegenbeispiels, ob sich Systemmodell und Systemimplementierung konform verhalten, zumindest bezogen auf den ausgewählten Testfall.

Es gibt einen Unterschied zwischen dem Einsatz von Eigenschaften beim Model-Checking und bei der Testfallgenerierung. Beim Model-Checking soll nachgewiesen werden, dass eine Eigenschaft für alle Pfade gilt, oder es zumindest einen Pfad gibt, für den die Eigenschaft gilt. Man gliedert die Eigenschaften dementsprechend in *universelle* und *existentielle* Eigenschaften. Universelle Eigenschaften müssen in jeder Systemsituation wahr sein und erfordern möglicherweise unendlich viele Pfade unendlicher Länge, um sie nachzuweisen [Pre03]. Bei existentiellen Eigenschaften muss mindestens ein Pfad durch das Modell existieren, für den sie wahr sind. Sie lassen sich mit einer endlichen Menge endlicher Pfade prüfen. Beim Model-Checking soll ein Beweis geführt werden, dass die Eigenschaft gilt und ansonsten ein Gegenbeispiel berechnet werden. Das Model-Checking bedient sich dafür mathematischer Verfahren, die eine eindeutige Aussage über die Korrektheit der Behauptung, dass das Systemmodell die Systemeigenschaft erfüllt, liefern. Die Verfahren *beweisen* somit die Korrektheit des Modells.

Im Gegensatz dazu wird bei der Testfallgenerierung mithilfe von Testmodellen und Systemeigenschaften kein Beweis der Korrektheit des Systemmodells gegenüber der Systemeigenschaft geführt. Das Ziel ist hierbei, das System zu testen, nicht das Testmodell. Beim Systemtest wird anhand exemplarischer Testfälle nachgewiesen, dass das zu testende System mit dem durch den *Testfall* spezifizierten Verhalten übereinstimmt. Dennoch kann es vorkommen, dass der Modelltest bei großen Modellen die einzige Möglichkeit der automatischen Verifikation darstellt, da der formale Beweis zu aufwändig wäre. In diesem Fall kann man aus einem vereinfachten Testmodell Testfälle generieren, die gegen das Systemmodell laufen.

Um einen Eindruck der Weiterentwicklungen im Bereich Model-Checking und der unterschiedlichen Vorgehensweisen zu erhalten, wird an dieser Stelle kurz auf die klassische Variante des Model-Checking (am Beispiel des CTL Model-Checkings und des TCTL Model-Checkings), auf das Symbolische Model-Checking mit Binären Entscheidungsdiagrammen und das Bounded Model-Checking eingegangen. Neue Wege in der Testfallgenerierung finden sich bei [BFPT06], [RT06] und [Pel09].

3.2.2 Explizites Model-Checking

Beim Expliziten Model-Checking wird die Transitionsrelation explizit über Adjazenzlisten (Nachbarschaftslisten) dargestellt. Eine Adjazenzliste eines (gerichteten oder ungerichteten) Graphen ist ein Array, dass zu allen Knoten die zum Knoten adjazenten Knoten enthält. Die Repräsentation von Graphen als Adjazenzlisten im Computer führt je nach Größe des Graphen zu hohen Speicher- und Analyseaufwänden. Dennoch war diese in den frühen 80er Jahren erstmals vorgestellte Vorgehensweise des Model-Checkings ein erfolgreicher und wegweisender Anfang aller nachfolgenden Weiterentwicklungen [CBRZ01].

Model-Checking mit Branching-time Logic Wir nehmen an, die deklarative Spezifikation der zu prüfenden Systemeigenschaften sei in CTL erstellt worden und der Zustands-Transitions-Graph eines Systems sei gegeben. Das Transitionssystem des Zustands-Transitions-Graphen sei endlich und nicht terminierend. Es gilt zu zeigen, dass ein Transitionssystem TS eine gegebene CTL Formel Φ erfüllt oder nicht, also ob gilt $TS \models \Phi$.

Definition 3.2.1 (Erfüllbarkeitsmenge)**.**
Bei einem gegebenen Transitionssystem TS, ist die Erfüllbarkeitsmenge $Sat_{TS}(\Phi)$*, oder kurz* $Sat(\Phi)$ *für eine CTL Zustandsformel* Φ *definiert als:*

$$Sat(\Phi) = \{s \in S \mid s \models \Phi\}.$$

Das Transitionssystem TS erfüllt die CTL Formel Φ *genau dann, wenn* Φ *in allen Initialzuständen von TS erfüllt ist:*

$$TS \models \Phi \Leftrightarrow \forall s_0 \in I \text{ gilt } s_0 \models \Phi.$$

Das ist gleichbedeutend zu $I \subseteq Sat(\Phi)$*.*

□

Um die Erfüllbarkeitsmengen generieren zu können, müssen alle atomaren Propositionen eines Zustands direkt für jeden beliebigen Zustand im Zustandsraum angegeben werden können. Dafür benötigt man eine *explizite* Repräsentation des Zustandsraum. Dieses Vorgehen verleiht der vorgestellten Variante des Model-Checkings den Namen *Explizites Model-Checking*. Diese älteste Form des Model-Checkings kann nur bei ausreichend kleinen Transitionssystemen angewandt werden.

Die grundlegende Vorgehensweise beim expliziten CTL Model-Checking lautet [CGP00]:

- Die CTL Spezifikation wird als Syntaxbaum dargestellt.
- Der Syntaxbaum wird von den Blättern ausgehend nach innen durchlaufen („bottom-up“). Dabei werden für Teilformeln Erfüllbarkeitsmengen $Sat(\Phi_i)$ generiert, die rekursiv zu der Erfüllbarkeitsmenge der Gesamtformel $Sat(\Phi)$ aufgebaut werden.
- Ist die Menge der Initialzustände eine Teilmenge der Erfüllbarkeitsmenge $Sat(\Phi)$, so erfüllt das Transitionssystem die CTL Spezifikation.

Der Syntaxbaum einer CTL Spezifikation ist für das Beispiel

$$\Phi = \underbrace{\exists\Diamond x}_{\Phi_1} \wedge \underbrace{\forall(y\,\mathcal{U}\,\underbrace{\exists\Box\neg z}_{\Phi_3})}_{\Phi_2}$$

in Abbildung 3.5 aufgespannt.

Die Blätter des Syntaxbaums bestehen entweder aus den Wahrheitswerten *true* und *false* oder aus atomaren Propositionen. Die Knoten des Baums repräsentieren die Teilformeln Φ_i der

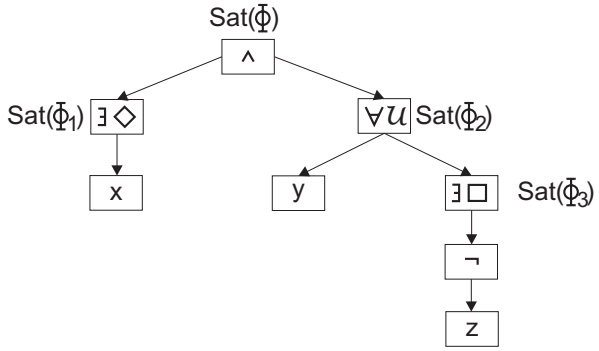


Abbildung 3.5: *Syntaxbaum einer CTL Formel*

Formel Φ. Nachdem die Erfüllbarkeitsmenge einer Teilformel Φ_i erstellt wurde, wird sie wie eine zusätzliche atomare Proposition a_{Φ_i} behandelt und Teil der übergeordneten Teilformeln.

Die Entscheidung, welche Zustände in einer Erfüllbarkeitsmenge $Sat(.)$ enthalten sind, ist über die folgende Charakterisierung festgelegt [BK08].

Satz 3.2.1 (Charakterisierung der Sat(.) Mengen für CTL Formeln in ENF)**.**
Sei $\mathcal{K} = (S, S_0, T, L)$ ein Kripke Struktur. Für alle CTL Formeln Φ, Ψ über AP gilt:

i) $Sat(true) = S$

ii) $Sat(a) = \{s \in S \mid a \in L(s)\}$, *für alle* $a \in AP$

iii) $Sat(\Phi \wedge \Psi) = Sat(\Phi) \cap Sat(\Psi)$

iv) $Sat(\neg\Phi) = S \backslash Sat(\Phi)$

v) $Sat(\exists \bigcirc \Phi) = \{s \in S \mid Post(s) \cap Sat(\Phi) \neq \emptyset\}$

vi) $Sat(\exists(\Phi \,\mathcal{U}\, \Psi))$ *ist die kleinste Teilmenge* M *aus* S*, für die gilt*

  a) $Sat(\Psi) \subseteq M$ *und*

  b) $s \in Sat(\Phi)$ *und* $Post(s) \cap M \neq \emptyset \Rightarrow s \in M$

vii) $Sat(\exists\Box\Phi)$ *ist die größte Teilmenge* M *aus* S*, für die gilt*

  a) $M \subseteq Sat(\Phi)$ *und*

  b) $s \in M \Rightarrow Post(s) \cap M \neq \emptyset$

wobei $Post(s) = \{s' \in S \mid (s, s') \in T\}$

□

Beispiel 9. *Die Kripke-Struktur aus Abbildung 3.1 (Beispiel 4, Abschnitt 3.1.2) wird durch eine weitere Transition erweitert zu der Kripke-Struktur $\mathcal{M}$ in Abbildung 3.6. Die Erfüllbarkeit der CTL-Formel $\Phi = \forall\Box(\texttt{light} \rightarrow \forall\Diamond\neg\texttt{power})$ auf der Kripke-Struktur soll gezeigt werden. Dafür wandelt man die CTL-Formel in die existentielle Normalform um (vgl. Beispiel 8, Abschnitt 3.1.3).*

$$\Phi = \underbrace{\neg\,\underbrace{\exists(true\,\mathcal{U}\,(\underbrace{\texttt{light} \wedge \underbrace{\exists\Box\texttt{power}}_{Sat(\Phi_3)}}_{Sat(\Phi_2)}))}_{Sat(\Phi_1)}}$$

Abbildung 3.6 enthält auf der rechten Seite den Syntaxbaum von Φ in ENF.

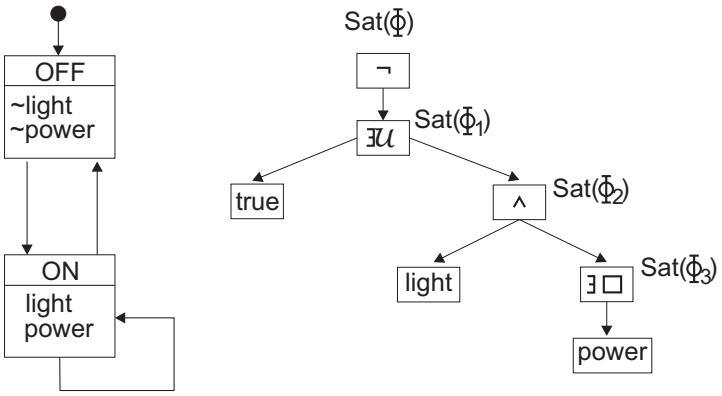


Abbildung 3.6: *Graph einer Kripke-Struktur und Syntaxbaum einer CTL-Formel*

Die Menge der Zustände von $\mathcal{M}$ lautet $S = \{\texttt{OFF}, \texttt{ON}\}$, die Folgezustände sind $Post(\texttt{OFF}) = \texttt{ON}$ und $Post(\texttt{ON}) = \{\texttt{OFF}, \texttt{ON}\}$. Die Erfüllbarkeitsmengen $Sat(.)$ lauten

$$
\begin{aligned}
Sat(\texttt{true}) &= \{\texttt{OFF, ON}\} \\
Sat(\texttt{light}) &= \texttt{ON} \\
Sat(\texttt{power}) &= \texttt{ON} \\
Sat(\Phi_3) &= \texttt{ON} \\
Sat(\Phi_2) &= Sat(\texttt{light}) \cap Sat(\Phi_3) = \texttt{ON} \\
Sat(\Phi_1) &= Sat(\exists true\, \mathcal{U}\ Sat(\Phi_2)) = \texttt{ON} \\
Sat(\Phi) &= Sat(\neg\Phi_1) = S \backslash Sat(\Phi_1) = \texttt{OFF}
\end{aligned}
$$

Da OFF ein Initialzustand ist, und somit gilt, dass $\{\texttt{OFF}\} \subseteq I$, gilt $\mathcal{M} \models \Phi$.

□

Model-Checking mit Timed Branching-time Logic Die Problemstellung beim TCTL Model-Checking ist äquivalent zum CTL Model-Checking. Zu zeigen ist, ob bei gegebener TCTL Formel Φ und Timed Automaton $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$ gilt, dass $\mathcal{A} \models \Phi$. Man geht davon aus, dass $\mathcal{A}$ non-zeno ist. Timelocks dürfen existieren.

Eine Graphenanalyse auf dem Zustands-Transitions-Graph $\mathcal{T}(\mathcal{A})$ ersetzt das Problem $\mathcal{A} \models \Phi$ und untersucht stattdessen $\mathcal{T}(\mathcal{A}) \models \Phi$. Dies ist aber nicht ohne weiteres möglich, da $\mathcal{T}(\mathcal{A})$ unendlich viele Zustände besitzt. Um dennoch über eine Graphenanalyse die Erfüllbarkeit der TCTL Formel zu untersuchen, wird ein endlicher Zustands-Transitions-Graph aus $\mathcal{T}(\mathcal{A})$ und Φ gewonnen, das so genannte *Region-Transitionssystem* $RTS(\mathcal{A}, \Phi)$. Die Zustände im RTS sind Äquivalenzklassen der Zustände aus $\mathcal{A}$, die alle dieselbe atomare Uhr-Invariante erfüllen und von denen „gleiche“ zeitdivergente Pfade ausgehen. Diese Zustände werden als *TCTL äquivalent* bezeichnet. Da die Anzahl der Äquivalenzklassen endlich ist, bieten sie die Grundlage für das TCTL Model-Checking. Erreicht wurde damit, dass man nun, anstatt zu beweisen, dass $\mathcal{A} \models \Phi$ mit einer TCTL Formel Φ gilt, bewiesen wird, dass eine CTL Formel auf dem Region-Transitionssystem erfüllt ist [BK08].

$$\mathcal{A} \models_{TCTL} \Phi \iff RTS(\mathcal{A}, \Phi) \models_{CTL} \hat{\Phi}$$

wobei $\hat{\Phi}$ eine CTL Formel ist, die aus der TCTL Formel Φ gewonnen wird. Die Transformation der TCTL in die CTL Formel wird durch Eliminierung der Zeitparameter erreicht. Dafür wird dem Timed Automaton eine neue Uhr z hinzugefügt, die eine Uhrauswertung η erweitert zu

$$\eta\{z := d\}(x) = \begin{cases} \eta(x) & \text{, wenn } x \in C \\ d & \text{, wenn } x = z. \end{cases}$$

Satz 3.2.2.
Sei $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$ ein Timed Automaton und $\Phi\, \mathcal{U}^J \Psi$ eine TCTL Formel über X und AP. Für eine Uhr $z \notin C$ sei

$$\mathcal{A} \oplus z = (\Sigma, L, L_0, X \cup \{z\}, I, T).$$

Für jeden Zustand s aus $\mathcal{T}(\mathcal{A})$ gilt

1. $s \models_{TCTL} \exists(\Phi\, \mathcal{U}^J \Psi)$ *genau dann, wenn*
   $s\{z := 0\} \models_{TCTL} \exists\Big((\Phi \vee \Psi)\, \mathcal{U}\, ((z \in J) \wedge \Psi)\Big)$
2. $s \models_{TCTL} \forall(\Phi\, \mathcal{U}^J \Psi)$ *genau dann, wenn*
   $s\{z := 0\} \models_{TCTL} \forall\Big((\Phi \vee \Psi)\, \mathcal{U}\, ((z \in J) \wedge \Psi)\Big)$

□

Eine TCTL Formel $\Phi = \exists\Diamond^{\leq 1} Closed$ wird durch die Eliminierung zu der CTL Formel $\hat{\Phi} = \exists\Diamond((z \leq 1) \wedge Closed)$. Für das Vorkommen nicht-verschachtelter Zeitbedingungen ist die Auflösung einfach. Zu klären ist noch, wie am besten im Falle verschachtelter Zeitbedingungen vorgegangen werden sollte. Eine verschachtelte Bedingung wäre zum Beispiel $\Phi = \forall\Box^{\geq 3}\exists\Diamond^{]1,2]}\, Closed$. Auch in diesem Fall kann die TCTL Formel durch Hinzufügen von Uhren in eine CTL Formel gewandelt werden. Allerdings benötigt man dazu zwei Uhren z_1 und z_2. Dann gilt

$$\hat{\Phi} = \forall\Box\Big((z_1 \geq 3) \rightarrow \exists\Diamond(z_2 \in [1,2] \wedge Closed)\Big).$$

Dieses Vorgehen führt dazu, dass die Anzahl der Uhren im Timed Automaton steigt und damit auch der Zustandsraums wächst. Mit jeder neuen Uhr nimmt die Größe des Region-Transitionssystem exponentiell zu.

Eine alternative Variante zur Zeitparametereliminierung benötigt nur eine weitere Uhr z. Mit Blick auf die rekursive Berechnung von Teilformeln Φ_i, wie sie in Definition 3.2.1 vorgestellt wurde, kann die Uhrvariable z pro Teilformelberechnung wiederverwendet werden.

Nach der Eliminierung der Zeitparameter müssen die Äquivalenzklassen auf den Zuständen des Timed Automaton gebildet und das Region-Transitionssystem aufgestellt werden. Danach können die bekannten CTL Model-Checking Verfahren auf das um zusätzliche Uhren erweiterte Region-Transitionssystem angewandt werden. Die Definition der Äquivalenzklassen soll an dieser Stelle nicht weiter vertieft werden. Näheres dazu kann unter Baier und Katoen [BK08] und Clarke et al. [CGP00] nachgelesen werden. Die Grundidee des TCTL Model-Checkings in algorithmischer Form wird in Abschnitt 5.3 skizziert.

3.2.3 Model-Checking auf Binären Entscheidungsdiagrammen

Das explizite Aufspannen des Zustandsraums, wie in Abschnitt 3.2.2 beschrieben, ist auf realistischen Systemgrößen nicht effizient. In den 90ern wurden erstmals größere Systeme durch den Einsatz symbolischer Variablen und Funktionen handhabbar. Die Grundidee beruht dabei auf der Repräsentation von Zustands- und Transitions*mengen* anstatt der expliziten Repräsentation von Adjazenzlisten pro Zustand. Die Vorgehensweise der Bestimmung von Erfüllbarkeitsmengen von Teilformeln, wie es beim CTL Model-Checking üblich ist, argumentiert ebenfalls auf Mengen von Zuständen. Das symbolische Model-Checking kann einige Ansätze wiederverwenden [CBRZ01].

Das Rechnen auf Zustandsmengen anstatt auf einzelnen Zuständen kann durch die Einführung von Boole-Funktionen erreicht werden. Hierfür wird jeder Zustand über einen binären Vektor kodiert. Dies geschieht über die Funktion $enc : S \rightarrow \{0,1\}^n$, S ist die Menge aller Zustände, n die Länge des Bit-Vektors. Die Bit-Vektorlänge kann über $n \geq \lceil \log |S| \rceil$ mit $|S| \geq 2$ geschätzt werden. Die Funktion ist zwar nicht surjektiv, man kann aber für jeden Bit-Vektor ohne zugeordneten Zustand einen Pseudo-Zustand annehmen, der über die Transitionsrelation nicht referenziert ist

und somit bei späteren Berechnungen auch nicht beachtet wird. Die Transitionsrelation wird über die Boole-Funktion $\Delta : \{0,1\}^{2n} \rightarrow \{0,1\}$ abgebildet. Sie gibt 1 zurück, falls das Zustandspaar (s, s') in der Transitionsrelation enthalten ist und 0 sonst [CBRZ01].

Beispiel 10. *Ein Transitionssystem sei anhand des Graphen in Abbildung 3.7 gegeben.*

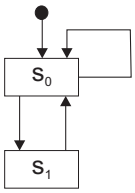


Abbildung 3.7: *Transitionssystem mit zwei Zuständen und drei Transitionen*

$S = \{s_0, s_1\}$. *Um die Zustände zu kodieren reicht ein Bit-Vektor der Länge 1,* $n \geq \lceil \log |S| \rceil \geq \lceil \log 2 \rceil = 1 (\log 2 \approx 0,3)$. *Für den Zustand* s_0 *nimmt der Bit-Vektor* x *den Wert* 0 *an, für den Zustand* s_1 *den Wert* 1. *Somit gilt* $enc(s_0) = 0$ *und* $enc(s_1) = 1$.

Die Transitionsrelation kann über die Boole-Funktion $\Delta = \neg x \vee \neg x'$ *beschrieben werden.* x' *meint in diesem Fall einen der Zustände aus* $Post(x)$*, der Menge der möglichen Folgezustände, die nach* x *über die Transitionsrelation erreichbar sind [BK08].*

x	x'	$\Delta = \neg x \vee \neg x'$
0	0	1
0	1	1
1	0	1
1	1	0

Tabelle 3.1: *Wahrheitstafel der Boole-Funktion* Δ

Eine Überprüfung anhand der Wahrheitstafel in Tabelle 3.1 zeigt, dass Δ *genau die erlaubten Zustandsübergange durch den Wahrheitswert* 1 *bestätigt.*

□

Die Darstellung einer Boole-Funktion und ihres Wertebereichs über Wahrheitstafeln ist für das obige kleine Beispiel ausreichend. Grundsätzlich ist die Verarbeitung von Wahrheitstafel durch Algorithmen aufwendig. Mit Boole-Funktionen können manche Operationen, wie zum Beispiel die Äquivalenz zweier Boole-Funktionen, effizienter anhand von *Binären Entscheidungsdigrammen (BDDs)* (engl: Binary Decision Diagrams) durchgeführt werden [CBRZ01]. Ein BDD ist ein gerichteter, azyklischer Graph, der bei einer fest vorgegebenen Ordnung der Variablen eine kanonische Form der Boole-Funktion darstellt.

Um zeigen zu können, ob ein Zustands-Transitions-Graph TS eine Eigenschaft Φ erfüllt oder nicht, wird für gewöhnlich ein Fixpunkt-Algorithmus angewendet [CBRZ01]. Hierbei wird der Algorithmus solange ausgeführt und dabei immer der neue Wert einer Funktion als Argument in die Funktion eingesetzt, bis das Argument und der Wert der Funktion gleich sind, kurz gesagt $f(x) = x$ gilt. Dieses Vorgehen wird bei der Untersuchung, ob zwei charakteristische Funktionen gleich sind, verwendet. Eine charakteristische Funktion $\chi_B : \{0,1\}^n \rightarrow \{0,1\}$ gibt an, ob ein Element in einer Menge B enthalten ist oder nicht. Die Gleichheit von Mengen wird damit über die Gleichheit von charakteristischen Funktionen bestimmt. Will man zum Beispiel die CTL-Eigenschaft $\Phi = \exists \Diamond \phi$ gegen den Zustands-Transitions-Graph TS prüfen, also untersuchen, ob $TS \models \Phi$ gilt, wird berechnet, ob ein Zustand, der ϕ erfüllt von einem Initialzustand aus erreichbar ist. Dazu nutzt man das BDD Format der Transitionsrelation.

Das Festlegen der passenden Boole-Funktion, um die Transitionsrelation zu repräsentieren ist nicht trivial und neben dem Aufstellen des BDDs mit die größte Herausforderung beim Symbolischen Model-Checking. Die Problematik der Zustandsraumexplosion verlagert sich in den Aufbau der BDDs. Obwohl es durch dieses Verfahren möglich wurde, bereits Systeme der Größe eines Steuergerätes in einem Fahrzeug behandeln zu können, konnten realistische industrielle Systemgrößen, mit zum Beispiel mehreren Steuergeräten, dennoch nicht betrachtet werden. Dies führte Anfang des 21. Jahrhunderts zu einer neuen Forschungsrichtung, dem *Bounded Model-Checking*. Es wird in folgendem Kapitel behandelt.

3.2.4 Bounded Model-Checking

Das Bounded Model-Checking ist die aktuell vielversprechendste Vorgehensweise, um große Systeme modellbasiert Testen zu können. Die Stärke liegt, aufgrund der Idee, den Zustandsraum nur teilweise abzurollen, im Finden von Testfällen. Das Bounded Model-Checking arbeitet ebenfalls auf der symbolischen Repräsentation von Zuständen über Bit-Vektoren. Die Transitionsrelation und das für den Model-Checker zu lösende Problem wird in Prädikatenlogik abgelegt. Die daraus entstehende Formel heißt *Bounded Model-Checking Instanz* (BMC-Instanz). Die BMC-Instanz ist genau dann lösbar, wenn das dazugehörende Zustands-Transitions-System einen endlichen Pfad enthält, der die Zielzustände erreicht. Diese Zielzustände hängen von dem zu lösenden Problem ab, das in Prädikatenlogik festgehalten wird. Die Eigenschaft führt zusammen mit der vorangegangenen Pfadauswahl durch das System/Testmodell zu einer Bedingung, im Englischen als „constraint“ bezeichnet, die gelöst werden muss. Die Lösbarkeit der BMC-Instanz wird von Satisfiability Modulo Theories (SMT) solvern[1] [RT06] bestimmt. Da es Boolsche Problemstellungen, constraints auf Integer oder reellwertiger Basis gibt, werden je nach mathematischer Problemstellung unterschiedliche SAT solver eingesetzt.

Eine BMC-Instanz hat folgenden Aufbau [Pel09]:

$$[[M,P]]_k = \bigwedge_{j=0}^{c-1} R(i(k_0+j), s(k_0+j), s(k_0+j+1)) \\ \wedge P(i(k_0), s(k_0), o(k_0), \ldots, i(k_0+c), s(k_0+c), o(k_0+c))$$

mit

- $i(k) = (x_1(k), \ldots, x_m(k))$ ist der Input-Vektor zum Berechnungsschritt k
- $s(k) = (s_1(k), \ldots, s_n(k))$ ist der Zustandsvektor zum Berechnungsschritt k
- $o(k) = \lambda(i(k), s(k))$ ist der Output-Vektor zum Berechnungsschritt k
- R ist die Transitionsrelation, die Vorzustände $s(k)$ und Inputs $i(k)$ zum Berechnungsschritt k den Folgezuständen zum Berechnungsschritt $k+1$ zuordnet
- P ist eine Eigenschaft bezüglich Inputs, Outputs und interner Zustände, die während der c Berechnungsschritte erfüllt sein müssen

Die BMC-Instanz zum Prüfen der CTL Formel $\Phi = \exists\Diamond\phi$ auf einem Zustands-Transitions-Graph TS mit Transitionsrelation T sieht für $c = 2$ wie folgt aus.

$$[[TS, \Phi]]_2 = R(i(0), s(0), s(1)) \wedge R(i(1), s(1), s(2)) \wedge \underbrace{(\phi(s_0) \vee \phi(s_1) \vee \phi(s_2))}_{\exists\Diamond\phi}$$

Die erhaltene BMC-Instanz wird als das zu lösende Problem (engl: constraint) an den SMT solver übergeben. Je nach Anwendungsgebiet können Input-, Output- und Zustandsvektor auch nicht-Boolsche Datentypen enthalten, zum Beispiel Integerwerte mit großen Datenbereichen

[1] Algorithmen, die über die Erfüllbarkeit einer constraint entscheiden.

oder Fließkommazahlen. Die sinnvolle Kombination unterschiedlicher SMT solver für unterschiedliche Datentypen beschleunigt oder ermöglicht sogar erst die Testfallgenerierung auf diesen Modellen.

Eine Vorgabe von c Berechnungsschritten, um die Abrolltiefe der BMC-Instanz zu beschränken, erklärt den Begriff *Bounded* in Bounded Model-Checking. Zusammen mit der BMC-Instanz wird die Transitionsrelation nur endlich viele vorgegebene Schritte abgerollt, und damit auch der Zustandsraum nur soweit betrachtet. Kann über die Erfüllbarkeit innerhalb dieser Schranke c nicht entschieden werden, kann man die Schranke erhöhen. Dies führt in der Konsequenz aber auch zu einem größeren Zustandsraum und den damit einhergehenden Problemen. Es gibt bereits mehrere Ansätze, um die BMC-Instanzen noch flexibler aufzubauen. Es kann zum Beispiel eine Cone-of-Influence Analyse vorgeschaltet werden, die berechnet, welche Teile in einem Modell sich überhaupt auf die interessanten Zustände auswirken. Dadurch kann die Größe der Transitionsrelation angepasst werden. Zudem kann in einer Kombination aus Simulation und constraint solving die BMC-Instanz verkleinert werden. Durch eine Simulation kann ein Teilpfad in die Nähe der interessanten Zustände gefunden werden und die BMC-Instanz beginnt dann nicht mehr in den Initialzuständen, sondern in Zuständen „in der Nähe“ der Zielzustände. Weitere Informationen dazu finden sich bei [CBRZ01, Pel09, PZck].

Beispiel 11. *Abbildung 2.6 zeigt das Statechart* `SC_Warnblinken`*.*
Die Transitionen *lauten*

$$
\begin{aligned}
\tau_1 &= (\texttt{OFF}, 0, in_Emer = PRESSED, \emptyset, \texttt{ON}) \\
\tau_2 &= (\texttt{ON}, 0, in_Emer = PRESSED, \emptyset, \texttt{ACTIVE}) \\
\tau_3 &= (\texttt{ACTIVE}, 0, \emptyset, \emptyset, \texttt{PASSIVE}) \\
\tau_4 &= (\texttt{PASSIVE}, 0, (lampsOnRequest! = NONE) \wedge \\
&\quad (curValue\ ! = lampsOnRequest), \emptyset, \texttt{ACTIVE}) \\
\tau_5 &= (\texttt{ON}, 0, lampsOnRequest = NONE, \emptyset, \texttt{OFF})
\end{aligned}
$$

Die Transitionsrelation *setzt sich zusammen aus*

$$
\begin{aligned}
\textit{auslöser}(\tau_1) &= \texttt{OFF} \wedge (in_Emer = PRESSED) \\
\textit{auswirkung}(\tau_1) &= \texttt{ON}' \\
\textit{auslöser}(\tau_2) &= \texttt{ON} \wedge (in_Emer\ ! = RELEASED) \\
\textit{auswirkung}(\tau_2) &= \texttt{ACTIVE}' \wedge (curValue = lampsOnRequest) \\
\textit{auslöser}(\tau_3) &= \texttt{ACTIVE} \wedge (lampsOnRequest\ ! = NONE) \wedge \\
&\quad (curValue\ ! = lampsOnRequest) \wedge \\
&\quad (in_Emer\ ! = RELEASED) \\
\textit{auswirkung}(\tau_3) &= \texttt{PASSIVE}' \wedge (curValue = lampsOnRequest) \\
\textit{auslöser}(\tau_4) &= \texttt{PASSIVE} \wedge (lampsOnRequest = NONE) \wedge \\
&\quad (in_Emer\ ! = RELEASED) \\
\textit{auswirkung}(\tau_4) &= \texttt{ACTIVE}' \\
\textit{auslöser}(\tau_5) &= \texttt{ON} \wedge (in_Emer = RELEASED) \\
\textit{auswirkung}(\tau_5) &= \texttt{OFF}'
\end{aligned}
$$

Input-Vektoren

$$
\begin{aligned}
i(0) &= in_Emer = PRESSSED \\
i(1) &= \emptyset \\
i(2) &= lampsOnRequest = LEFT \\
i(3) &= \emptyset \\
i(4) &= lampsOnRequest = NONE \\
i(5) &= in_Emer = RELEASED
\end{aligned}
$$

Zustandsformeln

$$\begin{aligned}
\varphi(0) &= \texttt{OFF}(0) \wedge (in_Emer(0) = RELEASED) \wedge \\
&\quad (curValue(0) = NONE) \wedge (lampsOnRequest(0) = NONE) \\
\varphi(1) &= \texttt{ON}(1) \wedge (in_Emer(1) = PRESSSED) \\
\varphi(2) &= \texttt{ACTIVE}(2) \wedge (curValue(2) = lampsOnRequest(1)) \\
\varphi(3) &= \texttt{PASSIVE}(3) \wedge (curValue(3) = lampsOnRequest(2)) \\
\varphi(4) &= \texttt{PASSIVE}(4) \wedge (curValue(4) = lampsOnRequest(3)) \\
\varphi(5) &= \texttt{ACTIVE}(5) \wedge (lampsOnRequest(5) = NONE) \\
\varphi(6) &= \texttt{OFF}(6) \wedge (in_Emer(6) = RELEASED)
\end{aligned}$$

Erster Teil der BMC-Instanz.

$$\begin{aligned}
&\bigvee_{j=0}^{5} R(i(k_0+j), s(k_0+j), s(k_0+j+1)) = \\
&\quad (in_Emer(0) = PRESSSED) \wedge \\
&\quad \big(0, (\texttt{OFF}(0), \begin{pmatrix} in_Emer(0) = RELEASED \\ curValue(0) = NONE \\ lampsOnRequest(0) = NONE \end{pmatrix}, \vec{0})\big) \wedge \\
&\quad \big(0, (\texttt{ON}(1), \begin{pmatrix} in_Emer(1) = PRESSED \\ curValue(1) = NONE \\ lampsOnRequest(1) = NONE \end{pmatrix}, \vec{0})\big) \wedge \ldots
\end{aligned}$$

Zweiter Teil der BMC-Instanz.

Ein Testfall fordert, dass der Position `ACTIVE` *innerhalb von 4 Zeitschritten die Position* `PASSIVE` *folgt. Dies wird in* P *über* $c = 6$ *Abrollschritte festgehalten.* z *ist eine dem Statechart* `SC_Warnblinken` *künstlich hinzugefügte Uhr.*

$$\begin{aligned}
&P(i(0), loc(0), \ldots, i(6), loc(6)) \equiv \\
&\quad \bigvee_{j=0}^{5} \big(\texttt{ACTIVE}(k_0+j) \wedge (z(k_0+j+1) < 4) \wedge \texttt{PASSIVE}(k_0+j+1)\big)
\end{aligned}$$

□

3.3 Requirements Tracing

Requirements Tracings steht für den Bezug von Anforderungen zu anderen, im Softwareentwicklungsprozess anfallenden Daten und Artefakten. Die verschiedenen Vorgehensweisen des Requirements Tracings sollen im vorliegenden Kapitel näher betrachtet werden. Zuerst werden bestehende Ansätze zusammengefasst (Abschnitt 3.3.1) und eine Eingrenzung der für die Arbeit passenden Verfahren vorgenommen. Abschnitt 3.3.2, Abschnitt 3.3.3 und Abschnitt 3.3.4 untersuchen im Anschluss die semi-formalen und formalen Herangehensweisen an das Requirements Tracing auch unter dem Gesichtspunkt der Effizienz und Umsetzbarkeit.

3.3.1 Bestehende Ansätze allgemein

Es existieren verschiedene Ausprägungen des Requirements Tracings im Modellbasierten Test [ARNRSG06, NZR07]. Liegt das Testmodell als Zustandsautomat vor, ist die einfachste Variante des Requirements Tracings das Annotieren der Anforderungen direkt an Testmodellelementen wie Zuständen oder Transitionen. Der Vorgang ist zwar leicht und schnell auszuführen, bietet aber auch nur wenig Spielraum bei der Umsetzung. Beschreibt eine Anforderung zum Beispiel

Systemverhalten und wird dadurch erst eine gewisse Abfolge von Zustandsübergängen der Anforderung wirklich gerecht, kann dies nicht adäquat abgebildet werden. Als Gegensatz dazu kann eine zusätzliche Modellierung der Anforderung operational (zum Beispiel durch Petri Netze oder Use-Case Diagramme) oder deklarativ (zum Beispiel anhand von Temporallogik) erfolgen. Sie erfordert zusätzlich zur Formalisierung der Anforderungen als Zustandsautomaten eine weitere Notation [NARZ05]. Dadurch wird zwar eine maximale Beschreibungsmächtigkeit erreicht, aber auch ein hoher Mehraufwand generiert, der nur schwer argumentiert werden kann.

Information über die Beziehung zwischen Anforderung und Testartefakt kann *explizit* oder *implizit* festgehalten werden. Implizit bedeutet, dass im Modell und/oder im Algorithmus bereits bei der Erstellung vermerkt wird, aufgrund welcher Anforderung die Implementierung vorgenommen wurde. Der explizite Verweis wird getrennt vom Modell durch den Tester und/oder einen Algorithmus festgelegt.

Die einfachste implizite Vorgehensweise bei der Anforderungsverfolgung innerhalb des Testprozesses legt in einer Matrixstruktur eine Eins zu Eins Verknüpfungen zwischen Anforderung und Testartefakt ab. Auf den modellbasierten Test übertragen bedeutet das Vorgehen, dass Elemente eines Testmodells mit Anforderungen verknüpft werden, allerdings ohne auf die Beziehung der einzelnen Elemente untereinander Bezug zu nehmen. Führt man einen Testfall gegen das Testmodell aus und durchläuft dieser ein bestimmtes Testmodellelement, bezieht sich der Testfall als Rückkehrschluss auf alle dem Element zugeordneten Anforderungen. Die durch eine derartige Zuordnung entstehende Matrix heißt *Traceabilitymatrix*. Die Matrix wird vom Testfallgenerator aufgrund der festgelegten Beziehungen zwischen den Anforderungen und den Modellelementen automatisch erzeugt. Je nach gewünschtem Detaillierungsgrad kann man Anforderungen zum Beispiel auf Gruppen von Zustandsautomaten, einzelne Zustandsautomaten oder Zustände und Transitionen innerhalb der Zustandsautomaten verweisen lassen. Ramesh und Jarke [RJ01] bezeichnen dieses Vorgehen als „Low-End Use of Traceability“. Es bezeichnet Vorgehensweisen, die keine zusätzliche Information als die bereits existierende zur Anforderungsverfolgung verwenden. Bestehende Artefakte und deren Elemente werden dabei durch *Links* mit den Anforderungen verknüpft.

Eine weitere implizite Anforderungsverfolgung im Bereich der Link-Erstellung beschreiben Bouquet et al. [BJL+05]. Sie vermerken bei der Erstellung die Anforderungsinformation in einem Testmodell. Das Modell ist keine graphische Repräsentation des Systems, sondern entspricht Programmcode. Es wird aus natürlichsprachlichen Anforderungen gewonnen. Während der Erstellung wird die Information für den Verweis auf Anforderungen in das Modell eingetragen. Fünf verschiedene Beziehungsarten zwischen Modellcode und Anforderung ermöglichen ein benutzerfreundliches Arbeiten. Der Anforderungsbezug wird als Teil des Testmodells programmiert, ähnlich zu Kommentaren in Programmcode. Einige am Markt erhältliche Modellierungs- oder Testfallermittlungswerkzeuge erlauben ebenfalls den Verweis auf Anforderungen direkt am graphischen Testmodellelement. Diese Vorgehensweise kann als weit verbreitet eingestuft werden.

Folgende explizite Formalismen der Anforderungsverfolgung werden auch im Fall der expliziten Testfallspezifikation verwendet und sind seit langem erprobt. Zu den natürlichsprachlichen bis formalen, expliziten Beschreibungen der Anforderungen, die teilweise eine graphische Repräsentation besitzen, zählen Petri-Netze [Rei85], Szenarios [NARZ05, Poh10], Use-Cases als eine abstrakte Szenariobeschreibung [FS05], Sequence Charts [NZR07], endliche Automaten, reguläre Ausdrücke, Constraints und logische Ausdrücke [ULP06].

Constraint Diagrams [Kle00], real-time symbolic diagrams [Sch01] und Varianten davon [Hoe99] haben zum Ziel, die ursprünglich formelbasierte Erstellung von Anforderungsbeziehungen anhand einer graphischen Notation zu unterstützen.

Um den Informationsgehalt der Anforderungsverfolgung zu verbessern, formalisieren diese Verfahren die Anforderungen zusätzlich zu der Formalisierung der Anforderungen als Testmodell.

Pretschner [Pre03] untersuchte in seiner Dissertation eine explizite Vorgehensweise der *Testfallspezifikation* für das *Model Checking*. Testfälle sind dabei eine Obermenge der Systemanforderungen und werden abstrakt formal beschrieben. Das Vorgehen basiert auf dem Wissen darüber, dass eine Anforderung existentielle und universelle Eigenschaften haben kann. Pretschner propagiert, dass existentielle Eigenschaften als Szenarien aufgefasst und mit entsprechenden

Formalismen beschrieben werden können, zum Beispiel als Sequenz- oder Use-Case-Diagramm. Universelle Eigenschaften lassen sich gut anhand von logischen Ausdrücken, auch für verteilte, parallele und asynchrone Systeme beschreiben. Mögliche Logiken, die grundsätzlich zum Beschreiben von Eigenschaften eines Zustands-Transitions-Systems verwendet werden können, sind temporale Logiken wie CTL und LTL [CGP00, DO03]. Sie werden ebenfalls verwendet, um die Testfallermittlung durchzuführen oder den Testfallgenerator bei der Testfallauswahl zu lenken [FW07, RH01]. Mithilfe von Syntax, Semantik und Operatoren lassen sich durch temporale Logiken Transitionssequenzen für reaktive Systeme beschreiben. Transitionssequenzen können dann mit Anforderungen identifiziert werden.

In der Praxis werden universelle Eigenschaften teilweise approximativ durch existentielle Eigenschaften beschrieben, um von dem Vorteil der Benutzerfreundlichkeit, zum Beispiel von Sequenz oder Use-Case-Diagrammen, profitieren zu können.

Die Idee, einen Bezug zwischen Anforderungen und Testfällen mittels aussagenlogischer Ausdrücke herzustellen, verfolgen ebenfalls Callahan et al. [CSE96], Ammann et al. [ADX01], Li Tan et al [TSL04] und Fraser und Wotawa [FW08b]. Sowohl bei Pretschner als auch bei den anderen, im Model Checking eingeordneten Arbeiten, sind der Startpunkt der Untersuchungen die Eigenschaften der Systemanforderungen. Der Testingenieur muss sich aufgrund der natürlichsprachlich vorliegenden Anforderungen überlegen, welche Eigenschaften das Modell erfüllen muss, um der Anforderung gerecht zu werden, welche davon universell und welche existentiell sind. Liegen diese Eigenschaften dann in logischen Ausdrücken vor, können sie zur Testfallermittlung herangezogen werden oder dem nachträglichen Abgleich der generierten Testfälle dienen.

Die rein formale Anforderungsverfolgung, bei der die Anforderung formalisiert abgelegt wird, um sie dann auf Testfälle beziehen zu können, bietet den höchstmöglichen Informationsgehalt und kann zur automatischen Testfallermittlung eingesetzt werden. Sie ist allerdings aufwändig in der Erstellung und Wartung und zudem fehleranfällig [Pre03]. Graphische Repräsentationen der logischen Ausdrücke [BKO05, Sch01] erleichtern zwar die Erstellung und verbessern die Verständlichkeit der Formeln, die Festlegung der existentiellen und universellen Eigenschaften entfällt jedoch nicht. Im folgenden Abschnitt werden ausgesuchte graphische Notationen und deren Formalisierung genauer betrachtet.

Graphische Notationen in der Anforderungsspezifikation können helfen, neben der operationalen auch die deklarative Spezifikation leichter und fehlerfreier festzulegen. Um dies zu erreichen kann grundsätzlich auf zwei Arten vorgegangen werden. Einerseits kann das Ziel sein, eine Anforderung in einer vorgegebenen Wunschlogik zu spezifizieren, die anhand einer graphischen Notation abgebildet werden soll. Andererseits kann eine graphische Notation zur Beschreibung von Anforderungen bereits existieren und diese soll in eine (möglicherweise neue) Logik übersetzt werden.

Unterschiedliche Ansätze verfolgen Bemühungen, die Anforderungsverfolgung automatisch zu generieren [EG02, ZSPMK03]. Um eine vollständig automatische und aussagekräftige Anforderungsverfolgung zu erhalten, sollten die Anforderungen oder zu verfolgenden Artefakte zumindest semi-formal vorliegen.

3.3.2 Visuelle Notationen

Einer Anforderungsbeschreibung muss eine zumindest semi-formale Notation zugrunde liegen, um automatische Auswertungen umsetzen zu können. Eine graphische Notation ist dabei nicht zwingend notwendig. Allerdings steigt die Akzeptanz und Verständlichkeit der Beschreibungssprache bei graphischen Repräsentationen. Dieser Ansatz gilt, solange der Inhalt der zu beschreibenden Eigenschaft nicht zu komplex und umfangreich ist.

Im vorliegenden Abschnitt werden anwendbare graphische Notationen vorgestellt, die theoretisch zur Anforderungsbeschreibung herangezogen werden können.

Message Sequence Charts (MSC)

Ein MSC (Message Sequence Chart) ist eine graphische Sprache zur Modellierung von Szenarien, die auf die Kommunikation zwischen Prozessen konzentriert ist. Ein MSC ist eine zweidimensionale Graphik, die ebenfalls eine formale Repräsentation partieller Ordnung ist. Ihre Hauptverwendung liegt in der Modellierung und Verifikation von Protokollkommunikation. Es ist sowohl möglich, einzelne Szenarien zu beschrieben, die als Basic Message Sequence Chart bezeichnet werden, als auch mehrere einzelne Charts zu einem High-Level Message Sequence Chart zu kombinieren, so dass ein unendlich großer Zustandsraum aufgespannt werden kann. Folgende Ausführungen sind [MP05], [APP95] und [Uni99] entnommen.

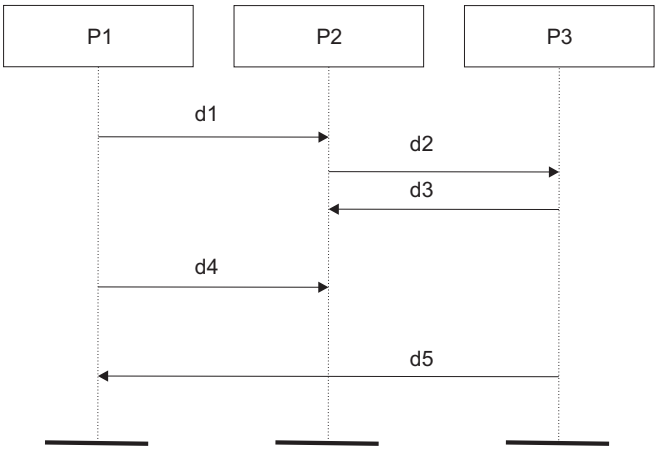


Abbildung 3.8: *Basic MSC*

Abbildung 3.8 stellt ein Basic MSC dar. Die Zeitlinien enthalten die Namen der Prozesse, zwischen denen Nachrichten versendet werden können. Die Reihenfolge der versendeten Nachrichten ist geordnet, in dem Sinne, dass nacheinander angelegte Nachrichten auch zeitlich nacheinander oder gleichzeitig eintreten.

Es wird davon ausgegangen, dass Nachrichten nach dem FIFO (first in, first out) Prinzip weitergeleitet werden. Als Konsequenz können sich Nachrichten innerhalb eines Kanals nicht überkreuzen.

Da die meisten Systeme mehrere (oder sogar unendlich viele) MSCs benötigen, um sie adäquat zu beschreiben, benötigt man einen high-level Formalismus, um miteinander verknüpfte MSCs in einem so genannten HMSC (high-level Message Sequence Chart) beschreiben zu können. Jeder Pfad durch ein HMSC entspricht dann einer Ausführung bzw. einem Szenario des Systems.

Drei Aspekte machen MSCs zu einer eher schwachen Notation, um Anforderungen, die in eine Temporallogik übersetzt werden sollen, zu spezifizieren [HT04]. Einerseits können keine Bedingungen an Nachrichten gestellt werden. Es ist nicht möglich, zu spezifizieren „wenn P die Nachricht M zu Q sendet, *muss* Q die Nachricht zu R weiterleiten“. Ob eine Aktion passieren *muss* oder *kann*, ist somit nicht spezifizierbar. Andererseits ist der Bezug zwischen dem MSC und der Implementierung zum Beispiel in einem Zustandsautomaten nur mit hohem Aufwand herzustellen. Drittens können Bedingungen an zeitliche Abhängigkeiten nur innerhalb eines MSCs festgelegt werden und nicht zwischen mehreren MSCs in einem HMSC. Diese zeitlichen Abhängigkeiten zwischen parallelen Prozessen sind für eine ausführliche Anforderungsbeschreibung notwendig. Eine Erweiterung der MSC, um die Schwachstellen zu beheben, wird im übernächsten Paragraph *Live Sequence Charts* [DH01] vorgestellt.

Sequenzdiagramme

Sequenzdiagramme (zum Beispiel aus der UML [RHQ+05]) unterscheiden sich nicht maßgeblich von MSCs. Meist wird bei Fragestellungen zu Syntax und Semantik von Sequenzdiagrammen auf MSCs verwiesen. Ein Unterschied besteht jedoch darin, dass Sequenzdiagramme generische Sequenzen enthalten können. Generische Sequenzdiagramme beschreiben Interaktionen zwischen frei wählbaren Objekten. Es ist also nicht vorgeschrieben, welche Objekte tatsächlich bei einer konkreten Sequenz zur Ausführung kommen. Aufgrund dessen definierten Cho et al. [CKCB02] eine Semantik speziell für generische Sequenzdiagramme.

Generische Sequenzdiagramme sind auf einer variablen Objektmenge aufgebaut. Die Objekte können nicht fixiert werden, da sich die Konfiguration der Objekte über die Zeit ändert. Die bekannten Quantifizierer $\forall$ und $\exists$ können nur auf einer fixen Objektmenge angewandt werden, was Cho et al. veranlasste, die Einsatzmöglichkeit der bekannten Quantoren in einer neuen Semantik zu überdenken. Die Semantik basiert auf der Temporallogik *Half-order Dynamic Temporal Logic* (HDTL), die die so genannte *freeze quantification* [Hen90] nutzt.

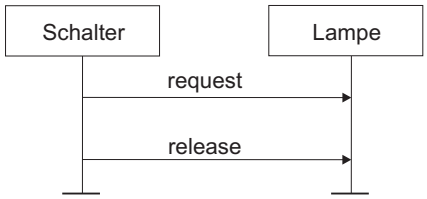


Abbildung 3.9: *Generisches Sequenzdiagramm Blinken*

Abbildung 3.9 zeigt sehr vereinfacht die Kommunikation zwischen einem Schalter und einer Lampe in einem Fahrzeug. Das Sequenzdiagramm ist generisch, das heißt, es spiegelt eine Summe von Schalter- und Lampenvarianten wider. Sowohl ein Blinkvorgang als auch das Einschalten der Nebelschlusslichter ist dadurch abbildbar. Beim Blinken als auch beim Nebelschlusslicht sind unterschiedliche Schalter und Lampen beteiligt. Beim Blinken wären das zum Beispiel Warnblinkschalter oder Blinkerhebel und die im Fahrzeug zum Blinken verbauten Lampen, von vorne links bis hinten rechts. Das Nebelschlusslicht wird wiederum von einem anderen Schalter betätigt und dieser steuert andere Lampen als die Blinklampen im Fahrzeug an.

Ein *trace* durch das Sequenzdiagramm ist eine endliche Abfolge von Ereignissen, die Instanzen der generischen Objekte des Diagramms enthält.

Live Sequence Charts (LSC)

Mit den 2001 von Damm und Harel [DH01] eingeführten Live Sequence Charts (LSCs) wurden einige Schwachstellen der Message Sequence Charts (MSCs) bei der Spezifikation von Eigenschaften behoben. Mit die wichtigste Schwachstelle betrifft die Möglichkeit zwischen verpflichtendem und möglichem Eintreten von Ereignissen zu unterscheiden. Zudem können Teilbereiche eines LSCs als universell notwendig, andere als existentiell, also nur auf mindestens einen Systemlauf beschränkt, festgelegt werden. Häufig ist es bei der Spezifikation einer Eigenschaft nötig, zuerst eine Vorbedingung zumindest existentiell zu erreichen, von der ausgehend dann alle weiteren Entwicklungen universell erfüllt sein müssen. Für diesen Fall wurde der so genannte Pre-Chart Bereich eingeführt. Neben diesen erwähnten existieren einige weitere Erweiterungen der MSCs, die zusammen genommen ein LSC umfassen.

Der Aufbau der graphischen Notation eines LSC lehnt sich stark an die im ITU definierte MSC Notation an, um die Vorteile des bereits etablierten MSC Standards nutzen zu können.

In Abbildung 3.10 sind die zwei Varianten universelles LSC (zu erkennen an der durchgezogenen Umrandung) und existentielles LSC (zu erkennen an der gestrichelten Umrandung) abgebildet. Das universelle LSC wendet die darin enthaltenen Restriktionen auf alle Systemläufe an.

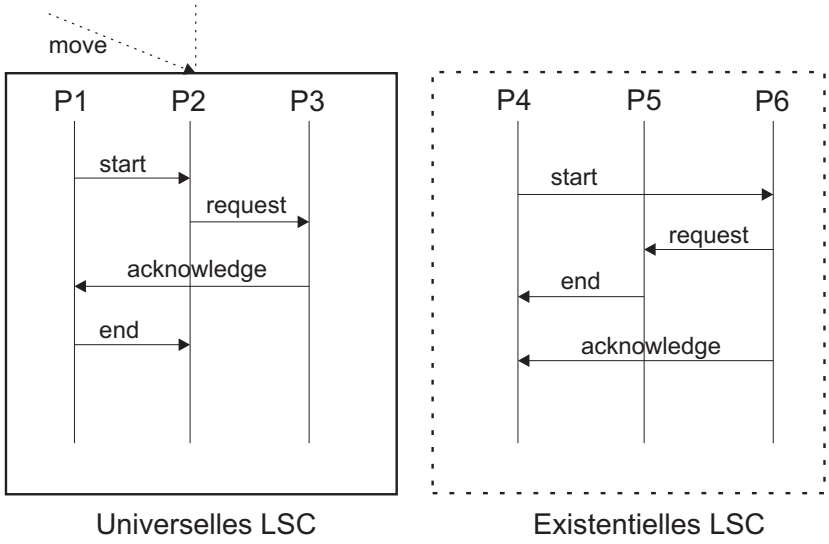


Abbildung 3.10: *Universelles und existentielles LSC [DH01]*

Existentielle LSCs müssen nur in mindestens einem Systemlauf erfüllt sein.

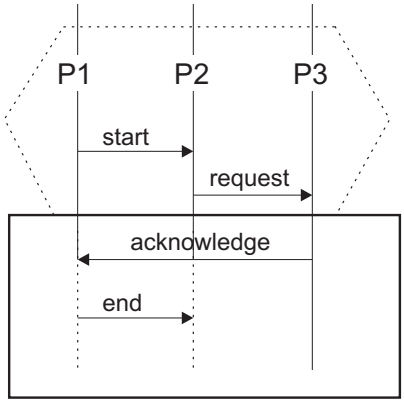


Abbildung 3.11: *Darstellung von Pre-Charts und „möglichen“ Bedingungen in LSCs [DH01]*

Einem universellen oder existentiellen LSC kann ein Pre-Chart vorgeschaltet werden. Dies ist in Abbildung 3.11 zu sehen (das gestrichelte Sechseck). Ein Pre-Chart stellt eine Vorbedingung dar, die erfüllt sein muss, bevor das eigentliche LSC zur Ausführung kommt. Die gestrichelten Anteile der P1 und P2 Zeitlinien markieren einen Bereich, dessen Aktionen nur *möglicherweise* eintreten können. Die Nachricht *end* kann, muss aber nicht von P1 an P2 gesendet werden.

Visual Timed event Scenarios (VTS)

Visual Timed event Scenarios (VTS) erlauben es, im Gegensatz zu MSCs, auch komplexe Systemereignisse mit Hilfe von Szenarios zu beschreiben. Der Testingenieur soll durch eine graphische Repräsentation bei der Beschreibung einer Systemeigenschaft unterstützt werden und zudem ein formal korrektes Szenario erhalten, das gegen ein Modell ausgeführt werden kann. Braberman et al. [BKO05] zeigen, dass die Überprüfung, ob ein Timed Automaton eine VTS-Szenario erfüllt, entscheidbar ist.

Der dazugehörende Formalismus unterstützt Gleichungen partieller Ordnung und Echtzeitbedingungen.

Abbildung 3.12 zeigt beispielhaft ein VTS Szenario. Das Model-Checking auf Timed Automata und VTS Szenarios kann bei Braberman et al. [BKO05] nachgelesen werden.

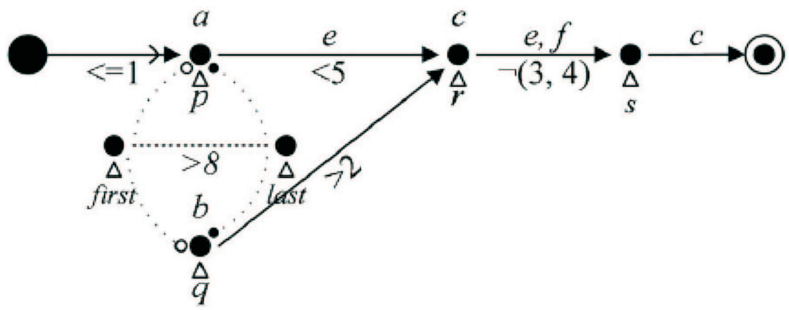
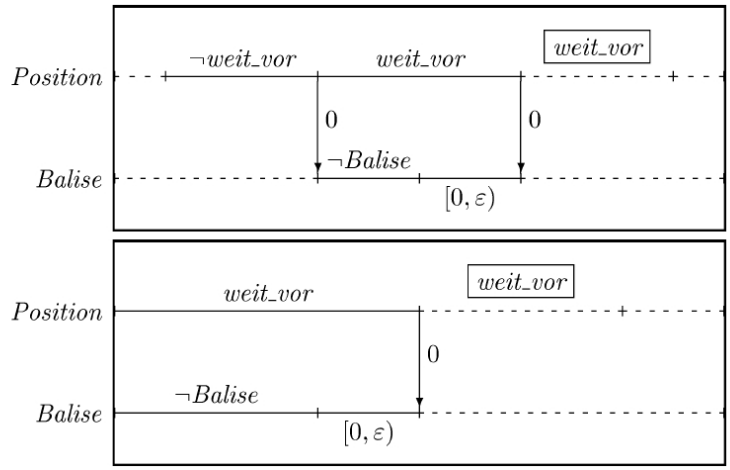


Abbildung 3.12: *VTS Szenario [BKO05]*

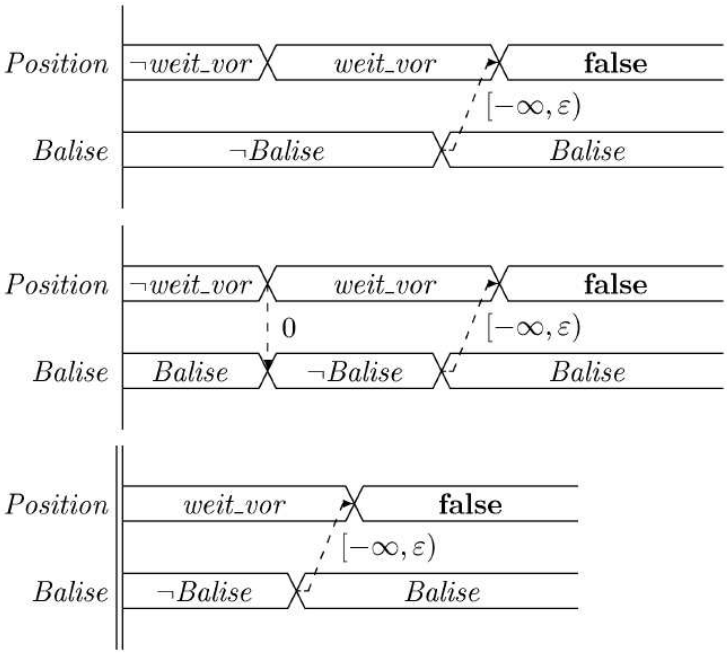
Symbolic Timing Diagrams

Eine Variante zur graphischen Repräsentation temporallogischer Ausdrücke beschreibt Schlör [Sch01] in seiner Dissertation an der Universität Oldenburg, die *Symbolic Timing Diagrams*. Mehrere Veröffentlichungen und Dissertation sind an der Universität Oldenburg im Rahmen der graphischen Beschreibung speziell von Timing Eigenschaften entstanden. So auch die Master Thesis von Hoenicke [Hoe99], der einen Vergleich von Constraint Diagrams und Real-Time Symbolic Timing Diagrams (eine Erweiterung des Ansatzes von Schlör) durchführt. Beide Formalismen dienen dazu, Systemeigenschaften von Echtzeitsystemen zu erfassen. Im Gegensatz zu der möglichen automatischen Auswertung der Real-Time Symbolic Timing Diagrams können die Constraint Diagrams bisher nicht automatisch verifiziert werden.

Abbildung 3.13(a) und Abbildung 3.13(b) zeigen die unterschiedliche Darstellung der Anforderung an einen Zug, rechtzeitig vor einer Gefahrenstelle zum Stehen zu kommen. Der Zug kann per Funk Schranken und Weichen fernbedienen. In regelmäßigen Abständen fährt der Zug über ein im Gleisbett verlegtes passives Element (genannt Balise), das vom Zug angeregt wird. Die Balise liefert bei Anregung eine Kennung, wodurch der Zug erkennen kann, auf welchem Streckenabschnitt er sich befindet. Hoenicke zeigt in seiner Arbeit, wie die Anforderungen an die Zugsteuerung in Real-time Symbolic Timing Diagrams und Constraint Diagrams umgesetzt werden kann [Hoe99].



(a) Constraint Diagram [Hoe99]



(b) Real-time Symbolic Timing Diagram [Hoe99]

Abbildung 3.13: *Symbolic Timing und Constraint Diagram*

3.3.3 Logik-Patterns

Logik-Patterns sind vordefinierte logische Bausteine, die zu einer neuen logischen Aussage verknüpft werden können. Dwyer et al. [DAC99] schlagen vor, die gebräuchlichsten temporallogischen Aussagen anhand von Patterns zu hinterlegen, die dann anhand ihrer natürlichsprachlichen Beschreibung eingesetzt werden können, um eine Eigenschaft zu spezifizieren. Dadurch muss der Spezifikateur der Eigenschaft über die Umsetzung einer Aussage in einen temporallogischen Ausdruck nur wenig bis gar nicht informiert sein. Zudem können der natürlichsprachlichen Beschreibung der Aussage unterschiedliche Temporallogiken zugeordnet werden, zum Beispiel eine CTL und eine LTL Variante und so kann je nach Bedarf eine Formel in LTL und eine in CTL erzeugt werden. Das *Precedence Pattern* ist ein von Dwyer et al. vorgestelltes Pattern, das immer dann genutzt wird, wenn ein Zustand q einem Zustand p vorausgehen muss. Eine Blinkanforderung, die dieses Pattern benötigt, wäre zum Beispiel `SLH-xxx` : *Ein Lampe blinkt nur, wenn der Blinkhebel betätigt wurde.* Die natürlichsprachliche Eigenschaft

Globally (`Blinkhebel=(Links oder Rechts)` precedes `Blinklampe=An`)

resultiert in der CTL-Formel

$$\Phi = \neg\exists\big(\neg(\texttt{Blinkhebel=(Links } \vee \texttt{ Rechts)})\,\mathcal{U}\;((\texttt{Blinklampe=An}) \wedge \neg(\texttt{Blinkhebel=(Links } \vee \texttt{ Rechts)}))\big)$$

3.3.4 Observerautomaten

Observerautomaten sind Zustandsautomaten, die allein die wesentlichen Stimulationen einer einzelnen Anforderung beschreiben [BHJP05]. Sie werden nicht zur Testfallgenerierung genutzt, sondern laufen parallel zur Testausführung mit. Wurde ein Observerautomat vollständig strukturell überdeckt, wurde die Anforderung am Testobjekt erzeugt. Ein Observerautomat für die Blinkanforderung *`SLH-yyy` : Bei Betätigung des Warnblinkschalters werden alle Blinklampen angesteuert.*, ist ein Zustandsautomat, der den Warnblinkschalter und alle Blinklampen beobachtet. Sobald der Warnblinktaster betätigt wurde, wird auf die Hellphase aller Blinklampen gewartet. Sind alle Lampe angesteuert worden, ist die Anforderung am Testobjekt stimuliert worden. Das korrekte Verhalten der Lampenansteuerung ist innerhalb des Testmodells abgelegt und nicht Teil des Observerautomaten. Der Observerautomat *beobachtet* nur.

Kapitel 4

Computation Sequence Chart

Ein *Computation Sequence Chart* (CSC) ist ein Formalismus, der es ermöglicht, Anforderungen und Testmodelle miteinander zu verknüpfen [Gro11]. Ein CSC setzt sich aus Elementen bestehender Zustands-Transitions-Graphen zusammen und beschreibt eine Gruppe von Systemausführungen („*computations*“), die das spezifizierte Verhalten einer Einzelanforderung widerspiegeln. Vorliegendes Kapitel erläutert die graphische Notation, Syntax und Semantik eines CSC.

Ein CSC soll zusammen mit dem Testmodell zur Testfallgenerierung und zum Abgleich mit existierenden Testfällen herangezogen werden. Dafür benötigt das CSC eine logische Repräsentation. Die Frage der Erfüllbarkeit eines CSCs auf einem Testmodell wird über den Einsatz temporallogischer Formeln umgesetzt. Temporallogische Formeln sind im Rahmen der Modellbasierten Testfallermittlung etabliert. Zu jeder Temporallogik existieren Publikationen zur Erfüllbarkeit ihrer Formeln auf einem Modell. Durch den Einsatz der Temporallogik TCTL wird die Definition einer neuen Erfüllbarkeitsrelation $\models$ explizit für CSCs vermieden.

Die Interpretation einer CSC Anforderungsbeschreibung als TCTL-Formel Φ ist Inhalt des vorliegenden Kapitels.

4.1 Grundlagen

Beim Modellbasierten Test wird Systemverhalten in Zustands-Transitions-Graphen abgebildet. Der Zustands-Transitions-Graph entspricht dem erwarteten Systemverhalten. Ein atomares Systemverhalten, wie es in einer atomaren natürlichsprachlichen Anforderung beschrieben ist, lässt sich am leichtesten über den Zustands-Transitions-Graphen identifizieren, indem ein Teilpfad durch den Graph gewählt wird, der der atomaren Anforderung am besten gerecht wird.

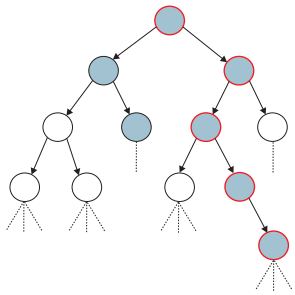


Abbildung 4.1: *Pfad durch den Berechnungsbaum*

Aus Sicht der Zustands-Transitions-Graphen, die das Verhalten des Systems abbilden, entspricht eine Anforderung ebenfalls einem oder mehreren Pfaden durch den aufgespannten Berechnungsbaum des Graphen. Abbildung 4.1 zeigt einen Berechnungsbaum. Der rot hervorge-

hobene Pfad durch den Baum kann einerseits als ein spezifisches Systemverhalten verstanden werden und erfüllt andererseits ausgewählte Systemeigenschaften. Der Berechnungsbaum beschreibt alle, vom Startpunkt aus möglichen Ausführungen des Systems. Eine einzelne Anforderung beschreibt eine endliche Menge dieser Ausführungen. Somit lassen sich alle Systemanforderungen, seien es Systemverhaltensbeschreibungen oder Systemeigenschaften, über die Spezifikation einer Menge von Pfaden durch den Berechnungsbaum beschreiben. Eine Systemeigenschaft muss dann bei jeder Systemausführung, die alle im Berechnungsbaum des Zustands-Transitions-Graphen zu finden sind, gelten. Es liegt nahe, das Requirments Tracing anhand dieser Pfadauswahl durchzuführen. Die Festlegung erwünschter Pfade durch einen Berechnungsbaum kann über Temporallogiken erfolgen. Die Spezifikation der temporallogischen Formeln ist jedoch aufwändig und fehleranfällig.

Ein einzelnes Systemverhalten lässt sich leichter über einen ausgewählten Pfad durch das Systemmodell abbilden, wie in Abbildung 4.2 zu sehen.

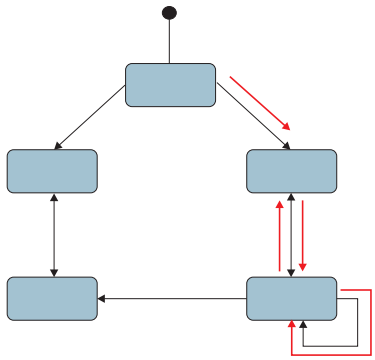


Abbildung 4.2: *Systemverhalten im Systemmodell*

Der rote Pfad durch den Zustandsautomaten entspricht einem spezifizierten Systemverhalten. Das Systemmodell entspricht der Summe aller verhaltensbeschreibenden Systemanforderungen. Da Anforderungen an ein System atomar definiert sind, kann mit Hilfe einer Klasse von Pfaden die Anforderung abgebildet werden.

Der Pfad durch den Automaten in Abbildung 4.2 entspricht, je nach Datenbelegung alternativen Pfaden durch den Berechnungsbaum. Dies führt zu der Entscheidung, als passende Temporallogik die Branching-time Logik Timed CTL anstatt Timed LTL zu wählen.

Beide Varianten, sowohl die Spezifikation der Systemeigenschaften über temporallogische Ausdrücke als auch die Auswahl von charakteristischen Teilpfaden durch das Systemmodell müssen sinnvoll kombiniert werden. Dies führt als Konsequenz zu einer graphischen Repräsentation von Pfaden durch das Systemmodell, genannt *Computation Sequence Chart*, die in Temporallogik umgewandelt werden.

Zur Definition der Syntax und Semantik eines CSC seien folgende Voraussetzungen gegeben. Ein Testmodell sei entweder eine Komposition aus Timed Statecharts $\mathbf{S} = S_1 \parallel S_2 \parallel \ldots \parallel S_n$ oder aus Timed Automata $\mathbf{A} = \mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \ldots \parallel \mathcal{A}_n$. Zur Wiederholung die Definition des Timed Automaton und Timed Statechart.

Definition 4.1.1 (Timed Automaton, [CGP00])**.**
Ein Timed Automaton *ist ein 6-Tupel* $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$*, mit*

- Σ *ist das endliche* Alphabet
- L *ist die endliche Menge der* Positionen
- $L_0 \subseteq L$ *ist die Menge der* Anfangspositionen
- X *ist die Menge der* Uhren

- $I : L \to C(X)$ *ist eine Abbildung von Positionen auf Bedingungen an Uhren, die* Positionsinvariante
- $T \subseteq L \times \Sigma \times C(X) \times 2^X \times L$ *ist eine Menge von Transitionen. Das 5-Tupel* $\langle l, a, \varphi, \lambda, l' \rangle$ *entspricht einer Transition von der Position* l *zur Position* l' *markiert mit Aktionen* a*, einer zeitlichen Bedingung* φ*, die angibt, wann die Transition genommen werden kann und einer Menge an Uhren* $\lambda \subseteq X$*, die auf Null gesetzt werden, sobald die Transition schaltet.*

□

Definition 4.1.2 (Timed Statechart, [PZck])**.**
Gegeben sei eine Variablenmenge V *und eine Timermenge* T*. Sei Expr*(V) *die Menge der Aussagen auf den Variablen aus* V *und Bexpr* (V) *die Menge Boolscher Aussagen auf* V*. Ein* Timed Statechart *ist ein 6-Tupel* $S = (L, L_0, P, G, A, \Lambda)$*, mit*

- L *ist die endliche Menge der* Positionen
- $L_0 \subseteq L$ *ist die Menge der* Anfangspositionen
- P *ist die Menge der* Transitionsprioritäten
- G *ist die Menge der* Transitionsbedingungen*.*
- $A = \mathbb{P}(V \times$ *Expr*$(V))$ *ist die Menge der* Aktionen*.*
- $\Lambda \subseteq L \times P \times G \times A \times L$ *ist die Menge der* Transitionen*. Das 5-Tupel* $\langle l, p, g, \alpha, l' \rangle$ *entspricht einer Transition von der Position* l *zur Position* l' *markiert mit* α*, einer Bedingung* $g \in$ *Bexpr* (V)*, die angibt, wann die Transition genommen werden kann und einer Priorität* p*.*

□

Die Menge aller paralleler Automaten eines Testmodells sei allgemein, sowohl für Timed Statecharts als auch für Timed Automata, mit SC bezeichnet. Es gilt somit entweder $SC =$ **S** oder $SC =$ **A**. $m \in SC$ ist ein einzelner Automat aus der Menge der parallelen Automaten SC.

Im Rahmen der Live Sequenz Chart Semantik (Abschnitt 3.3.2) ist ein Automat $m \in SC$ mit der *Instanz* i zu vergleichen. LSCs verstehen unter einer Instanz keine Automaten, sondern genau die ergänzenden Komponenten dazu, wie etwa Aktivitäten oder Objekte. Bei LSCs oder MSCs kommunizieren diese Aktivitäten oder Objekte über Nachrichten miteinander. Auf diese Weise wird der Fluss der Kommunikation abgebildet. An dieser Stelle unterscheiden sich CSC und LSC.

Sei $\mathcal{L}$ die Menge der Positionen von SC, mit $L(m)$ ist die Menge der Positionen eines einzelnen Automaten. Für den Produktautomaten SC gilt $\mathcal{L} = L(m_1) \times \ldots \times L(m_n)$.

Die Transitionsrelation des Testmodells sei als $R(SC)$ bezeichnet. Dass ein Zustandspaar Teil der Transitionsrelation ist, sei als $\langle l, \nu \rangle \, R \, \langle l', \nu' \rangle$ gekennzeichnet. Die *Transitive Hülle* R^* der Transitionsrelation enthält alle übergeordneten, indirekt erreichbaren Paare der Relation

$$\begin{aligned} \langle l, \nu \rangle \, R^* \, \langle l', \nu' \rangle \; :\Leftrightarrow \quad & \exists n \geq 1 \wedge \exists \langle l_1, \nu_1 \rangle, \ldots, \langle l_n, \nu_n \rangle \; : \\ & \langle l, \nu \rangle \; R \; \langle l_1, \nu_1 \rangle \; ; \; \ldots \; ; \; \langle l_n, \nu_n \rangle \; R \; \langle l', \nu' \rangle \end{aligned}$$

Beispiel 12. *Abbildung 4.3 zeigt zwei parallele Automaten. Sei* m_1 *der als „A“ und* m_2 *der als „B“ bezeichnete Automat. Für die Menge der Positionen gilt* $L(m_1) = \{A.a, A.b, A.c\}$ *und* $L(m_2) = \{B.a, B.b, B.c, B.d\}$*. Über die Punkt-Notation „(*Name des Automaten*).(*Position*)“ wird eine Eindeutigkeit der Positionen innerhalb eines Testmodells erreicht. Positionen müssen nur innerhalb eines Automaten, Automaten nur innerhalb einer Hierarchieebene eindeutige Namen besitzen. Die Punkt-Notation wird, falls notwendig, auf Hierarchieebenen erweitert „(*Hierachieebene 1*).(*Hierachieebene 2*). ... .(*Position*)“. Die Punkt-Notation wird im Folgenden*

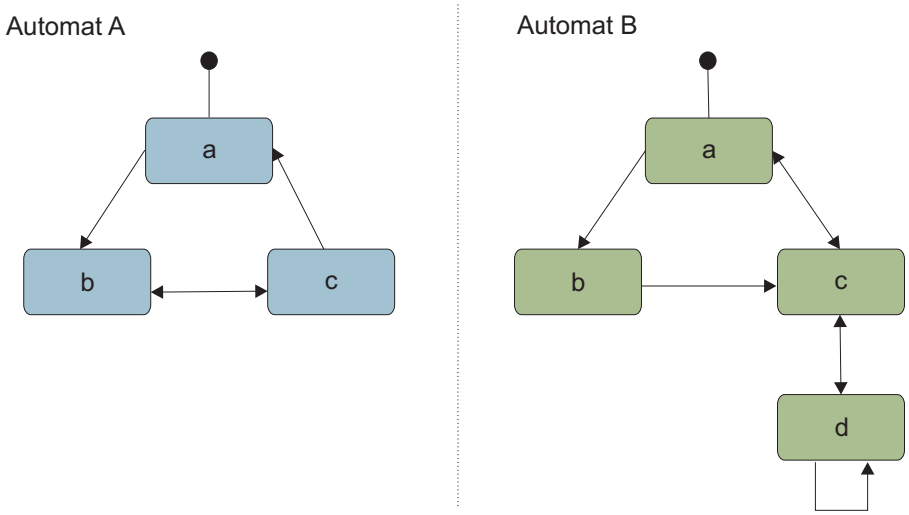


Abbildung 4.3: *Parallele Zustandsautomaten*

nur angewandt, wenn eine eindeutige Zuordnung von Positionen zu Automaten sonst nicht möglich ist.

Über die Transitionsrelation werden die Zustandsübergänge eines Automaten festgelegt. Die vereinfachte Darstellung in Abbildung 4.3 lässt nur Aktionstransitionen erkennen, deren Transitionsbedingungen ebenfalls weggelassen wurden. Eine Transitionsrelation dieser schematischen Darstellung lautet für m_1

$$R(m_1) = \{(a,b),(b,c),(c,b),(c,a)\}.$$

Die transitive Hülle dieser Relation lautet

$$R^*(m_1) = \{(a,b),(b,c),(c,b),(c,a),(a,c),(a,a),(b,a),(b,b),(c,c)\}.$$

□

Neben Automaten sind Entscheidungstabellen wichtige Elemente bei der Erstellung eines Testmodells. In einer Entscheidungstabelle ist jeder Variable, die in einem Boolschen Ausdruck eingesetzt wird, eine Input-Spalte zugeordnet. Die Output-Spalten der Entscheidungstabellen enthalten die Variablen, die als Aktion ausgeführt werden. Eine Entscheidungstabelle ist eine tabellarische Darstellung eines Automaten, dessen Transitionsbedingungen sich aus UND-verknüpften boolschen Ausdrücken zusammensetzen.

Für vorliegendes Kapitel soll genauer zwischen Zustandsautomat und Entscheidungstabelle unterschieden werden. Sei wie bisher SC die Menge der Zustandsautomaten eines Testmodells. Sei ET die Menge der Entscheidungstabellen des Testmodells. MC ist die Vereinigung aller Zustandsautomaten SC und Entscheidungstabellen ET; $(SC \cup ET) = MC$. $m \in MC$ bezeichnet einen Zustandsautomaten, $e \in MC$ eine Entscheidungstabelle. Für die Transitionsrelation des Testmodells gilt $R(SC) \cup R(ET) = R(MC)$.

Sei $T(e)$ die Menge der Transitionen einer Entscheidungstabelle und davon $\tau = \langle l, p, g, \alpha, l' \rangle$ (im Falle eines Timed Automaton $\tau = \langle l, a, \varphi, \lambda, l' \rangle$) eine Transition mit Transitionsbedingung $g \in$ Bexpr (V) als boolscher Ausdruck. Jede Transition τ ist über ihre Transitionsbedingung g und ihre Zugehörigkeit zur Entscheidungstabelle e eindeutig bestimmt. Diese eindeutige Zuordnung gilt nur für Entscheidungstabellen. Innerhalb eines Zustandsautomaten reicht allein die Angabe der Transitionsbedingung nicht. Es müssen in diesem Fall zusätzlich Quell- und Zielpositionen angegeben werden.

Beispiel 13. *Abbildung 4.4 zeigt eine Entscheidungstabelle, die in den Input-Spalten Bezug auf die Variablen* $v_i = \{x, y, z\}$ *nimmt und in den Output-Spalten auf die Variablen* $v_o = \{a, b\}$ *schreibt.*

	x	y	z	a	b
1	== 5	== true	== 10	= true	= true
2	== 5	== true	!= 10	= true	= false
3	== 5	== false		= false	= true
4	!= 5			= false	= false
5					
...					

Abbildung 4.4: *Entscheidungstabelle*

Die Tabelle sei als e bezeichnet. Die Menge der Transitionen lautet

$$T(e) = \{(x == 5 \wedge y == \textit{true} \wedge z == 10),$$
$$(x == 5 \wedge y == \textit{true} \wedge z\,! = 10),$$
$$(x == 5 \wedge y == \textit{false}),$$
$$(x\,! = 5)\},$$

wobei jede Einzeltransition über die Transitionsbedingung identifiziert wird. Die Eindeutigkeit einer Entscheidungstabellenzeile muss *über die Transitionsbedingung gegeben sein, da sonst ein Nicht-Determinismus innerhalb eines deterministischen Systems modelliert wäre. Dies würde zu einem nicht vorhersagbaren Systemverhalten führen. Wie bei Zustandsautomaten auch, müssen Variablennamen nur innerhalb einer Entscheidungstabelle eindeutig sein. Die Eindeutigkeit auf Testmodellebene wird durch die Punkt-Notation erreicht.*

□

Ein CSC bietet die Möglichkeit der Beschreibung von Transitionswechseln und Verweildauern in Positionen. Innerhalb eines CSC entspricht m genau der Repräsentation eines Automaten und den darin interessanten Positionswechseln und e einer Entscheidungstabelle und den darin interessanten Transitionswechseln.

Im Rahmen der CSC Definition ist $L(m)$ die Menge der Positionen eines Automaten m. Die Menge aller Positionen der Zustandsautomaten in SC lautet

$$L(SC) = \{(l_1, \ldots, l_n) \mid l_i \in L(m_i),\, i = 1, \ldots, n\}$$

Die Menge der Entscheidungstabellenzeilen einer Entscheidungstabelle e lautet $T(e)$. Die Menge aller Entscheidungstabellenzeilen lautet

$$T(ET) = \{(g_1, \ldots, g_m) \mid g_i \in T(e_i),\, i = 1, \ldots, n\}$$

Die Menge der Positionen und Entscheidungstabellenzeilen des Produktautomaten MC lautet

$$LT(MC) = \{(l_1, \ldots, l_n, g_1, \ldots, g_m) \mid$$
$$l_i \in L(m_i),\, i = 1, \ldots, n,\, g_j \in T(e_j),\, j = 1, \ldots, m\}$$

4.1.1 Graphische Notation eines CSCs

Die graphische Notation eines CSCs setzt sich aus den in Abbildung 4.5 dargestellten Elementen zusammen (die folgenden Abschnitte 4.2 und 4.3 definieren die einzelnen Komponenten).

Unter einem *Zeitstrahl* versteht man eine von oben nach unten zu lesende Struktur zum Anordnen von Elementen. „Von oben nach unten“ bedeutet ein zeitliches Fortschreiten beginnend bei $t_0 = 0$. Die Variable $t \in \mathbb{R}$ repräsentiert den Zeitverlauf. Ein Zeitstrahl wird einem Zustandsautomaten oder einer Entscheidungstabelle zugeordnet. Positionen, die innerhalb eines Zeitstrahls angeordnet sind, sind auch zeitlich als geordnete Folge zu interpretieren.

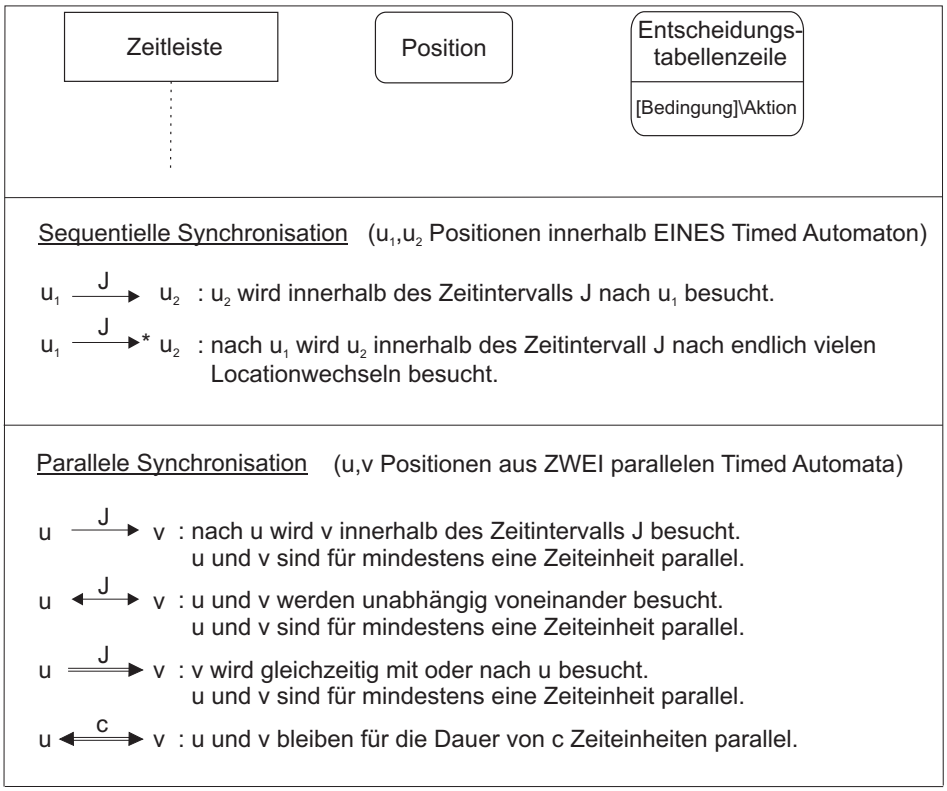


Abbildung 4.5: *Elemente der graphischen Notation*

Sei im Folgenden die graphische Notation eines Computation Sequence Chart (CSC) mit RC bezeichnet. Abbildung 4.6 zeigt ein CSC, das anhand der beiden Zustandsautomaten aus Abbildung 4.3 erstellt wurde. Ein CSC RC, wie in Abbildung 4.6 zu sehen, setzt sich zusammen aus Zeitstrahlen für Zustandsautomaten, Zeitstrahlen für Entscheidungstabellen und Relationen zwischen den Elementen der Zeitstrahle.

In der graphischen Notation des CSC wird ein Automat m eindeutig auf einen Zeitstrahl r_m für Zustandsautomaten abgebildet. Alle Elemente auf einem Zeitstrahl sind geordnet. Im Gegensatz zur Modellierung von einzelnen Automaten können Positionen in einem Zeitstrahl eines CSCs auch mehrfach vorkommen. Um sich wiederholende Positionen eines Automaten innerhalb eines Zeitstrahls unterscheiden zu können, muss eine Zeitstrahlposition mit einer Ordnungsmarkierung versehen werden. Die auf den Zeitstrahl eingetragenen Elemente werden von oben nach unten über eine natürlichzahlige Nummerierung in eine Ordnung gebracht.

Die Domäne eines einzelnen Zeitstrahls r_m innerhalb eines CSC RC lautet

$$\text{dom}\,(r_m) = \{\langle j,k\rangle \mid k \in L(m), j = 1,\ldots,n\}.$$

Im Folgenden wird anstatt $\langle j,k\rangle$ die Kurzform k_j verwendet.

Alle Zeitstrahle r_m beginnen zum Zeitpunkt 0, also implizit mit dem Eintritt in die Anfangspositionen des Automaten m zum Zeitpunkt 0. Der Eintritt in die Anfangsposition muss deshalb nicht explizit im CSC vermerkt werden.

Die Ordnung der Elemente auf einem Zeitstrahl resultiert in einer *zeitlichen* Ordnung der Elemente. Zur weiteren Interpretation der zeitlichen Ordnung werden Abbildungen benötigt, die Eintritts- und Austrittszeitpunkte in Positionen beschreiben. Sei υ : dom $(RC) \rightarrow s_E^n$ eine Abbildung der Positionen eines CSC auf Folgen nicht-negativer reeller Zahlen $s_E : \mathbb{N} \rightarrow \mathbb{R}_+$. Die Abbildung υ gibt die Zeitpunkte des *Eintretens* in eine CSC Position während einer Testmodellausführung wieder, die Abbildung ω : dom $(RC) \rightarrow s_A^n$ die Zeitpunkte des *Austritts* aus einer Position; $s_A : \mathbb{N} \rightarrow \mathbb{R}_+$.

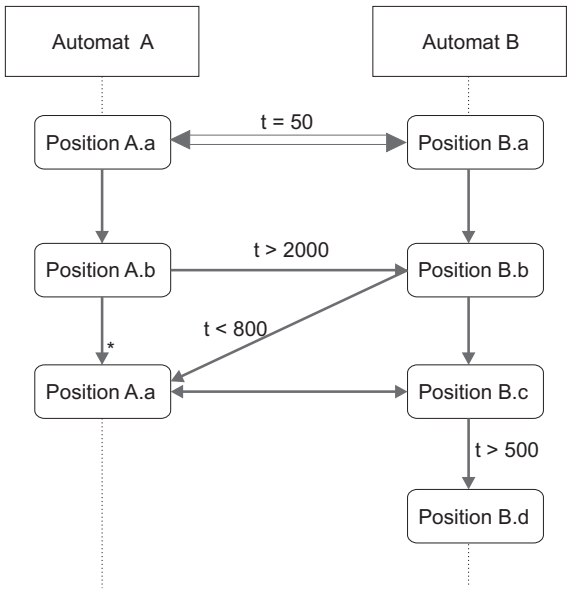


Abbildung 4.6: *Beispiel der graphischen Notation*

Die Ordnung der Positionen innerhalb eines Zeitstrahls spiegelt sich einerseits in der zeitlichen Ordnung ihrer Eintrittszeitpunkte und andererseits ihrer Austrittszeitpunkte wider.

Für CSC Positionen $k_1, \ldots, k_n \in r_m$ und deren Eintrittszeitpunkte gilt:

$$\upsilon(k_1) \leq \ldots \leq \upsilon(k_n).$$

Für die Austrittszeitpunkte von CSC Positionen gilt:

$$\omega(k_1) \leq \ldots \leq \omega(k_n).$$

Im Falle einer Aktionstransition $\langle k_j, \nu \rangle \overset{action}{\longrightarrow} \langle k_{j+1}, \nu \rangle$ im Automaten m gilt für die zugeordneten CSC Positionen

$$\omega(k_j) = \upsilon(k_{j+1}).$$

Für eine Verzögerungstransition $\langle k_j, \nu \rangle \overset{delay\ d}{\longrightarrow} \langle k_j, \nu + d \rangle$ werden immer die kumulierten Verzögerungen bis zum Austritt aus dem Zustand k_j angenommen. Eine unendliche Verweildauer $d = \infty$ in der Position k_j ist erlaubt.

$$\omega(k_j) = \upsilon(k_j) + d.$$

Die Positionen eines Zeitstrahls r_m sind somit zeitlich geordnet. Da Aktionstransitionen in Nullzeit berechnet werden (das heißt, innerhalb eines Berechnungsschrittes) und eine Verzögerungstransition nicht zwingend modelliert sein muss, können Positionswechsel innerhalb eines Zeitstrahls auch in Nullzeit passieren. Dennoch existiert eine Ordnung aufgrund der Transitionsrelation der Positionen innerhalb eines Automaten. Anders ist dies zwischen zwei nebenläufigen Automaten. Eine Reihenfolge der Positionen muss explizit spezifiziert werden, wenn sie zur Beschreibung einer Anforderung benötigt wird.

Um eine Ordnung von Positionen innerhalb eines Zeitstrahls und zwischen unterschiedlichen Zeitstrahlen festlegen zu können, werden *Ordnungsrelationen* definiert. Die Relationen zwischen den Positionen eines oder mehrerer Zeitstrahle spiegeln die zeitliche Abhängigkeit der Eintrittszeitpunkte in Positionen wider. Positionen eines CSC können auf 6 unterschiedliche Arten zueinander in Beziehung stehen.

Innerhalb eines Zeitstrahls gelten die Ordnungsrelationen $\xrightarrow{J}_s$ und $\xrightarrow{J}_s^*$. Das kleingeschriebene s steht für *sequentiell* und kennzeichnet, dass die Relation nur innerhalb eines Zeitstrahls gültig ist.

Zwischen zwei Zeitstrahlen gelten vier weitere Relationen $\xrightarrow{J}_p$, $\xleftrightarrow{J}_p$,$\overset{J}{\Longrightarrow}_p$ und $\overset{c}{\Longleftrightarrow}_p$. Das kleingeschriebene p steht für *parallel* und kennzeichnet, dass die Relationen nur zwischen zwei Zeitstrahlen gültig sind.

$c \in \mathbb{N}$ ist eine Konstante, die bei Bedarf auf $c = \infty$ gesetzt werden kann. J ist ein Intervall natürlicher Zahlen, das die Dauer sowie den Anfangszeitpunkt der Gültigkeit der Relation angibt. J ist optional. Eine fehlende Angabe wird als $J = [0, \infty)$ interpretiert.

Ist eine Entscheidungstabellenzeile Teil der Anforderungsbeschreibung, kann dies über einen Zeitstrahl für die Entscheidungstabelle im CSC abgelegt werden. Da es im Falle der Entscheidungstabelle darauf ankommt, eine gewünschte Zeile zu überdecken, werden innerhalb eines CSC auf einem Zeitstrahl keine Positionen, sondern Transitionsbedingungen eingetragen. Eine Transitionsbedingung ist eindeutig mit einer Entscheidungstabellenzeile verknüpft. Allerdings können manche Variablenabfragen innerhalb der Transitionsbedingung Freiheitsgrade erlauben. Deshalb kann es von Bedeutung sein, über welche exakte Datenbelegung eine Transitionsbedingung zu `true` auswertet. Dies ist bei der Definition von Ordnungsrelationen für Entscheidungstabellenzeilen zu beachten.

Der Zeitstrahl einer Entscheidungstabelle sei mit r_e bezeichnet. Die Domäne eines einzelnen Zeitstrahls r_e innerhalb eines CSCs RC lautet

$$\text{dom}\,(r_e) = \{\langle j, g\rangle \mid \tau \in T(e), j = 1, \ldots, n\}.$$

Im Folgenden wird anstatt $\langle j, g\rangle$ die Kurzform g_j verwendet.

Die Ordnung der Elemente auf einem Zeitstrahl resultiert in einer *zeitlichen* Ordnung der Elemente. Zur weiteren Interpretation der zeitlichen Ordnung werden Abbildungen benötigt, die Eintritts- und Austrittszeitpunkte in die einzige Position der Entscheidungstabelle beschreiben. Sei $\upsilon : \text{dom}\,(RC) \to s_E^n$ eine Abbildung der Entscheidungstabellenzeilen eines CSC auf Folgen nicht-negativer reeller Zahlen; $s_E : \mathbb{N} \to \mathbb{R}_+$. υ gibt die Zeitpunkte des *Eintretens* in die Entscheidungstabellenposition bezogen auf eine Entscheidungstabellenzeile während einer Ausführung durch das Testmodell wieder, die Abbildung $\omega : \text{dom}\,(RC) \to s_A^n$ die Zeitpunkte des *Austritts* aus der Position; $s_A : \mathbb{N} \to \mathbb{R}_+$.

Da Aktionstransitionen augenblicklich, das heißt, ohne Zeitverlust ausgeführt werden, ist der Begriff der Verweildauer und eines Eintritts- oder Austrittszeitpunktes in einer Transition nicht sinnvoll. Dennoch gibt es während einer Ausführung Zeitpunkte, zu denen Transitionen genommen werden und eine darauf folgende Verweildauer in der einzigen Position der Entscheidungstabelle. Die Synchronisation dieser Zeitpunkte ergibt wiederum Sinn, vor allem, da sie auch mit Positionen der anderen Automaten synchronisierbar sein sollten.

4.1.2 Transformation in TCTL

Die Transformation der CSC-Relationen in TCTL-Formeln ist notwendig, um CSCs bei der Testfallgenerierung an einen Testfallgenerator übergeben zu können, der mit TCTL-Formeln Testfälle ermitteln kann. Es gibt auch Testfallgeneratoren, die auf Prädikatenlogik aufsetzen. Die Transformation in Prädikatenlogik wird eingesetzt, um die TCTL-Formeln zu beweisen.

Für die Transformation der CSC-Relationen in TCTL-Formeln bzw. Prädikatenlogik wird im Folgenden eine Position bzw. eine Entscheidungstabellenzeile als Prädikat verstanden.

Eine Position ist ein Platzhalter für atomare Propositionen, die in ihr wahr sind. Wird eine Position innerhalb einer TCTL Formel oder prädikatenlogischen Formel verwendet, ist sie dort als Prädikat zu verstehen, das über die in ihr wahren atomaren Propositionen gebildet wird.

Eine Entscheidungstabellenzeile ist eindeutig über ihre Transitionsbedingung beschrieben. Wird eine Entscheidungtabellenzeile innerhalb einer TCTL Formel oder einer prädikatenlogischen Formel verwendet, ist sie dort als Prädikat zu verstehen, das über die Transitionsbedingung gebildet wird.

4.1.3 Beweise durch Prädikatenlogik

Für den Beweis, dass die definierten TCTL-Formeln die Bedeutung einer CSC-Relation wiedergeben, werden diese in Prädikatenlogik transformiert. Dafür werden zuerst die Zeitparameter mit Hilfe zusätzlicher Uhren eliminiert. Danach wird die TCTL-Formel prädikatenlogisch aufgeschrieben. Da die Ordnungsrelationen der Zustandsautomaten und Entscheidungstabellen sich nur minimal unterscheiden, werden exemplarisch die Zustandsautomat-Relationen vorgestellt.

Die Transformation der TCTL in die CTL Formel wird durch Eliminierung der Zeitparameter erreicht (siehe Abschnitt 3.2.2) [BK08]. Dafür wird dem Timed Automaton oder Timed Statechart eine neue Uhr z hinzugefügt, die eine Uhrauswertung η erweitert zu

$$\eta\{z := d\}(x) = \begin{cases} \eta(x) & \text{, wenn } x \in C \\ d & \text{, wenn } x = z. \end{cases}$$

Folgender Satz beschreibt die Transformationen zur Eliminierung der Zeitparameter nach [BK08].

Satz 4.1.1.
Sei ein Timed Automaton $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$ *oder ein Timed Statechart* $\mathcal{B} = (V, T, L, L_0, P, G, A, \Lambda)$ *gegeben und* $\Phi\ \mathcal{U}^J\ \Psi$ *eine TCTL Formel über* X *und* AP*. Für eine Uhr* $z \notin C$ *sei*

$$\mathcal{A} \oplus z = (\Sigma, L, L_0, X \cup \{z\}, I, T) \text{ bzw. } \mathcal{B} \oplus z = (V, T \cup \{z\}, L, L_0, P, G, A, \Lambda).$$

Für jeden Zustand s *aus* $\mathcal{T}(\mathcal{A})$ *gilt*

1. $s \models_{TCTL} \exists(\Phi\ \mathcal{U}^J\ \Psi)$ *genau dann, wenn*
   $s\{z := 0\} \models_{TCTL} \exists\Big((\Phi \vee \Psi)\ \mathcal{U}\ ((z \in J) \wedge \Psi)\Big)$

2. $s \models_{TCTL} \forall(\Phi\ \mathcal{U}^J\ \Psi)$ *genau dann, wenn*
   $s\{z := 0\} \models_{TCTL} \forall\Big((\Phi \vee \Psi)\ \mathcal{U}\ ((z \in J) \wedge \Psi)\Big)$

Beweis: Siehe [BK08], Seite 707.

□

Lemma 4.1.1.
Sei ein Timed Automaton $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$ oder ein Timed Statechart $\mathcal{B} = (V, T, L, L_0, P, G, A, \Lambda)$ gegeben und $\Phi\ \mathcal{U}^J\ \Psi$ eine TCTL Formel über X und AP. Für eine Uhr $z \notin C$ sei

$$\mathcal{A} \oplus z = (\Sigma, L, L_0, X \cup \{z\}, I, T) \text{ bzw. } \mathcal{B} \oplus z = (V, T \cup \{z\}, L, L_0, P, G, A, \Lambda).$$

Für jeden Zustand s aus $\mathcal{T}(\mathcal{A})$ gilt

1. $s \models_{TCTL} \exists(\Diamond^J\ \Phi)$ *genau dann, wenn*
   $s\{z := 0\} \models_{TCTL} \exists\Big(\textit{true}\ \mathcal{U}\ ((z \in J) \wedge \Phi)\Big)$

2. $s \models_{TCTL} \forall(\Diamond^J\ \Phi)$ *genau dann, wenn*
   $s\{z := 0\} \models_{TCTL} \forall\Big(\textit{true}\ \mathcal{U}\ ((z \in J) \wedge \Phi)\Big)$

Beweis: Es gelten die Äquivalenzen (siehe Abschnitt 3.1.3)

$\exists\Diamond^J\Phi \Leftrightarrow \exists\ \textit{true}\ \mathcal{U}^J\Phi$

$\forall\Diamond^J\Phi \Leftrightarrow \forall\ \textit{true}\ \mathcal{U}^J\Phi$

Dann gilt:

$$\begin{array}{rl}
 & s \models_{TCTL} \exists(\Diamond^J\ \Phi) \\
\Leftrightarrow & s \models_{TCTL} \exists(\textit{true}\ \mathcal{U}^J\ \Phi) \\
\overset{Satz 4.1.1}{\Leftrightarrow} & s\{z := 0\} \models_{TCTL} \exists\Big((\textit{true}\ \vee \Phi)\ \mathcal{U}\ ((z \in J) \wedge \Phi)\Big) \\
\Leftrightarrow & s\{z := 0\} \models_{TCTL} \exists\Big(\textit{true}\ \mathcal{U}\ ((z \in J) \wedge \Phi)\Big)
\end{array}$$

$s \models_{TCTL} \forall(\Diamond^J\ \Phi)$ entsprechend.

□

Um die TCTL-Formeln in Prädikatenlogik umzuschreiben, wurden folgende Transformationen eingesetzt:

Definition 4.1.3 (Erfüllbarkeitsrelation für TCTL, [BK08])**.**
Sei $\mathcal{A} = (\Sigma, L, L_0, X, I, T)$ ein Timed Automaton (alternativ ein Timed Statechart), $a \in AP$ eine atomare Proposition, $g \in AC(X)$ und $J \subset \mathbb{R}_{\geq 0}$ ein Zeitintervall, dessen Grenzen natürliche Zahlen sind. Für einen Zustand $s = \langle l, \nu \rangle$ im Zustandsautomaten $\mathcal{T}(\mathcal{A})$ seien Φ, Ψ TCTL Zustandsformeln und φ eine TCTL Pfadformel. Die Erfüllbarkeitsrelation $\models$ wird induktiv wie folgt definiert.

$$\begin{array}{rcl}
s \models true & \Leftrightarrow & \textit{für alle } s \\
s \models a & \Leftrightarrow & a \in I(s) \\
s \models g & \Leftrightarrow & \nu \in g \\
s \models \neg\Phi & \Leftrightarrow & \neg s \models \Phi \\
s \models \Phi \wedge \Psi & \Leftrightarrow & (s \models \Phi) \textit{ und } (s \models \Psi) \\
s \models \exists\ \varphi & \Leftrightarrow & \pi \models \varphi \textit{ für mindestens ein } \pi \in \textit{Pfade}_{div}(s) \\
s \models \forall\ \varphi & \Leftrightarrow & \pi \models \varphi \textit{ für alle } \pi \in \textit{Pfade}_{div}(s)
\end{array}$$

$$
\begin{aligned}
\pi \models \Phi \mathcal{U}^J \Psi \quad &\Leftrightarrow \quad \exists i \geq 0.\, s_i + d \models \Psi \textit{ für mindestens ein } d \in [0, d_i] \\
&\qquad \textit{mit } \sum_{k=0}^{i-1} d_k + d \in J \textit{ und} \\
&\qquad \forall j \leq i.\, s_j + d' \models \Phi \vee \Psi \textit{ für alle } d' \in [0, d_j] \\
&\qquad \textit{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{i-1} d_k + d \\
\pi \models \Diamond^J \Phi \quad &\Leftrightarrow \quad \exists i \geq 0.\, s_i + d \models \Phi \textit{ für mindestens ein } d \in [0, d_i] \\
&\qquad \textit{mit } \sum_{k=0}^{i-1} d_k + d \in J \\
\pi \models \Box^J \Phi \quad &\Leftrightarrow \quad \forall i \geq 0.\, s_i + d \models \Phi \textit{ für alle } d \in [0, d_i] \\
&\qquad \textit{mit } \sum_{k=0}^{i-1} d_k + d \in J \\
\pi \models \Box \Phi \quad &\Leftrightarrow \quad \forall i \geq 0.\, s_i \models \Phi \\
&\Leftrightarrow \quad \forall i \geq 0.\, \textit{Pfade}_{div}(s_i) \models \Phi
\end{aligned}
$$

mit $s = \langle l, \nu \rangle$ *und* $\pi \in \textit{Pfade}_{div}(s)$.

□

4.2 Sequentielle Ordnungsrelationen

Die sequentiellen Relationen aus Abbildung 4.5 und ihre Übersetzung in Temporallogik werden im Folgenden definiert. Ihre Transformation in Prädikatenlogik ist Inhalt der Sätze.

Seien k_j, k_{j+1} zwei aufeinander folgende Positionen eines Zeitstrahls r_m. Dann sind die sequentiellen Ordnungsrelationen eines CSC wie folgt definiert:

$$
\begin{aligned}
k_j \xrightarrow{J}_s k_{j+1} :\Leftrightarrow \quad & k_j, k_{j+1} \in L(m) \wedge \\
& \langle k_j, \nu \rangle\, R(m)\, \langle k_{j+1}, \nu + d \rangle \wedge \\
& \upsilon(k_j) \leq \upsilon(k_{j+1}) \wedge \\
& \upsilon(k_{j+1}) - \upsilon(k_j) \in J
\end{aligned}
$$

Die Relation $k_j \xrightarrow{J}_s k_{j+1}$ fordert, dass es einen direkten Transitionsübergang von k_j nach k_{j+1} gibt, der innerhalb der Zeitschranke J genommen werden muss.

Die Definition der Relation $k_j \xrightarrow{J}_s k_{j+1}$ in TCTL lautet wie folgt:

Definition 4.2.1.
Sei SC die Menge aller Automaten eines Testmodells. Ein einzelner Automat sei mit $m \in SC$ bezeichnet. $k, k' \in L(m)$ seien zwei Positionen. Dann definieren sich die sequentiellen Abhängigkeiten eines CSCs anhand der TCTL wie folgt:

$$
\begin{aligned}
k \xrightarrow{>c}_s k' : \Phi &= \forall\Box^{[0,\infty)} \Big(k \rightarrow \big((\exists\Box^{\leq c}\, k) \wedge \exists (k\, \mathcal{U}^{>c}\, k') \big) \Big) \\
k \xrightarrow{\geq c}_s k' : \Phi &= \forall\Box^{[0,\infty)} \Big(k \rightarrow \big((\exists\Box^{<c}\, k) \wedge \exists (k\, \mathcal{U}^{\geq c}\, k') \big) \Big) \\
k \xrightarrow{<c}_s k' : \Phi &= \forall\Box^{[0,\infty)} \Big(k \rightarrow \exists \big(k\, \mathcal{U}^{<c}\, k' \big) \Big) \\
k \xrightarrow{\leq c}_s k' : \Phi &= \forall\Box^{[0,\infty)} \Big(k \rightarrow \exists \big(k\, \mathcal{U}^{\leq c}\, k' \big) \Big) \\
k \longrightarrow_s k' : \Phi &= \forall\Box^{[0,\infty)} \Big(k \rightarrow \exists \big(k\, \mathcal{U}\, k' \big) \Big)
\end{aligned}
$$

□

Die Verwendung der Variablenzuweisung $\alpha \in A$ resultiert bei jeder Ordnungsrelation in einer zusätzlichen, durch $\wedge$ hinzugefügten Bedingung innerhalb der TCTL Formel Φ. Am Beispiel der Relation $\xrightarrow{>c}$ sieht das wie folgt aus

$$k \xrightarrow{>c;\alpha} k' : \Phi = \forall\Box^{[0,\infty)}\Big(k \rightarrow \big((\exists\Box^{\leq c}\, k) \wedge \exists(k\, \mathcal{U}^{>c}\, (k' \wedge \alpha))\big)\Big).$$

Seien g_j, g_{j+1} zwei aufeinander folgende Entscheidungstabellenzeilen eines Zeitstrahls r_e. Dann sind die sequentiellen Ordnungsrelationen wie folgt definiert:

$$\begin{aligned} g_j \xrightarrow{J}_s g_{j+1} \;:\Leftrightarrow\; & g_j, g_{j+1} \in T(e) \wedge \\ & \langle l, \nu\rangle \overset{g_{j+1}}{\Longrightarrow} \langle l, \nu + d\rangle \wedge \\ & \upsilon(g_j) \leq \upsilon(g_{j+1}) \wedge \\ & \upsilon(g_{j+1}) - \upsilon(g_j) \in J \end{aligned}$$

Die Relation fordert, dass Transition g_{j+1} nach g_j genommen wird (und dadurch die daran verknüpften Aktionen ausgeführt werden), und das innerhalb der Zeitschranke J.

Die Definition der Relation $g_j \xrightarrow{J}_s g_{j+1}$ in TCTL ist identisch zu Definition 4.2.1.

Die Beweise der TCTL-Formeln anhand der Prädikatenlogik werden im Folgenden ausgeführt.

Satz 4.2.1.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{<c}_s k'$ gilt folgende Äquivalenz:

$$\begin{aligned} s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\ & \Big(k \notin I(s_i) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i\{z := 0\}). \\ & \quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c)) \textit{ für mindestens ein } d \in [0, d_n] \\ & \quad\quad \textit{mit } \sum_{k=0}^{n-1} d_k + d \in [0, c) \textit{ und} \\ & \quad \forall i \leq j \leq n.\, (k \in I(s_j)) \vee (k' \in I(s_j)) \textit{ für alle } d' \in [0, d_j] \\ & \quad\quad \textit{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\,\big)\Big) \end{aligned}$$

Beweis:

$$\begin{aligned} \Phi \quad = \quad & \forall\Box^{[0,\infty)}\Big(k \rightarrow \exists\big(k\, \mathcal{U}^{<c}\, k'\big)\Big) \\ \Leftrightarrow \quad & \forall\Box\Big(k \rightarrow \exists\big(k\, \mathcal{U}^{<c}\, k'\big)\Big) \\ \Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. s_i + d \models k \rightarrow \exists\big(k\, \mathcal{U}^{<c}\, k'\big) \\ \Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. s_i + d \models \neg k \vee \exists\big(k \mathcal{U}^{<c}\, k'\big) \\ \Leftrightarrow \quad & \text{(Eliminierung der Zeitparameter, Satz 4.1.1)} \\ & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\ & s_i\{z := 0\} \models \neg k \vee \exists\big((k \vee k')\, \mathcal{U}\, (k' \wedge (z \in [0, c)))\big) \end{aligned}$$

$\Leftrightarrow$ Für alle $\pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0.$

$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in \mathit{Pfade}_{div}(s_i\{z := 0\}).$

$\exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c))$ für mindestens ein $d \in [0, d_n]$

mit $\underbrace{\sum_{k=0}^{n-1} d_k + d}_{z} \in [0, c)$ und

$\forall i \leq j \leq n.\, s_j + d' \models k \vee k'$ für alle $d' \in [0, d_j]$

mit $\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\,\big)\Big)$

$\Leftrightarrow$ Für alle $\pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0.$

$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in \mathit{Pfade}_{div}(s_i\{z := 0\}).$

$\exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c))$ für mindestens ein $d \in [0, d_n]$

mit $\sum_{k=0}^{n-1} d_k + d \in [0, c)$ und

$\forall i \leq j \leq n.\, (k \in I(s_j)) \vee (k' \in I(s_j))$ für alle $d' \in [0, d_j]$

mit $\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\,\big)\Big)$

□

In Worten: Für alle Pfade ab dem Initialzustand s_0 gilt, dass für alle Folgezustände s_i gilt, dass sich das System entweder nicht in Position k befindet oder es befindet sich in Position k und dann gilt für alle Teilpfade beginnend ab Zustand s_i (in dem auch Position k besucht wird), dass innerhalb des Zeitintervalls $[0, c)$ die Position von k nach k' wechselt. Das Zeitintervall wird überprüft, indem eine Hilfsuhr z auf 0 gesetzt wird, sobald die Position k besucht wird und darauf folgend k' besucht werden muss, solange die Auswertung der Uhr z noch im Zeitintervall $[0, c)$ liegt.

Beispiel 14. *Gegeben sei eine Berechnungsfolge*

$$\pi = \langle k, 0\rangle\, \langle k, 0.5\rangle\, \langle k', 0.5\rangle\, \langle k', 3\rangle\, \langle k, 0\rangle \ldots$$

$$\pi \in \underbrace{\langle k, 0\rangle}_{s_0} \overset{0.5}{\Longrightarrow} \underbrace{\langle k', 0.5\rangle}_{s_1} \overset{2.5}{\Longrightarrow} \underbrace{\langle k, 0\rangle}_{s_2} \ldots$$

Es soll gezeigt werden, dass der Pfad π die Synchronisation $k \xrightarrow{<4} k'$ erfüllt.

$\Phi \quad = \quad \forall\Box\Big(k \to \exists\big(k\, \mathcal{U}^{[0,4)}\, k'\big)\Big)$

$\Leftrightarrow$ • *Für alle* $\pi \in \mathit{Pfade}_{div}(s_0).$

▷ *es gibt im Beispiel nur den einen Pfad* π

• $\forall i \geq 0.\Big(k \notin I(s_i)$

▷ $k \in I(s_0), k \notin I(s_1), k \in I(s_2), \ldots$

• $\vee\,\big($*für mindestens ein* $\pi' \in \mathit{Pfade}_{div}(s_i).$

▷ *interessant sind* $i = 0, 2, \ldots$

- $\exists n \geq i.k' \in I(s_n + d)$ *für mindestens ein* $d \in [0, d_n]$
  - $\triangleright\ n = 1.s_1 + d \models k'$ *für ein* $d \in [0, 2.5]$
- *mit* $\sum_{k=0}^{n-1} d_k + d \in [0, 4)$ *und*
  - $\triangleright\ d_0 + d = 0.5 + [0, 2.5] = [0.5, 3] \in [0, 4)$
- $\forall i \leq j \leq n.(k \in I(s_j + d')) \vee (k' \in I(s_j + d'))$ *für alle* $d' \in [0, d_j]$
  - $\triangleright\ j = 0.s_0 + d' \models k$ *für alle* $d' \in [0, 0.5] \wedge$
  - $\triangleright\ j = 1.s_1 + d' \models k'$ *für alle* $d' \in [0, 2.5]$
- *mit* $\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\ )\Big)$
  - $\triangleright\ j = 0.[0, 0.5] \leq [0.5, 3] \wedge$
  - $\triangleright\ j = 1.[0.5, 3] \leq [0.5, 3]$

□

Satz 4.2.2.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{\leq c}_s k'$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& \Big(k \notin I(s_i) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i\{z := 0\}). \\
& \quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c]) \\
& \quad\quad \textit{für mindestens ein } d \in [0, d_n] \textit{ und} \\
& \quad \forall i \leq j \leq n.\, s_j + d' \models k \vee k' \textit{ für alle } d' \in [0, d_j] \\
& \quad\quad \textit{mit } \sum_{k=i}^{j-1} d_k + d' \leq z\ \big)\Big)
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & s_0 \models \forall \Box^{[0,\infty)} \Big(k \rightarrow \exists\big(k\ \mathcal{U}^{\leq c}\ k'\big)\Big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.s_i \models k \rightarrow \exists\big(k\ \mathcal{U}^{\leq c}\ k'\big) \\
\Leftrightarrow \quad & \text{(Eliminierung der Zeitparameter, Satz 4.1.1)} \\
& \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& s_i\{z := 0\} \models \neg k \vee \exists\big((k \vee k')\ \mathcal{U}\ (k' \wedge (z \in [0, c]))\big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& s_i\{z := 0\} \models \neg k \vee s_i\{z := 0\} \models \exists\big((k \vee k')\ \mathcal{U}\ (k' \wedge (z \in [0, c]))\big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& \Big(k \notin I(s_i\{z := 0\}) \vee \big(\text{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i\{z := 0\}). \\
& \quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c]) \text{ für mindestens ein } d \in [0, d_n] \\
& \quad\quad \text{mit } \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z} \in [0, c] \text{ und} \\
& \quad \forall i \leq j \leq n.\, s_j + d' \models k \vee k' \text{ für alle } d' \in [0, d_j] \\
& \quad\quad \text{mit } \sum_{k=i}^{j-1} d_k + d' \leq \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z}\ \big)\Big)
\end{aligned}
$$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$

$\exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c])$

für mindestens ein $d \in [0, d_n]$ und

$\forall i \leq j \leq n.\, s_j + d' \models k \vee k'$ für alle $d' \in [0, d_j]$

mit $\sum_{k=i}^{j-1} d_k + d' \leq z\ \big)\Big)$

□

Satz 4.2.3.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{>c}_s k'$ gilt folgende Äquivalenz:

$s_0 \models \Phi \quad \Leftrightarrow$ *Für alle* $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$\Big(k \notin I(s_i) \vee \big($*für mindestens ein* $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$

$\big((\forall m \geq i. s_m + d \models k \wedge (z \in [0, c])$ *für alle* $d \in [0, d_m]$

mit $\sum_{k=i}^{m-1} d_k + d \in [0, c])$ *und*

$(\exists n \geq i.\, s_n + d \models k' \wedge (z \in (c, \infty))$*für mindestens ein* $d \in [0, d_n]$

mit $\sum_{k=i}^{n-1} d_k + d \in (c, \infty)$ *und*

$\forall i \leq j \leq n.\, s_j + d' \models k \vee k'$ *für alle* $d' \in [0, d_j]$

mit $\sum_{k=i}^{j-1} d_k + d' \leq \sum_{k=i}^{n-1} d_k + d\ \big)\big)\Big)$

Beweis:

$s_0 \models \Phi \quad \Leftrightarrow \quad s_0 \models \forall\Box^{[0,\infty)}\Big(k \rightarrow \big((\exists\Box^{\leq c}\ k) \wedge \exists(k\, \mathcal{U}^{>c}\ k')\big)\Big)$

$\Leftrightarrow \quad s_0 \models \forall\Box\Big(k \rightarrow \big((\exists\Box^{[0,c]}\ k) \wedge \exists(k\, \mathcal{U}^{(c,\infty)}\ k')\big)\Big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$s_i \models k \rightarrow \big((\exists\Box^{[0,c]}\ k) \wedge \exists(k\, \mathcal{U}^{(c,\infty)}\ k')\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$s_i\{z := 0\} \models k \rightarrow \big((\exists\Box\ (k \wedge (z \in [0, c]))) \wedge$

$(\exists(k \vee k')\, \mathcal{U}\ (k' \wedge (z \in (c, \infty)))))\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$s_i\{z := 0\} \models \neg k \vee s_i\{z := 0\} \models \big((\exists\Box\ (k \wedge (z \in [0, c]))) \wedge$

$(\exists(k \vee k')\, \mathcal{U}\ (k' \wedge (z \in (c, \infty)))))\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$

$\big((\forall m \geq i.s_m + d \models k \wedge (z \in [0,c])$ für alle $d \in [0, d_m]$

mit $\underbrace{\sum_{k=i}^{m-1} d_k + d}_{z} \in [0,c])$ und

$(\exists n \geq i.\, s_n + d \models k' \wedge (z \in (c,\infty))$für mindestens ein $d \in [0, d_n]$

mit $\underbrace{\sum_{k=i}^{n-1} d_k + d}_{z} \in (c,\infty)$ und

$\forall i \leq j \leq n.\, s_j + d' \models k \vee k'$ für alle $d' \in [0, d_j]$

mit $\sum_{k=i}^{j-1} d_k + d' \leq \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z}\,)\big)\Big)$

□

Satz 4.2.4.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{\geq c}_s k'$ gilt folgende Äquivalenz:

$s_0 \models \Phi \quad \Leftrightarrow$ *Für alle* $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$\Big(k \notin I(s_i) \vee \big($*für mindestens ein* $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$

$\big((\forall m \geq i.s_m + d \models k \wedge (z \in [0,c))$ *für alle* $d \in [0, d_m]$

mit $\underbrace{\sum_{k=i}^{m-1} d_k + d}_{z} \in [0,c))$ *und*

$(\exists n \geq i.\, s_n + d \models k' \wedge (z \in [c,\infty))$*für mindestens ein* $d \in [0, d_n]$

mit $\underbrace{\sum_{k=i}^{n-1} d_k + d}_{z} \in [c,\infty)$ *und*

$\forall i \leq j \leq n.\, s_j + d' \models k \vee k'$ *für alle* $d' \in [0, d_j]$

mit $\sum_{k=i}^{j-1} d_k + d' \leq \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z}\,)\big)\Big)$

Beweis: Die Beweisführung folgt dem Beweis von Satz 4.2.3 ($k \xrightarrow{>c} k'$), allein die Intervallgrenzen ändern sich von $[0,c]$ nach $[0,c)$ und von (c,∞) nach $[c,\infty)$.

□

Satz 4.2.5.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \longrightarrow_s k'$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& \Big(k \notin I(s_i) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i). \\
& \qquad \exists n \geq i.\, s_n + d \models k' \textit{ für mindestens ein } d \in [0, d_n] \\
& \qquad\quad \textit{mit } \sum_{k=0}^{n-1} d_k + d \in [0, \infty] \textit{ und} \\
& \qquad \forall i \leq j \leq n.\, s_j + d' \models k \vee k' \textit{ für alle } d' \in [0, d_j] \\
& \qquad\quad \textit{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d \,\big)\Big)
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \forall \Box^{[0,\infty)} \Big(k \rightarrow \exists \big(k \,\mathcal{U}\, k'\big)\Big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. s_i + d \models k \rightarrow \exists \big(k \,\mathcal{U}\, k'\big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. s_i + d \models \neg k \vee \exists \big(k \,\mathcal{U}\, k'\big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& \Big(k \notin I(s_i) \vee \big(\text{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i). \\
& \qquad \exists n \geq i.\, s_n + d \models k' \text{ für mindestens ein } d \in [0, d_n] \\
& \qquad\quad \text{mit } \sum_{k=0}^{n-1} d_k + d \in [0, \infty] \text{ und} \\
& \qquad \forall i \leq j \leq n.\, s_j + d' \models k \vee k' \text{ für alle } d' \in [0, d_j] \\
& \qquad\quad \text{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d \,\big)\Big)
\end{aligned}
$$

In diesem Fall benötigt man keine zusätzliche Hilfsuhr z, da die Formel der CTL Darstellung entspricht und somit keinen expliziten Zeitbezug fordert.

□

Eine wichtige Eigenschaft des CSC kann man kurz als das *Überspringen* von Positionen bezeichnen. Die Relation der Positionen eines CSC muss nicht unbedingt allein die Transitionsrelation des Automaten m erfüllen, d.h. zwei aufeinander folgende Positionen eines CSC müssen nicht direkt über eine Transition in m verknüpft sein. Es muss allerdings möglich sein, über eine endliche Menge von Teilschritten den Positionswechsel zu realisieren. Daraus resultiert die zweite sequentielle Ordnungsrelation:

$$
\begin{aligned}
k_j \overset{J}{\longrightarrow}{}^*_s\, k_{j+1} :\Leftrightarrow \quad & k_j, k_{j+1} \in L(m) \wedge \\
& \langle k_j, \nu \rangle\, R^*(m)\, \langle k_{j+1}, \nu + d \rangle \wedge \\
& \upsilon(k_j) \leq \upsilon(k_{j+1}) \wedge \\
& \upsilon(k_{j+1}) - \upsilon(k_j) \in J
\end{aligned}
$$

Die Definition der Relation $k_j \overset{J}{\longrightarrow}{}^*_s\, k_{j+1}$ in TCTL lautet wie folgt:

Definition 4.2.2.
Sei SC die Menge aller Automaten eines Testmodells. Ein einzelner Automat sei mit $m \in SC$

bezeichnet. $k, k' \in L(m)$ *seien zwei Positionen. Dann definieren sich die sequentiellen Abhängigkeiten eines CSCs anhand der TCTL wie folgt:*

$$
\begin{aligned}
k \overset{>c}{\longrightarrow}{}^*_s\, k' : \Phi &= \forall\Box^{[0,\infty)}\Big(k \to \big((\exists\Box^{\leq c}\, \neg k') \wedge (\exists\Diamond^{>c} k')\big)\Big) \\
k \overset{\geq c}{\longrightarrow}{}^*_s\, k' : \Phi &= \forall\Box^{[0,\infty)}\Big(k \to \big((\exists\Box^{<c}\, \neg k') \wedge (\exists\Diamond^{\geq c} k')\big)\Big) \\
k \overset{<c}{\longrightarrow}{}^*_s\, k' : \Phi &= \forall\Box^{[0,\infty)}\Big(k \to \exists\Diamond^{<c}\, k'\Big) \\
k \overset{\leq c}{\longrightarrow}{}^*_s\, k' : \Phi &= \forall\Box^{[0,\infty)}\Big(k \to \exists\Diamond^{\leq c}\, k'\Big) \\
k \longrightarrow^*_s k' : \Phi &= \forall\Box^{[0,\infty)}\Big(k \to \exists\Diamond^{[0,\infty)}\, k'\Big)
\end{aligned}
$$

□

Die Verwendung der Variablenzuweisung $\alpha \in A$ resultiert bei jeder Ordnungsrelation in einer zusätzlichen, durch $\wedge$ hinzugefügten Bedingung innerhalb der TCTL Formel Φ. Am Beispiel der Relation $\overset{>c}{\longrightarrow}$ sieht das wie folgt aus

$$
g \overset{>c;\alpha}{\longrightarrow} g' : \Phi = \forall\Box^{[0,\infty)}\Big(g \to \big((\exists\Box^{\leq c}\, g) \wedge \exists(g\, \mathcal{U}^{>c}\, (g' \wedge \alpha))\big)\Big).
$$

Seien g_j, g_{j+1} zwei aufeinander folgende Entscheidungstabellenzeilen eines Zeitstrahls r_e. Dann sind die sequentiellen Ordnungsrelationen wie folgt definiert:

$$
\begin{aligned}
g_j \overset{J}{\longrightarrow}{}^*_s\, g_{j+1} \;:\Leftrightarrow\; & g_j, g_{j+1} \in T(e) \wedge \\
& \langle l, \nu\rangle\, R^*(m)\, \langle l, \nu + d\rangle \overset{g_{j+1}}{\Longrightarrow} \langle l, \nu + d'\rangle \wedge \\
& \upsilon(g_j) \leq \upsilon(g_{j+1}) \wedge \\
& \upsilon(g_{j+1}) - \upsilon(g_j) \in J
\end{aligned}
$$

Es gibt bei der *-Relation die Möglichkeit, weitere Entscheidungstabellenzeilen zwischen g_j und g_{j+1} zu besuchen.

Die Definition der Relation $g_j \overset{J}{\longrightarrow}_s g_{j+1}$ in TCTL ist identisch zu Definition 4.2.2.

Die Beweise der TCTL-Formeln anhand der Prädikatenlogik werden im Folgenden ausgeführt.

Satz 4.2.6.
Sei k *eine Position bzw. Entscheidungstabellenzeile. Für* $\Phi = k \overset{<c}{\longrightarrow}{}^*_s\, k'$ *gilt folgende Äquivalenz:*

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& \Big(k \notin I(s_i) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i\{z := 0\}). \\
& \quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c)) \textit{ für mindestens ein } d \in [0, d_n] \\
& \quad \textit{mit } \sum_{k=0}^{n-1} d_k + d \in [0, c)
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
\Phi \quad &= \quad \forall\Box^{[0,\infty)}\Big(k \rightarrow \exists\Diamond^{<c}\, k'\Big) \\
&\Leftrightarrow \quad \forall\Box\Big(k \rightarrow \exists\Diamond^{<c}\, k'\Big) \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0. s_i + d \models k \rightarrow \exists(\Diamond^{<c}\, k') \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0. s_i + d \models \neg k \vee \exists\big(\Diamond^{<c}\, k'\big) \\
&\Leftrightarrow \quad \text{(Eliminierung der Zeitparameter, Lemma 4.1.1)} \\
&\qquad \text{Für alle } \pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad s_i\{z := 0\} \models \neg k \vee \exists\big(\mathit{true}\ \ \mathcal{U}\ (k' \wedge (z \in [0,c)))\big) \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad \Big(k \notin I(s_i) \vee \big(\text{für mindestens ein } \pi' \in \mathit{Pfade}_{div}(s_i\{z := 0\}). \\
&\qquad\quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0,c)) \text{ für mindestens ein } d \in [0, d_n] \\
&\qquad\qquad \text{mit } \underbrace{\sum_{k=0}^{n-1} d_k + d}_{z} \in [0,c) \text{ und} \\
&\qquad\quad \forall i \leq j \leq n.\, s_j + d' \models \mathit{true} \text{ für alle } d' \in [0, d_j] \\
&\qquad\qquad \text{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\ \big)\Big) \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad \Big(k \notin I(s_i) \vee \big(\text{für mindestens ein } \pi' \in \mathit{Pfade}_{div}(s_i\{z := 0\}). \\
&\qquad\quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0,c)) \text{ für mindestens ein } d \in [0, d_n] \\
&\qquad\qquad \text{mit } \sum_{k=0}^{n-1} d_k + d \in [0,c)
\end{aligned}
$$

□

In Worten: Für alle Pfade ab dem Initialzustand s_0 gilt, dass für alle Folgezustände s_i gilt, dass sich das System entweder nicht in Position k befindet oder es befindet sich in Position k und dann gilt für alle Teilpfade beginnend ab Zustand s_i (in dem auch Position k besucht wird), dass innerhalb des Zeitintervalls $[0, c)$ die Position k' erreicht wird. Das Zeitintervall wird überprüft, indem eine Hilfsuhr z auf 0 gesetzt wird, sobald die Position k besucht wird und darauf folgend k' besucht werden muss, solange die Auswertung der Uhr z noch im Zeitintervall $[0, c)$ liegt.

Satz 4.2.7.

Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{\leq c}{}^*_s\, k'$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& \Big(k \notin I(s_i) \vee \big(\textit{für mindestens ein } \pi' \in \mathit{Pfade}_{div}(s_i\{z := 0\}). \\
& \quad \exists n \geq i.\, s_n + d \models k' \wedge (z \in [0,c]) \textit{ für mindestens ein } d \in [0, d_n] \\
& \quad\quad \textit{mit } \sum_{k=i}^{n-1} d_k + d \in [0,c]
\end{aligned}
$$

Beweis:

$s_0 \models \Phi \quad \Leftrightarrow \quad s_0 \models \forall\Box^{[0,\infty)}\Big(k \rightarrow \exists\Diamond^{\leq c}\, k'\Big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0. s_i \models k \rightarrow \exists\big($*true* $\mathcal{U}^{\leq c}\, k'\big)$

$\Leftrightarrow$ (Eliminierung der Zeitparameter, Lemma 4.1.1)
Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i\{z := 0\} \models \neg k \vee \exists\big($*true* $\mathcal{U}\ (k' \wedge (z \in [0, c])))\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i\{z := 0\} \models \neg k \vee s_i\{z := 0\} \models \exists\big($*true* $\mathcal{U}\ (k' \wedge (z \in [0, c])))\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(k \notin I(s_i\{z := 0\}) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$
$\exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c])$ für mindestens ein $d \in [0, d_n]$
mit $\underbrace{\sum_{k=i}^{n-1} d_k + d}_{z} \in [0, c]$ und
$\forall i \leq j \leq n.\, s_j + d' \models$ *true* für alle $d' \in [0, d_j]$
mit $\sum_{k=i}^{j-1} d_k + d' \leq \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z}\ \big)\Big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$
$\exists n \geq i.\, s_n + d \models k' \wedge (z \in [0, c])$ für mindestens ein $d \in [0, d_n]$
mit $\sum_{k=i}^{n-1} d_k + d \in [0, c]$

□

Satz 4.2.8.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{>c}{}^*_s\, k'$ gilt folgende Äquivalenz:

$s_0 \models \Phi \quad \Leftrightarrow$ *Für alle* $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i\{z := 0\} \models \exists i \geq 0.\, s_i + d \models \Phi$ *für mindestens ein* $d \in [0, d_i]$
mit $\sum_{k=0}^{i-1} d_k + d \in J$

Beweis:

$s_0 \models \Phi \quad \Leftrightarrow \quad s_0 \models \forall\Box^{[0,\infty)}\Big(k \rightarrow \big((\exists\Box^{\leq c}\, \neg k') \wedge (\exists\Diamond^{>c} k')\big)\Big)$

$\Leftrightarrow \quad s_0 \models \forall\Box\Big(k \rightarrow \big((\exists\Box^{\leq c}\, \neg k') \wedge (\exists\Diamond^{>c} k')\big)\Big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i \models k \rightarrow \big((\exists\Box^{[0,c]}\, \neg k') \wedge \exists($*true* $\mathcal{U}^{(c,\infty)}\, k')\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i\{z := 0\} \models k \rightarrow \big((\exists\Box\ (\neg k' \wedge (z \in [0, c]))) \wedge$
$(\exists$*true* $\mathcal{U}\ (k' \wedge (z \in (c, \infty)))))\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i\{z := 0\} \models \neg k \vee s_i\{z := 0\} \models \big((\exists \Box\ (\neg k' \wedge (z \in [0, c]))) \wedge$
$(\exists$*true* $\mathcal{U}\ (k' \wedge (z \in (c, \infty)))))\big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$
$\big((\forall m \geq i. s_m + d \models \neg k' \wedge (z \in [0, c])$ für alle $d \in [0, d_m]$
mit $\underbrace{\sum_{k=i}^{m-1} d_k + d}_{z} \in [0, c])$ und
$(\exists n \geq i.\, s_n + d \models k' \wedge (z \in (c, \infty))$für mindestens ein $d \in [0, d_n]$
mit $\underbrace{\sum_{k=i}^{n-1} d_k + d}_{z} \in (c, \infty)$ und
$\forall i \leq j \leq n.\, s_j + d' \models$ *true* für alle $d' \in [0, d_j]$
mit $\sum_{k=i}^{j-1} d_k + d' \leq \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z}\ ))\Big)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(k \notin I(s_i) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$
$\big((\forall m \geq i. s_m + d \models \neg k' \wedge (z \in [0, c])$ für alle $d \in [0, d_m]$
mit $\underbrace{\sum_{k=i}^{m-1} d_k + d}_{z} \in [0, c])$ und
$(\exists n \geq i.\, s_n + d \models k' \wedge (z \in (c, \infty))$für mindestens ein $d \in [0, d_n]$
mit $\underbrace{\sum_{k=i}^{n-1} d_k + d}_{z} \in (c, \infty)$ und
$\forall i \leq j \leq n.\, s_j + d' \models$ *true* für alle $d' \in [0, d_j]$
mit $\sum_{k=i}^{j-1} d_k + d' \leq \underbrace{\sum_{k=i}^{n-1} d_k + d}_{z}\ ))\Big)$

□

Satz 4.2.9.

Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \xrightarrow{\geq c}{}^*{}_s\ k'$ gilt folgende Äquivalenz:

$s_0 \models \Phi \quad \Leftrightarrow$ *Für alle* $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i\{z := 0\} \models \exists i \geq 0.\, s_i + d \models \Phi$ *für mindestens ein* $d \in [0, d_i]$
mit $\sum_{k=0}^{i-1} d_k + d \in J$

Beweis: Die Beweisführung folgt dem Beweis von Satz 4.2.8 ($k \xrightarrow{>c} k'$), allein die Intervallgrenzen ändern sich von $[0, c]$ nach $[0, c)$ und von (c, ∞) nach $[c, \infty)$. □

Satz 4.2.10.
Sei k eine Position bzw. Entscheidungstabellenzeile. Für $\Phi = k \longrightarrow_s^* k'$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& s_i\{z := 0\} \models \exists i \geq 0.\, s_i + d \models \Phi \textit{ für mindestens ein } d \in [0, d_i] \\
& \qquad \textit{mit } \sum_{k=0}^{i-1} d_k + d \in J
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & s_0 \models \forall\Box^{[0,\infty)}\Big(k \rightarrow \exists\Diamond^{[0,\infty)}\ k'\Big) \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& s_i\{z := 0\} \models \neg k \vee \Diamond\ k' \\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
& s_i\{z := 0\} \models \exists i \geq 0.\, s_i + d \models \Phi \text{ für mindestens ein } d \in [0, d_i] \\
& \qquad \text{mit } \sum_{k=0}^{i-1} d_k + d \in J
\end{aligned}
$$

□

4.3 Parallele Ordnungsrelationen

Die parallelen Relationen aus Abbildung 4.5 und ihre Übersetzung in Temporallogik werden im Folgenden definiert. Ihre Transformation in Prädikatenlogik ist Inhalt der Sätze.

Sei p_s eine Position auf dem Zeitstrahl r_m und q_t eine Position auf dem Zeitstrahl $r_{m'}$. Die Zeitstrahle gehören zum CSC RC, $r_m, r_{m'} \in RC$. $m, m' \in MC$ sind zwei Automaten aus dem Testmodell. Seien $e, e' \in ET$ zwei Entscheidungstabellen aus einem Testmodell. Die Zeitstrahle $r_e, r_{e'} \in RC$ gehören zum CSC RC. Sei g_s eine Entscheidungstabellenzeile auf dem Zeitstrahl r_e und h_t eine Entscheidungstabellenzeile auf dem Zeitstrahl $r_{e'}$. Dann sind die parallelen Ordnungsrelationen wie folgt definiert:

$$
\begin{aligned}
p_s \xrightarrow{J}_p q_t \ :\Leftrightarrow \quad & p_s \in L(m),\, q_t \in L(m') \wedge \\
& \upsilon(p_s) < \upsilon(q_t) \wedge \\
& \upsilon(q_t) - \upsilon(p_s) \in J \wedge \\
& \omega(p_s) \geq \upsilon(q_t)
\end{aligned}
$$

Die Relation besagt, dass Position/Entscheidungstabellenzeile p_s vor Position/Entscheidungstabellenzeile q_t besucht werden muss und dies innerhalb der durch J vorgegebenen zeitlichen Schranken. Außerdem müssen p_s und q_t mindestens für eine Zeiteinheit parallel besucht werden.

Die Definition der Relation $p_s \xrightarrow{J}_p q_t$ in TCTL lautet wie folgt:

Definition 4.3.1.
Sei SC die Menge aller Automaten eines Testmodells. Zwei unterschiedliche Automaten seien mit $m, m' \in SC$ bezeichnet. $p \in m$ und $q \in m'$ sind zwei Positionen der unterschiedlichen Automaten.

Dann definieren sich die sequentiellen Abhängigkeiten eines CSCs anhand der TCTL wie folgt:

$$
\begin{aligned}
p \xrightarrow{>c}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big((p \wedge \neg q) \rightarrow \big((\exists\Box^{\leq c}\ (p \wedge \neg q)) \wedge \exists(p\,\mathcal{U}^{>c}\ (p \wedge q))\big)\Big)\\
p \xrightarrow{\geq c}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big((p \wedge \neg q) \rightarrow \big((\exists\Box^{<c}\ (p \wedge \neg q)) \wedge \exists(p\,\mathcal{U}^{\geq c}\ (p \wedge q))\big)\Big)\\
p \xrightarrow{<c}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big((p \wedge \neg q) \rightarrow \exists\big(p\,\mathcal{U}^{<c}\ (p \wedge q)\big)\Big)\\
p \xrightarrow{\leq c}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big((p \wedge \neg q) \rightarrow \exists\big(p\,\mathcal{U}^{\leq c}\ (p \wedge q)\big)\Big)\\
p \longrightarrow_p q : \Phi &= \forall\Box^{[0,\infty)}\Big((p \wedge \neg q) \rightarrow \exists\big(p\,\mathcal{U}\ (p \wedge q)\big)\Big)
\end{aligned}
$$

□

Satz 4.3.1.
Für $\Phi = p \longrightarrow_p q$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& \Big((p \notin I(s_i) \vee q \in I(s_i)) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i).\\
& \qquad \exists n \geq i.\, s_n + d \models p \wedge q \textit{ für mindestens ein } d \in [0, d_n]\\
& \qquad\quad \textit{mit } \sum_{k=0}^{n-1} d_k + d \in [0,\infty] \textit{ und}\\
& \qquad \forall i \leq j \leq n.\, s_j + d' \models p \textit{ für alle } d' \in [0, d_j]\\
& \qquad\quad \textit{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\ \big)\Big)
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \forall\Box^{[0,\infty)}\Big((p \wedge \neg q) \rightarrow \exists\big(p\,\mathcal{U}\ (p \wedge q)\big)\Big)\\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& s_i + d \models (p \wedge \neg q) \rightarrow \exists\big(p\,\mathcal{U}\ (p \wedge q)\big)\\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& s_i + d \models \neg(p \wedge \neg q) \vee \exists\big(p\,\mathcal{U}\ (p \wedge q)\big)\\
\Leftrightarrow \quad & \text{(DeMorgansches Gesetz [Sch00a])}\\
& \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& s_i + d \models (\neg p \vee q) \vee \exists\big(p\,\mathcal{U}\ (p \wedge q)\big)\\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& \Big((p \notin I(s_i) \vee q \in I(s_i)) \vee \big(\text{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i).\\
& \qquad \exists n \geq i.\, s_n + d \models p \wedge q \text{ für mindestens ein } d \in [0, d_n]\\
& \qquad\quad \text{mit } \sum_{k=0}^{n-1} d_k + d \in [0,\infty] \text{ und}\\
& \qquad \forall i \leq j \leq n.\, s_j + d' \models p \vee (p \wedge q) \text{ für alle } d' \in [0, d_j]\\
& \qquad\quad \text{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\ \big)\Big)
\end{aligned}
$$

$\Leftrightarrow$ (Absorptionsgesetz [Sch00a])
Für alle $\pi \in \mathit{Pfade}_{div}(s_0).\forall i \geq 0.$
$\Big((p \notin I(s_i) \vee q \in I(s_i)) \vee$ (für mindestens ein $\pi' \in \mathit{Pfade}_{div}(s_i).$
$\exists n \geq i.\, s_n + d \models p \wedge q$ für mindestens ein $d \in [0, d_n]$
mit $\sum_{k=0}^{n-1} d_k + d \in [0, \infty]$ und
$\forall i \leq j \leq n.\, s_j + d' \models p$ für alle $d' \in [0, d_j]$
mit $\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\ )\Big)$

□

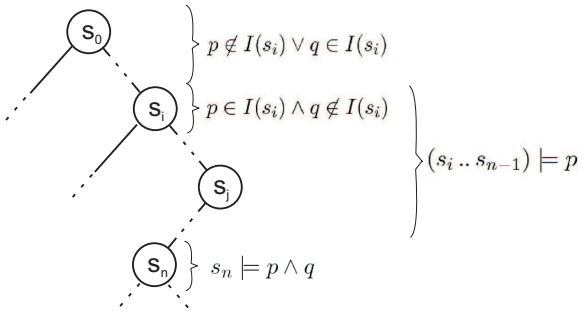


Abbildung 4.7: *Zustandswechsel zur Erfüllung der Formel* $p \longrightarrow_p q$

Abbildung 4.7 zeigt Zustandswechsel zur Erfüllung der parallelen Relation $p \longrightarrow_p q$.

Folgende Relation besagt, dass q_t gleichzeitig mit oder später als p_s eintreten kann. Bei vorgegebener zeitlicher Schranke muss diese allerdings eingehalten werden. Außerdem müssen p_s und q_t mindesten für eine Zeiteinheit parallel besucht werden.

$$p_s \overset{J}{\Longrightarrow}_p q_t \ :\Leftrightarrow \quad \begin{array}{l} p_s \in L(m),\, q_t \in L(m') \wedge \\ \upsilon(p_s) \leq \upsilon(q_t) \wedge \\ \upsilon(q_t) - \upsilon(p_s) \in J \wedge \\ \omega(p_s) \geq \upsilon(q_t) \end{array}$$

Die Relation besagt, dass h_t gleichzeitig mit oder später als g_s eintreten kann. Bei vorgegebener zeitlicher Schranke muss diese allerdings eingehalten werden. Außerdem müssen g_s und h_t mindestens für eine Zeiteinheit parallel besucht werden.

Die Definition der Relation $p_s \overset{J}{\Longrightarrow}_p q_t$ in TCTL lautet wie folgt:

Definition 4.3.2.
Sei SC die Menge aller Automaten eines Testmodells. Zwei unterschiedliche Automaten seien mit $m, m' \in SC$ bezeichnet. $p \in m$ und $q \in m'$ sind zwei Positionen der unterschiedlichen Automaten.

Dann definieren sich die sequentiellen Abhängigkeiten eines CSCs anhand der TCTL wie folgt:

$$
\begin{aligned}
p \overset{>c}{\Longrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(p \to \big((\exists\Box^{\leq c}\,(p \wedge \neg q)) \wedge \exists(p\,\mathcal{U}^{>c}\,(p \wedge q))\big)\Big)\\
p \overset{\geq c}{\Longrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(p \to \big((\exists\Box^{<c}\,(p \wedge \neg q)) \wedge \exists(p\,\mathcal{U}^{\geq c}\,(p \wedge q))\big)\Big)\\
p \overset{<c}{\Longrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(p \to \exists\big(p\,\mathcal{U}^{<c}\,(p \wedge q)\big)\Big)\\
p \overset{\leq c}{\Longrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(p \to \exists\big(p\,\mathcal{U}^{\leq c}\,(p \wedge q)\big)\Big)\\
p \Longrightarrow_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(p \to \exists\big(p\,\mathcal{U}^{[0,\infty)}\,(p \wedge q)\big)\Big)
\end{aligned}
$$

□

Satz 4.3.2.
Für $\Phi = p \Longrightarrow_p q$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& \Big(p \notin I(s_i) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i).\\
& \qquad \exists n \geq i.\, s_n + d \models p \wedge q \textit{ für mindestens ein } d \in [0, d_n]\\
& \qquad\quad \textit{mit } \sum_{k=0}^{n-1} d_k + d \in [0,\infty] \textit{ und}\\
& \qquad \forall i \leq j \leq n.\, s_j + d' \models p \textit{ für alle } d' \in [0, d_j]\\
& \qquad\quad \textit{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\,\big)\Big)
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad \Leftrightarrow \quad & \forall\Box^{[0,\infty)}\Big(p \to \exists\big(p\,\mathcal{U}^{[0,\infty)}\,(p \wedge q)\big)\Big)\\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& s_i + d \models p \to \exists\big(p\,\mathcal{U}^{[0,\infty)}\,(p \wedge q)\big)\\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& s_i + d \models \neg p \vee \exists\big(p\,\mathcal{U}^{[0,\infty)}\,(p \wedge q)\big)\\
\Leftrightarrow \quad & \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0.\\
& \Big(p \notin I(s_i) \vee \big(\text{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i).\\
& \qquad \exists n \geq i.\, s_n + d \models p \wedge q \text{ für mindestens ein } d \in [0, d_n]\\
& \qquad\quad \text{mit } \sum_{k=0}^{n-1} d_k + d \in [0,\infty] \text{ und}\\
& \qquad \forall i \leq j \leq n.\, s_j + d' \models p \vee (p \wedge q) \text{ für alle } d' \in [0, d_j]\\
& \qquad\quad \text{mit } \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\,\big)\Big)
\end{aligned}
$$

$\Leftrightarrow$ (Absorptionsgesetz [Sch00a])
Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(p \notin I(s_i) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i).$
$\exists n \geq i.\, s_n + d \models p \wedge q$ für mindestens ein $d \in [0, d_n]$
mit $\sum_{k=0}^{n-1} d_k + d \in [0, \infty]$ und
$\forall i \leq j \leq n.\, s_j + d' \models p$ für alle $d' \in [0, d_j]$
mit $\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\,\big)\Big)$

□

Folgende Relation besagt, dass Position (oder Entscheidungstabellenzeile) p_s vor q_t aber auch q_t vor p_s eintreten darf, allerdings muss die zeitliche Differenz der Eintrittszeitpunkte innerhalb der Schranke J liegen. Außerdem müssen p_s und q_t mindesten für eine Zeiteinheit parallel besucht werden.

$$p_s \overset{J}{\longleftrightarrow}_p q_t \;:\Leftrightarrow\quad p_s \overset{J}{\Longrightarrow}_p q_t \vee q_t \overset{J}{\Longrightarrow}_p p_s$$

Die Definition der Relation $p_s \overset{J}{\longleftrightarrow}_p q_t$ in TCTL lautet wie folgt:

Definition 4.3.3.
Sei SC die Menge aller Automaten eines Testmodells. Zwei unterschiedliche Automaten seien mit $m, m' \in SC$ bezeichnet. $p \in m$ und $q \in m'$ sind zwei Positionen der unterschiedlichen Automaten. Dann definieren sich die sequentiellen Abhängigkeiten eines CSCs anhand der TCTL wie folgt:

$$\begin{aligned}
p \overset{>c}{\longleftrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(\big(p \rightarrow \exists(p\,\mathcal{U}^{>c}\,(p \wedge q))\big) \vee \big(q \rightarrow \exists(q\,\mathcal{U}^{>c}\,(p \wedge q))\big)\Big)\\
p \overset{\geq c}{\longleftrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(\big(p \rightarrow \exists(p\,\mathcal{U}^{\geq c}\,(p \wedge q))\big) \vee \big(q \rightarrow \exists(q\,\mathcal{U}^{\geq c}\,(p \wedge q))\big)\Big)\\
p \overset{<c}{\longleftrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(\big(p \rightarrow \exists(p\,\mathcal{U}^{<c}\,(p \wedge q))\big) \vee \big(q \rightarrow \exists(q\,\mathcal{U}^{<c}\,(p \wedge q))\big)\Big)\\
p \overset{\leq c}{\longleftrightarrow}_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(\big(p \rightarrow \exists(p\,\mathcal{U}^{\leq c}\,(p \wedge q))\big) \vee \big(q \rightarrow \exists(q\,\mathcal{U}^{\leq c}\,(p \wedge q))\big)\Big)\\
p \longleftrightarrow_p q : \Phi &= \forall\Box^{[0,\infty)}\Big(\exists\Diamond^{[0,\infty)}\,(p \wedge q)\Big)
\end{aligned}$$

□

Satz 4.3.3.
Für $\Phi = p \longleftrightarrow_p q$ gilt folgende Äquivalenz:

$s_0 \models \Phi \quad \Leftrightarrow$ *Für alle $\pi \in$ Pfade$_{div}(s_0).\forall i \geq 0.$*
für mindestens ein $\pi' \in$ Pfade$_{div}(s_i).$
$\exists n \geq i.\, s_n + d \models p \wedge q$ *für mindestens ein* $d \in [0, d_n]$
mit $\sum_{k=0}^{n-1} d_k + d \in [0, \infty]$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad &\Leftrightarrow \quad \forall\Box^{[0,\infty)}\Big(\exists\Diamond^{[0,\infty)}\ (p \wedge q)\Big) \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad s_i + d \models \exists\Diamond^{[0,\infty)}\ (p \wedge q) \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad \text{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i). \\
&\qquad\quad \exists n \geq i.\, s_n + d \models p \wedge q \text{ für mindestens ein } d \in [0, d_n] \\
&\qquad\qquad \text{mit } \sum_{k=0}^{n-1} d_k + d \in [0, \infty]
\end{aligned}
$$

□

Folgende Relation besagt, dass Positionen (oder Entscheidungstabellenzeilen) g_s und h_t mindestens für eine Dauer von c Zeiteinheiten parallel besucht werden müssen. Darüber, wann dies zu geschehen hat, wird keine Aussage getroffen.

$$
\begin{aligned}
p_s \overset{c}{\Longleftrightarrow}_p q_t \ :\Leftrightarrow \quad & p_s \in L(m),\, q_t \in L(m') \wedge \\
& \upsilon(p_s) \leq \omega(q_t) \wedge \\
& \upsilon(q_t) \leq \omega(p_s) \wedge \\
& \omega(p_s) - \upsilon(q_t) \geq c
\end{aligned}
$$

Die Definition der Relation $p_s \overset{c}{\Longleftrightarrow}_p q_t$ in TCTL lautet wie folgt:

Definition 4.3.4.
Sei SC die Menge aller Automaten eines Testmodells. Zwei unterschiedliche Automaten seien mit $m, m' \in SC$ bezeichnet. $p \in m$ und $q \in m'$ sind zwei Positionren der unterschiedlichen Automaten. Dann definieren sich die sequentiellen Abhängigkeiten eines CSCs anhand der TCTL wie folgt:

$$
p \overset{c}{\Longleftrightarrow}_p q : \Phi \quad = \quad \forall\Box^{[0,\infty)}\Big((p \wedge q) \rightarrow \exists\Box^{\leq c}\ (p \wedge q)\Big)
$$

□

Satz 4.3.4.
Für $\Phi = p \Longleftrightarrow_p q$ gilt folgende Äquivalenz:

$$
\begin{aligned}
s_0 \models \Phi \quad &\Leftrightarrow \quad \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad \Big((p \notin I(s_i) \vee q \notin I(s_i)) \vee \big(\textit{für mindestens ein } \pi' \in \textit{Pfade}_{div}(s_i). \\
&\qquad\quad \forall n \geq i.\, s_n + d \models p \wedge q \textit{ für mindestens ein } d \in [0, d_n] \\
&\qquad\qquad \textit{mit } \sum_{k=0}^{n-1} d_k + d \in [0, \infty]\ \big)\Big)
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
s_0 \models \Phi \quad &\Leftrightarrow \quad \forall\Box^{[0,\infty)}\Big((p \wedge q) \rightarrow \exists\Box^{\leq c}\ (p \wedge q)\Big) \\
&\Leftrightarrow \quad \text{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\
&\qquad s_i + d \models (p \wedge q) \rightarrow \exists\Box^{\leq c}\ (p \wedge q)
\end{aligned}
$$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i + d \models \neg(p \wedge q) \vee \exists\Box^{\leq c}\ (p \wedge q)$

$\Leftrightarrow$ (DeMorgansches Gesetz [Sch00a])
Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$s_i + d \models (\neg p \vee \neg q) \vee \exists\Box^{\leq c}\ (p \wedge q)$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big((p \notin I(s_i) \vee q \notin I(s_i)) \vee \big($für mindestens ein $\pi' \in$ *Pfade*$_{div}(s_i).$
$\forall n \geq i.\, s_n + d \models p \wedge q$ für mindestens ein $d \in [0, d_n]$
mit $\sum_{k=0}^{n-1} d_k + d \in [0, \infty]\ \big)\Big)$

□

Die weiteren Relationen, die sich aus $\xrightarrow{J}_p$, $\xleftrightarrow{J}_p$,$\overset{J}{\Longrightarrow}_p$ und $\overset{c}{\Longleftrightarrow}_p$ ableiten, können in ähnlicher Form wie bereits vorgestellt, bewiesen werden.

Die Relationen können auch mit Bedingungen an Variablenbelegungen versehen werden. Sei V die Menge der Variablen eines Testmodells, ausgenommen der Uhrvariablen. $A = \mathbb{P} \times \text{Expr}(V)$ ist die Menge der *Aktionen*. Eine Aktion $\alpha \in A$ beschreibt Variablenzuweisungen bezüglich der Variablen aus V und der erlaubten Aussagen aus Expr(V). Diese Variablenzuweisungen können unabhängig vom Testmodell neu definiert werden und einer Ordnungsrelation zugeordnet werden. Zusätzlich zu den zeitlichen Bedingungen müssen dann auch die Bedingungen an Variablenzuweisungen in der resultierenden Position erfüllt sein. Auf diese Art kann eine gewünschte Transition erzwungen werden, falls zwischen zwei Positionen mehrere Transitionen modelliert sind. Überprüft werden kann das anhand der Zuordnungsfunktion I, die jedem Zustand die Menge der atomaren Propositionen zuordnet, die in ihm wahr sind. Am Beispiel der $\longrightarrow_p$ Relation ergibt sich folgende Definition für Positionen und Entscheidungstabellenzeilen.

$$p_s \xrightarrow{J;\,\alpha}_p q_t \ :\Leftrightarrow \qquad p_s \xrightarrow{J}_p q_t \wedge \alpha \in I(\langle q_t, \nu \rangle)$$

Die Anwendung der Variablenzuweisungen kann auf alle anderen Ordnungsrelationen entsprechend übertragen werden.

Beim Einsatz der Relationen mit zusätzlichen Bedingungen und Variablen ist darauf zu achten, dass sie nicht im Widerspruch zu der zu überdeckenden Entscheidungstabellenzeile stehen. Die Anwendung der Variablenzuweisungen kann auf alle anderen Ordnungsrelationen entsprechend übertragen werden.

Bemerkung 1. *Bei der Synchronisation paralleler Abläufe durch Automaten zeigt sich die einzige Unterscheidung der Modellierungsnotationen* Timed Automata *und* Timed Statechart *im Bereich der CSC. Genauer gesagt ist es keine Frage der Notation, sondern der zu Grunde gelegten Semantik. Auf Timed Automata wird meist eine* Interleaving Semantik *angewandt. Die Interleaving Semantik schachtelt parallele Positionen zu einer Reihenfolge. Aufgrund der Unterscheidbarkeit der Positionen existieren für n parallele Positionen n-Fakultät Möglichkeiten, die Positionen in eine Reihenfolge zu bringen. Diese Reihenfolge hat beim späteren Model-Checking eine Bedeutung. Bei fortgeschrittenen Model-Checking Methoden müssen nicht alle Möglichkeiten der Reihenfolgenbildung betrachtet werden, allerdings wird an dieser Stelle deutlich, dass bei Timed Automata innerhalb eines CSC eine Reihenfolge der parallelen Positionen beachtet werden muss. Vorgegriffen auf die späteren Ausführungen zum Einsatz temporaler Logiken bei CSCs sei gesagt, dass Temporallogiken zwischen interleaved sequences unterscheiden können. Ein zweiter daraus resultierender Aspekt betrifft die möglicherweise gewünschte Aussage, zwei Positionen sollen in demselben Berechnungsschritt eintreten. Dies kann bei einer Interleaving Semantik innerhalb eines CSC so nicht spezifiziert werden.*

Timed Statecharts liegt meist eine so genannte Step Semantik *mit echter Parallelität (true parallelism) zugrunde. Bei ihr kann von zwei parallelen Positionen innerhalb eines Berechnungsschrittes gesprochen werden. Sie verschachtelt die parallelen Positionen nicht, sondern sie werden so nach außen publiziert, als wären sie innerhalb eines steps gleichwertig ausgeführt worden (auch wenn sie innerhalb des Prozessors nacheinander berechnet werden).*

□

Sowohl die sequentiellen als auch die parallelen Ordnungsrelationen sind total, da für je zwei Elemente a, b aus der Domäne des Produktautomaten MC gilt: entweder $\upsilon(a) \leq \upsilon(b)$ oder $\upsilon(b) \leq \upsilon(a)$.

4.4 Ordnungsrelationen zwischen Zustandsautomaten und Entscheidungstabellen

Entscheidungstabellenzeilen und Positionen eines Zustandsautomaten können synchronisiert werden. Sequentielle Ordnungsrelationen sind nicht möglich, da sich ein Zeitstrahl nur entweder auf einen Zustandsautomat oder auf eine Entscheidungstabelle bezieht.

Sei $e \in ET$ eine Entscheidungstabelle und $m \in SC$ ein Zustandsautomat aus einem Testmodell. Die Zeitstrahle $r_e, r_m \in RC$ gehören zum CSC RC. Sei $g \in T(e)$ eine Entscheidungstabellenzeile auf dem Zeitstrahl r_e und $k \in L(m)$ eine Position auf dem Zeitstrahl r_m. Dann sind die parallelen Ordnungsrelationen wie folgt definiert:

$$g \xrightarrow{J}_p k \;:\Leftrightarrow\; \begin{array}{l} g \in T(e),\, k \in L(m) \wedge \\ \upsilon(g) < \upsilon(k) \wedge \\ \upsilon(k) - \upsilon(g) \in J \wedge \\ \omega(g) \geq \upsilon(k) \end{array}$$

$$g \xrightarrow{J;\alpha}_p k \;:\Leftrightarrow\; g \xrightarrow{J}_p k \wedge \alpha \in I(\langle k, \nu \rangle)$$

$$g \Longrightarrow^{J}_p k \;:\Leftrightarrow\; \begin{array}{l} g \in T(e),\, k \in L(m) \wedge \\ \upsilon(g) \leq \upsilon(k) \wedge \\ \upsilon(k) - \upsilon(g) \in J \wedge \\ \omega(g) \geq \upsilon(k) \end{array}$$

$$g \xleftrightarrow{J}_p k \;:\Leftrightarrow\; g \Longrightarrow^{J}_p k \vee k \Longrightarrow^{J}_p g$$

$$g \Longleftrightarrow^{c}_p k \;:\Leftrightarrow\; \begin{array}{l} g \in T(e),\, k \in L(m) \wedge \\ \upsilon(g) \leq \omega(k) \wedge \\ \upsilon(k) \leq \omega(g) \wedge \\ \omega(g) - \upsilon(k) \geq c \end{array}$$

Dieselben Relationen können auch von k nach g eingesetzt werden.

4.5 Komposition mehrerer CSC-Relationen

Die Aneinanderreihung mehrerer Positionen k_i innerhalb eines Zeitstrahls r_m oder alternativ mehrerer Entscheidungstabellenzeilen g_i eines Zeitstrahls r_e erfolgt über eine Verschachtelung der einzelnen TCTL-Subformeln Φ_i.

Sei $l_i \sim l_{i+1}$ die Kurzschreibweise aller sequentiellen Relationen innerhalb eines Zeitstrahls r_m oder r_e. Sei $p_i \sim q_j$ die Kurzschreibweise aller paralleler Relationen innerhalb eines Zeitstrahls r_m oder r_e. $\hat{\Phi}_i$ und $\hat{\Psi}_i$ enthalten die Subformeln der sequentiellen und parallelen Relationen. Für jede sequentielle bzw. parallele Relation lassen sich nach folgendem Schema die Subformeln identifizieren.

$$l_i \longrightarrow_s^* l_{i+1} : \forall\Box\Big(\underbrace{l_j}_{\hat{\Psi}_i} \rightarrow \underbrace{\exists\Diamond\, l_{j+1}}_{\hat{\Phi}_i}\Big)$$

$$p_i \longrightarrow_p q_j : \forall\Box\Big(\underbrace{\big(p_i \wedge \neg q_j\big)}_{\hat{\Psi}_i} \rightarrow \underbrace{\exists\big(p_i\, \mathcal{U}\, (p_i \wedge q_j)\big)}_{\hat{\Phi}_i}\Big)$$

Im Speziellen gilt für

$$p \overset{\leq c}{\longleftrightarrow}_p q : \forall\Box\Big(\big(\underbrace{p}_{\hat{\Psi}_i} \rightarrow \underbrace{\exists(p\, \mathcal{U}^{\leq c}\, (p \wedge q))}_{\hat{\Phi}_i}\big) \vee \big(\underbrace{q}_{\hat{\Psi}'_i} \rightarrow \underbrace{\exists(q\, \mathcal{U}^{\leq c}\, (p \wedge q))}_{\hat{\Phi}'_i}\big)\Big)$$

$$p \longleftrightarrow_p q : \forall\Box\Big(\underbrace{\exists\Diamond^{[0,\infty)}\, (p \wedge q)}_{\hat{\Psi}_i}\Big)$$

Die allgemeine Form einer sequentiellen Relation lautet somit:

$$l_i \sim l_{i+1}\ (p_i \sim q_j) : \forall\Box\Big(\hat{\Psi}_i \rightarrow \hat{\Phi}_i\Big),$$

wobei die Fälle

$$p \longleftrightarrow_p q : \forall\Box\Big(\hat{\Psi}_i\Big) \text{ und}$$

$$p \overset{\sim c}{\longleftrightarrow}_p q : \forall\Box\Big(\big(\hat{\Psi}_i \rightarrow \hat{\Phi}_i\big) \vee \big(\hat{\Psi}'_i \rightarrow \hat{\Phi}'_i\big)\Big) \text{ zu beachten sind.}$$

Für die sequentielle Relation $\Phi = l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$ *ohne* explizite zeitliche Einschränkung wird folgende TCTL Formel definiert:

Definition 4.5.1.
Sei SC die Menge aller Automaten eines Testmodells. Ein einzelner Automat sei mit $m \in SC$ bezeichnet. l_1, l_2 und $l_3 \in L(m)$ seien drei Positionen. Dann definiert sich die sequentielle Relation $\Phi = l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$ wie folgt:

$$\Phi = \forall\Box\Big(\underbrace{l_1 \rightarrow \big(\exists(l_1\, \mathcal{U}\, l_2)\big)}_{l_1 \longrightarrow_s l_2} \rightarrow \underbrace{\exists(l_2\, \mathcal{U}\, l_3)\big)}_{l_2 \longrightarrow_s l_3}\Big)$$

□

Abbildung 4.8 zeigt die benötigten Zustandswechsel zur Erfüllung der Formel $\Phi = l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$.

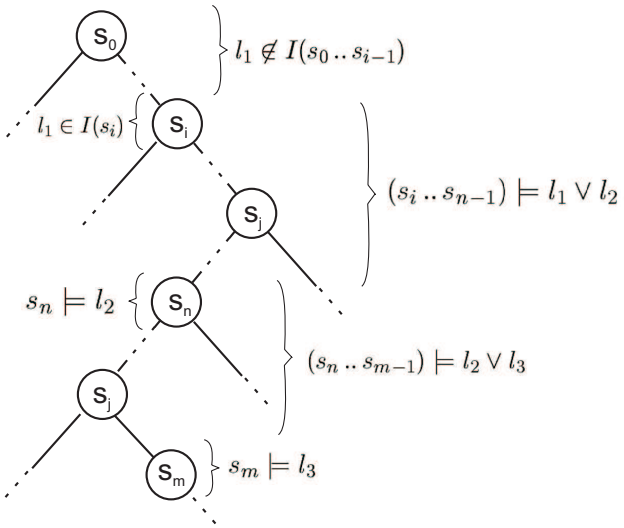


Abbildung 4.8: *Zustandswechsel zur Erfüllung der Formel* $l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$

Für die sequentielle Relation $\Phi = l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$ *ohne* explizite zeitliche Einschränkung gilt folgender Satz:

Satz 4.5.1.
Für $\Phi = l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$ *gilt folgende Äquivalenz:*

$s_0 \models \Phi \quad \Leftrightarrow$ *Für alle* $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(l_1 \notin I(s_i) \vee \big($*entweder für kein* $\pi' \in$ *Pfade*$_{div}(s_i).\exists n \geq i.$
$(s_n + d \models l_2$ *für mindestens ein* $d \in [0, d_n] \ldots$ *und*
$\forall i \leq j \leq n.\, s_j + d' \models l_1 \vee l_2$ *für alle* $d' \in [0, d_j] \ldots)$ *oder*
(*für mindestens ein* $\pi \in$ *Pfade*$_{div}(s_n).\forall m \geq n.$
$(s_m + d \models l_3$ *für mindestens ein* $d \in [0, d_m] \ldots$ *und*
$\forall n \leq j \leq m.\, s_j + d' \models l_2 \vee l_3$ *für alle* $d' \in [0, d_j] \ldots)))))\Big)$

Beweis:

$$s_0 \models \Phi \quad \Leftrightarrow \quad s_0 \models \forall\Box\Big(\underbrace{l_1 \rightarrow \big(\exists(l_1 \,\mathcal{U}\, l_2)\big)}_{l_1 \longrightarrow l_2} \rightarrow \underbrace{\exists(l_2 \,\mathcal{U}\, l_3))}_{l_2 \longrightarrow l_3}\Big)$$

$$\Leftrightarrow \quad s_0 \models \forall\Box\Big(\neg l_1 \vee \big(\exists(l_1 \,\mathcal{U}\, l_2) \longrightarrow \exists(l_2 \,\mathcal{U}\, l_3)\big)\Big)$$

$$\Leftrightarrow \quad s_0 \models \forall\Box\Big(\neg l_1 \vee \big(\neg\exists(l_1 \,\mathcal{U}\, l_2) \vee \exists(l_2 \,\mathcal{U}\, l_3)\big)\Big)$$

$\Leftrightarrow$ Für alle $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$
$\Big(l_1 \notin I(s_i) \vee \big($entweder für kein $\pi' \in$ *Pfade*$_{div}(s_i).\exists n \geq i.$
$(s_n + d \models l_2$ für mindestens ein $d \in [0, d_n] \ldots$ und
$\forall i \leq j \leq n.\, s_j + d' \models l_1 \vee l_2$ für alle $d' \in [0, d_j] \ldots)$ oder
(für mindestens ein $\pi \in$ *Pfade*$_{div}(s_n).\forall m \geq n.$
$(s_m + d \models l_3$ für mindestens ein $d \in [0, d_m] \ldots$ und
$\forall n \leq j \leq m.\, s_j + d' \models l_2 \vee l_3$ für alle $d' \in [0, d_j] \ldots)))))\Big)$

□

Allgemeiner leitet sich daraus für 3 Positionen oder Entscheidungstabellenzeilen mit den Teilformeln Φ_i und Φ_{i+1} ab:

$$\left.\begin{array}{l}\Phi_i = \forall\Box\Big(l_i \rightarrow \underbrace{\exists\big(l_i\ \mathcal{U}\ l_{i+1}\big)}_{\hat{\Phi}_i}\Big) \\ \Phi_{i+1} = \forall\Box\Big(l_{i+1} \rightarrow \underbrace{\exists\big(l_{i+1}\ \mathcal{U}\ l_{i+2}\big)}_{\hat{\Phi}_{i+1}}\Big)\end{array}\right\} \Phi = \forall\Box\Big(l_i \rightarrow \big(\hat{\Phi}_i \rightarrow \hat{\Phi}_{i+1}\big)\Big)$$

Definition 4.5.2 (TCTL-Gesamtformel eines Zeitstrahls)**.**
Sei eine zeitlich geordnete Folge von Positionen oder Entscheidungstabellenzeilen l_i mit $i = 0, \ldots, n$ eines Zeitstrahls gegeben. Sie wird durch eine geordnete Folge paarweiser TCTL-Formeln $\hat{\Phi}_0, \hat{\Phi}_1, \ldots, \hat{\Phi}_{n-1}$ abgebildet. Dann gilt für die Gesamtformel Φ des Zeitstrahls

$$\Phi = \forall\Box\Big(\hat{\Psi}_0 \rightarrow \big(\hat{\Phi}_0 \rightarrow (\hat{\Phi}_1 \rightarrow (\ldots(\hat{\Phi}_{n-1} \rightarrow \hat{\Phi}_n)\ldots))\big)\Big)$$

□

Für die sequentielle Relation $\Phi = l_1 \overset{<300}{\longrightarrow}_s l_2 \overset{\geq 500}{\longrightarrow}_s l_3$ *mit* expliziter zeitlicher Einschränkung erhält man die TCTL Formel

$$\Phi = \forall\Box\Big(\underbrace{l_1 \rightarrow \big(\exists(\ l_1\ \mathcal{U}^{<300}\ l_2\ )}_{l_1 \longrightarrow l_2} \rightarrow \underbrace{((\exists\Box^{<500}\ l_2) \wedge \exists(l_2\ \mathcal{U}^{\geq 500}\ l_3\ ))\big)}_{l_2 \longrightarrow l_3}\Big)$$

Es gilt folgender Satz:

Satz 4.5.2.
Für $\Phi = l_1 \overset{<a}{\longrightarrow}_s l_2 \overset{\geq b}{\longrightarrow}_s l_3$ gilt folgende Äquivalenz:

$$\begin{array}{lll} s_0 \models \Phi & \Leftrightarrow & \textit{Für alle } \pi \in \textit{Pfade}_{div}(s_0).\forall i \geq 0. \\ & & \Big(l_1 \notin I(s_i) \vee \big(\textit{entweder für kein } \pi' \in \textit{Pfade}_{div}(s_i\{z_1 := 0\}).\exists n \geq i. \\ & & \quad \neg(s_n + d \models l_2 \wedge (z_1 \in [0,a]) \textit{ für mindestens ein } d \in [0,d_n] \textit{ und} \\ & & \quad\quad \forall i \leq j \leq n.\, s_j + d' \models l_1 \vee l_2 \textit{ für alle } d' \in [0,d_j] \\ & & \quad\quad\quad \textit{mit } \sum_{k=i}^{j-1} d_k + d' \leq z_1) \vee \\ & & \quad (\textit{für mindestens ein } \pi'' \in \textit{Pfade}_{div}(s_n\{z_2 := 0\}).\exists m \geq n. \\ & & \quad\quad (s_m + d \models l_3 \wedge (z_2 \in [b,\infty)) \textit{ für mindestens ein } d \in [0,d_m] \textit{ und} \\ & & \quad\quad\quad \forall n \leq j \leq m.\, s_j + d' \models l_2 \vee l_3 \textit{ für alle } d' \in [0,d_j] \\ & & \quad\quad\quad\quad \textit{mit } \sum_{k=n}^{j-1} d_k + d' \leq z_2)))\Big) \end{array}$$

Beweis:

$$s_0 \models \Phi \Leftrightarrow s_0 \models \forall\Box\Big(\underbrace{l_1 \rightarrow \big(\exists(\ l_1\ \mathcal{U}^{<a}\ l_2\ )}_{l_1 \longrightarrow l_2} \rightarrow \underbrace{((\exists\Box^{<b}\ l_2) \wedge \exists(l_2\ \mathcal{U}^{\geq a}\ l_3\ ))\big)}_{l_2 \longrightarrow l_3}\Big)$$

Die Umformungen folgen dem Beweis von Satz 4.5.1.

□

Komposition zweier Zeitstrahle Die Komposition zweier und mehrerer Zeitstrahle wird über einen Algorithmus gehandhabt. Zuerst soll davon ausgegangen werden, dass eine paarweise Betrachtung der Zeitstrahle innerhalb eines CSC vorgenommen wird. Seien r und r' zwei Zeitstrahle. l_i seien die Positionen oder Entscheidungstabellenzeilen auf r, l'_j die Positionen oder Entscheidungstabellenzeilen auf r'. Im Folgenden werden Positionen und Entscheidungstabellenzeilen allgemein mit *Elemente* bezeichnet. Um eine Gesamtformel auf einem CSC mit zwei Zeitstrahlen und parallelen Relationen berechnen zu können, werden die Gesamtformeln der einzelnen Zeitstrahle nicht benötigt. Die Vorgehensweise arbeitet in diesem Fall rekursiv und verarbeitet immer zuerst eine parallele Relation, bevor die zu den Elementen gehörenden sequentiellen Relationen betrachtet werden.

Die Verknüpfung mehrerer Relationen durch den Algorithmus wird durch das Zeichen $\otimes$ gekennzeichnet.

Die Arbeitsschritte sehen wie folgt aus

1. Einlesen der ersten beiden Elemente l_0, l'_0 der Zeitstrahle r und r'
2. Falls eine oder mehrere parallele Relationen von einem Element oder beiden Elementen ausgeht, verarbeiten der parallelen Relationen zu einer Teilformel.
3. Verarbeiten der sequentiellen Relationen $l_0 \sim l_1$ und $l'_0 \sim l'_1$ zu Teilformeln.
4. Zusammenfügen der Teilformeln.
5. Einlesen der nächsten beiden Elemente (falls vorhanden) und wiederholen der Schritte 2 bis 4, bis das Ende beider Zeitstrahle erreicht ist.

Die Schritte 2 bis 4 werden als *Teilformelschritt* bezeichnet und sind in den folgenden Abbildung anhand einer gestrichelten Umrandung der beteiligten Elemente gekennzeichnet.

Verknüpfung paralleler und sequentieller Ordnungsrelationen
Die Erzeugung einer Gesamt-TCTL-Formel.

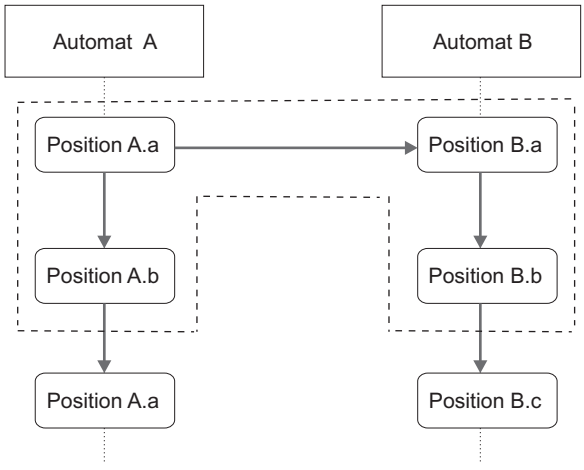


Abbildung 4.9: *CSC mit paralleler und sequentieller Relation*

Eine einzelne Betrachtung der Relationen ergibt

$$\text{“}A.a\text{“} \longrightarrow \text{“}B.a\text{“} : \forall\Box\Big(\underbrace{\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“}\big)}_{\hat{\Psi}_1} \rightarrow \underbrace{\exists\big(\text{“}A.a\text{“}\ \mathcal{U}\ (\text{“}A.a\text{“} \wedge \text{“}B.a\text{“})\big)}_{\hat{\Phi}_1}\Big)$$

$$\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“} : \forall\Box\Big(\text{“}A.a\text{“} \rightarrow \underbrace{\exists(\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“})}_{\hat{\Phi}_2}\Big)$$

$$\text{“}B.b\text{“} \longrightarrow \text{“}B.b\text{“} : \forall\Box\Big(\text{“}B.a\text{“} \rightarrow \underbrace{\exists(\text{“}B.a\text{“}\ \mathcal{U}\ \text{“}B.b\text{“})}_{\hat{\Phi}_3}\Big)$$

Das Ergebnis des Teilformelschritts für den eingerahmten Bereich aus Abbildung 4.9 sieht wie folgt aus

$$\begin{aligned}&(\text{“}A.a\text{“} \longrightarrow \text{“}B.a\text{“}) \otimes (\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“}) \otimes (\text{“}B.a\text{“} \longrightarrow \text{“}B.b\text{“}) :\\ &\quad \forall\Box\Big(\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“}\big) \rightarrow\\ &\qquad \big(\exists(\text{“}A.a\text{“}\ \mathcal{U}\ (\text{“}A.a\text{“} \wedge \text{“}B.a\text{“})) \rightarrow\\ &\qquad\quad (\exists(\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“}) \wedge \exists(\text{“}B.a\text{“}\ \mathcal{U}\ \text{“}B.b\text{“}))\big)\big)\Big)\end{aligned}$$

Allgemein haben die Einzelrelationen die Form

$$p_1 \longrightarrow q_1 : \forall\Box\Big(\underbrace{\big(p_1 \wedge \neg q_1\big)}_{\hat{\Psi}_1} \rightarrow \underbrace{\exists\big(p_1\ \mathcal{U}\ (p_1 \wedge q_1)\big)}_{\hat{\Phi}_1}\Big)$$

$$p_1 \longrightarrow p_2 : \forall\Box\Big(p_1 \rightarrow \underbrace{\exists(p_1\ \mathcal{U}\ p_2)}_{\hat{\Phi}_2}\Big)$$

$$q_1 \longrightarrow q_2 : \forall\Box\Big(q_1 \rightarrow \underbrace{\exists(q_1\ \mathcal{U}\ q_2)}_{\hat{\Phi}_3}\Big)$$

Definition 4.5.3. *Die Formel für einen* Teilformelschritt *bestehend aus einer parallelen und zwei sequentieller Positionen setzt sich wie folgt zusammen:*

$$(p_1 \longrightarrow q_1) \otimes (p_1 \longrightarrow p_2) \otimes (q_1 \longrightarrow q_2) : \forall\Box\Big(\hat{\Psi}_1 \rightarrow \big(\hat{\Phi}_1 \rightarrow (\hat{\Phi}_2 \wedge \hat{\Phi}_3)\big)\big)\Big)$$

□

Die Gesamtformel für das komplette CSC aus Abbildung 4.9 lautet

$$\begin{aligned}&(\text{“}A.a\text{“} \longrightarrow \text{“}B.a\text{“}) \otimes (\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“}) \otimes (\text{“}B.a\text{“} \longrightarrow \text{“}B.b\text{“})\\ &\qquad\qquad \otimes (\text{“}A.b\text{“} \longrightarrow \text{“}A.a\text{“}) \otimes (\text{“}B.b\text{“} \longrightarrow \text{“}B.c\text{“}) :\\ &\quad \forall\Box\Big(\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“}\big) \rightarrow\\ &\qquad \big(\exists(\text{“}A.a\text{“}\ \mathcal{U}\ (\text{“}A.a\text{“} \wedge \text{“}B.a\text{“})) \rightarrow\\ &\qquad\quad \big((\exists(\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“}) \wedge \exists(\text{“}B.a\text{“}\ \mathcal{U}\ \text{“}B.b\text{“})) \rightarrow\\ &\qquad\qquad (\exists(\text{“}A.b\text{“}\ \mathcal{U}\ \text{“}A.a\text{“}) \wedge \exists(\text{“}B.b\text{“}\ \mathcal{U}\ \text{“}B.c\text{“}))\big)\big)\Big)\end{aligned}$$

Nach einer Ergänzung der allgemeinen Einzelrelationen um

$$p_2 \longrightarrow p_3 : \forall\Box\Big(p_2 \rightarrow \underbrace{\exists(p_2\ \mathcal{U}\ p_3)}_{\hat{\Phi}_4}\Big)$$

$$q_2 \longrightarrow q_3 : \forall\Box\Big(q_2 \rightarrow \underbrace{\exists(q_2\ \mathcal{U}\ q_3)}_{\hat{\Phi}_5}\Big)$$

gilt allgemein

$$(p_1 \longrightarrow q_1) \otimes (p_1 \longrightarrow p_2) \otimes (q_1 \longrightarrow q_2) \otimes (p_2 \longrightarrow p_1) \otimes (q_2 \longrightarrow q_3) :$$
$$\forall\Box\Big(\hat{\Psi}_1 \rightarrow \big(\hat{\Phi}_1 \rightarrow \big((\hat{\Phi}_2 \wedge \hat{\Phi}_3) \rightarrow (\hat{\Phi}_4 \wedge \hat{\Phi}_5)\big)\big)\Big)$$

Ein weiteres Beispiel enthält zwei sequentielle und zwei parallele Relationen (siehe Abbildung 4.10).

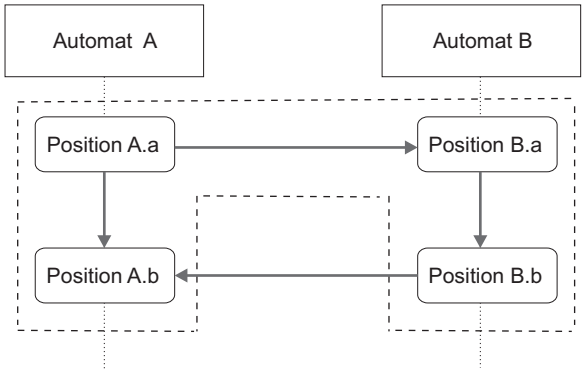


Abbildung 4.10: *Zwei sequentielle und zwei parallele Relationen*

Mit dem Vorwissen aus der Formelerstellung zu Abbildung 4.9 ergibt sich folgende allgemeine Formel für das CSC aus Abbildung 4.10. Benötigt wird dazu die zweite parallele Relation

$$q_2 \longrightarrow p_2 : \forall\Box\Big(\underbrace{\big(q_2 \wedge \neg p_2\big)}_{\hat{\Psi}_6} \rightarrow \underbrace{\exists\big(q_2 \,\mathcal{U}\, (q_2 \wedge p_2)\big)}_{\hat{\Phi}_6}\Big)$$

Definition 4.5.4. *Die Formel für einen* Teilformelschritt *bestehend aus zwei parallelen und zwei sequentiellen Positionen setzt sich wie folgt zusammen:*

$$(p_1 \longrightarrow q_1) \otimes (p_1 \longrightarrow p_2) \otimes (q_1 \longrightarrow q_2) \otimes (q_2 \longrightarrow p_2) :$$
$$\forall\Box\Big(\underbrace{\hat{\Psi}_1 \rightarrow \big(\hat{\Phi}_1 \rightarrow \big(\hat{\Phi}_2 \wedge \hat{\Phi}_3}_{\overline{\Phi}} \wedge (\hat{\Psi}_6 \rightarrow \hat{\Phi}_6)\big)\big)\Big)$$

□

Das nächste Beispiel enthält 4 sequentielle und zwei parallele Relationen (siehe Abbildung 4.11).

Dann ergibt sich als Gesamtformel für Abbildung 4.11

$$\forall\Box\Big(\underbrace{\hat{\Psi}_1 \rightarrow \big(\hat{\Phi}_1 \rightarrow \big(\hat{\Phi}_2 \wedge \hat{\Phi}_3}_{grün} \wedge \underbrace{(\hat{\Psi}_6 \rightarrow (\hat{\Phi}_6 \rightarrow (\hat{\Phi}_4 \wedge \hat{\Phi}_5))))}_{blau}\big)\Big)$$

Eine alternative Gruppierung der Formelkomponenten zeigt die Zusammensetzung der Formel aus bereits berechneten Teilformelschritten und neu hinzukommenden parallelen sowie sequentiellen Relation.

$$\forall\Box\Big(\underbrace{\hat{\Psi}_1 \rightarrow \big(\hat{\Phi}_1 \rightarrow \big(\hat{\Phi}_2 \wedge \hat{\Phi}_3}_{bestehende\,Teilformel} \wedge (\quad \underbrace{\hat{\Psi}_6 \rightarrow (\hat{\Phi}_6}_{parallele\,Relation} \quad \rightarrow (\quad \underbrace{\hat{\Phi}_4 \wedge \hat{\Phi}_5}_{sequentielle\,Relationen} \quad))))\big)\Big)$$

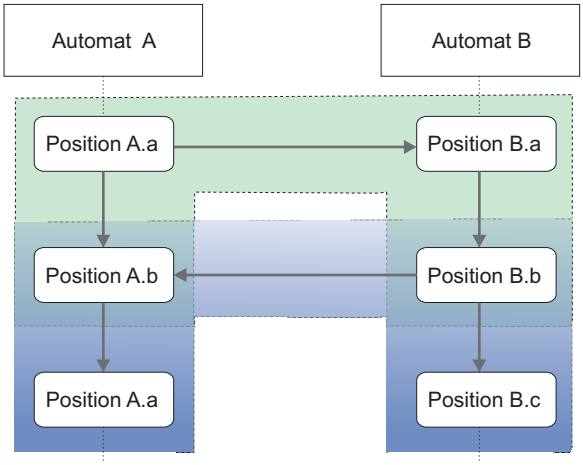


Abbildung 4.11: *Kombination zweier Teilschritte*

Mehrfachrelationen Von einem Element kann mehr als eine parallele Relationen zu unterschiedlichen Elementen ausgehen. Diese Mehrfachrelationen werden aufgelöst, indem *Pseudo-Elemente* in das CSC eingetragen werden. Diese Pseudoelemente sind Vielfache eines Elements und müssen bis zu einem sequentiellen Elementwechsel global wahr bleiben.

Folgende Abbildung 4.12 zeigt die Algorithmus-interne Auflösung von Mehrfachrelationen durch Pseudo-Elemente.

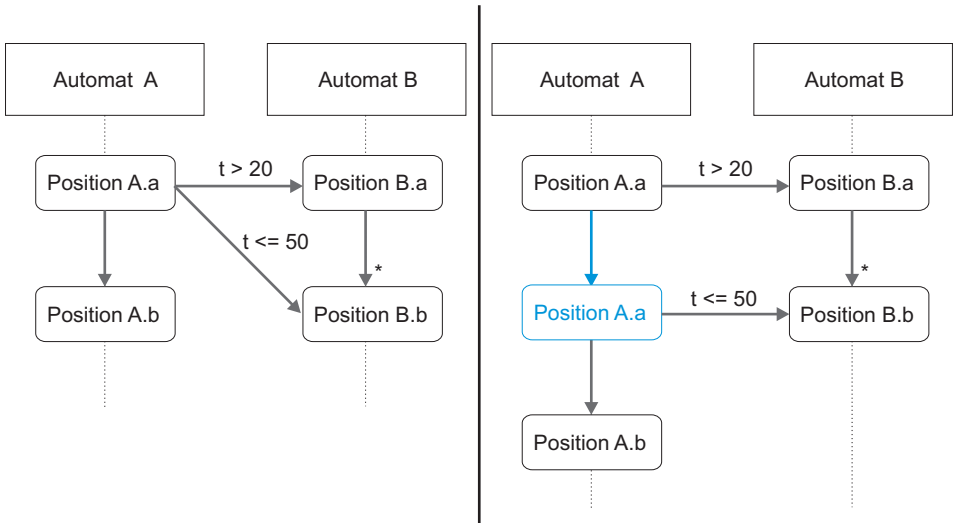


Abbildung 4.12: *CSC mit ausgehenden Mehrfachrelationen*

Die Einführung des Pseudo-Elements in Abbildung 4.12 resultiert in folgenden Teilschrittfor-

meln

$$\text{“}A.a\text{“} \xrightarrow{>20} \text{“}B.a\text{“} \quad : \quad \forall\Box\Big(\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“}\big) \rightarrow \big((\exists\Box^{\leq 20}(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“})) \wedge (\exists(\text{“}A.a\text{“}\ \mathcal{U}^{>20}(\text{“}A.a\text{“} \wedge \text{“}B.a\text{“})))\big)\Big)$$

$$\text{“}B.a\text{“} \longrightarrow^{*} \text{“}B.b\text{“} \quad : \quad \forall\Box\Big(\text{“}B.a\text{“} \rightarrow \exists\Diamond\text{“}B.b\text{“}\Big)$$

$$\text{“}A.a\text{“} \xrightarrow{\leq 50} \text{“}B.b\text{“} \quad : \quad \forall\Box\Big(\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.b\text{“}\big) \rightarrow \big(\exists(\text{“}A.a\text{“}\ \mathcal{U}^{\leq 50}(\text{“}A.a\text{“} \wedge \text{“}B.b\text{“}))\big)\Big)$$

$$\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“} \quad : \quad \forall\Box\Big(\text{“}A.a\text{“} \rightarrow (\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“})\Big)$$

Die Gesamtformel lautet

$$\begin{aligned}&(\text{“}A.a\text{“} \longrightarrow \text{“}B.a\text{“}) \otimes (\text{“}A.a\text{“} \longrightarrow \text{“}B.b\text{“}) \otimes (\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“}) \otimes (\text{“}B.a\text{“} \longrightarrow \text{“}B.b\text{“}) : \\ &\quad \forall\Box\Big(\big(((\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“}) \rightarrow \\ &\qquad ((\exists\Box^{\leq 20}(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“})) \wedge (\exists(\text{“}A.a\text{“}\ \mathcal{U}^{>20}(\text{“}A.a\text{“} \wedge \text{“}B.a\text{“}))))) \wedge \\ &\qquad ((\text{“}A.a\text{“} \wedge \neg\text{“}B.b\text{“}) \rightarrow (\exists(\text{“}A.a\text{“}\ \mathcal{U}^{\leq 50}(\text{“}A.a\text{“} \wedge \text{“}B.b\text{“}))))) \rightarrow \\ &\qquad\quad \big((\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“}) \wedge (\Diamond\text{“}B.b\text{“})\big)\Big)\end{aligned}$$

Die Mehrfachrelationen müssen nicht allein von einem Element ausgehende Relationen sein. Abbildung 4.13 zeigt eine Situation, in der sowohl eine ausgehende als auch eine eingehende Relation zur Auflösung der Mehrfachrelationen führt.

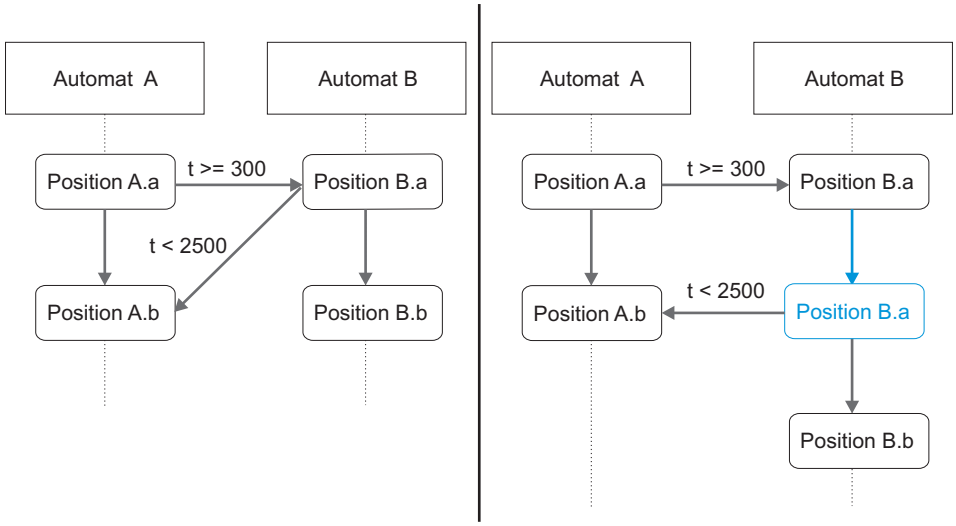


Abbildung 4.13: *CSC mit ein- und ausgehenden Mehrfachrelationen*

$$\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{s1} \rightarrow \hat{\Phi}_{s1}\Big)$$

$$\text{“}B.a\text{“} \longrightarrow \text{“}B.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{s2} \rightarrow \hat{\Phi}_{s2}\Big)$$

$$\text{“}A.a\text{“} \xrightarrow{\geq 300} \text{“}B.a\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{p1} \rightarrow \hat{\Phi}_{p1}\Big)$$

$$\text{“}B.a\text{“} \xrightarrow{<2500} \text{“}A.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{p2} \rightarrow \hat{\Phi}_{p2}\Big)$$

Definition 4.5.5. *Allgemein setzt sich eine Teilschrittformel* Φ *über* Mehrfachrelationen *wie folgt zusammen:*

$$\Phi = \forall\Box\Big(\big((\hat{\Psi}_{p1} \to \hat{\Phi}_{p1}) \wedge (\hat{\Psi}_{p2} \to \hat{\Phi}_{p2})\big) \to \big((\hat{\Phi}_{s1}) \wedge (\hat{\Phi}_{s2})\big)\Big)$$

□

In Abbildung 4.14 sind zwei Relationen über Kreuz eingetragen. Die Regel der Verundung von Mehrfachrelationen bleibt auch in diesem Fall erhalten.

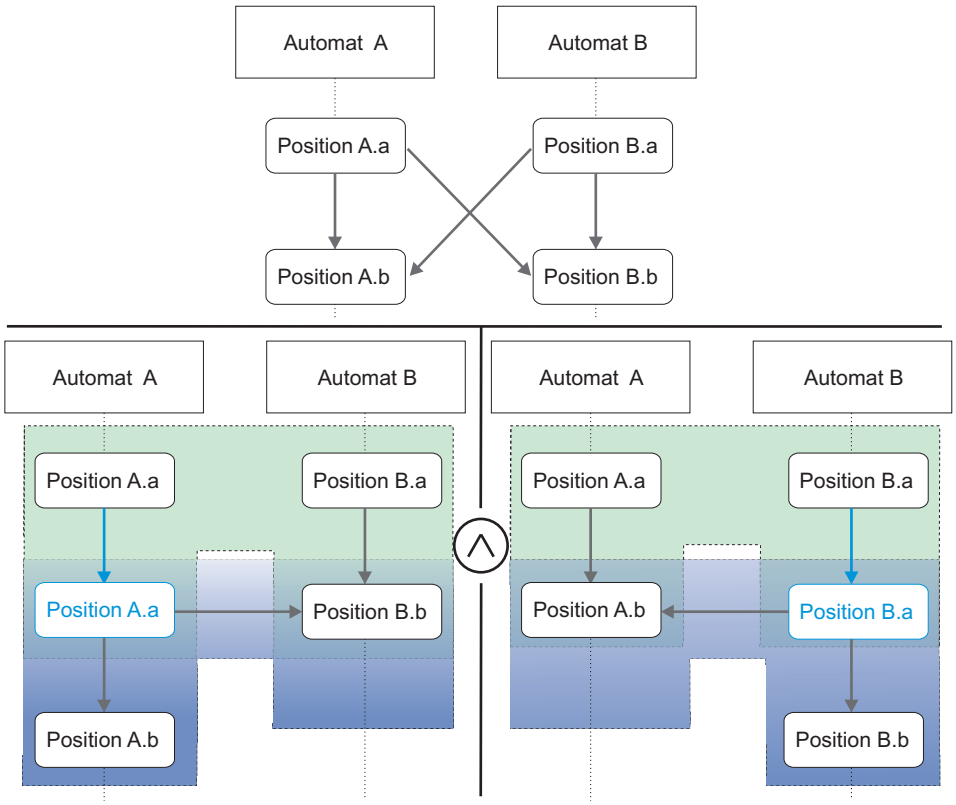


Abbildung 4.14: *CSC mit gekreuzten Mehrfachrelationen*

$$\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{s1} \to \hat{\Phi}_{s1}\Big)$$
$$\text{“}B.a\text{“} \longrightarrow \text{“}B.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{s2} \to \hat{\Phi}_{s2}\Big)$$
$$\text{“}A.a\text{“} \longrightarrow \text{“}B.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{p1} \to \hat{\Phi}_{p1}\Big)$$
$$\text{“}B.a\text{“} \longrightarrow \text{“}A.b\text{“} \quad : \quad \forall\Box\Big(\hat{\Psi}_{p2} \to \hat{\Phi}_{p2}\Big)$$

Definition 4.5.6. *Die Formel für sich* überkreuzende *Mehrfachrelationen lautet:*

$$\Phi = \forall\Box\Big(\underbrace{(\hat{\Psi}_{s1} \to \hat{\Phi}_{s1}) \wedge (\hat{\Psi}_{s2} \to \hat{\Phi}_{s2})}_{grün} \wedge \underbrace{(\hat{\Psi}_{p1} \to \hat{\Phi}_{p1}) \wedge (\hat{\Phi}_{p2} \to \hat{\Psi}_{p2})}_{blau}\Big)$$

□

Positionsübergreifende Mehrfachrelationen erweitern den Teilschritt um sequentielle Anteile. Ein Beispiel ist in Abbildung 4.15 gegeben.

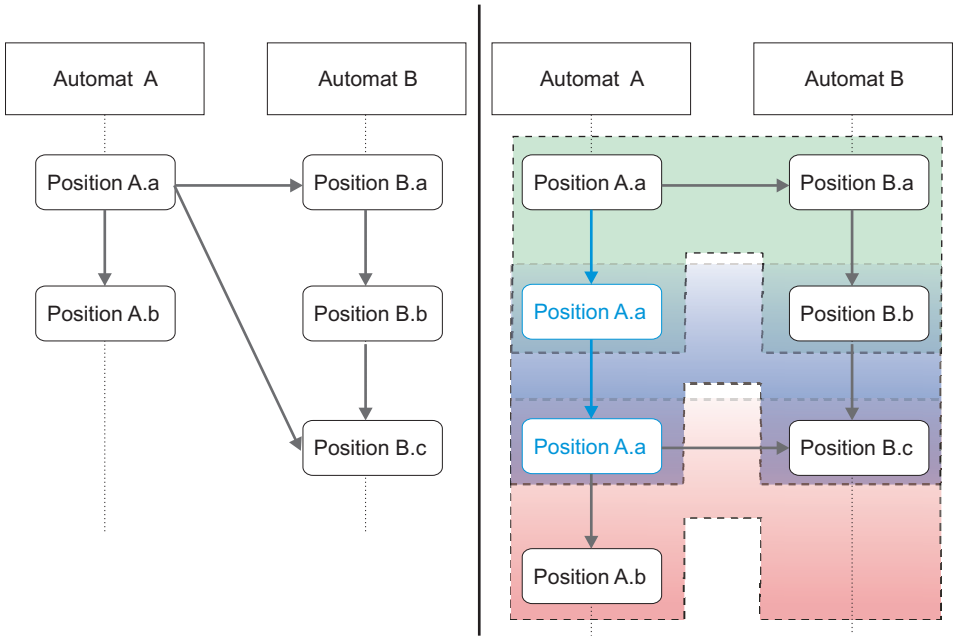


Abbildung 4.15: *CSC mit positionsübergreifenden Mehrfachrelationen*

$$\text{“}B.a\text{“} \longrightarrow \text{“}B.b\text{“} \quad : \quad \forall\Box\left(\hat{\Psi}_{s1} \rightarrow \hat{\Phi}_{s1}\right)$$
$$\text{“}B.b\text{“} \longrightarrow \text{“}B.c\text{“} \quad : \quad \forall\Box\left(\hat{\Psi}_{s2} \rightarrow \hat{\Phi}_{s2}\right)$$
$$\text{“}A.a\text{“} \longrightarrow \text{“}A.b\text{“} \quad : \quad \forall\Box\left(\hat{\Psi}_{s3} \rightarrow \hat{\Phi}_{s3}\right)$$
$$\text{“}A.a\text{“} \longrightarrow \text{“}B.a\text{“} \quad : \quad \forall\Box\left(\hat{\Psi}_{p1} \rightarrow \hat{\Phi}_{p1}\right)$$
$$\text{“}A.a\text{“} \longrightarrow \text{“}B.c\text{“} \quad : \quad \forall\Box\left(\hat{\Psi}_{p2} \rightarrow \hat{\Phi}_{p2}\right)$$

Die Formel für die *positionsübergreifende* Mehrfachrelationen in obigem Beispiel hat folgende Struktur.

$$\Phi = \forall\Box\Big(\underbrace{\hat{\Psi}_{p1} \rightarrow \big(\hat{\Phi}_{p1} \rightarrow \big(\hat{\Phi}_{s1} \wedge \hat{\Phi}_{s3}}_{grün} \wedge (\underbrace{\hat{\Phi}_{s2} \wedge \hat{\Phi}_{s3}}_{blau} \wedge (\underbrace{\hat{\Psi}_{p2} \rightarrow (\hat{\Phi}_{p2} \rightarrow (\hat{\Phi}_{s3}}_{rosa})))))\Big)$$

Komposition mehrerer Zeitstrahle Werden mehrere Zeitstrahle zur Erstellung eines CSC benötigt, geht der Algorithmus zuerst paarweise vor und kombiniert anschließend pro Teilformelschritt die paarweisen Ergebnisse. Die folgenden Beispiele mit 3 Zeitstrahlen enthalten ohne Beschränkung der Allgemeinheit die Vorgehensweise auf n Zeitstrahlen.

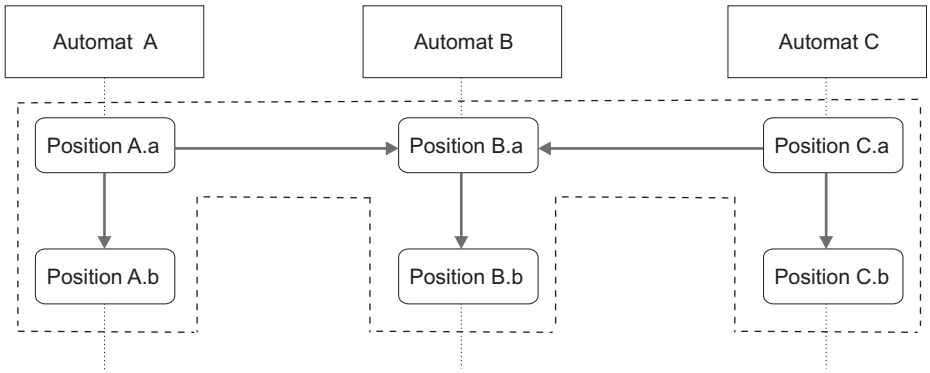


Abbildung 4.16: *CSC mit 3 Zeitstrahlen*

Die Teilschrittformeln der paarweisen Zeitlinien $A \otimes B$, $B \otimes C$ und $A \otimes C$ lauten

$$A \otimes B \quad : \forall\Box\Big(\overbrace{\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“}\big)}^{\hat{\Psi}} \rightarrow \overbrace{\big(\exists(\text{“}A.a\text{“}\ \mathcal{U}\ (\text{“}A.a\text{“} \wedge \text{“}B.a\text{“}))}^{\hat{\Phi}^p} \rightarrow \underbrace{(\exists(\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“}) \wedge \exists(\text{“}B.a\text{“}\ \mathcal{U}\ \text{“}B.b\text{“})))}_{\hat{\Phi}^s}\big)\Big)$$

$$B \otimes C \quad : \forall\Box\Big(\big(\text{“}C.a\text{“} \wedge \neg\text{“}B.a\text{“}\big) \rightarrow \big(\exists(\text{“}C.a\text{“}\ \mathcal{U}\ (\text{“}C.a\text{“} \wedge \text{“}B.a\text{“})) \rightarrow (\exists(\text{“}B.a\text{“}\ \mathcal{U}\ \text{“}B.b\text{“}) \wedge \exists(\text{“}C.a\text{“}\ \mathcal{U}\ \text{“}C.b\text{“})))\big)\Big)$$

$$A \otimes C \quad : \forall\Box\Big(\exists\big((\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“}) \wedge \exists(\text{“}C.a\text{“}\ \mathcal{U}\ \text{“}C.b\text{“})\big)\Big)$$

Definition 4.5.7. *Die Gesamtformel der ersten Teilschrittformeln dreier Zeitstrahle ergibt sich als Konjunktion der $\hat{\Psi}$, $\hat{\Phi}^p$ und $\hat{\Phi}^s$ Anteile der Teilschritte.*

$$A \otimes B \otimes C \quad : \forall\Box\Big((\hat{\Psi}_{A\otimes B} \wedge \hat{\Psi}_{B\otimes C} \wedge \hat{\Psi}_{A\otimes C}) \rightarrow \big((\hat{\Phi}^p_{A\otimes B} \wedge \hat{\Phi}^p_{B\otimes C} \wedge \hat{\Phi}^p_{A\otimes C}) \rightarrow (\hat{\Phi}^s_{A\otimes B} \wedge \hat{\Phi}^s_{B\otimes C} \wedge \hat{\Phi}^s_{A\otimes C})\big)\Big)$$

□

Für Abbildung 4.16 ergibt sich die Gesamtformel der ersten Teilschrittformeln

$$\begin{aligned}
A \otimes B \otimes C \quad & : \forall\Box\Big((\hat{\Psi}_{A\otimes B} \wedge \hat{\Psi}_{B\otimes C} \wedge \hat{\Psi}_{A\otimes C}) \rightarrow \big((\hat{\Phi}^p_{A\otimes B} \wedge \hat{\Phi}^p_{B\otimes C} \wedge \hat{\Phi}^p_{A\otimes C}) \rightarrow \\
& \qquad (\hat{\Phi}^s_{A\otimes B} \wedge \hat{\Phi}^s_{B\otimes C} \wedge \hat{\Phi}^s_{A\otimes C})\big)\Big) \\
\iff \quad & \forall\Box\Big((\hat{\Psi}_{A\otimes B} \wedge \hat{\Psi}_{B\otimes C}) \rightarrow \big((\hat{\Phi}^p_{A\otimes B} \wedge \hat{\Phi}^p_{B\otimes C}) \rightarrow \\
& \qquad (\hat{\Phi}^s_{A\otimes B} \wedge \hat{\Phi}^s_{B\otimes C} \wedge \hat{\Phi}^s_{A\otimes C})\big)\Big) \\
\iff \quad & \forall\Box\Big(\big(\text{“}A.a\text{“} \wedge \neg\text{“}B.a\text{“} \wedge \text{“}C.a\text{“}\big) \rightarrow \\
& \qquad \big((\exists(\text{“}A.a\text{“}\ \mathcal{U}\ (\text{“}A.a\text{“} \wedge \text{“}B.a\text{“})) \wedge \exists(\text{“}C.a\text{“}\ \mathcal{U}\ (\text{“}C.a\text{“} \wedge \text{“}B.a\text{“}))) \rightarrow \\
& \qquad\quad (\exists(\text{“}A.a\text{“}\ \mathcal{U}\ \text{“}A.b\text{“}) \wedge \exists(\text{“}B.a\text{“}\ \mathcal{U}\ \text{“}B.b\text{“}) \wedge \exists(\text{“}C.a\text{“}\ \mathcal{U}\ \text{“}C.b\text{“})))\big)\Big)
\end{aligned}$$

Eine Position kann sich mit mehr als einer weiteren Position synchronisieren. Dies kann durch Wechselwirkungen dazu führen, dass zeitliche Konflikte die Konsistenz der Relationen innerhalb eines CSC verhindern. Diese zeitlichen Konflikte können mithilfe eines Ungleichungssystems der zeitlichen Bedingungen festgestellt werden. Wie dies behandelt wird, ist im Abschnitt 5.2 beschrieben. Stellt sich nach der Konflikterkennung heraus, dass es keine zeitlichen Konflikte gibt, kann über die Lösbarkeit des CSC jedoch noch keine allgemeingültige Aussage getroffen werden. Die abgelegten Ordnungsrelationen innerhalb eines CSC können trotzdem noch im Widerspruch zum Testmodell und dessen Transitionsrelation stehen. Ob eine Ausführung durch das Testmodell berechnet werden kann, die ein CSC erfüllt, lässt sich erst beim Model-Checking oder der Testfallgenerierung sagen.

4.6 Abgrenzung des Verfahrens

Für die Abgrenzung eines Computation Sequence Chart von Message Sequence Charts, Live Sequence Charts und weiteren, in Abschnitt 3.3 vorgestellten Methoden werden im Folgenden ihre wichtigsten Charakteristika hervorgehoben. Die Referenzen und Verweise auf wichtige Publikation sind in Abschnitt 3.3 enthalten.

Der Unterschied zwischen Szenariobeschreibungen (MSC, Sequenzdiagramm, LSC) und Zustandsdiagrammen (Timed Statechart, Timed Automaton) liegt in der Verwendung von Zuständen, Objekten und Instanzen.

MSCs und LSCs gehören zu den *Inter-Objekt* Spezifikationen. Ein MSC beschreibt die Kommunikation von Systemobjekten über die Zeit hinweg anhand des Sendens und Empfangens von Nachrichten. Ein LSC enthält Instanzen, die über Ereignisse und Variablen definiert sind. Eine Instanz kann Nachrichten senden und empfangen und Bedingungen erfüllen. Die Motivation hinter LSCs ist es, auf Aktivitäten (zum Beispiel in Activity Charts) oder Objekte (zum Beispiel in der objektorientierten Modellierung) Bezug nehmen zu können und den Daten- und Informationsfluss abzubilden. Im Gegensatz zu LSCs können MSCs nicht abbilden, ob eine Bedingung eintreten *kann* und als weitere Möglichkeit, ob sie eintreten *muss* [DH01].

Statecharts und allgemein Zustandsautomaten zählen zu den *Intra-Objekt* Spezifikationen. Sie beschreiben ebenfalls das Verhalten eines Systems, konzentrieren sich aber darauf, das Systemverhalten vollständig zu spezifizieren, also universell, indem sie ein System als Abfolge von Systemzuständen beschreiben. Diese Art der Systemmodellierung wird meist nach der Szenariobeschreibung ausgeführt. In einem Statechart werden zusammenhängende Systemzustände, deren Wechsel anhand von Transitionen beschrieben werden kann, zu hierarchisch gegliederten Automaten zusammengefasst. Welche Systemobjekte an welchem Systemzustand beteiligt sind, wird bei der Erstellung des Testmodells nicht herausgestellt. Ein Testmodell ist funktionsorientiert und nicht objektorientiert modelliert. Das heißt, der aus der Komposition der Zustandsautomaten erzeugbare Code kann nicht eindeutig auf die später implementierte Software auf Steuergeräten zugeordnet werden. Die Funktionalität wird ohne Blick auf die Architektur modelliert, um Systemvarianten unabhängig modellieren zu können und so die Wiederverwendung zu erhöhen.

Ein CSC hat *nicht* zum Ziel, eine Systemspezifikation anhand von Szenarien durchzuführen. Die Motivation liegt in der Verlinkung von Testmodellen (Statecharts, etc.) und natürlichsprachlichen Anforderungen. Deshalb ist es nicht notwendig, Bezüge zwischen Instanzen (Aktivitäten, Objekten, etc.) herzustellen. Im vorliegenden Ansatz zielt die Ergänzung der Statecharts darauf ab, Abläufe durch das Statechart abzubilden. Die Instanzen der hier vorgestellten CSCs sind selber Statecharts und zwar genau die bereits modellierten, jetzt allerdings mit dem Blick auf die aufgefaltete Darstellung als Berechnungsbaum. In einem CSC wird eine Gruppe von Abläufen durch ein einzelnes Statechart beschrieben, wobei dabei Bezug auf andere parallel laufende Statecharts genommen werden kann. Um interessante „Knotenpunkte“ innerhalb eines Laufs spezifizieren zu können, kann Einfluss auf die Konfiguration des Modells genommen werden, auf die Variablenbelegungen und *wann* dies passieren soll. Ein CSC entspricht nur im Hinblick auf die sequentielle Sicht auf Abläufe der Inter-Objekt Modellierung, wie sie MSCs und allgemein Szenariobeschreibungen spezifizieren. MSCs und LSCs beschreiben Szenarien, die meist existentiell

aufgefasst werden, um mögliches Verhalten des Systems in frühen Designphasen festzuhalten. Natürlichsprachliche Anforderungen verhalten sich ähnlich, müssen aber auf das Testmodell und seine Elemente übertragen werden, um Rückschlüsse auf die Beziehung zwischen Testmodell und Anforderung ziehen zu können.

Visual Timed event Scenario (VTS) und Symbolic Timing Diagram sind zwei vollständig neu zu erlernende graphische Notationen, die mit den gängigen Notationen nicht viel gemein haben. Das liegt an der Motivation, mit ihnen so viele Systemeigenschaften und Verhaltensweisen wie möglich spezifizieren zu können. Beide Notationen sind mächtig genug, um nahezu alle Anforderungen zu spezifizieren. Da sie formalisiert werden können, ist Model-Checking und Testfallgenerierung mit ihnen möglich.

Ein VTS Szenario ist für den industriellen Einsatz zu aufwändig, da der Testingenieur sich bereits die graphische Notation der Zustandsautomaten aneignen muss. Die Symbolic Timing Diagrams sind intuitiver aufgebaut, sie fordern aber dennoch eine zweite Interpretation der Anforderung, die nicht mehr zustandsbasiert ist. Die Problematik bei beiden Ansätzen ist, dass die Umsetzung der Anforderungen in ein Testmodell bei der Erstellung der graphischen Repräsentation einer Einzelanforderung nicht weiterhilft.

Logik-Patterns erleichtern die Arbeit mit temporallogischen Ausdrücken und anderen Logiken. Sie bieten viel Flexibilität, da ihre Übersetzung in unterschiedliche Logiken leicht möglich ist. Logik-Patterns fordern dennoch eine Auseinandersetzung mit der Denkweise deklarativer Spezifikationen. Eine Anforderung deklarativ zu spezifizieren unterscheidet sich sehr von der operationalen Spezifikation. Logik-Pattern verbessern zwar die Lesbarkeit der logischen Formeln, der Umgang mit Logiken muss aber dennoch beherrscht werden. Zudem erzwingen sie, wie auch VTS und Symbolic Timing Diagrams eine zusätzliche Formalisierung der Anforderung, die das existierende Testmodell außen vor lässt.

Observerautomaten sind Intra-Objekt Spezifikationen. Auch wenn sie sich nicht allein aus existierenden Testmodellelementen zusammensetzen, sind sie von der Denkweise der Modellierung identisch zu der Erstellung des Testmodells. Sie sind Zustandsautomaten und erlauben somit die vollständige Syntax und Semantik der gewählten Testmodellnotation (zum Beispiel Timed Statechart oder Timed Automata). Ziel der vorliegenden Arbeit war es, für die beiden wichtigsten Modellierungsnotationen von Echtzeitsystemen eine Anforderungsbeschreibung zu finden, die von den bekanntesten Model-Checking Verfahren direkt verwendet werden können. Model-Checking auf Basis von Prädikatenlogik und CTL bzw. LTL wird von den meisten Verfahren zur Testfallgenerierung eingesetzt und war deshalb die bevorzugte Variante. Um einen Observerautomaten zur Generierung von Testfällen nutzen zu können, muss er in Prädikaten- oder Temporallogik vorliegen. Die Übersetzung eines Zustandsautomaten in eine Temporal- oder Prädikatenlogik wurde von Sowmya und Ramesh [SR98] ermöglicht, indem sie die Transformation von Statecharts in eine neue Temporallogik genannt FNLOG vorstellten. FNLOG kann von den gebräuchlichen Model-Checking Verfahren nicht direkt genutzt werden. Die Übersetzung der Observerautomaten in TCTL wurde zugunsten des neuen Ansatzes der Computation Sequence Charts nicht weiter verfolgt. Der Einsatz von Observerautomaten, die in Temporallogik transformiert werden können, stellt dennoch die beste Alternative zu den hier vorgestellten Computation Sequence Charts dar.

Die meisten vorgestellten Verfahren zur graphischen Spezifikation von Anforderungen sind in ihrem Sprachumfang und ihrer Ausdrucksmächtigkeit der zugrundegelegten Logik sehr flexibel. Manche visuelle Notationen sind intuitiv, andere schwerer zu erlernen. Ihre Vorteile zahlen sich aus, wenn zum Beispiel ein bestehendes Entwicklungsmodell verifiziert werden soll. In diesem Fall ist es sinnvoll, alle Anforderungen von dem Entwicklungsmodell losgelöst nochmals zu formalisieren und gegen das Modell zu testen. Dann ist es nicht von Bedeutung, ob sich die graphische Notation der Anforderungspezifikation von der Notation des Testmodells unterscheidet. Das Testmodell, im Falle des Beispiels das Entwicklungsmodell, wurde nicht vom Testingenieur erstellt. Ob es in Matlab Simulink oder als Timed Statecharts vorliegt, ist für den Testingenieur nicht von Interesse, da er die Anforderungen nochmal unabhängig vom Testmodell formalisieren muss.

Das Modellbasierte Testen, wie es in der vorliegenden Arbeit betrachtet wird, verwendet ein

Testmodell, dass der Testingenieur allein zum Zweck des Testens erstellt hat. Die Unabhängigkeit zum Entwicklungsmodell oder Testobjekt allgemein ist dadurch geben, dass er selber alle Anforderungen im Testmodell formalisiert hat. Müsste der Testingenieur nochmal alle Anforderung in einer neuen graphischen Spezifikation ablegen, könnte der Mehraufwand nicht mehr argumentiert werden. Deshalb ist das wichtigste Merkmal der hier vorgestellten graphischen Notation, dass sie aus bereits existieren Testmodellelementen und Pfaden durch das Testmodell aufgebaut wird.

Kapitel 5

CSC Testprozess

5.1 Anpassungen des modellbasierten Testprozesses

Der Einsatz von CSCs im modellbasierten Testprozess soll diesen nicht grundlegend ändern. Er stellt vielmehr eine Erweiterung des Prozesses an wenigen Eckpunkten dar. Abbildung 5.1 zeigt die Anpassung des modellbasierten Testprozesses aufgrund der Einführung von CSCs. Ein CSC wird bereits bei der Erstellung des Testmodells verfasst. Sobald eine einzelne Anforderung im Testmodell umgesetzt wurde, werden die Teilpfade durch das Testmodell im CSC festgehalten, die zu Testmodellerweiterungen bzw. -anpassungen geführt haben. Zur Unterstützung der Erstellung wurde im Rahmen der Dissertation eine Plug-In Lösung für ENTERPRISE ARCHITECT[1] entwickelt, die einzelne Arbeitsschritte automatisiert durchführt. Die Automatisierung erlaubt, im Testmodell Zustände oder Entscheidungstabellenzeilen auszuwählen und einem CSC zu übergeben. Existiert in einem CSC bereits ein Zeitstrahl eines Zustandsautomaten, wird der Zustand eingehängt, ansonsten wird ein neuer Zeitstrahl eingefügt. Die Relationen zwischen Elementen des CSCs müssen manuell gezogen werden, da hierbei das Wissen des Testers benötigt wird. Wird innerhalb eines Zustandsautomaten eine Position gelöscht oder umbenannt, wird dem Testingenieur dies über die Plug-In Lösung mitgeteilt. Es liegt in seiner Verantwortung, die Konsistenz von Zustandsautomat und CSCs zu garantieren. Ändern sich allein Transitionen innerhalb des Automaten, kann der Testingenieur auf die betroffenen CSCs wechseln und den Inhalt, wenn nötig, anpassen.

Liegen nach der Fertigstellung des Testmodells alle Anforderungen als CSC vor, können sie einzeln oder als Ganzes entweder als TCTL oder als prädikatenlogische Formel an den Testfallgenerator zur anforderungsbasierten Testfallgenerierung übergeben werden. Die Formalisierung des CSCs erfolgt automatisch über ein Skript. Der Algorithmus ist in Anhang A skizziert. Dem Skript vorgeschaltet sind Routinen, die grundlegende Notationsregeln und Konsistenzchecks durchführen. Sie werden im nächsten Abschnitt 5.2 betrachtet.

5.2 CSC Korrektheitsprüfung

Folgende Regeln werden definiert, um eine Formalisierung eines CSCs durchführen zu können. Die Einhaltung der Regeln wird vom Skript zur Formelerstellung geprüft.

- Positionen innerhalb eines Zeitstrahls müssen in dem zugeordneten Zustandsautomaten existieren.
- Positionen innerhalb eines Zeitstrahls müssen mit mindestens einer Position über eine der sequentiellen Beziehungen verknüpft sein.

[1] Modellierungswerkzeug der Firma SPARX SYSTEMS

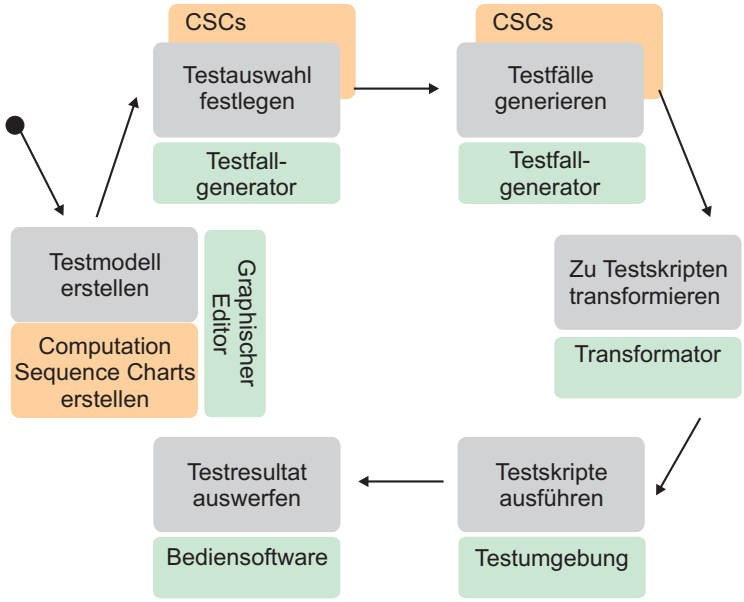


Abbildung 5.1: *Computation Sequence Charts im modellbasierten Testprozess*

- Innerhalb eines Zeitstrahls darf eine Position maximal eine eingehende und eine ausgehende sequentielle Beziehung besitzen.
- Eine Position im Zeitstrahl A darf nur dann mehrere parallele Beziehungen zu Positionen im Zeitstrahl B haben, falls die Positionen zeitlich nicht bereits eingetreten sind (keine Rückwärtsverweise).
- Zwischen zwei parallelen Position darf es nur eine Beziehung geben.
- Die Entscheidungstabellenzeilen innerhalb eines Zeitstrahls müssen in der zugeordneten Entscheidungstabelle existieren. Gleiches gilt für Positionen eines Zustandsautomaten.
- Die zeitlichen Bedingungen an Transitionen dürfen nur die Relationen $<, \leq, =, \geq, >$ enthalten und die Zeitwerte müssen natürliche Zahlen > 0 sein.

Erlaubt sind folgende Aktionen.

- Zwischen zwei Zeitstrahlen dürfen keine oder endliche viele parallele Beziehungen erstellt werden.
- Positionen eines Zeitstrahls können auf zwei weitere Zeitstrahle verweisen.

Ein CSC, welches oben beschriebene Regeln beachtet, kann auf Lösbarkeit hin untersucht werden. Die Entscheidung über die Lösbarkeit des CSCs geschieht anhand eines Ungleichungssystems der zeitlichen Kausalitäten. Jede Beziehung, sequentiell sowohl als auch parallel, führt zu einer Ungleichung, bestehend aus dem Zeitstempel des Eintritts in Position A und dem Zeitstempel des Eintritts in Position B.

Beispiel 15. *Die Funktion $timetick : L \rightarrow \mathbb{R}$ liefert den Zeitpunkt des Eintritts in die Position $l \in L$ zurück. Wandelt man die zeitlichen Kausalitäten des CSC aus Abbildung 5.2 anhand der Funktion $timetick$ in Ungleichungen, ergibt sich folgendes Ungleichungssystem.*

Zeitliche Bedingungen führen zu

$$timetick(A.b) - timetick(A.a) > 1200 \tag{5.1}$$
$$timetick(B.b) - timetick(B.a) <= 400 \tag{5.2}$$
$$timetick(A.a) - timetick(B.a) <= 660 \tag{5.3}$$

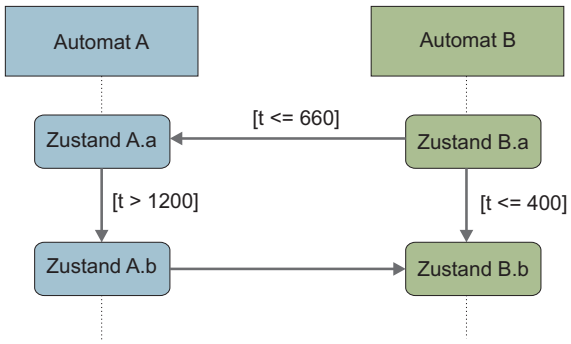


Abbildung 5.2: *CSC*

Eigenschaft sequentieller Relationen führt zu

$$timetick(A.a) - timetick(A.b) < 0 \tag{5.4}$$
$$timetick(B.a) - timetick(B.b) < 0 \tag{5.5}$$

Eigenschaft der parallelen Relationen führt zu

$$timetick(A.b) - timetick(B.b) \leq 0 \tag{5.6}$$
$$timetick(B.a) - timetick(A.a) \leq 0 \tag{5.7}$$

Eigenschaft der Dreiecksbeziehung nachfolgender sequentieller Position zu paralleler Position des Vorgängers führt zu

$$timetick(B.a) - timetick(A.b) \leq 0 \tag{5.8}$$
$$timetick(A.a) - timetick(B.b) \leq 0 \tag{5.9}$$

□

Die Ungleichungen werden automatisch erstellt. Die Lösbarkeit des Ungleichungssystems kann mithilfe der Fourier-Motzkin Elimination [BW95] entschieden werden.

An dieser Stelle sei erwähnt, dass die Lösbarkeit eines CSCs nicht auf die Möglichkeit schließen lässt, einen passenden Testfall generieren zu können. Es kann trotz theoretischer Lösbarkeit des CSCs immer noch der Fall sein, dass das Testmodell die TCTL Formel nicht erfüllt. Die Aussage über die Lösbarkeit lässt allein darauf schließen, dass es keine zeitlichen Widersprüche im CSC gibt.

5.3 CSC Testfallgenerierung

Abbildung 5.3 zeigt die Anpassungen im Vorlauf der Testfallgenerierung unter Berücksichtigung von CSCs. Um CSCs bei der Testfallgenerierung einzubinden, müssen sie zuerst in eine Logik gewandelt werden. Dies kann je nach Bedarf zum Beispiel TCTL oder Prädikatenlogik sein. Diese Formalisierung erfolgt automatisch über ein Skript. Dem Skript vorgeschaltet sind Routinen, die grundlegende Notationsregeln und Konsistenzchecks durchführen (Abschnitt 5.2).

Bei der Testfallgenerierung mit CSCs wird ein CSC eingelesen und von dem in Anhang A vorgestellten Algorithmus in eine TCTL Formel bzw. eine prädikatenlogische Formel transformiert. Vor der Formelgenerierung wird eine Korrektheitsprüfung durchgeführt (Abschnitt 5.2).

Symbolisches Model-Checking Sei Φ die TCTL Formel eines CSC RC. Die Vorgehensweise zur Testfallgenerierung unter Berücksichtigung von Φ auf *Timed Automata* ist ähnlich zur CTL Testfallgenerierung aus Abschnitt 3.2.2.

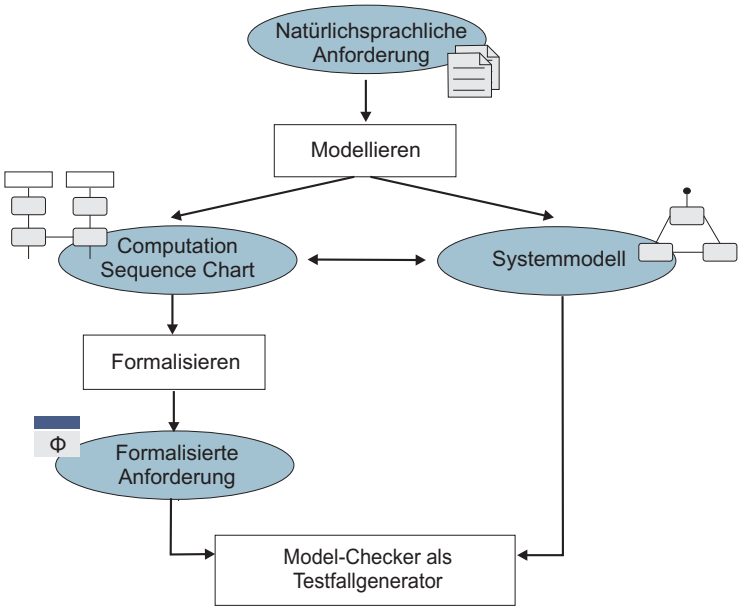


Abbildung 5.3: *Computation Sequence Charts in der Testfallgenerierung*

Algorithmus 1 zeigt die Grundzüge des TCTL Model-Checking Algorithmus ([BK08], S.737). Die allgemeine Vorgehensweise lautet:

- Die TCTL Formel Φ des CSC wird in die existentielle Normalform gebracht.
- Der Syntax-Baum von Φ wird aufgespannt.
- Von den Blättern ausgehend werden die Erfüllbarkeitsmengen $Sat(\Phi_i)$ rekursiv berechnet. Dabei wird Φ_i (und dem Timed Automaton $\mathcal{A}$) eine neue Uhr z hinzugefügt, um den Zeitparameter von Φ_i zu eliminieren (siehe hierzu Abschnitt 3.2.2). Sobald $Sat(\Phi_i)$ für ein i berechnet wurde, wird die Uhr z wieder frei und kann dann durch ein Reset für $i+1$ verwendet werden. Somit kann dieselbe Uhr z für jedes Φ_i wiederverwendet werden und der Zustandsraum wird allein um eine Uhr-Variable erweitert.
- Ist die Menge der Initialzustände eine Teilmenge der Erfüllbarkeitsmenge $Sat(\Phi)$, so erfüllt das Transitionssystem die TCTL Spezifikation des CSC.

Je nach Systemgröße kann das explizite oder das symbolische Model-Checking genutzt werden, um die Erfüllbarkeitsmengen $Sat(.)$ zu bestimmen (Abschnitt 3.2.1). Ist der Zustandsraum explizit aufgespannt bzw. sind die Binären Entscheidungsdiagramme erstellt, erfolgt die Berechnung der Erfüllbarkeit von Φ auf dem Transitionssystem in konstanter Zeit [CBRZ01].

Algorithmus 1 TCTL Model-Checking (Grundidee)

Eingabe: Timed Automaton $\mathcal{A}$ (non-zeno, timelock frei), TCTL Formel Φ
Ausgabe: JA, falls $\mathcal{A} \models \Phi$, sonst NEIN

$R := RTS(\mathcal{A} \oplus z, \Phi)$;
{* Region-Transitionssystem RTS *}
{* mit Zustandsraum S_{rts} und Zuordnungsvorschrift L_{rts} *}

for all $i \leq \mid \Phi \mid$ **do**
  for all $\Psi \in Sub(\Phi)$ mit $\mid \Psi \mid = i$ **do**
    if $\Psi = true$ **then**
      $Sat_R(\Psi) := S_{rts}$;
    else if $\Psi = a$ **then**
      $Sat_R(\Psi) := \{s \in S_{rts} \mid a \in L_{rts}(s)\}$;
    else if $\Psi = \Psi_1 \wedge \Psi_2$ **then**
      $Sat_R(\Psi) := \{s \in S_{rts} \mid \{a_{\Psi_1}, a_{\Psi_2}\} \subseteq L_{rts}(s)\}$;
    else if $\Psi = \neg\Psi'$ **then**
      $Sat_R(\Psi) := \{s \in S_{rts} \mid a_{\Psi'} \notin L_{rts}(s)\}$;
    else if $\Psi = \exists(\Psi_1 \, \mathcal{U}^J \Psi_2)$ **then**
      $Sat_R(\Psi) := Sat_{CTL}(\exists((a_{\Psi_1} \vee a_{\Psi_2}) \, \mathcal{U} \, ((z \in J) \wedge a_{\Psi_2})))$;
    else if $\Psi = \forall(\Psi_1 \, \mathcal{U}^J \Psi_2)$ **then**
      $Sat_R(\Psi) := Sat_{CTL}(\forall((a_{\Psi_1} \vee a_{\Psi_2}) \, \mathcal{U} \, ((z \in J) \wedge a_{\Psi_2}))$;
    else
      return
    end if
    for all $s \in S_{rts}$ with $s\{z := 0\} \in Sat_R(\Psi)$ **do**
      $L_{rts}(s) := L_{rts}(s) \cup \{a_\psi\}$;
    end for
  end for
end for
if $L_{rts} \subseteq Sat_R(\Phi)$ **then**
  return JA
else
  return NEIN
end if

Beispiel 16. *Abbildung 5.4 zeigt die Zustandswechsel einer Berechnung zur Erfüllung der Formel* $l_1 \overset{<300}{\longrightarrow} l_2 \overset{\geq 500}{\longrightarrow} l_3$*. Die Abbildung enthält ebenfalls den Verlauf zweier Uhren, die herangezogen werden, um die TCTL-Formel in eine CTL-Formel zu transformieren.*

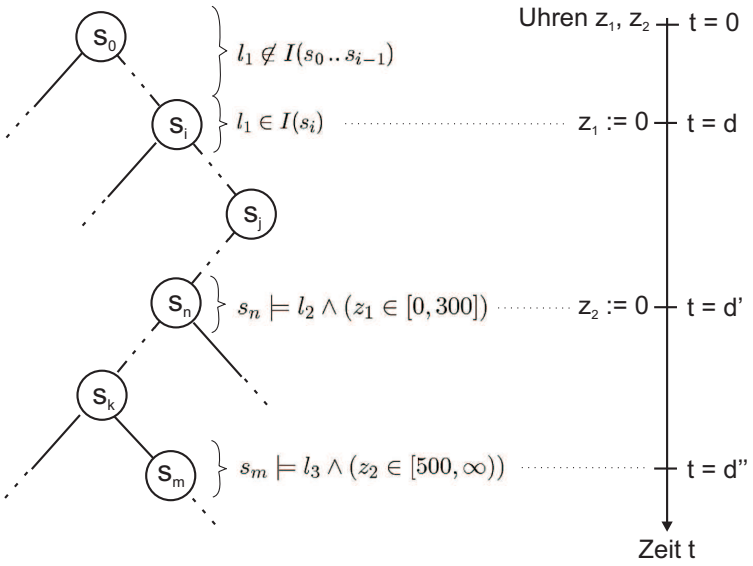
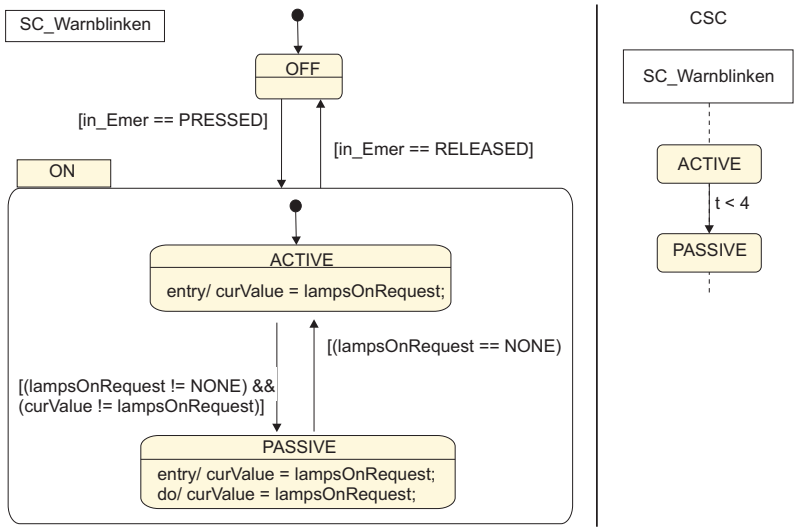


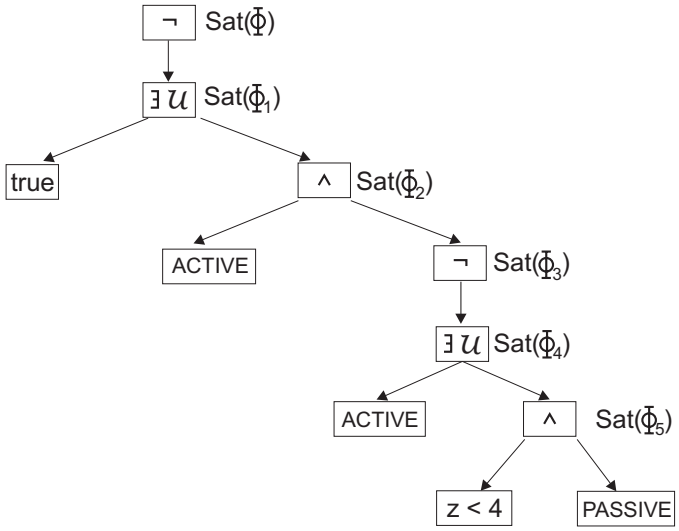
Abbildung 5.4: *Zustandswechsel zur Erfüllung der Formel* $l_1 \overset{<300}{\longrightarrow} l_2 \overset{\geq 500}{\longrightarrow} l_3$

□

Beispiel 17. *Abbildung 5.5(a) zeigt das Statechart Warnblinken und ein CSC, das aus den Positionen* `ACTIVE` *und* `PASSIVE` *gebildet wurde. Das CSC fordert den Übergang von Position* `ACTIVE` *zu Position* `PASSIVE` *innerhalb von 4 Zeiteinheiten. Die dazugehörende Relation lautet* $\texttt{ACTIVE} \xrightarrow{t<4} \texttt{PASSIVE}$*. Die korrespondierende TCTL-Formel*



(a) Statechart SC_Warnblinken und ein dazugehörendes CSC



(b) Syntaxbaum von Φ'

Abbildung 5.5: *Zustandsautomat, CSC und Syntaxbaum*

$$\Phi = \forall\Box\Big(\texttt{ACTIVE} \rightarrow \exists\big(\texttt{ACTIVE}\, \mathcal{U}^{[0,4)}\, \texttt{PASSIVE}\big)\Big)$$

lautet in ENF

$$\Phi' = \neg\exists(true\, \mathcal{U}\, (\texttt{ACTIVE} \wedge \neg\exists(\texttt{ACTIVE}\, \mathcal{U}\, (\texttt{PASSIVE} \wedge (z < 4))))),$$

wobei bereits eine neue Uhr z *hinzugefügt wurde. Der dazugehörende Syntaxbaum ist in Abbildung 5.5(b) zu sehen. Mithilfe von explizitem oder symbolischem Model-Checking kann die Erfüllbarkeit von* Φ' *auf dem Zustandsautomaten* `SC_Warnblinken` *nachgewiesen und ein Testfall generiert werden.*

□

Bounded Model-Checking Die Generierung von Testfällen aus *Timed Statecharts* unter Berücksichtigung eines CSCs mit Hilfe von Bounded Model-Checking bietet zwei Alternativen.

1) Mit TCTL-Zwischenrepräsentation

- Die TCTL Formel Φ wird durch Hinzufügen von neuen Uhren z_i in eine CTL Formel $\hat{\Phi}$ transformiert.
- Die CTL Formel $\hat{\Phi}$ wird in Prädikatenlogik umgeformt. Die resultierende prädikatenlogische Formel sei P.
- P wird in die konjunktive Normalform $\hat{P}$ gewandelt ([Rec06], S. 38).
- Die prädikatenlogische Repräsentation $\hat{P}$ des CSC wird Teil der BMC-Instanz

$$[[M,P]]_k = \bigwedge_{j=0}^{c-1} T(i(k_0+j), s(k_0+j), s(k_0+j+1)) \\ \wedge \hat{P}(i(k_0), s(k_0), o(k_0), \ldots, i(k_0+c), s(k_0+c), o(k_0+c)).$$

2) Ohne TCTL-Zwischenrepräsentation

- Die prädikatenlogische Formel P des CSCs wird direkt erstellt.
- P wird in die konjunktive Normalform $\hat{P}$ gewandelt.
- Die prädikatenlogische Repräsentation $\hat{P}$ des CSC wird Teil der BMC-Instanz

$$[[M,P]]_k = \bigwedge_{j=0}^{c-1} T(i(k_0+j), s(k_0+j), s(k_0+j+1)) \\ \wedge \hat{P}(i(k_0), s(k_0), o(k_0), \ldots, i(k_0+c), s(k_0+c), o(k_0+c)).$$

Der Vorteil von Alternative 1) liegt in der leichteren Verständlichkeit der TCTL-Zwischenrepräsentation. Da die TCTL-Formel wahrscheinlich für den Testingenieur nicht von Bedeutung sein wird, empfiehlt sich die leichter umzusetzende Variante 2).

Um die Komplexität der zu lösenden BMC-Instanz so gering wie möglich zu halten, empfiehlt es sich ebenfalls, die CSC-Formel stückweise aufzubauen. Die einzelnen Teilschrittformeln sollten in die BMC-Instanz schrittweise eingebaut werden. Das stückweise Generieren von Testschritten, indem Zwischenziele gesetzt werden, nach deren Berechnung jeweils eine neue BMC-Instanz bestimmt wird, ist eine bekannte Vorgehensweise, die auch in diesem Fall hilfreich sein kann. Dabei wird eine Lösung für die BMC-Instanz nicht nur vom Initialzustand ausgehend, sondern auch ab einem vorher bestimmten Berechnungsfragment aufgebaut [PZck]. Auf diese Art verringert sich die Komplexität der BMC-Instanz.

Beispiel 18. *Abbildung 5.5(a) zeigt das Statechart Warnblinken und ein CSC, dass aus den Positionen* `ACTIVE` *und* `PASSIVE` *gebildet wurde. Das CSC fordert den Übergang von Position* `ACTIVE` *zu Position* `PASSIVE` *innerhalb von 4 Zeiteinheiten. Die dazugehörende Relation lautet* `ACTIVE` $\xrightarrow{t<4}$ `PASSIVE`.

Der dazugehörende prädikatenlogische Ausdruck lautet

$\Phi =$ *Für alle* $\pi \in$ *Pfade*$_{div}(s_0).\forall i \geq 0.$

$\Big($ACTIVE $\notin I(s_i) \vee \big($*für mindestens ein* $\pi' \in$ *Pfade*$_{div}(s_i\{z := 0\}).$

$\exists n \geq i.\, s_n + d \models$ PASSIVE $\wedge\, (z \in [0,4))$ *für mindestens ein* $d \in [0, d_n]$

mit $\sum_{k=0}^{n-1} d_k + d \in [0,4)$ *und*

$\forall i \leq j \leq n.\, ($ACTIVE $\in I(s_j)) \vee ($PASSIVE $\in I(s_j))$ *für alle* $d' \in [0, d_j]$

mit $\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{n-1} d_k + d\ \big)\Big)$

Der erste Teil der BMC-Instanz bei einer Abrolltiefe von $c = 6$ *lautet (vgl. Beispiel 11, Abschnitt 3.2.4):*

$$\bigvee_{j=0}^{5} R(i(k_0+j), s(k_0+j), s(k_0+j+1)) =$$
$$(in_Emer(0) = PRESSSED) \wedge$$
$$\big(0, (\texttt{OFF}(0), \begin{pmatrix} in_Emer(0) = RELEASED \\ curValue(0) = NONE \\ lampsOnRequest(0) = NONE \end{pmatrix}, \vec{0})\big) \wedge$$
$$\big(0, (\texttt{ON}(1), \begin{pmatrix} in_Emer(1) = PRESSED \\ curValue(1) = NONE \\ lampsOnRequest(1) = NONE \end{pmatrix}, \vec{0})\big) \wedge \ldots$$

Der zweite Teil der BMC-Instanz hat die Form

$$P(i(0), loc(0), \ldots, i(6), loc(6)) \equiv$$
$$\bigvee_{j=0}^{5} \big(\texttt{ACTIVE}(k_0+j) \wedge (z(k_0+j+1) < 4) \wedge \texttt{PASSIVE}(k_0+j+1)\big)$$

P *ist aus* Φ *abgeleitet und bereits in konjunktiver Normalform. Die BMC-Instanz könnte zum Beispiel mit dem Input*

$$\omega = \Big\langle (0, (in_Emer = PRESSED, lampsOnRequest = NONE), \vec{0}, \vec{0}) \Big\rangle,$$
$$\Big\langle (0.5, (in_Emer = PRESSED, lampsOnRequest = LEFT), \vec{0}, \vec{0}) \Big\rangle$$

gelöst werden. □

5.4 CSC Abgleich mit generierten Testfällen

Ein generierter Testfall ist ein endlicher Pfad durch das Testmodell. Er repräsentiert eine mögliche Folge von Belegungen der Modellvariablen.

Sei π ein Testfall eines Testmodells MC. Die Vorgehensweise, um zu entscheiden, ob ein Testfall π eine TCTL Formel Φ erfüllt, $\pi \models \Phi$, unterscheidet sich nicht von der Variante, die Erfüllbarkeit der Formel auf einem Modell zu entscheiden, $MC \models \Phi$. Anstatt eine Transitionsrelation für das Testmodell aufstellen zu müssen, ist die Voraussetzung zur Berechnung der Erfüllbarkeit bei einem Testfall bereits gegeben. Der Testfall enthält keine alternativen Systemverhaltensweisen. Ob der Testfall eine gültige Belegung unter Φ darstellt, kann bewiesen oder widerlegt werden.

Ähnlich zur Vorgehensweise bei Timed Automata (Abschnitt 5.3) wird zuerst der TCTL Syntaxbaum erzeugt und rekursiv von den Blättern des Baumes ausgehend, die Erfüllbarkeitsmengen $Sat(\Phi_i)$ der Teilformeln generiert. Hierbei muss nicht zwingend über eine zusätzliche Uhr z eine Eliminierung der Zeitparameter vorgenommen werden. Da alle Zeitstempel der Positionswechsel, also alle Verweildauern in einer Position, bekannt sind, kann eine direkte Beurteilung der zeitlichen Restriktionen vorgenommen werden.

Das Bounded Model-Checking verfährt ebenso wie in Abschnitt 5.3 beschrieben. Die Gesamtformel Φ wird entweder über die TCTL-Zwischenrepäsentation oder direkt in Prädikatenlogik erstellt und ihre Erfüllbarkeit auf dem „Modell“ π wird berechnet. Da π keine komplexe Transitionsrelation erzeugt ist auch die BMC-Instanz überschaubar, so dass in den meisten Fällen auf eine stückweise Berechnung verzichtet werden kann.

Beispiel 19. *Gegeben sei eine Berechnungsfolge*

$$\begin{aligned}\pi \quad = \quad & \langle(0,(\overbrace{\texttt{OFF}}^{Position},\overbrace{0}^{Uhr\ t}))\rangle\,\langle(0.5,(\texttt{ON},0.5))\rangle\,\langle(0.5,(\texttt{ACTIVE},0.5))\rangle \\ & \langle(1,(\texttt{ACTIVE},1))\rangle\,\langle(1,(\texttt{PASSIVE},0))\rangle\,\langle(3,(\texttt{PASSIVE},2))\rangle \\ & \langle(3,(\texttt{ACTIVE},2))\rangle\end{aligned}$$

Es soll gezeigt werden, dass der Pfad π die Synchronisation ACTIVE $\xrightarrow{<4}$ PASSIVE *erfüllt.*

Wählt man das explizite oder symbolische Model-Checking, wird die zur Relation gehörende TCTL-Formel in ENF erstellt:

$$\Phi = \neg\exists(true\,\mathcal{U}\,(\texttt{ACTIVE} \wedge \neg\exists(\texttt{ACTIVE}\,\mathcal{U}\,(\texttt{PASSIVE} \wedge (z < 4)))))$$

Die Berechnungsfolge erhält nach dem (optionalen) Hinzufügen der Uhr z und einer Beschränkung auf Zustände aus dem Region-Transitionssystem die verbleibenden Zustände

$$\begin{aligned}\pi \quad = \quad & \langle(0,(\overbrace{\texttt{OFF}}^{Position},(\overbrace{0}^{Uhr\ t},\overbrace{0}^{Uhr\ z})))\rangle\,\langle(0.5,(\texttt{ON},(0.5,0.5)))\rangle \\ & \langle(0.5,(\texttt{ACTIVE},(0.5,0)))\rangle\,\langle(1,(\texttt{PASSIVE},(0,0.5)))\rangle \\ & \langle(3,(\texttt{ACTIVE},(2,0)))\rangle\end{aligned}$$

Zur Wiederholung zeigt Abbildung 5.6 den Syntaxbaum aus Beispiel 17.

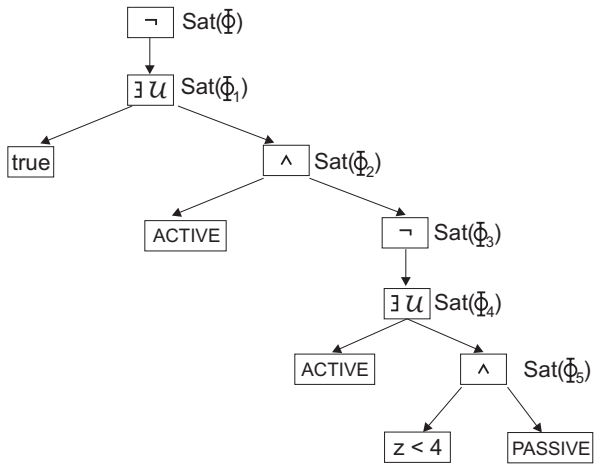


Abbildung 5.6: *Syntaxbaum von Φ*

Für die Erfüllbarkeitsmengen lauten die Ergebnisse

$$
\begin{aligned}
Sat(\texttt{true}) &= \{\langle(0,(\texttt{OFF},(0,0)))\rangle, \langle(0.5,(\texttt{ON},(0.5,0.5)))\rangle, \\
&\quad \langle(0.5,(\texttt{ACTIVE},(0.5,0)))\rangle, \langle(1,(\texttt{PASSIVE},(0,0.5)))\rangle, \\
&\quad \langle(3,(\texttt{ACTIVE},(2,0)))\rangle\} \\
Sat(\texttt{ACTIVE}) &= \{\langle(0.5,(\texttt{ACTIVE},(0.5,0)))\rangle, \langle(3,(\texttt{ACTIVE},(2,0)))\rangle\} \\
Sat(\texttt{PASSIVE}) &= \langle(1,(\texttt{PASSIVE},(0,0.5)))\rangle \\
Sat(z<4) &= \{\langle(0,(\texttt{OFF},(0,0)))\rangle, \langle(0.5,(\texttt{ON},(0.5,0.5)))\rangle, \\
&\quad \langle(0.5,(\texttt{ACTIVE},(0.5,0)))\rangle, \langle(1,(\texttt{PASSIVE},(0,0.5)))\rangle, \\
&\quad \langle(3,(\texttt{ACTIVE},(2,0)))\rangle\} \\
Sat(\Phi_5) &= \langle(1,(\texttt{PASSIVE},(0,0.5)))\rangle \\
Sat(\Phi_4) &= \{\langle(0.5,(\texttt{ACTIVE},(0.5,0)))\rangle, \langle(1,(\texttt{PASSIVE},(0,0.5)))\rangle\} \\
Sat(\Phi_3) &= \{\langle(0,(\texttt{OFF},(0,0)))\rangle, \langle(0.5,(\texttt{ON},(0.5,0.5)))\rangle, \\
&\quad \langle(3,(\texttt{ACTIVE},(2,0)))\rangle, \\
Sat(\Phi_2) &= \langle(3,(\texttt{ACTIVE},(2,0)))\rangle \\
Sat(\Phi_1) &= \langle(1,(\texttt{PASSIVE},(0.5,0.5)))\rangle \langle(3,(\texttt{ACTIVE},(2,0)))\rangle \\
Sat(\Phi) &= Sat(\neg\Phi_1) = S \backslash Sat(\Phi_1) = \\
&\quad \{\langle(0,(\texttt{OFF},(0,0)))\rangle, \langle(0.5,(\texttt{ON},(0.5,0.5)))\rangle, \\
&\quad \langle(0.5,(\texttt{ACTIVE},(0.5,0)))\rangle\}
\end{aligned}
$$

Da der Initialzustand $\langle(0,(\texttt{OFF},(0,0)))\rangle$ der Berechnungsfolge in $Sat(\Phi)$ enthalten ist, ist π eine gültige Belegung für die Formel Φ.

Alternativ kann mithilfe des Bounded Model-Checkings die Erfüllbarkeit berechnet werden.

□

5.5 Von der Anforderung zum CSC

Ein mögliches Verhalten eines Systems, beschrieben als einzelne natürlichsprachliche Anforderung, lässt sich als eine Folge von Zustandswechseln im Testmodell abbilden. Der Bezug einer einzelnen Anforderung auf Zustandssequenzen eines Testmodells liegt im Ermessen des Testers. Seine Fähigkeit, die Anforderung zu formalisieren, wird bereits der Erstellung des Testmodells eingesetzt. CSCs werden am besten parallel zur Testmodellerstellung gebildet. Sobald Zustände, Entscheidungstabellenzeilen und Relationen modelliert wurden, um das Verhalten der einzelnen Anforderung im Testmodell abzubilden, sollten die benötigten Zustandswechsel als CSC abgelegt werden. Eine nachträgliche Erstellung ist zwar möglich, allerdings auch aufwändiger.

Ob das CSC das Verhalten einer natürlichsprachlichen Anforderung adäquat widerspiegelt, muss durch Reviews sichergestellt werden. Ebenso wie bei der Erstellung des Testmodells ist es hierbei möglich, die Anforderung falsch zu interpretieren. Eine aufwändigere, aber dennoch sinnvolle Vorgehensweise wäre es, das Testmodell von einer anderen Person erstellen zu lassen. Werden Testmodell und CSC von unabhängigen Personen erstellt, steigt das Vertrauen in die Korrektheit des Testmodells und der zugehörigen CSCs.

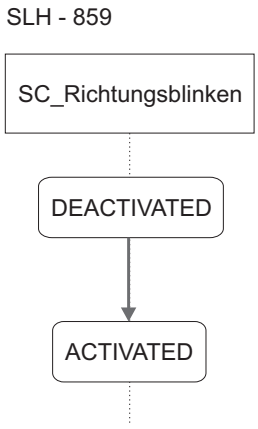
Beispiel 20. *Fortsetzung des Beispiels* „Blinkfunktionalität“ *aus Abschnitt 2.5.*

Um die einzelnen Anforderungen der Blinkfunktionalität zur anforderungsbasierten Testfallgenerierung nutzen zu können, müssen sie zuerst als CSC spezifiziert werden. Wesentliche Automaten, die das Soll-Verhalten des zu testenden Systems beschreiben, sind SC_Entprellung, SC_Richtungsblinken und SC_Warnblinken. Die Umgebungsautomaten sind für das Verfolgen

von Anforderungen nicht von Bedeutung. Die Belegung von Systemstimuli ist Aufgabe des Testfallgenerators und eine weitere Einschränkung der Systemnutzung erschwert die Testfallgenerierung zusätzlich. Aus Sicht der Anforderungsverfolgung ist es allein von Bedeutung, dass ein spezifiziertes Systemverhalten beobachtbar wird. Ob dafür der Blinkerhebel nach links oder nach rechts bewegt wird, ist meist nicht von Bedeutung. Falls doch einmal die Unterscheidung von Links- und Rechtsblinken notwendig wird, sollte dies innerhalb der CSCs über explizite Variablenbelegungen erzwungen werden, da sonst dem Testfallgenerator die Wahl überlassen wird.

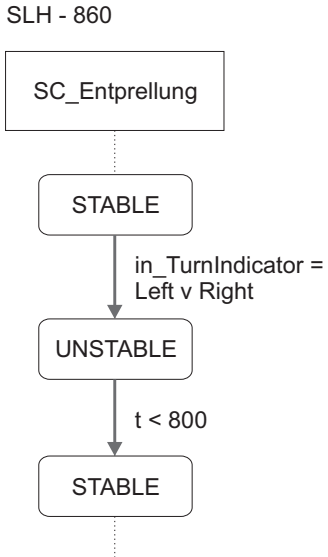
Ein Auszug des Lastenhefts der Aussenlichtfunktionalität ist in Tabelle 2.2 enthalten und die dazugehörenden CSCs sind in Abbildung 5.7 abgelegt (CSCs 5.7(a) bis 5.7(g)).

SLH-859: Voraussetzung für die Aktivierung von Richtungsblinken ist Schlüssel auf `Radiostellung` oder `Motor an`.



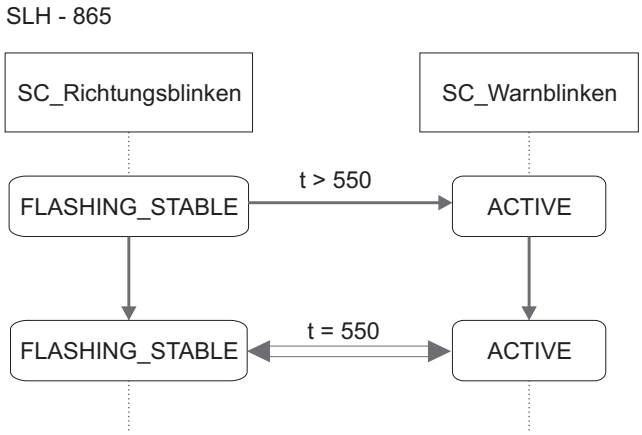
(a) SLH-859

SLH-860: Die Blinkhebelstellung für Richtungsblinken `Links` oder `Rechts` wird eingelesen und entprellt.



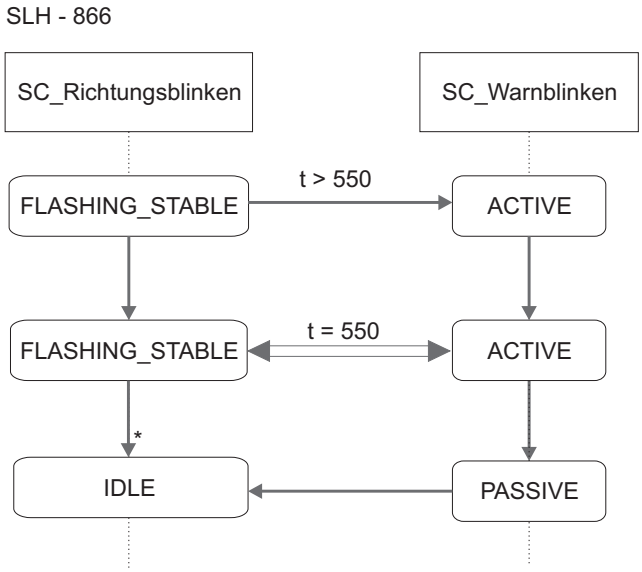
(b) SLH-860

SLH-865: Richtungsblinken kann durch Warnblinken unterbrochen werden.



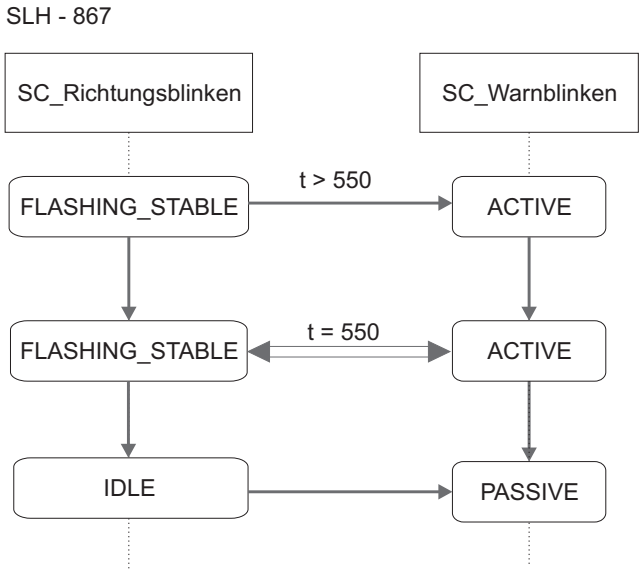
(c) SLH-865

SLH-866: Ein durch Warnblinken unterbrochenes Richtungsblinken wird bei Wegfall der Warnblinkanforderung wieder aufgenommen.



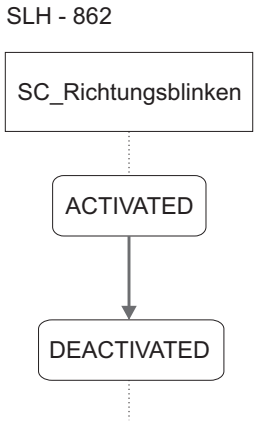
(d) SLH-866

SLH-867: Ein durch Warnblinken unterbrochenes Richtungsblinken wird bei Wegfall der Richtungsblinkanforderung zurückgenommen.



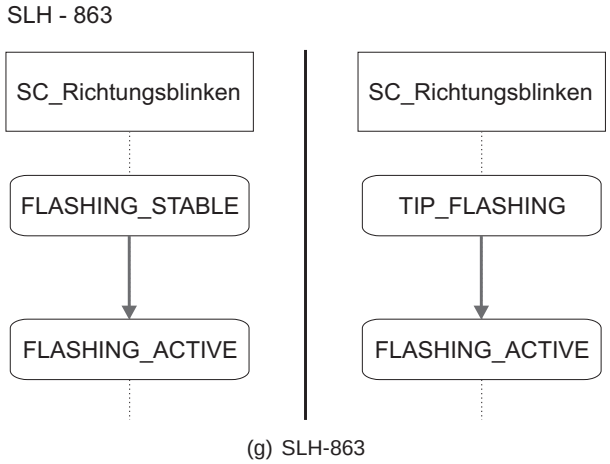
(e) SLH-867

SLH-862: Wegfall der Funktionsvoraussetzung.



(f) SLH-862

SLH-863: Eine neue Blinkanforderung deaktiviert die aktuelle Blinkanforderung.



(g) SLH-863

Abbildung 5.7: *CSCs der Anforderungen aus Tabelle 2.2*

Kapitel 6

Zusammenfassung und Ausblick

Ein Kernpunkt der Testfallermittlung ist die Anforderungsüberdeckung der ursprünglich an das System gestellten Anforderungen. Gemeint sind die originalen, natürlichsprachlichen Anforderungen, die von Kunden im Entwicklungsprozess spezifiziert wurden, meist abgelegt in Pflichten- und Lastenheften. Ziel der Arbeit ist es, eine Anforderungsüberdeckung im Modellbasierten Test zu entwickeln, die einerseits über das zielgerichtete Generieren von Testfällen für Anforderungen und andererseits durch den Abgleich bereits generierter Testfälle mit Anforderungen, erreicht wird.

Die Verfolgung der Testartefakte und Dokumentation der Testfallgenerierung ist neben der Anforderungsüberdeckung ein weiterer Bestandteil des Modellbasierten Tests. Um eine Zuordnung von Testfällen, am Testsystem gelaufenen Testskripten und sonstigen, während des Testprozesses anfallenden Artefakten zu Anforderungen sicherzustellen, ist eine Anforderungsverfolgung notwendig. Zusätzlich zum besseren Verständnis von Testfällen werden Wiederverwendung und Wartbarkeit von Testmodellen durch Requirements Traceability verbessert. Ziel ist es, eine schnelle, einfache und werkzeuggestützte Verlinkung von Anforderungen und Testartefakten zu erhalten. Der dadurch erzeugte Mehraufwand darf die Effizienz des Modellbasierten Testprozesses nicht maßgeblich beeinflussen.

Die Akzeptanz einer neuen Methode zur automatischen Testfallermittlung hängt stark von der Nachvollziehbarkeit der Prozessschritte und seiner Ergebnisse ab. Akzeptanz ist ein wichtiger Erfolgsfaktor bei der Einführung einer neuen Testfallermittlungsmethode wie dem Modellbasierten Test. Vorrangig von Interesse sind dabei nicht die im Testfallgenerator implementierten Algorithmen der Testauswahl. Es geht vielmehr darum, die Motivation von Testfällen nachvollziehen zu können. Bei der Testfallauswahl beruhen die Entscheidungen des Testingenieurs auf den Anforderungen, ihm bekannte Testfallermittlungsverfahren, seinem Wissen über Fehlerquellen des zu testenden Systems und seinen allgemeinen Erfahrungen bei der Fehlerfindung. Im Gegensatz dazu sind für den Testfallgenerator Testfälle motiviert aufgrund von Modellüberdeckungen und Testauswahlkriterien, die maximiert werden sollen. Beide Motivationen sind notwendig und müssen als zwei Sichtweisen auf denselben Testfall unterstützt und visualisiert werden. Eine Nachvollziehbarkeit bis hin zu Anforderungen kann helfen, sie miteinander zu verknüpfen.

Test-Verantwortliche, Zertifizierer und Auditoren müssen auch nachträglich nachvollziehen können, welche Testfälle ausgewählt wurden und an welchen Kriterien sich die Auswahl orientierte, um in der Vergangenheit getroffene Entscheidungen und Testendekriterien bewerten zu können. Die Dokumentation der Testfallermittlung sowohl als auch der Testfälle ist ein geforderter Teilschritt der Qualitätsabsicherung. Der Bezug zwischen Testfällen und Anforderungen muss Bestandteil der Dokumentation sein. Dies fordern Qualitätshandbücher und Sicherheitsnormen, wie die ISO26262.

Der praktische Gewinn aus Requirements Tracing im modellbasierten Testen anhand von Computation Sequence Charts begründet sich dadurch, dass es

- die Nachvollziehbarkeit der Testprozesserstellung für alle am Testprozess beteiligten Par-

teien erreicht,

- die Verständlichkeit der entstehenden Test-Suiten garantiert, indem jeder Testfall neben der Motivation der strukturellen Testmodellüberdeckung auch den Bezug zu einer oder mehrerer Anforderungen erhält,
- ein Anforderungsüberdeckung durch realitätsnahe Testfälle ermöglicht, da allein für eine Anforderung generierte Testfälle genau das spezifizierte Systemverhalten zeigen,

Der Beitrag vorliegender Arbeit stellt einen expliziten, formalen Beschreibungsformalismus für das Requirements Tracing und die anforderungsbasierte Testfallgenerierung anhand von Model-Checking dar. Der Formalismus heißt **Computation Sequence Chart**, kurz CSC. Ein CSC spezifiziert eine Einzelanforderung über eine Folge von Zustandswechseln aus einem Testmodell. Die Zustandsfolge darf Zyklen enthalten, Zustandswechsel dürfen endlich viele Zwischenschritte benötigen und parallele und sequentielle Verläufe können synchronisiert werden.

Im Folgenden sind die Vorteile eines CSC aufgezählt.

- Ein CSC wird direkt aus den Elementen eines Testmodells gewonnen.
- In einem CSC können zeitliche Abhängigkeiten von Zuständen spezifiziert werden.
- In einem CSC können Variablenbelegungen spezifiziert werden.
- In einem CSC kann derselbe Zustand mehrfach vorkommen.
- Über ein CSC können Zyklen in Pfaden spezifiziert werden.
- Ein CSC kann automatisch in TCTL und in Prädikatenlogik transformiert werden.
- Ein Testfallgenerator kann Testfälle generieren, die in einem CSC spezifizierte Abläufe enthalten.
- Ein Testfallgenerator kann generierte Testfälle einem CSC, und damit einer Anforderungen, zuordnen.
- Durch Änderungen einer Anforderung betroffene Modellelemente können über das CSC leicht identifiziert werden.

Die Verfolgung der Anforderungs-Testmodellbeziehung darf nur einen geringfügigen Mehraufwand im Testprozess erzeugen. Die Anforderungsbeschreibung besitzt eine leicht verständliche graphische Repräsentation. Der Testingenieur setzt das CSC durch bereits im Testmodell vorhandene Modellelemente zusammen. Dabei kann er zusätzlich aus 6 Synchronisationsrelationen zwischen Systemzuständen wählen. Die Erstellung des CSC ist werkzeugunterstützt und geführt. Ein CSC stellt einen Bezug zwischen Anforderung und Testmodell her, der keine zweite Formalisierung der Anforderungen benötigt. Es stellt lediglich eine Teilpfadauswahl des bereits existierenden Testmodells dar. Die „Denkweise“ bei der Testmodellerstellung und der CSC Erstellung ist dadurch identisch. Die Überprüfung auf syntaktische und semantische Korrektheit, die Generierung CSC-bezogener Testfälle sowie der Abgleich mit generierten Testfällen erfolgt automatisch.

Die in dieser Arbeit vorgestellte graphische Notation zum Requirements Tracing ermöglicht, durch den Einsatz der Temporallogik TCTL, das Tracing nicht nur zu nutzen, um eine einfache Verknüpfung von Anforderung und Testfall zu erhalten, sondern auch, Testfälle explizit aufgrund von Anforderungen zu generieren. Die Synchronisationsrelationen eines CSC werden mit TCTL Teilformeln hinterlegt und über einen Algorithmus zu einer Gesamtformel Φ kombiniert. Die Mächtigkeit des temporallogischen Sprachumfangs wird durch die vorgegebenen Synchronisationsmechanismen und ihre TCTL Repräsentation reduziert, allerdings gewinnt das Verfahren gleichzeitig den praktischen und notwendigen Vorteil, dass ein Verständnis von der im Hintergrund erzeugten Temporallogik vom Testingenieur nicht erwartet wird.

Die Arbeit stellt die Einbindung der CSC-Erstellung in den klassischen modellbasierten Testprozess und in das Model-Checking zur Testfallgenerierung vor. Die Testfallgenerierung mit CSCs und der Abgleich von generierten Testfällen und CSCs wird beispielhaft für das Symbolische Model-Checking auf Timed Automata und TCTL Formeln und das Bounded Model-Checking auf Timed Statecharts und prädikatenlogischen Formeln erläutert. Beim Symbolischen Model-Checking wird dem Testfallgenerator die TCTL Gesamtformel eines CSC direkt übergeben. Ihre Erfüllbarkeit kann zum Beispiel über den Einsatz von BDDs berechnet werden. Die TCTL Gesamtformel Φ eines CSC wird bei der Testfallgenerierung mit Bounded Model-Checking in eine prädikatenlogische Formel transformiert oder alternativ direkt prädikatenlogisch aufgestellt und als zu erfüllende Eigenschaft dem Testfallgenerator übergeben. Der Abgleich bereits generierter Testfälle erfolgt über dieselben Model-Checking Methoden, die auch bei der Testfallgenerierung eingesetzt werden. Ein Testfall TC entspricht einem Pfad durch das Testmodell und ist somit eine gültige Folge von Variablenbelegungen. Mit Hilfe der Model-Checking Verfahren kann bewiesen werden, ob gilt $TC \models \Phi$ oder $TC \not\models \Phi$.

6.1 Ausblick

Die Flexibilität von Computation Sequence Charts und ihre werkzeuggestützte Erstellung könnte in Zukunft durch folgende Weiterentwicklungen verbessert werden.

Um auch unabhängig von Positionen in Statecharts oder Entscheidungstabellenzeilen Variablenänderungen während einer Anforderung spezifizieren zu können, könnten künstliche Positionen in CSCs eingeführt werden, die beliebige prädikatenlogische Ausdrücke enthalten dürfen. Auf diese Art können komplexere Anforderungen, die über die Darstellung als Sequenz von Zustandswechseln hinausgehen, realisiert werden.

Eine weitere Möglichkeit, um Eigenschaften von Anforderungen spezifizieren können, sind Bestandteil der Live Sequence Charts und werden dort als *Pre-Chart* bezeichnet. Ein Pre-Chart muss vor dem eigentlich Ablauf erfüllt sein. Auf diese Art könnten beispielsweise die Funktionsvoraussetzung und Funktionsauslöser einer Anforderung in einem CSC abgebildet werden. Dies wurde bisher als nicht notwendig eingestuft, da die Funktionsvoraussetzungen und Funktionsauslöser bereits Teil des Testmodells sind. Der Testfallgenerator kennt diese Bedingungen und sie werden somit automatisch bei der Testfallgenerierung für eine Anforderung beachtet. Gäbe es jedoch mehrere mögliche Funktionsauslöser, kann die Vorgabe, einen speziellen Auslöser zu wählen, dem Testfallgenerator bisher nur über eine zusätzliche Zeitleiste übergeben werden.

Eine zweite Ergänzung wird ebenfalls von Live Sequence Charts umgesetzt. Über eine gestrichelte oder durchgezogene Umrandung des LSCs kann spezifiziert werden, ob die Anforderung auf mindestens einem Pfad (existentiell) oder auf allen Pfaden (universell) erfüllt sein muss. Die Möglichkeit zwischen existentiellen und universellen Anforderung unterscheiden zu können, wurde bisher nicht umgesetzt, da viele natürlichsprachliche Anforderungen eigentlich universell wären, sie werden aber aufgrund des besseren Verständnisses meist existentiell beschrieben. Ein sehr wichtiger Aspekt, der im Rahmen der Dissertation beachtet wurde, war es, so wenig Mehraufwand wie möglich für den Testingenieur zu generieren. Eigentlich existentielle Anforderung universell zu formulieren wurde aus diesem Grund nicht in Betracht gezogen, auch wenn dies grundsätzlich die bessere Vorgehensweise wäre. Um für eine Anforderung einen Testfall generieren zu können, reicht die existentielle Betrachtung der Anforderung aus.

Um den Mehraufwand so gering wie möglich zu halten, können die Arbeitsschritte zur Erstellung der Computation Sequence Charts noch weiter automatisiert werden. Eine Softwarelösung ist Case-Tool abhängig. Für die DAIMLER AG wurde eine Plug-In Lösung implementiert, die viele Arbeitsschritte abnimmt, wie zum Beispiel das Auswählen von Modellelementen und das Anordnen der Elemente auf Zeitleisten. Weitere Automatisierung im Bereich der Erstellung leerer CSCs und ihrer Zuordnung zu Anforderungen im Lastenheft wäre denkbar. Um die Wiederverwendung von CSCs zu unterstützen, sollten sie vom Testmodell losgelöst abgelegt werden. Werkzeugseitige Verbesserungen werden von der DAIMLER AG weiterhin verfolgt.

Anhang A

Algorithmus

In vorliegendem Kapitel wird eine exemplarische Implementierung der CSC Generierung skizziert. Die vorgenommenen Implementierungen werden in ihrem algorithmischen Kern mittels üblichem Pseudo-Code präsentiert; diese bauen auf den in den vorausgegangenen Kapiteln gegebenen Definitionen und Sätzen auf und folgen in ihrer Logik den dort gegebenen Beweisen. Der Bezug zwischen Algorithmus und Definition beziehungsweise Satz wird, falls direkt möglich, im Pseudo-Code angegeben.

Die Herleitung der TCTL Formeln aus einem CSC folgt einem schrittweisen Aufbau. Die Vorgehensweise enthält folgende Grundsätze.

- Innerhalb eines CSCs werden alle Zeitstrahle paarweise verarbeitet.
- Zu jedem Zeitstrahl werden ein Anfangs- und ein Endzustand hinzugefügt. Anfangs- und Endzustand sind über eine sequentielle Transition mit dem nachfolgenden beziehungsweise vorangehenden Zustand verknüpft.
- Mehrfachrelationen werden durch das Hinzufügen von Pseudozuständen aufgelöst.
- Beginnend bei den ersten echten Zuständen wird geprüft, ob eine parallele Beziehung zwischen beiden eingetragen ist. Wenn ja, wird abhängig von dem Ursprungszustand die Funktion zur Erstellung der TCTL Teilformeln aufgerufen.
- Die Teilformelfunktion verarbeitet zuerst die erste parallele Beziehung des Quellzustand, falls eine existiert.
- Nach den parallelen Beziehungen werden die sequentiellen Beziehungen pro Einzelzustand verarbeitet.
- Im Anschluss werden alle Teilschrittformeln der paarweisen Zeitstrahle zu einer Gesamtformel verknüpft.
- Danach bewegt sich jeder Zeitstrahl zu seinem nächsten Zustand, falls vorhanden und die nächsten Teilschrittformeln werden berechnet.
- Dies wird solange wiederholt, bis alle Zeitstrahle ihre Endzustände erreicht haben.

Innerhalb einer TCTL Formel wird ein *Zustand* einer Anforderungstabelle mit eindeutigem Pfad zum Ursprungsautomaten eingetragen. Der Name setzt sich aus den Namen der übergeordneten Funktionshierarchieebenen, durch Punkt getrennt, zusammen. Damit kann der Testfallgenerator diesen Zustand intern mit der Menge der atomaren Propositionen gleich setzen, die in diesem Zustand wahr sein müssen.

Bei einer *Entscheidungstabellenzeile*, also eine Bedingung/Aktion Kombination einer Transition, wird die Transitionsbedingung als eine Menge UND-verknüpfter atomarer Propositionen dargestellt.

Die logischen Ausdrücke werden eins zu eins in die TCTL Formel mit aufgenommen. Anhand des Zustands, in dem sich das Testmodell befindet wird entschieden, ob zu dem gewünschten Zeitpunkt der logische Ausdruck zu wahr oder zu falsch auswertet.

Die in den nachstehenden Algorithmen blau hervorgehobenen Codebereiche sind eine Kurzschreibweise für Ausdrücke, in denen TCTL-Formeln zusammengesetzt werden. Dafür bedient sich der Algorithmus an Klassen, die in einer TCTL-Library abgelegt sind. Jede Klasse entspricht einem Operator. Über die Vererbung der Klassen wird sichergestellt, dass die TCTL-Semantik nicht verletzt wird. So darf zum Beispiel einem Exists-Operator nur eine Pfadformel übergeben werden und die daraus entstehende Zustandsformel nur über Zustands-Operatoren wie dem Und-Operator weiter kombiniert werden.

Eine beispielhafte TCTL-Formel $\Phi = \exists\Phi \rightarrow (\exists\Phi_1 \rightarrow (\exists\Phi_2 \wedge \exists\Phi_3))$ sieht wie folgt aus:

$$\begin{aligned}\Phi = new\ TCTLImpliesNode\big(&new\ TCTLExistsNode(\Phi),\\ &new\ TCTLImpliesNode(new\ TCTLExistsNode(\Phi_1),\\ &\quad new\ TCTLAndNode(new\ TCTLExistsNode(\Phi_2),\\ &\quad\quad new\ TCTLExistsNode(\Phi_3))))\big)\end{aligned}$$

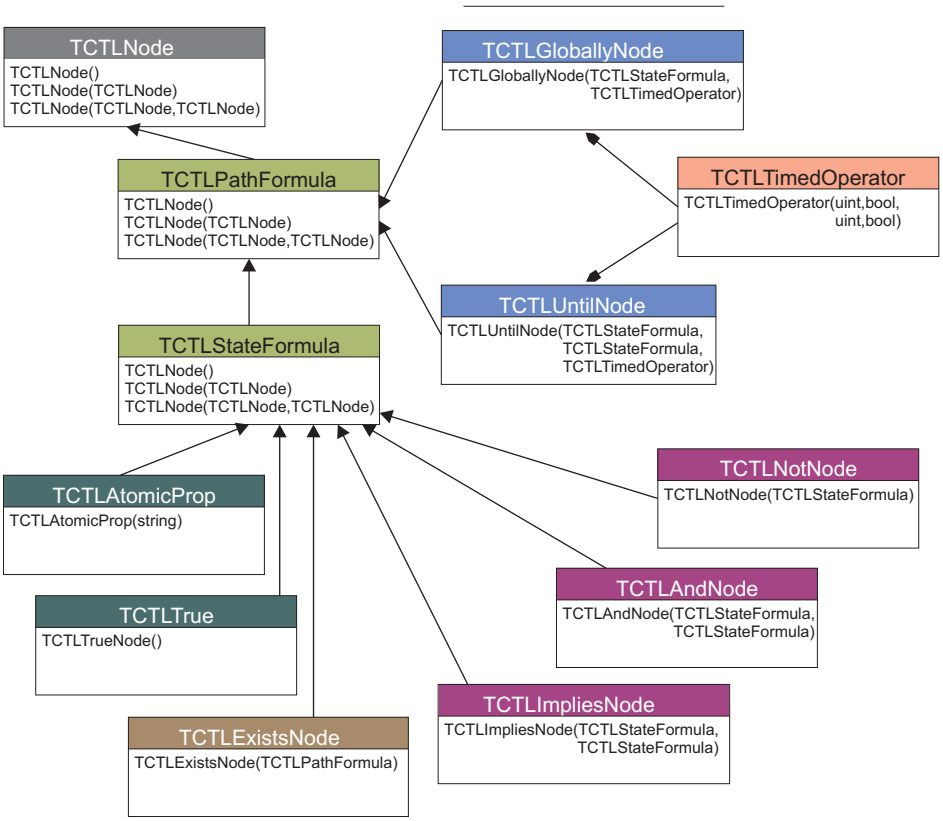


Abbildung A.1: *TCTL-Library Klassendiagramm*

Abbildung A.1 zeigt einen Ausschnitt des Klassendiagramms der TCTL-Library.

Der Algorithmus zum Einlesen eines CSCs und zum Transformieren dieses CSCs in die temporallogische Repräsentation arbeitet auf folgenden Klassen. Die `Timeline`-Klasse repräsentiert einen Zeitstrahl. Sie enthält `State` und `Transition` Instanzen. Die Klasse `State` repräsentiert einen Zustand eines CSCs und speichert alle Verweise auf Transitionen, die mit ihm verknüpft sind. Die Klasse `Transition` enthält Verweise auf Ursprungs- und Zielzustände und

alle Transitionsbedingungen, wie zum Beispiel zeitliche Grenzen oder erwünschte Variablenbelegungen.

Die Klasse `InterTimelines` speichert den Inhalt von zwei Zeitstrahlen. Je nach Anzahl der Zeitstrahle eines CSCs wird eine `InterTimeline` Instanz pro paarweise gebildeter Zeitstrahle erzeugt. Die `InterTimeline` Klasse löst beim Einlesen von zwei Zeitstrahlen jegliche Art von Mehrfachrelationen durch das Einführen von Pseudozuständen auf. Pseudozustände werden als spezielle `Duplicates` angelegt, die einen Verweis auf den Zustand beinhalten, den sie duplizieren. Durch das Auflösen von Mehrfachrelationen, kann ein `InterTimeline` Objekt die einzelnen Teilschrittformel selber berechnen und über einen Iterator einzeln zurückgeben.

Die Klasse CSC beinhaltet alle `InterTimelines`. Sie kann diese durchlaufen und die Schleife beenden, wenn alle `InterTimelines` keine weiteren Teilformeln mehr berechnen können.

Beim Erzeugen der Instanzen werden Prüfungen durchgeführt, die sicherstellen, dass alle Regeln aus Abschnitt 5.2 eingehalten wurden. Bei Fehlern bricht das Programm ab.

Die Klasse `StepFunction` speichert die Formelbestandteile eines Teilformelschritts. Die $\hat{\Psi}$, $\hat{\Phi^p}$ und $\hat{\Phi^s}$ Anteile eines Teilformelschritts werden getrennt abgelegt, um sie später in die Gesamtformel einzufügen.

Die Klasse `TCTLFormula` repräsentiert die Gesamtformel. Sie baut sich während der Berechnungen rekursiv auf. Wird ein neuer Formelbestandteil eingefügt, stellt die `TCTLFormula` Instanz sicher, dass dies an der richtigen Stelle mit entsprechender Klammerung geschieht.

Algorithmus 2 ist der Einstiegspunkt in die Generierung der Gesamtformel eines CSCs. An diesem Punkt sind alle graphischen Elemente wie Zeitstrahl, Zustand, Transition, ... bereits in Listen oder anderen Strukturen abgelegt und überprüft worden und können durchlaufen werden. Der Algorithmus folgt der Komposition mehrerer Zeitstrahle in Abschnitt 4.5 und der darin enthaltenen Definition 4.5.7.

Algorithmus 2 GenerateTCTLFormula

```
{Pairwise generation}
while !CSC.InterTimelines.End() do
  for all InterTimeline it do
    {Generate the StepFunction based on the next two states}
    stepFunction.Add(it.ProcessStepFunction())
  end for
  CombineSubformulas(stepFunction)
end while

{The outer operator is Always(Globally())}
if tctlFormula! = null then
  {A(G(Φ))}
end if
```

Zur Erstellung der Gesamtformel eines CSCs wird paarweise vorgegangen. In jedem `InterTimeline` Objekt werden die ersten beiden Zustände gewählt. Diese Zustände nutzt die Funktion `ProcessStepFunktion`, um damit eine Teilschrittformel zu bestimmen. Die Formelbestandteile der Teilschrittformel aller `InterTimeline` Objekte werden der Funktion `CombineSubformulas()` übergeben, die daraus, zusammen mit global abgelegten, bereits erstellten Formelanteilen die Gesamtformel `tctlFormula` aufbaut. Danach bewegen sich alle `InterTimeline` Objekte zu den nächsten Zuständen der Zeitstrahle, solange bis alle `InterTimeline` Objekte am Ende ihrer Zeitstrahle angekommen sind.

Die Funktion `ProcessStepFunction` (Algorithmus 3) geht wie in Abschnitt 4.5, Definition 4.5.3 definiert vor:

1. Falls eine parallele Transition zwischen Zustand s_1 und Zustand s_2 existiert, ablegen der TCTL-Formel in Φ_1

2. Falls keine parallele Transition zwischen Zustand s_1 und Zustand s_2 existiert, wird der Wechsel in s_1 und s_2 festgehalten und in Φ_1 gespeichert (vgl. Abbildung 4.14).

3. Gehen von s_1 und s_2 sequentielle Transitionen aus, werden diese in TCTL-Formeln abgelegt.

Eingabewerte der Funktion `ProcessStepFunction()` sind zwei Zustände des `InterTimeline` Objekts. Sie stellen die Anfangszustände einer neuen Teilschrittformel dar. Die beiden Zustände wissen, ob es eine parallele Relation zwischen ihnen gibt und ob ihnen ein weiterer Zustand sequentiell folgt. Muss eine parallele Relation verarbeitet werden, wird dafür die Funktion `ComputeParTransition()` aufgerufen. Sie bekommt die Transition übergeben und bestimmt die temporallogische Formel. Diese Formel wird als Teilformel in dem Speicherobjekt `stepFunction` gespeichert. Dasselbe Verfahren wird bei sequentiellen Transitionen und der Funktion `ComputeSeqTransition()` angewandt. Sie speichert die temporallogischen Formeln ebenfalls in dem Speicherobjekt `stepFunction`, welche im Anschluss an die Funktion `CombineSubformulas()` übergeben wird. `CombineSubformulas()` erstellt aus `stepFunction` und bestehenden global abgelegten Formelanteilen die neue Gesamtformel.

Algorithmus 3 ProcessStepFunction (Definition 4.5.3)

{Process the parallel transition, if available}
if $ExistsParallelTransition()$ **then**
  $\Phi_1 = (\hat{\Psi}_{p1} \rightarrow \hat{\Phi}_{p1}) = ComputeParTransition(trans)$
else
  $\{\Phi_1 = ((\hat{\Psi}_{s1} \rightarrow \hat{\Phi}_{s1}) \wedge (\hat{\Psi}_{s2} \rightarrow \hat{\Phi}_{s2}))\}$
end if
stepFunction.AddPar(Φ_1)
{Sequential transitions}
$\Phi_2 = \hat{\Phi}_{s1} = ComputeSeqTransition(trans)$
$\Phi_3 = \hat{\Phi}_{s2} = ComputeSeqTransition(trans)$
stepFunction.AddSeq(Φ_2, Φ_3)

Die sequentiellen und parallelen TCTL-Formeln für einzelne Transitionen ergeben sich aus den Definitionen in Abschnitt 4.4. In den folgenden zwei Funktionen `ComputeParTransition` (Algorithmus 4) und `ComputeSeqTransition` (Algorithmus 5) ist ein Auszug der Transitionen dargestellt.

Im Algorithmus `ComputeParTransition` sind alle Formeln für parallele Transitionen hart codiert hinterlegt. Der Übergabewert ist eine parallele Transition. Jede parallele Transition hat einen Namen, über den der Typ der Relation eindeutig referenziert werden kann. Dieser Name wurde der Transition bereits beim Einlesen des CSC gegeben. Abhängig von der zeitlichen Bedingung an der Transition wird dann die passende Formel zurückgegeben. Die Formeln sind in Kapitel 4 definiert. Der Algorithmus stellt eine einfache Eins zu Eins Transformation von paralleler Transition in temporallogische Formel dar. Da keine Berechnungen erfolgen, kann an dieser Stelle kein fälschlicher Zugriff auf ungültige Wertebereiche auftreten. Ist der Typ der parallelen Transition nicht bekannt, bricht der Gesamtalgorithmus an dieser Stelle ab.

Der Algorithmus für sequentielle Transitionen funktioniert vergleichbar zu Algorithmus 4.

Algorithmus 4 ComputeParTransition

if $trans == TR_PAR_ONE_WAY$ **then**
  ...
end if
if $trans == TR_PAR_BIDIRECT$ **then**
  {Definition 4.3.3, Satz 4.3.3}
  if $trans.GetTimimgCond() == NULL$ **then**
    $\hat{\Psi} = \exists\Diamond(s_1 \wedge s_2)$
    $\hat{\Phi} = NULL$
  else
    int $val = trans.GetTimimgCond().Bound$
    if $trans.GetTimimgCond().Operator.Equals(\text{“} > \text{“})$ **then**
      $\hat{\Psi} = s_1$
      $\hat{\Phi} = \exists(s_1\, \mathcal{U}(val, \infty)\ (s_1 \wedge s_2))$
      $\hat{\Psi}' = s_2$
      $\hat{\Phi}' = \exists(s_2\, \mathcal{U}(val, \infty)\ (s_1 \wedge s_2))$
    else if $trans.GetTimimgCond().Operator.Equals(\text{“} \leq \text{“})$ **then**
      ...
    else
      ...
    end if
  end if
  ...
end if
$trans.SetProcessed()$

Algorithmus 5 ComputeSeqTransition

if $trans == TR_SEQ_ONE_WAY$ **then**
  {Definition 4.2.1, Satz 4.2.5}
  if $trans.GetTimimgCond() == NULL$ **then**
    $\hat{\Psi} = s_1$
    $\hat{\Phi} = \exists(s_1\, \mathcal{U}\ s_2)$
  else
    int $val = trans.GetTimimgCond().Bound$
    {Definition 4.2.1, Satz 4.2.3}
    if $trans.GetTimimgCond().Operator.Equals(\text{“} > \text{“})$ **then**
      $\hat{\Psi} = s_1$
      $\hat{\Phi} = ((\exists\Box[0, val]\ s_1) \wedge (\exists s_1\, \mathcal{U}(val, \infty)\ s_2))$
      {Definition 4.2.1, Satz 4.2.2}
    else if $trans.GetTimimgCond().Operator.Equals(\text{“} \leq \text{“})$ **then**
      $\hat{\Psi} = s_1$
      $\hat{\Phi} = \exists(s_1\, \mathcal{U}[0, val]\ s_2)$
    else
      ...
    end if
  end if
end if
if $trans == TR_SEQ_BROKEN$ **then**
  ...
end if
$trans.SetProcessed()$

Abhängig von der Existenz der einzelnen Teilschrittformeln ergibt sich die Gesamtformel anhand der Funktion `CombineSubformulas` (Algorithmus 6). Im ersten Teil der Funktion wird aus allen Teilschrittformeln der `InterTimelines` eine Teilschrittformel über alle Zeitstrahle erstellt. Dafür wird das Speicherobjekt `stepFunktion` durchlaufen und die einzelnen Bestandteile der Formel stückweise nach den Defintionen in Abschnitt 4.5 kombiniert. Das Objekt `tctlFormula` ergänzt die bestehende Gesamtformel um die neuen Bestandteile und beachtet dabei, an welcher Stelle der Formel die neue Teilschrittformel eingefügt werden muss.

Algorithmus 6 CombineSubformulas (Definition 4.5.3)

for all stepFunction sf **do**
  $\hat{\Psi} = \hat{\Psi} \wedge sf.First()$
  $\hat{\Phi^p} = \hat{\Phi^p} \wedge sf.Second()$
  $\hat{\Phi^s} = \hat{\Phi^s} \wedge sf.Third()$
end for
if $\hat{\Psi}$ && $\hat{\Phi^p}$ && Φ_3 **then**
  $tctlFormula.Add(\hat{\Psi} \rightarrow (\hat{\Phi^p} \rightarrow \hat{\Phi^s}))$
else if $\hat{\Psi}$ && $!\hat{\Phi^p}$ && $\hat{\Phi^s}$ **then**
  $tctlFormula.Add(\hat{\Psi} \rightarrow \hat{\Phi^s})$
else if $!\hat{\Psi}$ && $\hat{\Phi^p}$ && $!\hat{\Phi^s}$ **then**
  $tctlFormula.Add(\hat{\Psi} \rightarrow \hat{\Phi^p})$
else if $\hat{\Psi}$ && $!\hat{\Phi^p}$ && $!\hat{\Phi^s}$ **then**
  $tctlFormula.Add(\hat{\Psi})$
else if $!\hat{\Psi}$ && $!\hat{\Phi^p}$ && $!\hat{\Phi^s}$ **then**
  return
else
  {All other cases lead to an error and stop the program.}
  return err
end if

Danksagung

Professor Schweiggert gilt mein größter Dank. Er hat mich zu dieser Arbeit ermutigt, mich jederzeit mit aller Kraft unterstützt und er hat das richtige Maß an Beständigkeit gefördert. Professor Schweiggert ließ mich an seinem fachlichen und persönlichen Erfahrungsschatz teilhaben. Er war mir stets Mentor und Vorbild bei wissenschaftlichen und beruflichen Entscheidungen. Ich verdanke ihm viel.

Ich danke ebenfalls Prof. Partsch für sein Interesse und seine konstruktive Kritik an meiner Arbeit. Er hat meine Arbeit bereichert und vorangetrieben. Meine Vorgesetzten und Kollegen der Daimler AG gaben mir die Freiheit, wissenschaftlich zu arbeiten und boten mir durch die Teilnahme an Konferenzen und Publikationen die Möglichkeit, meine Arbeit anderen zu präsentieren und sie mit ihnen zu diskutieren.

Meinen Freunden danke ich für ihre Rücksicht und ihr Verständnis. Die Promotionszeit verlangt nicht allein vom Promovierenden ein hohes Maß an Beharrlichkeit und Geduld.

Nicht zuletzt danke ich meiner Familie, die mich in jeglicher Hinsicht unterstützt und immer an mich geglaubt hat. Ohne sie wäre diese Arbeit für mich nicht möglich gewesen.

Vielen Dank!

Abbildungsverzeichnis

1.1 Das Automobil und Wechselwirkungen mit seiner Umwelt . . . 2
1.2 Realität - Modell - Funktionsentwicklung . . . 2
1.3 (In Anlehnung an [Par10]) Von der Realität zum System under Test (SUT) . . . 3
1.4 Der Entwicklungsprozess nach dem V-Modell . . . 5
1.5 Die Spezifikation kombiniert mit dem Testmodell ermöglicht den anforderungsbezogenen Test . . . 6

2.1 Testprozess beim Modellbasierten Testen . . . 18
2.2 Hierarchieebenen innerhalb der Außenlichtfunktionalität . . . 25
2.3 Zustandsautomat Entprellung . . . 26
2.4 Zustandsautomat der Blinkerhebelbedienung . . . 26
2.5 Zustandsautomat Richtungsblinken . . . 28
2.6 Zustandsautomat Warnblinken . . . 29

3.1 Kripke Graph des Systems Licht . . . 33
3.2 Berechnungsbaum ([CGP00], S. 28) . . . 34
3.3 Graph des Timed Automaten eines zeitlich eingeschränkten Lichtschalters . . . 38
3.4 Aufbau eines Testsystems im Rahmen der modellbasierten Testfallgenerierung . . . 41
3.5 Syntaxbaum einer CTL Formel . . . 50
3.6 Graph einer Kripke-Struktur und Syntaxbaum einer CTL-Formel . . . 51
3.7 Transitionssystem mit zwei Zuständen und drei Transitionen . . . 53
3.8 Basic MSC . . . 59
3.9 Generisches Sequenzdiagramm Blinken . . . 60
3.10 Universelles und existentielles LSC [DH01] . . . 61
3.11 Darstellung von Pre-Charts und „möglichen“ Bedingungen in LSCs [DH01] . . . 61
3.12 VTS Szenario [BKO05] . . . 62
3.13 Symbolic Timing und Constraint Diagram . . . 63

4.1 Pfad durch den Berechnungsbaum . . . 65
4.2 Systemverhalten im Systemmodell . . . 66
4.3 Parallele Zustandsautomaten . . . 68
4.4 Entscheidungstabelle . . . 69
4.5 Elemente der graphischen Notation . . . 70
4.6 Beispiel der graphischen Notation . . . 71
4.7 Zustandswechsel zur Erfüllung der Formel $p \longrightarrow_p q$ . . . 88
4.8 Zustandswechsel zur Erfüllung der Formel $l_1 \longrightarrow_s l_2 \longrightarrow_s l_3$ . . . 95
4.9 CSC mit paralleler und sequentieller Relation . . . 97
4.10 Zwei sequentielle und zwei parallele Relationen . . . 99
4.11 Kombination zweier Teilschritte . . . 100
4.12 CSC mit ausgehenden Mehrfachrelationen . . . 100
4.13 CSC mit ein- und ausgehenden Mehrfachrelationen . . . 101
4.14 CSC mit gekreuzten Mehrfachrelationen . . . 102
4.15 CSC mit positionsübergreifenden Mehrfachrelationen . . . 103

4.16 CSC mit 3 Zeitstrahlen 104

5.1 Computation Sequence Charts im modellbasierten Testprozess 110
5.2 CSC 111
5.3 Computation Sequence Charts in der Testfallgenerierung 112
5.4 Zustandswechsel zur Erfüllung der Formel $l_1 \overset{<300}{\longrightarrow} l_2 \overset{\geq 500}{\longrightarrow} l_3$ 114
5.5 Zustandsautomat, CSC und Syntaxbaum 115
5.6 Syntaxbaum von Φ 118
5.7 CSCs der Anforderungen aus Tabelle 2.2 123

A.1 TCTL-Library Klassendiagramm 130

Tabellenverzeichnis

2.1 Klassifikation modellbasierter Testverfahren 19
2.2 Auszug Systemlastenheft Blinkfunktionalität 24

3.1 Wahrheitstafel der Boole-Funktion Δ 53

Abkürzungsverzeichnis

BDD	Binary Decision Diagram
BMC	Bounded Model-Checking
CSC	Computation Sequence Chart
CTL	Branching-time Logic
HiL	Hardware-in-the-Loop
HMSC	Highlevel Message Sequence Chart
LSC	Live Sequence Chart
LTL	Linear Time Logic
MSC	Message Sequence Chart
SAT	Boolean Satisfiability (problem)
SC	Statechart
SMT	Satisfiability Modulo Theories
TA	Timed Automata
TCTL	Timed Branching-time Logic
VTS	Visual Timed event Scenarios

Literaturverzeichnis

[AADO00] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt. Evaluation of three specification-based testing criteria. In *Proc. 6th international conference on engineering of complex computer systems*, pages 179–187, 2000.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science LICS '90*, pages 414–425, 1990.

[ADX01] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *Proc. 7th IEEE International Conference on Engineering of Complex Computer Systems*, pages 212–221, 2001.

[AH92] R. Alur and T. A. Henzinger. *Logics and models of real time: A survey*, volume 600 of *Lecture Notes in Computer Science*, chapter Real-Time: Theory in Practice, pages 74–106. Springer, Springer Berlin / Heidelberg, 1992.

[Alu98] R. Alur. Timed automata. In *Summer School on Verification of Digital and Hybrid Systems*. NATO-ASI, 1998.

[Alu99] Rajeev Alur. *Timed Automata*, volume 1633 of *Lecture Notes in Computer Science*, chapter Computer-Aided Verification, pages 8–22. Springer, Springer Berlin / Heidelberg, 1999.

[APP95] R Alur, D Peled, and W. Penczek. Model-checking of causality properties. In *Logic in Computer Science, 1995. LICS '95. Proceedings., Tenth Annual IEEE Symposium on*, pages 90 – 100, San Diego, CA, USA, 1995. AT&T Bell Labs., Murray Hill, NJ.

[ARNRSG06] N. Aizenbud-Reshef, B.T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[Ass90] IEEE Standards Association. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standards Association, 1990.

[BFPT06] B. Badban, M. Fränzle, J. Peleska, and T. Teige. Test automation for hybrid systems. In *Proc. 3rd International Workshop on Software Quality Assurance (SOQUA 2006)*, Portland, OR, USA., November 6 2006. Extended version of the submission published in Neelam Gupta, Yves Ledru, Johannes Mayer (eds.), co-located with the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (ACM SIGSOFT 2006 / FSE 14).

[BHJP05] J. Blom, A. Hessel, B. Jonsson, and P. Petterson. Specifying and generating test cases using observer automata. In *Proc. 4th International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lectures Notes in Computer Science*, pages 125 – 139. Springer Verlag, 2005.

[BJL+05] F. Bouquet, E. Jaffual, B. Legeard, F. Peureux, and M. Utting. Requirements traceability in automated test generation - application to smart card software validation. In *Proc. 1st international workshop on advances in model-based testing*, pages 1–7, 2005.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKO05] V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. In *Proc. IEEE Transactions on Software Engineering*, volume 31, pages 1028 – 1041, Piscataway, NJ, USA, December 2005. IEEE Press.

[BR05] M. Broy and A. Rausch. Das neue V-Modell XT. *Informatik-Spektrum*, 28(3):220–229, Juni 2005.

[Bro79] *Der grosse Brockhaus*. F.A. Brockhaus, 1979.

[BW95] A. J. C. Bik and H. A. G. Wijshoff. Implementation of fourier-motzkin elimination. In *1st Annual Conference: Advanced School for Computing and Imaging*, pages 377–386, 1995.

[BY04] J. Bengtsson and W. Yi. *Timed automata: Semantics, algorithms and tools*, volume 3098 of *Lecture Notes in Computer Science*, chapter Lectures on Concurrency and Petri Nets, Advances in Petri Nets (ACPN'03), pages 87–124. Springer Verlag, Springer Berlin / Heidelberg, 2004.

[CBRZ01] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*. Kluwer Academic Publishers, 2001.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*, volume Second Printing. The MIT Press, 2000.

[CKCB02] S. M. Cho, H. H. Kim, S. D. Cha, and D. H. Bae. A semantics of sequence diagrams. In *Information Processing Letters*, volume 84, pages 125–130. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 2002.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Draft automated software testing using model-checking. Proceedings of the SPIN workshop, WVU technical report NASA-IVV-96-022, 1996.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626 – 643, December 1996.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[Dan03] W. Dangelmaier. *Produktion und Information*. Springer Verlag, 2003.

[DGG04] A. Denise, M. C. Gaudel, and S. D. Gouraud. A generic method for statistical testing. In *Proc. 15th International Symposium on Software Reliability Engineering ISSRE 2004*, pages 25–34, 2–5 Nov. 2004.

[DH01] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DO03] H. Dierks and E.-R. Olderog. Temporale Spezifikationslogiken. *at-Automatisierungstechnik*, 51:A1–A4, Februar 2003.

[EG02] A. Egyed and P. Gruenbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Proc. 17th IEEE International Conference on Automated Software Engineering ASE 2002*, pages 163–171, 2002.

[Eng10] A. Engel. *Verification, validation and testing of engineered systems*. John Wiley and sons, 2010. ISBN 978-0-470-52751-1.

[Fis10] G. Fischer. *Lineare Algebra*, volume 17. Teubner Verlag, Wiesbaden, 2010. google.books.

[For07] P. Forbrig. *Objektorientierte Softwareentwicklung mit UML*, volume 3. Hanser, 2007.

[FS05] M. Friske and H. Schlingloff. Von Use Cases zu Test Cases: Eine systematische Vorgehensweise. TU Braunschweig Report, 2005. Technische Universität Braunschweig.

[fS09] International Organization for Standardization. *Road vehicles - Functional safety (ISO 26262, Draft)*. International Organization for Standardization, 2009.

[FSW08] M. Friske, B.-H. Schlingloff, and S. Weißleder. Composition of model-based test coverage criteria. In U. Nickel H. Giese, M. Huhn and B. Schätz, editors, *MBEES - Model-Based Engineering of Embedded Systems IV*, volume TUBS-SSE of *Dagstuhl-Workshop*. TU Braunschweig Report, April 2008.

[FW07] G. Fraser and F. Wotawa. Using LTL rewriting to improve the performance of model-checker based test-case generation. In *Proc. 3rd international workshop on advances in model-based testing*, pages 64–74, 2007.

[FW08a] G. Fraser and F. Wotawa. Ordering coverage goals in model checker based testing. In *Proc. IEEE International Conference on Software Testing Verification and Validation Workshop ICSTW '08*, pages 31–40, 2008.

[FW08b] G. Fraser and F. Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, 16(2):161–183, Juni 2008.

[GB03] H. Giese and S. Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, 6 2003.

[GF94] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. *Proceedings of the First International Conference on Requirements Engineering*, 1:94–101, April 1994.

[GG93] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63 – 82, June 1993. Wiley.

[GNRS09] H. Götz, M. Nickolaus, T. Roßner, and K. Salomon. *Modellbasiertes Testen*. iX Studie. Heise Zeitschriften Verlag, Hannover, Januar 2009.

[Gom11] H. Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

[Gro11] V. Grosch. Requirements traceability within model-based testing: Applying path fragments and temporal logic. In Seppo Virtanen, editor, *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, volume 2, pages 1 – 21, 2011.

[GS05] C. Gaston and D. Seifert. *Evaluating Coverage based Testing*, volume 3472 of *Lecture Notes in Computer Science*, chapter Model-Based Testing of Reactive Systems, pages 293–322. Springer, Springer Berlin / Heidelberg, 2005.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[HB01] W. Hesse and H. v. Braun. Wo kommen die Objekte her? Ontologisch-erkenntnistheoretische Zugänge zum Objektbegriff. In K. Bauknecht, W. Brauer, and Th. Mück, editors, *Workshop „Erkenntnistheorie-Semiotik-Ontologie (ESO): Die Bedeutung philosophischer Disziplinen für die Softwaretechnik“*, volume 157 of *Informatik 2001 - Tagungsband der GI/OCG-Jahrestagung*, pages 776 – 781. Universität Wien, Österr. Computer-Gesellschaft, 2001.

[Hen80] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januar 1980.

[Hen90] T. Henzinger. Half-order modal logic: how to prove real-time properties. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 281–296, New York, NY, USA, 1990. ACM.

[Heu09] H. Heuser. *Lehrbuch der Analysis Teil 1*, volume 17. Teubner Verlag, Wiesbaden, November 2009. google.books.

[HN96] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[Hoe99] J. Hoenicke. Graphische Spezifikationssprachen: Der Zusammenhang zwischen Constraint Diagrams und real-time symbolic timing diagrams. Master's thesis, Universität Oldenburg, June 1999.

[Hra93] H. Hrachovec. Formale Semantik im Verhältnis zur Erkenntnistheorie. Ein Blickwechsel. *Zeitschrift für philospohische Forschung*, 47:165 – 183, 1993.

[HT04] D. Harel and P.S. Thiagarajan. *Message Sequence Charts*, chapter UML for Real. Design of Embedded Real-Time Systems., pages 77–105. Springer Verlag, 2004.

[Kle00] C. Kleuker. *Constraint Diagrams*. Universität Oldenburg, 2000.

[KPK+10] S. Kollmann, V. Pollex, K. Kempf, F. Slomka, M. Traub, T. Bone, and J. Becker. Comparative application of real-time verification methods to an automotive architecture. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, November 2010.

[Lig09] P. Liggesmeyer. *Software-Qualität*, volume 2. Spektrum Akademischer Verlag, 2009.

[Lot10] W. Lotter. Einfach mehr. *brand eins*, 06/2010, 2010.

[Lun06] J. Lunze. *Ereignisdiskrete Systeme: Modellierung und Analyse dynamischer Systeme mit Automaten, Markovketten und Petrinetzen*. Oldenbourg Wissenschaftsverlag GmbH, 2006.

[Mor02] M. T. Morales. *Systemtheorie, Diskurstheorie und das Recht der Transzendentalphilosophie : Kant - Luhmann - Habermas*. Number 324 in Epistemata / Reihe Philosophie. Königshausen und Neumann GmbH, Würzburg, 2002. google.books.

[MP05] A. Muscholl and D. Peled. *Deciding Properties of Message Sequence Charts*, volume 3466 of *Lecture Notes in Computer Science*, chapter Scenarios: Models, Transformations and Tools, pages 43 – 65. Springer, Springer Berlin / Heidelberg, 2005.

[NARZ05] L. Naslavsky, T. A. Alspaugh, D. J. Richardson, and H. Ziv. Using scenarios to support traceability. In *Proc. 3rd international workshop on traceability in emerging forms of software engineering*, pages 25–30, 2005.

[NZR07] L. Naslavsky, H. Ziv, and D. J. Richardson. Towards traceability of model-based testing artifacts. In *Proc. 3rd international workshop on advances in model-based testing*, pages 105–114, 2007.

[OXL99] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proc. 5th IEEE International Conference on Engineering of Complex Computer Systems ICECCS '99*, pages 119–129, 1999.

[Par10] H. Partsch. *Requirements-Engineering systematisch*. Springer, 2010.

[Pel09] J. Peleska. Test automation problems and their reduction to bounded model checking. Department of Mathematics and Computer Science, University of Bremen, 2009.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46 – 57. IEEE Computer Society Press, 1977.

[Poh10] K. Pohl. *Requirements Engineering. Fundamentals, Principles, and Techniques*. Springer Verlag, Berlin, 2010.

[Pre03] A. Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. Dissertation, Technische Universität München, März 2003.

[PW07] R. Pitschinetz and J. Wegener. Software cte/xl. www.systematic-testing.com, 2007.

[PZck] J. Peleska and C. Zahlten. Industrial-strength model-based testing of embedded real-time systems. (in Druck).

[Rec06] P. Rechenberg. *Informatik-Handbuch*. Hanser, 2006.

[Rei85] W. Reisig. *Systementwurf mit Netzen*. Springer-Verlag, 1985.

[RH01] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. 8th annual IEEE international conference and Workshop on the engineering of computer based systems*, pages 83–91, 2001.

[RH03] S. Rayadurgam and M. P. E. Heimdahl. Generating mc/dc adequate test sequences through model checking. In *Proc. 28th Annual IEEE/NASA Software Engineering Workshop*, pages 91–96, December 2003.

[RHQ+05] C. Rupp, J. Hahn, S. Queins, M. Jeckle, and B. Zengler. *UML 2 glasklar*, volume 2 of *Java Spektrum*. Hanser, 2005.

[RJ01] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE-J-SE*, 27(1):58–93, Jan. 2001.

[RPWTU84] C. V. Ramamoorthy, A. Prakash, T. Wei-Tek, and Y. Usuda. Software engineering: Problems and perspectives. *Computer*, 17(10):191–209, 1984.

[RT06] S. Ranise and C. Tinelli. Satisfiability modulo theories. In *Trends and controversies: IEEE Magazine on Intelligent Systems*, volume 21, pages 71–81, 2006.

[SAM00] K. Sutinen, L. Almefelt, and J. Malmqvist. Implementation of requirements traceability in system engineering tools. *Product Models 2000*, pages 313–328, November 2000.

[Sch99] P. Schefe. Softwaretechnik und Erkenntnistheorie. *Informatik-Spektrum*, 22:122–135, 1999. 10.1007/s002870050131.

[Sch00a] U. Schöning. Boolsche Funktionen, Logik, Grammatiken und Automaten. Vorlesungsskript, 2000.

[Sch00b] U. Schöning. *Logik für Informatiker*, volume 5. Spektrum Akademischer Verlag, 2000.

[Sch01] R. Schloer. *Symbolic Timing Diagrams: A visual formalism for model verification*. Phd thesis, Universität Oldenburg, March 2001.

[Sch10a] F. Schweiggert. Anforderungsmanagement. Lehrgang bei NC Software AG Kassel, TAE, 2010.

[Sch10b] F. Schweiggert. Software-/System-Entwurf. Lehrgang bei NC Software AG Kassel, TAE, 2010.

[Sch10c] F. Schweiggert. Validation & Verifikation. Lehrgang bei NC Software AG Kassel, TAE, 2010.

[SM02] F. Schweiggert and I. Melzer. Software Engineering . Universität Ulm, Fakultät Mathematik und Wirtschaftswissenschaften, April 2002.

[SR98] A. Sowmya and S. Ramesh. Extending statecharts with temporal logic. *Software Engineering, IEEE Transactions on*, 24(3):216 –231, mar 1998.

[TSL04] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proc. IEEE International Conference on Information Reuse and Integration IRI 2004*, pages 483–498, 2004.

[UL07] M. Utting and B. Legeard. *Practical Model-Based Testing*. Elsevier Inc., San Francisco, 2007.

[ULP06] M. Utting, B. Legeard, and A. Pretschner. A taxonomy of model-based testing. Working Paper Series, Hamilton, New Zealand, 2006. Dept. of Computer Science, University of Waikato.

[Uni99] International Telecommunication Union. *ITU-T Recommendation Z.120: Formal Description Techniques (FDT) - Message Sequence Chart*. International Telecommunication Union, November 1999.

[UP91] H. Ulrich and G. J. B. Probst. *Anleitung zum ganzheitlichen Denken und Handeln*, volume 3. Pail Haupt Verlag, 1991.

[Ves88] F. Vester. *Leitmotiv vernetztes Denken. Für einen besseren Umgang mit der Welt.*, volume 19. Heyne Sachbuch, 1988.

[Weg01] J. Wegener. *Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen*. Shaker, 2001.

[Wie95] R. Wieringa. An introduction to requirements traceability. Technical report, Vrije Universiteit, Amsterdam, November 1995.

[WJ91] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE-J-SE*, 17(7):703–711, 1991.

[ZSPMK03] A. Zisman, G. Spanoudakis, E. Perez-Miana, and P. Krause. Tracing software requirements artefacts. In *Proc. International Conference on Software Engineering Research and Practise*, volume 1, pages 448 – 455, 2003.