Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion

vorgelegt von
Diplom-Informatiker
Steffen Helke

An der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. H. Ehrig
Berichter: Prof. Dr. S. Jähnichen
Berichter: Prof. Dr. M. Wirsing

Tag der wissenschaftlichen Aussprache: 4.7.2007

Berlin 2008
D 83

Zusammenfassung

In dieser Arbeit wird ein Ansatz zur Verifikation von Statecharts mit unendlichen Daten vorgestellt. Es wird ein Verfahren konzipiert, mit dem die Gültigkeit einer temporallogischen Formel des universalen Fragements der CTL für eine Statecharts-Spezifikation überprüft werden kann. Das entwickelte Verfahren beruht auf einer struktur- und eigenschaftserhaltenden Datenabstraktionstechnik und ist in einem logikbasierten Framework prototypisch umgesetzt. Eine notwendige Voraussetzung zur Umsetzung des Vorhabens ist der Aufbau einer Beweisinfrastruktur in dem interaktiven Theorembeweiser Isabelle/HOL, in der eine Datenabstraktion für Statecharts-Spezifikationen maschinengestützt vorgenommen werden kann.

Zunächst wird in Isabelle/HOL die abstrakte Syntax und eine synchrone Schrittsemantik der Spezifikationssprache Statecharts formalisiert. Grundlage der Formalisierung sind Hierarchische Automaten, die eine strukturelle Zerlegung von Statecharts in Sequentielle Automaten erlauben. Die aus der Literatur bekannte Beschreibung Hierarchischer Automaten wird durch zwei Optimierungen für die Zwecke eines Theorembeweisers adaptiert. Zum einen werden in der Formalisierung Konstruktionsoperatoren eingeführt, mit denen Hierarchische Automaten schrittweise aus ihren sie definierenden Sequentiellen Automaten zusammengesetzt werden können. Zum anderen wird die baumartige Struktur Hierarchischer Automaten durch einen primitiv-rekursiven Datentyp repräsentiert und darauf aufbauend die Syntax und Semantik von Statecharts durch rekursive Konstantendefinitionen beschrieben. Durch dieses Vorgehen ist es möglich, in der Beweisführung auf effiziente Induktionstaktiken zurückzugreifen.

Ein wichtiger Beitrag der Arbeit besteht in der Formalisierung komplexer Datenräume für Statecharts. Hierfür werden partitionierte Datenräume, vollständige und partielle Datenraumbelegungen, sowie totale und partielle Update-Funktionen eingeführt. Mit Hilfe der Definitionen wird das komplizierte Verhalten auf den Datenräumen formuliert. Teil der Beschreibung ist eine Interleaving-Semantik, mit der die Effekte beim konkurrierenden Schreiben synchron ausgeführter Update-Funktionen angegeben werden.

Neben der Beschreibung von Statecharts mit Datenräumen wird in der Arbeit eine Formalisierung der temporalen Logik CTL angegeben, deren Semantik auf Kripkestrukturen basiert. Es wird ein Operator definiert, mit dem aus einem Hierarchischen Automaten eine sein Verhalten definierende Kripkestruktur abgeleitet werden kann. Mit Hilfe des Operators kann ausgedrückt werden, dass eine CTL-Formel in einer Statecharts-Spezifikation erfüllt ist.

Der Hauptbeitrag der Arbeit besteht in der Konzeption einer Abstraktionstheorie, mit der der Datenraum eines Hierarchischer Automaten für eine gegebene Abstraktionsfunktion abstrahiert werden kann. Die Technik basiert auf der Konstruktion von Überapproximationen und ist für das universale Fragment der CTL eigenschaftserhaltend. Das Ergebnis der Abstraktion ist durch einen Hierarchischen Automaten repräsentiert. Das Verfahren ist strukturerhaltend, d.h. die strukturellen Merkmale eines zu abstrahierenden Hierarchischen Automaten bleiben während der Abstraktion erhalten. Die Abstraktion erfolgt in kompositionaler Art und Weise. Zunächst wird ein zu abstrahierender Hierarchischer Automat in die ihn definierenden Sequentiellen Automaten zerlegt und diese separat voneinander abstrahiert. Anschließend werden die abstrahierten Einzelteile zu einem Hierarchischen Automaten zusammengefügt.

Um den praktischen Umgang mit dem Framework zu erleichtern, werden abschließend zwei Taktiken vorgestellt, die im Rahmen der Arbeit konzipiert und in dem Theorembeweiser Isabelle/HOL implementiert worden sind. Die Taktiken nutzen neben Isabelle auch andere Verifikationswerkzeuge, um den Beweisprozess effizienter zu gestalten. Die Tragfähigkeit des Ansatzes wird für eine Fallstudie gezeigt.

Inhaltsverzeichnis

Kapitel 1
Einführung

1.1 Motivation und Problemstellung

Qualitätssicherung ist ein zentraler Bestandteil der Softwareentwicklung. Trotz hoher Sicherheitsanforderungen wird in realen Entwicklungen die Qualitätssicherung nur unzureichend angewendet. Einer der Gründe hierfür ist die ungenügende Werkzeugunterstützung, die zu hohen Kosten für ein Unternehmen führt. Hohe Produktqualität ist aber insbesondere für eingebettete sicherheitskritische Systeme notwendig, da hier das Versagen der eingesetzten Software den Verlust von Menschenleben nach sich ziehen kann.

Eingebettete Systeme unterscheiden sich von anderen Computersystemen dadurch, dass sie direkt auf eine physische Umgebung einwirken. Sie sind heute in allen Bereichen unseres Lebens vorzufinden. Sie kommen zum Beispiel im Flugzeugbau, in der Bahntechnik oder im Automobilbereich vor. Heute werden 98% aller hergestellten Mikroprozessoren in eingebetteten Anwendungen eingesetzt. Ein Automobil kann bis zu 80 verschiedene Steuergeräte beinhalten [Bro03]. Ein großer Anteil der eingebetteten Systeme ist sicherheitskritisch. Um die Qualität der dort eingesetzten Software zu verbessern, sind in den letzten Jahren Versuche unternomen worden, die Standards für Zertifizierungsagenturen um Auflagen zum Einsatz rigoroser Methoden für die Qualitätssicherung zu erweitern [BHP$^+$00, ABG$^+$03, FP06].

Wir stellen in dieser Dissertation Ergebnisse vor, auf Basis derer die Qualität für eingebettete sicherheitskritische Softwaresysteme verbessert werden kann. Wir präsentieren hierfür ein neuartiges Verfahren, mit dem die Spezifikationen derartiger Systeme maschinengestützt verifiziert werden können. In der Praxis wird das Verhalten eingebetteter Systeme häufig mit Hilfe von *Statecharts* beschrieben. Entsprechend unterstützen wir in dieser Dissertation den Statecharts-Formalismus. Um einen Verifikationsprozess auch von komplexen Statecharts-Spezifikationen zu ermöglichen, stellen wir eine Technik vor, mit der das Verhalten einer Spezifikation durch ein effizienter zu verifizierendes Statechart abstrakt interpretiert werden kann. Bevor wir das Ziel dieser Dissertation konkreter fassen, geben wir eine kurze Einführung in den behandelten Formalismus und stellen Probleme vor, die bei der Verifikation von Statecharts-Spezifikationen auftreten.

Statecharts [Har87] bezeichnen eine graphische Spezifikationssprache, die insbesondere zur Beschreibung des Verhaltens reaktiver Systeme eingesetzt wird. Die Steuerungssoftware eingebetteter Systeme ist in der Regel durch reaktives Verhalten charakterisiert. *Reaktive Systeme* [MP91] verarbeiten aperiodisch auftretende Ereignisse ihrer Umgebung, veranlassen entsprechende Aktionen und weisen im Allgemeinen eine unbeschränkte Laufzeit auf. Die erste Variante von Statecharts wurde von David Harel [Har87] vorgeschlagen. Im Wesentlichen erweitert der Statecharts-Formalismus *Mealy Automaten* [HU79] um hierarchische Strukturierungsmechanismen. Durch diese kann eine Spezifikation sehr kompakt dargestellt werden.

Das Einsatzgebiet von Statecharts kann in einem rigorosen Softwareentwicklungszyklus für eingebettete Systeme sehr vielgestaltig sein. Zunächst stellt eine ausgearbeitete Spezifikation eine gute Dokumentation der Software dar. Sie kann aber auch die Grundlage für die automatische Erzeugung von Testfällen sein. Weiterhin können detailliert ausgearbeitete Statecharts-Spezifikationen zur Erzeugung ausführbarer Programmcodes genutzt werden.

Um Statecharts-Spezifikationen für die genannten Bereiche erfolgreich einsetzen zu können, müssen folgende Eigenschaften erfüllt sein. Die Spezifikation muss

1. aus zueinander passenden Teilen aufgebaut sein und

2. alle wichtigen Sicherheitseigenschaften umsetzen.

Da Spezifikationen in realen Anwendungsbeispielen sehr groß werden, ist eine manuelle Überprüfung dieser Eigenschaften nicht realistisch durchführbar oder zumindest sehr fehleranfällig. Aus diesen Gründen ist der Einsatz geeigneter Software zur maschinengestützten Analyse von Spezifikationen notwendig.

Im industriellen Umfeld wurde das *Model-Checking* als die geeignete Verifikationstechnik entdeckt, um Software eingebetteter Systeme zu überprüfen [CGH$^+$93]. Model-Checking ist eine Technik, mit der die Gültigkeit temporallogischer Eigenschaften auf einem Zustandsgraphen vollautomatisch überprüft werden kann. Ist eine Eigenschaft nicht erfüllt, wird ein Gegenbeispiel zurückgegeben. Model-Checking beruht auf einem Verfahren, bei dem die Zustände des Zustandsgraphen explizit oder symbolisch – abhängig von der eingesetzten Technik – aufgezählt werden. Während dieser Aufzählung wird an jedem Zustand die Gültigkeit der zu überprüfenden temporallogischen Eigenschaft bestimmt. Es gibt inzwischen diverse effiziente Implementierungen dieser Technik in Form so genannter Model-Checker [McM93, Hol03, Ros94], die auch kommerziell vertrieben werden. Mit den traditionellen Algorithmen des Model-Checking können zwar unendliche Pfade in einem Zustandsgraphen überprüft werden, jedoch ist das Verfahren konzeptuell auf Zustandsgraphen mit endlich vielen Zuständen beschränkt. Hinzu kommt, dass auch bei Zustandsgraphen mit endlich vielen Zuständen die Anwendbarkeit eingeschränkt sein kann. Das ist immer dann der Fall, wenn zu viele Zustände überprüft werden müssen. Hier terminieren die Algorithmen zwar in endlicher Zeit, die langen Wartezeiten sind aber für einen praktischen Umgang ungeeignet.

Viele graphische Verhaltensbeschreibungen – so auch Statecharts – sind zunächst auf endlich vielen Zuständen definiert. Häufig wird aber die graphische Beschreibung um einen Datenraum ergänzt. Beim Ausführen von Transitionen können Datenvariablen dieses Datenraumes gelesen und geschrieben werden. Im Allgemeinen sind die Datenvariablen auf unendliche Definitionsbereiche deklariert, so dass die Verhaltensbeschreibung der gesamten Spezifikation einen Zustandsgraphen mit unendlich vielen semantischen Zuständen bildet. Zur Überprüfung solcher Zustandsgraphen sind die klassischen Techniken des Model-Checking nicht geeignet. Es gibt bereits erste Versuche, klassische Model-Checker um neuartige Techniken zu erweitern, mit denen auch unendliche Zustandssysteme überprüft werden können [Bry06].

Eine Möglichkeit zur Überprüfung unendlicher Zustandssysteme ist die Kombination von Model-Checking und eigenschaftserhaltenden Abstraktionstechniken. *Eigenschaftserhaltende Abstraktion* ist eine Technik zur Reduktion von Zustandsgraphen. Mit diesem Verfahren kann ein für das Model-Checking zu großes Zustandssystem durch ein kleineres Zustandssystem abstrakt interpretiert werden. In der Regel führt die Reduktion zu einem deutlich kleineren und vor allem endlichen Zustandssystem, so dass darauf mit einem Model-Checker

Analysen zumeist effizient durchgeführt werden können. Weiterhin sind eigenschaftserhaltende Abstraktionen so konstruiert, dass für eine Klasse verifizierbarer Eigenschaften gilt, dass aus ihrer Gültigkeit in den abstrakten Modellen auch ihre Gültigkeit in den konkreten Modellen folgt. Für die Gültigkeit von Eigenschaften, die auf abstrakten Modellen nicht erfüllt sind, gibt es in Bezug auf die konkreten Systeme zwei mögliche Schlussfolgerungen: entweder die Eigenschaften sind in den konkreten Modellen tatsächlich auch verletzt oder aber die Abstraktionen waren zu grob gewählt, so dass die Eigenschaften in den konkreten Modellen tatsächlich gültig sind. Im zweiten Fall sind durch die Abstraktionen so viel Informationen über die konkreten Modelle verloren gegangen, dass ein Nachweis der Eigenschaften in den abstrakten Systemen unmöglich ist.

Um unendliche oder sehr große endliche Zustandssysteme mit Model-Checking überprüfen zu können, werden Abstraktionstechniken bereits heute im industriellen Umfeld eingesetzt [CGP02, SEGW03]. Allerdings kann ein Abstraktionsprozess nicht immer durch entsprechende Softwarewerkzeuge unterstützt werden. Das macht die Anwendung dieser Verfahren sehr fehleranfällig. Gegenstand aktueller Forschung ist es, Abstraktionsprozesse durch den Einsatz von Theorembeweisern sicherer zu gestalten. Im Gegensatz zu Model-Checkern können im Theorembeweisern auch unendliche Zustandsräume erfasst werden, so dass hier durch das Ableiten mathematischer Aussagen die Beziehung zwischen konkreten und abstrakten Modellen genauer untersucht werden kann. Es gibt in der Literatur bereits Arbeiten, die zeigen, wie durch Kombination von Model-Checking und Theorembeweisen der Abstraktionsprozess für unendliche Zustandssysteme qualitativ verbessert werden kann [SS99, LGS$^+$95]. Speziell für den Abstraktionsprozess von Statecharts-Spezifikationen sind aber bisher nur wenige Versuche unternommen worden [TSOR04]. Ein Grund hierfür ist die komplexe Semantik von Statecharts, die eine einfache Anwendung bereits entwickelter Abstraktionskonzepte nicht ohne Weiteres zulässt. Insbesondere existiert bisher kein Verfahren, bei dem durch den Abstraktionsprozess die vorteilhaften Strukturierungsmechanismen von Statecharts nicht verloren gehen. Dies ist aber insbesondere für die Transparenz von automatisch generierten abstrakten Modellen wünschenswert. Wir wollen mit dieser Dissertation einen Beitrag leisten, den beschriebenen Mangel zu beheben. Im folgenden Abschnitt formulieren wir konkrete Ziele, um das Vorhaben dieser Arbeit genauer einzugrenzen.

1.2 Zielsetzung und Lösungsansatz

Hauptziel dieser Dissertation ist es, ein Verfahren zu konzipieren, mit dem Statecharts-Spezifikationen unter Verwendung eigenschafts- und strukturerhaltender Abstraktionstechniken weitestgehend automatisch verifiziert werden können. Insbesondere die Verifikation von Statecharts-Spezifikationen mit unendlichen Datenräumen soll dabei unterstützt werden. Konkret verfolgen wir den Lösungsansatz, Statecharts-Modelle unter Verwendung des Theorembeweisers Isabelle/HOL zu abstrahieren und die so gewonnenen abstrakten Modelle durch den Model-Checker SMV zu verifizieren. Um dieses Ziel zu erreichen, behandeln wir folgende zwei Arbeitspakete.

Im ersten Arbeitspaket wird die abstrakte Syntax und die Semantik von Statecharts in dem Theorembeweiser Isabelle/HOL formalisiert. Ausgangspunkt der Formalisierung ist die Beschreibung von Statecharts mit Hilfe *Hierarchischer Automaten*. Hierarchische Automaten stellen eine in der Literatur bereits etablierte und in verschiedenen Anwendungen erprobte Beschreibung von Statecharts dar. Durch die Verwendung Hierarchischer Automaten erlangt

die Arbeit gegenüber anderen, weniger verbreiteten Beschreibungen von Statecharts, einen stärkeren Praxisbezug. Wir werden ferner dafür Sorge tragen, dass trotz dieser Anforderung eine, im Hinblick auf die Beweisführung in einem Theorembeweiser, möglichst effiziente Formalisierung entsteht. Wir entwickeln hierfür Optimierungen, für die wir nachweisen, dass sie die Eigenschaften der ursprünglich eingesetzten Semantik erhalten. Weiterhin stellt die formale Behandlung von Datenräumen bei der Umsetzung der Formalisierung von Statecharts eine besondere Herausforderung dar. Der Grund hierfür ist, dass in der Literatur nur wenige ausgearbeitete Statecharts-Semantiken existieren, die nichttriviale Datenräume berücksichtigen. In einem letzten Arbeitsschritt werden wir die Formalisierung um eine Theorie für die temporale Logik CTL erweitern. In dieser Theorie ist es möglich, eine Aussage zu formulieren, die die Gültigkeit einer CTL-Formel in Bezug auf eine Statecharts-Spezifikation beschreibt. Alle in diesem Arbeitspaket zu entwickelnden Theorien setzen wir in dem Theorembeweiser Isabelle/HOL um.

Im zweiten Arbeitspaket dieser Dissertation werden wir eine Abstraktionstechnik für Statecharts-Spezifikationen entwerfen. Diese Technik soll dazu dienen, den Datenraum einer Statecharts-Spezifikation zu abstrahieren. Beispielsweise werden wir auf diese Art einen unendlichen Datenraum zu einem endlichen Datenraum abstrahieren, so dass das abstrakte Modell mit Model-Checking verifiziert werden kann. Wir werden mit dieser Technik das Ergebnis der Abstraktion – also das abstrakte Modell – durch einen zur Eingabe strukturell identischen Statechart ausdrücken. Weiterhin untersuchen wir, für welches Fragment der temporalen Logik CTL die Abstraktion eigenschaftserhaltend ist. Um den Abstraktionsprozess möglichst praktikabel zu halten, wird die Abstraktion in kompositionaler Art und Weise vorgenommen. Dies bedeutet, dass der Statechart in einzelne Komponenten zerlegt wird, die unabhängig voneinander abstrahiert werden können. Anschließend werden die so gewonnenen abstrahierten Einzelteile wieder zu einer Statecharts-Spezifikation zusammengesetzt. Die Abstraktionstechnik wird in einer Isabelle/HOL-Theorie auf Basis der im ersten Arbeitspaket entwickelten Statecharts-Formalisierung umgesetzt. Um die praktische Arbeit beim Ableiten von Theoremen über Statecharts-Spezifikationen zu verbessern, werden geeignete Beweistaktiken entwickelt und implementiert. Wir werden hierbei automatische Beweiswerkzeuge außerhalb von Isabelle in den Verifikationsprozess einbinden. Zum einen wird eine Taktik entwickelt, mit der Statecharts-Spezifikationen, die auf einem endlichen Datenraum definiert sind, überprüft werden können. Zum anderen wird eine Taktik entworfen, mit der ein Datenabstraktionsprozess für eine Statecharts-Spezifikation auf Basis boolescher Prädikate außerhalb von Isabelle berechnet werden kann.

Abschließend fassen wir die Ziele aus den beiden Arbeitspaketen zusammen. Das grobe Ziel dieser Dissertation besteht in der Entwicklung eines logikbasierten Frameworks, mit dem Statecharts-Modelle, die auf unendlichen Datenräumen definiert sind, zu großen Teilen automatisch verifiziert werden können. Um das Vorhaben zu realisieren, werden dazu folgende Teilaufgaben gelöst.

1. Entwicklung einer strukturerhaltenden HOL-Theorie für Statecharts mit Hilfe Hierarchischer Automaten, in der Beweise über Statecharts-Spezifikationen mit unendlichen Datenräumen geführt werden können.

2. Entwicklung einer HOL-Theorie für die Temporallogik CTL, die mit der Theorie für Hierarchische Automaten semantisch integriert wird.

3. Entwicklung einer strukturerhaltenden und eigenschaftserhaltenden Datenabstraktions-

theorie für Statecharts, die innerhalb der HOL-Logik auf Basis der bereits entwickelten Statecharts-Formalisierung umgesetzt wird.

4. Anbindung des Model-Checkers SMV über die Orakelschnittstelle von Isabelle/HOL zur effizienten Verifikation Hierarchischer Automaten, die endliche Datenräumen beinhalten. Entwicklung und prototypische Implementierung eines Abstraktionsalgorithmus zum Generieren eigenschaftserhaltender Datenabstraktionen für Hierarchische Automaten.

Nachdem wir die Ziele für diese Arbeit festgelegt und einen ersten Lösungsansatz skizziert haben, stellen wir im folgenden Abschnitt den Aufbau dieser Dissertation vor.

1.3 Aufbau dieser Dissertation

Die Ausarbeitung dieser Dissertation besteht aus zehn Kapiteln. Die Kernkapitel haben wir zu zwei Teilen zusammengefasst. Die Teile korrespondieren zu den beiden Arbeitspaketen, die wir im letzten Abschnitt beschrieben haben. Im Folgenden stellen wir die Struktur der beiden Teile vor.

Der erste Teil umfasst die Kapitel 2 bis 5 und stellt eine HOL-Theorie zur maschinengestützten Analyse von Statecharts-Spezifikationen vor. Kapitel 2 gibt eine Einführung in Statecharts und temporale Logiken. Ferner werden die in dieser Dissertation eingesetzten Softwarewerkzeuge vorgestellt. Kapitel 3 beschreibt die Formalisierung der abstrakten Syntax und Semantik von Statecharts mit Hilfe Hierarchischer Automaten in Isabelle/HOL. Ferner präsentiert dieses Kapitel, wie eine in der Literatur verfügbare Beschreibung von Statecharts für die Anforderungen in einem Theorembeweiser optimiert werden kann. Kapitel 4 führt eine Erweiterung der Statecharts-Formalisierung um Datenräume ein. Kapitel 5 stellt schließlich eine Theorie zur Formalisierung der temporalen Logik CTL vor und führt diese mit der Formalisierung von Statecharts aus Kapitel 3 zusammen.

Die Kapitel 6, 7 und 8 bilden den zweiten Teil dieser Dissertation. Dieser Teil präsentiert eine struktur- und eigenschaftserhaltende Datenabstraktionstheorie für Statecharts. Kapitel 6 stellt hierfür zunächst das entwickelte Abstraktionskonzept vor. Kapitel 7 beschreibt eine Abstraktionstheorie, mit der das in Kapitel 6 vorgestellte Konzept in Isabelle/HOL durch Konstruktionsoperatoren umgesetzt wird. Kapitel 8 gibt an, wie der Abstraktionsprozess für Hierarchische Automaten mit Hilfe geeigneter Taktiken möglichst effizient gestaltet werden kann. Es wird eine Taktik präsentiert, mit der in Isabelle/HOL formalisierte Hierarchische Automaten durch den Model-Checker SMV verifiziert werden können. Ferner stellt Kapitel 8 eine zweite Taktik vor, die einen Abstraktionsalgorithmus in die Werkzeugumgebung einbindet. Dieser Algorithmus errechnet die Datenabstraktion Hierarchischer Automaten auf Basis von Prädikaten. Weiterhin wird für eine Beispielanwendung die Funktionsweise des in dieser Dissertation entwickelten Frameworks vorgestellt.

Kapitel 9 gibt einen Überblick über verwandte Arbeiten. Zum einen werden Arbeiten betrachtet, in denen die maschinengestützte Verifikation von Statecharts mit Hilfe von Model-Checkern oder Theorembeweisern erfolgt. Zum anderen werden Ansätze aus der Literatur vorgestellt, die sich mit eigenschaftserhaltenden Abstraktiontechniken beschäftigen.

Kapitel 10 fasst die Ergebnisse dieser Dissertation zusammen und gibt einen Ausblick auf weiterführende Arbeiten.

Teil I

HOL-Theorie für mechanische Beweise über Statecharts-Spezifikationen

Kapitel 2

Grundlagen

Wir legen in diesem Kapitel die Grundlagen zum besseren Verständnis des Folgenden. Zunächst geben wir eine Einführung in die abstrakte Syntax und Semantik von Statecharts und stellen ein Beispiel für eine Statecharts-Spezifikation vor. Im darauf folgenden Abschnitt beschreiben wir Kripke-Strukturen und Temporale Logiken. Schließlich gehen wir im letzten Abschnitt auf Softwarewerkzeuge ein, die im Rahmen dieser Arbeit für die maschinengestützte Verifikation von Statecharts eingesetzt werden.

2.1 Statecharts

Mit *Statecharts* wird eine graphische Spezifikationssprache bezeichnet, die auf endlichen Zustandsautomaten basiert. Statecharts dienen zur Beschreibung reaktiven Verhaltens, wie es beispielsweise in eingebetteten Systemen vorkommt. Durch Sprachkonzepte zur hierarchischen und parallelen Zustandskomposition kann auch komplexes Verhalten sehr kompakt beschrieben werden. Parallel komponierte Systemteile können über Ereignisse kommunizieren, die über *Broadcast*-Mechanismen verteilt werden. Zusätzlich wird einem Statechart ein *partitionierter Datenraum* zugeordnet, der bei der Ausführung von Transitionen partiell gelesen und geschrieben werden kann.

Der Statecharts-Formalismus wurde bereits in den 1980er Jahren entwickelt [Har87]. In den folgenden Jahren bildeten sich mehrere Varianten des Formalismus heraus, die häufig ähnliche Syntax aber unterschiedliche Semantiken aufweisen. Ein systematischer Vergleich von zwanzig ausgewählten Vertretern wurde von Michael von der Beek vorgenommen [vdB94]. Durch die kommerzielle Entwicklung des Werkzeugs Statemate [HLN+90] konnte sich eine Semantik – die so genannte *Statemate-Semantik*[1] – gegenüber anderen Varianten durchsetzen [HN96]. Wenig später entstand mit der Einführung der UML (*Unified-Modelling-Language*) ein alternativer Statechartsdialekt mit weiter Verbreitung: die so genannten *UML-State-Machines* [HG97, Obj03]. UML-State-Machines wurden entworfen, um ein geeignetes Beschreibungsmittel für das dynamische Verhalten von objektorientierten Klassen zur Verfügung zu haben. Wir orientieren uns bei der Entwicklung der in dieser Arbeit vorgestellten Ergebnisse überwiegend an der Statemate-Semantik. Viele der Ergebnisse sind aber allgemeingültig und können daher mit geringem Aufwand auf UML-State-Machines übertragen werden. Zur sprachlichen Vereinfachung werden wir in dieser Arbeit UML-State-Machines auch als *UML-Statecharts* und Statecharts, die auf einer Statemate-Semantik beruhen, auch als *Statemate-Statecharts* bezeichnen.

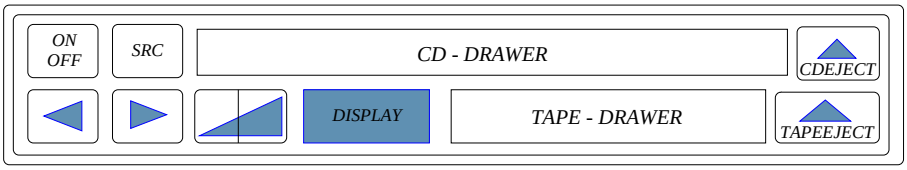
[1] Diese Semantik wurde für das Werkzeug Statemate entwickelt und trägt deshalb diesen Namen.

Dieser Abschnitt ist folgendermaßen aufgebaut. Zunächst stellen wir ein Beispiel für eine Statecharts-Spezifikation vor. Es handelt sich bei dem Beispiel um die Steuerung einer Musikanlage, wie sie in einem Automobil eingebaut sein könnte. Dieses Beispiel haben wir bewusst einfach gehalten. Es wird im Folgenden eingesetzt, um die Statecharts-Formalisierung intuitiv verständlich zu halten. Danach geben wir einen Überblick über die abstrakte Syntax von Statecharts und identifizieren die in dieser Arbeit behandelte Untermenge. Anschließend zeigen wir, wie Statecharts mit Hilfe *Erweiterter Hierarchischer Automaten* beschrieben werden können. Abschließend führen wir die Statemate-Semantik für die von uns betrachtete Untermenge ein.

2.1.1 Beispielspezifikation: Musikanlage im Automobil

In diesem Abschnitt wird ein kleines Beispiel zur Steuerung einer Musikanlage in einem Automobil präsentiert[2]. Die Steuerung dieser Musikanlage werden wir durch eine Statecharts-Spezifikation beschreiben. Sollte der Leser gar keine Kenntnisse über Statecharts besitzen, raten wir dazu, Teile aus Abschnitt 2.1.2 beim Lesen vorzuziehen, da wir erst in diesem Abschnitt die Syntax und Semantik von Statecharts systematisch einführen.

Die folgende Abbildung veranschaulicht schematisch die Benutzerschnittstelle der Musikanlage.



Das Bedienungsfeld des Gerätes ist möglichst minimal gehalten. In der folgenden Spezifikation betrachten wir nur eine Teilfunktionalität der Steuerung. Insbesondere wird die Regulierung der Lautstärke, die Darstellung interner Informationen auf dem digitalen Bildschirm und die Funktionalitäten zur Speicherung von Radiosendern nicht betrachtet.

Das System besitzt drei Signalquellen: ein Radio (*Tuner*), einen Kassetten-Spieler (*Tape-Deck*) und einen CD-Spieler (*CD-Player*). Das Gerät verfügt über folgende Schalter.

- *ON/OFF*: An- und Ausschalten des Gerätes
- *SRC*: Änderung der Signalquelle
- ▶: Änderung eines Radiokanals, Vorwärtsspulen im Kassetten-Spieler oder Anwählen des nächsten Titels auf einer CD
- ◀: Änderung eines Radiokanals, Rückwärtsspulen im Kassetten-Spieler oder Anwählen des vorherigen Titels auf einer CD

Abbildung 2.1 zeigt eine Statecharts-Spezifikation dieser Anwendung.

Die Spezifikation besteht auf der obersten Abstraktionsstufe aus drei parallel komponierten Zuständen. Mit Hilfe der im oberen Teil von Abbildung 2.1 dargestellten Zustände *CDPlayer* und *TapeDeck* wird modelliert, ob eine CD oder eine Kassette im Gerät vorhanden

[2]Dieses Beispiel wurde als Übungsaufgabe für Studierende in Zusammenarbeit mit dem Kollegen André Nordwig entwickelt.

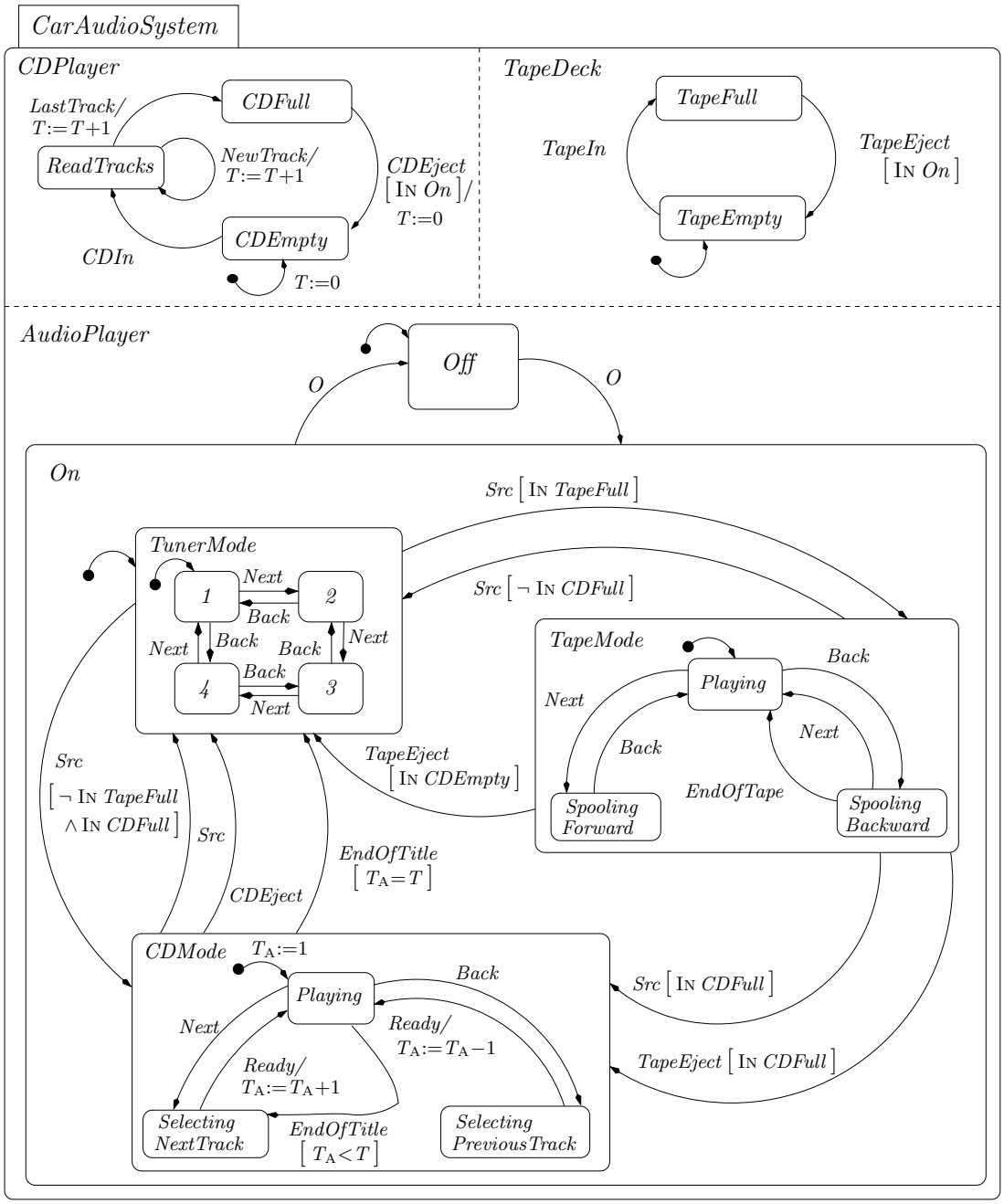


Abbildung 2.1: Statecharts-Spezifikation eines *Car-Audio-System*

ist oder nicht. Zusätzlich wird im Zustand *CDPlayer* beim Einlegen einer CD die Anzahl der spielbaren Tracks bestimmt und das Ergebnis in der Datenvariable T abgelegt. Hierbei wird das Auffinden eines neuen Tracks durch das Ereignis *NewTrack* angezeigt. Beim Auffinden des letzten Tracks auf der CD wird das Ereignis *LastTrack* empfangen. Der Definitionsbereich der Datenvariable T ist vom Typ Integer und unbeschränkt.

Der im Vergleich mit den Zuständen *CDPlayer* und *TapeDeck* deutlich komplexere Zustand *AudioPlayer* beschreibt das zentrale Verhalten der Steuerung, auf das im Weiteren detailliert eingegangen wird.

Initial wird mit dem Systemstart – repräsentiert durch das Ereignis O – der *TunerMode* aktiviert. Es kann immer nur eine Signalquelle zu einem Zeitpunkt aktiv sein. Durch das Betätigen des *SRC*-Schalters kann die aktivierte Signalquelle in einer festgelegten Reihenfolge geändert werden. Wenn beispielsweise keine CD im CD-Spieler vorhanden ist, wechselt die Steuerung vom *TapeMode* in den *TunerMode*. Ist hingegen eine CD im CD-Spieler vorhanden, wechselt die Steuerung vom *TapeMode* in den *CDMode*. Das Einfügen eines neuen Mediums (CD oder Kassette) führt ohne das Betätigen des *SRC*-Schalters zu keiner Änderung der aktivierten Signalquelle.

Der *TunerMode* besteht aus vier Kanälen, die in zyklischer Reihenfolge durch Betätigen der Tasten ▶ und ◀ geändert werden können. Das Betätigen der Tasten ▶ und ◀ wird durch die Ereignisse *Next* und *Back* in der Statecharts-Spezifikation von Abbildung 2.1 repräsentiert.

Wird der *TapeMode* aktiviert, beginnt der Kassetten-Spieler mit dem Abspielen der eingelegten Kassette. Durch Betätigen der Tasten ▶ und ◀ kann die Kassette vorwärts und rückwärts gespult werden. Durch das Betätigen der jeweils anderen Taste kann das Vorwärts- bzw. das Rückwärtsspulen gestoppt werden. Wenn beim Rückwärtsspulen der Anfang des Kassettenbandes erreicht wurde, beginnt der Kassetten-Spieler mit dem Abspielen der eingelegten Kassette.

Wenn der *CDMode* aktiviert wurde, beginnt der CD-Spieler mit dem Abspielen der eingelegten CD. In der Datenvariable $T_{\mathtt{A}}$ wird die Nummer des gerade aktiven Tracks abgelegt. Analog zu der Datenvariable T ist $T_{\mathtt{A}}$ als Integer-Variable deklariert. Nach dem Abspielen eines Titels auf der CD wechselt der CD-Spieler automatisch zum nächsten Titel und inkrementiert $T_{\mathtt{A}}$ entsprechend. Für den Fall, dass der letzte Track der CD abgespielt wurde, wird der *CDMode* verlassen und der *TunerMode* aktiviert. Durch Betätigen der Tasten ▶ und ◀ kann man zum nächsten oder vorherigen Titel auf der CD wechseln. Wenn ein neuer Titel selektiert wurde, muss das System die Position des Lesekopfes anpassen. Der erfolgreiche Abschluss dieses Vorgangs wird durch das Ereignis *Ready* modelliert.

Im folgenden Abschnitt werden wir die abstrakte Syntax von Statecharts anhand des eingeführten Beispiels erläutern.

2.1.2 Abstrakte Syntax

Die Syntax von Statecharts besteht im Wesentlichen aus Zuständen, die durch markierte Transitionen miteinander verknüpft sind. Zustände sind durch an den Ecken abgerundete Rechtecke und Transitionen durch gerichtete Kanten zwischen zwei Zuständen visualisiert.

Zustände

Zustände können in andere Zustände eingebettet werden. Zum Beispiel ist in der Statecharts-Spezifikation aus Abbildung 2.1 der Zustand *CDFull* in den Zustand *CDPlayer* eingebettet.

Wir sprechen dann davon, dass der Zustand *CDPlayer* durch den Zustand *CDFull direkt verfeinert* wird. Eingebettete Zustände können wiederum durch andere Zustände verfeinert werden. So ist in Abbildung 2.1 zum Beispiel der Zustand *CDMode* in den Zustand *On* eingebettet und *CDMode* selbst wiederum durch den Zustand *Playing* verfeinert. Wir sprechen auch davon, dass der Zustand *On* durch den Zustand *Playing indirekt* oder *transitiv verfeinert* wird. Weiterhin bezeichnen wir *CDMode* als *direkten Unterzustand* von *On* und *On* als *direkten Vorgänger* von *CDMode*. Der Vorgängerzustand eines eingebetteten Zustands ist eindeutig. Abhängig davon, ob und wie ein Zustand verfeinert ist, können drei Zustandsarten unterschieden werden.

Basis-*Zustände* bezeichnen Zustände, die durch keine weiteren Unterzustände verfeinert sind. In Abbildung 2.1 sind zum Beispiel *CDFull* und *Playing* Basis-Zustände.

Oder-*Zustände* bezeichnen Zustände, die durch einen oder mehrere Zustände direkt verfeinert werden. Dabei gilt, dass von den direkten Unterzuständen immer nur einer zu einem Zeitpunkt aktiviert werden kann. In Abbildung 2.1 ist *AudioPlayer* zum Beispiel ein Oder-Zustand, da er durch die drei Zustände *TunerMode*, *TapeMode* und *CDMode* direkt verfeinert wird und diese Zustände nicht gleichzeitig aktiviert werden können.

Und-*Zustände* bezeichnen Zustände, die durch mindestens zwei Zustände direkt verfeinert werden. Dabei gilt, dass die direkten Unterzustände Oder-Zustände sind und diese beim Betreten des Und-Zustands gleichzeitig aktiviert werden. In Abbildung 2.1 ist der Zustand *CarAudioSystem* ein Und-Zustand, da er durch die Oder-Zustände *CDPlayer*, *TapeDeck* und *AudioPlayer* direkt verfeinert wird. Die gestrichelte Linie zwischen den Unterzuständen gibt an, dass diese Zustände beim Betreten von *CarAudioSystem* gleichzeitig aktiviert werden. Wir sprechen auch davon, dass die Zustände *CDPlayer*, *TapeDeck* und *AudioPlayer parallel komponiert* sind.

Jedes Statechart besitzt einen eindeutigen *Wurzelzustand*, der keine Vorgängerzustände besitzt. In Abbildung 2.1 ist dieser Zustand mit *CarAudioSystem* bezeichnet. Oder-Zustände müssen einen direkten Unterzustand als *initialen Zustand* auszeichnen. Der initiale Zustand des Oder-Zustands *AudioPlayer* ist zum Beispiel *Off*. Der initiale Zustand wird auch als *Default-Zustand* bezeichnet.

Partitionierte Datenräume

Einer Statecharts-Spezifikation kann ein partitionierter Datenraum zugeordnet sein. Die Partitionen dieses Datenraumes können durch Transitionen gelesen und partiell geschrieben werden. Partitionen werden mit Variablennamen referenziert und sind typisiert. Durch parallel komponierte Zustände können Partitionen von Transitionen konkurrierend geschrieben werden. Dabei entstehende Konflikte werden mit einer Interleaving-Semantik aufgelöst (vgl. Abschnitt 2.1.4). Das Beispiel in Abbildung 2.1 enthält zwei Datenpartitionen, die jeweils aus einer Integer-Variable bestehen.

Transitionen

Eine Transition beschreibt einen gerichteten Zustandsübergang, der von einem *Startzustand* ausgeht und in einen *Zielzustand* übergeht. In Abbildung 2.1 ist zum Beispiel eine Transition beschrieben, die in dem Startzustand *TunerMode* beginnt und in den Zielzustand

TapeMode übergeht. Weiterhin ist diese Transition mit einem so genannten *Label* markiert. Ein Label legt einerseits Bedingungen fest, unter denen die Transition *schalten* kann und beschreibt andererseits die Effekte der Transition. Die Transition zwischen *TunerMode* und *TapeMode* kann nur unter der Bedingung schalten, dass der Zustand *TapeFull* aktiviert ist und das Ereignis *Src* anliegt. Man beachte, dass ein Ereignis dann anliegt, wenn es im vorhergehenden Zustandsübergang erzeugt oder von der Umgebung empfangen wurde (vgl. Abschnitt 2.1.4). Das Label der betrachteten Transition definiert keine Effekte. Allgemein ist die Struktur eines Labels durch drei Komponenten festgelegt.

$$\mathit{Trigger}\,[\,\mathit{Condition}\,]/\mathit{Action}$$

Jede dieser Komponenten ist optional. Die erste Komponente *Trigger* repräsentiert ein Ereignis, das die Transition auslöst. Die zweite Komponente *Condition* definiert ein Prädikat, das zusätzlich zum Anliegen von *Trigger* erfüllt sein muss, damit die Transition schalten kann. Das Prädikat *Condition* kann folgende atomare Bedingungen definieren, die negiert ($\neg$) oder durch Konjunktion ($\wedge$) bzw. Disjunktion ($\vee$) miteinander verknüpft werden können.

- Das Anliegen eines Ereignisses e (En e).
- Das Vorkommen eines aktiven Zustands s (In s).
- Die Gültigkeit eines Prädikats auf dem zugrunde liegenden Datenraum.

Mit dem Aktionsteil *Action* eines Labels werden die Effekte der Transition beschrieben. Folgende Effekte können angegeben werden.

- Die Erzeugung von Ereignissen.
- Die Änderung der Wertebelegung des zugrunde liegenden Datenraums.

Die Änderung der Wertebelegung des zugrunde liegenden Datenraums wird durch eine so genannte Update-Funktion beschrieben.

Einschränkungen

In dieser Arbeit haben wir uns auf einen Teil der Ausdruckssprache des Statecharts-Formalismus [Har87] eingeschränkt. Insbesondere unterstützen wir folgende syntaktische Ausdrucksmittel nicht.

Interlevel-Transitionen: Diese speziellen Transitionen können zur Modellierung von Zustandsübergängen über Zustandsgrenzen hinweg eingesetzt werden. Mathematisch präziser sind dies Transitionen, in denen der direkte Vorgängerzustand des Startzustands vom direkten Vorgängerzustand des Zielzustands verschieden ist. Abbildung 2.2 zeigt einen Ausschnitt aus der in Abschnitt 2.1.1 vorgestellten Spezifikation in leicht modifizierter Form. Hier wurde unter anderem eine Interlevel-Transition hinzugefügt, die vom Startzustand *TapeMode* zum Zielzustand *Off* führt. Diese Transition modelliert, dass sich nach dem vollständigen Abspielen einer Kassette – angezeigt durch das Ereignis *EndOfTape* – die Musikanlage abschaltet. Interlevel-Transitionen sind ein mächtiges Konzept. Ähnlich zu *Goto*-Anweisungen in Programmiersprachen erschweren sie aber die Definition von modularen und strukturerhaltenden Modellen zur allgemeinen Beschreibung einer Spezifikationssprache. Unsere Arbeit basiert

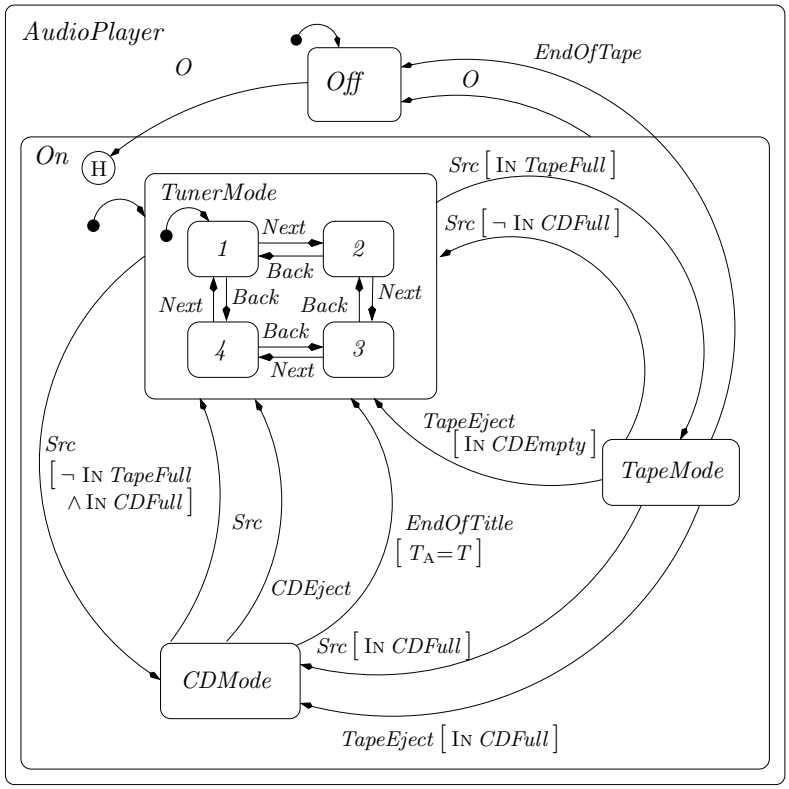


Abbildung 2.2: History und Interlevel-Transition in einem *Car-Audio-System*-Modell

auf einer Beschreibung von Statecharts durch Hierarchische Automaten, in denen Interlevel-Transitionen nicht modelliert werden können. Mit der Einführung von Erweiterten Hierarchischen Automaten [MLS97, Mik00] ist eine Möglichkeit geschaffen worden, die Beschreibung von Interlevel-Transitionen strukturerhaltend auszudrücken. Eine Anpassung der in dieser Dissertation entstandenen Ergebnisse auf Erweiterte Hierarchische Automaten ist aus unserer Sicht leicht möglich.

History-Konnektoren: Beim Wiedereintritt in einen ODER-Zustand kann es sinnvoll sein, wieder den Unterzustand zu aktivieren, der beim vorherigen Verlassen des ODER-Zustands aktiv war. Der Statecharts-Formalismus bietet zur Beschreibung eines solchen Verhaltens *History-Konnektoren* an, die durch ein mit einem Kreis umrandetes H visualisiert werden. In Abbildung 2.2 haben wir einen History-Konnektor verwendet. Der History-Konnektor bewirkt in diesem Modell, dass beim Betreten des ODER-Zustands *On* die zuletzt aktive Signalquelle aktiviert wird. Angenommen beim Abschalten der Musikanlage war der *TunerMode* aktiv, so wird beim erneuten Betreten von *On* durch Verwendung des History-Konnektors auch wieder der Unterzustand *TunerMode* betreten. Der hier eingesetzte History-Konnektor bezieht sich auf direkte Unterzustände. Der Statecharts-Formalismus bietet einen weiteren History-Konnektor – visualisiert durch ein mit einem Kreis umrandetes H^* – an, mit dem auch indirekte Unterzustände behandelt werden können. Das eben beschriebene Beispiel könnte mit diesem History-Konnektor derart erweitert werden, dass nach dem erneuten Betreten des Zustands *On* nicht nur der *TunerMode* aktiviert, sondern zusätzlich der zuletzt aktive Sendekanal betreten wird.

Aktionen beim Betreten und Verlassen von Zuständen: Ein Zustand kann ähnlich zu einer Transition mit Aktionen markiert werden, die, abhängig vom verwendeten Schlüsselwort ENTRY oder EXIT, mit dem Betreten oder Verlassen des Zustands assoziiert werden.

Static-Reactions: Ähnlich zu den Aktionen beim Betreten und Verlassen von Zuständen können Zustände mit einem Label der Form *Trigger* [*Condition*]/*Action* markiert werden. Solange der Zustand aktiv ist, wird in jedem Zeitschritt, in dem *Trigger* anliegt und *Condition* erfüllt ist, *Action* ausgeführt. Hierbei wird der betroffene Zustand nicht verlassen.

Zeitaspekte: Der Statecharts-Formalismus erlaubt es, Zeitaspekte, wie zum Beispiel Timeout oder Schedule von Ereignissen und Aktionen, zu spezifizieren und stellt dafür eine reichhaltige Syntax zur Verfügung.

2.1.3 Hierarchische Automaten

Hierarchische Automaten wurden erfolgreich zur strukturierten Beschreibung von Statecharts eingesetzt [MLS97, LMM99b]. Im Gegensatz zu Statecharts zeichnet sich dieser Formalismus durch eine reduzierte Syntax und eine einfachere operationale Semantik aus. Trotz dieser Einschränkungen kann ein Großteil der reichhaltigen Syntax von Statecharts mit Hierarchischen Automaten beschrieben werden. Mit *Erweiterten Hierarchischen Automaten* – eine Spracherweiterung von Hierarchischen Automaten – ist insbesondere die Modellierung von Interlevel-Transitionen, unter Beibehaltung der einfachen Semantik, möglich. Es gibt eine Reihe von Formalismen [JM94, Mar91, Mar92], die ähnlich wie bei Hierarchischen Automaten zur Beschreibung von Modellen mit hierarchischen Strukturen geeignet sind, die aber keine Formalisierung von Interlevel-Transitionen erlauben. Obwohl wir in unserer Arbeit Interlevel-Transitionen nicht betrachten, ist im Hinblick auf eine Weiterführung der Arbeit die Ver-

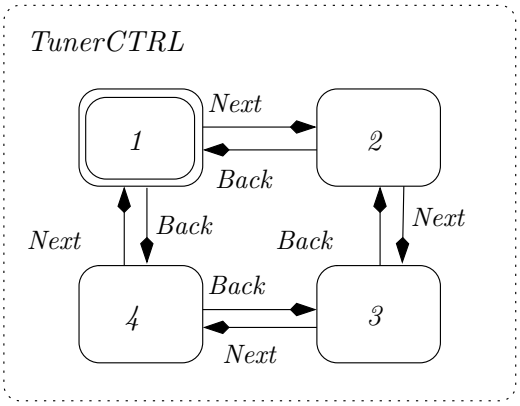


Abbildung 2.3: Sequentieller Automat zur Verhaltensbeschreibung eines *Tuners*

wendung Hierarchischer Automaten zur Formalisierung von Statecharts sinnvoll, da diese Formalisierung mit geringem Aufwand durch die Behandlung von Interlevel-Transitionen auf Basis Erweiterter Hierarchischen Automaten vervollständigt werden kann. In der Literatur sind semantische Beschreibungen von Statecharts durch Erweiterte Hierarchische Automaten sowohl für Statecharts mit der Statemate-Semantik [MLS97, Mik00] als auch für UML-State-Machines [LMM99b] angegeben.

In diesem Abschnitt stellen wir die Beschreibung von Statemate-Statecharts durch Hierarchische Automaten [MLS97, Mik00] vor. In Abschnitt 2.1.4 wird eine auf Hierarchischen Automaten basierende Semantik für Statecharts angegeben.

Ein Hierarchischer Automat besteht in der Regel aus mehreren *Sequentiellen Automaten*, die parallel und/oder hierarchisch miteinander komponiert sind. Ein Sequentieller Automat ist hierbei eine endliche Zustandsmaschine, die aus folgenden vier Komponenten besteht.

- Menge von Zuständen
- Initialer Zustand
- Menge von Labeln
- Transitionsrelation

Zusätzlich kann einem Sequentiellen Automaten ein partitionierter Datenraum zugeordnet sein, der bei der Ausführung von Transitionen partiell gelesen und geschrieben wird. Abbildung 2.3 zeigt den Sequentiellen Automaten *TunerCTRL*, mit dem das Verhalten der Senderauswahl eines Radios beschrieben ist.

Analog zum Statecharts-Formalismus werden in einem Sequentiellen Automaten die Zustände durch an den Ecken abgerundete Rechtecke und die Transitionen durch gerichtete Kanten zwischen den Zuständen visualisiert. Auch die Ausdrucksprache zur Markierung von Transitionen in Form von Labeln ist identisch zum Statecharts-Formalismus. Im Gegensatz zu Statecharts verfügt ein Sequentieller Automat aber über keine Ausdrucksmittel zur Beschreibung von hierarchischen Strukturen. Initiale Zustände werden anders als bei Statecharts nicht durch Pfeile, sondern graphisch durch doppelte Umrandungen hervorgehoben (vgl. Zustand *1* in Abbildung 2.3).

Sollen in einer Abbildung mehrere Sequentielle Automaten dargestellt werden, so werden die Zustände und Transitionen durch gepunktete Rechtecke in graphische Einheiten zusammengefasst. Eine graphische Einheit bezeichnet hierbei genau einen Sequentiellen Automaten und ist mit einem Namen – in Abbildung 2.3 z.B. *TunerCTRL* – versehen. Jede Transition und jeder Zustand eines Hierarchischen Automaten muss genau einem seiner Sequentiellen Automaten zuzuordnen sein.

Abbildung 2.4 zeigt die Komposition von mehreren Sequentiellen Automaten zu einem Hierarchischen Automaten, die der Statecharts-Spezifikation aus Abschnitt 2.1.1 entspricht. Hierzu wurde das Statechart entlang seiner Hierarchie in Komponenten zerlegt, die sich durch Sequentielle Automaten repräsentieren lassen. Fügt man die so gewonnenen Sequentiellen Automaten mit Hilfe einer Kompositionsfunktion in einer Baumstruktur wieder zusammen, so erhält man einen Hierarchischen Automaten, der das ursprüngliche Statechart repräsentiert. Eine solche Kompositionsfunktion sei beispielsweise CF. Dann ordnet CF jedem Zustand s des betrachteten Hierarchischen Automaten eine Menge M von Sequentiellen Automaten zu. Die Menge M mit $M = (CF\ s)$ beschreibt hierbei alle *direkten Sequentiellen Unterautomaten* von s. Wir sprechen auch davon, dass der Zustand s durch die Sequentiellen Automaten aus M *direkt verfeinert* wird. Weiterhin kann aus der Kardinalität von M – repräsentiert durch den Operator $\#$ – auf die Zustandsart von s geschlossen werden.

$\#M = 0$ gibt an, dass s ein Basis-Zustand ist.

$\#M = 1$ gibt an, dass s ein Oder-Zustand ist.

$\#M > 1$ gibt an, dass s ein Und-Zustand ist.

Eine Kompositionsfunktion muss die Sequentiellen Automaten eines Hierarchischen Automaten in einer Baumstruktur anordnen. Deshalb müssen folgende drei Eigenschaften durch die Kompositionsfunktion erfüllt sein.

1. Es muss ein eindeutiger Wurzelautomat ausgezeichnet sein.

2. Alle Sequentiellen Automaten mit Ausnahme des Wurzelautomaten besitzen einen eindeutigen Vorgängerzustand.

3. Es existieren keine Zyklen.

Eine mathematisch präzise Beschreibung der Wohlgeformtheitseigenschaften einer Kompositionsfunktion ist in der Definition 3.6 auf Seite 47 angegeben.

2.1.4 Semantik

Zusätzlich zu der im letzten Abschnitt eingeführten strukturellen Beschreibung von Statecharts durch Hierarchische Automaten, soll in diesem Abschnitt ein Eindruck von der Semantik gegeben werden. Wir stützen uns dabei auf die Statemate-Semantik [Har87], die von Erich Mikk für Hierarchische Automaten konkretisiert wurde [MLS97, Mik00]. In diesem Abschnitt wird lediglich ein intuitives Verständnis der Semantik geben. Eine präzise mathematische Beschreibung ist in Abschnitt 3.3 zu finden.

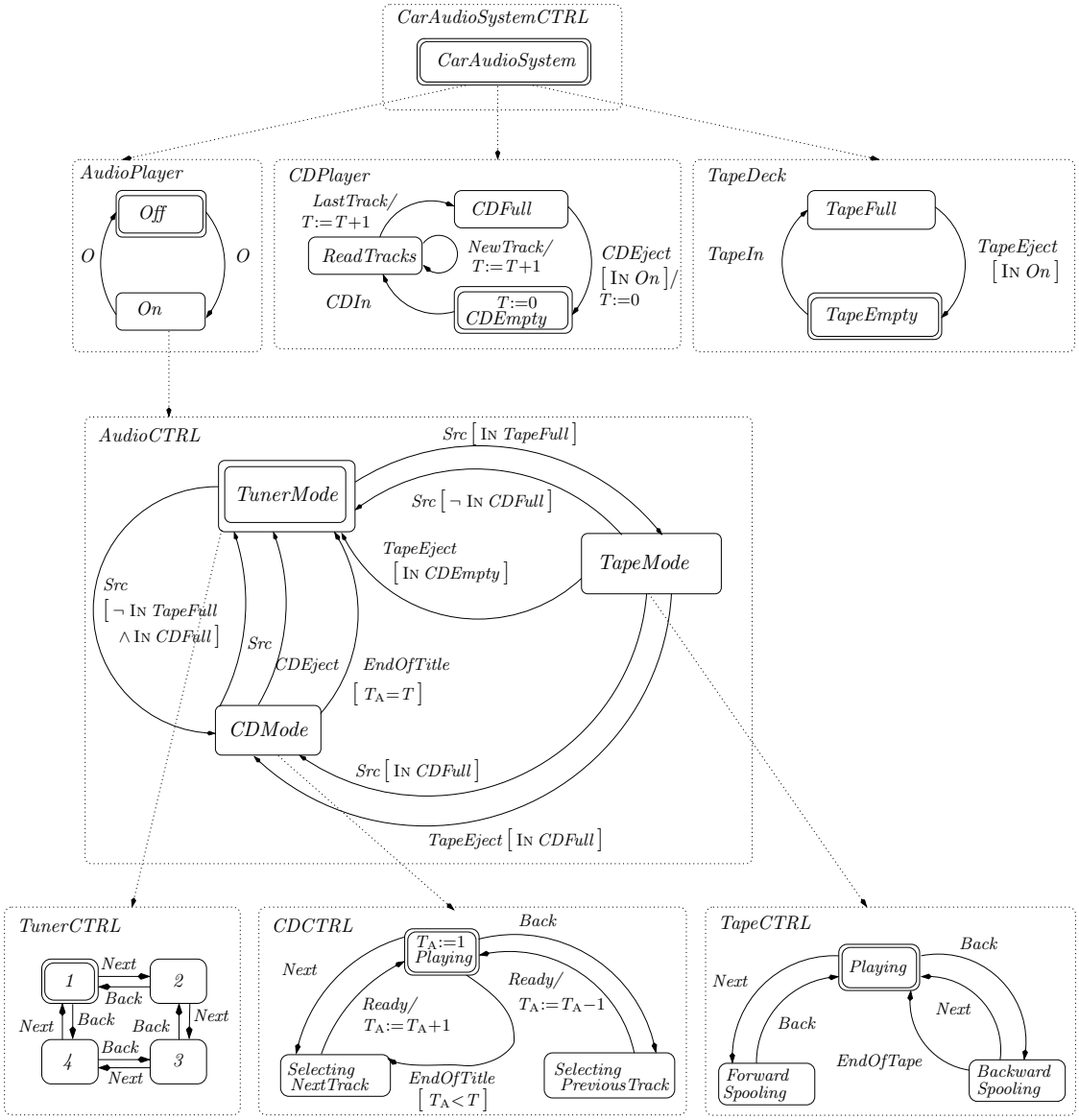


Abbildung 2.4: Hierarchischer Automat der Statecharts-Spezifikation *Car-Audio-System*

Semantische Zustände und Berechnungspfade

Die semantische Beschreibung eines Statecharts wird durch eine Menge von *Berechnungspfaden* festgelegt. Der *Berechnungspfad* eines Statecharts ist eine Sequenz von semantischen Zuständen dieses Statecharts.

Ein semantischer Zustand wird auch als *Status* bezeichnet und besteht aus folgenden Komponenten:

- Menge von aktiven Zuständen
- Menge von anliegenden Ereignissen
- aktuelle Datenbelegung eines Statecharts.

Die erste dieser Komponenten wird in der Literatur häufig auch als *Konfiguration* [Har87] bezeichnet. Für eine Konfiguration C eines Statecharts SC werden folgende vier Eigenschaften gefordert.

1. C enthält den Wurzelzustand von SC
2. Wenn C einen ODER-Zustand A enthält, so muss C auch genau einen direkten Unterzustand von A enthalten.
3. Wenn C einen UND-Zustand A enthält, so müssen in C auch alle direkten Unterzustände von A enthalten sein.
4. C enthält nur Zustände, die sich durch die obigen drei Regeln ergeben.

Die zweite Komponente eines Status enthält Ereignisse, die entweder durch den Statechart erzeugt oder von der Umgebung empfangen worden sind.

Ein *Berechnungspfad* beginnt mit dem *initialen Status*. Der initiale Status besteht aus einer initialen Konfiguration, einer leeren Ereignismenge und einer initialen Datenbelegung. Die initiale Konfiguration wird aus den initialen Zuständen des Statecharts unter Beachtung der Wohlgeformtheitseigenschaften einer Konfiguration errechnet. Die initiale Konfiguration für das Beispiel aus Abbildung 2.1 ist durch folgende Zustandsmenge beschrieben.

{ *CarAudioSystem*, *CDPlayer*, *TapeDeck*, *AudioPlayer*, *CDEmpty*, *TapeEmpty*, *Off* }

Initiale Datenbelegungen sind für eine Statecharts-Spezifikation im Allgemeinen vorgegeben. Die in dieser Arbeit vorgestellte Formalisierung von Statecharts (vgl. Kapitel 3) legt sogar fest, dass genau eine initiale Datenbelegung für eine gegebene Statecharts-Spezifikation vorliegt. Für das Beispiel aus Abbildung 2.1 ist die Datenvariable T mit 0 und die Datenvariable $T_{\mathtt{A}}$ mit 1 initialisiert.

Im Folgenden wird ein Eindruck davon vermittelt, wie aus einem initialen Status über einzelne Berechnungsschritte ein Berechnungspfad bestimmt werden kann.

Synchrone Schrittsemantik

Innerhalb eines Berechnungspfads wird der Übergang von einem Status zu einem direkten Folgestatus als *Schritt* bezeichnet. Der Statecharts-Formalismus gehört zu der Familie der *synchronen* Sprachen [vH05]. Synchrone Sprachen sind dadurch charakterisiert, dass parallel

komponierte Subsysteme ihre Schritte gleichzeitig ausführen. Weiterhin ist ein Schritt atomar, d.h. er bildet keine semantischen Zwischenzustände im Gegensatz zu einer Mikroschrittsemantik [PS91].

Ein Schritt wird durch das Ticken einer fiktiven globalen Uhr angestoßen. Ein semantischer Zustandsübergang wird häufig auch als *atomarer synchroner Zeitschritt* bezeichnet. Um den direkten Folgestatus S' für einen gegebenen Status S berechnen zu können, müssen die folgenden Regeln beachtet werden.

Aktivierte Transitionen: Zunächst werden jene Transitionen ermittelt, die im Status S *aktiviert* sind. Das sind die Transitionen, deren Startzustand in der Konfiguration von S liegt, und für die *Trigger* und *Condition* durch die anliegenden Ereignisse und die aktuelle Datenbelegung erfüllt werden.

Maximale konfliktfreie Transitionsmengen: Auf Basis der Menge aller aktivierten Transitionen werden maximale Untermengen von Transitionen identifiziert, die beim gleichzeitigen Ausführen zu einem Folgestatus führen. Es wird zu einem Status S im Allgemeinen mehrere maximale konfliktfreie Untermengen von Transitionen geben, so dass das daraus induzierte nichtdeterministische Verhalten durch mehrere Berechnungspfade mit unterschiedlichen Folgestatus beschrieben ist.

Prioritätsregeln für Transitionen: Der Nichtdeterminismus im Berechnungsmodell wird durch einen so genannten Transitionsselektionsalgorithmus eingeschränkt. Dieser Algorithmus definiert Prioritätsregeln für Transitionen, die auf unterschiedlichen Hierarchiestufen operieren. Für die Statemate-Semantik wurde festgelegt, dass jene Transitionen höher priorisiert sind, die auf höheren Hierarchieebenen operieren.

Effekt von parallel ausgeführten Transitionen: Für eine konfliktfreie Untermenge von Transitionen TS wird der Folgestatus S' folgendermaßen bestimmt.

1. Die Konfiguration von S' wird durch die Zielzustände der Transitionen in TS und deren Oberzuständen festgelegt.

2. Die Ereignismenge von S' wird durch die während des Schrittes erzeugten Ereignisse bestimmt. Ereignisse können während eines Schrittes durch die ausgeführten Transitionen aus TS oder durch die Umgebung erzeugt werden.

3. Die Datenbelegung von S' wird aus den Update-Funktionen der ausgeführten Transitionen errechnet. Unbeschriebene Partitionen des Datenraumes erhalten die Datenbelegung des vorhergehenden Status S. Konflikte, die beim konkurrierenden Schreiben von Partition durch gleichzeitig ausgeführte Transitionen entstehen (*Racing-Effekt*), werden durch eine Interleaving-Semantik in einem Nichtdeterminismus aufgelöst.

Triviale Berechnungsschritte: Sollten im Status S keine Transitionen feuern, so wird trotzdem ein Schritt ausgeführt, wobei Konfiguration und Datenbelegung des Folgestatus S' zu Konfiguration und Datenbelegung des vorhergehenden Status S identisch sind. Die Ereignismenge von S' wird in diesem Falle lediglich durch die von der Umgebung erzeugten Ereignisse bestimmt.

Wir haben in diesem Abschnitt die abstrakte Syntax und Semantik von Statecharts eingeführt. Mit Hilfe dieses Formalismus kann das Verhalten von sicherheitskritischen

Softwaresystemen sehr detailliert beschrieben werden. Allerdings werden solche Statecharts-Spezifikationen für praktische Beispiele entsprechend umfangreich. Folglich ist es ohne geeignete Werkzeugunterstützung häufig schwer zu beurteilen, ob die Spezifikation notwendige Sicherheitseigenschaften einhält oder verletzt.

Im folgenden Abschnitt führen wir temporale Logiken ein, mit deren Hilfe wichtige Sicherheitseigenschaften knapp und präzise formuliert werden können. Im darauf folgenden Abschnitt stellen wir Werkzeuge vor, die es ermöglichen, die Gültigkeit von temporallogischen Eigenschaften in den Spezifikationen sicherheitskritischer Systeme nachzuweisen.

2.2 Temporale Logiken

Wichtige sicherheitskritische Eigenschaften von eingebetteten Systemen sind häufig zeitabhängig, so dass die klassische Prädikatenlogik nicht ausreicht, um diese zu formulieren. Aus diesem Grunde stellen wir in diesem Abschnitt temporale Logiken vor, da hiermit Aussagen über die Zeit ausgedrückt werden können. Da temporale Logiken auf Basis von so genannten Kripke-Strukturen interpretiert werden, führen wir zunächst diesen Formalismus kurz ein. Darauf aufbauend werden wir im zweiten Teil des Abschnitts einen Eindruck von gängigen temporalen Logiken geben und diese miteinander vergleichen.

2.2.1 Kripke-Strukturen

Temporale Logiken sind gewöhnlich durch Kripke-Strukturen interpretiert [CGP00]. Eine *Kripke-Struktur* [Kri63] ist ein zustandsbasierter Transitionsgraph, mit dem das Verhalten eines reaktiven Systems beschrieben werden kann. Eine Kripke-Struktur besteht aus folgenden Komponenten:

- Menge von Zuständen
- Menge von initialen Zuständen
- Totale Transitionsrelation auf den Zuständen
- Labelfunktion, die jedem Zustand eine Menge von atomaren Aussagen zuordnet.

Im Gegensatz zu Statecharts oder Hierarchischen Automaten verfügen Kripke-Strukturen über keine Ausdrucksmittel, mit denen das Systemverhalten strukturiert oder kompakt ausgedrückt werden kann. In Kapitel 5 ist eine formale Definition für Kripke-Strukturen angegeben und gezeigt, wie das durch einen Hierarchischen Automaten definierte Verhalten mit Hilfe einer Kripke-Struktur beschrieben werden kann.

Um temporallogische Aussagen für eine Kripke-Struktur interpretieren zu können, betrachtet man die Zeit als Abfolge von Berechnungsschritten. Die Berechnungsschritte einer Kripke-Struktur werden durch Pfade beschrieben. Ein Pfad ist eine unendliche Sequenz von Zuständen. Er wird, von einem initialen Zustand ausgehend, mit Hilfe der Transitionsrelation abgeleitet. Um nichtdeterministisches Verhalten modellieren zu können, kann die Transitionsrelation mehrere Folgezustände für einen Zustand festlegen. Deshalb werden häufig die Berechnungen einer Kripke-Struktur durch unendliche Berechnungsbäume beschrieben. Der Wurzelzustand eines Berechnungsbaumes ist durch einen initialen Zustand der Kripke-Struktur festgelegt. Die Verzweigungen im Berechnungsbaum repräsentieren Nichtdeterminismen. Das

Konstruieren von Berechnungsbäumen für eine Kripke-Struktur wird in der Literatur auch als Abwickeln oder Auffalten bezeichnet.

2.2.2 Lineare und verzweigende Zeitlogik

Ein wesentliches Kriterium zur Einordnung von temporalen Logiken ist die Frage, ob in der Logik Aussagen über Verzweigungen auf den zugrunde liegenden Berechnungsbäumen formuliert werden können, oder ob die Ausdrucksmächtigkeit auf Aussagen über lineare Berechnungsabläufe beschränkt ist. Wir führen in diesem Abschnitt wichtige temporale Operatoren ein, mit denen Eigenschaften in linearer und verzweigender Zeitlogik formuliert werden können. Weiterhin geben wir drei wichtige temporale Logiken an, die auf Basis der eingeführten Operatoren definiert werden können und sich bezüglich des genannten Kriteriums unterscheiden. Abschließend stellen wir die temporale Logik CTL gesondert vor.

Temporale Operatoren

Zunächst geben wir temporale Operatoren an, mit denen Aussagen auf einem linearen Berechnungspfad formuliert werden können.

X P	Auf dem nächstfolgenden Zustand des Pfades gilt P.
F P	Auf mindestens einem Zustand des Pfades gilt P.
G P	Auf allen Zuständen des Pfades gilt P.
P U Q	Auf einem Zustand S des Pfades gilt Q und auf allen Zuständen vor S gilt P.
P W Q	Q gilt auf allen Zuständen des Pfades bis zu einem Zustand auf dem P gilt.

Das Folgeverhalten ab dem Zustand einer Kripke-Struktur besteht im Allgemeinen aus mehreren Berechnungspfaden, die häufig zu einem Berechnungsbaum zusammengefaßt werden. Folgende Operatoren erlauben es, die Folgepfade eines Zustands zu quantifizieren.

A P	Auf allen Folgepfaden des Zustands gilt P.
E P	Auf mindestens einem Folgepfad des Zustands gilt P.

Mit Hilfe der eingeführten Operatoren können, aufbauend auf den atomaren Aussagen einer Kripke-Struktur, komplexe temporallogische Formeln zusammengesetzt werden. Temporallogische Formeln werden in Zustands- und Pfadformeln unterschieden. *Zustandsformeln* können aus folgenden Bestandteilen aufgebaut sein.

1. Jede atomare Aussage ist eine Zustandsformel.
2. Die Negation einer Zustandsformel ist eine Zustandsformel.
3. Die Verknüpfung von zwei Zustandsformeln mit Konjunktion oder Disjunktion ist eine Zustandsformel.
4. Die Applikation einer Pfadformel auf den Pfadquantor A oder E ist eine Zustandsformel.

Pfadformeln werden auf Basis folgender Regeln zusammengesetzt.

1. Jede Zustandsformel ist eine Pfadformel.
2. Die Negation einer Pfadformel ist eine Pfadformel.
3. Die Verknüpfung von zwei Pfadformeln mit Konjunktion oder Disjunktion ist eine Pfadformel.
4. Die Applikation einer Pfadformel auf die temporalen Operatoren X, F oder G ist eine Pfadformel.
5. Die Verknüpfung von zwei Pfadformeln mit den temporalen Operatoren U oder W ist eine Pfadformel.

Alle Zustandsformeln, die sich auf Basis der angegebenen Regeln konstruieren lassen, bilden die Formelklasse CTL* [EH86].

Temporale Logiken

Der Name der temporalen Logik CTL* deutet bereits darauf hin, dass es sich um eine Erweiterung der weniger mächtigen Logik CTL (*Computation-Tree-Logic*) [CE81] handelt. Tatsächlich kann die verzweigende Zeitlogik CTL in CTL* ausgedrückt werden. Die Einschränkung der Ausdruckssprache von CTL gegenüber CTL* besteht darin, dass nur Pfadformeln zugelassen sind, in denen direkt vor jedem verwendeten temporalen Operator ein Pfadquantor steht. Um diese Einschränkung auszudrücken, wird das Regelwerk zum Bilden von Pfadformeln folgendermaßen abgeändert.

1. Die Applikation einer Zustandsformel auf die temporalen Operatoren X, F oder G ist eine Pfadformel.
2. Die Verknüpfung von zwei Zustandsformeln mit den temporalen Operatoren U oder W ist eine Pfadformel.

Das Regelwerk zum Bilden einer Zustandsformel ist hingegen in der Ausdruckssprache von CTL und CTL* identisch.

In CTL* sind auch Aussagen über lineare Berechnungspfade formulierbar. Entsprechend kann neben CTL auch die lineare Zeitlogik LTL (*Linear-Temporal-Logic*) [Pnu77] in CTL* ausgedrückt werden. Die Einschränkung der Ausdruckssprache von LTL gegenüber CTL* besteht darin, dass nur Zustandsformeln der Form $\mathrm{A}\,\boldsymbol{F}$ zulässig sind, wobei $\boldsymbol{F}$ eine Pfadformel ist, die mit folgenden Regeln gebildet wird.

1. Jede atomare Aussage ist eine Pfadformel.
2. Es gelten die Regeln 2 bis 5 zum Bilden einer Pfadformel für CTL*.

Der Zusammenhang zwischen den beschriebenen Logiken ist in Abbildung 2.5 graphisch veranschaulicht. Aus der Abbildung ist ersichtlich, dass es Formeln gibt, die sowohl in CTL als auch in LTL enthalten sind. Ferner gibt es CTL*-Formeln, die weder in CTL noch in LTL ausgedrückt werden können.

Im Folgenden stellen wir die temporale Logik CTL genauer vor, da sie in dieser Dissertation eine wichtige Rolle spielt.

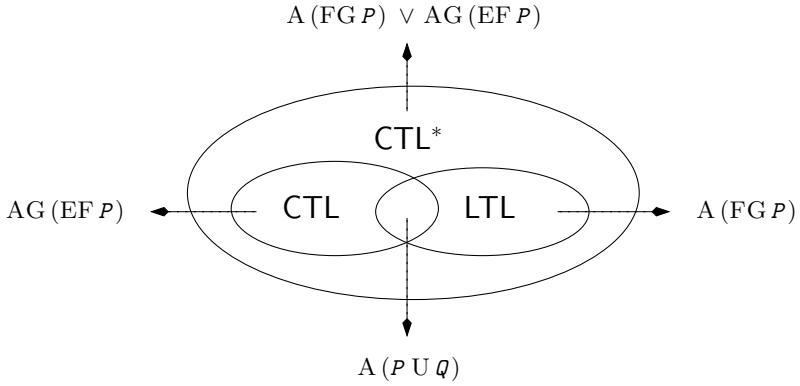


Abbildung 2.5: Zusammenhänge zwischen drei wichtigen temporalen Logiken

Computation-Tree-Logic

Die besondere Charakteristik der Ausdruckssprache von CTL [CE81] hat dazugeführt, dass sich spezielle Operatoren für diese Logik durchgesetzt haben. Folgende zehn Operatoren werden hierbei verwendet.

$$\text{AX, AF, AG, AU, AW}$$
$$\text{EX, EF, EG, EU, EW}$$

Es handelt sich bei diesen Operatoren um syntaktische Abkürzungen für jeweils einen Pfadquantor, der gefolgt von einem temporalen Operator ist. Um welchen Pfadquantor und um welchen temporalen Operator es sich jeweils handelt, ist dem Namen des entsprechenden CTL-Operators zu entnehmen. Alle Operatoren der CTL lassen sich auf die drei Operatoren EX, EG und EU unter Verwendung von Negation und Disjunktion zurückführen [CGP00]. Diese Eigenschaft kann dazu genutzt werden, eine möglichst kompakte Semantik für die temporale Logik CTL anzugeben.

Die Semantik von temporallogischen Formeln ist auf Basis einer Kripke-Struktur (vgl. Abschnitt 2.2.1) definiert. Sei F eine Formel der temporalen Logik CTL und M eine Kripke-Struktur, so ist F an einem Zustand S in M erfüllt, wenn folgende Beziehung erfüllt ist.

$$M, S \vDash F$$

Hierbei ist die Relation $\vDash$ durch folgende induktive Definition beschrieben.

$$\begin{aligned}
M, S \vDash F &\Leftrightarrow \begin{cases}\text{Wenn } F \text{ eine atomare Aussage ist, so enthält die Labelfunktion} \\ \text{von } M \text{ am Zustand } S \text{ die Formel } F.\end{cases} \\
M, S \vDash \neg F &\Leftrightarrow M, S \nvDash F \\
M, S \vDash F \vee G &\Leftrightarrow M, S \vDash F \vee M, S \vDash G \\
M, S \vDash \text{EX } F &\Leftrightarrow \begin{cases}\text{Es existiert ein Pfad } P \text{ aus } M, \text{ der mit dem Zustand } S \text{ beginnt. Für} \\ \text{den nächstfolgenden Zustand } T \text{ auf dem Pfad } P \text{ gilt } M, T \vDash F.\end{cases}
\end{aligned}$$

$$M, S \models \text{EU } F\ G \quad \Leftrightarrow \begin{cases} \text{Es existiert ein Pfad } P \text{ aus } M, \text{ der mit dem Zustand } S \text{ beginnt.} \\ \text{Ferner existiert ein } i \geq 0, \text{ so dass für den } i\text{-ten Zustand } T \text{ auf} \\ \text{dem Pfad } P \text{ gilt } M, T \models G. \text{ Schließlich gilt für alle Zustände } U, \text{ die} \\ \text{auf dem Pfad } P \text{ vor T liegen, } M, U \models F. \end{cases}$$

$$M, S \models \text{EG } F \quad \Leftrightarrow \begin{cases} \text{Es existiert ein Pfad } P \text{ aus } M, \text{ der mit dem Zustand } S \text{ beginnt.} \\ \text{Ferner gilt für alle Zustände } T \text{ auf dem Pfad } P, \text{ dass } M, T \models F \text{ gilt.} \end{cases}$$

Eine CTL-Formel F ist in einer gegebenen Kripke-Struktur M dann gültig, wenn ein initialer Zustand S existiert, so dass $M, S \models F$ gilt. Als Alternative zu der vorgestellten induktiven Semantik-Definition kann jeder Operator der CTL durch einen Fixpunkt beschrieben werden [EC80]. In Abschnitt 5.3 auf Seite 88 sind Definitionen für alle CTL-Operatoren auf Basis von Fixpunkten angegeben.

In dieser Dissertation wird die temporale Logik CTL verwendet, um ein logikbasiertes Framework für die Verifikation von Statecharts-Spezifikationen zu entwickeln. Eine ausführliche Formalisierung dieser Logik ist in Kapitel 5 für den Theorembeweiser Isabelle/HOL angegeben. Weiterhin verwenden wir in Kapitel 6 zur Definition einer Datenabstraktionstheorie für Statecharts die Sublogik ∀CTL. Diese Logik beschränkt die Ausdruckssprache von CTL auf Formeln, in denen keine Pfadquantifizierungen mit dem Existenzquantor E vorkommen. Um implizite Existenzquantoren zu vermeiden, wird ferner gefordert, dass Negationen nur auf atomare Aussagen angewendet werden dürfen. Die Sublogik ∀CTL wird in der Literatur auch als universales Fragment der temporalen Logik CTL bezeichnet. Man findet dort auch die Bezeichnung ACTL.

Wir haben in diesem Abschnitt einen Überblick über gängige temporale Logiken gegeben. In der Literatur sind sehr ausführliche Beschreibungen zu diesem Thema zu finden [CGP00, Win01]. Im folgenden Abschnitt stellen wir Softwarewerkzeuge vor, die im Rahmen dieser Dissertation eingesetzt werden, um maschinengestützte Analysen auf Statecharts-Spezifikationen durchzuführen.

2.3 Maschinengestützte Verifikation

In diesem Abschnitt stellen wir zunächst Software vor, mit deren Hilfe die Verifikation von sicherheitskritischen Systemen maschinengestützt erfolgen kann. Anschließend geben wir eine kurze Einführung in die Konzepte des interaktiven Theorembeweisers Isabelle/HOL, die für das Verständnis der vorliegenden Dissertation besonders wichtig sind.

2.3.1 Verifikationswerkzeuge

Im Wesentlichen lassen sich die Softwarewerkzeuge zur formalen Verifikation von sicherheitskritischen Systemen in zwei Gruppen unterteilen. Es gibt einerseits Verifikationswerkzeuge, die durch eine starke Spezialisierung einen hohen Grad an Automatisierung erreichen. Andererseits existieren Verifikationswerkzeuge, die möglichst wenig Annahmen an die zu prüfenden Modelle stellen und deshalb im Allgemeinen eine Interaktion mit dem Benutzer erfordern. Im Rahmen dieser Dissertation werden für die Verifikation von Statecharts-Spezifikationen die automatischen Beweiser SMV [McM93] und SVC [Dil], sowie der interaktive Theorembeweiser Isabelle [Pau87b] eingesetzt.

Symbolic-Model-Verifier

Die Technik des *Model-Checking* zur Verifikation von Eigenschaften auf endlichen Modellen wurde unabhängig voneinander durch Emerson und Clarke [CE81] sowie durch Quielle und Sifakis [QS82] vorgeschlagen. Diese Technik dient dazu, die Gültigkeit einer temporallogischen Formel in einem Zustandssystem – meist angegeben durch eine Kripke-Struktur – zu überprüfen. Model-Checking ist eingeschränkt auf endliche Zustandssysteme und ist durch einen Algorithmus implementiert, der keine Benutzerinteraktion erfordert. Die traditionellen Algorithmen für das Model-Checking basieren entweder auf Fixpunktberechnungen oder auf Berechnungsvorschriften, die aus der Automatentheorie abgeleitet sind. In Abhängigkeit von der zu unterstützenden Temporallogik wird eines der beiden Verfahren ausgewählt. Da sich beispielsweise die Operatoren der CTL durch Fixpunkte beschreiben lassen, wird für das Model-Checking von CTL-Formeln häufig ein auf Fixpunkten basierendes Verfahren eingesetzt.

Die Algorithmen des im Rahmen dieser Dissertation eingesetzten Model-Checkers SMV (*Symbolic-Model-Verifier*) [McM93] basieren auf Fixpunktberechnungen. Der Name der Software deutet darauf hin, dass es sich um ein symbolisches Berechnungsverfahren handelt. Hierbei werden zunächst die Zustände und die Transitionsrelation der zu überprüfenden Kripke-Struktur durch boolesche Funktionen repräsentiert. Diese Funktionen werden dann durch so genannte ROBDDs (*Reduced-Ordered-Binary-Decision-Diagrams*) [Bry86] in der Implementierung dargestellt. ROBDDs sind eine kanonische Normalform für boolesche Funktionen. Sie bieten sowohl eine Basis für eine kompakte Darstellung der booleschen Funktionen, als auch für deren effiziente Manipulation.

Für das im Rahmen dieser Dissertation entwickelte Framework haben wir die originale Implementierung von SMV verwendet. Allerdings haben sich im Laufe der Zeit einige Varianten dieses populären Model-Checkers herausgebildet. Als ein wichtiger Vertreter ist der Model-Checker Cadence-SMV zu nennen, der in der Forschungsgruppe von Ken McMillan entstanden ist [McM]. Weiterhin wurde an der Carnegie Mellon University eine Erweiterung der originalen Implementierung mit dem Namen NuSMV entwickelt [CCGR99]. Schließlich kombinierte man an der Carnegie Mellon University die bestehende Implementierung von SMV mit einigen Entscheidungsprozeduren von SAT-Lösern zu dem so genannten Framework SyMP [Ber99].

Standford-Validity-Checker

SVC (*Standford-Validity-Checker*) [Dil] ist ein automatischer Theorembeweiser für das entscheidbare Fragment der FOL (*First-Order-Logic*). Dieses Fragment schließt die Quantifizierung von Variablen aus. Unterstützt werden auch Gleichungen, nichtinterpretierte Funktionen und Konstanten, sowie propositionale Konnektoren. Der SVC ist in der Forschungsgruppe von David L. Dill an der Standford University entstanden und wird dort seit Jahren weiterentwickelt. Ursprünglich wurde das Werkzeug zur Verifikation von Mikro-Prozessoren implementiert. In jüngeren Arbeiten wird der Beweiser aber auch zur Überprüfung von Spezifikationen sicherheitskritischer Systeme eingesetzt. Ferner wurde er in einem Experiment als alternative Entscheidungsprozedur in dem verbreiteten Theorembeweiser PVS [ORS92] verwendet.

In Abschnitt 8.2 dieser Ausarbeitung stellen wir vor, wie mit Hilfe des SVC eine eigenschaftserhaltende Abstraktion für eine Statecharts-Spezifikation errechnet werden kann.

Isabelle

Isabelle [NP] ist ein interaktiver Theorembeweiser. Eine erste Implementierung wurde bereits 1986 von Lawrence C. Paulson vorgestellt [Pau87b, Pau87a]. Seit 1992 wird das Werkzeug allerdings im Rahmen einer engen Kooperation zwischen den Forschungsgruppen von Lawrence C. Paulson (Cambridge University) und Tobias Nipkow (Technische Universität München) entwickelt und gilt heute als gemeinschaftliche Erfindung.

Isabelle basiert auf einer so genannten Meta-Logik, die mit verschiedenen Objekt-Logiken instanziiert werden kann. Deshalb wird das Werkzeug auch als generischer Theorembeweiser bezeichnet. Beispiele für mögliche Instanzen der Meta-Logiken sind FOL (*First-Order-Logic*), ZF (*Zermelo-Fraenkel-Set-Theorie*) oder CTT (*Constructive-Type-Theorie*). Im Kontext dieser Arbeit wird mit der meist genutzten Instanz HOL (*Higher-Order-Logic*) gearbeitet. Die Meta-Logik selbst repräsentiert ein Fragment der intuitionistischen HOL mit Typklassenpolymorphie [Pau89].

Isabelle ist in der funktionalen Programmiersprache Standard ML [Pau96] implementiert. Das umgesetzte Konzept basiert auf der Idee, zunächst nur die Meta-Logik in ML zu implementieren. Die Objekt-Logiken sind hingegen in der Meta-Logik als logische Theorien formuliert. Durch dieses Vorgehen gilt das Werkzeug als sehr zuverlässig und leicht anpassbar.

Die Objekt-Logik HOL ist beispielsweise durch ein axiomatisches Kalkül in der Meta-Logik ausgedrückt. Hierfür werden zunächst neun Basisaxiome für fünf Basisoperatoren der HOL in einer Isabelle-Theorie eingeführt. Für diese Basisaxiome ist aus der Literatur bekannt, dass sie ein vollständiges und korrektes Kalkül für die HOL bilden. Alle anderen Operatoren und Theoreme der Objekt-Logik werden *konservativ* aus dem axiomatisch Eingeführtem abgeleitet. Dies bedeutet, dass sich alle neuen Operatoren auf Basisoperatoren und alle neuen Theoreme auf Basisaxiome zurückführen lassen. Aufbauend auf den Theorien einer Objekt-Logik, können neue Theorien erstellt werden. Im Rahmen dieser Dissertation werden Isabelle-Theorien zur Formalisierung von Statecharts-Spezifikationen (vgl. Kapitel 3), zur Formalisierung der temporalen Logik CTL (vgl. Kapitel 5) und zur Formalisierung einer Abstraktionstheorie (vgl. Kapitel 7) vorgestellt, die aus den Theorien für die Objekt-Logik HOL konservativ abgeleitet sind.

Um Theoreme in Isabelle ableiten zu können, werden Inferenzmechanismen eingesetzt, die eine Kombination von Resolution und Unifikation höherer Ordnung darstellen. Isabelle ist ein taktischer Theorembeweiser. Dies bedeutet, dass das Beweisvorgehen zum Ableiten von Theoremen automatisiert werden kann. Hierbei werden einfache Beweisschritte, so genannte Taktiken, auf Basis einer Beweisstrategie zu einer komplexen Taktik zusammengesetzt. Eine komplexe Taktik wird auch als *Tactical* bezeichnet. Eine ML-basierte Taktiksprache ermöglicht das Implementieren von eigenen Tacticals zur automatischen Behandlung spezieller Beweisprobleme. Die aktuelle Version von Isabelle enthält bereits leistungsstarke Taktiken, deren Beweisvorgehen über Parameter geeignet gesteuert werden kann. Hier sind insbesondere ein klassischer Beweiser und ein Simplifizierer zu nennen.

Obwohl gerade in den letzten Jahren der Automatisierungsgrad von Isabelle deutlich gesteigert werden konnte, besteht eine unüberwindbare Grenze für nicht entscheidbare Logiken, die mit keiner noch so geschickten Beweisstrategie aufgehoben werden kann. Ferner existieren neben Isabelle viele andere leistungsfähige Beweiswerkzeuge, die für verschiedene entscheidbare Teilprobleme eine Lösung vollautomatisch und in kürzerer Zeit errechnen, als es mit einer Beweistaktik innerhalb von Isabelle möglich wäre. Um dieses Potential auch für Isabelle nutzbar zu machen, besteht die Möglichkeit, über eine so genannte Orakelschnittstelle an-

dere vertrauenswürdige Beweiswerkzeuge einzubinden. Im Rahmen dieser Dissertation wird in Abschnitt 8.1 eine Anbindung des Model-Checkers SMV zur effizienten Verifikation von endlichen Statecharts-Spezifikationen vorgestellt. Ferner geben wir in Abschnitt 8.2 an, wie ein Abstraktionsalgorithmus für Statecharts-Spezifikationen in den Beweisverlauf über die Orakelschnittstelle an Isabelle eingebunden werden kann.

In der Literatur ist umfangreiches Material zur detaillierten Einführung in die Funktionsweise von Isabelle verfügbar [NPW02, Pau92, Pau94]. Im Folgenden gehen wir auf die Logik HOL genauer ein. Wir stellen wichtige logische Konzepte vor, die beim Entwickeln einer Isabelle/HOL-Theorie benötigt werden.

2.3.2 Formalisierungen in Isabelle/HOL

HOL ist eine klassische Logik höherer Stufe, die sich unter anderem dadurch auszeichnet, dass alle in ihr vorkommenden Entitäten streng typisiert sind. Sie wurde deshalb bei ihrer ersten Erwähnung in der Literatur auch als *Simple-Type-Theory* bezeichnet [Chu40]. Typisierte Logiken bieten den Vorteil, dass sie im Allgemeinen in einem Softwarewerkzeug effizienter zu behandeln sind. So ist beispielsweise das Type-Checking für die klassische HOL vollautomatisch möglich. Speziell Isabelle/HOL unterstützt neben einer einfachen Typtheorie Konzepte für polymorphe Typen und für axiomatische Typklassen. Letztere werden im Rahmen dieser Arbeit nicht verwendet.

Eine in Isabelle/HOL formulierte Theorie besteht hauptsächlich aus Typ- und Konstantendefinitionen, Theoremen und Beweisen. Im Folgenden beschreiben wir, wie Typen und Konstanten in Isabelle/HOL definiert werden. Wir legen damit zugleich eine für diese Ausarbeitung verbindliche Syntax fest.

Typdefinitionen

Ein Grundtyp T ist in Isabelle/HOL entweder durch einen booleschen Typ $bool$ oder durch einen mit dem Typkonstruktor für Funktionen $\rightarrow$ zusammengesetzten Typ repräsentiert.

$$T = bool \mid T \rightarrow T$$

Ein Grundtyp enthält mindestens ein Element. Es können zunächst nur totale Funktionen angegeben werden. Ferner gilt, dass mindestens eine unendliche Domain existieren muss. Ein Funktionstyp der Form $\alpha_1 \rightarrow (\ldots \rightarrow \alpha_n \rightarrow \beta)$ wird durch den Ausdruck $[\alpha_1, \ldots, \alpha_n] \rightarrow \beta$ abgekürzt.

Auf den Grundtypen aufbauend, sind in der aktuellen Version von Isabelle/HOL unter anderem folgende Typkonstruktoren bereits definiert. Die Bezeichner α und β repräsentieren hierbei zwei beliebige polymorphe Typen.

Produkt:	$\alpha * \beta$	Summe:	$\alpha + \beta$
Menge:	$\alpha\ set$	Liste:	$\alpha\ list$
Integer:	int	Natürliche Zahl:	nat
Option:	$\alpha\ option$	Partielle Funktion:	$\alpha \rightarrow\!\!\!\!\!\!+\ \beta$

Obwohl für die Grundtypen von Isabelle/HOL nur totale Funktionen zugelassen sind, wurde eine Möglichkeit gefunden, Partialität zu modellieren. Hierbei wird die Undefiniertheit einer

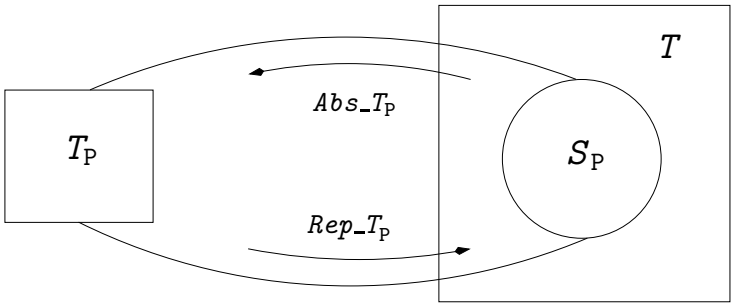


Abbildung 2.6: Typabstraktion über einer Menge

Funktion durch eine spezielle Konstante explizit ausgedrückt. In Isabelle/HOL wird dieses Verhalten mit Hilfe eines Option-Typs folgendermaßen formal gefasst. Wir verwenden in der Beschreibung eine Typdefinition, die mit dem Symbol $\equiv_{\mathtt{t}}$ angegeben wird. Das Symbol $\equiv_{\mathtt{s}}$ repräsentiert hingegen eine Typabkürzung.

$$\begin{aligned} \alpha\ \mathit{option} &\equiv_t \mathit{None} \mid \mathit{Some}\,\alpha \\ \alpha \rightarrow\!\!\!\!\!\!+\ \beta &\equiv_s \alpha \longrightarrow (\beta\ \mathit{option}) \end{aligned}$$

Der Konstruktor *None* repräsentiert in der angegebenen Typdefinition das undefinierte Ergebnis einer Funktionsanwendung. Das definierte Ergebnis einer Funktionsanwendung wird mit Hilfe des Konstruktors *Some* in den Option-Typ eingebettet. Diese Einbettung kann für definierte Werte mit dem Operator *the* umgekehrt werden. Mit den Operatoren *dom* und *ran* kann für eine gegebene partielle Funktion der Definitions- bzw. Wertebereich ermittelt werden.

Ein zentrales Konzept zur Einführung von neuen Typdefinitionen ist die Typabstraktion über einer Menge. Hierbei wird zunächst über einem bereits existierenden Typ T ein Prädikat P definiert. Danach wird aus P eine Menge $S_{\mathtt{P}}$ abgeleitet, die alle die Elemente enthält, die P erfüllen. Schließlich wird $S_{\mathtt{P}}$ zu einem neuen Typ $T_{\mathtt{P}}$ abstrahiert. In Abbildung 2.6 ist das Vorgehen für die Typabstraktion über einer Menge graphisch veranschaulicht.

Der neu geschaffene Typ $T_{\mathtt{P}}$ ist isomorph zu der durch das Prädikat P beschriebenen Menge. Der Isomorphismus ist durch die beiden Injektionsfunktionen $\mathit{Rep\_T}_{\mathtt{P}}$ vom Typ $T_{\mathtt{P}} \rightarrow T$ und $\mathit{Abs\_T}_{\mathtt{P}}$ vom Typ $T \rightarrow T_{\mathtt{P}}$ repräsentiert. Die Injektionsfunktionen werden vom Isabelle-System für eine Typdefinition automatisch eingeführt. Ferner werden folgende Theoreme automatisch bewiesen. Einer festen Konvention folgend werden sowohl die Bezeichner für die Injektionsfunktionen, als auch die Namen für die Theoreme aus dem Typnamen abgeleitet.

$$\mathit{Abs\_T}_{\mathtt{P}}\,(\mathit{Rep\_T}_{\mathtt{P}}\ A) = A \qquad \text{(RepTPInv)}$$

$$(\mathit{Rep\_T}_{\mathtt{P}}\ A) \in S_{\mathtt{P}} \qquad \text{(RepTP)}$$

$$\frac{A \in S_{\mathtt{P}}}{\mathit{Rep\_T}_{\mathtt{P}}\,(\mathit{Abs\_T}_{\mathtt{P}}\ A) = A} \qquad \text{(RepAbsTP)}$$

Um zu garantieren, dass der neue Typ T_{P} nicht leer ist, muss zunächst ein Theorem abgeleitet werden, das die Erfüllbarkeit des Prädikates P belegt. Erst danach kann der Typ in Isabelle/HOL definiert werden. Hierbei ist der Name des zuvor abgeleiteten Theorems anzugeben.

Konstantendefinitionen

Konstanten sind in Isabelle/HOL *Terme*. Terme sind auf Basis einer Menge von Variablen V und einer Menge von Konstanten C durch folgende Grammatik definiert.

$$T = V \mid C \mid (T\,T) \mid \lambda V.\ T$$

Wenn wir in einer Theorie neue Konstanten einführen, erweitern wir entsprechend die Menge von gültigen Konstanten C. Konstanten besitzen einen eindeutigen Typ, der im Deklarationsteil mit dem Symbol $::_{\mathsf{c}}$ festgelegt wird. Mit dem Symbol $\equiv_{\mathsf{c}}$ wird eine nichtrekursive Konstantendefinition angegeben. Eine Typannotierung kann mit dem Symbol :: vorgenommen werden.

Primitiv-rekursive Definitionen

Um rekursive Konstantendefinitionen in Isabelle/HOL behandeln zu können, stehen verschiedene Ausdrucksmittel zur Verfügung. So können beispielsweise induktive Mengen und verschachtelte oder primitiv-rekursive Konstanten definiert werden. Insbesondere für die Beweisführung mit *primitiv-rekursiven* Definitionen bietet Isabelle/HOL einen hohen Grad an Automatisierung. Wir führen an dieser Stelle das Konzept für primitiv-rekursive Definitionen am Beispiel der in Isabelle/HOL verfügbaren Theorie über polymorphe Listen ein.

Die Basis einer primitiv-rekursiven Definition bildet ein primitiv-rekursiver Datentyp. Der Datentyp für Listen ist folgendermaßen definiert.

$$\begin{array}{lll} \alpha\ \mathit{list} \equiv_{\mathsf{t}} & \mathit{Nil} & (\text{“}[]\text{”}) \\ & \mid \mathit{Cons}\ \alpha\ (\alpha\ \mathit{list}) & (\mathsf{infixr}\ \text{“\#”}\ 65) \end{array}$$

In der Typdefiniton werden zwei Konstruktoren eingeführt. Der Konstruktor Nil repäsentiert eine leere Liste. Der Konstruktor Cons repäsentiert eine Funktion, die für ein Element vom Typ α und für eine Liste vom Typ $\alpha\ \mathit{list}$ eine neue Liste konstruiert. Mit der in Klammern angegebenen Mixfix-Syntax werden abkürzende Schreibweisen für die Konstruktoren angegeben. So kann der Operator Nil auch mit $[]$ und der Operator Cons auch mit der Infix-Notation $\#$ angesprochen werden. Die Zahl 65 legt die Stärke der Bindung des Operators $\#$ fest. Führt man in Isabelle/HOL einen primitiv-rekursiven Datentyp ein, so wird unter anderem automatisch eine Induktionsregel in Form eines Theorems abgeleitet.

Aufbauend auf einem primitiv-rekursiven Datentyp können primitiv-rekursive Konstantendefinitionen angegeben werden. Diese bestehen aus einer Regelmenge von rekursiven Gleichungen. Hierbei wird mit jeder vorkommenden Gleichung ein Konstruktor des zugrunde liegenden Datentyps behandelt. Als Beispiel geben wir zunächst den Deklarationsteil und danach die Regelmenge der primitiv-rekursiven Konstante map an. Hierbei handelt es sich um das bekannte Listenfunktional, mit dem eine Funktion F vom Typ $\alpha \to \beta$ auf alle Elemente einer Liste vom Typ $\alpha\ \mathit{list}$ angewendet wird.

$$map ::_{\mathsf{c}} [\alpha \to \beta, \alpha\ list] \to \beta\ list$$

$$\begin{aligned} map\ F\ [] &= [] \\ map\ F\ (A \# L) &= (F\ A) \# (map\ F\ L) \end{aligned}$$

Die in der Regelmenge definierten Gleichungen sind als Theoreme verfügbar und können für die Ableitung von neuen Theoremen verwendet werden.

Eine ausführliche Darstellung aller syntaktischen Elemente von Isabelle/HOL würde den Rahmen dieser Arbeit sprengen. Wir verweisen nochmals auf die Literatur [NPW02, Pau94]. Wenn wir in den folgenden Kapiteln Operatoren von Isabelle/HOL verwenden, die in diesem Abschnitt noch nicht eingeführt wurden, werden wir an der Stelle ihrer Verwendung eine kurze Erläuterung geben.

Kapitel 3

Hierarchische Automaten in Isabelle/HOL

In diesem Kapitel wird eine strukturerhaltende HOL-Theorie zur Formalisierung Hierarchischer Automaten vorgestellt. Diese Theorie bildet die Grundlage, um Beweise über Statecharts-Spezifikationen in Isabelle/HOL führen zu können.

Wir geben zunächst in Abschnitt 3.1 eine Formalisierung der abstrakten Syntax Hierarchischer Automaten an, die sich an Vorarbeiten aus der Literatur orientiert [Mik00]. In Abschnitt 3.2 beschreiben wir zwei Konzepte, mit deren Hilfe die Formalisierung aus Abschnitt 3.1 im Hinblick auf eine effizientere Beweisführung verbessert werden kann. Im letzten Abschnitt dieses Kapitels gehen wir auf die semantische Interpretation Hierarchischer Automaten ein. In Analogie zur Formalisierung der abstrakten Syntax beschreiben wir zunächst eine an Vorarbeiten aus der Literatur orientierte Semantik. Anschließend wird im Hinblick auf eine effizientere Beweisführung die Formalisierung der Semantik optimiert.

3.1 Abstrakte Syntax

Wir führen im ersten Teil dieses Abschnitts eine Formalisierung der abstrakten Syntax Sequentieller Automaten ein. Darauf aufbauend stellen wir im zweiten Teil eine Formalisierung der abstrakten Syntax Hierarchischer Automaten vor. Hierbei wird ein Hierarchischer Automat aus mehreren Sequentiellen Automaten zusammengesetzt.

3.1.1 Sequentielle Automaten

Wir stellen zunächst eine Definition für Sequentielle Automaten vor. Wir geben dazu einen *Typ* zur Beschreibung von Sequentiellen Automaten an, indem wir die Typabstraktion über einer Menge bilden. Eine allgemeine Einführung in die Isabelle-Syntax, unter anderem zur Beschreibung von Typabstraktionen über einer Menge, ist in Abschnitt 2.3.2 auf Seite 37 angegeben.

Definition 3.1 (Sequentieller Automat (SA)) *Sei σ ein Typ von Zustandsbezeichnern, ϵ ein Typ von Ereignisbezeichnern und δ ein Typ für den Datenraum, dann ist ein Sequentieller Automat* `SA` *über $(\sigma, \epsilon, \delta)$ durch ein Quadrupel (S, I, L, T) repräsentiert. Hierbei ist*

- `S` *die Menge von Zuständen,*
- `I` *der initiale Zustand,*
- `L` *die Menge von Labeln und*

- *T die Menge von Transitionen*

des Sequentiellen Automaten SA. Diese Komponenten müssen die interne Konsistenzbedingung SeqAutoCorrect über Sequentielle Automaten erfüllen, die in Definition 3.3 angegeben ist. Der Typ $(\sigma, \epsilon, \delta)$ *seqauto besteht aus allen Sequentiellen Automaten über* $(\sigma, \epsilon, \delta)$.

$$
\begin{aligned}
(\sigma, \epsilon, \delta)\ \mathit{seqauto} \equiv_{\mathtt{t}} \{\, & (S, I, L, T) \mid \\
& (S :: \sigma\ \mathit{set}) \\
& (I :: \sigma) \\
& (L :: (\sigma, \epsilon, \delta)\ \mathit{label}\ \mathit{set}) \\
& (T :: (\sigma, \epsilon, \delta)\ \mathit{trans}\ \mathit{set}). \\
& \mathit{SeqAutoCorrect}\ S\ I\ L\ T \,\} \\
& \quad \textit{gerechtfertigt durch}\ \mathsf{SeqAutoNonEmpty}
\end{aligned}
\tag{3.1}
$$

□

Die Typabstraktion über einer Menge ist nur dann korrekt, wenn mindestens ein Element in der zu abstrahierenden Menge vorhanden ist. Dies haben wir für den Typ $(\sigma, \epsilon, \delta)$ *seqauto* mit dem Theorem SeqAutoNonEmpty nachgewiesen.

Um auf die Komponenten eines Sequentiellen Automaten zugreifen zu können, führen wir folgende Selektionsoperatoren ein. So kann die Menge von Zuständen mit dem Operator *States*, der initiale Zustand mit dem Operator *InitState*, die Menge von Labeln mit dem Operator *Label* und die Transitionsrelation mit dem Operator *Trans* selektiert werden. Weiterhin führen wir einen Operator *Events* ein, mit dem alle in einem Sequentiellen Automaten vorkommenden Ereignisse bestimmt werden können. Zusätzlich zu den Selektionsoperatoren ist ein Operator *Rep_seqauto* gegeben, mit dem ein Element des Typs $(\sigma, \epsilon, \delta)$ *seqauto* in seine Repräsentation überführt werden kann. Umgekehrt können die Komponenten eines Sequentiellen Automaten durch die Abstraktionsfunktion *Abs_seqauto* aus der Repräsentation in den Typ $(\sigma, \epsilon, \delta)$ *seqauto* abstrahiert werden. Folgendes Theorem verdeutlicht den Zusammenhang zwischen Selektionsoperatoren und Repräsentationsoperator.

Abgeleitetes Theorem 3.2 (Repräsentation durch Selektion)

$$\mathit{Rep\_seqauto}\ SA = (\mathit{States}\ SA, \mathit{InitState}\ SA, \mathit{Label}\ SA, \mathit{Trans}\ SA) \qquad \text{(RepSeqautoTuple)}$$

□

In Definition 3.1 werden Typsynonyme zur Beschreibung von Transitionen und Labeln verwendet. Im Folgenden erklären wir diese Abkürzungen.

Eine *Transition* besteht aus einem Startzustand, einem Label und einem Zielzustand und wird durch folgendes Typsynonym beschrieben. Der Typkonstruktor $*$ repräsentiert das Kartesische Produkt.

$$(\sigma, \epsilon, \delta)\ \mathit{trans} \equiv_{\mathtt{s}} \sigma * (\sigma, \epsilon, \delta)\ \mathit{label} * \sigma$$

Um auf den Start- bzw. Zielzustand einer Transition zugreifen zu können, führen wir die Selektionsoperatoren *Source* und *Target* ein. Das Label einer Transition kann mit dem Operator *Label* selektiert werden.

Ein *Label* besteht aus einer Expression, einem Guard und einer Aktion. Der Typ aller Label wird durch folgendes Typsynonym beschrieben.

$$(\sigma, \epsilon, \delta)\ \mathit{label} \equiv_{\mathtt{s}} (\sigma, \epsilon)\ \mathit{expr} * \delta\ \mathit{guard} * (\epsilon, \delta)\ \mathit{action}$$

Mit einer *Expression* können atomare Aussagen, wie das Anliegen von Ereignissen (*En*) oder das Vorhandensein eines aktiven Zustandes (*In*), ausgedrückt werden. Diese atomaren Aussagen können negiert (*Not*) oder durch Konjunktion (*And*) oder Disjunktion (*Or*) miteinander verknüpft werden. Eine Expression ist durch folgenden rekursiven Datentyp definiert.

$$\begin{aligned}(\sigma, \epsilon)\,\mathit{expr} \equiv_{\mathtt{t}}\ & \mathit{True} \\ \mid\ & \mathit{In}\ \sigma \\ \mid\ & \mathit{En}\ \epsilon \\ \mid\ & \mathit{Not}\ (\sigma, \epsilon)\,\mathit{expr} \\ \mid\ & \mathit{And}\ (\sigma, \epsilon)\,\mathit{expr}\ (\sigma, \epsilon)\,\mathit{expr} \\ \mid\ & \mathit{Or}\ (\sigma, \epsilon)\,\mathit{expr}\ (\sigma, \epsilon)\,\mathit{expr}\end{aligned}$$

Ein *Guard* repräsentiert ein Prädikat, das von der aktuellen Datenbelegung des Datenraumes abhängt und durch das Typsynonym $\delta\ \mathit{guard}$ beschrieben ist. Hierbei repräsentiert $\delta\ \mathit{data} \rightarrow \mathit{bool}$ eine totale Funktion von $\delta\ \mathit{data}$ nach bool. Der Typkonstruktor $\delta\ \mathit{data}$ definiert den Typ für eine gültige Datenbelegung auf dem Datenraum δ. Er ist in Kapitel 4 im Detail beschrieben.

$$\delta\ \mathit{guard} \equiv_{\mathtt{s}} \delta\ \mathit{data} \rightarrow \mathit{bool}$$

Eine *Aktion* beschreibt den Effekt einer Transition und besteht aus einer Menge von generierten Ereignissen und einer Update-Funktion, mit der der Datenraum partiell geschrieben werden kann. Der Typ von partiellen Update-Funktion ist durch $\delta\ \mathit{pupdate}$ repräsentiert. Für eine präzise Definition partieller Update-Funktionen verweisen wir auf Kapitel 4, da zur Einführung der Formalisierung der abstrakten Syntax Sequentieller Automaten ein tieferes Verständnis für diesen Datentyp nicht erforderlich ist.

$$(\epsilon, \delta)\ \mathit{action} \equiv_{\mathtt{s}} \epsilon\ \mathit{set} * \delta\ \mathit{pupdate}$$

In Definition 3.1 haben wir für die Komponenten eines Sequentiellen Automaten Eigenschaften zur *Internen Konsistenz* mit dem Prädikat *SeqAutoCorrect* gefordert. Im Folgenden präsentieren wir die mathematisch präzise Definition dieses Prädikates durch eine Konstante. Eine Konstante wird in dieser Arbeit durch $::_{\mathtt{c}}$ deklariert und durch $\equiv_{\mathtt{c}}$ definiert. Die Endlichkeit einer Menge wird durch das Prädikat *finite* ausgedrückt.

Definition 3.3 (Wohlgeformter Sequentieller Automat) *Sei* `S` *eine Menge von Zuständen,* `I` *ein initialer Zustand,* `L` *eine Menge von Labeln und* `T` *eine Menge von Transitionen, dann bilden diese Komponenten einen wohlgeformten Sequentiellen Automaten, wenn*

folgende Bedingungen erfüllt sind.

$$\begin{aligned} \mathit{SeqAutoCorrect} ::_{\mathsf{c}} & [\,\sigma\ \mathit{set},\ \sigma,\ (\sigma,\epsilon,\delta)\ \mathit{label}\ \mathit{set},\ (\sigma,\epsilon,\delta)\ \mathit{trans}\ \mathit{set}\,] \\ & \rightarrow \mathit{bool} \\ \mathit{SeqAutoCorrect}\ S\ I\ L\ T \equiv_{\mathsf{c}}\ & I \in S\ \wedge\ \mathit{finite}\ S\ \wedge \\ & S \neq \varnothing\ \wedge\ \mathit{finite}\ T\ \wedge \\ & \forall\,(s,l,t) : T.\ \ s \in S \wedge t \in S \wedge l \in L \end{aligned} \tag{3.2}$$

□

Das in Definition 3.3 angegebene Prädikat garantiert für einen Sequentiellen Automaten *SA*, dass sein initialer Zustand und die Start- und Zielzustände aller seiner Transitionen in der Zustandsmenge des *SA* enthalten sind. Analog dazu müssen auch die Label aller seiner Transitionen in der Labelmenge des *SA* enthalten sein. Ferner garantiert das Prädikat in Definition 3.3, dass Zustands- und Transitionsmenge endlich sind. Mit dieser letzten Forderung erreichen wir, dass jeder wohlgeformte Sequentielle Automat durch einen Model-Checker verifiziert werden kann, wenn der zugrunde liegende Datenraum endlich ist.

Unter Verwendung der eingeführten Typdefinition 3.1 und den darauf geltenden Operationen führen wir in Abbildung 3.1 die Konstante *TunerCTRL* ein, um den Sequentiellen Automaten für die *Tuner*-Steuerung aus dem *Car-Audio-System*-Modell zu formalisieren (vgl. auch Abbildung 2.3).

Die polymorphen Typen für die Zustands- und Ereignisbezeichner werden durch den atomaren Typ *string* instanziiert. Für das *Car-Audio-System* wurde ein aus zwei Partitionen bestehender Datenraum angegeben, wobei jede der Partitionen aus einer Integer-Variable besteht. In Abbildung 3.1 ist ein solcher Datenraum durch den Typ *ds* repräsentiert. Auf eine genaue Formalisierung von *ds* wollen wir an dieser Stelle verzichten und auf das Kapitel 4 verweisen. Hier ist unter anderem (vgl. Abbildung 4.2 auf Seite 74) die Formalisierung eines aus zwei Partitionen bestehenden Datenraumes angegeben.

Sollte eine Statecharts-Spezifikation vorliegen, in der keine Datenvariablen verwendet werden, so kann der polymorphe Typ für den Datenraum mit dem atomaren Typ *unit* instanziiert werden. Dieser Typ enthält genau ein Element, das mit `()` bezeichnet ist.

Jede Komponente des Sequentiellen Automaten wird durch eine eigene Konstante repräsentiert. Hierbei beschreibt die Konstante *TunerStates* die Zustandsmenge, die Konstante *TunerInitState* den initialen Zustand, die Konstante *TunerLabels* die Labelmenge und die Konstante *TunerTrans* die Transitionsrelation des Sequentiellen Automaten *TunerCTRL*.

Ein leerer Guard wird durch die Konstante *GuardDefault* beschrieben und definiert ein immer erfüllbares Prädikat über dem Datenraum. Ein leerer Aktionsteil wird durch die Konstante *ActionDefault* beschrieben. Er definiert, dass keine Ereignisse erzeugt werden und die Wertebelegung des Datenraumes nicht verändert wird.

Um Beweise auf dem in Abbildung 3.1 formalisierten Sequentiellen Automaten führen zu können, ist es im Allgemeinen notwendig, so genannte Selektionstheoreme abzuleiten. Selektionstheoreme ermöglichen den konstruktiven Zugriff auf die einzelnen Komponenten eines Sequentiellen Automaten. Zur Selektion der Zustände des in Abbildung 3.1 dargestellten Sequentiellen Automaten haben wir folgendes Selektionstheorem abgeleitet.

$$\mathit{States}\ \mathit{TunerCTRL} = \{\,\text{“1”},\ \text{“2”},\ \text{“3”},\ \text{“4”}\,\} \tag{SelectStatesTunerCTRL}$$

$$
\begin{aligned}
&\mathit{TunerStates} ::_c \mathit{string\ set}\\
&\mathit{TunerStates} \equiv_c \{\text{“1”}, \text{“2”}, \text{“3”}, \text{“4”}\}\\
&\mathit{TunerInitState} ::_c \mathit{string}\\
&\mathit{TunerInitState} \equiv_c \text{“1”}\\
&\mathit{TunerLabels} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
&\mathit{TunerLabels} \equiv_c \{(\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault})\}\\
&\mathit{TunerTrans} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
&\mathit{TunerTrans} \equiv_c \{(\text{“1”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“2”}),\\
&\qquad (\text{“2”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“1”}),\\
&\qquad (\text{“2”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“3”}),\\
&\qquad (\text{“3”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“2”}),\\
&\qquad (\text{“3”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“4”}),\\
&\qquad (\text{“4”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“3”}),\\
&\qquad (\text{“4”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“1”}),\\
&\qquad (\text{“1”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“4”})\}\\
&\mathit{TunerCTRL} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{TunerCTRL} \equiv_c \mathit{Abs\_seqauto}\ (\mathit{TunerStates}, \mathit{TunerInitState},\\
&\qquad\qquad \mathit{TunerLabels}, \mathit{TunerTrans})
\end{aligned}
$$

Abbildung 3.1: Formalisierung der *Tuner*- Steuerung aus dem *Car-Audio-System*- Modell

Die Ableitung von Selektionstheoremen für einen Sequentiellen Automaten setzt implizit den Nachweis der Wohlgeformtheit für diesen Sequentiellen Automaten voraus. Mit folgendem Theorem, das den Zusammenhang zwischen Repräsentations- und Abstraktionsoperator am Beispiel der Typabstraktion für Sequentielle Automaten herstellt, wird diese Abhängigkeit deutlich.

$$\frac{\mathit{SeqAutoCorrect}\ S\ I\ L\ T}{\mathit{Rep\_seqauto}(\mathit{Abs\_seqauto}(S, I, L, T)) = (S, I, L, T)} \qquad \text{(RepAbsSeqAuto)}$$

Der Nachweis der Wohlgeformtheitseigenschaften (vgl. Definition 3.3) ist für viele Sequentielle Automaten leicht zu erbringen. Wir haben im Rahmen dieser Arbeit eine Taktik implementiert, die in vielen Fällen einen automatischen Beweisprozess ermöglicht.

Die vorgestellte Formalisierung Sequentieller Automaten werden wir verwenden, um Hierarchische Automaten in einer strukturierten Art und Weise zu beschreiben.

3.1.2 Hierarchische Automaten

In diesem Abschnitt geben wir eine Definition für die abstrakte Syntax Hierarchischer Automaten an. Wir verwenden analog zur Definition Sequentieller Automaten die Typabstraktion über einer Menge, um einen geeigneten *Typ* zur Beschreibung von Hierarchischen Automaten einzuführen.

Definition 3.4 (Hierarchischer Automat (HA)) *Sei σ ein Typ von Zustandsbezeichnern, ϵ ein Typ von Ereignisbezeichnern und δ ein Typ für den Datenraum, dann ist ein Hierarchischer Automat* `HA` *über $(\sigma, \epsilon, \delta)$ durch ein Quadruple (S_A, E, F_{Comp}, D) repräsentiert. Hierbei ist*

- S_A *die Menge von Sequentiellen Automaten,*
- E *die Menge von Ereignissen,*
- F_{Comp} *die Kompositionsfunktion und*
- D *die initiale Datenbelegung des Datenraumes*

des Hierarchischen Automat `HA`*. Diese Komponenten müssen die interne Konsistenzbedingung* `HierAutoCorrect` *für Hierarchische Automaten erfüllen, die in Definition 3.5 angegeben ist. Der Typ $(\sigma, \epsilon, \delta)$* `hierauto` *besteht aus allen Hierarchischen Automaten über $(\sigma, \epsilon, \delta)$.*

$$
\begin{aligned}
(\sigma, \epsilon, \delta)\,\texttt{hierauto} \equiv_t \{\, & (S_A, E, F_{Comp}, D) \mid \\
& (S_A :: ((\sigma, \epsilon, \delta)\,\texttt{seqauto}\,\texttt{set})) \\
& (E :: \epsilon\,\texttt{set}) \\
& (F_{Comp} :: (\sigma \rightharpoonup ((\sigma, \epsilon, \delta)\,\texttt{seqauto}\,\texttt{set}))) \\
& (D :: \delta\,\texttt{data}). \\
& \texttt{HierAutoCorrect}\ S_A\ E\ F_{Comp}\ D\,\} \\
& \quad \textit{gerechtfertigt durch } \mathsf{HierAutoNonEmpty}
\end{aligned}
\tag{3.3}
$$

□

Der in Definition 3.4 verwendete Typkonstruktor $\rightharpoonup$ beschreibt eine partielle Funktion[1] und wird dazu eingesetzt, die Kompositionsfunktion für einen Hierarchischen Automaten zu deklarieren. Das Theorem HierAutoNonEmpty belegt, dass die in Definition 3.4 abstrahierte Menge nicht leer ist.

Um auf die Komponenten eines Hierarchischen Automaten zugreifen zu können, führen wir folgende Selektionsoperatoren ein. Die Menge an Sequentiellen Automaten kann mit dem Operator `SAs`, die Menge an Ereignissen mit dem Operator `Events`, die Kompositionsfunktion mit dem Operator `CompFun` und der initiale Datenwert mit dem Operator `InitData` für einen Hierarchischen Automaten bestimmt werden. Die Repräsentanten eines Hierarchischen Automaten können mit dem Operator `Rep_hierauto` ermittelt werden. Gegebene Repräsentanten kann man umgekehrt mit dem Operator `Abs_hierauto` wieder zu einem Hierarchischen Automaten des Typs $(\sigma, \epsilon, \delta)$ `hierauto` zusammensetzen[2]. Zusätzlich führen wir einen Operator

[1] In Abschnitt 2.3.2 auf Seite 37 wird erläutert, wie partielle Funktionen in Isabelle/HOL beschrieben werden können.

[2] Im Folgenden werden wir beim Einführen weiterer Typdefinitionen darauf verzichten, die Namen für Repräsentations- und Abstraktionsfunktion explizit anzugeben, da sich diese uniform aus dem Typnamen ableiten lassen (vgl. Abschnitt 2.3.2).

States ein, der angewendet auf einen Hierarchischen Automaten alle in ihm vorkommenden Zustände zurückgibt. Der Operator $Root_{\mathtt{SA}}$ liefert einen eindeutigen Sequentiellen Automaten, der sich auf der obersten Hierarchiestufe des Hierarchischen Automaten befindet (vgl. Definition 3.6).

Definition 3.5 (Wohlgeformter Hierarchischer Automat) *Sei $S_{\mathtt{A}}$ eine Menge von Sequentiellen Automaten, E eine Menge von Ereignissen, $F_{\mathtt{Comp}}$ eine Kompositionsfunktion und D eine initiale Datenbelegung des Datenraums, dann bilden diese Komponenten einen wohlgeformten Hierarchischen Automaten, wenn folgende vier Bedingungen erfüllt sind.*

$$
\begin{aligned}
&\mathit{HierAutoCorrect} ::_{\mathtt{c}} [\,(\sigma, \epsilon, \delta)\ \mathit{seqauto}\ \mathit{set},\ \epsilon\ \mathit{set},\\
&\qquad\qquad \sigma \nrightarrow ((\sigma, \epsilon, \delta)\mathit{seqauto}\ \mathit{set}),\ \delta\ \mathit{data}\,]\\
&\qquad\qquad \rightarrow \mathit{bool}\\
\mathit{HierAutoCorrect}\ S_{\mathtt{A}}\ E\ F_{\mathtt{Comp}}\ D \equiv_{\mathtt{c}}\ & \mathit{finite}\ S_{\mathtt{A}}\ \wedge\\
&\Big(\bigcup A \in S_{\mathtt{A}}\,.\ \mathit{Events}\ A\Big) \subseteq E\ \wedge\\
&\forall\ A, B : S_{\mathtt{A}}\,.\ A \neq B \Rightarrow \mathit{States}\ A \cap \mathit{States}\ B = \varnothing\ \wedge\\
&\mathit{IsCompFun}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}
\end{aligned}
$$

Die Eigenschaft **IsCompFun** *fordert die Wohlgeformtheitseigenschaft für die Kompositionsfunktion $F_{\mathtt{Comp}}$ bezüglich $S_{\mathtt{A}}$ und ist in Definition 3.6 angegeben.* □

Definition 3.5 garantiert, dass ein wohlgeformter Hierarchischer Automat *HA* aus einer endlichen Anzahl Sequentieller Automaten aufgebaut ist. Ferner gilt, dass die zur Definition der Sequentiellen Automaten verwendeten Ereignisse in der Ereignismenge von *HA* enthalten sind. Außerdem ist sichergestellt, dass die Zustandsmengen aller Sequentiellen Automaten paarweise disjunkt sind. Das Kernstück in der Modellierung Hierarchischer Automaten ist die Kompositionsfunktion zur Beschreibung der hierarchischen Struktur von *HA*. Die Wohlgeformtheitseigenschaften der Kompositionsfunktion sind in Definition 3.6 ausführlich beschrieben.

Definition 3.6 (Wohlgeformte Kompositionsfunktion) *Sei $S_{\mathtt{A}}$ eine Menge von Sequentiellen Automaten, $F_{\mathtt{Comp}}$ eine Kompositionsfunktion, dann bildet $F_{\mathtt{Comp}}$ eine wohlgeformte Kompositionsfunktion bezüglich $S_{\mathtt{A}}$, wenn folgende Bedingungen erfüllt sind.*

$$
\begin{aligned}
&\mathit{IsCompFun} ::_{\mathtt{c}} [\,(\sigma, \epsilon, \delta)\ \mathit{seqauto}\ \mathit{set},\ \sigma \nrightarrow ((\sigma, \epsilon, \delta)\mathit{seqauto}\ \mathit{set})\,]\\
&\qquad\qquad \rightarrow \mathit{bool}\\
\mathit{IsCompFun}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}} \equiv_{\mathtt{c}}\ & \mathit{dom}\ F_{\mathtt{Comp}} = \Big(\bigcup A \in S_{\mathtt{A}}\,.\ \mathit{States}\ A\Big)\ \wedge\\
&\mathit{RootExists}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}\ \wedge\\
&\mathit{OneAncestor}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}\ \wedge\\
&\mathit{NoCycles}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}
\end{aligned}
$$

- *Das Prädikat* **RootExists** *garantiert, dass ein eindeutiger Wurzelautomat existiert, der mit dem Operator $Root_{\mathtt{SA}}$ aus $S_{\mathtt{A}}$ und $F_{\mathtt{Comp}}$ bestimmt werden kann.*

$$\mathit{RootExists}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}} \equiv_{\mathtt{c}} \exists_1 A : S_{\mathtt{A}}\,.\ A \notin \Big(\bigcup\ \mathit{ran}\ F_{\mathtt{Comp}}\Big)$$

$$
\begin{aligned}
&\mathit{CompFunCAS} \ ::_{\mathsf{c}} \ \mathit{string} \rightarrow\!\!\!\!\!\!+ \ ((\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto\,set}) \\
&\mathit{CompFunCAS} \equiv_{\mathsf{c}} \{ \text{“}\mathit{CarAudioSystem}\text{”} \mapsto \{\mathit{CDPlayer}, \mathit{TapeDeck}, \mathit{AudioPlayer}\}, \\
&\qquad\qquad \text{“}\mathit{On}\text{”} \mapsto \{\mathit{AudioCTRL}\}, \\
&\qquad\qquad \text{“}\mathit{TunerMode}\text{”} \mapsto \{\mathit{TunerCTRL}\}, \\
&\qquad\qquad \text{“}\mathit{TapeMode}\text{”} \mapsto \{\mathit{TapeCTRL}\}, \\
&\qquad\qquad \text{“}\mathit{CDMode}\text{”} \mapsto \{\mathit{CDCTRL}\}\} \\
&\qquad\qquad \cup \{S \mapsto \varnothing \mid S \in \{ \text{“}\mathit{Off}\text{”}, \text{“}\mathit{CDFull}\text{”}, \text{“}\mathit{CDEmpty}\text{”}, \text{“}\mathit{ReadTracks}\text{”}, \\
&\qquad\qquad\qquad\qquad \text{“}\mathit{TapeFull}\text{”}, \text{“}\mathit{TapeEmpty}\text{”}, \text{“}\mathit{1}\text{”}, \text{“}\mathit{2}\text{”}, \text{“}\mathit{3}\text{”}, \text{“}\mathit{4}\text{”}, \\
&\qquad\qquad\qquad\qquad \text{“}\mathit{CDCTRL.Playing}\text{”}, \text{“}\mathit{NextTrack}\text{”}, \\
&\qquad\qquad\qquad\qquad \text{“}\mathit{PreviousTrack}\text{”}, \text{“}\mathit{TapeCTRL.Playing}\text{”}, \\
&\qquad\qquad\qquad\qquad \text{“}\mathit{ForwardSpooling}\text{”}, \text{“}\mathit{BackwardSpooling}\text{”}\}\}
\end{aligned}
$$

Abbildung 3.2: Formalisierung der Kompositionsfunktion für das *Car-Audio-System*- Modell

- *Das Prädikat* `OneAncestor` *garantiert, dass jeder Sequentielle Automat bis auf den Wurzelautomaten einen eindeutigen Vorgängerzustand besitzt.*

$$
\begin{aligned}
\mathit{OneAncestor}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}} \ \equiv_{\mathsf{c}} \ \forall A : (S_{\mathtt{A}} \setminus (\mathit{Root}_{\mathtt{SA}}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}))\,. \\
\exists_1 s.\ s \in \Big(\bigcup A' : (S_{\mathtt{A}} \setminus A).\ \mathit{States}\ A'\Big) \ \wedge \\
A \in \mathit{the}(F_{\mathtt{Comp}}\ s)
\end{aligned}
$$

- *Das Prädikat* `NoCycles` *garantiert, dass* $F_{\mathtt{Comp}}$ *keine Zyklen enhält. Hierfür wird für jede nichtleere Untermenge* S*, die sich aus den Zuständen aller Sequentiellen Automaten bilden läßt, gefordert, dass in* S *ein Zustand enthalten ist, dessen Nachfolgezustände mit* S *einen leeren Schnitt bilden.*

$$
\begin{aligned}
\mathit{NoCycles}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}} \ \equiv_{\mathsf{c}} \ \forall S :\ \mathbb{P}\ \Big(\bigcup A : S_{\mathtt{A}}.\ \mathit{States}\ A\Big)\ . \\
S \neq \varnothing \Rightarrow \\
\exists\, s : S.\ S \cap \Big(\bigcup A : \mathit{the}(F_{\mathtt{Comp}}\ s).\ \mathit{States}\ A\Big) = \varnothing
\end{aligned}
$$

□

Abbildung 3.2 zeigt für das *Car-Audio-System*- Modell mit der Konstanten `CompFunCAS` ein Beispiel für die Definition einer Kompositionsfunktion.

In diesem Abschnitt haben wir eine Formalisierung Hierarchischer Automaten angegeben, die sich an Vorarbeiten von Erich Mikk [MLS97, Mik00] orientiert. Wir werden im nächsten Abschnitt sehen, dass die Formalisierung der Kompositionsfunktion einige Nachteile in sich birgt. Insbesondere für den effizienten beweistechnischen Umgang mit Statecharts-Spezifikationen mittlerer Komplexität ist eine Optimierung unbedingt erforderlich.

3.2 Optimierungen

Wenn wir die im letzten Abschnitt vorgestellte Formalisierung Hierarchischer Automaten anwenden wollen, müssen wir analog zu dem Vorgehen bei Sequentiellen Automaten für jeden in Isabelle formulierten Hierarchischen Automaten die Selektionstheoreme ableiten. Diese Ableitung erfordert den Nachweis der Wohlgeformtheitseigenschaften (vgl. Definition 3.5). Der Beweis für die Wohlgeformtheit der Kompositionsfunktion ist dabei besonders komplex. Insbesondere für den Nachweis der Zyklenfreiheit dieser Funktion (vgl. Definition 3.6) ist ein Prädikat für jede nichtleere Untermenge zu zeigen, die sich aus den Zuständen eines Hierarchischen Automaten HA ableiten läßt. Dies ergibt $2^{\#(States\ HA)}$ Beweisverpflichtungen und ist damit schon bei relativ kleinen Hierarchischen Automaten nicht mehr mit vertretbarem Aufwand zu beweisen. Selbst, wenn wir für Beweise über konkreten Statecharts-Spezifikationen die Wohlgeformtheitseigenschaft axiomatisch annehmen, zeigen unsere Erfahrungen, dass das Ableiten von vielen Theoremen in der angegebenen Formalisierung sehr aufwendig ist. Der Grund dafür ist, dass die im letzten Abschnitt angegebene Formalisierung der Kompositionsfunktion nicht durch ein Induktionsprinzip unterstützt wird. Damit kann ein wichtiges Beweisprinzip nicht zur Anwendung kommen. Eine Möglichkeit, dieses Problem zu lösen, besteht darin, die baumartige Struktur eines Hierarchischen Automaten in einer Formalisierung explizit darzustellen. Wir haben dies auf zwei unterschiedliche Arten realisiert, die wir im Folgenden vorstellen.

3.2.1 Strukturerhaltende Konstruktionsoperatoren

Eine Möglichkeit die hierarchische Struktur von Hierarchischen Automaten beweistechnisch besser nutzbar zu machen, besteht darin, bei der Kodierung geeignete Konstruktionsoperatoren zu verwenden. Für diese Konstruktionsoperatoren leiten wir allgemeingültige Theoreme ab, die bei Beweisen über konkreten Hierarchischen Automaten gewinnbringend eingesetzt werden können. Dieses Vorgehen wurde in ähnlicher Weise von Thomas Santen [San00, San99] für Beweise über Object-Z-Spezifikationen [Smi92, DR00] erfolgreich eingesetzt.

Wir schlagen zur schrittweisen Konstruktion eines Hierarchischen Automaten folgendes Vorgehen vor. Zunächst wird aus dem Sequentiellen Automaten der obersten Hierarchiestufe ein so genannter *Pseudo*-HA konstruiert. Ein *Pseudo*-HA bezeichnet hierbei einen Hierarchischen Automaten ohne hierarchische Struktur, der aus einem Sequentiellen Automaten und einer initialen Datenbelegung für den Datenraum zusammengesetzt werden kann. Danach werden die Zustände dieses Pseudo-HA unter Verwendung von Konstruktionsoperatoren schrittweise mit Sequentiellen Automaten verfeinert, so dass mehrere Hierarchiestufen entstehen. Mit folgendem Operator kann ein Pseudo-HA konstruiert werden.

Definition 3.7 (Konstruktion eines Pseudo-HA) *Sei SA ein Sequentieller Automat und D eine initiale Datenraumbelegung, dann ist die Konstruktion eines Pseudo-HA aus SA und D folgendermaßen definiert.*

$$\begin{aligned} PseudoHA &::_{\mathsf{c}} [(\sigma, \epsilon, \delta)\, seqauto, \delta\, data] \rightarrow (\sigma, \epsilon, \delta)\, hierauto \\ PseudoHA\ SA\ D &\equiv_{\mathsf{c}} Abs\_hierauto(\{SA\}, Events\ SA, EmptyMap\ (States\ SA), D) \end{aligned} \tag{3.4}$$

□

Der Operator *EmptyMap* konstruiert hierbei aus einer Zustandsmenge eine Kompositionsfunktion. Diese Konstruktion stellt sicher, dass ein Pseudo-HA nur BASIS-Zustände enthält.

$$\begin{aligned} &\mathit{EmptyMap} \ ::_{\mathsf{c}} \ \sigma \ \mathit{set} \to ((\sigma \rightharpoonup\!\!\!\!\!+ \ (\sigma, \epsilon, \delta) \ \mathit{seqauto}) \ \mathit{set}) \\ &\mathit{EmptyMap} \ S \ s \ \equiv_{\mathsf{c}} \ \mathsf{if} \ s \in S \ \mathsf{then} \ (\mathit{Some} \ \varnothing) \ \mathsf{else} \ \mathit{None} \end{aligned}$$

Mit dem Konstruktionsoperator $\boxplus$ kann ein bereits konstruierter Hierarchischer Automat um einen Sequentiellen Automaten erweitert werden. Hierzu wird der Automat an einem zu definierenden Zustand S mit dem übergebenen Sequentiellen Automaten verfeinert. Folgende Definition gibt diese Konstruktion an.

Definition 3.8 (Erweiterung eines HA durch einen SA) *Sei HA ein Hierarchischer Automat, S ein Zustand und SA ein Sequentieller Automat, dann ist die Erweiterung des HA durch SA in dem Zustand S folgendermaßen definiert.*

$$\begin{aligned} &\_ \boxplus \_ \ ::_{\mathsf{c}} \ [(\sigma, \epsilon, \delta) \ \mathit{hierauto}, \ \sigma * (\sigma, \epsilon, \delta) \ \mathit{seqauto}] \\ &\qquad\qquad \to (\sigma, \epsilon, \delta) \ \mathit{hierauto} \\ &\_ \boxplus \_ \ \equiv_{\mathsf{c}} \ (\lambda \ \mathit{HA} \ (S, \mathit{SA}) \ . \\ &\qquad\qquad \mathsf{let} \ S_{\mathtt{A}}^{'} \ = \ \{ \ \mathit{SA} \ \} \cup (\mathit{SAs} \ \mathit{HA}); \\ &\qquad\qquad\quad E^{'} \ = \ \mathit{Events} \ \mathit{HA} \cup \mathit{Events} \ \mathit{SA}; \\ &\qquad\qquad\quad F_{\mathtt{Comp}}^{'} = \ \mathit{CompFun} \ \mathit{HA} \ \oplus \ (S, \mathit{SA}) \\ &\qquad\qquad\quad D^{'} \ = \ \mathit{InitData} \ \mathit{HA} \\ &\qquad\qquad \mathsf{in} \ \ \mathit{Abs\_hierauto}(S_{\mathtt{A}}^{'}, E^{'}, F_{\mathtt{Comp}}^{'}, D^{'})) \end{aligned} \tag{3.5}$$

□

In Definition 3.5 wird der Operator $\oplus$ eingesetzt, um eine Kompositionsfunktion $F_{\mathtt{Comp}}$ an einem Zustand S mit einem Sequentiellen Automaten SA zu erweitern. Eine Erweiterung wird nur dann vorgenommen, wenn der Zustand S im Definitionsbereich von $F_{\mathtt{Comp}}$ enthalten ist. Um Zyklen in der erweiterten Kompositionsfunktion zu vermeiden, muss weiterhin gelten, dass S kein gültiger Zustand von SA ist. Mit dem Operator $\oplus_{\mathtt{P}}$ werden zwei partielle Funktionen zu einer neuen partiellen Funktion zusammengefügt. Angenommen zwei partielle Funktionen F^1 und F^2 werden durch den Ausdruck $F^1 \ \oplus_{\mathtt{P}} \ F^2$ miteinander verknüpft, dann wird die neue partielle Funktion für Eingaben, die ausschließlich im Definitionsbereich von F^1 liegen, das in F^1 definierte Ergebnis liefern. In allen anderen Fällen wird das in F^2 definierte Ergebnis zurückgegeben.

$$\begin{aligned} &\_ \oplus \_ \ ::_{\mathsf{c}} \ [\sigma \rightharpoonup\!\!\!\!\!+ \ (\sigma, \epsilon, \delta) \ \mathit{seqauto} \ \mathit{set}, \ \sigma * (\sigma, \epsilon, \delta) \ \mathit{seqauto}] \\ &\qquad\qquad \to (\sigma \rightharpoonup\!\!\!\!\!+ \ (\sigma, \epsilon, \delta) \ \mathit{seqauto} \ \mathit{set}) \\ &\_ \oplus \_ \ \equiv_{\mathsf{c}} \ (\lambda \ F_{\mathtt{Comp}} \ \ (S, \mathit{SA}) \ . \\ &\qquad\qquad\quad \mathsf{if} \ \ (S \in \mathit{dom} \ F_{\mathtt{Comp}} \ \wedge \ S \notin \mathit{States} \ \mathit{SA}) \\ &\qquad\qquad\quad \mathsf{then} \\ &\qquad\qquad\qquad F_{\mathtt{Comp}} \oplus_{\mathtt{P}} \{S \mapsto (\mathit{the} \ (F_{\mathtt{Comp}} \ S)) \cup \{\mathit{SA}\}\} \oplus_{\mathtt{P}} (\mathit{EmptyMap} \ (\mathit{States} \ \mathit{SA})) \\ &\qquad\qquad\quad \mathsf{else} \\ &\qquad\qquad\qquad F_{\mathtt{Comp}}) \end{aligned}$$

$$
\begin{aligned}
&\mathit{CAS} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{hierauto}\\
&\mathit{CAS} \equiv_{\mathsf{c}} (\mathit{PseudoHA}\ \mathit{CarAudioSystemCTRL}\ \mathit{InitDS})\\
&\qquad \boxplus (\mathit{“CarAudioSystem”}, \mathit{AudioPlayer})\\
&\qquad \boxplus (\mathit{“CarAudioSystem”}, \mathit{CDPlayer})\\
&\qquad \boxplus (\mathit{“CarAudioSystem”}, \mathit{TapeDeck})\\
&\qquad \boxplus (\mathit{“On”}, \mathit{AudioCTRL})\\
&\qquad \boxplus (\mathit{“TunerMode”}, \mathit{TunerCTRL})\\
&\qquad \boxplus (\mathit{“TapeMode”}, \mathit{TapeCTRL})\\
&\qquad \boxplus (\mathit{“CDMode”}, \mathit{CDCTRL})
\end{aligned}
$$

Abbildung 3.3: Schrittweise Konstruktion des *Car-Audio-System* - Modells

Unter Verwendung der eingeführten Operatoren in Definition 3.7 und 3.8 haben wir die Beispielspezifikation aus Abschnitt 2.1.1 formalisiert. Abbildung 3.3 zeigt mit der Konstantendefinition *CAS* die schrittweise Konstruktion des Hierarchischen Automaten. Die Konstante *InitDS* repräsentiert eine Datenraumbelegung, durch die die Datenvariable T mit 0 und die Datenvariable $T_{\mathtt{A}}$ mit 1 initialisiert wird. Für eine detaillierte Beschreibung der Formalisierung von Datenraumbelegungen verweisen wir auf Kapitel 4.

Der Gewinn einer solchen Konstruktion liegt darin, dass unter Anwendung geeigneter Theoreme der Nachweis der Wohlgeformtheitseigenschaft (vgl. Definition 3.5) sehr effizient und häufig sogar vollautomatisch erbracht werden kann. Folglich ist auch die Ableitung von Selektionstheoremen für einen konstruierten Hierarchischen Automaten automatisch möglich.

Um diesen Grad an Automatisierung zu erreichen, haben wir in Isabelle/HOL eine Taktik implementiert, mit der die Selektionstheoreme für einen konstruierten Hierarchischen Automaten automatisch abgeleitet werden können. Hierfür zerlegt die Taktik schrittweise einen konstruierten Hierarchischen Automaten in seine Komponenten und leitet separat Selektionstheoreme für die einzelnen Komponenten ab. Anschließend werden diese Selektionstheoreme zu einem Selektionstheorem des konstruierten Hierarchischen Automaten zusammengefügt. Um eine schrittweise Zerlegung möglich zu machen, ist es zusätzlich notwendig zu zeigen, dass die Konstruktion die Wohlgeformtheitseigenschaften eines Hierarchischen Automaten respektiert. Im Folgenden wollen wir einige Theoreme vorstellen, die von dieser Taktik verwendet werden.

Für die Konstruktion eines Pseudo-HA konnten wir zeigen, dass seine definierenden Komponenten die Wohlgeformtheit ohne weitere Annahmen immer erfüllen.

Abgeleitetes Theorem 3.9 (Wohlgeformtheit von Pseudo-HA)

$$\frac{S_{\mathtt{A}}^{'} = \{\mathit{SA}\} \qquad E^{'} = \mathit{Events}\ \mathit{SA} \qquad F_{\mathtt{Comp}}^{'} = \mathit{EmptyMap}\ (\mathit{States}\ \mathit{SA})}{\mathit{HierAutoCorrect}\ S_{\mathtt{A}}^{'}\ E^{'}\ F_{\mathtt{Comp}}^{'}\ D} \quad \text{(PseudoHACorrect)}$$

□

Auf Basis dieses Theorems können weitere Theoreme abgeleitet und zur Selektion von Komponenten eines Pseudo-HA genutzt werden.

Abgeleitete Theoreme 3.10 (Selektionstheoreme für Pseudo-HA)

$$SAs\ (PseudoHA\ SA\ D) = \{SA\} \qquad \text{(PseudoHASelectSAs)}$$

$$Events\ (PseudoHA\ SA\ D) = Events\ SA \qquad \text{(PseudoHASelectEvents)}$$

$$CompFun(PseudoHA\ SA\ D) = EmptyMap(States\ SA) \qquad \text{(PseudoHASelectCompFun)}$$

$$InitData\ (PseudoHA\ SA\ D) = D \qquad \text{(PseudoHASelectInitData)}$$

□

Für die Konstruktion mit dem Operator $\boxplus$ konnten wir analog zu Theorem 3.9 nachweisen, dass die Wohlgeformtheitseigenschaft erfüllt ist. Allerdings gilt diese Eigenschaft nur unter einigen Annahmen an die Konstruktion.

Abgeleitetes Theorem 3.11 (Wohlgeformtheit von $\boxplus$ -Konstruktionen)

$$\frac{\begin{array}{c} States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA \\ S_A^{'} = \{\ SA\ \} \cup (SAs\ HA) \qquad E^{'} = Events\ HA\ \cup Events\ SA \\ F_{Comp}^{'} = CompFun\ HA \oplus (S, SA) \qquad D^{'} = InitData\ HA \end{array}}{HierAutoCorrect\ S_A^{'}\ E^{'}\ F_{Comp}^{'}\ D^{'}} \qquad \text{(HAAddSACorrect)}$$

□

Entsprechend sind die Selektionstheoreme bei einer Konstruktion mit dem Operator $\boxplus$ nur unter diesen Annahmen anwendbar.

Abgeleitete Theoreme 3.12 (Selektionstheoreme für $\boxplus$ -Konstruktionen)

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{SAs\ (HA \boxplus (S, SA)) = \{SA\} \cup SAs\ HA} \qquad \text{(HAAddSASelectSAs)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{Events\ (HA \boxplus (S, SA)) = Events\ SA \cup Events\ HA} \qquad \text{(HAAddSASelectEvents)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{CompFun\ (HA \boxplus (S, SA)) = CompFun\ HA \oplus (S, SA)} \qquad \text{(HAAddSASelectCompFun)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{InitData\ (HA \boxplus (S, SA)) = InitData\ HA} \qquad \text{(HAAddSASelectInitData)}$$

□

Die von uns entwickelte Taktik versucht für einen mit Konstruktionsoperatoren aufgebauten Hierarchischen Automaten, die Selektionstheoreme abzuleiten. Folgendes Theorem wurde beispielweise mit der Taktik für den Hierarchischen Automaten aus Abbildung 3.3 abgeleitet.

Abgeleitetes Theorem 3.13 (SAs des *Car-Audio-System*- Modells)

$$\begin{aligned} \mathit{SAs\ CAS} = \{&\mathit{CarAudioSystemCTRL}, \mathit{AudioPlayer}, \mathit{CDPlayer}, \mathit{TapeDeck},\\ &\mathit{AudioCTRL}, \mathit{TunerCTRL}, \mathit{CDCTRL}, \mathit{TapeCTRL}\} \end{aligned} \tag{SelectSAsCAS}$$

□

Um dieses Theorem abzuleiten, wird zunächst folgende Beweisverpflichtung für die Selektion von Sequentiellen Automaten auf der entfalteten *CAS*- Konstanten erzeugt.

$$\mathit{SAs}\ ((\mathit{PseudoHA\ CarAudioSystemCTRL\ InitDS}) \boxplus \ \ldots\ \boxplus\ \ldots) = \ ?\mathsf{Result}$$

Der Bezeichner ?Result repräsentiert hierbei eine Metavariable, die während des zu führenden Beweises schrittweise instanziiert wird. Zunächst wendet die Taktik das Theorem HAAddSASelectSAs durch Rückwärtsresolution sieben mal auf die Beweisverpflichtung an. Jede dieser Anwendungen erfordert es, dass durch die Taktik auch die Annahmen von Theorem HAAddSASelectSAs nachgewiesen werden können. War dies erfolgreich, kann durch die einmalige Anwendung des Theorems PseudoHASelectSAs der Beweis abgeschlossen werden.

Zusätzlich zu dem Konstruktionsoperator $\boxplus$ definieren wir einen Konstruktionsoperator $\boxdot$, der einen Hierarchischen Automaten an einem Zustand durch einen anderen Hierarchischen Automaten verfeinert.

Definition 3.14 (Erweiterung eines HA durch einen HA) *Sei S ein Zustand, HA^1 und HA^2 Hierarchische Automaten, dann ist die Erweiterung von HA^1 durch HA^2 im Zustand S folgendermaßen definiert.*

$$\begin{aligned} &\_\boxdot\_ ::_{\mathsf{c}} [(\sigma,\epsilon,\delta)\,\mathit{hierauto}, \sigma * (\sigma,\epsilon,\delta)\,\mathit{hierauto}]\\ &\qquad \rightarrow (\sigma,\epsilon,\delta)\,\mathit{hierauto}\\ &\_\boxdot\_ \equiv_{\mathsf{c}} (\lambda\, \mathit{HA}^1\ (S, \mathit{HA}^2)\ .\\ &\qquad \mathsf{let}\ (S^1_{\mathtt{A}}, E^1, F^1_{\mathtt{Comp}}, D^1) = \mathit{Rep\_hierauto}(\mathit{HA}^1 \boxplus (S, \mathit{Root}_{\mathtt{SA}}\, \mathit{HA}^2))\\ &\qquad\quad (S^2_{\mathtt{A}}, E^2, F^2_{\mathtt{Comp}}, D^2) = \mathit{Rep\_hierauto}\, \mathit{HA}^2\\ &\qquad\qquad S^{'}_{\mathtt{A}} = S^1_{\mathtt{A}} \cup S^2_{\mathtt{A}};\\ &\qquad\qquad E^{'} = E^1 \cup E^2;\\ &\qquad\qquad F^{'}_{\mathtt{Comp}} = F^1_{\mathtt{Comp}} \uplus F^2_{\mathtt{Comp}}\\ &\qquad \mathsf{in}\ \ \mathit{Abs\_hierauto}(S^{'}_{\mathtt{A}}, E^{'}, F^{'}_{\mathtt{Comp}}, D^1)) \end{aligned} \tag{3.6}$$

□

Auch für diesen Konstruktionsoperator haben wir Theoreme zur Wohlgeformtheit und zur Selektion von einzelnen Komponenten angegeben, die in Anhang A.3 dieser Ausarbeitung aufgeführt sind.

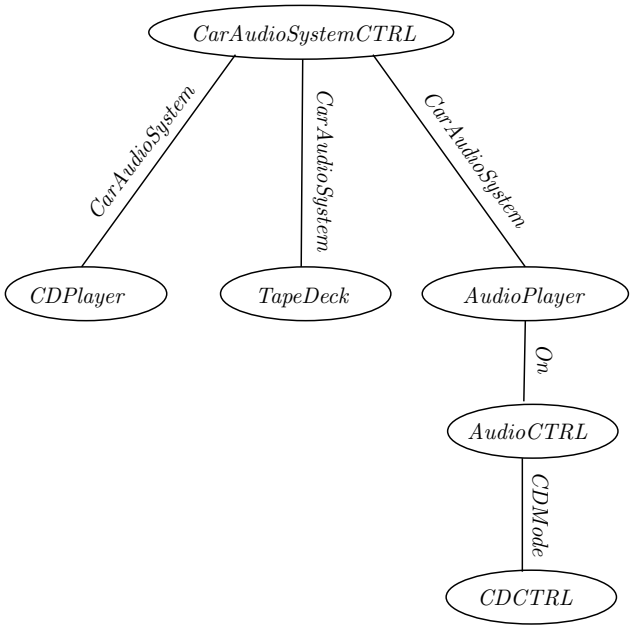


Abbildung 3.4: Kompositionsbaum am Beispiel des *Car-Audio-System*-Modells

3.2.2 Kompositionsbäume zur Repräsentation von Kompositionsfunktionen

In diesem Abschnitt soll eine zweite Möglichkeit aufgezeigt werden, wie die baumartige Struktur eines Hierarchischen Automaten in einer Formalisierung explizit dargestellt werden kann.

Die Grundidee besteht darin, die Kompositionsfunktion aus Definition 3.3 nicht mehr durch eine partielle Funktion, sondern durch einen *Kompositionsbaum* zu repräsentieren. Wir haben dazu folgenden primitiv-rekursiven Datentyp $(\sigma, \epsilon, \delta)$ `comptree` definiert.

$$\begin{aligned}(\sigma,\epsilon,\delta)\,\texttt{comptree} \;\equiv_{\mathtt{t}}\; & \mathit{Node}_{\mathtt{T}}\ (\sigma,\epsilon,\delta)\texttt{seqauto} \\ & (\sigma \rightarrow ((\sigma,\epsilon,\delta)\texttt{comptree list}))\end{aligned}$$

Der Datentyp repräsentiert einen Baum, dessen Knoten jeweils aus einem Sequentiellen Automaten und einer *Nachfolgefunktion* bestehen. Hierbei ordnet die Funktion den Zuständen des Sequentiellen Automaten eine Liste von Bäumen zu, an denen der Sequentiellen Automat verfeinert wird. Entsprechend bildet die Funktion Basis - Zustände auf die leere Liste, Oder - Zustände auf einelementige Listen und Und - Zustände auf mehrelementige Listen ab. Dieses Vorgehen entspricht in etwa der schrittweisen Konstruktion eines Hierarchischen Automaten mit Hilfe von Konstruktionsoperatoren aus Abschnitt 3.2.1. Da sich jetzt aber die baumartige Struktur Hierarchischer Automaten im Typ selbst und nicht nur in der Konstruktionsvorschrift widerspiegelt, sind wir in der Lage, Beweise noch effizienter zu gestalten. Wir nutzen hierbei aus, dass primitiv-rekursive Datentypen in Isabelle/HOL durch Induktionstaktiken unterstützt sind, die im Allgemeinen das Beweisvorgehen erheblich erleichtern.

In Abbildung 3.4 ist der Kompositionsbaum für das *Car-Audio-System* - Modell aus Abschnitt 2.1.1 veranschaulicht. Ein Knoten ist in der Abbildung mit seinem, ihn definierenden Sequentiellen Automaten markiert. Die Funktion eines Knotens ist durch herausführende

Kanten graphisch dargestellt. Hierbei entspricht jede Kante einer durch die Funktion definierten Verfeinerung. Die Markierung an einer Kante gibt den verfeinerten Zustand an. BASIS - Zustände wurden in der Abbildung nicht dargestellt.

Analog zu den Wohlgeformtheitseigenschaften einer Kompositionsfunktion (vgl. Definition 3.6) legen wir Wohlgeformtheitseigenschaften für Kompositionsbäume fest.

Definition 3.15 (Wohlgeformter Kompositionsbaum) *Sei $T_{\mathtt{Comp}}$ vom rekursiven Datentyp $(\sigma, \epsilon, \delta)$ `comptree`, dann ist $T_{\mathtt{Comp}}$ ein wohlgeformter Kompositionsbaum, wenn folgende Bedingungen erfüllt sind.*

$$\begin{aligned} \mathit{IsCompTree} &::_{\mathtt{c}} (\sigma, \epsilon, \delta)\ \mathtt{comptree} \to \mathtt{bool} \\ \mathit{IsCompTree}\ T_{\mathtt{Comp}} &\equiv_{\mathtt{c}} \mathit{MutuallyDistinct}_{\mathtt{T}}\ T_{\mathtt{Comp}} \ \wedge \\ &\quad\ \mathit{RootExists}_{\mathtt{T}}\ T_{\mathtt{Comp}} \end{aligned}$$

Die geforderten Eigenschaften werden als primitiv-rekursive Funktionen definiert.

- *Das Prädikat $\mathit{MutuallyDistinct}_{\mathtt{T}}$ garantiert, dass die Zustandsmengen aller in $T_{\mathtt{Comp}}$ vorkommenden Sequentiellen Automaten paarweise disjunkt sind.*

$$\begin{aligned} &\mathit{MutuallyDistinct}_{\mathtt{T}}\ (\mathit{Node}_{\mathtt{T}}\ \mathit{SA}\ F) = \\ &\qquad \mathit{States}\ \mathit{SA} \cap \Big(\bigcup S : \mathit{States}\ \mathit{SA}.\ \mathit{States}_{\mathtt{TL}}\ (F\ S)\Big) = \varnothing \ \wedge \\ &\qquad \forall\, S_1, S_2 : \mathit{States}\ \mathit{SA}. \\ &\qquad\qquad S_1 \neq S_2 \Rightarrow \mathit{States}_{\mathtt{TL}}\ (F\ S_1) \cap \mathit{States}_{\mathtt{TL}}\ (F\ S_2) = \varnothing \ \wedge \\ &\qquad \forall\, S : \mathit{States}\ \mathit{SA}.\ \mathit{MutuallyDistinct}_{\mathtt{TL}}\ (F\ S) \end{aligned}$$

$$\begin{aligned} \mathit{MutuallyDistinct}_{\mathtt{TL}}\ [\,] &= \mathit{true} \\ \mathit{MutuallyDistinct}_{\mathtt{TL}}\ (T \# L) &= \\ &\qquad \mathit{States}_{\mathtt{T}}\ T \cap \mathit{States}_{\mathtt{TL}}\ L = \varnothing \ \wedge \\ &\qquad \mathit{MutuallyDistinct}_{\mathtt{T}}\ T \ \wedge \\ &\qquad \mathit{MutuallyDistinct}_{\mathtt{TL}}\ L \end{aligned}$$

- *Das Prädikat $\mathit{RootExists}_{\mathtt{T}}$ garantiert, dass es in $T_{\mathtt{Comp}}$ einen eindeutigen Wurzelzustand gibt. Die formalen Definitionen von $\mathit{RootExists}_{\mathtt{T}}$, $\mathit{States}_{\mathtt{T}}$ und $\mathit{States}_{\mathtt{TL}}$ sind in Anhang A.4 zu finden.*

□

Primitiv-rekursive Definitionen, wie wir sie z.B. für die Konstante $\mathit{MutuallyDistinct}_{\mathtt{T}}$ in Definition 3.15 eingesetzt haben, sind in dieser Ausarbeitung in Abschnitt 2.3.2 auf Seite 39 eingeführt worden. Sie bestehen aus einer Regelmenge rekursiver Gleichungen. Hierbei wird mit jeder vorkommenden Gleichung ein Konstruktor des zugrunde liegenden rekursiven Datentyps behandelt. Für den Operator $\mathit{MutuallyDistinct}_{\mathtt{T}}$ ist entsprechend nur eine Regel für den Konstruktor $\mathit{Node}_{\mathtt{T}}$ erforderlich. Da allerdings der von uns eingeführte Datentyp `comptree` in seiner Definition zusätzlich den primitiv-rekursiven Datentyp `list` verwendet, ist es notwendig, eine weitere Konstante $\mathit{MutuallyDistinct}_{\mathtt{TL}}$ einzuführen, die auf Listen von Kompositionsbäumen operiert. Auf Basis dieser Konstante wird die Regelmenge entsprechend um Gleichungen für die Konstruktoren des Datentyps `list` erweitert. Hierbei ist die

leere Liste durch den Konstruktor [] und das Einfügen eines Elements am Anfang der Liste durch den Konstruktor # repräsentiert. Mit den Operatoren $States_{\mathtt{T}}$ und $States_{\mathtt{TL}}$ können die in einem Kompositionsbaum bzw. in einer Liste von Kompositionsbäumen vorkommenden Zustände zurückgegeben werden.

Basierend auf der Definition eines wohlgeformten Kompositionsbaumes geben wir nun die baumbasierte Variante eines Hierarchischen Automaten an.

Definition 3.16 (Baumbasierter Hierarchischer Automat ($\mathsf{HA}_{\mathtt{T}}$)) *Sei σ ein Typ von Zustandsbezeichnern, ϵ ein Typ von Ereignisbezeichnern und δ ein Typ für den Datenraum, dann ist ein baumbasierter Hierarchischer Automat $HA_{\mathtt{T}}$ über $(\sigma, \epsilon, \delta)$ durch ein Tripel $(E, T_{\mathtt{Comp}}, D)$ repräsentiert. Hierbei ist*

- *E die Menge von Ereignissen,*
- *$T_{\mathtt{Comp}}$ der Kompositionsbaum und*
- *D die initiale Datenraumbelegung*

des Hierarchischen Automat $HA_{\mathtt{T}}$. Diese Komponenten müssen die interne Konsistenzbedingung $HierAutoCorrect_{\mathtt{T}}$ für Hierarchische Automaten erfüllen, die in Definition 3.17 angegeben ist. Der Typ $(\sigma, \epsilon, \delta)\, hierauto_{\mathtt{T}}$ besteht aus allen Hierarchischen Automaten über $(\sigma, \epsilon, \delta)$.

$$
\begin{aligned}
(\sigma, \epsilon, \delta)\, hierauto_{\mathtt{T}} \equiv_{\mathtt{t}} \{\, & (E, T_{\mathtt{Comp}}, D) \mid \\
& (E :: \epsilon\ set) \\
& (T_{\mathtt{Comp}} :: ((\sigma, \epsilon, \delta)\ comptree)) \\
& (D :: \delta\ data). \\
& HierAutoCorrect_{\mathtt{T}}\ E\ F_{\mathtt{Comp}}\ D \,\} \\
& \quad \textit{gerechtfertigt durch } \mathsf{HierAutoTreeNonEmpty}
\end{aligned}
\tag{3.7}
$$

□

Um auf die Komponenten eines baumbasierten Hierarchischen Automaten zugreifen zu können, führen wir folgende Selektionsoperatoren ein. Die Menge an Sequentiellen Automaten kann durch den Operator $SAs_{\mathtt{T}}$, die Ereignismenge durch den Operator $Events_{\mathtt{T}}$, der Kompositionsbaum durch den Operator $CompTree$ und die initiale Datenraumbelegung durch den Operator $InitData_{\mathtt{T}}$ selektiert werden. Man beachte, dass die Menge an Sequentiellen Automaten im Gegensatz zur Definition 3.4 für Hierarchische Automaten nicht mehr explizit in der Repräsentation des baumbasierten Hierarchischen Automaten verwaltet, sondern bei einer Selektion aus dem Kompositionsbaum errechnet wird.

Definition 3.17 (Wohlgeformter baumbasierter Hierarchischer Automat) *Sei E eine Menge von Ereignissen, $T_{\mathtt{Comp}}$ ein Kompositionsbaum vom Typ $\mathtt{comptree}$ und D eine initiale Belegung des Datenraumes, dann bilden diese Komponenten einen wohlgeformten baumbasierten Hierarchischen Automaten, wenn folgende zwei Bedingungen erfüllt sind.*

$$\begin{aligned} &\mathit{HierAutoCorrect}_{\mathtt{T}} ::_{\mathtt{c}} [\epsilon\ \mathit{set}, (\sigma, \epsilon, \delta)\ \mathit{comptree}, \delta\ \mathit{data}] \\ &\qquad \rightarrow \mathit{bool} \\ &\mathit{HierAutoCorrect}_{\mathtt{T}}\ E\ T_{\mathtt{Comp}}\ D \equiv_{\mathtt{c}} \left(\bigcup A \in (\mathit{SAs}_{\mathtt{T}}\ T_{\mathtt{Comp}})\,.\ \mathit{Events}\ F\right) \subseteq E \wedge \\ &\qquad \mathit{IsCompTree}\ T_{\mathtt{Comp}} \end{aligned}$$

Die Eigenschaft $\mathtt{IsCompTree}$ fordert die Wohlgeformtheitseigenschaft für Kompositionsbäume $T_{\mathtt{Comp}}$ und ist in Definition 3.15 angegeben. □

Mit der Definition baumbasierter Hierarchischer Automaten haben wir im Vergleich mit der ursprünglichen Definition Hierarchischer Automaten eine bessere Basis geschaffen, um neue Theoreme innerhalb der vorgeschlagenen Theorie mit vertretbarem Aufwand ableiten zu können. So ist beispielsweise der Beweis der Wohlgeformtheit für einen baumbasierten Hierarchischen Automaten leichter zu führen. Da im Theorembeweiser Isabelle/HOL primitivrekursive Datentypen durch mächtige Induktionstaktiken unterstützt werden, sind auch andere Theoreme mit allgemeineren Aussagen einfacher abzuleiten. So haben wir – analog zu Abschnitt 3.2.1 – auf Basis baumbasierter Hierarchischer Automaten Konstruktionsoperatoren definiert. Die Beweisführung zur Ableitung der Wohlgeformtheits- und Selektionstheoreme für baumbasierte Konstruktionsoperatoren ist deutlich einfacher im Vergleich zu der Beweisführung für die korrespondierenden Theoreme in der funktionsbasierten Formalisierung von Hierarchische Automaten (vgl. Theoreme 3.12 und 3.11).

Ein Nachteil der baumbasierten Formalisierung Hierarchischer Automaten sind die schwer überschaubaren Konstantendefinitionen (vgl. Definition 3.15), die aufgrund der primitivrekursiven Datentypen sehr technisch ausfallen. Um garantieren zu können, dass die effizientere Formalisierung mit primitiv-rekursiven Datentypen die von Erich Mikk intendierten Eigenschaften der ursprünglichen Formalisierung Hierarchischer Automaten widerspiegelt, geben wir in Definition 3.18 einen Operator $\mathit{Lift}_{\mathtt{HA}}$ an. Mit diesem Operator werden aus einem baumbasierten Hierarchischen Automaten die Komponenten für einen funktionsbasierten Hierarchischen Automaten konstruiert. Anschließend zeigen wir, dass die konstruierten Komponenten die Wohlgeformtheitseigenschaften eines Hierarchischen Automaten aus Definition 3.5 auf Seite 47 erfüllen.

Definition 3.18 (Transformation eines HA_{T} zu einem HA) *Sei $\mathit{HA}_{\mathtt{T}}$ ein baumbasierter Hierarchischer Automat, so ist die Konstruktion eines Hierarchischen Automaten durch folgenden Operator beschrieben.*

$$\begin{aligned} \mathit{Lift}_{\mathtt{HA}} &::_{\mathtt{c}} (\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathtt{T}} \rightarrow (\sigma, \epsilon, \delta)\ \mathit{hierauto} \\ \mathit{Lift}_{\mathtt{HA}}\ \mathit{HA}_{\mathtt{T}} &\equiv_{\mathtt{c}} \mathsf{let}\ S_{\mathtt{A}}^{'} = \mathit{SAs}_{\mathtt{T}}\ \mathit{HA}_{\mathtt{T}}; \\ &\qquad E^{'} = \mathit{Events}_{\mathtt{T}}\ \mathit{HA}_{\mathtt{T}}; \\ &\qquad F_{\mathtt{Comp}}^{'} = \mathit{Lift}_{\mathtt{CF}}\ (\mathit{CompTree}\ \mathit{HA}_{\mathtt{T}}); \\ &\qquad D^{'} = \mathit{InitData}_{\mathtt{T}}\ \mathit{HA}_{\mathtt{T}} \\ &\quad \mathsf{in}\ \mathit{Abs\_hierauto}(S_{\mathtt{A}}^{'}, E^{'}, F_{\mathtt{Comp}}^{'}, D^{'}) \end{aligned}$$

Hierbei beschreibt der Operator $Lift_{\mathrm{CF}}$ *die Transformation eines Kompositionsbaumes in eine Kompositionsfunktion und ist in Definition 3.19 angegeben.* □

Die Transformation $Lift_{\mathrm{CF}}$ konstruiert für einem Kompositionsbaum eine partielle Funktion, die von einem Zustand S des baumbasierten Hierarchischen Automaten auf die Menge von Sequentiellen Automaten abbildet, durch die S direkt verfeinert wird. Wenn im Kompositionsbaum keine Sequentiellen Automaten für einen Zustand gefunden werden, so wird der Funktionswert an dieser Stelle undefiniert. Die Anwendung einer Funktion auf alle Elemente einer Liste ist durch den Operator *map* und die Transformation einer Liste in eine Menge durch den Operator *set* repräsentiert. Ferner verwenden wir den Auswahloperator @ von Hilbert, um einen Zustand S' zu wählen, der einerseits in der Zustandsmenge des Sequentiellen Automaten SA enthalten ist und der andererseits S als direkten Unterzustand besitzt. Sollte kein S' existieren, das diese Eigenschaften erfüllt, so liefert der Auswahloperator von Hilbert ein beliebiges unbekanntes Element aus dem Grundtypen von S' zurück.

Analog zur primitiv-rekursiven Konstantendefinition von $MutuallyDistinct_{\mathrm{T}}$ aus Definition 3.15 ist der Operator $Lift_{\mathrm{CF}}$ mit Hilfe eines zusätzlichen Operators $Lift_{\mathrm{CFL}}$ definiert, der auf Listen von Kompositionsbäumen operiert. Beide Operatoren rufen sich wechselseitig auf.

Definition 3.19 (Transformation eines Kompositionsbaumes in eine Funktion)
Sei T_{Comp} *ein Kompositionsbaum, so ist die Transformation* $Lift_{\mathrm{CF}}\ T_{\mathrm{Comp}}$ *zu einer Kompositionsfunktion durch folgende primitiv-rekursive Konstantendefinition festgelegt.*

$$
\begin{aligned}
&Lift_{\mathrm{CF}}\ (Node_{\mathrm{T}}\ SA\ F)\ S = \\
&\qquad \mathsf{if}\ S \in States_{\mathrm{T}}\ (Node_{\mathrm{T}}\ SA\ F)\ \mathsf{then} \\
&\qquad\quad \mathsf{if}\ S \in States\ SA\ \mathsf{then} \\
&\qquad\qquad Some(set\,(map\ Root_{\mathrm{SA}}\ (F\ S))) \\
&\qquad\quad \mathsf{else} \\
&\qquad\qquad Lift_{\mathrm{CFL}}\ (F\ (@\,S'.\ S' \in States\ SA\ \wedge\ S \in States_{\mathrm{TL}}\,(F\ S')))\ S \\
&\qquad \mathsf{else}\ None
\end{aligned}
$$

$$
\begin{aligned}
&Lift_{\mathrm{CFL}}\ []\ \_ = None \\
&Lift_{\mathrm{CFL}}\ (T \# L)\ S = \\
&\qquad \mathsf{if}\ S \in States_{\mathrm{T}}\ T\ \mathsf{then} \\
&\qquad\quad Lift_{\mathrm{CFL}}\ L\ S \\
&\qquad \mathsf{else}\ Lift_{\mathrm{CF}}\ T\ S
\end{aligned}
$$

□

Man beachte, dass die Repräsentation einer Kompositionsfunktion als Kompositionsbaum mehrdeutig ist, da der Datentypen *comptree* auf Listen basiert. Deshalb werden im Allgemeinen mehrere Kompositionsbäume existieren, die die selbe Kompositionsfunktion repräsentieren. Damit kann der Operator $Lift_{\mathrm{CF}}$ zur Partitionierung verschiedener Kompositionsbäume in Äquivalenzklassen eingesetzt werden.

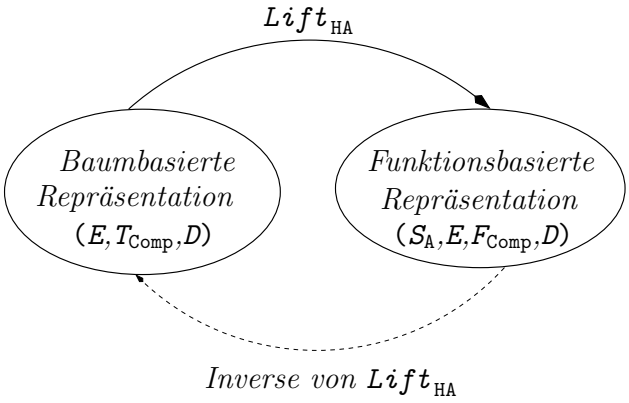


Abbildung 3.5: Transformation zwischen baum- und funktionsbasierten Hierarchischen Automaten

Abbildung 3.5 illustriert, welche Tranformationen zwischen funktions- und baumbasierten Hierarchischen Automaten durch den Operator $Lift_{\mathrm{HA}}$ möglich sind. In unseren Arbeiten wurde in Isabelle/HOL folgendes Theorem formal abgeleitet.

Abgeleitetes Theorem 3.20 (Konsistenz zwischen $\mathsf{HA_T}$ und HA)

$$\frac{SA = SAs_{\mathrm{T}}\ HA_{\mathrm{T}} \qquad E = Events_{\mathrm{T}}\ HA_{\mathrm{T}} \qquad F_{\mathrm{Comp}} = Lift_{\mathrm{CF}}\ (CompTree\ HA_{\mathrm{T}}) \qquad D = InitData_{\mathrm{T}}\ HA_{\mathrm{T}}}{HierAutoCorrect\ SA\ E\ F_{\mathrm{Comp}}\ D} \quad \text{(ConsistencyOfHAs)}$$

□

Mit der Gültigkeit dieses Theorems wird belegt, dass ein baumbasierter Hierarchischer Automat die Eigenschaften des funktionsbasierten Hierarchischen Automaten erfüllt. Würden wir zusätzlich beweisen, dass $Lift_{\mathrm{HA}}$ surjektiv ist, so könnten wir Theoreme, die in der baumbasierten Formalisierung Hierarchischer Automaten effizient abgeleitet wurden, in die funktionsbasierte Formalisierung liften. Wir haben uns in dieser Arbeit allerdings auf die Ableitung des Theorems 3.20 beschränkt, da es lediglich um eine Validierung der effizienteren Formalisierung ging, die die Basis für Beweise auf Statecharts-Spezifikationen bildet. Hingegen war es nicht unser Ziel, eine Beweisinfrastruktur aufzubauen, die mit beiden Formalisierungen gleichzeitig umgehen kann. Aus ähnlichen Gründen wurde die Rückrichtung – in Abbildung 3.5 repräsentiert durch einen gestrichelten Pfeil – von uns noch nicht genauer untersucht.

3.3 Semantik

Der zentrale Baustein in der Semantik Hierarchischer Automaten sind *Status*[3], auf Basis derer semantische Berechnungschritte angegeben werden können. Ähnlich zum vorhergehenden Abschnitt präsentieren wir zunächst eine Formalisierung von Status, die sich an den Arbeiten von Erich Mikk orientiert [MLS97, Mik00]. Darauf aufbauend wird in Abschnitt 3.3.2 eine synchrone Schrittsemantik für Hierarchische Automaten angegeben. In Abschnitt 3.3.3 geben wir eine optimierte Formalisierung an. Die Optimierung wird dadurch erzielt, dass die Menge

[3]In der deutschen Sprache werden Singular und Plural des Wortes *Status* identisch geschrieben.

aktiver Zustände in einem Status durch einen Konfigurationsbaum beschrieben wird, der aufgrund seiner primitiv-rekursiven Eigenschaft beim Führen von Beweisen einen höheren Grad an Automatisierung ermöglicht.

3.3.1 Konfigurationen und Status

In diesem Abschnitt definieren wir semantische Zustände (*Status*) Hierarchischer Automaten. Die Basis der Definition bildet die Typabstraktion über einer Menge, mit der wir einen geeigneten Typ zur Beschreibung von Status angeben.

Definition 3.21 (Status) *Sei σ ein Typ von Zustandsbezeichnern, ϵ ein Typ von Ereignisbezeichnern und δ ein Typ für den Datenraum, dann ist ein Status* `ST` *über $(\sigma, \epsilon, \delta)$ durch ein Quadruple* $(\texttt{HA}, \texttt{E}, \texttt{C}, \texttt{D})$ *repräsentiert. Hierbei ist*

- `HA` *der Hierarchische Automat, in dem der Status möglich ist,*
- `C` *die Menge von aktiven Zuständen (Konfiguration),*
- `E` *die Menge von anliegenden Ereignissen und*
- `D` *die aktuelle Datenraumbelegung*

des Status `ST`*. Diese Komponenten müssen die interne Konsistenzbedingung* `StatusCorrect` *über Status erfüllen, die in Definition 3.22 angegeben ist. Der Typ $(\sigma, \epsilon, \delta)$* `status` *besteht aus allen Status über $(\sigma, \epsilon, \delta)$.*

$$
\begin{aligned}
(\sigma, \epsilon, \delta)\ \texttt{status} \equiv_{\mathrm{t}} \{\, & (\texttt{HA}, \texttt{C}, \texttt{E}, \texttt{D}) \mid \\
& (\texttt{HA} :: (\sigma, \epsilon, \delta)\ \texttt{hierauto}) \\
& (\texttt{C} :: \sigma\ \texttt{set}) \\
& (\texttt{E} :: \epsilon\ \texttt{set}) \\
& (\texttt{D} :: \delta\ \texttt{data}). \\
& \texttt{StatusCorrect HA C E D}\ \} \\
& \quad \textit{gerechtfertigt durch}\ \textsf{StatusNonEmpty}
\end{aligned}
\tag{3.8}
$$

□

Das Theorem StatusNonEmpty belegt, dass die in Definition 3.21 abstrahierte Menge nicht leer ist.

Um auf die Komponenten eines Status zugreifen zu können, führen wir folgende Selektionsoperatoren ein. Der Hierarchische Automat eines Status kann mit dem Operator `HA`, die Konfiguration mit dem Operator `Conf`, die Menge an Ereignissen mit dem Operator `Events` und die aktuelle Datenraumbelegung mit dem Operator `Data` selektiert werden.

Das Prädikat `StatusCorrect` beschreibt die sinnvolle Zusammensetzung der Komponenten eines Status.

Definition 3.22 (Wohlgeformter Status) *Sei* `HA` *ein Hierarchischer Automat,* `C` *eine Menge von aktiven Zuständen,* `E` *eine Menge von Ereignissen und* `D` *eine aktuelle Datenraumbelegung, dann bilden diese Komponenten einen wohlgeformten Status, wenn folgende Bedingungen erfüllt sind.*

$$
\begin{aligned}
&\mathit{StatusCorrect} ::_{\mathsf{c}} [\,(\sigma,\epsilon,\delta)\,\mathit{hierauto},\,\sigma\,\mathit{set},\,\epsilon\,\mathit{set},\,\delta\,\mathit{data}\,] \\
&\qquad\qquad \rightarrow \mathit{bool} \\
&\mathit{StatusCorrect}\ \mathit{HA}\ \mathit{C}\ \mathit{E}\ \mathit{D} \equiv_{\mathsf{c}} \ \mathit{E} \subseteq \mathit{Events}\ \mathit{HA}\ \wedge \\
&\qquad\qquad \mathit{DataSpace}\ \mathit{HA} = \mathit{DataSpace}\ \mathit{D}\ \wedge \\
&\qquad\qquad \mathit{IsConfSet}\ (\mathit{SAs}\ \mathit{HA})\ (\mathit{CompFun}\ \mathit{HA})\ \mathit{C}
\end{aligned}
\tag{3.9}
$$

Die Eigenschaft `IsConfSet` *fordert die Wohlgeformtheit für eine Konfiguration* `C` *bezüglich der Sequentiellen Automaten und der Kompositionsfunktion des Hierarchischen Automaten* `HA` *und ist in Definition 3.23 angegeben.* □

Das Prädikat aus Definition 3.22 garantiert für einen Status *ST*, dass in *ST* nur solche Ereignisse anliegen können, die in der Definition des Hierarchischen Automaten von *ST* auch eingeführt sind. Ferner wird sichergestellt, dass die aktuelle Datenraumbelegung von *ST* sich auf einen Datenraum bezieht, der zu dem Datenraum des Hierarchischen Automaten von *ST* identisch ist. Man beachte, dass in Kapitel 4 eine ausführliche Beschreibung von Datenräumen angegeben ist. In diesem Kapitel wird auch der Operator zum Selektieren des Datenraumes aus einer Datenraumbelegung eingeführt. Die Konfiguration von *ST* erfüllt die in Definition 3.23 angegebenen Eigenschaften.

Definition 3.23 (Wohlgeformte Konfiguration) *Sei* $S_{\mathtt{A}}$ *eine Menge von Sequentiellen Automaten,* $F_{\mathtt{Comp}}$ *eine Kompositionsfunktion und* `C` *eine Menge von aktiven Zuständen, dann bilden diese Komponenten eine wohlgeformte Konfiguration, wenn folgende Bedingungen erfüllt sind.*

$$
\begin{aligned}
&\mathit{IsConfSet} ::_{\mathsf{c}} [\,(\sigma,\epsilon,\delta)\,\mathit{seqauto}\,\mathit{set},\,\sigma \nrightarrow ((\sigma,\epsilon,\delta)\,\mathit{seqauto}\,\mathit{set}),\,\sigma\,\mathit{set}\,] \\
&\qquad\qquad \rightarrow \mathit{bool} \\
&\mathit{IsConfSet}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}\ \mathit{C} \equiv_{\mathsf{c}} \ \mathit{C} \subseteq \left(\bigcup \mathit{A} \in S_{\mathtt{A}} .\ \mathit{States}\ \mathit{A}\right)\ \wedge \\
&\qquad\qquad \mathit{RootExists}_{\mathtt{C}}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}\ \mathit{C}\ \wedge \\
&\qquad\qquad \mathit{DownwardClosure}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}\ \mathit{C}
\end{aligned}
$$

- *Das Prädikat* $\mathit{RootExists}_{\mathtt{C}}$ *garantiert, dass ein eindeutiger Zustand aus dem Wurzelautomat in der Konfiguration enthalten ist.*

$$
\mathit{RootExists}_{\mathtt{C}}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}}\ \mathit{C} \equiv_{\mathsf{c}} \exists_1 S : \mathit{C}.\ S \in \mathit{States}(\mathit{Root}_{\mathtt{SA}}\ S_{\mathtt{A}}\ F_{\mathtt{Comp}})
$$

- *Das Prädikat* `DownwardClosure` *garantiert, dass für jeden Sequentiellen Automaten* `A`*, der einen Zustand* `S` *aus* `C` *verfeinert,* `C` *auch einen eindeutigen Zustand* `S'` *aus* `A` *enthält. Zusätzlich wird gefordert, dass für einen Zustand* `S`*, der nicht in* `C` *enthalten ist, auch*

die Unterzustände von S *nicht in* C *vorkommen.*

$$\begin{aligned} \mathit{DownwardClosure}\ S_{\mathrm{A}}\ F_{\mathrm{Comp}}\ C \equiv_{\mathsf{c}} \forall\, S : \Big(\bigcup A \in S_{\mathrm{A}}.\ \mathit{States}\ S_{\mathrm{A}}\Big)\ .\\ \forall\, A : \mathit{the}(F_{\mathrm{Comp}}\ S).\\ \quad \mathsf{if}\ S \in C\ \mathsf{then}\\ \quad\quad \exists_1 S'.\ S' \in \mathit{States}\ A \wedge S' \in C\\ \quad \mathsf{else}\\ \quad\quad \forall\, S : \mathit{States}\ A.\ S \notin C \end{aligned}$$

□

Mit Definition 3.21 haben wir den zentralen Baustein der Semantik Hierarchischer Automaten in Form eines *Status* angegeben. Darauf aufbauend ist im folgenden Abschnitt die synchrone Schrittsemantik für Hierarchische Automaten formalisiert.

3.3.2 Synchrone Schrittsemantik

Die Konfiguration und entsprechend auch der Status eines Hierarchischen Automaten kann sich verändern, wenn der Hierarchische Automat – angestoßen durch eine fiktive globale Uhr – einen synchronen Zeitschritt ausführt. Zu Beginn eines Zeitschrittes werden maximale Mengen konfliktfreier Transitionen bestimmt, eine von diesen Transitionsmengen nichtdeterministisch ausgewählt und auf Basis dieser die Folgekonfiguration bzw. der Folgestatus errechnet. Um maximale Mengen von konfliktfreien Transitionen für einen gegebenen Status bestimmen zu können, benötigen wir eine Reihe von Definitionen, die im Folgenden eingeführt werden sollen.

Definition 3.24 (Nachfolger-Relation für Zustände (χ)) *Sei* HA *ein Hierarchischer Automat, dann sind die Nachfolger der Zustände von* HA *durch die Nachfolger-Relation* χ *festgelegt.*

$$\begin{aligned} \chi &::_{\mathsf{c}} (\sigma, \epsilon, \delta)\ \mathit{hierauto} \to (\sigma * \sigma)\ \mathit{set}\\ \chi\ HA &\equiv_{\mathsf{c}} \{(S, S').\ S \in \mathit{States}\ HA \ \wedge\\ &\qquad \exists\, A : \mathit{SAs}\ HA.\ A \in \mathit{the}(\mathit{CompFun}\ HA\ S) \ \wedge\\ &\qquad\quad S' \in \mathit{States}\ A\} \end{aligned}$$

Durch die Relationen χ^+ *und* χ^* *sei der transitive und der transitiv-reflexive Abschluss von* χ *für einen gegebenen* HA *bezeichnet.* □

Definition 3.24 legt eine Relation χ für einen Hierarchischen Automaten HA fest, mit der jedem Zustand S des HA seine direkten Nachfolgezustände zugeordnet werden können. Alle Nachfolgezustände von S sind mit Hilfe der Kompositionsfunktion aus den direkten Sequentiellen Unterautomaten von S abgeleitet.

Definition 3.25 (Priorität von Transitionen) *Sei* `HA` *ein Hierarchischer Automat und seien* `t` *und* `t'` *zwei Transitionen aus* `HA`*, dann gilt das Prädikat* `HigherPriority`*, falls* `t` *eine höhere Priorität als* `t'` *besitzt.*

$$\begin{aligned} \mathtt{HigherPriority} &::_{\mathsf{c}} [(\sigma,\epsilon,\delta)\,\mathtt{hierauto}, (\sigma,\epsilon,\delta)\,\mathtt{trans} * (\sigma,\epsilon,\delta)\,\mathtt{trans}] \\ &\quad \rightarrow \mathtt{bool} \\ \mathtt{HigherPriority\ HA} &\equiv_{\mathsf{c}} \lambda(t,t').\ t \in \mathtt{Trans\ HA} \ \wedge\ t' \in \mathtt{Trans\ HA} \ \wedge \\ &\qquad (\mathtt{Source}\ t, \mathtt{Source}\ t') \in (\chi^{+}\ \mathtt{HA}) \end{aligned}$$

□

Definition 3.25 legt fest, unter welcher Bedingung eine Transition `t` eine höhere Priorität als eine Transition `t'` besitzt. Dies ist immer dann der Fall, wenn der Startzustand von `t'` ein direkter oder indirekter Nachfolgezustand des Startzustandes von `t` ist.

Definition 3.26 (Aktivierte Transitionen (AT)) *Sei* `ST` *ein Status, so sei die Menge von aktivierten Transitionen in* `ST` *durch* `AT` *festgelegt.*

$$\begin{aligned} \mathtt{AT} &::_{\mathsf{c}} (\sigma,\epsilon,\delta)\,\mathtt{status} \rightarrow (\sigma,\epsilon,\delta)\,\mathtt{trans\,set} \\ \mathtt{AT\ ST} &\equiv_{\mathsf{c}} \bigcup A \in \mathtt{SAs\ ST}.\ \mathtt{EnabledTrans\ ST\ A} \end{aligned}$$

Hierbei beschreibt `EnabledTrans` *die Menge an aktivierten Transitionen in* `ST` *für einen Sequentiellen Automaten* `SA`*.*

$$\begin{aligned} \mathtt{EnabledTrans} &::_{\mathsf{c}} [(\sigma,\epsilon,\delta)\,\mathtt{status}, (\sigma,\epsilon,\delta)\,\mathtt{seqauto}] \\ &\quad \rightarrow (\sigma,\epsilon,\delta)\,\mathtt{trans\,set} \\ \mathtt{EnabledTrans\ S\ SA} &\equiv_{\mathsf{c}} \{\, T.\ \mathtt{Source}\ T \in \mathtt{Conf\ ST} \ \wedge \\ &\qquad (\mathtt{Conf\ ST}, \mathtt{Events\ ST}, \mathtt{Data\ ST}) \models \mathtt{Label}\ T \,\} \end{aligned}$$

Für eine aktivierte Transition gilt, dass ihr Startzustand in der aktuellen Konfiguration von `ST` *liegt und dass Trigger, Condition und Guard aus dem Label der Transition durch die in* `ST` *aktiven Zustände, anliegenden Ereignisse und die aktuelle Datenraumbelegung erfüllt werden.* □

Definition 3.27 (Maximale Mengen konfliktfreier Transitionen (MTS)) *Sei* `ST` *ein Status, so seien alle möglichen maximalen konfliktfreien Transitionsmengen durch* `MTS` *beschrieben.*

$$\begin{aligned} \mathtt{MTS} &::_{\mathsf{c}} (\sigma,\epsilon,\delta)\,\mathtt{status} \rightarrow ((\sigma,\epsilon,\delta)\,\mathtt{trans\,set})\,\mathtt{set} \\ \mathtt{MTS\ ST} &\equiv_{\mathsf{c}} \{\, T.\ \mathtt{MaxNonConflict\ ST\ T} \} \end{aligned}$$

Hierbei garantiert das Prädikat `MaxNonConflict` *die Einhaltung der Prioritätsregeln auf aktivierten Transitionen, die Beachtung des Maximalitätskriteriums und die Konfliktfreiheit der*

Transitionen.

$$\begin{aligned} \mathit{MaxNonConflict} ::_{\mathsf{c}} & [(\sigma,\epsilon,\delta)\,\mathit{status}, (\sigma,\epsilon,\delta)\,\mathit{trans\,set}] \\ & \rightarrow \mathit{bool} \\ \mathit{MaxNonConflict}\,\mathit{ST}\,T \equiv_{\mathsf{c}} & T \subseteq \mathit{AT}\,\mathit{ST} \wedge \\ & \forall\, A : \mathit{SAs}\,\mathit{ST}.\ \#(T \cap \mathit{Trans}\,A) \leq 1 \\ & \forall\, t : (\mathit{AT}\,\mathit{ST}). \\ & \quad t \in T \Leftrightarrow \nexists\, t' : (\mathit{AT}\,\mathit{ST}).\ \mathit{HigherPriority}\,(\mathit{HA}\,\mathit{ST})\,(t', t) \end{aligned}$$

Transitionen sind zueinander konfliktfrei, wenn ihre gleichzeitige Ausführung zu einem wohlgeformten Folgestatus im Sinne der Definition 3.22 führt. □

Das in Definition 3.27 angegebene Prädikat *MaxNonConflict* garantiert für eine maximale konfliktfreie Transitionsmenge *T*, dass *T* nur aktivierte Transitionen im Sinne der Definition 3.26 enthält. Ferner wird sichergestellt, dass von jedem Sequentiellen Automaten höchstens eine Transition in *T* enthalten sein darf. Für alle Transitionen in *T* gilt weiterhin, dass keine andere aktivierte Transition existiert, die eine höhere Priorisierung im Sinne der Definition 3.25 besitzt. Umgekehrt gilt aber auch, wenn zu einer aktivierten Transition keine höher priorisierte Transition existiert, so muss sie in *T* enthalten sein. Dadurch wird das Maximalitätskriterium sichergestellt.

Die Definition des Operators *MTS* zeigt, dass es mehr als eine maximale konfliktfreie Transitionsmenge und damit unterschiedliche Folgestatus geben kann. Diese semantische Darstellung repräsentiert das nichtdeterministische Verhalten von Statecharts.

Im Folgenden werden wir den initialen Status eines Hierarchischen Automaten definieren. In der Definition wird das relationale Bild einer Relation *R* für eine Menge *S* durch folgenden Ausdruck $R\,(\!|\,S\,|\!)$ beschrieben. Der Operator $\times$ bezeichnet das Kartesische Produkt.

Definition 3.28 (Initialer Status) *Sei HA ein Hierarchischer Automat, so ist der initiale Status von HA mit InitStatus festgelegt.*

$$\begin{aligned} \mathit{InitStatus} ::_{\mathsf{c}} & (\sigma,\epsilon,\delta)\,\mathit{hierauto} \rightarrow (\sigma,\epsilon,\delta)\,\mathit{status} \\ \mathit{InitStatus}\,\mathit{HA} \equiv_{\mathsf{c}} & \mathit{Abs\_status}\,(\mathit{HA}, \mathit{InitConf}\,\mathit{HA}, \varnothing, \mathit{InitData}\,\mathit{HA}) \end{aligned}$$

Hierbei definiert InitConf die initiale Konfiguration des HA.

$$\begin{aligned} \mathit{InitConf} ::_{\mathsf{c}} & (\sigma,\epsilon,\delta)\,\mathit{hierauto} \rightarrow \sigma\,\mathit{set} \\ \mathit{InitConf}\,\mathit{HA} \equiv_{\mathsf{c}} & ((\mathit{InitStates}\,\mathit{HA} \times \mathit{InitStates}\,\mathit{HA}) \cap (\chi\,\mathit{HA}))^*\ (\!|\,\mathit{InitRootState}\,\mathit{HA}\,|\!) \end{aligned}$$

Der Operator InitStates liefert die initialen Zustände aller Sequentiellen Automaten aus HA. Der Operator InitRootState liefert den initialen Zustand des Wurzelautomaten von HA. □

Mit Definition 3.28 legen wir fest, dass im initialen Status eines Hierarchischen Automaten *HA* keine Ereignisse anliegen. Ferner setzen wir die aktuelle Datenbelegung auf die in der Definition des *HA* angegebene initiale Datenbelegung. Die initiale Konfiguration konstruieren wir folgendermaßen. Zunächst schränken wir die Nachfolgerelation χ auf die Zustandspaare ein, in denen beide Zustände initiale Zustände aus *HA* sind. Danach konstruieren wir den transitiv-reflexiven Abschluß auf der eingeschränkten Relation. Schließlich erhalten wir die

initiale Zustandsmenge (Konfiguration), in dem wir auf der so konstruierten Relation das relationale Bild für den initialen Zustand des Wurzelautomaten aus HA bilden.

In Definition 3.29 legen wir fest, wie sich für eine gegebene konfliktfreie Transitionsmenge und für einen aktuell anliegenden Status der Folgestatus errechnen läßt. Um die Definition übersichtlich zu halten, wird zusätzlich eine totale Update-Funktion auf dem Datenraum angenommen. Diese Update-Funktion beschreibt einen der Effekte auf dem Datenraum, die durch das gleichzeitige Ausführen der partiellen Update-Funktionen aus der gegebenen Transitionsmenge entstehen können. Wie eine solche totale Update-Funktion mathematisch präzise unter Beachtung einer Interleaving-Semantik bestimmt werden kann, werden wir in Kapitel 4 angeben. Der Operator ! repräsentiert die Applikation einer totalen Update-Funktion auf eine aktuelle Datenraumbelegung.

Definition 3.29 (Folgestatus) *Sei ST ein Status, T eine Menge konfliktfreier Transitionen und U eine totale Update-Funktion auf dem Datenraum, die aus den in T vorliegenden partiellen Update-Funktionen errechnet wurde, so ist der Folgestatus von ST unter Ausführung von T mit dem Operator $StepStatus$ festgelegt.*

$$
\begin{aligned}
StepStatus ::_c\ & [\,(\sigma,\epsilon,\delta)\ status,\ (\sigma,\epsilon,\delta)\ trans\ set,\ \delta\ update\,] \\
& \rightarrow (\sigma,\epsilon,\delta)\ status \\
StepStatus\ ST\ T\ U \equiv_c\ & \mathsf{let}\ \ (HA, C, E, D) = Rep\_status\ ST; \\
& \qquad\qquad C' = StepConf\ HA\ C\ T; \\
& \qquad\qquad E' = ActionEvents\ T; \\
& \qquad\qquad D' = U\,!\,D \\
& \mathsf{in}\ \ \ Abs\_status\ (HA, C', E', D')
\end{aligned}
$$

Hierbei definiert der Operator $ActionEvents$ für eine Transition alle von ihr erzeugbaren Ereignisse. Der Operator $StepConf$ beschreibt die Folgekonfiguration für einen Hierarchischen Automaten HA, eine Konfiguration C und eine Menge konfliktfreier Transitionen T.

$$
\begin{aligned}
StepConf ::_c\ & [\,(\sigma,\epsilon,\delta)\ hierauto, \sigma\ set, (\sigma,\epsilon,\delta)\ trans\ set\,] \\
& \rightarrow \sigma\ set \\
StepConf\ HA\ C\ T \equiv_c\ & C \setminus ((\chi^*\ HA)\ (\!|\ Source\ T\ |\!))\ \cup \\
& Target\ T\ \cup \\
& ((\chi^+\ HA)\ (\!|\ Target\ T\ |\!)) \cap (InitStates\ HA)
\end{aligned}
$$

□

Das Kernstück in Definition 3.29 ist die Konstruktion der Folgekonfiguration mit Hilfe des Operators $StepConf$. Für einen Hierarchischen Automaten HA, eine Konfiguration C und eine Menge konfliktfreier Transitionen T bilden wir die Folgekonfiguration in drei Schritten. Zunächst wird in einem ersten Schritt C um die Startzustände der auszuführenden Transitionen aus T reduziert. Hierbei werden nicht nur die Startzustände der Transition, sondern auch deren direkte und indirekte Nachfolgezustände aus C entfernt. In einem zweiten Schritt werden die Zielzustände der auszuführenden Transitionen aus T zu der reduzierten Konfigurationsmenge hinzugefügt. Schließlich werden in einem dritten Schritt die direkten und indirekten Nachfolgezustände der Zielstände von T hinzugefügt, wenn es sich bei diesen um initiale Zustände aus HA handelt.

Unter Verwendung der Definitionen 3.28 und 3.29 zur Beschreibung des initialen Status und des Folgestatus kann eine induktive Mengendefinition zur Beschreibung der Menge aller erreichbaren Status in einem Hierarchischen Automaten angegeben werden.

Definition 3.30 (Erreichbare Status) *Sei* HA *ein Hierarchischer Automat, so definieren die folgenden zwei Regeln eine induktiv ableitbare Menge* $ReachStatus$*, die den initialen Status des* HA *und alle von ihm erreichbaren Status enthält.*

$$InitStatus\ HA \in ReachStatus$$

$$\frac{ST \in ReachStatus\ HA;\ T \in MTS\ ST;\ U \in SolveRacing_{\mathsf{T}}\ T}{StepStatus\ ST\ T\ U \in ReachStatus\ HA}$$

□

Der Operator $SolveRacing_{\mathsf{T}}$ definiert eine Menge von totalen Update-Funktionen. Jede in dieser Menge enthaltene Funktion beschreibt einen möglichen Effekt, der durch das konkurrierende Schreiben der übergebenen Transitionen auf dem Datenraum entstehen kann. Eine mathematisch präzise Beschreibung des Operators haben wir in Kapitel 4 angegeben.

3.3.3 Konfigurationsbäume zur Repräsentation von Konfigurationen

Analog zur Formalisierung aus dem Abschnitt 3.2.2 zur effizienteren Beschreibung von Kompositionsfunktionen wollen wir die Konfiguration eines Status durch einen primitiv-rekursiven Datentyp ausdrücken. Hierzu wird die Menge an aktiven Zuständen durch folgenden rekursiven Datentyp $\sigma\ conftree$ ersetzt, mit dem n-äre Bäume von Zuständen darstellbar sind.

$$\sigma\ conftree \equiv_{\mathsf{t}} Node_{\mathsf{C}}\ \sigma\ (\sigma\ conftree)\ list$$

Auf Basis dieses Datentyps führen wir zunächst eine Definition für einen baumbasierten Status ein, indem wir Definition 3.21 geeignet anpassen.

Definition 3.31 (Baumbasierter Status) *Sei* σ *ein Typ von Zustandsbezeichnern,* ϵ *ein Typ von Ereignisbezeichnern und* δ *eine Typ für den Datenraum, dann ist ein baumbasierter Status* ST_{T} *über* $(\sigma, \epsilon, \delta)$ *durch ein Quadruple* $(HA_{\mathsf{T}}, T_{\mathtt{Conf}}, E, D)$ *repräsentiert. Hierbei ist*

- HA_{T} *der baumbasierte Hierarchische Automat,*
- $T_{\mathtt{Conf}}$ *der Konfigurationsbaum, bestehend aus aktiven Zuständen,*
- E *die Menge von anliegenden Ereignissen und*
- D *die aktuelle Datenraumbelegung*

des baumbasierten Status ST_{T}*. Diese Komponenten müssen die interne Konsistenzbedingung* $StatusCorrect_{\mathsf{T}}$ *über baumbasierten Status erfüllen, die in Definition 3.32 angegeben ist. Der*

Typ $(\sigma, \epsilon, \delta)\ \mathit{status}_{\mathrm{T}}$ *besteht aus allen baumbasierten Status über* $(\sigma, \epsilon, \delta)$.

$$
\begin{aligned}
(\sigma, \epsilon, \delta)\ \mathit{status}_{\mathrm{T}} \equiv_{\mathrm{t}} \{\, & (\mathit{HA}_{\mathrm{T}}, T_{\mathtt{Conf}}, E, D) \mid \\
& (\mathit{HA}_{\mathrm{T}} :: (\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}}) \\
& (T_{\mathtt{Conf}} :: \sigma\ \mathit{conftree}) \\
& (E :: \epsilon\ \mathit{set}) \\
& (D :: \delta\ \mathit{data}). \\
& \mathit{StatusCorrect}_{\mathrm{T}}\ \mathit{HA}_{\mathrm{T}}\ T_{\mathtt{Conf}}\ E\ D \,\} \\
& \quad \textit{gerechtfertigt durch } \mathsf{StatusTreeNonEmpty}
\end{aligned}
\tag{3.10}
$$

□

Das Theorem StatusTreeNonEmpty belegt, dass die in Definition 3.31 abstrahierte Menge nicht leer ist. Das Prädikat $\mathit{StatusCorrect}_{\mathrm{T}}$ beschreibt die sinnvolle Zusammensetzung der Komponenten eines baumbasierten Status.

Definition 3.32 (Wohlgeformter baumbasierter Status) *Sei* HA_{T} *ein Hierarchischer Automat,* $T_{\mathtt{Conf}}$ *ein Konfigurationsbaum bestehend aus aktiven Zuständen in* HA_{T} *und* E *eine Menge von Ereignissen, die in* HA_{T} *anliegen, dann bilden diese Komponenten einen wohlgeformten baumbasierten Status, wenn folgende Bedingungen erfüllt sind.*

$$
\begin{aligned}
\mathit{StatusCorrect}_{\mathrm{T}} ::_{\mathrm{c}} & [\,(\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}}, \sigma\ \mathit{conftree}, \epsilon\ \mathit{set}, \delta\ \mathit{data}\,] \\
& \rightarrow \mathit{bool} \\
\mathit{StatusCorrect}_{\mathrm{T}}\ \mathit{HA}_{\mathrm{T}}\ T_{\mathtt{Conf}}\ E\ D \equiv_{\mathrm{c}} & E \subseteq \mathit{Events}\ \mathit{HA}_{\mathrm{T}}\ \wedge \\
& \mathit{DataSpace}\ \mathit{HA}_{\mathrm{T}} = \mathit{DataSpace}\ D\ \wedge \\
& \mathit{IsConfTree}\ (\mathit{CompTree}\ \mathit{HA}_{\mathrm{T}})\ \ T_{\mathtt{Conf}}
\end{aligned}
\tag{3.11}
$$

Die Eigenschaft $\mathit{IsConfTree}$ *garantiert die Wohlgeformtheit für einen Konfigurationsbaum* $T_{\mathtt{Conf}}$ *bezüglich des Kompositionsbaumes* $T_{\mathtt{Comp}}$ *des baumbasierten Hierarchischen Automaten* HA_{T}.

$$
\begin{aligned}
\mathit{IsConfTree}\ T_{\mathtt{Comp}}\ (\mathit{Node}_{\mathrm{C}}\ S\ L) &= S \in \mathit{States}\ (\mathit{Root}_{\mathtt{SA}}\ T_{\mathtt{Comp}})\ \wedge \\
&\quad \mathit{IsConfTree}_{\mathrm{L}}\ ((\mathit{Root}_{\mathtt{Fun}}\ T_{\mathtt{Comp}})\ S)\ L \\
\\
\mathit{IsConfTree}_{\mathrm{L}}\ \mathit{CT}_{\mathrm{L}}\ [\,] &= (\mathit{CT}_{\mathrm{L}} = [\,]) \\
\mathit{IsConfTree}_{\mathrm{L}}\ \mathit{CT}_{\mathrm{L}}\ (S \# L) &= (\mathit{CT}_{\mathrm{L}} \neq [\,])\ \wedge \\
&\quad \mathit{IsConfTree}\ (\mathit{Head}\ \mathit{CT}_{\mathrm{L}})\ S\ \wedge \\
&\quad \mathit{IsConfTree}_{\mathrm{L}}\ (\mathit{Tail}\ \mathit{CT}_{\mathrm{L}})\ L
\end{aligned}
$$

Die Anwendung des Operators $\mathit{Root}_{\mathtt{SA}}$ *auf eine Kompositionsfunktion liefert den Sequentiellen Automaten des Wurzelknotens. Die Anwendung des Operators* $\mathit{Root}_{\mathtt{Fun}}$ *auf eine Kompositionsfunktion liefert eine Nachfolgefunktion. Die Anwendung der so berechneten Nachfolgefunktion auf einen Zustand liefert eine Liste von Unterbäumen.* □

Die in Definition 3.32 angegebene Wohlgeformtheitseigenschaft eines Konfigurationsbaums haben wir ähnlich zu der Wohlgeformtheitseigenschaft eines Kompositionsbaums (vgl. Definition 3.15) durch eine primitiv-rekursive Konstantendefinition ausgedrückt. In der Definition

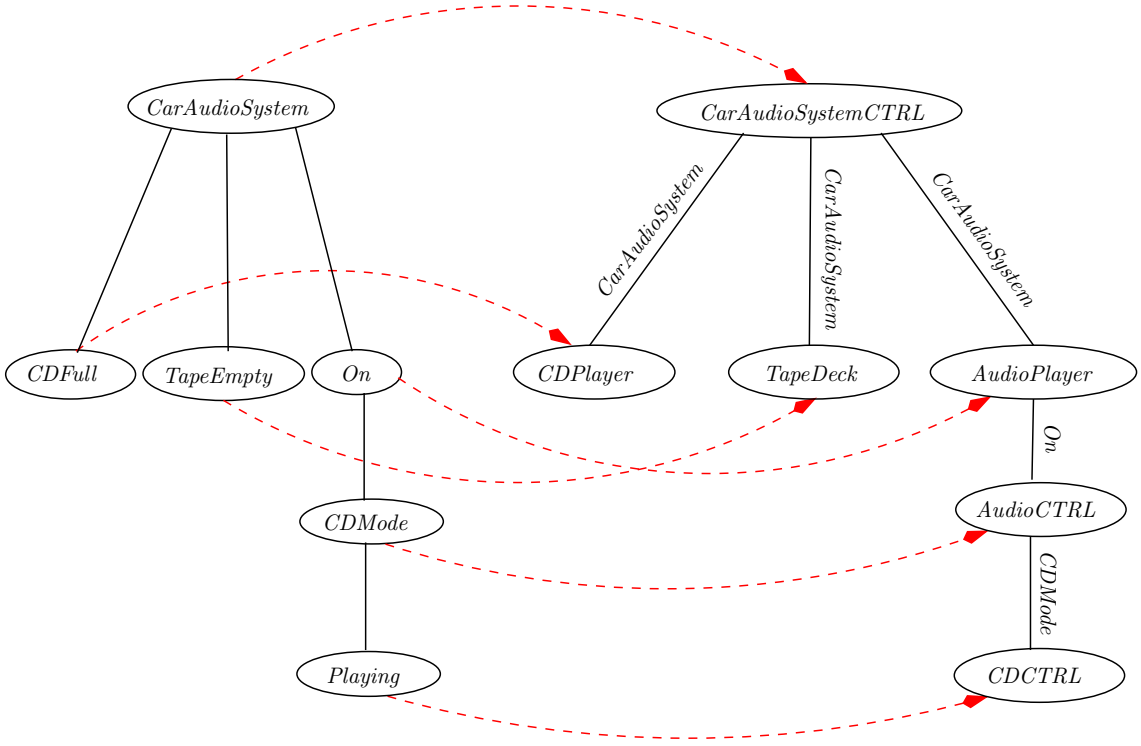


Abbildung 3.6: Wohlgeformtheit eines Konfigurationsbaumes für das *Car-Audio-System*-Modell

wird die strukturelle Ähnlichkeit von Konfigurations- und Kompositionsbaum ausgenutzt. Die Bäume werden schrittweise traversiert und korrespondierende Knoten miteinander verglichen. Dieses algorithmische Vorgehen haben wir in Abbildung 3.6 für die Beispielspezifikation des *Car-Audio-Systems* veranschaulicht.

Abbildung 3.6 zeigt auf der linken Seite einen Konfigurationsbaum. Wir haben einen Status gewählt, in dem gerade der *CD-Player* spielt. Auf der rechten Seite der Abbildung ist der zugehörige Kompositionsbaum dargestellt. Um die Wohlgeformtheitseigenschaft nachzuweisen, wird zunächst die Strukturgleichheit von Konfigurations- und Kompositionsbaum überprüft. Weiterhin wird gezeigt, dass korrespondierende Knoten aus Konfigurations- und Kompositionsbaum miteinander verträglich sind. Die Korrespondenzen zwischen den Knoten sind in der Abbildung durch gestrichelte Pfeile veranschaulicht. Zwei miteinander korrespondierende Knoten tragen unterschiedliche Informationen: Der Knoten des Konfigurationsbaumes ist mit einem Zustand `S`, der Knoten des Kompositionsbaumes mit einem Sequentiellen Automaten `SA` versehen. Eine Verträglichkeit zwischen den Knoten ist dann gegeben, wenn der Zustand `S` in den Zuständen von `SA` enthalten ist. So ist beispielsweise der Zustand *Playing* in den Zuständen des Sequentiellen Automaten *CDCTRL* enthalten.

3.4 Zusammenfassung

Wir haben in diesem Kapitel Isabelle/HOL-Theorien vorgestellt, in denen die abstrakte Syntax und die Semantik Hierarchischer Automaten formalisiert sind. Die vorgestellten Ergebnisse haben wir bereits veröffentlicht [HK01a]. Als Ausgangsposition für die Formalisierung wählten

wir eine von anderen Wissenschaftlern bereits mehrfach erprobte Beschreibung Hierarchischer Automaten [MLS97, Mik00]. Unserer Kenntnis nach gab es bisher keine Formalisierung dieser Beschreibung in einem Theorembeweiser.

Wir stellten im ersten Abschnitt dieses Kapitels zunächst eine Theorie für Isabelle/HOL vor, in der die abstrakte Syntax Sequentieller Automaten formalisiert ist. Darauf aufbauend entwickelten wir eine Theorie zur Beschreibung der abstrakten Syntax Hierarchischer Automaten. Hierbei wurde ein Hierarchischer Automat aus mehreren Sequentiellen Automaten auf Basis einer Kompositionsfunktion zusammengesetzt. Wir folgten bei der Umsetzung der originalen Beschreibung Hierarchischer Automaten aus der Literatur [MLS97]. Die beiden in diesem ersten Abschnitt vorgestellten Theorien ermöglichen es, Statecharts-Spezifikationen in dem Theorembeweiser Isabelle/HOL in Form Hierarchischer Automaten darzustellen. Damit ist die Voraussetzung dafür geschaffen, dass Beweise über der abstrakten Syntax von Statecharts-Spezifikationen in Isabelle/HOL geführt werden können.

Um das praktische Arbeiten in Isabelle/HOL zu verbessern, stellten wir im zweiten Abschnitt eine alternative Formalisierung der abstrakten Syntax Hierarchischer Automaten vor. In dieser Beschreibung wurde die baumartige Struktur Hierarchischer Automaten explizit modelliert. Anstelle der Kompositionsfunktion wurde ein Kompositionsbaum auf Basis primitiv-rekursiver Datentypen formalisiert. Dieser von uns komplett neu entwickelte Ansatz bietet gegenüber der ersten Beschreibung die Möglichkeit, Induktionsbeweise über der abstrakten Syntax Hierarchischer Automaten zu führen. Unsere Erfahrungen zeigen, dass die Anwendung dieses Beweisprinzips die Verifikation von konkreten Statecharts-Spezifikationen häufig erleichtert. Ferner setzten wir in diesem Abschnitt die beiden entwickelten Formalisierungen zueinander in Beziehung. Wir entwickelten hierfür einen Operator, mit dem aus einem baumbasierten Hierarchischen Automaten die Komponenten eines funktionsbasierten Hierarchischen Automaten konstruiert werden können. Wir zeigten, dass die so konstruierten Komponenten die in der Literatur angegebenen Wohlgeformtheitseigenschaften Hierarchischer Automaten erfüllen.

Im dritten Abschnitt präsentierten wir eine Formalisierung der Semantik Hierarchischer Automaten. Hierzu wurden zunächst semantische Zustände (*Status*) beschrieben und darauf aufbauend die synchrone Schrittsemantik Hierarchischer Automaten formalisiert. In Analogie zur Beschreibung der abstrakten Syntax Hierarchischer Automaten entwickelten wir zunächst eine Theorie, die sich an einer Beschreibung aus der Literatur orientiert. Anschließend stellten wir eine alternative Theorie vor, mit der die ursprüngliche Formalisierung im Hinblick auf eine effizientere Beweisführung optimiert werden konnte. Die Optimierung wurde dadurch erzielt, dass die Menge aktiver Zustände in einem Status durch einen Konfigurationsbaum beschrieben wurde, der aufgrund seiner primitiv-rekursiven Eigenschaft beim Führen von Beweisen einen höheren Grad an Automatisierung ermöglicht.

Zusammenfassend haben wir mit diesem Kapitel eine Grundlage dafür geschaffen, dass Beweise über Statecharts-Spezifikationen in Form von Hierarchischen Automaten in Isabelle/HOL sowohl auf der abstrakten Syntax, als auch auf der Semantik in vielen Fällen effizient geführt werden können. Mit der Formalisierung von Datenräumen, die einer Statecharts-Spezifikation im Allgemeinen zugrunde liegen, haben wir uns in diesem Kapitel nur am Rande beschäftigt. Da ein wichtiger Schwerpunkt dieser Dissertation die Entwicklung von Konzepten zur Datenabstraktion von Statecharts ist, stellen wir im folgenden Kapitel eine Formalisierung von partitionierten Datenräumen, Datenraumbelegungen und Update-Funktionen für Statecharts ausführlich vor.

Kapitel 4

Datenräume Hierarchischer Automaten

In der Formalisierung aus Kapitel 3 adressierten wir Datenräume, die Hierarchischen Automaten zugrunde liegen können. Datenräume wurden sowohl in der syntaktischen Beschreibung als auch in der semantischen Interpretation Hierarchischer Automaten behandelt. Um aber die Formalisierung in Kapitel 3 nicht mit zu vielen Details zu überfrachten, haben wir zur Beschreibung der semantischen Effekte auf dem zugrunde liegenden Datenraum Operatoren verwendet, die nicht formalisiert wurden. Dies betrifft insbesondere Definition 3.30 auf Seite 66, in der wir alle vom initialen Status eines Hierarchischen Automaten erreichbaren Folgestatus durch eine induktive Menge beschrieben haben. In dieser Definition verwenden wir einen Operator $\mathit{SolveRacing}_{\mathrm{U}}$, der für alle in einem synchronen Zeitschritt ausführbaren Transitionen eine Menge von Update-Funktionen zurückliefert. Jede dieser Update-Funktionen repräsentiert einen möglichen Effekt auf dem Datenraum, der nach Ausführung des betrachteten Zeitschrittes entstehen kann. In diesem Kapitel führen wir unter anderem eine präzise Definition für den Operator $\mathit{SolveRacing}_{\mathrm{U}}$ ein.

Wir stellen eine Theorie vor, in der Datenräume, Datenraumbelegungen und Update-Funktionen Hierarchischer Automaten detailliert beschrieben sind. Zunächst motivieren wir an einem kleinen Beispiel die Anforderungen für die Formalisierung. In Abschnitt 4.2 beschreiben wir Definitionen für Datenräume. Darauf aufbauend werden in Abschnitt 4.3 Update-Funktionen eingeführt. Schließlich definieren wir in Abschnitt 4.4 eine Semantik, die festlegt, welche Effekte auf dem Datenraum durch die synchrone Ausführung mehrerer Transitionen entstehen.

4.1 Beispiel für eine Statecharts-Spezifikation mit Daten

In Abbildung 4.1 haben wir ein Beispiel für eine Statecharts-Spezifikation dargestellt, an dem wir im Folgenden die Formalisierung der Datenräume Hierarchischer Automaten erläutern wollen.

Der Datenraum in der Beispielspezifikation besteht aus zwei Datenvariablen vom Typ Integer, die mit x und y bezeichnet sind. Der Wurzelzustand S_1 in Abbildung 4.1 ist ein UND-Zustand. Er wird durch die drei ODER-Zustände S_2, S_3 und S_4 verfeinert. Diese Zustände sind wiederum jeweils durch zwei BASIS-Zustände verfeinert, die durch eine Transition miteinander verbunden sind. So enthält beispielsweise der Zustand S_2 die BASIS-Zustände S_5 und S_6. Die Transition zwischen den beiden Zuständen ist dann aktiviert, wenn das Ereignis E_1 anliegt und der Zustands S_5 aktiv ist. Der Effekt der Transition wird durch die Update-Funktion Update_1 beschrieben. Diese Funktion definiert, dass der Datenvariablen x der Wert zugewiesen wird, mit dem vor Ausführung der Transition die Datenvariable y belegt war.

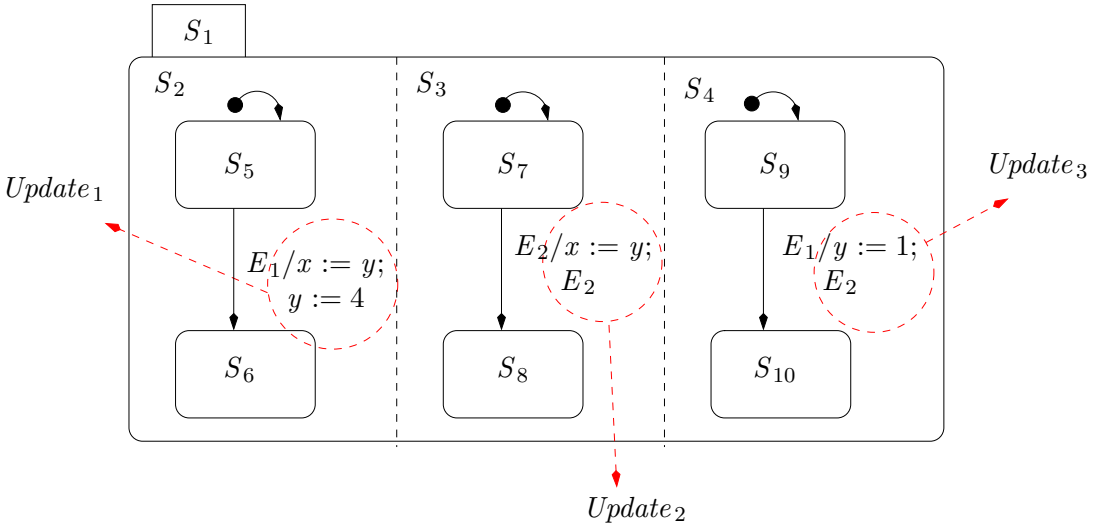


Abbildung 4.1: Beispielspezifikation für einen Statechart mit einem Datenraum

Die Datenvariable y wird mit 4 beschrieben. Neue Ereignisse werden durch diese Transition nicht erzeugt. Man beachte, dass durch die Update-Funktion $Update_1$ der Datenraum vollständig beschrieben wird. Dies ist im Allgemeinen nicht der Fall. So wird beispielsweise im Zustand S_3 der Datenraum durch die Update-Funktion $Update_2$ nur partiell beschrieben. Partialität bedeutet hier, dass lediglich der Datenvariablen x ein neuer Wert zugewiesen wird. Die Datenvariable y bleibt hingegen unbeschrieben. Update-Funktionen in einer Statecharts-Spezifikation definieren also im Allgemeinen nur eine unvollständige Datenraumbelegung. Wir bezeichnen sie deshalb auch als *partielle Update-Funktionen*.

Im Gegensatz dazu führt die Ausführung eines synchronen Zeitschrittes in der Semantik von Statecharts zu einer vollständigen Datenraumbelegung. Entsprechend verwenden wir zur Beschreibung der Semantik *totale Update-Funktionen*. In Abschnitt 4.4 zeigen wir, wie aus partiellen Update-Funktionen totale Update-Funktionen abgeleitet werden können. Wir legen damit fest, welche Datenbelegung für unbeschriebene Partitionen des Datenraumes nach Ausführung eines synchronen Zeitschrittes angenommen werden soll. Ferner beschreiben wir, welche Datenbelegung zugewiesen wird, wenn mehrere synchron auszuführende Transitionen den Datenraum konkurrierend beschreiben. Diese Konkurrenzsituation zwischen mehreren Transitionen wird in der Literatur auch als *Racing* bezeichnet.

Ein weiterer wichtiger Aspekt der Formalisierung ist, dass die Datenräume von Statecharts-Spezifikationen eine Struktur besitzen. Diese wird durch eine Partitionierung des Datenraumes beschrieben. Eine Partition kann hierbei aus mehreren Datenvariablen bestehen. Eine partielle Update-Funktion muss eine Partition immer vollständig beschreiben. Besteht also eine Partition aus mehreren Datenvariablen, so werden durch eine partielle Update-Funktion entweder alle oder gar keine Datenvariablen in dieser Partition beschrieben. Das Beispiel aus Abbildung 4.1 besteht aus zwei Partitionen, die jeweils durch eine Datenvariable repräsentiert sind. Die beiden Datenvariablen können nicht zu einer Partition zusammengefasst werden, da die Update-Funktionen $Update_2$ und $Update_3$ jeweils nur eine Datenvariable beschreiben. Partitionierungen des Datenraumes spielen also in der Semantik von Statecharts eine entscheidende Rolle und sind bei der Formalisierung zu berücksichtigen.

Im Folgenden geben wir an, wie die beschriebenen Anforderungen in unserer Formalisierung Hierarchischer Automaten umgesetzt sind. Wir weisen darauf hin, dass in der Literatur bisher nur wenige Beschreibungen von Statecharts zu finden sind, die auch die Formalisierung von Datenräumen in ausführlicher Art und Weise behandeln [BG97, Büs03]. Insbesondere die Vorarbeiten von Erich Mikk [Mik00], an die wir uns in dieser Arbeit stark anlehnen, unterstützen keine Datenräume. Aus diesem Grunde war es für diese Dissertation notwendig, eine eigene Formalisierung zur Beschreibung von Datenräumen Hierarchischer Automaten vorzuschlagen.

4.2 Partitionierte Datenräume

In diesem Abschnitt führen wir eine Formalisierung von Datenräumen für Hierarchische Automaten ein. Im ersten Teil beschreiben wir, wie die Partitionierung eines Datenraumes in unserer Formalisierung umgesetzt ist. Im zweiten Teil dieses Abschnitts stellen wir die Formalisierung von Datenraumbelegungen vor.

Die Partitionierung des Datenraumes haben wir in der folgenden Definition durch eine Liste von Mengen repräsentiert. Hierbei ist jede Partition durch eine Menge beschrieben, wobei die Elemente der Menge alle möglichen Datenbelegungen der Partition angeben. Eine Menge in der Liste legt also den Definitionsbereich einer Partition fest.

Definition 4.1 (Partitionierter Datenraum) *Sei δ ein Typ für den Datenraum, dann ist ein partitionierter Datenraum D über δ durch eine Liste von Mengen L repräsentiert. L muss die interne Konsistenzbedingung $DataSpaceCorrect$ über partitionierten Datenräumen erfüllen, die in Definition 4.2 angegeben ist. Der Typ $\delta\ dataspace$ besteht aus allen wohlgeformten partitionierten Datenräumen über δ.*

$$\begin{aligned}\delta\ dataspace \equiv_{\mathsf{t}} \{\, L \mid\ & \\ & (L :: \delta\ set\ list). \\ & DataSpaceCorrect\ L \,\}\end{aligned}$$

gerechtfertigt durch DataSpaceNonEmpty

□

Um den Definitionsbereich eines Datenraumes an der i-ten Partition selektieren zu können, führen wir den Operator $PartDom$ ein. Die Anzahl der Partitionen kann mit dem Operator $PartNum$ für einen Datenraum zurückgeliefert werden. Die wohlgeformte Partitionierung eines Datenraumes ist folgendermaßen festgelegt.

Definition 4.2 (Wohlgeformter partitionierter Datenraum) *Sei L eine Liste von Mengen, die einen Grundtyp δ partitionieren, dann ist die Partitionierung wohlgeformt, wenn folgende Bedingungen erfüllt sind.*

$$\begin{aligned}DataSpaceCorrect\ & ::_{\mathsf{c}} \delta\ set\ list \to bool \\ DataSpaceCorrect\ L\ & \equiv_{\mathsf{c}} distinct\ L\ \wedge \\ & \qquad \forall\, D^1, D^2 : set\ L.\ D^1 \neq D^2 \Rightarrow D^1 \cap D^2 = \varnothing\ \wedge \\ & \qquad \bigcup set\ L = \mathsf{UNIV}\end{aligned}$$

□

$ds \equiv_{\mathsf{t}} V_0\ int$	$DS ::_{\mathsf{c}} ds\ dataspace$
$\mid\ V_1\ int$	$DS \equiv_{\mathsf{c}} Abs\_dataspace\,[\,ran\ V_0, ran\ V_1\,]$

Abbildung 4.2: Beispiel für einen Datenraum mit zwei *Integer*- Partitionen

Das Prädikat $distinct$ in Definition 4.2 fordert, dass keine doppelten Elemente in einer Liste vorkommen. Die Konstante UNIV beschreibt die maximale Menge über einem Grundtyp.

Ferner garantieren wir mit Definition 4.2 für einen partitionierten Datenraum, dass in der Liste zur Repräsentation der Partitionen jede Partition nur einmal vorkommt. Weiterhin wird gefordert, dass alle Partitionen zueinander paarweise disjunkt sind. Fügt man schließlich alle Partitionen zu einer zusammen, so erhält man die maximale Menge des polymorphen Grundtypen.

Um die Formalisierung von disjunkt partitionierten Datenräumen zu veranschaulichen, haben wir in Abbildung 4.2 einen aus zwei Partitionen bestehenden Datenraum DS formalisiert. Mit DS haben wir den Datenraum für das Beispiel aus Abbildung 4.1 repräsentiert. Beide Partitionen sind entsprechend durch eine *Integer*- Variable beschrieben.

Auf der linken Seite in Abbildung 4.2 haben wir zunächst einen Datentyp ds definiert, bei dem die Elemente der ersten Partition mit dem Konstruktor V_0 und entsprechend die Elemente der zweiten Partition mit dem Konstruktor V_1 erzeugt werden. Auf der rechten Seite der Abbildung wird der polymorphe Typ für den Datenraum mit ds instanziiert. Weiterhin nutzen wir die Bildbereiche der Konstruktoren, um eine zweielementige Liste anzugeben, die als Repräsentant für den neu einzuführenden Datenraum DS dient. Es ist leicht zu sehen, dass die in Definition 4.2 geforderten Eigenschaften gelten, so dass folgendes Theorem abgeleitet werden kann.

$$DataSpaceCorrect\,[\,ran\ V_0, ran\ V_1\,] \qquad \text{(DataSpaceCorrectDS)}$$

Alternativ kann der polymorphe Typ für den Datenraum auch durch die binäre Summe $(int + int)$ instanziiert und der Repräsentant für DS aus den Bildbereichen der beiden Injektionsfunktionen des Summentyps gebildet werden.

Bei der Belegung des Datenraumes unterscheiden wir zwischen einer *partiellen* und einer *vollständigen* Belegung. *Partielle Datenraumbelegungen* bilden eine wichtige Grundlage für partielle Update-Funktionen, die im Aktionsteil eines Statecharts vorkommen. Erst durch die Auswertung aller synchron auszuführenden partiellen Update-Funktionen können totale Update-Funktionen berechnet werden, die dann entsprechend in den Typ von vollständigen Datenraumbelegungen abbilden. Weiterhin wird von der initialen Datenraumbelegung erwartet, dass sie eine vollständige Belegung des Datenraumes definiert. Zur sprachlichen Vereinfachung bezeichnen wir vollständige Datenraumbelegungen im Gegensatz zu partiellen Datenraumbelegungen auch einfach als Datenraumbelegungen.

Definition 4.3 (Datenraumbelegung) *Sei δ ein Typ für den Datenraum, dann ist eine vollständige Datenraumbelegung V über δ durch ein Tupel (L, D) repräsentiert. Hierbei ist*

- L *die Liste von Datenbelegungen für einzelne Partitionen und*
- D *der Datenraum*

der Datenraumbelegung V. Diese Komponenten müssen die interne Konsistenzbedingung $DataCorrect$ über Datenraumbelegungen erfüllen, die in Definition 4.4 angegeben ist. Der Typ $\delta\ data$ besteht aus allen wohlgeformten Datenraumbelegungen über δ.

$$\begin{aligned}\delta\ data \equiv_{\mathsf{t}} \{\, (L, D) \mid\ & \\ & (L :: \delta\ list) \\ & (D :: \delta\ dataspace). \\ & DataCorrect\ L\ D \,\} \\ & \quad \textit{gerechtfertigt durch } \mathsf{DataNonEmpty}\end{aligned}$$

□

Um die Datenbelegung der i-ten Partition in einer Datenraumbelegung selektieren zu können, führen wir den Operator $PartData$ ein. Der zugrunde liegende Datenraum in einer Datenraumbelegung kann mit dem Operator $DataSpace$ selektiert werden.

In Definition 4.4 ist angegeben, unter welchen Bedingungen eine Liste von Datenbelegungen für einzelne Partitionen bezüglich eines Datenraums wohlgeformt ist. Der Operator # gibt hierbei die Länge einer Liste an.

Definition 4.4 (Wohlgeformte Datenraumbelegung) *Sei L eine Liste von Datenbelegungen und D ein Datenraum, dann sind die Datenbelegungen bezüglich des Datenraumes wohlgeformt, wenn folgende Bedingungen erfüllt sind.*

$$\begin{aligned}DataCorrect\ &::_{\mathsf{c}} [\delta\ list, \delta\ dataspace] \to bool \\ DataCorrect\ L\ D\ &\equiv_{\mathsf{c}} \#\ L = PartNum\ D\ \wedge \\ &\qquad \forall\ i : \{\, n.\, n < PartNum\ D \,\}.\, (PartData\ L\ i) \in (PartDom\ D\ i)\end{aligned}$$

□

Definition 4.4 garantiert, dass eine Datenraumbelegung genauso viele Einträge enthält, wie es Partitionen in dem zugehörigen Datenraum gibt. Ferner ist sichergestellt, dass die für eine Partition angegebene Datenbelegung den Definitionsbereich der Partition nicht verletzt.

Um die Formalisierung von Datenraumbelegungen zu veranschaulichen, stellen wir in Abbildung 4.3 die Definition einer möglichen initialen Datenraumbelegung $InitDS$ für das Beispiel aus Abbildung 4.2 vor. Initial nehmen wir hierzu an, dass die erste Partition mit 7 und die zweite Partition mit 9 beschrieben werden soll. Weiterhin verwenden wir den Typ ds, die Konstruktoren V_0 und V_1 sowie die Konstante DS, die wir in Abbildung 4.2 bereits eingeführt haben. Es ist leicht zu sehen, dass die in der Definition 4.4 geforderten Eigenschaften gelten, so dass folgendes Theorem abgeleitet werden kann.

$$DataCorrect\ [\, V_0\ 7, V_1\ 9\,]\ DS \qquad \text{(DataCorrectInitDS)}$$

$$InitDS ::_{c} ds\ data$$
$$InitDS \equiv_{c} Abs\_data\ ([\ V_0\ 7, V_1\ 9\ ], DS)$$

Abbildung 4.3: Beispiel für eine initiale Datenraumbelegung

Ähnlich zur Definition 4.3 geben wir nun eine Definition für partielle Datenraumbelegungen an. Die Partialität wird unter Verwendung des in Abschnitt 2.3.2 auf Seite 37 eingeführten *option*-Typ modelliert.

Definition 4.5 (Partielle Datenraumbelegung) *Sei δ ein Typ für den Datenraum, dann ist eine partielle Datenraumbelegung V_P über δ durch ein Tupel (L, D) repräsentiert. Hierbei ist*

- L *die Liste von optionalen Datenbelegungen für einzelne Partitionen und*
- D *der Datenraum.*

der partiellen Datenraumbelegung V_P. Diese Komponenten müssen die interne Konsistenzbedingung `PDataCorrect` *über partiellen Datenraumbelegungen erfüllen, die in Definition 4.4 angegeben ist. Der Typ δ* `pdata` *besteht aus allen wohlgeformten partiellen Datenraumbelegungen über δ.*

$$\begin{aligned}\delta\ pdata \equiv_{t} \{\, (L, D) \mid\ & (L :: \delta\ option\ list) \\ & (D :: \delta\ dataspace). \\ & PDataCorrect\ L\ D \,\} \\ & \textit{gerechtfertigt durch } \mathsf{PDataNonEmpty}\end{aligned}$$

□

Um die Datenbelegung der i-ten Partition in einer partiellen Datenraumbelegung selektieren zu können, führen wir den Operator `PartPData` ein. Dieser Operator liefert für unbelegte Partitionen ein `None` zurück und stellt für definierte Partitionsbelegungen dem Rückgabewert die Konstante `Some` voran.

Die Definition 4.6 zur Wohlgeformtheit von partiellen Datenraumbelegungen ist sehr ähnlich zur Definition 4.4, nur mit dem Unterschied, dass Teile der Eigenschaft für undefinierte Partitionen nicht erfüllt sein müssen.

Definition 4.6 (Wohlgeformte partielle Datenraumbelegung) *Sei* L *eine Liste von optionalen Datenbelegungen und* D *ein Datenraum, dann sind die optionalen Datenbelegungen bezüglich des Datenraumes wohlgeformt, wenn folgende Bedingungen erfüllt sind.*

$$\begin{aligned} \mathit{PDataCorrect} &::_{\mathsf{c}} [\delta\ \mathit{option}\ \mathit{list}, \delta\ \mathit{dataspace}] \to \mathit{bool} \\ \mathit{PDataCorrect}\ L\ D &\equiv_{\mathsf{c}} \#\ L = \mathit{PartNum}\ D\ \wedge \\ &\qquad \forall\ i : \{\, n \,.\, n < \mathit{PartNum}\ D \,\}. \\ &\qquad\quad (\mathit{PartPData}\ L\ i) \neq \mathit{None} \Rightarrow (\mathit{PartPData}\ L\ i) \in (\mathit{PartDom}\ D\ i) \end{aligned}$$

□

In diesem Abschnitt haben wir Definitionen für partitionierte Datenräume, vollständige Datenraumbelegungen und partielle Datenraumbelegungen eingeführt. Wir werden diese Definitionen im nächsten Abschnitt dazu benutzen, Update-Funktionen zu formalisieren.

4.3 Update-Funktionen

Nachdem wir die formale Basis für einen disjunkt partitionierten Datenraum geschaffen haben, wollen wir nun eine adäquate Formalisierung von Update-Funktionen angeben. Hierbei ist zu beachten, dass mit der Update-Funktion einer Transition nicht immer alle Partitionen des Datenraumes geschrieben werden müssen. Dies bedeutet, dass wir einen formalen Begriff für *partielle Update-Funktionen* benötigen.

Da die Semantik Hierarchischer Automaten auch für unbeschriebene Partitionen bei der Ausführung eines synchronen Zeitschritts im Nachzustand eine Wertebelegung festlegt, ist für die Formalisierung der Semantik zusätzlich ein Begriff von *totalen Update-Funktionen* notwendig. Zur sprachlichen Vereinfachung bezeichnen wir totale Update-Funktionen auch einfach nur als Update-Funktionen.

Definition 4.7 (Update-Funktion) *Sei* δ *ein Typ für einen Datenraum, dann ist eine totale Update-Funktion* U *über* δ *durch eine Abbildung beschrieben, die jeder Datenraumbelegung eine neue eindeutige Datenraumbelegung zuordnet. Hierbei muss* U *die interne Konsistenzbedingung* $\mathit{UpdateCorrect}$ *erfüllen, die in der Definition 4.8 angegeben ist. Der Typ* $\delta\ \mathit{update}$ *besteht aus allen wohlgeformten Update-Funktionen über* δ*.*

$$\begin{aligned} \delta\ \mathit{update} \equiv_{\mathsf{t}} \{\, U \mid & \\ & (U :: \delta\ \mathit{data} \to \delta\ \mathit{data})\,. \\ & \mathit{UpdateCorrect}\ U \,\} \end{aligned}$$

gerechtfertigt durch UpdateNonEmpty

□

Wie bereits in Abschnitt 3.3.1 auf der Seite 65 erwähnt, repräsentiert der Operator ! die Applikation einer Update-Funktion auf eine aktuelle Datenraumbelegung.

Definition 4.8 (Wohlgeformte Update-Funktion) *Sei* U *eine Abbildung von Datenraumbelegungen nach Datenraumbelegungen, dann ist diese Abbildung wohlgeformt, wenn folgende Bedingung erfüllt ist.*

$$\begin{aligned} \mathit{UpdateCorrect} &::_{\mathsf{c}} (\delta\ \mathit{data} \to \delta\ \mathit{data}) \to \mathit{bool} \\ \mathit{UpdateCorrect}\ U &\equiv_{\mathsf{c}} \forall D.\ \mathit{DataSpace}\ D = \mathit{DataSpace}\ (U\ D) \end{aligned}$$

□

Mit der Wohlgeformtheitseigenschaft aus der Definition 4.8 fordern wir also, dass eine Update-Funktion immer den Datenraum erhalten muss. Für eine Statecharts-Spezifikation in unserer Formalisierung erreichen wir dadurch den gewünschten Effekt, dass ein initial festgelegter Datenraum durch wohlgeformte Update-Funktionen nicht verändert werden kann.

Ein Spezialfall unter den Update-Funktionen ist die so genannte *Default-Update-Funktion*. Diese Funktion repräsentiert die Wiederherstellung der Datenbelegung auf dem Datenraum.

Definition 4.9 (Default-Update-Funktion) *Sei δ ein Typ für den Datenraum, dann ist eine Default-Update-Funktion für δ durch die Konstante* `UpdateDefault` *folgendermaßen beschrieben.*

$$\begin{aligned} &\mathtt{UpdateDefault} ::_{\mathsf{c}} (\delta\ \mathtt{update}) \\ &\mathtt{UpdateDefault} \equiv_{\mathsf{c}} \mathtt{Abs\_update}(\lambda\ D.\ D) \end{aligned}$$

□

Eine Default-Update-Funktion wird in der Statecharts-Semantik z.B. dazu benötigt, semantische Zustandsübergänge zu beschreiben, in denen die synchron ausgeführten Transitionen den Datenraum nicht beschreiben (vgl. Definition 4.13).

Im Folgenden führen wir partielle Update-Funktionen ein.

Definition 4.10 (Partielle Update-Funktion) *Sei δ ein Typ für den Datenraum, dann ist eine partielle Update-Funktion* `U` *über δ durch eine Abbildung beschrieben, die jeder Datenraumbelegung eine eindeutige partielle Datenraumbelegung zuordnet. Hierbei muss* `U` *die interne Konsistenzbedingung* `PUpdateCorrect` *erfüllen, die in Definition 4.11 angegeben ist. Der Typ δ* `pupdate` *besteht aus allen partiellen Update-Funktionen über δ.*

$$\begin{aligned} \delta\ \mathtt{pupdate} \equiv_{\mathsf{t}} \{\, &U \mid \\ &(U :: \delta\ \mathtt{data} \to \delta\ \mathtt{pdata})\,. \\ &\mathtt{PUpdateCorrect}\ U \,\} \end{aligned}$$

gerechtfertigt durch PUpdateNonEmpty

□

Der Operator !! repräsentiert die Applikation einer partiellen Update-Funktion auf eine aktuelle Datenraumbelegung.

Definition 4.11 (Wohlgeformte partielle Update-Funktion) *Sei* `U` *eine Abbildung von Datenraumbelegungen zu partiellen Datenraumbelegungen, dann ist diese Abbildung wohlgeformt, wenn folgende Bedingung erfüllt ist.*

$$\begin{aligned} \mathtt{PUpdateCorrect} &::_{\mathsf{c}} (\delta\ \mathtt{data} \to \delta\ \mathtt{pdata}) \to \mathtt{bool} \\ \mathtt{PUpdateCorrect}\ U &\equiv_{\mathsf{c}} \forall D.\ \mathtt{DataSpace}\ D = \mathtt{DataSpace}\ (U\ D) \end{aligned}$$

□

```
Update_3 ::_c ds pupdate
Update_3 ≡_c Abs_pupdate (λ D.
                  if DS = DataSpace D then
                      Abs_pdata ( [ None, Some (V_1 1) ], DS )
                  else DataAsPData D )
```

Abbildung 4.4: Beispiel für eine partielle Update-Funktion

Mit der Wohlgeformtheitseigenschaft aus Definition 4.11 fordern wir analog zur Definition 4.8, dass eine partielle Update-Funktion immer den Datenraum erhalten muss.

Um die Formalisierung einer partiellen Update-Funktion zu veranschaulichen, zeigen wir in Abbildung 4.4 die Definition der Update-Funktion *Update*$_3$ für das Beispiel aus Abbildung 4.1. In der dort angegebenen Definition verwenden wir den Typ `ds`, den Konstruktor V_1 und den Datenraum `DS`, die wir bereits in Abbildung 4.2 eingeführt haben. Wir überprüfen zunächst, ob die der Update-Funktion übergebene Datenraumbelegung auf dem Datenraum `DS` basiert. Ist dies der Fall, so wird eine partielle Datenraumbelegung zurückgegeben, in der die erste Partition unbeschrieben und die zweite Partition mit 1 beschrieben ist. Sollte die der Update-Funktion übergebene Datenraumbelegung zu `DS` verschieden sein, so wird die übergebene Datenraumbelegung in eine partielle Datenraumbelegung umgewandelt und ohne weitere Veränderungen zurückgegeben. Der dazu verwendete Operator `DataAsPData` ist in Anhang A.5 zu finden. Es ist leicht zu erkennen, dass in beiden durch die Update-Funktion $Update_3$ unterschiedenen Fällen, die Wohlgeformtheitseigenschaft im Sinne der Definition 4.11 erfüllt ist.

Analog zu Definition 4.9 geben wir einen Spezialfall für partielle Update-Funktionen an. Mit der so genannten *Partiellen Default-Update-Funktion* wird die *leere* Update-Funktion repräsentiert. Dies bedeutet, dass alle Datenpartitionen nach Ausführung dieser Update-Funktion mit der Konstanten `None` belegt werden.

Definition 4.12 (Partielle Default-Update-Funktion) *Sei δ ein Typ für den Datenraum, dann ist eine partielle Default-Update-Funktion über δ durch die Konstante* `PUpdateDefault` *folgendermaßen beschrieben.*

```
PUpdateDefault ::_c (δ pupdate)
PUpdateDefault ≡_c Abs_pupdate
                     (λ D. Abs_pdata
                         ( replicate (PartNum(DataSpace D)) None,
                           DataSpace D) )
```

□

Hierbei definiert der Ausdruck `replicate N E` eine `N`-elementige Liste, in der alle Felder mit dem Wert `E` beschrieben sind.

4.4 Semantische Interpretation von Racing-Effekten

Da die Semantik von Statecharts festlegt, dass in einem Zeitschritt im Allgemeinen mehrere Transitionen gleichzeitig ausgeführt werden, kann es zu Konflikten beim Schreiben des Datenraumes kommen. So eine Situation tritt immer dann ein, wenn gleichzeitig auszuführende Transitionen auf die selben Datenpartitionen schreiben. Das Eintreten eines solchen Konflikts beim konkurrierenden Schreiben von Datenräumen wird in der Literatur auch als *Racing-Effekt* bezeichnet. In dieser Arbeit werden *Racing-Effekte* durch die Anwendung einer so genannten *Interleaving-Semantik* in einem Nichtdeterminismus aufgelöst.

Um den Effekt von *Racing* zu erläutern, ziehen wir noch einmal das Beispiel aus Abbildung 4.1 heran. Wir betrachten einen semantischen Zeitschritt, in dem zu Beginn die Zustände S_1, S_2, S_3, S_4, S_5, S_7 und S_9 aktiv sind und die Ereignisse E_1 und E_2 anliegen. In dieser Situation sind drei Transitionen aktiviert, da die Vorbedingungen dieser lediglich aus den Triggern E_1 bzw. E_2 bestehen. Die aktivierten Transitionen bilden im Sinne der Definition 3.27 eine maximale konfliktfreie Transitionsmenge und werden deshalb gleichzeitig ausgeführt. Zum Abschluß des Zeitschrittes werden folglich die Zustände S_5, S_7 und S_9 verlassen und die Zustände S_6, S_8 und S_{10} betreten. Zusätzlich wird das Ereignis E_2 generiert. Ein weiterer Effekt der simultanen Ausführung der Transitionen ist *Racing*, da die Integer - Variable y durch die partiellen Update-Funktionen $Update_1$ und $Update_3$ konkurrierend geschrieben wird.

Mit Abbildung 4.5 wollen wir veranschaulichen, wie der in unserem Beispiel aufgetretene *Racing*-Effekt durch eine Interleaving-Semantik beschrieben werden kann. Hierzu betrachten wir alle Folgestatus, die durch das synchrone Ausführen der Transitionen entstehen können. Die Komponenten eines Status sind in der Abbildung zur besseren Übersicht auf die Kerninformationen reduziert. So wird beispielsweise die aktuelle Datenraumbelegung als zweielementige Liste visualisiert und die Informationen über den Datenraum als implizit angenommen.

Die vor Ausführung des Zeitschrittes aktuelle Datenraumbelegung sei mit $[X_{\mathrm{Pre}}, Y_{\mathrm{Pre}}]$ vorausgesetzt. Die Zuweisung $x := y$ gibt an, dass x mit Y_{Pre} überschrieben werden soll. Diese Zuweisung für x ist in $Update_1$ und $Update_2$ angegeben. In $Update_3$ wurde keine Zuweisung für x festgelegt. Damit ist nach synchroner Ausführung der Transitionen der Wert von x eindeutig mit Y_{Pre} definiert. Im Gegensatz dazu wird die Partition von y konkurrierend durch $Update_1$ und $Update_3$ geschrieben, so dass im Ergebnis entweder y mit 4 oder mit 1 belegt sein kann. Die von uns umgesetzte Interleaving-Semantik lässt beide Varianten zu, was in Abbildung 4.5 durch zwei nichtdeterministisch erreichbare Folgestatus veranschaulicht ist.

Tritt der Schreibkonflikt nur in einer Partition auf, ist der entstehende Nichtdeterminismus leicht abzuleiten. Werden allerdings mehrere Partitionen konkurrierend geschrieben, so muss in der Semantik zusätzlich beachtet werden, dass der Schreibvorgang einer partiellen Update-Funktion als ein atomarer Vorgang aufgefasst wird.

Betrachten wir noch einmal das Beispiel aus Abbildung 4.1 und nehmen an, dass zusätzlich in $Update_2$ die Datenvariable y mit 4 und in $Update_3$ die Datenvariable x mit 3 beschrieben wird. In diesem Falle werden offensichtlich beide Partitionen konkurrierend geschrieben. Weiterhin betrachten wir wiederrum die synchrone Ausführung von allen drei Transitionen unter den identischen Voraussetzungen, wie wir sie in Abbildung 4.5 dargestellt haben. Da der Schreibvorgang einer partiellen Update-Funktion als atomare Aktion angesehen wird, kann es auch für das adaptierte Beispiel nur zwei Folgestatus mit den Datenraumbelegungen $[Y_{\mathrm{Pre}}, 4]$ und $[3, 1]$ geben. Folgestatus mit den Datenraumbelegungen $[Y_{\mathrm{Pre}}, 1]$ oder $[3, 4]$ sind hingegen nicht möglich. Wenn also eine partielle Update-Funktion den Schreibvorgang gegenüber

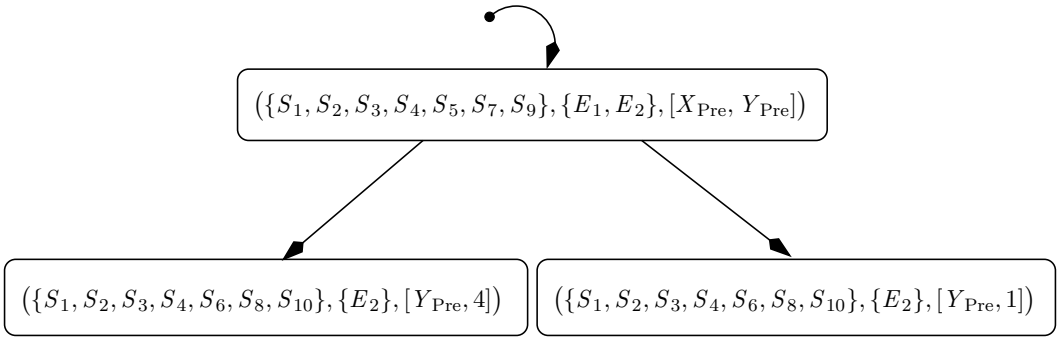


Abbildung 4.5: Semantische Interpretation von *Racing* durch *Interleaving*

Konkurrierenden gewinnt, so gewinnt sie ihn für alle Partitionen, die sie beschreiben möchte.

Der folgende Operator errechnet für eine endliche Anzahl von partiellen Update-Funktionen alle möglichen totalen Update-Funktionen auf dem Datenraum. Hierbei wird der Schreibvorgang einer partiellen Update-Funktion als ein atomarer Schritt aufgefasst, so dass zwar die Reihenfolge der Ausführung der partiellen Updates beliebig, aber eine tiefere Verzahnung in Bezug auf die Schreibvorgänge einzelner Partitionen ausgeschlossen ist.

Definition 4.13 (Interleaving-Semantik für Update-Funktionen) *Sei $S_{\mathtt{U}}$ eine Menge von synchron auszuführenden partiellen Update-Funktionen, dann ist das durch Racing induzierte nichtdeterministische Verhalten durch folgende Menge von totalen Update-Funktionen beschrieben.*

$$\begin{aligned} \mathit{SolveRacing}_{\mathtt{U}} &::_{\mathsf{c}} (\delta\, \mathit{pupdate}\ \mathit{set}) \rightarrow (\delta\, \mathit{update}\ \mathit{set}) \\ \mathit{SolveRacing}_{\mathtt{U}}\ S_{\mathtt{U}} &\equiv_{\mathsf{c}} \{U.\ (S_{\mathtt{U}}, U) \in \mathsf{foldset}\ \ \oplus_{\mathtt{U}}\ \ \mathit{UpdateDefault}\} \end{aligned}$$

□

In Definition 4.13 haben wir den in der Standard Isabelle-Distribution gegebenen Operator foldset verwendet. Mit diesem Operator können für die übergebenen partiellen Update-Funktionen alle möglichen Verknüpfungen mit Hilfe des Operators $\oplus_{\mathtt{U}}$ ermittelt werden. Hierbei ist der rekursive Anker der Verknüpfung durch den Operator *UpdateDefault* gegeben. Zur besseren Veranschaulichung zeigen wir in Abbildung 4.6 alle für das Beispiel aus Abbildung 4.1 entstehenden Verknüpfungen.

Wir erhalten also für n synchron ausgeführte partielle Update-Funktionen zunächst 2^n totale Update-Funktionen. Allerdings lässt sich nach einer semantischen Überprüfung der syntaktisch gebildeten Permutationen die erhaltene Menge häufig weiter reduzieren. So fallen in unserem Beispiel aus Abbildung 4.6 jeweils die ersten beiden und die letzten vier Update-Funktionen zu einer Update-Funktion zusammen. Damit erhalten wir genau zwei Update-Funktionen, mit denen die beiden Folgestatus aus Abbildung 4.5 erreichbar sind.

Im Folgenden präsentieren wir eine Definition für den binären Operator $\oplus_{\mathtt{U}}$ zur Verknüpfung von zwei Update-Funktionen.

$$
\begin{aligned}
&\mathit{SolveRacing}_{\mathtt{U}}\ (\{\,\mathit{Update}_1, \mathit{Update}_2, \mathit{Update}_3\,\}) \;=\\
&\qquad \{\,\mathit{Update}_1 \oplus_{\mathtt{U}} \mathit{Update}_2 \oplus_{\mathtt{U}} \mathit{Update}_3 \oplus_{\mathtt{U}} \mathit{UpdateDefault},\\
&\qquad\ \ \mathit{Update}_1 \oplus_{\mathtt{U}} \mathit{Update}_3 \oplus_{\mathtt{U}} \mathit{Update}_2 \oplus_{\mathtt{U}} \mathit{UpdateDefault},\\
&\qquad\ \ \mathit{Update}_2 \oplus_{\mathtt{U}} \mathit{Update}_1 \oplus_{\mathtt{U}} \mathit{Update}_3 \oplus_{\mathtt{U}} \mathit{UpdateDefault},\\
&\qquad\ \ \mathit{Update}_2 \oplus_{\mathtt{U}} \mathit{Update}_3 \oplus_{\mathtt{U}} \mathit{Update}_1 \oplus_{\mathtt{U}} \mathit{UpdateDefault},\\
&\qquad\ \ \mathit{Update}_3 \oplus_{\mathtt{U}} \mathit{Update}_1 \oplus_{\mathtt{U}} \mathit{Update}_2 \oplus_{\mathtt{U}} \mathit{UpdateDefault},\\
&\qquad\ \ \mathit{Update}_3 \oplus_{\mathtt{U}} \mathit{Update}_2 \oplus_{\mathtt{U}} \mathit{Update}_1 \oplus_{\mathtt{U}} \mathit{UpdateDefault}\,\}
\end{aligned}
$$

Abbildung 4.6: Mögliche Ausführungsreihenfolgen partieller Update-Funktionen

Definition 4.14 (Verknüpfung von Update-Funktionen) *Sei $U_{\mathtt{P}}$ eine partielle Update-Funktion und $U_{\mathtt{T}}$ eine totale Update-Funktion, so ist die Verknüpfung beider Funktionen zu einer totalen Update-Funktion folgendermaßen beschrieben.*

$$
\begin{aligned}
&\_ \oplus_{\mathtt{U}} \_ \;::_{\mathtt{c}}\; [\,\delta\,\mathit{pupdate}, \delta\,\mathit{update}\,] \to \delta\,\mathit{update}\\
&\_ \oplus_{\mathtt{U}} \_ \;\equiv_{\mathtt{c}}\; \lambda\ U_{\mathtt{P}}\ U_{\mathtt{T}}\,.\ \lambda\ D.\ (U_{\mathtt{P}}\,!!\ D) \oplus_{\mathtt{D}} (U_{\mathtt{T}}\,!\ D)
\end{aligned}
$$

Hierbei beschreibt der Operator $\oplus_{\mathtt{D}}$ die Verknüpfung von Datenraumbelegungen, die in der Definition 4.15 angegeben ist. □

Verknüpfen wir eine partielle Update-Funktion $U_{\mathtt{P}}$ mit einer totalen Update-Funktion $U_{\mathtt{T}}$ mit $U_{\mathtt{P}} \oplus_{\mathtt{U}} U_{\mathtt{T}}$, so kann $U_{\mathtt{T}}$ nur die Partitionen beschreiben, die von $U_{\mathtt{P}}$ nicht beschrieben werden. Damit gewinnt die Update-Funktion $U_{\mathtt{P}}$ den Schreibvorgang gegenüber der konkurrierenden Update-Funktion $U_{\mathtt{T}}$. Durch diese Festlegung kann bei einer Funktionsverkettung, wie sie in Abbildung 4.6 dargestellt ist, erreicht werden, dass die am weitesten links stehende partielle Update-Funktion den Schreibvorgang gegenüber allen anderen synchron ausgeführten Update-Funktion gewinnt. Die geringste Priorität hat hingegen die Update-Funktion $\mathit{UpdateDefault}$.

Der beschriebene Effekt ist in Definition 4.14 auf die Verknüpfung von Datenraumbelegungen zurückgeführt, die in folgender Definition durch den Operator $\oplus_{\mathtt{D}}$ angegeben ist.

Definition 4.15 (Verknüpfung von Datenraumbelegungen) *Sei $D_{\mathtt{P}}$ eine partielle Datenraumbelegung, $D_{\mathtt{T}}$ eine totale Datenraumbelegung und der zugrunde liegende Datenraum von $D_{\mathtt{P}}$ und $D_{\mathtt{T}}$ identisch, so ist die Verknüpfung beider Datenraumbelegungen zu einer totalen Datenraumbelegung folgendermaßen beschrieben.*

$$
\begin{aligned}
&\_ \oplus_{\mathtt{D}} \_ \;::_{\mathtt{c}}\; [\,\delta\,\mathit{pdata}, \delta\,\mathit{data}\,] \to \delta\,\mathit{data}\\
&\_ \oplus_{\mathtt{D}} \_ \;\equiv_{\mathtt{c}}\; \lambda\ D_{\mathtt{P}}\ D_{\mathtt{T}}\ .\\
&\qquad\qquad \mathsf{let}\ \ (L_{\mathtt{P}}, DS_{\mathtt{P}}) = \mathit{Rep\_pdata}\ D_{\mathtt{P}};\\
&\qquad\qquad\qquad\ (L_{\mathtt{T}}, DS_{\mathtt{T}}) = \mathit{Rep\_data}\ D_{\mathtt{T}};\\
&\qquad\qquad\qquad\qquad\ L = \mathit{map}\ \oplus_{\mathtt{D}}\ (\mathit{zip}\ L_{\mathtt{P}}\ L_{\mathtt{T}})\\
&\qquad\qquad \mathsf{in}\ \ \mathit{Abs\_data}\,(L, DS_{\mathtt{P}})
\end{aligned}
$$

Hierbei beschreibt der Operator $\oplus_{\mathtt{O}}$ *die Verknüpfung von einer optionalen Datenbelegung mit einer totalen Datenbelegung zu einer totalen Datenbelegung.*

$$\oplus_{\mathtt{O}} \equiv_{\mathtt{c}} \lambda\,(D_{\mathtt{O}}, D_{\mathtt{T}})\,.\,\mathsf{if}\ D_{\mathtt{O}} = \mathit{None}\ \mathsf{then}\ D_{\mathtt{T}}\ \mathsf{else}\ \mathit{the}\,D_{\mathtt{O}}$$

□

Die in der Definition eingesetzten Operatoren ***zip*** und ***map*** repräsentieren Funktionale auf Listen mit der üblichen Semantik. Der Operator $\oplus_{\mathtt{D}}$ liefert nur dann ein sinnvolles Ergebnis, wenn die übergegebenen Datenraumbelegungen auf dem gleichen Datenraum basieren. Diese Bedingung wird aber von dem Operator $\oplus_{\mathtt{U}}$ garantiert.

Nachdem wir Definition 4.13 an einem Beispiel illustriert und durch die Definitionen 4.14 und 4.15 konkretisiert haben, wollen wir zum Abschluss dieses Kapitels angeben, wie der in der Definition 3.30 eingesetzte Operator $\mathit{SolveRacing}_{\mathtt{T}}$ mit Hilfe des Operators $\mathit{SolveRacing}_{\mathtt{U}}$ definiert werden kann.

Definition 4.16 (Interleaving-Semantik für Transitionen) *Sei* $S_{\mathtt{T}}$ *eine Menge von synchron auszuführenden Transitionen, dann ist das durch Racing induzierte nichtdeterministische Verhalten durch folgende Menge von totalen Update-Funktionen beschrieben.*

$$\mathit{SolveRacing}_{\mathtt{T}} ::_{\mathtt{c}} ((\sigma, \epsilon, \delta)\,\mathit{trans}\,\mathit{set}) \rightarrow (\delta\,\mathit{update}\,\mathit{set})$$
$$\mathit{SolveRacing}_{\mathtt{T}}\ S_{\mathtt{T}} \equiv_{\mathtt{c}} \mathit{SolveRacing}_{\mathtt{U}}\,(\mathit{PUpdates}\ S_{\mathtt{T}})$$

□

Hierbei beschreibt der Operator ***PUpdates*** die Selektion der partiellen Update-Funktionen aus einer Menge von Transitionen.

4.5 Zusammenfassung

Wir haben in diesem Kapitel Isabelle/HOL-Theorien vorgestellt, in denen die Datenräume Hierarchischer Automaten formalisiert sind. Viele der vorgestellten Ideen haben wir bereits veröffentlicht [HK03b]. Datenräume sind ein wichtiger Bestandteil sowohl in der syntaktischen Beschreibung als auch in der semantischen Interpretation von Statecharts. Trotz dieser Tatsache ist uns bisher kein anderer Ansatz aus der Literatur bekannt, mit dem eine vergleichbar präzise Formalisierung von Datenräumen umgesetzt wurde.

In der vorliegenden Formalisierung haben wir zunächst auf Basis eines polymorphen Grundtypen einen Typ für partitionierte Datenräume eingeführt. Die Partitionen können hierbei aus beliebigen anderen Grundtypen von Isabelle/HOL aufgebaut sein. Anschließend haben wir eine Formalisierung von Datenraumbelegungen angegeben, in der wir zwischen vollständigen und partiellen Datenraumbelegungen unterscheiden.

Partielle Datenraumbelegungen werden in unserer Formalisierung dazu verwendet, um partielle Update-Funktionen zu definieren. Partielle Update-Funktionen werden in der abstrakten Syntax Hierarchischer Automaten benutzt, um den Effekt einer Transition auf dem Datenraum zu beschreiben.

Die initiale Datenraumbelegung eines Hierarchischen Automaten haben wir hingegen als vollständige Datenraumbelegung formalisiert. Ferner verwenden wir vollständige Datenraumbelegungen, um die Semantik Hierarchischer Automaten zu beschreiben. So enthält der Status

eines Hierarchischen Automaten eine vollständige Datenraumbelegung. Bei der Ausführung eines synchronen Zeitschrittes wird dem Datenraum eine neue Datenraumbelegung zugewiesen. Diese semantische Zustandsveränderung auf dem Datenraum modellieren wir mit Hilfe von totalen Update-Funktionen. Eine totale Update-Funktion errechnen wir hierbei aus den partiellen Update-Funktionen, die in dem betrachteten Zeitschritt synchron ausgeführt werden. Schreibkonflikte auf Datenpartitionen lösen wir durch eine Interleaving-Semantik in einem Nichtdeterminismus auf.

Zusammenfassend haben wir in diesem Kapitel mit partitionierten Datenräumen, mit vollständigen und partiellen Datenraumbelegungen sowie mit totalen und partiellen Update-Funktionen eine Semantik definiert, die das komplizierte Verhalten auf den Datenräumen Hierarchischer Automaten beschreibt. Im folgenden Kapitel geben wir eine Formalisierung der temporalen Logik CTL an, in der CTL-Aussagen über Hierarchischen Automaten formuliert werden können. Hierbei sind auch Aussagen formulierbar, die sich auf die Datenräume Hierarchischer Automaten beziehen.

Kapitel 5

CTL in Isabelle/HOL

In diesem Kapitel stellen wir HOL-Theorien vor, in denen sowohl die Syntax als auch die Semantik der Temporallogik CTL formuliert sind. Dazu geben wir zunächst eine Formalisierung von Kripke-Strukturen an. Anschließend stellen wir die Formalisierung von CTL-Operatoren vor und zeigen, wie diese auf Basis von Kripke-Strukturen durch eine induktive Beschreibung semantisch interpretiert werden können. In Abschnitt 5.3 zeigen wir, wie CTL-Operatoren alternativ durch Fixpunkte des μ-Kalküls beschrieben werden können. Im letzten Abschnitt dieses Kapitels geben wir eine HOL-Theorie an, in der die Theorien zur Beschreibung Hierarchischer Automaten und die Theorien zur Beschreibung der Temporallogik CTL zusammengeführt sind. Hierzu beschreiben wir die Semantik eines Hierarchischen Automaten durch eine Kripke-Struktur. Auf Basis dieser Beschreibung definieren wir einen Operator, mit dessen Hilfe die Gültigkeit von CTL-Formeln bezüglich Hierarchischer Automaten ausgedrückt werden kann.

5.1 Kripke-Strukturen

Kripke-Strukturen bilden das Fundament zur semantischen Interpretation von CTL-Formeln. Im Folgenden führen wir einen Typ für Kripke-Strukturen ein.

Definition 5.1 (Kripke-Struktur) *Sei σ ein Typ von Zustandsbezeichnern und α ein Typ von atomaren Aussagen, dann ist eine Kripke-Struktur `M` über (σ, α) durch ein Quadrupel (S, I, T, L) repräsentiert. Hierbei ist*

- *`S` die Menge von Zuständen,*
- *`I` die Menge von initialen Zuständen,*
- *`T` die Transitionsrelation und*
- *`L` die Labelfunktion*

der Kripke-Struktur `M`. Diese Komponenten müssen die interne Konsistenzbedingung `KripkeCorrect` über Kripke-Strukturen erfüllen, die in Definition 5.2 angegeben ist. Der Typ

(σ, α) *kripke besteht aus allen Kripke-Strukturen über* (σ, α).

$$
\begin{aligned}
(\sigma,\alpha)\,\mathit{kripke} \equiv_{\mathsf{t}} \{\, & (S, I, T, S) \mid \\
& (S :: \sigma\ \mathit{set}) \\
& (I :: \sigma\ \mathit{set}) \\
& (T :: (\sigma * \sigma)\ \mathit{set}) \\
& (L :: \sigma \rightharpoonup (\alpha\ \mathit{set})). \\
& \mathit{KripkeCorrect}\ S\ I\ T\ L \,\} \\
& \quad \textit{gerechtfertigt durch } \mathsf{KripkeNonEmpty}
\end{aligned}
\tag{5.1}
$$

□

In Analogie zur Bezeichnung der semantischen Zustände Hierarchischer Automaten nennen wir die Zustände einer Kripke-Struktur auch *Status*. Um auf die Komponenten einer Kripke-Struktur zugreifen zu können, führen wir folgende Selektionsoperatoren ein. Die Menge an Zuständen kann mit dem Operator *Status*, die Menge an initialen Zuständen mit dem Operator *InitStatus*, die Transitionsrelation mit dem Operator *TransRel* und die Labelfunktion mit dem Operator *LabelFun* bestimmt werden.

Definition 5.2 (Wohlgeformte Kripke-Struktur) *Sei* S *eine Menge von Zuständen,* I *ein Menge von initialen Zuständen,* T *eine Transitionsrelation und* L *eine Labelfunktion, dann bilden diese Komponenten eine wohlgeformte Kripke-Struktur, wenn folgende Bedingungen erfüllt sind.*

$$
\begin{aligned}
\mathit{KripkeCorrect} ::_{\mathsf{c}} & [\sigma\ \mathit{set}, \sigma\ \mathit{set}, (\sigma * \sigma)\ \mathit{set}, \sigma \rightharpoonup (\alpha\ \mathit{set})\ ] \\
& \rightarrow \mathit{bool} \\
\mathit{KripkeCorrect}\ S\ I\ T\ L \equiv_{\mathsf{c}} & I \subseteq S \ \wedge\ T \subseteq S \times S \ \wedge\ \mathit{dom}\ T = S \ \wedge\ \mathit{dom}\ L = S
\end{aligned}
$$

□

Definition 5.2 garantiert, dass alle initialen Status einer Kripke-Struktur M in der Menge aller gültigen Status aus M enthalten sind. Ferner wird gefordert, dass die Transitionsrelation nur über gültigen Status aus M gebildet werden darf. Außerdem muss sowohl die Transitionsrelation als auch die Labelfunktion bezüglich aller gültigen Status aus M total sein.

Die Semantik einer Kripke-Struktur wird durch *Berechnungspfade* festgelegt. Berechnungspfade sind durch unendliche Sequenzen von Status beschrieben. Folgende Definition gibt die Formalisierung von Berechnungspfaden für eine Kripke-Struktur an.

Definition 5.3 (Berechnungspfade in einer Kripke-Struktur) *Sei* M *eine Kripke-Struktur und* S *ein Status aus* M*, dann beschreibt folgende Menge alle in* M *vom Status* S *ausgehenden Berechnungspfade.*

$$
\begin{aligned}
\mathit{Paths}\ & ::_{\mathsf{c}} [\ (\sigma,\alpha)\,\mathit{kripke}, \sigma\ ] \rightarrow (\mathit{nat} \rightarrow \sigma)\ \mathit{set} \\
\mathit{Paths}\ M\ S\ & \equiv_{\mathsf{c}} \{\, P.\ S = (P\ 0)\ \wedge\ \forall\ i.\ (P\ i, P\ (i+1)) \in \mathit{TransRel}\ M \,\}
\end{aligned}
$$

□

In Definition 5.3 haben wir unendliche Sequenzen als eine Funktion repräsentiert, die von natürlichen Zahlen auf Status abbildet. Ein gültiger Berechnungspfad in einer Kripke-Struktur M ab einem Status S wird durch eine Funktion beschrieben, die zwei Eigenschaften erfüllt. Zum einen muss die Funktion an der 0-ten Stelle auf den Status S abbilden. Damit stellen wir sicher, dass der Berechnungspfad mit dem Status S beginnt. Zum anderen fordern wir, dass alle direkt aufeinanderfolgenden Einträge in der Funktion durch die Transitionsrelation von M in Beziehung stehen. Damit stellen wir sicher, dass ein Berechnungspfad für jeden aufgeführten Status nur einen mit der Transitionsrelation von M erreichbaren Folgestatus als Nachfolger definiert.

Wir haben uns in der Definition 5.3 dafür entschieden, unendliche Sequenzen mit Hilfe von Funktionen zu beschreiben. Diese Art der Beschreibung besticht durch ihre Einfachheit und ist für unsere Formalisierung gut geeignet. In der Literatur sind Arbeiten sowohl zu dieser Beschreibung als auch zu alternativen Beschreibungen für unendliche Sequenzen angegeben [DGM97]. Besonders effizient sind hierbei Ansätze, die auf einer Erweiterung der HOL-Logik mit Bereichstheorie beruhen. Diese Erweiterung ist auch unter dem Namen HOLCF-Logik bekannt [Reg94, Reg95]. Sequenzen werden in diesen Arbeiten durch einfache rekursive Bereichsgleichungen ausgedrückt. Die Praktikabilität dieses Ansatzes wurde an einem Trace-Modell für I/O-Automaten erprobt [MN97]. Da wir im Rahmen dieser Dissertation auf Basis von Berechnungspfaden keine umfangreichen Isabelle-Beweise führen, haben wir auf eine derartige Erweiterung verzichtet.

5.2 CTL-Operatoren

In diesem Abschnitt führen wir die Operatoren der CTL (*Computation Tree Logik*) [CE81] ein und legen ihre Semantik bezüglich einer gegebenen Kripke-Struktur fest. Dazu geben wir zunächst einen rekursiven Datentyp $(\sigma,\alpha)ctl$ an, mit dem alle Operatoren der CTL repräsentiert werden können.

$$
\begin{aligned}
(\sigma,\alpha)ctl \equiv_t\ & Atom\ \alpha \\
\mid\ & Neg\ (\sigma,\alpha)ctl \\
\mid\ & And\ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl \\
\mid\ & Or\ \ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl \\
\mid\ & Imp\ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl \\
\mid\ & EX\ \ (\sigma,\alpha)ctl \\
\mid\ & AX\ \ (\sigma,\alpha)ctl \\
\mid\ & EF\ \ (\sigma,\alpha)ctl \\
\mid\ & AF\ \ (\sigma,\alpha)ctl \\
\mid\ & EG\ \ (\sigma,\alpha)ctl \\
\mid\ & AG\ \ (\sigma,\alpha)ctl \\
\mid\ & EU\ \ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl \\
\mid\ & AU\ \ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl \\
\mid\ & ER\ \ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl \\
\mid\ & AR\ \ (\sigma,\alpha)ctl\ (\sigma,\alpha)ctl
\end{aligned}
$$

Die semantische Interpretation von CTL-Formeln wird auf Basis des rekursiven Datentyps $(\sigma, \alpha)\,ctl$ mit einer primitiv-rekursiven Konstantendefinition angegeben.

Definition 5.4 (Erfüllbarkeit von CTL-Formeln) *Sei* F *vom rekursiven Datentyp* $(\sigma, \alpha)\,ctl$*,* M *eine Kripke-Struktur vom Typ* $(\sigma, \alpha)\,kripke$ *und* S *ein gültiger Zustand aus* M*, dann ist mit folgender primitiv-rekursiven Konstantendefinition festgelegt, ob* F *in* M *ab dem Zustand* S *erfüllt ist.*

$$
\begin{aligned}
(M,S) \models (Atom\ P) &= P \in the(LabelFun\ M)\ S \\
(M,S) \models (Neg\ F) &= \neg\ (M,S) \models F \\
(M,S) \models (And\ F\ G) &= (M,S) \models F \wedge (M,S) \models G \\
(M,S) \models (Or\ F\ G) &= (M,S) \models F \vee (M,S) \models G \\
(M,S) \models (Imp\ F\ G) &= (M,S) \models F \Rightarrow (M,S) \models G \\
(M,S) \models (EX\ F) &= \exists\ T.\ (S,T) \in (TransRel\ M) \wedge (M,T) \models F \\
(M,S) \models (AX\ F) &= \forall\ T.\ (S,T) \in (TransRel\ M) \Rightarrow (M,T) \models F \\
(M,S) \models (EF\ F) &= \exists\ T.\ (S,T) \in (TransRel\ M)^* \wedge (M,T) \models F \\
(M,S) \models (AF\ F) &= \forall\ P : Paths\ M\ S.\ \exists\ i.\ (M,\ P\ i) \models F \\
(M,S) \models (EG\ F) &= \exists\ P : Paths\ M\ S.\ \forall\ i.\ (M,\ P\ i) \models F \\
(M,S) \models (AG\ F) &= \forall\ P : Paths\ M\ S.\ \forall\ i.\ (M,\ P\ i) \models F \\
(M,S) \models (EU\ F\ G) &= \exists\ P : Paths\ M\ S.\ \exists\ i.\ (M,\ P\ i) \models G \wedge \forall\ j.\ j < i \Rightarrow (M,\ P\ j) \models F \\
(M,S) \models (AU\ F\ G) &= \forall\ P : Paths\ M\ S.\ \exists\ i.\ (M,\ P\ i) \models G \wedge \forall\ j.\ j < i \Rightarrow (M,\ P\ j) \models F \\
(M,S) \models (ER\ F\ G) &= \exists\ P : Paths\ M\ S.\ \forall\ i.\ (M,\ P\ i) \models G \wedge \exists\ j.\ j < i \Rightarrow (M,\ P\ j) \models F \\
(M,S) \models (AR\ F\ G) &= \forall\ P : Paths\ M\ S.\ \forall\ i.\ (M,\ P\ i) \models G \wedge \exists\ j.\ j < i \Rightarrow (M,\ P\ j) \models F
\end{aligned}
$$

□

Definition 5.4 setzt eine Semantik für CTL-Formeln um, wie sie in der Literatur sehr häufig angegeben wird [CGP00]. Wir verzichten deshalb auf eine ausführliche Erläuterung. Im folgenden Abschnitt geben wir eine andere Möglichkeit an, wie CTL-Operatoren semantisch interpretiert werden können.

5.3 Fixpunktcharakterisierung

Alternativ lassen sich die Operatoren der CTL-Logik durch eine Fixpunktcharakterisierung im μ-Kalkül [EC80, CGP00] beschreiben. Diese Art der Formalisierung wollen wir zunächst für die unären Operatoren der CTL-Logik vorstellen.

Definition 5.5 (Fixpunktcharakterisierung unärer CTL-Operatoren) *Sei* T *die Transitionsrelation einer Kripke-Struktur und* P *ein Prädikat an einem Zustand* U*, dann definiert ein unärer CTL-Operator, angewendet auf* T *und* P*, ein neues Prädikat am Zustand* U *mit folgender Konstantendeklaration.*

$$\_ ::_c [\ (\sigma * \sigma)\ set,\ \sigma\ pred\ ] \rightarrow \sigma\ pred$$

Die Operatoren EX_P *und* AX_P *fordern das Prädikat* P *für direkte Nachfolgezustände von* U.

$$EX_P\ T\ P \equiv_c \lambda\ U.\ \exists\ V.\ P\ V \wedge\ (U, V) \in T$$
$$AX_P\ T\ P \equiv_c \lambda\ U.\ \forall\ V.\ P\ V \vee\ (U, V) \notin T$$

Die Operatoren EF_P, AF_P, EG_P *und* AG_P *definieren hingegen auch Aussagen über indirekte Nachfolgezustände von* U, *die mittels der Operatoren des* μ*-Kalküls zur Bestimmung des kleinsten Fixpunkts (*μ*) und des größten Fixpunkts (*ν*) angegeben sind.*

$$EF_P\ T\ P \equiv_c \mu\ (\lambda\ Q.\ P \vee_P\ (EX_P\ T\ Q))$$
$$AF_P\ T\ P \equiv_c \mu\ (\lambda\ Q.\ P \vee_P\ (AX_P\ T\ Q))$$
$$EG_P\ T\ P \equiv_c \nu\ (\lambda\ Q.\ P \wedge_P\ (EX_P\ T\ Q))$$
$$AG_P\ T\ P \equiv_c \nu\ (\lambda\ Q.\ P \wedge_P\ (AX_P\ T\ Q))$$

□

Der Typ $\sigma\ pred$ ist durch folgendes Typsynonym gegeben.

$$\sigma\ pred \equiv_s \sigma \to bool$$

Die Junktoren $\wedge_P$ und $\vee_P$ repräsentieren entsprechend Verknüpfungen auf Prädikaten in folgender Art und Weise.

$$\_ \wedge_P \_ ::_c [\sigma\ pred, \sigma\ pred] \to \sigma\ pred$$
$$P \wedge_P Q \equiv_c \lambda\ U.\ P\ U\ \wedge\ Q\ U$$

Auch für andere Junktoren, wie z.B. $\Rightarrow_P$ oder $\neg_P$ sind entsprechende Konstantendefinitionen auf Prädikaten eingeführt. Auszüge aus der Theorie haben wir in Anhang A.6 angegeben.

Analog zur Definition 5.5 formalisieren wir in Definition 5.6 die binären Operatoren der CTL-Logik.

Definition 5.6 (Fixpunktcharakterisierung binärer CTL-Operatoren) *Sei* T *die Transitionsrelation einer Kripke-Struktur,* P *und* R *Prädikate, dann definiert ein binärer* CTL*-Operator, angewendet auf* T, P *und* R, *ein neues Prädikat mit folgender Konstantendeklaration.*

$$\_ ::_c [(\sigma * \sigma)\ set, \sigma\ pred, \sigma\ pred] \to \sigma\ pred$$

Die Operatoren EU_P, AU_P, ER_P *und* AR_P *definieren Aussagen über indirekte Nachfolgezustände.*

$$EU_P\ T\ P\ R \equiv_c \mu\ (\lambda\ Q.\ R \vee_P\ (P \wedge_P\ (EX_P\ T\ Q)))$$
$$AU_P\ T\ P\ R \equiv_c \mu\ (\lambda\ Q.\ R \vee_P\ (P \wedge_P\ (AX_P\ T\ Q)))$$
$$ER_P\ T\ P\ R \equiv_c \nu\ (\lambda\ Q.\ R \wedge_P\ (P \vee_P\ (EX_P\ T\ Q)))$$
$$AR_P\ T\ P\ R \equiv_c \nu\ (\lambda\ Q.\ R \wedge_P\ (P \vee_P\ (AX_P\ T\ Q)))$$

□

Zur Validierung der Definitionen 5.5 und 5.6 haben wir eine Reihe von Theoremen abgeleitet, die bekannte Zusammenhänge zwischen CTL-Operatoren beschreiben. Die gewählten Theoreme sind an den Arbeiten von Alexander Bolotov und Michael Fisher [BF99] orientiert, die eine Resolutionsmethode für die CTL-Logik vorgeschlagen haben. Diese Methode beruht auf einer Normalform für CTL-Formeln, dem Konzept der Schrittresolution und einer besonderen Form von temporalen Resolutionsregeln. Die Autoren geben ein vollständiges Axiomensystem für die CTL an. Dieses besteht aus zehn Axiomen und zwei Inferenzregeln, von denen wir einige in unserer Theorie abgeleitet haben. Wir haben die von Alexander Bolotov und Michael Fisher vorgeschlagene Resolutionsmethode in unserem Framework nicht weiter untersucht. Allerdings bilden die von ihnen vorgeschlagenen Axiome und Referenzregeln eine gute Grundlage, um weitere Theoreme über CTL-Operatoren abzuleiten.

Um die Fixpunktcharakterisierung von CTL-Operatoren mit der induktiven Semantik aus Definition 5.4 in Beziehung zu setzen, führen wir einen geeigneten Operator ein. Mit Hilfe dieses Operators kann für eine gegebene temporallogische Formel der CTL und für eine gegebene Kripke-Struktur ein Prädikat angegeben werden, in dem alle CTL-Operatoren durch ihre Fixpunktcharakterisierung beschrieben sind.

Definition 5.7 (Fixpunktcharakterisierung einer CTL-Formel) *Sei* `F` *eine CTL-Formel vom rekursiven Datentyp* (σ, α)`ctl` *und* `M` *eine Kripke-Struktur vom Typ* (σ, α)`kripke`*, dann ist mit folgender primitiv-rekursiven Konstantendefinition ein Prädikat vom Typ* σ `pred` *festgelegt, das die Fixpunktcharakterisierung von* `F` *beschreibt.*

$$
\begin{aligned}
\mathit{Pred}\ M\ (\mathit{Atom}\ P) &= \lambda S.\ P \in \mathit{the}(\mathit{LabelFun}\ M)\ S \\
\mathit{Pred}\ M\ (\mathit{Neg}\ F) &= \neg_{\mathrm{P}}\ \mathit{Pred}\ M\ F \\
\mathit{Pred}\ M\ (\mathit{And}\ F\ G) &= \mathit{Pred}\ M\ F \wedge_{\mathrm{P}}\ \mathit{Pred}\ M\ G \\
\mathit{Pred}\ M\ (\mathit{Or}\ F\ G) &= \mathit{Pred}\ M\ F \vee_{\mathrm{P}}\ \mathit{Pred}\ M\ G \\
\mathit{Pred}\ M\ (\mathit{Imp}\ F\ G) &= \mathit{Pred}\ M\ F \Rightarrow_{\mathrm{P}}\ \mathit{Pred}\ M\ G \\
\mathit{Pred}\ M\ (\mathit{EX}\ F) &= \mathit{EX}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F) \\
\mathit{Pred}\ M\ (\mathit{AX}\ F) &= \mathit{AX}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F) \\
\mathit{Pred}\ M\ (\mathit{EF}\ F) &= \mathit{EF}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F) \\
\mathit{Pred}\ M\ (\mathit{AF}\ F) &= \mathit{AF}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F) \\
\mathit{Pred}\ M\ (\mathit{EG}\ F) &= \mathit{EG}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F) \\
\mathit{Pred}\ M\ (\mathit{AG}\ F) &= \mathit{AG}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F) \\
\mathit{Pred}\ M\ (\mathit{EU}\ F\ G) &= \mathit{EU}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F)\ (\mathit{Pred}\ M\ G) \\
\mathit{Pred}\ M\ (\mathit{AU}\ F\ G) &= \mathit{AU}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F)(\mathit{Pred}\ M\ G) \\
\mathit{Pred}\ M\ (\mathit{ER}\ F\ G) &= \mathit{ER}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F)\ (\mathit{Pred}\ M\ G) \\
\mathit{Pred}\ M\ (\mathit{AR}\ F\ G) &= \mathit{AR}_{\mathrm{P}}\ (\mathit{LabelFun}\ M)\ (\mathit{Pred}\ M\ F)(\mathit{Pred}\ M\ G)
\end{aligned}
$$

□

Definition 5.7 ermöglicht es, den in der Literatur angegebenen Zusammenhang zwischen der induktiven semantischen Interpretation von CTL-Formeln (vgl. Definition 5.4) und der

Fixpunktcharakterisierung (vgl. die Definitionen 5.5 und 5.6) durch folgende Aussage zu formulieren.

$$(\lambda\ S.\ (M, S \models F)) = \mathit{Pred}\ M\ F$$

Da die Gültigkeit dieser Aussage in der Literatur außer Frage steht, haben wir darauf verzichtet, in Isabelle/HOL einen Beweis hierfür abzuleiten. Hinzu kommt, dass für das zentrale Ziel dieser Dissertation – die Entwicklung einer Datenabstraktionstheorie für Statecharts – die Fixpunktcharakterisierung von CTL-Formeln keine wichtige Rolle spielt, da wir uns im Folgenden auf die induktive Semantik (vgl. Definition 5.4) für CTL-Formeln stützen. Auf Basis dieser Semantik entwickeln wir im folgenden Abschnitt eine Theorie, in der CTL-Formeln in Bezug auf eine Statecharts-Spezifikation semantisch interpretiert werden können. Konkreter beschreiben wir hier, wie eine solche Interpretation für Hierarchische Automaten vorgenommen werden kann.

5.4 Semantische Interpretation für Statecharts

In diesem Abschnitt stellen wir eine Theorie vor, in der einerseits die Formalisierung Hierarchischer Automaten (vgl. die Kapitel 3 und 4) und andererseits die Formalisierung von Kripke-Strukturen und CTL-Formeln zu einer Formalisierung zusammengeführt werden. Kripke-Strukturen dienen hierbei dazu, die Semantik Hierarchischer Automaten formal zu fassen. Folglich beschreiben wir die Zustände von Kripke-Strukturen durch semantische Zustände Hierarchischer Automaten (vgl. Definition 3.21 für *Status*). Ferner legen wir fest, welche atomaren Aussagen in den Status einer Kripke-Struktur gelten können. Atomare Aussagen definieren hierbei, welche Zustände aktiv sind, welche Ereignisse anliegen und welche Prädikate auf dem Datenraum erfüllt sind. Wir definieren folgenden Datentyp $(\sigma, \epsilon, \delta)\,\mathit{atomar}$, um atomare Eigenschaften Hierarchischer Automaten zu beschreiben.

$$\begin{aligned}(\sigma, \epsilon, \delta)\,\mathit{atomar} \equiv_{\mathtt{t}}\ & \mathit{True}\\ \mid\ & \mathit{False}\\ \mid\ & \mathit{In}\ \sigma\\ \mid\ & \mathit{En}\ \epsilon\\ \mid\ & \mathit{Val}\ \ (\delta\ \mathit{data}) \to \mathit{bool}\end{aligned}$$

Der Typ der Kripke-Struktur eines Hierarchischen Automaten $(\sigma, \epsilon, \delta)\,\mathit{hakripke}$ ist durch folgendes Typsynonym beschrieben.

$$(\sigma, \epsilon, \delta)\,\mathit{hakripke} \equiv_{\mathtt{s}} (\,(\sigma, \epsilon, \delta)\,\mathit{status}, (\sigma, \epsilon, \delta)\,\mathit{atomar}\,)\,\mathit{kripke}$$

Wir instanziieren in dieser Typabkürzung den Typ für Kripke-Strukturen $(\sigma, \alpha)\,\mathit{kripke}$ aus Definition 5.1 mit dem Typ für Status Hierarchischer Automaten $(\sigma, \epsilon, \delta)\,\mathit{status}$ und dem auf dieser Seite eingeführten Typ für atomare Aussagen Hierarchischer Automaten $(\sigma, \epsilon, \delta)\,\mathit{atomar}$.

Zentrale Bestandteile von Kripke-Strukturen sind die Labelfunktionen und die Transitionsrelationen. Im Folgenden geben wir an, wie diese Komponenten für Hierarchische Automaten konstruiert werden können.

Wir beginnen mit der Definition einer Labelfunktion, die den Status eines Hierarchischen Automaten eine Menge von in diesem Status gültigen atomaren Aussagen zuordnet. Unter Verwendung des oben angegebenen Datentyps für atomare Aussagen definieren wir für einen Hierarchischen Automaten die Labelfunktion mit folgendem Operator. Der Operator ‘ repräsentiert hierbei – ähnlich zum Operator *map* für Listen – die Anwendung einer Funktion auf alle Elemente einer Menge.

$$
\begin{aligned}
&\mathit{HAToLabelFun} ::_{\mathsf{c}} (\sigma,\epsilon,\delta)\,\mathit{hierauto} \to ((\sigma,\epsilon,\delta)\,\mathit{status} \nrightarrow (\sigma,\epsilon,\delta)\,\mathit{atomar\,set})\\
&\mathit{HAToLabelFun}\ \mathit{A}\ \mathit{ST} \equiv_{\mathsf{c}} \mathsf{if}\ (\mathit{HA}\ \mathit{ST}) = \mathit{A}\\
&\qquad\qquad \mathsf{then}\ \ \mathsf{let}\ \ \mathit{In}_{\mathtt{P}} = (\lambda\ \mathit{S}.\ \mathit{In}\ \mathit{S})\ \text{`}\ (\mathit{Conf}\ \mathit{ST})\\
&\qquad\qquad\qquad\qquad \mathit{En}_{\mathtt{P}} = (\lambda\ \mathit{E}.\ \mathit{En}\ \mathit{E})\ \text{`}\ (\mathit{Events}\ \mathit{ST})\\
&\qquad\qquad\qquad\qquad \mathit{Val}_{\mathtt{P}} = \{\mathit{P}_{\mathtt{V}}.\ \exists\ \mathit{P}.\ \mathit{P}_{\mathtt{V}} = \mathit{Val}\ \mathit{P} \wedge\ \mathit{P}\,(\mathit{Data}\ \mathit{ST})\}\\
&\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathit{Some}\,(\mathit{In}_{\mathtt{P}} \cup\ \mathit{En}_{\mathtt{P}} \cup\ \mathit{Val}_{\mathtt{P}} \cup\ \{\mathit{True}\})\\
&\qquad\qquad \mathsf{else}\ \mathit{None}
\end{aligned}
$$

Die Labelfunktion einer Kripke-Struktur *M* ist durch eine partielle Funktion beschrieben (vgl. Definition 5.1), die bezüglich der in *M* gültigen Status total sein muss. Entsprechend prüfen wir in der Definition des Operators *HAToLabelFun* zunächst, ob der Status *ST*, der der Labelfunktion übergeben wird, ein gültiger Status ist. Ein Status *ST* ist genau dann gültig, wenn der in *ST* enthaltene Hierarchische Automat mit dem Hierarchischen Automaten übereinstimmt, für den die Labelfunktion zu konstruieren ist. Ist die Eigenschaft für *ST* erfüllt, so geben wir eine Menge atomarer Aussagen zurück, die alle in *ST* aktiven Zustände, alle in *ST* anliegenden Ereignisse und alle in *ST* gültigen Prädikate auf dem Datenraum enthält. Zusätzlich stellen wir sicher, dass die atomare Aussage *True* in der zurückzugebenden Menge enthalten ist.

Um die Transitionsrelation einer Kripke-Struktur für einen Hierarchischen Automaten anzugeben, verwenden wir Definition 3.29 von Seite 65. In dieser Definition ist ein Operator *StepStatus* angegeben, der für einen gegebenen Status den Folgestatus errechnet. Dieser Operator setzt eine so genannte *Closed-System-Semantik* für Hierarchische Automaten um. Dies bedeutet, dass während eines semantischen Zustandsübergangs von der Umgebung keine neuen Ereignisse entgegengenommen werden können. Lediglich intern generierte Ereignisse werden in die Ereignismenge des Folgestatus aufgenommen. Eine *Closed-System-Semantik* ist für die Spezifikation von reaktiven Systemen im Allgemeinen ungeeignet, da sich diese Systeme gerade durch eine intensive Kommunikation mit der Umgebung auszeichnen. Das Gegenstück zu einer *Closed-System-Semantik* ist eine *Open-System-Semantik*, in der in jedem Zeitschritt Ereignisse aus der Umgebung empfangen werden können und entsprechend in der Ereignismenge des Folgestatus sichtbar sind. Durch folgende leichte Adaption von Definition 3.29 erhalten wir einen neuen Operator, mit dem eine *Open-System-Semantik* ausgedrückt werden kann.

Definition 5.8 (Folgestatus in einer Open-System-Semantik) *Sei ST ein Status, T eine Menge von konfliktfreien Transitionen, U eine totale Update-Funktion auf dem Datenraum*

und E_U eine Menge von Ereignissen, die aus der Umgebung empfangen werden, dann ist der Folgestatus von ST unter Ausführung von T mit dem Operator $StepStatusOpen$ folgendermaßen festgelegt.

$$
\begin{aligned}
StepStatusOpen ::_c\ & [(\sigma,\epsilon,\delta)\ status,\ (\sigma,\epsilon,\delta)\ trans\ set,\ \delta\ update,\ \epsilon\ set] \\
& \rightarrow (\sigma,\epsilon,\delta)\ status \\
StepStatusOpen\ ST\ T\ U\ E_U\ \equiv_c\ & \mathsf{let}\ \ (HA, C, E, D) = Rep\_status\ ST; \\
& \qquad\qquad C' = StepConf\ HA\ C\ T; \\
& \qquad\qquad E' = (ActionEvents\ T) \cup E_U; \\
& \qquad\qquad D' = U\,!\,D \\
& \mathsf{in}\ \ \ Abs\_status\ (HA, C', E', D')
\end{aligned}
$$

□

Die Änderung gegenüber Definition 3.29 besteht darin, dass wir eine Menge von Ereignissen annehmen, die während der Ausführung des synchronen Zeitschrittes aus der Umgebung empfangen werden. Wir stellen sicher, dass diese Menge von Ereignissen im Folgestatus anliegt. Unter Verwendung des Operators $StepStatusOpen$ definieren wir mit dem Operator $HAToStepRel$ die Transitionsrelation einer Kripke-Struktur für einen gegebenen Hierarchischen Automaten A.

$$
\begin{aligned}
HAToStepRel ::_c\ & (\sigma,\epsilon,\delta)\ hierauto \rightarrow ((\sigma,\epsilon,\delta)\ status\ *\ (\sigma,\epsilon,\delta)\ status)\ set \\
HAToStepRel\ A\ \equiv_c\ & \{\,(ST, ST')\,. \\
& \qquad (HA\ ST) = A\ \wedge \\
& \qquad \exists\ E : \mathbb{P}\ (Events\ A)\,. \\
& \qquad\quad ((MTS\ ST \neq \emptyset) \Rightarrow \\
& \qquad\qquad \exists\ T : MTS\ ST.\ \exists\ U : SolveRacing_T\ T. \\
& \qquad\qquad\quad ST' = StepStatusOpen\ ST\ T\ U\ E)\ \wedge \\
& \qquad\quad ((MTS\ ST = \emptyset) \Rightarrow \\
& \qquad\qquad\quad ST' = StepStatusOpen\ ST\ \emptyset\ UpdateDefault\ E)\,\}
\end{aligned}
$$

Die so konstruierte Transitionsrelation ordnet jedem in A gültigen Status ST einen Folgestatus ST' zu. Die Definition von $HAToStepRel$ unterscheidet zwei Fälle.

Der erste Fall tritt genau dann ein, wenn der Operator MTS für ST mindestens eine maximale konfliktfreie Transitionsmenge zurückliefert (vgl. Definition 3.27). In diesem Fall ist ST' genau dann ein gültiger Folgestatus von ST, wenn ST' mit Hilfe des Operators $StepStatusOpen$ folgendermaßen gebildet werden kann. Wir wählen zunächst eine Transitionsmenge T aus, die in der mit MTS errechneten Menge enthalten ist. Anschließend bilden wir mit Hilfe des Operators $SolveRacing_T$ eine Menge von Update-Funktionen, die alle möglichen Effekte auf dem Datenraum beschreibt. Schließlich wählen wir aus dieser Menge eine Update-Funktion U aus und errechnen den Folgestatus ST' mit dem Operator $StepStatusOpen$ auf Basis von ST, T und U. Die Ereignismenge E beschreibt die von der Umgebung empfangenen Ereignisse und ist eine beliebige Teilmenge der in A gültigen Ereignisse.

Der zweite Fall tritt genau dann ein, wenn der Operator *MTS* für *ST* eine leere Menge zurückliefert. In diesem Fall ist *ST'* genau dann ein gültiger Folgestatus von *ST*, wenn *ST'* mit dem Operator *StepStatusOpen* auf Basis von *ST*, der leeren Transitionsmenge und der Default-Update-Funktion gebildet werden kann.

Schließlich definieren wir in Definition 5.9 einen Operator, mit dem für einen gegebenen Hierarchischen Automaten die Kripke-Struktur konstruiert werden kann.

Definition 5.9 (Kripke-Strukturen Hierarchischer Automaten) *Sei* `A` *ein Hierarchischer Automat, dann ist die semantische Interpretation von* `A` *durch folgende Kripke-Struktur festgelegt.*

$$\begin{aligned} \mathit{HAToKripke} &::_{\mathsf{c}} (\sigma,\epsilon,\delta)\,\mathit{hierauto} \to (\sigma,\epsilon,\delta)\,\mathit{hakripke} \\ \mathit{HAToKripke}\ A &\equiv_{\mathsf{c}} \mathit{Abs\_kripke}\,(\{\,\mathit{ST}.\ \mathit{HA}\ \mathit{ST} = A\,\}, \\ &\qquad\qquad \{\,\mathit{InitStatus}\ A\,\}, \\ &\qquad\qquad \mathit{HAToStepRel}\ A, \\ &\qquad\qquad \mathit{HAToLabelFun}\ A) \end{aligned}$$

□

Folgendes in Isabelle/HOL abgeleitete Theorem belegt, dass die Konstruktion aus Definition 5.9 zu einer wohlgeformten Kripke-Struktur im Sinne der Definition 5.2 führt.

Abgeleitetes Theorem 5.10 (Wohlgeformtheit der Kripke-Struktur eines HA)

$$\frac{S = \{\,\mathit{ST}.\ \mathit{HA}\ \mathit{ST} = A\,\} \qquad I = \{\,\mathit{InitStatus}\ A\,\} \qquad T = \mathit{HAToStepRel}\ A \qquad L = \mathit{HAToLabelFun}\ A}{\mathit{KripkeCorrect}\ S\ I\ T\ L} \qquad \text{(HAToKripkeCorrect)}$$

□

Mit Definition 5.9 haben wir die Semantik Hierarchischer Automaten durch Kripke-Strukturen beschrieben. Auf Basis dieser Beschreibung können wir prüfen, ob CTL-Formeln in dem Verhalten Hierarchischer Automaten erfüllbar sind oder nicht. Wir betrachten CTL-Formeln folgenden Typs.

$$(\sigma,\epsilon,\delta)\,\mathit{hactl} \equiv_{\mathsf{s}} ((\sigma,\epsilon,\delta)\,\mathit{status},\ (\sigma,\epsilon,\delta)\,\mathit{atomar})\,\mathit{ctl}$$

Analog zur Typabkürzung von $(\sigma,\epsilon,\delta)\,\mathit{hakripke}$ instanziieren wir hier den Typ für CTL-Formeln $(\sigma,\alpha)\,\mathit{ctl}$ mit dem Typ für Status Hierarchischer Automaten $(\sigma,\epsilon,\delta)\,\mathit{status}$ und dem Typ für atomare Aussagen Hierarchischer Automaten $(\sigma,\epsilon,\delta)\,\mathit{atomar}$.

Auf Basis dieses Typs führen wir einen Operator ein, mit dem die Gültigkeit einer CTL-Formel in dem Verhalten eines Hierarchischen Automaten ausgedrückt werden kann.

$$\begin{aligned} &\_ \models_{\mathtt{HA}} \_ ::_{\mathsf{c}} [(\sigma,\epsilon,\delta)\,\mathit{hierauto}, (\sigma,\epsilon,\delta)\,\mathit{hactl}] \to \mathit{bool} \\ &\_ \models_{\mathtt{HA}} \_ ::_{\mathsf{c}} \lambda\,\mathit{HA}\,F\ .\ (\mathit{HAToKripke}\ \mathit{HA}, \mathit{InitStatus}\ \mathit{HA}) \models F \end{aligned}$$

Mit Hilfe des Operators $\models_{\mathtt{HA}}$ formulieren wir in den folgenden Kapiteln dieser Ausarbeitung die Gültigkeit einer CTL-Formel F für den initialen Status eines Hierarchischen Automaten HA in folgender Art und Weise.

$$HA \models_{\mathtt{HA}} F$$

Ist F hierbei eine atomare Aussage, so können folgende von uns in Isabelle/HOL abgeleiteten Theoreme für die Ableitung eines Beweises verwendet werden.

Abgeleitete Theoreme 5.11 (Atomare Aussagen in einem HA)

$$HA \models_{\mathtt{HA}} Atom\ True \qquad \text{(AtomTrueHA)}$$

$$HA \models_{\mathtt{HA}} Atom\,(En\,E) = E \in Events(InitStatus\ HA) \qquad \text{(AtomEventsHA)}$$

$$HA \models_{\mathtt{HA}} Atom\,(In\,S) = S \in States(InitStatus\ HA) \qquad \text{(AtomStatesHA)}$$

$$HA \models_{\mathtt{HA}} Atom\,(Val\,P) = P(InitData\ HA) \qquad \text{(AtomDataHA)}$$

□

Auf Basis dieser und noch weiterer in unserer Theorie abgeleiteter Theoreme sind wir dazu in der Lage, die Erfüllbarkeit von CTL-Formeln für Hierarchische Automaten abzuleiten.

5.5 Zusammenfassung

Wir haben in diesem Kapitel Isabelle/HOL-Theorien vorgestellt, in denen CTL-Operatoren und ihre Semantik beschrieben sind. Die Semantik haben wir auf zwei unterschiedliche Arten definiert. Zum einen formalisierten wir Kripke-Strukturen und die Menge, der in ihr gültigen Berechnungspfade. Darauf aufbauend definierten wir eine induktive Semantik, mit der für jede CTL-Formel überprüft werden kann, ob sie in den Berechnungspfaden der Kripke-Struktur erfüllt ist. Zum anderen definierten wir für jeden CTL-Operator eine auf Fixpunkten des μ-Kalküls basierende Semantik. Beide Definitionen der Semantik sind in der Literatur weit verbreitet.

Um eine CTL-Formel nicht nur bezüglich einer Kripke-Struktur, sondern auch bezüglich eines Hierarchischen Automaten auswerten zu können, entwickelten wir im letzten Abschnitt dieses Kapitels ein Vorgehen, mit dem aus einem Hierarchischen Automaten eine Kripke-Struktur konstruiert werden kann. Die Berechnungspfade dieser Kripke-Struktur beschreiben die Semantik eines Hierarchischen Automaten. Wir haben gezeigt, dass die von uns vorgeschlagene Konstruktion die Eigenschaften der Wohlgeformtheit von Kripke-Strukturen erfüllt.

Schließlich haben wir einen Operator angegeben, mit dessen Hilfe wir ausdrücken können, dass eine CTL-Formel in dem Verhalten eines Hierarchischen Automaten – bezogen auf den initialen Status – erfüllt ist. Diese Aussage beschreibt das so genannte *Model-Checking-Problem* für Hierarchische Automaten. Dieses Problem ist durch einen Model-Checker nur dann automatisch zu verifizieren, wenn der Datenraum, der dem zu überprüfenden Hierarchischen Automaten zugrunde liegt, endlich ist.

Mit diesem Kapitel schließen wir den ersten Teil dieser Ausarbeitung ab. Wir haben in diesem Teil die abstrakte Syntax und Semantik von Statecharts durch Hierarchische Automaten in Isabelle/HOL formalisiert. Ferner haben wir diese Formalisierung um die Beschreibung von partitionierten Datenräumen angereichert, die einer Statecharts-Spezifikation zugrunde liegen können. Schließlich stellten wir eine Formalisierung der temporalen Logik CTL vor und führten die Formalisierungen für Statecharts und der CTL in einer Theorie zusammen. Wir haben mit diesen Ergebnissen die notwendigen Voraussetzungen für die Entwicklung einer Datenabstraktionstechnik für Statecharts geschaffen. In Teil II dieser Ausarbeitung werden wir diese Technik vorstellen. Mit ihr können unendliche Datenräume Hierarchischer Automaten zu endlichen Datenräumen abstrahiert werden. Das dabei entstehende abstrakte Modell kann mit Hilfe eines Model-Checkers automatisch verifiziert werden.

Teil II

Struktur- und eigenschaftserhaltende Datenabstraktion für Statecharts

Kapitel 6

Konzeption der Abstraktionstheorie

Das in Teil I dieser Arbeit vorgestellte Framework enthält Formalisierungen für die Temporallogik CTL und die Spezifikationssprache Statecharts. Beide Formalisierungen sind in einer Theorie zusammengeführt. Wir haben damit eine Infrastruktur aufgebaut, in der Beweise über Statecharts-Spezifikationen geführt werden können. Ein Nachteil dieses Frameworks ist, dass die Beweise im Allgemeinen interaktiv zu führen sind. Um einen höheren Grad an Automatisierung zu erreichen, kombinieren wir in Teil II dieser Ausarbeitung den Theorembeweiser Isabelle/HOL mit anderen Verifikationswerkzeugen. Die Idee unseres Ansatzes besteht darin, die Datenräume von Statecharts-Spezifikationen in Isabelle/HOL zu abstrahieren und auf den dabei entstehenden abstrakten Statecharts-Spezifikationen durch bekannte Model-Checker automatische Analysen durchzuführen. Wir beschreiben hierzu eine Datenabstraktionstechnik für Statecharts, die eigenschafts- und strukturerhaltend ist.

Teil II dieser Ausarbeitung gliedert sich in drei Kapitel. Zunächst stellen wir in diesem Kapitel ein Konzept vor, auf dessen Basis eine Datenabstraktion für Statecharts-Spezifikationen umgesetzt werden kann und beschreiben, warum das Konzept zu einer eigenschaftserhaltenden Abstraktion führt. In Kapitel 7 setzen wir das Abstraktionskonzept in einer Isabelle/HOL-Theorie um. Schließlich stellen wir in Kapitel 8 zwei Taktiken vor, mit denen einerseits ein von uns entwickelter Abstraktionsalgorithmus und andererseits der Model-Checker SMV in den Beweisprozess von Isabelle eingebunden werden kann.

Speziell in diesem Kapitel beschreiben wir das Konzept der von uns entwickelten Datenabstraktionstechnik. Das Kapitel gliedert sich in drei Abschnitte. In Abschnitt 6.1 stellen wir Galois-Korrespondenzen vor und führen den Begriff der Überapproximation ein, mit dem das universale Fragment der CTL erhalten werden kann. Im darauf folgenden Abschnitt geben wir an, wie die Überapproximation eines Sequentiellen Automaten gebildet wird. Schließlich ist in Abschnitt 6.3 dargestellt, wie unter Verwendung der Überapproximation von Sequentiellen Automaten ein Hierarchischer Automat in kompositionaler Art und Weise abstrahiert werden kann.

6.1 Eigenschaftserhaltung und Galois-Korrespondenzen

In diesem Abschnitt wollen wir die Grundlagen für eigenschaftserhaltende Datenabstraktionstechniken legen. Wir verfolgen dabei das Ziel, die Eigenschaftserhaltung von Formeln der CTL zu untersuchen. *Eigenschaftserhaltung* bedeutet hierbei, dass aus der Gültigkeit einer CTL-Formel im abstrakten Modell immer die Gültigkeit der CTL-Formel im konkreten Modell abgeleitet werden kann. Umgekehrt können aus einer im abstrakten Modell verletzten

CTL - Formel allerdings zunächst keine Rückschlüsse auf deren Gültigkeit im konkreten Modell gemacht werden. Auf Basis der in Teil I vorgestellten Formalisierung, können wir die *Eigenschaftserhaltung* folgendermaßen formal fassen.

$$(Abs_{\mathtt{HA}}\, HA \models_{\mathtt{HA}} Abs_{\mathtt{CTL}}\, F) \Rightarrow (HA \models_{\mathtt{HA}} F)$$

Hierbei beschreibt der in Abschnitt 5.4 eingeführte Folgerungsoperator $\models_{\mathtt{HA}}$ die Gültigkeit einer temporallogischen CTL-Formel F ab dem initialen Zustand eines Hierarchischen Automaten HA. Wir setzen weiterhin voraus, dass die Operatoren $Abs_{\mathtt{HA}}$ und $Abs_{\mathtt{CTL}}$ zur Abstraktion von Hierarchischen Automaten und CTL-Formeln gegeben sind. Die Beantwortung der Frage, ob und wie solche Operatoren definiert werden können, ist Gegenstand dieses Kapitels.

Im Wesentlichen sind in der Literatur zwei Ansätze für eigenschaftserhaltende Abstraktionen zu finden: *Über- und Unterapproximation* [CGL94, CC77]. In einer *Überapproximation* kann das abstrakte Modell neues Verhalten enthalten, jedoch darf Verhalten, das im konkreten Modell möglich war, nicht verloren gehen. Das bedeutet, dass Eigenschaften des universalen Fragments von CTL ($\forall$CTL), die auf allen Pfaden des abstrakten Modells gültig sind, auch auf allen Pfaden des konkreten Modells folgen. Im Gegensatz dazu kann in einer *Unterapproximation* kein neues Verhalten hinzugefügt, aber Verhalten, das im konkreten Modell möglich war, reduziert werden. Entsprechend werden Eigenschaften des existenziellen Fragments von CTL ($\exists$CTL) durch unterapproximierte abstrakte Modelle erhalten. In der Arbeit von Dennis Dams [Dam96] werden zwei abstrakte Modelle aus einem zu verifizierenden konkreten Modell erzeugt, die eine Über- bzw. eine Unterapproximation darstellen. Abhängig von der zu verifizierenden Eigenschaft wird das geeignete abstrakte Modell ausgewählt. In jüngeren Arbeiten [HJS01] werden die Informationen von Über- und Unterapproximation in nur einem – dem so genannten *Modalen* – Transitionssystem repräsentiert. *Modale Transitionssysteme* [LT88, Lar89] unterscheiden *may* und *must*-Transitionen und basieren auf einer dreiwertigen Logik. Zur automatischen Verifikation von Eigenschaften auf Modalen Transitionssystemen wurde eine spezielle Technik des Model-Checkings vorgeschlagen [Hut02, GHJ01]. Die dort beschriebenen Algorithmen besitzen einen Aufwand, der mit traditionellen Model-Checkern vergleichbar ist.

Die theoretische Basis für Über- und Unterapproximationen bildet die Theorie der Galois-Korrespondenzen [CC77, MSS86]. Wir haben eine Formalisierung von Galois-Korrespondenzen in Isabelle/HOL angegeben [HK03a]. Diese Theorie wird hier nur in Auszügen vorgestellt, da eine ausführliche Darstellung für das Grundverständnis der Abstraktionstheorie von Statecharts nicht notwendig ist. Zunächst geben wir eine Definition für *Galois-Korrespondenzen* an.

Definition 6.1 (Galois-Korrespondenz) *Seien C und A vollständige Verbände mit den Ordnungsrelationen $\leq_{\mathtt{C}}$ bzw. $\leq_{\mathtt{A}}$, weiterhin seien α und γ monotone Abbildungen mit $\alpha : C \longrightarrow A$ und $\gamma : A \longrightarrow C$, dann bilden diese Komponenten eine Galois-Korrespondenz, wenn folgende Bedingung erfüllt ist.*

$$\forall\, a : A, c : C.\ (\alpha\, c \leq_{\mathtt{A}} a) = (c \leq_{\mathtt{C}} \gamma\, a)$$

□

Die Abbildung α aus Definition 6.1 wird häufig auch als *Abstraktionsfunktion* und die Abbildung γ als *Konkretisierungsfunktion* bezeichnet.

Weiterhin erfüllen die Abbildungen in einer Galois-Korrespondenz die folgenden *Adjunktionstheoreme*. Diese Theoreme ermöglichen es, α durch γ bzw. γ durch α, unter Verwendung von Supremum und Infimum, auszudrücken.

$$\alpha\ c = \bigwedge \{\, a : A \mid c \leq_C \gamma\ a \,\}$$
$$\gamma\ a = \bigvee \{\, c : C \mid \alpha\ c \leq_A a \,\}$$

In dieser Arbeit setzen wir Galois-Korrespondenzen ein, um Datenräume von abstrakten und konkreten Prädikaten miteinander in Beziehung zu setzen.

Definition 6.2 (Galois-Korrespondenz für Prädikate) *Sei δ_C `pred` der Typ für Prädikate über einem konkreten Datenraum δ_C. Sei weiterhin δ_A `pred` der Typ für Prädikate über einem abstrakten Datenraum δ_A, dann ist eine Galois-Korrespondenz `GC` über (δ_C, δ_A) durch ein Tupel (F_A, F_C) repräsentiert. Hierbei ist*

- F_A *die Abstraktionsfunktion und*
- F_C *die Konkretisierungsfunktion*

der Galois-Korrespondenz `GC`. Diese Komponenten müssen die interne Konsistenzbedingung `GaloisCorrect` über Galois-Korrespondenzen erfüllen, die in Definition 6.3 angegeben ist. Der Typ (δ_C, δ_A) `galois` besteht aus allen Galois-Korrespondenzen über (δ_C, δ_A).

$$\begin{aligned}(\delta_C, \delta_A)\ \texttt{galois} \equiv_t \{\, (F_A, F_C) \mid\ & \\ & (F_A :: \delta_C\ \texttt{pred} \longrightarrow \delta_A\ \texttt{pred}) \\ & (F_C :: \delta_A\ \texttt{pred} \longrightarrow \delta_C\ \texttt{pred}). \\ & \texttt{GaloisCorrect}\ F_A\ F_C \,\} \\ & \quad \textit{gerechtfertigt durch } \mathsf{GaloisNonEmpty}\end{aligned} \tag{6.1}$$

□

Der Typkonstruktor `pred` ist in Abschnitt 5.3 eingeführt. Das Theorem GaloisNonEmpty belegt, dass die in Definition 6.2 abstrahierte Menge nicht leer ist. Um auf die Komponenten einer Galois-Korrespondenz zugreifen zu können, führen wir folgende Selektionsoperatoren ein. Die Abstraktionfunktion kann mit dem Operator `Abs` und die Konkretisierungsfunktion mit dem Operator `Conc` bestimmt werden.

Definition 6.3 (Wohlgeformte Galois-Korrespondenz für Prädikate) *Sei F_A eine Abstraktionsfunktion und F_C eine Konkretisierungsfunktion, dann bilden diese Komponenten eine wohlgeformte Galois-Korrespondenz, wenn folgende Bedingungen erfüllt sind.*

$$\begin{aligned}\texttt{GaloisCorrect} ::_c [\, \delta_C\ \texttt{pred} \longrightarrow \delta_A\ \texttt{pred}, \delta_A\ \texttt{pred} \longrightarrow \delta_C\ \texttt{pred} \,] & \\ \longrightarrow \texttt{bool} & \\ \texttt{GaloisCorrect}\ F_A\ F_C \equiv_c \forall\ pc\ pa\ .\ \texttt{Valid}\,(F_A\ pc \Rightarrow_P pa) = \texttt{Valid}\,(pc \Rightarrow_P F_C\ pa) & \\ \wedge\ \texttt{Mono}\ F_A \wedge \texttt{Mono}\ F_C &\end{aligned}$$

□

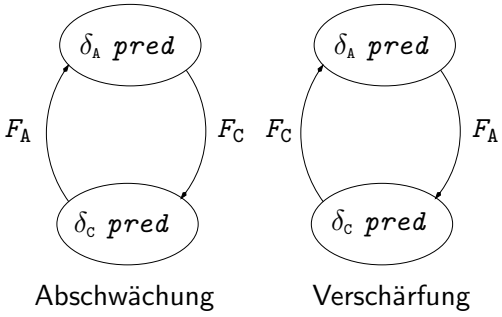


Abbildung 6.1: Galois-Korrespondenzen für Über- und Unterapproximationen

Der in Definition 6.3 verwendete Operator $\Rightarrow_P$ ist in Abschnitt 5.3 eingeführt worden und beschreibt die Ordnungsrelation auf Prädikaten. Der Operator *Mono* garantiert die Monotonie einer Funktion. Der Operator *Valid* repräsentiert die Allgemeingültigkeit eines Prädikates.

$$Valid ::_c \sigma\ pred \rightarrow bool$$
$$Valid\ P \equiv_c \forall s.\ P\ s$$

In Abbildung 6.1 sind zwei Galois-Korrespondenzen dargestellt, in denen die Datenräume von abstrakten und konkreten Prädikaten miteinander in Beziehung gesetzt werden. Auf der linken Seite in Abbildung 6.1 ist eine Galois-Korrespondenz *GC* vom Typ $(\delta_c, \delta_A)\ galois$ angegeben, die für Überapproximationen eines Modells eingesetzt werden kann. Hierbei basiert das zu abstrahierende Modell auf dem Datenraum δ_c und die zu konstruierende Überapproximation des Modells auf dem Typ δ_A. Es wird ausgenutzt, dass durch die Anwendung der Abstraktionsfunktion F_A Prädikate vom Typ $\delta_c\ pred$ abgeschwächt werden können. Folgendes in Isabelle/HOL abgeleitetes Theorem belegt diese Aussage.

$$\frac{F_A = Abs\,GC \qquad F_C = Conc\,GC}{Valid\,(pc \Rightarrow_P F_C\,(F_A\,pc))} \qquad \text{(WeakerPredicate)}$$

Analog dazu ist auf der rechten Seite in Abbildung 6.1 eine Galois-Korrespondenz *GC* vom Typ $(\delta_A, \delta_c)\ galois$ angegeben, die für Unterapproximationen eines Modells eingesetzt werden kann. Hierbei basiert wiederum das zu abstrahierende Modell auf dem Datenraum δ_c und die zu konstruierende Unterapproximation des Modells auf dem Typ δ_A. Man beachte aber, dass im Vergleich zur Galois-Korrespondenz auf der linken Seite der Abbildung eine Neuinterpretation von abstrakter und konkreter Datendomain erfolgt ist. Mit der Konkretisierungsfunktion F_C werden in dieser Galois-Korrespondenz Prädikate vom Typ $\delta_c\ pred$ in Prädikate vom Typ $\delta_A\ pred$ transformiert. Es wird ausgenutzt, dass durch die Anwendung der Konkretisierungsfunktion F_C Prädikate vom Typ $\delta_c\ pred$ verschärft werden können. Folgendes in Isabelle/HOL abgeleitetes Theorem belegt diese Aussage.

$$\frac{F_A = Abs\,GC \qquad F_C = Conc\,GC}{Valid\,(F_A\,(F_C\,pa) \Rightarrow_P pa)} \qquad \text{(StrongerPredicate)}$$

Im Sinne der hier eingeführten Terminologie für Galois-Korrespondenzen stellt also die Überapproximation eines Prädikates eine Abstraktion und die Unterapproximation eines Prädikates eine Konkretisierung dar.

Für Galois-Korrespondenzen, die einen Zusammenhang zwischen Datenräumen von abstrakten und konkreten Prädikaten festlegen, werden Abstraktions- und Konkretisierungsfunktion häufig aus einer so genannten *Retrieve-Relation* R abgeleitet [LGS+95]. In Analogie zur Theorie der Datenverfeinerung [Jon90, Hoa72, DB01] repräsentiert R eine Relation, mit der abstrakte und konkrete Datenwerte zueinander in Beziehung gesetzt werden. Auf Basis von R werden dann über die *Schwächste Vorbedingung* und die *Strengste Nachbedingung* [Bac88] Funktionen abgeleitet, mit denen Prädikate über abstrakten und konkreten Datenwerten zueinander in Beziehung gesetzt werden können.

Wir verwenden in dieser Arbeit spezielle Retrieve-Relationen, die die Funktionseigenschaft erfüllen und die total sind. Eine Retrieve-Funktion ist durch folgende Typabkürzung beschrieben. Hierbei ist $(\delta_c, \delta_a)\ abs$ der Typ der HOL-Funktionen von δ_c nach δ_a.

$$(\delta_c, \delta_a)\ abs \equiv_s \delta_c \to \delta_a$$

Mit folgenden Konstantendefinitionen definieren wir auf Basis einer Retrieve-Funktion häufig eingesetzte Galois-Korrespondenzen zur Transformation von Prädikaten [LGS+95].

$$\begin{aligned}
SP &::_c (\delta_c, \delta_a)\ abs \to (\delta_c\ pred \to \delta_a\ pred)\\
SP\ R\ P &\equiv_c \lambda\ a.\ \exists\ c.\ (R\ c) = a \wedge P\ c\\
WP &::_c (\delta_c, \delta_a)\ abs \to (\delta_a\ pred \to \delta_c\ pred)\\
WP\ R\ P &\equiv_c \lambda\ c.\ \forall\ a.\ (R\ c) = a \Rightarrow P\ a\\
SP^{-1} &::_c (\delta_c, \delta_a)\ abs \to (\delta_a\ pred \to \delta_c\ pred)\\
SP^{-1}\ R\ P &\equiv_c \lambda\ c.\ \exists\ a.\ (R\ c) = a \wedge P\ a\\
WP^{-1} &::_c (\delta_c, \delta_a)\ abs \to (\delta_c\ pred \to \delta_a\ pred)\\
WP^{-1}\ R\ P &\equiv_c \lambda\ a.\ \forall\ c.\ (R\ c) = a \Rightarrow P\ c
\end{aligned}$$

Auf Basis dieser definierten Funktionen können nun Galois-Korrespondenzen für die beiden in Abbildung 6.1 dargestellten Fälle angegeben werden.

Definition 6.4 (Prädikatentransformer als Galois-Korrespondenz) *Sei durch δ_c ein konkreter und durch δ_a ein abstrakter Datenraum repräsentiert, weiterhin sei $R : (\delta_c, \delta_a)\ abs$ eine Abstraktionsfunktion, so lassen sich aus R die Galois-Korrespondenz $GaloisOver$ zur Abschwächung und die Galois-Korrespondenz $GaloisUnder$ zur Verschärfung von Prädikaten über dem konkreten Datenraum definieren.*

$$\begin{aligned}
GaloisOver &::_c (\delta_c, \delta_a)\ abs \to (\delta_c, \delta_a)\ galois\\
GaloisOver\ R &\equiv_c Abs_galois\ (SP\ R, WP\ R)\\
GaloisUnder &::_c (\delta_c, \delta_a)\ abs \to (\delta_a, \delta_c)\ galois\\
GaloisUnder\ R &\equiv_c Abs_galois\ (SP^{-1}\ R, WP^{-1}\ R)
\end{aligned}$$

□

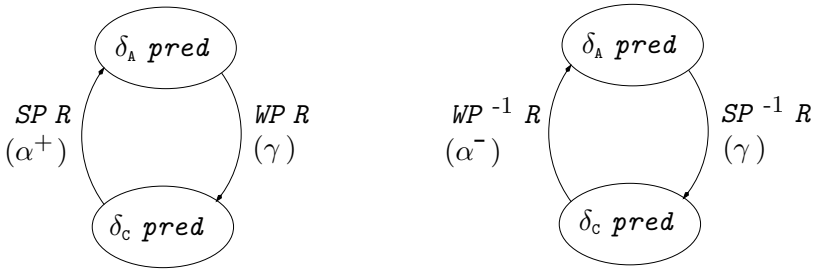


Abbildung 6.2: Galois-Korrespondenzen mit Prädikatentransformern

Folgende in Isabelle/HOL abgeleiteten Theoreme belegen, dass die in Definition 6.4 zum Bilden einer Galois-Korrespondenz verwendeten Funktionenpaare die Wohlgeformtheitseigenschaft von Galois-Korrespondenzen erfüllen.

$$GaloisCorrect\ (SP\ R)\ (WP\ R) \qquad \text{(OverApproxGalois)}$$

$$GaloisCorrect\ (SP^{-1}\ R)\ (WP^{-1}\ R) \qquad \text{(UnderApproxGalois)}$$

Zwischen der Abstraktionsfunktion von *GaloisOver* und der Konkretisierungsfunktion von *GaloisUnder* gilt folgende Dualitätsbeziehung.

$$SP\ R\ (\neg_P\ P) = \neg_P\ (WP^{-1}\ R\ P) \qquad \text{(DualityWPInvSP)}$$

Da die Retrieve-Relation in unserer Theorie auf eine totale Funktion spezialisiert ist, können wir zusätzlich ableiten, dass die Konkretisierungsfunktion von *GaloisOver* mit der Abstraktionsfunktion von *GaloisUnder* identisch ist.

$$WP\ R = SP^{-1}\ R \qquad \text{(WPIsSPInv)}$$

Die abgeleiteten Eigenschaften über Galois-Korrespondenzen werden in Kapitel 7 ausgenutzt, um die Definitionen für Konstruktionsoperatoren zur Abstraktion zu definieren und in Abschnitt 8.2, um die ML-Implementierung des Abstraktionsalgorithmus zu vereinfachen.

Zusammenfassend stellen wir die wichtigsten Ergebnisse der eingeführten Theorie über Galois-Korrespondenzen in Abbildung 6.2 noch einmal graphisch dar. Zur besseren Lesbarkeit der Schaubilder wird in den folgenden Abschnitten dieses Kapitels die Abstraktionsfunktion von *GaloisOver* mit α^+ und die Konkretisierungsfunktion von *GaloisUnder* mit α^- abgekürzt. Die Konkretisierungsfunktion von *GaloisOver* und die zu ihr identische Abstraktionsfunktion von *GaloisUnder* (vgl. Theorem WPIsSPInv) werden künftig mit γ abgekürzt. Wir verzichten bei diesen Bezeichnungen absichtlich auf die explizite Darstellung des Parameters *R*.

6.2 Datenabstraktion für Sequentielle Automaten

Basierend auf der Formalisierung von Sequentiellen Automaten aus Abschnitt 3.1.1 führen wir in diesem Abschnitt eine Technik zur Generierung von Überapproximationen für Sequentielle Automaten ein. Die Technik dient dazu, den zugrunde liegenden Datenraum eines Sequentiellen Automaten zu abstrahieren. Die Struktur des Sequentiellen Automaten bleibt dabei erhalten. Zunächst geben wir einen intuitiven Eindruck von der Abstraktiontechnik anhand eines kleinen Beispiels. Danach stellen wir die semantischen Besonderheiten von synchronen Sprachen vor und beschreiben ein systematisches Vorgehen, mit dem die Überapproximation eines Sequentiellen Automaten schrittweise gebildet werden kann. Abschließend begründen wir, warum eine Unterapproximation von Sequentiellen Automaten in unserer Theorie in vielen Fällen nicht möglich ist.

6.2.1 Beispiel für eine Prädikatenabstraktion

Die Prädikatenabstraktion [GS97, DDS99] ist eine weit verbreitete Technik zur Konstruktion von eigenschaftserhaltenden Abstraktionen. Hierbei ist der abstrakte Datenraum aus einer endlichen Anzahl von booleschen Variablen zusammengesetzt. Jede dieser Variablen repräsentiert ein Prädikat über dem konkreten Datenraum. Die Qualität des abstrakten Modells hängt von der Güte der gewählten Prädikate ab. Im Allgemeinen werden diese von einem Experten des zu behandelnden Problembereichs festgelegt. Unsere Abstraktionstheorie ist nicht auf die Prädikatenabstraktion beschränkt. Allerdings ist die Prädikatenabstraktion ein häufiger und ein intuitiv verständlicher Anwendungsfall. Deshalb verwenden wir in dem einführenden Beispiel aus Abbildung 6.3 eine Prädikatenabstraktion.

Links oben in Abbildung 6.3 ist der zu abstrahierende Sequentielle Automat dargestellt. Er besteht aus den Zuständen S_1 und S_2 und einer Transition zwischen ihnen. Der dem Sequentiellen Automaten zugrunde liegende Datenraum ist rechts neben dem konkreten Modell angegeben und besteht aus einer natürlichen Zahl x. Die Transition ist in Abhängigkeit von x spezifiziert. Sie ist nur dann aktiviert, wenn der Guard $x > 5$ erfüllt ist. Bei Ausführung der Transition wird x inkrementiert.

Unsere Abstraktionstechnik definiert für einen Sequentiellen Automaten und eine gegebene Abstraktionsfunktion ein abstraktes Modell. Das abstrakte Modell ist durch einen strukturell ähnlichen Sequentiellen Automaten repräsentiert und beschreibt eine Überapproximation des konkreten Modells. Folglich erhält der abstrakte Sequentielle Automat die Eigenschaften des universalen Fragments der CTL. Der mittlere Teil in Abbildung 6.3 zeigt den abstrakten Sequentiellen Automaten für das einführende Beispiel. Man beachte, dass das abstrakte Modell von der Struktur sehr ähnlich zu dem konkreten Modell ist. So sind die Zustände S_1 und S_2 direkt übernommen worden. Weiterhin wurde die konkrete Transition durch zwei abstrakte Transitionen repräsentiert. Lediglich die selbstbezügliche Transition am Zustand S_1 wurde komplett neu hinzugefügt. Rechts neben dem abstrakten Sequentiellen Automaten ist der abstrakte Datenraum angegeben. Er besteht aus den booleschen Variablen B_1 und B_2. Die darunter definierte Abstraktionsfunktion legt fest, dass B_1 durch das Prädikat $x > 4$ und B_2 durch das Prädikat $x \leq 6$ repräsentiert ist. Diese Prädikate werden von einem Experten vorgegeben, der damit eine aus seiner Sicht sinnvolle Partitionierung des Datenraumes festlegt.

Der Kern der Abstraktionstechnik besteht darin, datenbehaftete Bestandteile des konkreten Modells, wie Guards und Update-Funktionen, durch ihre abstrakten Gegenstücke zu

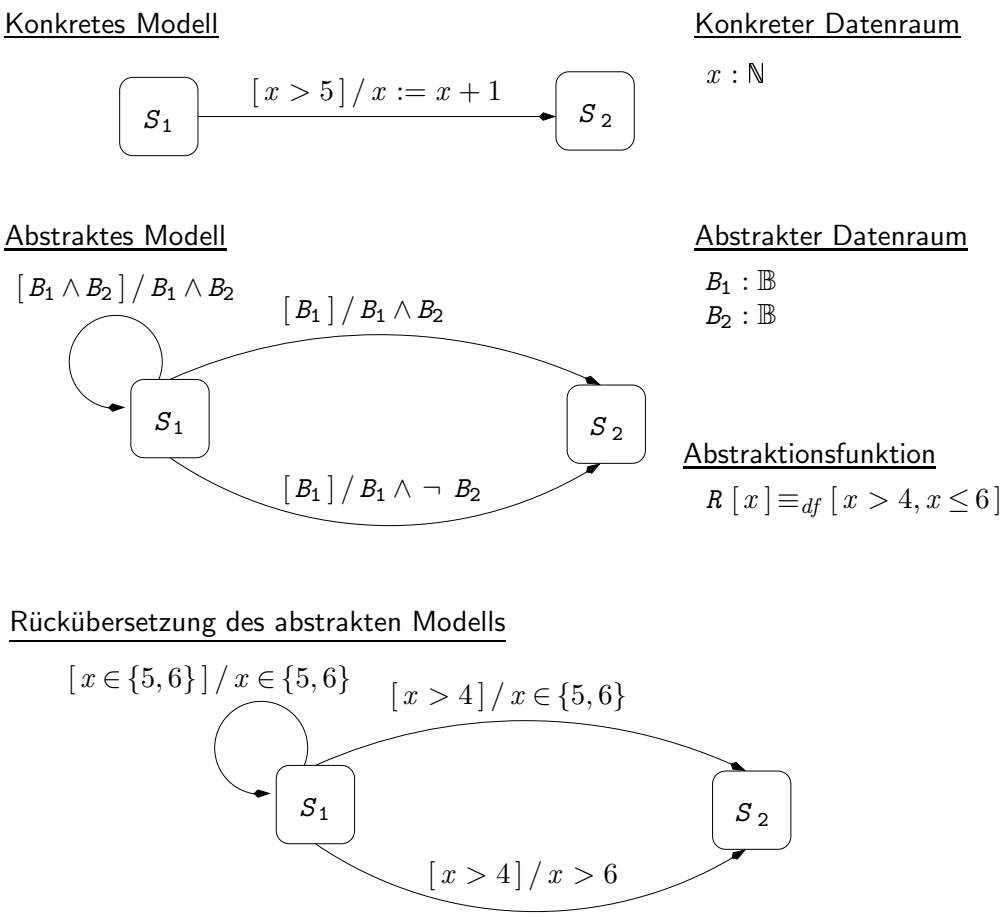


Abbildung 6.3: Datenabstraktion eines Sequentiellen Automaten für eine Prädikatenabstraktion

ersetzen. Da die dabei entstehenden Ausdrücke häufig nur schwer zu verstehen sind, schlagen wir zum besseren Verständnis des Abstraktionsprozesses eine Rückübersetzung des abstrakten Modells vor. Hierbei werden die auf Basis der booleschen Variablen ausgedrückten abstrakten Guards und Update-Funktionen durch die sie repräsentierenden Prädikate ersetzt. Durch dieses Vorgehen erhält der Benutzer unserer Abstraktionstechnik einen klaren Eindruck davon, wieviel Verhalten durch die von ihm vorgegebene Abstraktionsfunktion zum ursprünglichen Modell hinzugefügt worden ist. Die Rückübersetzung unseres einführenden Beispiels ist im unteren Teil von Abbildung 6.3 angegeben.

Im Folgenden beschreiben wir ein systematisches Vorgehen zur Konstruktion der Überapproximation für einen gegebenen Sequentiellen Automaten und eine gegebene Abstraktionsfunktion. Wir geben zunächst eine knappe Einführung in die semantischen Besonderheiten synchroner Sprachen, da die Sequentiellen Automaten zu dieser Sprachfamilie gehören.

6.2.2 Synchrone Sprachfamilie und implizites Verhalten

Statecharts und folglich auch Hierarchische Automaten gehören zur Familie der *synchronen Sprachen*, die Anfang der 1980er Jahre entwickelt wurden. Prominente Vertreter unter den synchronen Programmiersprachen sind Esterel [BG92], Lustre [HCRP91] und Signal [GGBdM91]. Als wichtige synchrone Spezifikationsprachen wollen wir den Statecharts-Dialekt Argos [Mar91] und deren Weiterentwicklung – genannt SynchCharts [And96] – erwähnen. Letztere lassen sich direkt in Esterel abbilden und werden von dem Werkzeug EsterelStudio unter dem Namen Safe-State-Machines (SSMs) [And03] unterstützt. Eine knappe und übersichtliche Einführung zu synchronen Sprachen ist zum Beispiel von Rheinhard von Hanxleden [vH05] zusammengestellt worden.

Synchrone Sprachen basieren auf einer *synchronen Schrittsemantik*, die wir in Abschnitt 3.3 bereits eingeführt und für Hierarchische Automaten formalisiert haben. Man beachte, dass wir in dieser Arbeit die Semantik Sequentieller Automaten als einen Spezialfall der Semantik Hierarchischer Automaten betrachten. Hierzu wird ein Sequentieller Automat als ein Hierarchischer Automat ohne Hierarchie angesehen (vgl. Definition 3.7). Die Kernidee einer synchronen Schrittsemantik besteht darin, dass in jedem semantischen Systemzustand – synchronisiert durch eine globale Uhr – alle parallel geschalteten Komponenten des Systems einen gemeinsamen Berechnungsschritt ausführen. Im Folgenden betrachten wir eine spezielle Situation in dieser Semantik, in der so genanntes *implizites Verhalten* auftritt.

Sollten zu Beginn eines Berechnungsschritts keine aktivierten Transitionen (vgl. Definition 3.26) existieren, so wird ein *trivialer Berechnungsschritt* ausgeführt. Der Effekt eines trivialen Berechnungsschritts besteht darin, dass der Datenraum mit der zuvor gültigen Datenraumbelegung erneut beschrieben wird. Es werden weder neue Ereignisse erzeugt, noch andere Zustände aktiviert. Lediglich von der Umgebung können neue Ereignisse empfangen werden. Um das Verhalten von trivialen Berechnungsschritten in einem Sequentiellen Automaten graphisch zu veranschaulichen, vervollständigen wir diese Automaten mit *impliziten Transitionen*. Eine implizite Transition wird mit einem geeigneten Guard und einer *Default-Update-Funktion* (vgl. Definition 4.9) markiert. Abbildung 6.4 zeigt hierfür ein Beispiel.

Der in Abbildung 6.4 dargestellte Sequentielle Automat besteht aus den Zuständen S_1 und S_2. Die von S_1 nach S_2 führende Transition ist mit dem Guard G_C und der Update-Funktion U_C markiert. Wir betrachten einen semantischen Zustandsübergang, in dem der Zustand S_1 aktiv ist und in dem die aktuelle Datenbelegung den Guard G_C nicht erfüllt. In dieser Situation wird ein trivialer Berechnungsschritt ausgeführt, der in der Abbildung durch

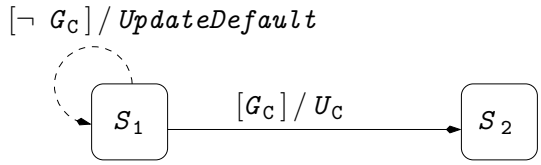


Abbildung 6.4: Implizites Verhalten eines Sequentiellen Automaten

eine selbstbezügliche Transition repräsentiert ist. Diese Transition ist gestrichelt gezeichnet, um zu veranschaulichen, dass es sich um implizites Verhalten handelt. Der Guard der gestrichelten Transition $\neg\ G_C$ ist so konstruiert, dass das implizite Verhalten immer genau dann auftritt, wenn die andere aus S_1 herausführende Transition nicht aktiviert werden kann. Im Allgemeinen ist der Guard der impliziten Transition als Konjunktion der negierten Guards aller anderen aus S_1 herausführenden Transitionen zu konstruieren.

6.2.3 Überapproximation von Sequentiellen Automaten

Um die Überapproximation eines Sequentiellen Automaten unter Berücksichtigung von implizitem Verhalten zu bilden, konstruieren wir schrittweise einen neuen abstrakten Automaten. Unter der Überapproximation eines Sequentiellen Automaten verstehen wir präziser formuliert die Überapproximation des zugrunde liegenden Datenraumes, d.h. die Struktur des Automaten bleibt in wesentlichen Teilen erhalten und datenabhängige Bestandteile werden durch geeignete abstrakte Gegenstücke ersetzt. Die Konstruktion besteht aus den folgenden Schritten:

1. Wir konstruieren einen zum konkreten Modell strukturell sehr ähnlichen Automaten. Strukturelle Merkmale, die hierbei erhalten werden, sind die Zustände und die Transitionen zwischen ihnen. Ferner werden die Namen der Zustände und die vom Datenraum unabhängigen Teile der Transitionsmarkierungen (Label) aus dem konkreten Modell übernommen. Datenunabhängige Bestandteile eines Labels sind die Expression und die im Aktionsteil angegebenen Ereignisse.

2. Wir abstrahieren datenabhängige Bestandteile der Label. Hierzu wird für jede Transition jeweils eine Überapproximation des Guards und eine Überapproximation der Update-Funktion gebildet.

3. Wir führen selbstbezügliche Transitionen ein, um implizites Verhalten des konkreten Modells im abstrakten Modell explizit zu repräsentieren.

Im Folgenden motivieren wir diese Vorgehensweise und begründen, warum es notwendig ist, implizites Verhalten des konkreten Modells im abstrakten Modell explizit zu machen.

Um eine Überapproximation des zu abstrahierenden Sequentiellen Automaten zu erhalten, darf im abstrakten Modell nur neues Verhalten hinzugefügt, aber kein bestehendes Verhalten entfernt werden. Dies müssen wir beachten, wenn wir die Guards und die Update-Funktionen der Transitionen abstrahieren. Hinzu kommt, dass es im Allgemeinen unmöglich ist, ein Prädikat über dem konkreten Datenraum durch ein Prädikat über dem abstrakten Datenraum exakt zu beschreiben. In Abschnitt 6.1 sind mit Definition 6.4 zwei Galois-Korrespondenzen angegeben, mit denen Prädikate verschärft oder abgeschwächt werden können. Die Abschwächung

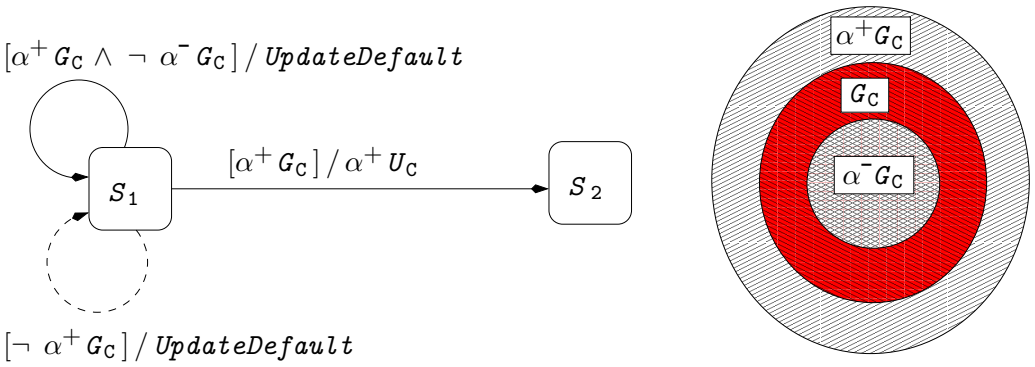


Abbildung 6.5: Überapproximation eines Sequentiellen Automaten unter Verwendung der Funktionen α^+ und α^-

eines Prädikats wird durch die Funktion α^+ und die Verschärfung mit α^- angegeben. Auf der rechten Seite in Abbildung 6.5 ist die Abstraktion eines Guards G_C unter Verwendung der beiden Funktionen graphisch durch ein Kreisdiagramm veranschaulicht.

Überapproximation von Guards

Wir schlagen zunächst vor, mit α^+ abstrakte Guards aus den konkreten Guards des zu abstrahierenden Sequentiellen Automaten zu konstruieren. Die Abschwächung der Guards führt hierbei auf den ersten Blick zu einer Überapproximation des Verhaltens, da die abstrakten Transitionen öfter als ihre korrespondierenden Gegenstücke aus dem konkreten Modell aktiviert werden können. Allerdings führt die Abschwächung eines Guards gleichzeitig zu einer Reduktion von implizitem Verhalten. Der Grund dafür ist, dass die Guards der impliziten Transitionen bei diesem Vorgehen automatisch verschärft werden können. Dieser unerwünschte Effekt entsteht dadurch, dass die Guards der impliziten Transitionen aus der Negation der abgeschwächten Guards konstruiert sind.

Zur Lösung des beschriebenen Problems schlagen wir vor, eine selbstbezügliche Transition am Startzustand der zu abstrahierenden Transition einzufügen, um implizites Verhalten aus dem konkreten Modell explizit zu repräsentieren.

Zur besseren Veranschaulichung werden wir dieses Vorgehen für den Sequentiellen Automaten aus Abbildung 6.4 beschreiben. Hierzu geben wir an, wie man schrittweise zu der auf der linken Seite in Abbildung 6.5 dargestellten Überapproximation gelangt: Zunächst werden die strukturellen Merkmale des konkreten Sequentiellen Automaten übernommen, d.h. auch der abstrakte Automat besteht aus den beiden Zuständen S_1 und S_2 und einer Transition zwischen diesen. Danach werden die datenabhängigen Bestandteile des Labels durch ihre abstrakten Gegenstücke ersetzt. Hierzu werden der Guard G_C und die Update-Funktion U_C mit α^+ abstrahiert. Wie abstrakte Update-Funktionen mit Hilfe von α^+ konstruiert werden können, erklären wir in Abschnitt 6.2.3. Schließlich wird durch das Einfügen einer selbstbezüglichen Transition im Zustand S_1 sichergestellt, dass kein implizites Verhalten aus dem konkreten System verloren geht. Dies ist notwendig, da die Negation des abgeschwächten Guards $\alpha^+ G_C$ (vgl. die gestrichelte Transition in Abbildung 6.5) eine Verschärfung gegenüber dem ursprünglichen Guard der impliziten Transition $\neg\ G_C$ (vgl. die gestrichelte Transition in Abbildung 6.4) darstellt. Um diesen Effekt auszugleichen, wird der Guard der neu eingefügten

Transition als Konjunktion aus der Überapproximation von G_C und der negierten Unterapproximation von G_C konstruiert. Das dabei entstehende Prädikat ist in der Graphik auf der rechten Seite in Abbildung 6.5 durch den äußeren und mittleren Ring des Kreisdiagramms repräsentiert.

Betrachten wir noch einmal das einführende Beispiel aus Abbildung 6.3. Die dort im abstrakten Sequentiellen Automaten am Zustand S_1 eingefügte selbstbezügliche Transition beschreibt implizites Verhalten. Der Guard $B_1 \wedge B_2$ an dieser Transition ergibt sich aus der Vereinfachung des folgenden Ausdrucks.

$$\alpha^+(x > 4) \wedge \neg\, \alpha^-(x > 4)$$

Sowohl in dem einführenden Beispiel als auch in dem Sequentiellen Automaten aus Abbildung 6.4 existiert nur eine aus dem Zustand S_1 herausführende Transition. Für n Transitionen würde der Guard der selbstbezüglichen Transition entsprechend folgendermaßen konstruiert.

$$[\alpha^+ G_C^1 \wedge \neg\, \alpha^- G_C^1 \wedge \alpha^+ G_C^2 \wedge \neg\, \alpha^- G_C^2 \wedge \ \ldots\ \wedge \alpha^+ G_C^n \wedge \neg\, \alpha^- G_C^n]$$

Ferner sind wir bei dem vorgestellten Konzept davon ausgegangen, dass die Aktivierung einer Transition lediglich von der Erfüllbarkeit ihres Guards abhängt. Sollten in den Labeln der zu abstrahierenden Transitionen auch Expressions vorkommen, so sind diese beim Einfügen von impliziten Transitionen geeignet zu berücksichtigen. In Kapitel 7 ist eine Theorie angegeben, in der Konstruktionsoperatoren zur Überapproximation Sequentieller Automaten definiert sind. Bei der Formalisierung dieser Operatoren haben wir die Behandlung von Expressions berücksichtigt.

Überapproximation von Update-Funktionen

In Analogie zur Überapproximation eines Guards schlagen wir vor, die abstrakten Update-Funktionen aus den konkreten Update-Funktionen mit Hilfe von α^+ zu konstruieren. Dazu werden zunächst die Update-Funktionen als binäre Prädikate über ihrem Vor- und Nachzustand interpretiert und anschließend mit α^+ abstrahiert. Das bei der Abstraktion entstehende binäre Prädikat repräsentiert im Allgemeinen mehr als eine abstrakte Update-Funktion. So wurde zum Beispiel in dem einführenden Beispiel aus Abbildung 6.3 die konkrete Update-Funktion $x := x + 1$ durch die zwei abstrakten Update-Funktionen $B_1 \wedge B_2$ und $B_1 \wedge \neg\, B_2$ repräsentiert. Um beide Update-Funktionen im abstrakten Sequentiellen Automaten darstellen zu können, ist die konkrete Transition durch zwei abstrakte Transitionen ausgedrückt worden. Angenommen die Update-Funktion einer konkreten Transition läßt sich in der Überapproximation nur durch n verschiedene abstrakte Update-Funktionen darstellen, so sind für die konkrete Transition im abstrakten Modell n Transitionen einzuführen, wobei jede der Transitionen mit einer der konstruierten abstrakten Update-Funktionen zu markieren ist.

Man beachte, dass die Überapproximation von Update-Funktionen im Allgemeinen den Nachzustand abschwächt und damit neues Verhalten hinzufügt. Im Gegensatz zur Überapproximation der Guards wird aber kein implizites Verhalten reduziert. Ferner sind wir bei dem vorgestellten Konzept davon ausgegangen, dass die Update-Funktionen den partitionierten Datenraum total beschreiben. In Abschnitt 6.3 zur Datenabstraktion für Hierarchische Automaten verfeinern wir diesen Ansatz um die Überapproximation von partiellen Update-Funktionen.

6.2.4 Unterapproximation von Sequentiellen Automaten

Das von uns vorgestellte Konzept zur Abstraktion von Sequentiellen Automaten beruht darauf, einen möglichst strukturell ähnlichen Sequentiellen Automaten als abstraktes Modell zu erzeugen. Wir haben uns dieses Ziel gesetzt, um eine hohe Transparenz im gesamten Abstraktionsprozess garantieren zu können. Dies ist ohne Zweifel ein Vorteil unserer Arbeit gegenüber anderen Arbeiten aus der Literatur. Allerdings birgt dieser Ansatz den Nachteil, dass wir unsere Abstraktionstheorie auf Überapproximationen einschränken müssen. Dies ist in der speziellen Semantik von synchronen Sprachen begründet (vgl. Abschnitt 6.2.2). Die Vervollständigung von nicht definiertem Verhalten durch so genanntes implizites Verhalten schränkt die Möglichkeit zur Reduktion von Verhalten substantiell ein. Betrachten wir nochmals das Beispiel aus Abbildung 6.4. Würden wir vorschlagen, den Guard g_c durch α^- zu verschärfen, so würde der Guard an der impliziten Transition gleichzeitig abgeschwächt. Damit würde im abstrakten Modell neues Verhalten auftreten und zu einer inkorrekten Unterapproximation führen. Folglich ist es unmöglich für das Beispiel aus Abbildung 6.4 eine fehlerfreie Unterapproximation zu bilden, in der das abstrakte Modell ein strukturell ähnlicher Sequentieller Automat ist. Die einzige Möglichkeit, Verhalten zu reduzieren, besteht darin, nichtdeterministische Verzweigungen eines Sequentiellen Automaten auf deterministische einzuschränken. Dieses Vorgehen ist aber für die Definition eines allgemeinen Abstraktionskonzepts nicht ausreichend. Da es uns nicht möglich ist, das Ergebnis der Unterapproximation eines Sequentiellen Automaten wieder als Sequentiellen Automaten auszudrücken, ist unser Abstraktionsansatz auf die Erhaltung des universalen Fragments der CTL ($\forall$ CTL) beschränkt.

6.3 Datenabstraktion für Hierarchische Automaten

In diesem Abschnitt stellen wir eine Abstraktionstechnik zur Überapproximation Hierarchischer Automaten vor, die in kompositionaler Art und Weise erfolgt. Dies bedeutet, dass ein Hierarchischer Automat zunächst in die ihn definierenden Sequentiellen Automaten zerlegt wird. Anschließend werden diese separat voneinander abstrahiert und die entstehenden Überapproximationen wieder zu einem abstrakten Hierarchischen Automaten zusammengefügt. Dass dieses Vorgehen unter bestimmten Randbedingungen zu einer korrekten Überapproximation des Hierarchischen Automaten führt, wird in diesem Abschnitt begründet. Zunächst gehen wir in Analogie zum letzten Abschnitt auf das implizite Verhalten Hierarchischer Automaten ein. Danach beschreiben wir die Besonderheiten beim konkurrierenden und partiellen Beschreiben von Datenpartitionen durch Update-Funktionen. Schließlich erläutern wir, wie die Überapproximation von partiellen Update-Funktionen zu konstruieren sind und welche Eigenschaften von der Abstraktionsfunktion für einen korrekten Abstraktionsprozess erfüllt werden müssen.

6.3.1 Implizites Verhalten in Hierarchischen Automaten

Implizites Verhalten tritt immer dann auf, wenn zu Beginn eines semantischen Zustandsübergangs keine Transitionen aktiviert sind. In dieser Situation wird ein trivialer Berechnungsschritt ausgeführt, in dem die Datenvariablen mit ihrem vorherigen Wert erneut beschrieben werden. Im letzten Abschnitt wurde dieses implizite Verhalten durch gestrichelte Transitionen modelliert, die immer nur dann aktiviert werden, wenn keine anderen Transitionen in einem Sequentiellen Automaten aktiviert sind.

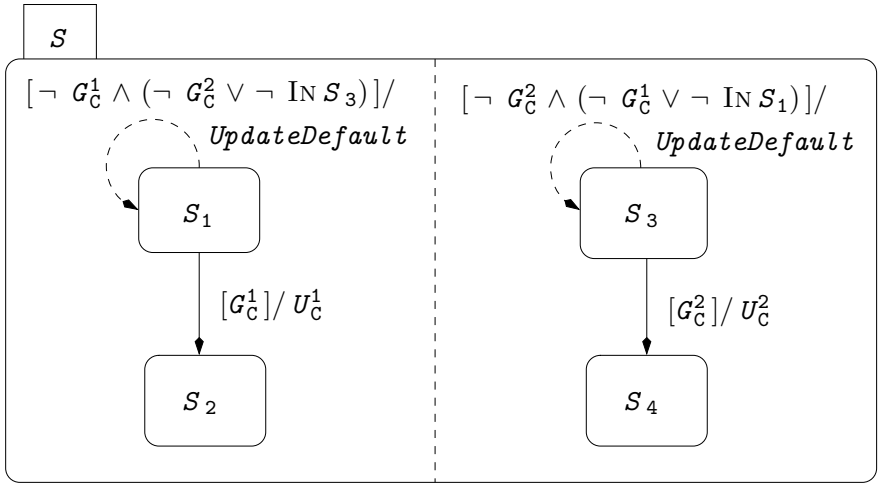


Abbildung 6.6: Implizites Verhalten in einem Hierarchischen Automaten

In Analogie zur Abbildung 6.4 zeigt Abbildung 6.6 implizites Verhalten in einem Hierarchischen Automaten. Hierbei ist zu beachten, dass der Guard einer selbstbezüglichen Transition, die implizites Verhalten beschreibt, von Kontextinformationen abhängt. Diese Kontextinformationen liegen außerhalb des Sequentiellen Automaten, in dem die selbstbezügliche Transition definiert ist.

Betrachten wir den Sequentiellen Automaten auf der linken Seite in Abbildung 6.6. Hier wird der Guard der selbstbezüglichen Transition am Zustand S_1 nur dann erfüllt, wenn der Guard der aus S_1 herausführenden Transition G^1_C nicht erfüllt ist. Zusätzlich wird ein Prädikat gefordert, mit dem sichergestellt wird, dass im parallel komponierten Sequentiellen Automaten auch keine Transition schalten kann.

Es wird deutlich, dass bei der Definition von implizitem Verhalten im Allgemeinen starke Abhängigkeiten zwischen den einzelnen Sequentiellen Automaten eines Hierarchischen Automaten bestehen. Dies steht aber der Definition einer Abstraktionstechnik entgegen, die die Abstraktion in kompositionaler Art und Weise umsetzt. In Abschnitt 6.3.3 schlagen wir eine Variante zur Überapproximation eines Sequentiellen Automaten vor, in der keine Kontextinformationen von anderen parallel komponierten Sequentiellen Automaten benötigt werden. Lediglich lokal verfügbare Informationen werden ausreichen, um die Überapproximation zu konstruieren.

6.3.2 Partitionen auf dem Datenraum und partielle Update-Funktionen

Der Datenraum eines Hierarchischen Automaten besteht im Allgemeinen aus einer endlichen Anzahl untereinander disjunkter Partitionen. Diese Partitionen sind global sichtbar und können durch die Update-Funktionen von synchron auszuführenden Transitionen konkurrierend geschrieben werden. Hierbei können Schreibkonflikte auftreten, die durch eine Interleaving-Semantik in nichtdeterministischen Verhalten aufgelöst werden. Eine präzise mathematische Beschreibung partitionierter Datenräume und darauf schreibender Update-Funktionen ist in Kapitel 4 vorgestellt. Dort ist unter anderem beschrieben, dass Update-Funktionen so definiert werden können, dass sie nicht auf alle Partitionen des Datenraumes

schreiben. Sie werden deshalb in dieser Arbeit auch als *partielle Update-Funktionen* (vgl. Definition 4.10) bezeichnet. Die Semantik Hierarchischer Automaten legt für einen semantischen Zustandsübergang fest, welcher Datenwert einer unbeschriebenen Datenpartition zugewiesen wird. Beschreibt die partielle Update-Funktion einer Transition eine Datenpartition nicht, so wird zunächst geprüft, ob eine andere synchron auszuführende Transition existiert, die diese Datenpartition mit einem Datenwert beschreibt. Ist dies der Fall, so wird der Datenpartition der Datenwert dieser synchron auszuführenden Transition zugewiesen. Sollte allerdings keine der synchron auszuführenden Transitionen die Datenpartition beschreiben, so wird die zuvor gültige Datenbelegung erneut zugewiesen.

Ähnlich zu der in Abschnitt 6.3.1 beschriebenen Situation wird also eine Vervollständigung des explizit modellierten Verhaltens durch implizites Verhalten vorgenommen. Ob implizites Verhalten für eine von einer Update-Funktion unbeschriebene Datenpartition auftritt, ist im Allgemeinen lokal in einem Sequentiellen Automaten nicht zu entscheiden. Folglich ist es schwierig, die Überapproximation einer partiellen Update-Funktion unabhängig von anderen synchron schreibenden Update-Funktionen anzugeben. In Abschnitt 6.3.3 definieren wir spezielle Anforderungen an die Abstraktionsfunktion, die garantieren, dass die Überapproximation einer partiellen Update-Funktion mit lokal verfügbaren Informationen konstruierbar ist.

6.3.3 Überapproximation von Hierarchischen Automaten

Um die Überapproximation eines Hierarchischen Automaten zu bilden, konstruieren wir in kompositionaler Art und Weise einen neuen abstrakten Hierarchischen Automaten. Unter Überapproximation eines Hierarchischen Automaten verstehen wir die Abstraktion des zugrunde liegenden Datenraums. Die Struktur des Automaten wird also weitestgehend erhalten, und datenabhängige Bestandteile werden durch ihre abstrakten Gegenstücke ersetzt. Der Konstruktionsprozess gliedert sich in die folgenden Schritte.

1. Zerlegen des Hierarchischen Automaten in seine ihn definierenden Sequentiellen Automaten.

2. Bilden von Überapproximationen aller Sequentiellen Automaten aus Schritt 1 analog zum Vorgehen in Abschnitt 6.2.3. Ausgenommen hiervon ist die dort beschriebene Behandlung von Update-Funktionen.

3. Einfügen *partieller Default-Update-Funktionen* (vgl. Definition 4.12) an den selbstbezüglichen Transitionen, die im abstrakten Modell zur Beschreibung von implizitem Verhalten dienen.

4. Konstruktion von Überapproximationen für alle konkreten partiellen Update-Funktionen, indem die Updates für alle Datenpartitionen separat voneinander abstrahiert werden. Dieser Schritt ist nur durchführbar, wenn die Abstraktionsfunktion die Struktur des Datenraums erhält.

5. Zusammenfügen aller abstrahierten Sequentiellen Automaten – in umgekehrter Reihenfolge ihrer Zerlegung – zu einem abstrakten Hierarchischen Automaten.

Im Folgenden begründen wir, warum das angegebene Vorgehen zu einer korrekten Überapproximation Hierarchischer Automaten führt.

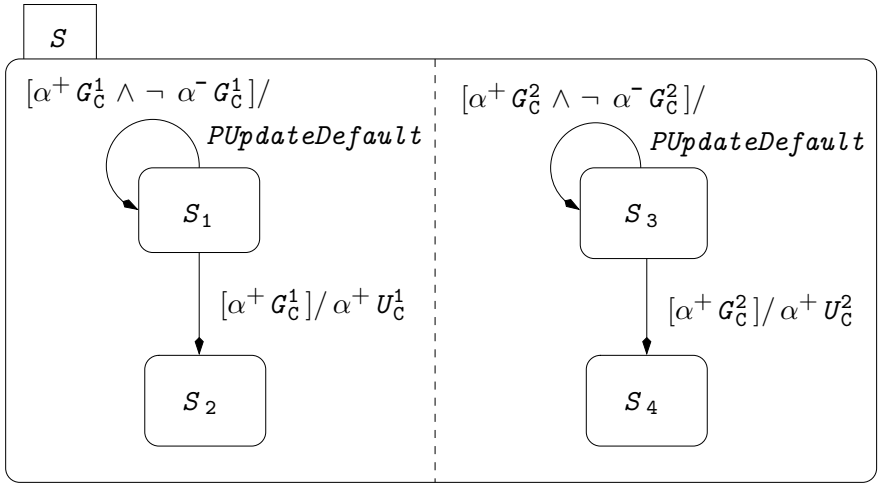


Abbildung 6.7: Überapproximation mit Hilfe von partiellen Default-Update-Funktionen

Implizites Verhalten durch partielle Default-Update-Funktionen

Zunächst wollen wir auf die Korrektheit der Modellierung von implizitem Verhalten eingehen. Wenden wir das beschriebene Vorgehen zur Abstraktion auf den Hierarchischen Automaten aus Abbildung 6.6 an, so erhalten wir die in Abbildung 6.7 dargestellte Überapproximation. Man beachte, dass die Guards der neu eingefügten selbstbezüglichen Transitionen für einen Sequentiellen Automaten lokal konstruiert sind. Kontextinformationen von parallel komponierten Sequentiellen Automaten sind – anders als bei der Definition von implizitem Verhalten in Abbildung 6.6 – bei der Konstruktion nicht berücksichtigt worden und können damit zu einer Abschwächung der Guards führen. Folglich sind die selbstbezüglichen Transitionen an den Zuständen S_1 und S_2 häufiger aktiviert, als es für die Erhaltung von implizitem Verhalten notwendig ist.

Es könnte der Eindruck entstehen, dass durch dieses Vorgehen mehr implizites Verhalten als notwendig modelliert und damit nicht die bestmögliche Überapproximation konstruiert wird. Dieser Eindruck täuscht jedoch. Die besondere Semantik partieller Default-Update-Funktionen ermöglicht es, dass das häufigere Ausführen der Transitionen auf dem Datenraum nicht sichtbar wird. Der Grund dafür ist, dass eine partielle Default-Update-Funktion beim Schreiben des Datenraumes die geringste Priorität von allen anderen partiellen Update-Funktionen besitzt. Eine partielle Default-Update-Funktion hat also nur dann einen in der Semantik nachweisbaren Effekt auf dem Datenraum, wenn keine andere synchron ausgeführte Transition den Datenraum beschreibt.

Überapproximation von partiellen Update-Funktionen

Um das in Abschnitt 6.3.2 angesprochene Problem zur Behandlung von partitionierten Datenräumen zu lösen, beschreiben wir eine Technik, mit der eine möglichst präzise Überapproximation von partiellen Update-Funktion konstruiert werden kann. Eine wichtige Voraussetzung hierfür ist, dass die Abstraktionsfunktion die Partitionen des Datenraumes erhält. Um dies sicherzustellen, definieren wir spezielle Eigenschaften für die zu verwendende Abstraktionsfunktion und fordern, dass diese für die Überapproximation eines Hierarchischen

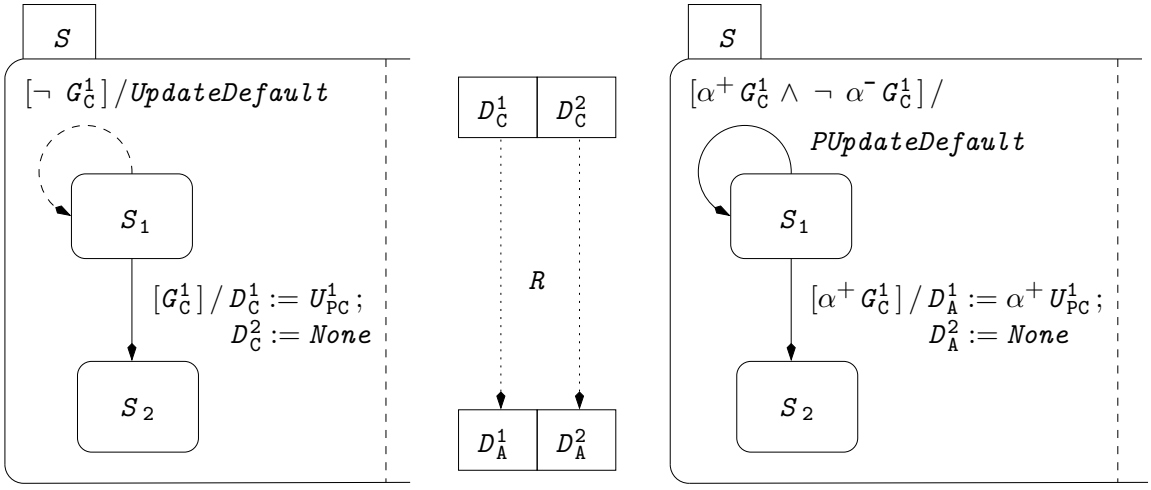


Abbildung 6.8: Strukturerhaltene Überapproximation partieller Update-Funktionen

Automaten explizit nachgewiesen werden.

Angenommen das Beispiel aus Abbildung 6.6 ist auf einem aus zwei Partitionen bestehenden Datenraum definiert. Die Partitionen seien jeweils durch eine Integer-Variable repräsentiert und mit D^1_C und D^2_C bezeichnet. Wir nehmen weiterhin an, dass die Update-Funktion U^1_C den Datenraum nur partiell beschreibt. Konkret wird die erste Datenpartition mit der Funktion U^1_{PC} und die zweite Partition gar nicht beschrieben. Die Konkretisierungen des Beispiels sind zur Veranschaulichung auf der linken Seite in Abbildung 6.8 angegeben. In der graphischen Repräsentation eines Hierarchischen Automaten ist es unüblich, eine unbeschriebene Datenpartition darzustellen. Zur besseren Veranschaulichung weichen wir von dieser Konvention ab und weisen der unbeschriebenen Datenpartition D^2_C die Konstante *None* zu.

Um eine präzise Überapproximation der partiellen Update-Funktion U^1_C lokal in einem Sequentiellen Automaten konstruieren zu können, fordern wir, dass die Abstraktionsfunktion die binäre Struktur des Datenraumes erhält, d.h. die beiden konkreten Datenpartitionen D^1_C und D^2_C werden im abstrakten Datenraum durch zwei korrespondierende Datenpartitionen D^1_A und D^2_A repräsentiert. Hierbei ist zu beachten, dass die abstrakten Datenpartitionen entweder D^1_C oder D^2_C, aber nicht beide konkreten Datenpartitionen gleichzeitig interpretieren dürfen. Wir haben diese Eigenschaft für eine Abstraktionsfunktion R in der Mitte von Abbildung 6.8 graphisch veranschaulicht.

In einer Prädikatenabstraktion sind D^1_A und D^2_A durch boolesche Variablen charakterisiert, wobei eine abstrakte Datenpartition im Allgemeinen aus mehreren booleschen Variablen zusammengesetzt sein kann. Jede dieser booleschen Variablen repräsentiert die Gültigkeit eines Prädikats über der zu repräsentierenden konkreten Datenpartition. Folgende Abstraktionsfunktion für eine Prädikatenabstraktion verletzt offensichtlich die von uns geforderte Eigenschaft für Abstraktionsfunktionen, da die erste abstrakte Datenpartition von D^1_C und D^2_C abhängt.

$$R\,[D^1_C, D^2_C] \equiv_{df} [D^1_C \leq D^2_C, D^2_C \leq 2]$$

Würden wir R trotzdem verwenden, um eine Überapproximation für U^1_C lokal zu berechnen, so wäre diese Überapproximation sehr ungenau. Der Grund dafür ist, dass die Abstraktions-

funktion eine abstrakte Interpretation unabhängig von D_C^2 nicht erlaubt. Lokal ist aber keine Information über die ungeschriebene Datenpartition D_C^2 vorhanden, so dass das Prädikat $D_C^1 \leq D_C^2$ erfüllt sein kann oder nicht. Es ist also möglich, auch für Abstraktionsfunktionen, die die Struktur des Datenraumes nicht erhalten, korrekte Überapproximationen zu konstruieren. Allerdings sind diese Überapproximationen im Allgemeinen sehr ungenau und deshalb für praktische Beispiele nicht sinnvoll einsetzbar.

Hält man sich hingegen an die Vorgabe und definiert eine Abstraktionsfunktion so, dass die Struktur des Datenraumes erhalten bleibt, so kann eine präzise Überapproximation konstruiert werden. Hierzu wird die zu abstrahierende Update-Funktion in Teilfunktionen zerlegt. Jede Teilfunktion beschreibt den Effekt der Update-Funktion für genau eine Datenpartition. Da die Abstraktionsfunktion jede konkrete Datenpartition durch genau eine abstrakte Datenpartition repräsentiert, können die Teilfunktionen getrennt voneinander abstrahiert werden. Unbeschriebene Datenpartitionen sind entsprechend im abstrakten Modell auch als unbeschriebene Datenpartitionen zu repräsentieren.

Die rechte Seite in Abbildung 6.8 deutet an, wie für unser Beispiel die Überapproximation von U_C^1 konstruiert werden kann. Für die Darstellung des konkreten Modells auf der linken Seite in Abbildung 6.8 ist U_C^1 bereits in zwei Teilfunktionen zerlegt worden. Die eine Teilfunktion ist mit U_{PC}^1 bezeichnet und beschreibt den Effekt von U_C^1 auf der ersten Datenpartition. Die andere Teilfunktion ist durch die Zuweisung der Konstante *None* definiert. Diese Zuweisung repräsentiert, dass die Datenpartition D_C^2 von U_C^1 nicht beschrieben wird. Hat man eine Zerlegung in Teilfunktionen vorgenommen, werden diese anschließend unabhängig voneinander abstrahiert. Hierzu wenden wir analog zum Vorgehen in Abschnitt 6.2.3 die Funktion α^+ auf U_{PC}^1 an und weisen das Ergebnis der ersten abstrakten Datenpartition D_A^1 zu. Die zweite abstrakte Datenpartition D_A^2 wird nicht beschrieben, was analog zu der Darstellung im konkreten Modell durch die Zuweisung der Konstante *None* umgesetzt ist.

6.4 Zusammenfassung

Wir haben in diesem Kapitel ein Konzept vorgestellt, mit dem Datenräume Hierarchischer Automaten abstrahiert werden können. Die Ergebnisse dieses Kapitels wurden von uns auf einer internationalen Konferenz veröffentlicht [HK05].

Im ersten Abschnitt dieses Kapitels legten wir mit einer Formalisierung von Galois-Korrespondenzen eine wichtige Grundlage für unsere Abstraktionstheorie. Wir sind dadurch in der Lage, Prädikate durch eine Abstraktion gezielt zu verschärfen oder abzuschwächen. Auf Basis von Galois-Korrespondenzen stellten wir im nächsten Abschnitt ein Konzept zur Datenabstraktion Sequentieller Automaten vor. Wir konstruierten für eine gegebene Abstraktionsfunktion die Überapproximationen eines Sequentiellen Automaten, indem wir die datenabhängigen Bestandteile durch abstrakte Gegenstücke ersetzten und implizites Verhalten durch selbstbezügliche Transitionen repräsentierten.

Im dritten Abschnitt dieses Kapitels stellten wir schließlich eine Abstraktionstechnik vor, mit der für eine Abstraktionsfunktion die Überapproximation eines Hierarchischen Automaten gebildet werden kann. Die Abstraktion des Hierarchischen Automaten erfolgt in kompositionaler Art und Weise. Hierzu haben wir den zu abstrahierenden Hierarchischen Automaten in seine ihn definierenden Sequentiellen Automaten zerlegt und diese separat voneinander abstrahiert. Anschließend haben wir die Überapproximationen aller Sequentiellen Automaten wieder zu einem Hierarchischen Automaten zusammengesetzt. Um dieses Vorgehen geeignet

umzusetzen, haben wir spezielle Eigenschaften an die Abstraktionsfunktion formuliert.

Das Ergebnis der Abstraktion ist also durch einen Hierarchischen Automaten repräsentiert, der eine Struktur aufweist, die ähnlich zum ursprünglichen Automaten ist. Dieses besondere Merkmal unterscheidet unseren Ansatz von vielen anderen in der Literatur angegebenen Abstraktionsverfahren. Wir sind dadurch in der Lage, dem Benutzer unserer Abstraktionstechnik eine für ihn verständliche Beschreibung des abstrakten Modells anzubieten. Ferner ist der dem Ergebnis zugrunde liegende Datenraum endlich, so dass eine automatische Verifikation durch Model-Checking auf dem abstrakten Modell möglich ist. Schließlich haben wir in diesem Kapitel erläutert, warum die von uns entwickelte Datenabstraktiontechnik eigenschaftserhaltend bezüglich des universalen Fragments der CTL ist.

Im folgenden Kapitel setzen wir das vorgestellte Konzept zur Datenabstraktion Hierarchischer Automaten in einer Isabelle-Formalisierung um. Hierfür definieren wir einen geeigneten Abstraktionsoperator, der für eine Abstraktionsfunktion aus einem Hierarchischen Automaten einen abstrakten Hierarchischen Automaten konstruiert.

Kapitel 7

Umsetzung der Abstraktionstheorie in Isabelle/HOL

In diesem Kapitel setzen wir das in Kapitel 6 vorgestellte Konzept zur Überapproximation Hierarchischer Automaten in einer Formalisierung für Isabelle/HOL um. Diese neue Theorie erweitert die Formalisierung Hierarchischer Automaten aus Teil I um Konstruktionsoperatoren zur Abstraktion. Die Konstruktionsoperatoren erlauben es, aus einem gegebenen Hierarchischen Automaten und einer gegebenen Abstraktionsfunktion eine Überapproximation des Hierarchischen Automaten zu bilden. Das Kapitel gliedert sich in vier Abschnitte.

Zunächst formalisieren wir einen Datentyp für Abstraktionsfunktionen, der die in Abschnitt 6.3.3 angesprochenen Eigenschaften garantiert. In Analogie zur Einführung des Abstraktionskonzepts aus Kapitel 6 geben wir in Abschnitt 7.2 Konstruktionsoperatoren an, mit denen die Überapproximation eines Sequentiellen Automaten gebildet werden kann. Diese werden wir in Abschnitt 7.3 dazu verwenden, Konstruktionsoperatoren zur Abstraktion für Hierarchische Automaten in kompositionaler Art und Weise zu definieren. Schließlich stellen wir im letzten Abschnitt dieses Kapitels einen Operator vor, mit dem die Unterapproximation einer temporalen Formel des universalen Fragments der CTL gebildet werden kann.

7.1 Eigenschaften der Abstraktionsfunktion

In diesem Abschnitt sind Eigenschaften für die zu verwendende Abstraktionsfunktion angegeben. Wenn diese Eigenschaften garantiert werden, ist es uns möglich, eine präzise Überapproximation für einen Hierarchischen Automaten zu konstruieren.

In Abbildung 7.1 ist der Aufbau einer Abstraktionsfunktion graphisch veranschaulicht. Sie zeigt die Strukturerhaltung eines aus n Partitionen bestehenden Datenraumes. Die oberen Felder in der Abbildung symbolisieren die Partitionen des konkreten Datenraumes $D_{\mathtt{C}}^{1}$ bis $D_{\mathtt{C}}^{\mathtt{n}}$. Analog dazu symbolisieren die unteren Felder die Partitionen des abstrakten Datenraumes $D_{\mathtt{A}}^{1}$ bis $D_{\mathtt{A}}^{\mathtt{n}}$. Wir fordern, dass durch eine Abstraktionsfunktion R jede konkrete Datenpartition unabhängig von anderen Datenpartitionen auf eine korrespondierende abstrakte Datenpartition abgebildet wird. Dies bedeutet, dass R sich in die Teilfunktionen R^{1} bis $R^{\mathtt{n}}$ zerlegen läßt.

Basierend auf der Formalisierung von partitionierten Datenräumen aus Abschnitt 4 lassen sich die Charakteristika von Abstraktionsfunktionen durch einen Typ beschreiben, der in Definition 7.1 angegeben ist.

Das Theorem AbsNonEmpty belegt, dass die in der Definition abstrahierte Menge nicht leer ist.

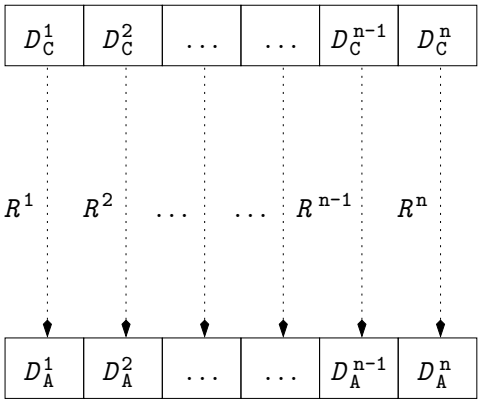


Abbildung 7.1: Zerlegung einer strukturerhaltenden Abstraktionsfunktion

Definition 7.1 (Abstraktionsfunktion) *Sei δ_C ein Typ für den konkreten Datenraum und δ_A ein Typ für den abstrakten Datenraum, dann ist eine Abstraktionsfunktion R über (δ_C, δ_A) durch ein Tripel (L, D_C, D_A) repräsentiert. Hierbei ist*

- *L die Liste von Teilfunktionen zur Abstraktion einzelner Datenpartitionen,*
- *D_C der konkrete Datenraum und*
- *D_A der abstrakte Datenraum*

der Abstraktionsfunktion R. Diese Komponenten müssen die interne Konsistenzbedingung $AbsCorrect$ über Abstraktionsfunktionen erfüllen, die in Definition 7.2 angegeben ist. Der Typ (δ_C, δ_A) abs besteht aus allen Abstraktionsfunktionen über (δ_C, δ_A).

$$
\begin{aligned}
(\delta_C, \delta_A)\ abs \equiv_t \{\, & (L, D_C, D_A) \mid \\
& (L :: (\delta_C \to \delta_A)\ list) \\
& (D_C :: \delta_C\ dataspace) \\
& (D_A :: \delta_A\ dataspace). \\
& AbsCorrect\ L\ D_C\ D_A\}
\end{aligned}
$$

gerechtfertigt durch AbsNonEmpty

□

Um auf die Komponenten einer Abstraktionsfunktion zugreifen zu können, führen wir folgende Selektionsoperatoren ein. Die Liste an Teilfunktionen zur Abstraktion einzelner Datenpartitionen kann mit dem Operator *AbsFuns*, der konkrete Datenraum mit dem Operator *CDataSpace* und der abstrakte Datenraum mit dem Operator *ADataSpace* für eine gegebene Abstraktionsfunktion bestimmt werden.

Definition 7.2 (Wohlgeformte Abstraktionsfunktion) *Sei L eine Liste von Teilfunktionen zur Abstraktion von einzelnen Datenpartitionen, D_C ein konkreter Datenraum und D_A ein*

abstrakter Datenraum, dann bilden diese Komponenten eine wohlgeformte Abstraktionsfunktion, wenn folgende Bedingungen erfüllt sind.

$$\begin{aligned} &\mathit{AbsCorrect} ::_{\mathsf{c}} [(\delta_{\mathsf{c}} \rightarrow \delta_{\mathsf{A}})\ \mathit{list},\ \delta_{\mathsf{c}}\ \mathit{dataspace},\ \delta_{\mathsf{A}}\ \mathit{dataspace}] \rightarrow \mathit{bool} \\ &\mathit{AbsCorrect}\ L\ D_{\mathsf{C}}\ D_{\mathsf{A}} \equiv_{\mathsf{c}} \ \#\ L = \mathit{PartNum}\ D_{\mathsf{C}} \wedge \mathit{PartNum}\ D_{\mathsf{C}} = \mathit{PartNum}\ D_{\mathsf{A}} \wedge \\ &\qquad\qquad \forall\ i : \{\, n.\ n < \mathit{PartNum}\ D_{\mathsf{C}} \,\}.\ \mathit{ran}\,(L!\,i) \subseteq \mathit{PartDom}\ D_{\mathsf{A}}\ i \end{aligned}$$

□

Definition 7.2 garantiert für eine wohlgeformte Abstraktionsfunktion R, dass sowohl die Anzahl der in R definierten Teilfunktionen als auch die Anzahl der in R abstrakten Datenpartitionen zu der Anzahl der konkreten Datenpartitionen identisch ist. Ferner wird durch die Definition sichergestellt, dass der Bildbereich der i-ten Teilfunktion aus R eine Untermenge des Definitionsbereichs der i-ten Partition des abstrakten Datenraumes ist.

Abstraktionsfunktionen sollen in unserer Formalisierung dazu eingesetzt werden, totale und partielle Datenraumbelegungen (vgl. die Definitionen 4.3 und 4.5 aus Kapitel 4) zu abstrahieren. Hierfür führen wir zwei Operatoren ein. Für eine gegebene Abstraktionsfunktion R kann mit dem Operator $\mathit{AbsBy}^{+}_{\mathsf{D}}$ eine totale Datenraumbelegung und mit dem Operator $\mathit{AbsBy}^{+}_{\mathsf{PD}}$ eine partielle Datenraumbelegung abstrahiert werden. Mit diesen Operatoren wird sichergestellt, dass die i-te Teilfunktionen aus R auf die i-te Datenraumbelegung des konkreten Modells appliziert wird. Ferner wird durch diese Operatoren garantiert, dass das Ergebnis der Applikation an der i-ten Datenraumbelegung des abstrakten Modells eingetragen wird. Der Operator $\mathit{AbsBy}^{+}_{\mathsf{PD}}$ stellt zusätzlich sicher, dass unbeschriebene Datenpartitionen des konkreten Modells als unbeschriebene Datenpartitionen im abstrakten Modell repräsentiert werden. Weiterhin kann mit dem Operator range der Bildbereich einer Abstraktionsfunktion ermittelt werden. Unter dem Bildbereich einer Abstraktionsfunktion verstehen wir alle mit dem Operator $\mathit{AbsBy}^{+}_{\mathsf{D}}$ erreichbaren Datenraumbelegungen. Die genauen Definitionen der Operatoren sind in Anhang A.5 dargestellt.

7.2 Konstruktion von überapproximierten Sequentiellen Automaten

Um die Überapproximation eines Sequentiellen Automaten zu berechnen, definieren wir in Definition 7.3 den Operator $\mathit{AbsBy}^{+}_{\mathsf{SA}}$.

Definition 7.3 (Konstruktion der Überapproximation eines SA) *Sei SA ein Sequentieller Automat und R eine Abstraktionsfunktion, dann ist die Überapproximation von SA für R folgendermaßen definiert.*

$$\begin{aligned} &\_\ \mathit{AbsBy}^{+}_{\mathsf{SA}}\ \_ ::_{\mathsf{c}} [(\sigma, \epsilon, \delta_{\mathsf{c}})\ \mathit{seqauto}, (\delta_{\mathsf{c}}, \delta_{\mathsf{A}})\ \mathit{abs}] \rightarrow (\sigma, \epsilon, \delta_{\mathsf{A}})\ \mathit{seqauto} \\ &\_\ \mathit{AbsBy}^{+}_{\mathsf{SA}}\ \_ \equiv_{\mathsf{c}} (\lambda\ \mathit{SA}\ R\ . \\ &\qquad \mathsf{let}\ \ (S_{\mathsf{C}}, I_{\mathsf{C}}, L_{\mathsf{C}}, T_{\mathsf{C}}) = \mathit{Rep\_seqauto}\ \mathit{SA}; \\ &\qquad\qquad S_{\mathsf{A}} = S_{\mathsf{C}}; \\ &\qquad\qquad I_{\mathsf{A}} = I_{\mathsf{C}}; \\ &\qquad\qquad L_{\mathsf{A}} = L_{\mathsf{C}}\ \mathit{AbsBy}^{+}_{\mathsf{Ls}}\ R; \\ &\qquad\qquad T_{\mathsf{A}} = T_{\mathsf{C}}\ \mathit{AbsBy}^{+}_{\mathsf{Ts}}\ R; \\ &\qquad \mathsf{in}\ \ \mathit{Abs\_seqauto}\ (S_{\mathsf{A}}, I_{\mathsf{A}}, L_{\mathsf{A}}, T_{\mathsf{A}})) \end{aligned} \tag{7.1}$$

□

Der Operator ist folgendermaßen aufgebaut. Zunächst wird der zu abstrahierende Sequentielle Automat in seine Repräsentation überführt. Hierdurch können wir auf seine einzelnen Komponenten, wie die Zustandsmenge S_C, die initiale Zustandsmenge I_C, die Labelmenge L_C und die Transitionsrelation T_C zugreifen. Da wir eine strukturerhaltende Abstraktion definieren wollen, werden die strukturellen Merkmale über die Zustände aus S_C und aus I_C ohne Modifikation für den zu konstruierenden Sequentiellen Automaten übernommen. Die Abstraktion der Labelmenge und der Transitionsrelation wird durch die noch zu definierenden Konstruktionsoperatoren $AbsBy^+_{Ls}$ und $AbsBy^+_{Ts}$ umgesetzt. Im Folgenden werden wir uns auf die Konstruktion der abstrakten Transitionsrelation beschränken. Die Abstraktion der Labelmenge wird hingegen nicht vorgestellt. Der Grund dafür ist, dass die darin enthaltenen Label zusätzlich in den Transitionen der Transitionsrelation kodiert sind. Somit kann aus der abstrakten Transitionsrelation die abstrakte Labelmenge sehr einfach abgeleitet werden.

Zur Abstraktion der Transitionsrelation wird jede Transition mit dem Operator $AbsBy^+_T$ separat abstrahiert. Das Ergebnis dieses Operators ist eine Menge von Transitionen, da im Allgemeinen eine Transition des konkreten Modells durch mehrere Transitionen des abstrakten Modells repräsentiert ist.

Definition 7.4 (Konstruktion der Überapproximation einer Transition) *Sei T eine Transition und R eine Abstraktionsfunktion, dann ist die Überapproximation von T für R folgendermaßen definiert.*

$$
\begin{aligned}
&\_\ AbsBy^+_T\ \_ ::_C [(\sigma, \epsilon, \delta_C)\ trans, (\delta_C, \delta_A)\ abs] \\
&\qquad\qquad \rightarrow (\sigma, \epsilon, \delta_A)\ trans \\
&\_\ AbsBy^+_T\ \_ \equiv_C (\lambda\ T\ R\ . \\
&\qquad\qquad \mathsf{let}\ \ (SS_C, L_C, TS_C) = T; \\
&\qquad\qquad\qquad\qquad SS_A = SS_C; \\
&\qquad\qquad\qquad\qquad TS_A = TS_C; \\
&\qquad\qquad\qquad\qquad Ls_A = L_C\ AbsBy^+_L\ R; \\
&\qquad\qquad\qquad\qquad L_{Self} = L_C\ AbsBy^+_{Self}\ R; \\
&\qquad\qquad \mathsf{in}\quad \{(SS, L_A, TS)\ .\ L_A \in Ls_A \wedge SS = SS_A \wedge TS = TS_A\} \\
&\qquad\qquad\qquad \cup \{(SS_A, L_{Self}, SS_A)\}\ )
\end{aligned}
$$

□

Der in Definition 7.4 angegebene Operator $AbsBy^+_T$ ist folgendermaßen aufgebaut. Zunächst wird die zu abstrahierende Transition T in drei Bestandteile zerlegt. Hierbei wird mit SS_C der Startzustand, mit TS_C der Zielzustand und mit L_C das Label von T beschrieben. Um die strukturellen Merkmale zu erhalten, werden Start- und Zielzustand unverändert für die Konstruktion der abstrakten Transition übernommen.

Ferner wird mit dem noch zu definierenden Operator $AbsBy^+_L$ die Abstraktion für das Label L_C umgesetzt. Man beachte, dass der Operator eine Menge von abstrakten Labeln – in Definition 7.4 mit Ls_A bezeichnet – definiert. Für jedes Label dieser Menge wird eine abstrakte Transition eingeführt.

Schließlich wird mit dem noch zu definierenden Operator $AbsBy^+_{Self}$ das Label für eine selbstbezügliche Transition konstruiert. Diese Transition ist notwendig, um implizites Verhalten des konkreten Modells explizit zu erhalten. Sie wird am Startzustand SS_C eingefügt.

Man beachte, dass bei der Konstruktion dieser selbstbezüglichen Transition nicht die stärkste Überapproximation für das implizite Verhalten entsteht. Das liegt daran, dass andere Transitionen, die eventuell auch aus dem Startzustand $SS_{\mathtt{C}}$ herausführen, nicht berücksichtigt werden (vgl. Abschnitt 6.2.3). Es ist prinzipiell möglich einen Operator zu definieren, der eine derartig präzise Konstruktion festlegt. Nachteil einer solchen Definition ist allerdings, dass im Gegensatz zum Operator $AbsBy^{+}_{\mathtt{Self}}$ die Konstruktion nicht mehr lokal für eine Transition, sondern für eine Menge von Transitionen durchzuführen ist. Im Ergebnis leidet darunter der strukturelle Aufbau und damit die Transparenz der Operatoren ein wenig.

Im Folgenden wird auf den Aufbau des Konstruktionsoperators $AbsBy^{+}_{\mathtt{L}}$ genauer eingegangen.

Definition 7.5 (Konstruktion der Überapproximation eines Labels) *Sei L ein Label und R eine Abstraktionsfunktion, dann ist die Überapproximation von L für R folgendermaßen definiert.*

$$
\begin{aligned}
&\_\ AbsBy^{+}_{\mathtt{L}}\ \_ ::_{\mathtt{c}} [(\sigma, \epsilon, \delta_{\mathtt{c}})\ label, (\delta_{\mathtt{c}}, \delta_{\mathtt{A}})\ abs] \\
&\qquad\qquad \rightarrow (\sigma, \epsilon, \delta_{\mathtt{A}})\ label\ set \\
&\_\ AbsBy^{+}_{\mathtt{L}}\ \_ \equiv_{\mathtt{c}} (\lambda\ L\ R\ . \\
&\qquad\qquad \mathsf{let}\ \ (E_{\mathtt{C}}, G_{\mathtt{C}}, Es_{\mathtt{C}}, U_{\mathtt{C}}) = L; \\
&\qquad\qquad\qquad\qquad E_{\mathtt{A}} = E_{\mathtt{C}}; \\
&\qquad\qquad\qquad\qquad G_{\mathtt{A}} = G_{\mathtt{C}}\ AbsBy^{+}_{\mathtt{G}}\ R; \\
&\qquad\qquad\qquad\qquad Es_{\mathtt{A}} = Es_{\mathtt{C}}; \\
&\qquad\qquad\qquad\qquad Us_{\mathtt{A}} = U_{\mathtt{C}}\ AbsBy^{+}_{\mathtt{U}}\ R; \\
&\qquad\qquad \mathsf{in}\quad \{\, (E, G, Es, U_{\mathtt{A}})\ .\ E = E_{\mathtt{A}} \wedge G = G_{\mathtt{A}} \wedge Es = Es_{\mathtt{A}} \wedge U_{\mathtt{A}} \in Us_{\mathtt{A}} \,\})
\end{aligned}
$$

□

Zunächst wird das zu abstrahierende Label in seine Bestandteile zerlegt. Hierbei beschreibt $E_{\mathtt{C}}$ die Expression, $G_{\mathtt{C}}$ den Guard, $Es_{\mathtt{C}}$ die im Aktionsteil angegebene Ereignismenge und $U_{\mathtt{C}}$ die partielle Update-Funktion des Labels. Das Kernstück der Konstruktion besteht einerseits in der Abstraktion des Guards mit dem Operator $AbsBy^{+}_{\mathtt{G}}$ und andererseits in der Abstraktion der partiellen Update-Funktion mit dem Operator $AbsBy^{+}_{\mathtt{U}}$.

Die Definition des Operators $AbsBy^{+}_{\mathtt{G}}$ ist direkt aus dem Operator α^{+}(vgl. Abschnitt 6.1) abgeleitet und repräsentiert die strengste Nachbedingung von R in Bezug auf den zu abstrahierenden Guard G. In der Definition wird die Abstraktion einer konkreten Datenraumbelegung $D_{\mathtt{C}}$ mit dem Operator $AbsBy^{+}_{\mathtt{D}}$ ermittelt.

Definition 7.6 (Konstruktion der Überapproximation eines Guards) *Sei G ein Guard und R eine Abstraktionsfunktion, dann ist die Überapproximation von G für R folgendermaßen definiert.*

$$
\begin{aligned}
&\_\ AbsBy^{+}_{\mathtt{G}}\ \_ ::_{\mathtt{c}} [\delta_{\mathtt{c}}\ guard, (\delta_{\mathtt{c}}, \delta_{\mathtt{A}})\ abs] \rightarrow \delta_{\mathtt{A}}\ guard \\
&\_\ AbsBy^{+}_{\mathtt{G}}\ \_ \equiv_{\mathtt{c}} (\lambda\ G\ R\ D_{\mathtt{A}}\ .\ \exists\ D_{\mathtt{C}}\ .\ (G\ D_{\mathtt{C}})\ \wedge\ (D_{\mathtt{C}}\ AbsBy^{+}_{\mathtt{D}}\ R) = D_{\mathtt{A}})
\end{aligned}
$$

□

Für einen noch vorzustellenden Konstruktionsoperator, der zur Beschreibung von implizitem Verhalten eingesetzt wird, benötigen wir auch eine Konstruktionsvorschrift für die Unterapproximation eines Guards. Mit folgender Definition führen wir diesen Operator ein.

Definition 7.7 (Konstruktion der Unterapproximation eines Guards) *Sei* G *ein Guard und* R *eine Abstraktionsfunktion, dann ist die Unterapproximation von* G *für* R *folgendermaßen definiert.*

$$\_\ \mathit{AbsBy}^{-}_{\mathsf{G}}\ \_ ::_{\mathsf{c}} [\delta_{\mathsf{c}}\ \mathit{guard}, (\delta_{\mathsf{c}}, \delta_{\mathsf{A}})\ \mathit{abs}] \to \delta_{\mathsf{A}}\ \mathit{guard}$$
$$\_\ \mathit{AbsBy}^{-}_{\mathsf{G}}\ \_ \equiv_{\mathsf{c}} (\lambda\ G\ R\ D_{\mathsf{A}}\ .\ \forall\ D_{\mathsf{C}}\ .\ ((D_{\mathsf{C}}\ \mathit{AbsBy}^{+}_{\mathsf{D}}\ R) = D_{\mathsf{A}})\ \Rightarrow\ (G\ D_{\mathsf{C}}))$$

□

Im Folgenden beschreiben wir die Konstruktion zur Überapproximation von partiellen Update-Funktionen.

Definition 7.8 (Konstruktion der Überapproximation einer Update-Funktion) *Sei* U_{C} *eine partielle Update-Funktion und* R *eine Abstraktionsfunktion, dann ist die Überapproximation von* U_{C} *für* R *folgendermaßen definiert.*

$$\begin{aligned}
&_\ \mathit{AbsBy}^{+}_{\mathsf{U}}\ _ ::_{\mathsf{c}} [\delta_{\mathsf{c}}\ \mathit{pupdate}, (\delta_{\mathsf{c}}, \delta_{\mathsf{A}})\ \mathit{abs}] \to \delta_{\mathsf{A}}\ \mathit{pupdate\ set}\\
&_\ \mathit{AbsBy}^{+}_{\mathsf{U}}\ _ \equiv_{\mathsf{c}} (\lambda\ U_{\mathsf{C}}\ R\ .\\
&\qquad \{\ U_{\mathsf{A}}\ .\ \forall\ D_{\mathsf{A}}\ .\\
&\qquad\quad \mathsf{if}\ D_{\mathsf{A}} \in (\mathit{range}\ R)\ \mathsf{then}\\
&\qquad\qquad \exists\ D_{\mathsf{C}}\ .\ (U_{\mathsf{A}}\,!!\,D_{\mathsf{A}}) = (U_{\mathsf{C}}\,!!\,D_{\mathsf{C}})\ \mathit{AbsBy}^{+}_{\mathsf{PD}}\ R\ \wedge\\
&\qquad\qquad\qquad D_{\mathsf{A}} = D_{\mathsf{C}}\ \mathit{AbsBy}^{+}_{\mathsf{D}}\ R\\
&\qquad\quad \mathsf{else}\ (U_{\mathsf{A}}\,!!\,D_{\mathsf{A}}) = (\mathit{DataAsPData}\ D_{\mathsf{A}})\ \}\ )
\end{aligned}$$

□

Der Operator $\mathit{AbsBy}^{+}_{\mathsf{U}}$ konstruiert für eine auf dem konkreten Datenraum definierte partielle Update-Funktion U_{C} eine endliche Menge von partiellen Update-Funktionen, die auf dem abstrakten Datenraum definiert sind. Der Operator garantiert, dass sich das Verhalten jeder konstruierbaren abstrakten Update-Funktionen U_{A} durch das Verhalten von U_{C} simulieren läßt. Abbildung 7.2 veranschaulicht graphisch die umgesetzte Simulationseigenschaft. Allerdings fordern wir diese Eigenschaft nur für Eingaben, die auch im Bildbereich der Abstraktionsfunktion R liegen, da nur für diese Datenraumbelegungen eine sinnvolle Interpretation im konkreten Datenraum vorgenommen werden kann. Das Verhalten für Eingaben außerhalb des Bildbereiches von R ist für eine korrekte Überapproximation nicht relevant und wir könnten beliebiges Verhalten erlauben. Um aber nicht unnötig viele Update-Funktionen zu konstruieren, fordern wir für eine Eingabe D_{A}, die außerhalb des Bildbereiches von R liegt, dass D_{A} von U_{A} unverändert als Ergebnis zurückgegeben wird.

Alternativ zur Definition 7.8 kann die Überapproximation einer Update-Funktion auch prädikatenbasiert erfolgen. Hierfür wird eine partielle Update-Funktion U_{C} durch ein binäres Prädikat interpretiert, mit dem der Zusammenhang zwischen den Ein- und Ausgaben von U_{C} beschrieben werden kann. Dieses Prädikat wird anschließend auf Basis der in Abschnitt 6.1 angegebenen Galois-Korrespondenzen in ein Prädikat über dem abstrakten Datenraum

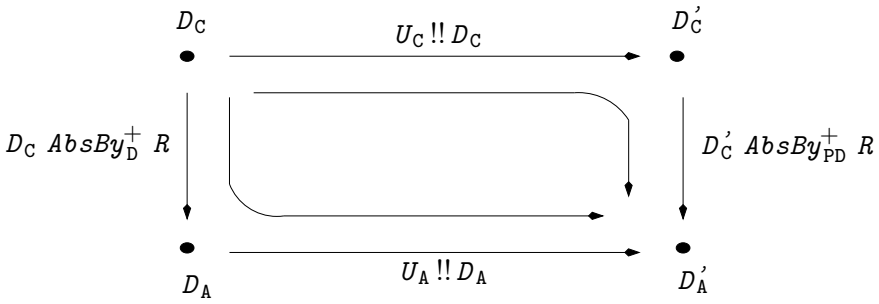


Abbildung 7.2: Simulationseigenschaft zur Überapproximation von Update-Funktionen

transformiert. Verwenden wir hierbei die Abstraktionsfunktion α^+, so ist das Ergebnis der Transformation eine Überapproximation von U_C.

Um Definition 7.8 zu validieren, zeigen wir, dass das Verhalten der mit dem Operator $AbsBy_U^+$ konstruierten Update-Funktionen mit dem Verhalten zusammenfällt, das bei einer prädikatenbasierten Überapproximation entsteht.

Hierfür führen wir zunächst mit dem Operator $AbsPredBy_U^+$ die Konstruktion einer prädikatenbasierten Überapproximation ein.

$$
\begin{aligned}
&\_\ AbsPredBy_U^+\ \_ ::_c [\delta_c\ pupdate, (\delta_c, \delta_A)\ abs] \rightarrow (\delta_A\ data * \delta_A\ pdata)\ pred \\
&\_\ AbsPredBy_U^+\ \_ \equiv_c (\lambda\ U_C\ R\ .\ \\
&\qquad\qquad (\lambda\ (D_A, D_A')\ .\ \exists\ D_C\ D_C'\ .\ (U_C !! D_C) = D_C'\ \wedge \\
&\qquad\qquad\qquad\qquad (D_C'\ AbsBy_{PD}^+\ R) = D_A'\ \wedge \\
&\qquad\qquad\qquad\qquad (D_C\ AbsBy_D^+\ R) = D_A))
\end{aligned}
$$

Um ein durch den Operator $AbsPredBy_U^+$ konstruiertes Prädikat mit den konstruierten Update-Funktionen aus Definition 7.8 in Beziehung setzen zu können, führen wir einen weiteren Operator UP ein. Mit UP kann eine abstrakte Update-Funktion U_A in ein binäres Prädikat überführt werden.

$$
\begin{aligned}
UP &::_c [(\delta_c, \delta_A)\ abs, \delta_A\ pupdate] \rightarrow (\delta_A\ data * \delta_A\ pdata)\ pred \\
UP\ R\ U_A &\equiv_c \lambda\ (D_A, D_A')\ .\ (U_A !! D_A) = D_A'\ \wedge\ D_A \in (range\ R)
\end{aligned}
$$

Das Prädikat beschreibt nicht das gesamte Verhalten von U_A, sondern ist auf Verhalten für Eingaben beschränkt, die im Bildbereich der Abstraktionsfunktion R liegen. Folgendes in Isabelle/HOL abgeleitetes Theorem belegt, dass das Verhalten der Update-Funktionen aus Definition 7.8 mit dem bei einer prädikatenbasierten Überapproximation entstehenden Verhalten identisch ist.

Abgeleitetes Theorem 7.9 (Überapproximation von Update-Funktionen)

$$
U_C\ AbsPredBy_U^+\ R = \bigvee_P\ (UP\ R)\ `\ (U_C\ AbsBy_U^+\ R) \qquad \text{(UpdateIsPred)}
$$

□

Um die beiden Konstruktionen miteinander vergleichen zu können, ist zunächst die mit dem Operator $AbsBy^{+}_{\mathsf{U}}$ konstruierte Menge an abstrakten Update-Funktionen mit Hilfe von UP in eine Menge von binären Prädikaten transformiert worden. Mit dem Theorem UpdateIsPred haben wir schließlich gezeigt, dass die Disjunktion dieser Prädikate mit dem Prädikat identisch ist, das der Operator $AbsPredBy^{+}_{\mathsf{U}}$ definiert.

Nachdem wir die Konstruktionsoperatoren für die Abstraktion eines Labels eingeführt haben, wird zum Abschluß dieses Abschnitts noch der Operator $AbsBy^{+}_{\mathtt{Self}}$ vorgestellt. Das mit diesem Operator konstruierte Label wird in Definition 7.4 dazu verwendet, eine selbstbezügliche Transition zu markieren. Mit dieser Transition wird implizites Verhalten des konkreten Modells modelliert.

Definition 7.10 (Konstruktion von implizitem Verhalten) *Sei* L *ein Label und* R *eine Abstraktionsfunktion, dann kann das durch die Überapproximation des Labels* L *reduzierte Verhalten dem Modell wieder hinzugefügt werden. Hierzu wird eine selbstbezügliche Transition eingefügt, deren Label folgendermaßen konstruiert ist.*

$$
\begin{aligned}
&\_\ AbsBy^{+}_{\mathtt{Self}}\ \_ :: _{\mathsf{c}}\ [(\sigma, \epsilon, \delta_{\mathsf{c}})\ label, (\delta_{\mathsf{c}}, \delta_{\mathsf{A}})\ abs] \\
&\qquad\qquad \rightarrow (\sigma, \epsilon, \delta_{\mathsf{A}})\ label \\
&\_\ AbsBy^{+}_{\mathtt{Self}}\ \_ \equiv_{\mathsf{c}} (\lambda\ L\ R\ . \\
&\qquad \mathsf{let}\ \ (E_{\mathsf{C}}, G_{\mathsf{C}}, \_, \_) = L; \\
&\qquad\qquad\qquad E_{\mathsf{A}} = E_{\mathsf{C}}; \\
&\qquad\qquad\qquad G^{+}_{\mathsf{A}} = G_{\mathsf{C}}\ AbsBy^{+}_{\mathsf{G}}\ R; \\
&\qquad\qquad\qquad G^{-}_{\mathsf{A}} = G_{\mathsf{C}}\ AbsBy^{-}_{\mathsf{G}}\ R; \\
&\qquad\qquad\qquad G_{\mathsf{A}} = G^{+}_{\mathsf{A}} \wedge \neg\ G^{-}_{\mathsf{A}}; \\
&\qquad\qquad\qquad Es_{\mathsf{A}} = \varnothing; \\
&\qquad\qquad\qquad U_{\mathsf{A}} = PUpdateDefault; \\
&\qquad \mathsf{in}\quad (E_{\mathsf{A}}, G_{\mathsf{A}}, Es_{\mathsf{A}}, U_{\mathsf{A}})\ )
\end{aligned}
$$

□

Ähnlich zu den anderen in diesem Abschnitt vorgestellten Definitionen wird das übergebene Label L zunächst in seine Bestandteile zerlegt. Da nicht alle Bestandteile für die Konstruktion relevant sind, werden lediglich die Expression E_{C} und der Guard G_{C} selektiert.

Das zu konstruierende Label wird benötigt, um das implizite Verhalten zu modellieren, das beim Bilden der Überapproximation von L reduziert wird. Implizites Verhalten kann aber nur dann reduziert werden, wenn die Expression von L erfüllt ist. Aus diesem Grund wird E_{C} ohne weitere Modifikation für E_{A} übernommen.

Um den abstrakten Guard G_{A} zu konstruieren, bilden wir die Konjunktion aus der Überapproximation von G_{C} und der negierten Unterapproximation von G_{C}. Unter Ausnutzung des Theorems DualityWPInvSP aus Abschnitt 6.1 kann die Definition weiter vereinfacht werden, indem wir die negierte Unterapproximation von G_{C} durch die Überapproximation von $\neg\ G_{\mathsf{C}}$ ersetzen.

Da die Semantik von Hierarchischen Automaten das Erzeugen von internen Ereignissen für implizites Verhalten ausschließt, ist die Ereignismenge im Aktionsteil des zu konstruierenden Labels leer.

Die in der Konstruktion verwendete Update-Funktion *PUpdateDefault* gibt an, wie der Datenzustand beim Auftreten von impliziten Verhalten zu verändern ist (vgl. Abschnitt 6.3.3).

7.3 Konstruktion von überapproximierten Hierarchischen Automaten

Um die im letzten Abschnitt eingeführten Konstruktionsoperatoren zur Konstruktion von Sequentiellen Automaten für Hierarchische Automaten nutzbar zu machen, definieren wir folgenden Konstruktionsoperator in kompositionaler Art und Weise.

Definition 7.11 (Konstruktion der Überapproximation eines HA**)** *Sei* HA *ein Hierarchischer Automat und* R *eine Abstraktionsfunktion, dann ist die Überapproximation von* HA *für* R *folgendermaßen definiert.*

$$
\begin{aligned}
&\_\ AbsBy^{+}_{\mathrm{HA}}\ \_ ::_{\mathrm{c}} [\,(\sigma,\epsilon,\delta_{\mathrm{c}})\ hierauto, (\delta_{\mathrm{c}},\delta_{\mathrm{A}})\ abs\,] \\
&\qquad\qquad \rightarrow (\sigma,\epsilon,\delta_{\mathrm{A}})\ hierauto \\
&\_\ AbsBy^{+}_{\mathrm{HA}}\ \_ \equiv_{\mathrm{c}} (\lambda\ HA\ R\ . \\
&\qquad\qquad \mathsf{let}\ \ (SA_{\mathrm{C}}, E_{\mathrm{C}}, F_{\mathrm{C}}, D_{\mathrm{C}}) = Rep\_hierauto\ HA; \\
&\qquad\qquad\qquad\qquad SA_{\mathrm{A}} = (\lambda\ SA\ .\ SA\ AbsBy^{+}_{\mathrm{SA}}\ R)\ `SA_{\mathrm{C}}\,; \\
&\qquad\qquad\qquad\qquad E_{\mathrm{A}} = E_{\mathrm{C}}; \\
&\qquad\qquad\qquad\qquad F_{\mathrm{A}} = F_{\mathrm{C}}\ AbsBy^{+}_{\mathrm{CF}}\ R; \\
&\qquad\qquad\qquad\qquad D_{\mathrm{A}} = D_{\mathrm{C}}\ AbsBy^{+}_{\mathrm{D}}\ R; \\
&\qquad\qquad \mathsf{in}\ \ Abs\_hierauto\ (SA_{\mathrm{A}}, E_{\mathrm{A}}, F_{\mathrm{A}}, D_{\mathrm{A}}))
\end{aligned}
\tag{7.2}
$$

□

Zunächst transformieren wir den zu abstrahierenden Hierarchischen Automat in seine Bestandteile. Hierbei beschreibt SA_{C} die Menge an Sequentiellen Automaten, E_{C} die Menge an Ereignissen, F_{C} die Kompositionsfunktion und D_{C} die initiale Datenraumbelegung. Das Kernstück der Abstraktion besteht in der Anwendung des Operators $AbsBy^{+}_{\mathrm{SA}}$ auf alle Elemente der Menge SA_{C}. Dabei wird jeder Sequentielle Automat des Hierarchischen Automaten unabhängig abstrahiert. Die Menge E_{C} wird durch die Abstraktion nicht verändert. Um die Strukturmerkmale des Hierarchischen Automaten bei der Konstruktion zu erhalten, wird F_{C} im Wesentlichen unverändert übernommen. Der Operator $AbsBy^{+}_{\mathrm{CF}}$ ersetzt lediglich alle in F_{C} behandelten Sequentiellen Automaten durch ihre abstrakten Gegenstücke. Analog zur Abstraktion von SA_{C} wird hierzu der Operator $AbsBy^{+}_{\mathrm{SA}}$ verwendet. D_{C} repräsentiert eine totale Datenraumbelegung und wird mit Hilfe des Operator $AbsBy^{+}_{\mathrm{D}}$ in eine abstrakte Datenraumbelegung D_{A} transformiert.

Mit Definition 7.3 haben wir in diesem Abschnitt einen Operator eingeführt, mit dessen Hilfe eine Überapproximation für Hierarchische Automaten konstruierbar ist. Bezugnehmend auf den einleitenden Abschnitt 6.1 zur Abstraktionstheorie ist damit die Frage beantwortet, ob es möglich ist, einen Operator Abs_{HA} anzugeben. Ferner haben wir bereits in Kapitel 6 ausführlich erläutert, warum diese Konstruktion zu einer Überapproximation und damit zu einer eigenschaftserhaltenden Abstraktion für das universale Fragments der CTL führt. Die bisher informell beschriebene Aussage können wir durch folgenden Ausdruck formalisieren.

$$(Abs_{\mathrm{HA}}\, HA \models_{\mathrm{HA}} Abs_{\mathrm{CTL}}\, F) \Rightarrow (HA \models_{\mathrm{HA}} F)$$

In dieser Formalisierung benötigen wir allerdings neben dem bereits eingeführten Operator zur Abstraktion Hierarchischer Automaten einen Operator, mit dem temporallogische Formeln des universalen Fragments der CTL geeignet abstrahiert werden können. Einen solchen Operator werden wir im folgenden Abschnitt einführen.

7.4 Konstruktion von unterapproximierten CTL-Formeln

Um eine temporallogische Formel zu verifizieren, ist es in der Regel wünschenswert, die Abstraktionsfunktion so zu definieren, dass die Formel exakt beschrieben werden kann. In so einem Fall wird die temporallogische Eigenschaft also weder abgeschwächt noch verschärft. Für die Prädikatenabstraktion von ∀CTL-Formeln kann das beispielsweise dadurch umgesetzt werden, dass alle in den Formeln vorkommenden atomaren Aussagen durch boolesche Variablen des abstrakten Datenraumes explizit repräsentiert werden. Wenn wir in den vorherigen Kapiteln dieser Arbeit angegeben haben, dass die Überapproximation Hierarchischer Automaten bezüglich des universalen Fragments der CTL eigenschaftserhaltend ist, so sind wir immer davon ausgegangen, dass sich die zu verifizierende ∀CTL-Formel auf dem abstrakten Datenraum exakt beschreiben lässt.

Trotzdem kann es sinnvoll sein, eine zu verifizierende ∀CTL-Formel in der Abstraktion nur approximativ zu beschreiben, da dadurch die Komplexität des abstrakten Datenraumes reduziert und der sich anschließende Beweisprozess mit Model-Checking effizienter geführt werden kann.

Um für ein solches Vorgehen eine eigenschaftserhaltende Abstraktion zu garantieren, wird in der Literatur vorgeschlagen, eine Unterapproximation der zu verifizierenden temporallogischen Formel zu bilden [SS99]. Dadurch wird die Eigenschaft verschärft. Folglich wird versucht, auf dem abstrakten Modell eine stärkere Aussage zu beweisen, als ursprünglich gefordert. Ist dieser Beweis erfolgreich, so folgt daraus sofort, dass die ursprünglich schwächere Eigenschaft auf dem konkreten Modell erfüllt ist.

Basierend auf der in Kapitel 5 eingeführten Formalisierung von CTL geben wir mit folgender Definition die Konstruktionsvorschrift für die Unterapproximation einer ∀CTL-Formel an.

Definition 7.12 (Konstruktion der Unterapproximation einer CTL-Formel) *Sei* F *eine* ∀CTL*-Formel vom Typ* $(\sigma, \epsilon, \delta_c)$ `hactl` *und* R *eine Abstraktionsfunktion vom Typ* (δ_c, δ_a) `abs`*, dann ist mit folgender primitiv-rekursiven Konstantendefinition festgelegt, wie eine Unterapproximation von* F *mit dem Typ* $(\sigma, \epsilon, \delta_a)$ `hactl` *konstruiert werden kann.*

$$
\begin{aligned}
(\mathit{Atom}\ P)\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{Atom}\ (P\ \mathit{AbsBy}^{-}_{\mathrm{A}}\ R)\\
(\mathit{Neg}\ (\mathit{Atom}\ P))\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{Neg}\ (\mathit{Atom}\ (P\ \mathit{AbsBy}^{+}_{\mathrm{A}}\ R))\\
(\mathit{And}\ F\ G)\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{And}\ (F\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\ (G\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\\
(\mathit{Or}\ F\ G)\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{Or}\ (F\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\ (G\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\\
(\mathit{Imp}\ F\ G)\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{Imp}\ (F\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\ (G\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\\
(\mathit{AX}\ F)\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{AX}\ (F\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)\\
(\mathit{AF}\ F)\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R &= \mathit{AF}\ (F\ \mathit{AbsBy}^{-}_{\mathrm{CTL}}\ R)
\end{aligned}
$$

$$
\begin{aligned}
(AG\ F)\ AbsBy^{-}_{\mathrm{CTL}}\ R &= AG\ (F\ AbsBy^{-}_{\mathrm{CTL}}\ R)\\
(AU\ F\ G)\ AbsBy^{-}_{\mathrm{CTL}}\ R &= AU\ (F\ AbsBy^{-}_{\mathrm{CTL}}\ R)\ (G\ AbsBy^{-}_{\mathrm{CTL}}\ R)\\
(AR\ F\ G)\ AbsBy^{-}_{\mathrm{CTL}}\ R &= AR\ (F\ AbsBy^{-}_{\mathrm{CTL}}\ R)\ (G\ AbsBy^{-}_{\mathrm{CTL}}\ R)
\end{aligned}
$$

□

Das Kernstück der Konstruktion besteht in der Unterapproximation atomarer Aussagen durch den Operator $AbsBy^{-}_{\mathrm{A}}$, der mit folgender Definition eingeführt wird.

Definition 7.13 (Konstruktion der Unterapproximation einer atomaren Aussage) *Sei AP eine atomare Formel vom Typ $(\sigma, \epsilon, \delta_{\mathrm{c}})\ atomar$ und R eine Abstraktionsfunktion vom Typ $(\delta_{\mathrm{c}}, \delta_{\mathrm{A}})\ abs$, dann ist mit folgender primitiv-rekursiven Konstantendefinition festgelegt, wie eine Unterapproximation von AP mit dem Typ $(\sigma, \epsilon, \delta_{\mathrm{A}})\ atomar$ konstruiert werden kann.*

$$
\begin{aligned}
True\ AbsBy^{-}_{\mathrm{A}}\ R &= True\\
(In\ S)\ AbsBy^{-}_{\mathrm{A}}\ R &= In\ S\\
(En\ E)\ AbsBy^{-}_{\mathrm{A}}\ R &= En\ E\\
(Val\ P)\ AbsBy^{-}_{\mathrm{A}}\ R &= Val\ (P\ AbsBy^{-}_{\mathrm{G}}\ R)
\end{aligned}
$$

□

In Definition 7.13 wurde der Operator $AbsBy^{-}_{\mathrm{G}}$ verwendet, um die Unterapproximation eines atomaren Prädikates zu bilden. Dieser Operator wurde ursprünglich dazu eingeführt, um die Unterapproximation für einen Guard zu konstruieren. Da Guards in unserer Formalisierung durch atomare Prädikate repräsentiert sind, können wir den Operator $AbsBy^{-}_{\mathrm{G}}$ an dieser Stelle wiederverwenden. In Definition 7.12 wurde zum Bilden der Unterapproximation einer negierten atomaren Formel der Operator $AbsBy^{+}_{\mathrm{A}}$ verwendet. Die Definition für diesen Operator ist in Analogie zu Definition 7.13 unter Verwendung des Operators $AbsBy^{+}_{\mathrm{G}}$ leicht anzugeben. Wir verzichten hier auf eine ausführliche Darstellung.

7.5 Zusammenfassung

In diesem Kapitel haben wir eine in Isabelle/HOL umgesetzte Formalisierung der Datenabstraktionstheorie aus Kapitel 6 vorgestellt. Viele der präsentierten Ergebnisse sind bereits veröffentlicht [HK05].

Im ersten Abschnitt definierten wir einen Datentyp für wohlgeformte Abstraktionsfunktionen, mit dem die Struktur eines Datenraumes bei der Abstraktion erhalten werden kann. Anschließend stellten wir eine Reihe von Konstruktionsoperatoren vor, mit deren Hilfe für eine wohlgeformte Abstraktionsfunktion die Überapproximation eines Sequentiellen Automaten in Isabelle/HOL definiert wurde. Im darauf folgenden Abschnitt definierten wir schließlich einen Operator, der die Überapproximation eines Hierarchischen Automaten festlegt. Der Operator ist in Anlehnung an das Konzept aus Kapitel 6 in kompositionaler Art und Weise angegeben. Entsprechend haben wir den Operator zur Überapproximation Hierarchischer Automaten mit Hilfe des Operators zur Überapproximation Sequentieller Automaten definiert. Im letzten Abschnitt dieses Kapitels stellten wir schließlich einen Operator vor, mit dem die Unterapproximation einer temporallogischen Formel des universalen Fragments der CTL definiert wird.

Die von uns entwickelte Abstraktionstheorie ermöglicht es, die Eigenschaftserhaltung Hierarchischer Automaten bezüglich des universalen Fragments der CTL formal zu fassen. Im folgenden Kapitel widmen wir uns der Fragestellung, wie das praktische Arbeiten zum Verifizieren konkreter Statecharts-Spezifikationen in unserer Theorie möglichst effizient gestaltet werden kann. Wir stellen hierfür zwei Taktiken vor, mit denen vertrauenswürdige Beweiswerkzeuge in den Beweisprozess von Isabelle/HOL eingebunden werden.

Kapitel 8

Praktische Analysen für Statecharts

Um die vorgestellte Abstraktionstheorie für die Verifikation von praktischen Anwendungsbeispielen nutzbar zu machen, haben wir effiziente Beweistaktiken implementiert. Die Beweisführung wird hierbei durch externe Verifikationswerkzeuge unterstützt, die wir über die Orakelschnittstelle von Isabelle/HOL im Rahmen dieser Dissertation in Taktiken eingebunden haben. Abbildung 8.1 zeigt einen Überblick über die entwickelte Werkzeugumgebung.

Dieses Kapitel gliedert sich in drei Abschnitte. Zunächst stellen wir eine Beweistaktik vor, mit deren Hilfe in Isabelle/HOL formalisierte Statecharts-Spezifikationen unter Verwendung des Model-Checkers SMV verifiziert werden können. Diese Taktik ist nur dann anwendbar, wenn die zu verifizierenden Spezifikationen auf einem endlichen Datenraum basieren. Für den Fall, dass eine Statecharts-Spezifikation auf einem unendlichen Datenraum definiert ist, stellen wir in Abschnitt 8.2 einen im Rahmen dieser Dissertation implementierten Abstraktionsalgorithmus vor. Dieser Algorithmus basiert auf einer Prädikatenabstraktion und erzeugt für eine gegebene Abstraktionsfunktion eine Statecharts-Spezifikation, die auf einem endlichen Datenraum definiert ist. Zur Auswertung anfallender Beweisaufgaben nutzen wir den automatischen Theorembeweiser SVC [Dil]. Im letzten Abschnitt dieses Kapitels stellen wir eine Fallstudie vor, die mit Hilfe der in dieser Dissertation entwickelten Werkzeugumgebung verifiziert wurde. Es handelt sich hierbei um die Steuerung eines Kühlkreislaufs in einem nuklearen Kraftwerk.

8.1 Model-Checking

In diesem Abschnitt beschreiben wir eine Isabelle-Taktik, mit deren Hilfe Statecharts-Spezifikationen, die auf einem endlichen Datenraum definiert sind, durch den Model-Checker SMV analysiert werden können. Zunächst beschreiben wir den allgemeinen Aufbau der Taktik und geben einen Eindruck davon, wie sie in dem Theorembeweiser Isabelle bei der Ableitung von Theoremen eingesetzt werden kann. Die Taktik basiert auf einer Übersetzungstheorie von Erich Mikk [Mik00], die wir im zweiten Teil dieses Abschnitts anhand eines Beispiels skizzieren. Da Erich Mikk in seinen Arbeiten die Behandlung von Datenvariablen nicht unterstützt, widmen wir uns im letzten Teil dieses Abschnitts einem Konzept, mit dem eine entsprechende Erweiterung vorgenommen werden kann [HK03b].

8.1.1 Taktik zur Anbindung von SMV

Um die Gültigkeit einer CTL-Formel in einer Statecharts-Spezifikation effizient nachweisen zu können, haben wir eine Isabelle-Taktik in ML [Pau96] implementiert, die diesen Nachweis mit

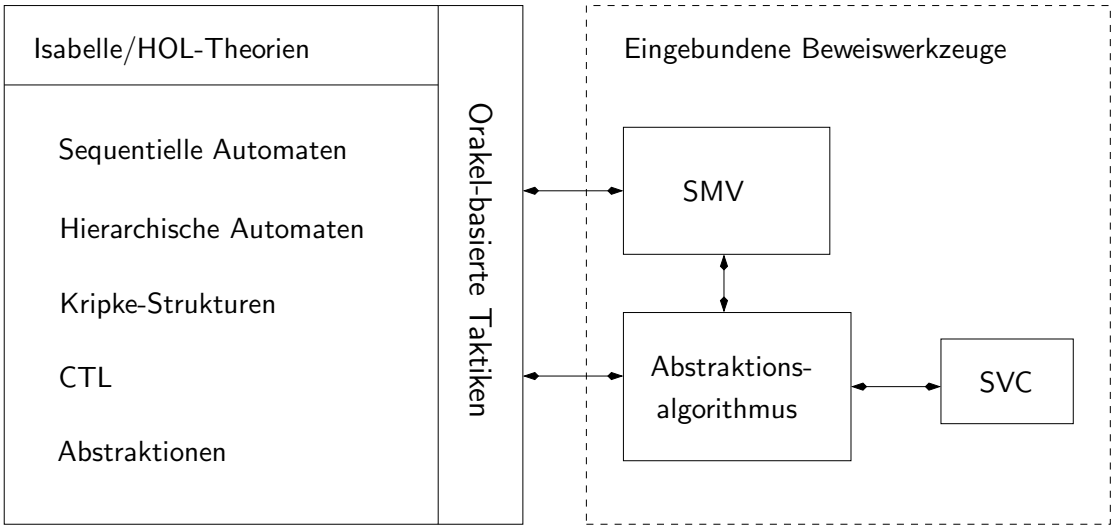


Abbildung 8.1: Aufbau des Frameworks: Isabelle-Theorien und eingebundene Beweiswerkzeuge

Hilfe des Model-Checkers SMV vornimmt. Ein Prototyp für diese Taktik entstand im Rahmen der Diplomarbeit von Tuvshintur Tserendorj [Tse03], durch die wir erste wichtige Erfahrungen sammeln konnten. Die jetzt vorliegende Taktik stellt eine Neuimplementierung des in der Diplomarbeit entstandenen Prototypen dar und unterstützt jetzt vollständig den in Kapitel 2.1 beschriebenen Sprachumfang von Statecharts. Aus der Sicht eines Isabelle-Benutzers gestaltet sich das Arbeiten mit der Taktik folgendermaßen. Angenommen in Isabelle/HOL wurde, um die Gültigkeit von F in HA abzuleiten, folgendes Beweisziel abgesetzt.

$$HA \models_{\mathrm{HA}} F$$

Wird unsere Taktik auf diesen Beweiszustand angewendet, so wird zunächst die Isabelle-interne Termstruktur analysiert und die dabei gewonnenen Informationen über F und HA in einer baumartigen ML-Datenstruktur abgelegt. Anschließend wird auf Basis dieser Datenstruktur eine Eingabedatei für SMV erzeugt und durch Aufruf des Model-Checkers das Beweisziel überprüft.

Sollte der Verifikationsprozess nicht erfolgreich sein und ein Gegenbeispiel zurückliefern, so wird die Isabelle-Taktik scheitern. Entsprechend bleibt der Beweiszustand in Isabelle nach Ausführung der Taktik unverändert.

Gelingt es hingegen die Aussage mit SMV zu verifizieren, so wird die Taktik das Beweisziel in Isabelle als bewiesen deklarieren und ein entsprechendes Theorem einführen. Da allerdings beim Nachweis mit unserer Taktik die Orakelschnittstelle genutzt wird, bekommt das abgeleitete Theorem eine spezielle Markierung. Diese Markierung zeigt an, dass der Beweis nicht allein auf Basis von in Isabelle abgeleiteten Theoremen geführt wurde und andere, eventuell weniger vertrauenswürdige Werkzeuge für den Beweis zum Einsatz gekommen sind.

8.1.2 Übersetzung Hierarchischer Automaten nach SMV

Die Eingabesprache von SMV ist dazu geeignet, zustandsbasierte dynamische Systeme zu beschreiben. Eine präzise Syntax und Semantik der Sprache wurde von Ken McMillan angegeben [McM93]. Wir geben hier nur eine kurze Einführung.

Das zu verifizierende Modell – in unserer Arbeit die Statecharts-Spezifikation – wird durch eine so genannte Modulbeschreibung spezifiziert. Der Zustandsraum eines Moduls wird durch eine Menge von Zustandsvariablen festgelegt. Zustandsvariablen werden mit einem geeigneten Typ durch das Schlüsselwort `VAR` eingeführt. Eine Initialisierung ist mit dem Schlüsselwort `INIT` möglich. Folgezustände von Zustandsvariablen können durch eine Transitionsrelation mit Hilfe der Schlüsselwörter `ASSIGN` und `TRANS` spezifiziert werden. Mit `ASSIGN` wird der Folgezustand einer Zustandsvariablen durch die Zuweisung einer neuen Belegung angegeben. Die Sprachsemantik von SMV legt fest, dass mehrere `ASSIGN`-Anweisungen für einen Zustandsübergang des Systems synchron auszuführen sind. Dies entspricht einer synchronen Schrittsemantik, die es ermöglicht, den Statecharts-Formalismus adäquat in der Eingabesprache von SMV abzubilden.

Man beachte, dass `ASSIGN`-Anweisungen nur operationales Spezifizieren unterstützen. Dies bedeutet, dass die Transitionsrelation durch direkte Zuweisungen an den Folgezustand einer Zustandsvariablen modelliert werden muss. Im Gegensatz dazu kann bei Verwendung des Schlüsselwortes `TRANS` das Verhalten der Transitionsrelation deutlich abstrakter modelliert werden. Hier sind zur Beschreibung von Transitionen propositionale Formeln und Gleichungen vorgesehen.

Zusätzlich ist es mit dem Schlüsselwort `DEFINE` möglich, einfache Prädikate zur Beschreibung des Kontrollflusses zu formulieren.

Eine zu überprüfende CTL-Formel kann mit Hilfe des Schlüsselwortes `SPEC` angegeben werden. Ferner gibt es für alle Operatoren der CTL entsprechende Bezeichner.

Die Konzeption unserer Taktik basiert auf der von Erich Mikk vorgeschlagenen Übersetzungsvorschrift zur Transformation Erweiterter Hierarchischer Automaten in die Eingabesprache von SMV. Wir wollen dem Leser anhand des in Abschnitt 4 vorgestellten Beispiels einen intuitiven Eindruck von der schrittweisen Übersetzung geben (vgl. Abbildung 4.1 auf Seite 72). Für eine ausführliche Darstellung der zugrunde liegenden Theorie verweisen wir auf die Dissertation von Erich Mikk [Mik00].

Zunächst werden für alle Sequentiellen Automaten des zu verifizierenden Hierarchischen Automaten so genannte *Automaten*-Variablen eingeführt. Die *Automaten*-Variable eines Sequentiellen Automaten wird durch einen Aufzählungstyp deklariert, mit dem eindeutig beschrieben werden kann, welcher Zustand des Sequentiellen Automaten gerade aktiv ist. Die linke Seite in Abbildung 8.2 zeigt die Deklaration von Automaten-Variablen für das Beispiel aus Abbildung 4.1 auf Seite 72. Hierbei wird der Wurzelautomat durch die Automaten-Variable $\texttt{S}_1\texttt{\_CTRL}$ repräsentiert. Entsprechend beschreiben die Variablen $\texttt{S}_2\texttt{\_CTRL}$, $\texttt{S}_3\texttt{\_CTRL}$ und $\texttt{S}_4\texttt{\_CTRL}$ die Sequentiellen Automaten, die den Wurzelautomaten verfeinern.

Ein Ereignis wird durch eine boolesche Variable modelliert. Diese gibt an, ob das Ereignis aktuell anliegt oder nicht. In Abbildung 8.2 sind zur Modellierung der Ereignisse unseres Beispiels die Variablen $\texttt{En\_E}_1$ und $\texttt{En\_E}_2$ eingeführt.

Auf der rechten Seite in Abbildung 8.2 ist für jeden Zustand des Hierarchischen Automaten genau ein `In`-Prädikat definiert. Mit diesem wird modelliert, ob der Zustand aktuell aktiv ist oder nicht. Ein Zustand kann nur dann aktiv sein, wenn er im Zusammenspiel mit allen anderen gleichzeitig aktiven Zuständen eine wohlgeformte Konfiguration bildet. Um diese

```
VAR     S1_CTRL : {S1};        DEFINE  In_S1 := S1_CTRL=S1;
        S2_CTRL : {S5,S6};             In_S5 := In_S1 & S2_CTRL=S5;
        S3_CTRL : {S7,S8};             In_S6 := In_S1 & S2_CTRL=S6;
        S4_CTRL : {S9,S10};            In_S7 := In_S1 & S3_CTRL=S7;
        En_E1   : boolean;             In_S8 := In_S1 & S3_CTRL=S8;
        En_E2   : boolean;             In_S9 := ...; In_S10 := ...;
```

Abbildung 8.2: SMV-Repräsentation von Sequentiellen Automaten, Zuständen und Ereignissen

Eigenschaft zu garantieren, weisen die Prädikate untereinander Abhängigkeiten auf, mit denen die baumartige Struktur einer Konfiguration nachgebildet wird. So besteht beispielsweise eine Abhängigkeit zwischen den Prädikaten `In_S8` und `In_S1`, da der Zustand S_8 nur dann aktiv sein kann, wenn gleichzeitig der Oberzustand S_1 aktiv ist.

Abbildung 8.3 zeigt für unser Beispiel, wie das Verhalten der Transitionen eines Sequentiellen Automaten durch eine `TRANS`-Anweisung in der Eingabesprache von SMV repräsentiert werden kann. Zusätzlich zu den bereits eingeführten Variablen aus Abbildung 8.2 werden hier so genannte `Outgoing`- und `Active`-Prädikate verwendet, deren Bedeutung wir im Folgenden kurz erläutern.

Für jeden Zustand des Hierarchischen Automaten wird genau ein `Outgoing`-Prädikat eingeführt, mit dem überprüft wird, ob der Zustand verlassen werden kann. Das `Outgoing`-Prädikat eines Zustands S fordert einerseits, dass S aktiv ist und andererseits, dass eine der aus S herausführenden Transitionen aktiviert werden kann. Entsprechend garantiert das Prädikat `Outgoing_S7`, dass der Zustand S_7 aktiv ist und dass das Ereignis E_2 anliegt. Der Wurzelzustand S_1 kann hingegen niemals verlassen werden, was durch das Prädikat `Outgoing_S1` kodiert ist. Man beachte, dass die booleschen Konstanten *True* und *False* in SMV durch `1` und `0` repräsentiert sind.

Ähnlich zu den `Outgoing`-Prädikaten wird für jeden Sequentiellen Automaten genau ein `Active`-Prädikat eingeführt. Mit `Active`-Prädikaten kann man die Priorisierung von Transitionen (vgl. Definition 3.25) beschreiben. Die Semantik Hierarchischer Automaten legt fest, dass Transitionen, die auf höheren Hierarchiestufen operieren, gegenüber Transitionen, die auf niederen Hierarchiestufen operieren, priorisiert werden müssen. So beschreibt beispielsweise das `Active`-Prädikat `S3_CTRL_Active`, dass Transitionen innerhalb des Sequentiellen Automaten `S3_CTRL` nur dann ausgeführt werden dürfen, wenn in den direkten und indirekten Vorgängerzuständen von S_3 keine Transitionen aktiviert sind. Man beachte, dass die Negation in SMV durch `!` repräsentiert ist. Das `Active`-Prädikat `S1_CTRL_Active` des Wurzelautomaten ist trivialer Weise immer erfüllt, da für S_1 keine Vorgängerzustände existieren.

Nachdem wir `Outgoing`- und `Active`-Prädikate eingeführt haben, erläutern wir im Folgenden die `TRANS`-Anweisung aus Abbildung 8.3. Mit dieser Anweisung wird das Verhalten der Transitionen in `S3_CTRL` beschrieben. Zunächst wird durch die Bedingung `S3_CTRL_Aktiv` garantiert, dass nur dann Transitionen ausgeführt werden, wenn in den Vorgängerzuständen von S_3 keine Transitionen aktiviert sind. Ist dieses Prädikat erfüllt, so wird in einer Fallunterscheidung für jeden Zustand aus `S3_CTRL` überprüft, ob in ihm Transitionen schalten können. Man beachte hierbei, dass die Fallunterscheidung in SMV durch eine `case`-Anweisung repräsentiert ist. Die Überprüfung erfolgt mit Hilfe von `Outgoing`-Prädikaten. Sind Transi-

```
DEFINE  Outgoing_S1 := 0;
        Outgoing_S7 := In_S7 & En_E2;
        S1_CTRL_Active := 1;
        S3_CTRL_Active := S1_CTRL_Active & In_S1 & !Outgoing_S1;

TRANS   S3_CTRL_active -->
         case
           In_S7 :
             case
               Outgoing_S7 : (En_E2 & next(S3_CTRL) = S8);
               1           : next(S3_CTRL) = S7;
             esac;
           In_S8 : next(S3_CTRL) = S8;
         esac;
```

Abbildung 8.3: SMV-Repräsentation für die Transition eines Hierarchischen Automaten

tionen aktiviert, so wird deren Effekt durch ein Prädikat beschrieben. Ist beispielsweise das Prädikat `Outgoing_S7` erfüllt, so wird das Ereignis E_2 erzeugt und der Sequentielle Automat `S3_CTRL` in den Zustand S_8 überführt. Man beachte hierbei, dass der Nachzustand der Variablen `S3_CTRL` durch `next(S3_CTRL)` referenziert wird. Ist hingegen das Prädikat `Outgoing_S7` nicht erfüllt, so verbleibt der Sequentielle Automat im Zustand S_7.

Sind in einem Zustand eines Sequentiellen Automaten mehrere Transitionen gleichzeitig aktiviert, so liegt ein Nichtdeterminismus vor. In diesem Fall schlägt Erich Mikk vor, die Effekte aller aktivierten Transitionen durch Disjunktion miteinander zu verknüpfen. Um zu garantieren, dass die disjunktive Verknüpfung exklusiv ist, führt er zusätzlich für jeden Sequentiellen Automaten eine so genannte `Arbiter`-Variable ein. Diese `Arbiter`-Variable gibt an, welche Transition nichtdeterministisch ausgewählt wird. Die `Arbiter`-Variable eines Sequentiellen Automaten repräsentiert zu einem Zeitpunkt immer nur eine Transition. Unser Beispiel enthält keinen Nichtdeterminismus, so dass der Sinn einer `Arbiter`-Variable daran nicht illustriert werden kann. Für eine ausführliche Beschreibung zur Verwendung von `Arbiter`-Variablen verweisen sie wir auf die Arbeiten von Erich Mikk [Mik00].

Abbildung 8.4 zeigt, wie Ereignisse erzeugt und gelöscht werden können. Hierzu werden spezielle `Indicator`-Prädikate definiert, die angeben, ob eine aktivierte Transition ausgewählt wurde oder nicht. Um die Transitionen unseres Beispiels benennen zu können, vergeben wir eine aufsteigende Nummerierung entlang ihres Auftretens in Abbildung 4.1 auf Seite 72, wobei wir auf der linken Seite mit 1 beginnen. Im Folgenden erläutern wir den typischen Aufbau eines `Indicator`-Prädikates am Beispiel der Transition T_2 (vgl. Abbildung 8.4).

Das `Indicator`-Prädikat einer Transition garantiert zunächst, dass keine höher priorisierten Transitionen aktiviert sind. Deshalb enthält die Definition von `Indicator_T2` das `Active`-Prädikat `S2_CTRL_Active`. Weiterhin wird durch ein `Indicator`-Prädikat sichergestellt, dass die betreffende Transition aktiviert ist. Für das Prädikat `Indicator_T2` ist dies durch die Variable `Outgoing_S2` ausgedrückt. Schließlich beschreibt ein `Indicator`-Prädikat einen Zustandsübergang, in dem die Transition ausgeführt wird. Der Sequentielle Automat, in dem die Transition enthalten ist, muss sich also nach der Ausführung des Zustandsübergangs

```
DEFINE Indicator_T2 := S2_CTRL_Active & Outgoing_S2 & next(S2_CTRL) = S3;
       Indicator_T3 := S3_CTRL_Active & Outgoing_S4 & next(S3_CTRL) = S5;

ASSIGN next(En_E2)  := case
                          (Indicator_T2 | Indicator_T3) : 1;
                          1 : {0,1};
                       esac;
```

Abbildung 8.4: SMV-Repräsentation zum Generieren und Löschen von Ereignissen

im Zielzustand der Transition befinden. Im Falle von Nichtdeterminismen ist zusätzlich die Belegung der `Arbiter`-Variable zu berücksichtigen. In der Definition von `Indicator_T2` ist gefordert, dass der Nachzustand von `S2_CTRL` mit dem Zustand S_3 belegt ist.

In unserem Beispiel wird ein Ereignis E_2 immer dann erzeugt, wenn die Transition T_2 oder T_3 ausgeführt wird. Sind hingegen die zu den beiden Transitionen korrespondierenden `Indicator`-Prädikate nicht erfüllt, so kann das Ereignis nach einem Zustandsübergang auftreten oder nicht (vgl. mit dem zweiten Teil der `case`-Anweisung aus Abbildung 8.4). Diese Behandlung setzt eine so genannte *Open-System-Semantik* um (vgl. Definition 5.8 auf Seite 92), in der nach einem Zustandsübergang eine Menge von Ereignissen aus der Umgebung empfangen werden kann. Für eine *Closed-System-Semantik* würde entsprechend im zweiten Teil der `case`-Anweisung aus Abbildung 8.4 spezifiziert, dass das Ereignis E_2 nach einem Zustandsübergang nicht auftreten darf.

Wir haben in diesem Abschnitt beschrieben, wie Hierarchische Automaten in der Eingabesprache von SMV repräsentiert werden können. Dieses Konzept wurde von Erich Mikk entwickelt und unterstützt keine Datenräume. Im Folgenden stellen wir eine von uns entwickelte Erweiterung des Konzeptes um die Behandlung von Datenräumen vor. Für die Anbindung von SMV an Isabelle/HOL haben wir dieses erweiterte Konzept umgesetzt.

8.1.3 Erweiterung um Datenräume

Da Erich Mikk in seinen Arbeiten die Behandlung von Datenvariablen nicht unterstützt, erweitern wir sein Konzept um diesen Anteil [HK03b]. Hierbei ist insbesondere die semantische Interpretation von *Racing*-Effekten adäquat umzusetzen (vgl. Abschnitt 4.4 auf Seite 80).

Der Datenraum des zu abstrahierenden Hierarchischen Automaten kann aus beliebig vielen Partitionen bestehen, wobei eine Partition aus mehreren Variablen aufgebaut sein kann. Zur Deklaration von Variablen verwenden wir nur die Typen *Integer* und *Boolean*, wobei Integer-Variablen auf einen endlichen Definitionsbereich eingeschränkt sein müssen. Eine Erweiterung um andere Datentypen ist prinzipiell möglich, wenn sie sich durch die Typen der Eingabesprache von SMV repräsentieren lassen.

Um einen Datenraum in SMV zu repräsentieren, führen wir für jede Partition entsprechende Zustandsvariablen ein. Abbildung 8.5 zeigt die Deklaration von Zustandsvariablen für das Beispiel aus Abbildung 4.1. Der Datenraum besteht aus zwei Partitionen, die durch die Integer-Variablen x und y repräsentiert sind. Der Definitionsbereich der Variablen ist in diesem Beispiel auf das Intervall $0\,..\,255$ eingeschränkt. Wir nehmen an, dass x initial mit 8 und y mit 7 belegt sei. Die Ausführung von Transitionen führt durch die an den Transitio-

```
VAR    X : 0 .. 255;          ASSIGN    init(X) := 8;
       Y : 0 .. 255;                    init(Y) := 7;

TRANS  (Indicator_T1 & next(X) = Y) | (Indicator_T2 & next(X) = Y) |
       (! Indicator_T1 & ! Indicator_T2 & next(X) = X)

TRANS  (Indicator_T1 & next(Y) = 4) | (Indicator_T3 & next(Y) = 1) |
       (! Indicator_T1 & ! Indicator_T3 & next(Y) = Y)
```

Abbildung 8.5: SMV-Repräsentation von partitionierten Datenräumen

nen befindlichen Update-Funktionen zu Effekten auf dem Datenraum. `Trans`-Anweisungen, mit denen diese Effekte beschrieben werden können, zeigt der untere Teil in Abbildung 8.5. Hierbei werden die Datenvariablen separat voneinander behandelt, indem für jede Datenvariable eine `Trans`-Anweisung eingeführt wird. Mit einer `Trans`-Anweisung spezifizieren wir alle möglichen Effekte, die auf der behandelten Datenvariable entstehen können, und verknüpfen diese durch eine Disjunktion. Ein Effekt entsteht immer dann, wenn eine Transition ausgeführt wird, deren Update-Funktion die zu behandelnde Datenvariable beschreibt. Ob eine Transition ausgeführt wird, kann mit dem korrespondierenden `Indicator`-Prädikat überprüft werden. Durch Ausführung der Transition T_1 wird beispielsweise x mit dem Wert belegt, den y vor Ausführung von T_1 hatte. Sollte während eines Zustandsübergangs keine der Transitionen die Datenvariable beschreiben, so wird der zuvor gültige Datenzustand wiederhergestellt. Um diese Bedingung zu beschreiben, werden zunächst die Indikatorprädikate von allen auf die Datenvariable schreibenden Transitionen bestimmt, diese anschließend negiert und schließlich durch Konjunktion miteinander verknüpft. So wird auf x die zuvor gültige Datenbelegung wiederhergestellt, wenn in einem Zustandsübergang weder die Transition T_1 noch die Transition T_2 ausgeführt wird.

Die angegebene Modellierung zur Beschreibung von Effekten auf dem Datenraum setzt die Semantik von schreibenden Update-Funktionen aus Abschnitt 4.1 nicht exakt um. Allerdings wird durch die Modellierung kein Verhalten reduziert, sondern lediglich neues Verhalten hinzugefügt. Zum besseren Verständnis betrachten wir nochmal das Beispiel aus Abbildung 4.1 auf Seite 72 und erweitern in Analogie zu Abschnitt 4.4 die Update-Funktion der Transition T_2 um die Zuweisung $y := 4$ und die Update-Funktion der Transition T_3 um die Zuweisung $x := 3$. Wie wir in Abschnitt 4.4 ausführlich beschreiben, führt diese Erweiterung in der Semantik Hierarchischer Automaten zu keinen neuen Effekten auf dem Datenraum. Der Grund dafür ist, dass die Ausführung von Update-Funktion auf dem Datenraum atomar ist, d. h. eine Update-Funktion gewinnt einen Schreibvorgang gegenüber konkurrierenden Update-Funktionen immer für alle Partitionen, die sie beschreiben möchte. Die von uns in SMV umgesetzte Semantik verletzt diese Eigenschaft und erzeugt für die angegebene Erweiterung des Beispiels im Folgestatus zwei neue Datenraumbelegungen. So kann beispielsweise der Effekt auftreten, dass nach Ausführung der Transitionen x mit 3 und y mit 4 belegt ist. Hierbei gewinnt einerseits zwar T_2 den Schreibvorgang gegen T_3 für x. Andererseits verliert T_2 aber gleichzeitig den Schreibvorgang gegen T_3 für y. Dies widerspricht der Statemate-Semantik von Update-Funktionen und deshalb handelt es sich bei dem von uns implementierten Überset-

zungsalgorithmus um eine Überapproximation des Verhaltens. Folglich ist die Anwendbarkeit der Taktik gegenwärtig auf das universale Fragment der CTL beschränkt. Im Rahmen dieser Arbeit ist diese Einschränkung aber unproblematisch, da wir beabsichtigen, die Taktik in Kombination mit dem implementierten Abstraktionsalgorithmus zu verwenden (vgl. Abschnitt 8.2). Das Abstraktionskonzept ist aber ohnehin auf das universale Fragment der CTL eingeschränkt.

Eine Erweiterung des Übersetzungsalgorithmus zur präzisen Beschreibung Hierarchischer Automaten in SMV ist möglich und für die nächste Ausbaustufe der Taktik geplant. Hierzu werden wir die in der Dissertation von Robert Büssow vorgeschlagene Behandlung von Datenvariablen aufgreifen [Büs03]. Robert Büssow entwickelte in seiner Arbeit ein Konzept zum Model-Checking von μSZ-Spezifikationen [BGGK97], das durch eine Anbindung an den Model Checker SMV realisiert wurde. Dieses Konzept beinhaltet auch die Behandlung von konkurrierenden Schreibvorgängen auf partitionierte Datenräume. Die Kernidee besteht darin, für die Datenvariablen zusätzlich so genannte `Lock`-Variablen einzuführen, mit denen ein kontrollierter Schreibzugriff erfolgen kann. Durch Optimierungsstrategien werden die Variablen identifiziert, bei denen ein Schreibkonflikt auftreten kann. Dadurch kann eine unnötige Vergrößerung des Zustandsraums vermieden werden.

8.2 Automatisierte Prädikatenabstraktion

In diesem Abschnitt beschreiben wir eine Isabelle-Taktik, die durch den Einsatz einer Prädikatenabstraktion die Analyse von - semantisch gesehen – unendlich großen Statecharts-Spezifikationen mit dem Model-Checker SMV ermöglicht. Zunächst geben wir einen intuitiven Eindruck von der Arbeitsweise der Taktik. Im zweiten Teil dieses Abschnitts stellen wir eine Technik vor, mit der die Überapproximation von atomaren Prädikaten algorithmisch ermittelt werden kann. Danach beschreiben wir die Kernidee unseres Algorithmus und gehen dabei insbesondere auf die algorithmische Berechnung der Überapproximationen für Guards und Update-Funktionen ein. Abschließend geben wir an, welche Ausgaben durch die Abstraktionstaktik erzeugt werden.

8.2.1 Taktik zur Anbindung eines Abstraktionsalgorithmus

Im letzten Abschnitt haben wir eine Isabelle-Taktik zur effizienten Verifikation von Statecharts-Spezifikationen mit Hilfe des Model-Checkers SMV vorgestellt. Bei ihrer Benutzung können folgende Probleme auftreten.

1. Die Taktik ist nicht anwendbar, weil die zu überprüfende Statecharts-Spezifikation auf einem unendlichen Datenraum definiert ist.

2. Die Taktik terminiert nicht, weil die zu überprüfende Statecharts-Spezifikation zwar auf einem endlichen, aber zu großen Datenraum definiert ist, so dass der Model-Checker SMV kein Ergebnis in akzeptabler Zeit zurückliefern kann.

Um diese Probleme zu lösen, haben wir in Kapitel 6 ein Vorgehen angegeben, mit dem der Datenraum einer Statecharts-Spezifikation vor dem Einsatz eines Model-Checkers abstrahiert werden kann. Wir haben begründet, warum das dort vorgeschlagene Abstraktionskonzept bezüglich des universalen Fragments der CTL eigenschaftserhaltend ist. Ferner ist das Konzept

in Kapitel 7 dieser Ausarbeitung in einer Isabelle/HOL-Theorie umgesetzt. Wendet ein Isabelle-Benutzer die in dieser Theorie definierten Konstruktionsoperatoren an, so erhält er folgendes Beweisziel.

$$(A\ AbsBy^{+}_{\mathrm{HA}}\ R) \vDash_{\mathrm{HA}} (F\ AbsBy^{-}_{\mathrm{CTL}}\ R)$$

Hierbei ist A ein Hierarchischer Automat, der die zu analysierende Statecharts-Spezifikation repräsentiert. Weiterhin beschreibt F eine Formel des universalen Fragments der CTL und R eine Abstraktionsfunktion, mit der der Datenraum von A abstrakt interpretiert werden kann. Zu überprüfen ist, ob die mit R konstruierte Unterapproximation von F in der mit R konstruierten Überapproximation von A erfüllt ist. Die Definitionen für die Konstruktionsoperatoren $AbsBy^{+}_{\mathrm{HA}}$ und $AbsBy^{-}_{\mathrm{CTL}}$ sind auf den Seiten 127 und 128 dieser Ausarbeitung zu finden. Sollte das abgesetzte Beweisziel gelten, so folgt daraus, dass F in A gilt, da die eingesetzte Konstruktion eigenschaftserhaltend ist.

Wir nehmen an, dass der abstrakte Datenraum, in den R abbildet, endlich ist. Folglich besitzt auch die Überapproximation von A einen endlichen Datenraum. Obwohl damit die zentrale Voraussetzung zum Model-Checking erfüllt ist, kann die im letzten Abschnitt vorgestellte Taktik nicht direkt auf das Beweisziel angewendet werden. Der Grund dafür ist, dass Teile in der Konstruktion, wie zum Beispiel die Konstruktionen für Guards und Update-Funktionen, mit Hilfe von Existenzquantoren gebildet sind. Für diese Teile ist aber eine direkte Übersetzung in die Eingabesprache von SMV nicht möglich.

Durch eine gezielte Vereinfachung der Konstruktionsoperatoren in logisch äquivalente Ausdrücke ist das Beweisziel solange umzuformen, bis wir einen quantorenfreien Ausdruck erhalten, der nach SMV übersetzt werden kann. Dieser Beweisprozess ist im Allgemeinen interaktiv zu führen und nur schwer automatisierbar, da insbesondere das automatische Eliminieren von Existenzquantoren in einem Theorembeweiser als schwierig einzuschätzen ist.

Als Alternative zur schrittweisen Vereinfachung der Konstruktionsoperatoren bieten wir dem Isabelle-Benutzer eine Taktik an, die das in Kapitel 6 vorgeschlagene Abstraktionskonzept außerhalb der Isabelle-Logik durch einen ML-Algorithmus umsetzt. Mit diesem Abstraktionsalgorithmus kann für einen Hierarchischen Automaten die Überapproximation und für eine $\forall$ CTL-Formel die Unterapproximation konstruiert werden. In Analogie zu den Konstruktionsoperatoren werden die Ergebnisse der Abstraktion wieder als Hierarchische Automaten bzw. als $\forall$ CTL-Formeln ausgedrückt. Im Gegensatz zu der allgemeineren Definition der Konstruktionsoperatoren ist der Algorithmus auf eine Prädikatenabstraktion beschränkt. Folglich ist die Taktik nur dann anwendbar, wenn die Abstraktionsfunktion R auf eine Prädikatenabstraktion zugeschnitten ist. Dazu ist zunächst der Datenraum von A durch geeignete Prädikate zu beschreiben. Anschließend wird die Abstraktionsfunktion so konstruiert, dass jedes dieser Prädikate durch eine boolesche Variable im abstrakten Datenraum repräsentiert ist. Ein einführendes Beispiel für die Prädikatenabstraktion haben wir zu Beginn des Kapitels 6 auf der Seite 106 vorgestellt.

Im Rahmen der Diplomarbeit von Klemens Kanal entstand ein Prototyp für diese Taktik [Kan05], den wir in weiterführenden Arbeiten verfeinert haben. Dieser Prototyp setzt das in Kapitel 6 angegebene Abstraktionskonzept für Statecharts algorithmisch um. Im folgenden Abschnitt erläutern wir, wie die Überapproximation eines Prädikates algorithmisch errechnet werden kann. Der umgesetzte Ansatz basiert auf Arbeiten von Hassen Saïdi und Natarajan

Shankar [SS99], die eine ähnliche Implementierung zur Abstraktion von Transitionssystemen in dem Beweiser PVS [ORS92] entwickelten.

8.2.2 Algorithmus zur Überapproximation von Prädikaten

In diesem Abschnitt erläutern wir, wie die Überapproximation eines Prädikates algorithmisch berechnet werden kann. Wir weichen von der in den Konstruktionsoperatoren zur Abstraktion verwendeten Definition ab und wenden ein von Hassen Saïdi und Natarajan Shankar entwickeltes Verfahren an, da es sich besser für eine algorithmische Umsetzung eignet.

Dieses Verfahren beruht auf alternativen Definitionen für die in Abbildung 6.2 auf Seite 104 eingeführten Prädikatentransformer α^+ und α^-. In Abschnitt 6.1 haben wir die Definitionen auf Basis der strengsten Nachbedingung bzw. schwächsten Vorbedingung aus einer Retrieve-Relation abgeleitet. Im Gegensatz dazu verwenden wir für den Algorithmus Definitionen, die auf den Adjunktionstheoremen einer Galois-Korrespondenz aufbauen. Adjunktionstheoreme haben wir auf Seite 100 dieser Ausarbeitung bereits eingeführt. Auf Basis dieser Theoreme lassen sich α^+ und α^- folgendermaßen, unter Verwendung von Supremum und Infimum, definieren.

$$\alpha^+ P_C \equiv_{df} \bigwedge \{ P_A \mid P_C \Rightarrow_P \gamma P_A \}$$
$$\alpha^- P_C \equiv_{df} \bigvee \{ P_A \mid \gamma P_A \Rightarrow_P P_C \}$$

Hierbei sei P_C ein Prädikat vom Typ δ_c *pred* und P_A ein Prädikat vom Typ δ_A *pred*. Für den Abstraktionsalgorithmus haben wir uns auf eine Implementierung für α^+ beschränkt, da der duale Zusammenhang zwischen den beiden Prädikatentransformern es uns ermöglicht, α^- mit Hilfe von α^+ zu beschreiben (vgl. Abschnitt 6.1).

Um die Überapproximation eines Prädikates P_C auf Basis der Definition von α^+ in einem Algorithmus errechnen zu können, ist zunächst für alle P_A des Typs δ_A *pred* die Gültigkeit folgender Aussage zu prüfen.

$$P_C \Rightarrow_P \gamma P_A \tag{8.1}$$

Anschließend werden alle die Prädikate durch Konjunktion miteinander verknüpft, für die die Prüfung erfolgreich war. Folgende Schwierigkeiten sind für die algorithmische Umsetzung dieses Vorgehens zu lösen.

1. Es wird eine Implementierung des Prädikatentransformers γ benötigt.
2. Von den zu prüfenden P_A des Typs δ_A *pred* muss eine qualifizierte Auswahl getroffen werden, um den Aufwand vertretbar zu halten.

Im Folgenden erläutern wir, wie diese Probleme gelöst werden können.

Konkretisierung durch Substitution

Da es sich bei unserem Algorithmus um eine Prädikatenabstraktion handelt, ist eine Implementierung für γ leicht anzugeben. Der Grund dafür ist, dass bei einer Prädikatenabstraktion

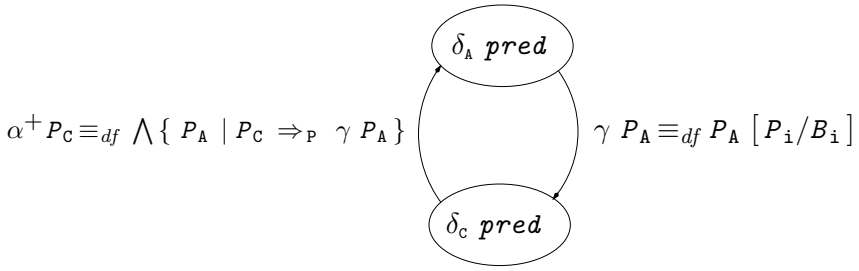


Abbildung 8.6: Galois-Korrespondenz mit Adjunktionstheorem

der abstrakte Datenraum aus einer endlichen Anzahl von booleschen Variablen zusammengesetzt ist. Jede dieser Variablen ist durch ein Prädikat auf dem konkreten Datenraum interpretiert. Entsprechend ist γ eine Substitution, die alle in einem Prädikat vom Typ δ_A *pred* vorkommenden booleschen Variablen durch ihre definierenden Prädikate vom Typ δ_C *pred* ersetzt.

Abbildung 8.6 zeigt eine Galois-Korrespondenz. Durch die Funktion α^+ auf der linken Seite der Abbildung kann ein Prädikat P_C über dem konkreten Datenraum δ_C in ein Prädikat über dem abstrakten Datenraum δ_A transformiert werden. Mit Hilfe der Substitutionsfunktion γ ist entsprechend die Konkretisierung eines Prädikates P_A über δ_A in ein Prädikat über δ_C möglich. Angenommen hierbei enthält P_A maximal k boolesche Variablen. Dann gibt der Ausdruck $P_A [P_i/B_i]$ für $i : 1..k$ an, dass in P_A jedes vorkommende B_i durch das korrespondierende Prädikat P_i zu ersetzen ist.

Im einführenden Beispiel des Abstraktionskapitels 6 haben wir das Substitutionsverfahren bereits eingesetzt. Dies war notwendig, um die Rückübersetzung des abstrakten Modells – dargestellt in Abbildung 6.3 auf Seite 106 – zu realisieren. Im Folgenden wollen wir das Vorgehen für die Rückübersetzung eines Guards an diesem Beispiel erläutern. Wir wählen hierfür den Guard an der selbstbezüglichen Transition des Zustands S_1 und transformieren diesen in zwei Schritten zu einem Prädikat über dem konkreten Datenraum.

$$\begin{aligned} \gamma\ (B_1 \wedge B_2) &\Leftrightarrow (B_1 \wedge B_2)\ [x > 4/B_1, x \leq 6/B_2\,] \\ &\Leftrightarrow x > 4 \wedge x \leq 6 \\ &\Leftrightarrow x \in \{5, 6\} \end{aligned}$$

Im ersten Schritt der Transformation wird B_1 durch das Prädikat $x > 4$ und B_2 durch das Prädikat $x \leq 6$ ersetzt. Im zweiten Schritt wird eine Simplifizierung vorgenommen, die zu dem Prädikat $x \in \{5, 6\}$ führt.

Das Beispiel zeigt deutlich, dass die Implementierung der Substitutionsfunktion γ leicht zu realisieren ist. Um die Überapproximation eines Prädikates angeben zu können, ist damit eine wichtige Voraussetzung für die Implementierung von α^+ erfüllt. Wir benötigen aber zusätzlich eine Strategie, mit der die durch den Typ δ_A *pred* beschriebene Formelmenge geeignet reduziert werden kann. Im Folgenden wird diese Strategie vorgestellt.

Qualifizierte Auswahl von zu prüfenden Formeln

Problematisch für eine praktische Umsetzung ist, dass Beweisverpflichtung 8.1 für alle P_A des Typs δ_A *pred* auszuwerten ist. Der Typ δ_A *pred* beschreibt hierbei eine Menge von ato-

Normalform(2^{2^k})	
$B_1 \wedge B_2$	$B_1 \vee (\neg\ B_1 \wedge B_2)$
$\neg\ B_1 \wedge B_2$	$B_1 \vee (\neg\ B_1 \wedge \neg\ B_2)$
$\neg\ B_1 \wedge \neg\ B_2$	$B_2 \vee (\neg\ B_1 \wedge \neg\ B_2)$
$B_1 \wedge \neg\ B_2$	$B_2 \vee (B_1 \wedge \neg\ B_2)$
$B_1 \vee B_2$	$(B_1 \wedge \neg\ B_2) \vee (\neg\ B_1 \wedge B_2)$
$\neg\ B_1 \vee B_2$	$(B_1 \wedge B_2) \vee (\neg\ B_1 \wedge \neg\ B_2)$
$\neg\ B_1 \vee \neg\ B_2$	$(B_1 \wedge \neg\ B_2) \vee (\neg\ B_1 \wedge \neg\ B_2)$
$B_1 \vee \neg\ B_2$	$(\neg\ B_1 \wedge B_2) \vee (\neg\ B_1 \wedge \neg\ B_2)$

Approximation$(3^k - 1)$	
$B_1 \vee B_2$	B_1
$\neg\ B_1 \vee B_2$	B_2
$\neg\ B_1 \vee \neg\ B_2$	$\neg\ B_1$
$B_1 \vee \neg\ B_2$	$\neg\ B_2$

Abbildung 8.7: Reduktion der Formelmenge für zwei boolesche Variablen

maren Formeln der Aussagenlogik. Entsprechend kann eine Formel dieses Typs in einer Normalform mit Disjunktion, Konjunktion und Negation beschrieben werden. Angenommen der abstrakte Datenraum $\delta_\mathbb{A}$ besteht aus k booleschen Variablen, so lassen sich basierend auf der Disjunktiven Normalform, 2^{2^k} verschiedene Formeln konstruieren. Um eine möglichst präzise Approximation zu errechnen, betrachten wir hierbei nur Formeln, in denen jeweils alle booleschen Variablen vorkommen. Die linke Seite in Abbildung 8.7 zeigt beispielhaft, wie diese Formelmenge für zwei boolesche Variablen B_1 und B_2 aussieht.

In einem Algorithmus wäre also die Beweisverpflichtung 8.1 für 2^{2^k} verschiedene Formeln auszuwerten. Um diesen hohen Rechenaufwand zu reduzieren, haben Hassen Saïdi und Natarajan Shankar so genannte Repräsentanten festgelegt. Repräsentanten sind Disjunktionen, die sich über den Literalen der booleschen Variablen bilden lassen, wobei auch Formeln betrachtet werden, in denen nicht alle booleschen Variablen vorkommen. Durch dieses Vorgehen ist der Beweisaufwand auf die Prüfung von $3^k - 1$ Beweisverpflichtungen beschränkt. Die rechte Seite in Abbildung 8.7 zeigt für das Beispiel die aus Repräsentanten bestehende Formelmenge.

Hassen Saïdi und Natarajan Shankar haben einen Beweis dafür angegeben, dass diese reduzierte Formelmenge ausreicht, um eine präzise Überapproximation zu errechnen [SS99]. Da das Lösen von $3^k - 1$ verschiedenen Beweisverpflichtungen bei entsprechend groß gewähltem k schnell an die Grenze der Praktikabilität stößt, schlagen sie einen Algorithmus vor, mit dem auch allgemeinere Überapproximationen errechnet werden können. Hierfür wird mit der allgemeinsten Überapproximation – repräsentiert durch das Prädikat `true` – begonnen und durch schrittweises Auswerten der Formelmenge ein Prädikat berechnet, mit dem man sich an die präziseste Überapproximation annähert. Der Algorithmus ist in dem Sinne skalierbar, dass er für eine festzulegende Schranke die Berechnung abbricht. Das bis zu diesem Zeitpunkt errechnete Zwischenergebnis wird als Überapproximation für das zu abstrahierende Prädikat zurückgegeben.

Wir haben bei der Implementierung unserer Taktik die vorgestellte Strategie zur Überapproximation von atomaren Prädikaten eingesetzt. Um jeweils die anfallende Beweisverpflichtung 8.1 für eine Formel auszuwerten, verwenden wir den SVC-Beweiser [Dil].

Im Folgenden geben wir einen Eindruck von den Besonderheiten unserer Implementierung zur Überapproximation Hierarchischer Automaten.

8.2.3 Algorithmus zur Überapproximation Hierarchischer Automaten

Der Algorithmus setzt das in Abschnitt 6.3.3 vorgeschlagene Abstraktionskonzept um. Dies bedeutet, dass er zunächst den zu abstrahierenden Hierarchischen Automaten in seine ihn definierenden Sequentiellen Automaten zerlegt. Danach wird für jeden Sequentiellen Automaten separat die Überapproximation errechnet. Abschließend werden die errechneten Überapproximationen wieder zu einem Hierarchischen Automaten zusammengefügt.

Damit ist der Aufbau des Algorithmus sehr ähnlich zur Struktur der Konstruktionsoperatoren aus Kapitel 7. Der zentrale Unterschied ist jedoch die Art und Weise, mit der die Überapproximationen von Guards und Update-Funktionen errechnet werden. Wir verwenden hierzu die im letzten Abschnitt vorgestellte Technik zur Überapproximation von atomaren Prädikaten. Im Folgenden werden wir am Beispiel aus Abbildung 6.3 auf Seite 106 erläutern, wie die Überapproximation sowohl für einen Guard, als auch für eine Update-Funktion errechnet werden kann.

Überapproximation von Guards

Angenommen es ist der Guard $x > 5$ algorithmisch zu abstrahieren. Wir verwenden zur Abstraktion die in Abbildung 6.3 angegebene Abstraktionsfunktion, mit der x durch die zwei booleschen Variablen B_1 und B_2 im abstrakten Datenraum interpretiert wird. Die Variablen B_1 und B_2 repräsentieren die atomaren Aussagen $x > 4$, und $x \leq 6$.

Um den Guard zu abstrahieren, wird für jede Formel P_A, die auf der rechten Seite in Abbildung 8.7 angegeben ist, die Gültigkeit der Beweisverpflichtung 8.1 geprüft. Hierbei ist P_C entsprechend mit $x > 5$ festzulegen. Wir erhalten folgende Beweisverpflichtungen mit den durch den automatischen Beweiser SVC bestimmten Ergebnissen.

$$\begin{aligned}
x > 5 \;\Rightarrow\; \gamma\, B_1 \quad & \surd \\
x > 5 \;\Rightarrow\; \gamma\, B_2 \quad & - \\
x > 5 \;\Rightarrow\; \gamma\, (\neg\, B_1) \quad & - \\
x > 5 \;\Rightarrow\; \gamma\, (\neg\, B_2) \quad & \surd \\
x > 5 \;\Rightarrow\; \gamma\, (B_1 \vee B_2) \quad & \surd \\
x > 5 \;\Rightarrow\; \gamma\, (\neg\, B_1 \vee B_2) \quad & - \\
x > 5 \;\Rightarrow\; \gamma\, (B_1 \vee \neg\, B_2) \quad & \surd \\
x > 5 \;\Rightarrow\; \gamma\, (\neg\, B_1 \vee \neg\, B_2) \quad & \surd
\end{aligned}$$

Alle als gültig markierten Formeln werden anschließend durch konjunktive Verknüpfung zu folgendem bereits simplifiziertem Prädikat zusammengefasst.

$$\begin{aligned}
\alpha\,(x > 5) \;&\Leftrightarrow\; \bigwedge \{\, B_1,\, \neg\, B_2,\, B_1 \vee \neg\, B_2,\, \neg\, B_1 \vee \neg\, B_2 \,\} \\
&\Leftrightarrow\; B_1 \wedge \neg\, B_2 \\
&\Leftrightarrow\; B_1
\end{aligned}$$

Wird ferner ausgenutzt, dass $\neg\, B_2$ aus B_1 folgt, so kann die Überapproximation für das Prädikat $x > 5$ noch weiter zu B_1 vereinfacht werden.

Klemens Kanal hat die angegebene Strategie in einem Algorithmus umgesetzt, mit dem die Überapproximation eines Guards automatisch errechnet werden kann [Kan05]. Der Algorithmus ist in dem Sinne optimiert, dass er nur die Disjunktionen mit dem SVC überprüft, die nicht durch bereits errechnete Ergebnisse subsumiert werden können. So ist in unserem Beispiel durch die erfolgreiche Überprüfung der Formel B_1 ein zusätzlicher Test für $B_1 \vee \neg\ B_2$ unnötig. Für eine ausführliche Beschreibung des Algorithmus verweisen wir auf die Diplomarbeit von Klemens Kanal [Kan05].

Überapproximation von Update-Funktionen

Zur Errechnung der Überapproximation einer Update-Funktion wird diese zunächst als binäres Prädikat über dem Vor- und Nachzustand des konkreten Datenraumes interpretiert. Anschließend errechnen wir – analog zum Errechnen der Überapproximation eines Guards (vgl. Abschnitt 8.2.2) – die Überapproximation der Nachbedingung dieses binären Prädikates. Wir erhalten so ein Prädikat über dem abstrakten Nachzustand, das im Allgemeinen mehr als eine abstrakte Datenraumbelegung beschreibt. Da die Update-Funktionen des zu errechnenden abstrakten Statecharts den Nachzustand auf dem abstrakten Datenraum eindeutig festlegen müssen, bilden wir auf der errechneten Überapproximation die kanonische disjunktive Normalform. Jedes der im Ergebnis entstehenden Disjunktionsglieder wird als eine abstrakte Update-Funktion interpretiert. Damit errechnet der von uns konzipierte Algorithmus nullstellige Update-Funktionen, deren Effekt konstant ist und nicht aus einer Datenraumbelegung des Vorzustandes errechnet wird.

Wir erläutern das Vorgehen am Beispiel für die Update-Funktion $x := x + 1$ aus Abbildung 6.3 auf Seite 106. Der Vorzustand sei im Folgenden durch ungestrichene und der Nachzustand durch gestrichene Variablen identifiziert. Ferner wird für den Vorzustand die Überapproximation des Guards der entsprechenden Transition angenommen, da nur unter dieser Voraussetzung die Update-Funktion überhaupt zur Anwendung kommen kann. In unserem Beispiel ist die Überapproximation des Guards durch B_1 beschrieben (vgl. letzter Abschnitt zur Überapproximation von Guards). Wenden wir die Konkretisierungsfunktion γ auf diesen Guard an, so erhalten wir das Prädikat $x > 4$, das wir im Folgenden für den Vorzustand der Update-Funktion annehmen.

Wir erhalten folgende mit dem SVC geprüften Ergebnisse.

$$
\begin{aligned}
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ B_1' &\quad \surd \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ B_2' &\quad - \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ (\neg\ B_1') &\quad - \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ (\neg\ B_2') &\quad - \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ (B_1' \vee B_2') &\quad \surd \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ (\neg\ B_1' \vee B_2') &\quad - \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ (B_1' \vee \neg\ B_2') &\quad - \\
\gamma\ B_1 \wedge x' = x + 1 \ \Rightarrow\ \gamma\ (\neg\ B_1' \vee \neg\ B_2') &\quad -
\end{aligned}
$$

Im Folgenden bilden wir die Konjunktion aller erfolgreich geprüften Formeln und anschließend die kanonische disjunktive Normalform.

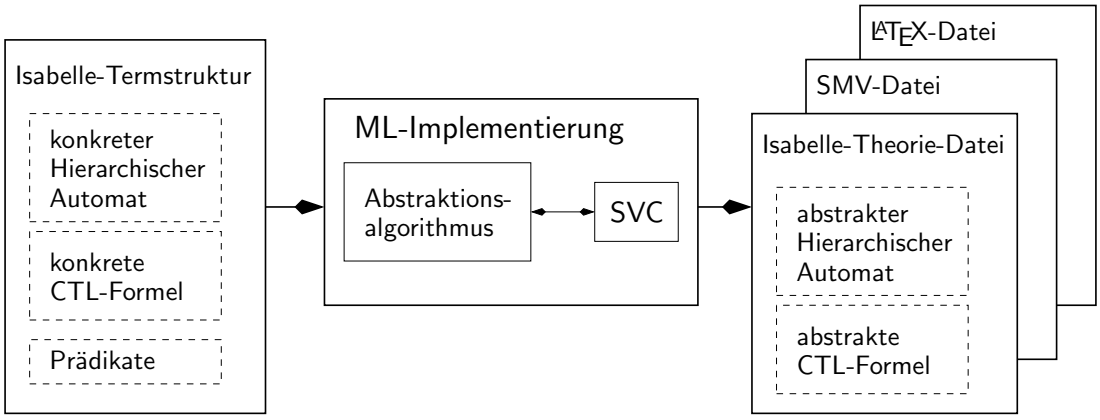


Abbildung 8.8: Ein- und Ausgaben des Abstraktionsalgorithmus

$$\alpha\ (\exists\, x\,.\ x' = x + 1 \wedge \gamma\ B_1) \Leftrightarrow \bigwedge \{\, B_1',\ B_1' \vee B_2' \,\}$$
$$\Leftrightarrow\ (B_1' \wedge B_2') \vee (B_1' \wedge \neg\ B_2')$$

Das Ergebnis ist durch zwei Disjunktionsglieder repräsentiert, die wir als Update-Funktionen auf dem abstrakten Datenraum interpretieren. Die Notation von Statecharts ist so konzipiert, dass eine Transition nur durch genau eine Update-Funktion markiert werden kann. Um trotzdem beide Update-Funktionen repräsentieren zu können, duplizieren wir die Transition (vgl. Abbildung 6.3 auf Seite 106).

Übersicht über die Taktik

Abbildung 8.8 stellt schematisch die Ein- und Ausgaben des Algorithmus dar. Eingelesen werden durch die ML-Implementierung ein Hierarchischer Automat, eine Formel des universalen Fragment der CTL und eine Menge von Prädikaten. Die einzulesenden Prädikate definieren hierbei die Abstraktionsfunktion. Der Algorithmus konstruiert auf Basis dieser Abstraktionsfunktion die abstrakten Gegenstücke für den Hierarchischen Automaten und für die CTL-Formel. Dabei wird der SVC-Beweiser genutzt, um die Gültigkeit einfacher atomarer Aussagen zu überprüfen. Schließlich werden die Ergebnisse der Abstraktion in drei verschiedenen Repräsentationen vom Algorithmus zurückgegeben.

1. Der abstrakte Hierarchische Automat wird auf Basis der in Teil I dieser Ausarbeitung dargestellten Formalisierung in einer Theorie-Datei für Isabelle/HOL ausgedrückt. Diese Datei kann in Isabelle eingelesen werden und bildet damit die Grundlage für mechanische Beweise über dem abstrakten Hierarchischen Automaten. Will sich beispielsweise ein Benutzer nicht auf die Korrektheit des Abstraktionsalgorithmus verlassen, so kann er auf Basis dieser Datei in Isabelle explizit nachweisen, dass ein abstrakter Hierarchischer Automat bezüglich eines anderen Hierarchischen Automaten für das universale Fragment der CTL eigenschaftserhaltend ist.

2. Der abstrakte Hierarchische Automat und die abstrakte CTL-Formel werden in einer Eingabe-Datei für SMV auf Basis des in Abschnitt 8.1 vorgestellten Übersetzungsschemas dargestellt. Diese Datei dient dazu, mit Hilfe des Model-Checkers SMV zu überprüfen, ob die abstrakte CTL-Formel in dem abstrakten Hierarchischen Automaten erfüllt ist.

3. Die Sequentiellen Automaten des abstrakten Hierarchischen Automaten werden in einer Eingabe-Datei für LaTeX repräsentiert. Dadurch ist es dem Benutzer unserer Taktik möglich, einen graphischen Eindruck von dem auf Basis der von ihm angegebenen Prädikate erzeugten abstrakten Hierarchischen Automaten zu bekommen.

Im folgenden Abschnitt zeigen wir an einer Fallstudie die Praktikabilität der vorgestellten Abstraktionstechnik.

8.3 Fallstudie: Kühlungssystem im Nuklearkraftwerk

In diesem Abschnitt führen wir zunächst eine Beispielspezifikation ein, mit der das Verhalten eines sicherheitskritischen reaktiven Kühlungssystems aus einem nuklearen Kraftwerk beschrieben wird. Danach führen wir automatisierte Analysen auf dieser Beispielspezifikation durch. Wir setzen hierzu die in diesem Kapitel vorgestellten Taktiken ein.

8.3.1 Statecharts-Spezifikation der Steuerungssoftware

In Abbildung 8.9 wird der Aufbau eines Kühlungssystems skizziert. In Abhängigkeit von den Druckverhältnissen innerhalb des Kühlungskreislaufs sind von der Steuerungssoftware die Ventile des Systems zu öffnen oder zu schließen.

Wenn beispielsweise der Wasserdruck im Kühlungssystem unter einen definierten Grenzwert fällt, ist von der Steuerungssoftware ein Ereignis zu generieren, das eine Druckerhöhung durch die Öffnung des oberen Ventils in Abbildung 8.9 bewirkt. Dadurch gelangt neue Kühlflüssigkeit in das Kühlungssystem. In Analogie dazu kann das untere Steuerungsventil ebenfalls durch ein Signal aktiviert werden, um bei einem Überdruck im Kühlungskreislauf Dampf abzulassen.

In der Literatur ist eine erste Beschreibung dieser Anwendung von David Parnas [CP93] zu finden. In der dort angegebenen Problembeschreibung ist die Funktionalität des Kühlungssystems auf das Hinzufügen von Kühlflüssigkeit (*Safety Injection*) beschränkt. Einige Jahre später wurde mit Hilfe von SCR-Spezifikationen (*Software Cost Reduction*) diese Beschreibung präzisiert und erweitert [BH97, BGL98]. Eine zentrale Erweiterung von Bultan et al. [BGL98] besteht darin, bei einem zu hohen Wasserdruck im Kühlungssystem den Druck durch Ablassen von Dampf zu reduzieren.

Auf Basis der Arbeiten aus der Literatur haben wir die Statecharts-Spezifikation in Abbildung 8.10 entwickelt. Dabei haben wir darauf geachtet, Ausdrucksmittel von Statecharts zu verwenden, die unter dem Blickwinkel einer Datenabstraktionstechnik konzeptuell wichtig sind. Die Spezifikation besteht aus drei parallel komponierten Zuständen, deren Verhalten im Folgenden nacheinander beschrieben werden soll. Man beachte, dass der Datenraum der Statecharts-Spezifikation – obwohl nicht explizit in Abbildung 8.10 angegeben – aus der Variablen *pressure* vom Typ *Integer* besteht.

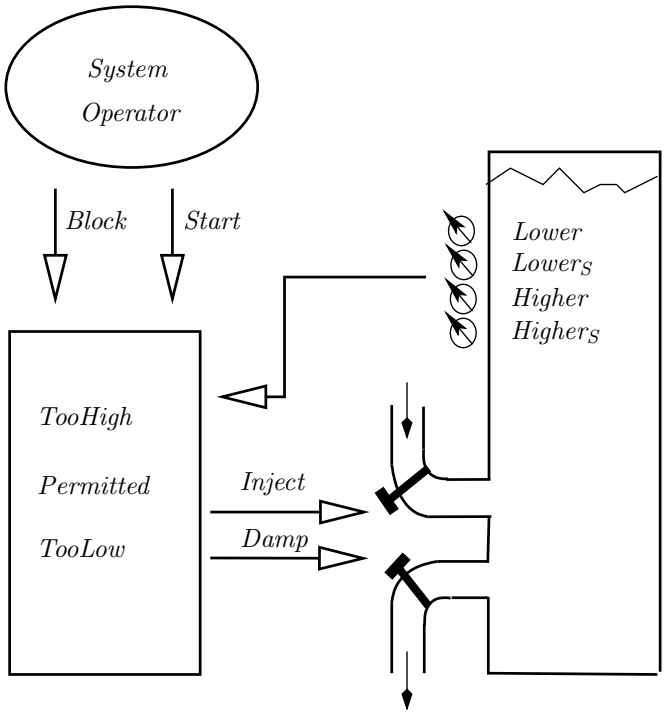


Abbildung 8.9: Skizze eines Kühlungssystems in einem nuklearen Kraftwerk

Der in Abbildung 8.10 ganz oben modellierte Zustand *DeviceCTRL* repräsentiert die eigentliche Steuerung der beiden Ventile. Initial ist die automatisierte Steuerung des Kühlungssystems blockiert. Ein Aktivieren der Automatisierung wird durch das Ereignis *Start* angezeigt. Die Automatisierung kann jederzeit durch Betätigen eines entsprechenden Knopfes – angezeigt durch das Ereignis *Block* – unterbrochen werden.

Zunächst befindet sich die Steuerung im Unterzustand *Idle*. Durch Abfragen der Unterzustände des parallel geschalteten Zustands *PressureMode* wird entschieden, ob die Steuerung in den Unterzustand *Damp* oder in den Unterzustand *Safety-Injection* überführt werden muss. Ist der Wasserdruck im Normalbereich, verbleibt die Steuerung im Zustand *Idle*. Der Zustand *Damp* wird wieder verlassen, wenn das interne Ereignis *DampOff* auftritt. Entsprechend wird der Zustand *Safety-Injection* verlassen, wenn das interne Ereignis *InjOff* auftritt.

Im mittleren Teil von Abbildung 8.10 ist der Zustand *PressureMode* modelliert. In Abhängigkeit von den Konstanten *permit* und *low* wird hier bestimmt, ob der in der internen Datenvariablen *pressure* abgelegte Wert, im Normal-Bereich (*Permitted*), im Überdruck-Bereich (*TooHigh*) oder im Unterdruck-Bereich (*TooLow*) liegt. Ist die Datenvariable *pressure* wieder im zulässigen Bereich, so wird beim Verlassen des Zustandes *TooHigh* das interne Ereignis *DampOff* bzw. beim Verlassen des Zustandes *TooLow* das interne Ereignis *InjOff* generiert.

In dem im unteren Teil von Abbildung 8.10 angegebenen Zustand *Measuring* ist modelliert, wie der Wert der Datenvariable *pressure* bestimmt wird. Wir betrachten zunächst das Systemverhalten, wenn der Unterzustand *Normal* aktiv ist. Hier wird in Abhängigkeit von zwei Sensoren die Druckveränderung festgestellt und entsprechend der Wert von *pressure* durch die Operationen Inc und Dec erhöht oder verringert. Das Ansteigen des Drucks wird durch das externe Ereignis *higher* und das Abfallen des Drucks durch das externe Ereignis

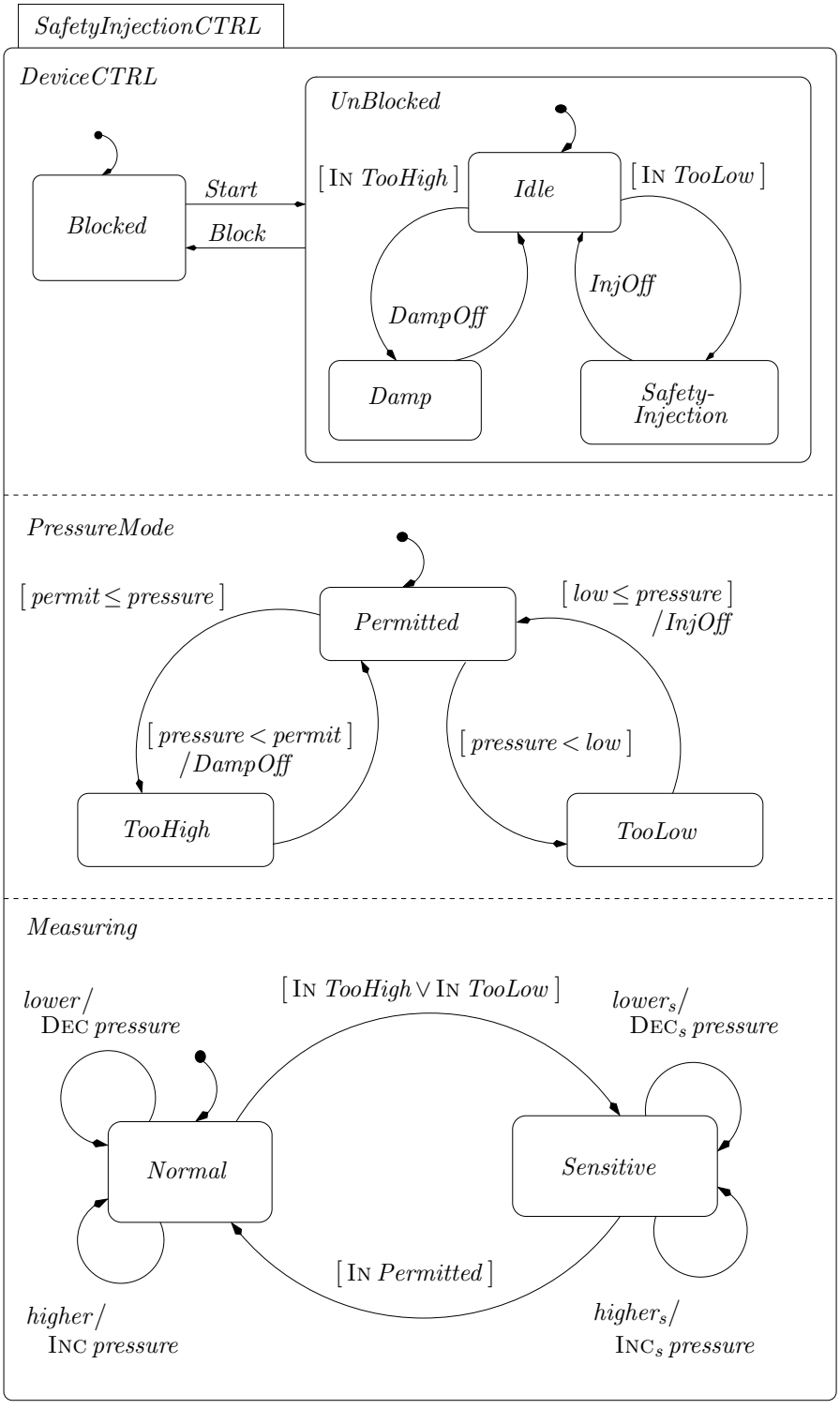


Abbildung 8.10: Statecharts-Spezifikation eines *Safety-Injection-Systems*

lower angezeigt. Initial wird von einem Normaldruck ausgegangen, dessen Wert der Datenvariablen einmalig zugewiesen wird. Verletzt der Wert von *pressure* den Normalbereich, so wird wegen der veränderten Druckverhältnisse eine sensiblere Messung vorgenommen. Dieses ist in der Modellierung durch einen Zustandswechsel von *Normal* zu *Sensitive* beschrieben. Im *Sensitive*-Zustand erfolgt die Messung analog zum *Normal*-Zustand, nur mit dem Unterschied, dass sensiblere Sensoren – angezeigt durch die Ereignisse $higher_s$ und $lower_s$ – verwendet werden. Entsprechend wird der Wert der Datenvariablen *pressure* in feineren Schritten erhöht bzw. verringert, was durch die Operationen $\textsc{Inc}_s$ und $\textsc{Dec}_s$ repräsentiert ist.

8.3.2 Analysen

Um eine maschinengestützte Analyse auf der im letzten Abschnitt eingeführten Statecharts-Spezifikation durchführen zu können, sind folgende Arbeitschritte zu erledigen.

1. Formalisierung der Statecharts-Spezifikation in Isabelle/HOL.
2. Formalisierung einer Abstraktionsfunktion in Isabelle/HOL.
3. Ableitung von temporalen Eigenschaften mit Hilfe der in den Abschnitt 8.2 vorgestellten Taktik, die auf der Statecharts-Spezifikation gelten sollen.

Formalisierung der Statecharts-Spezifikation

Wir beginnen mit der Formalisierung der Statecharts-Spezifikation in Isabelle/HOL auf Basis des in Teil I dieser Ausarbeitung angegebenen Vorgehens. In einem ersten Schritt führen wir hierfür einen Datentyp *ds* und eine Konstante *DS* zur Formalisierung des Datenraumes ein.

$$ds \equiv_{\mathsf{t}} V_0\ int$$

$$DS ::_{\mathsf{c}} ds\ dataspace$$
$$DS \equiv_{\mathsf{c}} Abs\_dataspace\,[\,ran\ V_0\,]$$

Der Datenraum besteht lediglich aus einer Partition, die durch eine Integer-Variable – in Abbildung 8.10 mit *pressure* benannt – beschrieben ist. Um auf die Partition des Datenraumes *DS* zugreifen zu können, führen wir folgende Selektionsoperatoren ein.

$$Sel_0 ::_{\mathsf{c}} ds \to int$$
$$Sel_0\,(V_0\,i) = i$$

$$Select_0 ::_{\mathsf{c}} ds\,data \to int$$
$$Select_0\,D \equiv_{\mathsf{c}} Sel_0\,(PartData\,D\,0)$$

Auf Basis der Formalisierung des Datenraumes geben wir nun eine Formalisierung der Sequentiellen Automaten an, mit der die Spezifikation modular beschrieben werden kann. Wir führen hierfür die fünf Konstanten *RootCTRL*, *DeviceCTRL*, *PressureMode*, *Measuring*

und *UnBlockedCTRL* ein. Jede dieser Konstanten definiert einen Sequentiellen Automaten. Beispielhaft wollen wir die Definition der Konstante *PressureMode* angeben. Wir folgen bei ihrer Definition dem in Abschnitt 3.1.1 dieser Ausarbeitung vorgestellten Vorgehen, indem wir die Komponenten des Sequentiellen Automaten durch eigene Konstanten beschreiben.

$$
\begin{aligned}
&\mathit{PressureStates} ::_c \mathit{string\ set}\\
&\mathit{PressureStates} \equiv_c \{\text{“}\mathit{Permitted}\text{”}, \text{“}\mathit{TooHigh}\text{”}, \text{“}\mathit{TooLow}\text{”}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureInit} ::_c \mathit{string}\\
&\mathit{PressureInit} \equiv_c \text{“}\mathit{Permitted}\text{”}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ActionInjOff} ::_c (\mathit{string}, \mathit{ds})\ \mathit{action}\\
&\mathit{ActionInjOff} \equiv_c (\{\text{“}\mathit{InjOff}\text{”}\}, \mathit{PUpdateDefault})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ActionDampOff} ::_c (\mathit{string}, \mathit{ds})\ \mathit{action}\\
&\mathit{ActionDampOff} \equiv_c (\{\text{“}\mathit{DampOff}\text{”}\}, \mathit{PUpdateDefault})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureLabels} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
&\mathit{PressureLabels} \equiv_c \{(\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\ D) < 20, \mathit{ActionDefault}),\\
&\qquad\qquad (\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\ D) < 10, \mathit{ActionInjOff}),\\
&\qquad\qquad (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\ D) < 20, \mathit{ActionDampOff}),\\
&\qquad\qquad (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\ D) < 10, \mathit{ActionDefault})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureTrans} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
&\mathit{PressureTrans} \equiv_c\\
&\qquad \{(\text{“}\mathit{Permitted}\text{”},\\
&\qquad\quad (\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\ D) < 20, \mathit{ActionDefault}),\\
&\qquad\quad \text{“}\mathit{TooHigh}\text{”}),\\
&\qquad\quad (\text{“}\mathit{TooLow}\text{”},\\
&\qquad\quad\ (\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\ D) < 10, \mathit{ActionInjOff}),\\
&\qquad\quad \text{“}\mathit{Permitted}\text{”}),\\
&\qquad\quad (\text{“}\mathit{TooHigh}\text{”},\\
&\qquad\quad\ (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\ D) < 20, \mathit{ActionDampOff}),\\
&\qquad\quad \text{“}\mathit{Permitted}\text{”}),\\
&\qquad\quad (\text{“}\mathit{Permitted}\text{”},\\
&\qquad\quad\ (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\ D) < 10, \mathit{ActionDefault})\\
&\qquad\quad \text{“}\mathit{TooLow}\text{”})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureMode} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{PressureMode} \equiv_c \mathit{Abs\_seqauto}\ (\mathit{PressureStates}, \mathit{PressureInit},\\
&\qquad\qquad\qquad\qquad \mathit{PressureLabels}, \mathit{PressureTrans})
\end{aligned}
$$

Wir gehen in der Definition des Sequentiellen Automaten *PressureMode* davon aus, dass die in Abbildung 8.10 verwendeten Konstanten *low* und *high* mit den Werten 10 und 20 belegt sind (vgl. Formalisierung der Guards in der Konstante *PressureLabels*). Für eine ausführliche Darstellung der anderen Sequentiellen Automaten verweisen wir auf den Anhang B.2.

Schließlich konstruieren wir mit folgender Beschreibung aus den Sequentiellen Automaten einen Hierarchischen Automaten. Der Operator *LiftInitData* bildet hierbei eine Liste von Datenbelegungen für die Partitionen in eine Datenraumbelegung ab. Da in unserem Beispiel nur eine Partition existiert, haben wir eine einelementige Liste angegeben. Wir nehmen für unsere Analyse an, dass die Integer-Variable *pressure* initial mit der Zahl 15 belegt ist.

$$
\begin{aligned}
&\mathit{SafetyInjectionSystem} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{hierauto}\\
&\mathit{SafetyInjectionSystem} \equiv_{\mathsf{c}} (\mathit{PseudoHA}\ \mathit{RootCTRL}\ (\mathit{LiftInitData}\,[V_0\,15]))\\
&\qquad\qquad \boxplus (\text{“}\mathit{SafetyInjectionCTRL}\text{”}, \mathit{DeviceCTRL})\\
&\qquad\qquad \boxplus (\text{“}\mathit{SafetyInjectionCTRL}\text{”}, \mathit{PressureMode})\\
&\qquad\qquad \boxplus (\text{“}\mathit{SafetyInjectionCTRL}\text{”}, \mathit{Measuring})\\
&\qquad\qquad \boxplus (\text{“}\mathit{UnBlocked}\text{”}, \mathit{UnBlockedCTRL})
\end{aligned}
$$

Formalisierung der Abstraktionsfunktion

Die im letzten Abschnitt eingeführte Statecharts-Spezifikation ist auf einem unendlichen Datenraum definiert, so dass zu deren Analyse kein traditioneller Model-Checker eingesetzt werden kann. Aus diesem Grunde definieren wir eine geeignete Abstraktionsfunktion, mit deren Hilfe der unendliche Datenraum durch einen endlichen Datenraum abstrakt interpretiert werden kann. Passend zum abstrakten Datenraum erzeugen wir eine abstrakte Statecharts-Spezifikation unter Verwendung der in Abschnitt 8.2 angegebenen Taktik.

Wie bereits in Definition 7.1 angegeben, besteht eine Abstraktionfunktion in unserer Formalisierung aus drei Komponenten.

1. eine Liste von Teilfunktionen
2. ein konkreter Datenraum
3. ein abstrakter Datenraum

Den konkreten Datenraum haben wir mit dem Datentyp *ds* und der Konstanten *DS* bereits eingeführt. In Analogie dazu führen wir einen Datentyp ds_{a} und eine Konstante DS_{a} ein, um den abstrakten Datenraum zu formalisieren. Dieser Datenraum besteht aus einer Partition, die durch zwei boolesche Variablen beschrieben ist.

$$
\begin{aligned}
&ds_{\mathsf{a}} \equiv_{\mathsf{t}} V_0^{\mathsf{a}}\ (\mathit{bool} * \mathit{bool})\\
&\\
&DS_{\mathsf{a}} ::_{\mathsf{c}} ds_{\mathsf{a}}\ \mathit{dataspace}\\
&DS_{\mathsf{a}} \equiv_{\mathsf{c}} \mathit{Abs\_dataspace}\,[\,\mathit{ran}\ V_0^{\mathsf{a}}\,]
\end{aligned}
$$

Wir interpretieren die Integer-Variable *pressure* durch die Prädikate $pressure < 10$ und $pressure < 20$. Hierbei soll in der ersten booleschen Variable von DS_{a} die Gültigkeit des Prädikates $pressure < 10$ und in der zweiten die Gültigkeit des Prädikates $pressure < 20$ kodiert werden. Um diesen Zusammenhang zu beschreiben, definieren wir die für eine Abstraktionsfunktion geforderte Liste von Teilfunktionen, wobei jede dieser Funktionen eine Partition des konkreten Datenraumes auf eine Partition des abstrakten Datenraumes abbildet. In unserem Beispiel handelt es sich um eine einelementige Liste, da jeweils nur genau eine Partition im abstrakten und konkreten Datenraum vorliegt. Mit folgender Definition führen wir eine Teilfunktion ein, mit der die Integer-Variable des konkreten Datenraumes durch zwei boolesche Variablen des abstrakten Datenraumes interpretiert wird.

$$\begin{aligned} &AbsFun ::_{\mathsf{c}} (ds \rightarrow ds_{\mathsf{a}})\ list \\ &AbsFun \equiv_{\mathsf{c}} [\lambda\, D.\ V_0^{\mathsf{a}}((Select_0\, D) < 10, (Select_0\, D) < 20)] \end{aligned}$$

Abschließend kann die Abstraktionsfunktion `R` aus den Einzelteilen folgendermaßen zusammengesetzt werden.

$$\begin{aligned} &R ::_{\mathsf{c}} (ds, ds_{\mathsf{a}})\ abs \\ &R \equiv_{\mathsf{c}} Abs\_abs(AbsFun, DS, DS_{\mathsf{a}}) \end{aligned}$$

Mit Folgendem Theorem haben wir in Isabelle/HOL gezeigt, dass die drei definierten Bestandteile die Eigenschaften der Wohlgeformtheit von Abstraktionsfunktionen aus Definition 7.2 erfüllen.

Abgeleitetes Theorem 8.1 (Wohlgeformtheit der Abstraktionsfunktion *R*)

$$AbsCorrect\ AbsFun\ DS\ DS_{\mathsf{a}} \qquad \text{(AbsFunCorrect)}$$

□

Ableiten von temporalen Eigenschaften

Wenden wir die in Abschnitt 8.2 vorgestellte Taktik an, um die Gültigkeit von temporallogischen Formeln des universalen Fragments der CTL nachzuweisen, wird automatisch eine abstrakte Statecharts-Spezifikation generiert. Im Folgenden stellen wir die abstrahierte Statecharts-Spezifikation vor. Wir beschränken uns in der Darstellung auf Sequentielle Automaten, die datenbehaftete Bestandteile enthalten. Zunächst stellen wir in Abbildung 8.11 die Abstraktion des Sequentiellen Automaten *PressureMode* vor. Die einzigen Bestandteile in *PressureMode*, die vom Datenraum abhängen, sind Guards. Entsprechend werden diese Bestandteile durch abstrakte Gegenstücke ersetzt. Da wir eine Abstraktionsfunktion gewählt haben, mit der die Prädikate der Guards exakt beschrieben werden können, entstehen keine zusätzlichen selbstbezüglichen Transitionen bei der Abstraktion. Nur in dem Falle, dass ein Guard durch die Abstraktion abgeschwächt würde, besteht die Notwendigkeit, mit Hilfe von selbstbezüglichen Transitionen implizites Verhalten zu repräsentieren.

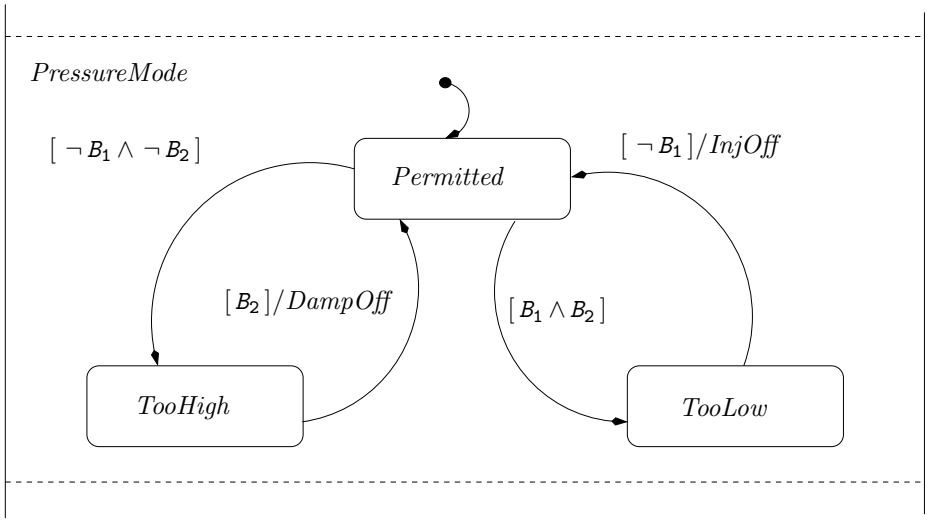


Abbildung 8.11: Ausschnitt der abstrahierten *Safety-Injection-Spezifikation*: *PressureMode*

In Analogie zur Abbildung 8.11 zeigen wir in Abbildung 8.12 die Abstraktion des Sequentiellen Automaten `Measuring`. Die einzigen Bestandteile, die vom Datenraum abhängen, sind hier die Update-Funktionen der selbstbezüglichen Transitionen. Jede dieser Transitionen wird in dem abstrakten Modell durch drei Transitionen repräsentiert. Eine genauere Approximation ist nicht möglich, da die Guards der Transitionen keine Annahmen über die Datenvariable *pressure* treffen. Lediglich der Effekt $B_1 \wedge \neg B_2$ kann ausgeschlossen werden, da dieses Prädikat im konkreten Modell nicht erfüllbar ist.

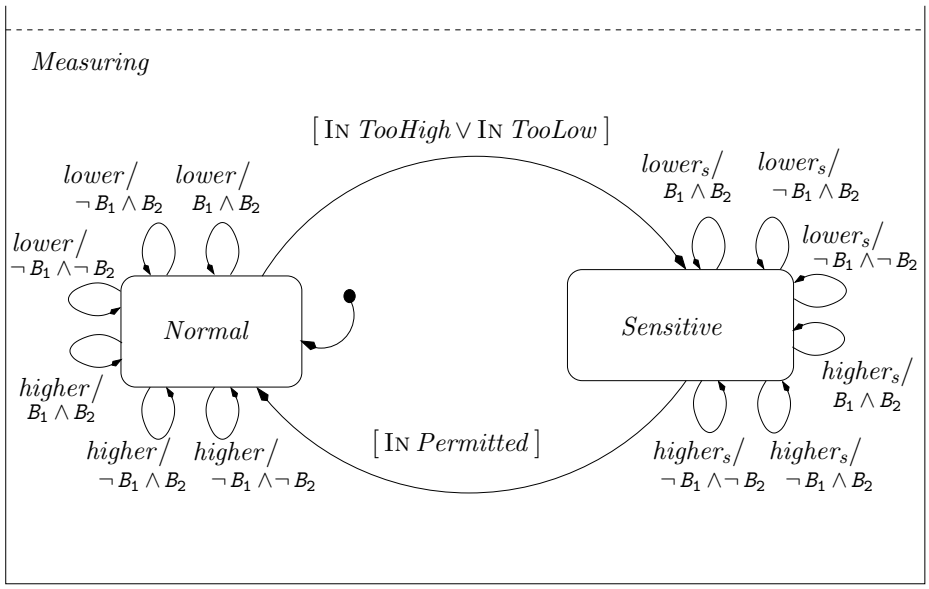


Abbildung 8.12: Ausschnitt der abstrahierten *Safety-Injection-Spezifikation*: *Measuring*

Folgende Theoreme haben wir auf Basis des vorgestellten abstrakten Modells verifiziert.

Abgeleitete Theoreme 8.2 (Aussagen für das *Safety-Injection-System*)

$$\frac{\begin{array}{c} A_1 = Atom\,(Val\,(\lambda\ D.\ (Select_0\ D) < 10)) \\ A_2 = Atom\,(In\ \text{“}Permitted\text{”}) \\ C = AX\,(Atom\,(In\ \text{“}TooLow\text{”})) \end{array}}{SafetyInjectionSystem \models_{HA}\ AG\,((A_1 \wedge_P A_2) \Rightarrow_P C)} \quad \text{(PermittedToTooLow)}$$

$$\frac{\begin{array}{c} A_1 = Atom\,(Val\,(\lambda\ D.\ 20 \leq (Select_0\ D))) \\ A_2 = Atom\,(In\ \text{“}Permitted\text{”}) \\ C = AX\,(Atom\,(In\ \text{“}TooHigh\text{”})) \end{array}}{SafetyInjectionSystem \models_{HA}\ AG\,((A_1 \wedge_P A_2) \Rightarrow_P C)} \quad \text{(PermittedToTooHigh)}$$

$$\frac{\begin{array}{c} A_1 = Atom\,(Val\,(\lambda\ D.\ (Select_0\ D)) < 20) \\ A_2 = Atom\,(In\ \text{“}TooHigh\text{”}) \\ C = AX\,(Atom\,(In\ \text{“}Permitted\text{”}) \wedge_P Atom\,(En\ \text{“}DampOff\text{”})) \end{array}}{SafetyInjectionSystem \models_{HA}\ AG\,((A_1 \wedge_P A_2) \Rightarrow_P C)} \quad \text{(TooHighToPermitted)}$$

$$\frac{\begin{array}{c} A_1 = Atom\,(Val\,(\lambda\ D.\ 10 \leq (Select_0\ D))) \\ A_2 = Atom\,(In\ \text{“}TooLow\text{”}) \\ C = AX\,(Atom\,(In\ \text{“}Permitted\text{”}) \wedge_P Atom\,(En\ \text{“}InjOff\text{”})) \end{array}}{SafetyInjectionSystem \models_{HA}\ AG\,((A_1 \wedge_P A_2) \Rightarrow_P C)} \quad \text{(TooLowToPermitted)}$$

□

Man beachte, dass die verwendete Taktik zur Ableitung dieser Theoreme Beweiswerkzeuge außerhalb von Isabelle benutzt. Aus diesem Grunde sind die Theoreme in Isabelle/HOL mit einer speziellen Markierung versehen. Es wird damit explizit darauf hingewiesen, dass alle für den Beweis eingesetzten Werkzeuge vertrauenswürdig sein müssen.

8.4 Zusammenfassung

In diesem Kapitel haben wir zwei Taktiken vorgestellt, mit denen die maschinengestützte Analyse einer Statecharts-Spezifikation überwiegend automatisch erfolgen kann. Mit Hilfe der ersten Taktik kann eine Statecharts-Spezifikation, die auf einem endlichen Datenraum basiert, durch den Model-Checker SMV verifiziert werden. Hierzu werden durch die Taktik während eines Beweisverlaufs zunächst die notwendigen Informationen über den zu analysierenden Statechart und die zu prüfende CTL-Formel aus der Termstruktur von Isabelle/HOL ausgelesen und anschließend in eine Eingabedatei für SMV übersetzt. Im ersten Abschnitt dieses Kapitels stellten wir ein Übersetzungsschema vor, das dieser Transformation zugrunde liegt. Es basiert teilweise auf Arbeiten von Erich Mikk. Wir haben es für unsere Zwecke angepasst und insbesondere um die Behandlung von Daten erweitert.

Die zweite Taktik ist für Statecharts-Spezifikationen gedacht, die entweder auf sehr großen endlichen oder sogar auf unendlichen Datenräumen basieren. Mit dieser Taktik kann der Datenraum einer Statecharts-Spezifikation auf Basis von Prädikaten algorithmisch abstrahiert

werden. Wir implementierten hierfür einen Abstraktionsalgorithmus in ML, der eine Prädikatenabstraktion zur Berechnung von Überapproximationen umsetzt. Der Algorithmus verwendet zum Auswerten von atomaren Aussagen das Beweiswerkzeug SVC. Das Ergebnis der Abstraktion ist wiederum eine Statecharts-Spezifikation, die in Form von verschiedenen Datenformaten zurückgegeben wird. So wird durch die Taktik eine Isabelle-Theorie-Datei, eine SMV-Eingabe-Datei und eine LaTeX-Datei zur Repräsentation der Statecharts-Spezifikation erzeugt.

Im letzten Abschnitt dieses Kapitels stellten wir eine Fallstudie vor. Es handelt sich hierbei um die Steuerung eines Kühlkreislaufs in einem nuklearen Kraftwerk. Zunächst führten wir eine Statecharts-Spezifikation ein, mit der das Systemverhalten beschrieben werden kann. Danach erläuterten wir, wie diese Statecharts-Spezifikaton in Isabelle/HOL formalisiert wird. Schließlich zeigten wir an dem Beispiel die Praktikabilität der implementierten Taktiken, indem wir eine Abstraktionsfunktion definierten, eine automatisierte Prädikatenabstraktion durchführten und schließlich Analysen mit dem Model-Checker SMV durchführten.

Kapitel 9

Verwandte Arbeiten

Um diese Dissertation geeignet zu motivieren, haben wir in den vorhergehenden Kapiteln bereits einige verwandte Arbeiten zitiert. Wir sind dabei insbesondere auf Ansätze aus der Literatur eingegangen, für die im Rahmen dieser Dissertation unmittelbar Erweiterungen entstanden sind.

Neben diesen Ansätzen gibt es noch eine ganze Reihe von Publikationen, die auch inhaltlich verwandt zu dieser Arbeit sind. Diese Arbeiten werden wir in diesem Kapitel systematisch vorstellen und mit der vorliegenden Dissertation vergleichen. Dieses Kapitel gliedert sich in drei Teile. Zunächst gehen wir auf Arbeiten aus der Literatur ein, in denen die maschinengestützte Verifikation von Statecharts mit Hilfe von Theorembeweisern oder Model-Checkern realisiert wurde. Der zweite Teil beschreibt Ansätze aus der Literatur zur automatisierten Generierung von eigenschaftserhaltenden Abstraktionen. Abschließend diskutieren wir verwandte Arbeiten, die sich in keine der anderen Teile einordnen lassen, aber trotzdem Bezüge zu dieser Arbeit aufweisen.

9.1 Maschinengestützte Verifikation für Statecharts

Wir gliedern diesen Abschnitt in zwei Bereiche. Zunächst stellen wir Arbeiten vor, die dokumentieren, wie Statecharts in einem Theorembeweiser formalisiert werden können. Danach präsentieren wir Arbeiten, deren Hauptanliegen in der Entwicklung von neuartigen Model-Checking-Techniken zur Analyse von Statecharts-Spezifikationen besteht.

9.1.1 Formalisierungen in Theorembeweisern

Die erste uns bekannte Formalisierung von Statecharts in einen Theorembeweiser wurde von Nancy Day im Jahre 1993 vorgenommen [Day93]. Im Rahmen ihrer Masterarbeit übersetzte sie Statemate-Statecharts in die Eingabesprache des Werkzeugs HOL-Voss [JS94]. Bei dem System HOL-Voss handelt es sich um eine Werkzeugintegration aus dem Theorembeweiser HOL [GM93] und einem symbolischen Model-Checker. Der Fokus der Arbeit von Nancy Day liegt dabei weniger auf einer effizienten Formalisierung innerhalb des HOL-Systems, sondern ihre Arbeit hat eher das Ziel, ein Frontend zur Verifikation von CTL-Formeln mit dem in HOL-Voss integrierten Model-Checker zur Verfügung zu stellen. Obwohl für die Formalisierung von Statecharts keine Hierarchischen Automaten verwendet werden, weist der Ansatz starke Bezüge zu unserer Arbeit auf. Insbesondere wird die Behandlung von Datenvariablen unterstützt. Die daraus resultierenden semantischen Effekte - zum Beispiel Racing – werden in der Formalisierung behandelt. Allerdings ist die Formalisierung von Datenräumen in unserer

Arbeit deutlich umfassender beschrieben. In der Arbeit von Nancy Day wird im Gegensatz zu unserer Arbeit nicht ausgeführt, wie temporale Formeln in Bezug auf eine Statecharts-Spezifikation semantisch interpretiert werden können. Ferner sind keine Abstraktionskonzepte für Statecharts-Spezifikationen beschrieben.

Weiterhin ist uns eine Arbeit zur Formalisierung von Statemate-Statecharts in dem Beweissystem KIV bekannt [TSOR04]. KIV (*Karlsruhe-Interactive-Verifier*) [Rei92] ist ein interaktiver Theorembeweiser der auf der LCF-Logik [GMW79] basiert. Die Autoren definieren ein Sequenzenkalkül zur Verifikation von Statecharts-Spezifikationen mit unendlichen Datenräumen. Allerdings sind die Datenräume eines Statecharts im Gegensatz zu unserer Arbeit durch eine separate algebraische Spezifikation beschrieben. Ferner unterscheidet sich der Ansatz darin, dass eine asynchrone Makro-Schrittsemantik für Statecharts formalisiert wurde, die sich von der in dieser Dissertation behandelten synchronen Schrittsemantik unterscheidet.

Formalisierungen für UML-Statecharts existieren auch für den Theorembeweiser PVS [ORS92]. So stellte Issa Traore im Jahre 2000 eine Formalisierung für UML-Statecharts vor, die in der Spezifikationssprache von PVS ausgedrückt ist [Tra00]. Insbesondere dokumentiert er in seiner Arbeit, wie UML-Statecharts mit einem in PVS verfügbaren Model-Checker verifiziert werden können. Aufbauend auf dieser Arbeit wurde von Demissie B. Aredo eine ähnliche, aber im Detail besser ausgearbeitete Formalisierung entwickelt [Are04]. In Abgrenzung zu unserer Arbeit handelt es sich um eine Formalisierung für UML-Statecharts. Ferner basiert die Formalisierung nicht auf Hierarchischen Automaten.

9.1.2 Model-Checker für Statecharts

Wir betrachten zunächst Arbeiten, in denen Techniken des Model-Checkings für Statemate-Statecharts untersucht werden. Anschließend stellen wir Arbeiten zum Model-Checking von UML-Statecharts vor.

Statemate-Statecharts

Diese Dissertation ist zu großen Teilen als eine Fortführung der Arbeiten von Erich Mikk zu sehen. Er stellt in seiner Dissertation eine Semantik für Statemate-Statecharts vor, die mit Hilfe von Erweiterten Hierarchischen Automaten beschrieben ist [MLS97]. Auf Basis dieser Semantik entwickelt er zwei Ansätze zur maschinengestützten Analyse von Statemate-Statecharts mit Model-Checkern [Mik00]. Einerseits definiert und implementiert er hierfür eine Vorschrift zur Übersetzung von Erweiterten Hierarchischen Automaten in die Eingabesprache des symbolischen CTL-Model-Checkers SMV. Andererseits entwickelt er ein Konzept, mit dem Erweiterte Hierarchische Automaten in der Spezifikationssprache Promela ausgedrückt werden können [MLSH99]. Promela ist eine Prozessalgebra, die als Eingabesprache für den LTL-Model-Checker SPIN [Hol03] verwendet wird. Es ist also auf Basis des von Erich Mikk vorgeschlagenen Übersetzungsschemas sowohl möglich, Eigenschaften der temporalen Logik CTL mit dem Model-Checker SMV, als auch Eigenschaften der temporalen Logik LTL mit dem Model-Checker SPIN zu überprüfen. Im Gegensatz zu unserer Arbeit ist die Behandlung von Datenvariablen nicht möglich.

Jan-Juan Hiemer schlägt in seiner Dissertation [Hie98] vor, Statemate-Statecharts mit dem CSP-Model-Checker FDR [Ros05] zu verifizieren. Er erläutert, wie Statecharts in CSP [Hoa85] repräsentiert werden können und beweist, dass die Transformation die Semantik der zu übersetzenden Statecharts-Spezifikation erhält. In dieser Arbeit wird – wie in der Arbeit

von Erich Mikk – nur die Verifikation von Statecharts-Spezifikationen unterstützt, die keine Datenräume besitzen. In jüngeren Arbeiten wurde von Bill Roscoe und Zhenzhong Wu die Übersetzung von Statemate-Statecharts nach CSP optimiert und unter anderem um eine einfache Behandlung von Daten erweitert [RW06]. Jedoch ist der Ansatz auf endliche Datenräume eingeschränkt. Konflikte beim konkurrierenden Schreiben des Datenraumes sind durch das Auftreten eines speziellen Fehler-Ereignisses repräsentiert. Eine semantische Interpretation des Racing-Effektes – beispielsweise durch eine Interleaving Semantik – ist im Gegensatz zu unserer Arbeit nicht umgesetzt.

Robert Büssow beschreibt in seiner Dissertation [Büs03] ein Verfahren zum Model-Checking von μSZ-Spezifikationen. Die von ihm unterstützte Spezifikationssprache μSZ [BGGK97] stellt eine Kombination aus den Spezifikationssprachen Z [Spi92] und Statemate-Statecharts dar. Der entwickelte Ansatz beruht auf zwei Übersetzungsschritten. Zunächst werden die Statecharts-Anteile einer μSZ-Spezifikation in eine Z-Spezifikation überführt. In einem zweiten Schritt werden sowohl das Ergebnis des ersten Transformationsschritts, als auch der originale Z-Anteil der μSZ-Spezifikation in die Eingabesprache des Model-Checkers SMV übersetzt. Der zweite Übersetzungsschritt ist nur dann durchführbar, wenn sich die zu transformierenden Z-Spezifikationen in einem so genannten reduzierten Z – eine echte Untermenge von Z – ausdrücken lassen. Mit dem von Robert Büssow entwickelten Verfahren kann die Gültigkeit von CTL-Formeln auf Modellen der Spezifikationensprachen μSZ, Z und Statemate-Statecharts überprüft werden. Er ist insbesondere dazu in der Lage, Statecharts-Spezifikationen zu behandeln, die auf einem Datenraum definiert sind. Allerdings wird im Gegensatz zu unserem Ansatz gefordert, dass die Datenräume endlich sind. Ein Datenabstraktionsprozess wird nicht unterstützt.

Udo Brockmeyer und Gunnar Wittich verwenden einen symbolischen Model-Checker der Firma Siemens zur Verifikation von Statemate-Modellen [BW98a, BW98b]. Ein Schwerpunkt ihrer Arbeit besteht in der Behandlung von Zeitanforderungen, die in TCTL (*Timed CTL*) beschrieben sind. *Timed CTL* ist eine Erweiterung der temporalen Logik CTL um diskrete Zeit. Eine Behandlung von Datenvariablen wird nicht unterstützt.

Der Model-Checker mit dem Namen Safety-Checker Blockset [TS05] wird von der Firma TNI-Software entwickelt und dient der Verifikation von so genannten Stateflow-Modellen. Stateflow-Modelle sind eine Variante von Statecharts, deren Semantik sehr ähnlich zur Statemate-Semantik ist. Der Formalismus wurde für die Werkzeugumgebung Matlab/Simulink [ABRW05] entwickelt. Der Model-Checker Safety-Checker Blockset ist in diese Werkzeugumgebung integriert. Über die Behandlung von Datenvariablen wird in den Arbeiten zu diesem Ansatz leider nichts ausgesagt.

UML-Statecharts

Johan Lilius und Ivan Porres Paltor haben eine Formalisierung für UML-State-Machines vorgeschlagen [LP99a]. Auf Basis dieser Formalisierung entwickelten sie in dem Verifikationswerkzeug vUML einen Algorithmus, mit dem durch Model-Checking die Gültigkeit von temporalen Eigenschaften der LTL bezüglich eines UML-Statecharts überprüft werden kann [LP99b, Por01]. Die dort verwendete Technik beruht auf einer Übersetzung von UML-Statecharts in die Eingabesprache Promela des Model-Checkers SPIN [Hol03]. Die Arbeit stellt eine Erweiterung des kurz vorher von Diego Latella, Istvan Majzik und Mieke Massink vorgeschlagenen Übersetzungsschemas dar [LMM99a]. Auch diese Autoren entwickelten einen Ansatz, mit dem UML-Statecharts durch den Model-Checker SPIN auf die Gültigkeit von Eigenschaften

der LTL überprüft werden können. Ferner beschreiben die Autoren einen Ansatz, mit dem Eigenschaften des universalen Fragments der CTL durch Verwendung des Model-Checkers JACK überprüft werden können [GLM99]. Die Arbeiten basieren auf einer Formalisierung von UML-Statecharts mit Hilfe von Erweiterten Hierarchischen Automaten [LMM99b]. Im Gegensatz zu unserer Arbeit wird allerdings die Behandlung von Datenräumen nicht unterstützt.

Ein anderer Ansatz zur automatischen Analyse von UML-Statecharts wurde von Timm Schäfer, Alexander Knapp und Stephan Merz vorgeschlagen [SKM01]. Die Autoren sind daran interessiert, Inkonsistenzen zwischen verschiedenen Sichten der UML aufzudecken. Hierfür entwickelten sie das Werkzeug Hugo, mit dessen Hilfe automatisch geprüft werden kann, ob ein UML-Statechart mit einem Collaboration-Diagramm verträglich ist. Teil der Werkzeugumgebung ist auch hier ein Übersetzungsalgorithmus, der UML-Statecharts als Eingabe für den Model-Checker SPIN aufbereitet. Die Collaboration-Diagramme werden zunächst in Büchi-Automaten überführt und anschließend durch so genannte *Never-Claims* repräsentiert. Ist der Interaktionsablauf eines Collaboration-Diagramms in einem zu prüfenden Statechart möglich, so wird durch den Model-Checker SPIN ein entsprechendes Gegenbeispiel für die generierten Never-Claims zurückgegeben. Wird hingegen kein Gegenbeispiel erzeugt, ist damit nachgewiesen, dass der Interaktionsablauf des Collaboration-Diagramms mit dem zu prüfenden Statechart nicht verträglich ist. Ferner wurde diese Arbeit von den Autoren um die Behandlung von Zeit erweitert [KMR02]. Zur Analyse wird hier anstelle von SPIN der Model-Checker Uppaal verwendet, mit dem Hierarchische Timed Automata verifiziert werden können. In diesem Zusammenhang wollen wir abschließend auf das Verifikationswerkzeug HUppaal hinweisen, mit dem Real-Time-Statecharts in Timed-Automata transformiert und dann auf dem Ergebnis der Transformation Eigenschaften der TCTL überprüft werden können [DMY02].

9.2 Abstraktionstechniken

Um umfangreiche Modelle automatisiert verifizieren zu können, ist der Einsatz von geeigneten Abstraktionstechniken unvermeidbar. In der Literatur sind viele Arbeiten zu finden, in denen sich die Autoren mit der Abstraktion von Zustandssystemen auseinandersetzen [Lon93, CGL94, LGS+95, Mer97]. Auch wenn im Allgemeinen solche Abstraktionstechniken nicht vollautomatisch sind, gibt es ein starkes Bemühen, Algorithmen zur Verfügung zu stellen, die sich durch eine Teilautomatisierung auszeichnen und nur wenig Interaktion mit dem Benutzer erfordern. Die Kernidee in vielen Arbeiten besteht in der Verwendung von Methoden zur *Abstrakten Interpretation* des zu verifizierenden Verhaltens [CC77]. Die zentrale Frage ist hierbei, bezüglich welcher Klasse von Eigenschaften die Abstraktion eigenschaftserhaltend ist. Dennis Dams entwickelte im Rahmen seiner Dissertation ein Framework zur abstrakten Interpretation von Zustandssystemen. Die in der Arbeit von Dennis Dams generierten abstrakten Modelle sind bezüglich der temporalen Logik CTL* [EH86] eigenschaftserhaltend [Dam96]. In Abgrenzung dazu haben wir in der vorliegenden Dissertation ein Verfahren zur Abstraktion von Statecharts-Spezifikationen entwickelt, das bezüglich des universalen Fragments der CTL eigenschaftserhaltend ist. In Kapitel 6 dieser Ausarbeitung haben wir begründet, warum diese Einschränkung in unserem Ansatz erforderlich ist.

Eine besonders weit verbreitete Abstraktionstechnik ist die Prädikatenabstraktion, bei der aus einer Menge von atomaren Aussagen über dem potentiell unendlichen Zustandsraum ein abstraktes endliches Modell gebildet wird. Im abstrakten Modell sind die Prädikate durch boolesche Variablen beschrieben. Die erste automatisierte Prädikatenabstraktion zum Model-

Checking von unendlichen Zustandssystemen wurde von Susanne Graf und Hassen Saïdi angegeben [GS97]. Dieser Ansatz wurde in den folgenden Jahren weiter verfeinert und in einigen Teilen optimiert [DDS99, SS99, CMM00]. Der von uns im Rahmen dieser Dissertation entwickelte Abstraktionsalgorithmus setzt eine Prädikatenabstraktion um (vgl. Abschnitt 8.2). Wir haben dabei insbesondere Ideen der Arbeiten von Hassen Saïdi und Natarajan Shankar aufgegriffen [SS99].

Ein wichtiges Kriterium zur Einordnung von eigenschaftserhaltenden Abstraktionstechniken ist die Frage, durch welchen Formalismus das abstrakte Modell repräsentiert wird. Wir haben in dieser Dissertation eine strukturerhaltende Abstraktionstechnik vorgestellt, bei der das Ergebnis des Abstraktionsprozesses durch einen zur Eingabe strukturell identischen Statechart repräsentiert ist. Im Gegensatz dazu schlagen beispielsweise Dominique Cansell, Dominique Méry und Stephan Merz eine Klasse von Diagrammen vor, die zur abstrakten Interpretation unendlicher Zustandssysteme eingesetzt werden können [CMM00]. Diese Diagramme werden – wie bei einer Prädikatenabstraktion üblich – aus einer Menge von Prädikaten abgeleitet, die der Benutzer vorgeben muss. Die automatisch generierbaren Diagramme werden deshalb auch als *Prädikatendiagramme* bezeichnet. In Abgrenzung zu unserer Arbeit sind die Autoren auf Basis der Prädikatendiagramme dazu in der Lage, Lebendigkeitseigenschaften zu erhalten und entsprechend nachzuweisen. Allerdings ist ungeklärt, ob die vorgeschlagene Technik auf strukturerhaltende Abstraktionen hierarchischer Zustandssysteme übertragen werden kann. Die Autoren entwickelten weiterhin eine Verfeinerungstechnik für Prädikatendiagramme, so dass bei einer zu groben Abstraktion das abstrakte Modell schrittweise um Informationen angereichert werden kann [CMM01].

Techniken, mit denen generierte abstrakte Modelle schrittweise wieder verfeinert werden können, wurden auch in anderen Arbeiten vorgestellt [CGJ+00, DD02]. Hierbei wird in dem Fall, dass die Verifikation einer temporalen Eigenschaft durch Model-Checking auf dem abstrakten Modell scheitert, das durch den Model-Checker erzeugte Gegenbeispiel genauer analysiert. Mit dieser Analyse wird überprüft, ob das Gegenbeispiel in dem Verhalten des konkreten Modells tatsächlich möglich ist, oder ob es nur durch Verhalten möglich wurde, das dem abstrakten Modell beim Bilden der Abstraktion hinzugefügt worden ist. Trifft letzteres zu, wird das abstrakte Modell so verfeinert, dass das Gegenbeispiel in dem Verhalten des abstrakten Modells nicht mehr möglich ist. Danach wird ein neuer Versuch unternommen, die temporale Eigenschaft mit Hilfe des Model-Checkers zu verifizieren und gegebenenfalls bei Rückgabe eines Gegenbeispiels erneut eine Verfeinerung vorgenommen.

In der Literatur sind eine ganze Reihe anderer Ansätze zur Abstraktion unendlicher Zustandssysteme zu finden. Eine systematische Darstellung aller dieser Arbeiten würde allerdings den Rahmen dieser Ausarbeitung sprengen.

9.3 Sonstige Arbeiten

Abschließend wollen wir auf zwei Arbeiten hinweisen, deren Zielsetzung nicht direkt mit der Zielsetzung dieser Dissertation übereinstimmt. Wir wollen damit zeigen, dass zwischen unserer Arbeit und anderen Anwendungsbereichen Querbezüge existieren, die es eventuell erlauben, Teile der von uns entwickelten Konzepte zu übertragen.

9.3.1 Code-Verifikation

In der Forschungsabteilung von Microsoft ist das so genannte SLAM-Projekt angesiedelt [BR02]. In diesem Projekt wird eine Technik entwickelt, die dem Korrektheitsnachweis von temporalen Sicherheitseigenschaften auf C-Programmen dient. Die zu prüfenden Sicherheitseigenschaften werden in einer für dieses Projekt entwickelten Spezifikationsprache SLIC (*Specification Language for Interface Checking*) beschrieben und in den Programm-Code eingebettet. In der Spezifikation werden die zu prüfenden Eigenschaften durch einen Automaten repräsentiert, auf dessen Basis das Programmverhalten während der Ausführung überwacht werden kann. Für die anfallenden Verifikationsaufgaben wird eine Prädikatenabstraktion eingesetzt, die für ein gegebenes C-Programm und eine Menge von Prädikaten, ein boolesches Programm konstruiert. Auf dem booleschen Programm wird schließlich mit Hilfe des Model-Checkers BEBOP versucht, die geforderten Eigenschaften zu zeigen. Der Model-Checker BEBOP wurde speziell zur Verifikation von booleschen Programmen entwickelt [BR00].

9.3.2 Hardware-Verifikation

Im Rahmen des VerifSoft-Projektes [Ver], wurde von Sergey Tverdyshev der Theorembeweiser Isabelle/HOL mit dem Model-Checker NuSMV und mit verschiedenen SAT-Lösern kombiniert [Tve05]. Die Arbeit ist auf die Verifikation von CTL- und LTL-Formeln für Hardware-Spezifikationen fokussiert. Zu diesem Zweck wurden auch Formalisierungen für die temporalen Logiken LTL und CTL in Isabelle/HOL angegeben. Genaue Ausführungen dieser Formalisierungen sind leider nicht veröffentlicht, so dass es schwierig ist, diese mit unserer CTL-Formalisierung aus Kapitel 5 zu vergleichen. Ein zentraler Unterschied zu unserer Arbeit besteht aber darin, dass Sergey Tverdyshev ein Modell nur dann verifizieren kann, wenn es in Isabelle/HOL als endliche Zustandsmaschine vorliegt. Dies bedeutet unter anderem, dass alle verwendeten Datentypen auf einem endlichen Definitionsbereich beruhen müssen. Ferner werden keine hierarchischen Zustandsautomaten unterstützt.

Kapitel 10

Zusammenfassung und Ausblick

In diesem Kapitel geben wir einen Überblick über die im Rahmen dieser Dissertation entstandenen Ergebnisse. Wir beurteilen, ob die Zielsetzung erfüllt werden konnte und ob der entwickelte Ansatz auch für praktische Anwendungen eingesetzt werden kann. Abschließend werden wir offene Fragen und noch zu behandelnde Problemstellungen im Kontext dieser Arbeit beleuchten.

10.1 Ergebnisse

Wir wollen in diesem Abschnitt die Ergebnisse dieser Dissertation zusammenfassen. Die Hauptaufgabe bestand darin, ein logikbasiertes Framework zu entwickeln, mit dessen Hilfe Statecharts-Spezifikationen, die auf unendlichen Datenräumen definiert sind, maschinengestützt verifiziert werden können.

Wir haben hierfür eine Formalisierung in dem Theorembeweiser Isabelle/HOL angegeben, in der Statecharts-Spezifikationen mit unendlichen Datenräumen durch Hierarchische Automaten beschrieben werden können. Darauf aufbauend haben wir eine Theorie entwickelt, in der der Datenraum einer Statecharts-Spezifikation für eine gegebene Abstraktionsfunktion struktur- und eigenschaftserhaltend abstrahiert werden kann. Hierbei wird das entstehende abstrakte Modell wieder durch eine Statecharts-Spezifikation repräsentiert. Ferner haben wir den Model Checker SMV an die Orakelschnittstelle von Isabelle/HOL angebunden, um für Statecharts-Spezifikationen, die auf einem endlichen Datenraum basieren, temporale Eigenschaften des universalen Fragments der CTL mit Hilfe von SMV effizient nachweisen zu können. Im Einzelnen sind folgende Ergebnisse entstanden.

Abstrakte Syntax und Semantik Hierarchischer Automaten: Wir haben in Isabelle/HOL Theorien angegeben, in denen sowohl die abstrakte Syntax, als auch die Semantik von Statecharts auf Basis Hierarchischer Automaten formalisiert sind. Wir orientierten uns hierbei an einer Beschreibung aus der Literatur [Mik00], für die es bisher noch keine Formalisierung in einem Theorembeweiser gab. Hierarchische Automaten erlauben eine strukturelle Zerlegung von Statecharts in so genannte Sequentielle Automaten. Diese strukturelle Zerlegung ermöglicht es, Beweise effizient und übersichtlich zu führen.

Optimierung durch baumbasierte Repräsentationen: Um die entwickelte Formalisierung von Statecharts für Beweise in Isabelle/HOL effizienter zu gestalten, entwickelten wir zwei Optimierungen. Zum einen haben wir Konstruktionsoperatoren eingeführt, mit denen Hierarchische Automaten schrittweise aus ihren sie definierenden Sequentiellen

Automaten zusammengesetzt werden können. Zum anderen haben wir eine alternative Formalisierung Hierarchischer Automaten angegeben, in der die baumartige Struktur eines Hierarchischen Automaten durch einen primitiv-rekursiven Datentyp repräsentiert worden ist.

1. **Konstruktionsoperatoren:** Für Hierarchische Automaten, die mit den Konstruktionsoperatoren definiert sind, leiteten wir Theoreme ab, die angeben, unter welchen Bedingungen eine solche Konstruktion zu einem wohlgeformten Hierarchischen Automaten führt. Auf Basis dieser Theoreme entwickelten wir eine Taktik, mit der die Wohlgeformtheit eines beliebigen, mit den Konstruktionsoperatoren definierten Hierarchischen Automaten automatisiert abgeleitet werden kann.
2. **Primitiv-rekursive Datentypen:** Sowohl in der abstrakten Syntax als auch in der Semantik haben wir die baumartige Struktur Hierarchischer Automaten durch primitive-rekursive Datentypen beschrieben. Dadurch erhalten wir die Möglichkeit, rekursive Konstantendefinitionen anzugeben, mit denen die Syntax und Semantik Hierarchischer Automaten formalisiert werden kann. Ferner kann die Ableitung von Theoremen über Hierarchischen Automaten auf Basis der primitiv-rekursiven Datentypen durch effiziente Induktionsbeweise unterstützt werden. Schließlich haben wir gezeigt, dass diese alternative Formalisierung Hierarchischer Automaten die Eigenschaften der ursprünglichen Formalisierung von Erich Mikk erfüllt.

Partitionierte Datenräume: Wir haben Theorien in Isabelle/HOL entwickelt, in denen Datenräume Hierarchischer Automaten formalisiert sind. Auf Basis eines polymorphen Grundtypen haben wir partitionierte Datenräume, vollständige und partielle Datenraumbelegungen, sowie totale und partielle Update-Funktion eingeführt. Mit diesen Definitionen sind wir in der Lage, das komplizierte Verhalten auf den Datenräumen Hierarchischer Automaten mathematisch präzise zu formulieren. Insbesondere haben wir eine Beschreibung der Interleaving-Semantik beim konkurrierenden Schreiben synchron auszuführender Update-Funktion angegeben.

Abstrakte Syntax und Semantik der temporalen Logik CTL: Wir haben Theorien für Isabelle/HOL angegeben, in denen die Operatoren der temporalen Logik CTL und ihre Semantik beschrieben sind. Um die Gültigkeit einer CTL-Formel semantisch zu bestimmen, haben wir zwei alternative Formalisierungen angegeben. Zum einen definierten wir eine induktive Semantik auf Basis von gültigen Berechnungspfaden in einer Kripke-Struktur. Zum anderen definierten wir für jeden CTL-Operator eine auf Fixpunkten des μ-Kalküls basierende Semantik. Beide Semantik-Definitionen sind in der Literatur weit verbreitet. Ferner definierten wir einen Operator, mit dem aus einem Hierarchischen Automaten eine sein Verhalten definierende Kripke-Struktur konstruiert werden kann. Wir haben gezeigt, dass diese Konstruktion die Wohlgeformtheitseigenschaften einer Kripke-Struktur erfüllt.

Konzeption einer Abstraktionstheorie: Wir haben ein Abstraktionskonzept vorgestellt, mit dem der Datenraum eines Hierarchischen Automaten für eine gegebene Abstraktionsfunktion abstrahiert werden kann. Diese Abstraktionstechnik basiert auf dem Bilden einer Überapproximation und ist bezüglich des universalen Fragments der temporalen Logik CTL eigenschaftserhaltend. Das Ergebnis der Abstraktion ist – analog zur Eingabe – durch einen Hierarchischen Automaten repräsentiert. Das Verfahren ist

strukturerhaltend, da die strukturellen Merkmale des zu abstrahierenden Hierarchischen Automaten bei der Abstraktion erhalten bleiben. Die Durchführung der Abstraktion erfolgt in kompositionaler Art und Weise. Zunächst wird hierfür der zu abstrahierende Hierarchische Automat in die ihn definierenden Sequentiellen Automaten zerlegt und diese separat voneinander abstrahiert. Anschließend werden die Abstraktionen der Sequentiellen Automaten wieder zu einem Hierarchischen Automaten zusammengefügt.

Konstruktion von Überapproximationen für Hierarchische Automaten: Wir haben eine Isabelle/HOL-Formalisierung angegeben, in der das von uns entwickelte Abstraktionskonzept umgesetzt ist. Wir formalisierten hierfür zunächst wohlgeformte Abstraktionsfunktionen, deren Eigenschaften garantieren, dass die Abstraktion eines Hierarchischen Automaten in kompositionaler Art und Weise erfolgen kann. Anschließend führten wir Operatoren ein, mit denen die Überapproximation eines Hierarchischen Automaten für eine gegebene Abstraktionsfunktion konstruiert werden kann.

Praktischer Umgang mit dem Framework: Wir haben zwei Taktiken implementiert, mit denen die maschinengestützte Analyse einer Statecharts-Spezifikation überwiegend automatisch erfolgen kann. Zum einen entwickelten wir eine Isabelle-Taktik, mit der die Gültigkeit einer CTL-Formel für einen Hierarchischen Automaten mit Hilfe des Model-Checkers SMV überprüft werden kann. Zum anderen stellten wir eine Isabelle-Taktik vor, mit der ein Algorithmus zur Datenabstraktion Hierarchischer Automaten in einen Beweisprozess von Isabelle eingebunden werden kann.

1. **Taktik zum Model-Checking:** Diese Taktik liest eine CTL-Formel und einen Hierarchischen Automaten aus der Isabelle-Termstruktur aus und übersetzt die gewonnenen Informationen in eine Eingabedatei für den Model-Checker SMV. Das zugrunde liegende Übersetzungsschema ist an Arbeiten von Erich Mikk angelehnt [Mik00] und wurde von uns um die Behandlung von Daten erweitert.
2. **Taktik zur Datenabstraktion:** Diese Taktik setzt eine Prädikatenabstraktion zur Abstraktion von Datenräumen Hierarchischer Automaten algorithmisch um. In dem Algorithmus haben wir das von uns vorgeschlagene Abstraktionskonzept für Hierarchische Automaten implementiert. Die algorithmische Abstraktion von Prädikaten basiert auf vorgeschlagenen Arbeiten aus der Literatur [SS99]. Zur Auswertung atomarer Aussagen haben wir den Beweiser SVC verwendet.

Mit Hilfe der entwickelten Taktiken haben wir eine Fallstudie verifiziert, um die Praktikabilität unseres Ansatzes zu belegen. Es handelt sich hierbei um eine Statecharts-Spezifikation, mit der die Steuerung eines Kühlungskreislaufs in einem nuklearen Kraftwerk modelliert wird.

In Abbildung 10.1 sind alle im Rahmen dieser Dissertation entwickelten Isabelle/HOL-Theorien graphisch dargestellt. Diese Theorien haben wir durch das Prinzip der konservativen Erweiterung aus den in der Standard-Distribution von Isabelle/HOL verfügbaren Theorien abgeleitet. In Tabelle 10.1 sind die Bezeichner der Isabelle-Theorien den in unserer Arbeit behandelten Konzepten zugeordnet. Die Theorien enthalten insgesamt etwas mehr als 200 Typ- und Konstantendefinitionen. Auf Basis dieser Definitionen haben wir etwa 450 Theoreme abgeleitet.

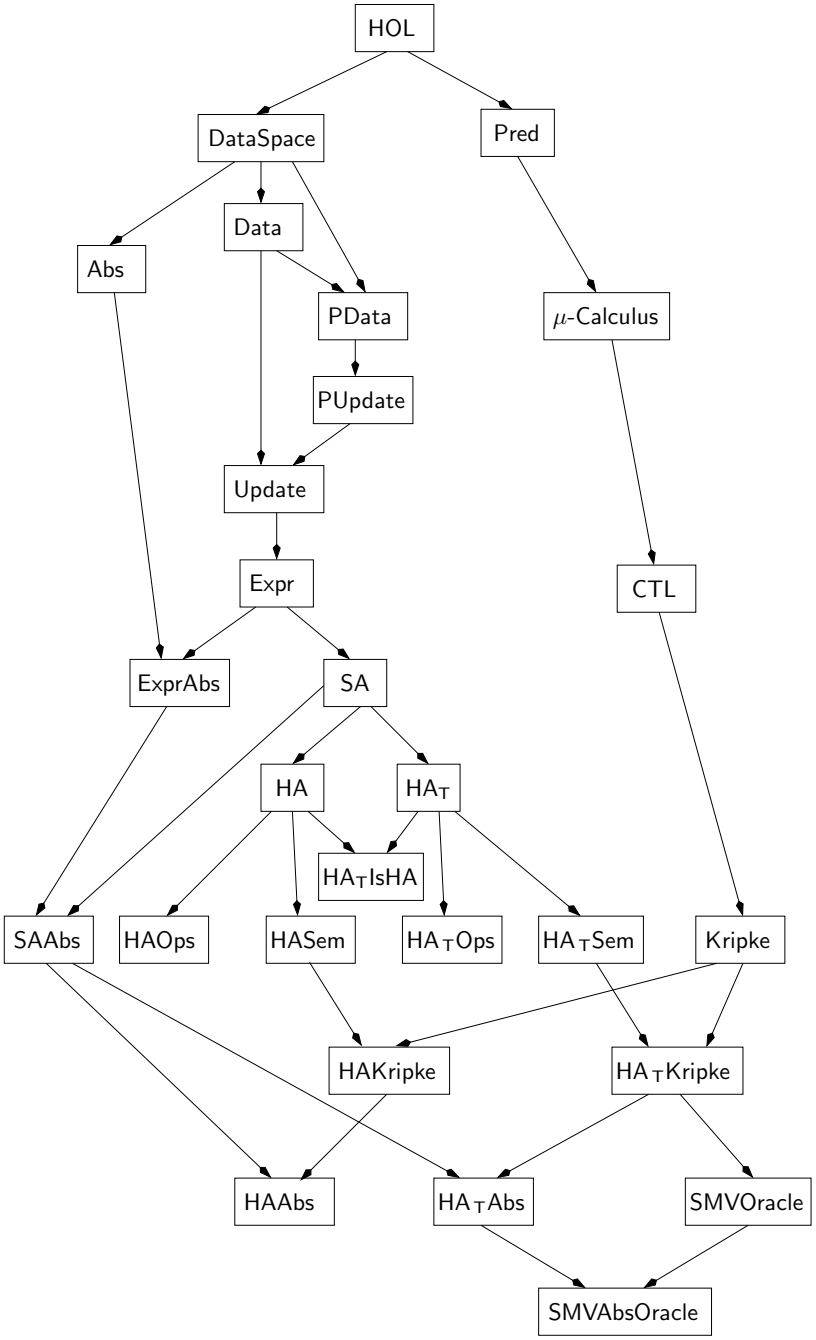


Abbildung 10.1: Abhängigkeiten zwischen den entwickelten Isabelle-Theorien

Tabelle 10.1: Zuordnung umgesetzter Konzepte zu entwickelten Theorien

Konzept	Theorien zur Umsetzung
Datenräume	DataSpace, Data, PData, Update, PUpdate
Sequentielle Automaten	SA, Expr
Hierarchische Automaten	HA, HASem, HAOps
Baumbasierte Hierarchische Automaten	HA_T, HA_TSem, HA_TOps, HA_TIsHA
Kripke-Strukturen und CTL	Kripke, HAKripke, HA_TKripke, CTL
Datenabstraktion	Abs, ExprAbs, SAAbs, HAAbs, HA_TAbs

Ferner wurden die in dieser Dissertation vorgestellten Ergebnisse auf nationalen und internationalen Konferenzen vorgetragen und veröffentlicht [HNSS00, HK01a, HK01b, HK03a, HK03b, HK05].

10.2 Diskussion

Die Verifikation von Statecharts-Spezifikationen durch geeignete Software zu unterstützen, ist ein offenes Forschungsgebiet. Insbesondere die Entwicklung von geeigneten Abstraktionstechniken, die möglichst automatisiert angewendet werden können, ist eine wichtige Aufgabe, der wir uns in dieser Dissertation gestellt haben. Ziel war es, ein logikbasiertes Framework zur Verfügung zu stellen, mit dem datenbehaftete Statecharts-Spezifikationen verifiziert werden können. Im letzten Abschnitt haben wir dargestellt, mit welchen Ergebnissen wir dieses Ziel erreicht haben. Diese Ergebnisse wollen wir in Bezug auf drei Kriterien diskutieren.

Semantische Fundierung: Ein besonderer Wert unserer Arbeit besteht darin, dass wir sowohl die Syntax als auch die komplexe Semantik von Statecharts in dem Theorembeweiser Isabelle/HOL formalisiert haben. Da wir davon ausgehen können, dass die rigorose Beweisführung in dem verwendeten Theorembeweiser korrekt ist, wird durch dieses Ergebnis ein erheblicher Zugewinn gegenüber einer Semantik-Beschreibung auf Papier erreicht, wie sie von Erich Mikk für Statecharts ohne Daten bereits entwickelt worden war. Ferner konnten wir für eine optimierte Formalisierung von Statecharts in Isabelle/HOL nachweisen, dass sie die Eigenschaften der originalen Beschreibung von Erich Mikk erhält. Dieses Ergebnis zeigt, wie theoretische Vorarbeiten durchgängig in eine mechanisierte Form gebracht werden können, ohne dabei auf Optimierungen zu verzichten.

Praktikabilität: Um die Praktikabilität unseres Ansatzes zu verbessern, haben wir Taktiken entwickelt, mit denen das interaktive Beweisvorgehen innerhalb eines Theorembeweisers automatisiert werden kann. Es war jedoch dabei nicht unser Ziel, Algorithmen zur Verfügung zu stellen, die auf Effizienz optimiert und bezüglich dieser mit praxisnahen Implementierungen zur Verifikation verglichen werden könnten. Vielmehr wollten

wir mit dieser Dissertation eine Verbindung zwischen Grundlagenforschung und anwendungsbezogener Forschung herstellen. Wir wählten deshalb einerseits einen Theorembeweiser aus, der überwiegend im akademischen Umfeld eingesetzt wird. Wir versuchten aber andererseits auch, durch die Einbindung von Taktiken die Praktikabilität zu steigern. Wir sind mit den Taktiken dazu in der Lage, einen praxistauglichen Model-Checker und einen Algorithmus zur Datenabstraktion für die maschinengestützte Analyse von Statecharts zu verwenden.

Benutzbarkeit: Ein besonderes Merkmal des von uns entwickelten Abstraktionskonzepts besteht gegenüber anderen Ansätzen in der Transparenz des Verfahrens. Wir haben diese Transparenz dadurch erreicht, dass das Ergebnis der Abstraktion in Analogie zur Eingabe durch einen Hierarchischen Automaten repräsentiert wird. Dadurch kann der Benutzer unserer Technik auf Basis eines ihm bekannten Formalismus prüfen, wieviel Verhalten durch die angegebene Abstraktionsfunktion dem ursprünglichen Modell hinzugefügt worden ist. Nachteil dieses Ansatzes ist allerdings, dass wir uns bei der Eigenschaftserhaltung auf das universale Fragment der CTL einschränken mussten. Eine Verhaltensreduktion wäre in vielen Fällen nicht mehr durch einen Hierarchischen Automaten zu repräsentieren gewesen.

10.3 Ausblick

In diesem Abschnitt wollen wir einige Überlegungen zu noch offenen Fragestellungen im Zusammenhang mit dieser Dissertation anstellen.

Verfeinerungskonzepte: Der von uns entwickelte Ansatz basiert auf einem Verfahren zur Datenabstraktion von Statecharts-Spezifikationen. In der Literatur werden Abstraktionstechniken häufig mit einem Verfahren zur schrittweisen Verfeinerung kombiniert. Hierbei wird in der Regel beim gescheiterten Nachweis einer temporalen Formel auf dem abstrakten Modell das zurückgegebene Gegenbeispiel genauer analysiert. Die Überprüfung erfolgt im Hinblick darauf, ob das zurückgegebene Gegenbeispiel tatsächlich im konkreten Modell auftreten kann. Sollte dies nicht der Fall sein, ist das Gegenbeispiel dem Verhalten zuzuordnen, das beim Bilden der Abstraktion zum ursprünglichen Modell hinzugefügt wurde. Mit einem geeigneten Verfahren zur Verfeinerung von Statecharts-Spezifikation wären wir dann in der Lage, das abstrakte Modell so zu verfeinern, dass das Gegenbeispiel nicht mehr zum zulässigen Verhalten des abstrakten Modells gehört. Dadurch wäre es möglich Eigenschaften zu verifizieren, die im bisherigen Verfahren nicht beweisbar sind.

Konstruktionsoperatoren zur Abstraktion: Die von uns im Rahmen dieser Dissertation entwickelte Isabelle-Theorie zur Datenabstraktion Hierarchischer Automaten enthält Konstruktionsoperatoren, mit denen die Überapproximation eines Hierarchischen Automaten definiert werden kann. Diese Konstruktionen enthalten Ausdrücke mit existenzquantifizierten Variablen, die ohne eine vorherige Simplifizierung nicht in die Eingabesprache eines Model-Checkers transformiert werden können. Im Rahmen einer Diplomarbeit haben wir erste Erfahrungen beim Simplifizieren der Konstruktionsoperatoren mit Hilfe eines *Contraint-Lösers* gesammelt [Sau06]. In dieser Arbeit wurde eine Tak-

tik entwickelt, mit der der Contraint-Löser Eclipse [AW06] in einen Isabelle-Beweisablauf eingebunden werden kann. In künftigen Arbeiten wollen wir auf Basis dieses Prototypen die Simplifizierung von Konstruktionsoperatoren zur Abstraktion automatisieren.

Anhang A
Isabelle-Theorien

Dieser Anhang enthält lediglich eine kleine Auswahl der in Isabelle/HOL zur Formalisierung Hierarchischer Automaten angegebenen Theorien.

A.1 Sequentielle Automaten

Nichtleerheit des Typs von SAs

$$(\,\{@\,S.\ \mathit{true}\},@\,S.\ \mathit{true},\varnothing,\varnothing\,)\in\{(S,I,L,T).\ \mathit{SeqAutoCorrect}\ S\ I\ L\ T\}$$

(SeqAutoNonEmpty)

Deklarationen und Typdefinitionen

$$\begin{aligned}(\sigma,\epsilon,\delta)\,\mathit{seqauto}\ \equiv_{\mathsf{t}}\ \{\,&(S,I,L,T)\mid\\ &(S::\sigma\ \mathit{set})\\ &(I::\sigma)\\ &(L::(\sigma,\epsilon,\delta)\ \mathit{labels})\\ &(T::(\sigma,\epsilon,\delta)\ \mathit{trans}).\\ &\mathit{SeqAutoCorrect}\ S\ I\ L\ T\,\}\\ &\quad \textit{gerechtfertigt durch}\ \text{SeqAutoNonEmpty}\end{aligned}$$

$$\begin{aligned}\mathit{States}\ &::_{\mathsf{c}}\ (\sigma,\epsilon,\delta)\,\mathit{seqauto}\to\sigma\ \mathit{set}\\ \mathit{States}\ &\equiv_{\mathsf{c}}\ \mathit{fst}\circ\mathit{Rep\_seqauto}\end{aligned}$$

$$\begin{aligned}\mathit{InitState}\ &::_{\mathsf{c}}\ (\sigma,\epsilon,\delta)\,\mathit{seqauto}\to\sigma\\ \mathit{InitState}\ &\equiv_{\mathsf{c}}\ \mathit{fst}\circ\mathit{snd}\circ\mathit{Rep\_seqauto}\end{aligned}$$

$$\begin{aligned}\mathit{Label}\ &::_{\mathsf{c}}\ (\sigma,\epsilon,\delta)\,\mathit{seqauto}\to(\sigma,\epsilon,\delta)\,\mathit{label}\ \mathit{set}\\ \mathit{Label}\ &\equiv_{\mathsf{c}}\ \mathit{fst}\circ\mathit{snd}\circ\mathit{snd}\circ\mathit{Rep\_seqauto}\end{aligned}$$

$$\begin{aligned}\mathit{Trans}\ &::_{\mathsf{c}}\ (\sigma,\epsilon,\delta)\,\mathit{seqauto}\to(\sigma,\epsilon,\delta)\,\mathit{trans}\ \mathit{set}\\ \mathit{Trans}\ &\equiv_{\mathsf{c}}\ \mathit{snd}\circ\mathit{snd}\circ\mathit{snd}\circ\mathit{Rep\_seqauto}\end{aligned}$$

Theoreme

$$Rep\_seqauto\ SA = (States\ SA, InitState\ SA, Label\ SA, Trans\ SA) \quad \text{(RepSeqAutoTuple)}$$

$$SeqAutoCorrect(States\ SA)\,(InitState\ SA)\,(Label\ SA)\,(Trans\ SA) \quad \text{(RepSeqAutoSelect)}$$

$$States\ SA \neq \varnothing \quad \text{(StatesNonEmpty)}$$

$$InitState\ SA \in States\ SA \quad \text{(InitStateIsStates)}$$

A.2 Hierarchische Automaten

Nichtleerheit des Typs von HA

$$(\{Abs\_seqauto(\{@\,S.\ true\}, @\,S.\ true, \varnothing, \varnothing)\}, \varnothing, EmptyMap\ \{@\,S.\ true\}, @\,D.\ true) \in \{(S_A, E, F_{Comp}, D).\ HierAutoCorrect\ S_A\ E\ F_{Comp}\ D\} \quad \text{(HierAutoNonEmpty)}$$

Deklarationen und Typdefinitionen

$$\begin{aligned}(\sigma,\epsilon,\delta)\,hierauto \equiv_t \{\,&(S_A, E, F_{Comp}, D) \mid\\ &(S_A :: ((\sigma,\epsilon,\delta)\,seqauto\,set))\\ &(E :: \epsilon\ set)\\ &(F_{Comp} :: (\sigma \rightharpoonup ((\sigma,\epsilon,\delta)\,seqauto\,set)))\\ &(D :: \delta\ data).\\ &HierAutoCorrect\ S_A\ E\ F_{Comp}\ D\,\}\\ &\quad gerechtfertigt\ durch\ \text{HierAutoNonEmpty}\end{aligned}$$

$$\begin{aligned}SAs &::_c (\sigma,\epsilon,\delta)\,hierauto \to (\sigma,\epsilon,\delta)\,seqauto\,set\\ SAs &\equiv_c fst \circ Rep\_hierauto\end{aligned}$$

$$\begin{aligned}Events &::_c (\sigma,\epsilon,\delta)\,hierauto \to \epsilon\ set\\ Events &\equiv_c fst \circ snd \circ Rep\_hierauto\end{aligned}$$

$$\begin{aligned}CompFun &::_c (\sigma,\epsilon,\delta)\,hierauto \to (\sigma \rightharpoonup ((\sigma,\epsilon,\delta)\,seqauto\,set))\\ CompFun &\equiv_c fst \circ snd \circ snd \circ Rep\_hierauto\end{aligned}$$

$$\begin{aligned}InitData &::_c (\sigma,\epsilon,\delta)\,hierauto \to \delta\\ InitData &\equiv_c snd \circ snd \circ snd \circ Rep\_hierauto\end{aligned}$$

Theoreme

$$\mathit{Rep\_hierauto}\ \mathit{HA} = (\mathit{SAs}\ \mathit{HA}, \mathit{Events}\ \mathit{HA}, \mathit{CompFun}\ \mathit{HA}, \mathit{InitData}\ \mathit{HA}) \quad \text{(RepHierAutoTuple)}$$

$$\mathit{HierAutoCorrect}\,(\mathit{SAs}\ \mathit{HA})\,(\mathit{Events}\ \mathit{HA})\,(\mathit{CompFun}\ \mathit{HA})\,(\mathit{InitData}\ \mathit{HA}) \quad \text{(RepHierAutoSelect)}$$

A.3 Konstruktionsoperatoren Hierarchischer Automaten

Deklarationen und Typdefinitionen

$$\begin{aligned} \mathit{EmptyMap} &::_{\mathsf{c}} \sigma\ \mathit{set} \to ((\sigma \rightharpoonup (\sigma, \epsilon, \delta)\ \mathit{seqauto})\ \mathit{set}) \\ \mathit{EmptyMap}\ S\ s &\equiv_{\mathsf{c}} \text{if } s \in S \text{ then } (\mathit{Some}\ \varnothing) \text{ else } \mathit{None} \end{aligned}$$

$$\begin{aligned} \mathit{PseudoHA} &::_{\mathsf{c}} [(\sigma, \epsilon, \delta)\ \mathit{seqauto},\ \delta\ \mathit{data}] \to (\sigma, \epsilon, \delta)\ \mathit{hierauto} \\ \mathit{PseudoHA}\ \mathit{SA}\ D &\equiv_{\mathsf{c}} \mathit{Abs\_hierauto}(\{\mathit{SA}\}, \mathit{Events}\ \mathit{SA}, \mathit{EmptyMap}\ (\mathit{States}\ \mathit{SA}), D) \end{aligned}$$

$$\begin{aligned} \_ \oplus \_ &::_{\mathsf{c}} [\sigma \rightharpoonup (\sigma, \epsilon, \delta)\ \mathit{seqauto}\ \mathit{set},\ \sigma * (\sigma, \epsilon, \delta)\ \mathit{seqauto}] \\ &\qquad \to (\sigma \rightharpoonup (\sigma, \epsilon, \delta)\ \mathit{seqauto}\ \mathit{set}) \\ \_ \oplus \_ &\equiv_{\mathsf{c}} (\lambda\ F_{\mathtt{Comp}}\ (S, \mathit{SA})\ . \\ &\qquad \text{if } (S \in \mathit{dom}\ F_{\mathtt{Comp}} \wedge S \notin \mathit{States}\ \mathit{SA}) \\ &\qquad \text{then} \\ &\qquad\quad F_{\mathtt{Comp}} \oplus_{\mathtt{P}} \{S \mapsto (\mathit{the}\ (F_{\mathtt{Comp}}\ S)) \cup \{\mathit{SA}\}\} \oplus_{\mathtt{P}} (\mathit{EmptyMap}\ (\mathit{States}\ \mathit{SA})) \\ &\qquad \text{else} \\ &\qquad\quad F_{\mathtt{Comp}}) \end{aligned}$$

$$\begin{aligned} \_ \boxplus \_ &::_{\mathsf{c}} [(\sigma, \epsilon, \delta)\ \mathit{hierauto},\ \sigma * (\sigma, \epsilon, \delta)\ \mathit{seqauto}] \\ &\qquad \to (\sigma, \epsilon, \delta)\ \mathit{hierauto} \\ \_ \boxplus \_ &\equiv_{\mathsf{c}} (\lambda\ \mathit{HA}\ (S, \mathit{SA})\ . \\ &\qquad \text{let } S_{\mathtt{A}}^{'} = \{\ \mathit{SA}\ \} \cup (\mathit{SAs}\ \mathit{HA}); \\ &\qquad\quad E^{'} = \mathit{Events}\ \mathit{HA} \cup \mathit{Events}\ \mathit{SA}; \\ &\qquad\quad F_{\mathtt{Comp}}^{'} = \mathit{CompFun}\ \mathit{HA} \oplus (S, \mathit{SA}) \\ &\qquad\quad D^{'} = \mathit{InitData}\ \mathit{HA} \\ &\qquad \text{in } \mathit{Abs\_hierauto}(S_{\mathtt{A}}^{'}, E^{'}, F_{\mathtt{Comp}}^{'}, D^{'})) \end{aligned}$$

$$\begin{aligned}
&_ \boxdot _ ::_{\mathsf{c}} [(\sigma,\epsilon,\delta)\, hierauto,\ \sigma * (\sigma,\epsilon,\delta)\, hierauto] \\
&\qquad \rightarrow (\sigma,\epsilon,\delta)\, hierauto \\
&_ \boxdot _ \equiv_{\mathsf{c}} (\lambda\, HA^{\,1}\ (S, HA^{\,2})\ . \\
&\qquad \text{let } (S_{\mathtt{A}}^{1}, E^{1}, F_{\mathtt{Comp}}^{1}, D^{1}) = Rep_hierauto\,(HA^{\,1} \boxplus (S, Root_{\mathtt{SA}}\, HA^{\,2})) \\
&\qquad\quad (S_{\mathtt{A}}^{2}, E^{2}, F_{\mathtt{Comp}}^{2}, D^{2}) = Rep_hierauto\, HA^{\,2} \\
&\qquad\qquad S_{\mathtt{A}}^{'} = S_{\mathtt{A}}^{1} \cup S_{\mathtt{A}}^{2}; \\
&\qquad\qquad E^{'} = E^{1} \cup E^{2}; \\
&\qquad\qquad F_{\mathtt{Comp}}^{'} = F_{\mathtt{Comp}}^{1} \oplus_{\mathtt{P}} F_{\mathtt{Comp}}^{2} \\
&\qquad \text{in } Abs_hierauto\,(S_{\mathtt{A}}^{'}, E^{'}, F_{\mathtt{Comp}}^{'}, D^{1}))
\end{aligned}$$

Theoreme für Pseudo-HA-Konstruktionen

$$\frac{S_{\mathtt{A}}^{'} = \{SA\} \qquad E^{'} = Events\ SA \qquad F_{\mathtt{Comp}}^{'} = EmptyMap\ (States\ SA)}{HierAutoCorrect\ S_{\mathtt{A}}^{'}\ E^{'}\ F_{\mathtt{Comp}}^{'}\ D} \quad \text{(PseudoHACorrect)}$$

$$SAs\ (PseudoHA\ SA\ D) = \{SA\} \quad \text{(PseudoHASelectSAs)}$$

$$Events\ (PseudoHA\ SA\ D) = Events\ SA \quad \text{(PseudoHASelectEvents)}$$

$$CompFun(PseudoHA\ SA\ D) = EmptyMap(States\ SA) \quad \text{(PseudoHASelectCompFun)}$$

$$InitData\ (PseudoHA\ SA\ D) = D \quad \text{(PseudoHASelectInitData)}$$

Theoreme für $\boxplus$ - Konstruktionen

$$\frac{\begin{array}{c} States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA \\ S_{\mathtt{A}}^{'} = \{\ SA\ \} \cup (SAs\ HA) \qquad E^{'} = Events\ HA \cup Events\ SA \\ F_{\mathtt{Comp}}^{'} = CompFun\ HA \oplus (S, SA) \qquad D^{'} = InitData\ HA \end{array}}{HierAutoCorrect\ S_{\mathtt{A}}^{'}\ E^{'}\ F_{\mathtt{Comp}}^{'}\ D^{'}} \quad \text{(HAAddSACorrect)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{SAs\ (HA \boxplus (S, SA)) = \{SA\} \cup SAs\ HA} \quad \text{(HAAddSASelectSAs)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{Events\ (HA \boxplus (S, SA)) = Events\, SA \cup Events\ HA} \quad \text{(HAAddSASelectEvents)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{CompFun\ (HA \boxplus (S, SA)) = CompFun\ HA \oplus (S, SA)} \quad \text{(HAAddSASelectCompFun)}$$

$$\frac{States\ SA \cap States\ HA = \varnothing \qquad S \in States\ HA}{InitData\ (HA \boxplus (S, SA)) = InitData\ HA} \quad \text{(HAAddSASelectInitData)}$$

Theoreme für $\boxdot$ - Konstruktionen

$$\frac{States\ HA^1 \cap States\ HA^2 = \varnothing \qquad S \in States\ HA^1 \qquad D' = InitData\ HA^1 \\ HA = HA^1 \boxplus (S, Root_{\mathrm{SA}}\ HA^2) \qquad S'_{\mathrm{A}} = (SAs\ HA) \cup (SAs\ HA^2) \\ E' = Events\ HA \cup Events\ HA^2 \qquad F'_{\mathrm{Comp}} = CompFun\ HA \oplus_{\mathrm{P}} CompFun\ HA^2}{HierAutoCorrect\ S'_{\mathrm{A}}\ E'\ F'_{\mathrm{Comp}}\ D'}$$

(HAAddHACorrect)

$$\frac{States\ HA^1 \cap States\ HA^2 = \varnothing \qquad S \in States\ HA^1}{SAs\ (HA^1 \boxdot (S, HA^2)) = SAs\ HA^1 \cup SAs\ HA^2}$$

(HAAddHASelectSAs)

$$\frac{States\ HA^1 \cap States\ HA^2 = \varnothing \qquad S \in States\ HA^1}{Events\ (HA^1 \boxdot (S, HA^2)) = Events\ HA^1 \cup Events\ HA^2}$$

(HAAddHASelectEvents)

$$\frac{States\ HA^1 \cap States\ HA^2 = \varnothing \qquad S \in States\ HA^1}{CompFun\ (HA^1 \boxdot (S, HA^2)) = (CompFun\ HA^1\ \oplus\ (S, Root_{\mathrm{SA}}\ HA^2)) \oplus_{\mathrm{P}} CompFun\ HA^2}$$

(HAAddHASelectCompFun)

A.4 Baumbasierte Hierarchische Automaten

Deklarationen und Typdefinitionen für die Wohlgeformtheit

$$(\sigma, \epsilon, \delta)\ comptree \equiv_{\mathrm{t}} Node_{\mathrm{T}}\ (\sigma, \epsilon, \delta)\ seqauto\ (\sigma \to ((\sigma, \epsilon, \delta)\ comptree\ list))$$

$$RootSA_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ comptree \to (\sigma, \epsilon, \delta)\ seqauto$$
$$RootSA_{\mathrm{T}}\ (Node_{\mathrm{T}}\ SA\ F) = SA$$

$$RanSA_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ comptree \to (\sigma, \epsilon, \delta)\ seqauto\ set$$
$$RanSA_{\mathrm{TL}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ comptree\ list \to (\sigma, \epsilon, \delta)\ seqauto\ set$$

$$RanSA_{\mathrm{T}}\ (Node_{\mathrm{T}}\ SA\ F) = \left(\bigcup S : States\ SA.\ set(map\ RootSA_{\mathrm{T}}\ (F\ S))\right) \cup \left(\bigcup S : States\ SA.\ RanSA_{\mathrm{TL}}\ (F\ S)\right)$$

$$RanSA_{\mathrm{TL}}\ [\,] = \varnothing$$
$$RanSA_{\mathrm{TL}}\ (T \# L) = RanSA_{\mathrm{T}}\ T \cup RanSA_{\mathrm{TL}}\ L$$

$$RootExists_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ comptree \to bool$$
$$RootExists_{\mathrm{T}}\ T \equiv_{\mathrm{c}} \exists_1 A.\ A = (RootSA_{\mathrm{T}}\ T) \wedge A \notin (RanSA_{\mathrm{T}}\ T)$$

$$States_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree \to \sigma\, set$$
$$States_{\mathrm{TL}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree\, list \to \sigma\, set$$

$$States_{\mathrm{T}}\ (Node_{\mathrm{T}}\ SA\ F) = States\ SA \cup \Big(\bigcup S : States\ SA.\ States_{\mathrm{TL}}\,(F\ S)\Big)$$

$$States_{\mathrm{TL}}\ [\,] = \varnothing$$
$$States_{\mathrm{TL}}\ (T \# L) = States_{\mathrm{T}}\ T \cup States_{\mathrm{TL}}\ L$$

$$MutuallyDistinct_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree \to bool$$
$$MutuallyDistinct_{\mathrm{TL}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree\, list \to bool$$

$$\begin{aligned} MutuallyDistinct_{\mathrm{T}}\ (Node_{\mathrm{T}}\ SA\ F) = {} & States\ SA \cap \Big(\bigcup S : States\ SA.\ States_{\mathrm{TL}}\,(F\ S)\Big) = \varnothing \ \wedge \\ & \forall\, S_1, S_2 : States\ SA. \\ & \quad S_1 \neq S_2 \Rightarrow States_{\mathrm{TL}}\ (F\ S_1) \cap States_{\mathrm{TL}}\ (F\ S_2) = \varnothing \ \wedge \\ & \forall\, S : States\ SA.\ MutuallyDistinct_{\mathrm{TL}}\ (F\ S) \end{aligned}$$

$$\begin{aligned} MutuallyDistinct_{\mathrm{TL}}\ [\,] &= true \\ MutuallyDistinct_{\mathrm{TL}}\ (T \# L) = {} & States_{\mathrm{T}}\ T \cap States_{\mathrm{TL}}\ L = \varnothing \ \wedge \\ & MutuallyDistinct_{\mathrm{T}}\ T \ \wedge \\ & MutuallyDistinct_{\mathrm{TL}}\ L \end{aligned}$$

$$TSAs_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree \to (\sigma, \epsilon, \delta)\, seqauto\ set$$
$$TSAs_{\mathrm{T}}\ T \equiv_{\mathrm{c}} \{RootSA_{\mathrm{T}}\ T\} \cup RanSA_{\mathrm{T}}\ T$$

$$TSAs_{\mathrm{TL}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree\, list \to (\sigma, \epsilon, \delta)\, seqauto\ set$$
$$TSAs_{\mathrm{TL}}\ L \equiv_{\mathrm{c}} (set(map\ RootSA_{\mathrm{T}}\ L)) \cup RanSA_{\mathrm{TL}}\ L$$

$$IsCompTree ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\, comptree \to bool$$
$$IsCompTree\ T \equiv_{\mathrm{c}} MutuallyDistinct_{\mathrm{T}}\ T \ \wedge\ RootExists_{\mathrm{T}}\ T$$

$$HierAutoCorrect_{\mathrm{T}} ::_{\mathrm{c}} [\epsilon\, set, (\sigma, \epsilon, \delta)\, comptree, \delta\, data\,] \to bool$$
$$HierAutoCorrect_{\mathrm{T}}\ E\ T\ D \equiv_{\mathrm{c}} \Big(\bigcup A \in (TSAs_{\mathrm{T}}\ T).\ Events\ F\Big) \subseteq E \ \wedge\ IsCompTree\ T$$

Nichtleerheit des Typs baumbasierter HA

$$(\varnothing, \mathit{Node}_{\mathrm{T}}\ (\mathit{Abs\_seqauto}(\{@\,S.\ \mathit{true}\}, @\,S.\ \mathit{true}, \varnothing, \varnothing))\ (\lambda\ S.\ [])\,, @\,D.\ \mathit{true}) \in \{(E, T, D).\ \mathit{HierAutoCorrect}_{\mathrm{T}}\ E\ T\ D\}$$

(HierAutoTreeNonEmpty)

Deklarationen und Typdefinitionen

$$(\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}} \equiv_{\mathrm{t}} \{\, (E, T_{\mathtt{Comp}}, D) \mid (E :: \epsilon\ \mathit{set})\ (T_{\mathtt{Comp}} :: ((\sigma, \epsilon, \delta)\ \mathit{comptree}))\ (D :: \delta\ \mathit{data}).\ \mathit{HierAutoCorrect}_{\mathrm{T}}\ E\ T_{\mathtt{Comp}}\ D \,\}$$

gerechtfertigt durch HierAutoTreeNonEmpty

$$\mathit{Events}_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}} \to \epsilon\ \mathit{set}$$
$$\mathit{Events}_{\mathrm{T}} \equiv_{\mathrm{c}} \mathit{fst} \circ \mathit{Rep\_hierauto}_{\mathrm{T}}$$

$$\mathit{CompTree} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}} \to (\sigma, \epsilon, \delta)\ \mathit{comptree}$$
$$\mathit{CompTree} \equiv_{\mathrm{c}} \mathit{fst} \circ \mathit{snd} \circ \mathit{Rep\_hierauto}_{\mathrm{T}}$$

$$\mathit{InitData}_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}} \to \delta$$
$$\mathit{InitData}_{\mathrm{T}} \equiv_{\mathrm{c}} \mathit{snd} \circ \mathit{snd} \circ \mathit{Rep\_hierauto}_{\mathrm{T}}$$

$$\mathit{SAs}_{\mathrm{T}} ::_{\mathrm{c}} (\sigma, \epsilon, \delta)\ \mathit{hierauto}_{\mathrm{T}} \to (\sigma, \epsilon, \delta)\ \mathit{seqauto\ set}$$
$$\mathit{SAs}_{\mathrm{T}}\ \mathit{HA}_{\mathrm{T}} \equiv_{\mathrm{c}} \mathit{TSAs}_{\mathrm{T}}\,(\mathit{CompTree}\ \mathit{HA}_{\mathrm{T}})$$

A.5 Partitionierte Datenräume

Deklarationen, Typdefinitionen und Theoreme für Datenräume

$$[\,\mathsf{UNIV}\,] \in \{L.\ \mathit{DataSpaceCorrect}\ L\}$$

(DataSpaceNonEmpty)

$$\begin{aligned}\delta\ \mathit{dataspace} \equiv_{\mathsf{t}} \{\, L \mid\ & (L :: \delta\ \mathit{set}\ \mathit{list}).\\ & \mathit{DataSpaceCorrect}\ L \,\}\\ & \textit{gerechtfertigt durch } \mathsf{DataSpaceNonEmpty}\end{aligned}$$

$$\begin{aligned}\mathit{PartNum} &::_{\mathsf{c}} \delta\ \mathit{dataspace} \to \mathit{nat}\\ \mathit{PartNum} &\equiv_{\mathsf{c}} \mathit{length} \circ \mathit{Rep\_dataspace}\end{aligned}$$

$$\begin{aligned}\mathit{PartDom} &::_{\mathsf{c}} [\delta\ \mathit{dataspace}, \mathit{nat}] \to \delta\ \mathit{set}\\ \mathit{PartDom}\ D\ N &\equiv_{\mathsf{c}} (\mathit{Rep\_dataspace}\ D)\ !\ N\end{aligned}$$

Deklarationen, Typdefinitionen und Theoreme für Datenraumbelegungen

$$([\,@\ T.\ \mathit{true}], \mathit{Abs\_dataspace}\,[\,\mathsf{UNIV}\,]) \in \{(L, D).\ \mathit{DataCorrect}\ L\ D\}$$

(DataNonEmpty)

$$\begin{aligned}\delta\ \mathit{data} \equiv_{\mathsf{t}} \{\, (L, D) \mid\ & (L :: \delta\ \mathit{list})\\ & (D :: \delta\ \mathit{dataspace}).\\ & \mathit{DataCorrect}\ L\ D \,\}\\ & \textit{gerechtfertigt durch } \mathsf{DataNonEmpty}\end{aligned}$$

$$\begin{aligned}\mathit{DataValue} &::_{\mathsf{c}} \delta\ \mathit{data} \to \delta\ \mathit{list}\\ \mathit{DataValue} &\equiv_{\mathsf{c}} \mathit{fst} \circ \mathit{Rep\_data}\end{aligned}$$

$$\begin{aligned}\mathit{DataSpace} &::_{\mathsf{c}} \delta\ \mathit{data} \to \delta\ \mathit{dataspace}\\ \mathit{DataSpace} &\equiv_{\mathsf{c}} \mathit{snd} \circ \mathit{Rep\_data}\end{aligned}$$

$$\begin{aligned}\_\ !\ \_ &::_{\mathsf{c}} [\delta\ \mathit{data}, \mathit{nat}] \to \delta\\ \_\ !\ \_ &\equiv_{\mathsf{c}} (\lambda\ D\ N\ .\ (\mathit{DataValue}\ D)\ !\ N)\end{aligned}$$

Deklarationen, Typdefinitionen und Theoreme für partielle Datenraumbelegungen

$$([\,\mathit{Some}\,(\,@\;\;T.\;\mathit{true}\,)\,],\mathit{Abs\_dataspace}\,[\,\mathsf{UNIV}\,]) \in \{(L,D).\,\mathit{PDataCorrect}\;L\;D\}$$

(PDataNonEmpty)

$$\begin{aligned}
\delta\ \mathit{pdata} \equiv_{\mathsf{t}} \{\,&(L,D)\mid\\
&(L :: \delta\ \mathit{option\ list})\\
&(D :: \delta\ \mathit{dataspace}).\\
&\mathit{PDataCorrect}\ L\ D\,\}\\
&\quad \textit{gerechtfertigt durch}\ \mathsf{PDataNonEmpty}
\end{aligned}$$

$$\begin{aligned}
\mathit{DataValue} &::_{\mathsf{c}} \delta\ \mathit{pdata} \to (\delta\ \mathit{option})\ \mathit{list}\\
\mathit{DataValue} &\equiv_{\mathsf{c}} \mathit{fst} \circ \mathit{Rep_pdata}
\end{aligned}$$

$$\begin{aligned}
\mathit{DataSpace} &::_{\mathsf{c}} \delta\ \mathit{pdata} \to \delta\ \mathit{dataspace}\\
\mathit{DataSpace} &\equiv_{\mathsf{c}} \mathit{snd} \circ \mathit{Rep_pdata}
\end{aligned}$$

$$\begin{aligned}
\mathit{DataAsPData} &::_{\mathsf{c}} \delta\ \mathit{data} \to \mathit{pdata}\\
\mathit{DataAsPData}\ D &\equiv_{\mathsf{c}} \mathsf{let}\ \ (L, D_{\mathsf{P}}) = \mathit{Rep_data}\ D;\\
&\qquad\qquad\ \ L_{\mathsf{O}} \quad\ = \mathit{map}\ \mathit{Some}\ L\\
&\qquad\ \mathsf{in}\quad \mathit{Abs_pdata}(L_{\mathsf{O}}, D_{\mathsf{P}})
\end{aligned}$$

$$\begin{aligned}
\mathit{PDataAsData} &::_{\mathsf{c}} \delta\ \mathit{pdata} \to \mathit{data}\\
\mathit{PDataAsData}\ D_{\mathsf{P}} &\equiv_{\mathsf{c}} \mathsf{let}\ \ (L_{\mathsf{O}}, D) = \mathit{Rep_pdata}\ D_{\mathsf{P}};\\
&\qquad\qquad\ \ L \quad\ = \mathit{map}\ \mathit{the}\ L\\
&\qquad\ \mathsf{in}\quad \mathit{Abs_data}(L, D)
\end{aligned}$$

$$\begin{aligned}
\oplus_{\mathsf{O}} &::_{\mathsf{c}} (\delta\ \mathit{option} * \delta) \to \delta\\
\oplus_{\mathsf{O}} &\equiv_{\mathsf{c}} \lambda\ (D_{\mathsf{O}}, D_{\mathsf{T}})\ .\ \mathsf{if}\ \ D_{\mathsf{O}} = \mathit{None}\ \ \mathsf{then}\ \ D_{\mathsf{T}}\ \ \mathsf{else}\ \ \mathit{the}\ D_{\mathsf{O}}
\end{aligned}$$

$$\begin{aligned}
_ \oplus_{\mathsf{D}} _ &::_{\mathsf{c}} [\,\delta\ \mathit{pdata}, \delta\ \mathit{data}\,] \to \delta\ \mathit{data}\\
_ \oplus_{\mathsf{D}} _ &\equiv_{\mathsf{c}} \lambda\ D_{\mathsf{P}}\ D_{\mathsf{T}}\ .\\
&\qquad \mathsf{let}\ \ (L_{\mathsf{P}}, DS_{\mathsf{P}}) = \mathit{Rep_pdata}\ D_{\mathsf{P}};\\
&\qquad\qquad (L_{\mathsf{T}}, DS_{\mathsf{T}}) = \mathit{Rep_data}\ D_{\mathsf{T}};\\
&\qquad\qquad\qquad L = \mathit{map}\ \oplus_{\mathsf{O}}\ (\mathit{zip}\ L_{\mathsf{P}}\ L_{\mathsf{T}})\\
&\qquad \mathsf{in}\ \ \mathit{Abs_data}\,(L, DS_{\mathsf{P}})
\end{aligned}$$

Deklarationen, Typdefinitionen und Theoreme für Update-Funktionen

$$(\lambda\ D.\ D) \in \{L.\ \mathit{UpdateCorrect}\ L\}$$

(UpdateNonEmpty)

$$\begin{aligned}\delta\ \mathit{update} \equiv_{\mathtt{t}} \{\, U \mid\ & (U :: \delta\ \mathit{data} \to \delta\ \mathit{data})\,. \\ & \mathit{UpdateCorrect}\ U\,\} \\ & \textit{gerechtfertigt durch}\ \text{UpdateNonEmpty}\end{aligned}$$

$$\begin{aligned}\_\ !\ \_ &::_{\mathtt{c}} [\delta\ \mathit{update}, \delta\ \mathit{data}] \to \delta\ \mathit{data} \\ \_\ !\ \_ &\equiv_{\mathtt{c}} (\lambda\ U\ D\,.\,(\mathit{Rep\_update}\ U)\ D)\end{aligned}$$

$$\begin{aligned}\mathit{UpdateDefault} &::_{\mathtt{c}} (\delta\ \mathit{update}) \\ \mathit{UpdateDefault} &\equiv_{\mathtt{c}} \mathit{Abs\_update}(\lambda\ D.\ D)\end{aligned}$$

Deklarationen, Typdefinitionen und Theoreme für Partielle Update-Funktionen

$$\mathit{DataAsPData} \in \{L.\ \mathit{PUpdateCorrect}\ L\}$$

(PUpdateNonEmpty)

$$\begin{aligned}\delta\ \mathit{pupdate} \equiv_{\mathtt{t}} \{\, U \mid\ & (U :: \delta\ \mathit{data} \to \delta\ \mathit{pdata})\,. \\ & \mathit{PUpdateCorrect}\ U\,\} \\ & \textit{gerechtfertigt durch}\ \text{PUpdateNonEmpty}\end{aligned}$$

$$\begin{aligned}\_\ !!\ \_ &::_{\mathtt{c}} [\delta\ \mathit{pupdate}, \delta\ \mathit{data}] \to \delta\ \mathit{pdata} \\ \_\ !!\ \_ &\equiv_{\mathtt{c}} (\lambda\ U\ D\,.\,(\mathit{Rep\_pupdate}\ U)\ D)\end{aligned}$$

$$\begin{aligned}\mathit{PUpdateDefault} &::_{\mathtt{c}} (\delta\ \mathit{pupdate}) \\ \mathit{PUpdateDefault} &\equiv_{\mathtt{c}} \mathit{Abs\_update}(\lambda\ D. \\ &\qquad \mathit{Abs\_pupdate}(\mathit{replicate}(\mathit{PartNum}(\mathit{DataSpace}D))\ \mathit{None}, \\ &\qquad\qquad \mathit{DataSpace}D))\end{aligned}$$

$$\begin{aligned}\_ \oplus_{\mathtt{U}} \_ &::_{\mathtt{c}} [\delta\ \mathit{pupdate}, \delta\ \mathit{update}] \to \delta\ \mathit{update} \\ \_ \oplus_{\mathtt{U}} \_ &\equiv_{\mathtt{c}} \lambda\ U_{\mathtt{P}}\ U_{\mathtt{T}}\,.\ \lambda\ D.\ (U_{\mathtt{P}}\ !!\ D) \oplus_{\mathtt{D}} (U_{\mathtt{T}}\ !\ D)\end{aligned}$$

$$\begin{aligned} \mathit{SolveRacing}_{\mathtt{U}} &::_{\mathtt{c}} (\delta\, \mathit{pupdate}\ \mathit{set}) \to (\delta\, \mathit{update}\ \mathit{set}) \\ \mathit{SolveRacing}_{\mathtt{U}}\ S_{\mathtt{U}} &\equiv_{\mathtt{c}} \{U.\ (S_{\mathtt{U}}, U) \in \mathsf{foldset}\ \ \oplus_{\mathtt{U}}\ \ \mathit{UpdateDefault}\} \end{aligned}$$

A.6 Prädikate auf Basis boolescher Funktionen

Deklarationen und Typdefinitionen

$$\sigma\ \mathit{pred} \equiv_{\mathtt{s}} \sigma \to \mathit{bool}$$

$$\begin{aligned} \mathit{Valid} &::_{\mathtt{c}} \sigma\ \mathit{pred} \to \mathit{bool} \\ \mathit{Valid}\, P &\equiv_{\mathtt{c}} \forall s.\ P\ s \end{aligned}$$

$$\begin{aligned} \_ \wedge_{\mathtt{P}} \_ &::_{\mathtt{c}} [\sigma\ \mathit{pred}, \sigma\ \mathit{pred}] \to \sigma\ \mathit{pred} \\ P \wedge_{\mathtt{P}} Q &\equiv_{\mathtt{c}} \lambda\, U.\ P\ U\ \wedge\ Q\ U \end{aligned}$$

$$\begin{aligned} \_ \vee_{\mathtt{P}} \_ &::_{\mathtt{c}} [\sigma\ \mathit{pred}, \sigma\ \mathit{pred}] \to \sigma\ \mathit{pred} \\ P \vee_{\mathtt{P}} Q &\equiv_{\mathtt{c}} \lambda\, U.\ P\ U\ \vee\ Q\ U \end{aligned}$$

$$\begin{aligned} \_ \Rightarrow_{\mathtt{P}} \_ &::_{\mathtt{c}} [\sigma\ \mathit{pred}, \sigma\ \mathit{pred}] \to \sigma\ \mathit{pred} \\ P \Rightarrow_{\mathtt{P}} Q &\equiv_{\mathtt{c}} \lambda\, U.\ P\ U\ \Rightarrow\ Q\ U \end{aligned}$$

$$\begin{aligned} \neg_{\mathtt{P}} &::_{\mathtt{c}} \sigma\ \mathit{pred}, \to \sigma\ \mathit{pred} \\ \neg_{\mathtt{P}}\ P &\equiv_{\mathtt{c}} \lambda\, U.\ \neg\ P\ U \end{aligned}$$

Anhang B
Beispiele für Statecharts in Isabelle/HOL

B.1 Musikanlage im Automobil

Formalisierung des Datenraumes

$$\begin{aligned} ds \equiv_{\mathsf{t}} \; & V_0\ int \\ \mid \; & V_1\ int \end{aligned}$$

$$\begin{aligned} Sel_0 &::_{\mathsf{c}} ds \to int \\ Sel_0\,(V_0\,i) &= i \end{aligned}$$

$$\begin{aligned} Sel_1 &::_{\mathsf{c}} ds \to int \\ Sel_1\,(V_1\,i) &= i \end{aligned}$$

$$\begin{aligned} Select_0 &::_{\mathsf{c}} ds\,data \to int \\ Select_0\,D &\equiv_{\mathsf{c}} Sel_0\,(PartData\,D\,0) \end{aligned}$$

$$\begin{aligned} Select_1 &::_{\mathsf{c}} ds\,data \to int \\ Select_1\,D &\equiv_{\mathsf{c}} Sel_1\,(PartData\,D\,1) \end{aligned}$$

$$\begin{aligned} DS &::_{\mathsf{c}} ds\ dataspace \\ DS &\equiv_{\mathsf{c}} Abs\_dataspace\,[\,ran\ V_0, ran\ V_1\,] \end{aligned}$$

$$\begin{aligned} LiftInitData &::_{\mathsf{c}} ds\ list \to ds\,data \\ LiftInitData\,L &\equiv_{\mathsf{c}} Abs\_data(L, DS) \end{aligned}$$

$$\begin{aligned} InitDS &::_{\mathsf{c}} ds\,data \\ InitDS &\equiv_{\mathsf{c}} LiftInitData\,[V_0\,0, V_0\,1\,] \end{aligned}$$

$$\begin{aligned} LiftPUpdate &::_{\mathsf{c}} (ds\,data \to (ds\,option)\ list) \to ds\,pupdate \\ LiftPUpdate\,L &\equiv_{\mathsf{c}} Abs\_pupdate(\lambda\ D.\ \text{if}\ ((DataSpace\,D) = DS) \\ &\qquad\qquad \text{then}\ Abs\_pdata(L\ D, DS) \\ &\qquad\qquad \text{else}\ (DataAsPData\,D)) \end{aligned}$$

Formalisierung des Wurzelautomaten *CarAudioSystemCTRL*

$$
\begin{aligned}
\mathit{CarAudioSystemStates} &::_c \mathit{string\ set}\\
\mathit{CarAudioSystemStates} &\equiv_c \{\text{“}\mathit{CarAudioSystem}\text{”}\}\\
\\
\mathit{CarAudioSystemInitState} &::_c \mathit{string}\\
\mathit{CarAudioSystemInitState} &\equiv_c \text{“}\mathit{CarAudioSystem}\text{”}\\
\\
\mathit{CarAudioSystemLabels} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
\mathit{CarAudioSystemLabels} &\equiv_c \varnothing\\
\\
\mathit{CarAudioSystemTrans} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
\mathit{CarAudioSystemTrans} &\equiv_c \varnothing\\
\\
\mathit{CarAudioSystemCTRL} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
\mathit{CarAudioSystemCTRL} &\equiv_c \mathit{Abs\_seqauto}\ (\mathit{CarAudioSystemStates},\\
&\qquad\qquad \mathit{CarAudioSystemInitState},\\
&\qquad\qquad \mathit{CarAudioSystemLabels},\\
&\qquad\qquad \mathit{CarAudioSystemTrans})
\end{aligned}
$$

Formalisierung des Sequentiellen Automaten *AudioPlayer*

$$
\begin{aligned}
\mathit{AudioPlayerStates} &::_c \mathit{string\ set}\\
\mathit{AudioPlayerStates} &\equiv_c \{\text{“}\mathit{On}\text{”}, \text{“}\mathit{Off}\text{”}\}\\
\\
\mathit{AudioPlayerInitState} &::_c \mathit{string}\\
\mathit{AudioPlayerInitState} &\equiv_c \text{“}\mathit{Off}\text{”}\\
\\
\mathit{AudioPlayerLabels} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
\mathit{AudioPlayerLabels} &\equiv_c \{(\mathit{En}\ \text{“}\mathit{O}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault})\}\\
\\
\mathit{AudioPlayerTrans} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
\mathit{AudioPlayerTrans} &\equiv_c\\
&\quad \{(\text{“}\mathit{Off}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{O}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault})\\
&\quad\ \text{“}\mathit{On}\text{”}),\\
&\quad (\text{“}\mathit{On}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{O}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault})\\
&\quad\ \text{“}\mathit{Off}\text{”})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{AudioPlayer} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{AudioPlayer} \equiv_c \mathit{Abs\_seqauto}\ (\mathit{AudioPlayerStates}, \mathit{AudioPlayerInitState},\\
&\qquad\qquad \mathit{AudioPlayerLabels}, \mathit{AudioPlayerTrans})
\end{aligned}
$$

Formalisierung des Sequentiellen Automaten *CDPlayer*

$$
\begin{aligned}
&\mathit{CDPlayerStates} ::_c \mathit{string\ set}\\
&\mathit{CDPlayerStates} \equiv_c \{\, \mathit{“CDFull”}, \mathit{“CDEmpty”}, \mathit{“ReadTracks”} \,\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDPlayerInitState} ::_c \mathit{string}\\
&\mathit{CDPlayerInitState} \equiv_c \mathit{“CDEmpty”}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UpdateIncT} ::_c \mathit{ds\ pupdate}\\
&\mathit{UpdateIncT} \equiv_c \mathit{LiftPUpdate}(\lambda\ D.\ [\,\mathit{Some}(V_0\,((\mathit{Select}_0\,D)+1)), \mathit{None}\,])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UpdateZeroT} ::_c \mathit{ds\ pupdate}\\
&\mathit{UpdateZeroT} \equiv_c \mathit{LiftPUpdate}(\lambda\ D.\ [\,\mathit{Some}(V_0\,0), \mathit{None}\,])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDPlayerLabels} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
&\mathit{CDPlayerLabels} \equiv_c \{\, (\mathit{En}\ \mathit{“CDIn”}, \mathit{GuardDefault}, \mathit{UpdateIncT}),\\
&\qquad\qquad (\mathit{And}\ (\mathit{En}\ \mathit{“CDEject”})\ (\mathit{In}\ \mathit{“On”}), \mathit{GuardDefault},\\
&\qquad\qquad\qquad \mathit{UpdateZeroT}),\\
&\qquad\qquad (\mathit{En}\ \mathit{“LastTrack”}, \mathit{GuardDefault}, \mathit{UpdateIncT}),\\
&\qquad\qquad (\mathit{En}\ \mathit{“NewTrack”}, \mathit{GuardDefault}, \mathit{UpdateIncT})\,\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDPlayerTrans} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
&\mathit{CDPlayerTrans} \equiv_c\\
&\qquad \{\, (\mathit{“CDEmpty”},\\
&\qquad\quad (\mathit{En}\ \mathit{“CDIn”}, \mathit{GuardDefault}, \mathit{ActionDefault})\\
&\qquad\quad \mathit{“ReadTracks”}),\\
&\qquad (\mathit{“CDFull”},\\
&\qquad\quad (\mathit{And}\ (\mathit{En}\ \mathit{“CDEject”})\ (\mathit{In}\ \mathit{“On”}), \mathit{GuardDefault},\\
&\qquad\qquad\qquad \mathit{UpdateZeroT}),\\
&\qquad\quad \mathit{“CDEmpty”})\\
&\qquad (\mathit{“ReadTracks”},\\
&\qquad\quad (\mathit{En}\ \mathit{“NewTrack”}, \mathit{GuardDefault}, \mathit{UpdateIncT})\\
&\qquad\quad \mathit{“ReadTracks”}),\\
&\qquad (\mathit{“ReadTracks”},\\
&\qquad\quad (\mathit{En}\ \mathit{“LastTrack”}, \mathit{GuardDefault}, \mathit{UpdateIncT}),\\
&\qquad (\mathit{“CDFull”}\,\}
\end{aligned}
$$

$$CDPlayer ::_c (string, string, ds)\ seqauto$$
$$CDPlayer \equiv_c Abs\_seqauto\ (CDPlayerStates, CDPlayerInitState, CDPlayerLabels, CDPlayerTrans)$$

Formalisierung des Sequentiellen Automaten *TapeDeck*

$$TapeDeckStates ::_c string\ set$$
$$TapeDeckStates \equiv_c \{\text{“}TapeFull\text{”}, \text{“}TapeEmpty\text{”}\}$$

$$TapeDeckInitState ::_c string$$
$$TapeDeckInitState \equiv_c \text{“}TapeEmpty\text{”}$$

$$TapeDeckLabels ::_c (string, string, ds)\ label\ set$$
$$TapeDeckLabels \equiv_c \{(En\ \text{“}TapeIn\text{”}, GuardDefault, ActionDefault), (And\ (En\ \text{“}TapeEject\text{”})\ (In\ \text{“}On\text{”}), GuardDefault, ActionDefault)\}$$

$$TapeDeckTrans ::_c (string, string, ds)\ trans\ set$$
$$TapeDeckTrans \equiv_c \{(\text{“}TapeEmpty\text{”}, (En\ \text{“}TapeIn\text{”}, GuardDefault, ActionDefault)\ \text{“}TapeFull\text{”}), (\text{“}TapeFull\text{”}, (And\ (En\ \text{“}TapeEject\text{”})\ (In\ \text{“}On\text{”}), GuardDefault\ ActionDefault)\ \text{“}TapeEmpty\text{”})\}$$

$$TapeDeck ::_c (string, string, ds)\ seqauto$$
$$TapeDeck \equiv_c Abs\_seqauto\ (TapeDeckStates, TapeDeckInitState, TapeDeckLabels, TapeDeckTrans)$$

Formalisierung des Sequentiellen Automaten *AudioCTRL*

$$AudioStates ::_c string\ set$$
$$AudioStates \equiv_c \{ \text{“}TunerMode\text{”}, \text{“}TapeMode\text{”}, \text{“}CDMode\text{”} \}$$

$$AudioInitState ::_c string$$
$$AudioInitState \equiv_c \text{“}TunerMode\text{”}$$

$$\begin{aligned}
&AudioLabels ::_c (string, string, ds)\ label\ set\\
&AudioLabels \equiv_c\\
&\quad \{ (And\ (En\ \text{“}Src\text{”})\ (In\ \text{“}TapeFull\text{”}), GuardDefault, ActionDefault),\\
&\quad\ (And\ (En\ \text{“}Src\text{”})\ (Not(In\ \text{“}CDFull\text{”})), GuardDefault, ActionDefault),\\
&\quad\ (And\ (En\ \text{“}TapeEject\text{”})\ (In\ \text{“}CDEmpty\text{”}), GuardDefault, ActionDefault),\\
&\quad\ (And\ (Not(In\ \text{“}TapeFull\text{”}))\ (Not(In\ \text{“}CDFull\text{”})), GuardDefault,\\
&\qquad\qquad ActionDefault),\\
&\quad\ (En\ \text{“}Src\text{”}, GuardDefault, ActionDefault),\\
&\quad\ (En\ \text{“}CDEject\text{”}, GuardDefault, ActionDefault),\\
&\quad\ (And\ (En\ \text{“}Src\text{”})\ (In\ \text{“}CDFull\text{”}), GuardDefault, ActionDefault),\\
&\quad\ (And\ (En\ \text{“}TapeEject\text{”})\ (In\ \text{“}CDFull\text{”}), GuardDefault, ActionDefault),\\
&\quad\ (En\ \text{“}EndOfTitle\text{”}, \lambda\ D.\ (Select_0\ D) = (Select_1\ D), ActionDefault)\ \}
\end{aligned}$$

$$
\begin{array}{l}
\mathit{AudioTrans} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans}\ \mathit{set} \\
\mathit{AudioTrans} \equiv_{\mathsf{c}} \\
\quad \{ (\text{“}\mathit{TunerMode}\text{”}, \\
\quad\quad (\mathit{And}\ (\mathit{En}\ \text{“}\mathit{Src}\text{”})\ (\mathit{In}\ \text{“}\mathit{TapeFull}\text{”}), \mathit{GuardDefault}, \mathit{ActionDefault}), \\
\quad\quad \text{“}\mathit{TapeMode}\text{”}), \\
\quad\quad (\text{“}\mathit{TapeMode}\text{”}, \\
\quad\quad\quad (\mathit{And}\ (\mathit{En}\ \text{“}\mathit{Src}\text{”})\ (\mathit{Not}\ (\mathit{In}\ \text{“}\mathit{CDFull}\text{”})), \mathit{GuardDefault}, \mathit{ActionDefault}), \\
\quad\quad\quad \text{“}\mathit{TunerMode}\text{”}), \\
\quad\quad (\text{“}\mathit{TapeMode}\text{”}, \\
\quad\quad\quad (\mathit{And}\ (\mathit{En}\ \text{“}\mathit{TapeEject}\text{”})\ (\mathit{In}\ \text{“}\mathit{CDEmpty}\text{”}), \mathit{GuardDefault}, \mathit{ActionDefault}), \\
\quad\quad\quad \text{“}\mathit{TunerMode}\text{”}), \\
\quad\quad (\text{“}\mathit{TunerMode}\text{”}, \\
\quad\quad\quad (\mathit{And}\ (\mathit{Not}\ (\mathit{In}\ \text{“}\mathit{TapeFull}\text{”}))\ (\mathit{Not}\ (\mathit{In}\ \text{“}\mathit{CDFull}\text{”})), \mathit{GuardDefault}, \\
\quad\quad\quad\quad\quad\quad \mathit{ActionDefault}), \\
\quad\quad\quad \text{“}\mathit{CDMode}\text{”}), \\
\quad\quad (\text{“}\mathit{CDMode}\text{”}, \\
\quad\quad\quad (\mathit{En}\ \text{“}\mathit{Src}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \\
\quad\quad\quad \text{“}\mathit{TunerMode}\text{”}), \\
\quad\quad (\text{“}\mathit{CDMode}\text{”}, \\
\quad\quad\quad (\mathit{En}\ \text{“}\mathit{CDEject}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \\
\quad\quad\quad \text{“}\mathit{TunerMode}\text{”}), \\
\quad\quad (\text{“}\mathit{TapeMode}\text{”}, \\
\quad\quad\quad (\mathit{And}\ (\mathit{En}\ \text{“}\mathit{Src}\text{”})\ (\mathit{In}\ \text{“}\mathit{CDFull}\text{”}), \mathit{GuardDefault}, \mathit{ActionDefault}), \\
\quad\quad\quad \text{“}\mathit{CDMode}\text{”}), \\
\quad\quad (\text{“}\mathit{TapeMode}\text{”}, \\
\quad\quad\quad (\mathit{And}\ (\mathit{En}\ \text{“}\mathit{TapeEject}\text{”})\ (\mathit{In}\ \text{“}\mathit{CDFull}\text{”}), \mathit{GuardDefault}, \mathit{ActionDefault}) \\
\quad\quad\quad \text{“}\mathit{CDMode}\text{”}), \\
\quad\quad (\text{“}\mathit{CDMode}\text{”}, \\
\quad\quad\quad (\mathit{En}\ \text{“}\mathit{EndOfTitle}\text{”}, \lambda\ D.\ (\mathit{Select}_0\ D) = (\mathit{Select}_1\ D), \mathit{ActionDefault}) \\
\quad\quad\quad \text{“}\mathit{TunerMode}\text{”}) \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{AudioCTRL} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto} \\
\mathit{AudioCTRL} \equiv_{\mathsf{c}} \mathit{Abs\_seqauto}\ (\mathit{AudioStates}, \mathit{AudioInitState}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{AudioLabels}, \mathit{AudioTrans})
\end{array}
$$

Formalisierung des Sequentiellen Automaten *TunerCTRL*

$$\begin{aligned}
\mathit{TunerStates} &::_c \mathit{string\ set}\\
\mathit{TunerStates} &\equiv_c \{\text{“1”}, \text{“2”}, \text{“3”}, \text{“4”}\}\\
\\
\mathit{TunerInitState} &::_c \mathit{string}\\
\mathit{TunerInitState} &\equiv_c \text{“1”}\\
\\
\mathit{TunerLabels} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
\mathit{TunerLabels} &\equiv_c \{(\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault})\}\\
\\
\mathit{TunerTrans} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
\mathit{TunerTrans} &\equiv_c \{(\text{“1”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“2”}),\\
&\qquad (\text{“2”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“1”}),\\
&\qquad (\text{“2”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“3”}),\\
&\qquad (\text{“3”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“2”}),\\
&\qquad (\text{“3”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“4”}),\\
&\qquad (\text{“4”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“3”}),\\
&\qquad (\text{“4”}, (\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“1”}),\\
&\qquad (\text{“1”}, (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}), \text{“4”})\}\\
\\
\mathit{TunerCTRL} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
\mathit{TunerCTRL} &\equiv_c \mathit{Abs_seqauto}\ (\mathit{TunerStates}, \mathit{TunerInitState},\\
&\qquad\qquad \mathit{TunerLabels}, \mathit{TunerTrans})
\end{aligned}$$

Formalisierung des Sequentiellen Automaten *TapeCTRL*

$$\begin{aligned}
\mathit{TapeStates} &::_c \mathit{string\ set}\\
\mathit{TapeStates} &\equiv_c \{\text{“Playing”}, \text{“FSpooling”}, \text{“BSpooling”}\}\\
\\
\mathit{TapeInitState} &::_c \mathit{string}\\
\mathit{TapeInitState} &\equiv_c \text{“Playing”}\\
\\
\mathit{TapeLabels} &::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
\mathit{TapeLabels} &\equiv_c \{(\mathit{En}\ \text{“Next”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad (\mathit{En}\ \text{“Back”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad (\mathit{En}\ \text{“EndOfTape”}, \mathit{GuardDefault}, \mathit{ActionDefault})\}
\end{aligned}$$

$$
\begin{aligned}
&\mathit{TapeTrans} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans}\ \mathit{set}\\
&\mathit{TapeTrans} \equiv_{\mathsf{c}}\\
&\quad \{ (\text{“}\mathit{Playing}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{Next}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{FSpooling}\text{”}),\\
&\qquad (\text{“}\mathit{FSpooling}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{Back}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{Playing}\text{”})\\
&\qquad (\text{“}\mathit{Playing}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{Back}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{BSpooling}\text{”}),\\
&\qquad (\text{“}\mathit{BSpooling}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{Next}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{Playing}\text{”}),\\
&\qquad (\text{“}\mathit{BSpooling}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{EndOfTape}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{Playing}\text{”}) \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{TapeCTRL} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{TapeCTRL} \equiv_{\mathsf{c}} \mathit{Abs\_seqauto}\ (\mathit{TapeStates}, \mathit{TapeInitState},\\
&\qquad\qquad\qquad\qquad \mathit{TapeLabels}, \mathit{TapeTrans})
\end{aligned}
$$

Formalisierung des Sequentiellen Automaten *CDCTRL*

$$
\begin{aligned}
&\mathit{CDStates} ::_{\mathsf{c}} \mathit{string}\ \mathit{set}\\
&\mathit{CDStates} \equiv_{\mathsf{c}} \{ \text{“}\mathit{CDPlaying}\text{”}, \text{“}\mathit{SelNextTrack}\text{”}, \text{“}\mathit{SelPrevTrack}\text{”} \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDInitState} ::_{\mathsf{c}} \mathit{string}\\
&\mathit{CDInitState} \equiv_{\mathsf{c}} \text{“}\mathit{CDPlaying}\text{”}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UpdateIncT}_{\mathsf{A}} ::_{\mathsf{c}} \mathit{ds}\ \mathit{pupdate}\\
&\mathit{UpdateIncT}_{\mathsf{A}} \equiv_{\mathsf{c}} \mathit{LiftPUpdate}(\lambda\ D.\ [\mathit{None}, \mathit{Some}(V_1\ ((\mathit{Select}_1\ D) + 1))])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UpdateDecT}_{\mathsf{A}} ::_{\mathsf{c}} \mathit{ds}\ \mathit{pupdate}\\
&\mathit{UpdateDecT}_{\mathsf{A}} \equiv_{\mathsf{c}} \mathit{LiftPUpdate}(\lambda\ D.\ [\mathit{None}, \mathit{Some}(V_1\ ((\mathit{Select}_1\ D) - 1))])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDLabels} \ ::_{\mathsf{c}} \ (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
&\mathit{CDLabels} \ \equiv_{\mathsf{c}} \{\,(\mathit{En}\ \text{“}\mathit{Next}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\qquad (\mathit{En}\ \text{“}\mathit{Back}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\qquad (\mathit{En}\ \text{“}\mathit{Ready}\text{”}, \mathit{GuardDefault}, \mathit{UpdateIncT}_{\mathtt{A}}),\\
&\qquad\qquad (\mathit{En}\ \text{“}\mathit{Ready}\text{”}, \mathit{GuardDefault}, \mathit{UpdateDecT}_{\mathtt{A}}),\\
&\qquad\qquad (\mathit{En}\ \text{“}\mathit{EndOfTitle}\text{”}, \lambda\ D.\ (\mathit{Select}_1\ D) < (\mathit{Select}_0\ D)\,, \mathit{ActionDefault})\,\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDTrans} \ ::_{\mathsf{c}} \ (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
&\mathit{CDTrans} \ \equiv_{\mathsf{c}}\\
&\quad \{\,(\text{“}\mathit{CDPlaying}\text{”},\\
&\qquad (\mathit{En}\ \text{“}\mathit{Next}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{SelNextTrack}\text{”}),\\
&\qquad (\text{“}\mathit{SelNextTrack}\text{”},\\
&\qquad\quad (\mathit{En}\ \text{“}\mathit{Ready}\text{”}, \mathit{GuardDefault}, \mathit{UpdateIncT}_{\mathtt{A}}),\\
&\qquad \text{“}\mathit{CDPlaying}\text{”})\\
&\qquad (\text{“}\mathit{CDPlaying}\text{”},\\
&\qquad\quad (\mathit{En}\ \text{“}\mathit{Back}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{SelPrevTrack}\text{”}),\\
&\qquad (\text{“}\mathit{SelPrevTrack}\text{”},\\
&\qquad\quad (\mathit{En}\ \text{“}\mathit{Ready}\text{”}, \mathit{GuardDefault}, \mathit{UpdateDecT}_{\mathtt{A}}),\\
&\qquad \text{“}\mathit{CDPlaying}\text{”}),\\
&\qquad (\text{“}\mathit{CDPlaying}\text{”},\\
&\qquad\quad (\mathit{En}\ \text{“}\mathit{EndOfTitle}\text{”}, \lambda\ D.\ (\mathit{Select}_1\ D) < (\mathit{Select}_0\ D), \mathit{ActionDefault}),\\
&\qquad \text{“}\mathit{SelNextTrack}\text{”})\,\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CDCTRL} \ ::_{\mathsf{c}} \ (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{CDCTRL} \ \equiv_{\mathsf{c}} \mathit{Abs\_seqauto}\ (\mathit{CDStates}, \mathit{CDInitState}, \mathit{CDLabels}, \mathit{CDTrans})
\end{aligned}
$$

Formalisierung des Hierarchischen Automaten *Car-Audio-System*

$$
\begin{aligned}
&\mathit{CarAudioSystem} \ ::_{\mathsf{c}} \ (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{hierauto}\\
&\mathit{CarAudioSystem} \ \equiv_{\mathsf{c}} (\mathit{PseudoHA}\ \ \mathit{CarAudioSystemCTRL}\ \ \mathit{InitDS})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{CarAudioSystem}\text{”}, \mathit{AudioPlayer})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{CarAudioSystem}\text{”}, \mathit{CDPlayer})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{CarAudioSystem}\text{”}, \mathit{TapeDeck})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{On}\text{”}, \mathit{AudioCTRL})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{TunerMode}\text{”}, \mathit{TunerCTRL})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{TapeMode}\text{”}, \mathit{TapeCTRL})\\
&\qquad\qquad\qquad\qquad \boxplus\ (\text{“}\mathit{CDMode}\text{”}, \mathit{CDCTRL})
\end{aligned}
$$

B.2 Kühlungssystem im Nuklearkraftwerk

Formalisierung des Datenraumes

$$ds \equiv_t V_0\ int$$

$$Sel_0 ::_c ds \to int$$
$$Sel_0\,(V_0\ i) = i$$

$$Select_0 ::_c ds\ data \to int$$
$$Select_0\ D \equiv_c Sel_0\,(PartData\ D\ 0)$$

$$DS ::_c ds\ dataspace$$
$$DS \equiv_c Abs\_dataspace\,[\,ran\ V_0\,]$$

$$LiftInitData ::_c ds\ list \to ds\ data$$
$$LiftInitData\ L \equiv_c Abs\_data(L, DS)$$

$$LiftPUpdate ::_c (ds\ data \to (ds\ option)\ list) \to ds\ pupdate$$
$$LiftPUpdate\ L \equiv_c Abs\_pupdate(\lambda\ D.\ \text{if}\ ((DataSpace\ D) = DS)$$
$$\text{then}\ Abs\_pdata(L\ D, DS)$$
$$\text{else}\ (DataAsPData\ D))$$

Formalisierung des Wurzelautomaten

$$RootStates ::_c string\ set$$
$$RootStates \equiv_c \{\text{“}SafetyInjectionCTRL\text{”}\}$$

$$RootInit ::_c string$$
$$RootInit \equiv_c \text{“}SafetyInjectionCTRL\text{”}$$

$$RootLabels ::_c (string, string, ds)\ label\ set$$
$$RootLabels \equiv_c \varnothing$$

$$RootTrans ::_c (string, string, ds)\ trans\ set$$
$$RootTrans \equiv_c \varnothing$$

$$RootCTRL ::_c (string, string, ds)\ seqauto$$
$$RootCTRL \equiv_c Abs\_seqauto\ (RootStates, RootInit, RootLabels, RootTrans)$$

Formalisierung des Sequentiellen Automaten *DeviceCTRL*

$$DeviceStates ::_c string\ set$$
$$DeviceStates \equiv_c \{“Blocked”, “UnBlocked”\}$$

$$DeviceInit ::_c string$$
$$DeviceInit \equiv_c “Blocked”$$

$$DeviceLabels ::_c (string, string, ds)\ label\ set$$
$$DeviceLabels \equiv_c \{(En\ “Reset”, GuardDefault, ActionDefault)$$
$$(And\ (En\ “Blocked”)\ (Not\,(En\ “Reset”)), GuardDefault, ActionDefault)\}$$

$$DeviceTrans ::_c (string, string, ds)\ trans\ set$$
$$DeviceTrans \equiv_c \{(“Blocked”,$$
$$(En\ “Reset”, GuardDefault, ActionDefault),$$
$$“UnBlocked”),$$
$$(“UnBlocked”,$$
$$(And\ (En\ “Blocked”)\ (Not\,(En\ “Reset”)), GuardDefault, ActionDefault),$$
$$“Blocked”)\}$$

$$DeviceCTRL ::_c (string, string, ds)\ seqauto$$
$$DeviceCTRL \equiv_c Abs\_seqauto\ (DeviceStates, DeviceInit, DeviceLabels, DeviceTrans)$$

Formalisierung des Sequentiellen Automaten *UnBlockedCTRL*

$$UnBlockedStates ::_c string\ set$$
$$UnBlockedStates \equiv_c \{“Idle”, “Damp”, “SafetyInjection”\}$$

$$UnBlockedInit ::_c string$$
$$UnBlockedInit \equiv_c “Idle”$$

$$UnBlockedLabels ::_c (string, string, ds)\ label\ set$$
$$UnBlockedLabels \equiv_c \{(In\ “TooHigh”, GuardDefault, ActionDefault),$$
$$(In\ “TooLow”, GuardDefault, ActionDefault),$$
$$(In\ “DampOff”, GuardDefault, ActionDefault),$$
$$(In\ “InjOff”, GuardDefault, ActionDefault)\}$$

$$
\begin{aligned}
&\mathit{UnBlockedTrans} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans}\ \mathit{set}\\
&\mathit{UnBlockedTrans} \equiv_{\mathsf{c}} \{(\text{“}\mathit{Idle}\text{”},\\
&\qquad\qquad (\mathit{In}\ \text{“}\mathit{TooHigh}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\qquad \text{“}\mathit{Damp}\text{”}),\\
&\qquad\qquad (\text{“}\mathit{Idle}\text{”},\\
&\qquad\qquad (\mathit{In}\ \text{“}\mathit{TooLow}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\qquad \text{“}\mathit{SafetyInjection}\text{”}),\\
&\qquad\qquad (\text{“}\mathit{Damp}\text{”},\\
&\qquad\qquad (\mathit{In}\ \text{“}\mathit{DampOff}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\qquad \text{“}\mathit{Idle}\text{”}),\\
&\qquad\qquad (\text{“}\mathit{SafetyInjection}\text{”},\\
&\qquad\qquad (\mathit{In}\ \text{“}\mathit{InjOff}\text{”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\qquad \text{“}\mathit{Idle}\text{”})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UnBlockedCTRL} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{UnBlockedCTRL} \equiv_{\mathsf{c}} \mathit{Abs\_seqauto}\ (\mathit{UnBlockedStates}, \mathit{UnBlockedInit},\\
&\qquad\qquad\qquad\qquad\qquad \mathit{UnBlockedLabels}, \mathit{UnBlockedTrans})
\end{aligned}
$$

Formalisierung des Sequentiellen Automaten *PressureMode*

$$
\begin{aligned}
&\mathit{PressureStates} ::_{\mathsf{c}} \mathit{string}\ \mathit{set}\\
&\mathit{PressureStates} \equiv_{\mathsf{c}} \{\text{“}\mathit{Permitted}\text{”}, \text{“}\mathit{TooHigh}\text{”}, \text{“}\mathit{TooLow}\text{”}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureInit} ::_{\mathsf{c}} \mathit{string}\\
&\mathit{PressureInit} \equiv_{\mathsf{c}} \text{“}\mathit{Permitted}\text{”}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ActionInjOff} ::_{\mathsf{c}} (\mathit{string}, \mathit{ds})\ \mathit{action}\\
&\mathit{ActionInjOff} \equiv_{\mathsf{c}} (\{\text{“}\mathit{InjOff}\text{”}\}, \mathit{PUpdateDefault})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ActionDampOff} ::_{\mathsf{c}} (\mathit{string}, \mathit{ds})\ \mathit{action}\\
&\mathit{ActionDampOff} \equiv_{\mathsf{c}} (\{\text{“}\mathit{DampOff}\text{”}\}, \mathit{PUpdateDefault})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureLabels} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{label\ set}\\
&\mathit{PressureLabels} \equiv_c \{(\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\, D) < 20, \mathit{ActionDefault}),\\
&\qquad\qquad (\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\, D) < 10, \mathit{ActionInjOff}),\\
&\qquad\qquad (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\, D) < 20, \mathit{ActionDampOff}),\\
&\qquad\qquad (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\, D) < 10, \mathit{ActionDefault})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureTrans} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
&\mathit{PressureTrans} \equiv_c\\
&\qquad \{(\text{“}\mathit{Permitted}\text{”},\\
&\qquad\quad (\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\, D) < 20, \mathit{ActionDefault}),\\
&\qquad\quad \text{“}\mathit{TooHigh}\text{”}),\\
&\qquad\quad (\text{“}\mathit{TooLow}\text{”},\\
&\qquad\quad\ (\mathit{ExprDefault}, \lambda\ D.\ \neg\ (\mathit{Select}_0\, D) < 10, \mathit{ActionInjOff}),\\
&\qquad\quad \text{“}\mathit{Permitted}\text{”}),\\
&\qquad\quad (\text{“}\mathit{TooHigh}\text{”},\\
&\qquad\quad\ (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\, D) < 20, \mathit{ActionDampOff}),\\
&\qquad\quad \text{“}\mathit{Permitted}\text{”}),\\
&\qquad\quad (\text{“}\mathit{Permitted}\text{”},\\
&\qquad\quad\ (\mathit{ExprDefault}, \lambda\ D.\ (\mathit{Select}_0\, D) < 10, \mathit{ActionDefault})\\
&\qquad\quad \text{“}\mathit{TooLow}\text{”})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PressureMode} ::_c (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{PressureMode} \equiv_c \mathit{Abs\_seqauto}\ (\mathit{PressureStates}, \mathit{PressureInit},\\
&\qquad\qquad\qquad\qquad \mathit{PressureLabels}, \mathit{PressureTrans})
\end{aligned}
$$

Formalisierung des Sequentiellen Automaten *Measuring*

$$
\begin{aligned}
&\mathit{MeasuringStates} ::_c \mathit{string\ set}\\
&\mathit{MeasuringStates} \equiv_c \{\text{“}\mathit{Normal}\text{”}, \text{“}\mathit{Sensitive}\text{”}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{MeasuringInit} ::_c \mathit{string}\\
&\mathit{MeasuringInit} \equiv_c \text{“}\mathit{Normal}\text{”}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UpdateMinusNormal} ::_c \mathit{ds\,pupdate}\\
&\mathit{UpdateMinusNormal} \equiv_c \mathit{LiftPUpdate}(\lambda\ D.\ [\mathit{Some}(V_0\,((\mathit{Select}_0\, D) - 1))])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ActionMinusNormal} ::_c (\mathit{string}, \mathit{ds})\ \mathit{action}\\
&\mathit{ActionMinusNormal} \equiv_c (\varnothing, \mathit{UpdateMinusNormal})
\end{aligned}
$$

$$UpdatePlusNormal ::_c dspupdate$$
$$UpdatePlusNormal \equiv_c LiftPUpdate(\lambda\ D.\ [\,Some(V_0\,((Select_0\,D)+1))\,])$$

$$ActionPlusNormal ::_c (string, ds)\,action$$
$$ActionPlusNormal \equiv_c (\varnothing, UpdatePlusNormal)$$

$$UpdateMinusSensitive ::_c dspupdate$$
$$UpdateMinusSensitive \equiv_c LiftPUpdate(\lambda\ D.\ [\,Some(V_0\,((Select_0\,D)-2))\,])$$

$$ActionMinusSensitive ::_c (string, ds)\,action$$
$$ActionMinusSensitive \equiv_c (\varnothing, UpdateMinusNormal)$$

$$UpdatePlusSensitive ::_c dspupdate$$
$$UpdatePlusSensitive \equiv_c LiftPUpdate(\lambda\ D.\ [\,Some(V_0\,((Select_0\,D)+2))\,])$$

$$ActionMinusNormal ::_c (string, ds)\,action$$
$$ActionMinusNormal \equiv_c (\varnothing, UpdateMinusNormal)$$

$$\begin{aligned}
&MeasuringLabels ::_c (string, string, ds)\,label\ set\\
&MeasuringLabels \equiv_c\\
&\quad \{\,(En\ \text{“}Lower\text{”}, GuardDefault, ActionMinusNormal),\\
&\quad\ (En\ \text{“}Higher\text{”}, GuardDefault, ActionPlusNormal),\\
&\quad\ (Or\ (In\ \text{“}TooHigh\text{”})\ (In\ \text{“}TooLow\text{”}), GuardDefault, ActionDefault),\\
&\quad\ (En\ \text{“}Lower\text{”}, GuardDefault, ActionMinusSensitive)\\
&\quad\ (En\ \text{“}Higher\text{”}, GuardDefault, ActionPlusSensitive)\\
&\quad\ (In\ \text{“}Permitted\text{”}, GuardDefault, ActionDefault)\,\}
\end{aligned}$$

$$
\begin{aligned}
&\mathit{MeasuringTrans} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{trans\ set}\\
&\mathit{MeasuringTrans} \equiv_{\mathsf{c}}\\
&\qquad \{ (\mathit{“Normal”},\\
&\qquad\quad (\mathit{En\ “Lower”}, \mathit{GuardDefault}, \mathit{ActionMinusNormal}),\\
&\qquad\ \mathit{“Normal”}),\\
&\qquad (\mathit{“Normal”},\\
&\qquad\quad (\mathit{En\ “Higher”}, \mathit{GuardDefault}, \mathit{ActionPlusNormal}),\\
&\qquad\ \mathit{“Normal”}),\\
&\qquad (\mathit{“Normal”},\\
&\qquad\quad (\mathit{Or}\ (\mathit{In\ “TooHigh”})\ (\mathit{In\ “TooLow”}), \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\ \mathit{“Sensitive”}),\\
&\qquad (\mathit{“Sensitive”},\\
&\qquad\quad (\mathit{In\ “Permitted”}, \mathit{GuardDefault}, \mathit{ActionDefault}),\\
&\qquad\ \mathit{“Normal”}),\\
&\qquad (\mathit{“Sensitive”},\\
&\qquad\quad (\mathit{En\ “Lower”}, \mathit{GuardDefault}, \mathit{ActionMinusSensitive}),\\
&\qquad\ \mathit{“Sensitive”}),\\
&\qquad (\mathit{“Sensitive”},\\
&\qquad\quad (\mathit{En\ “Higher”}, \mathit{GuardDefault}, \mathit{ActionPlusSensitive})\\
&\qquad\ \mathit{“Sensitive”}) \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Measuring} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{seqauto}\\
&\mathit{Measuring} \equiv_{\mathsf{c}} \mathit{Abs\_seqauto}\ (\mathit{MeasuringStates}, \mathit{MeasuringInit},\\
&\qquad\qquad\qquad\qquad \mathit{MeasuringLabels}, \mathit{MeasuringTrans})
\end{aligned}
$$

Formalisierung des Hierarchischen Automaten *Safety-Injection-System*

$$
\begin{aligned}
&\mathit{SafetyInjectionSystem} ::_{\mathsf{c}} (\mathit{string}, \mathit{string}, \mathit{ds})\ \mathit{hierauto}\\
&\mathit{SafetyInjectionSystem} \equiv_{\mathsf{c}} (\mathit{PseudoHA\ RootCTRL}\ (\mathit{LiftInitData}\ [V_0\, 15]))\\
&\qquad\qquad\qquad\qquad \boxplus (\mathit{“SafetyInjectionCTRL”}, \mathit{DeviceCTRL})\\
&\qquad\qquad\qquad\qquad \boxplus (\mathit{“SafetyInjectionCTRL”}, \mathit{PressureMode})\\
&\qquad\qquad\qquad\qquad \boxplus (\mathit{“SafetyInjectionCTRL”}, \mathit{Measuring})\\
&\qquad\qquad\qquad\qquad \boxplus (\mathit{“UnBlocked”}, \mathit{UnBlockedCTRL})
\end{aligned}
$$

Verzeichnis der Definitionen und Theoreme

Index für Definitionen und Theoreme

Literaturverzeichnis

[ABG+03] Alkassar, A., M. Broy, F. Gehring, M. Garschhammer, H.-G. Hegering, P. Keil, H. Kelter, U. Löwer, M. Pankow, A. Picot, A.-R. Sadeghi und M. Schiffers: *Kommunikations- und Informationstechnik 2010+3: Neue Trends und Entwicklungen in Technologie, Anwendungen und Sicherheit.* SecuMedia-Verlag, 2003.

[ABRW05] Angermann, A., M. Beuschel, M. Rau und U. Wohlfarth: *Matlab-Simulink-Stateflow.* Oldenbourg Verlag München, 2005.

[And96] André, C.: *SyncCharts: a Visual Representation of Reactive Behaviors.* Technischer Bericht RR 95–52, rev. RR (96–56), Sophia-Antipolis, I3S, Frankreich, 1996.

[And03] André, C.: *Semantics of S.S.M. (Safe State Machine).* Technischer Bericht, Esterel Technologies, 2003.

[Are04] Aredo, D.B.: *Formal Development of Open Distributed Systems: Integration of UML and PVS.* Doktorarbeit, Department of Informatics, University of Oslo, Norway, 2004.

[AW06] Apt, K. und M. G. Wallace: *Constraint Logic Programming using Eclipse.* Cambridge-University-Press, 2006.

[Bac88] Back, R. J. R.: *A Calculus of Refinements for Program Derivations.* Acta Informatica, 23:593–624, 1988.

[Ber99] Berezin, S.: *Combining Model Checking and Theorem Proving in Hardware Verification.* Doktorarbeit, Carnegie Mellon University, 1999.

[BF99] Bolotov, A. und M. Fisher: *A Clausal Resolution Method for CTL Branching-Time Temporal Logic.* Journal of Experimental and Theoretical Artificial Intelligence, 11:77–93, 1999.

[BG92] Berry, G. und G. Gonthier: *The Esterel Synchronous Programming Language: Design, Semantics, Implementation.* Science of Computer Programming, 19(2):87–152, 1992.

[BG97] Büssow, R. und W. Grieskamp: *Combining Z and Temporal Interval Logics for the Formalization of Properties and Behaviors of Embedded Systems.* In: *Proceedings of Asian Computing Science Conference (ASIAN)*, Band 1345 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1997.

[BGGK97] Büssow, R., R. Geisler, W. Grieskamp und M. Klar: *The μSZ Notation Version 1.0.* Technischer Bericht 97-26, Fakultät IV, Technische Universität Berlin, Deutschland, 1997.

[BGL98] Bultan, T., R. Gerber und C. League: *Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach.* In: *Proceedings of the International Symposium on Software Testing and Analysis*, Seiten 113–123. ACM, 1998.

[BH97] Bharadwaj, R. und C. Heitmeyer: *Verifying SCR Requirements Specifications using State Exploration.* In: *Proceedings of the ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.

[BHP+00] Broy, M., H.-G. Hegering, A. Picot, A. Buttermann, M. Garschhammer, R. Hauck und S. Vogel: *Kommunikations- und Informationstechnik 2010 : Trends in Technologie und Markt.* SecuMedia-Verlag, 2000.

[BR00] Ball, T. und S.K. Rajamani: *Bebop: A Symbolic Model Checker for Boolean Programs.* In: Havelund, K., J. Penix und W. Visser (Herausgeber): *Proceedings of the SPIN Workshop on Model Checking of Software (SPIN 2000)*, Band 1885 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 113–130. Springer-Verlag, 2000.

[BR02] Ball, T. und S.K. Rajamani: *The SLAM Project: Debugging System Software via Static Analysis.* In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL 2002)*, Band 37 der Reihe *SIGPLAN Notice*, Seiten 1–3. ACM SIGPLAN, 2002.

[Bro03] Broy, M.: *Automotive Software Engineering.* In: *25th International Conference on Software Engineering*, Seiten 719 – 720. IEEE 2003, 2003.

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

[Bry06] Bryant, R.E.: *Formal Verification of Infinite State Systems Using Boolean Methods.* IEEE Symposium on Logic in Computer Science (LICS), 0:3–4, 2006.

[BW98a] Brockmeyer, U. und G. Wittich: *Real-Time Verification of Statemate Designs.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 537–541, 1998.

[BW98b] Brockmeyer, U. und G. Wittich: *Tamagotchis Need Not Die — Verification of* Statemate *Designs.* In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Band 1384 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 217–231. Springer-Verlag, 1998.

[Büs03] Büssow, R.: *Model Checking Combined Z and Statechart Specifications.* Doktorarbeit, Fakultät IV, Technische Universität Berlin, Deutschland, 2003.

[CC77] Cousot, P. und R. Cousot: *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points.* In: *Proceedings of the 4th ACM Symposium on Principles of Programming Languages.* ACM Press, 1977.

[CCGR99] Cimatti, A., E. Clarke, F. Giunchiglia und M. Roveri: *NuSMV: A new Symbolic Model Verifier.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Band 1633 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 495–499. Springer-Verlag, 1999.

[CE81] Clarke, E. M. und E. A. Emerson: *Synthesis of Sychnronization skeletons for branching time temporal logic.* In: Kozen, D. (Herausgeber): *Logic of Programs*, Band 131 der Reihe *Lecture Notes in Computer Science (LNCS).* Springer-Verlag, 1981.

[CGH+93] Clarke, E.M., O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan und L.A. Ness: *Verification of the Futurebus+ Cache Coherence Protocol.* In: Agnew, D., L. Claesen und R. Camposano (Herausgeber): *International Symposium on Computer Hardware Description Languages and their Applications*, Seiten 5–20. Elsevier Science Publishers B.V., Niederlande, 1993.

[CGJ+00] Clarke, E.M., O. Grumberg, S. Jha, Y. Lu und H. Veith: *Counterexample-Guided Abstraction Refinement.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 154–169, 2000.

[CGL94] Clarke, E. M., O. Grumberg und D. E. Long: *Model Checking and Abstraction.* ACM Transactions on Programming Languages and Systems, 16(5):1512–1542, 1994.

[CGP00] Clarke, E., O. Grumberg und D. Peled: *Model Checking.* MIT-Press, 2000.

[CGP02] Chandra, S., P. Godefroid und C. Palm: *Software Model Checking in Practice: An Industrial Case Study.* In: *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 431–441. ACM Press, 2002.

[Chu40] Church, A.: *A Formulation of the Simple Theory of Types.* Journal of Symbolic Logic, Seiten 56–68, 1940.

[CMM00] Cansell, D., D. Méry und S. Merz: *Predicate Diagrams for the Verification of Reactive Systems.* In: *International Conference on Integrated Formal Methods (IFM 2000)*, Band 1945 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 380–397. Springer-Verlag, 2000.

[CMM01] Cansell, D., D. Méry und S. Merz: *Diagram Refinements for the Design of Reactive Systems.* Journal of Universal Computer Science, 7(2):159–174, 2001.

[CP93] Courtois, P.J. und D.L. Parnas: *Documentation for Safety Critical Software.* In: *Proceedings of the 15th International Conference on Software Engineering*, Seiten 315–323, 1993.

[Dam96] Dams, D.: *Abstract Interpretation and Partition Refinement for Model Checking*. Doktorarbeit, Eindhoven University of Technology, Niederlande, 1996.

[Day93] Day, N.: *A Model Checker for Statecharts*. Technischer Bericht TR 93–35, Department of Computer Science, University of British Columbia, 1993.

[DB01] Derrick, J. und E. Boiten: *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer-FACIT, 2001.

[DD02] Das, S. und D.L. Dill: *Counter-Example Based Predicate Discovery in Predicate Abstraction*. In: *Formal Methods in Computer-Aided Design*. Springer-Verlag, 2002.

[DDS99] Das, S., D.L. Dill und S.Park: *Experience with Predicate Abstraction*. In: *Proceedings of the International Conference on Computer-Aided Verification (CAV)*. Springer-Verlag, 1999. Trento, Italien.

[DGM97] Devillers, M., D. Griffioen und O. Müller: *Possibly Infinite Sequences in Theorem Provers: A Comprative Study*. In: Gunter, E.L. und A.P. Felty (Herausgeber): *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Band 1275 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 89–104. Springer-Verlag, 1997.

[Dil] Dill, D.: *SVC homepage. http://verify.stanford.edu/SVC/.*

[DMY02] David, A., O. Möller und W. Yi: *Formal Verification of UML Statecharts with Real-Time Extensions*. In: Kutsche, R.-D. und H. Weber (Herausgeber): *Proceedings of the Fundamental Approaches to Software Engineering: 5th International Conference (FASE 2002)*, Band 2306 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 218–232. Springer-Verlag, 2002.

[DR00] Duke, R. und G. Rose: *Formal Object-Oriented Specification Using Object-Z*. MacMillan Press, 2000.

[EC80] Emerson, E. A. und E. M. Clarke: *Characterizing Correctness Properties of Parallel Programs using Fixpoints*. In: *International Colloquium on Automata, Languages and Programming (ICALP)*, Band 85 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1980.

[EH86] Emerson, E.A. und J.Y. Halpern: *SSometimesänd Not Neverrevisited: On Branching versus Linear Time Temporal Logic*. Journal of the ACM, 33(1):151–178, 1986.

[FP06] Findeis, M. und I. Pabst: *Functional Safety in the Automtive Industry, Process and Methods*. In: *VDA - Alternative Refrigerant Winter Meeting*. Verband der Automobilindustrie, 2006.

[GGBdM91] Guernic, P.L., T. Gautier, M.L. Borgne und C. de Marie: *Programming Real-Time Applications with* Signal. Proceedings of the IEEE, 79(9):1321–1335, 1991.

[GHJ01] Godefroid, P., M. Huth und R. Jagadeesan: *Abstraction-Based Model Checking Using Modal Transitionansition Systems.* In: Larsen, K. G. und M. Nielsen (Herausgeber): *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, Band 2154 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 426–440. Springer-Verlag, 2001.

[GLM99] Gnesi, S., D. Latella und M. Massink: *Model Checking UML Statechart Diagrams using JACK.* In: Williams, A. (Herausgeber): *Proceedings of IEEE International High-Assurance Systems Engineering Symposium*, Seiten 46–55. IEEE Computer Society, 1999.

[GM93] Gordon, M. und T.F. Melham: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GMW79] Gordon, M., R. Milner und C. Wadsworth: *Edinburgh LCF - A Mechanised Logic of Computation*, Band 78 der Reihe *Lecture Notes in Computer Science (LNCS).* Springer-Verlag, 1979.

[GS97] Graf, S. und H. Saïdi: *Construction of Abstract State Graphs with PVS.* In: *Conference on Computer Aided Verification (CAV)*, Band 1254 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 72–83, 1997.

[Har87] Harel, D.: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 8:231–274, 1987.

[HCRP91] Halbwachs, N., P. Caspi, P. Raymond und D. Pilaud: *The Synchronous Data-Flow Programming Language* Lustre. Proceedings of the IEEE, 79(9):1305–1320, 1991.

[HG97] Harel, D. und E. Gery: *Executable Object Modeling with Statecharts.* IEEE Computer, Seiten 246–257, 1997.

[Hie98] Hiemer, J.-J.: *Statecharts in CSP – Ein Prozeßmodell in CSP zur Analyse von Statemate Statecharts.* Doktorarbeit, Technische Universität Berlin, Deutschland, 1998.

[HJS01] Huth, M., R. Jagadeesan und D. Schmidt: *Modal Transition Systems: A Foundation for three-valued Program Analysis.* In: Sands, D. (Herausgeber): *European Symposium on Programming, ESOP 2001*, Band 2028 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 155–169. Springer-Verlag, 2001.

[HK01a] Helke, S. und F. Kammüller: *Representing Hierarchical Automata in Interactive Theorem Provers.* In: Boulton, R.J. und P.B. Jackson (Herausgeber): *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Band 2152 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 233–248. Springer-Verlag, 2001.

[HK01b] Helke, S. und F. Kammüller: *Representing Hierarchical Automata in Interactive Theorem Provers.* Technischer Bericht, Dagstuhl Seminar, 2001.

[HK03a] Helke, S. und F. Kammüller: *A Framework for Property Preservation Based on Galois Connections*. In: Basin, D. und B. Wolff (Herausgeber): *TPHOLs 2003: Emerging Trends Proceedings*, Technical Report No. 189, Seiten 3–12. Albert-Ludwigs-Universität Freiburg, 2003.

[HK03b] Helke, S. und F. Kammüller: *Verification of Statecharts Including Data Spaces*. In: Basin, D. und B. Wolff (Herausgeber): *TPHOLs 2003: Emerging Trends Proceedings*, Technischer Report 189, Seiten 177–190. Albert-Ludwigs-Universität Freiburg, 2003.

[HK05] Helke, S. und F. Kammüller: *Structure Preserving Data Abstractions for Statecharts*. In: Wang, F. (Herausgeber): *Proceedings of Formal Techniques for Networked and Distributed Systems (FORTE)*, Band 3731 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 305–319. Springer-Verlag, 2005.

[HLN+90] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring und M. Trakhtenbrot: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering, 16(4):403–413, 1990.

[HN96] Harel, D. und A. Naamad: *The STATEMATE Semantics of Statecharts*. ACM Transactions on Software Engineering and Methodology, 5(4):293–333, 1996.

[HNS97] Helke, S., T. Neustupny und T. Santen: *Automating Test Case Generation from Z Specifications with Isabelle*. In: Bowen, J., M. Hinchey und D. Till (Herausgeber): *Proceedings of ZUM '97: The Z Formal Specification Notation, International Conference of Z Users*, Band 1212 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 52–71. Springer-Verlag, 1997.

[HNSS00] Helke, S., A. Nordwig, T. Santen und D. Sokenou: *Scaling up von V&V Techniken durch Integration und Abstraktion*. In: Wirsing, M., M. Gogolla, H.-J. Kreowski, T. Nipkow und W. Reif (Herausgeber): *Proceedings of the GI-Workshop Rigorose Entwicklung software-intensiver Systeme*, Technischer Report No. 5, Seiten 11–20. Ludwig-Maximilians-Universität München, 2000.

[Hoa72] Hoare, C. A. R.: *Proof of Correctness of Data Representations*. Acta Informatica, 1:271–281, 1972.

[Hoa85] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall, International Series in Computer Science, 1985.

[Hol03] Holzmann, G.J.: *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2003.

[HS01] Helke, S. und T. Santen: *Mechanized Analysis of Behavioral Conformance in the Eiffel Base Libraries*. In: Oliveira, J. und P. Zave (Herausgeber): *Proceedings of Formal Methods for Increasing Software Productivity (FME)*, Band 2021 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 20–42. Springer-Verlag, 2001.

[HU79] Hopcroft, J.E. und J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Hut02] Huth, M.: *Model Checking Modal Transition Systems Using Kripke Structures.* In: Cortesi, A. (Herausgeber): *Verification, Model Checking, and Abstract Interpretation, Third International Workshop (VMCAI 2002)*, Band 2294 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 302–316. Springer-Verlag, 2002.

[JM94] Jahanian, F. und A. Mok: *Modechart: A Specification Language for Real-Time Systems.* IEEE Transactions of Software Engineering, 20:933–947, 1994.

[Jon90] Jones, C. B.: *Systematic Software Development using VDM.* Prentice Hall International, 1990.

[JS94] Joyce, J.J. und C.-J.H. Seger: *The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover.* In: Joyce, J.J. und C.-J.H. Seger (Herausgeber): *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*, Band 780 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 185–198. Springer-Verlag, 1994.

[Kan05] Kanal, K.: *Automatisierte Abstraktion zur Verifikation von Statecharts mit unendlichen Datenräumen.* Diplomarbeit, Fakultät IV, Technische Universität Berlin, Deutschland, 2005.

[KH00] Kammüller, F. und S. Helke: *Mechanical Analysis of UML State Machines and Class Diagrams.* Technischer Bericht, Proceedings of the Workshop on Precise Semantics for the UML (ECOOP), 2000.

[KMR02] Knapp, A., S. Merz und C. Rauh: *Model Checking Timed UML State Machines and Collaborations.* In: Damm, W. und E.-R. Olderog (Herausgeber): *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT)*, Band 2469 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 395–414. Springer-Verlag, 2002.

[Kri63] Kripke, S.: *Semantical Considerations on Modal Logic.* Acta Philosphica Fennica, 16:83–94, 1963.

[Lar89] Larsen, K. G.: *Modal Specifications.* In: Sifakis, J. (Herausgeber): *Automatic Verification Methods for Finite State Systems*, Band 407 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 232– 246. Springer-Verlag, 1989.

[LGS+95] Loiseaux, C., S. Graf, J. Sifakis, A. Bouajjani und S. Bensalem: *Property Preserving Abstractions for the Verification of Concurrent Systems.* Formal Methods in System Design, 6, 1995.

[LMM99a] Latella, D., I. Majzik und M. Massink: *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams using the SPIN Model-Checker.* Formal Aspects of Computing, 11:637–664, 1999.

[LMM99b] Latella, D., I. Majzik und M. Massink: *Towards a Formal Operational Semantics of UML Statechart Diagrams*. In: *Proceedings of Formal Methods for Open Object-Based Distributed Systems*. Kluwer Academic Publishers, 1999.

[Lon93] Long, D.E.: *Model Checking, Abstraction and Compositional Verification*. Doktorarbeit, CMU School of Computer Science, 1993.

[LP99a] Lilius, J. und I. Porres Paltor: *Formalising UML State Machines for Model Checking*. In: France, R. und B. Rumpe (Herausgeber): *Proceedings of the International Conference on the Unified Modeling Language (UML)*, Band 1723 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 430–445. Springer-Verlag, 1999.

[LP99b] Lilius, J.L. und I. Porres Paltor: *vUML: a Tool for Verifying UML Models*. Technischer Bericht TUCS-TR-272, Turku-Centre for Computer Science, Finnland, 1999.

[LT88] Larsen, K. G. und B. Thomsen: *A Modal Process Logic*. In: *Third Annual Symposium on Logic in Computer Science*, Computer Society Press, Seiten 203–210. IEEE, 1988.

[Mar91] Maraninchi, F.: *The Argos Language: Graphical Representation of Automata and Description of Reactive Systems*. In: *Workshop on Visual Languages*. IEEE, 1991.

[Mar92] Maraninchi, F.: *Operational and Compositional Semantics of Synchronous Automaton Compositions*. In: *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, Band 630 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 550–564. Springer-Verlag, 1992.

[McM] McMillan, K.: *SMV homepage. http://www.kenmcmil.com/smv.html*.

[McM93] McMillan, K.: *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mer97] Merz, S.: *Rules for Abstraction*. In: Shyamasundar, R. K. und K. Ueda (Herausgeber): *Advances in Computing Science (ASIAN)*, Band 1345 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 32–45. Springer-Verlag, 1997.

[Mik00] Mikk, E.: *Semantics and Verification of Statecharts*. Doktorarbeit, Christian Albrechts Universität Kiel, Deutschland, 2000.

[MLS97] Mikk, E., Y. Lakhnech und M. Siegel: *Hierarchical Automata as Model for Statecharts*. In: *Proceedings of Asian Computing Science Conference (ASIAN)*, Band 1345 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1997.

[MLSH99] Mikk, E., Y. Lakhnech, M. Siegel und G.J. Holzmann: *Implementing Statecharts in Promela/SPIN*. In: *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Seiten 90–101. IEEE Computer Society, 1999.

[MN97] Müller, O. und T. Nipkow: *Traces of I/O Automata in Isabelle/HOLCF*. In: Bidoit, M. und M. Dauchet (Herausgeber): *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Band 1214 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 580–594. Springer-Verlag, 1997.

[MP91] Manna, Z. und A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[MSS86] Melton, A., D.A. Schmidt und G.E. Strecker: *Galois Connections and Computer Science Applications*. In: Pitt, D., S. Abramsky, A. Poigne und D. Rydeheard (Herausgeber): *Category Theory and Computer Programming*, Band 240 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 299–312. Springer-Verlag, 1986.

[NP] Nipkow, T. und L.C. Paulson: *Isabelle homepage. http://isabelle.in.tum.de/*.

[NPW02] Nipkow, T., L.C. Paulson und M. Wenzel: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Band 2283 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2002.

[Obj03] Object Management Group: *OMG Unified Modeling Language Specification, Version 1.5*, März 2003.

[ORS92] Owre, S., J.M. Rushby, und N. Shankar: *PVS: A Prototype Verification System*. In: Kapur, D. (Herausgeber): *International Conference on Automated Deduction (CADE)*, Band 607 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 748–752. Springer-Verlag, 1992.

[Pau87a] Paulson, L. C.: *The Foundation of a Generic Theorem Prover*, 1987.

[Pau87b] Paulson, L.C.: *Logic and Computation, Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.

[Pau89] Paulson, L.C.: *The Foundation of a Generic Theorem Prover*. Journal of Automated Reasoning, 5:363–397, 1989.

[Pau92] Paulson, L. C.: *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, 1992.

[Pau94] Paulson, L. C.: *Isabelle – A Generic Theorem Prover*, Band 828 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1994.

[Pau96] Paulson, L. C.: *ML for the Working Programmer*. Cambridge University Press, 1996.

[Pnu77] Pnueli, A.: *The Temporal Logic of Programs*. In: *18th IEEE-Symposium on Foundations of Computer Science (FOCS)*, Seiten 46–57. IEEE, 1977.

[Por01] Porres, I.: *Modeling and Analyzing Software Behaviour in UML*. Doktorarbeit, Abo Akademi University, Finland, 2001.

[PS91] Pnueli, A. und M. Shalev: *What is a Step: On the Semantics of Statecharts.* In: Ito, T. und A.R. Meyer (Herausgeber): *Theoretical Aspects of Computer Software*, Band 526 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 244–265. Springer-Verlag, 1991.

[QS82] Quielle, J. P. und J. Sifakis: *Specification and verification of concurrent systems in CESAR.* In: Dezani-Ciancaglini, M. und U. Montanari (Herausgeber): *Proceedings of the Fifth International Symposium on Programming*, Band 137 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1982.

[Reg94] Regensburger, F.: *HOLCF: Eine konservative Erweiterung von HOL um LCF.* Doktorarbeit, Technische Universität München, Deutschland, 1994.

[Reg95] Regensburger, F.: *HOLCF: Higher Order Logic of Computable Functions.* In: Schubert, T.E., P.J. Windley und J. Alves-Foss (Herausgeber): *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Band 971 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 293–307. Springer-Verlag, 1995.

[Rei92] Reif, W.: *The KIV System: Systematic Construction of Verified Software.* In: *International Conference on Automated Deduction (CADE 1992)*, Band 2392 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 753–757. Springer-Verlag, 1992.

[Ros94] Roscoe, A.W.: *Model-Checking CSP.* In: *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Seiten 353–378. Prentice Hall International, 1994.

[Ros05] Roscoe, B.: *The Theory and Practice of Concurrency.* Prentice Hall, 2005.

[RW06] Roscoe, B. und Z. Wu: *Verifying Statemate Statecharts Using CSP and FDR.* In: Liu, Z. und J. He (Herausgeber): *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*, Band 4260 der Reihe *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2006.

[San99] Santen, T.: *A Mechanized Logical Model of Z and Object-Oriented Specification.* Doktorarbeit, Fachbereich Informatik, Technische Universität Berlin, Deutschland, 1999.

[San00] Santen, T.: *A Mechanized Logical Model of Z and Object-Oriented Specification.* Shaker-Verlag, 2000.

[Sau06] Sauerbier, A.: *Prädikatenabstraktion von Statecharts mit Hilfe von Contraintlösern.* Diplomarbeit, Fakultät IV, Technische Universität Berlin, Deutschland, 2006.

[SEGW03] S, Ben-David, C. Eisner, D. Geist und Y. Wolfsthal: *Model Checking at IBM.* Formal Methods in System Design, 22(2):101–108, 2003.

[SHS03a] Seifert, D., S. Helke und T. Santen: *Conformance Testing for Statecharts.* Technischer Bericht 03-01, Fakultät IV, Technische Universität Berlin, Deutschland, 2003.

[SHS03b] Seifert, D., S. Helke und T. Santen: *Test Case Generation for UML Statecharts*. In: *Proceedings of International Conference on Perspectives of System Informatics (PSI 2003)*, Band 2890 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 462–468. Springer-Verlag, 2003.

[SKM01] Schäfer, T., A. Knapp und S. Merz: *Model Checking UML State Machines and Collaborations*. Electronic Notes in Theoretical Computer Science (ENTCS), 55(3):13, 2001.

[Smi92] Smith, G. P.: *An Object-Oriented Approach to Formal Specification*. Doktorarbeit, University of Queensland, 1992.

[Spi92] Spivey, J.M.: *The Z Notation – A Reference Manual*. Prentice Hall, 2nd Auflage, 1992.

[SS99] Saïdi, H. und N. Shankar: *Abstract and Model Check While You Prove*. In: Halbwachs, N. und D. Peled (Herausgeber): *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, Band 1633 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 443–454. Springer-Verlag, 1999.

[Tra00] Traore, I.: *An Outline of PVS Semantics for UML Statecharts*. Journal of Universal Computer Science, 6(11):1088–1108, 2000.

[TS05] TNI-Software: *Safety-Checker Blockset V2.2. User's Manual*, 2005.

[Tse03] Tserendorj, T.: *Automatische Verifikation von Statecharts durch Anbindung eines Model Checkers an einen Theorembeweiser*. Diplomarbeit, Fakultät IV, Technische Universität Berlin, Deutschland, 2003.

[TSOR04] Thums, A., G. Schellhorn, F. Ortmeier und W. Reif: *Interactive Verification of Statecharts*. In: Ehrig, H. (Herausgeber): *Integration of Software Specification Techniques for Applications in Engineering*, Band 3147 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 355–373. Springer-Verlag, 2004.

[Tve05] Tverdyshev, S.: *Combination of Isabelle/HOL with Automatic Tools*. In: Gramlich, B. (Herausgeber): *Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS)*, Band 3717 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 302–309. Springer-Verlag, 2005.

[vdB94] Beek, M. von der: *A Comparison of Statecharts Variants*. In: *ProCoS: Proceedings of the International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Seiten 128–448. Springer-Verlag, 1994.

[Ver] *Verisoft homepage. http://www.verisoft.de/index.html*.

[vH05] Hanxleden, R. von: *Modellierung Reaktiver Systeme – Synchrone Sprachen und Statecharts*. In: *Software Engineering Eingebetteter Systeme*, Seiten 377–405. Spektrum Akademischer Verlag, 2005.

[Win01] Winter, K.: *Model Checking Abstract State Machines*. Doktorarbeit, Fakultät IV, Technische Universität Berlin, Deutschland, 2001.