Zur Homepage der Dissertation

CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Abstraktion und Gegenbeispiel-gelenkte Konstruktion von ω-Automaten zur Verifikation Schritt-diskreter linearer Hybrider Systeme

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

von

Dipl.-Inform. Marc Segelken

Gutachter:

Prof. Dr. Werner Damm
Prof. Dr. Bernd Becker

Tag der Disputation: 11. Juli 2007

Vorwort

Eingebettete Systeme nehmen heute eine Vielzahl von Funktionen war und sind in vielen technischen Anwendungsbereichen nicht mehr weg zu denken. Die Wichtigkeit dieser Systeme wird unterstrichen von den zentralen Aufgabenbereichen, die Eingebette Systeme insbesondere in Transportsystemen inzwischen wahrnehmen. Nicht selten sind diese sicherheitskritisch, d.h. eine Fehlfunktion kann schnell nicht nur zu besonders hohen Folgekosten führen, sondern sogar Leib und Leben von Menschen gefährden. Daraus ergibt sich die große Praxisrelevanz für die Entwicklung und Anwendung automatisierter Verfahren im Rahmen des Entwicklungsprozesses, mit denen die korrekte Funktionsweise solcher Systeme nachgewiesen werden kann. Die Schwierigkeit der Bewältigung der Komplexität dieser Systeme zeigt sich daran, daß trotz jahrelanger weltweiter Forschung heute solche Verfahren nur auf kleine Teilsysteme erfolgreich angewendet werden können. Je nach Anwendungsbereich ist vielfach die Verwendung kontinuierlicher Größen zur Wahrnehmung regelungstechnischer Aufgaben ein erhebliches Hindernis, da gerade hier die industriellen Anforderungen in Bezug auf Systemgröße und die aktuellen Möglichkeiten nach Stand der Forschung noch weit auseinanderklaffen.

Ein solches Themengebiet eignet sich daher ideal für eine Dissertation und ich freue mich, daß ich in der Abteilung meines Forschungsinstitutes ideale Vorraussetzungen vorgefunden habe, um mich mit diesem Thema eingehend zu beschäftigen.

Hier wäre zunächst die technologische Grundlage zu erwähnen, die ich in Form von leistungsfähigen Werkzeugen zur formalen Verifikation aufbauend auf einer ideal geeigneten, abteilungsinternen Modellierungsprache bereits vorgefunden habe. Dazu kommen viele bereits existente Kontakte zu Partnern aus der Industrie, die nicht nur eine Vielzahl von Beispielanwendungen beisteuerten, sondern auch einen Einblick in typische Probleme im derzeitigen Entwicklungsprozess erlaubten, was mir insbesondere im Rahmen der EU-Projekte SafeAir und SafeAir 2, sowie weiteren Projekten mit bekannten Firmen der Automobilindustrie zu Nutze kam.

In der bemerkenswert großen Schnittmenge an gemeinsamen Problemen war die Notwendigkeit der Verifikation von eingebetteten Systemen mit kontinuierlichen Kontrollfunktionenen schnell wiederzufinden.

Auch hier gab es in meiner Abteilung bereits Vordenker, die vielversprechende Ideen bereits zu Papier gebracht [BDG+98] und z.T. bereits in exemplarischer, abgewandelter Form implementiertiert haben [Bri99], so daß ich bereits viel von deren Funktionsweise, Stärken und Schwächen lernen konnte. Auch, wenn auf algorithmischer Ebene letztlich keine direkte Verwandtschaft mehr zu erkennen sein sollte, so sind es doch zentrale Ideen dieser Vorreiter, die in der vorliegenden Arbeit mit eingeflossen sind.

Aufgrund des praxisnahen Einstiegs, den mir die beschriebene Umgebung in diese Thematik ermöglichte, bekam ich eine andere Perspektive für die Problemstellung, als dies bei der üblichen Herangehensweise, der Publikationsrecherche nach bestehenden akademischen Problemstellungen und deren bereits existenten Lösungsansätze, der Fall gewesen wäre. Dies führte im Vergleich zum "Mainstream" der existenten Ansätze zur Verifikation von hybriden Systemen zu einer etwas untypischen Herangehensweise, sowie zu einer Spezialisierung auf die Charakteristik industrieller Fallbeispiele, die sich erheblich von akademischen Beispielen unterscheidet.

Das Ergebnis ist ein Verfahren, das bei der praxisrelevanten Zielklasse der Modelle eine sehr erfreuliche Effizienz aufzeigt. Gleichzeitig ermöglicht der Vergleich mit akademisch getriebenen Ansätzen jedoch eine aufschlussreiche Betrachtung von Unterschieden, aber auch Gemeinsamkeiten, aus der sich Erkenntnisse zur weiteren Verbesserung nicht nur dieses hier vorgestellten Verfahrens, sondern auch

künftiger anderer Verfahren ableiten lassen. Ich wünsche dem Leser viel Spaß und neue Einsichten in die Thematik bei der Studie dieser Dissertation.

Zusammenfassung

Die sich stetig erhöhende Anzahl eingebetteter Systeme nicht nur in allgegenwärtigen elektronischen Geräten, sondern insbesondere auch in sicherheitskritischen Einsatzumgebungen wie dem Transportwesen (Fahrzeugelektronik, Avionik) stellt die Entwicklung dieser Systeme vor immer größer werdenden Herausforderungen, da trotz steigender Komplexität die sicherheitskritischen Anforderungen an solche Systeme gewährleistet werden müssen. Dank des modellbasierten Entwurfsprozesses können bereits in frühen Phasen der Entwicklung formale Verifikationsmethoden zum Nachweis sicherheitskritischer Eigenschaften eingesetzt werden, welche diese Eigenschaften entweder bestätigen oder Widerlegungen selbiger in Form von Simulationsläufen des Systems liefern können, die Eigenschaftsverletzungen aufzeigen.

Industriell eingesetzte Verfahren, wie z.B. das Model-Checking, sind jedoch auf diskrete Systeme eingeschränkt und können nicht direkt mit Berechnungen auf kontinuierlichen Variablen umgehen, welche zunehmend auch in eingebetteten Systemen zum Einsatz kommen. Zur Behandlung solcher Systeme mit unendlichen Zustandsräumen bieten sich u.a. Abstraktionstechniken an, welche unter geeigneter Erweiterung des Systemverhaltens die Elimination aller kontinuierlichen Berechnungen ermöglichen, ohne ein etwaiges Fehlverhalten des Systems dabei zu verlieren. Das so vereinfachte System kann dann anstelle des komplexeren Ausgangssystems dem Model-Checking zum Nachweis von Eigenschaften unterzogen werden. Da das Systemverhalten durch Abstraktion erweitert und nicht eingeschränkt wird, können somit auf vereinfachten, endlichen Modellen Eigenschaften für Systeme mit unendlichen Zustandsräumen nachgewiesen werden.

Die Verwendung von Abstraktionstechniken führt bei zu starker Vereinfachung der Systeme i.d.R. jedoch zu ungültigen Widerlegungen der nachzuweisenden Eigenschaften, welche als solche zwar erkannt werden können, jedoch für den Anwendungsfall in dieser Form nicht zweckdienlich sind. Zur Vermeidung dieser ungültigen Widerlegungen haben sich automatische iterative Abstraktionsverfeinerungstechniken bewährt, in denen abhängig von den bisherigen ungültigen Widerlegungen das abstrakte Modell gezielt dergestalt verfeinert wird, daß nachfolgende Anwendungen des Model-Checkings insbesondere die bekannten falschen Widerlegungen nicht wiederholt als Ergebnis liefern können. Die iterative Wiederholung von Verfeinerung der Abstraktion und Anwendung des Model-Checkers ermöglicht häufig eine Konvergenz des schrittweise verfeinerten abstrakten Systems zu einem Modell, auf dem schließlich eine Eigenschaft nachgewiesen oder nachvollziehbar widerlegt werden kann.

Gegenstand dieser Arbeit ist ein neues iteratives Abstraktionsverfeinerungsverfahren für (Schritt-diskrete) Hybride Systeme, welches als Semientscheidungsverfahren durch inkrementelle Konstruktion eines ω-Automaten und durch dessen Parallelkomposition mit dem abstrakten Modell jegliches bereits entdeckte ungültige Verhalten des abstrakten Modells bzgl. der Berechnungen auf dem kontinuierlichen Zustandsraum für künftige Iterationen ausschließt und somit bei wiederholter Anwendung des formalen Verifikationsverfahrens auf das verfeinerte abstrakte Modell entweder zu einer belastbaren Bestätigung der untersuchten Eigenschaft bzw. zu einer nachweisbar gültigen Fehlerdiagnose in Form eines Fehlerpfades gelangen kann.

Das Verfahren eignet sich für im industriellen Kontext beobachtbare kontrolldominierte Systeme, welche sich durch einen großen diskreten Zustandsraum und einer relativ geringen Interaktion desselben mit Berechnungen auf dem kontinuierlichen Zustandsraum auszeichnen. Durch Ausnutzung der hohen Multiplizität kontinuierlicher Berechnungen im diskreten Zustandsraum kann das Verfahren auf dieser Modellklasse häufig in wenigen Iterationen zu einem Ergebnis gelangen.

Abstract

Embedded systems are increasingly used throughout modern society. These systems operate devices ranging from handheld devices to components in safety critical systems, such as automotive and avionic systems. As the range of applications for these systems inside of safety critical operations expands, so does the need for better scaling methods of requirement checking. With model-based development processes, it is possible to apply formal verification methods, already in early phases of the development process, to ensure compliance to the requirements of system critical systems. These methods can approve or disprove requirements by providing simulation scenarios which reveal the faulty behavior.

Industrially applied methods like model-checking are restricted to discrete systems and cannot be applied directly to models containing computations on continuous variables, which are increasingly used in embedded systems. To cope with such models comprising infinite state spaces among other approaches abstraction techniques have proven of value. Abstraction techniques extend the behavior of a system such that continuous computations can be eliminated from the model without loosing any behavior possibly violating the property. Model-Checking is applied to the simplified system instead of the original system to prove properties. Since the behavior of the system is only extended and not restricted by abstraction, properties for systems with infinite state spaces can be approved on simplified finite models.

The application of abstraction techniques usually causes invalid falsification of properties if the systems are oversimplified. This is detected, in fact, but is not serviceable in the given use case. To avoid such false negatives automatic iterative abstraction refinement techniques have proven of value. Based on previous invalid falsification information the abstract model is selectively refined such that subsequent model-checking applications do not result in already known false negatives again. The iterative repetition of applying refinement and model-checking enables a convergence of the step-wise refined abstract system to a model, on which a property can finally be proved or comprehensibly disproved.

Subject of this thesis is a new iterative abstraction refinement approach as a semi-decision procedure for the verification of (step-discrete) hybrid systems, which, by construction of a learning ω-automaton and its parallel composition with the abstract model, excludes any spurious behavior being observed so far for subsequent iterations. Thus, repeated application of the formal verification procedure on the refined abstract model might lead to a resilient affirmation of the property under verification or to its verifiable valid refutation.

The approach is suitable for control dominated systems being observable in industrial contexts, which exhibit huge discrete state spaces coming along with relatively few interactions with the computations on the continuous state space. By exploiting repeated occurrences of continuous computations in the discrete state space the approach is able to provide a result within few iterations in many cases.

Danksagungen

Ich möchte an dieser Stelle einigen Personen ausdrücklich meinen Dank aussprechen, welche mir direkt oder indirekt die Abfassung dieser Dissertation ermöglicht oder diese unterstützt haben.

Mein Dank gilt als erstes Prof. Dr. Werner Damm, der die Rahmenbedingungen für die Erstellung dieser Arbeit geschaffen und in vielen Punkten wertvolle Hilfestellungen gegeben hat. Desweiteren danke ich sowohl Prof. Damm, als auch Prof. Dr. Bernd Becker für die Begutachtung dieser Arbeit.

Weiterhin gilt mein Dank Dr. habil. Hardi Hungar, von dem maßgeblich die im Vorwort erwähnten ursprünglichen Ideen ausgingen und der mich während der Arbeit betreut hat, sowie Prof. Dr. Martin Fränzle, welcher viele wichtige Anmerkungen beigesteuert hat. Ebenfalls danke ich allen Kollegen, die mir in Diskussionen und informativen Unterhaltungen zu vielen Erkenntnissen verholfen haben, die in diese Arbeit eingeflossen sind. Hier wären z.B. Dipl. Inform. Christian Herde, Dipl. Inform. Alexander Metzner, Dipl. Inform. Tino Teige, Dipl. Inform. Boris Wirtz und Dipl. Inform. Jürgen Niehaus zu nennen. Insbesondere möchte ich Dr. Hartmut Wittke für seine Hilfe bei allen Integrationsfragen und -problemen, für die überaus hilfreichen Informationen und Diskussionen, sowie seinem großem Engagement beim Korrekturlesen und Kommentieren einer Rohversion dieser Dissertation danken. Darüber hinaus gibt es eine Vielzahl von in dieser Arbeit verwendeten Software-Modulen, die neben den bereits genannten Kollegen von vielen weiteren erstellt und gepflegt wurden, wie z.B. Dr. Tom Bienmüller, Dipl. Inform. Rainer Lochman und Dr. Jürgen Bohn.

Für die gute Arbeitsatmosphäre im Bereich sicherheitskritischer Systeme im OFFIS-Institut, die ebenfalls die Erstellung der Dissertation sehr begünstigt hat, danke ich apl. Prof. Dr. Bernhard Josko. Auch die Projekte, aus denen diese Arbeit hervorgegangen ist, sind ohne die Mitwirkung der Projektteilnehmer nicht möglich gewesen, und so möchte ich den Partnern der SafeAir-Projekte danken und hier insbesondere Philippe Baufreton, der diese als Projektkoordinator begleitet hat.

Abschließend gilt mein Dank meiner gesamten Familie, die mir nicht zuletzt die für diese Arbeit nötige Ausbildung ermöglicht und mich für diese motiviert haben.

Inhaltsverzeichnis

Algorithmenverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Eingebettete Systeme finden sich heutzutage in nahezu allen elektronischen Systemen, wie sie in industriellen Prozeßautomatisierungen, Transportvehikel und Kommunikationssystemen bis hin zu einfachen Haushaltsgeräten zum Einsatz kommen. Sie sind dadurch charakterisiert, daß sie als Teil eines Gesamtsystems eine spezifische Aufgabe innerhalb ihrer (physikalischen) Umgebung wahrnehmen, mit welcher sie fortwährend Informationen und Signale über Kommunikationsleitungen oder über Sensoren und Aktuatoren austauschen, mit dieser also in intensiver Wechselwirkung stehen. Diese Systeme sind als reaktive Systeme konzipiert, die im Allgemeinen nicht terminieren und deren Verhalten somit auch nicht durch endliche Ein-/Ausgaberelationen beschrieben werden kann, wie dies bei transformierenden System der Fall ist.

Besaßen eingebettete Systeme früher eher informative Funktionalität, z.B. für Situationsanalysen (Anzeigen einer Geschwindigkeit) oder Situationsbewertung (Warnanzeigen bei Erreichen definierter Situationen), so werden sie heute zunehmend zur Aktionsausführung (X-by-wire Anwendungen) oder gar zur aktiven Kontrolle wichtiger Prozesse (Autopiloten) eingesetzt.

Durch die zunehmende Übernahme vieler Teilfunktionen durch eingebettete Systeme werden deren Anzahl und Komplexität immer höher. So finden sich heutzutage in einem marktüblichen Automobil schon mehr als 80 Mikroprozessoren, die in verteilter Form einen Gesamt-Softwareumfang von mehreren Millionen Zeilen Source-Code ausführen. Die Kosten für die Erstellung solcher Funktionalitäten schießt dabei überproportional in die Höhe. Bereits im Jahre 2000 machten diese bis zu 30% der Fahrzeugkosten aus (W.Reitzle, BMW,2000) und bereits zu dieser Zeit ist man davon ausgegangen, daß 90% der Innovationen eines Fahrzeugs sich in der Elektronik wiederfinden (Daimler Chrysler Technology Conference, Stuttgart, 2000).

Die von eingebetteten Systemen wahrgenommenen Aufgaben sind vielfach sicherheitskritischer Art, d.h. ein Versagen des Systems kann tödliche Konsequenzen oder doch zumindest ökonomisch katastrophale Folgen haben. Als Beispiele wäre hier ein Fehlverhalten der Bremsenansteuerung in einem Automobil zu nennen, welches sicherlich als sicherheitskritisch einzustufen ist. Selbst bei glimpflichem Ausgang des ersten Vorfalls eines fehlerhaften Bremssteuerungsverhaltens ist die finanzielle Belastung der umgänglich zu erfolgenden Rückrufaktion enorm. Daher sind diese Art von Systemen vor ihrem praktischen Einsatz im besonderen Maße hinsichtlich ihrer Korrektheit zu überprüfen - eine immer schwieriger werdende Herausforderung bei wachsender Komplexität, Anzahl und Interoperabilität eingebetteter Systeme in fast allen Einsatzbereichen.

Modellbasierte Entwurfsprozesse haben sich bei der Entwicklung solcher Systeme bewährt, da bereits in frühen Phasen der Entwicklung formale ausführbare Spe-

zifikationen vorliegen, die bereits auf die gegebenen Anforderungen und mögliche Probleme hin untersucht werden können. Dabei kommen im industriellen Kontext i.d.R. Simulations- und Test-basierte Verfahren zum Einsatz. Bei steigender Größe und Komplexität dieser Systeme ist durch solche Verfahren jedoch immer weniger zu gewährleisten, daß alle relevanten Szenarien durchgespielt werden können, zumal gleichzeitig die so genannte *Time-to-Market*-Spanne einen engen Zeitrahmen diktiert, welcher tendenziell immer kleiner wird.

An dieser Stelle sind Analysemethoden wie insbesondere die formale Verifikation der Modelle zur automatischen Überprüfung der textuell dokumentierten Spezifikationen eine große Hilfe, da sie bei wenig zu investierendem Aufwand in der Anwendung eine belastbare umfassende Zusicherung der Einhaltung der Spezifikation nachweisen können, bzw. eine Verletzung derselben einschließlich detaillierter Diagnoseinformationen aufdecken können. Die heutigen Grenzen beim Einsatz dieser Technologie sind allgemein in der Komplexität der Modelle zu finden, da auf dem Stand der Technik nur eine Modellgröße verifiziert werden kann, die i.d.R. nur die Verifikation einzelner Subkomponenten größerer Systeme gestattet. Insbesondere eine besondere Teilklasse eingebetteter Systeme, die so genannten Hybriden Systeme, welche (Zeit-kontinuierliche) Berechnungen auf einem kontinuierlichen Zustandsraum durchführen, stellen eine große Herausforderung dar und die bislang bekannten Methoden sind weit von einer Anwendung auf industrielle Modelle entfernt, da die Komplexitätsgrenze bereits von kleinen Modellen gesprengt wird.

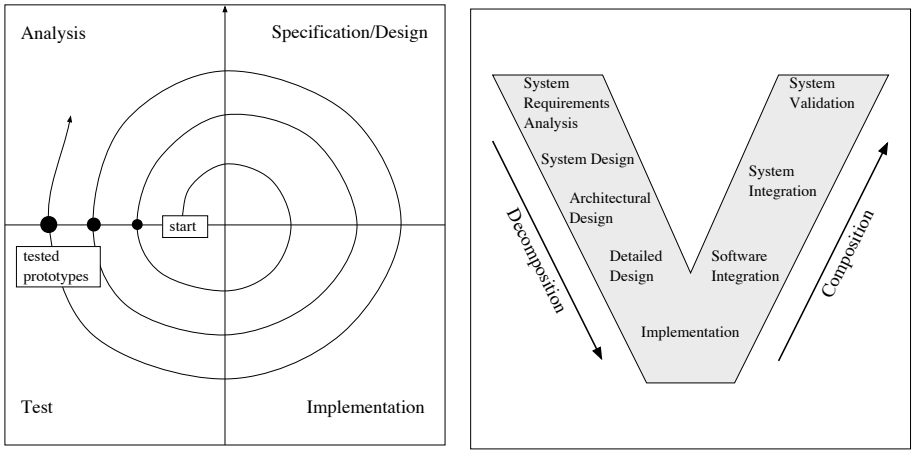
Zur Reduzierung der Modellkomplexität im Allgemeinen sind Abstraktionsverfahren ein vielversprechender Ansatz, da sie unter Beibehaltung des relevanten Verhaltens des Modells selbiges signifikant mitsamt seiner Komplexität vereinfachen können, ohne dabei notwendigerweise die Qualität der Beweisaussage zu mindern. Für Hybride Systeme sind Abstraktionsverfahren besonders interessant, da sie aus einem unendlich großen Zustandsraum, aufgespannt durch eine Anzahl endlicher diskreter Zustände kombiniert mit allen denkbaren Zuständen eines mehrdimensionalen Kontinuums, ein rein diskretes, endliches Modell erzeugen können, auf dem die nachzuweisende Eigenschaft trotz der Transformation bewiesen, bzw. einschließlich Diagnoseinformationen widerlegt werden kann.

Auch mit diesem Lösungsansatz sind existierende Methoden hinsichtlich der handhabbaren Modellgröße schnell an ihre Grenzen zu bringen. Dies gilt im Besonderen für Hybride Systeme, die neben vielen kontinuierlichen Dimensionen auch einen großen diskreten Zustandsraum aufweisen, wie dies häufig in industriellen Modellen der Fall ist. An dieser Stelle soll das in dieser These vorgestellte ω-CEGAR Verfahren [Seg07] die Komplexitätsgrenzen deutlich nach oben verschieben.

1.1 Modell-basierter Entwurfsprozeß

Ein Modell-basierter Entwurfsprozeß beinhaltet bereits in frühen Phasen der Entwicklung eines Systems ein ausführbares, abstraktes Modell der Spezifikation, welches zumeist in graphischer Form vorliegt. Ein solches Vorgehen wird insbesondere von Computer Aided Software Engineering Werkzeugen (CASE-*Tools*) unterstützt und birgt folgende Vorteile in sich:

- Anforderungen werden strukturell modelliert (*Divide & Conquer*)
- Festlegung von Schnittstellen, Kommunikationsverhalten und Interoperabilität erfolgt frühzeitig
- Graphische Repräsentationen begünstigen Austausch und Verständnis der Spezifikation zwischen Ingenieuren



(a) Spiralmodell (b) V-Modell

Abbildung 1.1: Modelle des Entwurfsprozesses

- Gegenüber textuellen Anforderungsdefinitionen stellen Spezifikationen in Form von abstrakten Modellen formal eindeutige Anforderungen dar und erleichtern die Anforderungsanalyse
- Ausführbarkeit durch Simulation erlaubt frühe Analyse des Entwurfs auf der Systemebene und ermöglicht frühe Detektion von Entwurfsfehlern
- Durch Einsatz von CASE-Tools besteht die Möglichkeit der graphischen Animation des Modells während der Simulation
- Inkrementelle Verfeinerung der Systemkomponenten unterstützt Entwurfs- und Entwicklungsprozeß
- CASE-Tools ermöglichen automatische Code-Generierung, d.h. die Spezifikationssprache wird bis zur Erzeugung des Produktionscodes für Zielprozessor (*Targetcode*) beibehalten
- CASE-Tools ermöglichen automatische Generierung von Testszenarios zur Validierung von Spezifikation und Implementation, sowie dem in Hardware umgesetzten System

Die Vorgehensweise in einem modellbasierten Entwurfsprozeß wird dabei sowohl durch das Spiralmodell [Boe86] als auch durch das V-Modell [ESt97] organisiert, welche in Abbildung 1.1 gegenübergestellt sind. Beide Modelle beschreiben ein iteratives Vorgehen und legen die jeweiligen Phasen des Entwurfsprozesses fest. Im Spiralmodell zeigt der Winkel der Position in der Spirale die jeweilige Entwurfsphase an. Demgegenüber beschreibt das V-Modell eher die Entwurfsphasen mit Hinblick auf die Strukturierung und Abstraktion der Anforderungen auf der Ebene des Systementwurfs, des Architekturentwurfs, der Verfeinerung des Entwurfs und der Implementation. Die mit diesen Phasen einhergehende Dekomposition des Modells in Teilkomponenten wird in dem zweiten Strang des V-Modells durch die jeweilig dazugehörige Integration auf Software- und Systemebene wieder aufgehoben.

Es existieren verschiedene Qualitätsstandards, welche in Abhängigkeit des Anwendungsbereichs unterschiedlich abgestufte Qualitätsabsicherungen nicht nur bzgl. des Produkts, sondern auch bzgl. des Entwicklungsprozesses definieren und

einfordern. Hier wäre z.B. der Standard Do-178B [RTC92] zu nennen, der in der Luftfahrt von großer Bedeutung ist.

1.2 Rolle der Formalen Verifikation

Zur Sicherung des Qualitätsstandards der erstellten Systeme werden in industriellen Prozessen vorwiegend die Verfahren Simulation und Testen eingesetzt. Bei Letzterem existieren verschiedene Abstufungen der zu erzielenden Überdeckung (*Coverage*), welche Kriterien einfacherer Art wie Anweisungsüberdeckungen (*Statement Coverage*) verwenden, oder aber auch komplexere Kriterien wie die Überdeckung sämtlicher Kontrollflußpfade des Systems (*Path Coverage*). Unabhängig von der Definition des Testkriteriums ist festzustellen, daß die Anzahl der möglichen verschiedenen Simulationsläufe vorgegebener Länge exponentiell in der Größe des erreichbaren Zustandsraums[1] des Systems anwächst, was eine Vollständigkeit von Test-basierten Verfahren zum Nachweis der Korrektheit eines Systems bei nicht-trivialer Größe praktisch unmöglich macht. Auf der anderen Seite sind Testverfahren aufgrund der Unvollständigkeit der Abdeckungen für Systeme beliebiger Größe anwendbar. Daher werden Testverfahren und automatische Testmuster-Generierungen vorwiegend für große Modelle und zur Validierung von Hardware-Implementierungen eingesetzt.

Aufgrund der Unvollständigkeit der Testverfahren bleiben in der Praxis immer wieder Randfälle des Systemverhaltens beim Testen unbeachtet, welche durch unerwartete oder nicht bedachte Kombinationen von Eingabebelegungen des Systems verursacht werden können. Genau hier sind jedoch oftmals die Fehlerquellen verborgen, welche zu katastrophalen Folgen führen können.

Genau an dieser Stelle sollten formale Verifikationsverfahren zum Einsatz kommen, da diese bezüglich einer formal spezifizierten Eigenschaft eine vollständige Analyse des Systems durchführen und mit einem mathematisch abgesicherten Ergebnis aufwarten können. Die nachzuweisenden Eigenschaften entspringen typischerweise der anfänglich erstellten Anforderungsdefinition, werden jedoch im Rahmen der im V-Modell empfohlenen strukturellen Dekomposition der Funktionalität durchaus auch mit lokalen Eigenschaften ergänzt, die sich nur auf Subkomponenten beziehen und häufig deren Zusammenspiel adressieren. Dieser Umstand kann zur Komplexitätsbewältigung im Rahmen von kompositionellen Verifikationsansätzen ausgenutzt werden [Wit06].

Formale Verifikationsmethoden können auf jede Komponente und jedes System angewendet werden, soweit es in einer allen Komponenten gemeinsamen Sprache repräsentiert ist und sich innerhalb der Komplexitätsgrenzen dieser Methoden bewegt. Im Spiralmodell sind formale Verifikationsmethoden somit in den Test- und Analysequadranten einzusetzen. Bei dem Modell-basierten Entwurfsprozeß liegt schon früh ein ausführbares Modell vor, welches entsprechend ebenso früh schon formalen Verfikationsprozessen unterzogen werden kann. Dadurch können bereits an dieser Stelle Fehler in der Spezifikation aufgedeckt werden. Im Rahmen der existenten Komplexitätsbeschränkungen kann die formale Verifikation beginnend an diesem Punkt im V-Modell-basierten Entwurfsprozeß alle Schritte bis zur Software-Integration begleiten und dabei die Korrektheit der Verfeinerungen und Implementierungen der beteiligten Komponenten sicherstellen.

Der anschließende Schritt der Systemintegration schließt i.d.R. die Ebene der konkreten Hardware-Integration mit ein, welche sich formalen Verifikationsverfahren entzieht, da diese eine homogen repräsentierte Verhaltensbeschreibung erfordern. Hier muß entsprechend wieder auf Test-Verfahren zurückgegriffen werden,

[1] genauer: der Umfang der Transitionsrelation beschränkt auf erreichbare Zustände

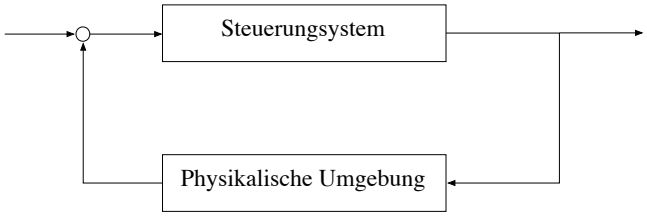


Abbildung 1.2: Geschlossener Regelkreis zwischen Steuerungssystem und physikalischer Umgebung

um ein korrektes Verhalten so weit wie praktisch möglich abzusichern.

Die Techniken der formalen Verifikation sind zu unterteilen in die Klasse der halb- und vollautomatischen. Erstere sind die deduktiven Verifikationsverfahren, welche Expertenwissen über deren Benutzung voraussetzen und mit erheblichem Aufwand in der Anwendung verbunden sind. Dieser Umstand ist ursächlich für deren geringe Verbreitung im industriellen Umfeld, da Ingenieure für gewöhnlich weder über das benötigte Expertenwissen verfügen, noch eine ausreichende Zeitspanne für eine solche Anwendung zur Verfügung steht. Von einem ökonomisch betrachteten Standpunkt sind solche Verfahren daher zu teuer, obwohl diese keine grundsätzlichen Beschränkungen hinsichtlich Modellkomplexität oder Endlichkeit des Zustandsraums voraussetzen. Im Gegensatz dazu sind die vollautomatischen Verifikationsverfahren aufgrund der Push-Button Technik prinzipiell sehr einfach anwendbar, wenn man einmal von der ohnehin zu erfolgenden Erstellung der nachzuweisenden formalen Eigenschaft absieht. Als Nachteil dieser Verfahren ist dafür jedoch die Einschränkung der analysierbaren Modelle hinsichtlich Komplexität und Endlichkeit des Zustandsraums in Kauf zu nehmen, wobei letztere Einschränkung durch Einsatz geeigneter Abstraktionsverfahren relativiert werden kann.

1.3 Besonderheiten Hybrider Systeme

Eingebettete Systeme nehmen neben ihren diskreten Verhalten hinsichtlich interner diskreter Zustandsabläufe und ihrer Kommunikation mit der Außenwelt über diskrete Signale auch häufig die Steuerung regelungstechnischen Aufgaben war. Aufgabengebiete sind hier beispielsweise die Stabilisierung instabiler Regelstrecken, Regulierungen auf Sollwerte, Trajektorienfolgen durch Folgeregler, Störentkopplungen und Überwachungen mit Fehlerdiagnose. Beispiele für solche Aufgaben finden sich bei Autopiloten in der Luftfahrt und Schiffahrt und in der Kraftfahrzeugtechnik bei Tempomaten, Antiblockiersystemen, Aktivlenkungen, usw.

Regelungstechnische Aufgabe beziehen sowohl in der Sensorik, als auch in der Aktuatorik die Verarbeitung analoger Wertgrößen ein, die als solche auch intern im Steuerungssystem eingelesen und repräsentiert werden müssen. Da analoge Größen ihrer Natur gemäß einen kontinuierlichen Wertebereich aufweisen, sind in der Digitaltechnik üblicherweise Kodierungen in Gleit- oder Festpunktdarstellung üblich, welche in diskreten Berechnungsschritten verarbeitet werden. Auf der anderen Seite muß bei vollständiger Berücksichtigung des geschlossenen Regelungssystems die durch Differentialgleichungen beschriebene Zeit-kontinuierliche Veränderung physikalischer Größen einbezogen werden, da eine digitalisierte Betrachtungsweise nur approximierend die Abläufe beschreiben kann.

Eine allgemeingültige Modellierung geschlossener Regelkreise muß daher notwendigerweise Zeitkontinuität mathematisch beschreiben, woraus sich die Defi-

nition Hybrider Systeme als Kombination von endlichen Automaten mit parallelen, mit diesen in Wechselwirkung stehenden Differentialgleichungen ergibt. Diese beinhalten ebenfalls Schwellwertbedingungen und unstetige Sprünge im Wertkontinuum, welches den diskreten Verhaltensanteil widerspiegelt und Hybride Systeme als solche charakterisiert.

Die verwendete Digitaltechnik eingebetteter Systeme reduziert diese formal auf diskrete Automaten endlichen Zustandsraums, so daß eine exakte Modellierung selbiger auf wertkontinuierliche Darstellungen verzichten kann und streng genommen auch muß, da numerisch motivierte Abweichungen idealisierter Berechnungen andernfalls nicht nachvollziehbar wären. Von diesem Standpunkt aus ist ein Hybrides System erst dann gegeben, wenn der gesamte Regelkreislauf betrachtet wird, mit einem rein diskreten Steuerungssystem und einer rein kontinuierlichen physikalischen Umwelt.

Diese Betrachtung ist in der formalen Verifikation aus komplexitätstechnischer Sicht jedoch problematisch, da bei heutigen eingebetteten Systemen bereits Prozessoren mit doppelt-genauer Gleitkommadarstellung zum Einsatz kommen, was bei exakter diskreter Modellierung des Steuerungssystems eine Zustandsraumvergrößerung um den Faktor 2^{64} für jede verwendete Gleitkommavariable beinhaltet. Zudem wird die Zustandsübergangsrelation bei Nachbildung paralleler arithmetischer Operationen insbesondere bei Multiplikations- und Divisionsoperationen die heutige Komplexitätsgrenze auch von symbolisch arbeitenden Model-Checkern sprengen, so daß eine formale Verifikation ohne Abstraktionsmethoden nicht möglich ist. Verfahren zur formalen Verifikation solcher Steuerungssysteme berücksichtigen daher im Allgemeinen nicht die durch die Digitaltechnik bedingte rein diskrete Modellrepräsentation[2], sondern behandeln als Wert-kontinuierlich anzusehende Größen weiterhin als solche, um die Komplexität des diskreten Verhaltensanteils des Hybriden Systems nicht weiter zu erhöhen. Dennoch bleibt auch bei dieser Betrachtungsweise folgende Unterscheidung zwischen diskreten Steuerungssystemen mit Wert-kontinuierlichen Verhaltensanteilen und geschlossenen Regelkreisen unter Einbeziehung physikalischer Gesetze zu beobachten:

- Geschlossene Regelkreise (*closed loop*) müssen für die Modellierung der physikalischen Umgebung Zeitkontinuität durch Differentialgleichungen berücksichtigen, während
- Steuerungssysteme innerhalb dieser Regelkreise (*open loop*) in der heute üblichen digitalen Verarbeitungstechnik nur Schritt-diskrete Berechnungen durchführen können und insbesondere nicht durch Differentialgleichungen beschrieben sind.

Dementsprechend folgt hieraus die Kategorisierung der Schritt-diskreten Hybriden Systeme, welche eine relevante Teilmenge der Zeit-kontinuierlichen Hybriden Systeme darstellen. Die Anforderungen und nachzuweisenden Eigenschaften werden im Rahmen des V-Modells des Entwurfsprozesses für die Steuerungssysteme entworfen, da die Anforderungen für den geschlossenen Regelkreis sich so nicht für einen Systementwurf eignen. Daher ist häufig das Steuerungssystem auch Gegenstand der formalen Verifikation, und nicht der geschlossene Regelkreis. Als Beispiel sei hier ein Autopilotsystem angeführt, welches bei Absinken der Geschwindigkeit automatisch den Schub erhöhen soll. Dies wäre die in der Praxis zu verifizierende Eigenschaft für das Flugzeug, und nicht, ob es mit Hinblick auf aerodynamische Naturgesetze und terrestrischen Bedingungen unter allen Umständen in der Luft

[2] Ausnahme bilden hier natürlich beispielsweise die formale Verifikation von Gleitkommaeinheiten von Prozessoren, bei denen eine vollständige Berücksichtigung der diskreten Modellierung ausdrücklich erwünscht ist.

bleibt, bis ihm das Kerosin ausgeht, wenn dies auch die zugrunde liegende Motivation ist.

Die Verifikation solcher Eigenschaften bilden die Problemstellung, der sich diese Dissertation für Schritt-diskrete Hybride Systeme widmet. Hier wird ein Verfahren vorgestellt werden, das diese Systeme bis zu einer Größe verifizieren kann, die hinsichtlich des Umfangs des diskreten Zustandsraums erheblich höher liegt, als die der vorhandenen Verfahren zur Verifikation Hybrider Systeme.

Der folgende Abschnitt vermittelt einen kurzen Überblick über die existenten Verfahren.

1.4 Überblick vorhandener Ansätze zur formalen Verifikation Hybrider Systeme

Es gibt eine Vielzahl von Ansätzen zur formalen Verifikation Hybrider Systeme, welche sich alle der Abstraktionstechnik bedienen, um den unendlichen Zustandsraum in einen endlichen zu transformieren. Eine Sonderrolle hierbei bilden die bereits erwähnten deduktiven Ansätze, welche mit sog. Theorembeweiser durch logisches Schließen eine Eigenschaft beweisen oder widerlegen können (z.B. PVS, [ORS92, OS03]). Da diese nicht mit Zustandsräumen rechnen, sind sie für hybride Modelle gleichermaßen geeignet wie für diskrete. Darüber hinaus gibt es viele Verfahren zur Verifikation Zeit-kontinuierlicher Hybrider Systeme, die im kontinuierlichen Zustandsraum geeignete Überapproximationen von Berechnungsabläufen bestimmen. Hier werden häufig Quader-, Polyeder- oder Ellipsoid-förmige Mengenbeschreibungen verwendet, um erreichbare Zustände zu charakterisieren und mithin den kontinuierlichen Zustandsraum zu partitionieren. Typisches Problem hierbei ist die aufwendige Zustandsrepräsentation für eine größere Anzahl von Dimensionen des kontinuierlichen Zustandsraums, welche zu verschiedenartigen Ansätzen der Vereinfachung selbiger geführt haben, z.T. auch mit Hilfe vom Benutzer zu spezifizierender Vorgaben wie Prädikate (z.B. [ADI02]) oder Normalvektoren (z.B. bei PHAVer [Fre05]), mit welchen eine kompaktere Darstellung und dadurch vereinfachte Durchführung der Verifikation erreicht werden soll.

Neben den bereits genannten wären weitere Vertreter dieser Form der Verifikation HyTech [HH94], HyperTech [HHMWT00], d/dt [ADM02], Kronos [BDM+98], Uppaal [BLL+96], veriShift [BT00], checkmate [SK00], sowie das in [ADI06] beschriebene Verfahren.

Die Ansätze sind hinsichtlich der Komplexität der von ihnen verifizierbaren Systemen jedoch stark eingeschränkt. Die Anzahl der Dimensionen des kontinuierlichen Zustandsraums darf einstellige Anzahlen i.d.R. nicht überschreiten und die Anzahl diskreter Zustände ist im Allgemeinen auf wenige Tausend beschränkt. Damit sind industrielle Anwendungen dieser Verfahren in dem derzeitigen Entwicklungsstand praktisch ausgeschlossen.

Desweiteren existieren Verfahren, die den kontinuierlichen Zustandsraum nicht direkt partitionieren, sondern ohne Betrachtung der letztlich immer implizierten Partitionierung eine Abstraktionsverfeinerung im diskreten Zustandsraum durchführen. Ebenso wie das Infinite-State-Cegar Verfahren [CFH+03] gehört das in dieser Dissertationsschrift vorgestellte ω-Cegar Verfahren [Seg07] dazu.

Unerwähnt bleiben sollen auch nicht die Ansätze des *Bounded Model Checkings*, wo SAT-basierte Verfahren das Verhaltensmodell des Hybriden Systems für eine vorgegebene Anzahl von Berechnungsschritten in eine MILP-Formel (*Mixed Integer Linear Programing*) übersetzen, welche dann mit Verfahren zur Lösung von Erfüllbarkeitsproblemen behandelt werden, wie z.B. in HySAT [FH05].

Für die historische Entwicklung dieser Arbeit von Bedeutung ist der Ansatz

des First-Order-Model-Checkings [BDG+98], welcher derzeit weiterentwickelt und evaluiert wird [DDH+06, DDH+07, DMO+07]. Bereits hier liegt die Idee zugrunde, große diskrete Zustandsräume durch eine BDD-Repräsentation mit dem bekannten CTL-Model-Check-Algorithmus zu verifizieren, während kontinuierliche Zustände durch eine kompakte, überapproximierende, symbolisch-algebraische Darstellung Berücksichtigung finden, die nach jedem Iterationsschritt des Model-Check-Suchalgorithmus durch einen integrierten First-Order-Model-Checker berechnet und überprüft werden.

Dieser Ansatz wurde in abgewandelter Form in [Bri99] dergestalt umgesetzt, daß potentiell auftretende Berechnungsfolgen in wählbarer Tiefe vorab berechnet und gegebenenfalls im abstrakten Modell unterbunden werden, sofern der Lösungsraum kontinuierlicher Zustände entsprechender diskreter Transitionsfolgen leer ist. Aufgrund der exponentiell anwachsenden Anzahl möglicher Berechnungsfolgen ist das Verfahren jedoch nicht praxistauglich für eine nicht-triviale Anzahl verschiedener regelungstechnischer Gesetze. Hervorzuheben ist jedoch hier, daß die Berechnungsfolgen bereits unabhängig von diskreten Transitionen betrachtet und ausgeschlossen wurden, so daß es für Systeme mit großem diskreten Zustandsraum geeignet war.

1.5 Neuer Ansatz zur Problemlösung

Das Problem der Verifikation von kontrolldominierten Schritt-diskreten Hybriden Systemen mit großen diskreten Zustandsräumen soll in dieser Dissertation durch ein neues iteratives Abstraktionsverfeinerungsverfahren namens ω-CEGAR behandelt werden. Dabei soll neben dem großen diskreten Zustandsraum auch ein großer kontinuierlicher Zustandsraum behandelt werden können, so daß mit Hilfe der verbreiteten CASE-Tools in industriellen Kontexten erstellte Modelle in ihrer typischen Charakteristik verifiziert werden können. Hierbei soll insbesondere auch die Zertifizierbarkeit ermöglicht werden, d.h. es muß möglich sein, die Einhaltung von Eigenschaften nachweisen zu können.

Die Zielstellung soll durch ein neues Abstraktionsverfahren erreicht werden, bei dem mit einer starken Vereinfachung des Systems beginnend ein abstraktes System schrittweise verfeinert wird, bis die zu verifizierende Eigenschaft nachgewiesen ist, oder diese nachweislich widerlegt werden kann. Im Unterschied zu existierenden Verfahren wird bei diesem neuen Ansatz aus den Iterationen mehr Information aus den ungültigen Widerlegungen gewonnen, wodurch eine stärkere Verfeinerung ermöglicht und mithin die Anzahl der benötigten Iterationen und damit die der Gesamtlaufzeit reduziert werden kann.

Die bereits erwähnten Ideen des First-Order-Model-Checkens aus [BDG+98] und [Bri99] finden dabei in sofern Verwendung, als daß sowohl ein BDD-basiertes Model-Check-Verfahren eingesetzt werden kann und ungültige Transitionsfolgen im diskreten Zustandsraum verhindert werden, als auch sich wiederholende kontinuierliche Berechnungsabläufe in verschiedenen diskreten Systemzustandsfolgen global ausgeschlossen werden, sofern sie keine mögliche Berechnungsfolge darstellen.

Umgesetzt wird das Verfahren durch umfassende Analysen ungültiger Widerlegungen zur Extraktion möglichst allgemeingültiger Gründe für deren Ungültigkeit (*Konflikte*) und den globalen Ausschluß dieser Gründe in künftigen Iterationen durch das Erlernen dieser Gründe durch einen ω-Automaten, welcher zur Verfeinerung des Systems mit einer einfachen Abstraktion kombiniert wird.

In dieser Dissertation wird neben der konzeptionellen Beschreibung des Verfahrens und seiner Umsetzung selbiges auf ein vorgestelltes Fallbeispiel angewandt, wodurch das Erreichen der gesetzten Zielstellung validiert wird.

1.6 Struktur der Dissertation

Diese Arbeit ist in acht Kapitel gegliedert, welche die folgenden Inhalte vorstellen:

Kapitel 2 gibt eine Einführung in die Grundlagen von Automaten, Transitionssystemen, Kripke-Strukturen, Temporalen Logiken, dem CTL-Model-Check-Algorithmus und dem Begriff der formalen Abstraktion.

In Kapitel 3 werden Hybride Automaten vorgestellt, sowohl in Zeit-kontinuierlicher, als auch in Schritt-diskreter Form. Hier findet eine sehr einfache syntaktische Struktur Verwendung, welche als konzeptionelle Repräsentation nachfolgend vorgestellte Algorithmen hinsichtlich der Formalisierung vereinfachen. Neben den einfach strukturieren Hybriden Automaten wird in diesem Kapitel auch die Struktur eines erweiterten Hybriden Automaten mit multiplen Transitionen zwischen zwei Zuständen vorgestellt. Mit der angegebenen Abbildung auf einfache Hybride Systeme kann das folgende Kapitel sich in seinen Ausführungen jedoch auf die einfache Automatenstruktur beschränken.

Das Kapitel 4 beinhaltet die konzeptionelle Beschreibung des iterativen Abstraktionsverfeinerungsverfahrens auf Basis der in Kapitel 3 definierten Hybriden Systeme, wobei der Schwerpunkt hier auf Schritt-diskrete Hybride Systeme gelegt wird. Hier wird die Motivation und Strategie des Verfahrensansatzes diskutiert und dieses zunächst in einfacher Form in seinen wesentlichen Phasen, der initialen Abstraktion, der Pfadanalyse, sowie der Verfeinerung der Abstraktion durch einen lernenden ω-Automaten beschrieben. Nach dem Beweis der Korrektheit des Verfahrens werden Terminierungsbedingungen und die Verwendung von CTL-Model-Checking diskutiert. Ebenfalls auf der konzeptionellen Ebene werden bereits zentrale Erweiterungen des Verfahrens vorgestellt, die die Konvergenz des Verfahrens erheblich beschleunigen können. Das Kapitel wird nach der Übertragung des Verfahrens auf Zeit-kontinuierliche Systeme durch einen Vergleich mit verwandten Arbeiten abgeschlossen.

Das Kapitel 5 führt den Leser in die Welt der im industriellen Kontext erstellten Schritt-diskreten Hybriden Systeme ein und beinhaltet neben der Kurzdarstellung einer Auswahl an CASE-Werkzeugen (*Computer Aided Systems Engineering*), deren Möglichkeiten und typische Verwendung die Vorstellung eines konkreten Fallbeispieles, einem einfachen Autopilotensystem zur Kontrolle der Höhe eines Flugobjekts im dreidimensionalen Raum unter Einwirkung der Störgröße Wind.

War Kapitel 5 in weiterem Sinne das Gegenstück zu Kapitel 3, da gegenüber den formal definierten, konzeptionellen Repräsentationen von Hybriden Systemen nun die in der Praxis existierenden Systeme beschrieben worden sind, so ist Kapitel 6 nun als entsprechendes Gegenstück zu Kapitel 4 anzusehen. Hier wird das konzeptionell in Kapitel 4 bereits vorgestellte Verfahren auf die Repräsentationsebene imperativer Sprachen, hier insbesondere die Sprache SMI, mit Bezug auf die in Kapitel 5 vorgestellten Modelle übertragen. Aufgrund der unterschiedlichen Darstellungen der Modelle sind auch hier die entscheidenden Verfahrensschritte weitestgehend formalisiert, um eine präzise Vorstellung der Umsetzung zu vermitteln. Darüber hinaus beinhaltet dieses Kapitel aus Vollständigkeitsgründen kurze Beschreibungen in nicht unmittelbar zum Verfahren gehörige Transformationsschritte, soweit sie für die praktische Anwendung des Verfahrens elementar sind.

In Kapitel 7 wird die Anwendung des Verfahrens zur Verifikation von Eigenschaften des in Kapitel 5 präsentierten Fallbeispiels vorgestellt, wobei hier auf zwei weitere Beispielmodelle Bezug genommen wird. Die aus den Verifikationsexperimenten gewonnenen statistischen Daten werden durch graphische Verlaufskurven der wichtigsten Eckdaten des iterativen Verfeinerungsprozesses veranschaulicht und diskutiert.

Abschließend enthält Kapitel 8 eine Zusammenfassung und einen Ausblick auf mögliche, zukünftige Verbesserungen und Erweiterungen des Verfahrens.

Kapitel 2

Mathematische Grundlagen und Algorithmen

In diesem Kapitel werden vorwiegend aus [CGP99] entliehene mathematische Grundlagen und Algorithmen vorgestellt, auf die in dieser Arbeit Bezug genommen wird. Im Einzelnen sind dies die Folgenden:

- Automaten für endliche und unendliche Wörter werden für die ω-Automatenkonstruktion benötigt.
- Transitionssysteme und Kripke-Strukturen repräsentieren die Datenstruktur für den Model-Check-Algorithmus.
- Temporale Logik wird sowohl bei der Beschreibung des Model-Check-Algorithmus, als auch für die Konstruktion des ω-Automaten benötigt.
- Der Model-Check-Algorithmus wird Ergebnis-bezogen angewendet, d.h. der Algorithmus selbst ist für die Arbeit nicht relevant, jedoch der Vollständigkeit halber ebenfalls dargestellt.
- Bisimulationsäquivalenz wird für die Verallgemeinerung der betrachteten Systeme benötigt.
- Abstraktion schließlich beschreibt die zentrale Eigenschaft des Verfahrens dieser Arbeit.

Wir werden in diesem und späteren Kapiteln einige wiederholt Verwendung findende Sysmbole im Randbereich aufführen, soweit diese nicht durch eigene Definitionen eingeführt werden. Dies soll das Auffinden selbiger erleichtern und dient somit als Lesehilfe.

2.1 Automaten für endliche und unendliche Wörter

2.1.1 Reguläre Automaten

Ein endlicher Automat A über endliche Wörter ist ein 5-Tupel (Q, Q_0, Σ, T, F), wobei

- Q eine Menge von *Zuständen* ist,
- $Q_0 \subseteq Q$ eine Menge von *Initialzuständen* kennzeichnet,
- Σ ein endliches *Alphabet* ist,

- $T \subseteq Q \times \Sigma \times Q$ eine *Transitionsrelation* bildet und
- $F \subseteq Q$ die Menge der *Finalzustände* darstellt.

Ein Lauf eines solchen Automaten bzgl. eines endliches Wortes $w = (\delta_1, \delta_2, \ldots, \delta_n) \in \Sigma^n \subseteq \Sigma^*$ ist eine Abbildung $\rho_w : \{i \,|\, 0 \leq i \leq n\} \to Q$, so daß ρ_w

- der erste Zustand ein Initialzustand ist, d.h. $\rho_w(0) \in Q_0$, und
- aufeinanderfolgende Zustände der Transitionsrelation genügen, d.h. $\forall_{0 \leq i \leq n} : (\rho_w(i), \delta_{i+1}, \rho_w(i+1)) \in T$.

Ein Wort $w \in \Sigma^n$ wird von dem Automaten *akzeptiert*, wenn $\rho_w(n) \in F$. Der Lauf ρ_w wird in diesem Falle als *akzeptierend* bezeichnet. Für ein Wort $w = (\delta_1, \delta_2, \ldots, \delta_k, \ldots, \delta_n)$ bezeichnen wir ein Teilwort $(\delta_1, \delta_2, \ldots, \delta_k)$ als Präfix und $(\delta_k, \ldots, \delta_n)$ als Suffix von w. Einen Präfix assoziieren wir außerdem mit dem Zustand $\rho_w(k)$, der durch Einlesen des Präfix erreicht wird.

Diese Automaten zur Akzeptanz endlicher Wörter werden als reguläre Automaten bezeichnet und wir verwenden das Symbol $\mathfrak{R}$ zur Bezeichnung der Menge aller regulären Automaten.

Eine Besonderheit ist die mit regulären Automaten realisierbare sog. *1-Akzeptanz*, bei der auch unendliche Wörter aufgrund eines endlichen Präfixes akzeptiert werden können. Die so akzeptierbaren Wörter sind Teil einer *1-akzeptierbaren Sprache*. Hingegen ist das Komplement einer 1-akzeptierbaren Sprache, also alle Wörter bis auf solche, die aufgrund eines endlichen Präfixes auszuschließen sind, nicht durch einen regulären Automaten akzeptierbar. Für eine solche sog. *co-1-Akzeptanz* ist bereits ein Büchi-Automat erforderlich.

2.1.2 ω-Automaten

Im Gegensatz zu endlichen zu akzeptierenden Wörtern sind für reaktive Systeme, da diese nicht terminieren, unendliche Wörter aus Σ^ω zu behandeln. Die einfachsten Automaten zur Akzeptanz unendlicher Wörter sind *Büchi-Automaten*, welche die gleiche Struktur aufweisen wie endliche Automaten, die endliche Wörter akzeptieren. Im Unterschied zu diesen wird F nun als Menge akzeptierender Zustände bezeichnet und ein unendliches Wort $v \in \Sigma^\omega$ genau dann akzeptiert, wenn der dazugehörige unendliche Lauf $\rho_v : \mathbb{N}_0 \to Q$ eine Menge von Zuständen Q_{inf} unendlich oft beinhaltet und $Q_{inf} \cap F \neq \emptyset$. Wir bezeichnen die Menge aller Büchi-Automaten mit $\mathfrak{B}$.

Ist die durch einen Büchi-Automaten zu akzeptierende Sprache co-1-akzeptierbar, so hat der Automat eine einfachere Gestalt, da die nicht akzeptierenden Zustände $\bar{F} = Q \setminus F$ Senken bilden, d.h. es gibt keine Transitionen, die von einem Zustand aus $\bar{F}$ zu einem anderen Zustand aus Q führen.

2.1.3 Determinismus

Wenn der Zustand $\rho(k)$ für jedes Wort $(\delta_1, \delta_2, \ldots, \delta_k, \ldots)$ sowohl bei regulären, als auch bei ω-Automaten immer durch alle bisher gelesenen Buchstaben $\delta_{i, i \leq k}$ eindeutig ist, bezeichnen wir den Automaten als *deterministisch*.

2.1.4 Vollständigkeit

Wir bezeichnen einen Automaten als *vollständig*, wenn gilt $\forall p \in Q \forall \delta \in \Sigma : \exists q \in Q : (p, \delta, q) \in T$.

2.2 Kripke-Strukturen

Reaktive Systeme zeichnen sich aus durch eine fortdauernde Berechnung begleitet von ständigem Interagieren mit der Umgebung durch Berücksichtigung von Eingaben aus der Umgebung und Setzen berechneter Reaktionen des Systems. Für die mathematische Modellierung solcher Systeme wird von den Details der Umgebungsinteraktion abgesehen und das Verhalten auf die eigentlichen internen Berechnungsschritte reduziert. Im einfachsten Fall ist dies ein Transitionssystem, welches zwecks einfacherer Eigenschaftsspezifikation später zu einer Kripke-Struktur durch eine Labeling-Funktion erweitert wird.

2.2.1 Transitionssysteme

Definition 1 *Transitionssystem Ein* Transitionssystem *ist ein Tripel* $TS = (S, S_0, T)$*, wobei* S *eine (unendliche) Menge von Zuständen,* $S_0 \subset S$ *eine Menge von Startzuständen und* $T \subseteq S \times S$ *eine Transitionsrelation darstellen.*

Definition 2 *Berechnungspfad Ein* Berechnungspfad π *eines Transitionssystems TS ist eine (unendliche) Sequenz von Zuständen* $\pi = (s_0, s_1, \dots), s_i \in S, s_0 \in S_0$*, wobei Paare aufeinanderfolgender Zustände in der Relation T sein müssen:* $(s_i, s_{i+1}) \in T$*.*

Ein Zustand $s_k \in S$ ist *erreichbar*, wenn es einen Pfad gibt, der s_k beinhaltet, formell: $\exists \pi = (s_0, s_1, \dots, s_k, \dots)$. Wenn jedoch gilt $\nexists \pi = (s_0, \dots, s_k)$, dann ist der Zustand s_k *unerreichbar*.

2.2.2 Kripke-Strukturen

Für die Spezifikation komplexerer Eigenschaften wird das Transitionssystem mit einer *Labeling-Funktion* $L : S \to 2^{AP}$ erweitert, welche es erlaubt, den einzelnen Zuständen jeweilige atomare Eigenschaften aus einer Menge AP von atomaren Propositionen zuzuordnen.

Definition 3 *Kripke-Struktur Eine* Kripke-Struktur M *ist ein Quadrupel* $M = (S, S_0, T, L)$*, wobei* S*,* S_0 *und* T *wie im Transitionssystem definiert sind und die Labelingfunktion* $L : S \to 2^{AP}$ *jedem Zustand eine Menge von* atomaren Propositionen AP *zuordnet, welche in dem jeweiligen Zustand wahr sind.*

Bedeutung der Labeling-Funktion Die Labeling-Funktion L vereinfacht die Spezifikation von Eigenschaften und, wie wir in Sektion 2.4.1 sehen werden, auch die Formalisierung des CTL-Model-Check-Algorithmus, da mit Hilfe von L zum Einen mit einzelnen atomaren Propositionen $p \in AP$ auf eine Menge von Zuständen $S_p = \{s_1, s_2, \dots, s_n\}, \bigwedge_{i \in \{1,2,\dots,n\}} p \in L(s_i)$ Bezug genommen werden kann und zum Anderen auch bei Transformationen des Systems bzw. Bezugnahme auf andere Systeme mit gleicher atomarer Propositionsmenge die Eigenschaften unverändert wiederverwendet werden können. Daher wird sowohl in der einschlägigen Literatur wie auch in den folgenden Ausführungen dieses Kapitels anstelle von Transitionssystemen unter Verwendung einer solchen Labeling-Funktion auf Kripke-Strukturen zurückgegriffen.

Da die Labeling-Funktion jedoch jederzeit durch eine explizite Bezugnahme auf die jeweiligen Zustandsteilmengen ersetzt werden kann, werden wir in späteren Kapiteln, in denen die Spezifikation von Eigenschaften in den Hintergrund tritt, für eine einfachere Darstellung auf diese verzichten und uns wieder auf Transitionssysteme beziehen.

2.2.2.1 Symbolische Repräsentation

Da Kripke-Strukturen bei industriellen reaktiven System sehr groß werden können, ist eine explizite Darstellung bzgl. des benötigten Speichers und der Effizienz von Analyseverfahren nicht praktikabel. Daher wird auf eine kompakte symbolische Repräsentation zurückgegriffen, welche sowohl Mengen, als auch Relationen effizient darstellen und manipulieren kann.

Endliche Mengen Q mit $|Q| < 2^m$ mit m als kleinstmöglicher natürlicher Zahl, wie sie in den Tupelelementen S, S_0, T und F in der Kripke-Struktur vorhanden sind, lassen sich mit Hilfe von bijektiven Abbildungen $\mathbb{B}^m \to Q$, mit $\mathbb{B} \stackrel{\text{def}}{=} \{false, true\}$ durch entsprechende boolsche Kodierungen in aussagenlogische Ausdrücke transformieren. So können wir durch Festlegung einer Kodierung $\phi_S : \mathbb{B}^{|S|} \to S$ mit Hilfe einer charakteristischen Funktion $f : \mathbb{B}^{|S|} \to \mathbb{B}$ jede beliebige Teilmenge S_f von S darstellen, die durch $S_f = \left\{ s \in S \,\middle|\, \exists b \in \mathbb{B}^{|S|} : f(b) \wedge \phi_S(b) = s \right\}$ definiert ist. Da jede Transition $t \in T$ ein Paar von Zuständen $(s_1, s_2) \in S \times S$ repräsentiert, kann unter Verwendung der Kodierung ϕ_S für Zustände auch jedes Element $t = (s_1, s_2)$ der Transitionsrelation $T \subseteq S \times S$ durch die bijektive Abbildung $\phi_T : \mathbb{B}^{2 \cdot |S|} \to T$ mit $\phi_T((b_0, b_1, \ldots, b_{2 \cdot |S|})) = \left(\phi_S((b_0, b_1, \ldots, b_{|S|})), \phi_S((b_{|S|+1}, b_{|S|+2}, \ldots, b_{2 \cdot |S|}))\right)$ kodiert und die Transitionsrelation T mithin als charakteristische Funktion $f_T : \mathbb{B}^{2 \cdot |S|} \to \mathbb{B}$ in Form eines aussagenlogischen Ausdrucks angegeben werden, so daß $f_T(b) = \left(\phi_T(b) \in T\right)$.

2.2.2.2 OBDDs

Charakteristische Funktionen in Form von aussagenlogischen Ausdrücken ermöglichen bereits eine symbolische Darstellung und Manipulation von Mengen, sind jedoch bzgl. Kompaktheit und Manipulierbarkeit von großen Mengen immer noch problematisch. Beide Eigenschaften sind wesentlich effizienter durch *Ordered Binary Decision Diagrams* (OBDD) realisierbar.

Aussagenlogischen Ausdrücken auf Variablenvektoren $b \in \mathbb{B}^m$ lassen sich in einfache binäre Entscheidungsbäume überführen, indem alle möglichen Variablenbelegungen in Baumform aufgezählt werden.

Dabei besitzt jeder Variablenknoten (*Nichtterminal*) eine Verzweigung in Form einer gerichteten Kante jeweils für eine 0- oder 1-Belegung, die wiederum durch den Belegungsbaum verbleibender Variablen fortgeführt wird. Am Ende eines jeden Zweiges findet sich ein Wertknoten (*Terminal*), welcher der zugrundelegenden Belegung aller Variablen den entsprechenden Wahrheitswert zuordnet.

Diese Darstellungsform führt immer noch zu sehr großen Repräsentationen, jedoch besteht nun die Möglichkeit, die Darstellung in einen OBDD umzuwandeln, welcher in einem nachfolgenden Schritt mit großer Wahrscheinlichkeit in eine erheblich kompaktere Repräsentation überführt werden kann:

1. Festlegung einer totalen Ordnung $<$ der beteiligten Variablen und entsprechender Transformation des Baums, so daß ausgehende Kanten von Nichtterminalsymbolen nur zu Nichtterminalsymbolen führen, welche gemäß der gegebenen Ordnung nicht kleiner sind.

2. Bis Erreichen eines Fixpunkts wiederholt angewandte, folgende Operationen:

   - *Elimination von doppelten Terminalen*. Blätter des Entscheidungsbaumes mit gleichem Wahrheitswert erhalten eine Kante zu einem einzigen Terminalknoten, der diesen Wert repräsentiert.

- *Elimination von doppelten Nichtterminalen*. Sich auf dieselbe Variable beziehende Nichtterminale mit identischem Unterbaum werden zusammengelegt.
- *Elimination von redundanten Überprüfungen*. Sind die Unterbäume der von einem Nichtterminalknoten ausgehenden Kanten identisch, wird der Nichtterminalknoten durch einen seiner Unterbäume ersetzt.

Diese von Bryant [R.E86, Bry92] definierte Vorgehensweise führt zu einer kanonischen Darstellung des zugrunde liegenden boolschen Ausdrucks, so daß OBDDs genau dann isomorph sind, wenn der zugrunde liegende boolsche Ausdruck logisch äquivalent ist.

Die Größe eines OBDDs hängt nun erheblich von der Variablenreihenfolge der totalen Ordnung ab. So gibt es günstige Variablenordungen, in denen sich viele Nichtterminalknoten eliminieren lassen und ungünstigere, bei denen dies nicht möglich ist. Eine Bestimmung der optimalen Variablenreihenfolge ist für große Graphen jedoch aus Komplexitätsgründen nur theoretisch möglich, denn bereits das Bestätigen einer optimalen Reihenfolge ist ein NP-vollständiges Problem. Es existieren jedoch eine Reihe von Heuristiken, welche zumindest eine günstige Reihenfolge ermitteln, so daß in der Praxis dennoch mit kompakten Repräsentationen gearbeitet werden kann.

Neben dem sehr einfachen Fall der Negation eines OBDDs durch Vertauschen der beiden Terminalsymbole gibt es 15 weitere logische Operationen zur Verknüpfung zweier OBDDs, welche in [R.E86, Bry92] beschrieben sind und in polynomialer Zeit berechnet werden können. Damit existieren neben der kompakten Speichersparenden Repräsentation von Kripke-Strukturen durch OBDDs auch effiziente Manipulationsmöglichkeiten, welche beim Model-Checken benötigt werden.

2.3 Temporale Logik

Im Gegensatz zur Software Verifikation terminierender Prozesse, bei der die zeitlichen Berechnungsabläufe eines Programms nicht von Interesse sind, ist bei reaktiven Systemen keine Terminierung gegeben und mithin auch keine endliche Eingabe-Ausgabe-Relation, welche als Grundlage für eine Überprüfung der Richtigkeit der Berechnungen herangezogen werden könnte. Stattdessen ist der qualitativ-zeitliche Berechnungsablauf einer Reaktion des Systems auf Stimuli der Umgebung für die Korrektheit des Systems von Bedeutung, da das Zusammenspiel mit der Umgebung z.B. als festgelegte, minimale oder maximale Reaktionszeit unabdingbar für typische Anforderungen an reaktive Systeme ist.

Zur Spezifikation von Eigenschaften, welche auf solche zeitlich vorgeschriebenen Abfolgen basieren, wird auf *Temporale Logik* zurückgegriffen, wie z.B. CTL*. Dabei wird für Schritt-diskrete Systeme die Zeit nicht explizit modelliert, sondern typischerweise die Schrittzahl des Systems als temporale Metrik herangezogen, welche bezogen auf instantane Berechnungsschritte eine völlig ausreichende Granularität darstellt. Eigenschaften wie „im folgenden Schritt", „n-Schritte später", „schließlich", „bis" oder „niemals" sind somit in Temporaler Logik ausdrückbar, wobei diese zeitlichen Bedingungen in Formeln durch *temporale Operatoren* bzw. kombinierte Ausdrücke selbiger beschrieben werden. Die Operatoren beziehen sich immer auf einen im Zustand s_0 beginnenden Pfad $\pi = (s_0, s_1, \dots), s_i \in S$ und evaluieren zu *true*, wenn die dazugehörige Bedingung erfüllt ist. Die Operatoren sind die Folgenden:

- Der **X**-Operator fordert die Gültigkeit einer Bedingung im nächsten Schritt.

- Der **F**-Operator fordert die Gültigkeit einer Bedingung in einem zukünftigen Schritt.
- Der **G**-Operator fordert die Gültigkeit einer Bedingung in jedem zukünftigen Schritt.
- Der **U**-Operator fordert die Gültigkeit einer ersten Bedingung bis zum Eintreten einer zweiten Bedingung.
- Der **R**-Operator fordert die Gültigkeit einer zweiten Bedingung soweit nicht vorher eine erste Bedingung erfüllt wurde.

Temporale Operatoren angewandt auf eine Bedingung ergeben so genannte *Pfadformeln* (engl.: *path formula*). im Gegensatz zu den Bedingungen, welche auch als *Zustandsformeln* (engl.: *state formula*) bezeichnet werden. Letztere werden aus den Pfadquantoren **A** und **E** gebildet, welche den Pfadformeln vorangestellt werden können, sich somit auf einen Berechnungsbaum mit allen von s_0 ausgehenden Pfaden beziehen und die Gültigkeit der Pfadformel auf allen, bzw. auf mindestens einem Pfad fordern. Innerhalb der beiden genannten Formelklassen stehen desweiteren jeweils die logischen Operatoren $\neg$, $\vee$ und $\wedge$ für Verknüpfungen zur Verfügung.

Die Semantik von CTL* bezogen auf einen Zustand $s \in S$ einer Kripke-Struktur M ist im Folgenden definiert, wobei f, f_1 und f_2 Zustandsformeln, g, g_1 und g_2 Pfadformeln und $\pi^i = (s_i, s_{i+1}, \dots)$ einen Suffixpfad beginnend im i-ten Zustand eines Pfades $\pi = (s_0, s_1, \dots, s_i, s_{i+1}, \dots)$ darstellen:

$$
\begin{aligned}
M, s \models p &\Leftrightarrow p \in L(s) \\
M, s \models \neg f &\Leftrightarrow M, s \not\models f \\
M, s \models f_1 \vee f_2 &\Leftrightarrow (M, s \models f_1) \vee (M, s \models f_2) \\
M, s \models f_1 \wedge f_2 &\Leftrightarrow (M, s \models f_1) \wedge (M, s \models f_2) \\
M, s \models \mathbf{E}g &\Leftrightarrow \exists \pi = (s, s_1, \dots) \in \overrightarrow{M} : M, \pi \models g \\
M, s \models \mathbf{A}g &\Leftrightarrow \forall \pi = (s, s_1, \dots) \in \overrightarrow{M} : M, \pi \models g \\
M, \pi \models f &\Leftrightarrow \pi = (s, s_1, \dots) \wedge M, s \models f \\
M, \pi \models \neg g &\Leftrightarrow M, s \not\models g \\
M, s \models g_1 \vee g_2 &\Leftrightarrow (M, s \models g_1) \vee (M, s \models g_2) \\
M, s \models g_1 \wedge g_2 &\Leftrightarrow (M, s \models g_1) \wedge (M, s \models g_2) \\
M, \pi \models \mathbf{X}g &\Leftrightarrow \pi^1 \models f \\
M, \pi \models \mathbf{F}g &\Leftrightarrow \exists k \geq 0 : M, \pi^k \models g \\
M, \pi \models \mathbf{G}g &\Leftrightarrow \forall i \geq 0 : M, \pi^i \models g \\
M, \pi \models g_1 \mathbf{U} g_2 &\Leftrightarrow \exists k \geq 0 : \left(\forall 0 \leq i < k : M, \pi^i \models g_1\right) \wedge \left(M, \pi^k \models g_2\right) \\
M, \pi \models g_1 \mathbf{R} g_2 &\Leftrightarrow \forall k \geq 0 : \left(\exists 0 \leq i \leq k : M, \pi^i \models g_1\right) \vee \left(M, \pi^k \models g_2\right)
\end{aligned}
$$

Bereits die Operatoren $\neg$, $\vee$, **X**, **U** und **E** sind ausreichend, um alle möglichen Formeln in CTL* auszudrücken:

$$
\begin{aligned}
f \wedge g &\equiv \neg(\neg f \vee \neg g) \\
f \mathbf{R} g &\equiv \neg(\neg f \mathbf{U} \neg g) \\
\mathbf{F} f &\equiv \mathit{true} \mathbf{U} f \\
\mathbf{G} f &\equiv \neg \mathbf{F} \neg f \\
\mathbf{A} f &\equiv \neg \mathbf{E} \neg f
\end{aligned}
$$

Wird CTL* auf Formeln der Form $\mathbf{A}f$ beschränkt, wobei f eine Pfadformel repräsentiert, welche als Zustandsformeln ausschließlich atomare Propositionen verwendet, so ergibt sich die Sprache LTL als Teilsprache von CTL*. In LTL können Ereignisse entlang eines linearen Pfades beschrieben werden, jedoch keine Aussagen über eine Anzahl von Pfaden, welche von einem Zustand ausgehen können. Dies ist wiederum mit der Teilsprache CTL möglich, welche sich durch die eingeschränkte Verwendung von temporalen Operatoren durch geforderte direkte Anwendung eines Pfadquantors auf jede Pfadformel ergibt. CTL und LTL sind in ihrer Ausdrucksmächtigkeit nicht vergleichbar, d.h. es gibt Ausdrücke in LTL wie z.B. $\mathbf{A}(\mathbf{FGp})$, welche nicht in CTL darstellbar sind und umgekehrt, wie der Ausdruck $\mathbf{AG}(\mathbf{EF}p)$ zeigt. Weiterhin ist CTL* ausdrucksmächtiger als LTL∪CTL, was beispielsweise aus der Disjunktion der genannten LTL- und CTL-Formeln $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$ ersichtlich wird. LTL, CTL

Die Sprache CTL kann in die disjunkten Teilsprachen ACTL und ECTL unterteilt werden. Diese sind dadurch ausgezeichnet, daß Negationen ausschließlich in Zustandsformeln auftreten können und bei ACTL nur Allquantoren, bei ECTL nur Existenzquantoren Verwendung finden. ACTL, ECTL

Fairness Häufig ist man nur an Eigenschaften interessiert, welche auf so genannten fairen Berechnungspfaden liegen. Dies sind Pfade, auf denen bestimmte Ereignisse immer wiederkehren, wie z.B. Abläufe in Kommunikationsprotokollen, bei denen auf eine Kommunikationsanforderung auch tatsächlich entsprechende Kommunikationsantworten folgen und nicht unendlich lange gewartet wird. Während eine entsprechende Beschränkung auf solche Pfade in CTL* ausgedrückt werden kann, ist dies in CTL nicht möglich. Letzteres kann jedoch mit einer *fairen Semantik* erweitert werden, in der gefordert wird, daß nur solche Pfade in Betracht gezogen werden, auf denen jeweils mindestens ein Zustand aus jeder Zustandsmenge, genannt Fairness Constraint, aus der Menge $F \subseteq 2^S$ der Fairness Constraints unendlich oft enthalten ist.

Damit erweitert sich die Kripke-Struktur um die Menge der Fairness Constraints F zu einer fairen Kripke-Struktur $M = (S, S_0, T, L, F)$, wobei S, S_0, T und L wie zuvor beschrieben definiert sind, während $F \subseteq 2^S$ die Menge der Fairness Constraints bestehend aus Teilmengen der Zustände darstellt. Wir schreiben $M, s \models_F f$ wenn die Eigenschaft auch unter der Fairness-Erweiterung Gültigkeit hat.

2.3.1 Sicherheitseigenschaften

Eine häufig benötigte einfache Form von Eigenschaften, welche auf reaktiven Systemen häufig verifiziert wird, sind die so genannten Sicherheitseigenschaften. Dabei wird eine Menge von „schlechten" oder „unsicheren" Zuständen definiert, welche unter Einsatzbedingungen nie erreicht werden dürfen. Bezeichnet $S_U \subseteq S$ die Menge solcher Zustände und sind genau diese mit der atomaren Proposition $unsafe$ ausgezeichnet, d.h. $\forall s \in S_U : unsafe \in L(s)$, so lautet die temporallogische Formel dieser Eigenschaft $\mathbf{AG}\neg unsafe$, welche sowohl in CTL als auch LTL liegt.

2.4 Model-Checking-Algorithmus

Ein Modell M in Form einer Kripke-Struktur $M = (S, S_0, T, L)$ erfüllt genau dann eine in temporallogischer Form vorliegende Eigenschaft ϕ, wenn alle relevanten Zustände die Eigenschaft erfüllen. Relevante Zustände sind insbesondere alle möglichen Initialzustände, woraus sich die Eigenschaft für alle hieraus erreichbaren Zustände gleichermaßen ableitet. Der Model-Check-Algorithmus muß also die Menge

$S_\phi = \left\{ s \in S | M, s \models \phi \right\}$ bestimmen. Das Modell M erfüllt die Eigenschaft ϕ, wenn $S_0 \subseteq S_\phi$.

Im folgenden wird der Algorithmus für CTL-Formeln vorgestellt.

2.4.1 CTL

Der Algorithmus bearbeitet die gegebene CTL-Formel ϕ von innen nach außen und berechnet für jede Teilformel die Zustände, welche diese erfüllen. Verwaltet wird dies durch eine Kennzeichnungsfunktion $label : S \rightarrow 2^{CTL}$, welche für jeden Zustand die Teilformeln beinhaltet, die in dem betreffenden Zustand bereits nachgewiesen wurden. Der Algorithmus beginnt mit $label(s) = L(s)$ und terminiert nach vollständiger Bearbeitung von ϕ in $|\phi|$ Iterationen, wonach für jeden Zustand gilt $M, s \models \phi \Leftrightarrow \phi \in label(s)$.

Da jede CTL-Formel durch die Operatoren $\neg$, $\vee$, **EX**, **EU** und **EG** ausgedrückt werden kann, ist eine Betrachtung dieser ausreichend. Der Algorithmus verfährt für diese Operatoren wie folgt:

Operator $\neg$ Für Teilformeln der Form $\neg\phi$ wird die Menge der Zustände, die ϕ_i nicht erfüllen, gekennzeichnet:

Forall $s \in S, \phi \notin label(s)$
  $label(s) := label(s) \cup \{\neg\phi\}$
end

Operator $\vee$ Für Teilformeln der Form $\phi_1 \vee \phi_2$ wird die Menge der Zustände, die ϕ_1 oder ϕ_2 erfüllen, gekennzeichnet:

Forall $s \in S, (\phi_1 \in label(s) \vee \phi_2 \in label(s))$
  $label(s) := label(s) \cup \{\phi_1 \vee \phi_2\}$
end

Operator EX Für Teilformeln der Form $\mathbf{EX}\phi$ wird die Menge der Zustände, die einen ϕ erfüllenden Nachfolgezustand besitzen, gekennzeichnet:

Forall $s \in S, \exists(s, s') \in T \wedge \phi \in label(s')$
  $label(s) := label(s) \cup \{\mathbf{EX}\phi\}$
end

Operator EU Für Teilformeln der Form $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ wird zunächst die Menge der Zustände gekennzeichnet, die ϕ_2 erfüllen. Anschließend werden alle unmittelbaren Vorgängerzustände, in denen ϕ_1 gilt, ebenfalls entsprechend gekennzeichnet:

$S' := \left\{ s \in S \,\middle|\, \phi_2 \in label(s) \right\}$
Forall $s \in S'$
  $label(s) := label(s) \cup \left\{ \mathbf{E}[\phi_1 \mathbf{U} \phi_2] \right\}$
end
While $S' \neq \emptyset$
  choose $s \in S'$
  $S' := S' \backslash \{s\}$
  Forall $s' \in S, (s', s) \in T$
    if $\mathbf{E}[\phi_1 \mathbf{U} \phi_2] \in label(s) \wedge \phi_1 \in label(s')$
      $label(s') := label(s') \cup \left\{ \mathbf{E}[\phi_1 \mathbf{U} \phi_2] \right\}$
      $S' := S' \cup \{s'\}$

```
        end
    end
end
```

Operator EG Für Teilformeln der Form $\mathbf{EG}\phi$ wird die Menge der nicht-trivialen starken Zusammenhangskomponenten (SZKn) benötigt, wie sie mit dem in [HU90] beschriebenen Algorithmus von Tarjan berechnet werden können. Eine starke Zusammenhangskomponente eines gerichteten Graphen ist ein maximaler Teilgraph, in dem jede Kante von jeder anderen Kante aus (indirekt) erreicht werden kann. Besteht der Teilgraph nur aus einem Knoten ohne Kante auf sich selbst, wird die starke Zusammenhangskomponente als trivial bezeichnet.

Für die Berechnung von $\mathbf{EG}\phi$ wird von den starken Zusammenhangskomponenten ausgegangen, in denen ϕ gilt, und der Transitionsrelation rückwärts folgend alle Zustände gekennzeichnet, in denen ϕ ebenso gilt.

$S' := \left\{ s \in S \,\middle|\, \phi \in \mathit{label}(s) \right\}$
$SCC := \{C \,|\, C \text{ ist eine nicht-triviale SZKn von } S'\}$
$Q := \bigcup_{C \in SCC} \{s \,|\, s \in C\}$
Forall $s \in Q$
    $\mathit{label}(s) := \mathit{label}(s) \cup \{\mathbf{EG}\phi\}$
end
While $Q \neq \emptyset$
    choose $s \in Q$
    $Q := Q \backslash \{s\}$
    Forall $s' \in S', (s', s) \in T$
        if $\mathbf{EG}\phi \in \mathit{label}(s)$
            $\mathit{label}(s') := \mathit{label}(s') \cup \{\mathbf{EG}\phi\}$
            $Q := Q \cup \{s'\}$
        end
    end
end

Der gesamte CTL-Model-Check-Algorithmus besitzt einen Aufwand von $O\left(|\phi| \cdot (|S| + |T|)\right)$.

2.4.1.1 Fairness Constraints

Zur Berücksichtigung von Fairness Constraints führen wir den Begriff der fairen starken Zusammenhangskomponenten ein. Eine starke Zusammenhangskomponente C ist fair, wenn wenigstens ein Zustand jedes Fairness Constraints in dieser vorhanden ist, formell: $\forall F_i \in F : \exists s \in F_i : s \in C$. Ersetzen wir nun die Menge SCC in der Berechnung von $\mathbf{EG}\phi$ durch die Menge der fairen starken Zusammenhangskomponenten und legen wir die faire Semantik zugrunde, so wird der Operator **EG** bereits korrekt bzgl. der Fairness Constraints behandelt und der Wahrheitswert von $M, s \models_F \phi$ kann berechnet werden. Mit dessen Hilfe können wir nun eine atomare Proposition fair einführen und den Zuständen zuordnen, wenn ein fairer Pfad von diesem Zustand ausgehend existiert. Es werden genau die Zustände s mit fair gekennzeichnet, für die $M, s \models_F \mathbf{EG}\mathit{true}$ gemäß der fairen Semantik gilt. Die übrigen Operatoren können dann wie folgt behandelt werden ($p \in AP, \phi, \phi_1, \phi_2 \in$ CTL):

$$
\begin{aligned}
M, s \models_F p &\quad\Leftrightarrow\quad M, s \models p \wedge \mathit{fair} \\
M, s \models_F \mathbf{EX}\phi &\quad\Leftrightarrow\quad M, s \models \mathbf{EX}\phi \wedge \mathit{fair} \\
M, s \models_F \mathbf{E}[\phi_1 \mathbf{U} \phi_2] &\quad\Leftrightarrow\quad M, s \models \mathbf{E}\left[\phi_1 \mathbf{U} (\phi_2 \wedge \mathit{fair})\right]
\end{aligned}
$$

Die Komplexität des Algorithmus erweitert sich um die Komplexität $O(|F|)$ der Bestimmung fairer starker Zusammenhangskomponenten und wird damit insgesamt zu $O\left(|\phi| \cdot (|S| + |T|) \cdot |F|\right)$.

2.4.2 Symbolisches Model-Checken

Wie bereits in Kapitel 2.2.2.1 festgestellt wurde, sprengt das explizite Kennzeichnen eines Kripke-Struktur-Graphen bei größeren Systemen schnell die vorhandene Speicherkapazität und ist aufgrund der vielen benötigten Speicherzugriffe nicht effizient.

Daher wird beim symbolischen Model-Checken auf eine OBDD-Repräsentation sowohl der Kripke-Struktur, als auch aller in den Zwischenschritten benötigten Zustandsmengen Bezug genommen. Die sukzessive Berechnung neuer Zustandsmengen stellt sich dann als prädikatenlogische Transformation auf den aussagenlogischen Ausdrücken dar, welche die jeweiligen Zustandsmengen charakterisieren.

Der Operator EX Gemäß der Definitionen in Kapitel 2.3 für $M, s \models \mathbf{E}g$, $M, \pi \models \mathbf{X}g$, $M, \pi \models f$ und $M, s \models f$ und davon ausgehend, daß die Teilformel f bereits berechnet und das Ergebnis als Menge $L_f = \{s \in S | M, s \models f\}$ in aussagenlogisch-kodierter Form vorliegt, gilt es nun, für $\mathbf{EX}f$ die Zustandsmenge $\left\{s \in S \middle| \exists s' : (s, s') \in T \wedge s' \in L_f\right\}$ zu berechnen. Dies kann durch eines der üblichen Kalküle auf monadischer Prädikatenlogik zur Berechnung der Menge $L_{\mathbf{EX}f} = \left\{b \in \mathbb{B}^{|S|} \middle| \exists b' \in \mathbb{B}^{|S|} : f_T(bb') \wedge l_f(b')\right\}$ geschehen, wobei $l_f : \mathbb{B}^{|S|} \to \mathbb{B}$ die charakteristische Funktion für die Zustände ist, welche die Teilformel f erfüllen, $l_f(b) = \left(\phi_S(b) \in L_f\right)$. Alternativ bieten sich für diese Berechnung auch effizientere Verfahren[1] an, welche unter Zuhilfenahme von Ergebnis-Caches für die meisten Fälle schneller terminieren, trotzdem die Aufwandsklasse exponentiell bleibt. Da die charakteristischen Funktionen f_T und l_f bereits als OBDDs zur Verfügung stehen, ist auch das Ergebnis der Mengenberechnung von $L_{\mathbf{EX}f}$ wieder ein neuer OBDD in Form einer charakteristischen Funktion $l_{\mathbf{EX}f}$.

Der Fixpunkt-Algorithmus zur Berechnung komplexerer CTL-Operatoren Der CTL-Model-Check-Algorithmus auf OBDD-Basis ist ein iteratives Fixpunktverfahren, bei dem sich die im Rahmen der CTL-Operatoren sukzessive mit Teilformeln zu kennzeichnende Zustandsmenge als Ergebnis einer Funktion $\tau : 2^S \to 2^S$ergibt und der Fixpunkt erreicht ist, wenn $\tau(S') = S'$. Die nötigen Prädikat-Transformationen und Startmenge der jeweiligen CTL-Operatoren sind in Tabelle 2.1 zusammengefaßt. Da sowohl die jeweiligen CTL-Teilformeln f, f_1 und f_2, als auch die Zustandsmenge Q aus dem vorherigen Iterationsschritt als OBDDs vorliegen, ergibt auch die Anwendung von τ eine neue Zustandsmenge, die als OBDD repräsentiert ist.

Der Algorithmus zur Berechnung des Fixpunkts ist der folgende:

$Q = Q_0$
$Q' = \tau(Q)$
while $Q \neq Q'$
    $Q = Q'$
    $Q' = \tau(Q)$
end

[1] Siehe [CGP99], Kapitel 6.6

CTL-Operator	Prädikat-Transformer $\tau : Q \to Q$	Startmenge Q_0
$\mathbf{AF}f$	$\tau(Q) = f \vee \mathbf{AX}Q$	$\emptyset$
$\mathbf{EF}f$	$\tau(Q) = f \vee \mathbf{EX}Q$	$\emptyset$
$\mathbf{AG}f$	$\tau(Q) = f \wedge \mathbf{AX}Q$	S
$\mathbf{EG}f$	$\tau(Q) = f \wedge \mathbf{EX}Q$	S
$\mathbf{A}[f_1 \mathbf{U} f_2]$	$\tau(Q) = f_2 \vee (f_1 \wedge \mathbf{AX}Q)$	$\emptyset$
$\mathbf{E}[f_1 \mathbf{U} f_2]$	$\tau(Q) = f_2 \vee (f_1 \wedge \mathbf{EX}Q)$	$\emptyset$
$\mathbf{A}[f_1 \mathbf{R} f_2]$	$\tau(Q) = f_2 \wedge (f_1 \vee \mathbf{AX}Q)$	S
$\mathbf{E}[f_1 \mathbf{R} f_2]$	$\tau(Q) = f_2 \wedge (f_1 \vee \mathbf{EX}Q)$	S

Tabelle 2.1: Prädikat-Transformatoren für CTL-Formeln

2.4.2.1 Fairness

Beim symbolischen Model-Checken ist die Angabe von Fairness Constraints als Zustandsmengen für den Fixpunktalgorithmus sehr unhandlich. Daher werden die Menge der Fairness Constraints F ebenfalls durch CTL-Formeln P_k dargestellt, $F = \{P_1, P_2, \ldots, P_n\}$, wodurch sich wieder ein Fixpunktverfahren zur Berechnung der fairen Zustände anbietet, wobei $M, s \models_F \mathbf{EG}f$ mit Hilfe des Fixpunktverfahrens durch $\tau(Q) = f \wedge \bigwedge_{k=1}^{n} \mathbf{EXE}\left[f \mathbf{U} (Q \wedge P_k)\right]$ mit der Startmenge $Q_0 = S$ berechnet werden kann. Mit den übrigen Operatoren kann dann wie in Kapitel 2.4.1.1 beschrieben verfahren werden.

2.4.3 Berechnung von Erreichbarkeitspfaden

Bislang haben wir nur den Wahrheitswert einer CTL-Formel unter Fairness-Bedingungen ermittelt. Eine besondere Stärke des CTL-Model-Checkens ist jedoch außerdem die Berechnung von Gegenbeispielen bei Formeln mit dem universellen Pfadquantifizierer **A**, wie z.B. $\mathbf{AG}f$, bzw. die Berechnung von sog. Zeugen bei Formeln, die mit dem existentiellen Pfadquantifizierer **E**, wie z.B. $\mathbf{EG}f$. Im ersten Fall ist dies ein Gegenbeispiel-Pfad, der die Formel widerlegt, und im zweiten Fall ein die Formel bestätigender Pfad. Da universell und existentiell quantifizierte CTL-Formeln zueinander dual sind, beschränken wir uns im Folgenden auf existentiell quantifizierte CTL-Formeln und betrachten nur die Operatoren **EG**, **EX** und **EU**.

2.4.3.1 Operator EG

Zur Berechnung eines Zeugenpfades bzgl. des Operators **EG** betrachten wir wieder den Graphen der starken Zusammenhangskomponenten der Kripke-Struktur. Ein unendlicher Pfad muß eine endliche Menge dieser Komponenten besuchen, bis er in eine Komponente mündet, in welcher er einen unendlichen Suffix besitzt. Der Algorithmus berechnet somit die Pfade durch die jeweiligen Komponenten, konkateniert die so ermittelten Präfixe in der entsprechenden Reihenfolge und schließt daran den unendlichen Suffix der letzten Komponente an.

Vorbereitung Unter Einbeziehung der Fairness-Bedingungen betrachten wir wieder den Prädikatentransformer $\tau(Q) = f \wedge \bigwedge_{k=1}^{n} \mathbf{EXE}\left[f \mathbf{U}(Q \wedge P_k)\right]$ aus Kapitel 2.4.2.1 zur Berechnung der Zustände, die die Bedingung $\mathbf{EG}f$ erfüllen. Während der Berechnung des äußeren Fixpunkts wird für jede Fairness-Bedingung $P \in F$ die Sequenz der Zustandsmengen $Q_0^P \subseteq Q_1^P \subseteq Q_2^P \subseteq \ldots$, die bei der Berechnung der Teilformel $\mathbf{E}\left[f \mathbf{U}(Q \wedge P)\right]$ anfällt, gespeichert.

Suche eines kürzesten, alle Fairness-Bedingungen einmalig erfüllenden Pfads Diese Zustandsmengensequenzen dienen nun indirekt als Gradient in der Suche des kürzesten Pfades von einem Anfangszustand s zu einem Zustand t, für den $t \models \mathbf{E}[f\mathbf{U}(\mathbf{EG}f \wedge P)]$ gilt, indem gezielt in den Mengen $\bigcup_{P\in F} Q_0^P$, $\bigcup_{P\in F} Q_1^P$, $\bigcup_{P\in F} Q_2^P$ usw. in dieser Reihenfolge gesucht wird, bis ein solcher Zustand t gefunden ist, der in der Transitionsrelation T als Nachfolger von s eingetragen ist. Der Zustand t liegt genau auf dem kürzesten Pfad zu $(\mathbf{EG}f) \wedge P$. Haben wir t in $\bigcup_{P\in F} Q_i^P$ gefunden, so sind es noch i Transitionsschritte bis zu einem Zustand u, für den $u \models (\mathbf{EG}f) \wedge P$ gilt. Für diese i Transitionsschritte werden trivialerweise genau die Zustände ausgewählt, die auf einem solchen kürzesten Pfad liegen, d.h. $(t_1, t_2, \ldots, t_i)$ mit $(t, t_1), (t_{k-1}, t_k) \in T \wedge t_k \in \bigcup_{P\in F} Q_{i-k}^P$. Vorliegend ist nun ein Teilpfad, der bereits die Fairness-Bedingung P erfüllt. Die beschriebene Vorgehensweise wird nun für alle anderen Fairness-Bedingungen wiederholt, so daß die zusammengesetzten Teilpfade bereits allen Fairness-Bedingungen genügen. Den letzten hierbei erhaltenen Zustand bezeichnen wir mit s'.

Suche nach einem fehlenden Endlospfadteilstück Befinden sich t und s' bereits in derselben starken Zusammenhangskomponente und gilt $\{s'\} \wedge \mathbf{EXE}[f\mathbf{U}\{t\}]$, so wäre der dazugehörige Zeugenpfad bereits das fehlende Stück eines sich unendlich oft wiederholenden Teilpfades und die gesamte Zeugenpfadberechnung mithin abgeschlossen.

Andernfalls muß der bereits gefundene Teilpfad von s' ausgehend weiter verlängert werden, bis eine entsprechende starke Zusammenhangskomponente gefunden wird, in welcher die Eigenschaft $\{s'\} \wedge \mathbf{EXE}[f\mathbf{U}\{t\}]$ erfüllt wäre. Da $s' \models \mathbf{EG}f$ muß es eine solche geben und durch erneute Suche nach einem kürzesten, alle Fairness-Bedingungen einmalig erfüllenden Teilpfads von s' aus müssen wir dieser zwingend näher kommen, da noch kein Schließen eines Endlospfads möglich war und somit ausgeschlossen ist, daß bereits besuchte Zustände erneut einbezogen werden.

2.4.3.2 Operatoren EU und EX

Für **EX** ist die Berechnung des Pfades trivial, während für den Operator **EU** der gesuchte Teilpfad zur Erreichung der Until-Bedingung einfach mit Hilfe der Sequenz der Zustandsmengen $Q_0 \subseteq Q_1 \subseteq Q_2 \subseteq \ldots$ der dazugehörigen Fixpunktberechnung gefunden werden kann. Hierbei wird iterativ ein jeweiliger Nachfolgezustand aus der Transitionsrelation gesucht, der Element der jeweilig engeren Zustandsmenge Q_{i-1} ist, während der vorherige Element aus Q_i, nicht jedoch Q_{i-1} ist.

Um den Pfad abschließend durch einen unendlichen, fairen Suffix-Pfad zu ergänzen kann wiederum das Verfahren zur Berechnung eines Pfades für die Formel **EG***true* verwendet werden, welches bereits die Einhaltung der Fairness-Bedingung garantiert.

2.5 Bisimulationsäquivalenz

Für viele Transformationen oder Vereinfachungen von Kripke-Strukturen ist ein Äquivalenzbegriff hilfreich, in dem zwei Strukturen M und M' bzgl. einer Temporalen Logik T_L genau dann äquivalent sind, wenn sie die gleiche Menge von Formeln $Spec \subseteq T_L$ erfüllen. Bezüglich CTL* wird dies durch eine Bisimulationsrelation B erreicht:

Definition 4 (Bisimulationsrelation) *Eine Relation $B \subseteq S \times S'$ bzgl. zwei Strukturen $M = (S, S_0, T, L)$ und $M' = \left(S', S'_0, T', L'\right)$ heißt* Bisimulationsrelation *genau dann, wenn für alle $s \in S$ und $s' \in S'$ mit $(s, s') \in B$ die folgenden Bedingungen gelten:*

1. *Gleichheit der atomaren Propositionen: $L(s) = L'(s')$*
2. *Korrespondenz von Transitionsquell- und Zielzuständen von M nach M': $\forall s_1 \in S, (s, s_1) \in T : \exists s'_1 \in S', (s', s'_1) \in T', (s_1, s'_1) \in B$*
3. *Korrespondenz von Transitionsquell- und Zielzuständen von M' nach M: $\forall s'_1 \in S', (s', s'_1) \in T' : \exists s_1 \in S, (s, s_1) \in T, (s_1, s'_1) \in B$*

Definition 5 (Bisimulationsäquivalenz) *Zwei Strukturen $M = (S, S_0, T, L)$ und $M' = \left(S', S'_0, T', L'\right)$ heißen* bisimulationsäquivalent *(geschrieben $\equiv$), wenn es eine Bisimulationsrelation B gibt, so daß*

1. $\forall s_0 \in S_0 : \exists s'_0 \in S'_0, (s_0, s'_0) \in B$
2. $\forall s'_0 \in S'_0 : \exists s_0 \in S_0, (s_0, s'_0) \in B$

Wie in [CGP99] gezeigt gilt für bisimulationsäquivalente Strukturen und beliebigen Formeln f aus CTL*:

$$(M \equiv M') \implies (M \models f \Leftrightarrow M' \models f) \tag{2.1}$$

Bisimulationsäquivalenz kann bereits dazu genutzt werden, eine Struktur M auf eine bisimulationsäquivalente Struktur M' abzubilden, die möglicherweise kleiner und damit für die Verifikation einfacher zu behandeln ist. In dieser Arbeit hingegen wird Bisimulationsäquivalenz in Kapitel 3.3 verwendet, um die Struktur der Schritt-diskreten Hybriden Automaten zu erweitern. Für eine Reduktion der Komplexität von Strukturen, bzw. Transitionssystemen ist die im folgenden Abschnitt beschriebene schwächere Abstraktion die geeignetere Technik.

2.6 Abstraktion

Zur Reduzierung des Zustandsraumes eines zu verifizierenden Systems wird häufig auf Abstraktionstechniken zurückgegriffen. Die Verkleinerung des Zustandsraumes reduziert bei größeren Systemen erheblich die Verifikationskomplexität bzw. macht eine Verifikation von Systemen mit unendlich großen Zustandsräumen erst möglich.

Bei Durchführung von Abstraktionen verlieren wir Informationen über das exakte Verhalten des betrachteten Systems. Dadurch kann es Eigenschaften geben, deren Gültigkeit für das betrachtete System nicht mehr feststellbar ist. Wichtig ist jedoch, daß die Verifikation eine Eigenschaft nicht fälschlich als gültig ermittelt, ohne daß dies validiert werden kann. So muß garantiert sein, daß eine Sicherheitseigenschaft, die auf einem *abstrakten* System nachgewiesen werden kann, auch auf dem abstrahierten System Gültigkeit hat. Ein Verifikationsverfahren dieser Eigenschaft wird auch als *konservativ* bezeichnet.

Bei der Abstraktion werden einer oder mehrere Zustände des zu abstrahierenden Systems *TS*, auch als *konkretes* System bezeichnet, in Relation zu einem oder mehreren Zuständen des abstrakten Systems A unter Beibehaltung mindestens aller ein- und ausgehenden Transitionen mit entsprechender Zuordnung gesetzt, so daß jede in *TS* mögliche Transitionsfolge in dem Zustandsraum von A nachvollzogen werden kann. Umgekehrt gibt es typischerweise, und bei größerer Reduktion

des Systems geradezu unvermeidbar, Transitionsfolgen in dem abstrakten System A, welche nicht im konkreten System TS möglich sind.

Bei Abstraktionen hat somit jede Transition in TS eine korrespondierende Transition in A unter einer Relation α:

$$\begin{array}{ccc} s_1 & \sim_\alpha & \hat{s}_1 \\ \downarrow & \Longrightarrow & \downarrow \\ s_2 & \sim_\alpha & \hat{s}_2 \end{array}$$

Formal wird eine Abstraktion durch eine an [CGL94] angelehnte Abstraktionsrelation, auch *Simulationsrelation* genannt, beschrieben:

Definition 6 (Abstraktion) *Ein Transitionssystem* $A = \left(\hat{S}, \hat{S}_0, \hat{E}\right)$ *ist eine* Abstraktion *eines Transitionssystems* $TS = (S, S_0, E)$*, geschrieben* $A \geq TS$*, genau dann, wenn es eine Relation* $\alpha \subseteq S \times \hat{S}$ *gibt, so daß*

- $\hat{S}_0 = \{\hat{s}_0 \,|\, \exists s_0 \in S_0 : (s_0, \hat{s}_0) \in \alpha\}$ *und*
- $\hat{E} = \{(\hat{s}_1, \hat{s}_2) \,|\, \exists s_1, s_2 \in S : (s_1, s_2) \in E \wedge \{(s_1, \hat{s}_1), (s_2, \hat{s}_2)\} \subseteq \alpha\}$

Wie oben beschrieben gilt nun trivial das folgende Lemma:

Lemma 1 *Für ein Transitionssystem TS und einer Abstraktion A,* $A \geq TS$*, gilt bei korrespondierenden Initialzuständen,* $\forall s_0 \in S_0 : \exists \hat{s}_0 \in \hat{S}_0, (s_0, \hat{s}_0) \in \alpha$*:*

$$\forall \pi = (s_0, s_1, \ldots, s_n) : \pi \in \overrightarrow{TS} \rightarrow \exists \hat{\pi} = (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n) \in \overrightarrow{A}, \forall_{0 \leq i \leq n} (s_i, \hat{s}_i) \in \alpha$$

wobei $\overrightarrow{TS}$ *und* $\overrightarrow{A}$ *die Menge aller Berechnungspfade inTS und A darstellen.*

Für eine gegebene Menge von bzgl. ihrer Erreichbarkeit zu untersuchenden Zuständen $S_U \in S$ bedeutet dies insbesondere, daß $A \models \mathbf{AG}\neg\hat{S}_U \implies TS \models \mathbf{AG}\neg S_U, \hat{S}_U = \left\{\hat{s} \in \hat{S} \,|\, \exists s \in S_U : (s, \hat{s}) \in \alpha\right\}$. Hingegen gilt nicht $A \not\models \mathbf{AG}\neg\hat{S}_U \implies TS \not\models \mathbf{AG}\neg S_U$.

Eine durch Definition 6 beschriebene Abstraktion, für die entsprechend Lemma 1 gilt, wird auch als *Überapproximation* bezeichnet, da das Verhalten vom konkreten System vollständig in dem abstrakten System enthalten ist, d.h. alle Läufe des konkreten Systems sind unter Verwendung der Relation α auch im abstrakten System möglich. Demgegenüber gibt es auch *Unterapproximationen*, für die gilt, daß jeder Lauf der Unterapproximation einen korrespondierenden Lauf im konkreten System besitzt, jedoch nicht umgekehrt. Zur Vervollständigung des Bildes seien schließlich auch die *Approximationen* erwähnt, für die keine Zusicherungen bzgl. Entsprechung der Läufe gemacht werden können.

Die vorliegende Arbeit macht von letzteren Beiden keinen Gebrauch, sondern berechnet eine Folge von Überapproximationen zu einem gegebenen konkreten System.

2.6.0.3 Abstraktion und CTL-Model-Checking

Für eine Abstraktion der Definition 6 wurde in [CGL94] nachgewiesen, daß für eine temporallogische Formel φ in ACTL gilt:

$$(A \models \varphi) \implies (TS \models \varphi) \tag{2.2}$$

Genau wie bei Sicherheitseigenschaften gilt somit eine für das abstrakte System A in CTL spezifizierte nachgewiesene Eigenschaft φ ebenfalls für das konkrete

System TS. Wird ein Gegenbeispiel für $A \models \varphi$ ermittelt, so kann dieses in TS jedoch ungültig sein.

Anders verhält es sich mit in ECTL spezifizierten Eigenschaften, da ein Pfad in A nicht notwendigerweise auch in TS existiert und mithin Gleichung (2.2) nicht gilt. Hier ist jedoch der umgekehrte Fall, ein Ergebnis der Nicht-Existenz eines entsprechenden Pfades, auf das konkrete System übertragbar.

Sowohl für nicht gültige ACTL-Formeln, als auch für gültige ECTL-Formeln können mit dem in Abschnitt 2.4.3 beschriebenen Verfahren Gegenbeispiel-, bzw. Zeugenpfade $\hat{\pi}$ berechnet werden, die hinsichtlich der Existenz eines unter der Abstraktionsrelation α korrespondierenden Pfades π in TS (in-)validiert werden können. Somit können diese Ergebnisse bestätigt oder verworfen werden.

Widerlegungsbäume Eine Besonderheit stellen Wiederlegungsbäume dar, wie sie z.B. durch die ACTL-Formel $\mathbf{A}\,(\mathbf{X}\neg a \vee \mathbf{X}\neg b)$ erzeugt werden können. Eine Widerlegung kann sich in diesem Falle nicht auf einen einzelnen Pfad beschränken, da die Formel gegebenenfalls nur durch Hinzunahme des anderen Pfadzweigs widerlegt werden kann.

Bzgl. der Abstraktion ist ein solcher Baum nur dann im konkreten System gültig, wenn der Baum vollständig konkretisierbar ist, d.h. jeder Teilzweig muß einem konkreten Teilpfad entsprechen, der konsistent zu den gemeinsamen Präfixteilpfad der Verzweigung ist.

Kapitel 3

Hybride Automaten

In diesem Kapitel werden Hybride Automaten und deren Semantik sowohl für Zeit-kontinuierliche, als auch für Zeit-diskrete Hybride Systeme formal vorgestellt. Diese konzeptionelle Repräsentation wird im folgenden Kapitel zur konzeptionellen Beschreibung des iterativen Abstraktionsverfeinerungsverfahren herangezogen werden.

3.1 Zeit-kontinuierliche Hybride Automaten

Formale Verifikation von Hybriden Automaten sind Gegenstand vieler Forschungsansätze, wobei zumeist unterschiedliche Definitionen von Hybriden Systemen Verwendung finden, welche vielfach äquivalent oder doch zumindest von ähnlicher Charakteristik sind. So besteht ein Hybrides System typischerweise aus einer endlichen Anzahl von diskreten Orten, auch als *Zustände* oder *Modi* bezeichnet, und einem endlich-dimensionalen *kontinuierlichen Zustandsraum*. Die Zeit-kontinuierliche Semantik beschreibt die Zustandsübergänge des kontinuierlichen Zustandsraums mit Hilfe von den Modi zugeordneten *Differentialgleichungen*, während diskrete Zustandswechsel bei Erfüllung der entsprechenden Transitionsbedingung, im Folgenden auch *Guard Set* genannt, des kontinuierlichen Zustandsraumes, begleitet von einem unstetigen kontinuierlichen Zustandssprung durchgeführt werden, der im Folgenden als *Jump Set* bezeichnet wird. Die dem neuen Modus zugeordneten Differentialgleichungen bestimmen nun wiederum die kontinuierliche Zustandsentwicklung, bis dieser Modus ebenfalls wieder verlassen wird, usw.

Hier soll zunächst die Definition Hybrider Automaten aus [CFH+03] betrachtet werden, aus der im folgenden Kapitel die hier näher betrachteten Schritt-diskreten hybriden Automaten abgeleitet werden.

Definition 7 (Hybrider Automat) *Syntax eines Hybriden Automaten. Ein Hybrider Automat ist ein Tupel* $H = (Z, z_0, X, inv, X_0, T, g, j, f)$*, wobei*

- Z *eine endliche Menge von* Zuständen*, auch als* Modi *bezeichnet,*
- $z_0 \in Z$ *ein* initialer Zustand,
- $X \subseteq \mathbb{R}^n$ *der* kontinuierliche Zustandsraum *mit* n *als Anzahl der Dimensionen,* n
- $inv : Z \rightarrow 2^X$ *eine Abbildung ist, die einem Zustand* $z \in Z$ *eine* Invariante $inv(z) \subseteq X$ *zuordnet,*

- $X_0 \subseteq X$ *die Menge der* initialen kontinuierlichen Zustände, *welche zusammen mit* z_0 *als* $\{z_0\} \times X_0$ *die Menge der initialen hybriden Zustände von* H *ergeben,*
- $T \subseteq Z \times Z$ *die Menge der* diskreten Transitionen *zwischen den Modi,*
- $g : T \to 2^X$ *jeder Transition* $(z_1, z_2) \in T$ *ein* Guard Set $g((z_1, z_2)) \subseteq X$ *zuordnet,*
- $j : T \times X \to 2^X$ *jeder Transition* $(z_1, z_2) \in T$ *und jedem* $x \in g((z_1, z_2))$ *ein* Jump Set $j((z_1, z_2), x) \subseteq X$ *zuordnet und*
- $f : Z \to \left(X \to \mathbb{R}^n\right)$ *jedem Modus* $z \in Z$ *ein* kontinuierliches Vektorfeld $f(z)$ *zuordnet, welches die kontinuierliche Zustandsraumentwicklung mit einer Differentialgleichung* $\dot{\chi}(t) = f(z)(\chi(t))$ *beschreibt. Dabei soll davon ausgegangen werden, daß jede Differentialgleichung eine eindeutige Lösung für den Initialwert besitzt:* $\chi(0) \in inv(z)$.

Die Semantik eines Hybriden Automaten wird mit Hilfe eines Transitionssystems mit unendlich vielen Zuständen translativ definiert:

Definition 8 (Semantik) *Die Semantik eines Hybriden Automaten* H *entspricht einem Transitionssystem* $TS_H = (S, S_0, E)$ *mit:*

- *der Menge aller hybriden Zustände* (z, x) *von* H, $S = \{(z, x) \in Z \times X \,|\, x \in inv(z)\}$
- *der Menge der initialen Hybriden Zustände* $S_0 = \{z_0\} \times X_0$
- *den Transitionen* $(s_1, s_2) \in E$ *mit* $s_1 = (z_1, x_1)$, $s_2 = (z_2, x_2)$ *genau dann, wenn es eine Transition* $(z_1, z_2) \in T$ *gibt mit einer Trajektorie* $\chi : [0, \tau] \to X$ *für ein* $\tau \in \mathbb{R}^{>0}$, *so daß*
  - $\chi(0) = x_1, \chi(\tau) \in g((z_1, z_2))$,
  - $x_2 \in j((z_1, z_2), \chi(\tau))$,
  - $\dot{\chi}(t) = f(z_1)(\chi(t))$ *für* $t \in [0, \tau]$,
  - $\chi(t) \in inv(z_1)$ *für* $t \in [0, \tau]$,
  - $x_2 \in inv(z_2)$

Aufgrund der Zeit-kontinuierlichen Modellierung können diese Automaten nicht nur das Verhalten von Reaktiven Systemen in Form von z.B. eingebetteten Systemen modellieren, sondern eignen sich auch für eine genaue Modellierung einer rein physikalischen Umgebung, welche natürlicherweise Zeit-kontinuierliche Entwicklungen aufweist.

3.2 Hybride Automaten in Schritt-diskreter Betrachtung

Hybride Automaten sind in ihrer allgemeinsten Form aufgrund ihrer Allgemeingültigkeit für die Wissenschaft am interessantesten, jedoch gibt es für den praktischen Einsatz der hieraus hervorgegangenen Verfahren für die Verifikation zwei Punkte zu berücksichtigen:

1. Heutige Computer und Steuerungssysteme arbeiten ausschließlich Schritt-diskret. Differentialgleichungen werden, wenn nötig, durch explizit modellierte Zeit Schritt-diskret approximiert. Primärer Anwendungsfall des Model-Checkens ist das Beweisen von Aussagen über solche Schritt-diskreten Steuerungssysteme, wie in Abschnitt 1.3 diskutiert, welche jedoch eine andere Charakteristik aufweisen, als typische Zeit-kontinuierliche Modellierungen.

2. Die Leistungsfähigkeit heutiger existierender Ansätze zur Behandlung allgemeingültiger Zeit-kontinuierlicher Hybrider Systeme ist für den praktischen und insbesondere für den industriellen Einsatz nicht ausreichend.

Nach Punkt 1 ist es für den praktischen Einsatz relevant, sich mit einer restriktiveren Teilmenge der allgemeinen Zeit-kontinuierlichen Hybriden Systemen auseinanderzusetzen. Diese industriell relevante Teilmenge von Modellen ist in der dieser Arbeit zugrunde liegenden Implementierung adressiert worden und, wie die Ergebnisse später zeigen werden, kann Punkt 2 für diese relativiert werden.

Schritt-diskrete Systeme mit kontinuierlichen Variablen, wie sie mit den CASE Tools SIMULINK-STATEFLOW, STATEMATE, ASCET oder SCADE erstellt werden, weisen bezogen auf den im vorigen Abschnitt vorgestellten Hybriden System die zentrale semantische Besonderheit auf, daß es keine kontinuierliche Zeit gibt. Mit diesen Werkzeugen modellierte reaktive Systeme berechnen schrittweise auf Basis des internen Zustands und der aktuellen Eingabewerte aus der Umgebung die dazugehörigen Ausgabewerte, sowie den neuen internen Zustand. Das System kann nur diskret reagieren und keine Zeit-kontinuierlichen Ausgabewerte zur Verfügung stellen, genauso wenig wie interne Zustandsübergänge Zeit-kontinuierlich ablaufen können.

Diese existierende semantische Restriktion solcher Modelle bedeutet für den Zeit-kontinuierlichen Automaten:

- Die Invariante, welche ein Verlassen eines Modus erzwingen bzw. das betreten eines neuen Modus verhindern kann, ist trivialerweise $inv(z) = X$ für alle Modi $z \in Z$, d.h. gemäß der Semantik-Definition 8 übt die Invariante keinerlei restriktiven Einfluss auf das Transitionsgeschehen mehr aus.

- Die Differentialgleichung zur Beschreibung des kontinuierlichen Vektorfeldes ist trivialerweise $\dot{\chi}(t) = f(z)(\chi(t)) = 0$ für alle Modi $z \in Z$, d.h. ohne diskrete Transitionsübergänge finden keinerlei kontinuierliche Zustandsänderungen statt.

Damit können wir die Syntax und Semantik eines Schritt-diskreten hybriden Automaten vereinfacht darstellen und wie folgt definieren:

Definition 9 (Schritt-diskreter Hybrider Automat) *Syntax eines Schritt-diskreten Hybriden Automaten. Ein Schritt-diskreter Hybrider Automat ist ein Tupel* $H = (Z, z_0, X, X_0, T, g, j)$*, wobei*

- Z *eine endliche Menge von Zuständen,*

- $z_0 \in Z$ *ein initialer Zustand,*

- $X \subseteq \mathbb{R}^n$ *der kontinuierliche Zustandsraum mit* n *als Anzahl der Dimensionen,*

- $X_0 \subseteq X$ *die Menge der initialen kontinuierlichen Zustände, welche zusammen mit* z_0 *als* $\{z_0\} \times X_0$ *die Menge der initialen hybriden Zustände von* H *ergeben,*

- $T \subseteq Z \times Z$ *die Menge der diskreten Transitionen zwischen den Modi,*

- $g : T \to 2^X$ *jeder Transition* $(z_1, z_2) \in T$ *ein Guard-Set* $g((z_1, z_2)) \subseteq X$ *zuordnet und*

- $j : T \to (X \to 2^X)$ *jeder Transition* $(z_1, z_2) \in T$ *und jedem* $x \in g((z_1, z_2))$ *eine Jump-Set-Funktion* $j((z_1, z_2))(x) \subseteq X$ *zuordnet.*

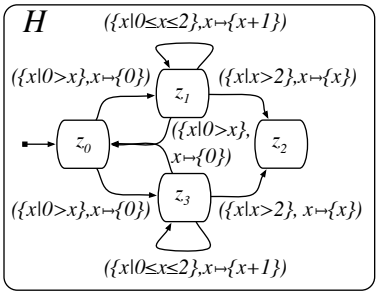


Abbildung 3.1: Beispiel eines einfachen Schritt-diskreten Hybriden Automaten H mit dem kontinuierlichen Zustandsraum $X = \mathbb{R}$

In obiger Definition ist zwecks geeigneterer Beschreibung des Verfahrens in späteren Kapiteln die Definition der Jump-Sets $j : T \times X \to 2^X$ syntaktisch abgeändert worden zu Jump-Set-Funktionen $j : T \to \left(X \to 2^X\right)$.

Die Semantik eines Schritt-diskreten Hybriden Automaten ist im Vergleich zu dem Zeit-kontinuierlichen Automaten signifikant vereinfacht darstellbar:

Definition 10 (Semantik Schritt-diskreter Hybrider Automaten) *Die Semantik eines Schritt-diskreten Hybriden Automaten H entspricht einem Transitionssystem $TS_H = (S, S_0, E)$ mit:*

- *der Menge aller hybrider Zustände (z, x) von H, $S = Z \times X$*
- *der Menge der initialen Hybriden Zustände $S_0 = \{z_0\} \times X_0$*
- *den Transitionen $(s_1, s_2) \in E$ mit $s_1 = (z_1, x_1)$, $s_2 = (z_2, x_2)$ genau dann, wenn $(z_1, z_2) \in T : x_1 \in g((z_1, z_2)) \wedge x_2 \in j((z_1, z_2))(x_1)$*

Einen Pfad $\pi = (s_0, s_1, s_2, \dots)$ von TS_H wird im Folgenden als Pfad von H bezeichnet und TS_H als das *Pfad-Transitionssystem* von H.

TS_H

Bzgl. eines gegeben Modells H bezeichnen wir die Menge aller Guard-Sets mit $G \stackrel{\text{def}}{=} \{g(t) | t \in T\}$ und die Menge aller Jump-Set-Funktionen mit $J \stackrel{\text{def}}{=} \{j(t) | t \in T\}$. Desweiteren definieren wir die Menge $GJ \stackrel{\text{def}}{=} \{(g(t), j(t)) | t \in T\} \subseteq G \times J$ als die Menge aller in H vorkommenden Paare von Guard Sets und Jump Set Funktionen.

G, J, GJ

In Abbildung 3.1 ist ein kleiner Beispiel-Automat graphisch veranschaulicht. Den Transitionen sind Guard Sets γ und Jump Set Funktionen ζ zugeordnet, welche in der Abbildung entsprechend mit (γ, ζ) beschriftet sind. Dies sind die Elemente in GJ. Da das dazugehörige Pfad-Transitionssystem unendlich viele Zustände besitzt, kann dies nicht angemessen dargestellt werden.

Lineare Schritt-diskrete Hybride Automaten Da die dieser Dissertation zugrunde liegende Implementierung sich auf lineare Schritt-diskrete Hybride Systeme beschränkt, führen wir diese hier ebenfalls formal ein.

Definition 11 *Ein Schritt-diskreter Hybrider Automat $H = (Z, z_0, X, X_0, T, g, j)$ heißt* linear, *falls alle Jump Set Funktionen sich durch eine lineare Transformation beschreiben lassen:*

$$\forall t \in T : j(t)(x) = A \cdot x + b$$

mit $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ und $x \in X$.

Damit haben wir nun die Schritt-diskreten linearen Hybriden Automaten in konzeptioneller Repräsentation vorgestellt. Diese werden für die konzeptionelle Beschreibung des Verfahrens im folgenden Kapitel herangezogen werden.

3.3 Erweiterung um multiple Transitionen

Sowohl der Zeit-kontinuierliche, als auch der Schritt-diskrete Hybride Automat kann den gegebenen Definitionen gemäß von einem Zustand z_1 zu einem Zustand z_2 höchstens eine Transition besitzen, welcher genau ein Guard-Set und eine Jump-Set-Funktion zugeordnet sind. Diese Restriktion vereinfacht später die Formalisierung des hier vorgestellten Verfahrens, entspricht jedoch nicht der Semantik der Modelle, die in den Sprachen der bereits genannten CASE-Tools bzw. der Sprache C modelliert werden können. Hier ist es typisch und auch sinnvoll von einem Zustand z_1 zu einem Zustand z_2 eine Menge von Transitionen zu verwenden, welche unterschiedlichen Guard-Sets und/oder unterschiedlichen Jump-Set-Funktionen zugeordnet sind. Wie im Folgenden gezeigt können solche Modelle jedoch syntaktisch bisimulationsäquivalent auf die in Definition 7 und 9 spezifizierten Hybriden Automaten reduziert werden, so daß sich eine Erweiterung selbiger Definitionen zur Behandlung solcher Modelle erübrigt.

Die Syntax dieser Mehr-Transitions-Automaten läßt sich wie folgt definieren:

Definition 12 (Erweiterter Schritt-diskreter Hybrider Automat) *Ein um Transitionsmengen* erweiterter Schritt-diskreter Hybrider Automat *ist definiert als* $H' = \left(Z', z_0', X', X_0', T'\right)$, *wobei*

- Z', z_0', X' *und* X_0' *analog definiert sind wie in Definition 9*
- $T' \subseteq Z' \times Z' \times G' \times J'$ *die Menge der diskreten Transitionen zwischen den Zuständen repräsentiert, wobei* G' *und* J' *Guard-Sets und Jump-Set-Funktionen ebenfalls analog zu Definition 9 darstellen*

Lemma 2 *Ein erweiterter Schritt-diskreter Hybrider Automat* $H' = \left(Z', z_0', X', X_0', T'\right)$ *ist bisimulationsäquivalent zu einem Schritt-diskreten Hybriden Automaten* $H = (Z, z_0, X, X_0, T, g, j)$ *mit*

- $Z = \{z_0\} \cup \bigcup_{(z', z_1', \gamma', \xi) \in T'} \left\{z_{z_1'\gamma'\xi}\right\}$
- $X = X'$
- $X_0 = X_0'$
- $T = \left(\bigcup_{(z_1', z_2', \gamma', \xi) \in T', z_1' = z_0'} \left\{(z_0, z_{z_2'\gamma'\xi})\right\}\right) \cup \left(\bigcup_{(z', z_1', \gamma_1, \xi_1) \in T'} \bigcup_{(z_1', z_2', \gamma_2, \xi_2) \in T'} \left\{(z_{z_1'\gamma_1\xi_1}, z_{z_2'\gamma_2\xi_2})\right\}\right)$
- $g\left((z_{z_1'\gamma_1\xi_1}, z_{z_2'\gamma_2\xi_2})\right) = \gamma_2$
- $j\left((z_{z_1'\gamma_1\xi_1}, z_{z_2'\gamma_2\xi_2})\right) = \xi_2$

unter der Relation $B = \left\{(z_0, z_0')\right\} \cup \bigcup_{(z', z_1', \gamma', \xi) \in T'} \left\{(z_{z_1\gamma'\xi}, z_1')\right\}$.

Beweis: Zu zeigen ist, daß B eine Bisimulationsrelation ist und als solche die Bedingung in Definition 4 erfüllt. Gegeben seien zwei beliebige Zustände $z_{z_1'\gamma_1\xi_1} \in Z \setminus \{z_0\}$ und $z_1' \in Z'$ mit $(z_{z_1'\gamma_1\xi_1}, z_1') \in B$. Der Zustand $z_0 \in Z$ wird aufgrund seiner abweichenden Bezeichnerstruktur und seiner Sonderstellung durch $\forall z \in Z : (z, z_0) \notin T$ von der folgenden Betrachtung ausgeklammert, kann jedoch recht einfach analog behandelt werden. Zunächst wird die Korrespondenz von Transitionsquell- und Zielzustände in beide Richtungen gezeigt:

- Jeder Zustand $z'_2 \in Z'$ mit $(z'_1, z'_2, \gamma_2, \xi_2) \in T'$ ist in H' ein möglicher Zielzustand der Transitionsrelation, welcher nach Definition von B korrespondiert mit dem Zustand $z_{z'_2\gamma_2\xi_2} \in Z$. Es bleibt zu zeigen, daß es eine Transition $(z_{z'_1\gamma_1\xi_1}, z_{z'_2\gamma_2\xi_2}) \in T$ gibt. Nach Definition von T ist genau dies der Fall, wenn es einen Zustand z'_1 gibt, der eine eingehende Transition $(z', z'_1, \gamma_1, \xi_1) \in T'$ besitzt[1] und eine ausgehende Transition $(z'_1, z'_2, \gamma_2, \xi_2) \in T'$. Ersteres ist erfüllt, da der Zustand $z_{z'_1\gamma_1\xi_1}$ nach Definition von Z andernfalls nicht existieren würde. Letzteres wurde bereits verwendet um einen beliebigen Zielzustand z'_2 zu bestimmen und ist somit trivial erfüllt, die Transition $(z_{z'_1\gamma_1\xi_1}, z_{z'_2\gamma_2\xi_2})$ also in T.

- Jeder Zustand $z_{z'_2\gamma_2\xi_2} \in Z$ mit $(z_{z'_1\gamma_1\xi_1}, z_{z'_2\gamma_2\xi_2}) \in T$ ist ein Zielzustand der Transitionsrelation, welcher nach Definition von B mit dem Zustand z'_2 korrespondiert. Es bleibt zu zeigen, daß es eine Transition $(z'_1, z'_2, \gamma_2, \xi_2) \in T'$ gibt. Aufgrund der Definition von T ist dies für zwei Zustände $z_{z'_1\gamma_1\xi_1}$ und $z_{z'_2\gamma_2\xi_2}$ trivial erfüllt.

Da $(z_0, z'_0) \in B$, ist auch die Bedingung in Definition 5 erfüllt und damit gilt $H \equiv H'$. □

Durch die Bisimulationsäquivalenz erweiterter Schritt-diskreter Hybrider Systeme und den einfachen Schritt-diskreten Hybriden Systemen haben wir gezeigt, daß ein für einfache Schritt-diskrete Hybride Systeme beschriebenes Verfahren auch auf erweiterte Systeme mit mehreren Transitionen zwischen zwei Zuständen übertragbar ist. Damit können wir die formale Beschreibung des iterativen Abstraktionsverfeinerungsverfahrens in Kapitel 4 auf die einfachere Form dieser Systeme beschränken, was demgemäß die Formalisierungen deutlich vereinfacht.

[1] Dies ist für den analogen Nachweis für z_0 nicht der Fall, aufgrund der für diesen Sonderfall explizit erweiterten Transitionsrelation T jedoch auch nicht erforderlich.

Kapitel 4

Konzeptionelle Beschreibung des ω-CEGAR Verfahrens

In diesem Kapitel wird das iterative Abstraktionsverfeinerungsverfahren zur Verifikation (Schritt-diskreter) Hybrider Systeme konzeptionell vorgestellt, d.h. alle Algorithmen und Darstellungen orientieren sich strikt an den in Kapitel 3 vorgestellten Definitionen von (Schritt-diskreten) Hybriden Systemen, ohne auf konkrete Repräsentationen von Modellen aus industriellen Kontexten einzugehen. Dieses Vorgehen soll die Allgemeingültigkeit des Verfahrens herausstellen und das Verständnis erleichtern. Die Bezugnahme auf konkrete Repräsentationen und Implementierungen erfolgt in Kapitel 6.

Im folgenden Abschnitt wird neben der Motivation des Einsatzes iterativer Abstraktionsverfeinerungstechniken und deren allgemeiner Funktionsweise die Strategie des hier vorgestellten Verfahrensansatzes mit Hinblick auf charakteristische Eigenschaften kontrolldominierter Systeme diskutiert.

Danach folgt die Vorstellung des neuen Verfahrens in einfacher Form in seinen wesentlichen Phasen, der initialen Abstraktion, der Pfadanalyse, sowie der Verfeinerung der Abstraktion durch einen lernenden ω-Automaten. Nach dem Beweis der Korrektheit des Verfahrens werden Terminierungsbedingungen und die Verwendung von CTL-Model-Checking diskutiert. Außerdem werden zentrale Erweiterungen des Verfahrens vorgestellt, die die Konvergenz des Verfeinerungsprozesses erheblich beschleunigen können.

Das Kapitel schließt nach der Vorstellung der Übertragung des Verfahrens auf Zeit-kontinuierliche Systeme durch einen Vergleich mit verwandten Arbeiten.

4.1 Motivation

Wir betrachten zunächst die allgemeine Funktionsweise iterativer Abstraktionsverfeinerungsverfahren, bevor die Charakteristik kontrolldominierter Schritt-diskreter Hybrider Systeme diskutiert wird. Darauf aufbauend wird dann die Strategie des in diesem Kapitel vorgestellten Abstraktionsverfeinerungsverfahrens für diese Systeme motiviert.

4.1.1 Iterative Abstraktionsverfeinerung

Wie bereits in Abschnitt 2.6 auf Seite 23 beschrieben, werden Abstraktionstechniken eingesetzt, um den Zustandsraum und mithin die Komplexität eines Systems zu reduzieren. Mit einer geeigneten Abstraktion lassen sich somit nicht nur größere Systeme model-checken, sondern selbst solche, die einen unendlichen Zustandsraum

aufweisen. Die Anwendung von Abstraktion ist daher sehr viel versprechend, jedoch bleibt das Problem, eine geeignete Abstraktion zu finden.

Die Abstraktion darf kein zu einfaches abstraktes System beinhalten, da sonst vom Model-Checker Pfade berechnet werden, die sich nicht auf das abstrahierte System übertragen lassen. Wir sprechen in diesem Fall von einer zu *groben* Abstraktion. Ist die Abstraktion nicht einfach genug, ist die angestrebte Komplexitätsreduktion nicht ausreichend, die Abstraktion ist damit zu *fein*. Im Falle von Hybriden Systemen ist es darüber hinaus nicht trivial, überhaupt eine diskrete Abstraktion zu finden, um vom kontinuierlichen Zustandsraum zu abstrahieren, da dieser unendlich groß ist.

Ein häufig angewandtes Verfahren zur Lösung dieses Abstraktionsproblems ist die iterative Abstraktionsverfeinerung. Dabei wird mit einer recht groben initialen Abstraktion A_0 des zu abstrahierenden Transitionssystems TS begonnen, bei der man sicher davon ausgehen kann, daß dieses nicht zu komplex ist. Wird nun vom Model-Checker ein Pfad $\hat{\pi}$ für das abstrakte Transitionssystem A_0 ermittelt, für den es keinen korrespondierenden Pfad π in TS gibt, so ist dieser Pfad als ungültig zu betrachten. Da A_0 offenbar das Ergebnis einer zu groben Abstraktion war, kann jedoch mit einer feineren Abstraktion A_1 der Vorgang wiederholt werden, wobei sinnvollerweise eine systematische Verfeinerung $A_0 \succeq A_1$ durchgeführt werden sollte. Bei erneutem Auftreten eines ungültige Pfades kann als Gegenmaßnahme abermals eine Verfeinerung des Abstraktionsschritts erfolgen. Auf diese Weise entsteht eine Folge von abstrakten Transitionssystemen $A_0, A_1, A_2, \ldots, A_n$, für die $A_0 \succeq A_1 \succeq A_2 \succeq \cdots \succeq A_n \succeq TS$ gilt. Damit nähern wir uns approximierend dem Verhalten von TS solange, bis wir für ein $n \in \mathbb{N}$ schließlich ein ausreichend genaues abstraktes Transitionssystem A_n erhalten, für das hinsichtlich einer Menge $\hat{S}_U$ von zu erreichenden Zuständen im abstrakten System entweder $A_n \models \mathbf{AG}\neg\hat{S}_U$ gilt oder der Model-Checker einen gültigen Pfad $\hat{\pi}$ bzgl. A_n berechnet. In ersterem Fall können wir aufgrund von Lemma 1 auf Seite 24 daraus schließen, daß $TS \models \mathbf{AG}\neg S_U$ mit S_U als der Menge der zu $\hat{S}_U$ in der Abstraktionsrelation befindlichen korrespondierenden Zustände und im Falles eines gültigen Pfades können wir aus diesem sogar einen Pfad π in TS berechnen, der in TS gültig ist und zu einem Zustand s_U aus S_U führt.

Wird die Folge von abstrakten Transitionssystemen $A_0, A_1, A_2, \ldots, A_n$ automatisch berechnet, so sprechen wir von einer automatischen iterativen Abstraktionsverfeinerung. Es gibt im Allgemeinen keine Abstraktionsverfeinerung, die ohne Berücksichtigung weiterer Informationen zielgerichtet eine entsprechende Folge generiert, da die Frage offen ist, welche Teile des Systems in welcher Form zu verfeinern sind. Daher wird hier häufig von der Pfad-gelenkten Abstraktionsverfeinerung (Counter Example Guided Abstraction Refinement, abgekürzt CEGAR) Gebrauch gemacht, wie z.B. in [CFH+03, ADI03]. Die grundsätzliche Idee hierbei ist die Ausnutzung der Information der Widerlegungen vorangegangener Iterationen, um eine gezielte Abstraktionsverfeinerung durchführen zu können. So werden beispielsweise in [CFH+03] Pfade, bzw. Pfadfragmente von $\hat{\pi}$ aus dem abstrakten Transitionssystem gezielt entfernt, um eine möglichst kleinschrittige Abstraktionsverfeinerung zu bewerkstelligen.

Grundsätzlich sind für einen CEGAR Ansatz die folgenden Schritte notwendig:

1. Generierung einer initialen Abstraktion A_0
2. Ermittlung eines Pfades $\hat{\pi}$ bzgl. der Abstraktion A_i durch einen Model-Check-Algorithmus
3. (In-)Validierung des Pfades $\hat{\pi}$.
4. Verfeinerung der Abstraktion A_{i+1}, so daß $\hat{\pi}$ nicht erneut auftreten kann.

Generierung einer initialen Abstraktion A_0 für TS
$A := A_0$
while $A \not\models \mathbf{AG}\neg\hat{S}_U$ **do** % Model-Checker Lauf
    $\hat{\pi} := (\hat{z}_0, \hat{z}_1, \ldots, \hat{z}_n), \hat{z}_n \in \hat{S}_U$ % Pfad vom Model-Checker für A
    if $\hat{\pi}$ ist ungültig **then**
        Generiere A' mit $A \succeq A'$ unter Berücksichtigung von $\hat{\pi}$
        $A := A'$
    else return π für TS konkretisiert aus $\hat{\pi}$ % gültiger Pfad
end
return *true* % $TS_H \models \mathbf{AG}\neg S_U$

Algorithmus 1: CEGAR-Algorithmus . Es wird entweder das Ergebnis *true* oder ein Pfad $\pi = (s_0, \ldots, s_n), s_n \in S_U$ bzgl. TS zurückgegeben.

Je nach Verfahren ist der Aufwand für diese Berechnungsschritte unterschiedlich. Grundsätzlich kann für die Verifikation Hybrider Systeme durch Abstraktionsverfeinerung jedoch gesagt werden, daß der Aufwand für den Model-Check-Algorithmus mit der Größe des Zustandsraums der abstrakten Transitionssysteme und der Pfadlänge wächst, während der Aufwand für die Validierung des Pfades mit der Anzahl kontinuierlicher Variablen (Dimensionen) und der Pfadlänge wächst. Da diese Schritte mehrmals wiederholt durchgeführt werden müssen, kann es abhängig von diesen Parametern zu beträchtlichen Gesamtlaufzeiten des Verifikationsprozesses kommen.

In Algorithmus 1 ist der iterative Prozess in Pseudo-Code dargestellt. Ein solches CEGAR-Verfahren wird auch in dem in dieser Arbeit geschilderten Verfahren eingesetzt. Bevor die Schritte im Einzelnen vorgestellt werden, wollen wir zunächst die Menge der zu verifizierenden Modelle näher betrachten.

4.1.2 Charakteristik Schritt-diskreter Modelle

Wie bereits in Kapitel 3.2 bemerkt wurde, stammen die hier betrachteten Systeme aus der Klasse der Schritt-diskreten Hybriden Systeme, welche sowohl mit den gebräuchlichen CASE-Tools, als auch direkt in der am weitesten verbreiteten Modellierungssprache C modelliert sind. Neben dieser, in Abschnitt 3.2 auf Seite 28 geschilderten, strukturellen Einschränkung gegenüber Zeit-kontinuierlichen Hybriden Systemen lassen sich jedoch weitergehende typische Eigenschaften solcher Modelle beobachten. Besonders auffällig insbesondere im Unterschied zu akademischen Beispielmodellen ist der große diskrete Zustandsraum solcher industriell eingesetzten Modelle, der nicht selten in der Größenordnung $> 2^{100}$ liegt. Dies wird im Wesentlichen durch zwei Modellierungsansätze impliziert:

1. Die Komponenten verwenden häufig neben kontinuierlichen Variablen auch diskrete zur Modellierung z.B. von Timeout-Ausdrücken, Validierungszählern und -Bits, Fehlerzähler, Verzögerungselemente zur Synchronisation von Daten, Enable-Bits in Filterkomponenten, unterschiedliche Taktungen, usw. Daneben sind häufig auch größere, parallel agierende Zustandsautomaten direkt für die Steuerung regelungstechnischer Gesetze in die Komponenten integriert bzw. mit für die Außenwelt kommunikationsrelevanten Aufgaben betraut.

2. Jedes System besteht üblicherweise aus (hierarchisch organisierten) parallel komponierten Unterkomponenten, welchen jeweils Spezialaufgaben zuge-

teilt sind. Zu diesen Aufgabenbereichen gehört auch die Abwicklung umfangreicherer Kommunikationsprotokolle, dem Scheduling verschiedener sowohl periodischer, als auch Ereignis-getriebener Tasks, der Kontrolle und etwaigen Aktivierung von Rückfallebenen in problematischen Situationen (Fail-Safe Strategien) oder der Ansteuerung von Benutzungsschnittstellen (HMI-Interface), welche i.d.R. vorwiegend durch diskrete Automaten behandelt werden. Die Anzahl der diskreten Zustände dieser parallel laufenden Automaten gehen multiplikativ in die Größe des diskreten Gesamtzustandsraums ein.

Beide Umstände tragen jeweils erheblich zur Vergrößerung des diskreten Zustandsraumes bei und in beiden sind die diskreten Zustandsübergänge nicht von einmaligen, ausschließlich diesen Übergängen zuzuordnenden kontinuierlichen Zustandsübergängen begleitet. Vielmehr finden sich sowohl die kontinuierlichen Vorbedingungen (*Guard Sets*) zur Aktivierung eines Übergangs, als auch die kontinuierlichen Zustandstransformationen (*Jump Set Funktionen*), an einer Vielzahl von weiteren diskreten Zustandsübergängen wieder. Dies liegt daran, daß z.B. die Kommunikationsprotokoll-relevanten Zustände anderer Unterkomponenten nur in selteneren Fällen eine Auswirkung auf die anzuwendenden regelungstechnischen Steuerungsaufgaben ausüben, auch wenn dies natürlich nicht ausgeschlossen ist und mithin auch die Interaktion beider Komponenten bzgl. der Verifikation zu betrachten ist.

Auf der anderen Seite wird die Anzahl der im System möglichen Berechnungsabläufe im kontinuierlichen Zustandsraum, welche durch alternierende Folgen von Guard Sets und Jump Set Funktionen beschrieben sind, durch Hinzunahme von diskreten Unterkomponenten oder diskreten Variablen nicht erhöht. Somit steht einer großen Anzahl von diskreten Zuständen eine vergleichsweise kleine Zahl von kontinuierlichen Vorbedingungen und Zustandstransformationen (*Regelungstechnische Gesetze* oder kurz: *Regelungsgesetze*) gegenüber. Diese Verhältnismäßigkeit ist durch Tabelle 4.1 dargestellt, in der sowohl für hier betrachtete industrielle Fallbeispiele, als auch für akademische Modelle diese Relation vergleichend dargestellt wird.

Wie in der Tabelle zu beobachten gilt für die industriellen Modelle

$$|GJ| \ll |Z| \lesssim |T| \tag{4.1}$$

wobei GJ die im Modell vorkommenden Guard Set/Jump Set Funktionspaare sind, Z die Menge der diskreten Zustände und T die Menge der Transitionen eines Schritt-diskreten Hybriden Systems, wie in Abschnitt 3.2 auf Seite 28 eingeführt. Es gibt also sehr viel mehr diskrete Zustände[1] und damit auch Transitionen in den Modellen, als verschiedene regelungstechnische Gesetze Anwendung finden.

Die Bestimmung der in der Tabelle angegebenen Anzahl der regelungstechnischen Gesetze basiert auf der Aufsummierung von verschiedenen in Simulationsläufen beobachteten Guard Set / Jump Set Paaren. Somit stellt diese Größe nur eine untere Grenze dar, die jedoch praktische Relevanz besitzt, wie später deutlich werden wird.

Für die Läufe solcher Systeme hat dies zur Folge, daß aufgrund der exzessiven Replikation von kontinuierlichen Berechnungsschritten sich nicht nur die einzelnen Berechnungsschritte, sondern auch deren Abfolge bei verschiedenen Läufen sehr häufig wiederholt. Dieser durch das hier vorgestellte Verfahren ausgenutzte Umstand wird noch in später präsentierten Statistiken dargestellt werden.

[1] Im Gegensatz zu der Transitionszahl ist die Zustandszahl technisch einfach anzugeben. Da die Zahl der erreichbaren Zustände notwendigerweise kleiner ist als die Anzahl der Transitionen, stellt die Zustandszahl eine sichere untere Grenze dar.

Modell	diskrete Zustände	kontinuierliche Variablen *total/input/state*	Regelungsgesetze
Window lifting system (ASCET,BMW)	2^{26}	27/2/4	~ 60
Autopilot System (SCADE,[VER98])	2^{51}	423/7/25	~ 80
Desante[a], (SCADE,Hispano-Suiza)	2^{1055}	358/14/0	8
Akademische Modelle			
Cruise Control System ([SFHK04])	2^4	6/0/6	~ 24
Distributed Robot Control ([AHH96])	2^9	12/0/12	< 1000
Mutual Exclusion example ([ADI02])	2^6	3/0/3	~ 16

[a] Castings zwischen int- und float-Größen sind hierbei abstrahiert

Tabelle 4.1: Industrielle open-loop Modelle vs. akademische closed-loop Modelle

4.1.3 Konsequenz für die Verifikation

In dem hier vorgestellten Abstraktionsverfahren sind Gründe für falsche Gegenbeispiele ausschließlich in der Unlösbarkeit der dazugehörigen abstrahierten Berechnungssequenzen zu suchen. Vor diesem Hintergrund ist es nahe liegend, nicht nur den Pfad, bzw. Teilpfad auszuschließen, der aufgrund einer Unvereinbarkeit mit der dazugehörigen konkreten Berechnungsfolge als ungültig zu erklären ist, sondern gleich alle Pfade auszuschließen, die aufgrund der gleichen Berechnungsfolge im kontinuierlichen Zustandsraum ebenfalls ungültig sind. Somit wird vermieden, im Verlauf der iterativen Abstraktionsverfeinerung je wieder einen ungültigen Pfad zu erhalten, der aus einem bereits bekannten Grund zu invalidieren ist.

In Modellen mit einem großen diskreten Zustandsraum existieren entsprechend viele ungültige Pfade, die als solche auf Grundlage der beinhalteten unlösbaren Berechnungsfolgen zusammenzufassen sind. Daher ist ein kategorischer Ausschluß von Pfadklassen geeignet, um in einem iterativen Verfeinerungsverfahren mit deutlich weniger Iterationen die Erreichbarkeit bzw. Unerreichbarkeit zu entscheiden, als dies bei Verfeinerungsverfahren wie [CFH+03] der Fall ist, wo nur Fragmente des ungültigen Pfades im Verfeinerungsschritt ausgeschlossen werden.

Das in diesem Kapitel vorgestellte Verfahren basiert auf dieser Überlegung und schließt ungültige Pfadklassen unter Verwendung eines inkrementell konstruierten ω-Automaten aus. Das Verfahren lernt in kompakter Form alle ungültigen Läufe dieser Pfadklassen und verhindert durch Verfeinerung, daß diese erneut auftreten können. Bevor das Verfahren vorgestellt wird, führen wir im folgenden Abschnitt einige, in diesem Kapitel häufiger verwendete Notationen ein.

4.2 Definition allgemeiner Funktionen und Notationen

Eine im Verfahren häufig vorkommende Datenstruktur sind Sequenzen von n Elementen eines beliebigen Typs E_T der Form $(e_1, e_2, \ldots, e_n) \in (E_T)^n$. Wir führen hierfür die folgende abkürzende Schreibweise ein: $\langle e_n \rangle$, $\langle e_n \rangle^{\geq 0}$

$$\langle e_n \rangle \stackrel{\text{def}}{=} (e_1, e_2, \ldots, e_n)$$

Für Sequenzen, deren Nummerierung bei 0 anstatt bei 1 beginnt, schreiben wir

$$\langle e_n \rangle^{\geq 0} \stackrel{\text{def}}{=} (e_0, e_1, \ldots, e_n)$$

$|\langle e_n \rangle|$

Die *Länge* einer Sequenz $\langle e_n \rangle$ wird durch $|\langle e_n \rangle| \stackrel{\text{def}}{=} n$ bzw. $\left|\langle e_n \rangle^{\geq 0}\right| \stackrel{\text{def}}{=} n{+}1$ bezeichnet.

Auf diesen Sequenzen werden Selektions- und Substitutionsfunktionen benötigt, welche an dieser Stelle Typ-unabhängig für verschiedene Arten von Sequenzen definiert werden.

Definition 13 (Selektion) *Für eine Sequenz $\langle e_n \rangle$ liefert die* Selektionsfunktion *$sel : (E_T)^n \times \mathbb{N} \rightarrow E_T$ das k-te Element der Sequenz, formal*

$$sel\,(\langle e_n \rangle, k) \stackrel{def}{=} e_k, k \leq n$$

$\langle e_n \rangle_{(k)}$

Als Kurzschreibweise verwenden wir $\langle e_n \rangle_{(k)} \stackrel{def}{=} sel\,(\langle e_n \rangle, k)$.

Definition 14 (Substitution) *Mit $\langle e_n \rangle = (e_1, e_2, \ldots, e_{k-1}, e_k, e_{k+1}, \ldots, e_n)$ als gegebene Sequenz liefert die* Substitutionsfunktion *$subst : (E_T)^n \times E_T \times \mathbb{N} \rightarrow (E_T)^n$ eine Sequenz gleichen Typs, in der das k-te Element, $k \leq n$, substituiert ist durch ein gegebenes e'_k, formal*

$$subst\left(\langle e_n \rangle, e'_k, k\right) \stackrel{def}{=} (e_1, e_2, \ldots, e_{k-1}, e'_k, e_{k+1}, \ldots, e_n)$$

$\langle e_n \rangle_{[(k)\backslash e'_k]}$

Als Kurzschreibweise verwenden wir $\langle e_n \rangle_{[(k)\backslash e'_k]} \stackrel{def}{=} subst\left(\langle e_n \rangle, e'_k, k\right)$.

Definition 15 (Konkatenation) *Zwei Sequenzen $\langle e_n \rangle$ und $\langle e'_m \rangle$ werden mit dem Konkatenationsoperator konkateniert zu einer neuen Sequenz mit:*

$$\langle e_n \rangle \cdot \langle e'_m \rangle \stackrel{def}{=} (e_1, e_2, \ldots, e_n, e'_1, e'_2, \ldots, e'_m)$$

4.3 Das ω-Cegar Verfahren

Dieser umfangreichere Abschnitt stellt das iterative Abstraktionsverfeinerungsverfahren zum Ausschluß von Klassen ungültiger Gegenbeispiele in einfachster Form vor. Komplexere Erweiterungen werden in später folgenden Abschnitten eingeführt.

Wie schematisch in Abbildung 4.1 dargestellt, beginnt das Verfahren mit einer einfachen Abstraktion, einem Automaten A_0 welcher die gleiche diskrete Struktur wie der hybride Automat H besitzt. In jeder Iteration werden die vom Model-Checker generierten Pfade analysiert und im Falle der Ungültigkeit minimale Teilberechnungssequenzen in dem dazugehörigen kontinuierlichen Zustandsraum der Konkretisierung dieser Pfade ermittelt, welche für deren Ungültigkeit hinreichend sind. Ein diese Teilberechnungssequenzen inkrementell lernender ω-Automat A_C akzeptiert nur ω-Wörter, welche keiner dieser hinreichenden Bedingungen für ungültige Pfade genügen. Diese Eigenschaft wird durch ein Kreuzprodukt mit der initialen Abstraktion A_0 kombiniert, d.h. der auf Grundlage aller bis zum i-ten Iterationsschritt bekannten Konflikte konstruierte Automat $A_{C_{i+1}}$ wird mit A_0 zur Konstruktion des im folgenden Iterationsschritt zu verifizierenden Automaten A_{i+1} durch $A_{i+1} = A_{C_{i+1}} \times A_0$ gebildet.

Aufgrund der Verwendung eines ω-Automaten zum Erlernen von Berechnungssequenzen werden wir das nun vorgestellte Verfahren zur Unterscheidung von anderen als ω-Cegar Verfahren bezeichnen.

Die folgenden Abschnitte beschreiben die geschilderten Verfahrensschritte *initiale Abstraktion*, *Pfadanalyse*, *Konstruktion des ω-Automaten* und *Parallelkomposition*.

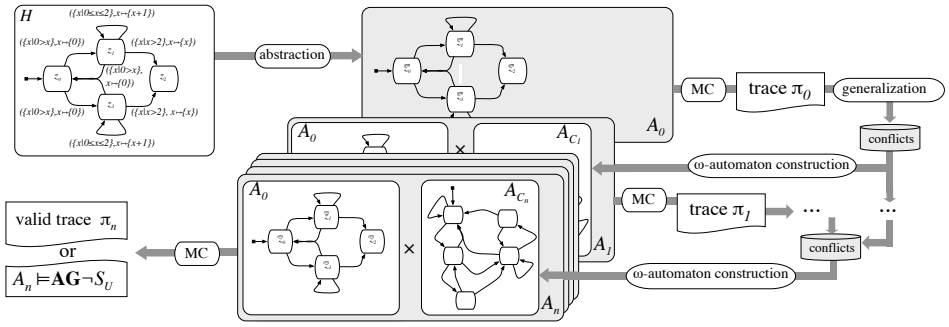


Abbildung 4.1: Schematische Übersicht des iterativen Abstraktionsverfeinerungsverfahrens.

4.3.1 Initiale Abstraktion

Ebenso wie viele andere iterative Abstraktionsverfeinerungsverfahren hybrider Systeme ([CFH+03]) beginnen wir initial mit genau dem Transitionssystem, welches die gleiche diskrete Struktur aufweist, wie das hybride System H.

Definition 16 (Initiale Abstraktion) *Das initial verwendete abstrakte Transitionssystem* $A_0 = \left(\hat{Z}, \hat{z}_0, \hat{T}\right)$ *bzgl. eines Pfad-Transitionssystems* $TS_H = (S, S_0, E)$ *eines Hybriden Systems* $H = (Z, z_0, X, X_0, T, g, j)$ *mit* $S = Z \times X$ *ist durch die Funktion* $\alpha_0 : S \to \hat{Z}$ *so definiert, daß für jeden Zustand* $z_k \in Z$ *ein isomorpher Zustand* $\hat{z}_k \in \hat{Z}$ *existiert, mit*

$$\alpha_0\left((z_k, x)\right) = \hat{z}_k$$

A_0 ist bzgl. diskreter Zustände und Transitionen zu H isomorph, während jedes kontinuierliche Verhalten ausgeklammert wird. Aufgrund der Definition von A_0 gilt trivialerweise $A_0 \geq TS_H$, d.h. A_0 ist eine Abstraktion des Pfad-Transitionssystems nach Definition 6 auf Seite 24.

Dieses Transitionssystem kann nun mit den in Kapitel 2.4 beschriebenen Verfahren durch einen Model-Checker behandelt werden und im Falle der Erreichbarkeit ein entsprechender Pfad generiert werden. Ist die Eigenschaft nicht erreichbar, so können wir nach Lemma 1 auf Seite 24 dieses Ergebnis auf das Pfad-Transitionssystem TS_H übertragen. Wird vom Model-Checker hingegen ein Pfad errechnet, so muß dieser, wie im folgenden Abschnitt beschrieben, validiert und im Falle der Ungültigkeit das Abstraktionssystem verfeinert werden.

Daneben kommt dem initialen abstrakten Transitionssystem jedoch eine Schlüsselrolle in der Verfeinerung für spätere Abstraktionsschritte zu. Wie bereits in Abbildung 4.1 skizziert, wird A_0 unter Ausnutzung der Isomorphie zu der diskreten Struktur von H in folgenden Iterationsschritten im Rahmen eines Kreuzprodukts mit dem zu konstruierenden ω-Automaten wiederverwendet.

Wir werden daher zur Unterscheidung von A_0 gegenüber beliebigen Abstraktionen A_i, bei Irrelevanz des Iterationsindex i auch einfach mit A bezeichnet, für A_0 das Tripel $A_0 = \left(\hat{Z}, \hat{z}_0, \hat{T}\right)$ und für A_i das Tripel $A_i = \left(\hat{S}, \hat{S}_0, \hat{E}\right)$ verwenden.

$A_0 = (\hat{Z}, \hat{z}_0, \hat{T})$
$A_i = (\hat{S}, \hat{S}_0, \hat{E})$

4.3.2 Pfadanalyse

Zur Analyse der Gültigkeit eines Pfades $\hat{\pi} = \langle \hat{s}_n \rangle^{\geq 0}$ mit der in Abschnitt 4.2 auf Seite 37 eingeführten abkürzenden Schreibweise $\langle \hat{s}_n \rangle^{\geq 0} \stackrel{\text{def}}{=} (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n)$, den der Model-Checker bzgl. des abstrakten Modells A ermittelt, muß festgestellt werden, ob es gemäß der Abstraktionsrelation α einen korrespondierenden Pfad

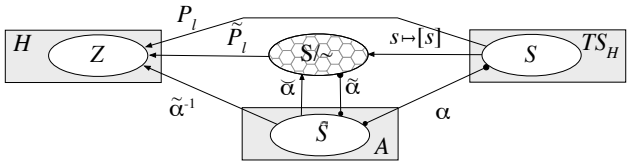


Abbildung 4.2: Abbildungsrelationen und -funktionen zwischen den Zustandsmengen des Hybriden System H, des Pfad-Transitionssystems TS_H und des abstrakten Transitionssystems A

$\pi = ((z_0, x_0), (z_1, x_1), \ldots, (z_n, x_n)) = \langle (z_n, x_n) \rangle^{\geq 0}$ in dem Pfad-Transitionssystem TS_H gibt, so daß $\forall 0 \leq i \leq n : ((z_i, x_i), \hat{s}_i) \in \alpha$ gilt. Ist dies der Fall, kann das Verfahren mit einem errechneten Pfad π terminieren, andernfalls ist aus dem ungültigen Pfad $\hat{\pi}$ eine Information zur nachfolgenden Verfeinerung von A zu ermitteln, was durch die Extraktion so genannter minimaler Konflikte erreicht wird.

Im Folgenden wird zunächst die Projektion θ des Pfades $\hat{\pi}$ auf das Schritt-diskrete Hybride System H eingeführt, mit deren Hilfe die Berechnungsfolge im kontinuierlichen Zustandsraum rekonstruiert werden kann. Hierfür wird eine Lösungsfunktion $\mathcal{L}$ definiert, welche die (Un-)Gültigkeit der Berechnungsfolge ermittelt. Die Lösungsfunktion wird im Falle der Ungültigkeit der Berechnungsfolge für die Extraktion und Minimierung von Konflikten aus selbiger von einem Suchalgorithmus verwendet, der eine Auswahl von Konflikten ermittelt.

4.3.2.1 Pfadprojektion

Für einen korrespondierenden Pfad $\pi = ((z_0, x_0), (z_1, x_1), \ldots, (z_n, x_n)) = \langle (z_n, x_n) \rangle^{\geq 0}$ in dem Pfad-Transitionssystem TS_H muß es insbesondere eine Folge $x_0, x_1, \ldots, x_n$ von Elementen des kontinuierlichen Zustandsraums X geben, die jeweils der alternierenden Folge von Guard Set Bedingungen und Jump Set Funktionen der gemäß Abstraktionsrelation dazugehörigen diskreten Transitionen aus der Transitionsmenge T des Hybriden Systems H genügen. Zur Berechnung dieser Folge muß bezüglich der Abstraktionsrelation α entsprechend gewährleistet sein, daß jedem Pfad $\hat{\pi}$ des abstrakten Systems A die dazugehörige Guard-Set/Jump-Set-Transformationssequenz zugeordnet werden kann, deren Lösbarkeit ermittelt werden muß.

Dies ist bei Abstraktionsrelationen der Fall, die eine Zuordung jedes abstrakten Zustands $\hat{s}$ zu genau einem Zustand z des Hybriden Systems H gewährleisten, wodurch mit Hilfe der Transitionsrelation T eindeutig die dazugehörigen Guard Set und Jump Set Funktion zuzuordnen ist. Im Folgenden wird eine solche Abbildungsfunktion $\tilde{\alpha}^{-1} : \hat{S} \rightarrow Z$ unter Angabe der notwendigen Eigenschaften bzgl. α formal hergeleitet.

$\tilde{\alpha}^{-1}$

Wir führen eine Partitionierungsfunktion $P_l : S \rightarrow Z$ ein, welche durch $P_l((z, x)) = z$ definiert ist, wie in Abbildung 4.2 dargestellt. Diese impliziert eine Äquivalenzrelation $s_1 \sim s_2 :\Leftrightarrow P_l(s_1) = P_l(s_2)$, so daß alle Zustände $s \in S$ des Pfad-Transitionssystems TS_H, die dem gleichen diskreten Zustand $z \in Z$ zuzuordnen sind, in einer Äquivalenzklasse $[s]$ der Quotientenmenge $S/\sim$ zusammengefaßt werden. Die Quotientenmenge $S/\sim$ ist isomorph zu Z und die eindeutige Abbildung $\tilde{P}_l : S/\sim \rightarrow Z$ mit $\tilde{P}_l([s]) = P_l(s)$ ist aufgrund der Definition von $\sim$ wohldefiniert.

Von α wird nun eine Äquivalenzklassenrelation $\tilde{\alpha} \subseteq S/\sim \times \hat{S}$ abgeleitet mit $\tilde{\alpha} = \{([s], \hat{s}) \,|\, (s, \hat{s}) \in \alpha\}$. Unter der Voraussetzung, daß $\tilde{\alpha}$ eine injektive Relation ist, ist die Äquivalenzklasse $[s]$ eines Zustands $\hat{s} \in \hat{S}$ durch $\tilde{\alpha}$ eindeutig und daher die inverse Relation $\breve{\alpha}$ eine wohldefinierte Funktion $\breve{\alpha} : \hat{S} \rightarrow S/\sim$. Die gesuchte

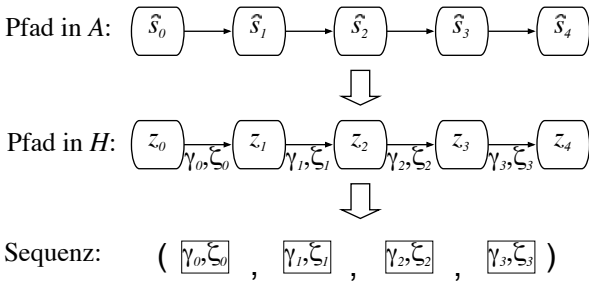


Abbildung 4.3: Projektion von Pfaden eines abstrakten Transitionssystems A bzgl. des Schritt-diskreten Hybriden Systems H auf eine GJ-Sequenz durch die Funktion θ

Abbildungsfunktion $\tilde{\alpha}^{-1} : \hat{S} \to Z$ ist dann definiert durch $\tilde{\alpha}^{-1} = \tilde{P}_l \circ \breve{\alpha}$.

Die für die Wohldefiniertheit von $\tilde{\alpha}^{-1}$ benötigte Voraussetzung der Injektivität der Relation $\tilde{\alpha}$ ist bei α_0 trivialerweise erfüllt, da α_0 isomorph zur Partitionierungsfunktion P_l ist und $\tilde{\alpha}$ somit eine bijektive Abbildung darstellt.

Guard Set / Jump Set Transformationssequenzen Mit Hilfe der Funktion $\tilde{\alpha}^{-1}$ können wir nun Pfade abstrakter Modelle auf Folgen diskreter Zustände aus Z abbilden und damit die zugrunde liegende Berechnungssequenz ermitteln.

Definition 17 (Guard Set / Jump Set Transformationssequenz) *Eine* Guard Set / Jump Set Transformationssequenz *(abgekürzt* GJ-Sequenz*) ist definiert durch* $\langle(\gamma_n, \zeta_n)\rangle = ((\gamma_1, \zeta_1), \ldots, (\gamma_n, \zeta_n)), \gamma_i \in G, \zeta_i \in J$*, wobei* G *die Menge aller Guard Sets und* J *die Menge aller Jump Set Funktionen eines Hybriden Systems sind. Wir bezeichnen die Menge aller GJ-Sequenzen mit* $C = \bigcup_{n\in\mathbb{N}} (G \times J)^n$. *GJ-Sequenz* *C*

Wie in Abbildung 4.3 dargestellt führen wir eine Projektion von Pfaden eines abstrakten Transitionssystems A mittels des dazugehörigen Schritt-diskreten Hybriden Systems H auf eine GJ-Sequenz mit der folgenden Definition durch:

Definition 18 (Pfadprojektion) *Gegeben seien ein Pfad* $\hat{\pi} = \langle \hat{s}_n \rangle^{\geq 0}$ *eines Transitionssystems* $A = \left(\hat{S}, \hat{s}_0, \hat{E}\right)$*, welches eine Abstraktion eines Pfad-Transitionssystems* TS_H *unter einer Abstraktionsrelation* α *darstellt. Mit* $\overrightarrow{A}$ *als Menge aller Pfade in* A *wird die dazugehörige GJ-Sequenz* $\langle(\gamma_n, \zeta_n)\rangle$ *berechnet durch eine Funktion* $\theta : \overrightarrow{A} \to C$*, die wie folgt definiert ist:*

$$\theta(\hat{\pi}) = \langle(\gamma_n, \zeta_n)\rangle \quad \textit{mit}$$
$$\gamma_i = g(t_i), \zeta_i = j(t_i), t_i = \tilde{\alpha}^{-1}\left((\hat{s}_{i-1}, \hat{s}_i)\right) := \left(\tilde{\alpha}^{-1}(\hat{s}_{i-1}), \tilde{\alpha}^{-1}(\hat{s}_i)\right)$$

Wir bezeichnen θ *auch als* Pfadprojektionsfunktion.

Damit sind wir nun in der Lage, aus einem Pfad eines abstrakten Transitionssystems A die Berechnungssequenz des dazugehörigen Schritt-diskreten Hybriden Systems H zu ermitteln. Dies ist für die im Folgenden geschilderten Auswertungen der Berechnungen elementar.

4.3.2.2 Auswertung der Berechnungen im kontinuierlichen Zustandsraum

Ein Pfad $\hat{\pi}$ ist gültig, wenn die Berechnungen im kontinuierlichen Zustandsraum des korrespondierenden Pfads π im Pfad-Transitionssystem TS_H unter dessen Transitionsrelation zu einem gültigen Zustand führen. Daher beschreiben wir in diesem Abschnitt das Vorgehen zur Auswertung der GJ-Berechnungssequenz.

Für die Pfadanalyse wird zunächst aus dem Pfad $\hat{\pi}$ die *GJ*-Sequenz $c = \theta(\hat{\pi}) = \langle(\gamma_n, \zeta_n)\rangle$ ermittelt, welche die schrittweisen Transformationen auf dem initialen kontinuierlichen Ausgangszustandsraum X_0 beschreibt. Gemäß der semantischen Definition von TS_H durch Definition 10 auf Seite 30 führt die für alle $i = 0, \ldots, n$ sukzessiv durchzuführende, jeweils alternierende Anwendung von Schnittmengenbildung mit den Guard Sets γ_i, gefolgt von Zustandstransformationen durch die Funktionen ζ_i zu einer Folge von kontinuierlichen Zustandsräumen $X_{seq} = \langle X_n\rangle^{\geq 0} \in 2^{X^{n+1}}$ mit $X_i = \left\{x' \middle| \exists x \in (X_{i-1} \cap \gamma_i) \wedge x' \in \zeta_i(x)\right\}$. Wenn $X_n \neq \emptyset$, dann existiert eine Folge $\langle x_n\rangle^{\geq 0}, x_i \in X_i$ und folglich auch ein korrespondierender Pfad π in TS_H mit $\pi = \left\langle\left(\tilde{\alpha}^{-1}(\hat{s}_n), x_n\right)\right\rangle^{\geq 0}$, welcher ein gültiges Gegenbeispiel darstellt.

X_{seq}

Gemäß dieser Beschreibung definieren wir für die spätere Bezugnahme die Funktion $\mathcal{L} : C \times 2^X \to 2^X$ zur Berechnung der Folge :

Definition 19 (Lösungsfunktion) *Gegeben sein eine GJ-Sequenz $c \in C$ mit $c = \langle(\gamma_n, \zeta_n)\rangle$ und eine Menge kontinuierlicher Zustände $\tilde{X} \subseteq X$. Die* Lösungsfunktion *$\mathcal{L}: C \times 2^X \to 2^X$ ist definiert als:*

$$\mathcal{L}\left(c, \tilde{X}\right) := \begin{cases} \tilde{X} & \text{wenn } |c| = 0 \\ \left\{\zeta_n(x) \middle| x \in \left(\mathcal{L}\left(\langle(\gamma_{n-1}, \zeta_{n-1})\rangle, \tilde{X}\right) \cap \gamma_n\right)\right\} & \text{wenn } |c| > 0 \end{cases}$$

Wir bezeichnen $\mathcal{L}$ bzgl. einer *GJ*-Sequenz c und einer Menge $\tilde{X}$ als *lösbar*, wenn $\mathcal{L}\left(c, \tilde{X}\right) \neq \emptyset$, andernfalls als *unlösbar*.

Die Bestimmung einer einzelnen Lösung $\langle x_n\rangle^{\geq 0}, x_i \in X_i$ für die *GJ*-Sequenz c eines vollständigen Pfades $\hat{\pi}$ ist allein mit der Funktion $\mathcal{L}$ nicht möglich, hier muß in allgemeinster Form eine *Rückpropagation* der Lösungsmenge $X_n = \mathcal{L}(c, X_0)$ durch die umgekehrte Reihenfolge der inversen Jump Set Funktionen erfolgen, so daß schließlich eine Teilmenge $X_0' \subseteq X_0$ bestimmt wird, in der alle gültigen Lösungen zu finden sind. Formal führen wir hierfür die Funktion $\tilde{\mathcal{L}}: C \times 2^X \to 2^X$ ein:

$\tilde{\mathcal{L}}$

$$\tilde{\mathcal{L}}\left(c, \tilde{X}\right) := \left\{\zeta_1^{-1}(x) \middle| x \in \left\{\zeta_2^{-1}(x) \middle| x \in \left\{\ldots \middle| x \in \left\{\zeta_n^{-1}(x) \middle| x \in X_n = \mathcal{L}\left(c, \tilde{X}\right)\right\} \ldots\right\}\right\}\right\}$$

Wenn $X_0' = \tilde{\mathcal{L}}(c, X_0) \neq \emptyset$, dann gibt es eine gültige Konkretisierung $\langle x_n\rangle^{\geq 0}, x_i \in X_i$ der Berechnungssequenz, welche mit $x_0 \in X_0'$ beginnt. Gemäß obigen Ausführungen ist dies immer genau dann der Fall, wenn $\mathcal{L}(x, X_0) \neq \emptyset$.

Wie später in Kapitel 6.5.1 genauer beschrieben, wird in der Praxis sowohl die Funktion $\mathcal{L}$, als auch die Funktion $\tilde{\mathcal{L}}$ zur Bestimmung einer konkreten Lösung durch einen Solver für lineare (Un-)Gleichungssysteme realisiert, welcher die folgenden Funktionalitäten bereitstellt:

- Die Funktion $\mathcal{L}' : C \times 2^X \to \mathbb{B}$ berechnet, ob $\exists \langle x_n\rangle^{\geq 0} \in X^{n+1}, x_0 \in X_0, x_1 \in X_1, \ldots, x_n \in X_n, \langle X_n\rangle^{\geq 0} = X_{seq}$.

$\mathcal{L}'$

- Die Funktion $\tilde{\mathcal{L}}' : C \times 2^X \to \mathbb{B} \times X^{n+1}$ ermittelt zusätzlich einen konkreten Repräsentanten $\langle x_n\rangle^{\geq 0}$, was ausschließlich für die finale Pfadberechnung von π bzgl. des Pfad-Transitionssystems TS_H relevant wird.

$\tilde{\mathcal{L}}'$

Für die Realisierung der Funktion $\mathcal{L}$ bzw. $\tilde{\mathcal{L}}$ können alternative Algorithmen verwendet werden. Dies schließt insbesondere Verfahren ein, welche nicht auf lineare Arithmetik beschränkt sind und somit auch nicht-lineare Berechnungen unterstützen, wie dies beispielsweise bei überapproximierenden Intervallverfahren der Fall wäre. Damit würden entsprechend auch nicht-lineare Hybride System verifiziert

werden können. Der iterative Abstraktionsverfeinerungsalgorithmus ist unabhängig von der Umsetzung dieser Funktionen soweit sie verlässliche Ergebnisse liefern.

Mit Hilfe der Funktion $\mathcal{L}$ und $\tilde{\mathcal{L}}$ bzw. deren Implementierungen $\mathcal{L}'$ und $\tilde{\mathcal{L}}'$ können wir nun eine *GJ*-Berechnungssequenz auf ihre Gültigkeit überprüfen und darüber hinaus eine Folge von kontinuierlichen Zuständen ermitteln, die der Berechnungssequenz genügen. Dies ist neben der beschriebenen Erkennung eines gültigen Pfades und dessen Konkretisierung auch für die im Folgenden beschriebene Identifikation von Konflikten relevant.

4.3.2.3 Identifikation von Konflikten

Im Falle eines im abstrakten System A ermittelten ungültigen Pfades $\hat{\pi}$ gäbe es keine passende Folge von kontinuierlichen Zuständen und folglich $\mathcal{L}(\theta(\hat{\pi}), X_0) = \emptyset$. Wie in Sektion 4.1.3 bereits erwähnt, ist es für Hybride Systeme mit großen Zustandsräumen nun sinnvoll, nicht nur genau diesen Pfad $\hat{\pi}$ für folgende Iterationsschritte auszuschließen, sondern gleich alle Pfade, die aus den gleichen Gründen ungültig sind. An dieser Stelle muß nun also mindestens ein Grund identifiziert werden, welcher einerseits ausreichend ist, diesen Pfad zu verhindern und andererseits allgemeingültig genug, um möglichst viele andere Pfade mit abzudecken. Diese Gründe, bestehend aus *GJ*-Sequenzen, bezeichnen wir als Konflikte mit folgender Definition:

Definition 20 (Konflikt) *Ein* Konflikt *c ist eine GJ-Sequenz mit* $\mathcal{L}\left(c, \tilde{X}\right) = \emptyset, \tilde{X} \subseteq X$ *mit X als dem kontinuierlichen Zustandsraums unseres Hybriden Modells H. Für* $\tilde{X} = X_0$ *mit* X_0 *als dem initiale kontinuierliche Zustandsraum von H wird der Konflikt als* initial*, für* $\tilde{X} = X$ *als* invariant *bezeichnet.*

Gemäß dieser Definition ist die Pfadprojektion $\theta(\hat{\pi})$ des in A ermittelten Pfades $\hat{\pi}$ als Konflikt zu bezeichnen, jedoch handelt es sich i.d.R. nicht um einen minimalen Konflikt. Nicht alle Elemente der *GJ*-Sequenz tragen notwendigerweise dazu bei, daß $\mathcal{L}\left(\theta(\hat{\pi}), \tilde{X}\right) = \emptyset$, sondern es sind vielmehr Teilsequenzen von $\theta(\hat{\pi})$, welche bereits zu einer leeren Lösungsmenge in $\mathcal{L}$ führen. Wir bezeichnen eine Teilsequenz als minimalen Konflikt, wenn jedes Entfernen von *GJ*-Paaren am Anfang oder am Ende der Folge zu einer nicht-leeren Lösungsmenge führt.

Selektionsstrategie Innerhalb eines Pfades $\hat{\pi}$ finden sich möglicherweise mehrere minimale Konflikte, die hier isoliert werden können. Da bereits ein minimaler Konflikt ausreichend für den Ausschluß von $\hat{\pi}$ ist, würde sich der kürzeste Konflikt zur ausschließlichen Berücksichtigung anbieten.

Durch Entfernen eines Teils oder aller Elemente eines isolierten Konflikts aus der gegebenen GJ-Sequenz können weitere Konflikte ermittelt werden, soweit diese nicht zufällig die entfernten Teile ebenfalls beinhaltet haben. Somit kann aus einer GJ-Sequenz eine Menge von Konflikten isoliert werden.

Es stellt sich jedoch die Frage, ob eine vollständige Extraktion aller Konflikte sinnvoll ist. Es ist durchaus möglich, daß der Ausschluß eines im k-ten Iterationsschritt gefundenen Konflikts c_1 in folgenden Iterationen der Abstraktionsverfeinerung bedingt durch die diskrete Struktur des Modells ein erneutes Auftreten anderer, ebenfalls im k-ten Schritt aufgetretener Konflikte $c_2, c_3, \ldots, c_j$ ausschließt, bzw. diese zumindest bzgl. der nachzuweisenden Eigenschaft keine Relevanz mehr besitzen. In diesem Fall würde das Erlernen der Konflikte $c_2, c_3, \ldots, c_j$ den Automaten unnötig vergrößern.

Betrachten wir hierzu die schematische Darstellung verschiedener möglicher Positionen von Konflikten in dem auf zwei Pfade beschränkten diskreten Zustandsgraphenausschnitt in Abbildung 4.4. Angenommen, der Model-Checker sucht

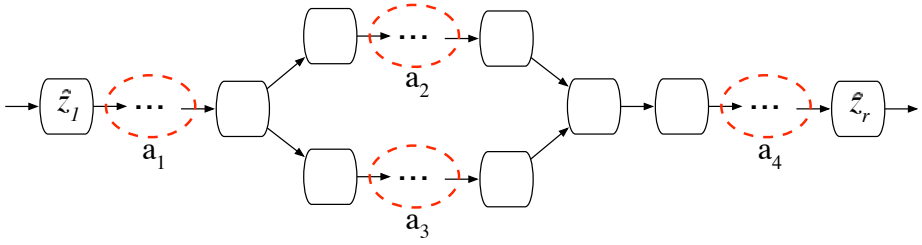


Abbildung 4.4: Schematische Betrachtung zu möglichen Konfliktpositionen in dem diskreten Zustandsgraphen

einen Pfad von $\hat{z}_1$ nach $\hat{z}_r$, wobei wir hier ausdrücklich den Zustandsraum $\hat{Z}$ aus A_0 betrachten. Die möglichen relevanten Bereiche, in denen Konflikte auftreten können, welche eine spätere Validierung des Pfades erzwingen, liegen in den *Bereichen* a_1 bis a_4, wobei ein Bereich hier informal als Menge von parallelen *Teilpfaden* verstanden sein soll, die u.a. die gleiche Berechnungssequenz im kontinuierlichen Zustandsraum durchführen. Wichtig bei diesem Beispiel ist, daß die den Teilpfaden eines Bereichs gemeinsamen Konflikte verschieden von denen der anderen Bereiche sind, da ein Auswahlproblem andernfalls nicht existent wäre. Es ist offensichtlich, daß ein etwaiger Konflikt c_1, der allen durch a_1 führenden Teilpfaden gemeinsam ist, oder ein Konflikt c_4, der alle durch a_4 führenden Teilpfade begleitet, bereits ausreichend ist, um beide Pfadgruppen durch a_2 und a_3 in Abwesenheit weiterer alternativer Pfade zu $\hat{z}_r$ auszuschließen. Demgegenüber sind Unterbindungen eines im Bereich a_2 ansässigen Konflikts c_2 bzw. Konflikts c_3 bzgl. des Bereichs a_3 nur geeignet, die jeweilige Pfadmenge durch selbige Bereiche auszuschließen. Es handelt sich bei a_1 und a_4 also um Schlüsselbereiche und bei c_1 und c_4 mithin um *Schlüsselkonflikte*, die vorzugsweise ausgeschlossen werden sollten, da sie die übrigen Konflikte irrelevant werden lassen.

Wie können solche Schlüsselkonflikte jedoch identifiziert werden? Da dies detaillierte Information über die diskrete Struktur des Hybriden Modells und der möglichen Pfade erfordert, scheint eine Antwort hierauf für umfangreiche diskrete Transitionsrelationen nur durch sehr aufwendige Pfadanalysemethoden ermittelbar zu sein. Für kleine diskrete Zustandsräume ist zu dieser Problematik in [FCJK05] ein Verfahren vorgeschlagen worden, welches sich auf die Berechnung von so genannten *Cut-Sets* durch Zustandssequenzen im diskreten Zustandsraum abstützt, wobei Cut-Sets hier auf konkrete Teilpfade bezogen einen Konflikt beschreiben. Dieser Ansatz läßt sich so jedoch nicht auf große Zustandsräume anwenden, da die Größe der Cut-Sets und deren Berechnung exponentiell in der Größe des Transitionsgraphen anwächst. Dieses Verfahren ist in Abschnitt 4.9.1.1 näher beschrieben. An diesem Punkt sind Backtracking-Verfahren im Vorteil, welche dies dynamisch während der Pfadsuche durchführen können, wie z.B. HySAT ([FH05]), was in Abschnitt 4.9.3 ausführlicher vorgestellt wird.

Dennoch bietet sich auch in dem hier diskutierten Verfahren eine Auswahlstrategie basierend auf drei Überlegungen an:

- Nach wahrscheinlichkeitstheoretischen Überlegungen besitzen kürzere Konflikte gegenüber längeren eine größere Wiederholungshäufigkeit im Hybriden System und minimale Konflikte daher eine höhere Allgemeingültigkeit in dem Sinne, daß sie auf mehr Pfade zutreffen als längere Konflikte.

- Da längere Konflikte hinsichtlich der benötigten Anzahl der Zustände des lernenden ω-Automaten sehr teuer sind, sollten kürzere Konflikten vorrangig

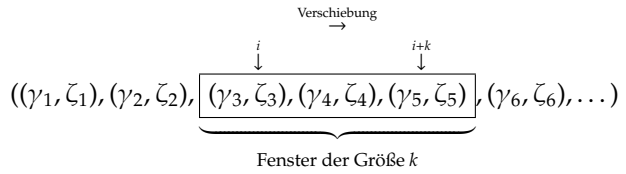
Berücksichtigung finden.

- Konflikte mit sich überschneidenden Intervallen sollte zur Berücksichtigung nur eines der beteiligten Konflikte führen, da eine erhöhte Wahrscheinlichkeit besteht, daß der Ausschluß nur eines Konflikts dazu führt daß die anderen in folgenden Iterationen nicht mehr auftreten, da zumindest der bislang einzig bekannte Pfad durch Bereiche dieser Konflikte damit ausgeschlossen ist.

Numerische Aspekte Unabhängig von dem verwendeten Verfahren zur Implementierung von $\mathcal{L}$ läßt sich bereits feststellen, daß der Berechungsaufwand und Ungenauigkeiten im Ergebnis von $\mathcal{L}$ mit der Größe der zu lösenden *GJ*-Sequenz ansteigt. Daher sollte möglichst vermieden werden, größere *GJ*-Sequenzen zu lösen bzw. deren diesbzgl. Ergebnisse der Funktion $\mathcal{L}$ zu verwenden. Es ist also aus numerischen Gründen von Vorteil, längere Konflikte solange nicht zu beachten, wie kürzere Konflikte noch ausreichend sind, eine effektive Verfeinerung des Modells zu erzielen.

Effiziente Suche durch Schlüssellochtechnik Die geschilderten Überlegungen führen zu einem Verfahren, in dem die Konfliktanalyse mit einer Schlüssellochtechnik (*Peephole*) durchgeführt wird, bei der ein Intervallfenster anwachsender Größe k über die GJ-Sequenz geschoben wird und Konflikte nur innerhalb des Fensters gesucht werden. Dabei bestimmt die Fenstergröße k die maximale Länge der gesuchten Konflikte. Das Fenster wird nach jeder Anwendung entweder um einen parametrierbaren Wert, z.b. 1 oder $k/2$ verschoben, bzw. falls Konflikte im letzten Fensterausschnitt gefunden wurden, wird der neue Fensteranfang auf das Ende des letzten Konflikts platziert.

Die Fenstergröße wird nach jeder vollständigen Durchsuchung der gesamten Sequenz entweder inkrementiert oder optional verdoppelt und der Pfad hiermit erneut untersucht, bis mindestens ein Konflikt gefunden wurde. Dies stellt sicher, daß bei Existenz kürzerer Konflikte längere nicht mehr untersucht werden. Innerhalb des Fensters ist die zu lösende Formel sehr viel kleiner und kann bedeutend schneller analysiert werden, als die Gesamtformel.



Algorithmus 2 beschreibt nach dieser Strategie die Funktion $r : C \to 2^C \times 2^C$ zur Reduzierung einer *GJ*-Sequenz auf alle initialen und invarianten minimalen Konflikte der Länge des kürzesten vorhandenen minimalen Konflikts, welche wir für den Verfeinerungsschritt verwenden wollen. Die Parametrisierung des dargestellten Algorithmus ist dabei eine Fensterverschiebung um jeweils einen Schritt (Variable i) und eine Vergrößerung des Fensters (Variable k) um einen Schritt nach jedem vollständigen Durchlauf. In dieser Parametrisierung ist der Aufwand der Analyse $O(l^3)$ mit $l = |\hat{\pi}|$, jedoch muß hinzugefügt werden, daß Konflikte sich selten auf den gesamten Pfad ausdehnen und auf wenige Schritte begrenzt sind und die Analyse nach Auffinden von Konflikten kleinstmöglicher Länge abbricht. Ist die kleinste Konfliktlänge der Konflikte eines Pfades mit k_{max} zu beziffern, so reduziert sich der Aufwand auf $O(l \cdot {k_{max}}^2)$. r

Neben der Funktion r definieren wir die Funktionen r_{ini} und r_{inv} zur selektiven Ermittlung initialer, bzw. invarianter Konflikte. r_{ini}, r_{inv}

$i := 0$
$k := 0$
$C_{ii} := \emptyset$
$C_{iv} := \emptyset$
while $C_{iv} \cup C_{ii} = \emptyset$ **do**
  while $i + k \leq n$ **do**
    if $i = 0 \wedge \mathcal{L}'\left(((\gamma_0, \zeta_0), \ldots, (\gamma_{i+k}, \zeta_{i+k})), X_0\right) = false$ **then**
      $C_{ii} := C_{ii} \cup \{((\gamma_0, \zeta_0), \ldots, (\gamma_{i+k}, \zeta_{i+k}))\}$
      $i := i + k$
    if $i > 0 \wedge \mathcal{L}'\left(((\gamma_0, \zeta_0), \ldots, (\gamma_{i+k}, \zeta_{i+k})), X\right) = false$ **then**
      $C_{iv} := C_{iv} \cup \{((\gamma_i, \zeta_i), \ldots, (\gamma_{i+k}, \zeta_{i+k}))\}$
      $i := i + k$
    $i := i + 1$
  end
  if $C_{iv} \cup C_{ii} = \emptyset$ **then** $k := k + 1, i := 0$
end
return (C_{ii}, C_{iv}) % $r_{ii} : C \to 2^C$ liefert C_{ii}, $r_{iv} : C \to 2^C$ liefert C_{iv}

Algorithmus 2: $r : C \to 2^C \times 2^C$. Berechnung reduzierter Konfliktmengen C_{ii} und C_{iv} aus einem Konflikt $c_\perp = ((\gamma_0, \zeta_0), \ldots, (\gamma_n, \zeta_n))$.

Binäre Intervallgrenzensuche Sofern Algorithmus 2 so parametrisiert wird, daß die Fenstergröße nach jedem Durchlauf um mehr als einen Schritt vergrößert wird, sind etwaige Konflikte innerhalb eines Fensters nicht mehr minimal und müssen weiter verkürzt werden. Da das Fensterintervall $I = [i, i + k]$ in Abwesenheit kürzerer Konflikte auch bis auf die Formelgröße anwachsen kann, wird auch innerhalb dieses Intervalls ein effizienteres Verfahren eingesetzt, als das naive versuchsweise Entfernen von einzelnen GJ-Paaren. An dieser Stelle wird daher eine binäre Suche nach den Intervallgrenzen des Konflikts durchgeführt. Dabei wird zunächst die Obergrenze i_{max} bestimmt, indem diese iterativ jeweils in die Mitte des verbleibend in Frage kommenden Suchraums platziert wird und abhängig vom Ergebnis der als nächstes zu untersuchende Suchraum festgelegt wird. Hernach wird das gleiche Vorgehen mit der unteren Grenze i_{min} praktiziert, so daß ein existenter Konflikt in einem Fenster der Intervallgröße k sicher in $2 \cdot \log k$ Schritten gefunden wird. Solches Vorgehen findet nicht notwendigerweise den kürzesten Konflikt, sofern mehrere innerhalb des Fensterausschnitts vorhanden sind, jedoch wird diese Zielsetzung schon in ausreichendem Maße durch die Anwendung der Fenstertechnik selbst erfüllt, da die Fenstergröße die Länge der aufzufindenden Konflikte begrenzt. Es ist zwar ungünstig, aber dennoch zweckdienlich, wenn in selteneren Fällen nur der zweitkleinste Konflikt verwendet wird, so daß hier auf Kosten der ausschließlichen Verwendung kleinster Konflikte die Effizienz der Pfadanalyse erhöht werden kann.

Damit haben wir nun ein Verfahren, um geeignete Konflikte eines ungültigen Pfades $\hat{\pi}$ zu bestimmen.

C_{ini}, C_{inv}

Verwendung der Konflikte Um zu verhindern, daß bereits gefundene initiale Konflikte $c_i \in C_{ini}$ oder invariante Konflikte $c_v \in C_{inv}$ je wieder in einem Pfad in späteren Iterationen diesen zu einem ungültigen machen, müssen diese global ausgeschlossen werden. In Betrachtung von GJ-Paaren (γ, ζ) als atomare Bezeichner

Σ

von Buchstaben eines Alphabets $\Sigma = GJ$ müssen nachfolgende Abstraktionen A

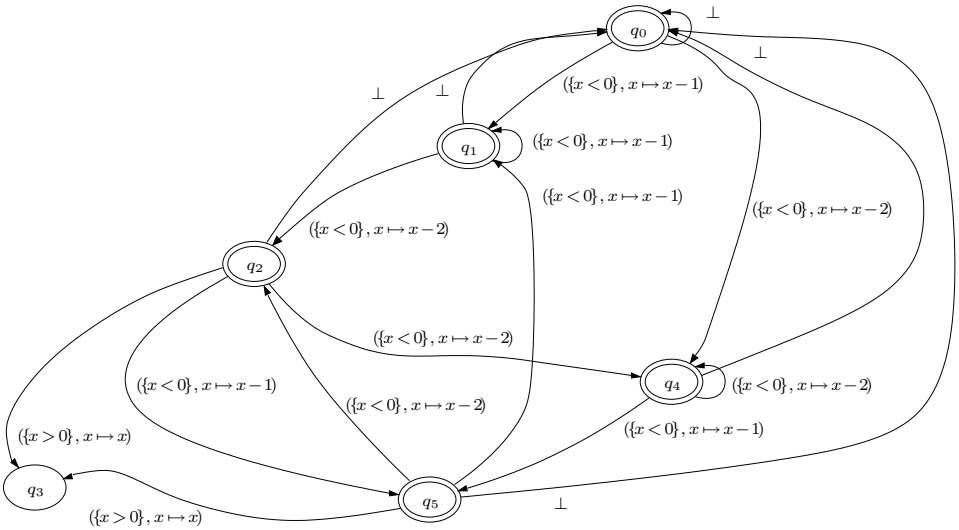


Abbildung 4.5: ω-Automat für den Ausschluss von $((\{x<0\}, x \mapsto x-1), (\{x<0\}, x \mapsto x-2), (\{x>0\}, x \mapsto x))$ und $((\{x<0\}, x \mapsto x-2), (\{x<0\}, x \mapsto x-1), (\{x>0\}, x \mapsto x))$. Bei Transitionen zu dem Startzustand q_0 repräsentiert das Zeichen $\perp$ dabei alle GJ-Paare, welche nicht durch eine explizite Transition zu einem anderen Zustand abgedeckt sind.

die folgende LTL-Formel erfüllen:

$$A \models \neg \left(\bigvee_{c_i \in C_{ini}} \lambda(c_i) \right) \wedge \neg \mathbf{F} \left(\bigvee_{c_v \in C_{inv}} \lambda(c_v) \right) \tag{4.2}$$

wobei λ die LTL-Formel $\lambda(c) = ((\gamma_1, \zeta_1) \wedge \mathbf{X}((\gamma_2, \zeta_2) \wedge \mathbf{X}(... \wedge \mathbf{X}(\gamma_n, \zeta_n))))$ für einen Konflikt $c = \langle (\gamma_n, \zeta_n) \rangle$ repräsentiert. Wie in Abbildung 4.1 auf Seite 39 skizziert wird dies durch ein Kreuzprodukt mit einem ω-Automaten A_C erreicht, für den Gleichung 4.2 gilt. Die Konstruktion des ω-Automaten basierend auf den bekannten Konflikten ist im folgenden Abschnitt beschrieben. λ

4.3.3 Konstruktion des ω-Automaten

Die Verfeinerung des abstrakten Modells A_k für Iterationsschritt $k+1$ wird durch die inkrementelle Erweiterung des ω-Automaten A_{C_k} zu $A_{C_{k+1}}$ erzielt, welcher für A_{k+1} durch Bildung des Kreuzprodukts $A_{k+1} = A_0 \times A_{C_{k+1}}$ verwendet wird, wie in Abbildung 4.1 auf Seite 39 dargestellt. In diesem Abschnitt wird beschrieben, wie ein solcher ω-Automat inkrementell und effizient konstruiert werden kann, welcher alle Läufe akzeptiert, soweit keine ihrer Teilsequenzen einem der auszuschließenden Konflikte entspricht.

Formal können die Läufe des Automaten als Interpretationen der LTL-Formeln $\neg \left(\bigvee_{c_i \in C_{ini}} \lambda(c_i) \right)$ für initiale, und $\neg \mathbf{F} \left(\bigvee_{c_v \in C_{inv}} \lambda(c_v) \right)$ für invariante Konflikte beschrieben werden, wobei die einzelnen GJ-Paare der Konfliktsequenzen als atomare Propositionen anzusehen sind.

In Abbildung 4.5 ist ein solcher Automat dargestellt, welcher die beiden Konfliktsequenzen $((\{x<0\}, x \mapsto x-1), (\{x<0\}, x \mapsto x-2), (\{x>0\}, x \mapsto x))$ und $((\{x<0\}, x \mapsto x-2), (\{x<0\}, x \mapsto x-1), (\{x>0\}, x \mapsto x))$ ausschließt, alle anderen Kombinationen jedoch erlaubt. Wir werden häufig die GJ-Paare, welche für diese Automaten die Zeichen eines Alphabets $\Sigma = GJ$ bilden, durch einfache Kleinbuch-

$a, b, c, d, ...$

staben $a, b, c, d, ...$ darstellen. Die unterschiedliche Verwendung von c als Zeichen eines Alphabets und c als Konflikt ergibt sich aus dem Kontext, oder wird explizit unterschieden. Der Automat in Abbildung 4.5 würde somit unter Verwendung von $a := (\{x < 0\}, x \mapsto x - 1)$, $b := (\{x < 0\}, x \mapsto x - 2)$ und $c := (\{x > 0\}, x \mapsto x)$ die Eigenschaft $\neg\mathbf{F}((a \wedge \mathbf{X}(b \wedge \mathbf{X}c)) \vee (b \wedge \mathbf{X}(a \wedge \mathbf{X}c)))$ erfüllen.

Da aus einer LTL-Formel ein Automat konstruiert werden soll, bieten sich hier zunächst bereits existierende Konstruktionsverfahren an wie [GPVW95] oder Weiterentwicklungen deren Algorithmus durch [DGV99] und [SB00]. Aufgrund ihrer Allgemeingültigkeit bzgl. mannigfaltig strukturierter LTL-Formeln erzeugen diese z.T. recht große Automaten und benötigen hierfür einen für unseren Zweck inakzeptablen Rechenaufwand, wie wir in Abschnitt 4.9.4 auf Seite 94 sehen werden. Dies erzwingt ein alternatives Konstruktionsverfahren für den hier vorliegenden speziellen Kontext.

Im Folgenden wird ein auf LTL-Formeln der oben genannten Struktur spezialisierter Algorithmus vorgestellt, welcher nahezu minimale, deterministische Automaten sehr effizient inkrementell konstruiert. Genauer erzeugen wir zunächst zwei Automaten:

$A_{C_\Re}, A_{C_\omega}$

- Einen regulären Automaten $A_{C_\Re}$ für initiale Konflikte C_{ini}, so daß $A_{C_\Re} \models \neg\left(\bigvee_{c \in C_{ini}} \lambda(c)\right)$. Dieser Automat akzeptiert alle endlichen Läufe, die keinen initialen Konflikt beinhalten.

- Einen ω-Automaten A_{C_ω} für invariante Konflikte C_{inv}, so daß gilt: $A_{C_\omega} \models \neg\mathbf{F}\left(\bigvee_{c \in C_{inv}} \lambda(c)\right)$. Dieser akzeptiert alle unendlichen Läufe, welche keine der invarianten Konflikte beinhaltet. Dieser Automat ist ein co-1-akzeptierender Büchi-Automat ohne Fairnessbedingungen.

Beide Automaten $A_{C_\Re}$ und A_{C_ω} werden nach ihrer Konstruktion durch Kreuzproduktbildung zu dem ω-Automat A_C zusammengefaßt, welcher ebenfalls ein co-1-akzeptierender Büchi-Automat ist.

A_C

Einen Vergleich mit den allgemeineren LTL-Formel basierten Büchi-Automatenkonstruktionsverfahren wird später in Kapitel 4.9.4 auf Seite 94 diskutiert, nachdem auch die Automaten-relevanten Erweiterungen des Verfahrens aus Kapitel 4.7.2.2 auf Seite 79 vorgestellt worden sind.

Sowohl der reguläre, als auch der ω-Automat werden durch einen ähnlichen Algorithmus konstruiert. Zunächst wird die Konstruktion des regulären Automaten $A_{C_\Re} = (Q_\Re, q_{\Re_0}, \Sigma, T_\Re, F_\Re)$ beschrieben und im Anschluss die Erweiterung des Algorithmus zur Erzeugung des ω-Automaten $A_{C_\omega} = (Q_\omega, q_{\omega_0}, \Sigma, T_\omega, F_\omega)$. Zur Definition regulärer und ω-Automaten, sowie der im folgenden Abschnitt verwendeten Läufe und der Funktion ρ siehe Abschnitt 2.1 auf Seite 11.

4.3.3.1 Regulärer Automat

Informal beschrieben funktioniert der in diesem Abschnitt beschriebene Algorithmus zur inkrementellen Konstruktion des regulären Automaten $A_{C_\Re} = (Q_\Re, q_{\Re_0}, \Sigma, T_\Re, F_\Re)$ wie folgt:

Unter Wiederverwendung von bereits vorhandenen Transitionen und Zuständen, die einen Präfix w_p der Länge k des hinzuzufügenden Wortes $w \in \Sigma^* = C$ bereits verarbeiten, werden vom Zustand $\rho_w(k)$ ausgehend, welcher wie in Kapitel 2.1 auf Seite 11 beschrieben der eingenommene Zustand eines Laufs ρ_w beim k-ten Zeichen ist, neue (akzeptierende) Finalzustände mit dazugehörigen Transitionen angehangen. Für das letzte Zeichen des Wortes w mündet die Transition in einen Terminalzustand, der kein akzeptierender Finalzustand ist und auch keine

$p \coloneqq q_0$
for $1 \leq i \leq n$ **do**
  if $\exists(p, \delta_i, q) \in T : q \neq \mathit{fin}$ **then**
    $p \coloneqq q$
  else
    $Q \coloneqq Q \cup \{q'\}$, mit q' als neuen Zustand
    $T \coloneqq (T \setminus \{(p, \delta_i, \mathit{fin})\}) \cup \{(p, \delta_i, q')\}$
    if $i \neq n$ **then**
      $F \coloneqq F \cup \{q'\}$
      $T \coloneqq T \cup \bigcup_{\delta \in \Sigma} \{(q', \delta, \mathit{fin})\}$
    else
      $T \coloneqq T \cup \bigcup_{\delta \in \Sigma} \{(q', \delta, q')\}$
    $p \coloneqq q'$
end
return $(Q, \{q_0\}, \Sigma, T, F)$

Algorithmus 3: $Add_{\mathfrak{R}} : \mathfrak{R} \times \Sigma^* \to \mathfrak{R}$. Erweiterung des Regulären Automaten $A_{C_{\mathfrak{R}}} = (Q, \{q_0\}, \Sigma, T, F)$ zum Ausschluss von Wörtern $(\delta_1, \delta_2, \ldots, \delta_n)$ mit $\delta_i \in \Sigma$ aus der akzeptierten Sprache. Die Variablen p und q sind Variablen über Q.

ausgehenden Transitionen besitzt. Der Automat akzeptiert somit jeden echten Präfix von w und nimmt bei einem vollständigen Lauf über w den Terminalknoten ein, welcher nicht mehr verlassen werden kann. Weiterhin gibt es einen terminalen Finalzustand fin, welcher immer dann eingenommen wird, wenn ein für keinen der beobachteten Wörter kritisches Zeichen gelesen wird. fin

Formal beschrieben wird dieses Vorgehen durch Algorithmus 3, welcher das Wort w zeichenweise in den Automaten einfügt. Initial wird mit dem Automaten $A_{C_{\mathfrak{R}}} = A^0_{C_{\mathfrak{R}}}$ begonnen, wobei $A^0_{C_{\mathfrak{R}}}$

$$A^0_{C_{\mathfrak{R}}} \coloneqq \left(\{q_{\mathfrak{R}_0}, \mathit{fin}\}, \{q_{\mathfrak{R}_0}\}, \Sigma, \{(q_{\mathfrak{R}_0}, \delta, \mathit{fin}), (\mathit{fin}, \delta, \mathit{fin}) \,|\, \delta \in \Sigma\}, \{q_{\mathfrak{R}_0}, \mathit{fin}\}\right)$$

Nachdem alle Wörter durch iterative Anwendung von Algorithmus 3 in $A_{C_{\mathfrak{R}}}$ berücksichtigt sind, wird eine Minimierung nach Algorithmus 4 durchgeführt, um den minimalen Automaten zu erhalten. Anschaulich ist der Automat vor der Minimierung ein Baum, bei dem beginnend bei den Blättern die Zweige soweit wie möglich iterativ zusammengelegt werden. Dies geschieht sehr effizient, da begonnen mit den nicht-finalen Terminalzuständen $Q_{\mathfrak{R}} \setminus F_{\mathfrak{R}}$, alle Zustände paarweise zusammengelegt werden, die in allen ausgehenden Transitionen übereinstimmen, wobei die Übereinstimmung sich auf Zeichen und Zielzustand beschränkt. Nichtakzeptierende Terminalzustände werden dabei alle zu einem einzigen zusammengefasst. Wo immer Zustände q_i und q_j zusammengefaßt werden konnten, indem alle eingehenden Transitionen von q_j zu q_i umgeleitet werden und q_j entfernt wird, wird mit allen direkten Vorgängerzuständen $\left\{p \,\middle|\, (p, \delta, q_i) \in T_{\mathfrak{R}}\right\}$ von q_i paarweise fortgefahren.

Für den so konstruierten Automat gilt aufgrund der Konstruktionsvorschrift: $A_{C_{\mathfrak{R}}} \models \neg\left(\bigvee_{c \in C_{ini}} \lambda(c)\right)$.

Beweis der Korrektheit der Automatenkonstruktion

Da die Minimierung des Automaten ein Standardverfahren darstellt und dessen akzeptierte Sprache nicht ändert [HU90], betrachten wir für den Korrektheitsbeweis das Konstruktionsverfahren ohne Minimierung. Vor dem Beweis der Automa-

$M \coloneqq \{Q \backslash F\}$
foreach $M_k \in M$ **do**
  foreach $q_i, q_j \in M_k, q_i \neq q_k$ **do**
    if $\forall p \in Q, \delta \in \Sigma : \exists(q_i, \delta, p) \in T \Leftrightarrow \exists(q_j, \delta, p) \in T$ **then**
      $T \coloneqq T \cup \left\{(p, \delta, q_i) \middle| \exists(p, \delta, q_j) \in T\right\}$
      $T \coloneqq T \setminus \left(\left\{(p, \delta, q_j) \in T\right\} \cup \left\{(q_j, \delta, p) \in T\right\}\right)$
      $F \coloneqq F \setminus \{q_j\}$
      $Q \coloneqq Q \setminus \{q_j\}$
      $M \coloneqq (M \setminus \{M_k\}) \cup \left\{p \middle| \exists \delta \in \Sigma : (p, \delta, q_i) \in T\right\}$
  end
end
return $(Q, \{q_0\}, \Sigma, T, F)$

Algorithmus 4: *minimize* : $\mathfrak{R} \to \mathfrak{R}$. Minimierung des Automaten $(Q, \{q_0\}, \Sigma, T, F)$

tenkonstruktion benötigen wir zunächst eine Invariante des gesamten Verfahrens, welche besagt, daß es keine Konflikte geben kann, die Teil der bereits in C_{ini} und C_{inv} berücksichtigten Konflikte sind.

Invariante 1 *Für alle zur Automatenkonstruktion verwendeten Konflikte* $(\delta_1, \delta_2, \ldots, \delta_n) \in C_{ini} \cup C_{inv}$ *gilt:*

$$\nexists j, k : 1 \leq j \leq k \leq n \wedge (j \neq 1 \vee k \neq n) \wedge (\delta_j, \ldots, \delta_k) \in C_{ini} \cup C_{inv}$$

Die Gültigkeit der Invariante folgt unmittelbar aus der Minimierung von Konflikten nach Algorithmus 2 auf Seite 46, bevor diese zur Automatenkonstruktion verwendet werden. □

Aus Invariante 1 folgt insbesondere, daß es in der Sprache des Automaten A_C keine Konflikte mehr geben kann, welche

1. echte Präfixe, Infixe oder Suffixe der bereits berücksichtigten Konflikte aus C_{inv} und C_{ini} sind, bzw.
2. einen der bereits berücksichtigten Konflikte aus C_{inv} und C_{ini} als Präfix, Infix oder Suffix beinhalten.

Für den Nachweis der Einhaltung des regulären Automaten bzgl. $A_{C_\mathfrak{R}} \models \neg\left(\bigvee_{c \in C_{ini}} \lambda(c)\right)$ formulieren wir nun mehrere, durch den Algorithmus 3 gewährleistete Schleifeninvarianten.

Invariante 2 *In Algorithmus 3 ist der bearbeitete Automat* $A_{C_\mathfrak{R}}$ *vor und nach jeder Schleifeniteration deterministisch und vollständig.*

Beweis: Der initial zu verwendende Automat $A^0_{C_\mathfrak{R}}$ ist trivial deterministisch und vollständig. In jedem Schleifendurchlauf, in dem ein neuer Zustand q' hinzugefügt wird, werden für diesen Transitionen bzgl. jedes Zeichens $\delta \in \Sigma$ zu dem Zustand *fin* bzw. dem Zustand q' selbst hinzugefügt, so daß die von q' ausgehende Transitionsmenge ebenfalls deterministisch und vollständig ist. Dem Vorgängerzustand p wird eine Transition (p, δ_i, q') hinzugefügt, welche jedoch bzgl. des gleichen Zeichens δ_i unmittelbar vorher mit der Transition (p, δ_i, fin) entfernt wurde. Somit bleibt auch die ausgehende Transitionsmenge des Zustands p deterministisch und vollständig.

Andere Zustände bzw. Transitionen sind vom Algorithmus nicht betroffen und die Aussage damit bewiesen. □

Invariante 3 *In Algorithmus 3 gilt vor und nach jeder Schleifeniteration für* $c' = (\delta_1, \delta_2, \ldots, \delta_n)$ *als hinzuzufügendem Konflikt die Bedingung:*

$$p = \rho_{c'}(i)$$

Beweis: Da der Automat schleifeninvariant deterministisch und vollständig ist, ist die Existenz eines Zustands $p = \rho_{c'}(i)$ immer gewährleistet und eindeutig. Die Bedingung gilt durch die Zuweisung $p := q_0$ vor dem ersten Schleifendurchlauf für $i = 0$. Es sind nun gemäß der Kontrollstruktur des Algorithmus zwei Fälle zu unterscheiden. Im ersten Fall wird eine etwaig vorhandene Transition von p mit der durch die Invariante garantierten Vorbedingung $p = \rho_{c'}(i-1)$ mit δ_i zu einem Zustand $q = \rho_{c'}(i)$ verwendet und p in Zeile 4 auf diesen gesetzt, so daß die Invariante wieder erfüllt ist. Im zweiten Fall wird ein Zustand q' mit entsprechender Transition $(\rho_{c'}(i-1), \delta_i, q')$ in den Zeilen 6 und 7 hinzugefügt, so daß nach der Zuweisung $p := q'$ am Ende dieses Kontrollzweigs die Einhaltung der Invariante ebenfalls gegeben ist. □

Invariante 4 *In Algorithmus 3 gilt vor und nach jeder Schleifeniteration mit* $c' = (\delta_1, \delta_2, \ldots, \delta_n)$ *als hinzuzufügendem Konflikt und* C_{ini} *als der Menge der bereits durch den Automaten* $A_{C_\Re}$ *ausgeschlossenen initialen Konflikte einschließlich* c' *die Bedingung*

$$\forall c \in C_{ini} \forall k \in \{1, \ldots, |c|\} : (c \neq c' \vee k \leq i) \implies \rho_c(k) \neq \mathit{fin}$$

Beweis: Die Invariante wird offenbar von dem initialen Automaten $A^0_{C_\Re}$ erfüllt. Der Zustand *fin* kann als Senkenzustand, als solcher durch Algorithmus 3 mangels möglichen Erweiterungen um ausgehende Transitionen unveränderbar, nicht mehr verlassen werden, so daß die Betrachtung auf Zustände $q \neq \mathit{fin}$ beschränkt werden kann. Der Algorithmus fügt in jeder Iteration höchstens einen Zustand q' mit $q' \neq \mathit{fin}$ der Zustandsmenge hinzu, wobei die Transitionsmenge ohne Beteiligung von q' unbeeinflußt bleibt, mit Ausnahme der Entfernung der Transition $(p, \delta_i, \mathit{fin})$, wodurch die Invariante trivial eingehalten wird. Es genügt somit, die ein- und ausgehenden Transitionen bzgl. q' zu betrachten, da die Invariante bzgl. aller übrigen Konflikte und Zustände unverändert Gültigkeit besitzt.

In Zeile 7 gilt gemäß Invariante 3 unter Berücksichtigung des bereits inkrementierten Zählers i unter Beibehaltung von p die Bedingung $p = \rho_{c'}(i-1)$. Somit bleibt die Invariante durch die hinzugefügte Transition (p, δ_i, q') für $k = i$ im Fall $c = c'$ erhalten, da wir durch diese $\rho_{c'}(k) = q'$ erhalten. Durch die für alle Zeichen des Alphabets Σ in Zeile 10 hinzugefügten Transitionen von q' nach *fin* wird die Invariante trivial erfüllt, da die Prämisse der Implikation für $i > k$ bei $c = c'$ nicht erfüllt ist. Da es keine weiteren, durch Algorithmus 4 möglicherweise hinzugefügten Transitionen mit dem Zielzustand *fin* gibt, ist die Invariante somit immer gültig. □

Invariante 5 *In Algorithmus 3 gilt vor und nach jeder Schleifeniteration die Bedingung:*

$$\forall q \in Q \forall \delta \in \Sigma : \left(\nexists c \in C_{ini} \nexists k \in \{1, \ldots, |c|\} : q = \rho_c(k-1) \wedge \delta = c_{(k)}\right) \implies (q, \delta, \mathit{fin}) \in T$$

Beweis: Die Invariante wird offenbar von dem initialen Automaten $A^0_{C_\Re}$ erfüllt. Algorithmus 3 fügt mit den Zeilen 6 und 7 in der i-ten Iteration für das Zeichen $\delta = \delta_i = c_{(i)}$ eine Transition (p, δ, q') vom durch Invariante 3 beschriebenen Vorgängerzustand $p = \rho_c(i-1)$ zu einem neuen Nachfolgezustand $q' = \rho_c(i)$ für den Konflikt $c \in C_{ini}$ hinzu, bei gleichzeitiger Entfernung der vorher existenten Transition $(p, \delta, \mathit{fin})$. Durch die Entfernung der Transition wird somit obige Invariante aufgrund der Existenz von c nicht verletzt. Desweiteren werden in Zeile 10 für den neuen Zustand q' und für jedes Zeichen $\delta \in \Sigma$ Transitionen $(q', \delta, \mathit{fin})$ hinzugefügt, durch welche die Invariante trivial erfüllt bleibt. Weiterhin macht die Invariante keine Aussage über den Zustand $\rho_c(k)$ mit $k = |c|$, so daß für die in Zeile 12 hinzugefügte Transition diesbzgl. keine Relevanz besteht. Damit erfüllen alle im Schleifenrumpf potentiell durchgeführten Modifikationen die Invariante. □

Nach dem Nachweis der benötigten Invarianten gilt es nun für die Korrektheit der Automatenkonstruktion für den regulären Automaten nachzuweisen, daß vom Initialzustand $q_{\Re_0}$ aus

1. jede endliche Sequenz $v_c = c \cdot v$ mit $v \in \Sigma^*$ als beliebigem endlichen Suffix, die eine Konfliktsequenz $c = (\delta_1, \delta_2, \ldots, \delta_n)$ als Präfix beinhaltet, in einen nicht-akzeptierenden Senkenzustand $\rho_{v_c}(|v_c|) \notin F, \nexists \delta \in \Sigma, q \in Q : (\rho_{v_c}(|v_c|), \delta, q) \in T$ führt und

2. jede beliebig lange Sequenz $v = (\delta'_1, \delta'_2, \ldots, \delta'_j)$, die keinen Konflikt als Präfix beinhaltet, akzeptiert wird, d.h. $\rho_v(j) \in F$.

Nachweis der Nicht-Akzeptanz von Konflikt-beinhaltenden Sequenzen (1): Nach Invariante 4 gilt für einen beliebigen Konflikt $c \in C_{ini}$ der Länge n am Ende von Algorithmus 3 $q_n = \rho_c(n) \neq \mathit{fin}$. Darüberhinaus gilt aufgrund der in der Zeile 8 erfolgenden Fallunterscheidung für den letzten, im Rahmen der Erweiterung für einen Konflikt c hinzugefügten Zustand $q_n \notin F$ und (q_n, δ, q_n) für alle $\delta \in \Sigma$, d.h. q_n ist ein nicht akzeptierender Senkenzustand und kann nicht mehr verlassen werden. □

Nachweis der Akzeptanz von Konflikt-freien Sequenzen (2): Der Nachweis erfolgt unmittelbar aus Invariante 5, da schließlich, nachdem die Sequenz v von jedem möglichen Konflikt $c \in C_{ini}$ abweicht, nur noch Transitionen zu dem Zustand fin existieren. Dieser ist ein Senkenzustand, da in keinem Schritt des Algorithmus eine ausgehende Transition zu einem anderen Zustand hinzugefügt wird. Damit gilt $\forall c \in C_{ini} \forall j : max(|c|) \leq j \leq |v| : \rho_v(j) = \mathit{fin}$. Da fin darüberhinaus mit $\mathit{fin} \in F$ akzeptierend ist, wird auch v akzeptiert. □

4.3.3.2 ω-Automat

Ähnlich wie bei dem regulären Automaten besteht die Idee des Konstruktionsalgorithmus für den ω-Automat $A_{C_\omega} = (Q_\omega, q_{\omega_0}, \Sigma, T_\omega, F_\omega)$ in der im Bedarfsfalle inkrementellen Hinzufügung von Zuständen und Transitionen, um unendliche Läufe auszuschließen, welche das hinzuzufügende Wort als Teil eines endlichen Präfixes beinhalten (co-1-Akzeptanz). Initial wird mit dem Automaten $A_{C_\omega} = A^0_{C_\omega}$ begonnen, wobei

$A^0_{C_\omega}$

$$A^0_{C_\omega} := (\{q_{\omega_0}\}, \{q_{\omega_0}\}, \Sigma, \{(q_{\omega_0}, \delta, q_{\omega_0}) | \delta \in \Sigma\}, \{q_{\omega_0}\})$$

Im Gegensatz zu dem regulären Automaten gibt es hier keinen Zustand *fin*, welcher den Automaten gewissermaßen „entschärft". Wann immer der reguläre Automat für ein beobachtetes Wort $w = (\delta_1, \delta_2, \ldots, \delta_j, \delta_{j+1}, \ldots, \delta_{k-1}, \delta_k, \ldots)$ im Zustand $\rho_w(k-1)$ aufgrund eines gelesenen Zeichens $\delta'_k \neq \delta_k$ in den Zustand *fin* gewechselt ist, muß der ω-Automat differenzierter reagieren:

- Falls es kein mit δ'_k beginnendes Wort gibt, d.h. $\nexists(\delta'_k, \delta'_{k+1}, \delta'_{k+2} \ldots) \in C_{inv}$, müssen wir in den initialen Default-Zustand q_{ω_0} wechseln, von dem aus alle Sequenzen aus C_{inv} wieder korrekt behandelt werden.

- Falls es ein Wort $v = (\delta_{j+1}, \ldots, \delta_k, \ldots) \in C_{inv}$ mit einer Teilpräfixübereinstimmung in $(\delta_{j+1}, \ldots, \delta_k)$ gibt, muß stattdessen der Zustand $\rho_v(k-j)$ eingenommen werden, da dieser Präfix in dem Gesamtlauf ja bereits zu beobachten war und entsprechend die Teilsequenz v im Falle des Einlesens des restlichen Teils von v korrekterweise in einem nicht-aktzeptierenden Terminalzustand enden muß.

Da immer $\rho(0) = q_{\omega_0}$ gilt, ist der erste Fall tatsächlich nur ein Spezialfall des zweiten mit einer Präfixübereinstimmung über 0 Zeichen.

Aufgrund dieser Überlegung muß der ω-Automat eine Vielzahl zusätzlicher Transitionen beinhalten, die wir *Zwischenkonfliktransitionen* nennen wollen, um obiger Bedingung zu genügen. Während der Konstruktion beinhaltet der Automat neben diesen aus Effizienzgründen noch *Hilfstransitionen*, welche im Unterschied dazu auch dann vorhanden sind, wenn das als nächstes zu lesende Zeichen $\delta'_k = \delta_k$ ist, entsprechend also in den Zustand $\rho_w(k)$ gewechselt werden muß. Diese Hilfstransitionen führen temporär zu einem nicht-deterministischen Automaten und werden auf Basis einer Prioritätenvergabe später wieder entfernt. Der Nicht-Determinismus ergibt sich aus dem Umstand, daß es für einen Zustand q durch Einführung der Hilfstransitionen mehrere Zielzustände geben kann, welche mit einem Zeichen δ gemäß der Transitionsrelation Nachfolgerzustände darstellen.

Dieser durch Hilfstransitionen verursachte Nichtdeterminismus erfordert bereits während des Konstruktionsverfahrens des ω-Automaten ebenso wie bei der abschliessenden Entfernung von Hilfstransitionen einen Auflösungsmechanismus. Weiterhin wird für alle Formalisierungen die Definition eines deterministischen Laufs $\rho^*_w : \{i \,|\, 0 \leq i \leq n\} \to Q$ in Erweiterung der Definition eines Laufs ρ benötigt, da nach Definition ein Lauf ρ für nicht-deterministische Automaten ebenfalls nicht-deterministisch und damit eine Relation wäre. Im Folgenden werden alle hierfür benötigten Relationen und Funktionen vorgestellt.

Wir definieren zunächst eine partielle Ordnung $<\subseteq Q_\omega \times Q_\omega$ auf den Zuständen mit Hilfe der minimalen Distanz zum initialen Zustand.

Definition 21 (Minimale Distanz) *Die* minimale Distanz *eines Zustands q zum Initialzustand q_{ω_0} ist definiert durch eine Funktion $d : Q_\omega \to \mathbb{N}$ definiert als*

$$d(q) = \begin{cases} 0 & gdw\ q = q_{\omega_0} \\ 1 + min\left(\left\{d(p) \,\middle|\, p \in Q \wedge \exists \delta \in \Sigma : (p, \delta, q) \in T\right\}\right) & gdw\ q \neq q_{\omega_0} \end{cases}$$

Die Distanz läßt sich sehr effizient berechnen, wenn die Implementierung die minimalen Distanzen explizit für jeden Zustand vorhält. Dann kann diese für jeden neuen Zustand, bzw. bei Hinzufügen oder Eliminieren von Transitionen vollständig lokal berechnet werden, unter Berücksichtigung der Auswirkung auf die ebenfalls zu aktualisierenden minimalen Distanzen der in der Transitionsrelation nachfolgenden Zustände.

Definition 22 (Partielle Ordnung) *Für zwei Zustände q_1 und q_2 aus Q_ω gilt in der Relation $<\subseteq Q_\omega \times Q_\omega$ genau dann $q_1 < q_2$, wenn $d(q_1) < d(q_2)$. Analog dazu sind die Relationsoperatoren $\leq, >, \geq$ definiert.*

Basierend auf der minimalen Distanz definieren wir desweiteren eine Funktion *tgt* zur Auflösung des durch die Hilfstransitionen verursachten möglichen Nichtdeterminismus:

Definition 23 (Priorisierter Zielzustand) *Die Funktion $tgt : Q_\omega \times \Sigma \to Q_\omega$ ermittelt einen Zielzustand maximaler Distanz zu dem Initialzustand q_{ω_0} mit*

$$tgt(p, \delta) = q \text{ mit } (p, \delta, q) \in T_\omega \wedge \nexists (p, \delta, q') \in T_\omega : q < q'$$

Für den hier vorgestellten Algorithmus ist dieser von der Funktion *tgt* ermittelte Zustand immer existent und eindeutig, wie wir mit der Vollständigkeit des Automaten und der immer gegebenen Existenz einer solchen Transition mit diesen Eigenschaften in den Invarianten 7 und 9 zeigen werden. Für einen Automaten, dessen Nicht-Determinismus durch die Funktion *tgt* aufgelöst werden kann, definieren wir nun einen *deterministischen Lauf.*

Definition 24 (Deterministischer Lauf) *Ein* deterministischer Lauf *eines Automaten bzgl. eines endliches Wortes $w = (\delta_1, \delta_2, \ldots, \delta_n) \in \Sigma^n \subseteq \Sigma^*$ ist eine Abbildung $\rho^*_w : \{i \,|\, 0 \leq i \leq n\} \to Q$, so daß* ρ^*_w

- *der erste Zustand ein Initialzustand ist, d.h. $\rho^*_w(0) \in Q_0$, und*
- *für aufeinanderfolgende Zustände gilt: $\rho^*_w(i+1) = tgt(\rho^*_w(i), \delta_{i+1})$.*

Mit Hilfe der Funktion *tgt* können wir nun Zwischenkonflikt- und Hilfstransitionen definieren. Eine Transition (p, δ, q) bezeichnen wir genau dann als *Zwischenkonflikttransition*, wenn $q = tgt(p, \delta) \wedge q \leq p$ gilt. Gilt hingegen $q \neq tgt(p, \delta)$, so bezeichnen wir diese als *Hilfstransition*.

Mit diesen Definitionen können wir nun mit Algorithmus 5 das Konstruktionsverfahren für den ω-Automaten beschreiben. Dieses gliedert sich in drei Schritte:

1. Iteratives Hinzufügen aller auszuschließender Teilsequenzen
2. Entfernung von Hilfstransitionen
3. Minimierung des Automaten

Die Strategie ist ähnlich zur Konstruktion des regulären Automaten: Das Wort w wird zeichenweise vom Algorithmus bearbeitet, existierende Zustände und Transitionen, die einen Präfix von w bereits verarbeiten, werden unverändert beibehalten. Sobald jedoch im Lauf $\rho^*_w(k-1) \not< \rho^*_w(k)$ gilt, handelt es sich um eine Zwischenkonflikttransition zwischen den Zuständen $\rho^*_w(k-1)$ und $\rho^*_w(k)$. $\rho^*_w(k)$ „kümmert" sich anschaulich gesprochen nur um alle Wörter, die einen Präfix der Länge kleiner gleich $d(\rho^*_w(k))$ haben. Um den mit der Transition $((\rho^*(k-1), \delta_k, \rho^*(k))$ einhergehenden Informationsverlust des mit $d\left(\rho^*_w(k-1)\right) \geq d\left(\rho^*_w(k)\right)$ längeren, von w gelesenen Präfix zu verhindern, wird an dieser Stelle ein neuer Zustand mit entsprechender Transition hinzugefügt. Die Zwischenkonflikttransition wird damit automatisch zur Hilfstransition und mithin später entfernt.

Konkret wird in Algorithmus 5 nun also die Funktion *tgt* zur Ermittlung des nächsten Zustands benutzt, um Hilfstransitionen zu ignorieren. Die Hilfstransitionen bekommen jedoch eine Schlüsselrolle bei der effizienten Bestimmung der Zustandsmengen H_p und $L_{q'}$. Die Bedeutung der Mengen ist wie folgt: $H_p, L_{q'}$

$p \coloneqq q_0$
for $1 \leq i \leq n$ **do**
  $q \coloneqq tgt(p, \delta_i)$
  if $p < q$ **then**
    $p \coloneqq q$
  else
    $Q \coloneqq Q \cup \{q'\}$, mit q' als neuem Zustand
    $H_p \coloneqq \left\{ h \,\middle|\, h \in F \wedge p < h \wedge \exists p' \in Q, \delta \in \Sigma : \{(p', \delta, p), (p', \delta, h)\} \subseteq T \right\}$
    if $i \neq n$ **then**
      $L_{q'} \coloneqq \left\{ l \,\middle|\, (p, \delta_i, l) \in T \right\}$
      $F \coloneqq F \cup \{q'\}$
    else
      $L_{q'} \coloneqq \emptyset$
      $T \coloneqq T \cup \bigcup_{\delta \in \Sigma} \{(q', \delta, q')\}$
    $T \coloneqq T \cup \bigcup_{h \in H_p \cup \{p\}} \{(h, \delta_i, q')\} \cup \bigcup_{l \in L_{q'}} \{(q', \delta, r) \,|\, \exists (l, \delta, r) \in T\}$
    $p \coloneqq q'$
end
return $(Q, \{q_0\}, \Sigma, T, F)$

Algorithmus 5: $Add_\omega : \mathfrak{B} \times \Sigma^* \to \mathfrak{B}$. Erweiterung des ω-Automaten $A_{C_\omega} = (Q, \{q_0\}, \Sigma, T, F)$ zum Ausschluss von Teilwörtern $(\delta_1, \delta_2, \ldots, \delta_n)$ aus der akzeptierten Sprache. Mit $\mathfrak{B}$ ist wie in Abschnitt 2.1 die Menge der Büchi-Automaten bezeichnet

- In $H_p \subseteq Q_\omega$ sind alle Zustände, welche einen Präfix haben, in dem der Präfix von p ein Suffix ist. Dies ist ersichtlich aus den Transitionen (p', δ, p) in der Berechnung von H_p, welche Zwischenkonflikt- und Hilfstransitionen darstellen. Somit ist eine Transition (h, δ_i, q') von jedem p' folgenden Nachfolgezustand $h \in H_p$ zu q' notwendig, um den kürzeren, durch p „observierten" Konflikt ebenfalls nachverfolgen zu können. Diese Transitionen werden automatisch von Zwischenkonflikt- zu Hilfstransitionen, wenn eine höher priorisierte Transition vorhanden sein sollte. Eine eingehendere Betrachtung dieser Zusammenhänge wird im Beweis zu Invariante 15 durchgeführt werden.
- In $L_{q'} \subseteq Q_\omega$ sind alle Zustände, deren Präfix ein Suffix im Präfix von q' darstellt, was hier ebenfalls durch die bereits existenten, von p ausgehenden Zwischenkonflikt- und Hilfstransitionen erkennbar ist. Somit ist hier analog zu dem vorangehenden Fall eine Transition (q', δ, r) von q' zu jedem Nachfolgerzustand r aller Zustände aus $L_{q'}$ notwendig, so daß die ausgehenden Transitionen des neuen Zustands q' ebenfalls zu anderen Zuständen führen, welche kürzere Konflikte mit partieller Übereinstimmung des Präfixes mit dem bisherigen Suffix des hinzuzufügenden Konflikts „bearbeiten".

Nachdem alle Wörter dem Automaten hinzugefügt worden sind, müssen die Hilfstransitionen entfernt werden durch eine Funktion $strip : \mathfrak{B} \to \mathfrak{B}$, welche nur die in der tgt-Funktion priorisierten Transitionen beibehält, d.h. $T_\omega \coloneqq \left\{ (p, \delta, q) \in T_\omega \,\middle|\, q = tgt(p, \delta) \right\}$. *strip*

Abschließend wird der Automat wieder mit Algorithmus 4 minimiert. Da reguläre und ω-Automaten gleicher syntaktischer Struktur sind und der Unterschied in der Semantik für den Minimierungsalgorithmus irrelevant ist, kann selbiger hier wiederverwendet werden. Durch die Minimierung kann der Automat trotz hinzugefügter Wörter sich wieder verkleinern. Die Größe des Automaten ist also nicht notwendigerweise mit der Anzahl der Wörter monoton anwachsend. Dieser

Umstand wird in Abschnitt 4.7.1 im Rahmen einer Erweiterung des Minimierungsverfahrens ausgenutzt.

Für den so konstruierten Automat $A_{C_\omega} = (Q, \{q_0\}, \Sigma, T, F)$ gilt aufgrund der Konstruktionsvorschrift: $A_{C_\omega} \models \neg\mathbf{F}\left(\bigvee_{\mathbf{c}\in\mathbf{C_{inv}}} \lambda(\mathbf{c})\right)$

Beweis der Korrektheit der Automatenkonstruktion

Zunächst benötigen wir erneut eine Menge von größtenteils induktiv beweisbaren Schleifeninvarianten zu Algorithmus 5. Die Beweise der Invarianten stützen sich dabei nicht nur auf die eigene Induktionsbehauptung ab, sondern z.T. auch auf die Induktionsbehauptungen anderer Invarianten, welche aufgrund zirkulärer Abhängigkeiten der Induktionsschrittbeweise in keiner konsistenten Reihenfolge präsentiert werden können. Da sich die vorgezogene Verwendung noch zu beweisender Invarianten ausschließlich auf die Bezugnahme zum vorherigen Schleifendurchlauf beschränkt, sind die Beweise dennoch induktiv abgesichert. Eine entsprechende Verschränkung der Beweisführung für die Sicherstellung aller Invarianten für die vorherige Iteration wäre möglich, eine strikte Trennung der Invarianten und deren Nachweis erleichtert die Verständlichkeit für den Leser jedoch beträchtlich.

Invariante 6 *In Algorithmus 5 sind alle Zustände des Automaten A_{C_ω} nach Durchlauf jeder Schleifeniteration erreichbar.*

Beweis: Bzgl. des anfänglich zu verwendenen Automaten $A^0_{C_\omega}$ ist die Erreichbarkeit des einzigen initialen Zustands q_{ω_0} trivial. In Abwesenheit möglicher Entfernungen von Transitionen genügt es zu zeigen, daß es mindestens eine Transition von einem bereits erreichbaren Zustand zu dem einzigen, in einem Schleifendurchlauf hinzugefügten Zustand q' gibt. Dies ist in Zeile 15 des Algorithmus für die Transition (p, δ_i, q'), welche sich in der Menge $\bigcup_{h\in H_p}\{(h, \delta_i, q')\}$ verbirgt, zutreffend, da p bereits vor erneuter Ausführung der Schleife existierte. □

Invariante 7 *In Algorithmus 5 ist der bearbeitete Automat A_{C_ω} schleifeninvariant vollständig.*

Beweis: Der initial zu verwendende Automat $A^0_{C_\omega}$ ist vollständig. Da ausschließlich Transitionen hinzugefügt und nicht entfernt werden, genügt der Vollständigkeitsnachweis für die ausgehenden Transitionen neu hinzugefügter Zustände. Die Zustandsmenge wird maximal um einen Zustand q' erweitert, wobei eine bzgl. des Alphabets Σ vollständige ausgehende Transitionsmenge durch die Transitionsmengenerweiterung $\bigcup_{l\in L_{q'}}\{(q', \delta, r)\,|\exists(l, \delta, r) \in T\}$ in Zeile 15 gewährleistet ist. Da die Zustände $l \in L_{q'}$ gemäß dieser Invariante bereits eine vollständige Menge ausgehender Transitionen aufweisen, genügt $L_{q'} \neq \emptyset$, um diese Transitionsmenge auf q' zu übertragen. Dies ist für den Fall $i \neq n$ aufgrund der ebenfalls vollständigen, vom Zustand p ausgehenden Transitionsmenge, aus der sich in Zeile 10 die Menge $L_{q'}$ errechnet, gegeben. Für den Fall $i = n$ wird in Zeile 14 die Vollständigkeit trivial gewährleistet. □

Invariante 8 *In Algorithmus 5 gilt die Schleifeninvariante*

$$\nexists q_1, q_2 \in Q : q_1 \neq q_2 \wedge d(q_1) = d(q_2) \wedge \exists\delta \in \Sigma, q \in Q\{(q, \delta, q_1), (q, \delta, q_2)\} \subseteq T$$

Beweis: Wir führen einen Widerspruchsbeweis durch. Angenommen, es gäbe zwei Zustände q' und q'', so daß $d(q_1) = d(q_2) \wedge \exists \delta \in \Sigma, q \in Q\{(q, \delta, q_1), (q, \delta, q_2)\} \subseteq T$. Nach Invariante 15 bedeutet dies, daß neben der Existenz eines Konflikts v mit $v = (\iota_1, \ldots, \iota_{k-1}, \ldots) \in C_{inv}$ gilt $\exists v' = (\iota_{j'+1} \ldots, \iota_k, \ldots), v'' = (\iota_{j''+1} \ldots, \iota_k, \ldots) \in C_{inv}$ mit $j', j'' \in \{1, \ldots, k\}$, so daß $q_1 = \rho^*_{v'}(k - j')$ und $q_2 = \rho^*_{v''}(k - j'')$. In Vorgriff auf Invariante 12 gilt jedoch $d(q_1) = d(\rho^*_{v'}(k - j')) = k - j'$, sowie $d(q_2) = d(\rho^*_{v''}(k - j'')) = k - j''$, und damit gilt aufgrund von $d(q_1) = d(q_2)$ auch $j' = j''$, d.h. die Präfixe der Wörter v' und v'' sind bis zum $(k - j')$-ten Zeichen identisch und stimmen beide mit dem Infix von v überein. Damit gilt $q_1 = q_2$ aufgrund des Determinismus des Automaten nach Invariante 9, d.h. obige Bedingung ist immer erfüllt. □

Invariante 9 *In Algorithmus 5 ist der bearbeitete Automat A_{C_ω} schleifeninvariant deterministisch unter Anwendung der Funktion tgt.*

Beweis: Diese Invariante ergibt sich unmittelbar aus Invariante 8 und der Definition der Funktion *tgt*, da paarweise Ungleichheit aller Distanzen der Zielzustände eine totale Ordnung auf den zugehörigen Transitionen induzieren. □

Invariante 10 *In Algorithmus 5 gilt vor und nach jeder Schleifeniteration für $c' = (\delta_1, \delta_2, \ldots, \delta_n)$ als hinzuzufügendem Konflikt die Bedingung:*

$$p = \rho^*_{c'}(i)$$

Beweis: Der einfache Beweis erfolgt analog zu dem in Invariante 3 auf Seite 51 über die Vorbedingung und der Fallunterscheidungsstruktur des Algorithmus. □

Invariante 11 *In Algorithmus 5 gilt die Schleifeninvariante*

$$\forall q \in Q \setminus F \forall \delta \in \Sigma : (q, \delta, q) \in T$$

Beweis: Diese Invariante ist unmittelbar aus der innersten Fallunterscheidung zwischen den Zeilen 9 und 14 ersichtlich, da hier ein Zustand entweder der Menge F hinzugefügt wird, oder die Zustandsmenge entsprechend erweitert wird, bei gleichzeitigem Verhindern von weiteren ausgehenden Transitionen von q' durch $L_{q'} = \emptyset$. Darüberhinaus ist in diesem Fall mit $i = n$ die letzte Iteration erreicht. Bei erneuter Anwendung des Algorithmus ist $p \notin F$ ausgeschlossen, da andernfalls Invariante 1 verletzt wäre. Somit können nachfolgende Anwendungen des Algorithmus keine Transitionen zu nicht-akzeptierenden Zuständen hinzufügen, soweit sie diese Zustände nicht selbst neu erzeugen. □

Invariante 12 *In Algorithmus 5 gilt vor und nach jeder Schleifeniteration die Bedingung:*

$$\forall c \in C_{inv} \forall k \in \{1, \ldots, |c|\} : (c \neq c' \vee k \leq i) \implies d(\rho^*_c(k)) = k$$

Beweis: Die Invariante ist für den anfänglichen Automaten $A^0_{C_\omega}$ in Abwesenheit von berücksichtigten Konflikten trivial erfüllt. Für den Induktionsbeweis dieser Schleifeninvariante betrachten wir die Menge aller potentiellen Änderungen der Transitionsrelation hinsichtlich deren Auswirkung. In Zeile 15 des Algorithmus 5 wird die Transitionsmenge um die folgenden Teilmengen erweitert, welche nachfolgend untersucht werden:

- $\{(p, \delta_i, q')\}$
- $\bigcup_{h \in H_p} \{(h, \delta_i, q')\}$
- $\bigcup_{l \in L_{q'}} \{(q', \delta, r) \,|\, \exists (l, \delta, r) \in T\}$

Die Transition (p, δ_i, q') führt unter Nichbeachtung der übrigen hinzukommenden Transitionen gemäß der Definition 21 mit $p = \rho_c^*(i-1)$ nach Invariante 10 der vorherigen Iteration mit $d(\rho_c^*(i-1)) = i-1$ trivial zu $d(q') = d(\rho_c^*(i)) = i$, so daß die Invariante hierdurch eingehalten wird.

Für die Transitionsmenge $\bigcup_{h \in H_p} \{(h, \delta_i, q')\}$ gilt aufgrund der Berechnung von H_p die Bedingung $p < h$ und somit wegen $d(q') = d(p) + 1$ auch $d(q') \leq d(h)$. Diese Transitionen beeinflussen $d(q')$ damit nicht, ebensowenig wie $d(h)$ aufgrund der Verwendung von h als Quellzustand.

Da aufgrund $q = tgt(p, \delta_i)$ und $p \geq q$ im Fallunterscheidungszweig ab Zeile 6 für alle $l \in L_{q'}$ die Aussage $d(l) \leq d(p)$ gilt, sowie aufgrund der Existenz der Transitionen (l, δ, r) ebenfalls $d(r) \leq d(l)+1$, so muß wegen $d(q') = d(p)+1$ auch gelten $d(r) \leq d(q')$. Damit kann die Distanz $d(r)$ der Zielzustände r zum Initialzustand q_{ω_0} in der Transitionsmenge $\bigcup_{l \in L_{q'}} \{(q', \delta, r) \,|\, \exists (l, \delta, r) \in T\}$ nicht durch selbige Transitionen verkürzt werden.

Die kürzesten Distanzen aller Zustände werden somit beibehalten, während die des neuen Zustands q' sich gemäß der Invariante erwartungsgemäß ergibt. □

Invariante 13 *In Algorithmus 5 gilt vor und nach jeder Schleifeniteration mit* $c' = (\delta_1, \delta_2, \ldots, \delta_n)$ *als hinzuzufügendem Konflikt und* C_{inv} *als der Menge der bereits durch den Automaten* A_{C_ω} *ausgeschlossenen invarianten Konflikte einschließlich* c' *die Bedingung:*

$$\forall c \in C_{inv} : (c \neq c' \vee i = |c'|) \implies \rho_c^*(|c|) \notin F$$

Beweis: Der anfängliche Automat $A^0_{C_\omega}$ erfüllt diese Invariante trivial. Wir betrachten für den Induktionsbeweis dieser Invariante erneut die im vorigen Beweis bereits aufgeführten, von Algorithmus 5 potentiell vorgenommenen Erweiterungen der Transitionsrelation in einem Schleifendurchlauf.

Aufgrund der vorangehenden Anweisungen in Algorithmus 5 ist an der Stelle der Erweiterung des Automaten um die Transition (p, δ_i, q') in Zeile 15 gewährleistet, daß $\nexists q \in Q : (p, \delta_i, q) \in T \wedge p < q$, da andernfalls der andere Anweisungsblock in Zeile 5 ausgeführt worden wäre und diese Transition mithin nicht hinzugefügt würde. Nach Invariante 12 kann es damit keine Transition $(p, \delta_i, q), q \neq q'$ geben, welche für den Lauf bzgl. irgendeines Konflikts benötigt würde. Entsprechend bleibt die Transitionsrelation für alle Läufe bei Einlesen von Konflikten $c \in C_{inv}, c \neq c'$ unbeeinträchtigt und damit insbesondere auch die Zielzustände $\rho_c^*(|c|)$ mitsamt deren Eigenschaft $\rho_c^*(|c|) \notin F$.

Wie im vorigen Beweis bereits nachgewiesen, gilt für alle Transitionen der Menge $\bigcup_{h \in H_p} \{(h, \delta_i, q')\}$ die Bedingung $d(q') \leq d(h)$, was diese als Zwischen- und Hilfstransitionen charakterisiert. Als solche können sie keine durch die Funktion *tgt* priorisierten Transitionen „verdrängen" und beeinflussen damit die Läufe $\rho_c^*(k)$ mit $1 \leq k \leq |c|$ nicht, welche sich gemäß Invariante 12 durchgehend auf priorisierte Transitionen zu Zuständen inkrementeller Distanz zu q_{ω_0} abstützen. Somit kann diese Transitionsmenge den Wahrheitsgehalt der Invariante nicht beeinflussen.

Die Transitionsmenge $\bigcup_{l \in L_{q'}} \{(q', \delta, r) \,|\, \exists (l, \delta, r) \in T\}$ beinhaltet ausschließlich von q' ausgehende Transitionen, welche aufgrund der offensichtlichen Irrelevanz des neuen Zustands q' für die Läufe bzgl. bestehender Konflikte $c \neq c'$ für die Invariante keine Auswirkungen hat. □

Invariante 14 *In Algorithmus 5 gilt die Schleifeninvariante:*

$$\forall q \in Q : ((\exists (q, \delta, p) \in T : q \neq p) \implies q \in F)$$

Beweis: Die Invariante ist für q_{ω_0} trivial erfüllt. Für jeden neu hinzugefügten Zustand q' ist die Invariante ebenfalls erfüllt, da entweder q' als nicht akzeptierender Senkenzustand hinzugefügt wird und diese Eigenschaft nach Invariante 11 beibehält, oder aber zu F hinzugefügt wird und in diesem Fall über die Hilfsmenge $L_{q'}$ eine Menge von ausgehenden Transitionen der Zustände $l \in L_{q'}$ übernimmt, welche nicht zu q' führen können, da q' bei Hinzufügung dieser Transitionen (l, δ, r) noch nicht existent war und damit $r \neq q'$. Die Menge $L_{q'}$ kann nicht leer sein, da andernfalls Invariante 7 verletzt wäre, denn es gibt neben der Konstruktion über die Hilfsmenge $L_{q'}$ keine andere Möglichkeit, von q' ausgehende Transitionen zu konstruieren. □

Invariante 15 *In Algorithmus 5 gilt die Schleifeninvariante mit $c' = (\delta_1, \delta_2, \ldots, \delta_n)$ als hinzuzufügendem Konflikt und C_{inv} als der Menge der bereits durch den Automaten A_{C_ω} ausgeschlossenen invarianten Konflikte einschließlich c'*

$$\begin{gathered} v, v' \in C_{inv} \wedge (v \neq c' \vee k \leq i) \wedge v = (\iota_1, \ldots, \iota_j, \ldots, \iota_k, \ldots) \wedge v' = (\iota_{j+1}, \ldots, \iota_{k-1}, \delta, \ldots) \\ \Longleftrightarrow \\ \left(\rho_v^*(k-1), \delta, \rho_{v'}^*(k-j)\right) \in T_\omega \wedge d\left(\rho_v^*(k-1)\right) = k-1 \wedge d\left(\rho_{v'}^*(k-j)\right) = k-j \end{gathered}$$

Beweis: Die Invariante gilt für $A^0_{C_\omega}$ trivial, da es hier noch keine berücksichtigten Konflikte gibt. Wir betrachten für den Beweis des Induktionsschritts erneut die bereits aufgeführten potentiellen Erweiterungen der Transitionsmenge eines Schleifendurchlaufs unter Berücksichtigung des hinzuzufügenden Konflikts c'.

Die Transition (p, δ_i, q') ist legitimiert durch den Fall $v = v' = c'$, $j = 0$, wodurch sich die Invariante zu der einfachen Aussage $\exists v \in C_{inv} : v = (\delta_1, \ldots, \delta_k, \ldots) \Longleftrightarrow (\rho_v^*(k-1), \delta_k, \rho_v^*(k)) \in T_\omega \wedge d\left(\rho_v^*(k-1)\right) = k-1 \wedge d\left(\rho_v^*(k)\right) = k$ vereinfacht. Die Existenz eines Konflikts v ist trivial durch den hinzuzufügenden Konflikt $c' = (\delta_1, \ldots, \delta_k, \ldots)$ erfüllt, die Äquivalenz damit gültig.

Als nächstes betrachten wir die Einhaltung der Invariante durch Erweiterung der Transitionsmenge um die in q' eingehende Transitionsmenge $\bigcup_{h \in H_p} \{(h, \delta_i, q')\}$. Aufgrund der durch Invariante 6 gewährleisteten Erreichbarkeit aller Zustände können wir mit p' als einem beliebigem Vorgängerzustand zu einem Zustand $h \in H_p$ immer zwei geeignete Sequenzen $v = (\iota_1, \ldots, \iota_j, \iota_{j+1}, \ldots, \iota_{k-1}, \iota_k, \ldots)$ und $v' = (\iota'_1, \ldots, \iota'_{k'-1}, \iota'_{k'}, \ldots)$ finden, für die gilt $\rho_v^*(d(p')) = p'$ und $\rho_{v'}^*(d(p)) = p$, sowie $k = d(p') + 1$ und $k' = d(p)$. Diese Sequenzen besitzen demnach einen kürzestmöglichen Präfix der Länge $k-1$ bzw. k', um p' bzw. p durch Einlesen dieses jeweiligen Präfixes zu erreichen. Weiterhin fordern wir von den Sequenzen die Eigenschaften $\rho_v^*(d(p') + 1) = h$ und $\rho_{v'}^*(d(p) + 1) = q'$, was aufgrund der Transitionen (p', δ, h) und (p, δ_i, q') in der geforderten Bedingung der Menge H_p, bzw. durch die direkt hinzugefügte, bereits oben diskutierte Transition immer möglich ist. Für diese Sequenzen gilt demnach $\iota_k = \delta$, sowie $\iota'_{k'} = \delta_i$. Wir können für jeden Zustand $h \in H_p$ ein entsprechendes Paar von Sequenzen mit obigen Eigenschaften finden, wobei aufgrund der für die Bestimmung der Menge H_p geforderten Existenz der Transition $(p', \delta, p) = \left(\rho_v^*(k-1), \delta, \rho_{v'}^*(k'-1)\right)$ nach dieser Invariante gilt $\exists j < k : \forall 1 \leq m \leq k-j-2 : v_{(j+m)} = v'_{(m)}$, sowie $\iota'_{k'-1} = \delta$, d.h. der Präfix von v' stimmt mit einem Teilwort aus v ab dem $(j+1)$-ten Zeichen über $(k-1)-(j-1) = k-j-2$ Zeichen überein, sowie darüberhinaus das Zeichen δ mit dem nächsten, im Präfix von v'

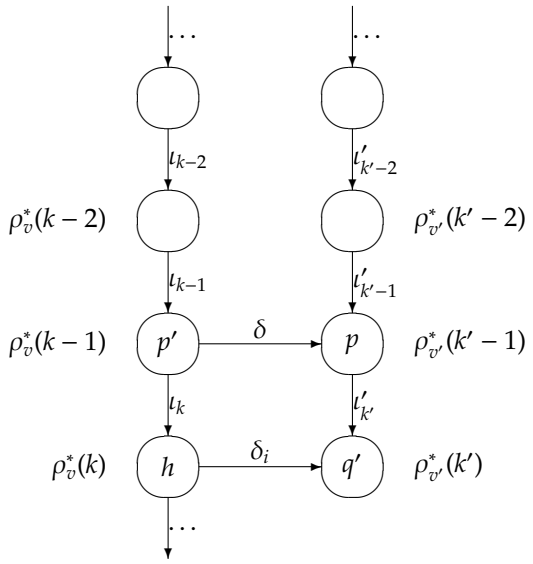


Abbildung 4.6: Schematische, unvollständige Darstellung des Automaten A_{C_ω}. Dargestellt ist die Sitation nach Abarbeitung des äußeren else-Zweigs eines Schleifendurchlaufs in Algorithmus 5. Es gilt $h \in H_p = \left\{h \middle| h \in F \wedge p < h \wedge \exists p' \in Q, \delta \in \Sigma : \{(p', \delta, p), (p', \delta, h)\} \subseteq T\right\}$

befindlichen Zeichen $\iota'_{k'-1}$. Somit haben wir $\iota_{j+1} = \iota'_1, \ldots, \iota_{k-1} = \iota'_{k'-2}$. Da motiviert durch die Existenz der Transition (p', δ, h) für die Definition der Menge H_p nun nach unserer Wahl von v gilt $\iota_k = \delta$, haben wir mit $\iota_k = \delta = \iota'_{k'-1}$ eine Verlängerung der Übereinstimmung zwischen v und v' um das gemeinsame Zeichen $\iota'_{k'} = \delta_i$, die nach der Invariante eine Transition von $\left(\rho^*_v(k), \iota'_{k'}, \rho^*_{v'}(k')\right)$ erfordert. Genau diese werden mit der Transitionsmenge $\bigcup_{h \in H_p} \{(h, \delta_i, q')\}$ für alle $h \in H_p$ hinzugefügt, so daß die Gültigkeit der Invariante bzgl. aller in den neuen Zustand q' eingehenden Transitionen gewährleistet ist. Zu diesen Überlegungen ist in Abbildung 4.6 der Automat unvollständig ausschnittsweise dargestellt.

Eine völlig analoge Betrachtung können wir hinsichtlich der von q' ausgehenden Transitionsmenge $\bigcup_{l \in L_{q'}} \{(q', \delta, r) | \exists (l, \delta, r) \in T\}$ durchführen. Aufgrund der Existenz der Transition (p, δ_i, l) zu jedem Zustand $l \in L_{q'}$ ist für jeden Zustand r, der aufgrund der Existenz einer Transition (l, δ', r) ein Nachfolgezustand von l ist, zur Einhaltung der Invariante bzgl. aller von q'ausgehenden Transitionen eine Transition (q', δ', r) notwendig. Genau diese werden mit der Transitionsmenge $\bigcup_{l \in L_{q'}} \{(q', \delta, r) | \exists (l, \delta, r) \in T\}$ als ausgehende Transitionen hinzugefügt. In Abbildung 4.7 findet sich eine erweiterte Darstellung des Automaten mit Konkretisierung der Zeichen der Transitionen, welche sich aufgrund obiger Überlegungen genau in dieser Struktur ergeben. Damit haben wir die Einhaltung der Invariante für alle in einem Schleifendurchlauf hinzugefügten Zustände und Transitionen gezeigt, womit der induktive Beweis abgeschlossen ist. □

Der folgende Satz stellt sicher, daß der vom Automaten eingenommene Zustand unabhängig von einem beliebig langen eingelesenem Wort ist, welches dem aktuell observierten Konflikt vorangestellt ist.

Satz 1 *Sei $v = x \cdot c_p$ eine beliebige Sequenz mit Konflikt-freiem Präfix x, c_p der Präfix eines Konflikts $c = c_p \cdot c_s$ und $A_{C_\omega} = (Q, \{q_0\}, \Sigma, T, F)$ ein nach Algorithmus 5 aus einer Menge von Konflikten C_{inv} konstruierter Automat. Wenn c_p der längste Präfix eines Konflikts im*

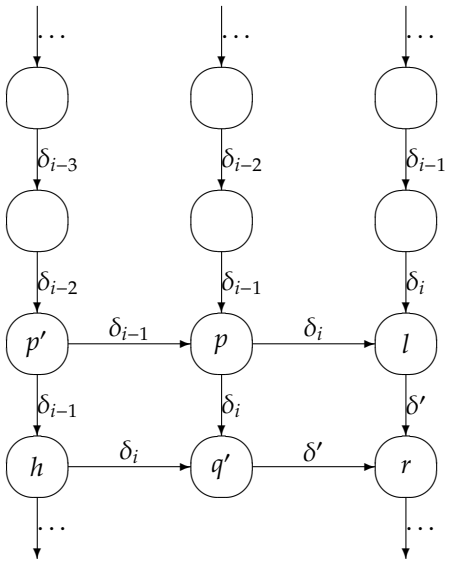


Abbildung 4.7: Schematische, unvollständige Darstellung des Automaten A_{C_ω} mit Darstellung der konkreten Zeichen der Transitionen. Dargestellt ist die Sitation nach Abarbeitung des äußeren else-Zweigs eines Schleifendurchlaufs in Algorithmus 5. Es gilt $h \in H_p = \left\{h \middle| h \in F \wedge p < h \wedge \exists p' \in Q, \delta \in \Sigma : \{(p', \delta, p), (p', \delta, h)\} \subseteq T\right\}$ und $l \in L_{q'} = \left\{l \middle| (p, \delta_i, l) \in T\right\}$

Suffix von v ist, d.h. $\nexists c'_p, c'_s \in \Sigma^* : c'_p \cdot c'_s \in C_{inv} \wedge c'_p$ *ist Suffix von* $v \wedge |c'_p| > |c_p|$*, dann gilt:*

$$\rho^*_v(|v|) = \rho^*_{c_p}\left(|c_p|\right)$$

Beweis von Satz 1: Jede beliebige konfliktfreie Sequenz x läßt sich als Konkatenation von Präfixen existierender Konflikte und Zeichen, die in keinem Konflikt verwendet werden, darstellen. Es genügt daher, den Satz für eine Konkatenation $v = c'_p \cdot c_p$ mit c'_p entweder als weiteren Präfix eines Konflikts $c' = c'_p \cdot c'_s \in C_{inv}$, oder mit c'_p als noch nicht verwendetem Zeichen σ nachzuweisen. Im Fall von $c'_p = \sigma$ ist der Nachweis trivial, da bei Einlesen von in Konflikten nicht verwendeten Zeichen σ immer der Initialzustand q_{ω_0} eingenommen wird, so daß jede vorangehende Sequenz offensichtlich irrelevant wird. Betrachten wir also den Fall $c'_p \cdot c'_s \in C_{inv}$, so daß $v = c'_p \cdot c_p = \left(\iota_1, \ldots, \iota_j, \iota_{j+1}, \ldots, \iota_k\right)$ mit $c'_p = \left(\iota_1, \ldots, \iota_j\right)$ und $c_p = \left(\iota_{j+1}, \ldots, \iota_k\right)$. Bei dieser Aufteilung von v ist c'_p nicht notwendigerweise der größte mögliche Präfix von c', da die folgende Teilsequenz $\left(\iota_{j+1}, \ldots, \iota_{j'}\right)$ mit $j \leq j' < k$ als Präfix von c_p mit dem Präfix von c'_s übereinstimmen und damit ebenfalls Bestandteil eines längsten Konlifktpräfixes $c''_p = \left(\iota_1, \ldots, \iota_{j'}\right)$ des Konflikts c' sein kann. Somit ist der Transitionsübergang von $\rho^*_{c'}(j')$ durch Einlesen des nächsten Zeichens $\iota_{j'+1}$ zu betrachten, welches nicht mehr Bestandteil von c' ist, d.h. $c'_{(j'+1)} \neq \iota_{j'+1}$. Nach Invariante 15 existieren unter dem Zeichen $\iota_{j'+1}$ Transitionen zu mehreren Zuständen l einer Menge L, für die gilt $\exists c'' = \left(\iota_m, \ldots, \iota_{j''}\right) \in C_{inv}, 1 \leq m < j'' < k : l = \rho^*_{c''}(j'')$. Nach Invariante 12 gilt $d(l) = j''$, sowie nach Invariante 8 das einmalige Vorkommen einer solchen Distanz für alle von $\rho^*_{c'}(j')$ mit dem Zeichen $\iota_{j'+1}$ erreichbaren Zielzustände l, so daß gemäß der Determinierung mit der Funktion *tgt* genau der Zustand $l \in L$ ausgewählt wird, der die maximale Distanz zu q_{ω_0} aufweist und damit nach Invariante 12 auch den längstmöglichen Präfix eines anderen Konflikts verarbeitet.

Wir erhalten damit $\rho^*_v(j'+1) = \rho^*_{c''}(j'') = \rho^*_{c_p}(j'-j)$. Von diesem Zustand aus ist das Erreichen des Zustands $\rho^*_{c_p}(|c_p|)$ bei Einlesen der verbleibenden Zeichenfolge $\left(\iota_{j'+1}, \ldots, \iota_k\right)$ trivial und obiger Satz damit bewiesen. □

Nachdem alle Invarianten und Sätze formuliert und bewiesen worden sind, gilt es nun nachzuweisen, daß von jedem beliebigen Zustand aus

1. jede endliche Sequenz $v_c = v \cdot c$, die eine Konfliktsequenz $c = (\delta_1, \delta_2, \ldots, \delta_k)$ als Suffix beinhaltet, in einen nicht-akzeptierenden Senkenzustand $\rho^*_{v_c}(|v_c|) \notin F, \nexists \delta \in \Sigma, q \in Q : \left(\rho^*_{v_c}(|v_c|), \delta, q\right) \in T$ führt und
2. jede beliebig lange Sequenz, $v = (\delta'_1, \delta'_2, \ldots, \delta'_j)$ die keinen Konflikt als Suffix beinhaltet, akzeptiert wird, d.h. $\rho^*_v(j) \in F$.

Nachweis der Nicht-Akzeptanz von Konflikt-beinhaltenden Sequenzen (1): Der Nachweis ist umittelbar aus den Invarianten 13 und 11, sowie dem Satz 1 gegeben, da Präfix-unabhängig im Falle eines Konflikts ein nicht-akzeptierender Senkenzustand eingenommen wird. □

Nachweis der Akzeptanz von Konflikt-freien Sequenzen (2): Aufgrund von Satz 1 ist bereits gewährleistet, daß im Falle des Einlesens konfliktfreier Sequenzen Zustandswechsel möglich sind, da eine einfache Verlängerung von v um das nächste Zeichen des mit c_p beginnenden Konflikts gemäß $\rho^*_{v\cdot(\iota)}(|v|+1) = \rho^*_{c_p\cdot(\iota)}\left(|c_p|+1\right)$ nach Invariante 12 notwendig zu einem Zustandswechsel zu einem Zustand höherer minimaler Distanz zu q_{ω_0} führen muß. Damit kann es sich bei $\rho^*_{c_p}(|c_p|)$ für $c_p \notin C_{inv}$ nicht um einen Senkenzustand handeln. Die Invariante 14 garantiert für diesen Fall jedoch, daß $\rho^*_{c_p}\left(|c_p|\right) \in F$ für $c_p \notin C_{inv}$ und $\rho^*_{c_p}\left(|c_p|\right) = \rho^*_v(|v|)$ damit akzeptierend ist. □

4.3.3.3 Kreuzprodukt beider Automaten

Beide Automaten $A_{C_\Re} = (Q_\Re, q_{\Re_0}, \Sigma, T_\Re, F_\Re)$ und $A_{C_\omega} = (Q_\omega, q_{\omega_0}, \Sigma, T_\omega, F_\omega)$ werden nun zu einem co-1-akzeptierenden ω-Automaten A_C durch Kreuzproduktbildung integriert. Der semantische Unterschied von regulären und ω-Automaten ist für das auf syntaktischer Ebene definierte Kreuzprodukt dabei nicht von Belang. Wir erhalten den ω-Automaten $A_C = (Q, q_0, \Sigma, T, F)$ mit

A_C

- $Q = Q_\Re \times Q_\omega$,
- $q_0 = (q_{\Re_0}, q_{\omega_0})$,
- $T_C = \left\{\left(\left(q_{\Re_1}, q_{\omega_1}\right), \sigma, \left(q_{\Re_2}, q_{\omega_2}\right)\right) \middle| \left(q_{\Re_1}, \sigma, q_{\Re_2}\right) \in T_\Re, \left(q_{\omega_1}, \sigma, q_{\omega_2}\right) \in T_\omega\right\}$ und
- $F = F_\Re \times F_\omega$

Für diesen Automaten gilt nun Eigenschaft (4.2), d.h.

$$A_C \models \neg\left(\bigvee_{c_i \in C_{ini}} \lambda(c_i)\right) \wedge \neg\mathbf{F}\left(\bigvee_{c_v \in C_{inv}} \lambda(c_v)\right)$$

Wir haben somit nun einen Automaten konstruiert, der jeden Lauf akzeptiert, der keine der bekannten Konflikte in Präfix- oder Infixstellung enthält. Als letzter Schritt bleibt die Kombination dieses Automaten mit der initialen Abstraktion A_0, was im folgenden Abschnitt beschrieben wird.

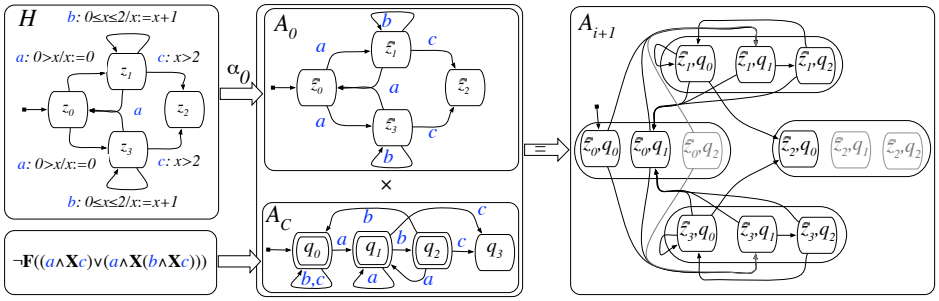


Abbildung 4.8: Vereinfachtes Beispiel der Abstraktionsverfeinerung. Einen unsicheren Zustand z_2 gegeben, wird der Pfad der Gegenbeispiele $\hat{\pi} = (\hat{z}_0, \hat{z}_1, \hat{z}_2)$ und $\hat{\pi} = (\hat{z}_0, \hat{z}_1, \hat{z}_1, \hat{z}_2)$ in A_{i+2} von (z_0, q_0) ausgehend ausgeschlossen. A_C ist dabei der ω-Automat, $A_C \models \neg\mathbf{F}((a \wedge \mathbf{X}c) \vee (a \wedge \mathbf{X}(b \wedge \mathbf{X}c)))$, mit $a = (\{x|0 > x\}, x \mapsto 0), b = (\{x|0 \leq x \leq 2\}, x \mapsto x + 1), c = (\{x|x > 2\}, x \mapsto x)$.

4.3.4 Abstraktionsverfeinerung durch Parallelkomposition

Um die Eigenschaft (4.2) von A_C auf den Automaten A zu übertragen, wird nun auch hier das Kreuzprodukt zwischen der initialen Abstraktion A_0 und A_C gebildet. Da $A_C = (Q, q_0, \Sigma, T_C, F)$ formal ein ω-Automat in Gestalt eines 5-Tupels ist, $A_0 = \left(\hat{Z}, \hat{Z}_0, \hat{E}\right)$ hingegen ein Transitionssystem in Form eines 3-Tupels, wird hier ein Kreuzprodukt verwendet, welches bei der Transitionsberechnung eine Übereinstimmung der Zeichen aus Σ mit den Guard Set / Jump Set Funktionspaaren der den auf H projizierten Transitionen aus A_0 voraussetzt. Nicht-akzeptierende Zustände aus A_C, welche im vorliegenden Falle co-1-akzeptierender Automaten immer Senken sind, werden im Kreuzprodukt nicht berücksichtigt und die dazugehörigen kritischen Transitionen somit in A verhindert. Formal definieren wir das Kreuzprodukt $\dot{\times} : \mathfrak{T} \times \mathfrak{B} \to \mathfrak{T}$ zwischen Transitionssystemen aus der Menge der Transitionssysteme $\mathfrak{T}$ und ω-Automaten aus der Menge Büchi-Automaten $\mathfrak{B}$, so daß $A = A_0 \dot{\times} A_C \stackrel{\text{def}}{=} \left(\hat{S}, \hat{S}_0, \hat{T}\right)$ mit

- $\hat{S} := \hat{Z} \times Q$
- $\hat{S}_0 := \hat{Z}_0 \times \{q_0\}$
- $\hat{T} := \{((\hat{z}_1, q_1), (\hat{z}_2, q_2)) | (\hat{z}_1, \hat{z}_2) \in \hat{E} \wedge (q_1, \sigma, q_2) \in T_C \wedge \exists t \in T : t = \tilde{\alpha}^{-1}((\hat{z}_1, \hat{z}_2)) \wedge \sigma = (g(t), j(t)) \wedge q_2 \in F\}$

Im Automat A genügen damit auch alle Pfade der Eigenschaft (4.2) und somit können in nachfolgenden Iterationen der Abstraktionsverfeinerung nur entweder gültige Pfade oder neue, noch nicht bekannte Konflikte im Rahmen der Pfadanalyse aufgedeckt werden.

In Abbildung 4.8 ist der Verfeinerungsschritt für ein Mini-Beispiel verdeutlicht, während Algorithmus 6 die Gesamtprozess der iterativen Abstraktionsverfeinerung noch einmal zusammenfaßt.

Bezüglich des abstrakten Zustandsraums ist anzumerken, daß viele Zustände $(\hat{z}, q) \in \hat{S}$ keine eingehenden Transitionen besitzen und daher trivial nicht erreichbar sind. Dies ist für alle Zustände $(\hat{z}, q) \in \hat{S}$ der Fall, für die $\forall (q', \sigma, q) \in T_C : \nexists t = (z', z) \in T \wedge \sigma = (g(t), j(t)) \wedge z = \tilde{\alpha}^{-1}(\hat{z})$ gilt, es also keine eingehenden Transitionen für z in H gibt deren GJ-Paar durch q „überwacht" wird. Der Zustandsraum von $\hat{S}$ kann dadurch wieder erheblich reduziert werden.

$A_{C_\Re} \coloneqq (\{q_{\Re_0}, \mathit{fin}\}, \{q_{\Re_0}\}, \Sigma, \{(q_{\Re_0}, \delta, \mathit{fin}) | \delta \in \Sigma\}, \{q_{\Re_0}, \mathit{fin}\})$
$A_{C_\omega} \coloneqq (\{q_{\omega_0}\}, \{q_{\omega_0}\}, \Sigma, \{(q_{\omega_0}, \delta, q_{\omega_0}) | \delta \in \Sigma\}, \{q_{\omega_0}\})$
$A \coloneqq A_0$
while $A \not\models \mathbf{AG}\neg\hat{S}_U$ **do** % Model-Checker Lauf
  $\hat{\pi} \coloneqq (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n), \hat{s}_n \in \hat{S}_U$ % Pfad von Model-Checker
  $(\mathit{result}, (x_0, x_1, \ldots, x_n)) \coloneqq \tilde{\mathcal{L}}'(\theta(\hat{\pi}), X_0)$
  if $\mathit{result} = \mathit{false}$ **then** % ungültiger Pfad
    foreach $c \in r_{\mathit{in}}(\theta(\hat{\pi}))$ **do** $A_{C_\Re} \coloneqq Add_\Re(A_{C_\Re}, c)$ **end**
    foreach $c \in r_{\mathit{in}}(\theta(\hat{\pi}))$ **do** $A_{C_\omega} \coloneqq Add_\omega(A_{C_\omega}, c)$ **end**
    $A \coloneqq (\mathit{Minimize}(\mathit{strip}(A_{C_\Re}) \times \mathit{Minimize}(\mathit{strip}(A_{C_\omega}))) \times A_0$
  else return $\pi = ((\tilde{\alpha}^{-1}(\hat{s}_0), x_0), (\tilde{\alpha}^{-1}(\hat{s}_1), x_1), \ldots, (\tilde{\alpha}^{-1}(\hat{s}_n), x_n))$ % gültiger Pfad
end
return *true* % $TS_H \models \mathbf{AG}\neg S_U$

Algorithmus 6: ω-Cegar Verfahren. Es wird entweder das Ergebnis *true* oder ein Pfad $\pi = (s_0, \ldots, s_n) \in \overrightarrow{TS_H}, s_n \in S_U$ zurückgegeben.

Damit haben wir nun vollständig beschrieben, wie in einem Iterationsschritt aus der Menge der bekannten Konflikte zuzüglich der Menge der neuen Konflikte ein verfeinertes Transitionssystem A konstruiert wird, mit dem im nächsten Schritt der Model-Check-Lauf wiederholt werden kann.

4.4 Korrektheit der verwendeten Abstraktion

In diesem Abschnitt wird die Korrektheit der im ω-Cegar verwendeten Abstraktionsrelation α nachgewiesen. Da die Definition selbiger noch aussteht, wird nach einer Beschreibung der durch das Verfahren implizierten Partitionierung des kontinuierlichen Zustandsraums unter dessen Zuhilfenahme diese formuliert.

4.4.1 Implizierte Partitionierung des kontinuierlichen Zustandsraums

Betrachten wir das Alphabet $\Sigma = GJ$ des Automaten A_C nicht mehr als Namen von Zeichen, sondern wieder als den Transitionen zugehörigen Guard Set / Jump Set Paaren, so erhalten wir einen Schritt-diskreten Hybriden Automaten $H_{A_C} \coloneqq (Q, q_0, X, X_0, T_H, g, j)$ gemäß Definition 9 mit

- $T_H \coloneqq \left\{(q_1, q_2) \middle| \exists(q_1, \delta, q_2) \in T_C\right\}$
- $g((q_1, q_2)) \coloneqq \gamma_t$ genau dann wenn $\exists(q_1, \delta, q_2) \in T_C : \delta = (\gamma_t, \zeta)$
- $j((q_1, q_2)) \coloneqq \zeta_t$ genau dann wenn $\exists(q_1, \delta, q_2) \in T_C : \delta = (\gamma, \zeta_t)$

Gemäß der durch ein Pfad-Transitionssystem nach Definition 10 auf Seite 30 beschrieben Semantik ist jedem diskreten Zustand darin eine erreichbare Teilmenge des kontinuierlichen Zustandsraums zugeordnet. Mit dieser Interpretation partitioniert der Automat A_C de facto den kontinuierlichen Zustandsraum X in $|Q|$ Partitionen, welche mit den einzelnen Zuständen assoziiert sind. Der Zustand $q_0 = (q_{\Re_0}, q_{\omega_0})$ repräsentiert dabei die initiale kontinuierliche Zustandsmenge X_0, eine davon ausgehende Transition zu einem Zustand q_1 mit einer Transition $(q_0, (\gamma, \zeta), q_1)$ entsprechend obiger Interpretation und der semanti-

schen Definition des Pfad-Transitionssystems den kontinuierlichen Zustandsraum $X_1 = \left\{x' \middle| \exists x \in (X_0 \cap \gamma) \wedge x' \in \zeta(x)\right\}$, usw.

Sei C_q die Menge aller Präfixe c von Konflikten, für die die zugehörigen Läufe von A_C im Zustand $q = (q_\Re, q_\omega) \in Q$ enden. Dann ist die dem Zustand q zugeordnete Partition $X_q \subseteq X$ des kontinuierlichen Zustandsraums beschrieben durch die Funktion $\chi : Q \to 2^X$ mit χ

$$\chi(q) \stackrel{\text{def}}{=} \begin{cases} \bigcup_{c \in C_q} \mathcal{L}(c, X_0) & \text{if } q_\Re \neq \mathit{fin} \\ \bigcup_{c \in C_q} \mathcal{L}(c, X) & \text{if } q_\Re = \mathit{fin} \end{cases}$$

Für lineare Systeme, in denen die Guard Sets durch konvexe Polyeder beschrieben werden können, sind gleichermaßen die Partitionen konvexe Polyeder, da Schnittmengen von Polyeder wieder durch Polyeder beschrieben sind.

4.4.2 Abstraktionsrelation

Bislang wurde die Abstraktionsrelation gemäß Definition 6 auf Seite 24 lediglich für A_0 angegeben, welche ein Spezialfall der allgemeinen Abstraktionsrelation α darstellt. Mit Hilfe der die vorigen Abschnitt betrachteten Partitionierung des kontinuierlichen Zustandsraums kann nun die Abstraktionsrelation formal beschrieben werden, die in dem ω-Cegar Verfahren angewandt wird. Die Abstraktionsrelation α ist in ihrer allgemeinen Form durch folgenden Satz beschrieben, wobei α $\hat{S} = \hat{Z} \times Q$ das in Abschnitt 4.3.4 beschriebene Kreuzprodukt der initialen Abstraktion $A_0 = \left(\hat{Z}, \hat{Z}_0, \hat{E}\right)$ und des ω-Automaten $A_C = (Q, q_0, \Sigma, T_C, F)$ mit $Q = Q_\Re \times Q_\omega$, $q_0 = (q_{\Re_0}, q_{\omega_0})$ darstellt:

Satz 2 *Das Transitionssystem $A = \left(\hat{S}, \hat{S}_0, \hat{T}\right), \hat{S} = \hat{Z} \times Q$ ist eine Abstraktion des Pfad-Transitionssystems $TS_H = (S, S_0, E), S = Z \times X$ unter der Abstraktionsrelation α mit*

$$\alpha = \left\{((z, x), (\hat{z}, q)) \in (Z \times X) \times \left(\hat{Z} \times Q\right) \middle| \tilde{\alpha}^{-1}(\hat{z}) = z \wedge x \in \chi(q)\right\}$$

Diese zentrale Aussage, welche die Vorgehensweise des ω-Cegar Verfahrens legitimiert, wird im folgenden Abschnitt bewiesen.

4.4.3 Beweis der Korrektheit

Nach Definition 6 sind die folgenden zwei Eigenschaften für die Korrektheit von Satz 2 zu zeigen:

- Nachweis der Korrelation der Initialzustände.

  Zu zeigen ist: $\forall (\hat{z}_0, q_0) \in \hat{Z}_0 \times \{q_0\} : \exists (z_0, x_0) \in S_0 \wedge ((\hat{z}_0, q_0), (z_0, x_0)) \in \alpha$.

  Da es genau einen Initialzustand $(\hat{z}_0, q_0) = (\hat{z}_0, (q_{\Re_0}, q_{\omega_0}))$ gibt, $\hat{z}_0$ isomorph zu z_0 ist und $\chi((q_{\Re_0}, q_{\omega_0})) = X_0$ nach Definition von χ ist, gilt für alle Initialzustände $(z_0, x_0) \in S_0, x_0 \in X_0$ die Relation α mit $((\hat{z}_0, q_0), (z_0, x_0)) \in \alpha$.

- Nachweis der Existenz der Transition t im folgenden Diagramm für alle Transitionen $\hat{t}$, wobei $x_1 \in \chi(q_1)$ und $x_2 \in \chi(q_2)$:

$$\begin{array}{ccc} (\hat{z}_1, q_1) & \sim_\alpha & (z_1, x_1) \\ \downarrow_{\hat{t}} & & \downarrow_t \\ (\hat{z}_2, q_2) & \sim_\alpha & (z_2, x_2) \end{array}$$

1. $\exists(z_1, z_2) \in T$ aufgrund der Isomorphie von A_0 zu H und der Konstruktionsbedingung $(\hat{z}_1, \hat{z}_2) \in \hat{T}$ für Transitionen in A.
2. $\exists(q_1, \sigma, q_2)$ mit $\sigma := (\gamma, \zeta) = (g((z_1, z_2)), j((z_1, z_2)))$ aufgrund der Konstruktionsbedingung $(q_1, \sigma, q_2) \in T_C \wedge \exists t \in T : t = \tilde{\alpha}^{-1}((\hat{z}_1, \hat{z}_2)) \wedge \sigma = (g(t), j(t))$ für Transitionen in A.
3. Für $\left(C_\perp = C_{ini} \wedge \tilde{X} = X_0\right) \vee \left(C_\perp = C_{inv} \wedge \tilde{X} = X\right)$ gilt $\exists c = (\delta_1, \delta_2, \ldots, \delta_k) \in C_\perp : X_1 := \chi(q_1) = \mathcal{L}\left((\delta_1, \delta_2, \ldots, \delta_k), \tilde{X}\right)$ aufgrund der Definition von χ. Unter Hinzunahme von 2. gilt außerdem $X_2 : \chi(q_2) = \mathcal{L}\left((\delta_1, \delta_2, \ldots, \delta_k, \sigma), \tilde{X}\right)$.
4. $X_2 = \mathcal{L}((\sigma), X_1)$ nach Definition von $\mathcal{L}$ und 3. Damit gilt insbesondere $\forall x_2 \in X_2 : \exists x_1 \in (\gamma \cap X_1) : x_2 = \zeta(x_1)$
5. Nach 1. und 4. sowie der Definition von TS_H gilt daher: $((z_1, x_1), (z_2, x_2)) \in E$.

Damit ist nachgewiesen, daß die durch das ω-CEGAR Verfahren implizierte Abstraktionsrelation α, die in jedem Iterationsschritt gilt, eine gültige Abstraktion des Pfad-Transitionssystems des Hybriden Automaten darstellt. Nach Lemma 1 auf Seite 24 gilt damit insbesondere, daß bei Unerreichbarkeit von Zuständen in A_i auf entsprechende Unerreichbarkeit von korrespondierenden Zuständen in TS_H geschlossen werden kann. □

4.5 Terminierung

Betrachten wir noch einmal die möglichen Ergebnisse einer Iteration des Verfeinerungsverfahrens:

1. Der Model-Checker weist nach, daß es keinen Pfad bzgl. der nachzuweisenden Eigenschaft φ gibt, $A \models \varphi$.
2. Es wird ein Pfad $\hat{\pi}$ gefunden, der in TS_H zu π konkretisiert werden kann.
3. Es wird ein ungültiger Pfad $\hat{\pi}$ gefunden, aus dem mindestens ein Konflikt extrahiert werden kann.
4. Aufgrund Unentscheidbarkeit oder numerischer Probleme in dem Algorithmus der Funktion $\mathcal{L}$ kann weder der Pfad konkretisiert, noch Konflikte identifiziert werden. Letzteres ist beispielsweise bei Auftreten von numerischen Instabilitäten in Solvern der Fall.

Das Verfahren terminiert offensichtlich mit einem der gewünschten Ergebnisse in den Fällen 1. und 2.

Im Fall 3. ist mindestens eine weitere Iteration notwendig. Da in diesem Fall immer mindestens der ermittelte Pfad $\hat{\pi}$ für nachfolgende Iterationen ausgeschlossen ist und die Anzahl der Pfade der Länge kleiner gleich $k \in \mathbb{N}$ aufgrund der Endlichkeit des diskreten Zustandsraums Z ebenfalls endlich ist, müssen schließlich alle Pfade bis zu einer endlicher Länge k durch das Verfahren gefunden werden. Daraus folgt:

- Wenn es einen Pfad endlicher Länge gibt, wird dieser auch gefunden, eine Terminierung des Verfahrens ist somit garantiert.

- Wenn es keinen Pfad endlicher Länge gibt, kann nur folgende Aussage gemacht werden: Es gibt keinen Pfad mit einer Länge kleiner gleich k. Das Verfahren terminiert in diesem Falle nicht, bzw. nur unter Vorgabe der Zahl k.

Tritt hingegen der Fall 4. ein, so fehlt das für das Verfahren kritische Ergebnis der Funktion $\mathcal{L}$ und das Verfahren muß abgebrochen werden. Theoretisch kann dieser Fall für entscheidbare Problemstellungen wie dem der Lösung von linearen Ungleichungssystemen, wie dies bei linearen Hybriden Systemen wie später beschrieben benötigt wird, nicht vorkommen, jedoch sind Implementierungen der Entscheidungsverfahren aus Effizienzgründen diesbzgl. differenzierter zu betrachten.

Wir halten daher fest, daß bei Nichtauswirkung numerischer Probleme das Verfahren für ein festes k immer terminiert und darüber hinaus ohne dieses terminieren kann, jedoch nicht zwingend muß.

4.6 CTL-Model-Checking und ω-Cegar

Bislang sind wir in den Ausführungen nur auf die Verifikation von Sicherheitseigenschaften eingegangen. Das Verfahren läßt sich mit einigen Einschränkungen jedoch auch auf CTL-Model-Checking übertragen, bei dem eine Eigenschaft φ durch eine ACTL- oder ECTL-Formel beschrieben ist und mit dem in Abschnitt 2.4.1 dargelegten Verfahren verifiziert werden muß. Wie bereits in Abschnitt 2.6.0.3 auf Seite 24 bemerkt, kann durch Anwendung des Model-Check-Verfahrens auf ein abstraktes System für ACTL- und ECTL-Formeln grundsätzlich immer

- entweder das Ergebnis auf das konkrete System übertragen werden, wie dies bei gültigen ACTL-Formeln und ungültigen ECTL-Formeln der Fall ist,
- oder das Ergebnis liegt als Gegenbeispiel bzw. Zeugenpfad vor und kann entsprechend (in-)validiert werden. Bei ungültigen Pfaden wird das Verfahren fortgesetzt.

Somit ist das ω-Cegar Verfahren für diese Art von Formeln grundsätzlich anwendbar. Für eine solche Anwendung sind jedoch zum Einen Zustandsformeln in φ mit Bezugnahme zu dem kontinuierlichen Zustandsraum zu behandeln und zum Anderen ist das Problem unendlicher Pfade zu lösen, wie in den folgenden Abschnitten dargestellt.

4.6.1 Transformation des Modells für Zustandsformeln in X

Sofern eine Zustandsformel φ Ausdrücke beinhaltet, welche sich auf den kontinuierlichen Zustandsraum erstrecken, müssen diese in einem dem Verifikationsprozess einmalig vorangehenden Transformationsprozess dergestalt in das Hybride System H eingearbeitet werden, daß für diese Ausdrücke explizite Zustände vorhanden sind und die als solche referenziert werden können, in denen der Ausdruck wahr ist.

Für ein Schritt-diskretes Hybrides System $H = (Z, Z_0, X, X_0, T, g, j)$ und eine in der Formel φ vorkommende Zustandsformel f, welche sich auf den kontinuierlichen Zustandsraum erstreckt, erzeugen wir ein neues, um eine Labeling-Funktion L erweitertes Schritt-diskretes Hybrides System $H' = \left(Z', Z_0', X, X_0, T', g', j', L\right)$ mit

- $Z' = \left\{ z^e \middle| z \in Z \wedge e \in \{f, \neg f\} \right\}$
- $Z_0' = \left\{ z_0^e \middle| z_0 \in Z_0 \wedge e \in \{f, \neg f\} \right\}$

- $T' = \left\{ \left(z_1^{e_1}, z_2^{e_2} \right) \middle| (z_1, z_2) \in T \wedge e_1, e_2 \in \{f, \neg f\} \right\}$
- $g'\left(\left(z_1^{e_1}, z_2^{e_2} \right) \right) = g\left((z_1, z_2) \right) \cap \{x \in X | e_2\}$
- $j'\left(\left(z_1^{e_1}, z_2^{e_2} \right) \right) = j\left((z_1, z_2) \right)$
- $L(z^e) = e$

Die Zustandsformel f ist damit nun nicht mehr eine zu interpretierende Zustandsformel, sondern eine atomare Proposition, so daß φ unverändert beibehalten werden kann. Sofern noch andere Zustandsformeln in φ betroffen sind, wird mit diesen obige Transformation ausgehend von H' wiederholt, wobei die bereits vergebenen atomaren Propositionen $L(z)$ durch die Labeling-Funktion für $L(z^e)$ übernommen und erweitert werden.

Für die Abstraktionen $A_i = \left(\hat{S}, \hat{S}_0, \hat{E} \right)$ wird gleichermaßen eine Labeling-Funktion $\hat{L}$ eingeführt, so daß aus diesen Transitionssystemen nun Kripke-Strukturen nach Definition 3 werden. Dabei soll gelten: $\forall \hat{s} \in \hat{S}, z \in Z : \hat{L}(\hat{z}) = L(z) \Leftrightarrow (z, \hat{s}) \in \alpha$.

Damit haben wird unter Verwendung von Labeling-Funktionen und einer Transformation des Hybriden Systems die Voraussetzung zur Anwendung des Model-Check-Algorithmus auf beliebige Zustandsformeln innerhalb einer CTL-Formel ausgedehnt.

4.6.2 Unendliche Pfade

Das grundsätzliche Problem bei CTL-Model-Checking sind die als unendlicher Pfad repräsentierten Gegenbeispiele, die vom Model-Checker bei Existenz von Fairness-Bedingungen generiert werden. Wie wir in Abschnitt 2.4.3 gesehen haben, wird für diese Pfade ein Suffix innerhalb einer starken Zusammenhangskomponente ermittelt, der periodisch dem bereits ermittelten Präfix des Pfades angehangen werden kann. Damit ist die Repräsentation des unendlichen Pfades endlich und kann zur Erzeugung längerer Präfixe von beliebiger endlicher Länge genutzt werden.

Genau dies ermöglicht es dem ω-Cegar Verfahren, die Konfliktsuche auf beliebig lange Präfixe des Pfades auszudehnen und, sofern dabei Konflikte gefunden werden, den Verifikationsprozess mit der Verfeinerung des Systems fortzusetzen. Bei einer Eigenschaft φ, die nach ausreichender Verfeinerung des abstrakten Modells A_i mit $A_i \models \varphi$ als gültig bestätigt werden kann, funktioniert das Verfahren demnach wie gehabt.

Anders hingegen stellt sich die Situation dar, in der ein Pfad unendlicher Länge tatsächlich gültig ist und keine Konflikte enthält. Dies ist mit der Lösungsfunktion $\mathcal{L}$ angewandt auf die unendliche GJ-Sequenz, auf die der Pfad mit der Pfadprojektionsfunktion θ projiziert wird, nicht entscheidbar und das Verfahren muß mit dem Ergebnis aufwarten, daß ein Gegenbeispiel zwar gefunden wurde, welches aber lediglich in einem endlichen Präfix gültig ist, wobei eine konkrete Präfixlänge errechnet und angegeben werden kann. Eine diagnostische Nutzbarkeit eines solchen Ergebnisses ist jedoch sicherlich vom Einzelfall abhängig.

4.6.3 Widerlegungsbäume

Wie bereits in Abschnitt 2.6.0.3 auf Seite 25 erwähnt, besteht bei Formeln wie z.B. $\mathbf{A}(\mathbf{X}\neg a \vee \mathbf{X}\neg b)$ das Gegenbeispiel aus einem Widerlegungsbaum. Dieser muß als Baum konkretisierbar sein, was im Falle der ω-Cegar Abstraktion immer genau

dann der Fall ist, wenn alle im Widerlegungsbaum befindlichen Fehlerpfade konkretisierbar sind und die den Pfaden gemeinsamen Präfixteilpfade einer gültigen Konkretisierung identisch sind. Betrachten wir im Folgenden die Lösungsmengen einzelner Zweige bezüglich einer Konkretisierung des gesamten Baumes.

Jeder im Baum vorhandene diskrete Fehlerpfad determiniert die *GJ*-Sequenz durch die Pfadprojektionsfunktion θ. Die *GJ*-Sequenz wiederum determiniert die Folge kontinuierlicher Zustandsmengen gemäß der Anwendung der Lösungsfunktion $\mathcal{L}$, wie in Abschnitt auf Seite 41 beschrieben. Die Lösungsfunktion $\mathcal{L}$ stützt sich zur Berechnung einer Zustandsmenge X_k ausschließlich auf die bis zum k-ten Schritt anzuwendenen *GJ*-Paare $(\gamma_i, \zeta_i), i \leq k$. Damit ist die Bestimmung der Folge der zu einem diskreten Pfad gehörigen kontinuierlichen Zustandsmengen insbesondere nicht vom weiteren Berechnungsverlauf abhängig, d.h. für die Berechnung von X_k sind die verbleibenden *GJ*-Paare $(\gamma_i, \zeta_i), i > k$ irrelevant. Somit ist für die Berechnung einer kontinuierlichen Zustandsmenge X_k ausschließlich der bisher zurückgelegte Transitionspfad ausschlaggebend, die Lösungsmengen der einzelnen Pfade des Widerlegungsbaumes entsprechend völlig unabhängig voneinander.

Nun ist jedoch ein Baum, in dem für jeden Pfad nach Anwendung der jeweiligen Lösungsmengenberechnung nach Abschnitt auf Seite 41 eine nicht-leere Menge ermittelt wird, nicht notwendigerweise auch als Ganzes konkretisierbar, was sich durch den Bestimmungsversuch einer konkreten Lösung zeigt. Benötigt wird eine Menge $X'_0 \subseteq X_0$, so daß nicht nur für eine Teilmenge, sondern für alle Elemente in X'_0 die Lösungsfunktion $\mathcal{L}$ für alle *GJ*-Sequenzen $c_i \in C_{Tree}$ mit C_{Tree} als der Menge aller den Pfaden zugeordneten *GJ*-Sequenzen des Widerlegungsbaumes eine nichtleere Menge ergeben. Nur für Initialzustände $x_0 \in X'_0$ ist somit gewährleistet, daß alle Pfade des Widerlegungsbaumes konkretisierbar sind. Diese Menge X'_0 berechnet sich als Schnittmenge der Rückpropagationen der Pfad-Lösungsmengen unter Zuhilfenahme der Funktion $\tilde{\mathcal{L}}$ auf Seite 42 durch

$$X'_0 = \bigcap_{c_i \in C_{Tree}} \tilde{\mathcal{L}}(c_i, X_0)$$

Der Widerlegungsbaum ist genau dann konkretisierbar, wenn $X'_0 \neq \emptyset$. Andernfalls ist der Baum nicht konkretisierbar und somit als ungültig zu betrachten. Sofern jeder Pfad des Baumes in Isolation betrachtet gültig ist, gibt es keinen linearen Konflikt, der durch den ω-Automaten ausgeschlossen werden kann. Die Fortsetzung des iterativen Verfahrens würde in einem solchen Fall eine deutlich komplexere Verfeinerung erfordern, welche an dieser Stelle jedoch nicht weiter betrachtet werden soll.

Das ω-Cegar Verfahren ist somit für allgemeines CTL-Model-Checken eingeschränkt anwendbar, soweit obige Widerlegungsfälle nicht eintreten[2].

4.7 Erweiterungen des Verfahrens

In dieser Sektion werden Erweiterungen des ω-Cegar Verfahrens vorgestellt, die zum Einen auf eine Verkleinerung des ω-Automaten durch Hinzufügung weiterer Konflikte abzielen und zum Anderen die Anzahl der benötigten Iterationen des Abstraktionsverfeinerungsprozess durch Verallgemeinerung von Konflikten reduzieren soll.

[2]Die hier beschriebene Behandlung unendlicher Pfade, sowie die Konkretisierung von Widerlegungsbäumen wurden in der Implementierung des Verfahrens nicht umgesetzt.

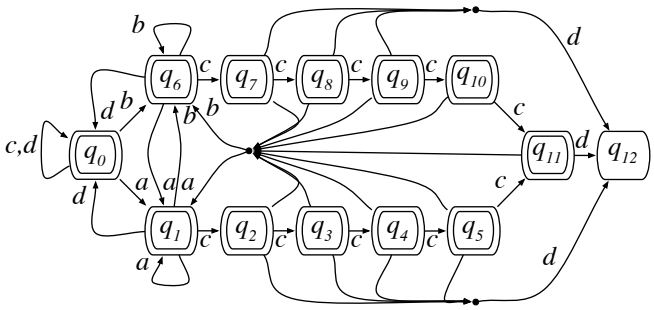


Abbildung 4.9: ω-Automat mit blockierter Minimierung

4.7.1 Minimierung mit Lernen

Die durch Algorithmus 4 beschriebene Minimierung des regulären und des ω-Automaten faßt genau die Zustände zusammen, bei denen die ausgehende Transitionsmenge gleich ist. Nun sind jedoch Situationen möglich, in denen die Menge der ausgehenden Transitionen sich nicht vollständig unterscheidet, sondern bis auf wenige Transitionen identisch ist. Da die Transitionen aufgrund der bereits beobachteten Konflikte existieren, können pathologische Situationen entstehen, bei denen einer oder wenige Konflikte, die noch nicht beobachtet wurden, ein beachtliches Minimierungspotential blockieren.

So ist der in Abbildung 4.9 dargestellte ω-Automat nicht weiter minimierbar, weil die Transition (q_{10}, d, q_{12}) fehlt. Wenn diese Transition existieren würde, könnten dadurch nicht nur die Zustände q_5 und q_{10} zusammengefasst werden, sondern im Anschluss daran auch alle Zustände $q_9, \ldots, q_6$ jeweils mit den Zuständen $q_4, \ldots, q_1$ in dieser Reihenfolge, so daß sich die Größe des Automaten in diesem Beispiel annähernd halbieren würde.

4.7.1.1 Analyse

Das Fehlen der entscheidenden Transition (q_{10}, d, q_{12}) liegt daran, daß die Sequenz[3] $c_{seq} = (b, c, c, c, c, d)$ offenbar nicht ausgeschlossen werden soll. Dies wiederum kann zwei Gründe haben:

1. Die Sequenz c_{seq} ist kein Konflikt.
2. Die Sequenz c_{seq} ist noch nicht aufgetreten.

Im ersten Fall ist an der Situation nichts zu ändern, da der minimale Automat nicht anders strukturiert sein kann, sofern die Sequenz c_{seq} explizit nicht ausgeschlossen werden darf. Vielversprechender ist hingegen der zweite Fall. Wenn die Sequenz noch nicht aufgetreten ist besteht die Möglichkeit, daß es sich tatsächlich bei c_{seq} um einen Konflikt handelt. Die Sequenz c_{seq} kann aufgrund der bisherigen vom Model-Checker berechneten Pfade noch nicht aufgetreten, oder womöglich aus strukturellen Gründen des Hybriden Modells sogar unmöglich sein. In beiden Fällen beantwortet eine einfache Berechnung von $\mathcal{L}\left(c_{seq}, X\right) \stackrel{?}{=} \emptyset$ die Frage, ob c_{seq} ein Konflikt ist und somit zum Automaten hinzugefügt werden darf. Im positiven Fall kollabiert der Zustandsraum des Automaten dadurch in der nachfolgenden Minimierung mit dem positiven Seiteneffekt, daß nun sogar

[3] Wir bezeichnen hier die Sequenz mit c_{seq} und nicht mit c, um diese explizit von dem Zeichen c des Alphabets $\Sigma = \{a, b, c, d, \ldots\}$ zu unterscheiden.

mehr Konflikte ausgeschlossen werden und mithin die Gesamtiterationszahl des Verfahrens möglicherweise weiter reduziert werden kann.

Für den Fall, daß ein Auftreten der Sequenz c_{seq} ohnehin unmöglich ist, wäre es sogar denkbar, c_{seq} zwecks freigesetztem Minimierungspotential dennoch auszuschließen. Dies erfordert jedoch einen vorherigen Nachweis von $A_0 \models \neg \mathbf{F}\lambda(c_{seq})$ mit Hilfe des Model-Checkers, was somit je nach Komplexität von A_0 eine vergleichsweise aufwendige Überprüfung darstellt. Daher würde sich vermutlich eine solche Maßnahme nur bei großen Problemen im komplexitätsbezogenen operativen Grenzbereich des Model-Checkers lohnen, um jede Möglichkeit an Komplexitätsreduktion auszunutzen. Hier wäre dann eine vorherige Analyse des resultierenden Minimierungspotentials in Abwägung der erwarteten Zustandsreduktion im Vergleich zum erforderlichen Berechnungsaufwand durch den Model-Checker angebracht um entscheiden zu können, ob dies tatsächlich durchgeführt werden soll.

4.7.1.2 Verfahren

Im Folgenden wird die Minimierung um eine Erkennung solcher Konflikte erweitert, die zu einer Minimierung des Automaten führen können. Die Minimierung wird weiterhin getrennt auf den regulären Automaten $A_{C_\Re}$ und den ω-Automaten A_{C_ω} angewandt. Hier wird das Verfahren für den ω-Automaten beschrieben, es läßt sich jedoch sehr einfach auf den regulären Automaten übertragen, weshalb wir im Folgenden generalisiert den Automaten $A_{C'}$behandeln.

Für die Erweiterung des Minimierungsprozesses zur Identifikation von zusätzlichen Konflikten mit Minimierungspotential für den Automaten benötigen wir zunächst weitere Definitionen von Funktionen zur Berechnung eines kürzesten Präfixes und eines kürzesten Suffixes von Läufen des Automaten bzgl. eines Zustands. Die im Folgenden definierten Funktionen wählen einen geeigneten beliebigen Lauf aus und wären prinzipiell nicht-deterministisch. Eine randomisierte Auswahl erfüllt die Aufgabe ebenso gut wie eine First-Match Strategie, welche in der Implementierung (typischerweise) verwendet wird.

Definition 25 (Prefix eines Zustands) *Die Funktion* $anyprefix : \mathfrak{B} \times Q \times C \to C$ *berechnet für einen Zustand q bzgl. eines Automaten* $A_{C'} = (Q, \{q_0\}, \Sigma, T_C, F)$ *und einer gegebenen GJ-Sequenz c eine neue Sequenz mit*

$$anyprefix(A_{C'}, q, c) = \begin{cases} c & gdw\ q = q_0 \\ (\delta) \cdot c & gdw\ q \neq q_0 \wedge \exists (p, \delta, q) \in T : \nexists (p', \delta', q) \in T_C : p' < p \end{cases}$$

Wir führen die abkürzende Schreibweise $\overleftarrow{q} \stackrel{def}{=} anyprefix(A_{C'}, q, ())$ *mit* $()$ *als leerer Sequenz ein, wenn der betreffende Automat* $A_{C'}$ *eindeutig aus dem Kontext hervorgeht.* $\overleftarrow{q}$

Die Funktion zur Berechnung eines Suffixes ergibt für einen nicht-akzeptierenden Zustand den kürzesten Pfad zu diesem:

Definition 26 (Suffix eines Zustands) *Die Funktion* $anysuffix : \mathfrak{B} \times Q \times C \to C$ *berechnet für einen Zustand q bzgl. eines Automaten* $A_{C'} = (Q, \{q_0\}, \Sigma, T_C, F)$ *und einer gegebenen GJ-Sequenz c eine neue Sequenz mit*

$$anysuffix(A_{C'}, q, c) = \begin{cases} c & gdw\ q \notin F \\ (\delta) \cdot c & gdw\ q \in F \wedge \exists (q, \delta, p) \in T : \nexists (q, \delta', p') \in T_C : p < p' \end{cases}$$

Analog zur Funktion anyprefix führen wir auch hier die abkürzende Schreibweise $\overrightarrow{q}$ $\overrightarrow{q} \stackrel{def}{=} anysuffix(A_{C'}, q, ())$ *ein, wenn der betreffende Automat* $A_{C'}$ *eindeutig aus dem Kontext hervorgeht.*

Desweiteren benötigen wir eine Funktion zur Änderung einer Abbildung $Q \to \mathbb{N}$:

Definition 27 *Die Funktion updatemap* $: (Q \to \mathbb{N}) \times Q \times \mathbb{N} \to (Q \to \mathbb{N})$ *ändert eine Abbildungsfunktion* $f : Q \to \mathbb{N}$ *mit:*

$$updatemap\,(f,(q',k)) \stackrel{def}{=} \left(q \mapsto \begin{cases} f(q) & gdw\ q \neq q' \\ k & gdw\ q = q' \end{cases} \right)$$

pmd

Diese Funktion wird benötigt für die Berechnung der Funktion $pmd : Q \to \mathbb{N}$ in Algorithmus 7, um den Wert der durch die Minimierung sich ändernden Distanz $d : Q \to \mathbb{N}$ nicht zu verlieren. Für jeden zusammengefassten Zustand q' steht somit durch die Funktion *pmd* auch nach der Minimierung die Information zur Verfügung, welche Distanz der am weitesten von q_0 entfernte Zustand besaß, der in q' eingegangen ist. Diese Information wird zur Klassifikation von zu generierenden Sequenzen benötigt, da unter Verwendung dieser die generierten Sequenzen auf solche beschränkt werden können, welche tatsächlich Minimierungspotential haben. Wie wir in Abschnitt 4.7.1.3 betrachten werden, können demgegenüber auch Sequenzen ohne Minimierungspotential generiert werden, welche dennoch durchaus erwünscht sein können.

Mit diesen Funktionen können wir uns durch den in Algorithmus 7 beschriebenen erweiterten Minimierungsverfahren *GJ*-Sequenzen konstruieren, welche nach Ergänzung des Automaten die fehlende Transition liefern könnten, sofern sie als Konflikt bestätigt werden. Die Konstruktion dieser Konflikte c bzgl. einer angestrebten Transition (q,δ,p) wird aus der Konkatenation eines Präfix von q mit dem Zeichen δ als Bindeglied zum Suffix von p erreicht, formal $c = \overleftarrow{q} \cdot (\delta) \cdot \overrightarrow{p}$, wobei (δ) eine Sequenz der Länge 1 des einzelnen *GJ*-Paares δ mit $\delta \in \Sigma = GJ$ darstellt. Falls q selbst eine Transition (q,δ,p') mit $p' \neq q_0$ besitzt, wird jedoch keine Sequenz erzeugt. Außerdem wird eine solche Sequenz nur dann erzeugt, wenn $d(q) < pmd(p)$, da wir andernfalls eine Sequenz auf Basis einer Zwischenkonflikttransition erzeugen würden. Dieser Fall ist ausführlich im folgenden Abschnitt behandelt, da hierdurch verlängerte Sequenzmuster generiert werden können.

Lemma 3 *Für eine Sequenz* $c = \overleftarrow{q} \cdot (\delta) \cdot \overrightarrow{p}$ *gilt* $A_C \models \mathbf{EF}\lambda(c)$*, d.h. die Sequenz c wird noch nicht von* A_C *ausgeschlossen.*

Beweis: Mit dem Präfix $\overleftarrow{q}$ wird der Zustand $\rho_c(d(q)) = q$ erreicht. Wenn es ein $\overrightarrow{q}$ gäbe, welcher identisch ist mit einem $(\delta) \cdot \overrightarrow{p}$, formal $\left\{ (\delta) \cdot c_p \,\middle|\, c_p = \overrightarrow{p} \right\} \cap \left\{ c_q \,\middle|\, c_q = \overrightarrow{q} \right\} \neq \emptyset$, dann muß gelten $\exists (q,\delta,p') \in T : p' \neq q_0$, was in Algorithmus 7 jedoch ausgeschlossen wird. □

Die Anzahl der zwecks Legitimierung einer benötigten Transition zu ergänzenden Konflikte hängt von der Anzahl vorangehender und nachfolgender Transitionsverzweigungen im Automaten ab und möglicherweise kann das Ziel des Zusammenfassens der betreffenden Zustände nicht erreicht werden. Dennoch ist es sinnvoll, die bereits gefundenen zusätzlichen Konflikte beizubehalten, da diese aufgrund ihrer Konstruktion sowohl im Präfix- als auch im Suffixteil wesentlich an bereits existierende Konflikte angelehnt sind und mit hoher Wahrscheinlichkeit, wenn überhaupt, nur sehr wenige neue Zustände für diese Konflikte nötig waren.

Aus dieser Überlegung heraus wenden wir ein in Algorithmus 8 dargelegtes, in den bisherigen Gesamtprozess eingebettetes Fixpunktiterationsverfahren an, welches solange alternierend minimiert und neue Konflikte ergänzt, bis keine weiteren zusätzlichen Konflikte mehr gefunden werden können.

$M := \{Q \backslash F\}$
$C_{new} := \emptyset$
$pmd := d$
foreach $M_k \in M$ **do**
  foreach $q_i, q_j \in M_k, q_i \neq q_k$ **do**
    if $\forall p \in Q, \delta \in \Sigma : \exists(q_i, \delta, p) \in T \Leftrightarrow \exists(q_j, \delta, p) \in T$ **then**
      $pmd := updatemap\left(pmd, \left(q_i, max\left(pmd(q_i), pmd(q_j)\right)\right)\right)$
      $T := T \cup \left\{(p, \delta, q_i) \middle| \exists(p, \delta, q_j) \in T\right\}$
      $T := T \setminus \left(\left\{(p, \delta, q_j) \in T\right\} \cup \left\{(q_j, \delta, p) \in T\right\}\right)$
      $F := F \setminus \{q_j\}$
      $Q := Q \setminus \{q_j\}$
      $M := (M \setminus \{M_k\}) \cup \left\{p \middle| \exists \delta \in \Sigma : (p, \delta, q_i) \in T\right\}$
      else
        if $\{q_i, q_j\} \notin N$ **then**
          $C' := \emptyset$
          $C' := C' \cup \bigcup_{(q_j, \delta, p) \in T, d(q_i) < pmd(p), tgrt(q_i, \delta) \neq \perp} \left\{\overleftarrow{q_i} \cdot (\delta) \cdot \vec{p}\right\}$
          $C' := C' \cup \bigcup_{(q_i, \delta, p) \in T, d(q_j) < pmd(p), tgrt(q_j, \delta) \neq \perp} \left\{\overleftarrow{q_j} \cdot (\delta) \cdot \vec{p}\right\}$
          if $C' \neq \emptyset \wedge \forall c \in C' : \mathcal{L}(c, X) = \emptyset$ **then**
            $C_{new} := C_{new} \cup C'$
          else
            $N := N \cup \left\{\{q_i, q_j\}\right\}$
  end
end
return $((Q, \{q_0\}, \Sigma, T, F), C_{new}, N)$

Algorithmus 7: Minimierung des Automaten $(Q, \{q_0\}, \Sigma, T, F)$ mit Berechnung von zusätzlichen Konflikten, welche zu weiteren Minimierungen führen (können). Eine bereits gescheiterte Minimierungsversuche protokollierende Menge $N \subseteq 2^Q$ wird berücksichtigt und gegebenenfalls erweitert. Neben dem minimierten Automaten wird eine Menge C_{new} zusätzlicher Konflikte und die Menge N zurückgegeben.

Für jeweils zwei Zustände q_i und q_j wird die Generierung neuer Konfliktkandidaten in der Iteration vorzeitig abgebrochen, sobald einer der erzeugten Sequenzen nicht als Konflikt bestätigt werden konnte, da in diesem Falle davon auszugehen ist, daß die vermisste Transition nicht ergänzt werden kann und darf. Dies geschieht wiederum im Minimierungsverfahren, in dem gescheiterte Zustandspaarungen durch zwei-elementige Teilmengen von Q in einer Menge N protokolliert werden, wodurch ein wiederholter Minimierungsversuch bezüglich dieser Zustände ausgeschlossen werden kann.

4.7.1.3 Instanziierung regulärer Ausdrücke

Wir haben bislang keine Sequenzen für angestrebte Transitionen (q, δ, p) erzeugt, für die $d(q) < pmd(p)$ gilt. Wie in Kapitel 4.3.3.2 geschildert, sind Zwischenkonflikttransitionen zu Zuständen kürzerer Distanz zum Initialzustand aufgrund von Teilpräfixübereinstimmung vorhanden, wobei der Zielzustand aufgrund seines kürzeren Präfixes auch weniger Information über die Historie eines Laufs „besitzt". Der Versuch, eine Zwischenkonflikttransition ohne Teilpräfixübereinstimmung etablieren zu wollen, hat daher einen besonderen Hintergrund und eröffnet neue Möglichkeiten.

$A_{C_R} \coloneqq (\{q_{\Re_0}, \mathit{fin}\}, \{q_{\Re_0}\}, \Sigma, \{(q_{\Re_0}, \delta, \mathit{fin}) \mid \delta \in \Sigma\}, \{q_{\Re_0}, \mathit{fin}\})$
$A_{C_\omega} \coloneqq (\{q_{\omega_0}\}, \{q_{\omega_0}\}, \Sigma, \{(q_{\omega_0}, \delta, q_{\omega_0}) \mid \delta \in \Sigma\}, \{q_{\omega_0}\})$
$A \coloneqq A_0$
while $A \not\models \mathbf{AG}\neg\hat{S}_U$ **do** % Model-Checker Lauf
  $\hat{\pi} \coloneqq (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n), \hat{s}_n \in \hat{S}_U$ % Pfad vom Model-Checker
  $(\mathit{result}, (x_0, x_1, \ldots, x_n)) \coloneqq \tilde{\mathcal{L}}'(\theta(\hat{\pi}), X_0)$
  if $\mathit{result} = \mathit{false}$ **then** % ungültiges Gegenbeispiel
    $C_{ini} \coloneqq r_{\mathit{ini}}(\theta(\hat{\pi}))$
    $C_{inv} \coloneqq r_{\mathit{inv}}(\theta(\hat{\pi}))$
    $N_{ini} \coloneqq \emptyset$
    $N_{inv} \coloneqq \emptyset$
    do
      foreach $c \in C_{ini}$ **do** $A_{C_\Re} \coloneqq \mathit{Add}_\Re(A_{C_\Re}, c)$ **end**
      foreach $c \in C_{inv}$ **do** $A_{C_\omega} \coloneqq \mathit{Add}_\omega(A_{C_\omega}, c)$ **end**
      $\left(A'_{C_\Re}, C_{ini}, N_{ini}\right) \coloneqq \mathit{Minimize}\left(\mathit{strip}(A_{C_\Re}), N_{ini}\right)$
      $\left(A'_{C_\omega}, C_{inv}, N_{inv}\right) \coloneqq \mathit{Minimize}\left(\mathit{strip}(A_{C_\omega}), N_{inv}\right)$
    while $C_{ini} \neq \emptyset \vee C_{inv} \neq \emptyset$
    $A \coloneqq \left(A'_{C_\Re} \times A'_{C_\omega}\right) \times A_0$
  else return $\pi \coloneqq \left(\left(\tilde{\alpha}^{-1}(\hat{s}_0), x_0\right), \left(\tilde{\alpha}^{-1}(\hat{s}_1), x_1\right), \ldots, \left(\tilde{\alpha}^{-1}(\hat{s}_n), x_n\right)\right)$ % gültiger Pfad
end
return *true* % $TS_H \models \mathbf{AG}\neg S_U$

Algorithmus 8: Gesamter iterativer Abstraktionsverfeinerungsprozess, ergänzt um erweiterte Minimierung. Es wird entweder das Ergebnis *true* oder ein Pfad $\pi = (s_0, \ldots, s_n) \in \overrightarrow{TS_H}, s_n \in S_U$ zurückgegeben, wobei $\overrightarrow{TS_H}$ die Menge aller Pfade in TS_H darstellt.

Die Funktion *pmd* wird benötigt, da die Minimierung die Distanzen und damit die <-Relation auf Zuständen durcheinander bringt, eine Klassifikation von Zwischentransitionen damit auf dieser Basis nicht mehr möglich ist.

Betrachten wir dazu abermals das Beispiel in Abbildung 4.9. In der Minimierung werden aufgrund ihrer jeweiligen Transitionen zum nicht akzeptierenden Zustand q_{12} die Zustände q_5 und q_{11} als potentiell zusammenfassbare Zustände behandelt. Alle ausgehenden Transitionen sind gleich, bis auf die Transition (q_5, c, q_{11}), für die die entsprechende Transition (q_{11}, c, q_{11}) nicht existiert, eine Zusammenfassung daher nicht möglich ist. Ohne die Einschränkung $d(q) < \mathit{pmd}(p)$ bei der Zusammenstellung neuer Sequenzen wird durch Algorithmus 7 die Sequenz $\overleftarrow{q_5} \cdot (c) \cdot \overrightarrow{q_{11}} = (a, c, c, c, c, c, c, d)$ erzeugt. Dies ist in soweit bemerkenswert, als daß diese Sequenz länger ist als jede bisherige, die der Automat überwacht. Das Vorhandensein eines Schemas, welches im vorliegenden Beispiel durch den regulären Ausdruck $ac^j d$ mit $1 \leq j \leq 5$ beschrieben werden kann, führt zu einer automatischen Verlängerung der Sequenz durch Reinstantiierung der sich wiederholenden Teilausdrücke.

Dieser Vorgang funktioniert gleichermaßen auf Sequenzen, die durch reguläre Ausdrücke der Form $\ldots(\delta_i \ldots \delta_k)^j \ldots$ mit einem längeren Wiederholungsteilwort beschrieben sind, wie das Beispiel in Abbildung 4.10 zeigt. Eine solche Situation entsteht immer, sobald zwei aufeinanderfolgende Instanzen des regulären Ausdrucks in dem Automaten präsent sind, da nach der Minimierung eine Transition zum gemeinsamen Suffix vorhanden ist. Die Zustände q_i und q_k könnten zusammengefasst werden, wenn die gestrichelt dargestellten Transitionen (q_i, δ, q_{k+1}) und (q_k, δ_i, q_{i+1})

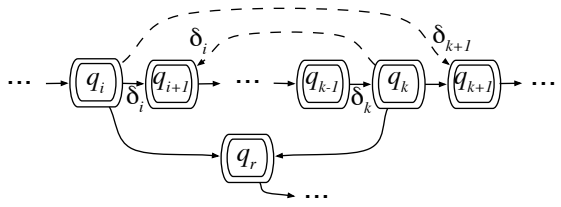


Abbildung 4.10: Erweiterte Minimierung zur Verlängerung von Sequenzen, die, als regulärer Ausdruck beschrieben, sich wiederholende Teilwörter beinhalten. Dargestellte Transitionsmenge ist unvollständig

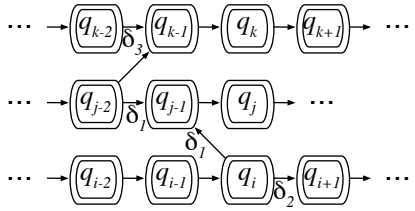


Abbildung 4.11: Erweiterte Minimierung über Zwischenkonflikttransitionen, die keine Zustände des eigenen Präfixes anspringen. Dargestellte Transitionsmenge ist unvollständig. Entfernung zum Zustand $\perp$ von links nach rechts ansteigend, wobei vertikal ausgerichtete Zustände die gleiche Entfernung aufweisen

vorhanden wären. Die Sequenzgenerierung der erweiterten Minimierung generiert somit aus $\ldots(\delta_i \ldots \delta_k)^1 \ldots$ als nächstes den Ausdruck $\ldots(\delta_i \ldots \delta_k)^2 \ldots$ usw.. Die Sequenzgenerierung im Rahmen des Fixpunktiterationsverfahrens innerhalb der Minimierungsphase terminiert erst dann, wenn erstmals eine der Instanzen der regulären Ausdrücke kein Konflikt mehr ist. Ist jedoch jede der Instanzen ein Konflikt, würde diese Sequenzgenerierung ohne entsprechende Begrenzung nicht terminieren.

Bislang haben wir im Rahmen der Aufhebung von Einschränkung $d(q) < pmd(p)$ nur solche Sequenzen betrachtet, welche eine Transition (q, δ, p) in Zustände des eigenen Präfixes p zu ergänzen versuchen, für die also $\exists i < d(q) : \rho_{\overleftarrow{q}}(i) = p$ gilt. Dadurch kamen die verlängerten Instanzen von regulären Ausdrücken zustande, welche als Muster in Konflikten vorhanden sind.

Betrachten wir nun die Generierung solcher Sequenzen, für die das nicht zutrifft, wie in Abbildung 4.11 dargestellt. Als Vorgängerzustände von q_{j-1} wären die Zustände q_{j-2} und q_i Kandidaten für eine Zusammenfassung, wobei jedoch die Transitionen $(q_{j-2}, \delta_2, q_{i+1})$ und (q_i, δ_3, q_{k-1}) fehlen.

- Sequenzen für eine Transition $t_1 \coloneqq (q_{j-2}, \delta_2, q_{i+1})$:
  Es gilt $\exists w \in \Sigma^{(d(q_i)-d(q_{j-2}))} : \overleftarrow{q_i} = w \cdot \overleftarrow{q_{j-2}}$ nach Invariante 15, da die Transition (q_i, δ_1, q_{j-1}) eine Zwischentransition ist mit $q_{j-1} < q_i$. Wenn die über die angenommene Transition t_1 induzierte Sequenz $\overleftarrow{q_{j-2}} \cdot (\delta_2) \cdot \overrightarrow{q_{i+1}}$ ein Konflikt wäre, so wäre der im Automaten behandelte Konflikt $w \cdot \overleftarrow{q_{j-2}} \cdot (\delta_2) \cdot \overrightarrow{q_{i+1}} = \overleftarrow{q_{i+1}} \cdot \overrightarrow{q_{i+1}}$ nicht minimal und durch Algorithmus 2 weiter reduziert worden. Da dies durch Algorithmus 2 jedoch nicht geschehen ist, ist $\overleftarrow{q_{i+1}} \cdot \overrightarrow{q_{i+1}}$ nicht weiter minimierbar und die Sequenz $\overleftarrow{q_{j-2}} \cdot (\delta_2) \cdot \overrightarrow{q_{i+1}}$ kein Konflikt.
- Sequenzen für eine Transition $t_2 \coloneqq (q_i, \delta_3, q_{k-1})$:

Aufgrund von Invariante 15 gilt bereits $t_2 \in T$, die Transition kann also nicht fehlen. Entsprechend werden auch keine Sequenzen generiert.

Wie an diesen beiden repräsentativen Transitionen ermittelt, ist eine Sequenzgenerierung in diesen Fällen sinnlos, da etwaige Sequenzen keine Konflikte sein können. Ein effizienterer Algorithmus sollte diesbzgl. Transitionen wie t_1 entsprechend klassifizieren und nicht verwenden. Da eine entsprechende, dennoch versuchte Sequenzgenerierung jedoch sowohl im Ergebnis, als auch vom Aufwand unkritisch ist, verzichten wir an dieser Stelle auf eine Verfeinerung des Algorithmus.

Die im Folgenden betrachtete Einsatzstrategie bezieht sich damit wieder auf die instantiierten regulären Konfliktmuster.

Einsatzstrategie Durch Instantiierung regulärer Ausdrücke erzeugte Sequenzen, sofern als Konflikt bestätigt, setzen kein Minimierungspotential frei, sondern fügen dem Automaten jeweils einen neuen Zustand q hinzu und sind daher nur sinnvoll, wenn damit künftige Iterationen der Abstraktionsverfeinerung eingespart werden können. Dies wiederum ist stark modellabhängig, da die erzeugten Sequenzen durchaus irrelevant sein können, da z.B. der Model-Checker ohnehin andere (kürzere) Pfade errechnet, oder aufgrund der diskreten Struktur des Modells diese nicht existieren. Je nach Modellcharakteristik bieten sich drei verschiedene Strategien an:

1. Verzicht auf verlängerte Sequenzgenerierung, d.h. Beibehaltung der Einschränkung $d(q) < pmd(p)$ in Algorithmus 7.

2. Generierung von verlängerten Sequenzen c nur bis zur Länge $|c| \leq max(C_\perp)$ mit $max(C_\perp)$ als die maximale Länge der bekannten Konflikte.

3. Generierung von verlängerten Sequenzen c bis zur Länge $|c| \leq |\hat{\pi}|$.

Da bei Strategie 1 der Automat nicht vergrößert wird, sollte diese als Standard herangezogen werden.

Wenn hingegen innerhalb der iterativen Abstraktionsverfeinerung jeweils mehrere Iterationen benötigt werden, bis die Länge der vom Model-Checker im Rahmen von Gegenbeispielen ermittelten Pfade $\hat{\pi}$ sich um einen Schritt verlängert, und der Quotient aus Iterationen pro Verlängerungschritt der Pfade annähernd gleich ist, dann liegt mit großer Wahrscheinlichkeit eine *Konfliktsystematik* vor, die, wie oben dargestellt, durch reguläre Ausdrücke beschrieben werden kann. Bei Vorliegen einer Konfliktsystematik kann eine Teilmenge aller Konflikte eines Hybriden Systems durch eine Menge $R = \{r_1, \ldots, r_n\}$ regulärer Ausdrücke beschrieben werden. Es wird in jeder Iteration jeweils nur ein Konflikt ermittelt, der durch eine neue Instanziierung einer dieser Ausdrücke beschrieben werden kann. Daher werden mindestens $n = |R|$ Iterationen benötigt, bis für jeden regulären Ausdruck $r_i \in R$ die um 1 verlängerte Instanz gefunden wird. In diesem Fall bietet sich Strategie 2 an, da bei der Entdeckung der ersten, um 1 verlängerten Konfliktinstanz sich die maximale Länge $max(C_\perp)$ der Konflikte ebenfalls um 1 erhöht, woraufhin die durch die anderen regulären Ausdrücke aus R beschrieben Konflikte noch in derselben Iteration „nachziehen" können, mithin also $n = |R|$ Iterationen in jeder Iteration eingespart werden können. Dies funktioniert insbesondere bei „synchronisierten" Konflikten die sich nur in einem Präfix gleicher Länge unterscheiden.

Als Beispiel hierfür können wir erneut den Automaten aus Abbildung 4.9 heranziehen. Hier scheint es sich um zwei reguläre Muster zu handeln, ac^id und bc^id. Voraussetzend, daß alle Sequenzen dieses Musters für alle $i \in \mathbb{N}$ auch Konflikte sind und es keinen Pfad an diesen Sequenzmustern vorbei gäbe, würde Strategie 1 jeweils zwei Iterationen benötigen, um den Pfad $\hat{\pi}$ um einen Schritt zu verlängern.

Strategie 2 benötigt hingegen jeweils nur einen Schritt, da die Entdeckung des Konflikts $ac^{i+1}d$ die Berücksichtigung von $bc^{i+1}d$ im Rahmen der damit inkrementierten maximalen Konfliktlänge ermöglicht.

Da die Konflikte in einem, vom Model-Checker ermittelte Pfad $\hat{\pi}$ notwendigerweise kürzer als dieser sind, verkörpert Strategie 3 ein ausgesprochen aggressives Vorgehen, welche die Verlängerung der Konfliktsequenzen um mehrere Schritte pro Iteration erlaubt. Letztlich können hier auch größere Begrenzungswerte verwendet werden, soweit sie proportional zur Pfadlänge sind. Andernfalls könnte die Pfadlänge diese Zahl übersteigen und möglicherweise auch die maximale Konfliktlänge, wodurch de facto wieder Strategie 1 angewandt würde.

4.7.2 Verwendung von GJ-Teilmengensequenzen

In diesem Abschnitt wird eine Erweiterung des Verfahrens beschrieben, in der durch Ausschluß von Sequenzen von Mengen von GJ-Paaren (*Teilmengensequenzen*) anstelle von Sequenzen von GJ-Paaren Konfliktmengen anstelle von einzelnen Konflikten ausgeschlossen werden. Diese Teilmengensequenzen bezeichnen wir daher auch als *verallgemeinerte Konflikte*. Diesen zugrunde liegt die Beobachtung, daß häufig mehrere, als GJ-Sequenzen repräsentierte Konflikte sich nur in einem GJ-Paar unterscheiden.

Ein konkretes Beispiel ist bei Vorhandensein mehrerer reelwertiger Variablen gegeben, welche für den jeweiligen Konflikt irrelevant sind. Betrachten wir dazu die folgenden drei GJ-Paare: $a = (\{(x,y)\,|x<0\}, ((x,y)\mapsto(x+1,y)))$, $b = (\{(x,y)\,|x<0\}, ((x,y)\mapsto(x+1,y+1)))$, $c = (\{(x,y)\,|x>2\}, ((x,y)\mapsto(x,y)))$. Offensichtlich ergeben sich hieraus die invarianten Konflikte (a,c) und (b,c), wobei die zur Unlösbarkeit beitragenden Umstände dieser Folgen identisch sind. In beiden Fällen ist die Teilformel $x<0 \wedge x+1>2$ nicht lösbar, und es ist irrelevant, ob die Variable y darüberhinaus eine Änderung erfährt, oder nicht. Bzgl. dieser beiden Konflikte können die GJ-Paare a und b demnach zu einer Gruppe $P=\{a,b\}$ zusammengefaßt werden, so daß wir anstelle von (a,c) und (b,c) nur noch einen Konflikt (P,c) erhalten, der auszuschliessen ist.

Anstelle des direkten Ausschlusses von GJ-Sequenzen kann demnach der Grad der Verallgemeinerung erhöht werden durch die Einführung von GJ-Teilmengensequenzen, definiert als $C^{\subseteq} \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} \left(2^{G\times J}\right)^n$. Allgemein gibt es Mengen von GJ-Paaren $P \subseteq G\times J$, so daß ausgehend von einem Konflikt c gilt: $C^{\subseteq}$

$$\exists k\in\mathbb{N}, P\subseteq G\times J : \forall(\gamma',\zeta')\in P : \mathcal{L}\left(c_{[(k)\setminus(\gamma',\zeta')]}, X\right) = \emptyset$$

Dabei greifen wir auf die in Abschnitt 4.2 definierte Notation zur Substitution einzelner Elemente in einer Sequenz zurück, d.h. $c_{[(k)\setminus(\gamma',\zeta')]}$ ist die Sequenz c, in der das k-te Element durch (γ',ζ') ersetzt wird.

Das k-te GJ-Paar der den Konflikt c repräsentierenden GJ-Sequenz kann also durch alle Elemente einer Menge P ausgetauscht werden, so daß alle hieraus erzeugten GJ-Sequenzen ebenfalls Konflikte sind. Es gibt demnach eine Menge von $|P|$ *ähnlichen* Konflikten, die sich nur in diesem einen Element unterscheiden. Unter Umständen gibt es nun für jedes i mit $1\le i\le n$ jeweils eine Menge P_i, die, als Austauschmenge für das i-te GJ-Paar $c_{(i)}$ verwendet, alle so erzeugten GJ-Sequenzen als Konflikt generiert. Für eine GJ-Teilmengensequenz $(P_1,P_2,\ldots,P_n)$ wären dies also $\prod_{1\le i\le n}|P_i|$ verschiedene GJ-Sequenzen. Wenn in ungünstigen Fällen das iterative Verfeinerungsverfahren viele dieser Konflikte einzeln pro Iterationsschritt auszuschließen hat, bevor das gewünschte Ergebnis ermittelt werden kann, so ist in solchen Fällen eine entsprechend große Zahl von Iterationen nötig.

Soweit die Mengen P_i jedoch aufgrund eines oder weniger Repräsentanten bestimmt werden können, kann ein Ausschluß der Teilmengensequenz alle bein-

halteten Konflikte vollständig abdecken. Der ω-Automat hat somit eine Sequenz $\langle(\gamma_n, \zeta_n)\rangle \in C$ immer genau dann abzulehnen, wenn es einen verallgemeinerten Konflikt in Form einer Teilmengensequenz $(P_1, P_2, \ldots, P_n)$ gibt, so daß $(\gamma_i, \zeta_i) \in P_i$.

Nachdem wir nun verallgemeinerte Konflikte durch Verwendung von Teilmengensequenzen motiviert durch ähnliche Konflikte eingeführt haben, soll die ω-Automatenkonstruktion im nächsten Abschnitt zum Ausschluß von Teilmengensequenzen erweitert werden. Im Anschluss daran wird eine Strategie zur Teilmengenbildung vorgestellt.

4.7.2.1 Determinisierung des ω-Automaten

Die Konstruktion eines ω-Automaten für den Ausschluß von Teilmengensequenzen unterscheidet sich notwendigerweise von dem bisherigen Konstruktionsverfahren, da zwar die vorgegebenen Teilmengen $P \subseteq 2^{GJ}$ bzgl. eines zu verarbeitenden Zeichen $(\gamma, \zeta) \in \Sigma = GJ$ durch $(\gamma, \zeta) \in P$ einfach bestimmt werden können, diese jedoch nicht notwendigerweise eindeutig sind, da $P \cap P' \neq \emptyset$ für $P \neq P'$ möglich ist und somit ein zu verarbeitendes GJ-Paar (γ, ζ) sowohl in P als auch in P' liegen kann.

Der ω-Automat konstruiert auf Basis von GJ-Teilmengensequenzen, d.h. mit einem Alphabet $\Sigma' \subseteq 2^{G\times J}$, besitzt daher Transitionen, die nicht mehr durch die zu verarbeitenden GJ-Paare $(\gamma, \zeta) \in \Sigma = GJ$ determiniert sind, der Automat ist somit also nichtdeterminististisch. Das ω-Cegar Verfahren setzt jedoch deterministische Automaten voraus, und so muß dieser vor der Verwendung entsprechend determinisiert werden. Da sich der ω-Automat A_C aus einem regulären Automaten $A_{C_\Re}$ und einem ω-Automaten A_{C_ω} zusammensetzt, wird die im folgenden beschriebene Determinisierung für $A_{C_\Re}$ und A_{C_ω} jeweils einzeln vor der Bildung des Kreuzprodukts $A_C = A_{C_\Re} \times A_{C_\omega}$ eingeschoben. Da die Determinisierung sich für $A_{C_\Re}$ und A_{C_ω} nicht unterscheidet, sei die folgende Darstellung diesbezüglich nicht differenziert und die Automaten mit $A_{C'}$ bezeichnet.

Σ'

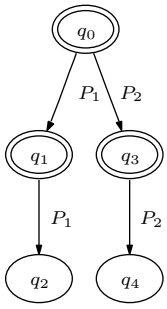
Analog zur klassischen Potenzmengenkonstruktion zur Determinisierung nichtdeterministischer Automaten benutzen wir den nach dem bisherigen Verfahren unter Verwendung des Teilmengen-Alphabets $\Sigma' = 2^\Sigma$ konstruierten Automaten $A_{C'} = (Q, q_0, \Sigma', T, F)$ zur Konstruktion eines determinisierten Automaten $A^*_{C'}$. Anschaulich gesehen verwenden wir A_C als gerichteten Graphen mit Knoten (Zustände) und attributierten Kanten (Transitionen). Jedem Knoten $q_j \in Q = \{q_0, q_1, \ldots, q_n\}$ wird ein Element $b_j \in \mathbb{B}$ zugeordnet, welches sich durch die Funktion $b^* : \{0, \ldots, n\} \times GJ \times \mathbb{B}^n \to \mathbb{B}$ mit

$A^*_{C'}$

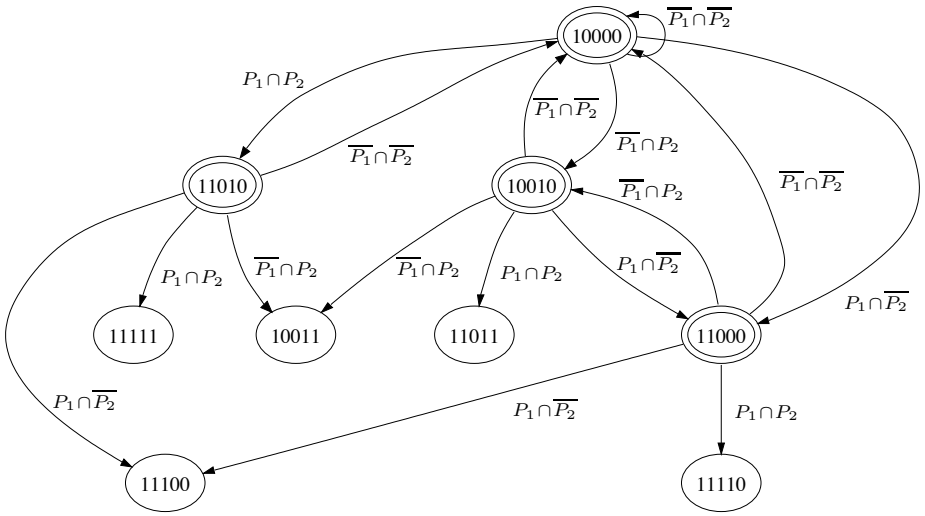
$$b^*(j, \delta, (b_0, b_1, \ldots, b_n)) = \begin{cases} 1 & \text{gdw } j = 0 \\ \bigvee_{q_i \in Q:(q_i, \delta', q_j) \in T, [q_j]=j, \delta \in \delta', q_i < q_j} b_{[q_i]} & \text{gdw } j > 0 \end{cases}$$

unter Verwendung einer Funktion $[] : Q \to \mathbb{N}$ mit $[q_i] = i$ berechnet. Der Automat $A^*_{C'} = \left(Q^*, q^*_0, \Sigma, T^*, F^*\right)$ wird dann folgendermaßen konstruiert:

- $Q^* := \{(1, b_1, b_2, \ldots, b_n) | b_i \in \mathbb{B}\}$
- $q^*_0 := (1, 0, ..., 0)$,
- $\Sigma = GJ$,
- $T^* := \left\{\left((b_0, \ldots, b_n), \delta, (b'_0, \ldots, b'_n)\right) \in Q^* \times \Sigma \times Q^* | b'_j = b^*(j, \delta, (b_0, \ldots, b_n))\right\}$
- $F^* := \left\{(1, b_1, \ldots, b_k, \ldots, b_n) \in Q^* \,\middle|\, \forall k \in \mathbb{N}, p \in Q \setminus F : [p] = k \implies b_k = 0\right\}$



(a) Nicht-deterministischer Automat $A_{C'}$. Dargestellt sind nur Transitionen (p, δ, q) für die $p < q$ gilt



(b) Deterministierter Automat $A^*_{C'}$

Abbildung 4.12: Beispiel einer Determinisierung eines ω-Automaten, der die Teilmengensequenzen (P_1, P_1) und (P_2, P_2) ausschließt. Die Alphabete sind $\Sigma' = \{P_1, P_2\} \subseteq 2^{GJ}$ und $\Sigma = GJ$, wobei jedes $\delta \in \Sigma$ in genau einer der für die Beschriftung verwendeten Mengen $P_1 \cap P_2$, $\overline{P_1} \cap \overline{P_2}$, $P_1 \cap \overline{P_2}$ oder $\overline{P_1} \cap P_2$ liegt

Dieser Automat besitzt $2^{|Q|-1}$ Zustände, ist damit sehr viel größer als $A_{C'}$ und außerdem nicht mehr minimal. Die Determinisierung wird für die Automaten $A_{C_\Re}$ und A_{C_ω} getrennt durchgeführt, bevor diese zu A_C komponiert werden, so daß die Größe des Zustandsraums Q_C von A_C bei Verwendung dieser Determinisierung durch $|Q_C| = 2^{|Q_\Re|+|Q_\omega|}$ beschrieben ist, wobei $Q_\Re$ und Q_ω die Zustandsmengen von $A_{C_\Re}$ und A_{C_ω} vor deren Determinisierung darstellen.

Wir haben das Automatenkonstruktionsverfahren somit jetzt zur Verwendung nicht-disjunkter Teilmengen als Zeichen eines Alphabets $\Sigma' = 2^{G \times J}$ erweitert und können im folgenden Abschnitt die Bestimmung geeigneter Teilmengen erläutern.

4.7.2.2 Dekomposition von Guard Sets und Jump Set Funktionen

Für eine Verallgemeinerung von Konflikten benötigen wir eine Gruppierung von GJ-Paaren zu Teilmengen P von $G \times J$. Für die Gruppierung gibt es die Möglichkeit, P allein auf Basis von beobachteten Konflikten im Rahmen der Iterationen des Verfahrens zu berechnen. Dies ist jedoch ein aufwendiges Unterfangen, da ohne Zuhilfenahme zusätzlicher Informationen eine große Anzahl von Kombinationen von GJ-Paaren als GJ-Sequenzen hinsichtlich einer möglichen Konflikteigenschaft

untersucht werden müssten, um entsprechende Gruppen spezifizieren zu können. Darüber hinaus wären solche Gruppierungen bei diesem Vorgehen erst nach vielen Iterationen definierbar, so daß erst ab einer gewissen Anzahl von Iterationen eine Verallgemeinerung von Konflikten ermöglicht würde.

Es gibt jedoch die Möglichkeit, eine Voranalyse der Guard Set Mengen G und Jump Set Funktionen J durchzuführen, welche bereits eine effektive Gruppierung in Mengen $P_1, P_2, \ldots, P_k$ erlaubt. Für Jump Set Funktionen ist dies durch eine Zuordnung selbiger zu ihren linear dekomponierten, ebenfalls endlichen Projektionen auf die einzelnen Dimensionen des kontinuierlichen Zustandsraums möglich, während die Guard Sets durch die in der konkreten Repräsentation des Hybriden Systems verwendeten Ausdrücke zu gruppieren sind. Letzteres ist auf der konzeptionellen Beschreibungsebene dieses Kapitels nicht ersichtlich, so daß wir dem Leser die Grundlage der Einteilung bis zum Kapitel 6 schuldig bleiben müssen. Da die Verwendung einer *Systematik* von GJ-Paar-Mengen $P_1, P_2, \ldots, P_k$ jedoch ein wichtiger Bestandteil des ω-CEGAR Verfahrens ist, kann auf eine konzeptionelle Beschreibung an dieser Stelle nicht verzichtet werden. Wir betrachten daher zunächst einige Voraussetzungen für die Verwendung der Mengen P, die wir durch Repräsentationen $(\tilde{g}, \tilde{j})$ beschreiben wollen, welche sich auf P projizieren lassen. Hintergrund ist hierbei, daß eine explizite Aufzählung einzelner GJ-Paare zwecks Skalierbarkeit vermieden werden soll. Im Anschluss entwickeln wir eine Systematik für Mengen von Guard Sets, gefolgt von einer Systematik für Jump Set Funktionen unter Verwendung einer vektoriellen linearen Dekomposition selbiger bzgl. einer Orthonormalbasis des kontinuierlichen Zustandsraums.

Kommen wir zunächst zu Repräsentation von Teilmengen $P \subseteq G \times J$, welche folgende Eigenschaften für eine effiziente Benutzbarkeit erfüllen müssen:

$\tilde{P}, \tilde{G}, []$

- Es wird eine kompakte Repräsentation $(\tilde{g}, \tilde{j})$ der Teilmengen benötigt, welche sich effizient auf die repräsentierte Menge $P = \tilde{G} \times \tilde{J}$ der beschriebenen GJ-Paare abbilden läßt, formal $[] : (\tilde{g}, \tilde{j}) \mapsto \left\{(\gamma, \zeta) \middle| \gamma \in \tilde{G}, \zeta \in \tilde{J}\right\}$.

- Insbesondere müssen durch $(\tilde{g}, \tilde{j})$ auch ein-elementige Teilmengen $\{(\gamma, \zeta)\} \subseteq GJ$ darstellbar sein, um bezüglich Gleichung (4.2) die gleiche Ausdrucksmächtigkeit zu ermöglichen, d.h. jede beliebige GJ-Sequenz ausschließen zu können, ohne andere GJ-Sequenzen ungewollt mit auszuschließen. Die Zuordnung eines GJ-Paares zu seiner Repräsentation erfolgt durch eine Funktion $convert : (\gamma, \zeta) \mapsto \left(\tilde{g}_\gamma, \tilde{j}_\zeta\right)$ mit $\left(\tilde{g}_\gamma, \tilde{j}_\zeta\right) \mapsto \{(\gamma, \zeta)\}$, wobei $convert$ auch komponentenweise definiert ist, d.h. $convert : \gamma \mapsto \tilde{g}_\gamma$ und $convert : \zeta \mapsto \tilde{j}_\zeta$.

Darüber hinaus ist aus Effizienzgründen die folgende Eigenschaft wünschenswert:

$\tilde{\gamma}_{\tilde{G}}, \tilde{\zeta}_{\tilde{J}}$

- Für jede zu verwendende Guard Set Menge $\tilde{G} \subseteq G$ muß jeweils eine einzelne Ersatz-Guard-Set Menge $\tilde{\gamma}_{\tilde{G}} \subseteq X$ angegeben werden können (trivial), sowie für jede zu verwendende Jump Set Funktionsmenge $\tilde{J} \subseteq J$ jeweils eine einzige abstrakte Ersatzfunktion $\tilde{\zeta}_{\tilde{J}} : X \to 2^X$.

Eine dementsprechende Gruppierung von GJ-Paaren zu GJ-Teilmengen wird durch eine Dekomposition sowohl von Guard Sets, als auch von Jump Set Funktionen erreicht. Im Folgenden wird zunächst eine Zerlegung der Guards Sets und im Anschluß daran eine Zerlegung der Jump Set Funktionen vorgestellt.

Guard Sets Alle Guard Sets $\gamma \in G$ lassen sich als Schnittmenge von m Mengen $\gamma_1^{u_1}, \gamma_2^{u_2}, \ldots, \gamma_m^{u_m} \subseteq X, u_i \in \mathbb{N}$ beschreiben:

$$\gamma = \bigcap_{1 \leq i \leq m} \gamma_i^{u_i}$$

	Guard Set Zerlegung			
	1	2	…	m
G	G_1	G_2	…	G_m
	X	X	…	X
γ_1	γ_1^1	γ_2^1	…	γ_m^1
γ_2	γ_1^2	γ_2^2	…	γ_m^1
⋮	⋮	⋮	⋮	⋮
⋮	⋮	$\gamma_2^{g_2}$	…	⋮
⋮	$\gamma_1^{g_1}$			⋮
⋮				$\gamma_m^{g_m}$
$\gamma_{\lvert G\rvert}$				

Tabelle 4.2: Systematische Übersicht über die Zerlegung von G. Dabei ist $d_i = |D_i|$. Jedes γ_j ist Schnittmenge von Mengenkombinationen aus jeweils einer Menge $\gamma_i^p \in G_i$

Wir bezeichnen die Mengen $\gamma_i^{u_i}$ als *Guard-Komponenten*. Es gibt, wie im systematischen Überblick in Tabelle 4.2 gezeigt, dabei für $i \in \{1, \ldots, m\}$ Mengen $G_i \coloneqq \left\{\gamma_i^u \middle| 1 \leq u \leq g_i\right\}$, aus denen jeweils eine Guard-Komponente für die Schnittmengenbildung ausgewählt werden kann. Die Berechnung der systematischen Zerlegung ist Modell-spezifisch und später näher in Abschnitt 6.3.3 beschrieben, wie bereits angedeutet wurde[4]. Jede Menge G_i enthält insbesondere das Element X und kann somit für die obige Schnittmengenbildung aus einer Anzahl von m Mengen als neutral betrachtet werden.

Jede Guard-Komponente γ_i^u kann abgebildet werden auf die Menge $P_{\gamma_i^u}$ all der Guard Sets, welche Teilmenge von γ_i^p sind, formal $\gamma_i^u \mapsto P_{\gamma_i^u} \stackrel{\text{def}}{=} \left\{\gamma \in G \middle| \gamma \subseteq \gamma_i^u\right\}$. Somit „selektiert" eine Guard-Komponente γ_i^u eine größere Menge von Guards Sets aus G.

Eine Menge von Guard Sets $\tilde{G} = \left\{\gamma_1, \gamma_2, \ldots, \gamma_{|\tilde{G}|}\right\} \subseteq G$, welche insbesondere auch ein-elementig sein kann, wird mit dieser Schnittmengenzerlegung also durch Angabe eines Mengenvektors der Länge m beschrieben mit $\tilde{g} \coloneqq \left(G_1', G_2', \ldots, G_m'\right), G_i' \subseteq G_i, 1 \leq i \leq m$ durch $\left\{\gamma \middle| \gamma \subseteq \bigcap_{1 \leq i \leq m} \bigcup_{\gamma_i^{u_i} \in G_i'} \gamma_i^{u_i}\right\}$. Somit ist das *Ersatz-Guard-Set* $\tilde{\gamma}_{\tilde{G}}$ beschrieben durch eine Funktion $[] : \left(2^X\right)^m \to 2^X$ mit

$\tilde{g}$

$\tilde{\gamma}_{\tilde{G}}$

$$\tilde{\gamma}_{\tilde{G}} = \left[\left(G_1', G_2', \ldots, G_m'\right)\right] \stackrel{\text{def}}{=} \bigcap_{1 \leq i \leq m} \bigcup_{\gamma_i^{u_i} \in G_i'} \gamma_i^{u_i}$$

Die Funktion $convert : 2^X \to \left(2^X\right)^m$ zur Abbildung von einzelnen GJ-Paaren auf ihre Repräsentation ist somit definiert als

$$convert(\gamma) \stackrel{\text{def}}{=} \left(G_1', G_2', \ldots, G_m'\right), G_i' = \left\{\gamma_i^{u_i}\right\} \wedge \gamma = \bigcap_{1 \leq i \leq m} \gamma_i^{u_i}$$

[4]Dem wissbegierigen Leser sei jedoch an dieser Stelle schon verraten, daß die tiefgestellten Indizes einem komparativen Ausdruck im Modell entsprechen, während die hochgestellten Indizes die Interpretationsmöglichkeiten des Ausdrucks im Kontext von diesem vorangestellten Berechnungen beschreibt, die dessen Wahrheitsaussage mit beeinflussen.

	Dimension			
	1	2	…	n
J	D_1	D_2	…	D_n
	$\zeta_{\mathbb{R}}$	$\zeta_{\mathbb{R}}$	…	$\zeta_{\mathbb{R}}$
ζ_1	$\zeta^1_{D_1}$	$\zeta^1_{D_1}$	…	$\zeta^1_{D_1}$
ζ_2	$\zeta^2_{D_1}$	$\zeta^2_{D_2}$	…	$\zeta^2_{D_n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\zeta^{d_1}_{D_1}$	$\vdots$	…	$\vdots$
$\vdots$		$\vdots$		$\zeta^{d_n}_{D_n}$
$\vdots$		$\zeta^{d_2}_{D_1}$		
$\zeta_{\lvert J\rvert}$				

Tabelle 4.3: Systematische Übersicht über die Zerlegung von J. Dabei ist $d_i \stackrel{\text{def}}{=} |D_i|$. Jedes ζ_i ist eine Linearkombination von jeweils einer Funktion pro Dimension multipliziert mit dem zugehörigen Einheitsvektor

Jump Set Funktionen Bei der Zerlegung der Jump Set Funktionen wird ausgenutzt, daß für den kontinuierlichen Zustandsraum X gilt: $X \subseteq \mathbb{R}^n$. Jump Set Funktionen $\zeta \in \left(X \to 2^X\right)$ können für den Fall, daß $\forall x \in X : \exists x' \in X : \zeta(x) = \{x'\}$ gilt, konkreter als injektive Funktionen $\zeta \in (\mathbb{R}^n \to \mathbb{R}^n)$ für $X \subseteq \mathbb{R}^n$, also einem kontinuierlichen Zustandsraum mit n Dimensionen, betrachtet werden. Dies erlaubt eine Linear-Dekomposition von Jump Set Funktionen in n kartesische n-dimensionale Vektorfunktionen $\zeta_{D_1}, \zeta_{D_2}, \ldots, \zeta_{D_n}$ mit $\zeta_{D_k} : \mathbb{R}^n \to \mathbb{R}$, so daß

$\zeta_{D_k}, \tilde{\zeta}$

$$\zeta(x) = \tilde{\zeta}\left(\zeta_{D_1}, \zeta_{D_2}, \ldots, \zeta_{D_n}\right)(x) \stackrel{\text{def}}{=} \zeta_{D_1}(x) \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \zeta_{D_2}(x) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \cdots + \zeta_{D_n}(x) \cdot \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

D_k

Dabei gibt es für jede Dimension k eine endliche Menge $D_k = \left\{\zeta^i_{D_k} \in (\mathbb{R}^n \to \mathbb{R}) \,|\, 1 \le i \le num(J,k)\right\}$ von $|D_k| = num(J,k)$ verschiedenen n-dimensionalen Vektorfunktionen, aus denen die jeweiligen ζ_{D_k} gewählt werden können, d.h. $\zeta_{D_k} \in D_k$. Die Funktion $num(J,k)$ ist eine Modell-spezifische dimensionsabhängige Funktion, die die Anzahl der verschiedenen Funktionsvorschriften für die Dimension k innerhalb von J repräsentiert[5]. Jede Teilfunktion $\zeta^i_{D_k} \in D_k$ kann somit der Menge jeder Kombination mit beliebigen Teilfunktionen anderer Dimensionen zugeordnet werden, formal: $\zeta^i_{D_k} \mapsto \left\{(\zeta_{D_1}, \zeta_{D_2}, \ldots, \zeta_{D_k}, \ldots, \zeta_{D_n}) \in J \,\middle|\, \zeta_{D_j} \in D_j \wedge \zeta_{D_k} = \zeta^i_{D_k}\right\}$. Wir bezeichnen die Funktionen ζ_{D_k} als *Jump-Komponenten*.

$\tilde{j}$

Eine Menge $\{\zeta_1, \zeta_2, \ldots, \zeta_b\} \subseteq J$ wird mit dieser kartesischen Funktionszerlegung wieder durch Angabe eines Mengenvektors der Länge n beschrieben mit $\tilde{j} := \left(D'_1, D'_2, \ldots, D'_n\right), D'_k \subseteq D_k, 1 \le k \le n$ wobei sich durch solch einen Mengenvektor eine Anzahl von $b = \prod_{1 \le k \le n} \left|D'_k\right|$ Jump Set Funktionen beschreiben lassen. Auch

[5] Für den neugierigen Leser sei an dieser Stelle vorweggenommen, daß sich die Anzahl der Möglichkeiten $num(J,k)$ aus den unterschiedlichen Berechnungsabläufen im Rahmen des Kontrollflusses des imperativen Programms erschließt, mit dem das Hybride System modelliert wird.

hier können ein-elementige Teilmengen beschrieben werden, die genau eine Jump Set Funktion $\zeta \in J$ beinhalten.

Die Angabe einer Ersatzfunktion ist anders als bei den Guard Sets diesmal nur möglich, wenn $\forall_{1\leq i\leq n} \left|D'_i\right| = 1$. Da mit dieser Einschränkung durch den Mengenvektor unzureichenderweise genau eine Funktion beschrieben wird, führen wir eine Funktion $\zeta_{\mathbb{R}} : \mathbb{R}^n \to \mathbb{R}$, welche eine beliebige reelle Zahl unabhängig vom Argument liefert. Diese ist typischerweise Element in jeder Menge D_i, so daß für $D'_i = \{\zeta_{\mathbb{R}}\}$ alle ζ_{D_i} abstrahiert werden, dies also $D'_i = D_i$ entspricht. $\zeta_{\mathbb{R}}$

Dann ist die *Ersatzfunktion* $\tilde{\zeta}_{\tilde{J}}$ beschrieben durch eine Funktion $[] : \left(2^{\mathbb{R}^n \to \mathbb{R}}\right)^n \to (\mathbb{R}^n \to \mathbb{R}^n)$ mit $\tilde{\zeta}_{\tilde{J}}$

$$\left[\left(D'_1, D'_2, \ldots, D'_n\right)\right] \stackrel{\text{def}}{=} \tilde{\zeta}\left(\zeta_{D_1}, \zeta_{D_2}, \ldots, \zeta_{D_n}\right) \text{ mit } \zeta_{D_i} \in \{\zeta_{D_i}\} = D'_i$$

Die Abbildung $convert : \left(X \to 2^X\right) \to \left(2^{\mathbb{R}^n \to \mathbb{R}}\right)^n$ ist definiert als

$$convert(\zeta) \stackrel{\text{def}}{=} \left(D'_1, D'_2, \ldots, D'_n\right), D'_i = \{\zeta_{D_i}\} \wedge \zeta = \tilde{\zeta}\left(\zeta_{D_1}, \zeta_{D_2}, \ldots, \zeta_{D_n}\right)$$

***GJ*-Teilmengen** Mit obiger Dekomposition erhalten wir eine Abbildung von Teilmengenbeschreibungspaaren $\left(\tilde{g}, \tilde{j}\right) = \left(\left(G'_1, G'_2, \ldots, G'_m\right), \left(D'_1, D'_2, \ldots, D'_n\right)\right)$ in *GJ*-Teilmengen durch

$$\left(\tilde{g}, \tilde{j}\right) \mapsto \left\{ (\gamma, \zeta) \in G \times J \;\middle|\; \gamma \subseteq \bigcap_{1\leq i\leq m} \bigcup_{\gamma_i^{u_i} \in G'_i} \gamma_i^{u_i} \wedge \zeta = \tilde{\zeta}\left(\zeta_{D_1}, \zeta_{D_2}, \ldots, \zeta_{D_n}\right), \zeta_{D_k} \in D'_k \right\}$$

Für die Ermittlung von *GJ*-Teilmengensequenzen in der Phase der Pfadanalyse müssen die Mengen $G_1, G_2, \ldots, G_m$ und $D_1, D_2, \ldots, D_n$ alle a priori bekannt sein, um entsprechende generalisierte Konflikte generieren zu können. Diese Mengen ergeben sich aus einer umfangreichen Analyse des Modells H, die sich stark an der konkreten Repräsentation von G und J orientiert und daher in Abschnitt 6.3.3.1 beschrieben wird. Wir bezeichnen im Folgenden die Sequenzen von Teilmengenbeschreibungspaaren mit C^*

$$C^* \stackrel{\text{def}}{=} \bigcup_{k\in\mathbb{N}} \left(\left(2^X\right)^m, \left(2^{\mathbb{R}^n \to \mathbb{R}}\right)^n\right)^k$$

Die vorgestellte Systematik ist nur eine Möglichkeit der Einführung von Gruppierungsmengen, die im $\mathbb{R}^n$ und aufgrund der im Schritt-diskreten Hybriden System für Guard Sets verwendeten Ausdrücke nahe liegend ist und, wie wir in Kapitel 6 sehen werden, ist die Systematik sogar durch rein syntaktisch operierende Algorithmen ermittelbar. Andere Systematiken sind denkbar und sogar kombinierbar mit der hier vorgestellten, so daß weitere, abstraktere Aspekte ähnlicher Konflikte einbezogen werden können, die sich nur aus analytischen Betrachtungen ergeben können.

4.7.2.3 Pfadanalyse für *GJ*-Teilmengensequenz-Konflikte

Nach der vorgestellten Systematisierung der *GJ*-Teilmengenbildung wird nun die zugehörige Erweiterung der Pfadanalyse betrachtet, um aus entsprechenden Teilmengen bestehende Konflikte aus dem Pfad zu extrahieren. Dabei beschränken wir uns hier zunächst auf Konflikte bestehend aus solchen Teilmengensequenzen, für die die Ersatzfunktion [] definiert ist.

Dazu erweitern wir zunächst die Lösungsfunktion $\mathcal{L}$, welche als $\mathcal{L} : C \times 2^X \to 2^X$ definiert ist, auf die Funktion $\mathcal{L}^* : C^* \times 2^X \to 2^X$ mit der folgenden Definition: $\mathcal{L}^*$

$i := 0$
$k := 0$
$C^*_{ii} := \emptyset$
$C^*_{iw} := \emptyset$
$c^* := ((\tilde{g}_0, \tilde{j}_0), (\tilde{g}_1, \tilde{j}_1), \ldots, (\tilde{g}_n, \tilde{j}_n))$ **mit** $(\tilde{g}_i, \tilde{j}_i) = convert(\gamma_i, \zeta_i)$
while $C^*_{iw} \cup C^*_{ii} = \emptyset$ **do**
  while $i + k \leq n$ **do**
    if $i = 0 \wedge \mathcal{L}^{*\prime}(((\tilde{g}_0, \tilde{j}_0), (\tilde{g}_1, \tilde{j}_1), \ldots, (\tilde{g}_{i+k}, \tilde{j}_{i+k})), X_0) = false$ **then**
      $C^*_{ii} := C^*_{ii} \cup \{r^{\sqcup}(((\tilde{g}_0, \tilde{j}_0), (\tilde{g}_1, \tilde{j}_1), \ldots, (\tilde{g}_{i+k}, \tilde{j}_{i+k})), X_0)\}$
      $i := i + k$
    if $i > 0 \wedge \mathcal{L}^{*\prime}(((\tilde{g}_0, \tilde{j}_0), (\tilde{g}_1, \tilde{j}_1), \ldots, (\tilde{g}_{i+k}, \tilde{j}_{i+k})), X) = false$ **then**
      $C^*_{iw} := C^*_{iw} \cup \{r^{\sqcup}(((\tilde{g}_i, \tilde{j}_i), (\tilde{g}_{i+1}, \tilde{j}_{i+1}), \ldots, (\tilde{g}_{i+k}, \tilde{j}_{i+k})), X)\}$
      $i := i + k$
    $i := i + 1$
  end
  if $C^*_{iw} \cup C^*_{ii} = \emptyset$ **then** $k := k + 1, i := 0$
end
return (C^*_{ii}, C^*_{iw}) % $r^*_{ii} : C \to 2^{C^*}$ liefert C^*_{ii}, $r^*_{iw} : C \to 2^{C^*}$ liefert C^*_{iw}

Algorithmus 9: $r^* : C \to 2^{C^*} \times 2^{C^*}$. Berechnung reduzierter Konfliktmengen C^*_{ii} und C^*_{iw} aus einem Konflikt $c = ((\gamma_0, \zeta_0), \ldots, (\gamma_n, \zeta_n))$.

Definition 28 (Lösungsfunktion) *Gegeben sei eine GJ-Teilmengensequenz $c^* \in C^*$ mit $c^* = \left(\left(\tilde{g}_1, \tilde{j}_1\right), ..., \left(\tilde{g}_n, \tilde{j}_n\right)\right)$ und eine Menge kontinuierlicher Zustände $\tilde{X} \subseteq X$. Die* Lösungsfunktion *$\mathcal{L}^* : C^* \times 2^X \to 2^X$ ist definiert als:*

$$\mathcal{L}^*\left(c^*, \tilde{X}\right) \stackrel{def}{=} \begin{cases} \tilde{X} & wenn\ |c^*| = 0 \\ \left\{\left[\tilde{j}_n\right](x) \,\middle|\, x \in \left(\mathcal{L}^*\left(\left(\left(\tilde{g}_1, \tilde{j}_1\right), ..., \left(\tilde{g}_{n-1}, \tilde{j}_{n-1}\right)\right), \tilde{X}\right) \cap [\tilde{g}_n]\right)\right\} & wenn\ |c^*| > 0 \end{cases}$$

Der Begriff eines Konflikts wird unter Verwendung von $\mathcal{L}^*$ anstelle von $\mathcal{L}$ analog zu Definition 20 entsprechend erweitert, d.h. wir erweitern den Begriff auf Mengen von *GJ*-Sequenzen, bei denen der „Grund" der Nichtlösbarkeit von $\mathcal{L}$ bzgl. dieser Sequenzen und der jeweiligen Menge X_0 bzw. X gleich ist. Die Funktionen $\mathcal{L}^{*\prime}$ und $\tilde{\mathcal{L}}^{*\prime}$ sind analog zu $\mathcal{L}'$ und $\tilde{\mathcal{L}}'$ definiert.

$\mathcal{L}^{*\prime}, \tilde{\mathcal{L}}^{*\prime}$

Die Isolation von Konflikten muß nun über Algorithmus 2 hinausgehend auch die Bestimmung möglichst großer Teilmengen mit einschließen und entsprechend erweitert werden. Algorithmus 9 beschreibt die Funktion $r^* : C \to 2^{C^*} \times 2^{C^*}$ aufbauend auf Algorithmus 2 und ist um einen Funktionsaufruf $r^{\sqcup} : C^* \times 2^X \to C^*$ erweitert, welcher nach der Eingrenzung des Intervalls auf einen Konflikt angewandt wird, bevor dieser der Ergebnismenge hinzugefügt wird. Während also r^* der Intervallbestimmung dient, ist $r^{\sqcup}$ für die Verallgemeinerung der einzelnen *GJ*-Paare innerhalb des Konflikts zuständig. Der von $r^{\sqcup}$ verwendete Algorithmus 10 substituiert versuchsweise für alle Elemente der Sequenz nacheinander die einzelnen Guard-, bzw. Jump-Komponenten durch die leere Menge bzw. die Funktion $\zeta_{\mathbb{R}}$ und prüft, ob die Sequenz nach wie vor ein Konflikt ist. Falls ja, wird die Substitution beibehalten, andernfalls nicht.

$r^{\sqcup}, r^*$

Wir erhalten durch diese Form der Pfadanalyse erheblich allgemeinere Konflikte, die wir mit dem Automaten A^*_C ausschließen können. Bzgl. dieser Erweiterung wird in Abschnitt 6.3.3 die Berechnung der Guard- und Jump-Komponenten Systematik anhand der konkreten Repräsentation der Hybriden Modelle geschildert,

$i := 0$
while $i \leq n$ **do**
  $j := 0$
  while $j \leq m$ **do**
    $(\tilde{\gamma}, \tilde{\zeta}) := c^*_{(i)}$
    $\tilde{\gamma}' := \tilde{\gamma}_{[(j)\setminus\emptyset]}$
    $c' := c^*_{[(i)\setminus(\tilde{\gamma}', \tilde{\zeta})]}$
    if $\mathcal{L}^{*\prime}(c', \tilde{X}) = \emptyset$ **then**
      $c^* = c'$
  end
  $j := 0$
  while $j \leq n$ **do**
    $(\tilde{\gamma}, \tilde{\zeta}) := c^*_{(i)}$
    $\tilde{\zeta}' := \tilde{\zeta}_{[(j)\setminus\{\zeta_{\mathbb{R}}\}]}$
    $c' := c^*_{[(i)\setminus(\tilde{\gamma}, \tilde{\zeta}')]}$
    if $\mathcal{L}^{*\prime}(c', \tilde{X}) = \emptyset$ **then**
      $c^* = c'$
  end
  $i := i + 1$
end
return c^*

Algorithmus 10: $r^{\sqcup} : C^* \times 2^X \rightarrow 2^{C^*}$. Verallgemeinerung eines Konflikts $c^* = ((\tilde{g}_0, \tilde{f}_0), (\tilde{g}_1, \tilde{f}_1), \ldots, (\tilde{g}_n, \tilde{f}_n))$ bzgl. einer Ausgangsmenge $\tilde{X}$ durch Teilmengenberechnung

während die erreichte Reduzierung der Iterationszahl in Abschnitt 7.3.2 vorgestellt und diskutiert wird.

4.8 Anwendung des Verfahrens auf Zeit-kontinuierliche Hybride Systeme

Die Einschränkung auf Schritt-diskrete Systeme, die wir bislang gemacht haben, war motiviert durch

1. die Funktionalität industrieller Modelle, vorliegend als Case-Tool-, bzw. C-Modell, welche keinerlei Zeit-Kontinuität beinhaltet,
2. die einfachere Darstellung des iterativen Abstraktionsverfeinerungsverfahrens, sowie
3. Implementierungsaspekte, welche in Kapitel 6 beschrieben werden.

Die Technik des Ausschlusses von diskreten Zustandspfaden kann jedoch ohne Einschränkung auch auf Zeit-kontinuierliche Systeme angewendet werden, wie in [CFH+03] gezeigt. Die Schwierigkeit bei kontinuierlichen Systemen liegt hier in der Realisierung der Lösungsfunktion $\mathcal{L}$ begründet. Diese gestaltet sich sehr viel aufwendiger, wenn kontinuierlichee Werteverläufe durch Differentialgleichungen beschrieben sind. Wie jedoch bereits erwähnt, ist die Umsetzung der Lösungsfunktion austauschbar, insbesondere z.B. auch durch das in [CFH+03] angewandte Verfahren.

Die nötigen Erweiterungen des ω-Cegar Verfahrens sind die folgenden:

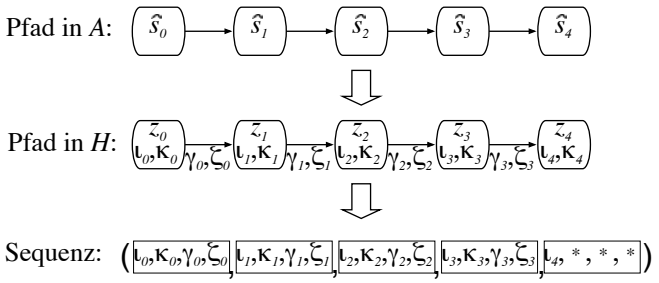


Abbildung 4.13: Erweiterung der Pfadprojektion für Zeit-kontinuierliche Hybride Systeme. Vgl. Abbildung 4.3 auf Seite 41

- Mit $I \subseteq 2^X$ als der Menge aller Invarianten und $F^f \subseteq 2^{(X \to \mathbb{R}^n)}$ als der Menge aller kontinuierlichen Vektorfelder des Zeit-kontinuierlichen Hybriden Automaten H verwenden wir anstelle der GJ-Sequenzen nun IF^fGJ-Sequenzen der Form
$$\langle(\iota_n, \kappa_n, \gamma_n, \zeta_n)\rangle, \iota_i \in I, \kappa_i \in F^f, \gamma_i \in G, \zeta_i \in J$$

- Die Funktion θ zur Berechnung von GJ-Sequenzen aus abstrakten Pfaden (Abschnitt 4.3.2.1) erweitert sich wie in Abbildung 4.13 dargestellt zu
$$\theta(\hat{\pi}) = \langle(\iota_n, \kappa_n, \gamma_n, \zeta_n)\rangle \text{ mit}$$
  - $\iota_i = inv\left(\tilde{\alpha}^{-1}(\hat{s}_{i+1})\right)$
  - $\kappa_i = f\left(\tilde{\alpha}^{-1}(\hat{s}_{i+1})\right)$
  - $\gamma_i = g\left(\tilde{\alpha}^{-1}\left((\hat{s}_i, \hat{s}_{i+1})\right)\right)$
  - $\zeta_i = j\left(\tilde{\alpha}^{-1}\left((\hat{s}_i, \hat{s}_{i+1})\right)\right)$

- Die Lösungsfunktion bzgl. einer IF^fGJ-Sequenz $c = \langle(\iota_n, \kappa_n, \gamma_n, \zeta_n)\rangle$ wird zu
$$\mathcal{L}(c, \tilde{X}) = \begin{cases} \tilde{X} & \text{wenn } |c| = 0 \\ \left\{ \begin{array}{l} \zeta_n(x') | x \in \left(\mathcal{L}\left(\langle(\iota_{n-1}, \kappa_{n-1}, \gamma_{n-1}, \zeta_{n-1})\rangle, \tilde{X}\right)\right) \cap \iota_n, \\ \exists \tau \in \mathbb{R}^{>0}, \chi : [0, \tau] \to X : \chi(0) = x, x' = \chi(\tau), \\ x' \in \gamma_n, \dot{\chi}(t) = \kappa_n(\chi(t)), t \in [0, \tau], \chi(t) \in \iota_n \end{array} \right\} & \text{wenn } |c| > 0 \end{cases}$$

- Das in der Automatenkonstruktion verwendete Alphabet Σ besteht nun aus $\Sigma = I \times F^f \times G \times J$.

- Die Transitionsrelation des Kreuzprodukts zur Verfeinerung des abstrakten Transitionsmodells A_{i+1} aus Abschnitt 4.3.4 wird zu
$$\hat{T} := \left\{ \begin{array}{l} ((\hat{z}_1, q_1), (\hat{z}_2, q_2)) \,|\, (\hat{z}_1, \hat{z}_2) \in \hat{E} \wedge (q_1, \sigma, q_2) \in T_C \wedge \exists t \in T : t = \tilde{\alpha}^{-1}((\hat{z}_1, \hat{z}_2) \wedge \\ \sigma = (inv(\tilde{\alpha}^{-1}(\hat{z}_1)), f(\tilde{\alpha}^{-1}(\hat{z}_1)), g(t), j(t)) \wedge q_2 \in F \end{array} \right\}$$

  Da in dem Kreuzprodukt das Übereinstimmungskriterium für eine Transition sich aufgrund der Einbeziehung von Invarianten und kontinuierlicher Vektorfelder des Ausgangszustands deutlich verschärft hat, wird es wesentlich mehr Zustände geben, die bedingt durch fehlende eingehende Transitionen trivial unerreichbar sind.

Damit ist das Verfahren einschließlich der erweiterten Minimierung aus Abschnitt 4.7.1 vollständig auf Zeit-kontinuierliche Hybride Systeme nach Definition 7 umgestellt.

Auch die Einführung von Teilmengensequenzen aus Abschnitt 4.7.2 läßt sich generell übertragen, sollte hier jedoch gegenüber der in Abschnitt 4.7.2.2 dargestellten Dekomposition entsprechend angepasst werden, so daß hier Invarianten und kontinuierliche Vektorfelder gleichermaßen bzgl. gemeinsamer Auswirkungen auf den Berechnungsablauf gruppiert werden. Auch hier bietet sich sicherlich eine Dimensions-spezifische Systematisierung an.

4.9 Vergleich zu anderen Verfahren

Von den existierenden, in der Einleitung bereits erwähnten iterativen Abstraktionsverfeinerungsverfahren haben zwei von diesen am meisten Gemeinsamkeiten mit dem hier vorgestellten Verfahren. Dies ist zum Einen der Infinite-State-Cegar Algorithmus angewandt auf Hybride Systeme [CFH+03] und zum Anderen die Erreichbarkeitsanalyse in Hybriden Systemen durch Pfad-gelenkte Prädikatenabstraktion [ADI02, ADI03]. Beide werden in den folgenden Abschnitten vorgestellt, sowie Gemeinsamkeiten und Unterschiede zum ω-Cegar Verfahren diskutiert.

Im Bereich des *Bounded-Model-Checking* (*BMC*) finden sich ebenfalls Parallelen zu dem ω-Cegar Ansatz. So wird in dem BMC-Werkzeug für Hybride Systeme namens HySAT [FH05] die Generalisierung von Konflikten auf SAT-basierter Ebene praktiziert und soll daher ebenfalls Erwähnung finden.

Im Anschluss wird eine alternative Konstruktion für Automaten auf Basis von LTL-Formeln vorgestellt.

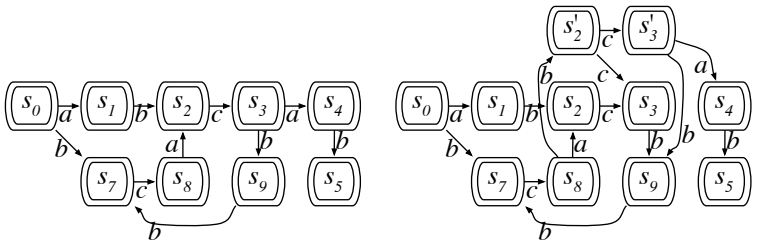
4.9.1 Infinite-State-Cegar für Hybride Systeme

Das Infinite-State-Cegar Verfahren aus [CFH+03] ist ein vollautomatisches iteratives Abstraktionsverfeinerungsverfahren, mit dem Zeit-kontinuierliche Hybride Systeme entsprechend der Definition 7 auf Seite 27 verifiziert werden können. Das Verfahren folgt dabei der bereits in [CGJ+00] erhobenen Maxime, daß es „wünschenswert ist, die gröbste Verfeinerung [zwischen den Iterationsschritten] zu erzielen, welche die [ungültigen] Gegenbeispiele eliminiert, da dies dem kleinsten abstrakten Modell entspricht, das für die Verifikation geeignet ist"[6].

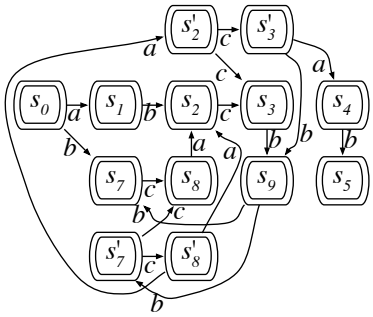
Für die Validierung der Pfade (Pfadanalyse) kommen dabei Polyeder-basierte Überapproximationen des Zustandsraums zum Einsatz, wobei zwischen verschieden groben Überapproximationen gewählt werden kann, um einen geeigneten Trade-off zwischen Genauigkeit und Berechnungsaufwand zu erreichen. Auch hier wird das für die Unlösbarkeit der kontinuierlichen Berechnungsschritte relevante Pfad-Teilstück bestimmt, jedoch streng gekoppelt an die diskreten Transitionen und diskreten Zustände des abstrakten Pfades. Das Validierungsverfahren ist in dem Ansatz ebenso wie in dem in dieser Arbeit vorgestellten Ansatz prinzipiell austauschbar.

Unterschiedlich hingegen ist die Verwendung der Validierungsinformation zur Generierung des abstrakten Transitionsmodells für den folgenden Iterationsschritt. Nachdem der Pfad für ungültig erklärt und das minimale Pfadfragment bestimmt worden ist, welches ursächlich für die Unlösbarkeit der Berechnung ist, so wird dieser Pfadausschnitt aus dem Modell dergestalt eliminiert, daß diese Transitionsfolge nicht mehr Teil eines Pfades in Präfix-, Infix- oder Suffixstellung sein kann.

[6] engl.: „It is desirable to obtain the coarsest refinement which eliminates the counterexample because this corresponds to the smallest abstract model that is suitable for verification."



(a) Transitionsystem vor 1. Verfeinerung (b) Transitionsystem nach 1. Verfeinerung



(c) Transitionsystem nach 2. Verfeinerung

Abbildung 4.14: Abstraktes Transitionssystem A. Transitionsbeschriftungen beziehen sich auf GJ-Paare, welche sich durch Projektion auf Hybrides System H ergeben.

Dabei werden dem Modell für ein Fragment der Länge n eine Anzahl von $n-1$ Zuständen hinzugefügt. Da an dieser Stelle nicht die Formalisierung des Verfahrens aus [CFH+03] erklärt werden soll, beschränken wir uns für die Verdeutlichung des Ansatzes auf ein Beispiel.

In Abbildung 4.14 ist ein abstraktes Transitionssystem abgebildet, wobei die Transitionsbeschriftungen a, b, c jeweils GJ-Paare darstellen, die nicht Teil des Transitionssystems sind, hier aber dennoch beigefügt worden sind, um die den Transitionen zuzuordnenden Berechnungen zu illustrieren. Angenommen, die Erreichbarkeit des Zustands s_5 soll verifiziert werden und der Model-Checker hat nun den Pfad $\hat{\pi}_1 = (s_0, s_1, s_2, s_3, s_4, s_5)$ errechnet. Nehmen wir an, die Pfadanalyse ergibt, daß das Pfadfragment (s_1, s_2, s_3, s_4) keine dazugehörige Berechnung im kontinuierlichen Zustandsraum besitzt, der Pfad daher für ungültig erklärt werden muß. Das Verfahren in [CFH+03] bewerkstelligt diesen Pfadausschluß durch eine Replikation aller vom ersten und letzten Zustand des Pfadfragments eingeschlossenen Zustände, in diesem Falle s_2 und s_3, welche repliziert werden durch s'_2 und s'_3. Die Transitionen werden dabei so ergänzt, daß alle Läufe, die nicht das auszuschließende Pfadfragment (s_1, s_2, s_3, s_4) besitzen, weiterhin beobachtbar sind. Dabei sind ein Zustand s und dessen Replikation s' in einem Lauf gleichermaßen als Zustand s zu betrachten.

Nach dieser Transformation ist also für die folgende Iteration als nächstes insbesondere der Lauf $\hat{\pi}_2 = (s_0, s_7, s'_2, s'_3, s_4, s_5)$ ein möglicher Pfad, der vom Model-Checker gefunden würde. Wie bereits durch die Transitionsbeschriftungen angedeutet, gehen wir in diesem Beispiel davon aus, daß die durch $\hat{\pi}_1$ induzierte GJ-Sequenz die Folge $\theta(\hat{\pi}_1) = (a, b, c, a, b)$ ist, der Grund für das ausgeschlossene Pfadfragment also die Teilsequenz (b, c, a) ist und (b, c, a) somit ein invarianter Konflikt. Nun ist die GJ-Folge für $\hat{\pi}_2$ $\theta(\hat{\pi}_2) = (b, c, a, c, a, b)$, enthält also

insbesondere wieder den Konflikt (b, c, a), wodurch $\hat{\pi}_2$ ebenfalls für ungültig erklärt werden muß. Und damit noch nicht genug, so wird der nächste Pfad nach entsprechender Verfeinerung des Modells über den Zustand s_9 verlaufend mit $\hat{\pi}_3 = (s_0, s_1, s_2, s_3, s_9, s'_7, s'_8, s'_2, s'_3, s_4, s_5)$ zum dritten Mal aufgrund desselben Konflikts invalidiert.

Dieses Beispiel verdeutlicht den Unterschied zwischen [CFH+03] und dem in dieser Arbeit vorgestellten Ansatz. Durch das Erlernen des Konflikts (b, c, a) in einem ω-Automaten wäre dieser generell in keinem Lauf des abstrakten Transitionssystems mehr vorhanden und mithin die zuletzt betrachteten Iterationen hinfällig. Damit kann die Anzahl der Iterationen in dem ω-Cegar Ansatz nur kleiner sein als in [CFH+03].

Maßnahmen, wie die (noch) weitergehende Verallgemeinerung durch Ausschluß von *GJ*-Teilmengensequenzen, wie in Abschnitt 4.7.2 beschrieben, sind bei diesem Verfahren nicht möglich, so daß die zu erwartende unterschiedliche Anzahl an benötigten Iterationen des ω-Cegar und des Infinite-State-Cegar Verfahrens noch sehr viel erheblicher differieren werden, als obige Überlegung bereits andeutet.

Die Autoren stellen in [CFH+03] fest, daß „die Berechnung der Nachfolgerzustände [der kontinuierlichen Berechnungen zur Validierung des Pfads] gewöhnlich der teuerste Schritt in der Verifikation hybrider Systeme darstellt"[7]. Abgesehen davon, daß für große diskrete Zustandsräume diese Aussage nicht unbedingt zutreffend ist, so ist doch etwas verwunderlich, daß unter Kenntnis dieser Problematik von den Autoren ein Verfahren vorgeschlagen wurde, welches eben diese Berechnung mehrfach im Rahmen der vielen Iterationen wiederholen muß. Aufgrund unserer in Abschnitt 4.1.2 geschilderten Beobachtung ist sogar ein Wiederholungsfaktor zu erwarten, der mit der Größe des Zustandsraums und der Länge der ermittelten Pfade in etwa in gleichem Maße ansteigt. Vor diesem Hintergrund erscheint der Versuch, die von den Autoren offenbar ohnehin als klein angenommenen diskreten Zustandsräume gemäß eingangs zitierter Maxime auch für die Abstraktionen so klein wie möglich zu halten, etwas verwunderlich.

Da sich der beschriebene Pfadausschluß in äquivalenter Weise im ω-Cegar Verfahren durch Ausschluss der jeweilig zugehörigen Zustandspaare der Transitionskette nachvollziehen lässt, indem beispielsweise das Pfadfragment (s_1, s_2, s_3, s_4) durch Erlernen der Sequenz $((s_1, s_2), (s_2, s_3), (s_3, s_4))$ mit $\Sigma = Z \times Z$ ausgeschlossen wird, sind diesbzgl. exakte Vergleiche hinsichtlich der Iterationszahlen zwischen beiden Verfahren möglich. Experimentelle Vergleiche in Kapitel 7.4 werden entsprechend zeigen, inwieweit sich obiges Problem bei größeren Modellen verschlimmert, zumal die beschriebenen Erweiterungen des ω-Cegar Verfahrens sich nicht auf das Infinite-State-Cegar Verfahren in [CFH+03] übertragen lassen.

4.9.1.1 Erweiterung um Zustandssequenzfragmente

Neuere Verbesserungen des Verfahrens sind in [FCJK05] geschildert und zielen im Wesentlichen auf eine effizientere Auswahl der auszuschließenden Fragmente eines Pfades ab. Die Verfeinerung des abstrakten Modells ist dabei unverändert, mit allen bereits diskutierten grundsätzlichen Konsequenzen bzgl. der Iterationsanzahl. Dennoch verfolgt das Verfahren einen interessanten Ansatz, um die Anzahl der Iterationen durch Effekte wie in Abschnitt 4.3.2.3 betrachtet und in Abbildung 4.4 auf Seite 44 dargestellt zu reduzieren, indem Pfadfragmente, wenn möglich, an zentraler Stelle im abstrakten Transitionssystem auszuschließen, um möglichst viele falsche Gegenbeispiele abzudecken.

[7] engl.: „The computation of sets of successor states is usually the most expensive step in hybrid system verification."

Das für diese Analyse erforderliche Verfahren ist in einen explizite Zustandsrepräsentationen verwendenden Model-Checker (*explicit state model-checker*) integriert, so daß kein gewöhnlicher Model-Checker (*finite-state model-checker*) für dieses Verfahren zum Einsatz kommen kann.

Die Analyse wird durch die Verwaltung von Mengen von Pfadfragmenten $\mathcal{F}$ in Form von Zustandssequenzen durchgeführt, von denen mindestens ein Fragment $\langle s_n \rangle \in \mathcal{F}$ jeden möglichen Pfad $\hat{\pi}$ zu der Menge der zu erreichenden Zustände $\hat{S}_U$ im diskreten Zustandsraum „schneidet" (*cut*), also als Teilsequenz in Präfix-, Infix- oder Suffixstellung in $\hat{\pi}$ vorkommt. Zur Berechnung einer minimalen Menge solcher Pfadfragmente kommen verallgemeinerte Algorithmen zur Berechnung von minimalen Cutting Sets von Graphen zum Einsatz. Im Falle einer vollständigen minimalen Pfadfragmentmenge $\mathcal{F}_{opt}$ kann für jeden Pfad $\hat{\pi}$ das ideale Pfadfragment bestimmt werden, welches

1. am einfachsten bzgl. der kontinuierlichen Berechnung zu (in-)validieren ist und
2. am meisten Pfade ausschließt.

Die Fragmente in $\mathcal{F}$ werden dabei während einem, von dem in Abschnitt 2.4 dargelegten Verfahren verschiedenen Model-Checking-Algorithmus im Falle der Gültigkeit sukzessive erweitert, bis entweder das Fragment eine bestimmte Länge l mit $l \leq 4$ überschritten hat, oder das Fragment bereits zu einem Zustand in $\hat{S}_U$ führt. Bei Fragmenten länger als l wird der gesamte Pfad zu $\hat{S}_U$ als Fragment hinzugefügt. Wenn das Fragment validiert werden kann, ist ein gültiger Pfad gefunden.

Das Verfahren dient gleichzeitig der Reduzierung des Validierungsaufwands, indem mit Gewichtungen der Transitionen die Validierungskosten abgeschätzt und die „günstigsten", d.h. am einfachsten zu validierenden Fragmente bei der Pfadsuche gewählt werden können.

Das Verfahren ist aufgrund der Berechnung von vollständigen Mengen von Pfadfragmenten nicht auf größere Transitionssysteme anwendbar, da es selbst bei endlicher Länge in einem Transitionssystem im Allgemeinen exponentiell viele Pfade gibt. Inwieweit symbolisches Model-Checken in diesem Verfahren anwendbar wäre, bliebe noch zu spezifizieren, da die Pfade während der Fragmenterweiterung berechnet werden und eine Umsetzung auf symbolischer Ebene nicht trivial erscheint. Die experimentellen Resultate in [FCJK05] ergeben eine Performanzverbesserung[8] von 1% gegenüber dem Verfahren in [CFH+03].

4.9.2 Prädikaten-Abstraktion

Das in [ADI02] beschriebene Verfahren zur Verifikation Hybrider Systeme basiert auf einer Prädikat-basierten Abstraktion, welche eine manuelle Selektion bzw. Ergänzung von Prädikaten voraussetzt. In einer in [ADI03] beschriebenen weiterentwickelten Version wird dieser Schritt automatisiert und in einer ausführlicheren Abhandlung in [ADI06] beschrieben. Wir beschreiben zunächst die manuelle Variante.

Die Prädikate, welche analog zu Guard Sets Teilmengen des kontinuierlichen Zustandsraums durch Vergleichsausdrücke beschreiben, können bei diesem Verfahren aus dem Modell abgeleitet werden, können bzw. müssen jedoch auch manuell ergänzt oder ausgewählt werden. Die zu verifizierenden Systeme sind Zeitkontinuierlich, die Berechnungen auf dem Modell gleichermaßen wie die Prädikate für eine effiziente Berechnung auf lineare beschränkt, was jedoch auch hier keine grundsätzliche Einschränkung darstellt. Wir beschränken die Beschreibung des

[8] Von 28, 1 auf 27, 8 Sekunden für ein einzelnes angegebenes Experiment

Verfahrens für einen einfacheren Vergleich auf Schritt-diskrete Hybride Systeme nach Definition 9. Das abstrakte Transitionssystem definiert sich hierfür wie folgt:

Zustandsraum Für k Prädikate wird der abstrakte Zustandsraum $\hat{Z}$ durch $\hat{Z} = Z \times \mathbb{B}^k$ beschrieben, wobei Z den diskreten Zustandsraum des Hybriden Systems nach Definition 7 bezeichnet. Jedem Prädikat p_i wird mithin also eine boolsche Variable b_i zugeordnet. Die Wahrheitswerte dieser boolschen Variablen korrelieren mit dem Wahrheitsgehalt der betreffenden Prädikate. Wir bezeichnen die Menge aller kontinuierlichen Zustände, für die ein boolscher Vektor $\langle b_k \rangle$ den jeweiligen Wahrheitsgehalt der dazugehörigen Prädikatenmenge beschreibt, mit $X_{\langle b_k \rangle}$, d.h. $X_{\langle b_k \rangle} \stackrel{\text{def}}{=} \{x \in X | \forall 1 \leq i \leq k : p_i(x) = b_i\}$.

Transitionsrelation Die Transitionsrelation sieht für zwei Zustände $\hat{z}_1 = (z_1, \langle b_k \rangle)$ und $\hat{z}_2 = (z_2, \langle b'_k \rangle)$ genau dann eine Transition $(\hat{z}_1, \hat{z}_2)$ vor, wenn gilt $t = (z_1, z_2) \in T \wedge \exists x \in X_{\langle b_k \rangle} \cap g(t) : \exists y \in X_{\langle b'_k \rangle} : y \in j(t)$. Die Berechnung der Transitionsrelation ist dabei insofern sehr aufwendig, als daß sie für jede Transition des abstrakten Systems einzeln durchgeführt werden muß.

Daher wird in [ADI02] eine Tiefensuche des Zustandsraums verwendet, welche die Menge der ausgehenden Transitionen für einen Zustand $(z_1, \langle b_k \rangle)$ während des Verifikationsprozesses (*on the fly*) berechnet.

Vergleich der Verfahren Bei der Betrachtung der Transitionsrelation ist festzustellen, daß für jeden Lauf alle Prädikate parallel observiert werden, unabhängig davon, ob diese möglicherweise in einen Konflikt führen können oder nicht. Daraus resultiert eine erhebliche Anzahl von zusätzlichen Zuständen und Transitionen, die für den Ausschluß bereits beobachteter ungültiger Pfade irrelevant sind.

Die Prädikatkombinationen beschreiben konvexe Polyeder und partitionieren entsprechend den kontinuierlichen Zustandsraum X in die Partitionsmenge $X_{\mathbb{B}^k} \stackrel{\text{def}}{=} \{X_{\langle b_k \rangle} | b_k \in \mathbb{B}^k\}$. Diese ist mit der in Abschnitt 4.4.1 auf Seite 64 beschriebenen Partitionierung $X_\chi = \{\chi(q) | q \in Q\}$ des ω-CEGAR Verfahrens zu vergleichen. Wenn die manuell vorgegeben Prädikate in ihren Kombination u.a. diese Menge beschreiben würden, formal $X_\chi \subseteq X_{\mathbb{B}^k}$, wäre das Verfahren theoretisch in der Lage, die gleichen Pfade zu ermitteln. Sehen wir einmal von der praktischen Unmöglichkeit ab, von einem Benutzer die Aufstellung der dazugehörigen Prädikate für große Systeme erwarten zu können, so bleiben weiterhin zwei Unterschiede im abstrakten Transitionssystem:

1. Die Anzahl der Partitionen $|X_{\mathbb{B}^k}| = 2^k$ ist fest vorgegeben und nur in 2er Potenzen veränderbar gegenüber $|X_\chi| = |Q|$. Somit ist die Partitionierung zwangsläufig suboptimal. Unter Umständen sind einige der Partitionen zwar leer, jedoch kann bzgl. eines n-flächigen Polyeders nicht vermieden werden, $2^{n/2}$ kollaterale Partitionen zu erhalten, da sämtliche Prädikate auch in negierter Form den dualen Zustandsraum beschreiben, welcher nur bei parallelen gegenüberliegenden Begrenzungen in entsprechenden Prädikatkombinationen leer sein kann.

2. Selbst bei Beschränkung auf die abstrakten Zustände $(z, \langle b_k \rangle)$, für die $\exists q \in Q$: $X_{\langle b_k \rangle} = \chi(q)$ gilt, sind dennoch erheblich mehr Transitionen vorhanden, da jeder mit dem Jump Set erreichbare Zielzustand im kontinuierlichen Zustandsraum bzgl. der Partitionen unterschieden und der zugehörige abstrakte Zustand eingenommen werden kann. Im Gegensatz dazu wird bei dem ω-CEGAR Verfahren nach Bildung des Kreuzprodukts nur zwischen für Konflikte relevanten Zuständen unterschieden. Ist kein Konflikt aufgrund unterschiedlicher Präfixe relevant, wird der Zustand q_0 eingenommen.

Durch die Vorgabe beliebiger Prädikate wird eine Abstraktionsmöglichkeit ähnlich der in Abschnitt 4.7.2 auf Seite 77 beschriebenen Teilmengensequenzen ermöglicht, ohne daß dadurch obige Aussagen zu relativieren wären, da insbesondere die Partitionierung bei Verwendung von *GJ*-Teilmengensequenzen sich nicht ändert.

Wenn das Verfahren effizient das abstrakte Transitionssystem berechnen würde, was bei Schritt-diskreten Systemen und limitierten *GJ*-Paaren durchaus ebenfalls in Form eines Kreuzprodukts mit einem getrennt konstruierten Automaten möglich wäre, so wäre bei Austausch der expliziten Tiefensuche durch effiziente Finite-State Model-Checker dieser Ansatz auch für große Hybride Systeme geeignet, wenn die jeweiligen Abstraktionen auch zu erheblich größeren und komplexeren Transitionssystemen führen. Da das Problem der Prädikatenbestimmung jedoch noch ungelöst ist, betrachten wir nun das diesbezügliche Vorgehen, das in [ADI03] beschrieben ist.

Automatische Prädikatenbestimmung Die automatische Prädikatenbestimmung bzgl. eines Pfades $\hat{\pi} = (\hat{z}_0, \hat{z}_1, \ldots, \hat{z}_i, \hat{z}_{i+1}, \ldots, \hat{z}_n)$ erfolgt auf Basis der ersten Transition $t = (z_i, z_{i+1})$, welche für die Ungültigkeit des Pfades verantwortlich ist. Es gilt für die Lösungsmenge R_i der durch den Pfad bis zum Zustand $\hat{z}_i$ durchgeführten, durch eine ähnlich zu der in Abschnitt 4.3.2.1 beschriebenen Pfadprojektion ermittelten Berechnungen $R_i := \mathcal{L}(\theta((\hat{z}_0, \hat{z}_1, \ldots, \hat{z}_i), X_0) \neq \emptyset$, während $R_{i+1} := \mathcal{L}(\theta((\hat{z}_0, \hat{z}_1, \ldots, \hat{z}_i, \hat{z}_{i+1}), X_0) = \emptyset$. Sowohl Darstellung als auch Verfahren zur Berechnung von R_i sind in [ADI03] verschieden von den in Abschnitt 4.3.2.1 eingeführten Funktionen θ und $\mathcal{L}$, jedoch eignet sich deren Wiederverwendung an dieser Stelle ideal für eine Beschreibung des Verfahrens in [ADI03] und erspart die Beschreibung von Details geringerer Relevanz. Durch die Einführung der Prädikate gilt: $\hat{z}_i = (z_i, \langle b_k \rangle) \wedge R_i \subseteq X_{\langle b_k \rangle}$. Es gilt $X_{\langle b_k \rangle} \cap g(t) \neq \emptyset$ mit $g(t)$ als dem der Transition t zugeordneten Guard Set, da sonst die Transition $(\hat{z}_i, \hat{z}_{i+1})$ nicht existiert hätte , während $R_i \cap g(t) = \emptyset$, weil andernfalls R_{i+1} nicht leer gewesen wäre. Aufgrund dieser Diskrepanz wird nun nach zusätzlichen m Prädikaten gesucht, wobei m möglichst klein sein sollte, welche eine Unterscheidung zwischen $X_{\langle b_k \rangle}$ und $g(t)$ ermöglichen. Damit hätte das verfeinerte Transitionssystem nach Berechnung der neuen Transitionsrelation dann keine Transition mehr von $\hat{z}_i = (z_i, \langle b_{k+m} \rangle)$ zu $\hat{z}_{i+1} = (z_{i+1}, \langle b'_{k+m} \rangle)$, da aufgrund der nun möglichen Unterscheidung $X_{\langle b_{k+m} \rangle} \cap g(t) = \emptyset$ gilt, so daß $\hat{\pi}$ nicht mehr auftreten kann.

Das Problem der Bestimmung möglichst weniger neuer Prädikate zur Unterscheidung zweier durch Mengen von Polyedern dargestellter Mengen ist bereits in drei Dimensionen NP-vollständig [DJ90]. Nur im einfachen Fall zweier sich nicht überschneidender Polyeder existiert eine Lösung mit linearem Aufwand in der Anzahl der Polyederflächen, die mit genau einem Prädikat auskommt, welches die beiden zugehörigen Halbräume unterscheidet. In dem Verfahren wird daher eine iterative Heuristik angewandt, welche die Polyedermengen in einzelne Polyeder unterteilt und diese jeweils durch ein neues Prädikat unterscheidet. Dabei können unterschiedliche Strategien für die Unterteilung Anwendung finden, wie z.B. Auswählen von Prädikaten, die jeweils die größte Menge von Polyedern trennen oder Maximierung der Volumen der zu trennenden Polyeder.

Vergleich der Partitionierung Abgesehen davon, daß der Einsatz von Heuristiken möglicherweise die Anzahl der in jedem Schritt notwendigen Prädikate möglicherweise mehr als für das Verfahren nötig erhöht, so sind bereits bei theoretisch optimaler Prädikat-Anzahl zur Trennung der oben beschriebenen Polyedermengen ein bis mehrere Prädikate zur Behandlung eines ungültigen Pfades notwendig. Daraus resultiert für m neue Prädikate, $m \geq 1$, eine Erhöhung der Partitionsanzahl um den Faktor 2^m in jeder Iteration, d.h. wenn vorher l Partitionen vorhanden waren,

so sind es nach dieser Verfeinerung bis zu $(l-1)\cdot 2^m$ zusätzliche neue Partitionen, soweit die Prädikate nicht zu existierenden Prädikaten parallele Halbräume beschreiben. Im Vergleich dazu wird bei dem ω-CEGAR Verfahren die Zahl der Partitionen additiv um eine Anzahl erhöht, die der Anzahl neuer Zustände im ω-Automat entspricht und insbesondere durch Minimierungseffekte sogar negativ sein kann. Da die Partitionen des ω-CEGAR genau diejenigen sind, die für den Ausschluß von ungültigen Pfaden benötigt werden, erscheint die Partitionierung des obigen Verfahrens im direkten Vergleich ineffizient, da trotz eines vielfachen an Partitionen zunächst kein zusätzlicher Nutzen erzielt wird. Es besteht jedoch die Möglichkeit, daß einige der zusätzlichen $(l-1)\cdot 2^m$ Partitionen präventiv an dem Ausschluß künftiger ungültiger Pfade beteiligt sein werden und damit Wiederverwendung finden.

Es gibt keine Maßnahme, die im Verlauf eines Pfades die nicht mehr relevanten, durch Guard Set Trennungen vorheriger Transitionen notwendig gewordenen Prädikate zu entfernen, so daß diese einschließlich der Konsequenzen für die Partitionierung und den damit einhergehenden zusätzlichen abstrakten Zuständen den gesamten Pfad, sowie künftige Iterationen der Abstraktionsverfeinerung begleiten. Demgegenüber ist im ω-CEGAR Verfahren eine Partition mitsamt deren abstrakten Zustand nur im Falle eines sich möglicherweise anbahnenden Konflikts im Pfad relevant und kann durch Minimierung sogar im Verlauf der Iterationen wieder mit anderen Zuständen im Rahmen der Minimierung vereinigt werden.

4.9.3 HySAT

Neben dem in Kapitel 2 vorgestellten CTL-basierten Model-Checking Verfahren existieren auch sog. *Bounded-Model-Checking* Verfahren (*BMC*), die im Gegensatz zu Erstgenanntem die Erreichbarkeit eines Zustands innerhalb einer festen Pfadlänge k berechnen können. Typische Vertreter dieser Technik sind *SAT*-basierte Verfahren (*Satisfiability*), die die als boolsche Funktion kodierte Schrittfunktion, die in Abschnitt 2.2.2.1 auf Seite 14 vorgestellt wurde, für k Schritte *abrollen* und somit durch Aufstellung einer boolschen Gleichung auf ein Erfüllbarkeitsproblem abbilden [BCC+99, GKvV95]. Diese Verfahren haben sich für kleine k als sehr effizient erwiesen, eignen sich jedoch in reiner Form nur zur *Falsifikation*, d.h. soweit kein Pfad der Länge k gefunden werden kann, können immer noch Pfade der Länge $l \neq k$ existieren. Daher sind SAT-basierte Techniken in dieser einfachen Form nur für Falsifikation von Eigenschaften in LTL-, nicht jedoch in CTL-Form geeignet[9]. Es gibt jedoch weitergehende Ansätze der SAT-basierten Techniken, mit denen induktiv auch vollständige Verifikation ermöglicht wird, wie z.B. [dMRS03], welche selbst die Verwendung von CTL-Formeln nicht ausschliessen.

SAT-basierte Techniken finden ebenfalls Anwendung für Bounded Model-Checking von Hybriden Systemen [WW99, ABC+02, dMRRS03, BDS02]. Hier können ebenfalls Strategien zum Ausschluss generalisierter Konflikte eingesetzt werden, wie das BMC-Werkzeugs HySAT [FH05] demonstriert. Das Hybride System liegt hier als logischer Ausdruck f_H vor, in dem einige boolsche Variablen den diskreten Zustandsraum, und andere den Wahrheitswert linearer (Un-)Gleichungen kodieren, welche die kontinuierlichen Berechnungen des Systems beschreiben. Ein *Backtracking-Verfahren* (Davis-Putnam-Loveland-Logemann, *DPLL*) wird verwendet, um gültige Belegungen der boolschen Variablen zu finden. Für Variablen, die lineare (Un-)Gleichungen kodieren, wird ein Solver mit der Lösung des korrespondierenden (Un-)Gleichungssystems betraut. Existiert keine Lösung, kann das

[9] Natürlich kann jede CTL-Formel für eine begrenzte Schrittzahl in eine LTL-Formel transformiert werden, indem alle im Sinne der CTL-Formel gültigen Läufe der Länge k als LTL-Formeln disjunktiv kombiniert werden. Dies wird auch als *bounded* CTL bezeichnet.

Backtracking Verfahren nach alternativen Belegungen suchen, bis entweder eine Lösung sowohl auf SAT-Ebene, als auch auf Ebene des kodierten (Un-)Gleichungssystems für den gesamten Ausdruck gefunden wird, oder die Existenz einer Lösung ausgeschlossen werden kann. Letzteres ist effizient durch Ausschluß umfassender Teilbelegungsmengen im Rahmen der Baum-förmigen Suchprozedur möglich.

Neben verschiedenen Optimierungen zur Effizienzsteigerung der Belegungssuche werden Konflikte im linearen (Un-)Gleichungssystem nach ähnlichem Muster verarbeitet wie bei dem ω-CEGAR Verfahren. Minimale Konflikte sind hier als Teile des (Un-)Gleichungssystems repräsentiert (*irreducible infeasible subsystems*), die durch Teilbelegungen der boolschen Variablen zur Kodierung des (Un-)Gleichungssystems vorliegen. Die Konflikt-repräsentierenden Teilbelegungen werden wiederum in aussagenlogischer Form f_c kodiert und in negierter Form der zu lösenden Gesamtgleichung f_H konjunktiv hinzugefügt.

Da ein Konflikt c bei k abgerollten Schritten auch hier nach Minimierung nur ein Teilintervall $[k_c, k_c + l_c]$ abdeckt, wird ein globaler Ausschluß des Konflikts durch Reinstanziierung von f_c in den übrigen $k - l_c$ Schritten mit $f_c^1, f_c^2, \ldots, f_c^{k-l_c}$ erreicht (*index shifting*), welche in negierter Form mit der Gesamtformel f konjugiert werden, d.h. in einem Backtracking Schritt erhalten wir die neue Gesamtformel f' aus f mit

$$f' := f \wedge \neg f_c^1 \wedge \neg f_c^2 \wedge \cdots \wedge \neg f_c^{k-l}$$

Somit ist auf SAT-Ebene ausgeschlossen, erneut eine Belegung der Variablen zu versuchen, die den Konflikt c beinhaltet. Diese Technik wird hier als *Isomorphieinferenz* (*isomorphy inference*) bezeichnet und entspricht damit im ω-CEGAR Verfahren dem Lernalgorithmus des ω-Automaten. Während die Verwendung eines ω-Automaten den Gegebenheiten des CTL-Model-Checkings angepasst ist, ist der Reinstanziierungsansatz eine effiziente Realisierung der gleichen Funktionalität auf SAT-Ebene für eine begrenzte Tiefe k.

Aufgrund der engen Integration des Solvers mit dem SAT-basierten Model-Check-Verfahren ist HYSAT sehr effizient. So ist auf kleinen diskreten Zustandsräumen das HYSAT-Verfahren auch deutlich effizienter als ω-CEGAR, da Aufrufe eines externen Model-Checkers hier nicht vorhanden sind. Für größere diskrete Zustandsräume liegen im HYSAT-Verfahren leider noch keine experimentellen Daten vor.

Ein direkter Vergleich mit dem ω-Cegar Verfahren ist jedoch aufgrund der unterschiedlichen Anwendungsfälle ohnehin nur begrenzt möglich. So ist HYSAT nicht für Zertifizierungszwecke geeignet, da ein Ergebnis der Form $H \models \varphi$ nicht ermittelt werden kann. Darüber hinaus existiert bei SAT-Techniken die bereits erwähnte Beschränkung auf LTL-Formeln begrenzter Schrittzahl für die Eigenschaft φ.

4.9.4 Automatenkonstruktionsverfahren

Für die Konstruktion von Büchi-Automaten aus LTL-Formeln werden üblicherweise die Verfahren [GPVW95] sowie deren in [DGV99] beschriebene Erweiterung zitiert. Das Tableau-basierte Verfahren beruht auf einer systematischen Zerlegung der LTL-Formel in atomare Teilformeln und dem Aufbau eines Graphen dessen Knoten, die späteren Zustände, mit Teilmengen der LTL-(Teil-)-Formel beschriftet werden. Der Graph ist prinzipiell exponentiell in der Größe der LTL-Formel und repräsentiert einen Automaten, der i.d.R. nicht minimal ist. Daher werden Heuristiken wie syntaktische Vereinfachungen verwendet, um den Zustandsraum der erzeugten Automaten zu reduzieren.

Insbesondere der Algorithmus in [SB00] erzielt erheblich kleinere Automaten als die Vorgängerverfahren durch Verwendung von Formelumstellungen, bool-

LTL-Formel	spez. Algorithmus Zustände	Zeit	Wring 1.1.0 Zustände	Zeit
$\neg\mathbf{F}(a \wedge \mathbf{X}b)$	2	0,1 s	4	0,1 s
$\neg\mathbf{F}(a \wedge \mathbf{X}(b \wedge \mathbf{X}c))$	4	0,1 s	12	0,2 s
$\neg\mathbf{F}(a \wedge \mathbf{X}(b \wedge X(b \wedge \mathbf{X}c)))$	8	0,1 s	20	0,6 s
$\neg\mathbf{F}(a \wedge \mathbf{X}(b \wedge X(c \wedge \mathbf{X}d)))$	8	0,1 s	36	1,6 s
$\neg\mathbf{F}((a \wedge \mathbf{X}b) \vee (a \wedge X(c \wedge Xb)))$	4	0,1 s	6	0,1 s
$\neg\mathbf{F}((a \wedge \mathbf{X}(b \wedge Xc) \vee (b \wedge X(a \wedge Xd)))$	16	0,1 s	25	1,0 s
$\neg\mathbf{F}((a \wedge \mathbf{X}(b \wedge Xc) \vee (a \wedge X(a \wedge X(d \wedge Xc))))$	64	0,1 s	28	2,6 s
$\neg\mathbf{F}((a \wedge \mathbf{X}(b \wedge Xc) \vee (b \wedge X(e \wedge X(f \wedge Xd))))$	32	0,1 s	180	100,2 s
$\neg\mathbf{F}((a \wedge \mathbf{X}(b \wedge Xc) \vee (b \wedge X(e \wedge X(f \wedge Xd))) \vee (x \wedge Xc))$	64	0,1 s	288	551,7 s
$\neg\mathbf{F}((a \wedge \mathbf{X}(b \wedge Xc) \vee (b \wedge X(e \wedge X(f \wedge Xd))) \vee (c \wedge X(f \wedge X(g \wedge Xh))))$	256	0,1 s	2160	161871 s

Tabelle 4.4: Vergleich der Automatengrößen des speziellen Konstruktionsalgorithmus unter Berücksichtigung der in Abschnitt 4.7.2.1 auf Seite 78 beschriebenen Determinisierung und dem LTL-Formel basierten Wring Algorithmus [SB00, FS01] für einige sehr kleine LTL-Formeln. Die Rechenzeiten wurden auf einem G4-PowerPC-Prozessor mit 1,42GHz mit 512MB RAM Hauptspeicher ermittelt

schen Optimierungstechniken und Vereinfachungen der Transitionsstruktur und Akzeptanzbedingungen des erzeugten Büchi-Automaten durch Simulation desselben. Der Algorithmus geht dabei von atomaren Propositionen aus, welche sich nicht gegenseitig ausschließen, d.h. die LTL-Formeln können auch atomare Propositionen beinhalten, welche im gleichen Schritt gelten können, d.h. die Läufe des konstruierten Automaten akzeptieren nicht Wörter bestehend aus atomaren Propositionen, sondern aus Teilmengen der Menge aller atomaren Propositionen. Daher ist für Vergleiche der in Abschnitt 4.7.2.1 auf Seite 78 beschriebene determinisierte Automat heranzuziehen.

Der Algorithmus funktioniert sehr verschieden von dem in Abschnitt 4.3.3.2 auf Seite 52 dargelegten, weshalb hier auf einen operativen Vergleich der Algorithmen verzichtet wird. Der Algorithmus ist in einer in Perl geschriebenen Implementierung namens *Wring* verfügbar [FS01], welche ersatzweise im Folgenden für experimentelle Vergleiche mit dem hier vorgestellten spezialisierten Algorithmus herangezogen wird.

Auch Wring erwartet gemäß des in [SB00] beschriebenen Verfahrens sich nicht ausschließende atomare Propositionen. Ein Vergleich mit Automaten mit einfachem Alphabet ist daher leider nicht möglich.

In Tabelle 4.4 sind für verschiedene LTL-Formeln der für diesen Kontext relevanten Struktur mit beiden Verfahren in entsprechende Büchi-Automaten übersetzt worden und die resultierenden Automatengrößen gegenübergestellt. Für einige Formeln finden sich auf Seite 97 die jeweilig erzeugten Automaten für einige der aufgeführten Formeln. Unter Berücksichtigung des Anwendungsfalls dieser Arbeit sind die Formeln als ausgesprochen klein zu bezeichnen. Dennoch zeigt sich bereits hier ein deutlicher Unterschied in den Automatengrößen, welcher insbesondere mit längeren Konfliktsequenzen exponentiell anzusteigen scheint.

Die Laufzeiten liegen für das spezialisierte Konstruktionsverfahren alle unter 0,1 Sekunden, während die Wring-Implementierung überproportional zur erzeugten Automatengröße anwächst und schon für sehr kleine Formeln den zeitlich akzeptablen Rahmen sprengt. Da es sich bei Wring um eine Perl-Implementierung handelt, können die Zeitangaben, wie von den Autoren in [SB00] angegeben, für

den Vergleich mit einer C-Implementierung um eine Größenordnung niedriger eingestuft werden, was jedoch den Unterschied bzgl. der Effizienz kaum verändert.

Determinisierter ω-Automat

Wring 1.1.0

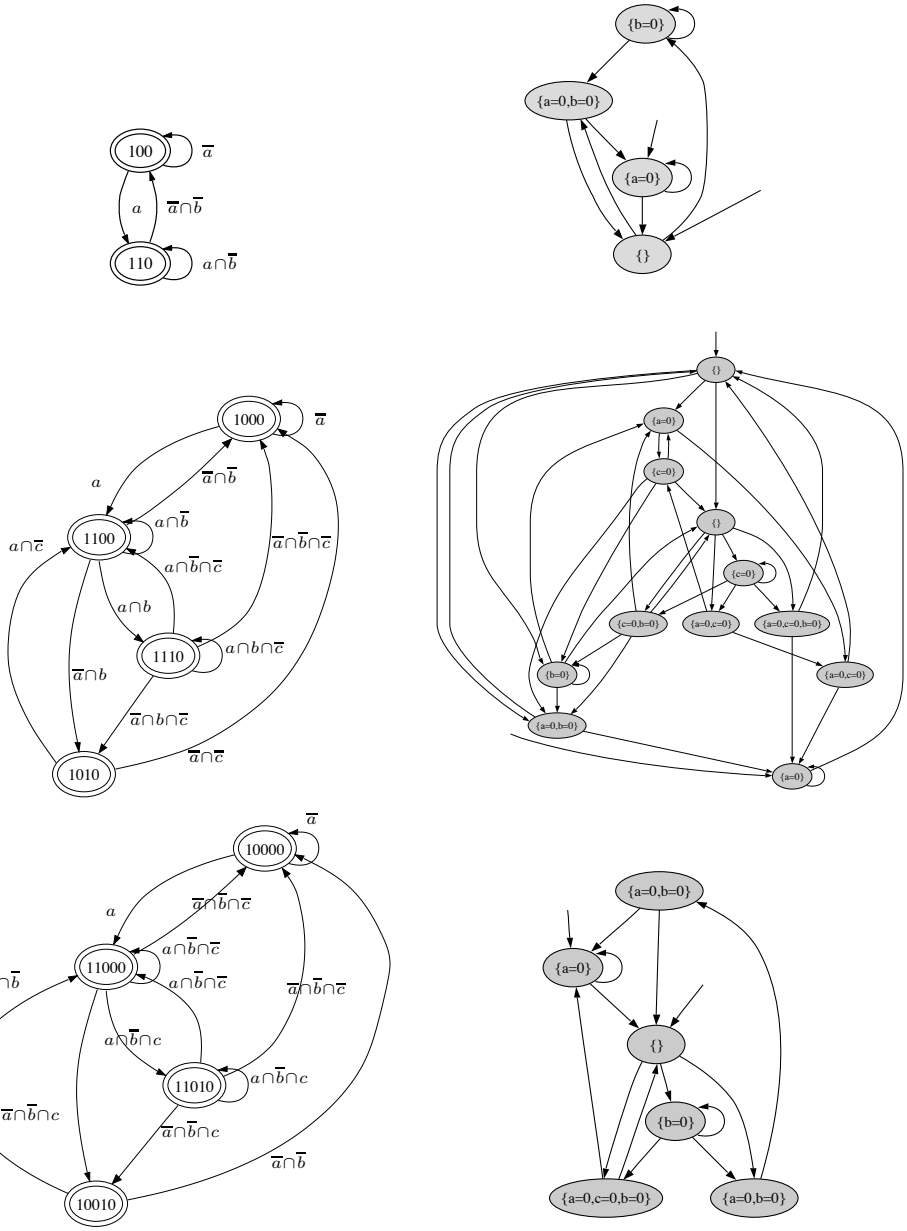


Tabelle 4.5: Gegenüberstellung der Automaten für die LTL-Formeln $\neg\mathbf{F}(a \wedge \mathbf{X}b)$, $\neg\mathbf{F}(a \wedge \mathbf{X}(b \wedge \mathbf{X}c))$ und $\neg\mathbf{F}((a \wedge \mathbf{X}b) \vee (a \wedge X(c \wedge Xb)))$ mit a, b, c als Teilmengen von GJ zum Ausschluß von Teilmengensequenzen. Links sind die determinisierten ω-Automaten gemäß der Konstruktion aus Abschnitt 4.7.2.1 auf Seite 78 dargestellt, rechts die mit dem Werkzeug Wring [FS01] nach [SB00] erzeugten. Da in der Wring-Darstellung die nicht-akzeptierenden Senken-Zustände nicht aufgeführt sind, sind diese auch in den Ergebnissen der ω-Automatenkonstruktion nicht dargestellt, zumal diese für das Kreuzprodukt nach Abschnitt 4.3.4 auf Seite 63 nicht relevant sind.

Kapitel 5

Fallbeispiel

In diesem Kapitel soll ein in späteren Experimenten verwendetes Modell, welches mit einem CASE-Tool erstellt worden ist, vorgestellt werden. Dies schließt eine kurze Vorstellung von CASE-Tools ein, mit der zunächst begonnen wird.

5.1 CASE-Tools

Wurde früher die Software für Steuerungssysteme noch in C-Code oder gar Assembler geschrieben, so hat sich in den letzten Jahren mehr und mehr der Einsatz von rechnergestützter Softwareentwicklung (*Computer Aided Software Engineering*) durchgesetzt. Die hierzu verwendeten Werkzeuge werden allgemein als CASE-Tools bezeichnet. Wie bereits in der Einleitung erwähnt, bieten diese gute Hilfestellungen in fast allen Schritten des V-Modell-basierten Entwurfsprozesses. Die einzelnen am Markt verbreiteten CASE-Tools unterscheiden sich in der Darstellung der Systeme, dem zur Verfügung stehenden Sprachumfang, sowie den zur Verfügung stehenden Simulations-, Analyse- und Code-Generierungsfunktionalitäten in unterschiedlichem Ausmaß. Im Folgenden seien zunächst typische Sprachelemente vorgestellt, bevor im Anschluss Besonderheiten und Einschränkungen der jeweiligen CASE-Tools diskutiert werden.

5.1.1 Sprachelemente

Mit allen CASE-Tools lassen sich *Zustandsautomaten* zur Spezifikation von diskreten Kontrolloperationen spezifizieren. Diese Zustandsautomaten verfügen gegenüber den in Kapitel 2.1 vorgestellten Automaten über erheblich mehr Sprachelemente zu deren Spezifikation. So können Zustandsautomaten hierarchisch gegliedert, parallel ausgeführt und mit ebenfalls hierarchischen Transitionen ausgestattet werden. Bei den meisten CASE-Tools können Transitionen durch Bedingungen eingeschränkt und durch Aktionen zur Ausführung beliebig komplexer Berechnungen erweitert werden.

Neben den Zustandsautomaten verfügen die meisten CASE-Tools über die Möglichkeit, *Datenflußgraphen* explizit zu modellieren. Hier können hierarchisch strukturierte Funktionselemente durch *unidirektionale Signale* verbunden werden. Die verwendbaren Elemente im Datenflußgraphen reichen von einfachen logischen und arithmetische Operatoren über Verzögerungselemente (*Latches*) bis hin zur Verwendung mehrdimensionaler Kennlinienfelder einschließlich nicht-linearer Interpolationsfunktionen. Dabei existiert grundsätzlich die Einschränkung, daß der Datenflußgraph keine Schleifen beinhalten darf, die nicht durch Verzögerungs-

elemente unterbrochen sind. Andernfalls würde die Ausführung der Berechnung nicht terminieren.

Desweiteren verfügen alle CASE-Tools über der Möglichkeit, extern spezifizierten C-Code in das Modell einzubinden, beispielsweise als Funktionsaufrufe in Transitionsaktionen oder Funktionen im Datenflußgraphen.

Neben strukturierten, diskreten und ganzzahligen Datentypen, verfügen alle CASE-Tools über den für diese Arbeit besonders interessanten Datentyp eines kontinuierlichen Wertebereichs.

Alle in diesem Abschnitt vorgestellten CASE-Tool Sprachen sind deterministisch, was jedoch nicht grundsätzlich zutreffend ist. So kann mit dem Werkzeug STATEMATE ebenso Nicht-Determinismus modelliert werden.

5.1.2 Schritt-Funktion

Der Synchronen Hypothese aus [BB91] folgend wird in der Semantik der jeweiligen CASE-Tool-Sprachen grundsätzlich von einer *instantanen* Ausführung aller Rechenoperationen ausgegangen, an denen keine explizite Zeitspanne gewartet werden soll. Dies bedeutet, daß Aktionen an Transitionen in Zustandsautomaten ebenso wie Datenflußgraphen abarbeitende Prozesse idealisierend keine Zeit verbrauchen. Ein Fortschreiten der Zeit wird allgemein durch übergeordnete Komponenten bestimmt, welche z.B. im Rahmen einer Zeitablaufsteuerung (*Scheduling*) das spezifizierte Modell periodisch oder ereignisgetrieben ausführen. Während Zeit verstreicht, kann sich weder der diskrete, noch der kontinuierliche Zustand des Systems ändern. Hier ist demnach die Definition 9 Schritt-diskreter Hybrider Systeme zutreffend, da es keine Zeit-kontinuierlichen Zustandsveränderungen gibt.

Es existieren also in allen CASE-Tool-Sprachen definierte Teilmengen von ununterbrechbaren terminierenden Berechnungsabläufen des Systems, die der Semantik nach instantan ausgeführt werden. Einfache Systeme verfügen über genau einen solchen Berechnungsablauf, der analog zu einem Funktionsaufruf eine Zustandstransformation vornimmt und die Kontrolle zurück gibt. Einen solchen Ablauf, der durch eine Funktion beschrieben werden kann, bezeichnen wir als *Schritt-Funktion* des Systems. In Analogie zu Transitionssystemen und Schritt-diskreten Hybriden Automaten beschreibt die einmalige Ausführung einer Schritt-Funktion einen diskreten Zustandswechsel über eine Transition. Wir sprechen in diesem Zusammenhang auch von einem *Schritt* des Systems.

5.1.3 Automatische Code-Generierung

Alle CASE-Tools verfügen über einen oder mehrere Code-Generatoren, welche aus den zumeist piktographisch vorliegenden Modellen ausführbaren Programmcode in einer gängigen Sprache generieren. Dies ist meistens ein Generator für die Sprache C [KR90], welche sich einfach mit handgeschriebenem C-Code integrieren, für die Zielprozessoren des eingebetteten Steuerungssystems kompilieren und auf diesen somit effizient ausführen läßt. Andere, durch Code-Generatoren erzeugte Sprachen sind ADA [TD95], Verilog [Hyd97], VHDL [Iee00] und Java [Gra]. Die Code-Generatoren sind dabei i.d.R. mannigfaltig parametrierbar hinsichtlich

- Zielprozessoren,
- der Verwendung von Gleitpunkt- oder Fixpunktkodierungen,
- der Verwendung von strukturierten Datentypen,

- der Verwendung von Unterfunktionsaufrufen gegenüber durchgeführtem *Inlining*,
- der Einführung von Wertebereichsbegrenzungscode (*Limiter*)
- u.v.a.

Da die Code-Generatoren Code in Sprachen erzeugen, die über eine definierte Semantik verfügen, entspricht dies in häufig vorkommender Ermangelung entsprechender Semantik-Definitionen der Ausgangssprachen[1] de facto einer indirekten, translativen Semantikbeschreibung. Daher macht i.d.R. auch der den CASE-Tools beiwohnende Simulator von dem generierten Code zur Durchführung der Simulation Gebrauch.

Der generierte Code dient primär der Erzeugung das Codes, welcher auf dem Zielsystem ausgeführt wird (*Target-Code*), eignet sich aufgrund seiner zentralen Rolle jedoch auch ideal als Ausgangspunkt einer Verhaltensbeschreibung des repräsentierten Modells, weswegen der von jedem CASE-Tool generierbare C-Code auch ideal für die Verifikation verwendet werden kann, wie wir im folgenden Kapitel sehen werden. Bemerkenswert ist hierbei der Einschluß von Code-Generierungsfehlern im Verhaltensmodell, d.h. weicht der generierte Code semantisch von der graphisch modellierten Intention des Entwicklers ab, wird dies bei entsprechend spezifizierter Verhaltensanforderung durch Verifikationsverfahren aufgedeckt. Daher ist in der Industrie ein zunehmendes Interesse an der bevorzugten Verifikation genau des C-Codes zu beobachten, der in der dazugehörigen Parametrierung auf der Zielarchitektur letztlich eingesetzt wird. Angesichts der zu beobachtenden Häufigkeit von Fehlern in Code-Generatoren ist ein solches Vorgehen begründet.

5.1.4 Exemplarisch verwendete CASE-Tools

5.1.4.1 SIMULINK/STATEFLOW

Das SIMULINK/STATEFLOW Werkzeug [The03] wird von der Firma *The MathWorks* vertrieben und erfreut sich großer Bekanntheit. Es wird für den Entwurf eingebetteter Systeme in allen Anwendungsbereichen eingesetzt. Die obig aufgeführten Sprachelemente sind in der diesem Werkzeug zugrunde liegenden Sprache alle vorhanden. Abbildung 5.1 zeigt die graphische Darstellung eines Systems in Simulink.

Besonderheiten im Sprachumfang sind hier insbesondere die sog. Ereignisse (*Events*), die in Zustandsautomaten als Aktionen ausgesandt werden können. Diese können entweder gezielt an andere Zustandsautomaten adressiert, oder global an alle Zustandsautomaten übertragen werden. Auf ein Event wird in der STATEFLOW-Semantik instantan reagiert, d.h. die Kontrolle wird umgehend an den Empfänger des Events weitergegeben, bis dieser das Event verarbeitet hat. Im Falle von globalen Events kann dies durchaus zu einer Endlosrekursion führen, da auch der absendende Automat das Event empfangen und verarbeiten kann. Bei dieser Semantik weicht der Transitionsbegriff erheblich von denen einfacher Transitionssysteme ab, da mehrere Transitionsübergänge in einem Schritt stattfinden können (*Transiente Zustände*).

5.1.4.2 SCADE

Das Scade-Werkzeug , welches intern auf die Datenfluß-orientierte Sprache LUSTRE [HCRP91] aufsetzt, wird von der Firma Esterel Technologies vertrieben und

[1] Die Sprache SCADE ist dank der ihr zugrunde liegenden Sprache LUSTRE [HCRP91] die einzige über eine definierte Semantik verfügende Sprache.

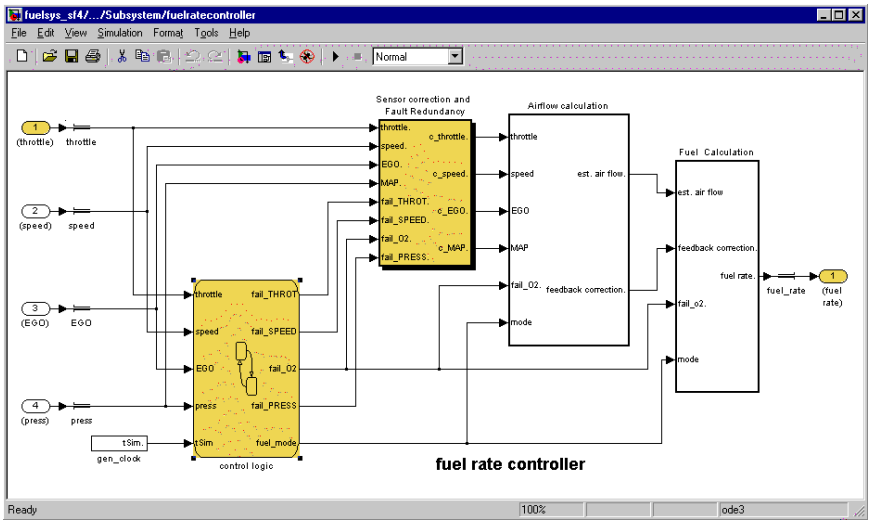


Abbildung 5.1: Das SIMULINK/STATEFLOW Werkzeug

wird vorwiegend im Luft- und Raumfahrtbereich für die Erstellung von Avionik-systemen verwendet. Die Sprachelemente und die Semantik dieser Sprache sind sehr einfach gehalten. Da SCADE im Wesentlichen eine graphische Erweiterung für LUSTRE ist, sind an komplexeren Sprachelementen auch nur solche anzutreffen, die in der Datenflußsprache LUSTRE modelliert werden können. Selbst Zustandsautomaten werden hier in Datenflußoperationen übersetzt, was die Verwendung von dynamischen Berechnungen an Transitionen ausschließt[2].

5.1.4.3 ASCET

Das Werkzeug ASCET wird von der Firma ETAS entwickelt und findet hauptsächlich Anwendung in der Fahrzeugtechnik. Entsprechend verfügt das Werkzeug über alle Wesentlichen in der Fahrzeugtechnik benötigten Funktionalitäten wie beispielsweise Unterstützung eines OSEK-kompatiblen Betriebssystems. Die Modellierungssprache umfasst objektorientierte Ansätze ohne Vererbung und weist einige Besonderheiten auf. So kann im Datenflußgraphen (*Blockdiagramme*) Kontrollfluß explizit modelliert werden, indem entsprechende Kontrollverbindungen die Ausführung von Datenflußgraphelementen auslösen können. Darüber hinaus ist in Datenflußgraphen die inhärente partielle Ordnung für semantische Ausführungsreihenfolge irrelevant, die Auswertung der einzelnen Knoten im Datenflußgraphen erfolgt nach explizit festgelegter Reihenfolge, wodurch zwangsläufig semantische Verzögerungselemente eingeführt werden können. Zustandsautomaten besitzen eine Anzahl von sog. Triggerfunktionen, welche jeweils eine Teilmenge der Transitionen ausführen können. In ASCET besteht weiterhin die Möglichkeit, Verhalten durch sog. ESDL Code zu spezifizieren, oder alternativ in C. In Abbildung 5.2 ist ein Bild des Werkzeugs dargestellt.

5.1.5 Den Zustandsbits auf der Spur

Wir haben in Abschnitt 4.1.2 bereits diskutiert, wodurch ein großer diskreter Zustandsraum aus systemfunktionaler Sicht in den Hybriden Modellen verursacht

[2]Die Mächtigkeit der Sprache LUSTRE ist natürlich grundsätzlich ausreichend, um jedes Sprachelement darauf semantisch korrekt abzubilden, jedoch wäre deren Verwendung bei solchen Umgehungen der Sprachrestriktionen dann zu hinterfragen.

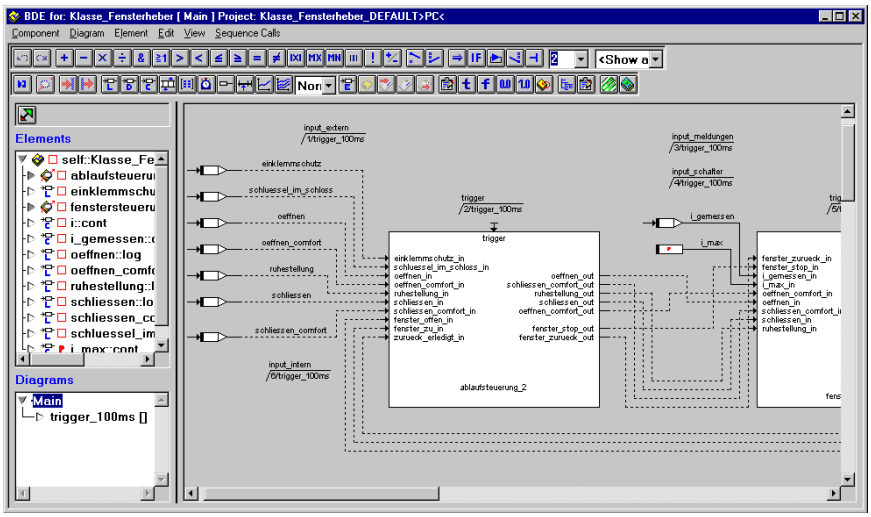


Abbildung 5.2: Das Werkzeug ASCET

wird. Mit Hinblick auf die existierenden CASE-Tools wollen wir nun detaillierter recherchieren, durch welche Sprachkonstrukte der diskrete Zustandsraum vergrößert wird.

Zunächst wären die offensichtlichen Elemente anzuführen:

- Zustandsautomaten
- Explizite Latches in Datenflußgraphen, insbesondere solche für Integer-Variablen größeren Wertebereichs

Neben diesen offensichtlichen existieren jedoch eine Reihe versteckter Quellen, welche möglicherweise, jedoch nicht zwingend, den Zustandsraum vergrößern:

- In Zustandsautomaten verwendete diskrete Variablen in den den Transitionen zugeordneten Aktionen
- In Datenflußgraphen verwendete diskrete Variablen (keine Latches)

Andere ebenfalls unauffällige Elemente führen zwingend zu einer Vergrößerung des Zustandsraums:

- Bibliothekselemente wie Zeit-diskrete Transferfunktionen oder Switch-Blöcke verwenden ihrerseits eigene Latches
- Unterschiedliche Taktungen (*Clocks*) von Teilen des Datenflußgraphen erzwingen nicht explizit sichtbare Latches für alle Ausgänge von bedingt auszuwertenden Teilen des Modells
- Zur einmaligen Durchführung von Initialisierungsberechnungen ist ein globales Zustandsbit zur Unterscheidbarkeit vorzuhalten

Da die jeweiligen Zustandsanzahlen zu multiplizieren sind, kommt bei Verwendung entsprechend vieler Elemente dieser Art schnell ein ansehnlicher diskreter Zustandsraum zusammen. Mit diesen Elementen werden die in Abschnitt 4.1.2 bereits aufgezählten, diskrete Zustände erfordernden Funktionalitäten modelliert. Wir werden für das nun folgend vorgestellte Fallbeispiel ebenfalls im Anschluß eine konkrete Analyse der Herkunft der diskreten Zustandsbits durchführen.

5.2 Modell eines Autopilot Systems

Nun wird ein größeres Fallbeispiel vorgestellt, welches mit einem CASE-Tool erstellt worden ist. Die Wahl fiel dabei auf ein mit SCADE modelliertes System, was aufgrund der Einfachheit der Sprache SCADE gut zu erläutern ist. Es handelt sich um ein stark vereinfachtes Autopilotsystem namens *Pilot*, welches die Höhe eines Flugobjekts kontrolliert und im Benutzerhandbuch [VER98] des SCADE-Werkzeugs verwendet wurde. Das System berechnet unter Berücksichtigung von im Sekundentakt vorliegender telemetrischer Informationen und einem Kommando den Soll-Geschwindigkeitsvektor des Flugobjekts. Weicht die durch nachfolgende Telemetrieeingaben ermittelte Trajektorie des Flugobjekts von der gewünschten ab, wird der Soll-Geschwindigkeitsvektor entsprechend korrigiert. Das System wechselt in einen Sicherheitsmodus (*Safety Mode*), wenn die Höhe einen kritischen Wert unterschreitet oder Telemetrieinformation bzw. hieraus errechnete Geschwindigkeitsvektoren ungültig sind. Die Telemetrieinformation wird durch zwei unabhängige Positionsvektoren angegeben, welche zusätzlich mit entsprechenden Gültigkeitsbits (*Qualifier Bits*) für jede Koordinate einhergehen. Positions- und Geschwindigkeitsangaben werden durch Vektoren in $\mathbb{R}^3$ repräsentiert.

Neben diesem Steuerungssystem beinhaltet das Modell ein einfaches Umgebungsmodell, welches abhängig von einem variierenden Windgeschwindigkeitsvektor die Trajektorie des Flugobjekts im Raum berechnet. In Abweichung von der in Abschnitt 1.3 aufgestellten These bezüglich der selteneren Verifikation von geschlossen Regelkreisen gegenüber der häufigeren Verifikation von isolierten Steuerungssystemen werden wir das Umgebungsmodell sowohl in der Vorstellung des Systems, als auch bei der späteren Durchführung von Beweisaufgaben miteinbeziehen, weil das Gesamtmodell hierdurch umfangreicher und anspruchsvoller wird. Da selbst das Umgebungsmodell die Störgröße Wind als Eingabegröße beinhaltet, handelt es sich immernoch um einen offenen Regelkreis.

Die Funktionen der im Beispiel vorkommenden graphischen Elemente ist im Anhang in Abbildung A.4 erläutert und sollten ausreichend sein, um mit den nachfolgenden Erläuterungen die Funktionalität des Modells nachzuvollziehen. In Abbildung 5.4 ist die hierarchisch oberste Ebene des Modells durch einen Operator namens *Flight*[3] dargestellt, in der linken Hälfte das Umgebungsmodell, bestehend aus dem Operator *ExternalConditions* mit zwei vorgeschalteten *fby*-Operatoren[4] und des Eingabesignals des Windvektors, und auf der rechten Seiten das Steuerungssystem *Pilot*, welches zwei Positionsvektoren und einen Kommandocode einschließlich dem diesen zugeordneten boolschen Gültigkeitssignal als Eingabe erhält. Der Ausgang des *Pilot*-Systems ist der Geschwindigkeitsvektor des Flugobjekts. Im oberen Teil der Abbildung ist ein Mechanismus zur Deaktivierung des Umgebungsmodells zu sehen, d.h. die Variable *automode* kontrolliert die Öffnung bzw. Schließung des Regelkreises. Wenn $automode = true$[5] wird die Berechnung neuer Flugobjektpositionen durch den *ExternalConditions*-Operator[6] ausgesetzt und die Eingabevektoren *PtA* und *PtB* werden stattdessen verwendet. Die Modellierung der Funktionalität von *ExternalConditions* ist der Abbildung 5.3 zu entnehmen. Hier wird für jeden Positionsvektor $\vec{p}$ unter Beibehaltung der diskreten Gültigkeitsinformation die neue Position $\vec{p}'$ mit $\vec{p}' = dt \cdot (\vec{v}_{wind} + \vec{v}_{obj}) + \vec{p}$ berechnet, was nach dem physikalischen Weg-Zeit-Gesetz $v = \frac{s}{t}$ gerade die in der Zeit dt zurückgelegte

External-Conditions

[3] Um Verwirrungen vorzubeugen: Das Gesamtmodell heißt *Pilot*, der Hierarchie-oberste Operator des Modells *Flight*, welcher wiederum ein Steuerungssystem *Pilot* als Sub-Operator verwendet.

[4] Diese verzögern das Signal um einen Clock-Zyklus, siehe auch Abbildung A.4

[5] Intuitiv könnte man angesichts des Bezeichners eine andere Semantik für diese Variable vermuten. Die Verwendung läßt jedoch keine andere Deutung zu.

[6] Dieser ist in einen Condact-Operator eingebettet, der eine bedingte Ausführung bewirkt. Siehe auch Beschreibung der Operatoren in Abbildung A.4

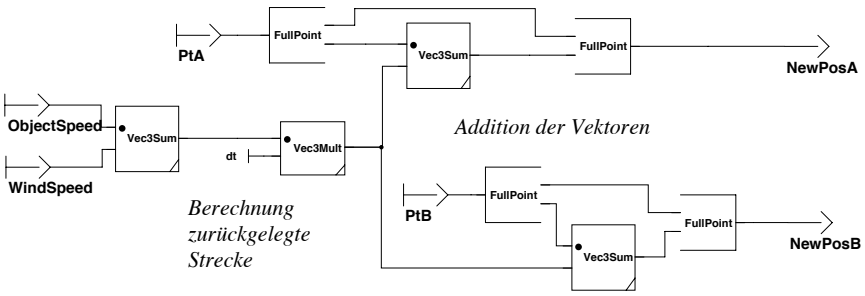


Abbildung 5.3: Der Operator *ExternalConditions* zur Berechnung der neuen Position des Flugobjekts

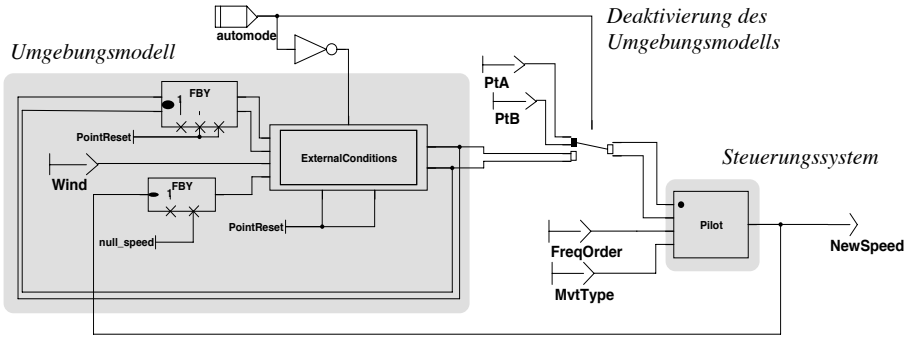


Abbildung 5.4: Das Autopilotsystem mit Umgebungsmodell zur geschlossenen Positionsberechnung mit der Störgröße Wind. Positions- und Windsignale sind Tripel kontinuierlichen Wertebereichs. Positionssignale beinhalten zudem für jede Koordinate ein Gültigkeitssignal.

Strecke des Objekts unter der Geschwindigkeitsresultierenden ist. Die Variable *dt* repräsentiert in dem Modell global die seit der letzten Auswertung verstrichene Zeit und ist im vorliegenden Fall eine Konstante mit dem Wert 1, in Übereinstimmung mit dem oben erwähnten Sekundentakt, in dem neue Telemetriedaten vorliegen sollen.

Pilot

Der Modellierung des *Pilot*-Operator ist in Abbildung 5.5 dargestellt. Der *Control*-Operator verarbeitet die Positionsangaben und berechnet daraus den Ist-Geschwindigkeitsvektor, sofern die Gültigkeitsbits der Positionskomponenten bestimmten Stabilitätsbedingungen genügen. Der Ist-Vektor wird hernach durch den *Command*-Operator weiterverarbeitet, indem abhängig vom Kommando Steig-, Sinkflug oder höhenneutrale Geschwindigkeitsvektorkorrekturen vorgenommen werden.

GetMvtMode

Der Operator *GetMvtMode* dient dabei einer Kommandokorrektur im Falle von ungültigen, im *Control*-Operator berechneten Konsistenzbits, so daß unabhängig vom außen angelegten Kommando beispielsweise im Falle ungültiger Telemetriedaten das Ersatzkommando für horizontalen Flug (*mvt_safety*) weitergeleitet wird. Die Modellierung dieses Operators ist in Abbildung 5.6 zu finden.

Control

Die Berechnung des Ist-Geschwindigkeitsvektors erfolgt durch den *Control*-Operator wie in Abbildung 5.7 spezifiziert. In der oberen Hälfte befindet sich die

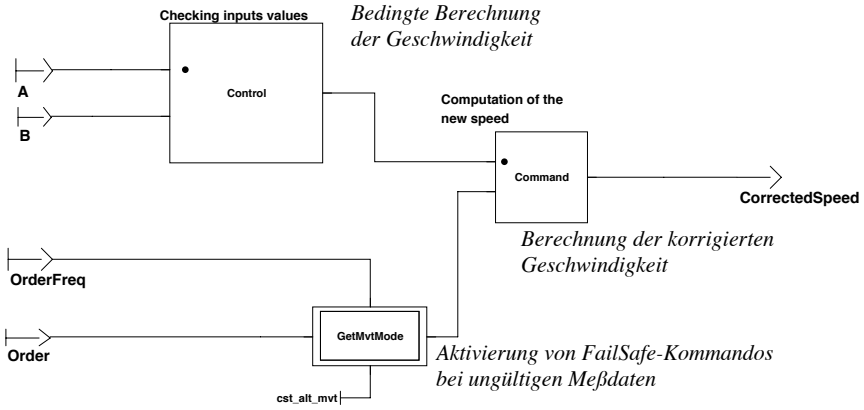


Abbildung 5.5: Funktionsweise des *Pilot*-Operators

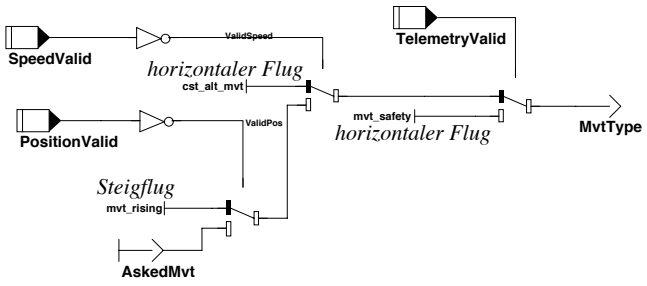


Abbildung 5.6: Der Operator *GetMvtMode* zur Berechnung von Ersatzkommandos bei unklarer Situationslage

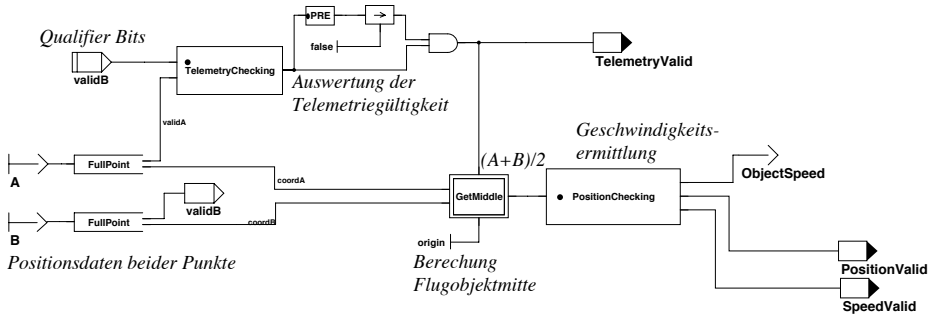


Abbildung 5.7: Berechnung der Geschwindigkeit durch den Operator *Control*

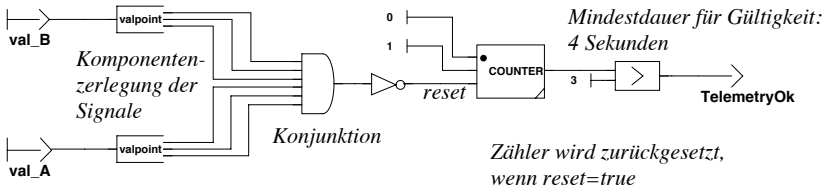


Abbildung 5.8: Der *TelemetryValid*-Operator

Auswertung der Gültigkeitsinformation der Telemetriemeßdaten, welche in der Datentyp-Struktur *FullPoint* der Positionsdaten verborgen waren. Die Berechnung erfolgt im *TelemetryValid*-Operator auf Basis einer durchgehenden Gültigkeit aller Koordinatenmeßwerte für mindestens 4 Sekunden[7], wie in Abbildung 5.8 zu erkennen ist. Das Ergebnis der Auswertung wird, nach Verlängerung der Mindesgültigkeitsdauer um eine weitere Sekunde durch die Konjunktion mit dem vorherigen Wert, zum Einen in die globale Variable *TelemetryValid* geschrieben, zum Anderen wird diese Information zur bedingten Ausführung der Berechnung der Flugobjektmitte beider Positionspunkte A und B für den nachfolgenden *PositionChecking*-Operator genutzt. Falls die Telemetriedaten nicht als gültig erachtet worden sind, wird von dem den *GetMiddle*-Operator einschließenden *Condact*-Operator der zuletzt berechnete Wert weitergegeben.

TelemetryValid

PositionChecking

Der *PositionChecking*-Operator errechnet aus dem Positionsmittelwert nun den Ist-Geschwindigkeitsvektor und weitere, die Position und Geschwindigkeit betreffende Gültigkeitsbits wie in Abbildung 5.9 dargestellt. Dazu wird von dem Positionsvektor $\vec{p}$ der vorherige Positionsvektor $\vec{p'}$ abgezogen, so daß wir $\Delta\vec{p} = \vec{p} - \vec{p'}$ erhalten. Zur Berechnung des Geschwindigkeitsvektors $\vec{v} \in \mathbb{R}^3$ wird im Rahmen einer Skalarmultiplikation die Berechnung $\vec{v} = \Delta\vec{p} \cdot \frac{1}{dt}$ durchgeführt und das Ergebnis für die Bestimmung des Gültigkeitssignals *SpeedOk* hinsichtlich seiner z-Koordinate und seines Betrags[8] auf korrekte Erfüllung entsprechender Mindestvorgaben überprüft. Zudem wird im oberen Teil der Abbildung das Ausgabesignal *PositionOk* durch eine Mindesthöhenüberprüfung ermittelt.

Command

Damit kommen wir nun zum dritten und letzten Operator in der *Pilot*-Funktionalität, dem *Command*-Operator, welcher in Abbildung 5.10 dargestellt ist. Dieser Berechnet zunächst mit dem *MvtCalculus*-Operator eine theoretische Geschwindig-

MvtCalculus

[7]Die Periodizität der Ausführung der Berechnungen ist mit 1 Hz spezifiziert.

[8]Dies ist die einzige Nicht-Linearität im Modell, der bei der Anwendung des ω-CEGAR Verfahrens durch einfache Abstraktion begegnet wird.

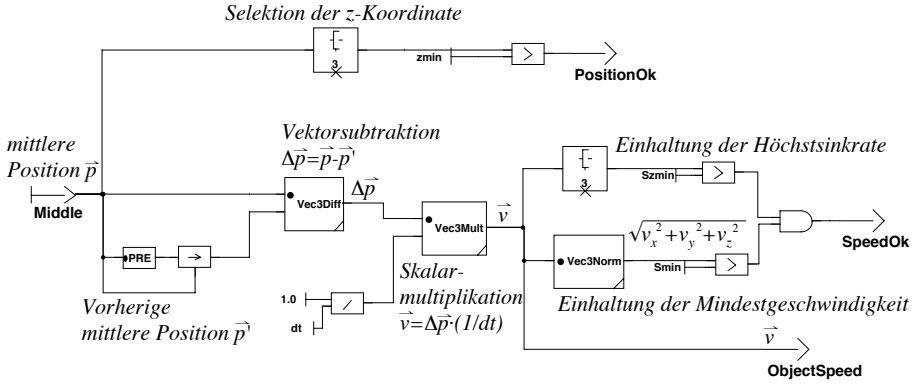


Abbildung 5.9: Der Operator *PositionChecking* zur Berechnung des Ist-Geschwindigkeitsvektors

keit auf Basis der Ist-Geschwindigkeit und des durchzuführenden Kommandos. Die Funktionsweise dieses Operators ist der Abbildung 5.11 zu entnehmen. Hier werden parallel verschiedene Flugbahnen zu den Szenarien Sinkflug, Steigflug und Warten, sowie die Sicherheitsmodi mit Beibehaltung der Flughöhe berechnet, von denen eine durch den Selektions-Operator als theoretische Geschwindigkeit weitergeleitet wird. Für die ersten drei Szenarien werden durch C-Code extern modellierte Funktionen verwendet, welche die folgenden einfachen Berechnungen[9] durchführen:

- Sinkflug (*Approach_FL*)

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} := \begin{cases} \begin{pmatrix} v_x \\ v_y \\ v_z - 5.0 \end{pmatrix} & \text{gdw } v_z + 5.0 > \mathit{Speedzmin} \\ \vec{v} & \text{sonst} \end{cases}$$

- Steigflug (*Rising_FL*)

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} := \begin{pmatrix} v_x \\ v_y \\ v_z + 30.0 \end{pmatrix}$$

- Warteposition[10] (*Waiting_FL*)

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} := \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix}$$

Regulation

Zurück im *Command*-Operator wird der theoretische Geschwindigkeitsvektor dem nachfolgenden *Regulation*-Operator zur Weiterverarbeitung übergeben, welcher in

[9] An dieser Stelle sind sicherlich komplexere, nicht-lineare Funktionen vorstellbar und realistischer. Da das Modell jedoch didaktischen Zwecken diente, ist hier auf eine einfache Funktionalität zurückgegriffen worden, was der Behandelbarkeit durch lineare Verfahren jedoch sehr entgegen kommt.

[10] Französischen Originalkommentaren im C-Code ist zu entnehmen, daß auch ein Einfrieren der v_z-Komponente angedacht war, was die Unterscheidung zu den Sicherheitsmodi erklären würde.

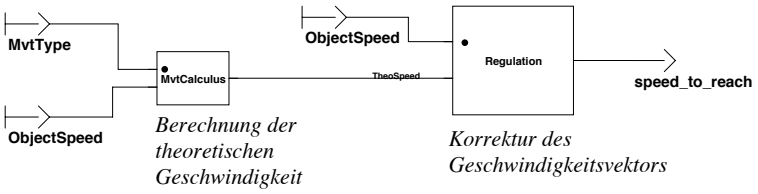


Abbildung 5.10: Der Operator *Command* dient der Berechnung des neuen Soll-Geschwindigkeitsvektors

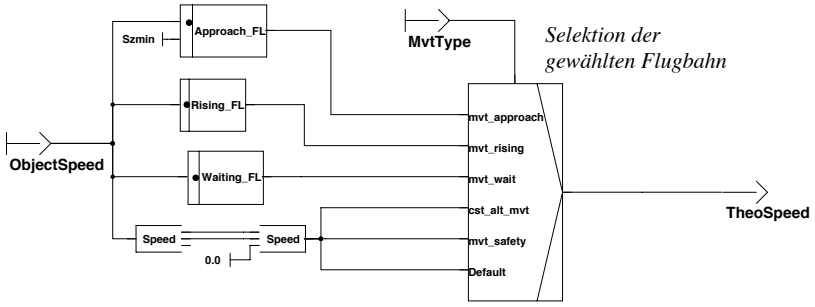


Abbildung 5.11: Der *MvtCalculus*-Operator zur Berechnung der theoretischen Geschwindigkeit. Die Variable *MvtType* ist vom Typ *Integer* und die Konstanten im Selektionsblock sind *mvt_approach* = 1, *mvt_rising* = 2, usw.

Abbildung 5.12 zu finden ist. Dieser erhält außerdem den Ist-Geschwindigkeitsvektor und hat nun die Aufgabe, selbigen in kleinen Schritten an die theoretische Geschwindigkeit heranzuführen. Dazu wird die zuletzt beobachtete Drift, der Unterschied zwischen Ist- und Soll-Geschwindigkeit des vorherigen Zyklusses, mit 5% gewichtet und die Ist-Geschwindigkeit hiermit korrigiert[11].

Abschließend soll auch die Ansteuerung der Variablen *automode* zur Steuerung der Öffnung und Schließung des Regelkreises nicht vorenthalten werden. In Abbildung 5.13 ist zu erkennen, wie das Eingabesignal *simulation* zusammen mit den Gültigkeitsinformationen der Geschwindigkeit und der errechneten Position von einem Zustandsautomaten verarbeitet werden. Der Regelkreis ist nur dann geschlossen, wenn der Automat sich nicht im Zustand *Auto* befindet[12].

Die übrigen im Modell verwendeten Operatoren *COUNTER*, *Vec3Diff*, *Vec3Mult*,

[11] Das SCADE-User-Manual beschreibt dies als Korrektur der theoretischen Geschwindigkeit in Abhängigkeit der im letzten Schritt beobachteten Drift, jedoch verwundert in diesem Falle, daß de facto die Ist-Geschwindigkeit korrigiert wird und die Drift mit 5% nur marginalen Einfluß ausübt.

[12] Entgegen der Intuition, aber im Modell so verwendet.

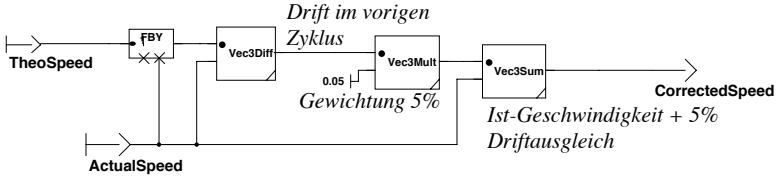


Abbildung 5.12: Korrektur des Geschwindigkeitsvektors durch den *Regulation*-Operator

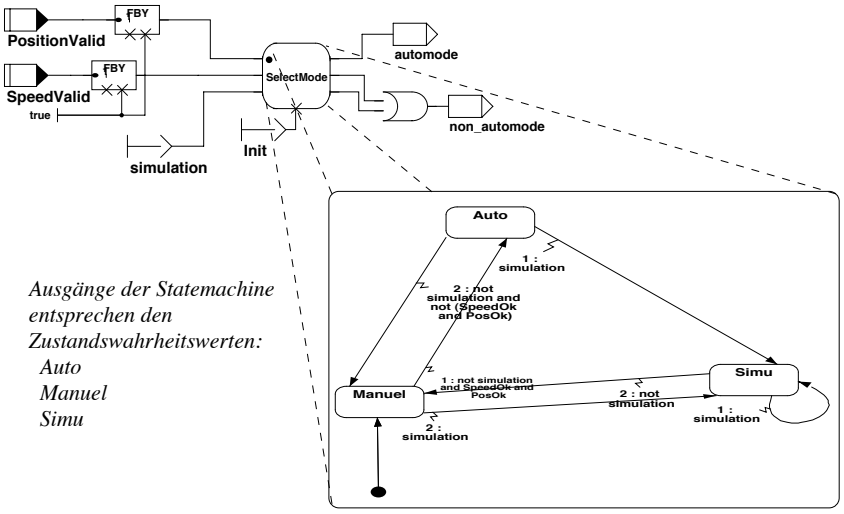


Abbildung 5.13: Berechnung des Inhalts der Variable *automode*. Dieses Diagramm ist der obersten Hierarchie-Ebene in Abbildung 5.4 zugehörig, das Signal *simulation* mithin also ein Eingangssignal des Gesamtmodells

Vec3Norm, *Vec3Prod*, *Vec3Sum* sind im Anhang A.3.1 als Lustre-Funktionen zu finden.

Da das Modell didaktischen Zwecken zur Erlernung der Benutzung des Scade-Werkzeugs diente, ist bedauerlicherweise über die dargelegten Funktionen des Modells hinausgehend keine Anforderungsdefinition existent, so daß sich in Kapitel 7 durchgeführte Beweisaufgaben auf spekulative Eigenschaften des Modells abstützen müssen.

5.2.1 Betrachtung des diskreten Zustandsraums

Eine statistische Angabe zur Anzahl verwendeter Variablen oder gar eine Angabe der Größe des diskreten Zustandsraums fällt auf dieser graphischen Repräsentation schwer, da zwar jede Verbindung zwischen Operatoren prinzipiell der Einführung einer Variablen entspricht, diese aber in vielen Fällen nur propagiert wird und daher nicht immer zur Vergrößerung des Zustandsraums beiträgt. Dennoch können wir mit Hilfe der in Abschnitt 5.1.5 gegebenen Hinweisen das Modell genauer betrachten. Im Bereich der offensichtlichen Zustände wären

- der Zustandsautomat mit drei möglichen Zuständen aufzuführen, sowie
- alle auf diskrete Variablen angewendete *pre*- oder *fby*-Operatoren, welche gemäß der Übersicht in Anhang A.4 die einzigen in Scade vorkommenden expliziten Verzögerungselemente darstellen.

Letztere finden sich z.B. in Abbildung 5.13 vor zwei der drei Eingängen des Zustandsautomaten, in Abbildung 5.7 im *Control*-Operator am Ausgang des *Telemetry-Checking*-Operator sowie in dem in Anhang A.3.1 definierten *COUNTER*-Operator, wobei dieser vom Typ Integer ist und damit besonders zur Zustandsraumvergrößerung beiträgt. Dieser wird später auf den Typ `int` in der Sprache C abgebildet, welche hierfür eine Mindestbreite von 16 Bits vorsieht. Weiterhin in diese Kategorie von Zustandsbits gehören auch die auf Telemetriedaten angewandten Verzögerungsoperatoren in den Abbildungen 5.4 vor den Eingängen des *ExternalConditions*-Operator, da diese Telemetriedaten für jede Koordinate ein Gültigkeitsbit

beinhalten. Da der Zustandsautomat mit seinen drei Zuständen in LUSTRE mit drei boolschen Latches kodiert wird, erhalten wir alle Zustandsbits zusammenzählend damit bislang 28 Zustandsbits.

An „versteckten" Zustandsbits sind jedoch noch weitere hinzu zu zählen. Eine Untersuchung der Zugriffe auf lokale oder globale Variablen im Modell zeigt, daß diese auf reine Propagation beschränkt sind und somit im vorliegenden Fall nicht zur Zustandsraumvergrößerung beitragen. Es wird jedoch ein Zustandsbit zur globalen Unterscheidung des Initialisierungsschrittes von den übrigen Schritten benötigt. Dieses Bit entscheidet, ob in der Datenflußabarbeitung bei den verwendeten *Init*- und *condact*-Operatoren der Inititialisierungswert, oder der reguläre Wert propagiert wird. Weiterhin wurde in Abschnitt 5.1.5 bereits erwähnt, daß durch unterschiedliche Taktungen implizierte bedingte Teilauswertungen des Datenflußgraphen die im bedingt ausgeführten Teil berechneten Ausgängewerte zwischengespeichert werden müssen. In der Sprache SCADE trifft dies auf den *condact*-Operator zu, welchen wir in den Abbildungen 5.4 um den *ExternalConditions*-Operator und den in einem *condact*-Operator eingeschlossenen *GetMvtMode*-Operator in Abbildung 5.5 wiederfinden. Bei ersterem sind somit sechs weitere Gültigkeitsbits betroffen und bei letzterem weitere drei, da das Kommando einen Wertebereich von eins bis sechs hat und somit mit drei Zustandsbits darzustellen ist. Es kommen also weitere 10 Zustandsbits zu den 28 hinzu, so daß das Gesamtmodell 38 Zustandsbits verwendet. Bei böswilliger Verwendung eines 32-Bit Zählers für den Operator *COUNTER* kämen noch weitere 16 Zustandsbits hinzu.

Es ist jedoch anzumerken, daß von den potentiell möglichen 2^{38} Zuständen nicht alle erreichbar sind, da z.B. im Falle des Zustandsautomaten nur drei Kombinationen der möglichen, mit drei Zustandsbits darstellbaren acht Kombinationen möglich sind. Grundsätzlich sind in der Sprache SCADE nicht alle der in Abschnitt 5.1.5 aufgeführten Ursachen für Zustandsbits existent. Dennoch demonstriert bereits dieses Beispiel sehr anschaulich, wie ein großer diskreter Zustandsraum bei einem auf den ersten Blick eher durch kontinuierliche Berechnungen dominierten Modell zustande kommen kann. Neben den Zustandsbits beinhaltet das Modell 10 diskrete Eingabebits für die Telemetriegültigkeitsbits und weiteren, in Abbildung 5.4 aufgeführten Signalen.

Abschließend sei noch erwähnt, daß ähnliche Überlegungen bzgl. der kontinuierlichen Variablen im Modell zu einer Anzahl von 24 zustandsbehafteten Dimensionen führen, welche noch durch 9 Eingabevariablen kontinuierlichen Typs zu ergänzen wären.

5.2.2 Generierter C-Code

Da der für das *Pilot*-System generierte C-Code für die Verifikation verwendet werden soll, werden wir diesen hier nun genauer betrachten. Verwendet wird der Code-Generator *scade_cg* in der Version *V2.3.2p01* mit der Option „`-blockexp`" zur Expandierung aller Funktionsaufrufe, die keine Bibliotheksfunktionen aufrufen. Der Code bekommt dadurch eine sehr einfache, flache Struktur, da die meisten Funktionsaufrufe entsprechend durch den Funktionskörper substituiert sind. Alternativ könnte ebenso ein nicht-expandierter Code generiert und verwendet werden, in dem jedem Operator eine C-Funktion zugeordnet wird, welche analog zu den graphischen Diagrammen jeweils hierarchisch aufgerufen wird. Das Ergebnis ist ein C-Code von insgesamt ca. 1500 Zeilen Umfang[13], der sämtliche verwendeten Variablen in strukturierter Form im globalen Gültigkeitsbereich einschließlich

[13] Zum Vergleich sei erwähnt, daß für industrielle Systeme der Umfang des generierten Codes durchaus bis zu 100000 Zeilen für ein einzelnes eingebettetes System umfassen kann, was insbesondere auch eine entsprechende Skalierung des Umfangs des diskreten Zustandsraums implizieren kann.

ihrer Typen definiert, was bereits über die Hälfte der Zeilen in Anspruch nimmt. Desweiteren existiert auch bei vollständig expandierten Funktionsaufrufen genau eine Funktion namens *Flight*, welche die gesamte, im vorigen Abschnitt vorgestellte Funktionalität umsetzt, d.h. ein Aufruf dieser Funktion führt zu einer vollständigen Abarbeitung aller Datenflußgraphen in einer Datenabhängigkeit-spezifischen Reihenfolge, soweit diese nicht durch *Condact*-Operatoren in Abwesenheit ihres zugeordneten Clock-Signals übergangen werden müssen. Die Funktion *Flight* ist mithin, wie in Abschnitt 5.1.2 definiert, als die *Schritt-Funktion* des Modells zu bezeichnen. Ebenso kann die Code-Generierung sich auf den *Pilot*-Operator beschränken, so daß es genau eine Funktion namens *Pilot* im generierten C-Code gäbe, welche die darunter liegenden Berechnungen einschließlich aller Suboperatoren durchführen und dann entsprechend die *Schritt-Funktion* repräsentieren würde.

Eine Verwendung des generierten C-Codes in einem Steuergerät sähe so aus, daß, motiviert durch die im Sekundentakt aktualisierten Meßwerte der Telemetriedaten, eine ebenso häufige Neuberechnung der Ausgaben des Systems nötig wäre und somit durch eine Form von Hintergrund-Betriebssystem die Funktion *Flight* bzw. *Pilot* periodisch in Abständen von einer Sekunde aufgerufen werden müßte. Die Kommunikation mit der Außenwelt erfolgt dabei über die Eingabe-/Ausgabevariablen, welche ebenfalls im globalen Gültigkeitsbereich definiert wurden. Die Eingabevariablen werden von der *Schritt-Funktion* gelesen und am Ende der durchgeführten Berechnungen die Ergebnisse in die Ausgabevariablen geschrieben.

Die Ausführungszeit der Schritt-Funktion mit den typischerweise anzusetzenden wenigen Millisekunden bei einer C-Funktion diesen Umfangs ist dabei kleiner als die Periode von einer Sekunde, so daß die in der oben erwähnten Synchronen Hypothese gemachten Annahme der instantanen Ausführung hier im Ergebnis zutreffend ist.

5.2.3 Verwendung des Beispiels

Für viele Prozessschritte insbesondere in Kapitel 6 sind einzelne Beispielaspekte darzustellen, die in dieser Fallstudie auch in Teilkomponenten sich nicht in kompakter Form wiederfinden, so daß eine übersichtliche Darstellung schwierig wird. Daher werden wir leider häufig von diesem Beispiel absehen müssen und Aspekt-spezifisch von varierenden, jedoch einfach zu verstehenden Minibeispielen Gebrauch machen.

Kapitel 6

Umsetzung des ω-CEGAR Verfahrens

Die in Kapitel 3 vorgestellten syntaktischen Repräsentationen Hybrider Systeme, sowie den darauf operierenden Algorithmen aus Kapitel 4 dienten der konzeptionellen Beschreibung des ω-CEGAR Verfahrens. Die tatsächlich vorliegende Repräsentation Hybrider Systeme, wie sie beispielsweise mit CASE-Tools oder einer Programmiersprache wie C modelliert werden, ist davon jedoch stark abweichend, wie wir im vorigen Kapitel gesehen haben. Es stellt sich damit die Frage nach einer geeigneten Repräsentationsebene für die Umsetzung des ω-CEGAR Verfahrens. Für große Systeme führt eine wie in Definition 3.2 auf Seite 28 beschriebene Darstellung Schritt-diskreter Systeme bereits für die initiale Abstraktion A_0 zu einem Automaten, der sich ohne symbolische Repräsentation und entsprechenden Techniken nicht mehr model-checken läßt, wie wir bereits in Abschnitt 2.2.2.1 auf Seite 14 bezüglich Kripke-Strukturen diskutiert haben. Darüber hinaus ist die Bildung des Kreuzprodukts mit dem ω-Automaten bei expliziter Darstellung nicht effizient durchführbar, da zum Einen der Umfang der Zustandsmengen der abstrakten Automaten A_i mit $i > 0$ die von A_0 noch weit übertreffen kann und zum Anderen die paarweise Zuordnung der Transitionen gleichen GJ-Paares explizit in derselben Größenordnung durchgeführt werden müsste. Es ist demnach eine abstraktere Repräsentationsebene zu finden, welche eine effiziente Anwendung des Verfahrens gestattet und hernach eine Transformation selbiger in die vom Model-Checker erwartete, symbolische Repräsentation zur Durchführung des in Abschnitt 2.4.2 beschrieben symbolischen Model-Checkens effizient ermöglicht.

Die Wahl traf auf eine einfache Sprache namens SMI, welche sich in dem Umfeld des Autoren im Bereich sicherheitskritischer Systeme bereits etabliert hat und für die bereits eine Reihe wiederverwendbarer Software-Module und -Werkzeuge existieren, was die Umsetzung einiger Verfahrensschritte deutlich vereinfacht hat. Hier ist insbesondere die bereits vorhandene Integration der Sprache SMI mit einem Standard-Model-Checker zu nennen, d.h. eine Transformation von Modellen in das vom Model-Checker erwartete Format war bereits vorhanden.

Auf der anderen Seite hat die Verwendung von SMI manche Verfahrensschritte erschwert oder unmöglich gemacht. Letzteres bezieht sich insbesondere auf die Anwendung des Verfahrens auf Zeit-kontinuierliche Hybride Systeme, welche mangels entsprechender Sprachkonstrukte nicht spezifizierbar sind und damit bedauerlicherweise mit der vorliegenden Implementierung nicht behandelt werden können, obwohl das Verfahren dies grundsätzlich ermöglicht, wie in Abschnitt 4.8 dargelegt. Für den praktischen Einsatz spielt dieser Aspekt jedoch keine Rolle, da im industriellen Bereich keine größeren Zeit-kontinuierlichen Systeme existieren,

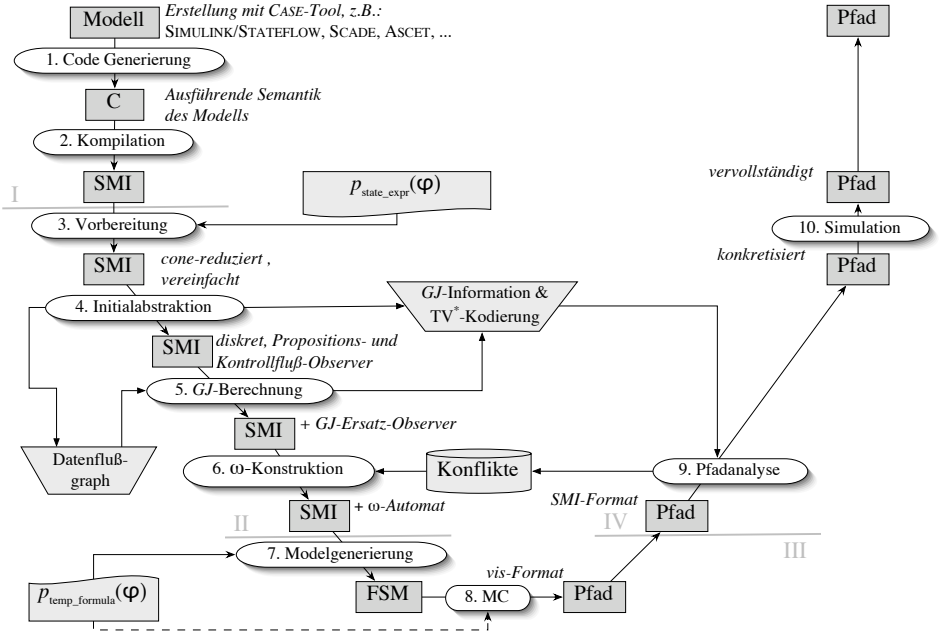


Abbildung 6.1: Übersicht der Verarbeitungsschritte

wie bereits mehrfach diskutiert wurde.

In diesem Kapitel wird die Implementierung des ω-CEGAR Verfahrens auf den konkreten Modellrepräsentationen beschrieben. Dies umfasst der Vollständigkeit halber auch die gesamte schrittweise Transformation des Modells von dem C zu dem SMI-Code ebenso wie die Verwendung des Model-Checkers. Abbildung 6.1 zeigt diesbezüglich einen vereinfachten Überblick über den Ablauf, dessen Schritte in diesem Kapitel näher beschrieben werden. Der Ablauf ist grob in vier Phasen aufgeteilt, welche ihrerseits aus mehreren Ablaufschritten bestehen:

I Transformation des Modells nach SMI (Abschnitt 6.2 auf Seite 120)

II Abstraktion, ω-Automatenkonstruktion und Kreuzprodukt auf SMI Ebene (Abschnitt 6.3 auf Seite 129)

III Modellgenerierung und Aufruf des Model-Checkers (Abschnitt 6.4 auf Seite 143)

IV Analyse des Pfades und Detektion von Konflikten (Abschnitt 6.5 auf Seite 145)

Die in Kapitel 4 geschilderte konzeptionelle Sicht des Verfahrens findet sich dabei in den Phasen 2 und 4 wieder und beinhaltet auch formale Beschreibungen, die aufgrund des großen Unterschieds der Umsetzung des Verfahrens auf der andersartigen Repräsentationsebene angebracht sind. Da die Sprache SMI in allen Schritten eine zentrale Rolle spielt, soll diese zunächst vorgestellt werden.

6.1 Die Sprache SMI

SMI steht abkürzend für *System Modeling Interface* und ist historisch aus dem Kontext der STATEMATE-Verifikation aus den Dissertationen [Bro99] und [Wit99]

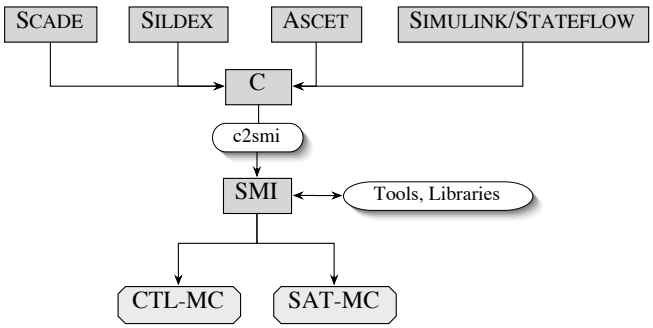


Abbildung 6.2: Rolle der SMI-Sprache

hervorgegangen. Motivation für deren Einführung war hierbei die Suche nach einer einfachen, einem imperativen Paradigma folgenden Verhaltensbeschreibung, welche die semantische Transformation von einem graphisch vorliegenden Verhaltensmodell in STATEMATE in ein Eingabe-Format für den Model-Checker durch einen Zwischenschritt vereinfachen sollte. SMI als Austauschformat bietet die inzwischen viel genutzte Möglichkeit, Modelle anderer Sprachen ebenfalls nach SMI zu übersetzen, um diese mit dem Model-Checker untersuchen zu können. Da SMI als Repräsentationsebene näher an den jeweiligen Modellrepräsentationen liegt als das Eingabeformat des Model-Checkers, gestaltet sich eine solche Integration deutlich einfacher.

Darüber hinaus gibt es für SMI neben der Verwendung alternativer Model-Checking Verfahren auch andere Anwendungsgebiete, wie beispielsweise die automatische Testmustergenerierung oder Sicherheitsanalysen, so daß auch hier eine größere Flexibilität erreicht wird. Aufgrund dieser vielfältigen Möglichkeiten und der zentralen Rolle von SMI sind im Rahmen verschiedener Projekte eine Vielzahl von Optimierungsverfahren und Transformationen für SMI verfügbar gemacht worden, welche die Integration weiterer Anwendungskontexte durch Wiederverwendung der entsprechenden Module z.T. sehr erleichtern. Ein Ausschnitt der SMI-Welt ist in Abbildung 6.2 dargestellt.

6.1.1 Sprachelemente

Eine genaue Definition der Sprache einschließlich der Semantik findet sich in [WBBL02]. Hier sollen nur die wichtigsten Sprachkonstrukte vorgestellt werden.

Die Sprache SMI ist eine sehr einfache imperative Sprache mit lediglich 7 verschiedenen Anweisungen, die sequentiell komponiert werden können:

- **SKIP** ist eine Anweisung ohne jede Auswirkung
- `@[`*target*`$,`*expression*`]` ist die Zuweisungsanweisung, welche den Ausdruck *expression* der Variablen *target*$ zuweist.
- **PAR** $S_1 || \ldots || S_n$ **RAP** ist eine Anweisung zur parallelen Ausführung der Programmblöcke $S_1, \ldots, S_n$
- **DCASE** `[]`c_1`:`S_1 `[]`$\ldots$ `[]`c_n`:`S_n **DESAC** ist die Anweisung zur deterministischen bedingten Ausführung der Programmblöcke von $S_1, \ldots, S_n$ abhängig von den Bedingungen $c_1, \ldots, c_n$. Es wird höchstens ein Programmblock ausgeführt.

- **NDCASE** `[]`c_1`:`S_1 `[]`... `[]`c_n`:`S_n **NDESAC** ist die Anweisung zur nichtdeterministischen bedingten Ausführung der Programmblöcke von $S_1, \ldots, S_n$ abhängig von den Bedingungen $c_1, \ldots, c_n$. Es wird höchstens ein Programmblock ausgeführt.
- **WHILE** c **DO** S **OD** ist die Schleifenanweisung, bei der S solange iterativ ausgeführt wird, wie die Bedingung c erfüllt ist.
- **BREAK** ist die Schleifenabbruchsanweisung

SMI-Programme dienen der Modellierung Reaktiver Systeme und müssen als solche eine Endlosschleife besitzen. Diese wird von allen SMI-basierten Werkzeugen als erste Anweisung des Programms erwartet, welche die gesamte Verhaltensbeschreibung in dem Schleifenkörper besitzt, d.h. ein SMI-Programm ohne Programmkopf[1] sieht folgendermaßen aus:

```
CODE
  DO STEP
    ...
  END STEP
END
```

Schritt des SMI-Programms

Dabei bezeichnen wir das Ausführen der zwischen **DO STEP** und **END STEP** aufgeführten Anweisungen[2] auch als die Ausführung eines *Schritts* des SMI-Programms. Ein solcher Schritt entspricht dem in Abschnitt 5.1.2 diskutierten Begriff der Schritt-Funktion.

$\mathcal{D}_{Cond}, \mathcal{D}_{Int_l^u}, \mathcal{D}_{Real}$

Die relevanten verfügbaren atomaren Datentypen sind: *Condition*, $Integer_l^u$ mit Einschränkung des Wertebereichs durch ein Intervall $[l, u]$ mit $l, u \in \mathbb{Z}$ und *Real*. Die Wertebereiche dieser Typen sind $\mathcal{D}_{Cond} \stackrel{\text{def}}{=} \{false, true\}$, $\mathcal{D}_{Int_l^u} \stackrel{\text{def}}{=} \{i \in \mathbb{Z} | l \leq i \leq u\}$ und $\mathcal{D}_{Real} \stackrel{\text{def}}{=} \mathbb{R}$. SMI besitzt außerdem die kompositionellen Typen *Array* und *Record*, wobei keine Restriktionen bzgl. der Schachtelungstiefe existieren. Es existieren weitere Datentypen in SMI, die jedoch für diese Arbeit nicht von Belang sind. Stattdessen führen wir den abstrakten Typ *Disc* ein, welcher alle Typen diskreten Wertebereichs umfassen soll. Wir bezeichnen die Menge aller Datentypen in SMI mit $\mathcal{T}_{SMI}$.

$\mathcal{T}_{SMI}, \mathcal{M}_{SMI}$

Variablen dieser Typen besitzen zusätzlich einen Modus M aus der Menge der Modi $\mathcal{M}_{SMI} \stackrel{\text{def}}{=} \{input, state, observer, constant\}$, welche wie folgt zu interpretieren sind:

- Eingabemodus (*input*): Variablen dieses Modus werden zu Beginn jeden Schrittes neu von der Umgebung beschrieben.
- Zustandsbehaftete Modi (*state*): Diese Modi werden für Variablen verwendet, welche ihren Zustand über Schrittgrenzen hinweg beibehalten. Es gibt für jede dieser Variablen v implizit auch eine Variable $v^{\$}$, die im Unterschied zu v als *gestrichen (primed)* bezeichnet wird. Demgemäß sprechen wir bei v auch von der *ungestrichenen (unprimed)* Variablen. Ungestrichene Variablen sind nur lesbar und referenzieren den Wert der Variablen am Ende des vorherigen Schrittes. Demgegenüber sind gestrichene Variablen, welche in der

[1] Dem Programm ist immer ein „**MODULE** *name*" als Programmkopf vorangestellt, was wir in allen Listings dieser Arbeit jedoch unterschlagen.

[2] Die äußere Schleife wird in der Syntax von SMI mit **WHILE** *true* **DO** ... **OD** spezifiziert, ist dann jedoch nicht zutreffender Weise verwechselbar mit einer divergenten Schleife, weswegen hier die für illustrative Zwecke auch Verwendung findende Darstellung **DO STEP** ... **END STEP** gewählt wird.

SMI-Syntax durch den Suffix '$$' gekennzeichnet werden, auch beschreibbar und beziehen sich immer auf den aktuellen, d.h. zuletzt geschriebenen Wert der Variable.

- Zustandslose Modi (*observer*): Variablen dieser Modi besitzen kein Zustandsverhalten über Schrittgrenzen hinweg, d.h. zu Beginn jeden Schrittes besitzen diese einen Typ-abhängigen Initialwert. Diese Variablen sind entsprechend nur in *gestrichener* Form existent, besitzen also in Ausdrücken immer den '$$'-Suffix.
- Konstanten (*constant*): Variablen dieses Modus sind nur lesbar.

Typen und Variablen mitsamt ihren Modi und etwaigen Initial-, bzw. Konstanten-Werten für zustandsbehaftete Variablen bzw. Konstanten werden in einer zwingend erforderlichen Symboltabelle definiert, welche in einer separaten Datei das SMI-Programm ergänzt.

Die Menge aller Variablen bezeichnen wir mit V_{SMI}, wobei wir Teilmengen hiervon durch Angabe des Typs und der Modusklasse durch die Notation V_T^M mit V_{SMI} $V_T^M, V_T^{\$M}$

$$V_T^M \stackrel{\text{def}}{=} \{v \in V_{SMI} | v \text{ ist vom Typ } T \in \mathcal{T}_{SMI} \text{ und Modus } M \in \mathcal{M}_{SMI}\}$$

referenzieren werden. Demgegenüber beziehen wir uns mit $V_T^{\$M}$ auf die Menge der gestrichenen Instanzen der Variablen. Modus M oder Typ T können dabei auch weggelassen werden, so daß

- $V_T \stackrel{\text{def}}{=} \bigcup_{M \in \mathcal{M}_{SMI}} V_T^M$ und $V_T, V^M, V_T^{\$}, V^{\$M}$
- $V^M \stackrel{\text{def}}{=} \bigcup_{T \in \mathcal{T}_{SMI}} V_T^M$,

wobei die Mengen $V_T^{\$}$ und $V^{\$M}$ analog dazu definiert sind. V_{SMI} setzt sich aus diesen disjunkten Teilmengen zusammen durch

$$V_{SMI} = \bigcup_{T \in \mathcal{T}_{SMI}} \left(V_T^{state} \cup V_T^{\$state} \cup V_T^{input} \cup V_T^{\$observer} \cup V_T^{constant} \right)$$

Ausdrücke der SMI-Sprache werden mit den einstelligen Operatoren *abs, asl, asr, not, aNot*, +, -, mit den zweistelligen +, -, *, /, *and* , *or, nor max, min*, +, -, =, /=, <, <=, >, >=, *, /, mod, and, or, impl, nand, nor, xor, xnor, aAnd, aOr, aImpl, aNand, aNor, aXor, aXnor*, sowie den dreistelligen *if-then-else* und *mux*[3] gebildet. Für eine bessere Lesbarkeit werden wir für alle SMI-Ausdrücke, sofern vorhanden, die besser lesbaren, mathematischen Zeichen für SMI-Operatoren verwenden, also z.B. '¬' für 'not', usw..

Für einen Ausdruck e definieren wir die Funktion $\mathcal{V}(e)$ als Abbildung auf die Menge der in e vorkommenden Variablen. $\mathcal{V}(e)$

6.1.2 SMI Semantik

Auf eine ausführliche vollständige Darstellung der Semantik von SMI wird an dieser Stelle verzichtet, da diese zum Einen in [WBBL02] nachzulesen ist und zum Anderen für das Verständnis der Umsetzung des Abstraktionsverfeinerungsverfahrens nicht alle Einzelheiten der Sprache SMI benötigt werden. Im vorigen Kapitel ist bei der Einführung der Sprachkonstrukte bereits eine intuitive Beschreibung deren Semantik gegeben worden. Hier soll nur der Teil formalisiert werden, der in diesem Kapitel benötigt wird[4].

[3] *if-then-else* und *mux* können auf semantisch äquivalente **DCASE**-Konstrukte abgebildet werden und werden daher in dieser Arbeit nicht weiter betrachtet. In den verwendeten Kontexten sind sie zudem nicht existent.

[4] Dabei werden gegenüber [WBBL02] abweichende Darstellungen verwendet.

6.1.2.1 Variablenbelegungen und Zuweisungen

σ

Wir führen für alle SMI-Variablen $v \in V_{SMI}$ Belegungen ein, welche jeweils als eine Funktion σ dargestellt werden und jeder Variablen v einen Wert w ihres Typ-abhängigen Wertebereichs $\mathcal{D}$ zuordnen, formal $\sigma : v \mapsto w$. Jede einmalige Ausführung des Schleifenrumpfes S der äußeren Endlos-Schleife überführt eine Belegung σ_i in eine neue Belegung σ_{i+1}. Dabei wird zu Beginn der Ausführungen des Schleifenrumpfes implizit aus der Belegung σ_i die Belegung σ^0_{i+1} mit

$$\sigma^0_{i+1} := \begin{cases} v, v^\$ \mapsto \sigma_i(v^\$) & \text{gdw } v \in V^{state} \\ v \mapsto \text{beliebiges } w, w \in \mathcal{D}_{T_v} & \text{gdw } v \in V^{input} \\ v^\$ \mapsto w, w \in \mathcal{D}_{T_v}, \nexists w' \in \mathcal{D}_{T_v} : w' < w & \text{gdw } v^\$ \in V^{\$observer} \\ v, v^\$ \mapsto \sigma_i(v) & \text{gdw } v \in V^{const} \end{cases}$$

Mit dieser Belegung werden die Variablenzugriffe in der Ausführung des Schleifenrumpfes zu Beginn ausgewertet und diese bei Ausführung der j-ten Zuweisung in einer Schleifenausführung $v^\$_m := e$ mit e als beliebigem Typ-konformen Ausdruck durch eine neue Belegung σ^j_{i+1} ersetzt mit

$$\sigma^j_{i+1} := \sigma[v^\$_m := e] \stackrel{\text{def}}{=} \begin{cases} v \mapsto e & \text{gdw } v = v^\$_m \\ v \mapsto \sigma^{j-1}_{i+1}(v) & \text{gdw } v \neq v^\$_m \end{cases}$$

Am Ende der Ausführung des Schleifenrumpfes ist nach Ausführung von k Zuweisungen die Belegung $\sigma_{i+1} = \sigma^k_{i+1}$ die bei der nächsten Operation zu verwendende. Die Anzahl k ist dabei abhängig vom jeweilig ausgeführten Kontrollflußpfad, welcher wiederum durch die Anfangsbelegung σ^0_{i+1} determiniert ist. Somit definiert die Ausführung des Schleifenrumpfes die Schritt-Funktion $\mathfrak{S} : S \mapsto (\sigma_i \mapsto \sigma_{i+1})$ und ein Lauf des SMI-Programms ergibt eine Folge von Belegungen $(\sigma_0, \sigma_1, \sigma_2, \dots)$ mit

Schritt-Funktion $\mathfrak{S}$

$$\sigma_0 \overset{\mathfrak{S}(S)}{\mapsto} \sigma_1 \overset{\mathfrak{S}(S)}{\mapsto} \sigma_2 \overset{\mathfrak{S}(S)}{\mapsto} \dots$$

mit σ_0 als Initialbelegung durch die Symboltabelle vorgegeben.

Wir definieren die Notation $\sigma \downarrow_V$ zur Bezugnahme auf eine Teilbelegung, welche im Gegensatz zu σ nur eine Belegung für Variablen aus $V \subseteq V_{SMI}$ definiert.

Teilmengen von Belegungen werden u.a. später für Analogien zu dem in Kapitel 3 und 4 vorgestellten Datenstrukturen herangezogen werden. Während im Hybriden Automaten der kontinuierliche Zustandsraum $X \subseteq \mathbb{R}^n$ durch n geordnete Dimensionsgrößen beschrieben worden ist, haben wir hier anstelle der Ordnung explizite Bezeichner, die Variablennamen. Da sich durch entsprechende Benennung der Dimensionen bzw. durch Definition einer Ordnung auf V_{Real} die Darstellungen ineinander überführen lassen, werden wir die formal benötigten Transformationen der einfacheren Darstellung halber als implizit gegeben annehmen.

6.1.2.2 Kontrollfluß

Der Kontrollfluß eines SMI-Schritts ergibt sich aus der sequentiellen Ausführung der Anweisungen eines SMI-Programmblocks. Dieser verzweigt sich bei der Schleifenanweisung **WHILE** und den Fallunterscheidungsanweisungen **DCASE** und **NDCASE**. Die Schleifenanweisung ist, sofern die Anzahl der Schleifendurchläufe endlich beschränkt ist, semantisch äquivalent zu einer endlichen Verschachtelung von **DCASE**-Anweisungen, wie in Abbildung 6.3 skizziert. Für diese können wir daher unsere Ausführungen an dieser Stelle auf die Fallunterscheidungsanweisungen beschränken[5].

[5] Die Anweisung **PAR** wäre der Vollständigkeit halber ebenfalls zu begutachten, kann jedoch in den C-basierten Fallbeispielen nicht auftreten und soll daher an dieser Stelle übergangen werden.

```
WHILE c$ DO
    S
OD
```

```
DCASE
    [] c$ :
        S
    DCASE
        [] c$ :
            S
            ...
            DCASE
                [] c$ :
                    S
                DESAC
            ...
    DESAC
DESAC
```

Abbildung 6.3: Semantisches Äquivalent zu **WHILE**-Anweisungen mit Schleifenbedingung $c^{\$}$ und Anweisungsblock S

```
CODE
S1
DCASE
    [] c1:
        S2
        DCASE
            [] c2:
                S3
            [] c3 :
                S4
        DESAC
        S5
DESAC
S6
END
```

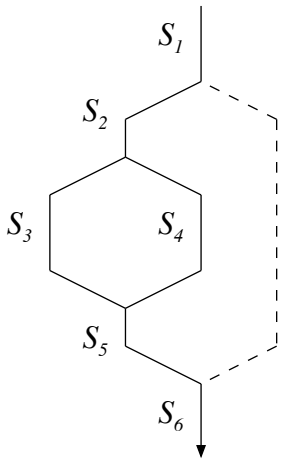


Abbildung 6.4: Kontrollflußdiagramm zu einem SMI-Programm, wobei $c_2 = \neg c_3$ gelten soll. Der gestrichelte Pfad ist implizit vorhanden, wann immer möglicherweise keine der zur Disposition stehenden Bedingungen erfüllt ist. In diesem Beispiel gilt dies in der Situation $\neg c_1$

Abhängig von dem Wahrheitswert der Auswertung der jeweils zugeordneten Vorbedingung c_i wird bei einer Fallunterscheidungsanweisung maximal einer der Blöcke S_i ausgewählt und der Kontrollfluß an diesen weitergegeben. Nach dessen Bearbeitung wird der Kontrollfluß wieder in den der Verzweigungsanweisung folgenden Anweisungen fortgeführt. Der Kontrollfluß kann somit eindeutig durch eine Folge von Anweisungen beschrieben werden.

Die verschiedenen möglichen Kontrollflüsse lassen sich in Kontrollflußdiagrammen veranschaulichen, welche Verzweigungen und Vereinigungen des Kontrollflusses beinhalten. In Abbildung 6.4 ist ein SMI-Programm und dessen Kontrollflußdiagramm schematisch dargestellt. Das gestrichelte Teilstück entspricht hier einem impliziten Kontrollfluß, in dem Bedingung c_1 nicht gilt. In Anlehnung an diese Diagrammdarstellung, in der der Kontrollfluß durch einen Pfad veranschaulicht werden kann, bezeichnen wir den Kontrollfluß auch als Kontrollflußpfad.

6.1.2.3 Ausdrücke diskreter Größen

An Ausdrücken diskreter Größen werden wir im Rahmen der Automatenkonstruktionen und Kommunikationsfunktionen derselben lediglich die Operatoren $=, \neq, \wedge, \vee, \neg$ verwenden, welche semantisch den gleich-bezeichneten üblichen logischen Operatoren entsprechen. Wir verzichten daher an dieser Stelle auf eine eingehende Vorstellung der Semantik dieser und der übrigen Operatoren mit dem Verweis auf [WBBL02].

6.1.2.4 Ausdrücke kontinuierlicher Größen

Bezüglich der Ausdrücke in SMI ist für die ω-CEGAR Implementierung nur die Semantik der Operatoren relevant, die auf kontinuierlichen Ausdrücken angewendet werden können. Die zweistelligen Operatoren $+, -, *$ und / sind in Infixnotation die Abbildungen $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, die semantisch den gleichnamigen vier Grundrechenarten entsprechen. In Ermangelung der Festlegung von konkreten Repräsentationen und etwaigen eingeschränkten Genauigkeiten in der Darstellung von Werten aus $\mathbb{R}$ soll an dieser Stelle darauf verwiesen werden, daß hier die C-Semantik übernommen worden ist[6]. Gleiches gilt für die zweistelligen Vergleichsoperatoren $<, \leq, >, \geq, =, \neq$, welche, ebenfalls in Infixnotation verwendet, die gleiche Semantik besitzen, wie die gleichnamigen Funktionen $\mathbb{R} \times \mathbb{R} \to \mathbb{B}$ aus C. Die einstelligen Operatoren $+ : \mathbb{R} \to \mathbb{R}$ und $- : \mathbb{R} \to \mathbb{R}$ lassen sich auf die zweistelligen abbilden mit $+e \stackrel{\text{def}}{=} 0 + e$ und $-e \stackrel{\text{def}}{=} 0 - e$.

6.1.3 Vergleich zum Schritt-diskreten Hybriden Automaten

Die in SMI spezifizierbaren Modelle weisen aufgrund des imperativen Paradigmas eine wesentlich andere syntaktische Struktur als ein Schritt-diskreter Hybrider Automat nach Definition 9 auf Seite 29 auf. Diese sind für die Implementierung des in Kapitel 4 vorgestellten Verfahrens von zentraler Bedeutung und sollen daher hier gegenüber gestellt werden. Tabelle 6.1 gibt einen einen Überblick der verschiedenen relevanten Aspekte.

6.2 Vom Modell zum SMI-Programm

Wie in Abbildung 6.2 dargestellt gibt es eine Anzahl von CASE-Tool-Sprachen, von denen eine Übersetzung nach SMI über den von diesen Werkzeugen generierten C-Codes existiert. Primär dient dieser Code der Ausführung auf einer Zielplattform eines eingebetteten Systems, für die das Modell entworfen wurde. In vielen Fällen wird der C-Code sogar von den Simulationswerkzeugen der CASE-Tools herangezogen, um das Modell selbst zu simulieren. Der C-Code definiert in solchen Fällen die Simulationssemantik der Modelle und ist daher eine weit zuverlässigere Verhaltensbeschreibung, als in Datenstrukturen verborgene piktographische Informationen, wie sie bei einer nicht-C-basierten Integration zu übersetzen wären.

Daneben ist der Vorteil einer einfacheren Übersetzung von C nach SMI aufgrund des gemeinsamen imperativen Paradigmas dieser Sprachen gegeben, trotz der vielfältigen Sprachkonstrukte in C, zu denen es kein direktes Äquivalent in SMI gibt. Weiterhin generieren nicht nur praktisch alle CASE-Tools die Sprache C, sondern dies ist auch die Sprache, in der häufig manuell erstellter Code vorliegt,

[6] Neben und vor dieser Arbeit existiert(e) nur ein weiterer Anwendungsfall, der diese Ausdrücke interpretieren muß. Dies ist die Simulation des SMI Programms, welche, manifestiert in einer entsprechenden Simulationsfunktion, sich der C-Semantik bedient.

	Schritt-diskreter Hybrider Automat	SMI
Eingabe	Implizit vorhanden durch Nicht-Determinismus an Transitionen, bzw. Jump Set Funktionen mit freien Dimensionen	Explizit vorhanden durch Eingabe-Variablen aller Typen
Nicht-determi-nismus	Vorhanden durch nicht-deterministische Transitionen	Wird realisiert durch **NDCASE** -Fallunterscheidungen oder zusätzlichen Inputs
Ausgabe	Implizit durch diskreten und kontinuierlichen Zustand (z, x), ähnlich einem Moore-Automat.	Explizit durch Ausgabe-Variablen, welche nicht zustandsbehaftet sein müssen, ähnlich einem Mealy-Automat
Zustand	Zustände sind eindeutig durch diskreten Zustand kombiniert mit kontinuierlichem Zustand beschrieben	Zustand ist repräsentiert durch Variablenbelegungen zustandsbehafteter Variablen, diskreter und kontinuierlicher Zustand durch jeweils diskrete und kontinuierliche Variablen.
Transi-tionen	Explizit vorhanden	Implizit durch Kontrollfluß-gesteuerte sequentielle Berechnungen auf Variablen. Hierbei sind auch Hilfsvariablen relevant.
Guard Sets	Den Transitionen zugeordnete Teilmengen des kontinuierlichen Zustandsraums	Boolsche Bedingungen durch die Vergleichsoperatoren $=, \neq, \leq, \geq, <, >$ auf Ausdrücke kontinuierlichen Typs. Die Ausdrücke sind im Kontext vorangegangener sequentieller Berechnungen einschließlich Hilfsvariablenbelegungen auszuwerten. Dabei werden nicht notwendigerweise alle Ausdrücke ausgewertet, bedingt durch den Kontrollfluß.
Jump Set Funktio-nen	Den Transitionen zugeordnete Funktionen $X \to 2^X$	Menge aller sequentiellen Berechnungen für kontinuierliche zustandsbehaftete Variablen, abhängig von Kontrollfluß und Hilfsvariablenbelegungen. Da die verfügbaren Operationen deterministisch sind, entspricht dies einer Abbildung $X \to X$

Tabelle 6.1: Vergleich SMI mit Schritt-diskretem Hybriden Automaten

so daß mit einem entsprechenden C-nach-SMI Compiler gleich mehrere Modellierungssprachen abgedeckt werden.

6.2.1 Code Generierung

Wie bereits in den Abschnitten 5.1.3 und 5.2.2 geschildert, wird aus den mit CASE-Tools erstellten Modellen mit Hilfe von C-Codegeneratoren C-Code erzeugt, der insbesondere eine Funktion beinhaltet, der der Schritt-Funktion des Modells entspricht. Genau diese ist, eingebettet in die obligatorische Endlosschleife für reaktive Systeme, Gegenstand der Übersetzung nach SMI.

6.2.2 C-Compiler

Begonnen in dem SafeAir-Projekt [BDL+01, BDL+02] und in verschiedenen Projekten [DSS+03, GGB+03] weiterentwickelt wurde ein entsprechender Compiler (*c2smi*), welcher die Übersetzung von C nach SMI durchführt. Dabei werden alle von den Code-Generatoren benutzten C-Sprachkonstrukte unterstützt, was insbesondere auch dynamische Pointerarithmetik beinhaltet.

Aufgrund der zentralen Funktion dieses Compilers wollen wir auf den generierten SMI Code an dieser Stelle genauer eingehen.

6.2.2.1 Übersetzung der C-Sprachelemente

Während für einfache Konstrukte der C-Sprache, wie `if-then-else`-Anweisungen, Zuweisungen, Schleifenkonstruktionen und große Teile der Ausdruckssprache ein semantisches Äquivalent in SMI existiert, besitzt C darüber hinaus viele High-Level Konstrukte, für die es in SMI kein Gegenstück gibt. Diese sind

- Funktionen
- Sprungbefehle (`goto`, `break`, `continue`, `case`, `return`)
- Zeiger und Zeigerarithmetik, dynamische Speicherverwaltung
- Casts

Da es kaum C-Programme ohne die Mehrzahl dieser Elemente gibt, mußte eine entsprechende Ersatzkonstruktion aller üblichen Verwendungen dieser in SMI gefunden werden:

Übersetzung von Funktionsaufrufen Funktionsaufrufe werden durch die Technik des *Inlining* nach SMI übersetzt, d.h. wo immer ein Funktionsaufruf im SMI-Programm stehen müsste, wird der gesamte Funktionsrumpf eingesetzt. Dabei werden etwaige Parameterzuweisungen zu Beginn ergänzt und die Eindeutigkeit aller Bezeichner unter Beibehaltung der Nachverfolgbarkeit durch Umbenennen aller Variablen sichergestellt.

Eine Inlining-Technik verbietet naturgemäß die Unterstützung von unbeschränkter Rekursion, mit anzugebender maximaler Rekursionstiefe ist durch einen entsprechend oft instantiierten Funktionskörper jedoch eine beschränkte Rekursion möglich. Für umfangreichere Funktionen mit mehreren potentiellen Stellen im Funktionskörper, an denen rekursive Aufrufe durchgeführt werden können, ist das Limit aufgrund der exponentiell wachsenden Programmgröße jedoch schnell erreicht.

```
int x;
int cond;
void f() {
  if(cond) {
    x = 1;
  }
  else {
    x = 2;
    return;
  }
  x = 3;
}

void main() {
  while(1) {
    f();
  }
}
```

```
CODE
 WHILE 1 DO
   DCASE
     [] cond:
       @[x$, 1]
       @[x$, 3]
     [] ¬(cond):
       @[x$, 2]
   DESAC
 OD
END
```

Abbildung 6.5: Beispiel für Übersetzung von `Return`-Anweisungen

Übersetzung von Sprungbefehlen Sprungbefehle fehlen in SMI gänzlich und somit muß jeder Sprung im Kontrollfluß des Programms durch bedingte Ausführung dazwischenliegender Anweisungen übersetzt werden. Bei jedem bedingten Sprung wird der Kontrollfluß verzweigt und für beide Zweige ein entsprechender SMI-Codeblock bis zu der Stelle generiert, bei der der Kontrollfluß wieder zusammenläuft. Abbildung 6.5 zeigt ein einfaches Beispiel der Übersetzung einer `Return`-Anweisung. Die übrigen Sprungbefehle[7] werden nach dem gleichen Prinzip übersetzt.

Übersetzung von Zeigerarithmetik und Adressmengenberechnung Zeiger werden als Variablen vom Datentyp *Integer* übersetzt. Jedem C-Objekt, d.h. Variablen, Elementen von Arrayvariablen und Komponenten von Strukturvariablen, wird in SMI eine Adresse zugeordnet, die für auf dem Heap befindliche Objekte global eindeutig ist und für lokale Variablen mit begrenzter Lebensdauer nur innerhalb deren Lebensdauer.

Aufgrund der strengen, statischen Typisierung in SMI sind Dereferenzierungen von Zeigern zwecks Lese- oder Schreiboperation auf dazugehörige Objekte nicht auf eine umfassende Array-Struktur abbildbar, wie dies üblicherweise von C-Compilern bzgl. einer dynamisch typisierten Speicherarchitektur umgesetzt wird. Daher werden Dereferenzierungen auf umfangreiche **DCASE**-Unterscheidungen in SMI abgebildet, bei der jedes potentiell mögliche Typ-konforme Objekt abhängig von dem Adresswert angesprochen wird.

Bei naiver Betrachtung der Addressübersetzung ohne weitere Analysen muß davon ausgegangen werden, daß bei jeder Dereferenzierung jedes Typ-konforme Objekt referenziert werden könnte, und daher präventiv bei jedem Zugriff eine Fallunterscheidung bzgl. aller in Frage kommenden Variablen eingeführt werden, was nicht nur die Größe des SMI-Programms quadratisch wachsen ließe, sondern auch die Komplexität der Verhaltensbeschreibung erheblich erhöhte.

Aus diesem Grunde wird durch ein Fixpunkt-Analyseverfahren die Menge der möglichen Addressinhalte jeder Zeigervariablen zu den jeweiligen Programmausführungszeitpunkten überapproximierend berechnet. Die Durchführung erfolgt mittels abstrakter Simulation, kombiniert mit der Einführung von strikt getrennten Segmenten, welche jede Zuweisung und Berechnung von Adressen bis auf die folgenden Abstraktionen nachvollzieht:

[7] Die Sprungbefehle `goto` und `continue` sind zum Zeitpunkt des Verfassens dieser Arbeit noch nicht unterstützt, da keiner der Codegeneratoren diese verwendet.

```
int a,c[3];
int *x = &a;
int *y = &c[1];
int **z = &y;

int cond;

void step_func() {
  if( cond) {
        *x = 1;
        **z = 2;
  }
  (*z)++;
}

void main() {
  step_func();
}
```

```
CODE
 DCASE
   [] cond:
     @[a$ , 1]
     @[c$[1], 2]
 DESAC
 @[y$ , y+1]
END
```

Abbildung 6.6: Berechnung potentieller Adressmengen von Zeiger-Objekten (1)

```
int a,c[3];
int *x = &a;
int *y = &c[1];
int **z = &y;

int cond;
void step_func() {
  if( cond) {
        *x = 1;
        **z = 2;
  }
  (*z)++;
}

void main() {
  while(1) {
    step_func();
  }
}
```

```
CODE
 WHILE 1 DO
   @[oob_error_c$ , false]
   DCASE
     [] cond:
       @[a$ , 1]
       @[l3$ , 2]
       DCASE
         [] y=7:
           @[c$[1], l3$]
         [] y=8:
           @[c$[2], l3$]
         [] ¬((y≥6)∧(y≤8)):
           @[oob_error_c$ , true]
       DESAC
   DESAC
   @[y$ , y+1]
 OD
END
```

Abbildung 6.7: Berechnung potentieller Adressmengen von Zeiger-Objekten (2)

- Bei Wiederzusammenführung von Kontrollflußverzweigungen werden die Adressmengen jeder Variablen aller Verzweigungen durch Vereinigung berechnet. Dabei wird nicht untersucht, ob einzelne Verzweigungen überhaupt zur Ausführung kommen konnten.
- Bei Addition von Adress-Offsets dynamischer Größe zu einer Adresse wird konservativ davon ausgegangen, daß das Ergebnis hernach die Adresse jeden Typ-konformen Objekts innerhalb desselben Segments beinhalten kann.

Das Fixpunkt-Analyseverfahren terminiert, sobald sich die Adressmenge keiner Variablen mehr in zwei aufeinander folgenden Iterationen ändert. Aufgrund der oben beschriebenen Abstraktionen in der Simulation ist dies im Allgemeinen bereits nach vergleichsweise wenigen Iterationen der Fall. Im schlimmsten Fall ist die Anzahl der benötigten Iterationen so hoch wie die größte Anzahl aller Variablen eines Typs $n = max(\{k_T | k_T = |Var_T|\})$, wobei Var_T die Menge aller C-Objekte des Typs T repräsentiert.

Zur Veranschaulichung des Ergebnisses dieser Berechnung ist in Abbildung 6.6 die SMI-Übersetzung von einem einfachen Beispielprogramm in C dargestellt. Hier können die Zeigerinhalte noch statisch eindeutig berechnet werden, daher sind keine Fallunterscheidungen nötig. Anders verhält es sich bereits in Abbildung 6.7, wo die für reaktive Systeme zwingende äußere nicht-terminierende Schleife ergänzt

```
int a,c[3];
int *x = &a;
int *y = &c[1];
int **z = &y;

int cond, cond2;

void step_func() {
  if( cond) {
        *x = 1;
        **z = 2;
  }
  (*z)++;
  if( cond2) {
        z=&x;
  }
}

void main() {
  while(1) {
    step_func();
  }
}
```

```
CODE
 WHILE 1 DO
   @[seg_error$, false]
   DCASE
     [] cond:
       @[l3$, 1]
       DCASE
         [] x=5:
           @[a$, l3$]
         [] ¬(x=5):
           @[seg_error$, true]
       DESAC
       DCASE
         [] z=10:
           @[l4$, y]
         [] z=9:
           @[l4$, x]
       DESAC
       @[l5$, 2]
       DCASE
         [] l4$=7:
           @[c$[1], l5$]
         [] l4$=5:
           @[a$, l5$]
         [] l4$=8:
           @[c$[2], l5$]
         [] ¬(l4$=5)∧¬((l4$≥6)∧(l4$≤8)):
           @[seg_error$, true]
       DESAC
   DESAC
   DCASE
     [] z=10:
       @[l6$, y]
     [] z=9:
       @[l6$, x]
   DESAC
   @[l7$, l6$+1]
   DCASE
     [] z=10:
       @[y$, l7$]
     [] z=9:
       @[x$, l7$]
   DESAC
   DCASE
     [] cond2:
       @[z$, 9]
   DESAC
 OD
END
```

Abbildung 6.8: Berechnung potentieller Adressmengen von Zeiger-Objekten (3)

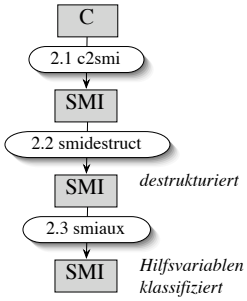


Abbildung 6.9: Verfeinerte Sicht auf die Schritte der Kompilation

ist. Die Zuweisung '**z=2' kann nicht mehr statisch eindeutig bestimmt werden und eine entsprechende Fallunterscheidung in SMI wird erforderlich, wie in den Zeilen 10 bis 17 zu sehen ist. Da der Wert von z sich nicht ändern kann, ist der generierte SMI Code insgesamt dennoch vergleichsweise übersichtlich. Dies ändert sich, wenn, wie in Abbildung 6.8 dargestellt, die Variable z bedingt auf die Adresse von x gesetzt werden kann. Dies hat insbesondere auch Einfluss auf die möglichen Adressbelegungen von x, so daß für die Zeile 11 im C-Code nun alternativ die Möglichkeit einer *Segmentverletzung* in Betracht gezogen werden muß.

Casts Die starke, statische Typisierung in SMI erzwingt eine strikte Einhaltung der Grenzen zwischen nicht kompatiblen Typen. Kompatibel sind in dieser Hinsicht lediglich die ganzzahligen Datentypen unterschiedlichen Wertebereichs innerhalb des kleinsten gemeinsamen Wertebereichs, wobei hierzu als Sonderfall auch Aufzählungstypen und Bedingungen gehören. Castings in C können daher nur im Rahmen explizit modellierbarer, semantisch äquivalenter Transformationen in SMI nachempfunden werden. Unterstützt wird dies bislang nur für Integer-Casts aller Art und Casts zwischen Fließkommazahlen und Integerwerten. Letzteres ist ausführlicher in Abschnitt 6.6 beschrieben.

6.2.2.2 Nachbearbeitung des SMI-Codes

Im vorangegangenen Abschnitt wurde die SMI-Code Generierung aus einem C-Programm skizziert. Dieses SMI-Programm ist jedoch für das CTL-Model-Checking in dieser Form noch ungeeignet, weil die große Anzahl an Variablen zu einem unnötig großen Zustandsraum führt. Der vorgegebene Modus aller Variablen mit persistenter Lebensdauer des Pendant im C-Programm ist zunächst bei konservativer Betrachtung als zustandsbehaftet anzunehmen, da ein Lesezugriff auf denselben unter Umständen die Belegung der Variablen im vorangegangenen Schritt referenzieren könnte.

Für viele dieser Variablen kann dies jedoch aus strukturellen Gründen mit Hilfe einer statischen Analyse ausgeschlossen und die entsprechenden SMI-Variablen mithin als zustandslos definiert werden. Der Model-Checker kann so später bei der Kodierung des Zustandsraums von diesen Variablen absehen.

Für diese Analyse[8] wurde ein eigenes Werkzeug namens *smiaux* implementiert, welches für alle Variablen alle Kontrollflußpfade des SMI Programms untersucht und feststellt, ob eine Variable vor dem jeweils ersten Lesezugriff innerhalb eines Schrittes nie oder möglicherweise schon beschrieben worden sein könnte

[8]Diese Analyse entspricht in algorithmischer Form der von uns in Abschnitt 5.2.1 bzgl. des Fallbeispiels durchgeführten Betrachtung der benötigten Anzahl der Zustandsbits.

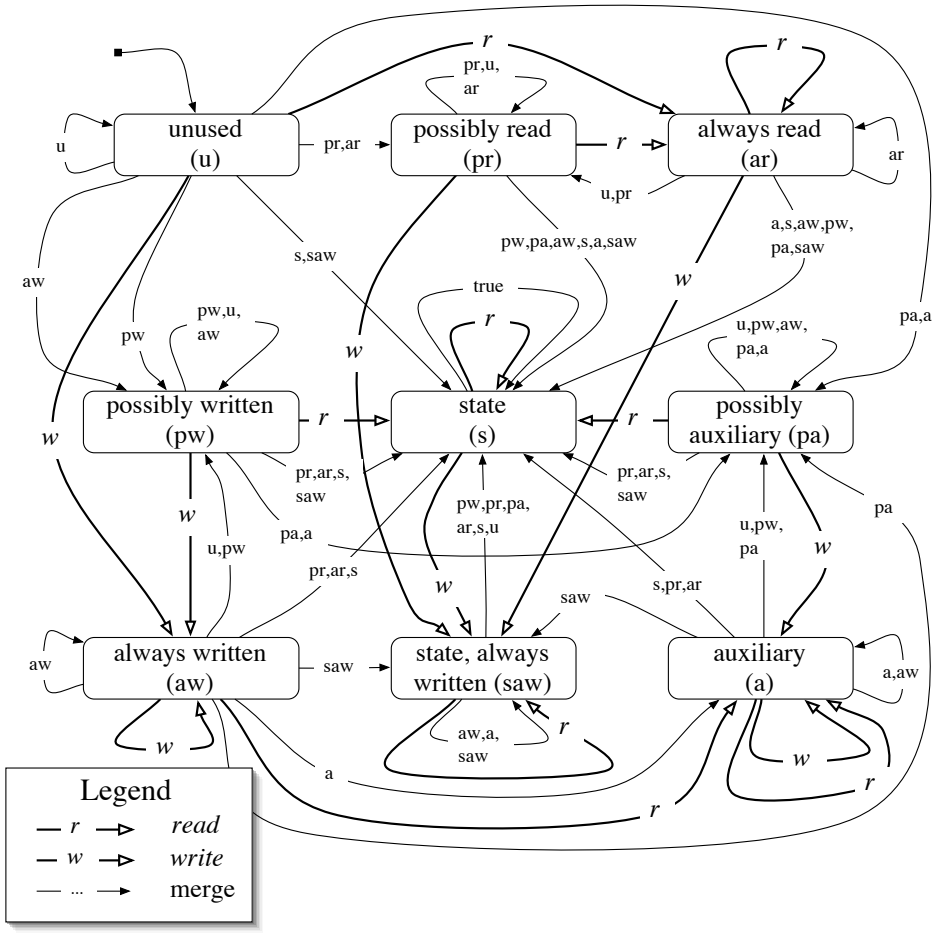


Abbildung 6.10: Zustandsdiagramm zur Berechnung des Modus für Variablen. Die Transitionsbeschriftungen für Fusionsoperationen (*merges*) sind Abkürzungen für den Zustand der gleichen Variablen eines anderen Kontrollflußpfads im Fusionspunkt

(zustandsbehaftet), oder ob diese immer schon beschrieben worden sein muß (zustandslos). Diese Information muß bei wieder zusammenlaufenden Kontrollflüssen (*Mergepoints*) entsprechend fusioniert werden (*merging*).

Der Algorithmus verwaltet für jede Variable $v \in V$ einen Zustand $s_v \in S_V$, welcher sich in Abhängigkeit von Variablenzugriffen entlang der Pfade und Fusionsoperationen in den Mergepoints gemäß der Funktionen $newstate_{access} : S_V \times \{read, write\} \rightarrow S_V$ und $newstate_{merge} : S_V \times S_V \rightarrow S_V$ gemäß des Zustandsdiagramms in Abbildung 6.10 ändern kann. Nach Abschluss der Analyse ist eine Variable in den Zuständen *unused*, *possibly written*, *always written*, *possibly auxiliary* und *auxiliary* als zustandslos zu behandeln.

Ein Beispiel dieser Berechnung ist in Abbildung 6.11 bzgl. des Kontrollflußdiagramms eines einfachen SMI-Programms dargestellt. Die Abbildung zeigt die Zustandswechsel des Zustands jeder Variablen im Verlauf der Berechnung, welche dem chronologischen Ablauf jeden Kontrollpfadsegments folgend den Zugriffen auf Variablen und Fusionen der Kontrollflußpfade durch Anwendung obiger Funktionen Rechnung trägt. Das Ergebnis der Berechnung führt zur Klassifikation der Variablen 'b' als zustandslose Hilfsvariable.

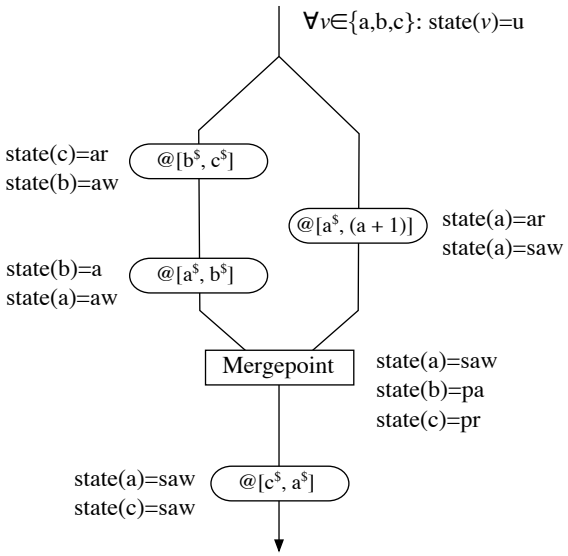


Abbildung 6.11: Beispiel für die Berechnung des Modus von Variablen anhand des Kontrollflusses. Der Zustand s_v einer Variablen v ist in der Abbildung durch die Funktion *state* gegeben

Über die Klassifikation hinausgehend wird im Rahmen obiger Berechnung für alle lesenden Variablenzugriffe auf eine Variable $v^\$$, deren Zustand zum chronologischen Zeitpunkt des Zugriffs[9] gemäß Kontrollflußpfad *unused*, *possibly written* oder *always written ist,* der Variablenzugriff durch das ungestrichene v ersetzt, da es sich in diesen Fällen mit absoluter Sicherheit um den Wert des vorherigen SMI-Schrittes handeln muß. Diese explizite Unterscheidung wird spätere Betrachtungen vereinfachen.

Destrukturierung In SMI gibt es eine strukturelle Hürde bei der Klassifikation von zustandslosen Variablen: Variablen strukturierten Typs können nur als Ganzes entweder als zustandslos oder zustandsbehaftet definiert sein, Komponentenspezifische Mischformen sind hingegen nicht möglich. Da dies in einigen Kontextes ein enormes Optimierungspotential ausschließt, ist vor der Analyse zur Berechnung zustandsloser Variablen eine Destrukturierung des SMI-Codes notwendig.

Zu diesem Zwecke ist das Tool *smidestruct* implementiert worden, welches jedes Datenobjekt d der Form 'a.b' bzw. 'c[k]' mit 'k' als Konstante aus $\mathbb{N}_0$ auf eine unstrukturierte Variable v_d nachvollziehbaren Bezeichners bijektiv abbildet. Zugriffe auf d werden jeweils durch v_d substituiert. Ausdrücke der Form 'c[< *beliebigerAusdruck* >]' kann es in dem durch *c2smi* erzeugten SMI-Code nicht geben, da diese im Rahmen der Dereferenzierung von Zeigern immer in explizite Fallunterscheidungen mit konstanten Feldzugriffen aufgeschlüsselt werden.

In Abbildung 6.9 sind noch einmal zusammenfassend die Schritte der Kompilierung aufgeschlüsselt.

[9] Zu unterscheiden vom Zustand am Ende der Analyse

6.3 Abstraktionsverfeinerung auf SMI-Ebene

Nachdem die Herkunft der in SMI vorliegenden Verhaltensbeschreibung geschildert wurde, wird im Folgenden die Umsetzung der Abstraktion und Abstraktionsverfeinerung des in Kapitel 4 konzeptionell beschriebenen Verfahrens auf der SMI-Ebene beschrieben. Insbesondere aufgrund der sequentiellen Ausführung von Anweisungen gegenüber den atomaren Transformationen des Schritt-diskreten Hybriden Automaten sind deutlich mehr Transformationsschritte nötig, als auf konzeptioneller Ebene dargestellt, wie in Abbildung 6.1 bereits angedeutet wurde.

Nach einigen Transformationen, die das SMI-Programm im Rahmen einer Vorverarbeitung durchlaufen muß, wird die Abstraktion des SMI-Programms geschildert. Wichtiger Punkt ist dabei die Projizierbarkeit des Pfades des abstrakten SMI-Programms auf die Berechnungssequenz des originalen SMI Programms analog zu Abschnitt 4.3.2.1 auf Seite 40. Insbesondere wird in Abschnitt 6.3.3 auf eine Erweiterung zur Steigerung der Effizienz der Projektion sowie deren Injektivität eingegangen. Zuletzt wird in diesem Abschnitt die Konstruktion des ω-Automaten einschließlich dessen Kombination mit dem abstrakten SMI-Programm erläutert.

6.3.1 Vorbearbeitung des SMI Programms

Das nach der Kompilation vorliegende SMI Programm beinhaltet auch nach den bereits vorgestellten Nachbearbeitungen unter Umständen die folgenden Eigenschaften:

- Das Modell besteht möglicherweise aus Teilen, welche aus strukturellen Gründen keinen Einfluß auf die zu verifizierende Eigenschaft haben können und unnötig die Komplexität sowohl für die Abstraktion, als auch für die spätere Anwendung des Model-Check-Algorithmus erhöhen.
- Das Modell kann Schleifen beinhalten, welche zwar durchaus später während der Modellgenerierung durch das Werkzeug *smi2blifmv* z.T. behandelt werden können, die zu verwendende Abstraktionstechnik kann mit Schleifen jedoch nicht umgehen.
- Variablen können auf einem Kontrollflußpfad mehrfach mit neuen Werten beschrieben werden, was eine Berechnung des Datenflußgraphen erheblich verkompliziert.
- Die Eigenschaft kann sich auf Aussagen zu kontinuierlichen Größen abstützen, welche vor der Abstraktion zunächst in explizite diskrete Zustände übertragen werden müssen.

Bevor mit der eigentlichen Abstraktion begonnen wird, wird ein weiterer, vorbereitender Transformationsschritt vorgenommen, in dem das SMI-Programm entsprechenden Vereinfachungen und Transformationen unterzogen wird. Dieser Schritt ist als Schritt 3 der Abbildung 6.1 auf Seite 114 dargestellt.

6.3.1.1 Cone-of-Influence-Reduktion

Das Tool *smicoi* führt die Analyse zur Bestimmung der für die Property relevanten Teile des Modells (*cone-of-influence*) auf syntaktischer Ebene durch. Dabei werden beginnend von der nachzuweisenden Eigenschaft φ transitiv strukturell mögliche Beeinflussungen der Variablen soweit zurückverfolgt und in einer Menge VC iterativ aufgesammelt, bis keine weiteren Variablen mehr hinzugefügt werden können, mithin also ein Fixpunkt erreicht ist. Im Anschluß an diese Analyse können alle

Zuweisungen zu Variablen v aus dem SMI-Programm entfernt werden, die nicht in VC sind. Genauere Einzelheiten zu diesem Verfahren sind in [Bie03] nachzulesen.

Eine solche Optimierung findet sich typischerweise ebenfalls in dem Model-Checker selbst [RGA+96, Gro96a, Gro96b], jedoch ist es für die Abstraktion von Vorteil, wenn bereits auf dieser Ebene unnötige Modellteile entfernt sind, zumal diese kontinuierliche Berechnungen mit einschließen können.

6.3.1.2 Abrollung von Schleifen

Schleifen mit einem Schleifenrumpf S, die einer festen syntaktischen Struktur folgen und bei denen die Anzahl n der Wiederholungen statisch aufgrund der Schleifenbedingung c und der unausweichlich auszuführenden Zuweisungen zu einer Schleifendurchlaufzählervariable innerhalb von S berechenbar sind, werden mit dem Werkzeug *smi4loopunroll* in n sequentiell aufeinanderfolgende Codeblöcke S ausgerollt. Dabei wird die Schleifendurchlaufzählervariable durch entsprechende Konstanten direkt substitutiert.

Folgen die Schleifen nicht dem erkannten Muster, müssen diese mit manuellen Eingriffen eliminiert werden, da das Abstraktionsverfahren keine Schleifen akzeptiert.

6.3.1.3 Erzeugung von Single-Assignment-Code

Für die Berechnung von GJ-Paaren in Abschnitt 6.3.3 wird ein Datenflußgraph des Modells bzgl. kontinuierlicher Variablen benötigt. Dieser wird, wie in Abschnitt 6.3.2 beschrieben, aus dem SMI-Programm erzeugt und setzt dabei voraus, daß entlang eines jeden Kontrollflußpfades des Modells jede Variable höchstes einmal einen Wert zugewiesen bekommt. Diese vom SMI-Programm geforderte Eigenschaft des Single-Assignment-Code beschränkt sich dabei auf Variablen vom Typ *Real*, da andere Variablen in der Abstraktion unbeachtet bleiben.

Die Erzeugung von Single-Assignment Code auf einem Kontrollflußpfad wird möglich durch die Verwendung zusätzlicher Hilfsvariablen $\{v_i | 0 \leq i \leq k\}$ mit k als Anzahl der Zuweisungen an die Variable v auf diesem Pfad. Für jede Zuweisung einer Variablen v wird stattdessen die Variable v_i für die i-te Zuweisung auf dem Pfad verwendet und entsprechend entlang des Pfades für den nächsten darauf folgenden Lesezugriff wird wieder v_i anstelle von v verwendet. Nach dem letzten Zugriff auf die Variable v_k wird diese für die korrekte Berechnung des nächsten Schrittes der äußersten Schleife wieder in die Variable v geschrieben.

Da das Vorgehen einer solchen Analyse abermals die systematische chronologische Bearbeitung des Kontrollflußgraphen erfordert, wurde für diesen Algorithmus erneut das Tool *smiaux* bemüht, in welchem diese Funktionalität integriert ist. Die Implementierung bestimmt durch Analyse zunächst die betreffende Variablenmenge, die einer solchen Transformation unterzogen werden muß. Unter Verwendung von Bindungslisten wird die oben beschriebene Substitution ausgeführt, wobei in den Kontrollflußfusionspunkten jedesmal sichergestellt sein muß, daß bezüglich einer Variablen v für alle zusammenlaufenden Zweige eine gemeinsame Instanz v_i gemäß der Bindungslisten die aktuelle ist, die Instanzen müssen also *synchronisiert* werden. Dies wird durch eine Zuweisung zur Variablen v_i am Ende aller zusammenlaufenden Kontrollflußpfade erreicht, wie in Abbildung 6.12 am Beispiel der Variablen v = 'a' und v_i = 'a4' dargestellt. Für zustandslose Variablen muß die Synchronisation nur solange aufrechterhalten werden, wie noch nachfolgende Zugriffe auf diese Variablen erwartet werden. Der gestrichelt dargestellte Pfadteil in Abbildung 6.12 ist implizit[10] und repräsentiert den Fall, daß in einem **DCASE** oder

[10] Eine Überprüfung, ob die Disjunktion aller Bedingungen eine Tautologie und der implizite Pfad

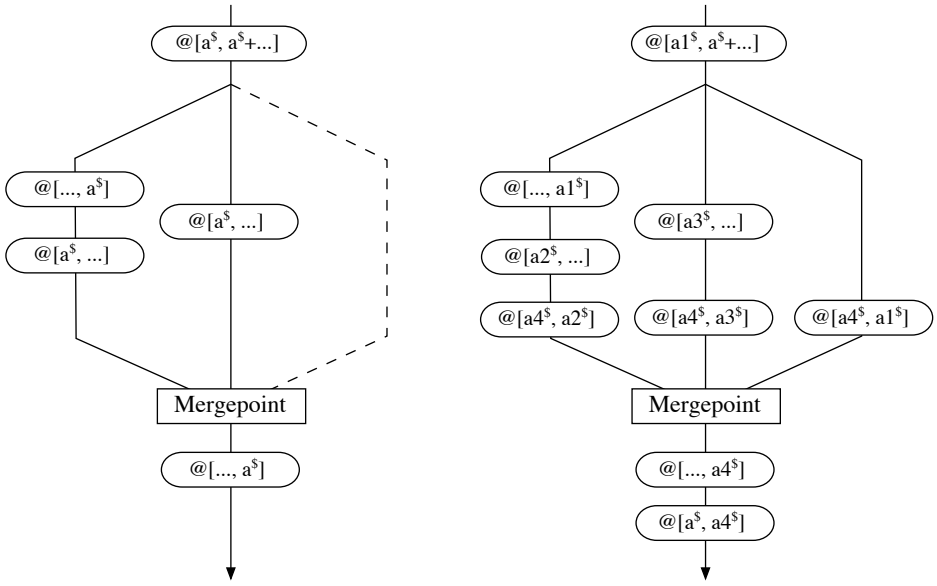


Abbildung 6.12: Beispiel für die Transformation eines SMI-Programms mit schematischen Zugriffen auf eine Variable 'a' in Single-Assignment-SMI, dargestellt als Kontrollflußpfaddiagramm. Der gestrichelte Pfad ist implizit

NDCASE keine der angegebenen Bedingungen erfüllt ist. Diese Pfadteile sind nun in explizite zu überführen und ebenfalls mit der erforderlichen Zuweisung zur gemeinsamen Instanz der relevanten Variablen auszustatten. Die abschließende Zuweisung $v := v_k$ muß nur für Zustandsvariablen erfolgen.

6.3.1.4 Kodierung von Zustandsformeln im SMI-Modell

Analog zum in Abschnitt 4.6.1 auf Seite 67 beschriebenen Ansatz zur Bezugnahme auf den kontinuierlichen Zustandsraum innerhalb von Zustandsformeln in der in CTL-spezifizierten Eigenschaft φ, müssen wir auch auf der SMI-Ebene eine vergleichbare Transformation durchführen. Hier gestaltet sich diese jedoch deutlich einfacher, da für jede Zustandsformel f lediglich eine boolsche SMI-Variable $obs^\$$ des Modus *Observer* dem SMI-Programm hinzugefügt werden muß. Als letzte Zuweisung am Ende der Endlosschleife im SMI-Programm und damit als letzte Zuweisung der SMI-Schrittfunktion, wird eine Zuweisung $obs^\$:= f$ angehangen. Damit hat die Variable $obs^\$$ automatisch den Zustandswert der Zustandsformel f, welche hernach in φ durch die Zustandsformel $obs^\$ = true$ substituiert werden kann.

6.3.2 Initialabstraktion

Das SMI-Programm liegt jetzt in einem Zustand vor, in dem die initiale Abstraktion als ein neues SMI-Programm erzeugt werden kann. Die Anforderungen an die initiale Abstraktion zur Erzeugung des abstrakten SMI-Programms $\mathcal{S}_{abs}$ aus dem konkreten, originalen SMI-Programm $\mathcal{S}_{orig}$ sind

$\mathcal{S}_{abs}, \mathcal{S}_{orig}$

- Elimination aller kontinuierlicher Berechnungen unter vollständiger Beibehaltung des diskreten Verhaltens und

somit ausgeschlossen ist, wird zusätzlich in *smiaux* durchgeführt.

- Sicherstellung der Berechenbarkeit der Pfadprojektion, wie in Abschnitt 4.3.2 auf Seite 39 gefordert.

Die folgenden Abschnitte schildern den Vorgang der Initialabstraktion einschließlich der Sicherstellung der Pfadprojizierbarkeit durch Einführung zusätzlicher Variablen in $\mathcal{S}_{abs}$.

6.3.2.1 Elimination kontinuierlichen Verhaltens

Die Trennung zwischen diskretem Verhalten und kontinuierlichen Berechnungen gestaltet sich auf der SMI-Ebene noch sehr übersichtlich, sofern keine Konversionen zwischen ganzzahligen und kontinuierlichen Größen vorhanden sind. Diese Problematik ist ausführlich in Abschnitt 6.6 diskutiert und soll der Einfachheit halber zunächst unbetrachtet bleiben. Kontinuierliches Verhalten erscheint im SMI-Programm dann ausschließlich in zwei verschieden Formen:

1. Vergleichsoperationen auf Ausdrücken kontinuierlichen Typs, z.B. $x - z \leq \pi \cdot y$
2. Zuweisungen kontinuierlicher Ausdrücke auf Variablen kontinuierlichen Typs, z.B. $x := y + z$

Die Elimination eines Vergleichoperationsausdrucks $e_1 \overset{\text{rel}}{\sim} e_2$ mit $\overset{\text{rel}}{\sim} \in \{<, \leq, >, \geq, =, \neq\}$ erreichen wir durch Substitution desselben mit einer boolschen Eingabevariable p, im Folgenden als *Propositionsvariable* bezeichnet mit $Expr(p) \overset{\text{def}}{=} e_1 \overset{\text{rel}}{\sim} e_2$, deren boolscher Wert in jedem Schritt von der Umgebung neu belegt werden kann, wodurch vom Model-Checker nicht-deterministisch ein Ergebnis der Vergleichsoperation geraten werden kann. Die Elimination der Zuweisungen gestaltet sich noch einfacher, da diese zunächst ersatzlos entfernt werden können. Dadurch werden auch alle Ausdrücke auf der rechten Seite der Zuweisung entfernt. Diese triviale Form der Abstraktion wird auch als *propositionale Abstraktion* bezeichnet.

6.3.2.2 Gewährleistung der Pfadprojektion

Die Einhaltung der Bedingung der Pfadprojizierbarkeit ist hingegen nicht trivial, da SMI zwischen zwei diskreten System-Zuständen durchaus unterschiedliche Berechnungen vollführen kann und somit nur aufgrund der Belegung diskreter Variablen nicht ersichtlich ist, welche Berechnungen auf dem kontinuierlichen Zustandsraum ausgeführt wurden. Dies entspricht somit eher einem Schritt-diskreten Hybriden Automaten gemäß Definition 12 auf Seite 31 aus Abschnitt 3.3, welcher mehrere Transitionen zwischen zwei Zuständen erlaubt. Nun könnte man natürlich die in Abschnitt 3.3 beschriebene Transformation zur Gewinnung eindeutiger Transitionen zwischen je zwei Zuständen durchführen und hernach für jede Transition, repräsentiert als ein Paar von Belegungen der diskreten SMI-Variablen, das entsprechende *GJ*-Paar ermitteln, jedoch ist dies aufgrund der Größe der Systeme nicht praktikabel und auch nicht nötig.

Da SMI über zustandslose Beobachtungsvariablen (*Observer*) verfügt, besteht grundsätzlich die Möglichkeit, mit deren Hilfe und entsprechender Verwendung nicht nur die jeweilig genommene Transition zwischen zwei Zuständen zu bestimmen, sondern darüber hinaus sogar direkt die Information bzgl. des zugehörigen *GJ*-Paares anzuzeigen. Damit kann die in Abschnitt 4.3.2 auf Seite 39 eingeführte Funktion θ in gewisser Hinsicht in das abstrakte SMI-Programm integriert werden, so daß in der späteren Pfadauswertung aus diesem die zugehörige *GJ*-Sequenz direkt ableitbar ist.

Propositions- und Trace-Variablen Für eine solche Aufbereitung des SMI-Programms ist zunächst sicherzustellen, daß durch die Abstraktion keine Information verloren geht, die nicht anderweitig wieder beschafft werden kann. Durch die Einführung von Propositionsvariablen als Substitut für das Ergebnis abstrahierter Vergleichsoperationen auf kontinuierlichen Ausdrücken können wir bereits mit Hilfe der Belegung der Propositionsvariablen am Ende eines SMI-Schritts rekonstruieren, wie der dazugehörige Vergleich ausgewertet werden muß, um einen Lauf des abstrakten SMI-Programms $\mathcal{S}_{abs}$ im dazugehörigen konkreten SMI-Programm $\mathcal{S}_{orig}$ nachzuvollziehen. Dies ist jedoch noch unzureichend, da die Information bzgl. zugewiesener kontinuierlich-wertiger Ausdrücke zu Variablen in $\mathcal{S}_{orig}$ in $\mathcal{S}_{abs}$ kein Pendant besitzt und bislang noch verloren geht. Wir können demnach mit Hilfe einer Belegung von Variablen aus $\mathcal{S}_{abs}$ am Ende eines SMI-Schrittes so noch nicht sagen, welche Zuweisungen ein dazugehöriger Lauf in $\mathcal{S}_{orig}$ ausgeführt hätte.

Diese Zuweisungen sind jedoch durch einen im Rahmen eines SMI-Schrittes genommenen Kontrollflußpfad eindeutig festgelegt, da dieser sich isomorph auf einen korrespondierenden Kontrollflußpfad in $\mathcal{S}_{orig}$ projizieren läßt. Es wäre somit ausreichend, das abstrakte SMI-Programm um Kontrollflußpfadbeobachtungsvariablen zu ergänzen, die am Ende des SMI-Schritts eine Rekonstruktion des Kontrollflußpfads erlauben. Einfacher verwendbar sind hingegen *Zuweisungsbeobachtungsvariablen* $t_v^\$$, die für jede ausgeführte Zuweisung $\mathfrak{a}$ zu einer Variablen $v \in V_{Real}^\$$ in $\mathcal{S}_{orig}$ eine eindeutige Belegung $\sigma(t_v^\$) = k_\mathfrak{a}$ in $\mathcal{S}_{abs}$ erhalten, wobei $k_\mathfrak{a} \in \{0, 1, \ldots, n_v\}$ mit n_v als der Anzahl von Zuweisungen zu der Variablen v in $\mathcal{S}_{orig}$. Wir ergänzen $\mathcal{S}_{abs}$ daher um solche Zuweisungsbeobachtungsvariablen, im Folgenden auch als *Trace-Variablen* bezeichnet, welche die spätere Rekonstruktion zugehöriger Zuweisungen aus $\mathcal{S}_{orig}$ gestatten. Dabei wird für jede Variable $v \in V_{Real}^\$$ in $\mathcal{S}_{orig}$ eine Trace-Variable $t_v^\$$ eingeführt, die vom Typ $Integer_0^{n_v}$ ist. Für jede Zuweisung $\mathfrak{a}$ in $\mathcal{S}_{orig}$ wird an der korrespondierenden Stelle in $\mathcal{S}_{abs}$ eine Zuweisung $t_v^\$:= k_\mathfrak{a}$ eingeführt, wobei $t_v^\$$ zu Beginn des SMI-Schritts mit 0 initialisiert wird. Entsprechend bedeutet eine Belegung $\sigma(t_v^\$) = 0$ am Ende eines durch $\Im$ berechneten SMI-Schrittes, daß der dazugehörige Kontrollflußpfad keine Zuweisung zu v beinhaltet hat.

In der nachfolgenden Auswertung der Belegungen müssen wir nicht zwischen syntaktisch gleichen Zuweisungen $\mathfrak{a}$ unterscheiden und verwenden daher für gleiche Zuweisungen auch gleiche Konstanten $k_\mathfrak{a}$. Gegenüber der Verwendung von Kontrollflußpfadbeobachtungsvariablen verlieren wir durch so eingesetzte Zuweisungsbeobachtungsvariablen die Information der Reihenfolge der Zuweisungen. Die in Abschnitt 6.3.1.3 auf Seite 130 beschriebene Einführung von Single-Assignment-Code erwirkt jedoch die Eigenschaft, daß jede Variable in $\mathcal{S}_{abs}$ höchstens einmal einen Wert zugewiesen bekommt. Aufgrund der in Abschnitt 6.2.2.2 auf Seite 126 beschriebenen Substitution von gestrichenen Variablen $v^\$$ durch v für Lesezugriffe, denen kein Schreibzugriff auf derselben Variablen vorangegangen sein kann, sind Abhängigkeiten von der Ausführungsreihenfolge der Zuweisungen damit beschränkt auf die einzuhaltende partielle Ordnung der Abhängigkeiten der zugewiesenen Ausdrücke. Die Zuweisungen können jetzt also unter einer deklarativen Semantik betrachtet werden.

In Abbildung 6.13 ist ein SMI-Programm und dessen einfache Abstraktion unter Verwendung von Propositions- und Trace-Variablen abgebildet. Den so erzeugten SMI-Codeblock bezeichnen wir mit $\mathcal{S}_{absA_0}$. $\mathcal{S}_{absA_0}$

Wir werden die Menge der Propositionsvariablen mit $PV \stackrel{\text{def}}{=} \{p_1, p_2, \ldots, p_{|PV|}\}$ bezeichnen und die Menge alle Trace-Variablen mit $TV \stackrel{\text{def}}{=} \left\{t_{v_1}^\$, t_{v_2}^\$, \ldots, t_{v_{|TV|}}^\$\right\}$. Für die Belegung aller Propositionsvariablen schreiben wir $\sigma(PV) \stackrel{\text{def}}{=} \{p_1 = \sigma(p_1), p_2 = \sigma(p_2), \ldots, p_{|PV|} = \sigma(p_{|PV|})\}$, sowie $\sigma(TV) \stackrel{\text{def}}{=} \left\{t_{v_1}^\$ = \sigma\left(t_{v_1}^\$\right), t_{v_2}^\$ = \sigma\left(t_{v_1}^\$\right), \ldots, t_{v_{|TV|}}^\$ = \sigma\left(t_{v_{|TV|}}^\$\right)\right\}$ für die Belegung aller PV, TV

```
CODE
  WHILE 1 DO
    DCASE
      [] c1:
        @[z$ , x]
      [] ¬(c1):
        @[z$ , 3.200]
    DESAC
    @[out$ , ((3.000+z$)<(0.2000))]
  OD
END
```

(a) $\mathcal{S}_{orig}$

```
CODE
  WHILE true DO
    @[T_l0$ , 0]
    DCASE
      [] c1:
        @[T_z$ , 1]
      [] ¬(c1):
        @[T_z$ , 2]
    DESAC
    @[out$ , (p0)]
  OD
END
```

(b) $\mathcal{S}_{abs}$

Abbildung 6.13: Beispiel für einfache Abstraktion unter Verwendung von Propositions- und Trace-Variablen

Trace-Variablen.

Wie wird nun aufgrund einer Belegung von Propositions- und Trace-Variablen das dazugehörige *GJ*-Paar berechnet? In einem sequentiellen SMI-Programm beinhalten diese Belegungen jeweils nur Bruchteile der gesuchten Information. Beispielsweise ist das Wissen um den Wahrheitsgehalt einer Propositionsvariablen p nur dann vollständig auswertbar, wenn der Kontext vorheriger Zuweisungen desselben SMI-Schrittes zu allen direkt und indirekt in $Expr(p)$ Einfluß nehmenden Variablen bekannt ist. Diese Information befindet sich wiederum in der Belegung der Trace-Variablen. Ohne diese Zusatzinformation ist eine Interpretation von Propositionsvariablenbelegungen im Allgemeinen unmöglich. Für die partielle Auswertung einer Propositionsvariablenbelegung definieren wir die Funktion

$\mathcal{E}$

$$\mathcal{E} : (\sigma, p) \mapsto \begin{cases} Expr(p) & \text{gdw } \sigma(p) = true \\ \neg Expr(p) & \text{gdw } \sigma(p) = false \end{cases}$$

$\mathcal{A}_\sigma$,$\mathfrak{A}_\sigma$
$\mathcal{E}_{Assignments}$
$\mathcal{E}$,$\mathcal{E}_{SMI}$

Wie bereits diskutiert kann die Belegung σ der Trace-Variablen auf eine Menge von deklarativen Zuweisungen $\mathcal{A}_\sigma = \{x_1 := e_1, x_2 := e_2, \ldots, x_k := e_k\}$ projiziert werden, was wir formal durch eine Funktion $\mathcal{E}_{Assignments} : \sigma(TV) \mapsto \mathcal{A}_\sigma$ beschreiben wollen. Weitergehend führen wir für die nachfolgende Verwendung die Funktion $\mathcal{E}$ ein, welche anstelle einer Zuweisungsmenge eine Konjunktion von entsprechenden Vergleichen generiert mit $\mathcal{E} : \sigma(TV) \mapsto (x_1 = e_1 \wedge x_2 = e_2 \wedge \cdots \wedge x_k = e_k)$. Daneben werden wir wir für die Beschreibung der Jump Set Funktion eines SMI-Schritts die Funktion $\mathcal{E}_{SMI}$ benötigen, welche anstelle der Zuweisungsmenge $\mathcal{A}_\sigma$ eine gemäß der durch Datenabhängigkeiten induzierten Ordnung sortierte SMI-Anweisungssequenz $\mathfrak{A}_\sigma$ erzeugt.

Damit können wir die Belegung σ einer Propositionsvariablen p nun vollständig auswerten, indem wir $\mathcal{E}(\sigma(TV))$ mit $\mathcal{E}(\sigma, p)$ zu dem folgenden Ausdruck konjugieren:

$$\mathfrak{g}_p^\sigma := \mathcal{E}(\sigma(TV)) \wedge \mathcal{E}(\sigma, p) \tag{6.1}$$

Der *ausgewertete Propositionsausdruck* $\mathfrak{g}_p^\sigma$ beschreibt durch die Interpretationsfunktion $\mathcal{I}$ gegebene Menge

$$\mathcal{I}\left(\mathfrak{g}_p^\sigma\right) \stackrel{\text{def}}{=} \left\{ \sigma' \,\middle|\, \mathfrak{g}_p^\sigma \left[V_{Real}^{\$} : v \backslash \sigma'(v) \right] = true \right\}$$

von Belegungen von SMI-Variablen kontinuierlichen Typs in $\mathcal{S}_{orig}$, für die der Ausdruck $\mathfrak{g}_p^\sigma$ wahr ist, wobei der Postfix-Operator $[V_{subst} : v \backslash e]$ in einem gegebenen Ausdruck alle Vorkommen der Variablen v, $v \in V_{subst}$, durch e substituiert. Da

ein Guard Set jedoch erst durch die Gesamtheit aller interpretierten Propositionsvariablenbelegungen beschrieben ist, ist dies immernoch nur eine Teilaussage eines Guard Set in Gestalt einer Obermenge, die weiter einzuschränken ist. Erst die Schnittmenge aller Interpretationen bezüglich der Belegung σ ergibt die Abbildung selbiger auf ein vollständiges Guard Set γ mit

$$\sigma \mapsto \gamma := \bigcap_{p \in PV} \mathcal{I}\left(\mathfrak{g}_p^\sigma\right) = \mathcal{I}\left(\bigwedge_{p \in PV} \mathfrak{g}_p^\sigma\right) \text{ mit } \mathfrak{g}_p^\sigma = \mathcal{E}\left(\sigma\left(TV\right)\right) \wedge \mathcal{E}\left(\sigma, p\right)$$

Die Ausdrücke $\mathfrak{g}_p^\sigma$ beschreiben somit genau solche Guard-Komponenten $\gamma_i^{u_i}$ wie in Abschnitt 4.7.2.2 auf Seite 79 vorgestellt, wobei der Index i auf die jeweilige Propositionsvariable p verweist, während der Propositionsvariablen-spezifische Index u_i die Belegungen σ bezeichnen, für die $\gamma_i^{u_i} = \mathcal{I}\left(\mathfrak{g}_p^\sigma\right)$ gilt[11].

In den Bemühungen, die Auswertung der Propositionsausdrücke im Kontext vorangegangener Zuweisungen durchführen zu können, haben wir nun auch schon die vollständige Transformation des kontinuierlichen Zustandsraums durch $\mathcal{A}_\sigma$ beschrieben. Gemäß der in Abschnitt 6.1.2 auf Seite 117 definierten SMI-Semantik entspricht die Ausführung der Zuweisungen aus $\mathcal{A}_\sigma$ als sortierte SMI-Anweisungssequenz $\mathfrak{A}_\sigma$ der Jump Set Funktion $\mathfrak{S} : (\mathfrak{A}_\sigma, \sigma_{Real}) \mapsto \sigma'_{Real}$ bezüglich einer Belegung σ_{Real} der kontinuierlichen Variablen im konkreten SMI-Programm. Somit ist die Abbildung von Trace-Variablen-Belegungen auf die Jump Set Funktion ζ beschrieben durch

$$\sigma \mapsto \zeta := \mathfrak{S}\left(\mathcal{E}_{SMI}\left(\sigma\left(TV\right)\right)\right)$$

In Abschnitt 4.3.3 auf Seite 47 haben wir die *GJ*-Paare uninterpretiert als Zeichen eines Alphabets Σ verwendet und zur Konstruktion des ω-Automaten genutzt. Wie wir an der hier definierten Projektion $\sigma \mapsto (\gamma, \zeta)$ sehen, sind *GJ*-Paare auf dieser Repräsentationsebene als Teilbelegungen $\sigma \downarrow PV \cup TV$ beschrieben, so daß sich deren Verwendung als Zeichen des Alphabets anbietet. Wir verwenden demgemäß $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$, wobei es sich bei den σ_i jeweils um partielle Belegungen, beschränkt auf die Menge $PV \cup TV$ handelt. Σ wird in jeder Iteration um neu beobachtete Belegungen ergänzt und umfaßt somit nur die bereits beobachteten. Σ

Wir haben nun eine Abbildung erhalten, die eine Belegung von Variablen in $\mathcal{S}_{abs}$ auf durch konjunktive Ausdrücke beschriebene Guard Sets und auf eine durch Ausführung einer Menge von Zuweisungen beschriebenen Jump Set Funktion dargestellt wird, und damit eine der Funktion θ aus Definition 18 auf Seite 41 entsprechenden Funktionalität beschrieben.

6.3.3 Minimale *GJ*-Repräsentationen

Die im vorangegangenen Abschnitt definierte Abbildung $\sigma \mapsto (\gamma, \zeta)$ beinhaltet zwei Probleme, welche sich aus der typischen ausschweifenden Verwendung von zustandslosen Hilfsvariablen insbesondere bei von automatischen Code-Generatoren generiertem C-Code, aber auch bei der Erzeugung von Single-Assignment-Code nach Abschnitt 6.3.1.3 zur schrittweisen Berechnung von neuen Belegungen für kontinuierliche Variablen ergeben:

1. *(Effizienz)* Für große Modelle bestehen die ausgewerteten Propositionsausdrücke $\mathfrak{g}_p^\sigma$ aus Gleichung (6.1) aus sehr vielen konjunktiven Ausdrücken und die rekonstruierten Zuweisungsmengen $\mathcal{A}_\sigma$ beinhalten sehr viele Zuweisungen. Dies induziert beträchtliche Ineffizienzen in der späteren Pfadanalyse durch wiederholt erforderliche Aufstellung großer konjunktiver Formeln.

[11] Für den Vergleich von Belegungen und Teilmengen des $\mathbb{R}^n$ siehe Bemerkung in Abschnitt 6.1.2 auf Seite 117

2. (*Injektivität*) Die Abbildung $\sigma \mapsto (\gamma, \zeta)$ ist nicht injektiv, d.h. es können mehrere Belegungen auf das gleiche *GJ*-Paar abgebildet werden, mit entsprechender Konsequenz für Konfliktanzahl, Iterationen und Automatengröße im ω-CEGAR Verfahren, welche sich dadurch allesamt vervielfachen.

Aus beiden Gründen ist eine Vorauswertung der Belegungen notwendig, welche eine direkte injektive Zuordnung von σ zu

- bereits ausgewerteten, jeweils als eine (Un-)Gleichung repräsentierten Propositionsausdrücken $\mathfrak{g}_p^\sigma$, sowie zu
- als Zuweisungsmengen der Kardinalität $|\mathcal{A}_\sigma| = 1$ repräsentierte Jump Set Funktionen erlaubt.

Das Ziel dieses Abschnitts ist daher die Hinzunahme weiterer diskreter Variablen $t_1^{*\$}, t_2^{*\$}, .., t_m^{*\$}$ in $\mathcal{S}_{abs}$, deren Belegungen eine effiziente injektive Zuordnung zu diesen ermöglichen. Zur Unterscheidung der Trace-Variablen aus dem vorigen Abschnitt bezeichnen wir diese ausschließlich als *vorausgewertete Trace-Variablen* oder *Super-Trace-Variablen* und bezeichnen deren Menge mit TV^*.

TV^*

Konzeptionell wird die notwendige Voranalyse durch eine umfassende Substitution aller Zuweisungen in $\mathcal{A}_\sigma$ erreicht, so daß für die Jump Set Funktionen nur eine einzige Zuweisung für jede zustandsbasierte Variable $v_k \in V_{Real}^{\$state} = \{v_1^\$, v_2^\$, \ldots, v_n^\$\}$ übrig bleibt, welche dann den Abbildungsvorschriften ζ_{D_k} aus Abschnitt 4.7.2.2 auf Seite 79 entsprechen. Da die Zuweisungsmenge $\mathcal{A}_\sigma$ auch bei der Erzeugung der Guard Set Konjunktionen benötigt wird, werden diese ebenfalls durch Substitution verkleinert, jedoch nicht notwendigerweise bis auf Zuweisungen zu Variablen aus $V_{Real}^{\$state}$, sondern bis auf genau die Variablenmenge, die in $Expr(p)$ vorkommen, welche unter Wiederverwendung der Funktion zur Ermittlung von Variablenvorkommen durch $\mathcal{V}(Expr(p))$ beschrieben ist. Die letzte Substitution erfolgt dann auf der Ebene der Konjunktion $\mathfrak{g}_p^\sigma$, so daß ein einzelner Vergleichsoperationsausdruck $e_1 \overset{\text{rel}}{\sim} e_2$ zurückbleibt.

Für die Durchführung der Substitutionen sind zwei Vorgehensweisen denkbar:

1. (*Eager Evaluation*) Die Berechnung der vollständigen Zuordnung erfolgt im Voraus.
2. (*Lazy Evaluation*) Die Berechnung der Zuordnung erfolgt nur für bereits beobachtete Belegungen der Propositions- und Trace-Variablen.

6.3.3.1 Eager Evaluation

Für die Umsetzung einer *Eager Valuation* Strategie zur Durchführung der Substitutionen sind wiederum zwei Ansätze zu möglich:

1. Rekursive Aufzählung aller möglichen Kontrollflußpfade
2. Rekursive Aufzählung aller möglichen Datenflußpfade

Gegenüber der Datenflußpfadaufzählung schränkt die Kontrollflußpfadaufzählung die Anzahl der möglichen Kombinationen von Zuweisungen genau dann ein, wenn unterschiedliche Zuweisungen in sich gegenseitig ausschließenden Zweigen existieren. Diese strukturell ausgeschlossenen Zuweisungskombinationen werden bei einer Kontrollflußpfadaufzählung nicht berücksichtigt. Eine Menge sequentieller Kontrollflußpfadverzweigungen S_{Case} resultiert hingegen in einem Produkt $\prod_{s \in S_{Case}} branches(s)$ verschiedener Zuweisungskombinationen, wobei $branches(s)$ die Anzahl der unterschiedenen Fälle einer Fallunterscheidungsanweisung angibt,

welche ihrerseits wieder durch eine solche Formel zu berechnen ist. Nach Vorwärtssubstitution wird sich die Menge aller strukturell möglichen Zuweisungskombinationen jedoch erheblich reduzieren, da durch sich wiederholende Zuweisungen in verschiedenen Zweigen, wie sie insbesondere durch das in Abschnitt 6.2.2 beschriebene praktizierte Inlining von Funktionsaufrufen erzeugt werden, verschiedene Trace-Variablenbelegungen auf gleiche Zuweisungsausdrücke hinauslaufen. Der Aufwand dieses Ansatzes ist mit $O(m^n \cdot |V_{Real}|)$ zu beziffern, wobei

- n die Anzahl der Fallunterscheidungsanweisungen und
- m die maximale Anzahl von Fallunterscheidungen in einer Fallunterscheidungsanweisung ist.

Durch eine einmalige Kontrollflußanalyse mit entsprechender Parallelverwaltung aller Variablen und deren möglichen zugewiesenen Ausdrücken und begleitender vollständiger Substitution derselben bei jeder Zuweisung im Kontrollfluß wäre dies auch in optimierter Form realisierbar, jedoch ohne Auswirkung auf die Aufwandsklasse.

Demgegenüber werden bei einer Datenflußpfadaufzählung auch strukturell ausgeschlossene Kombinationszuweisungen berücksichtigt. Der Vorteil dieses Ansatzes liegt jedoch in der effizienteren Berücksichtigung gleicher Zuweisungen an verschiedenen Positionen im SMI-Programm, welche hier lediglich eine Zuordnungstabelle vergrößern. Der Aufwand dieser Berechnung liegt in $O((a \cdot o)^l)$, wobei

- a die maximale Anzahl unterschiedlicher Zuweisungen zu ein- und derselben Variable ist,
- o die maximale Anzahl verschiedener Variablen in den zugewiesenen Ausdrücken und
- l die maximale Datenpfadlänge ohne Verzögerungselemente.

Während o unabhängig von der Modellgröße meist eine einstellige Konstante ist[12] und $a \sim m$, gilt immer $l < |V_{Real}|$. Ein Vergleich von l und n ist stark modellabhängig, für kontrolldominierte Modelle gilt jedoch im Allgemeinen $l < n$, da hier auch $|V_{Real}| < n$ gilt. Für solche Systeme haben wir daher $O((a \cdot o)^l) < O(m^n \cdot |V_{Real}|)$. Die unnötig berechneten strukturell ausgeschlossenen Kombinationen können vor diesem Hintergrund vernachlässigt werden, weswegen der Datenflußpfad-basierte Ansatz implementiert wurde und im Folgenden detaillierter beschrieben ist.

Datenflußpfad-basierte Auswertung Für jede Variable $v^\$$ wird eine bijektive Abbildung $M_{v^\$}$ berechnet, im Folgenden auch *Zuordnungstabelle* genannt. In dieser wird jedem möglichen, durch Substitution ermittelten Ausdruck e, der nach der Substitution immer als lineare Gleichung vorliegend ist, die Menge der dazugehörigen Trace-Variablen-Belegungen zugeordnet, die die Berechnung der Variablen $v^\$$ beschreibt. Dabei wird die auf dem Datenflußgraphen durch Abhängigkeiten induzierte partielle Ordnung ausgenutzt und zunächst $M_{v^\$}$ für die Variablen $v^\$$ berechnet, für welche in den potentiell zugewiesenen Ausdrücken r keine Variablen aus $V^\$_{Real}$ vorkommen, $\mathcal{V}(r) \cap V^\$_{Real} = \emptyset$, diese sich also ausschließlich auf Konstanten und Belegungen des vorangegangenen SMI-Schrittes abstützen. Da in $M_{v^\$}$

[12] Komplexere Ausdrücke sind für einzelne Variablenzuweisungen aufgrund der Unübersichtlichkeit nicht zu erwarten, da diese letztlich vom Modell-Entwickler vergeben werden. Komplexere Berechnungen teilen sich daher eher auf Zwischenschritte unter Verwendung von Hilfsvariablen auf, so daß diese Komplexitätsgröße logarithmisch in die Anzahl der Variablen und die maximale Länge des Datenflußgraphen eingeht.

Abwesenheit von nicht terminierenden, inneren Schleifen auch keine nicht durch Verzögerungselemente unterbrochenen Schleifen im Datenflußgraphen existieren können, gibt es immer Variablen dieser Eigenschaft. Nachdem für alle diese Variablen die dazugehörigen Abbildungen berechnet worden sind, können diese als Basis für die Berechnung der Abbildungsvorschriften der in der partiellen Ordnung des Datenflußgraphen folgenden Variablen berechnet werden, usw.

Mit $\mathcal{A}_{v^\$}$ als die Menge aller Zuweisungen zur Variablen $v^\$$ in $\mathcal{S}_{orig}$ ergibt sich die Berechnung der Abbildung $M_{v^\$}$ für $v^\$ \in V^\$_{Real}$ damit aus

$$M_{v^\$} = \bigcup_{\substack{\mathfrak{a}\in\mathcal{A}_{v^\$}:\\ \sigma'(t^\$_v)=k_\mathfrak{a},\mathfrak{a}=\text{'@}[v^\$,r]',\\ (\mathcal{V}(r)\cap V^\$)=\{v^\$_1,\ldots,v^\$_k\}}} \bigcup_{\substack{m_{v^\$_1},\ldots,m_{v^\$_k}:\\ m_{v^\$_i}=(e_{v^\$_i}\mapsto B_{v^\$_i})\in M_{v^\$_i},\\ B_{v^\$_i}=\{\sigma^1_{v^\$_i},\ldots,\sigma^n_{v^\$_i}\}}} \left\{ \begin{array}{c} \mathcal{N}(v^\$ = r[\mathcal{V}(r) : v^\$_i \backslash e_{v^\$_i}]) \mapsto \\ \bigcup_{\sigma_{v^\$_i}\in B_{v^\$_i}} \{\sigma_{v^\$_1} \cup \cdots \cup \sigma_{v^\$_k} \cup \sigma'\} \end{array} \right\}$$

wobei, wie bereits eingeführt, die Trace-Variable $t^\$_v$ genau dann unter einer Belegung σ' den Wert $k_\mathfrak{a}$ besitzt, wenn der auf $\mathcal{S}_{orig}$ projizierte Kontrollfluß eines Laufs von $\mathcal{S}_{abs}$ die Zuweisung $\mathfrak{a}$ ausführt.

Da semantisch äquivalente Gleichungen e in mannigfaltiger syntaktischer Form existieren können, ist eine Normalisierung wünschenswert, in der die syntaktische Repräsentation bei semantischer Äquivalenz übereinstimmend ist. Andernfalls würde das Ziel einer injektiven Abbildung von Super-Trace-Variablenbelegungen in *GJ*-Paare verfehlt. Eine geeignete Normalisierung dieser linearen Gleichungen ist durch eine Zusammenfassung aller Koeffizienten sowie einer lexikographischen Ordnung auf den Variablenbezeichnern gegeben. Eine solche Normalisierung führt
$\mathcal{N}$ die Funktion $\mathcal{N}$ in obiger Berechnungsvorschrift auf dem gegebenen Ausdruck durch und gewährleistet damit, daß semantisch äquivalente Ausdrücke auch syntaktisch gleich sind.

Da Zuweisungen nur auf gestrichenen Variablen vorkommen können, sind in der obigen Berechnungsvorschrift alle Variablen ausdrücklich in gestrichener Form verwendet. Für Vorkommen ungestrichener Variablen in r sind keine Zuordnungstabellen notwendig, da es sich bei diesen immer um Konstanten oder Werte aus dem vorherigen SMI-Schritt handelt und somit nichts zu substituieren ist.

Die Befolgung der partiellen Ordnung in der Reihenfolge der Berechnung wird automatisch durch das Prinzip der *lazy evaluation*[13] erreicht, d.h. $M_{v^\$_i}$ wird erst dann berechnet, wenn es benötigt wird, wobei einmal berechnete $M_{v^\$_i}$ natürlich wiederverwendet und nicht jedesmal neu berechnet werden. Die zur Berechnung von $M_{v^\$}$ benutzten Propositions- und Trace-Variablenbelegungen $B_{v^\$_i}$ sind durch obige Berechnung nur partielle und keine vollständigen Belegungen aller Propositions- und Trace-Variablen, wodurch sich deren Umfang erheblich einschränkt. Es werden nur die Einzelbelegungen berücksichtigt, die für die Determinierung der zugewiesenen Ausdrücke notwendig sind.

Analog zur Berechnung der Zuordnungstabellen für Variablen können wir Zu-
$M_{Expr(p)}$ ordnungstabellen $M_{Expr(p)}$ für Propositionsausdrücke berechnen:

[13] Nicht zu verwechseln mit der alternativen Strategie zur Berechnung der Zuordnung in Abschnitt 6.3.3.2.

$$M_{Expr(p)} = \bigcup_{\substack{\sigma'=(p\mapsto b),b\in\mathbb{B}:\\ V_p=\mathcal{V}(Expr(p))\cap V^\$=\{v_1^\$,\ldots,v_k^\$\}}} \bigcup_{\substack{m_{v_1^\$},\ldots,m_{v_k^\$}:\\ m_{v_i^\$}=(e_{v_i^\$}\mapsto B_{v_i^\$})\in M_{v_i^\$},\\ B_{v_i^\$}=\{\sigma^1_{v_i^\$},\ldots,\sigma^n_{v_i^\$}\}}} \left\{ \begin{array}{c} \mathcal{N}(\mathcal{E}(\sigma',p)[V_p : v_i^\$\backslash e_{v_i^\$}]) \mapsto \\ \bigcup_{\sigma_{v_i^\$}\in B_{v_i^\$}} \{\sigma_{v_1^\$} \cup \cdots \cup \sigma_{v_k^\$} \cup \sigma'\} \end{array} \right\}$$

In Abbildung A.1 auf Seite 186 findet sich ein Beispiel des Ergebnisses dieser Berechnungen für ein kleines SMI-Programm.

Vorausgewertete Trace-Variablen Die als Ergebnis vorliegenden Abbildungen $M_{v_1^\$}, M_{v_2^\$}, \ldots, M_{v_{|TV|}^\$}$ und $M_{Expr(p_1)}, M_{Expr(p_2)}, \ldots, M_{Expr(p_{|PV|})}$ werden nun zur „Dekodierung" der in den Trace-Variablenbelegungen verschlüsselten *GJ*-Paaren verwendet. Da für die Jump Set Funktion nur die zustandsbasierten Variablen von Interesse sind, beschränken wir uns bei den Zuordnungstabellen für Variablen auf diejenigen $M_{v^\$}$, für die $v^\$ \in V^{\$state}_{Real}$ gilt.

Wir führen für jede Zuordnungstabelle M_i eine Nummerierung $exprno_{M_i} : \mathbb{N} \to (M_i \cup \{true \mapsto \emptyset\})$ der einzelnen Ausdrücke und den zugeordneten Trace-Variablenbelegungen ein, wobei für alle Zuordnungstabellen gelten soll $exprno_{M_i}(0) = \{true \mapsto \emptyset\}$, d.h. der Wert 0 repräsentiert explizit einen *don't-care*-Wert. *exprno*$_M$

Für jedes M_i eines Propositionsausdrucks $Expr(p)$ bzw. einer zustandsbehafteten Variable $v^\$$ wird eine Super-Trace-Variable $t_i^{*\$}$ in $\mathcal{S}_{abs}$ eingeführt, welche am Ende des Schleifenrumpfes der äußeren Endlosschleife eine Ordnungszahl $n \in \mathbb{N}$ in Abhängigkeit der (partiellen) Trace-Variablenbelegungen durch $t_i^{*\$} := n$ zugewiesen bekommt. Dabei gilt

$$t_i^{*\$} := n \Leftrightarrow exprno_{M_i}(n) = (e \mapsto B) \wedge \exists \sigma' \in B : \sigma' = \sigma_{\downarrow TV^*}$$

In Abbildung A.2 auf Seite 187 ist dieser Zuweisungsblock für das Beispiel aus Abbildung A.1 auf Seite 186 dargestellt. Damit erhalten wir für jede Belegung σ von $\mathcal{S}_{abs}$ unter Zuhilfenahme der Zuordnungstabellen direkt die bereits zusammengefassten Ausdrücke, die die Guard Sets beschreiben bzw. die der Berechnung der Funktionswerte der einzelnen Dimensionen der Jump Set Funktion entsprechen.

Damit ist die Systematik der Dekomposition von *GJ*-Paaren aus Abschnitt 4.7.2.2 auf Seite 79 nun offen gelegt: In Tabelle 4.2 auf Seite 81 entsprechen die Mengen G_i den Ausdrücken der Zuordnungstabellen der Propositionsvariablen p_i mit

$$\gamma_i^n \mapsto \sigma\left(t_{p_i}^{*\$}\right) = n \mapsto \left\{exprno_{M_{Expr(p_i)}}(n) \mapsto B\right\} \in M_{Expr(p_i)} \tag{6.2}$$

Analog dazu entsprechen die Jump-Komponenten-Mengen D_i aus Tabelle 4.3 auf Seite 82 den Ausdrücken der Zuordnungstabellen zustandsbasierter Variablen mit

$$\zeta_{D_i}^n \mapsto \sigma\left(t_{v_i}^{*\$}\right) = n \mapsto \left\{exprno_{M_{v_i}}(n) \mapsto B\right\} \in M_{v_i} \tag{6.3}$$

Dabei wird die spezielle Funktion $\zeta_{\mathbb{R}}$ abgebildet durch $\zeta_{\mathbb{R}} \mapsto \sigma\left(t_{v_i}^{*\$}\right) = 0 \mapsto \{true \mapsto \emptyset\}$. Die in Abschnitt 4.7.2.2 auf Seite 79 eingeführt Funktion $num(J,k)$ liefert für die Menge der Jump Set Funktionen J und der k-ten Dimension gerade den Wert $\left|M_{v_k^\$}\right|$, wobei die zustandsbehaftete Variable v_k der Dimension k entspricht.

Der Nachteil der *Eager Evaluation* Strategie zur Berechnung der Zuordnung liegt in der exponentiellen Zunahme des Berechnungsaufwands für die Zuordnung mit der Größe des Datenflußgraphen bzw. der diskreten Struktur des Modells (genauer: Anzahl der Fallunterscheidungsanweisungen). Ob nun Kontrollflußpfad-

oder Datenflußpfad-bezogenes Vorgehen, spätestens bei der Verwendung mehrerer großer Kennlinienfelder explodieren die Größen $|M|$ der Zuordnungstabellen, so daß auf die im folgenden Abschnitt beschriebene *Lazy Evaluation* Strategie gewechselt werden muß.

6.3.3.2 Lazy Evaluation

Bei dieser Strategie werden die im vorigen Abschnitt beschriebenen Zuordnungstabellen nur für solche Belegungen von Trace-Variablen berechnet, die bereits in vorangegangenen Pfaden beobachtet worden sind. Dadurch wird das Explosionsproblem umgangen und die Zuordnungstabellen ebenso wie das abstrakte SMI-Programm iterativ verfeinert.

Für die Umsetzung ist hierbei wichtig, daß die Funktionen $exprno_M$ im Rahmen der Verfeinerung nur erweitert, nicht jedoch geändert werden, da die Belegungen der Super-Trace-Variablen das Alphabet Σ des ω-Automaten stellen. Eine Änderung der Zuordnung würde die Transformation sämtlicher Zeichen aller Konflikte nach sich ziehen.

Die *Lazy Evaluation* Strategie birgt gegenüber der *Eager Evaluation* Strategie den Nachteil zusätzlich benötigter Iterationen der Abstraktionsverfeinerung. Wann immer eine bislang unbekannte Propositions- und Trace-Variablenbelegung σ' in einem Pfad auftritt, welche nachträglich in der Pfadanalyse als einem *GJ*-Paar $\left(\gamma_i^{u_i}, \zeta_{D_j}^{u_j}\right)$ zuzuordnend festgestellt wird, das Teil eines bekannten und im ω-Automat verhinderten Konflikts ist, so diente diese Iteration nicht der Erweiterung des ω-Automaten, sondern lediglich der Erweiterung der Zuordnungstabellen. Da diese in $\mathcal{S}_{abs}$ durch Zuweisungen kodiert sind, wird dadurch natürlich auch das abstrakte Modell verfeinert, jedoch in kleineren Schritten. Die Lazy Evaluation Strategie sollte daher nur dann zum Einsatz kommen, wenn der Datenflußgraph bzgl. einer *Eager Evaluation* Strategie zu groß ist.

6.3.4 Konstruktion des ω-Automaten

Im Gegensatz zu der von Kapitel 4 aufgrund der SMI-Repräsentation erheblich anders ausgestalteten Umsetzung der Algorithmen in den vorherigen Abschnitten, ist die Konstruktion des ω-Automaten sehr stark an die in Abschnitt 4.3.3 beschriebene Verfahrensweise angelehnt. Der reguläre und der ω-Automat werden entsprechend der Algorithmen 3, 5 und 4 konstruiert und minimiert, einschließlich der in Algorithmus 7 beschrieben Erweiterung zur erweiterten Minimierung, sowie der etwaigen in Abschnitt 4.7.2.1 geschilderten Determinisierung des Automaten. Die Kreuzprodukte zur Komposition des regulären und ω-Automaten, sowie dessen Komposition mit der initialen Abstraktion gestaltet sich jedoch völlig anders, ebenso wie die Kodierung der Automaten in SMI.

6.3.4.1 Behandlung des Konfliktalphabets

Die Zeichen des Alphabets Σ stellen sich nun als Belegungen von Super-Trace-Variablen dar, bzw. in SMI kodiert als umfangreichere Ausdrücke aus Konjunktionen von Super-Trace-Variableneinzelbelegungen. Da diese Ausdrücke an vielen Transitionen der Automaten referenziert werden müssen, bietet sich zwecks Reduzierung des Umfangs des zu erzeugenden SMI-Codes eine einfache Abbildung auf boolsche Zeichenvariablen an, welche den Wahrheitswert des dazugehörigen Ausdruck erhalten. Dabei ist es ausreichend, nur die in Konflikten vorkommenden Super-Trace-Variablenbelegungen zu berücksichtigen. Wir führen entsprechend eine Menge $\Sigma_{TV^*} = \{\mathfrak{d}_0, \mathfrak{d}_1, \ldots, \mathfrak{d}_{|\Sigma_{TV^*}|}\}$ von SMI Variablen $\mathfrak{d}_k$ ein, welche in einem Block

nach Berechnung der Super-Trace-Variablenbelegungen den Wahrheitswert ihres zugehörigen Ausdrucks erhalten, $\mathfrak{d}_k := \left(t^{*\$}_{i_1} = k_1 \wedge t^{*\$}_{i_2} = k_2 \wedge \cdots \wedge t^{*\$}_{i_n} = k_n\right)$, wobei $\left\{t^{*\$}_{i_1}, t^{*\$}_{i_2}, \ldots, t^{*\$}_{i_n}\right\} \subseteq TV^*$ und die dazugehörigen Konstanten $k_1, k_2, \ldots, k_n$ den Belegungen entsprechen, die einer jeweiligen, in Abschnitt 4.7.2.2 auf Seite 79 beschriebenen kompakten Repräsentation $(\tilde{g}, \tilde{j})$ von Teilmengen von $G \times J$ entspricht und für mindestens einen Konflikt relevant ist.

6.3.4.2 Kodierung eines Automaten in SMI

Minimaler Automat Für die Kodierung des Zustands eines Automaten A_C nach Abschnitt 4.3.3 wird eine SMI-Variable s eingeführt, deren Belegung auf einen Zustand q aus $Q = \{q_0, q_1, \ldots, q_n\}$, abgebildet werden kann mit $(\sigma(s) = i) \mapsto q_i$. Der SMI-Automat enthält dann eine zweistufige Fallunterscheidung, wobei

1. in der in der ersten Stufe der Fall $\sigma(s) = i$ für alle $q_i \in Q$ zur Bestimmung des Ausgangszustands unterschieden wird, während
2. in der zweiten Stufe die neue Zustandsbelegung für s in Abhängigkeit der Belegung der boolschen Zeichenvariable δ zur Bestimmung des Zielzustands behandelt wird.

Desweiteren wird für in den initialen Zustand $q = q_{\omega_0}$ für ω-Automaten, bzw. den Zustand $q = fin$ für reguläre Automaten führende Transitionen (p, δ, q) eine Sonderbehandlung zur Reduzierung des erzeugten SMI-Codeumfangs durchgeführt. Da diese Transitionen die häufigsten sind, wird vermerkt, ob ein einer Transitionen (p, δ, q') mit $q' \neq q_0$ bzw. $q' \neq fin$ entsprechenden Zustandswechsel vorgenommen wurde. Falls nicht, wird am Ende der zweistufigen Fallunterscheidung die Zuweisung $s := 0$ bzw. $s := i_{fin}$ mit i_{fin} als der Zustandsnummer, die dem Zustand fin entspricht, durchgeführt. Transitionen zu nicht-akzeptierenden Zuständen werden durch eine Zuweisung $conflictobs := true$ behandelt, welche global einen aufgetretenen Konflikt anzeigt. Ein einfaches Beispiel ist in Abbildung 6.14 zu sehen.

Der Umfang des generierten SMI-Codes eines Automaten mit $n = |Q|$ Zuständen wächst mit $O(n \cdot |\Sigma|)$.

Determinisierter Automat Für den determinisierten Automaten A^*_C nach Abschnitt 4.7.2.1 wird analog zur dort bereits beschrieben operativen Semantik des als Zustandsgraphen zu interpretierenden Automaten A_C eine boolsche Variable b_j für jeden aktzeptierenden Zustandsknoten q_j mit $j > 0$ eingeführt, deren Wert genau dann auf *true* gesetzt wird, wenn die Zustandskodierungsvariable b_i einer der Vorgängerzustände $q_i, (q_i, \delta, q_j) \in T$ den Wert *true* besitzt und die Zeichenvariable δ der dazugehörigen Transition ebenfalls mit *true* belegt ist. Der SMI-Code enthält damit für jede Variable b_j die Zuweisung $b_j := b^*_j(\sigma)$, wobei b^*_j der Funktionskörper der gleichnamige Funktion aus Abschnitt 4.7.2.1 adaptiert auf die Belegung der Super-Trace-Variablen- und Zustandsbelegung der b_i ist. Für nicht-akzeptierende Zustände wird stellvertretend wieder die Variable *conflict* gesetzt, wobei der besseren Übersichtlichkeit halber für jeden nicht-akzeptierenden Zustand q_k eine eigene Zuweisung $conflict := conflict \vee b^*_k(\sigma)$ erzeugt wird.

Der Umfang des SMI-Codes bzgl. dieses Automaten wächst mit $O(n \cdot k)$, wobei k die maximale Anzahl von Vorgängerzuständen p eines Zustands q ist, für die $p < q$ gilt, womit k im Allgemeinen deutlich kleiner als $|\Sigma|$ ist.

6.3.4.3 Kreuzprodukte - Sequentialisierung in SMI

Kreuzprodukte werden in SMI durch Sequentialisierung der Schrittfunktionen der Automaten erreicht, d.h. ein SMI-Schritt umfasst in sequentieller Reihenfolge die

```
CODE
4  WHILE true DO
     DCASE
6      [] (p0=0)∧(T_x_old$=1):
         @[Sp0$ , 1]
8    DESAC
     DCASE
10     [] (p0=1)∧(T_x_old$=1):
         @[Sp0$ , 2]
12   DESAC
      DCASE
14     [] (p0=1)∧(T_x_old$=2):
         @[Sp0$ , 3]
16   DESAC
     ...
18   DCASE
       [] ((T_x_old$=1)):
20       @[x_old_ST$ , 1]
       [] ((T_x_old$=2)):
22       @[x_old_ST$ , 2]
     DESAC
24   @[conflict$ , false]
     @[sℜ0$ , false]
26   @[sℜ1$ , ((Sp2$=1)∧(x_old_ST$=2))∧((sℜ0=true))]
     @[conflict$ , conflict$∨(((Sp1$=1)∧(Sp2$=1))∧((sℜ1=true)))]
28   @[sω1$ , ((Sp0$=3)∧(x_old_ST$=2))∧((sω0=true))]
     @[sω2$ , ((Sp2$=1)∧(x_old_ST$=2))∧((sω1=true))]
30   @[conflict$ , conflict$∨(((Sp1$=1)∧(Sp2$=1))∧((sω2=true)))]
   OD
32 END
```

Abbildung 6.14: Beispiel der Kodierung eines Automaten in SMI

Berechnung der neuen Belegung für Variablen aus $\mathcal{S}_{absA_0}$, die Berechnung des neuen Zustands von $A_{C_\Re}$, sowie die Berechnung des neuen Zustands von A_{C_ω}, wobei für letztere jeweils die neue Belegung der Variablen $s_\Re$ und s_ω berechnet wird, welche diese Zustände kodieren.

Soweit die sequentialisierten Automaten unabhängig sind, kommt Sequentialisierung einer Parallelkomposition gleich, bei der alle Transitionskombinationen erlaubt sind. Dies entspräche einem uneingeschränkten Kreuzprodukt, in dem alle Transitionspaare vorhanden sind, wie dies zur Berechnung des Kreuzprodukts des regulären und des ω-Automaten nach Abschnitt 4.3.3.3 der Fall ist. Die Zustandsbelegungsberechnung dieser Automaten kann daher in beliebiger Reihenfolge erfolgen.

Die Eingaben dieser Automaten hingegen sind die Belegungen der Super-Trace-Variablen, welche ihrerseits aus Propositions- und Tracevariablenbelegungen berechnet werden, so daß diese Reihenfolge strikt einzuhalten ist.

Das spezielle Kreuzprodukt in Abschnitt 4.3.4 fordert zudem eine Entfernung der Transitionen, welche in einen nicht-akzeptierenden Zustand führen, d.h. die in SMI kodierten Automaten müssten Einfluß auf das bereits berechnete Geschehen in $\mathcal{S}_{absA_0}$ nehmen, was sich auf den ersten Blick als kniffelig erweist. Hierzu gibt es zwei Lösungsmöglichkeiten:

1. Der SMI-Code aus $\mathcal{S}_{absA_0}$ wird verschachtelt mit eingesetzten Funktionskörpern (*Inlining*) von Funktionen $p_i^{control} : \sigma \mapsto \mathbb{B}$ für alle Propositionsvariablen $P = \{p_1, p_2, \ldots, p_m\}$, welche einen Ersatzwert der Propositionsvariable p_i berechnet und in einer neuen Observer-Variablen $p_i'^{\$}$ schreibt durch $p_i'^{\$} := p_i^{control}(\sigma)$. Die Variable $p_i'^{\$}$ dient hernach als Substitut für die Propositionsvariable p_i in allen Ausdrücken in $\mathcal{S}_{absA_0}$. Die Funktion $p_i^{control}$ ist definiert

als

$$p_i^{control}(\sigma) \stackrel{\text{def}}{=} \left(\sigma(p_i) = \bigvee_{q_k \in Q \setminus F} b_k^*(\sigma) \right)$$

Anschaulich entspricht die Funktion einer Korrektur der zugehörigen Eingabebelegung im Falle eines andernfalls unvermeidbaren Konflikts. Die Berechnung kann dem Block $\mathcal{S}_{absA_0}$ nicht vorangestellt werden, da die Belegung σ zum Zeitpunkt der Abfrage der Belegung der Propositionsvariable p_i noch nicht existiert, jedoch aber relevant ist, da z.B. Trace-Variablenbelegungen des bisherigen Kontrollflusses den Guard-Ausdruck $\mathfrak{g}_p^\sigma$ erst determinieren, wie wir in Abschnitt 6.3.2.2 gesehen haben. Die Benutzung von expliziten Super-Trace-Variablen ist bei diesem Ansatz nur durch direkte, partielle Substitution der dazugehörigen Ausdrücke möglich, da die endgültige (vollständige) Belegung der Variablen in $\mathcal{S}_{absA_0}$ zu diesen Zeitpunkten noch nicht vorliegt.

2. Alternativ kann die im vorigen Abschnitt bereits vorgestellte Beobachtungsvariable *conflict* bei der im folgenden Abschnitt beschriebenen Übersetzung des SMI-Modells in das Format des Model-Checkers als so genannte Annahme (*Assumption*) verwendet werden, so daß die nachzuweisende Eigenschaft φ nur für diejenigen Läufe gelten muß, in denen immer $\sigma(conflict) = false$ gilt. Die entsprechende Behandlung dieser Beobachtungsvariable ist im folgenden Abschnitt beschrieben.

Lösung 2 birgt gegenüber der ersten Lösung keine Performanz-Nachteile beim späteren Model-Checking, wie experimentelle Untersuchungen gezeigt haben. Demgegenüber erlaubt es die Verwendung von expliziten Super-Trace-Variablen und Zeichenvariablen, welche den erzeugten SMI-Code übersichtlicher und kompakter gestalten. Daher ist der Lösung 2 der Vorzug gegeben worden.

Damit gestaltet sich die Abfolge der SMI-Blöcke der SMI-Schrittfunktion wie folgt:

1. Berechnung der neuen Belegung für Variablen aus $\mathcal{S}_{absA_0}$
2. Berechnung der Super-Trace-Variablenbelegung
3. Berechnung von boolschen Zeichenbelegungen
4. Berechnung des neuen Zustands von $A_{C_\Re}$, d.h. der Belegung der Variable(n), welche den Zustand von $A_{C_\Re}$ kodieren
5. Berechnung des neuen Zustands von A_{C_ω}, d.h. der Belegung der Variable(n), welche den Zustand von A_{C_ω} kodieren

Damit haben wir nun die Abstraktion A des Systems einschließlich der Verfeinerung des ω-Automaten und dessen Kombination mit der initialen Abstraktion A_0 konstruiert. als nächstes kann dieses abstrakte System dem Model-Checker zur Verifikation der Eigenschaft übergeben werden, wie im folgenden Abschnitt beschrieben.

6.4 Modellgenerierung und Model-Checking[14]

Das SMI-Programm $\mathcal{S}_{abs}$ liegt nun für den jeweiligen Verfeinerungsschritt vor und muß nun dem Model-Checker zur Überprüfung der Eigenschaft φ in Form einer

[14] Die in diesem Abschnitt geschilderten Abläufe sind vollständig außerhalb des Beitrages dieser Dissertationsschrift u.a. im Rahmen vorangegangener Arbeiten [Wit99, Bie99, Wit06] entstanden und sind aus Vollständigkeitsgründen mit einbezogen.

Kripke-Struktur $M = (S, S_0, T, L)$ vorgelegt werden. Sei

$$\Sigma_{SMI} \coloneqq \{\sigma \,|\forall v \in V_{state} : \exists! w \in \mathcal{D}_{Disc} : (v \mapsto w) \in \sigma\}$$

die Menge aller gültigen Belegungen zustandsbehafteter SMI-Variablen und φ die Eigenschaft in temporallogischer Form, dann wird die Kripke-Struktur vereinfacht dargestellt wie folgt erzeugt:

- $S \coloneqq \Sigma_{SMI}$
- $S_0 \coloneqq \{\sigma_0\}$
- $T \coloneqq \{(\sigma, \sigma') \,|\, \mathfrak{S}(\sigma) = \sigma'\}$
- $L \coloneqq \left\{\sigma \mapsto \left\{'v = w' \,\middle|\, v \in \mathcal{V}(\varphi) \wedge \sigma(v) = w\right\} \,|\sigma \in \Sigma_{SMI}\right\}$

Dabei ist $\mathfrak{S}$ die in Abschnitt 6.1.2 auf Seite 117 definierte Schrittfunktion des SMI-Programms und $\mathcal{V}(\varphi)$ die Menge der in φ vorkommenden Variablen. Diese Schritte spielen sich in der Implementierung wiederum mehrstufig ab und sind noch um die Abhandlung der Beobachtungsvariable *conflict* ergänzt.

6.4.1 Modellgenerierung

Zunächst erzeugt das Tool *smi2blifmv* aus einem diskreten SMI-Programm eine funktionale Darstellung aller einzelnen Bits [Bie01, Wit99], wobei hier sehr ähnlich zu der Berechnung der Super-Trace-Variablen Funktionen nach der Kontrollflußpfad-basierten *Eager Evaluation* Strategie vorgegangen wird. Bei dieser Berechnung kommen bereits symbolische OBDD-Techniken zum Einsatz. Das Ergebnis wird in dem BLIF-MV Format gespeichert [Kuk96], welches hierarchisch symbolische, sequentielle Systeme durch Variablen, Register, Leitungen und Relationstabellen beschreibt. Die Semantik dieses Formats entspricht einer Erweiterung der Standardsemantik synchroner monogetakteter digitaler Schaltungen.

6.4.2 Assumption-Observer

Nach der Modellgenerierung wird nun die Beobachtungsvariable *conflict* so in dem Modell auf BLIV-MV Ebene berücksichtigt, daß der Model-Checker nur Läufe beachtet, in denen dieser nie den Wert *true* erhält. Dazu wird zum Einen ein kleiner Automat $A_{conflict}$ wie in Abbildung 6.15 dargestellt mit dem Modell parallel komponiert und zum Anderen die CTL-Formel transformiert (Tool: *patchwork_mv*).

Der Automat $A_{conflict}$ geht in den Zustand q_1 und verweilt dort, wenn in dem Lauf irgendwann die Belegung $\sigma(conflict) = true$ beobachtet wurde, also ein Konflikt aufgetreten ist. Somit dient dieser Automat als „Gedächtnis" für das Auftreten eines solchen Ereignisses.

Die Eigenschaft φ muß so transformiert werden, daß alle Läufe, in denen $A_{conflict}$ im Zustand q_1 ist, nicht berücksichtigt werden. In der temporallogischen Formel φ werden daher alle Zustandsformeln f durch $f' \coloneqq (q_0 \implies f)$ substitutiert.

Dies bewirkt, daß alle Zustandsformeln in φ trivial erfüllt sind und mithin φ erfüllt ist, sobald ein Konflikt auftritt. Entsprechend können keine solchen Pfade vom Model-Checker generiert werden.

Einschränkung des Umgebungsverhaltens Dieser Mechanismus zum gezielten Ausschluß einer Menge von Pfaden zur Widerlegung einer Eigenschaft kann nicht nur wie geschildert zum Ausschluß von Konflikten eingesetzt werden, sondern

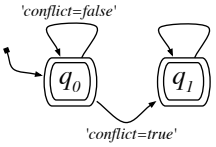


Abbildung 6.15: Automat $A_{conflict}$ zur Beobachtung der Belegung von *conflict*

darüber hinaus zur Spezifikation von Annahmen (*Assumptions*) über die Umgebung. Die Beobachtungsvariable bezieht sich dann auf Konjunktionen mehrerer Ausdrücke, die neben der Beobachtungsvariable des ω-Automaten auch Kombinationen, Wertebereiche oder Wertfolgen der Eingabesignale beschreiben, die ebenfalls für Gegenbeispiele keine Berücksichtigung finden sollen. Dieser Mechanismus wird in Kapitel 7 verwendet werden, um z.B. die Wertebereiche von Eingabevariablen des Typs *Real* einzuschränken. Wir werden dort eine Funktion *Assumption*
$Ass : e \mapsto b, b \in \mathbb{B}$ verwenden, die einen Automaten A_e mit zwei Zuständen für den übergebenen Ausdruck e in der oben beschriebenen Form einführt. Hier wird der Zustand q_1 eingenommen, sobald der Ausdruck e nicht wahr ist, die Assumption also verletzt wäre. Der Ausdruck $Ass(e)$ evaluiert zunächst zu *true* und wird permanent *false*, sobald die Eigenschaft e verletzt wurde.

6.4.3 Aufruf des Model-Checkers

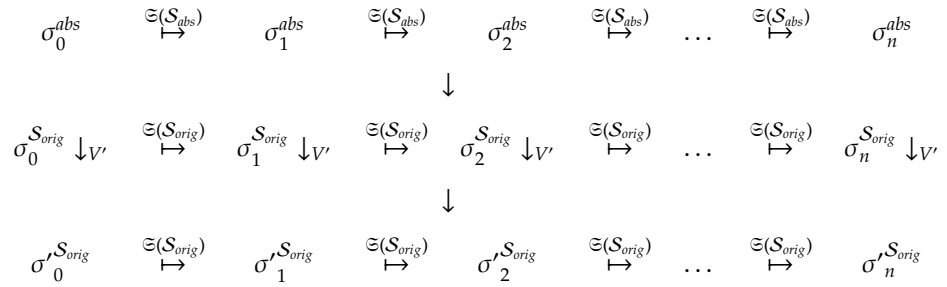
Die erzeugte BLIF-MV Datei wird daraufhin zusammen mit der CTL-Formel φ dem Model-Checker *vis* zur Anwendung eines wie in Abschnitt 2.4 beschriebenen Model-Check-Algorithmus übergeben. Für einfache Formeln der Form $\varphi = \mathbf{AG}\phi$ wird dabei ein sehr effizienter modifizierter Erreichbarkeitsalgorithmus [RGA+96, Gro96b] verwendet. Das Ergebnis ist entweder eine Bestätigung der Gültigkeit der Eigenschaft φ oder ein Gegenbeispiel in Form eines wie in Abschnitt 2.4.3 berechneten Simulationspfades, der auf das BLIF-MV Format aufsetzt.

Dieser wird vor der im folgenden Abschnitt dargelegten Pfadanalyse in einen SMI-Pfad der Länge n als Folge von Belegungen $\langle \sigma_n \rangle$ von Variablen aus V konvertiert.

6.5 Pfadanalyse

In diesem Abschnitt wird dargelegt, wie aus einem Lauf des abstrakten SMI-Programm $\mathcal{S}_{abs}$ in Form einer Sequenz von Belegungen entweder ein dazugehöriger Lauf in $\mathcal{S}_{orig}$ berechnet wird, oder, falls der Pfad für ungültig erklärt werden muß, eine Menge von Konflikten.

Im positiven Fall schließt sich an der Pfadanalyse noch ein Simulationsschritt zur Kontrolle und Vervollständigung der Belegungssequenz an, welche ebenfalls in diesem Abschnitt abgehandelt wird, womit sich in diesem Fall folgendes Bild mit $V' \subseteq V_{SMI}$ ergibt:



Der nächste Abschnitt behandelt, nach einigen einleitenden Worten zur Verwendung eines Solvers, die Realisierung der Lösungsfunktion $\mathcal{L}$ angewandt auf Belegungsfolgen in $\mathcal{S}_{abs}$. Dies beinhaltet ebenfalls die Integration des Solvers unter Gesichtspunkten der numerischen Stabilität. Im Anschluß wird die Detektion einer Menge minimaler Konflikte auf dieser Repräsentationsebene geschildert, sowie abschließend die Verwendung einer Simulation auf einer errechneten Belegungssequenz eines etwaig gültigen Laufs.

6.5.1 Integration und Verwendung des Solvers

Die in Abschnitt 4.3.2.2 definierte Berechnung $\mathcal{L}$ der einem abstrakten Pfad der Länge $k+1$ zugehörigen Folge von kontinuierlichen Zustandsräumen $(X_0, X_1, \ldots, X_k)$ bestehend aus $k+1$ Teilmengen von X ist in der Implementierung dergestalt abgewandelt, daß lediglich eine repräsentative Folge von Punkten $(x_0, x_1, \ldots, x_k), x_i \in X_i$ aus dem kontinuierlichen Zustandsraum ermittelt wird. Dies ist sowohl für den Nachweis der Existenz einer nicht-leeren Lösungsmenge $X_k \neq \emptyset$, als auch für die Darstellung eines konkretisierten Pfades völlig ausreichend. Ebenso ist der Nachweis der Nicht-Existenz einer solchen Lösung gleichbedeutend mit der Aussage, daß $X_k = \emptyset$.

Sofern die Berechnungen auf den kontinuierlichen Anteilen des Modells linearer Natur sind, kann diese Aufgabe von linearen Gleichungslösern wie z.B. lpsolve [BEN04] oder glpk [Mak03] übernommen werden. Bei Anwendung eines solchen Verfahrens sind die Ergebnisse jedoch mit einer gewissen Unsicherheit behaftet:

> „Existing LP-solvers do not claim to solve LPs to optimality. In fact, they come with hardly any guarantee. Feasible problems may be classified as infeasible and vice versa, a solution returned is not guaranteed to be feasible, and the objective value returned comes with no approximation guarantee. Some solvers guarantee that they will give an answer." [DFK+03]

Berechnete Lösungen können jederzeit durch Simulation validiert werden - nicht jedoch Aussagen über Nicht-Existenz einer Lösung. Für Zertifizierungszwecke ist daher ein überapproximierendes Verfahren wie z.B. Flowpipe-Approximation sicherer. Da das Verfahren der vorliegenden Arbeit jedoch die Abstraktionsmethodik in diesem Kontext als Schwerpunkt betrachtet und das Verfahren der numerischen Berechnung orthogonal zu dem Ansatz und daher austauschbar ist, wird im Rahmen dieser Arbeit mit diesem effizient einsetzbaren Verfahren Vorlieb genommen.

6.5.1.1 Implementierung der Lösungsfunktion

In einem Pfad bestehend aus einer Belegungssequenz $\left\langle \sigma_n^{abs} \right\rangle$ von Variablen aus $\mathcal{S}_{abs}$, wobei wir hier die in Abschnitt 4.2 auf Seite 37 definierte $\langle\rangle$-Notation für Folgen wiederaufgreifen, wird die Teilbelegung der Super-Trace-Variablen $\left\langle \sigma_n^{abs} \downarrow_{TV^*} \right\rangle$ für die Aufstellung eines Gleichungssystems genutzt. In der Implementierung operiert die

Lösungsfunktion daher auf Belegungssequenzen: $\mathcal{L}^{*\prime} : \left(\left\langle \sigma_n^{abs} \downarrow_{TV^*} \right\rangle, \sigma^*\right) \mapsto \mathbb{B}$. Dabei ist σ^* entweder die Initialbelegung $\sigma_0^{abs} \downarrow_{TV^*}$, welche den Zustand X_0 beschreibt, oder die alle Zustände in X beschreibende Belegung σ_X mit $\forall t^{*\$} \in TV^* : \sigma_X\left(t^{*\$}\right) = 0$. $\mathcal{L}^{*\prime}$

Nach den Gleichungen 6.2 und 6.3 auf Seite 139 wird für jede Belegung σ_i^{abs} der Belegungssequenz eine Konjunktion aller zugehörigen, durch Super-Trace-Variablenbelegungen kodierten Ausdrücke

$$e_i = \left(exprno_{M_1}\left(\sigma_i^{abs}\left(t_1^{*\$}\right)\right) \wedge exprno_{M_2}\left(\sigma_i^{abs}\left(t_2^{*\$}\right)\right) \wedge \cdots \wedge exprno_{M_m}\left(\sigma_i^{abs}\left(t_m^{*\$}\right)\right)\right) \tag{6.4}$$

für alle $t_j^{*\$} \in TV^* = \left\{t_1^{*\$}, t_2^{*\$}, \ldots, t_m^{*\$}\right\}$ erzeugt, so daß wir eine Sequenz von Ausdrücken $\langle e_n \rangle$ erhalten. Dabei bezeichnet die Funktion $exprno_M$, wie in Abschnitt 6.3.3.1 auf Seite 136 beschrieben, die Zuordnung der Super-Trace-Variablenbelegung zum dazugehörigen *GJ*-Paar, welches durch einen konjunktiven Ausdruck beschriebenen ist. Bevor wir aus dieser Sequenz eine große konjunktive Formel e erzeugen, müssen die Variablenvorkommen v der jeweiligen e_i durch zeitliche Instanzen v^i substituiert werden, wobei ungestrichene Vorkommen $v \in V_{Real}^{state}$ dem vorherigen Schritt $i-1$ zuzuordnen sind:

$$e_i' = e_i\left[V_{Real}^{input} V_{Real}^{\$state} : v^{\$} \backslash v^i\right]\left[V_{Real}^{state} : v \backslash v^{i-1}\right]$$

Variablen v^0 bilden dabei, sofern der Funktion $\mathcal{L}^{*\prime}$ als Anfangsbelegung $\sigma_0^{abs} \downarrow_{TV^*}$ vorgegeben wurde, die Initialbelegung der Variablen v und werden demgemäß durch die dazugehörigen Initialisierungskonstanten aus der SMI-Symboltabelle ersetzt. Damit ergibt sich die zu lösende Formel e aus

$$e = e_1' \wedge e_2' \wedge \cdots \wedge e_n' \tag{6.5}$$

Diese Formel wird durch die Funktion $\mathcal{L}^{*\prime}$ einem Solver zur Berechnung übergeben und das Ergebnis bzgl. der Lösbarkeit der Formel einschließlich einer Belegung $\sigma_{\mathcal{L}}$ aller Variablenvorkommen in e zurückgegeben.

Wurde e unter Berücksichtigung der gesamten Belegungssequenz $\left\langle \sigma_n^{abs} \right\rangle$ des vom Model-Checker gelieferten Pfads aufgestellt, so führt eine Lösung von e zur Terminierung das ω-CEGAR Verfahren wie in Algorithmus 6 auf Seite 64 bzw. Algorithmus 8 auf Seite 74 erläutert. In diesem Fall wird die vom Solver ermittelte Belegung $\sigma_{\mathcal{L}}$ in eine neue Belegungssequenz $\left\langle \sigma_n^{Real} \downarrow_{V_e} \right\rangle$ mit $V_e = V_{Real}^{input} \cup V_{Real}^{state}$ auf Variablen in $\mathcal{S}_{orig}$ übertragen mit $\sigma_i^{Real}(v) = \sigma_{\mathcal{L}}(v^i)$. Diese Sequenz beinhaltet lediglich Belegungen von Variablen aus V_{Real}^{state} und V_{Real}^{input} des Programms $\mathcal{S}_{orig}$, weswegen die Belegungen sowohl bzgl. der diskreten Variablen V_{Disc}, die sowohl in $\mathcal{S}_{orig}$, als auch in $\mathcal{S}_{abs}$ gleichermaßen vorhanden sind, als auch bzgl. der übrigen Variablen aus V_{Real} ergänzt werden müssen. Ersteres wird an dieser Stelle durch eine Fusion der Belegungssequenzen $\left\langle \sigma_n^{abs} \downarrow_{V_{Disc}} \right\rangle$ und $\left\langle \sigma_n^{Real} \downarrow_{V_e} \right\rangle$ durch paarweise Fusion der Belegungen gleicher Indizes zu einer neuen Sequenz $\left\langle \sigma_n^{\mathcal{S}_{orig}} \downarrow_{V_{Disc} \cup V_e} \right\rangle$ erreicht. $\left\langle \sigma_n^{\mathcal{S}_{orig}} \downarrow_{V_{disc} \cup V_e} \right\rangle$

Die Vervollständigung der Belegungssequenz bzgl. der übrigen Variablen aus V_{Real} geschieht später im in Abschnitt 6.5.3 auf Seite 150 geschilderten Simulationslauf unter Ausnutzung der Belegung der Variablen aus V_{Real}^{input}.

6.5.1.2 Integration des Solvers

Als Solver finden sowohl *lpsolve* [BEN04], als auch *glpk* [Mak03] Verwendung, da es sich während der Entwicklung bewährt hat, einen alternativen Gleichungslöser zwecks verbesserter Klassifikation etwaiger Inkonsistenzen in den Ergebnissen zu

ermöglichen. Somit kann z.B. bei Aufgabe eines Solvers aufgrund von eingetretener *Singularität* oder *schlechter Konditionierung* der jeweils andere Solver bemüht werden. Insbesondere bei größeren Modellen wächst die Gefahr eines falschen Unlösbar-Ergebnisses und kann durch redundante Überprüfung bestätigt oder widerlegt werden.

Die Anbindung der Solver geschieht durch eine Berechnung der benötigten Matrix aus einer großen Konjunktion in Form eines SMI-Ausdrucks. Jeder konjunktive Teilausdruck vorliegend als (Un-)Gleichung wird in einen Randbedingung (*Constraint*), repräsentiert durch eine Zeile in der Matrix, übersetzt. Da sämtliche konjunktiven Teilterme bereits durch die Funktion $\mathcal{N}$ in der Berechnung der Zuordnungstabellen in Abschnitt 6.3.3.1 auf Seite 136 normalisiert wurden, ist ein solcher Schritt dabei nicht erneut notwendig, es sind bereits alle Koeffizienten der Variablen zusammengefaßt.

Die auf dem Simplex-Verfahren basierenden Solver erwarten neben den Randbedingungen auch eine Optimierungsfunktion f_{opt}, welche zur numerischen Stabilisierung der Lösung verwendet wird, wie im folgenden Abschnitt beschrieben.

6.5.1.3 Vermeidung von Randlösungen

Notwendigkeit der ε-Variable Beide eingesetzten Gleichungslöser sind nicht in der Lage, echte Ungleichungen mit dem „<"-, oder „>"-Operator einzubeziehen. Ein einfacher Trick durch Einführung einer Variablen ε hilft jedoch weiter: Gleichungen der Form $x > c$ werden übersetzt in $x - \varepsilon \geq c$ mit $0 \leq \varepsilon \leq 1$, wobei ε additiv in die verwendete Optimierungsfunktion eingeht. Eine Lösung ist somit nur dann gültig, wenn der Solver eine Lösung für $\varepsilon > 0$ findet, $\varepsilon = 0$ also ausdrücklich ausgeschlossen wird. Neben der Berücksichtigung der echten Ungleichheitsoperatoren ist ein gewünschter positiver Seiteneffekt, daß der Lösungspunkt soweit möglich von allen kritischen durch echte Ungleichheitsbedingungen bedingten Rändern „weggezogen" wird, da der Solver versucht, den Wert für ε zu maximieren. Dies begünstigt numerisch stabilere Lösungen, die durch die spätere Validierung des Ergebnisses durch Simulation benötigt werden.

Einführung von Δ-Variablen War der geschilderte positive Seiteneffekt der ε-Variablen der numerischen Stabilität der Ergebnisse bereits sehr zuträglich, so ist er als Grund zur Einführung analoger Δ-Variablen zur Entschärfung der „≤"-, oder „≥"-Operatoren unerlässlich. Das Simplex-Verfahren liefert grundsätzlich vorzugsweise Lösungen, die auf den Eckpunkten der verschiedenen Constraints liegen und mithin für diese die Gleichheit einschließenden Operatoren genau die Lösungen, für die Gleichheit auch entsprechend gilt. Die konkrete Ergebnisdarstellung in Form einer begrenzt genauen Fließkommazahl liegt typischerweise minimal darunter oder darüber, so daß hierfür in der späteren validierenden Simulation ein anderes Resultat bzgl. der betroffenen Constraints ermittelt werden kann und damit das gesamte Ergebnis als nicht-bestätigbar abgelehnt werden muß.

Daher wird analog zur ε-Variablen für jedes entsprechende Constraint $x \geq c$ ersatzweise der Ausdruck $x - \Delta_i \geq c$ verwendet, wobei $0 \leq \Delta_i \leq 1$ und jedes Δ_i ebenfalls durch die Optimierungsfunktion maximiert wird. Es wird für jedes Constraint ein anderes Δ_i verwendet, da andernfalls ein ausnahmsweise nur durch exakte Gleichheit lösbarer Constraint ein $\Delta = 0$ erzwingen würde und mithin der gewünschte positive Effekt von Δ für die übrigen Constraints unnötig verloren ginge. Daher wird eine Menge $\{\Delta_0, \Delta_1, \ldots, \Delta_j\}$ entkoppelter Variablen eingeführt, die jedoch gleichermaßen durch die Optimierungsfunktion maximiert werden.

Um Umbuchungseffekte zugunsten ε und zu Ungunsten der Δ-Variablen zu verhindern, werden letztere in der Optimierungsfunktion stärker gewichtet und

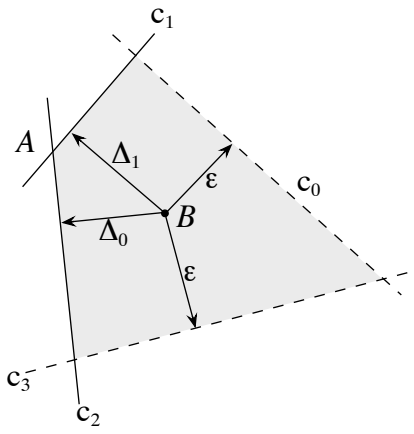


Abbildung 6.16: Verwendung von Δ-Variablen und ε zur Bestimmung möglichst unkritischer Lösungen. In diesem Beispiel dürfen die Randwerte der Bedingungen c_0 und c_3 von der Lösung nicht angenommen werden, im Gegensatz zu den Bedingungen c_1 und c_2, wo die Lösung auch auf den Randwerten liegen darf. Der Simplex-Algorithmus würde ohne Δ-Variablen den Punkt A berechnen. Durch die Δ-Variablen und der Gewichtungsfunktion wird hingegen eine Lösung in der numerisch idealen Mitte gefunden

gleichzeitig $j \cdot \varepsilon \geq \Delta_0 + \Delta_1 + \cdots + \Delta_j$ gefordert, um eine gewisse Balance zu erzwingen. Somit werden die Δ-Variablen nur soweit bevorzugt maximiert, wie ϵ immernoch größer oder gleich deren Durchschnittswert ist. Die zu maximierende Optimierungsfunktion für den Solver lautet $f_{opt} = \varepsilon + w * (\Delta_0 + \Delta_1 + \cdots + \Delta_j), w > 1$. In Abbildung 6.16 ist schematisch die Funktion von ε und den Δ-Variablen anhand eines Beispiels dargestellt.

Da die Δ-Variablen nicht wie die ε-Variable der Bestimmung der Lösbarkeit einer Formel dienen, sondern nur der numerischen Verbesserung der Lösung in Bezug auf die simulative Validierung, sind diese für die Konfliktbestimmung unerheblich und würden hier nur unnötig zur Komplexität der Formel beitragen. Daher werden sie nur bei der Berechnung der abschliessenden Lösung des Verfahrens eingesetzt, während bei der in Kapitel 6.5.2 beschriebenen Konfliktermittlung auf sie verzichtet wird.

Durch die Δ-Variablen kann jede Lösung bezüglich ihrer numerischen Güte quantitativ bewertet werden, da hier insbesondere die Information zu entnehmen ist, wieviele Ränder der gegebenen Randbedingungen durch die Lösung exakt eingenommen werden. Diese Information ist u.a. hilfreich, um die Ursache einer etwaigen Nichtbestätigung errechneter Lösungen durch die in Abschnitt 6.5.3 beschriebenen Simulation nachzuvollziehen.

6.5.2 Detektion von minimalen Konflikten

Die Detektion von minimalen Konflikten erfolgt nach der in Abschnitt 4.3.2.3 beschriebenen Vorgehensweise einschließlich der binären Suche nach Intervallgrenzen der Konflikte innerhalb des Suchfensters, sowie der in Abschnitt 4.7.2.3 vorgestellten Erweiterung auf Komponenten von Guard Sets und Jump Set Funktionen. Die Funktion r^* operiert dabei, wie die Lösungsfunktion $\mathcal{L}^{*\prime}$, auf Belegungssequenzen der Pfadlänge n und ermittelt minimale initiale und bzw. oder invariante

Konflikte der Länge k:

$$r^* : \langle \sigma_n \rangle \mapsto \left(\left\{ \left\langle \sigma_k^{ini} \downarrow_{TV^*} \right\rangle | k \leq n \right\}, \left\{ \left\langle \sigma_k^{inv} \downarrow_{TV^*} \right\rangle | k \leq n \right\} \right)$$

Die Abbildung erfolgt also von einem Pfad in $\mathcal{S}_{abs}$ in Form einer Belegungssequenz auf Konflikte in Form kürzerer, auf die Super-Trace-Variablen beschränkte Belegungssequenzen. Die Algorithmen 9 und 10 auf Seite 85 werden dabei einschließlich der in diesen nicht mitbeschriebenen binären Suche auf Intervallgrenzen von Konflikten für parametrierbare Fenstervergrößerungen > 1 auf den Belegungssequenzen durchgeführt, d.h. alle Paare $\left(\tilde{g}_i, \tilde{j}_i\right)$ in diesen Algorithmen sind Belegungen $\sigma_i \downarrow_{TV^*}$, welche in der Lösungsfunktion $\mathcal{L}^{*\prime}$, wie in Abschnitt 6.5.1.1 auf Seite 146 beschrieben, ausgewertet werden. Das Ergebnis sind, wie in Algorithmus 9 auf Seite 84 dargelegt, die Mengen der neu hinzugekommenen minimalen initialen und invarianten Konflikte in Form von Super-Trace-Variablenbelegunssequenzen.

Mitsamt ihrer Klassifikation als initial oder invariant werden diese in kompakter Repräsentation in eine Konflikt-Datenbankdatei wie in Abbildung 6.1 auf Seite 114 dargestellt geschrieben, so daß nachfolgende Iterationen des ω-CEGAR Verfahrens sich dieser Information bedienen und den ω-Automaten demgemäß verfeinern können.

Robustheit Aufgrund numerischer Instabilitäten im Verfahren des Solvers bei der Analyse der Gleichungssysteme ist nicht ausgeschlossen, daß bei der Isolation kürzester Konflikte numerische Probleme dergestalt auftreten, daß keine verlässliche Aussage bzgl. der (Un-)Lösbarkeit einer partiellen Belegungssequenz erhältlich ist. Es kann jedoch in diesem Fall von der problematischen Teilsequenz abgesehen werden, sofern noch andere Konflikte im Gesamtpfad vorhanden sind. Zur Erhöhung der Robustheit des ω-CEGAR Verfahrens gegenüber solchen numerischen Instabilitäten greift das Verfahren dann auf Konflikte zurück, die als solche vom Solver sicher klassifiziert werden können.

Erst wenn es bei Unlösbarkeit der Gesamtformel eines Pfades keinen Konflikt in der Sequenz gibt, der sicher isoliert werden kann, muß in Ermangelung genauer rechnender Solver das Verfahren ohne Ergebnis aufgeben, sofern es um die Zertifizierung einer Eigenschaft geht. Für eine mögliche Falsifikation besteht weiterhin die Möglichkeit, die fraglichen Teilsequenzen oder auch den gesamten Pfad als Konflikt zu behandeln und das Verfahren fortzusetzen, wenn beachtet wird, daß ein etwaiges nachfolgendes Ergebnis einer Nichterreichbarkeit nicht mehr belastbar ist.

6.5.3 Simulation der errechneten Lösung

Der hier beschriebene Schritt wird von akademisch eingesetzten Verfahren i.d.R. nicht durchgeführt, da der Nachweis der Existenz eines Pfades und die partielle Visualisierung hier im Allgemeinen ausreichend sind. Im industriellen Kontext sind jedoch die zusätzlichen Anforderungen:

1. Der Pfad muß die Belegung aller dem Benutzer bekannten Variablen beinhalten. Dies schließt auch die nicht-zustandsbehafteten Variablen aus $V_{Real}^{observer}$ ein.

2. Der Pfad muß in der Ausgangssprache, d.h. z.B. der zur Erstellung des Modells unterliegenden CASE-Tool Sprache simulierbar sein.

Daher wird abschließend vor der Präsentation des Ergebnispfads $\left\langle \sigma_n^{\mathcal{S}_{orig}} \downarrow_{V_{disc} \cup V_e} \right\rangle$ aus Abschnitt 6.5.1.1 auf Seite 146 dieser auf $\mathcal{S}_{orig}$ mit Hilfe der iterierten Ausfüh-

rung der Funktion $\Im$ durch das Simulationswerkzeug *smitrc_smisim* vollständig simuliert. Hierbei werden nur Belegungen der Variablen aus V^{input} berücksichtigt und das Simulationsergebnis in Form einer neuen Belegungssequenz $\left\langle \sigma'^{\mathcal{S}_{orig}}_{n} \right\rangle$ ermittelt. Fehlende, aufgrund der in Abschnitt 6.3.1.1 auf Seite 129 beschriebenen *Cone-of-Influence* Berechnung entfernte Eingabevariablen werden dabei mit beliebigen Vorgabewerten belegt.

$\left\langle \sigma'^{\mathcal{S}_{orig}}_{n} \right\rangle$

Ein Vergleich der Sequenzen $\left\langle \sigma^{\mathcal{S}_{orig}}_{n} \right\rangle$ und $\left\langle \sigma'^{\mathcal{S}_{orig}}_{n} \right\rangle$ bezüglich diskreter zustandsbehafteter Variablen zeigt hernach die durch Simulation bestätigte Gültigkeit des errechneten Pfades an. Da $\left\langle \sigma'^{\mathcal{S}_{orig}}_{n} \right\rangle$ im Gegensatz zu $\left\langle \sigma^{\mathcal{S}_{orig}}_{n} \right\rangle$ vollständige Belegungen in $\mathcal{S}_{orig}$ besitzt, wird dieser als endgültiges Ergebnis dem Benutzer präsentiert.

Eine Nichtbestätigung der errechneten Lösung ist in Abwesenheit von Implementierungsfehlern grundsätzlich auf numerische Probleme zurückzuführen, welche trotz der in Abschnitt 6.5.1.2 auf Seite 147 erläuterten Gegenmaßnahmen bei Lösungen auf Randwerten nach wie vor auftreten können. Insbesondere bei langen Pfaden summieren sich Darstellungsungenauigkeiten der Fließkommazahlen von anfänglichen Belegungen von kontinuierlichen Variablen, so daß am Ende der Belegungssequenz entscheidende Propositionsausdrücke andere als die in der abstrakten Simulation vom Model-Checker erratene Wahrheitswerte beinhalten können, mit entsprechen Konsequenzen für nachfolgende Belegungen. Ein solches Gegenbeispiel läßt sich bedauerlicherweise nicht simulieren, ist im akademischen Sinne jedoch dennoch gültig.

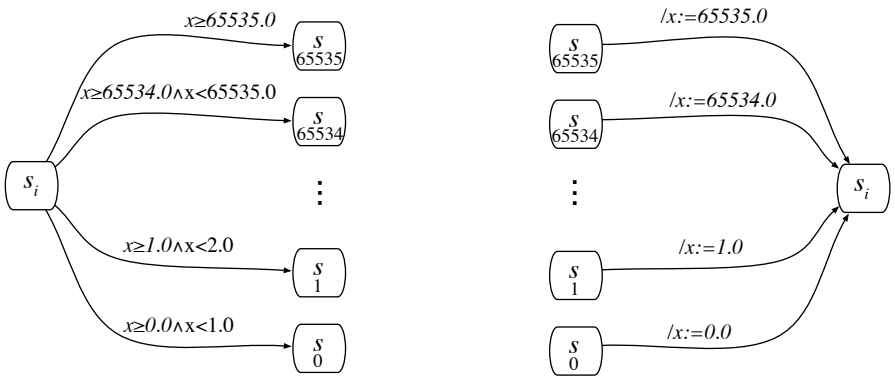
6.6 Behandlung von diskret-kontinuierlicher Casts

Wie bereits in Abschnitt 6.2.2.1 auf Seite 126 erwähnt, sind Typkonversionen (*Casts*) ausschließlich durch explizite Modellierung in SMI zu handhaben. Eine viel genutzte Typkonversion, die direkt das ω-CEGAR Abstraktionsverfahren betrifft, ist die Typkonvertierung zwischen den Typen *Integer* und *Real*, welche beide in beliebigen Ausdrücken vorkommen können.

Der Abstraktionsansatz kann diese nicht direkt unterstützen, weil außerhalb der gemeinsamen Transitionen als Synchronisationspunkte zwischen der diskreten und der kontinuierlichen Zustandswelt keinerlei Schnittstellen des Informationsaustauschs zwischen beiden existieren. So kann ein kontinuierlicher Zustand nur eine Auswahl möglicher ausgehender Transitionen eines diskreten Zustands einschränkend beschreiben, während diskrete Zustandstransitionen eine Auswahl an Konstanten kontinuierlichen Typs durch Jump Set Funktionen in den kontinuierlichen Zustandsraum transportieren können. Der Informationsaustausch kann somit nur wie in Abbildung 6.17 dargestellt erfolgen.

Für jeden möglichen Wert ist also eine explizite Transformation notwendig mit der Konsequenz, daß für große Bitbreiten der beteiligten Integer-Ausdrücke eine Vielzahl von neuen *GJ*-Paaren hinzukommen.

Die Alternative wäre die vollständige Einbeziehung der beteiligten Integer-Variablen in den Abstraktionsprozess, d.h. sie würden genauso behandelt werden, wie Variablen kontinuierlichen Typs. Da die verwendeten Solver sog. MILP-Solver sind (*Mixed Integer Linear Programing*), könnten diese mit Ausdrücken gemischt typisierter Variablen umgehen. Zu bedenken ist dabei jedoch, daß dabei die transitive Hülle der diskreten Ausdrücke vollständig miteinbezogen werden muß, da die betroffenen Variablen ihrerseits wieder mit anderen diskreten Variablen in direkter



(a) Konversion $\mathbb{R}$ nach $\mathbb{N}$ (b) Konversion $\mathbb{N}$ nach $\mathbb{R}$

Abbildung 6.17: Umsetzung von Typkonversionen zwischen $\mathbb{R}$ und $\mathbb{N}$ in Schritt-diskreten Hybriden Automaten am Beispiel einer 16-Bit Variablen

Wechselwirkung stehen, usw. Hierdurch muß u.U. nahezu das gesamte Modell mit abstrahiert werden, was jedoch nicht Sinn und Zweck dieses Verfahrens ist.

Eine Betrachtung industrieller Fallstudien zeigt, daß diese Form der Typkonversion zumeist dann benötigt wird, wenn hybride Systemkomponenten mit diskreten komponiert werden. In diesen Fällen wird die Typkonversion nicht etwa zur temporären Typ-fremden Speicherung einer Größe verwendet, sondern das gesamte Modul setzt seine Berechnungen auf diesen Größen in der gleichen diskreten oder kontinuierlichen Typwelt entsprechend fort. Damit wäre die nahezu vollständige Einbeziehung solcher Module der Regelfall.

Die explizite Typkonversion nach Abbildung 6.17 gestattet hingegen einen flexibleren Einsatz und ist daher implementiert worden. Da es sich um eine einmalige Vortransformation des Modells handelt, wäre diese im Rahmen der in Abschnitt 6.3.1 auf Seite 129 beschriebenen Transformationen anzusiedeln. Das entsprechende Typkonvertierungswerkzeug *smirealcaster* fügt explizite Konvertierungen zwischen beiden Typwelten ein, wobei unabhängig von der Konvertierungsrichtung nach Bestimmung des Vorzeichens sukzessive absteigende 2^{er} Potenzen von der zu konvertierenden absoluten Größe subtrahiert werden, soweit dies nicht zu negativen Werten führt, bis der Wert 0 erreicht ist. Mit der Funktion

$$transform(x,k) \stackrel{\text{def}}{=} \begin{cases} k + transform\left(x-k, \frac{k}{2}\right) & \text{gdw } x \geq k \\ transform\left(x, \frac{k}{2}\right) & \text{gdw } x < k \end{cases}$$

ergibt sich die Konvertierungsfunktion $cast_{\mathbb{R}\to\mathbb{Z}} : \mathbb{R} \to \mathbb{Z}$ mit $cast_{\mathbb{R}\to\mathbb{Z}}(x) \stackrel{\text{def}}{=} sign(x) \cdot transform(x, 2^b)$, wobei b die kleinste natürliche Zahl ist, für die $2^{b+1} > \max(abs(l), abs(u))$ mit *abs* als Funktion zur Berechnung des Absolutbetrages gilt, wenn die involvierte ganzzahlige Variable vom Typ $Integer_l^u$ ist. Analog dazu ist die Funktion $cast_{\mathbb{Z}\to\mathbb{R}} : \mathbb{Z} \to \mathbb{R}$ definiert. Diese Funktion wird von dem Werkzeug *smirealcaster* für jeden zu konvertierenden Ausdruck dem Wertebereich entsprechend oft abgerollt.

In Abbildung A.3 ist ein Beispiel für die Konvertierung auf SMI-Ebene dargestellt.

6.6.1 Komplexitätsbetrachtung

Wir erhalten für jede Konversion eines mit b Bits zu kodierenden ganzzahligen Wertebereichs eine Anzahl von b zusätzlichen Propositionen und ebensoviele zusätzliche Hilfsvariablen, welche jedoch in direkter Abhängigkeit zu den Propositionen stehen. Jede zu transformierende Konversion erhöht damit die Anzahl der potentiell vorhandenen GJ-Paare in der auf Seite 30 definierten Menge GJ des Systems um einen Faktor 2^b, was für eine Anzahl von k verwendeten Konversionen der Bitbreite b zu einem Faktor $2^{k \cdot b}$ führt. Dies zieht u.U. eine Erhöhung der Anzahl der Konflikte in ähnlicher Größenordnung nach sich, so daß die Prämissen des ω-Cegar Verfahrens von wenigen relevanten regelungstechnischen Gesetzen und wenigen Konflikten nicht mehr gegeben sind. Es ist im Allgemeinen daher nicht zu erwarten, daß größere, Konversionen beinhaltende Modelle mit dieser Technik verifiziert werden können. Dies schließt jedoch nicht aus, daß in Einzelfällen, wo die konvertierten Größen für die Berechnungen von untergeordneter Bedeutung sind, diese Technik eine Verifikation erst ermöglicht.

6.7 Selektiver Pfadfragmentausschluss

In Abschnitt 4.9.1 auf Seite 87 wurde bereits erwähnt, daß sich durch Modifikation des ω-Cegar Verfahrens das Infinite-State-Cegar Verfahren [CFH+03] nachahmen läßt und hierdurch belastbare Angaben zu den benötigten Iterationen des Verfeinerungsprozesses machen lassen. An dieser Stelle wird die Umsetzung dieser Modifikation auf der SMI-Ebene beschrieben.

Für einfache Schritt-diskrete Hybride Automaten nach Definition 9 auf Seite 29 sind Transitionssequenzen des Alphabets $\Sigma = T$ mit $T \subseteq Z \times Z$ auszuschließen, um die gleichen Pfade wie in [CFH+03] auszuschließen. Da sich die konkrete Repräsentation dieser Systeme auf der C-, bzw. SMI-Ebene darstellt und diese einem erweiterten Schritt-diskreten Automaten nach Definition 12 auf Seite 31 entsprechen, sind Transitionen durch Zustandspaare nicht mehr eindeutig spezifiziert. Daher wird in der Implementierung der Nachahmung des Infinite-State-Cegar Verfahrens das Alphabet $\Sigma = Z \times Z \times G \times J$ verwendet, d.h. wir verwenden Quadrupel, welche neben dem Ausgangs- und Zielzustand auch das GJ-Paar beinhalten.

6.7.1 Pfadanalyse

Technisch wird dies realisiert, indem die Pfadanalyse neben der Belegung der Super-Trace-Variablen auch die Belegung der übrigen diskreten Variablen $v \in V$ ergänzt, wobei V mit $V = V_{Disc}^{state} \backslash V_{\omega\text{-Automat}}$ alle diskreten Variablen ohne die Variablenmenge $V_{\omega\text{-Automat}}$ zur Kodierung des Zustands des ω-Automaten umfasst. Die in die Datenbank-Datei geschriebenen Konflikte sind nun also Sequenzen von Belegungen auf der Variablenmenge $V \cup TV^*$.

Die Lösungsfunktion bleibt hiervon unbeeinflusst, für die Konflikt-Minimierung wird Algorithmus 2 auf Seite 46 verwendet, d.h. es findet keine Berechnung von GJ-Komponenten statt, da Verallgemeinerung von Konflikten bei gezieltem Ausschluß einer Transistionssequenz keinen Sinn ergibt. Der Unterschied zum ω-Cegar Ablauf tritt erst in der Hinzufügung initialer und invarianter Konfliktmengen zur Datenbank-Datei zu Tage, da bei diesem Schritt die vollständige Belegungen oben genannter Variablen gespeichert werden.

6.7.2 Abstraktionsverfeinerung

Ebenso wie die Pfadanalyse ist der Prozess der Abstraktionsverfeinerung von dieser Modifikation nur marginal betroffen. Lediglich in der im Abschnitt 6.3.4.1 auf Seite 140 dokumentierten Berechnung der Menge der boolschen Zeichen des Konfliktalphabets werden die Zuweisungen um die Belegung der diskreten zustandsbehafteten Variablen ergänzt, d.h. die Zuweisungen zu den boolschen Zeichenvariablen werden zu

$\mathfrak{d}_k := \left(v_1 = l_1 \wedge v_2 = l_2 \wedge, \ldots, v_m = l_m \wedge t_{i_1}^{*\$} = k_1 \wedge t_{i_2}^{*\$} = k_2 \wedge \cdots \wedge t_{i_n}^{*\$} = k_n\right)$, wobei $\{v_1, v_2, \ldots, v_m\} = V_{Disc}^{state} \backslash V_{\omega\text{-Automat}}$ und, wie gehabt, $\left\{t_{i_1}^{*\$}, t_{i_2}^{*\$}, \ldots, t_{i_n}^{*\$}\right\} \subseteq TV^*$.

Da die Konflikte durch die modifizierte Pfadanalyse bereits als Belegungen eben dieser Variablen vorliegen, ist die nachfolgende Konstruktion des ω-Automaten davon unbeeinflusst und weitere Modifikationen daher nicht benötigt.

6.7.3 Variante: Berücksichtigung von parallelen Transitionen

Wie bereits erwähnt wird für den Ausschluß einer Transitionssequenz von einer Generalisierung durch Bestimmung von *GJ*-Teilmengensequenzen bei zu [CFH+03] identischer Umsetzung des Pfadausschlusses abgesehen. Dabei ist jedoch anzumerken, daß das in [CFH+03] beschriebene Verfahren auf Hybride Systeme der Definition 7 auf Seite 27 ausgeht, also insbesondere keine multiplen Transitionen zwischen zwei Zuständen erwartet. Insofern besteht in der Welt des Infinite-State-Cegar Verfahrens nicht die Notwendigkeit, versuchsweise auch parallele Transitionen zu überprüfen und möglicherweise mit auszuschließen, obwohl eine solche Überlegung naheliegend wäre. Die Verfeinerungstransformation würde diesbzgl. keine zusätzlichen Zustände benötigen, sondern lediglich einige zusätzliche Transitionen in die Transformation mit einschließen.

Die Verallgemeinerung der *GJ*-Sequenz durch Bestimmung relevanter *GJ*-Komponenten der auszuschließenden Konflikte entspricht einer solchen Erweiterung des Infinite-State-Cegar Verfahrens. Wir wollen den Autoren von [CFH+03] daher unterstellen, daß sie bei Einbeziehung real-existierender Modelle in ihren Experimenten das Verfahren entsprechend erweitern würden[15], und führen den Vergleich auch zu dieser Strategie durch Aktivierung der Generalisierung durch.

In dem folgenden Kapitel finden sich im Rahmen der experimentellen Untersuchungen zu beiden genannten Varianten des Vergleichsmodus Statistiken, die die Überlegungen aus Abschnitt 4.9.1 auf Seite 87 untermauern.

[15] Da die Autoren sich laut [FCJK05], wie in Abschnitt 4.9.1.1 erläutert, zuletzt um nicht-skalierbare Verfahren zur Selektion optimaler Konfliktausschlüsse durch Pfadfragmente bemüht haben, ist unklar, ob in naher Zukunft eine entsprechende Erweiterung wie hier geschildert zu erwarten wäre.

Kapitel 7

Experimentelle Ergebnisse

In diesem Kapitel werden experimentelle Ergebnisse des implementierten ω-CEGAR Verfahrens angewandt auf das in Abschnitt 5.2 auf Seite 104 vorgestellte Modell und anderen Schritt-diskreten linearen Hybriden Systemen vorgestellt. Dabei werden ebenfalls in begrenztem Umfang die Varianten des Verfahrens durch optional aktivierbare Erweiterungen wie z.B. der Verwendung von Teilmengensequenzen selektiv in die experimentellen Untersuchungen mit einbezogen.

7.1 Fallbeispielübersicht

In diesem Abschnitt werden die verwendeten Modelle kurz vorgestellt. Die zu verifizierenden Eigenschaften sind erst im folgenden Abschnitt beschrieben, da diese sich z.T. aus den Ergebnissen vorheriger Beweisaufgaben ableiten.

7.1.1 Autopilotsystem

Dies ist das in Kapitel 5.2 ausführlich vorgestellte Autopilotsystem zur Steuerung der Flugobjekthöhe in Abhängigkeit von Umgebungseinflüssen. Da die Durchführung von Verifikationsaufgaben mit der vorgestellten Umsetzung des Verfahrens lineare Systeme bedingen, werden wir von der Berechnung der Norm des Geschwindigkeitsvektors in Abbildung 5.9 abstrahieren, indem wir für den Ausgang des Operators *Vec3Norm* eine Eingangsvariable substituieren. Dies führt möglicherweise zu einer inkonsistenten Belegung der globalen Variablen *SpeedValid*, was für die durchgeführten Beweisaufgaben aufgrund der entbehrlichen Auswirkung dieser Variablen jedoch keine Probleme bereitet.

7.1.2 Fensterheber-System

Das Fensterheber-System ist ein von BMW entwickeltes, in ASCET modellieres System zur Steuerung der automatischen Fensterheber eines Fahrzeugs einschließlich Benutzerschnittstelle und Grobansteuerung der Elektromotoren. Das Modell hält eine interne Sollposition des Fensters vor, welche in Abhängigkeit der vom Benutzer sowohl durch die Fernbedienung, als auch durch die im Fahrzeug befindlichen Schalter getätigten Aktionen berechnet wird. Neben den Bedienelementen besitzt das Steuerungssystem Eingänge wie einen Einklemmschutzsensor zur Detektion von Hindernissen während des Schließens des Fensters und der gemessenen Fensterposition. Als Besonderheit beinhaltet dieses Modell eine Eingangsgröße dT, welche die seit der letzten Berechnung der Schrittfunktion verstrichene Zeit repräsentiert. Ähnlich wie im Autopilotsystem aus Abschnitt 5.2 wird diese Variable vom

Modell für die Berechnung der neuen Soll-Position des Fensters verwendet, jedoch diesmal nicht als Konstante, sondern als Eingang des Steuerungssystems, so daß das Modell auch bei unregelmäßiger Ausführung der Schrittfunktion eine korrekte Ansteuerung des Fensters gewährleisten kann. Hintergrund ist die Ausführung mehrere Steuerungssysteme auf einem Prozessor, welcher im Bedarfsfalle an höher priosierte Tasks vergeben werden muß. Daher kann kein periodischer Aufruf der Funktion des Steuerungssystems garantiert werden und das Betriebssystem stellt stattdessen eine akurate Variable dT zur Verfügung. Für die Durchführung von Verifikationsaufgaben auf diesem Modell bedeutet dies, daß beliebige Zeitabstände angenommen werden können.

Strukturell besteht das Modell aus zwei parallelen hierarchischen Zustandsautomaten, welche mit durch Datenflußdiagramme spezifizierten Zeitablaufs- und Sollzustandspositionsberechnungen interagieren. Die Berechnungen sind wesentlich durch Integratoren modeliert, in denen Zustandsvariablen kontinuierlichen Typs verstrichene Zeitspannen bzw. zurückgelegte Wegstrecken agglomerieren.

Das Modell ist von BMW nicht eingesetzt worden und wurde offenbar vor der endgültigen Fertigstellung verworfen. Bedauerlicherweise ist kein Anforderungskatalog für dieses Modell zu erhalten gewesen, weshalb sich mögliche Anforderungen im Bereich der Spekulation bewegen.

7.1.3 Fuelrate Controller System

Das *Fuelrate Controller System* ist in [The03] vorgestellt und dient der Einführung in die Modellierung typischer eingebetteter Systeme mit Simulink/Stateflow. Das Modell reguliert das Luft-Benzin-Gemisch eines Verbrennungsmotors durch Steuerung der Benzin-Einspritzung in den Verbrennungsraum. Durch die Meßwerte von Sensoren, die den Sauerstoffgehalt der Abgase und deren Druck misst, wird das aktuelle Verhältnis von Luft zu Benzin errechnet und gegebenenfalls durch Änderung der Benzinmengenzufuhr korrigiert. Neben diesen Sensoren stellen die Motorgeschwindigkeit und die Stellung des Gaspedals weitere Eingangsgrößen für das Fuelrate Controller System dar.

Das Modell verwendet einen Zustandsautomaten zur Steuerung verschiedener Betriebs- und Fehler-Modi. Desweiteren werden komplexere Berechnungen durch mehrere Kennlinienfelder zur Auswertung der Sensorik und Berechnung der Aktuatorik eingesetzt. Die Berechnungen sind grundsätzlich nichtlinear, jedoch führt die Verwendung von Kennlinienfelder zu linear approximierten Berechnungen.

Das Modell besitzt mit 256 Zuständen nur einen sehr kleinen diskreten Zustandsraum. Auf der anderen Seite finden durch die Kennlinienfelder eine Anzahl von $\sim$ 1500 Konstanten vom Typ *Real* in dem Modell Verwendung, was hier zu außerordenlich vielen möglichen Kombination und damit zu sehr vielen regelungstechnischen Gesetzen führt. Das Modell entspricht somit nicht den idealen Vorraussetzungen des ω-Cegar Verfahrens. Dennoch, bzw. gerade deshalb sind mit Hilfe dieses Modell hilfreiche Erkenntnisse bzgl. der Eignung des ω-Cegar Verfahrens zu ermitteln.

7.2 Ergebnisübersicht

In diesem Abschnitt wird zunächst ein tabellarischer Überblick über eine Menge ausgewählter Beweisaufgaben präsentiert. Im Anschluß daran werden die einzelnen Eigenschaften und deren Verifikationsergebnisse vorgestellt, wobei auf einige zur Veranschaulichung der typischen Verwendung der formalen Verifikation detailierter eingegangen wird. Abgesehen von Angaben im tabellarischen Überblick

werden in diesem Abschnitt keine Statistiken diskutiert. Dies erfolgt in nachfolgenden Abschnitten.

Die durchgeführten Verifikationsläufe wurden alle auf einer Workstation *Sun Fire V490* durchgeführt, welche das Betriebssystem SunOS 5.10 verwendet. Der Rechner besitzt 8GB RAM Haupspeicher und 4 CPUs, die mit 1350 MHz getaktet werden und 16MB Cache Speicher besitzen.

Im Folgenden werden die verifizierten Beweisaufgaben zu den Fallbeispielen vorgestellt.

Beweisaufgabe	$\mathfrak{n}$	$\lvert\Sigma\rvert/\lvert\Sigma_t\rvert/\lvert\Sigma_v\rvert$	$\lvert C_t\rvert/\lvert C_v\rvert$	$\lvert C_{min}\rvert$	$\#it$	$\lvert\pi\rvert$	$\lvert Q_{\mathfrak{R}}\rvert/\lvert Q_\omega\rvert$	Zeit
Autopilot System (*APS*), $\lvert Z\rvert = 2^{31}$								
φ_1^{APS}	5+17	47/3/58	2/36	3	29	13	2/53	13
φ_2^{APS}	5+17	145/3/209	2/137	17	102	17	2/268	75
φ_3^{APS}	5+17	43/2/44	1/26	0	23	13	2/31	9
φ_4^{APS}	5+18	77/4/65	3/39	0	49	515	3/60	44
φ_5^{APS}	5+18	46/4/42	3/25	0	31	$\nexists\pi$	3/25	8
φ_6^{APS}	4+8	57/1/85	0/57	1	58	9	2/65	16
φ_7^{APS}	4+8	49/1/64	0/44	1	42	10	2/40	10
φ_8^{APS}	4+8	59/1/79	0/54	1	55	10	2/53	15
φ_9^{APS}	4+8	91/1/163	0/167	39	120	12	2/167	81
φ_{10}^{APS}	4+8	120/1/222	0/258	44	203	14	2/290	720
φ_{11}^{APS}, abgebr.	4+8	132/1/255	0/1000	696	277	$\gtrdot$15	2/411	$> 1w$
φ_{12}^{APS}	5+11	86/4/99	3/51	0	32	515	2/77	44
φ_{13}^{APS}	5+11	44/4/35	3/21	0	11	$\nexists\pi$	2/19	4
φ_{14}^{APS}	5+18	41/2/43	1/24	0	21	13	33	16
φ_{15}^{APS}	3+8	134/1/191	0/521	307	185	16	2/258	600
Fensterheber System (*FHS*), $\lvert Z\rvert = 2^{26}$								
φ_1^{FHS}	2+3	92/12/66	14/75	5	63	10	5/39	21
φ_2^{FHS}	2+3	56/11/38	13/32	0	31	$\nexists\pi$	17	8
φ_3^{FHS}	2+3	183/13/117	12/1000	755	126	19	7/185	$29h$
φ_4^{FHS}, abgebr.	2+4	220/14/120	11/3805	3499	257	$\gtrdot$18	8/239	$> 2w$
φ_5^{FHS}	2+3	43/10/12	9/9	0	10	5	5/3	3
φ_6^{FHS}	2+3	46/7/15	6/12	0	8	10	3/3	2
φ_7^{FHS}	2+3	32/9/17	8/13	0	11	6	5/4	3
φ_8^{FHS}	2+3	52/8/29	7/20	0	12	9	4/9	4
Fuelrate Controller System (*FCS*), $\lvert Z\rvert = 2^8$								
φ_1^{FCS}, abgebr.	4+1	56/3/12	128/10	0	135	$\gtrdot$130	129/0	101
φ_2^{FCS}	4+1	14/3/7	2/6	0	7	2	3/0	2
φ_3^{FCS}	4+1	4/2/3	1/2	0	2	$\nexists\pi$	2/0	$20s$
φ_4^{FCS}, abgebr.	4+10	180/3/126	2/125	0	225	$\gtrdot$3	3/0	$11d$
φ_5^{FCS}	4+3	128/2/92	1/91	0	161	2	2/0	207
φ_6^{FCS}	3+3	349/3/192	16/191	0	329	17	17/0	393

Tabelle 7.1: Übersicht experimenteller Resultate. Dargestellt sind die Größe des diskreten Zustandsraums $|Z|$, die Anzahl der Dimensionen $\mathfrak{n}$ als Summe von Eingabe- und Zustandsvariablen innerhalb des Cone-of-Influence, die Anzahl unterschiedlicher *GJ* Paare $|\Sigma|$, sowie die Anzahlen $|\Sigma_t|$ und $|\Sigma_v|$ der Alphabete für den regulären Automaten $A_{C_{\mathfrak{R}}}$ und den ω-Automaten A_{C_ω} in getrennter Darstellung, welche unter Verwendung von Teilmengensequenzen die Anzahl der Teilmengen darstellen. Weiterhin sind $|C_t|/|C_v|$ die Anzahlen initialer und invarianter Konflikte, $|C_{min}|$ die Anzahl der in der Minimierung entdeckten Konflikte, $\#it$ die Anzahl der Iterationen des Verfeinerungsprozesses, $|\pi|$ die Länge des Pfades, welcher im Falle eines vorzeitigen Abbruchs des Prozesses das Zeichen $\gtrdot$ vorangestellt ist, $|Q_{\mathfrak{R}}|/|Q_\omega|$ die Anzahl der Zustände vor der Determiniserung und die letzte Angabe schließlich die benötigte Zeit des Verifikationsprozesses in Minuten, soweit nicht anders angegeben.

7.2.1 Verifikation des Autopilotsystems

Auf dem in Kapitel 5 vorgestellten Autopilotsystem gibt es, wie bereits erwähnt, keine offiziellen spezifizierten Eigenschaften, so daß die im Folgenden beschriebenen und formalisierten Beweisaufgaben intuitiv aus dem Verhaltensmodell abgeleitet sind. Da für einen der Beweise ein Gegenbeispiel von dem Wertebereich von Variablen des Typs *Integer* abhängt und eine erhöhte Gefahr von numerischen Problemen im Solver bei längeren Pfaden existiert, schränken wir für die Durchführung der Beweisaufgaben den Wertebereich dieses Typs auf das Interval $[-512; 511]$ ein.

7.2.1.1 Aktive Beeinflußung horizontaler Geschwindigkeitskomponenten

Eigenschaft φ_1^{APS} Aufgrund der vorgestellten Funktionalität des Autopilotsystems ist zu vermuten, daß das System aktiv nur den vertikalen Geschwindigkeitsvektor manipulieren soll, während die übrigen Komponenten des Geschwindigkeitssensors nur im Rahmen einer etwaigen, durch Wind verursachten Drift einer Korrektur unterzogen werden. Daraus leiten wir die Eigenschaft ab, daß bei völliger Windstille die x-und y-Komponenten sich nicht ändern dürfen, da andernfalls das Flugobjekt vom Kurs abkommen könnte. Wir beziehen uns für den Beweis exemplarisch auf die x-Komponente $v_x \coloneqq NewSpeed.\_F1$ und fordern $\Delta v_x = 0$ für durchgehende Windstille $\vec{v}_{wind} = 0$.

Diese Eigenschaft muß unter Einbeziehung des Umgebungsmodells spezifiziert werden, wobei die Eigenschaft nur gelten muß, wenn $automode = false$, da andernfalls neue, unkorrelierte Telemetrieinformationen über die Eingabevariablen *PtA* und *PtB* eingespeist werden. Da wir wissen, daß das Modell gültige Telemetrieinformationen erst nach 4 Schritten verarbeitet, wollen wir darüberhinaus fordern, daß $automode = false$ bereits für mindestens 4 Schritte gelten muß, bevor $\Delta v_x = 0$. Für $a \coloneqq (automode = false)$ und $NO_4 \coloneqq (a \wedge pre(a \wedge pre(a \wedge pre(a))))$ mit *pre* als Funktion zur expliziten Referenzierung des Wertes im vorherigen Schritt[1] lautet die nachzuweisende Eigenschaft damit in temporallogischer Form:

$$\mathbf{AG}(Ass(\underbrace{\vec{v}_{wind} = 0}_{Annahme}) \implies (\underbrace{NO_4 \implies (\Delta v_x = 0)}_{Eigenschaft}))$$

Die Zustandsformel $\vec{v}_{wind} = 0$ ist dabei die Annahme (*Assumption*), welche als Voraussetzung für die Einhaltung der Eigenschaft immer erfüllt sein muß. Wir machen daher von der in Abschnitt 6.4.2 auf Seite 144 eingeführten Funktion *Ass* zur automatischen Einführung eines Automaten Gebrauch, mit dessen Hilfe der Ausdruck $Ass(\vec{v}_{wind} = 0)$ solange zu *true* evaluiert, bis der Ausdruck $(\vec{v}_{wind} = 0)$ verletzt wird.

Aus Effizienzgründen kann in diesem Falle auch das Modell dergestalt modifiziert werden, daß anstelle der Eingabevariablen für $\vec{v}_{wind}$ ein konstanter Nullvektor Verwendung findet (*Freezing*).

Da aus numerischen Gründen die Verwendung von Gleichungen vermieden werden sollte, ersetzen wir die Forderung $\Delta v_x = 0$ durch die Forderung nach Einhaltung einer ϵ-Umgebung[2] für ein festes ϵ, d.h. wir verwenden $\Delta v_x - \epsilon < 0 \wedge \Delta v_x + \epsilon > 0$. Der Wert für Δv_x wird durch Hinzunahme einer entsprechenden Variablen im C-Programm explizit referenziert, ebenso wie der oben verwendete *pre*-Operator durch Einführung entsprechender Variablen umgesetzt wird. Obige Eigenschaft ist damit eine reine Sicherheitseigenschaft, d.h. die Formel spezifiziert lediglich die

[1] Wir bedienen uns hier der gleichnamigen SCADE/LUSTRE-Funktion gleicher Semantik.

[2] Dieses ϵ erfüllt eine andere Funktion als, und ist damit nicht zu verwechseln mit ε aus Abschnitt 6.5.1.3 auf Seite 148.

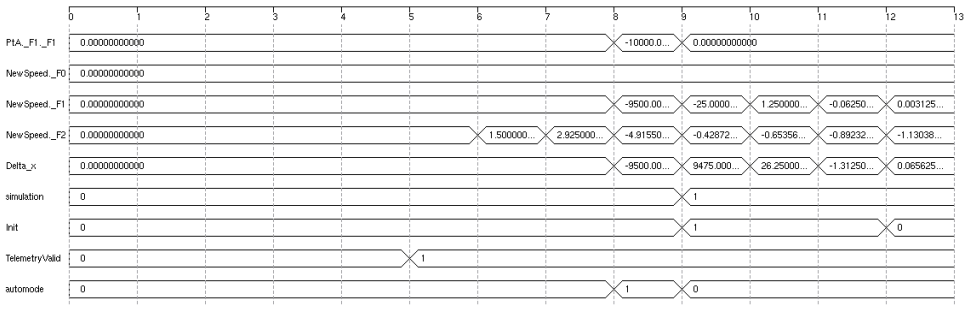


Abbildung 7.1: Gegenbeispiel für Beweis 1 als Sequenz von Belegungen (Auszug)

„guten" Systemzustände in Form einer Invarianten. Nach diesem Schema werden alle Eigenschaften spezifiziert.

Beweisergebnis Das Ergebnis in Form eines Gegenbeispiels ist in Abbildung 7.1 als Belegungssequenz einiger Variablen dargestellt, die insbesondere zeigen, wie sich die x-Komponente der Sollgeschwindigkeit als Variable *NewSpeed*.*_F*1 alias v_x ändert.

Eine Analyse der übrigen Variablenbelegungen zeigt, daß bevor in diesem Lauf durch $automode = false$ des Umgebungsmodell eingeschaltet wurde, extreme Positionsangaben gemessen wurden, welche zu einer entsprechend hohen intern berechneten Drift führten. Das Modell kompensiert eine Drift zu 5% im *Regulation*-Operator, wie bzgl. der Abbildung 5.12 auf Seite 109 beschrieben ist. Aufgrund der Verzögerungsoperatoren innerhalb dieser Rückkopplung stellt sich hier eine gedämpfte Schwingung ein, wie an den um den Nullpunkt schwingenden Werten der Variable *NewSpeed*.*_F*1 ersichtlich ist.

7.2.1.2 Abschwächung der Eigenschaft

Eigenschaft φ_2^{APS} Die beobachtete Schwingung ist aufgrund der extremen Telemetriemeßwerte und der sehr starken Dämpfung nicht als Fehler des Modells anzusehen, weswegen die Eigenschaft etwas abgeschwächt werden soll. Um stärkere Schwingungen zu Beginn der geschlossenen Simulationsläufe zu vermeiden, bzw. nicht zu beachten fordern wir, daß zum einen die Telemetriemeßwerte der x-Koordinate innerhalb eines engen Bereichs liegen sollen. Mit $PtA_x := PtA.\_F1.\_F1$ und $PtB_x := PtB.\_F1.\_F1$ benutzen wir als Annahme über die Umgebung $Tel_{restricted} := -100.0 \leq PtA_x \leq 100.0 \wedge -100.0 \leq PtB_x \leq 100.0$ wie in Abschnitt 6.4.2 auf Seite 144 beschrieben. Zum Anderen erhöhen wir die Anzahl der Schritte, die das Umgebungsmodell vor der Aktivierung der Eigenschaftseinhaltung bereits zugeschaltet sein muß durch $NO_6 := (a \wedge pre(a \wedge pre(a \wedge pre(a \wedge pre(a \wedge pre(a))))))$. Die Eigenschaft ist damit formal

$$\mathbf{AG}(Ass(\underbrace{\vec{v}_{wind} = 0 \wedge Tel_{restricted}}_{Annahme}) \implies (\underbrace{NO_6 \implies (\Delta v_x - \epsilon < 0 \wedge \Delta v_x + \epsilon > 0)}_{Eigenschaft}))$$

Beweisergebnis Das Ergebnis dieses Beweises ist als Widerlegungssequenz in Abbildung 7.2 zu sehen. Bei diesem Pfad ist das Signal *TelemetryValid* im 15. Schritt gerade gültig geworden, wodurch eine neue Positions- und Geschwindigkeitsberechnung angestoßen wird, die erneut eine Schwingung der Komponente v_x alias *NewSpeed*.*_F*1 bewirkt. Diese Schwingung ist auch im 17-ten Schritt noch so stark, daß abermals unsere Eigenschaft verletzt wird.

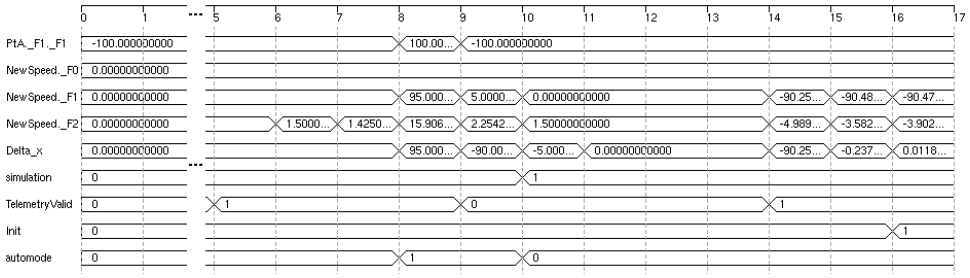


Abbildung 7.2: Gegenbeispiel zu abgeschwächter Eigenschaft

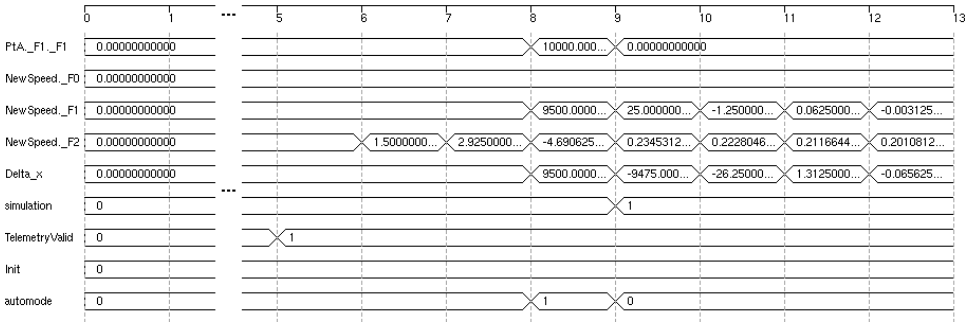


Abbildung 7.3: Gedämpfte Schwingung bei Änderung der Eigenschaftsspezifikation zur Verwendung von Geschwindigkeits-Halbräumen

7.2.1.3 Wechsel der Spezifikationsstrategie

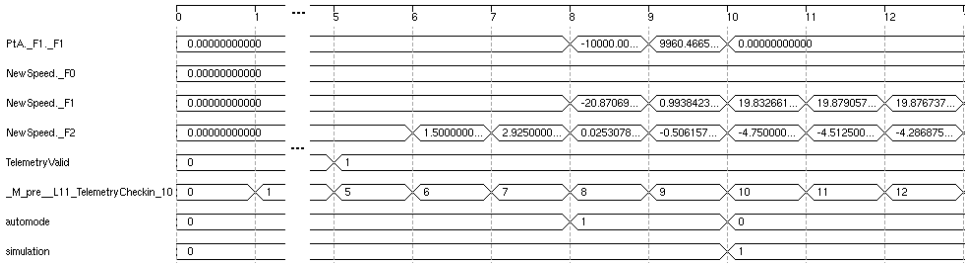
Eigenschaft φ_3^{APS} Um die gedämpfte Schwingung effektiv und effizient auszuklammern ändern wir die Eigenschaftsspezifikation dergestalt, daß wir anstelle der Verwendung von Δv_x eine willkürliche Konstante c_{v_x} o.B.d.A. zur Festlegung eines Geschwindigkeits-Halbraumes benutzen. Befindet sich die Geschwindigkeitskomponente v_x innerhalb eines Halbraumes, muß sie es in den folgenden Schritten ebenfalls sein. Eine Schwingung kann so sehr einfach ausgeschlossen werden, da hierdurch ein Halbraum nicht verlassen werden kann, wenn v_x mindestens zwei vorangehende Schritte innerhalb desselben verweilt. Eine Mindestverweildauer von einem Schritt ist nicht ausreichend, wie die folgende Formalisierung dieser Eigenschaft in der Verifikation offenbart:

$$\mathbf{AG}(Ass(\underbrace{\vec{v}_{wind} = 0}_{Annahme}) \implies \underbrace{((NO_4 \wedge pre(v_x) > c_{v_x}) \implies (v_x > c_{v_x}))}_{Eigenschaft})$$

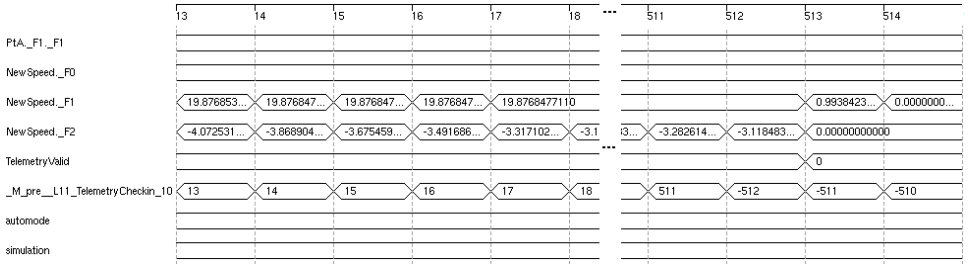
Mit $c_{v_x} := 0.0$ ergibt die Verifikation erneut einen 13-schrittigen Pfad mit einer Schwingung der v_x-Komponente, wie in Abbildung 7.3 dargestellt.

Eigenschaft φ_4^{APS} Nun erweitern wir die Eigenschaft wie oben erläutert, so daß die Geschwindigkeitskomponente v_x für zwei vorangehende Schritte im durch c_{v_x} spezifizierten Halbraum verweilen muß, bevor gefordert wird, daß dieser nicht mehr verlassen wird:

$$\mathbf{AG}(Ass(\underbrace{\vec{v}_{wind} = 0}_{Annahme}) \implies \underbrace{((NO_4 \wedge pre(v_x) > c_{v_x} \wedge pre(pre(v_x)) > c_{v_x}) \implies (v_x > c_{v_x}))}_{Eigenschaft})$$



(a) Pfad-Auszug Schritt 0-13



(b) Pfad-Auszug Schritt 13-515

Abbildung 7.4: Gegenbeispiel ohne Schwingungseffekte

Das Ergebnis ist ein in Abbildung 7.4 in Auszügen veranschaulichter Pfad der Länge 515. Zu beobachten ist hier, daß sich nach 514 Schritten der Wert der Variablen v_x ändert. Eine Analyse der übrigen Variablenbelegungen zeigt, daß dies durch einen Überlauf des Zählers in dem *TelemetryValid*-Operator verursacht wird, welcher bei Inkrementierung von 511 auf -512 wechselt. Der Überlauf bewirkt eine fälschlicherweise angenommene Ungültigkeit der Telemetriedaten, was wiederum Einfluß auf die Bestimmung der errechneten Position und damit auch auf die errechnete Geschwindigkeit hat. Der Pfad zeigt, daß die Geschwindigkeitskomponente v_x seit Schritt 17 des Systems konstant $v_x = 19.8768...$ betrug, bevor sie nun in Schritt 514 auf $v_x = 0.0$ gesetzt wird. Das Flugobjekt kommt damit erheblich vom Kurs ab und wird sein Ziel nach dieser Anzahl von Schritten, was bei der angebenen Schrittdauer von einer Sekunde einer Flugzeit von 8 Minuten und 35 Sekunden entspricht, nicht mehr erreichen. Das Gegenbeispiel deckt somit eindeutig einen Fehler im Modell auf, der einer Korrektur bedarf.

Ohne Einschränkung des Wertebereichs wäre, wie in Abschnitt 5.2.1 bereits diskutiert, ein Wertebereich von mindestens $[-32768; 32767]$ zu verwenden gewesen, was bei diesem Fehler eine Gegenbeispiellänge von 32770 Schritten und damit eine Kursabweichung nach 9 Stunden ergeben hätte. Die dazugehörige Pfadanalyse einhergehend mit großen Konjunktionsausdrücken für den Solver hätte neben numerischen Problemen sehr viel Berechnungszeit erfordert, weshalb wir zur Illustration des Fehlers den Wertebereich eingeschränkt haben.

7.2.1.4 Korrektur des Modells

Eigenschaft φ_5^{APS} Der Beweis aus dem vorigen Abschnitt soll nun mit einem korrigierten Modell wiederholt werden. Die Korrektur wird dabei durch eine bedingte Inkrementierung des Zählers errreicht, die ab einem Zählerstand von 511 nicht mehr durchgeführt wird.

Beweisergebnis Das vom ω-CEGAR Verfahren gelieferte Ergebnis bestätigt nun die Nicht-Erreichbarkeit der obigen Situation, d.h. der Geschwindigkeitshalbraum $v_x > c_{v_x}$ wird unter den geschilderten Randbedingungen nicht verlassen.

7.2.1.5 Kein Höhenverlust bei geringen vertikalen Winden

Eigenschaft φ_6^{APS} bis φ_{11}^{APS} Eine weitere Eigenschaft ist der Nachweis, ob bei vertikalen Winden v_{wind_z} innerhalb einer begrenzten Stärke das Flugobjekt an Höhe verlieren kann, obwohl kein Sinkflug angeordnet wurde ($MvtType \neq mvt\_approach$). Zur Formalisierung dieser Eigenschaft fordern wir, daß eine einmal überschrittene Flughöhe von $c_{alt} := 1000$m und einem erreichten Steigflug von über $c_{v_z} = 15\frac{\text{m}}{\text{s}}$ die Flughöhe c_{alt} nicht mehr unterschritten werden darf, sofern das Umgebungsmodell aktiviert ist ($automode = false$). Diese Vorbedingung wird dem C-Code als boolsche Variable mit der Zuweisung $obs := (obs \vee (v_z > c_{v_z} \wedge alt > c_{alt})) \wedge (automode = false)$ hinzugefügt und zur Formalisierung der Eigenschaft verwendet. Die Eigenschaft lautet damit unter Verwendung der Flughöhe des Systems $alt := (p_{1_z} + p_{2_z})/2$ mit p_{1_z} und p_{2_z} als die vertikalen Positionskomponenten der Telemetrie-Eingänge des *Pilot*-Operators in Abbildung 5.4 auf Seite 105:

$$\mathbf{AG}(Ass(\underbrace{c_{wind_{zmin}} < v_{wind_z} < c_{wind_{zmax}} \wedge MvtType \neq Approach}_{Annahme}) \implies (\underbrace{obs \implies (alt > c_{alt})}_{Eigenschaft}))$$

Die Eigenschaft wird mit verschiedenen Windstärken $-25.0 < v_{wind_z} < 25.0$ (φ_6^{APS}), $-15.0 < v_{wind_z} < 15.0$ (φ_7^{APS}), $-10.0 < v_{wind_z} < 10.0$ (φ_8^{APS}), $-7.5 < v_{wind_z} < 7.5$ (φ_9^{APS}), $-5.0 < v_{wind_z} < 5.0$ (φ_{10}^{APS}) und $-2.5 < v_{wind_z} < 2.5$ (φ_{11}^{APS}) untersucht. Bis $-5.0 < v_{wind_z} < 5.0$ werden Gegenbeispiele geliefert, die wie in Abbildung 7.5 dargestellt illustrieren, wie der Wind das Flugobjekt nach unten beschleunigt und dies nicht kompensiert wird. Bei schwachen Winden $-2.5 < v_{wind_z} < 2.5$ ist in aktzeptabler Zeit kein Gegenbeispiel zu berechnen und nur die Aussage möglich, daß ein Gegenbeispiel wenigstens 15 Schritte lang sein wird. Wir werden diesen Fall ausführlicher in Abschnitt 7.3 betrachten.

Eine Abschwächung der Eigenschaft dahingehend, daß ein Steigkommando als gegeben angenommen werden soll, führt ebenfalls zu einem ähnlichen Sinkflug selbst bei gesetzten Gültigkeitsbits der Geschwindigkeits-, Positions- und Telemetriedaten. Eine Analyse des Gegenbeispiels zeigt, daß aufgrund der sehr zurückhaltenden Anpassung der Sollgeschwindigkeit and die Theoretische Geschwindigkeit in dem *Regulation*-Operator in Abbildung 5.12 auf Seite 109 durch eine nur 5%-ige Angleichung etwaige Fallwinde der Stärke $> 2.5\frac{m}{s}$ nicht kompensiert werden können.

7.2.1.6 Weitere Beweise

An weiteren Beweisen für das Autopilot System wurden die folgenden durchgeführt:

- Die Eigenschaft φ_{12}^{APS} spezifiziert, daß eine einmal erreichte Höhe über 0.0 bei gegebenem Steigungskommando und eingehenden Telemetriedaten im Höhenbereich $[0.0; 100.0]$ bei ähnlich wie bei Beweis φ_4^{APS} unterdrückter Schwingung nicht unterschritten werden darf. Das Ergebnis ist ein ähnlicher Pfad wie bei φ_4^{APS}, der bei Überlauf des bereits erwähnten Zählers in eine solche Situation nach 514 Schritten führt. Nach der bereits bekannten Korrektur des Modells wird Beweisaufgabe φ_{13}^{APS} bei gleicher Eigenschaft erneut ausgeführt und die Eigenschaft als gültig bewiesen.

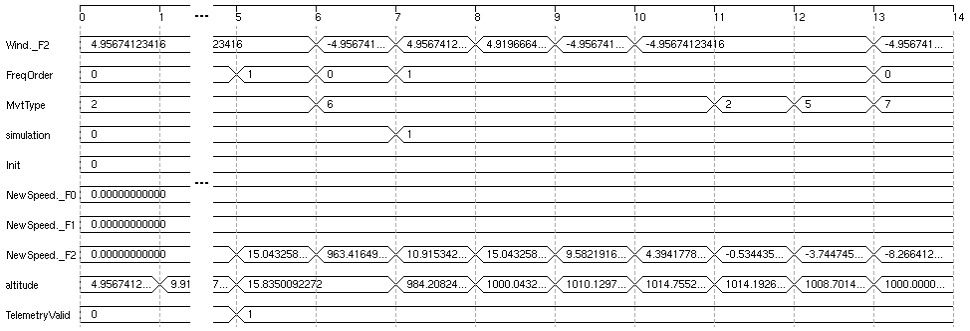


Abbildung 7.5: Sinkflug des Objekts bei vertikalen Winden. Die Konstanten in der Belegung der Variable *MvtType* sind der Abbildung 5.11 auf Seite 109 zu entnehmen

- Die Beweisaufgabe φ_{14}^{APS} entsprich φ_{3}^{APS}, mit dem Schwellwert $c_{v_x} := 10.0$. Diese ist für viele weiter Evaluierungen als Vergleich herangezogen worden.
- Die Beweisaufgabe φ_{15}^{APS} spezifiziert die Eigenschaft, daß bei einer Höhe unterhalb *smin* ein Steigflug erfolgen muß. Das Ergebnis ist erneut ein durch Schwingungseffekte belasteter Pfad.

7.2.2 Verifikation des Fensterheber Systems

Eigenschaft φ_1^{FHS} Auf dem Fensterheber System bieten sich Beweisaufgaben zur Erreichbarkeit bestimmter Sollwert-Positionen des Fensters an. Die Erreichbarkeit einer Sollwert-Position im vorgesehenen Wertebereich zwischen vorgegebenen Positionen pos_1 und pos_2 wird formalisiert durch

$$\mathbf{EF}(Ass(\underbrace{0.0 \leq dT \leq 1.0}_{Annahme}) \wedge (\underbrace{pos_1 \leq pos_{window} \leq pos_2}_{Eigenschaft}))$$

wobei wir annehmen, daß zwischen null und einer Sekunde zwischen zwei aufeinanderfolgende Aufrufe der Schrittfunktion verstreichen. Hier drücken wir dies durch eine Konjuktion der Annahme und der Eigenschaft aus, da wir nur an Pfaden interessiert sind, auf denen die Annahme durchgehend und die Eigenschaft in einem bestimmten Zeitpunkt gilt.

Für $pos_1 := 22.0$ und $pos_2 := 23.0$ liefert das Verfahren einen Pfad der Länge 10, in der nach einer kurzen Kalibrierungsphase eine Position $pos_{window} = 22.44$ erreicht wird.

Eigenschaft φ_2^{FHS} Als nächstes wird die Erreichbarkeit einer Sollwert-Position außerhalb des gültigen Wertebereichs überprüft:

$$\mathbf{EF}(Ass(\underbrace{0.0 \leq dT \leq 1.0}_{Annahme}) \wedge (\underbrace{pos_{window} > pos_{max}}_{Eigenschaft}))$$

Die Verifikation bestätigt hier die Unerreichbarkeit einer Position oberhalb des gültigen Wertebereichs.

Eigenschaft φ_3^{FHS} Eine komplexere Eigenschaft für dieses System ist die korrekte Reaktion auf ein Auslösen des Einklemmschutz-Sensors. Hier ist anzunehmen, daß eine erkannte Einklemm-Situation dazu führt, daß der Sollwert der Fensterposition nicht in Richtung Schließen verändert werden kann, damit keine Gefahr besteht, daß z.B. ein Kind seinen Kopf aus dem Fenster streckt, während das Fenster hochfährt. Um anfängliche Kalibrierungsphasen des Systems zu

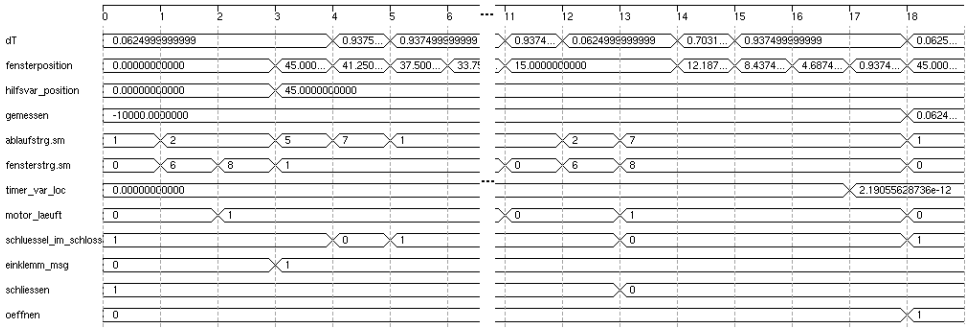


Abbildung 7.6: Gegenbeispiel zu Beweis φ_3^{FHS} als Sequenz partieller Belegungen

ignorieren, beginnen wir die Beobachtung unserer Eigenschaft erst nachdem die Sollwertposition des Fensters einmal eine nahezu ganz offene Fensterposition erreicht hat. Dies kann anhand eines durch die C-Anweisung $pos_{01} := pos_{01} \vee (0.0 < pos_{window} < 1.0)$ beschriebenen Zustandsbits, sowie einer durch die C-Anweisung $obstacle := (obstacle \vee einklemm_{msg}) \wedge einklemm_{msg}$ beschrieben Variablen, die eine Beibehaltung des Einklemmsignals anzeigt, wie folgt formalisiert werden:

$$\mathbf{AG}(Ass(\underbrace{0.0 < dT < 1.0}_{Annahme}) \implies \underbrace{(pos_{01} \wedge obstacle \implies pos_{window} < pos_{limit})}_{Eigenschaft})$$

Dabei legen wir willkürlich eine Grenzposition $pos_{limit} = 20.0$ fest, die nicht überschritten werden darf. Für den gültigen Wertebereich der Sollwertposition des Fensters gilt $0.0 \leq pos_{window} \leq 45.0$, womit der Wert für pos_{limit} knapp unterhalb der mittleren Position liegt.

Das Ergebnis ist ein 19-schrittiger Gegenbeispielpfad, welcher in Abbildung 7.6 zeigt, wie die Sollwertposition des Fensters (*fensterposition*) beginnend mit dem Wert 45.0 (geschlossen) langsam in eine Position zwischen 0.0 und 1.0 in Schritt 17 verschoben wird. Daraufhin wird der Sollwert im anschließenden Schritt wieder auf 45.0 gesetzt, obwohl die ganze Zeit der Sensor $einklemm_{msg}$ ein Hindernis angezeigt hat. Die Kombination der dargestellten Belegungen führt hier also zu einer Situation, welche obige Eigenschaft verletzt.

An weiteren Beweisen auf diesem Modell wurde die Erreichbarkeit einiger Zustände *state_i* bzw. Zustandskombinationen der Zustandsautomaten des Modells durch folgende Formel spezifiziert (Beweise φ_5^{FHS},φ_6^{FHS},φ_7^{FHS},φ_8^{FHS}):

$$\underbrace{\mathbf{EF}(state(fsm_1) = state\_i \wedge state(fsm_2) = state\_j)}_{Eigenschaft}$$

7.2.3 Verifikation des Fuelrate Controller Systems

Das Fuelrate Controller System beinhaltet, wie bereits angemerkt, mehrere große Kennlinienfelder und demgegenüber einen recht kleinen diskreten Zustandsraum. Insofern ist es für das ω-CEGAR Verfahren ein ungünstiges Modell, soll hier aber dennoch mitaufgeführt werden, um diesbzgl. Vergleiche durchführen zu können.

Auf diesem Modell werden die folgenden Beweise durchgeführt:

- Überprüfung der Erreichbarkeit eines Zustands namens $Multifail.FL4$ (φ_1^{FCS}), $Multifail.FL3$ (φ_2^{FCS})
- Überprüfung der Erreichbarkeit des Zustands *FuelDisabled.Overspeed* unter der Annahme, daß die gemessene Geschwindigkeit unter 600.0 liegt (φ_3^{FCS})

- Erreichbarkeit einer $fuelrate$ größer als ein konstanter Wert (φ_4^{FCS})
- Erreichbarkeit von $est\_air\_flow > 5.0 \wedge FuelDisabled.Overspeed$ (φ_5^{FCS})
- Erreichbarkeit von $tsim > 0.15 \wedge est\_air\_flow > 5.0$

Für alle Beweise wird aufgrund des durch Verwendung von großen Kennlinienfeldern verursachten großen Datenflußgraphen zur Berechnung der minimalen *GJ*-Repräsentationen nach Abschnitt 6.3.3 auf Seite 135 die Lazy Evaluation verwendet, d.h. einige Iterationen dienen lediglich der Verfeinerung der Abbildung und nicht des ω-Automaten.

7.3 Diskussion der Statistiken

Während wir bislang nur die Ergebnisse der Beweisaufgaben diskutiert haben, betrachten wir in diesem Abschnitt den Verlauf der Verifikationsprozesse unter statistischen Gesichtspunkten. Wir verwenden hierfür graphische Visualisierungen der einzelnen Verifikationsläufe durch Darstellungen der wichtigsten Kenngrößen in den einzelnen Iterationen der Prozesse. Die Kenngrößen sind dabei größtenteils die gleichen, die durch Tabelle 7.1 auf Seite 158 jeweils in ihrem Ergebnis aufgeführt sind.

Der Folgende Abschnitt führt die Diagramme und ihre Interpretation ein. Hernach werden die Erweiterungen der verallgemeinerter Konflikte, der Unterschied in der Evaluierungsstrategie der minimalen *GJ*-Repräsentationen, sowie der erweiterten Minimierung mit Erkennung weiterer Konflikte hinsichtlich der experimentellen Befunde betrachtet. Im Anschluss daran werden verschiedende typische Prozessverläufe diskutiert.

7.3.1 Bemerkungen zu Experimentaldaten

Im Folgenden werden nach der Vorstellung der Diagramme und deren Interpretation die Streuung der Statistiken anhand eines Beispiels betrachtet, um einen Eindruck über für die Vergleichbarkeit dieser Daten zu vermitteln.

7.3.1.1 Interpretation der Diagramme

Der iterative Verlauf der Abstraktionsverfeinerung läßt sich hinsichtlich der diesen begleitenden Kenngrößen wie Länge der vom Model-Checker ermittelten Pfade oder Größe des ω-Automaten durch ein geeignetes Diagramm darstellen, wie in Abbildung 7.7 dargestellt. Da solche Verlaufsdiagramme aufgrund ihrer umfassenden Präsentation nahezu aller Kenngrößen sehr informativ sind, werden sie für alle Beweise aufgeführt und sollen daher an dieser Stelle hinsichtlich der Herkunft und Bedeutung der einzelnen Kurven näher erkläutert werden.

Auf der Abszisse findet sich die Iterationszahl, während die Ordinate je nach Maßeinheit der jeweiligen Kurve als Mehrzweckskala dient. Die Legende im linken oberen Teil der Diagramme beschreibt die Zuordnung der Kurven zu den jeweiligen Größen:

- Die *Pfadlänge* bezeichnet $|\hat{\pi}|$, die Länge des vom Model-Checker errechneten Pfads $\hat{\pi}$ im abstrakten Modell. Da das CTL-Model-Checking-Verfahren immer einen der kürzesten Pfade ermittelt, ergibt sich ein monoton steigender Kurvenverlauf für diese Kenngröße. Ist also das Iterationsverfahren bei einer Pfadlänge von $|\hat{\pi}| = k$ angelangt, so kann es keinen Pfad $\hat{\pi}$ der Länge $|\hat{\pi}| < k$ geben, der vom Model-Checker in nachfolgenden Iterationen noch gefunden werden könnte.

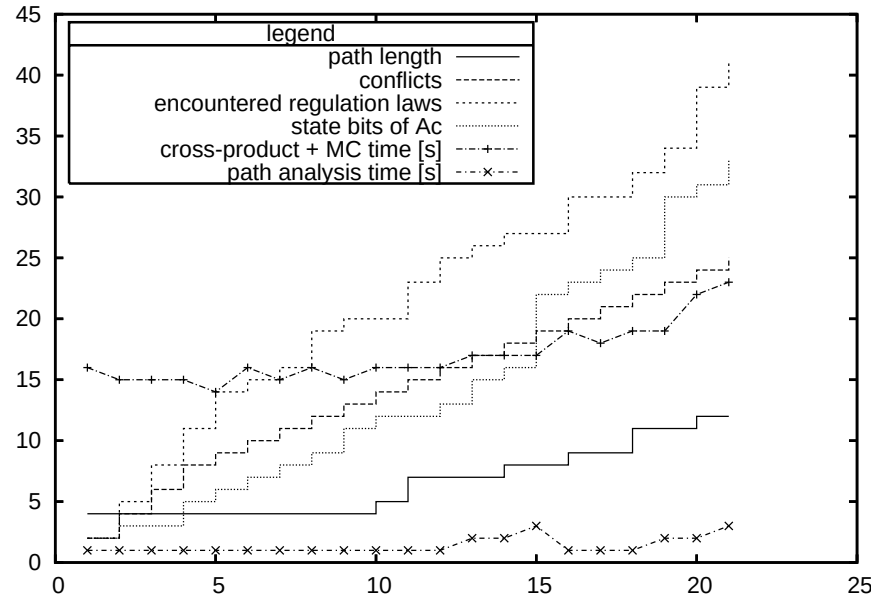


Abbildung 7.7: Verlauf eines Prozesses der iterativen Abstraktionsverfeinerung (Beweisaufgabe φ_{14}^{APS})

- Die *Konfliktanzahl* zeigt die agglomerierte Anzahl aller bislang aufgefundener Konflikte an. Demnach bedingt dies einen streng monoton steigenden Kurvenverlauf, da in jeder Iteration mindestens ein Konflikt gefunden werden muß, um ein Voranschreiten der Verfeinerung zu garantieren. In einem Schritt können durchaus mehrere Konflikte ermittelt werden, so daß dieser Wert überproportional zur Iterationszahl ansteigen kann. Einen Sonderfall stellt in dieser Hinsicht die Verwendung einer *Lazy Evaluation* Strategie bei der Auswertung der Trace-Variablenbelegungen dar, wie in Abschnitt 6.3.3.2 dargelegt. Hier kann der Wert durchaus für mehrere konsekutive Iterationen stagnieren.

- Die Anzahl der beobachteten *GJ*-Paare (*Regelungsgesetze*) ist eine weitere, agglomerierende und als solche monoton steigende Kenngröße, welche die Gesamtzahl der in allen bisherigen Iterationen bereits vorgekommenen, unterschiedlichen *GJ*-Paaren beziffert. Ein Vorkommen eines *GJ*-Paares ist dabei definiert durch das Vorkommen einer dazugehörigen Transition in der Zustandssequenz des Pfades $\hat{\pi}$, welche sich gemäß der in Abschnitt 4.3.2.1 auf Seite 40 geschilderten Pfadprojektion auf das *GJ*-Paar projizieren läßt.

- Die *Anzahl der Zustandsbits* des ω-Automaten A_C gibt die Größe desselben als Logarithmus der Anzahl der Zustände zur Basis 2 an, welches gerade der Anzahl der Zustandsbits entspricht, die zur Kodierung des ω-Automaten benötigt werden. Diese Kenngröße entwickelt sich im Allgemeinen nicht monoton, da durch Minimierungseffekte, wie in Abschnitt 4.7.1 beschrieben, der Automat durch Hinzunahme weiterer Konflikte auch kleiner werden kann.

- Die *Zeit*, die für die *Kreuzproduktbildung* und für das *Model-Checking* benötigt wird, ist in Sekunden angegeben und entspricht genau der für die in dem Abschnitt 6.4 erläuterten Generierung des symbolischen Modells für den Model-Checker, sowie der für das Model-Checking Verfahren benötig-

ten Zeit. Diese Wert wird erheblich von der Anzahl der Zustandsbits des ω-Automaten beeinflußt, jedoch in keinem fest quantifizierbaren Verhältnis. Die benötigte Zeit ist aufgrund der eingesetzten Heuristiken im Model-Checker starken Schwankungen unterworfen, die durchaus einen Faktor von 1000 in aufeinanderfolgenden Iterationen ausmachen können.

- Die *Pfadanalysezeit* ist schließlich die für die Konfliktisolation benötigte, ebenfalls in Sekunden angegebene Zeit der Anwendung der Algorithmen 9 und 10. Insbesondere schließt diese Zeit alle im Rahmen einer Pfadanalyse anfallenden Aufrufe des Solvers mit ein und kann für lange Pfade somit durchaus einen erheblichen Anteil ausmachen. Mitunter sind bei diesem Wert „Ausreißer" zu beobachten, welche sich auf numerische Instabilitäten im Solver zurückführen lassen. Dieser folgt offenbar in solchen Fällen einem aufwendigen Algorithmus, um dennoch ein Resultat zu erbringen.

Über die Beschreibung der Kurven hinaus bleibt noch anzumerken, das die Verlaufsdiagramme nur die Iterationen mit ungültigen Pfaden umfassen, d.h. die letzte, mit einem gültigen Ergebnis endende Iteration ist nicht im Diagramm enthalten.

7.3.1.2 Varianz

Während der Implementierung des Verfahrens ist auffällig geworden, daß bereits kleinste Änderungen in der Repräsentation der abstrakten Modellbeschreibung oder der Auswahlstrategie von Konflikten in einem nicht-konkretisierbaren Pfad zu deutlichen Variationen der quantitativen Kennzahlen der iterativen Abstraktionsverfeinerung führen. Als Extremfall sei hier die Umbenennung einer Modellvariable genannt, welche bereits zu einer geänderten Anzahl von Iterationen zur Durchführung ein- und desselben Beweises führen kann.

Dies liegt daran, daß die Konfliktmenge $C^*_\perp$, die in dem ω-Automaten ausgeschlossen werden muß um einen gültigen Pfad zu erreichen, bzw. die Unerreichbarkeit nachzuweisen, aus den folgenden Gründen nicht eindeutig ist:

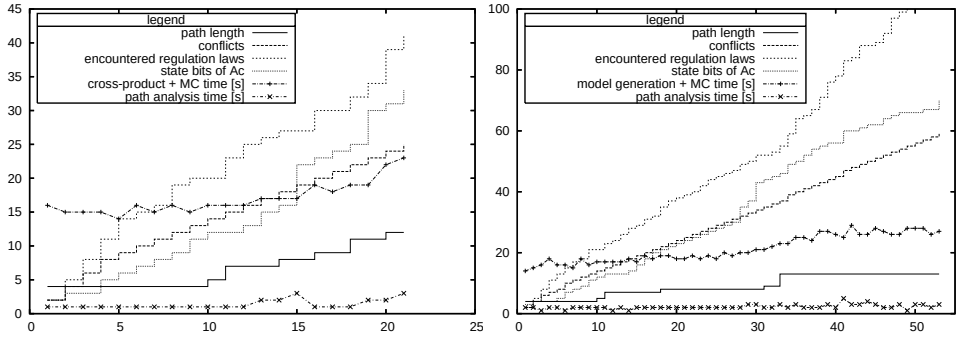
1. Bei Existenz gleich langer unterschiedlicher Pfade, von denen nur einige gültig sind und andere nicht, entzieht es sich der Kontrolle des Verfahrens, die Pfadwahl des Model-Checkers zu beeinflussen, soweit die Pfade noch in dem Konflikt-Automaten erlaubt sind. Eine glückliche Wahl eines gültigen Pfads entbindet das Verfahren entsprechend von der Berücksichtigung weiterer Konflikte in den noch nicht gewählten Pfaden.

2. Wie bereits in Abbildung 4.4 auf Seite 44 dargestellt gibt es verschiedene Alternativen bei der Konfliktselektion, um ungültige Pfade auszuschließen. Die genannten Selektionskritereien sind keineswegs eindeutig, wie z.B. in dem Fall von zwei kürzesten Konflikten gleicher Länge.

Selbst die minimale Konfliktmenge ist nicht eindeutig, da aufgrund von Punkt 2 zwei unterschiedliche, gleich große Konfliktmengen mit Konflikten jeweils gleicher Länge gleichermaßen alle ungültigen Pfade ausschließen können. Dies erklärt die hohen Abweichungen sowohl in der Iterationszahl, als auch insbesondere in der finalen Konfliktautomatengröße in den Statistiken.

Als Beispiel sei an dieser Stelle ein Fall angeführt, in dem lediglich die Nummerierung $exprno_M$ der Ausdrücke in den Zuordnungstabellen aus Abschnitt 6.3.3.1 geändert wurde[3]. Das Modell war damit semantisch äquivalent, weil die Numme-

[3] Die neue Nummerierung wurde im Rahmen der in Abschnitt 6.3.3.2 beschriebenen Einführung der *Lazy Evaluation* zwingend erforderlich, da für die Funktion $exprno_M$ trotz inkrementeller Erweiterung der Zustandstabellen bereits vergebene Indizes beibehalten werden mußten. Dieses Vergabeschema wurde für das *Eager Evaluation* Vorgehen hernach ebenfalls benutzt, im Gegensatz zu der vorher verwendeten Nummerierung nach lexikographischer Ordung der Ausdrücke.



(a) Nummerierung nach lexikographischer Ordnung

(b) Nummerierung nach Reihenfolge der Berechnung

Abbildung 7.8: Varianz im Verlauf der iterativen Verfeinerung bei Änderung der Nummerierung $exprno_M$ der Elemente der Zuordnungstabellen

rierung der Ausdrücke keine semantischen Auswirkungen hat und lediglich der Kodierung der vom ω-Automaten zu beobachtenden Zeichen des Alphabets dient. Die Konsequenzen für die Verläufe der iterativen Verfeinerungsverfahrens sind in Abbildung 7.8 gegenübergestellt, wobei auf dem *Flight*-System der gleiche Beweis φ_{14}^{APS} ausgeführt wurde.

Der Verlauf zeigt, daß mehr als doppelt soviele Iterationen nötig wahren, um den 13-schrittigen Pfad zu ermitteln. Es waren entsprechend mehr Konflikte auszuschließen, welche zu einem entsprechend größeren ω-Automaten führten und die analysierten Pfade haben zusammen eine größere Anzahl von unterschiedlichen *GJ*-Paaren beinhaltet. Vor dem Hintergrund dieser möglichen Varianz der quantitativen Kennzahlen werden wir von der Untersuchung von Parametrierungen mit marginaler Auswirkung absehen. Für belastbare Aussagen wäre eine entsprechend große Anzahl von Experimenten auf einer Vielzahl von geeigneten Fallbeispielen nötig, welche dem Autor jedoch nicht zur Vefügung stehen.

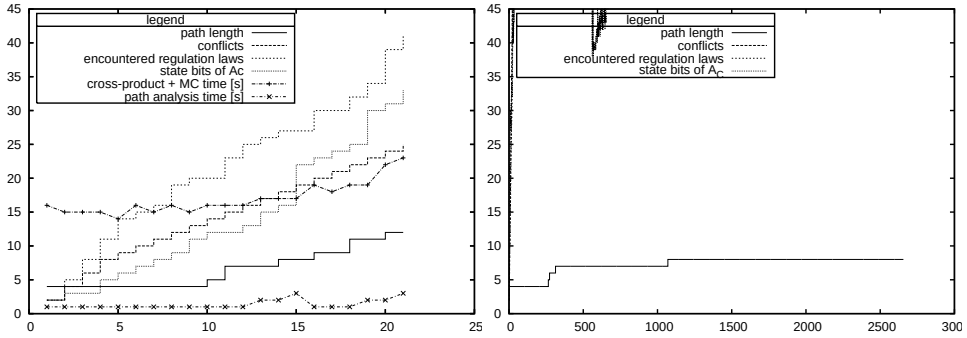
Die statistischen Kenngrößen sind daher vorbehaltlich möglichen, durch minimale Änderungen motivierten Abweichungen größeren Ausmasses zu betrachten.

7.3.2 Teilmengensequenzenerweiterung

In diesem Abschnitt soll untersucht werden, inwieweit die Einführung von Teilmengensequenzen von Konflikten nach der in Abschnitt 4.7.2.2 beschriebenen Systematik das ω-CEGAR Verfahren beschleunigt. Dazu wird die Verwendung von Teilmengensequenzen deaktiviert und der Beweis erneut durchgeführt. In Gegenüberstellung mit der aktivierten Teilmengensequenzoption des Verfahrens ist das Ergebnis einer Deaktivierung in Abbildung 7.9 dargestellt.

Der Verlauf der Kurven ist eindeutig: Während bei Verwendung von Teilmengensequenzen nur 23 Iterationen benötigt werden, um den gültigen Fehlerpfad der Länge 13 zu finden, sind ohne diese Erweiterung über 2700 Iterationen notwendig. Es können leider keine endgültigen Zahlen angegeben werden, da das Verfahren bei den eingesetzten Rechnern in einigen Wochen nur bis zu diesem Punkt voranschreitet, an dem Pfade der Länge 8 untersucht werden. Bei Extrapolation des Verlaufs ist jedoch zu vermuten, daß wenigstens die doppelte Anzahl an Iterationen nötig sein wird, um den gesuchten Pfad zu erreichen.

Dies zeigt die enorme Beschleunigung des Verfahrens durch die Verwendung von Teilmengensequenzen. Ohne diese wäre das Verfahren auf dem vorliegenden System praktisch nicht anwendbar.



(a) Aktivierte Teilmengensequenzverwendung (b) Deaktivierte Teilmengensequenzverwendung, unvollständig

Abbildung 7.9: Gegenüberstellung von aktivierter und deaktivierter Verwendung von Teilmengensequenzen

Automat	SMI Zeilen	Zustandsbits	t_{MG}	t_{MC}	$t_{MG} + t_{MC}$
minimal	42.477.852	14	2.5 h	20 s	2.5 h
determinisiert	25.222.647	465	90 s	180 s	270 s

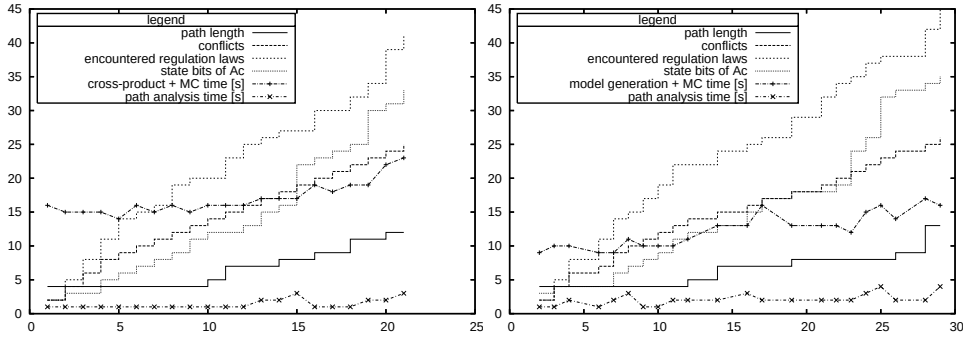
Tabelle 7.2: Vergleich von Zeiten für Modellgenerierung t_{MG} und Model-Checking t_{MC} bei unterschiedlich verwendeter Kodierung des ω-Automaten bei einem Umfang von ca. 8500 Konflikten. Die Zustandsbits sind aus dem regulären Automaten für initiale Konflikte und dem ω-Automaten für invariante Konflikte bereits zusammenaddiert

Anmerkung In diesem Diagramm wurde die für die Modellgenerierung und dem Model-Checker benötigte Zeit nicht angegeben. Anzunehmen gewesen wäre, daß insbesondere die Model-Checker Zeit t_{MC} deutlich langsamer ansteigt, da bei Nicht-Verwendung von Teilmengensequenzen für den ω-Automat keine Determinisierung nach Abschnitt 4.7.2.1 benötigt wird und demnach der ω-Automat bei gleicher Anzahl und Länge von Konflikten bezüglich der Anzahl seiner Zustände Q_C um einen Faktor $\frac{2^{|Q_{\mathfrak{R}}|+|Q_\omega|}}{|Q_{\mathfrak{R}}|+|Q_\omega|}$ kleiner ist, als der determinisierte Automat. Es war jedoch zu beobachten, daß das zur Modellgenerierung verwendete, in Abschnitt 6.4.1 vorgestellte Werkzeug *smi2blifmv*, welches zur Generierung der funktionalen Darstellung aller einzelnen Bits eingesetzt wird, erheblich mehr Rechenzeit t_{MG} für diese Aufgabe bei Verwendung des kleineren Automaten benötigt.

Ursächlich hierfür ist vermutlich, daß der Zustandsraum hier, wie in Abschnitt 6.3.4.2 beschrieben, logarithmisch kodiert wird und die Funktion zur Berechnung der Schrittfunktion für ein einzelnes Bit deutlich komplexer ist. Demgegenüber ist bei dem nach Abschnitt 4.7.2.1 determinisierten Automaten mit der in Abschnitt 6.3.4.2 beschrieben Kodierung der Zustandsübergangsfunktion einzelner Zustandsbits deutlich einfacher, da nur die Bits der Vorgängerzustände in einfachen Abhängigkeiten in der Funktion benötigt werden. Die Aufstellung der Zustandsübergangsfunktion der jeweiligen Bits erfolgt gerade in der Modellgennerierung, so daß ein erhöhter Aufwand in diesem Schritt bei logarithmischer Kodierung nachvollzogen werden kann.

In Tabelle 7.2 ist der Effekt an einem sehr großen ω-Automaten[4] beispielhaft wiedergegeben. Auch der Umfang des SMI-Codes, der für die Kodierung des Automaten benötigt wird, ist im Falle logarithmischer Kodierung deutlich größer, weil

[4]Dieser Vergleich wurde aufgrund des deutlicheren Unterschieds einem Verifikationsprozeß aus Abschnitt 7.4 entnommen.



(a) Eager Evaluation (b) Lazy Evaluation

Abbildung 7.10: Gegenüberstellung *Eager Evaluation* und *Lazy Evaluation* Ansatz

erheblich mehr Transitionen explizit beschrieben werden müssen, wie bereits die Umfangsabschätzungen des generierten SMI-Codes in Abschnitt 6.3.4.2 verdeutlichen. Aufgrund dieses Effekts wird in allen Experimenten immer der determinisierte, nicht-logarithmisch kodierte Automat verwendet und dementsprechend die Zeit für Modellgenerierung und Model-Checking nicht mit aufgeführt, wenn der logarithmisch kodierte Automat explizit hätte verwendet werden müssen, wie dies in dem Beispiel in Abbildung 7.9 der Fall ist.

7.3.3 Eager vs. Lazy Evaluation

Die in Abschnitt 6.3.3.2 geschilderte Verwendung eines Lazy Evaluation Ansatzes, d.h. der Auswertung neuer Belegungen der Tracevariablen führt zu zusätzlichen Verfeinerungsschritten, in denen nicht der ω-Automat, sondern nur das von diesem verwendete Alphabet verfeinert wird. Hier soll exemplarisch untersucht werden, wie ein Verfeinerungsprozess sich bei Einsatz dieser Strategie gegenüber der Eager Evaluation Strategie ändert.

Eine direkte Gegenüberstellung von entsprechenden Verifikationsläufen ist in Abbildung 7.10 zu finden. Glücklicherweise ist die Varianz bei diesem Experiment gering und die Iterationszahlen sind insofern identisch, als daß genau 7 Iterationen bei der *Lazy Evaluation* Strategie nur der Verfeinerung des Alphabets im Rahmen neu aufgetretener Belegungen der Trace-Variablen dienen. In dem Diagramm sind diese 7 Schritte an den Stellen erkennbar, an denen keine neuen Konflikte ermittelt worden sind[5], also den Iterationsschritten 1, 5, 13, 15, 18, 20 und 27. Darüberhinaus hat es offenbar keine nachteiligen Konsequenzen gehabt, so daß diese Strategie zwar mehr Iterationen benötigt, aber dennoch in akzeptabler Iterationszahl zum Ziel führt. Somit sollte auch im Falle großer Datenflußgraphen, bei denen eine Eager Evaluation Strategie nicht angewendet werden kann, eine Anwendbarkeit der Implementierung des ω-CEGAR Verfahrens gewährleistet sein.

7.3.4 Erweiterte Minimierung

An dieser Stelle soll der Effekt der um den Lernprozeß erweiterten Minimierung nach Abschnitt 4.7.1 auf Seite 70 diskutiert werden. Die in Tabelle 7.1 auf Seite 158

[5]Streng genommen müßte in genau diesen Schritten sich auch die Anzahl der aufgetretenen *GJ*-Paare erhöhen, jedoch wird die Statistik in solchen Schritten in der Implementierung nicht vollständig aktualisiert, so daß die neu hinzugekommenen *GJ*-Paare erst mit nachfolgenden Inkrementierungen der Konfliktzahlen einhergehend im Diagramm erhöht werden.

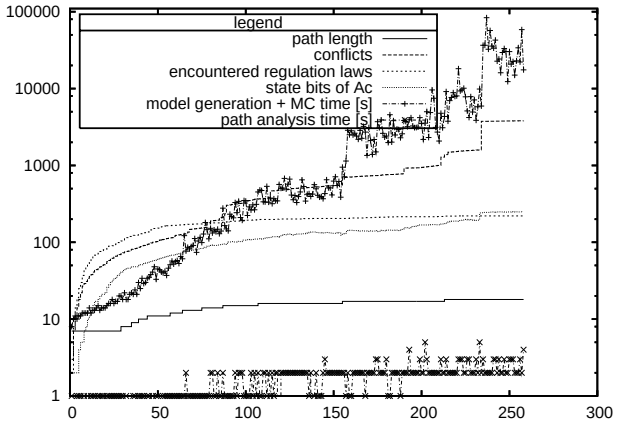


Abbildung 7.11: Verlauf des Verifikationsprozesses für Eigenschaft φ_4^{FHS} auf logarithmischer Skala

aufgeführten Verifikationsprozesse wurden alle mit dieser Erweiterung mit Strategie 1, dem Verzicht auf verlängerte Sequenzgenerierung, durchgeführt, d.h. die Erweiterung wurde nur zum Lernen von Konflikten verwendet, die eine Verkleinerung des ω-Automaten bewirken bzw. als Seiteneffekte von Verkleinerungsversuchen entdeckt werden. Ein Blick auf Tabelle 7.1 zeigt, daß der Anteil der Anzahl der in der Minimierung entdeckten Konflikte C_{min} gegenüber der durch Pfadanalyse ermittelten Konflikte mit der Größe des ω-Automaten anzusteigen scheint. Dieser Effekt ist bei dem Autopilot System dabei weniger ausgeprägt als bei dem Fensterheber System. Bei mit kleinerem ω-Automaten beendeten Verifikationsprozessen sind häufig keine Konflikte in der Minimierung gefunden worden, während in Beweis φ_4^{FHS} die Anzahl $|C_{min}|$ mit 3499 den wesentlichen Teil der insgesamt 3816 Konfikte ausmacht.

Wir betrachten den Verifikationslauf des Beweises φ_4^{FHS} in Abbildung 7.11 nun genauer. Den Kurvenverläufen ist zu entnehmen, daß die Anzahl der Konflikte in regelmäßigen Abständen zunehmend größere Sprünge macht. An diesen Stellen sind Konflikte in der Pfadanalyse aufgetreten, welche durch die Minimierung zu vielen weiteren Konflikten führen, welche sich durch den ω-Automaten erlernen lassen, ohne daß dieser sich nennenswert vergrößert. Werden in der Pfadanalyse Konflikte ermittelt, die zu einer Erweiterung des ω-Automaten um mehrere Zustandsbits führen, so führen diese zu der Entdeckung von umso mehr Konflikten in der anschließenden Minimierung. Dies ist besonders deutlich an der letzten Stufe im Verlauf der Zustandsbits des ω-Automaten und den nachfolgenden Sprung der Konfliktanzahl erkennbar. Der Automat wird vor der Hinzunahme der zusätzlichen Konflikte vergrößert. Leider führt dieser Verifikationsprozess nur sehr langsam zu größeren Pfadlängen und wurde daher vorzeitig abgebrochen.

Das Beispiel zeigt, daß tatsächlich eine Vielzahl von zusätzlichen Konflikten durch den Automaten erlernt werden kann, ohne daß sich dieser vergrößert. Daher ist bei allen durchgeführten Beweisaufgaben diese Erweiterung verwendet worden.

7.3.5 Diskussion der Statistiken

In Abbildung 7.12 sind weitere Prozeßverläufe einiger Beweise aus Tabelle 7.1 auf Seite 158 dargestellt. Weitere Diagramme zu den übrigen, noch nicht dargestellten Verläufen finden sich im Anhang ab Seite 193.

Im Folgenden werden die Kenngrößen hinsichtlich der Statistiken betrachtet.

Diese sind natürlich nicht in Isolation zu betrachten und dienen nur als systematische Einstiege in die verschiedenen diskutierten Aspekte der statistischen Daten.

Anzahl der Regelungsgesetze Die Anzahl der Regelungsgesetze (*GJ*-Paare) $|\Sigma|$ bewegt sich in einem, verglichen mit den Größen der diskreten Zustandsräume, sehr kleinen Bereich. Selbst bei den Beweisaufgaben, bei denen sehr viele Iterationen durchgeführt wurden und die Pfadlänge mehrere Schritte umfasst, ist die Anzahl der unterschiedlichen *GJ*-Paare niedrig. Als Beispiel sei hier Beweisaufgabe φ_{11}^{APS} angeführt, wo in 277 Iterationen nur 135 verschiedene *GJ*-Paare entdeckt wurden. Da hier die durchschnittliche Pfadlänge der Iterationen in der Größenordnung von 10 Schritten liegt, hätten damit $\sim 10 \cdot 277 = 2770$ verschiedene *GJ*-Paare besucht werden können, es sind jedoch nur etwa 5% der maximal möglichen in den ermittelten Pfaden aufgetreten. Dies unterstreicht, daß es nicht viele verschiedene *GJ*-Paare gibt, die für den Beweis der Eigenschaft Relevanz haben.

Etwas anders verhält es sich mit dem Fuelrate Controller System. Hier wurde bereits angemerkt, daß selbiges sich aufgrund des kleinen diskreten Zustandsraumes, sowie der durch mehrere große Kennlinienfelder verursachten großen Anzahl unterschiedlicher *GJ*-Paare, nicht besonders gut für das ω-CEGAR Verfahren eignet. Entsprechend beobachten wir hier auch ein, verglichen mit der Anzahl diskreter Zustände, großes Alphabet Σ und aufgrund der vielfältigen Kombinationsmöglichkeiten der Kennlinienfeldzugriffe eine große Anzahl an Konflikten, die ein rasches Fortschreiten der Pfadlänge unterbinden (z.B. φ_4^{FCS}).

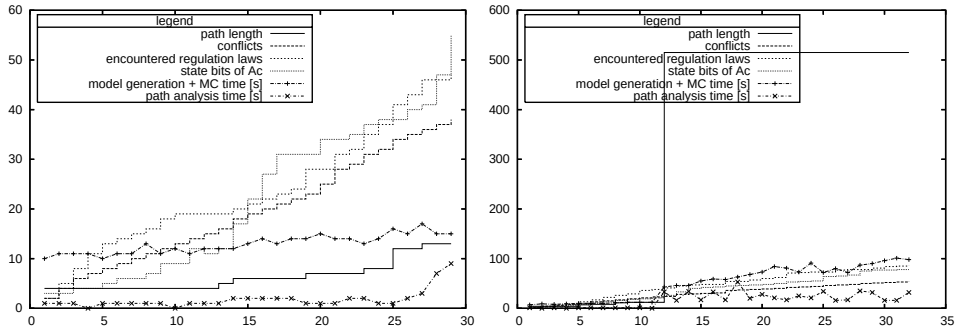
Die Anzahlen der verschiedenen Teilmengen $|\Sigma_i|$ und $|\Sigma_v|$ des regulären und des ω-Automaten sind z.T. größer als die von $|\Sigma|$, was sich aus den Kombinationsmöglichkeiten der Systematiken erklärt. So kann bereits aus einem einzigen *GJ*-Paar, welches durch n Super-Trace-Variablen mit einem Wertebereich von angenommenen k möglichen Belegungen pro Variable[6] eine Anzahl von k^n Teilmengen (-belegungen) generiert werden, indem für Jump Set Funktionskomponenten die Funktion $\zeta_{\mathbb{R}}$, bzw. bei Guard Set Komponenten die Menge X verwendet wird. In vielen Fällen gilt $|\Sigma_v| > |\Sigma|$, jedoch liegt dies ursächlich an den verallgemeinerten Konflikten, die ebenfalls Pfade zu *GJ*-Paaren ausschließen (könnten), die in der Statistik nicht auftreten.

Dimensionen Die Anzahl $\mathfrak{n}$ der Dimensionen (zustandsbehafteten Variablen kontinuierlichen Typs) variiert in den aufgeführten Verifikationsläufen je nach Modell und des individuellen Cone-of-Influence der Beweisaufgabe. Es finden sich viele Beispiele, in denen trotz größerem $\mathfrak{n}$ längere Pfade mit weniger Iterationen ermittelt wurden, als in anderen Fällen mit deutlich kleinerem $\mathfrak{n}$. Hier wäre insbesondere Beweisaufgabe φ_4^{APS} im Vergleich zu φ_{11}^{APS} oder, wenn wir Modellübergreifende Vergleiche erwägen, die Beweisaufgabe φ_4^{FHS}. Eine Korrelation der Dimensionsanzahl $\mathfrak{n}$ mit der Komplexität der Verifikation ist damit in den betrachteten Beispielen nicht ersichtlich.

Darüber hinaus ist festzustellen, daß $\mathfrak{n}$ mit bis zu 23 Dimensionen hier z.T. deutlich größer ist, als in den aufgeführten akademischen Beispielen aus Tabelle 4.1 auf Seite 37.

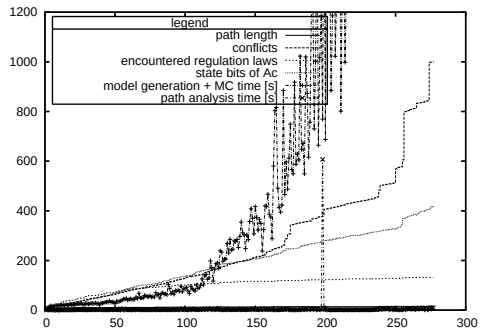
Anzahl der Konflikte Die Anzahl der Konflikte variiert erheblich zwischen den Beweisaufgaben. So sind Pfade der Länge 515 mit nur 42 Konflikten zu ermitteln (φ_4^{APS}), während bei anderen Beweisaufgaben selbst nach 1000 Konflikten nur ausgeschlossen werden kann, daß es einen Pfad gibt, der kürzer als 15 Schritte

[6] Die Wertebereiche der Super-Trace-Variablen differieren, was für diese Beispielrechnung jedoch irrelevant ist.

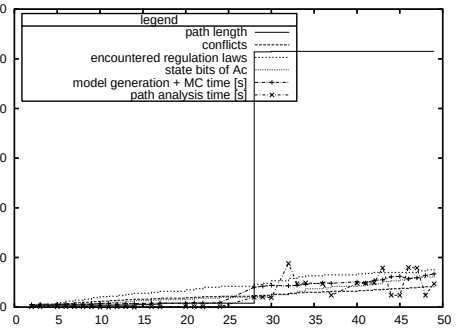


(a) Beweisaufgabe φ_1^{APS}

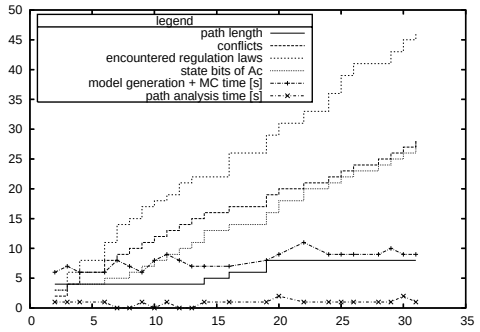
(b) Beweisaufgabe φ_{12}^{APS}



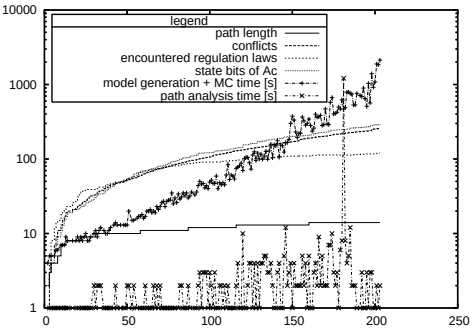
(c) Beweisaufgabe φ_{11}^{APS}



(d) Beweisaufgabe φ_4^{APS}

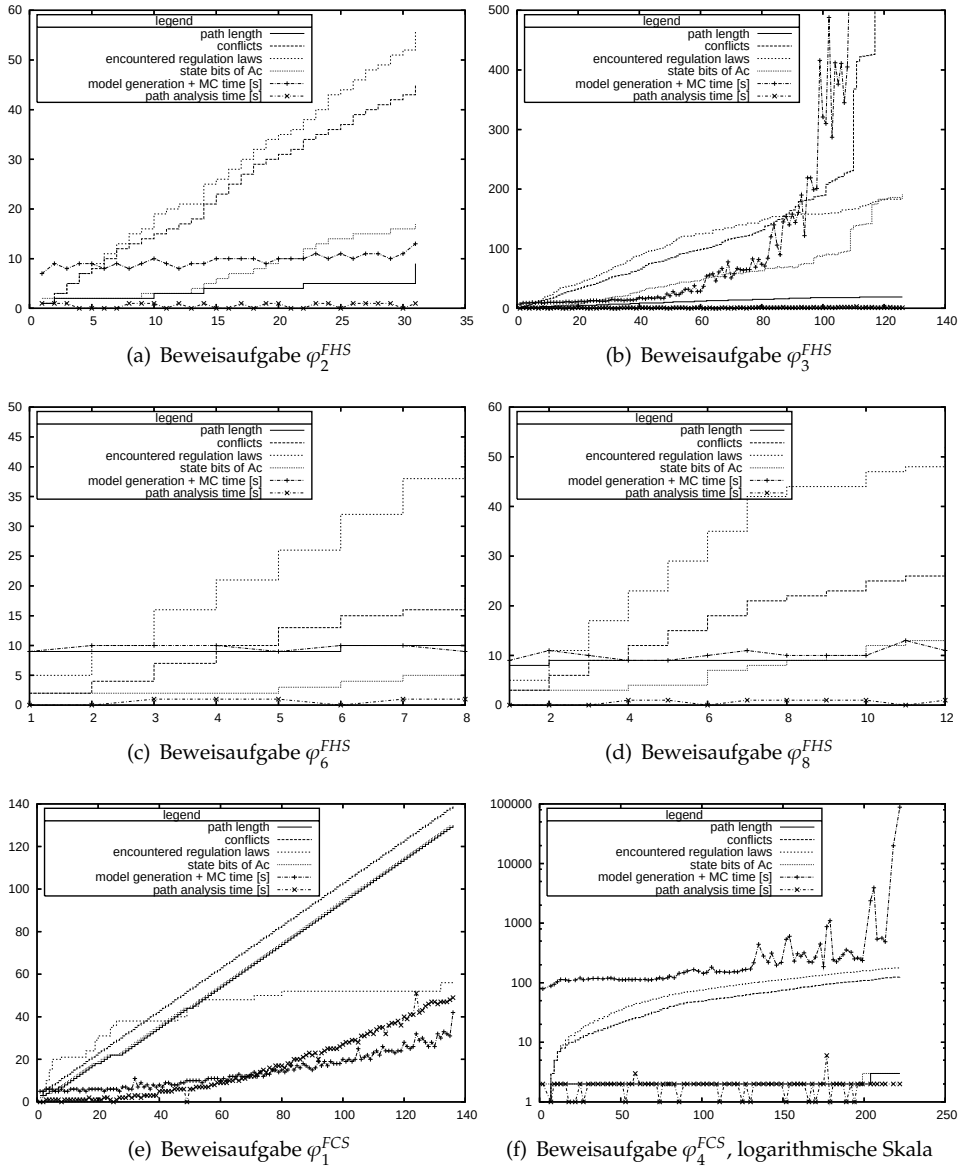


(e) Beweisaufgabe φ_5^{APS}



(f) Beweisaufgabe φ_6^{APS}

Abbildung 7.12: Verläufe von Verifikationsprozessen verschiedener Beweise auf dem Autopilotsystem



(a) Beweisaufgabe φ_2^{FHS}

(b) Beweisaufgabe φ_3^{FHS}

(c) Beweisaufgabe φ_6^{FHS}

(d) Beweisaufgabe φ_8^{FHS}

(e) Beweisaufgabe φ_1^{FCS}

(f) Beweisaufgabe φ_4^{FCS}, logarithmische Skala

Abbildung 7.13: Verläufe von Verifikationsprozessen verschiedener Beweise auf dem Fensterheber System und dem Fuelrate Controller System

ist, obwohl der Cone-of-Influenz bzgl. kontinuierlicher Variablen deutlich kleiner ist. Hier ist also eine große Abhängigkeit von der Beweisaufgabe zu beobachten. Schauen wir uns die Beweisspezifikationen an, so ist insbesondere bei φ_{11}^{APS} die Abhängigkeit von der Beweisparametrierung deutlich. Die Beweise φ_{6}^{APS} bis φ_{11}^{APS} unterscheiden sich nur in dem Parameter der angenommenen Windstärke und die Anzahl der Konflikte variiert zwischen 57 und > 1000. Die Begründung ist hier also im kontinuierlichen Zustandsraum zu suchen und da auch die Pfadlängen von 9 auf ≥ 15 ansteigen, ist hier primär das Erreichen einer bestimmten Teilmenge von X der primäre Hintergrund der Komplexität. Wie an diesem Beispiel ersichtlich, ist die ω-CEGAR Implementierung in seiner jetzigen Form für solche Situationen noch nicht geeignet. Wie bereits in Abschnitt 4.7.2.2 auf Seite 79 andiskutiert, können hier durch analytisch motivierte Teilmengenbildungen vermutlich bessere Ergebnisse erreicht werden.

Die Auswirkung der erweiterten Minimierung mit Konflikterkennung haben wir bereits in Abschnitt 7.3.4 auf Seite 171 betrachtet.

Anzahl der Iterationen Für die Anzahl der benötigten Iterationen gilt ähnliches, wie für die Konflikte hinsichtlich der Unterschiede zwischen den verschiedenen Beweisaufgaben. Wie bereits bemerkt kann die Anzahl der Konflikte bei Einsatz der Lazy Evaluation Strategie zur Berechnung der minimalen *GJ*-Repräsentationen kleiner als die Anzahl der Iterationen sein. Besonders deutlich wird dies in Beweisaufgabe φ_{5}^{FCS}, wo dieses Vorgehen unumgänglich ist, wie in Abschnitt 7.2.3 auf Seite 165 begründet. In 161 Iterationen wurden nur 92 Konflikte gefunden, d.h. daß bis zu 69 Iterationen nur zur Verfeinerung des Alphabets des ω-Automaten verwendet wurden und die Anzahl der Iterationen vor diesem Hintergrund zu relativieren ist.

Länge der Pfade Die Pfadlängen hängen von der Eigenschaft und dem Modell ab und die grundsätzliche Unabhängigkeit der Pfadlänge von der Verfahrenskomplexität haben die Beweise großer Pfadlängen gezeigt. An dieser Stelle ist der Verlauf der Pfadlängen während des iterativen Verifikationsprozesses von größerem Interesse.

Der Fortschritt der Pfadlänge während des Verifikationsprozesses ist in den meisten Fällen z.T. deutlich unterhalb der Iterationszahl angesiedelt, insbesondere, wenn vor jeder Verlängerung viele Konflikte erlernt werden müssen, die nicht durch die erweiterte Minimierung erkannt werden können. So finden sich insbesondere in den Beispielen φ_{11}^{APS}, φ_{4}^{FHS} und φ_{4}^{FCS} nur sehr zögerliche Pfadlängenvergrößerungen. Demgegenüber verläuft in Beispiel φ_{1}^{FCS} der Fortschritt der Pfadlänge weitgehend proportional zur Iterationszahl, d.h. in fast jedem Schrittt führt die Ermittlung eines Konflikts zu einem um einen Schritt längeren Pfad in der nächsten Iteration. Dieses Verhalten ist typisch für einfache Integratorfunktionalitäten, bei denen Konstanten zu einer zustandsbehafteten Variablen addiert werden.

Besonders interessant ist hier der Fall φ_{12}^{APS}, da der Verlauf der Pfadlänge, wie in Abbildung 7.12(b) dargestellt, nach einer kurzen Anfangsphase sprunghaft auf 515 wechselt. Bei diesem, durch den überlaufenden Zähler verursachten Beispiel „erkennt" das Verfahren (der Model-Checker), daß erst nach einem Wertewechsel auf dem *TelemetryValid*-Signal ein Gegenbeispiel möglich wird und überspringt dabei alle nötigen Schritte innerhalb einer Iteration. Dennoch sind nach dem Pfadlängenwechsel weitere Iterationen nötig, bis auch hier ein gültiger Pfad gefunden wird. Dieses Beispiel zeigt, daß das Verfahren insbesondere in solchen Fällen gut geeignet ist, in denen die Pfadlänge eines gültigen Gegenbeispiels primär durch den diskreten Verhaltensanteil bestimmt wird, und nur sekundär durch Zustandswechsel

im kontinuierlichen Raum. Das andere Extrem wäre hier das bereits diskutierte Beispiel φ_{11}^{APS}, wo die Abhängigkeit der Pfadlänge von Randbedingungen des kontinuierlichen Zustandsraums offensichtlich ist.

An dieser Stelle ist weiterhin festzustellen, daß das Verfahren in mehreren Fällen mit dem Ergebnis der Unerreichbarkeit der spezifizierten Eigenschaft terminiert, wodurch keine abschließende Pfadlänge angegeben werden kann. Dieser Fall unterstreicht insbesondere die Eignung des Verfahrens für Zertifizierungszwecke, da ein solches Ergebnis nach Lemma 1 auf Seite 24 verläßlich ist.

Größe des ω-Automaten Die Anzahl der Zustände des ω-Automaten ist nicht direkt mit der Iterations- oder Konfliktanzahl korrelliert, wie wir insbesondere in Abbildung 7.11 auf Seite 172 gesehen haben. Der Grund hierfür ist die kompakte Kodierung der Konflikte, die in Präfixen oder Suffixen übereinstimmen und sich so Zustände und Transitionen teilen können. Dieser Umstand deutet darauf hin, daß diese Konflikte aus analytischer Sicht ebenfalls *ähnlich* sind und bei geeigneter Gruppenbildung (Teilmengenbildung) zu allgemeineren Konflikten zusammengefaßt werden könnten. Bezogen auf die Partitionierung des kontinuierlichen Zustandsraums durch die Zustände des ω-Automaten durch die in Abschnitt 4.4.1 auf Seite 64 eingeführte Funktion χ läßt dies darauf schließen, daß durch syntaktisch unterschiedliche *GJ*-Sequenzen zumindest Inklusionsbeziehungen zwischen den Zuständen existieren, die sich durch gemeinsame Suffixe äußern. Sofern diese Konflikte nicht durch erweiterte Minimierung gefunden werden, kann der ω-Automat diese zwar ohne nennenswerte Zustandsraumvergrößerungen erlernen, jedoch sind dennoch viele Iterationen hierzu notwendig. Analytisch motivierte Teilmengenbildung würde diese Zusammenhänge möglicherweise a priori erkennen und die Iterationszahlen erheblich reduzieren können.

Die größten ω-Automaten der aufgeführten Beispiele bewegen sich im Rahmen bis ca. 400 Zustandsbits, wobei ein Model-Check-Lauf dann bereits über einen Tag dauern kann (in Abbildung 7.12(e) in dem dargestellten Ausschnitt nicht erkennbar), wobei dieser Wert um einen mehrstelligen Faktor in aufeinanderfolgenden Iterationen varieren kann, wie bereits alle umfangreicheren Verifikationsläufe verdeutlichen. Da diese Dauer mehrfach iteriert wird, ist damit für die Größe des diskreten Zustandsraums des Autopilotensystems mit 31 Zustandsbits die praktisch-relevante Komplexitätsgrenze erreicht. Hierbei muß natürlich angemerkt werden, daß die Leistungsfähigkeit der Rechner maßgeblich diese Grenze beeinflußen und der verwendete Rechner[7] diesbzgl. nicht dem aktuellen technischen Stand entspricht.

Weiterhin ist die Größe des ω-Automaten erheblich von der Länge der Konflikte abhängig. Wann immer Sprünge in der Anzahl der Zustandsbits in den graphischen Verläufen zu erkennen sind, werden diese i.d.R. durch einen einzigen, zusätzlich erlernten Konflikt verursacht. Für diesen müssen aufgrund seiner Länge, genauer aufgrund seines neuen Präfixes oder Suffixes, der nicht mit bereits erlernten Konflikten geteilt werden kann, entsprechend mehrere neue Zustände hinzugefügt werden. Auf der anderen Seite benötigen Konflikte der Länge 1 keine zusätzlichen Zustandsbits, da hier nur Transitionen hinzugefügt werden, die zu einem nicht-akzeptierenden Zustand führen. Dieser trägt jedoch im Kreuzprodukt nach Abschnitt 4.3.4 auf Seite 63 nicht zur Vergrößerung des Zustandsraumes bei. Besonders deutlich ist dieser Fall in der Durchführung der Beweisaufgabe φ_4^{FCS} erkennbar, wo 125 invariante Konflikte nur einen einzigen Zustand zu deren Kodierung benötigen (= 0 Zustandsbits). Diese haben ausnahmslos die Länge 1 und sind durch Kennlinienfelder und deren vielfältigen Wertkombinationen verursacht.

[7] Wie bereits erwähnt eine Workstation *Sun Fire V490* mit 1350 MHz CPU-Taktung

Model-Check-Zeit Stark von den Automatengrößen abhängig ist, wie bereits erwähnt, die vom Model-Checker benötigte Zeit zur Widerlegung der Eigenschaft und der Berechnung eines Gegenbeispiels innerhalb einer Iteration des Verfahrens. Ebenfalls erwähnt wurde bereits die hohe Streuung dieser Kenngröße bei nahezu identischer Automatengröße. Von diesen, durch die heuristischen BDD-Techniken erklärbaren Streuungen abgesehen ist jedoch generell in allen umfangreicheren Verifikationsläufen zu beobachten, daß die benötigte Zeit im Wesentlichen exponentiell in der Iterationszahl anwächst, was wiederum auf die anwachsende Automatengröße zurückzuführen ist. Es ist daher für das Verfahren von zentraler Bedeutung, einen Pfad mit einem kleinen ω-Automaten zu finden, bevor wir in den Bereich des starken Anstiegs der benötigten Model-Check-Zeit geraten.

Eine Besonderheit ist in dem Beispiel φ_4^{FCS} zu sehen. Obwohl das Modell nur 256 erreichbare Zustände besitzt und der ω-Automat nur 3 Zustandsbits umfasst, wachsen die Model-Check-Zeiten in exorbitante Größenordnungen und verhindern hier die vollständige Verifikation der Beweisaufgabe. Dies ist ursächlich in der wiederverwendeten Integration des Model-Checkers, in der aus technischen Gründen für Propositionsvariablen Zustandsbits erzeugt werden, die bei den vielen Kennlinienfeldern des Modells von erheblicher Anzahl sind. Durch Verfeinerung des Alphabets im Rahmen der Lazy Evaluation Strategie der Berechnung minimaler *GJ*-Repräsentationen geraten diese zunehmend in den Cone-of-Influence und erhöhen damit signifikant die Komplexität des Modells. Bedauerlicherweise liegen daher für dieses Modell nur wenige prägnante Ergebnisse vor.

Pfadanalysezeit In nahezu allen Beispielen ist bei Betrachtung der Verläufe eine gleichbleibend geringe Zeit für die Pfadanalyse festzustellen. Dies unterstreicht die hohe Effizienz, mit der der Solver[8] die Lösungsfunktion für kurze Pfade anwenden kann. Bei dieser Kenngröße ist das Anwachsen der Zeitspanne mit der Pfadlänge zu beobachten, hier insbesondere in den Beispielen φ_4^{APS} und φ_{12}^{APS}, sowie φ_1^{FCS}. Demgegenüber sind deutliche Abweichungen hiervon wie z.B. bei φ_{11}^{APS} durch numerische Probleme zu erklären, bei denen der Solver durch aufwendige Verfahren die Konditionierung der Formel im Rahmen seines ebenfalls iterativen Lösungsalgorithmus wiederholt zu verbessern versucht. Exemplarische Untersuchungen zu längeren Pfaden durch Erhöhung der Bitzahl des überlaufenden Zählers in den Beispielen φ_4^{APS} und φ_{12}^{APS} haben gezeigt, daß bis zu einer Pfadlänge von über 2000 Schritten auf diesem Modell mit Glück noch Ergebnisse erzielt werden können. Über numerische Probleme hinaus gestaltet sich die Pfadanalyse für deutlich längere Pfade als sehr aufwendig, da mit der in Abschnitt 4.3.2.3 auf Seite 43 beschriebenen Fenstertechnik zur Ermittlung von Konflikten eine sehr lange Sequenz mehrfach abzusuchen ist. Daher ist die Pfadlänge in der Praxis nicht nur durch numerische Probleme, sondern auch durch den erforderlichen Berechnungsaufwand begrenzt. I.d.R. ist jedoch die numerisch bedingte Grenze die vorherrschende.

Gesamtlaufzeiten Die Gesamtlaufzeiten setzen sich additiv aus der Pfadanalysezeit und der Model-Check-Zeit zusammen. Die benötigte Zeit zur Generierung des ω-Automaten ist im Vergleich hierzu zu vernachlässigen, wie bereits Tabelle 4.4 auf Seite 95 angedeutet hat. Eine Ausnahme bildet jedoch die erweiterte Minimierung bei großen Automaten. Bei dem in Abbildung 7.11 auf Seite 172 dargestellten Verlauf werden bei den Konfliktanzahl-Sprüngen die Konflikte nicht innerhalb einer Minimierung entdeckt, sondern im Rahmen des in Algorithmus 8 auf Seite 74 beschriebenen Fixpunktverfahrens durch iterierte Minimierung des Automaten. Mitunter werden die Konflikte hierbei in kleinen Gruppen ermittelt, so daß viele

[8] Primär wurde lpsolve verwendet und nur bei numerischen Problemen auf glpk gewechselt.

Minimierungen nötig sind, welche sich bei großen Automaten beträchtlich aufsummieren können. Diese Kenngröße ist in den Statistiken nicht aufgeführt, kann aber indirekt durch die Gesamtlaufzeit abzüglich der Model-Check-Zeit und der Pfadanalysezeit ermittelt werden. Eine ebenfalls präsente Integrations-bedingte Zeitspanne[9] innerhalb jeder Iteration kann vor dem Hintergrund langer Verifikationszeiten vernachlässigt werden.

Wir haben in diesem Abschnitt systematisch die einzelnen Kenngrößen betrachtet, diesbzgl. markante Fälle herangezogen und Zusammenhänge diskutiert. Dabei hat sich die in Abschnitt 4.1.2 auf Seite 35 diskutierte Charakteristik der Modelle als zutreffend vor dem Hintergrund der geringen Anzahl regelungstechnischer Gesetze (*GJ*-Paare) erwiesen. Ein wesentliches Ergebnis ist die Beobachtung, daß das ω-Cegar Verfahren sehr effizient in den Verifikationsläufen agiert, in denen die Pfadlänge eines gültigen Gegenbeispiels primär durch den diskreten Verhaltensanteil des Modells bedingt ist. Im Falle von primär durch kontinuierliche Zustandsberechnungen motivierte Pfadlängen ist das Verfahren ebenfalls effektiv, bedarf jedoch einer Effizienzsteigerung, wie sie durch geeignete, analytisch motivierte Teilmengenermittlung möglich scheint. Das Verfahren eignet sich zur Zertifizierung von Eigenschaften, wie der Nachweis der Nicht-Erreichbarkeit einiger Eigenschaften demonstriert.

Abschließend bleib anzumerken, daß auch für die nicht beendeten Verifikationsläufe ein Teilergebnis vorliegt, welches in der Nicht-Existenz eines Pfades bis zur Pfadlänge $|\hat{\pi}| - 1$ besteht. Bei Pfadlängen von 18 (φ_4^{FHS}), bzw. 130 (φ_1^{FCS}) kann bereits diese Information für diagnostische Untersuchungen des Modells hilfreich sein.

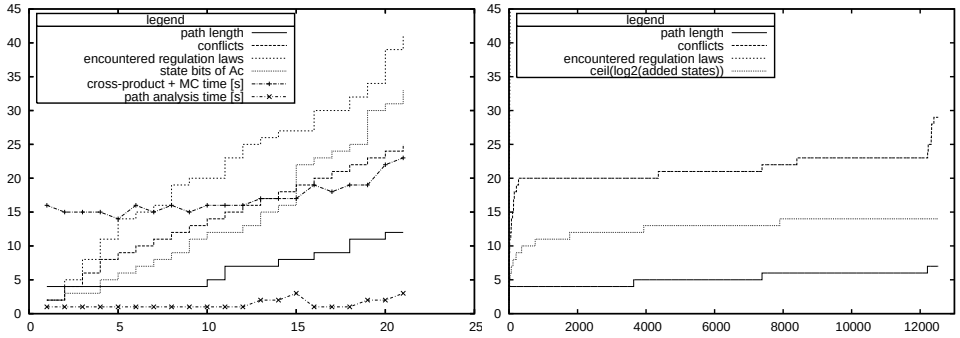
7.4 Vergleich mit Infinite-State-Cegar

In Abschnitt 6.7 wurde die Umsetzung einer Verfahrenserweiterung zum selektiven Ausschluß von Pfadfragmenten vorgestellt, mit der das in Abschnitt 4.9.1 beschriebene Infinite-State-Cegar Verfahren soweit nachgeahmt werden kann, daß genaue Aussagen zu der Anzahl benötigter Iterationen möglich sind. Diesen Nachahmungsmodus verwendend vergleichen wir nun die Anzahl benötigter Iterationen des ω-Cegar Verfahrens und des Infinite-State-Cegar Verfahrens einschließlich des Verlaufs der iterativen Verfeinerung.

Das Ergebnis des Verlaufs ist in Abbildung 7.14 dargelegt, wobei anzumerken ist, daß im Falle des Infinite-State-Cegar-Nachahmungsmodus der Verifikationsprozess noch nicht abgeschlossen ist und derzeit bei Pfaden der Länge 9 angelangt ist. Da der Nachahmungsmodus aufgrund der Verwendung des für diesen Einsatzzweck ungeeigneten ω-Automaten ein erheblich größeres Modell zu verifizieren hat als in [CFH+03], dauert der Prozeß zum Einen so lange, daß auch nach mehreren Wochen der Verfeinerungsprozeß nur bis zu dem abgebildeten Stand voranschreitet, zum Anderen verbietet sich aus diesem Grund eine Angabe der Model-Check-Zeiten der einzelnen Iterationen, die, von den ersten Iterationen abgesehen, deutlich länger sind und keine Aussagekraft haben.

Signifikant sind hingegen die in dem Diagramm angegebenen Daten zur Iterationsanzahl, der Pfadlänge, der Anzahl der gefundenen verschiedenen Konflikte, die Anzahl der im Rahmen der Pfadauswertung entdeckten verschiedenen *GJ*-Paare und ausgehend von der Initialabstraktion A_0 die mit der Verfeinerung einhergehende Vergrößerung der Abstraktionen A_i. Bei der letzten Kenngröße ist zu beachten,

[9] Diese umfasst alle Aktivitäten, die sich aus der Wiederverwendung existenter Softwaremodule und -Integrationen ergibt, welche nicht für diesen Einsatzzweck optimiert sind.



(a) ω -Cegar (b) Infinite-State-Cegar

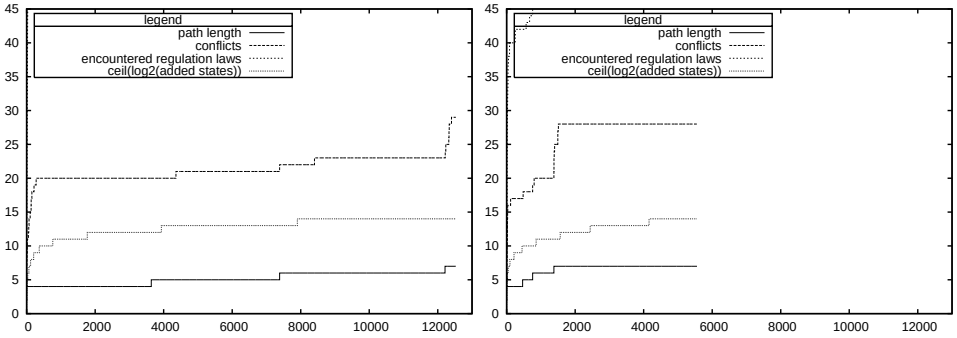
Abbildung 7.14: Gegenüberstellung von ω-Cegar und Infinite-State-Cegar für Beweisspezifikation φ_{14}^{APS}

daß diese additiv in die Größe des Zustandsraums von A_0 eingeht, und nicht multiplikativ, wie bei dem ω-Cegar Verfahren. So umfasst der Zustandsraum von A_i also für einen im Diagramm angegebenen Wert 19 eine Anzahl von $numstates(A_0) + 2^{19}$ Zustände. Dieser Wert wird rechnerisch aus der Anzahl und Länge der Konflikte ermittelt, da bekannt ist, daß für ein Pfadfragment der Länge n eine Anzahl von $n - 1$ zusätzlicher Zustände nötig ist, um genau dieses Fragment auszuschließen.

Betrachten wir nun die Anzahl der Iterationen, so ist festzustellen, daß auch nach über 12500 Iterationen nur eine Pfadlänge von 7 Schritten erreicht ist, d.h. es ist lediglich auszuschließen, daß ein gültiger Pfad einer Länge kleiner als 7 existiert. Demgegenüber liefert das ω-Cegar Verfahren bereits nach 23 Iterationen einen Pfad der Länge 13. Ein Blick auf die Anzahl der gefundenen Konflikte zeigt im Fall Infinite-State-Cegar, daß auch nach 12500 Iterationen weniger als 30 verschiedene Konflikte entdeckt worden sind, d.h. die gleichen Konflikte sind wieder und wieder die Ursache für die Ungültigkeit eines Pfades. Diese 30 Konflikte hätte man auch innerhalb von 30 Iterationen finden können. Bei rigorosem Ausschluß erneuten Auftretens dieser Konflikte sind sogar bereits 25 von ihnen ausreichend[10], um den gültigen Pfad zu ermitteln, wie das ω-Cegar Verfahren innerhalb von 23 Iterationen demonstriert. Hinsichtlich der entdeckten *GJ*-Paare ist die zugehörige Kurve für Infinite-State-Cegar so steil ansteigend, daß sie in dem Diagramm nicht sichtbar ist. Daher sei an dieser Stelle gesagt, daß der Wert sehr schnell auf 130 ansteigt, um dann deutlich langsamer einen Wert von 175 nach 12500 Iterationen zu erreichen. Es sind demnach viele *GJ*-Paare entlang der Pfade vorhanden, die mit ω-Cegar nicht „besucht" werden und offenbar auch nicht müssen, da hier nur 40 Paare „entdeckt" worden sind. Die vom Model-Checker ermittelten Pfade verlaufen folglich deutlich zielgerichter, da ein Großteil des Modells außen vor gelassen werden konnte.

Kommen wir nun zu einer Betrachtung der Gesamtlaufzeit. Wie bereits erwähnt, liegen für Infinite-State-Cegar konkrete Meßdaten weder für einzelne Iterationen, noch für den gesamten Verifikationsprozeß vor und man könnte mutmaßen, daß zwar viel mehr Iterationen benötigt wurden, die Verifikationszeiten der Iterationen aber (über-)kompensierend geringer sind, da der abstrakte Automat A_i sehr viel langsamer in seiner Komplexität anwächst. Zur Widerlegung einer solchen Spekulation können wir jedoch eine untere Schranke der Laufzeit von Infinite-State-Cegar abschätzen: Die in jeder Iteration benötigte Zeit setzt sich neben dem Aufwand für die Abstraktionsverfeinerung zusammen aus der vom Model-Checker benötigten Zeit $t_{MC}(A_i)$ zur Durchführung des Model-Check-

[10] Diese Zahl ist jedoch unter der in Abschnitt 7.3.1.2 diskutierten Varianz zu betrachten.



(a) Infinite-State-Cegar

(b) Infinite-State-Cegar mit Ausschluß paralleler Transitionen

Abbildung 7.15: Gegenüberstellung von einfachem Infinite-State-Cegar und dem mit zusätzlichem Ausschluß von Paralleltransitionen

Algorithmus auf dem abstrakten Modell A_i einschließlich der Ermittlung des Pfades und der für die Pfadanalyse benötigten Zeit $t_{analyze}$. Da wir nun annehmen können, daß $t_{MC}(A_i) \geq t_{MC}(A_0)$, ist $t_{MC}(A_0)$ eine untere Schranke. Die Zeit $t_{analyze}$ ist für beide Verfahren annähernd gleich, da die Intervallbestimmung nach Algorithmus 2 gleich ist und bei ω-Cegar nur ein zu vernachlässigender Anteil für die Bestimmung der Teilmengen durch Algorithmus 10 hinzukommt. Nehmen wir gemäß Diagramm 7.14(a) für $t_{MC}(A_0) = 16s$ und für $t_{analyze} = 1s$ an, zusammen also 17 Sekunden für eine Iteration von Infinite-State-Cegar. Wenn das Verfahren nach 12500 Iterationen terminiert wäre, hätten wir somit eine Gesamtlaufzeit von 212500 Sekunden oder 59 Stunden. Dies ergibt einen Mindestfaktor von 221, den das Infinite-State-Cegar Verfahren länger benötigt als das ω-Cegar Verfahren, wobei erneut darauf hingewiesen werden muß, daß der Prozeß des Infinite-State-Cegar Verfahrens noch andauert.

Damit sind die in Abschnitt 4.9.1 aufgestellten und diskutierten Spekulationen bestätigt. Auf Modellen mit größerem diskreten Zustandsraum ist eine Strategie des selektiven Pfadfragmentausschlusses angesichts der Vielzahl auszuschließender Pfadfragmente ein ineffizienter Ansatz.

7.4.1 Ausschluß paralleler Transitionen

In Abschnitt 6.7.3 wurde von einer möglichen Verbesserung des Infinite-State-Cegar Verfahrens für Automaten nach Definition 12 durch Analyse paralleler Transitionen mit unterschiedlichen *GJ*-Paaren diskutiert, welches durch Kombination des Nachahmungsmodus mit dem Teilmengensequenzausschluß ebenfalls imitiert werden kann. In Abbildung 7.15 können beide Varianten verglichen werden, wobei der Prozeß dieses Mal nur bis etwa 5500 Iterationen durchgeführt wurde[11].

Zu erkennen ist, daß hierdurch eine deutliche Beschleunigung zu erzielen ist, jedoch bleibt die Leistung des Verfahrens weiterhin weit hinter ω-Cegar zurück. Indes der Trend, daß selbst einfache Maßnahmen zur Generalisierung von Konflikten die Iterationszahl deutlich reduzieren kann, ist auch durch dieses Schaubild bestätigt.

[11] Auch hier ist eine wochenlange Prozesslaufzeit allein für die dargestellte Kurve zu veranschlagen.

7.5 Ergebniszusammenfassung

Die experimentellen Untersuchungen des ω-CEGAR Verfahrens haben folgende Ergebnisse geliefert:

- Die Verwendung von Teilmengensequenzen führt zu einer erheblichen Beschleunigung des Verfahrens.
- Die erweiterte Minimierung führt zum Erlernen einer großen zusätzlichen Anzahl von Konflikten, ohne den ω-Automaten zu vergrößern.
- Die Lazy Evaluation Strategie erlaubt die Anwendung des Verfahrens auf Modelle mit großen Datenflußgraphen bei akzeptabler Erhöhung der Iterationszahlen.
- Kontrolldominierte Systeme beinhalten wenige relevante Regelungsgesetze.
- Es gibt keine Korrelationen der Anzahl benötigter Iterationen zu den Modellkenngrößen Dimensionsanzahl, Pfadlänge oder Größe des diskreten Zustandsraums.
- Bei vorwiegend durch den diskreten Verhaltensanteil beeinflußten Widerlegungen ist das Verfahren sehr effizient.
- Ein Ergebnis bis zu einer begrenzten komplexitätsabhängigen Pfadlänge k ist immer ermittelbar.
- Das Verfahren eignet sich zur Zertifizierung von Sicherheitseigenschaften.
- Gegenüber selektiven Pfadfragmentausschlüssen ist das ω-CEGAR Verfahren bei kontrolldominierten Systemen um Größenordnungen effizienter.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein iteratives Abstraktionsverfeinerungsverfahren zur Verifikation (Schritt-diskreter) Hybrider Systeme vorgestellt. Durch die Generalisierung von Konflikten in ungültigen Gegenbeispielen kombiniert mit einem lernenden ω-Automaten wird das Wissen um verallgemeinerte Konflikte zur Verfeinerung der Abstraktionen verwendet, indem durch eine Parallelkomposition der Abstraktion mit dem ω-Automat global ein wiederholtes Auftreten dieser Konflikte ausgeschlossen wird. Das für Schritt-diskrete lineare Hybride Systeme implementierte Verfahren ist auf Zeit-kontinuierliche nicht-lineare Hybride Systeme übertragbar und kann neben der Verifikation einfacher Sicherheitseigenschaften (Erreichbarkeitseigenschaften) bedingt auch zur Verifikation von temporallogischen Spezifikationen in CTL herangezogen werden.

Das Verfahren ist besonders für kontrolldominierte Hybride Systeme mit großen diskreten Zustandsräumen und wenig Wechselwirkung mit den kontinuierlichen Berechnungen anwendbar, wie sie mit Case-Tools als eingebettete Systeme erstellt werden können. Entsprechend ist die Implementierung des Verfahrens mit vielen gebräuchlichen Case-Tools über die C-Code-Generierung integriert und hierfür optimiert.

Experimentelle Resultate zeigen die Effektivität und für kontrolldominierte Systeme die Effizienz des Verfahrens auf den Fallbeispielen, wobei die Fallstudie eines einfachen Autopilotensystems und die Verifikation einiger exemplarischer Eigenschaften detailiert vorgestellt wurden. Die Beispiele haben gezeigt, daß z.T. durch sehr wenige Iterationen nicht-triviale Widerlegungen von Eigenschaften berechnet, bzw. Eigenschaften bestätigt werden können. Die Erweiterungen für in der Praxis vorkommende Schritt-diskrete lineare Hybride Systeme, wie insbesondere der Ausschluß von Teilmengensequenzen auf Basis von beobachteten typischen Strukturen dieser durch Case-Tools erstellten Modelle, erweist sich dabei als äußerst wirksam.

Das Verfahren skaliert mit der Größe des diskreten Zustandsraums insofern, als daß die Anzahl der benötigten Iterationen primär von den Wechselwirkungen von diskretem und kontinuierlichem Verhalten abhängig und damit von der Größe des diskreten Zustandsraums weitgehend unabhängig ist. Die Strategie zum globalen Ausschluß verallgemeinerter Konflikte im Rahmen der Abstraktionsverfeinerung erweist sich bei dem betrachteten Modell als deutlich effizienter als das nach anderer Maxime agierende Infinite-State-Cegar Verfahren zur minimalen Verfeinerung des abstrakten Systems.

Neben den bereits vorgestellten Erweiterungen gibt es noch viele Möglichkei-

ten, das ω-CEGAR Verfahren zu erweitern und für den industriellen Einsatz zu verbessern:

- Der wichtigste Punkt ist dabei der konsequente Ausbau von Systematiken zur Gruppierung der regelungstechnischen Gesetze, wie dies in einfacher Form durch die durch Dekomposition von Guard Sets und Jump Set Funktionen strukturell motivierten Teilmengen exemplarisch gezeigt wurde. Insbesondere für Modelle mit vielen Konstanten kontinuierlichen Typs, wie dies typischerweise bei der Verwendung von Kennlinienfelder der Fall ist, kann durch Schwellwert-abhängige Grupierungen eine signifikante Verallgemeinerung von Konflikten erreicht werden, die sich in Form von Teilmengensequenzen effizient durch den ω-Automaten ausschließen lassen. Es können sogar mehrere unabhängige Gruppierungsmuster eingeführt werden, so daß die Konflikte immer in maximaler Allgemeingültigkeit durch optimal bestimmte Teilmengensequenzen beschrieben werden können. Das Wachstum des ω-Automaten kann so auf ein Minimum beschränkt werden, bei gleichzeitigem Maximum an Generalisierung und damit Verfeinerung des Systems. Die Anzahl der Möglichkeiten bei diesem Erweiterungsansatz sind noch nicht abzusehen.

- Konflikte, die sich durch reguläre Ausdrücke unendlich oft instanziieren lassen, können als solche durch Induktionsbeweise erkannt und sehr kompakt in den ω-Automaten integriert werden. Der Lernprozeß würde lange Teilmengensequenzen mit sich wiederholenden Teilmustern durch Transitionen zu einem bereits existenten Zustand im ω-Automat realisieren und so zu einem sehr viel kompakteren Automaten führen.

- Das Verfahren kann durch Einsatz überapproximierender Pfadanalysetechniken auf nicht-lineare Hybride Systeme erweitert werden.

- Das Verfahren kann unter Verwendung alternativer Repräsentationsebenen der Modelle auf Zeit-kontinuierliche Systeme übertragen werden.

- Das Verfahren könnte enger mit dem Model-Checker integriert werden, so daß Zustände in der initialen Abstraktion A_0, die bereits in vorherigen Iterationen als nicht mehr erreichbar festgestellt wurden, aus A_0 dauerhaft entfernt werden, so daß der wachsende ω-Automaten A_C mit einer schrumpfenden initialen Abstraktion A_0 kombiniert wird.

- Schließlich ist, motiviert durch die Beobachtung einer z.T. sehr hohen Varianz in den Statistiken, welche durch günstige und ungünstige ausgeschlossene Konflikte bedingt sind, eine Heuristik zur Optimierung der Menge der verwendeten Konflikte denkbar. So könnte der ω-Automat Konflikte wieder „verlernen", die erheblich zu seiner Komplexität beitragen. Soweit dabei nicht gerade erlernte Konflikte betroffen sind, besteht durchaus die Möglichkeit, daß die entstehende Lücke, die nachfolgende ungültige Pfade nun wieder nutzen können, durch kürzere Konflikte wieder geschlossen werden können. Im Extremfall können Konflikte sogar obsolet werden, da ihr Auftreten in der Kombination von A_0 mit durch nachfolgend erlernten Konflikte ohnehin ausgeschlossen ist.

Dies ist bereits ein umfangreicherer, jedoch sicher nicht erschöpfender Katalog von vielversprechenden Erweiterungsmöglichkeiten. Unter diesem Gesichtspunkt steht die Erforschung der Verwendung von ω-Automaten zur Verifikation Hybrider Systeme vermutlich erst am Anfang.

Anhang A

Anhang

A.1 Verwendete Software-Module

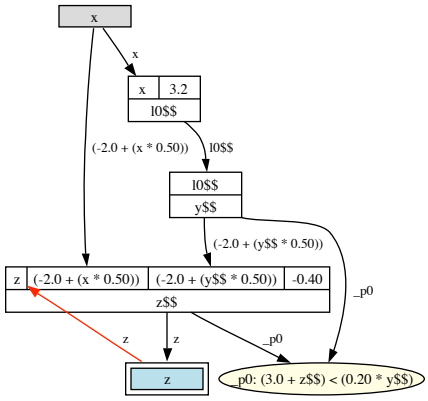
Bezeichnung	Autor	Beschreibung
c2smi	Marc Segelken	Compiler zur Erzeugung von SMI-Code aus C-Code (6.2.2)
smidestruct	Marc Segelken	Destrukturierung des SMI-Programms durch Substitution strukturierter Variablen (6.2.2.2)
smiaux	Marc Segelken	Klassifizierung von Hilfsvariablen, d.h. solche mit zustandslosem Verhalten. Erzeugung von Single-Assignment-Code (6.2.2.2)
iLabs	Marc Segelken	Berechnung von Zuordnungstabellen und $\mathcal{S}_{abs}$ einschließlich ω-Automat (6.3)
iLabs_trace	Marc Segelken	Pfadanalyse (6.5)
smirealcaster	Marc Segelken	Transformiert Casting-Ausdrücke $\mathbb{R} \leftrightarrow \mathbb{Z}$ (6.6)
Integration	Hartmut Wittke	Rahmenwerk der Zusammenarbeit aller beteiligten Werkzeuge (Abbildung 6.1)
smicoi	Tom Bienmüller	Berechnung des *Cone-of-Influence* (6.3.1.1)
smi4loopunroll	Rainer Lochmann	Rollt Schleifen im SMI-Programm ab (6.3.1.2)
smi2blifmv	Jürgen Bohn	Übersetzt SMI Programm in Blif-Mv Format für Model-Checker (6.4)
smitrc_smisim	Tom Bienmüller	Simuliert eine SMI Belegungssequenz $\langle \sigma_n^{\mathcal{S}_{orig}} \rangle$ zur Vervollständigung und Bestätigung dessen Gültigkeit (6.5.3)
patchwork_mv	Hartmut Wittke	Einbindung der Eigenschaft in das Blif_Mv-Modell unter Berücksichtigung des ω-Automaten-Observers (6.4.2)
vis	[Gro96a, RGA+96]	Model-Checker, verifiziert die abstrakten Modelle in jedem Iterationsschritt des ω-Cegar Prozesses (6.4.3)
lpsolve	[BEN04]	MILP-Solver, löst Gleichungen im Rahmen der Pfadanalyse (6.5.1.2)
glpk	[Mak03]	MILP-Solver, löst Gleichungen im Rahmen der Pfadanalyse (6.5.1.2)

A.2 Beispiele

```
/*INPUTS*/
extern float x;
extern int in;
extern bool c1;
/*OUTPUTS*/
bool out;
float y, z;

void step_func() {
  y = c1 ? x : 3.2;
  switch( in) {
        case 0: z = -2.0 + x*0.5;
          break;
        case 1: z = -2.0 + y*0.5;
          break;
        case 2: z = -0.4;
  }
  out = (3.0+z) < 0.2*y;
}

void main() {
  while(1) {
        step_func();
  }
}
```

```
CODE
 WHILE 1 DO
   DCASE
     [] c1:
       @[l0$ , x]
     [] ¬(c1):
       @[l0$ , 3.200]
   DESAC
   @[y$ , l0$]
   DCASE
     [] (in=0):
       @[z$ , (-2.000+(x·0.5000))]
     [] ¬((in=0)):
       DCASE
         [] (in=1):
           @[z$ , (-2.000+(y$·0.5000))]
         [] ¬((in=1)):
           DCASE
             [] (in=2):
               @[z$ , -0.4000]
             [] ¬((in=2)):
               SKIP
           DESAC
       DESAC
   DESAC
   @[out$ , ((3.000+z$)<(0.2000·y$))]
 OD
END
```

(a) C-Programm und SMI-Kompilat



(b) Datenflußgraph

```
MAPTABLE l0$
  (l0$=x)                  ↦ {-1**}
  (l0$=3.20)               ↦ {-2**}
MAPTABLE y$
  (y$=x)                   ↦ {-11*}
  (y$=3.20)                ↦ {-21*}
MAPTABLE z$
  (z$=z)                   ↦ {-**1}
  (z$=-2.00+0.50·x)        ↦ {-112,-**4}
  (z$=-0.40)               ↦ {-212,-**3}
MAPTABLE p0
  (3.00-0.20·x+z<0)        ↦ {1111}
  (2.36+z<0)               ↦ {1211}
  (1.00+0.30·x<0)          ↦ {1112,1114}
  (0.36+0.50·x<0)          ↦ {1214}
  (2.60-0.20·x<0)          ↦ {1113}
  false                    ↦ {1212,1213}
  (3.00-0.20·x+z>0)        ↦ {0111}
  (2.36+z>0)               ↦ {0211}
  (1.00+0.30·x>0)          ↦ {0112,0114}
  (0.36+0.50·x>0)          ↦ {0214}
  (2.60-0.20·x>0)          ↦ {0113}
  true                     ↦ {0212,0213}
```

(c) Zuordnungstabellen

Abbildung A.1: Beispielergebnis Zuordnungstabellenberechung. Dargestellt sind C-Programm, SMI-Kompilat, dessen Datenflußgraph sowie die errechneten Zuordnungstabellen wie in Abschnitt 6.3.3.1 auf Seite 136 erläutert. Die Zuordnungstabellen enthalten Belegungsvektoren platzsparend als Strings der Länge 4 kodiert, mit „*" als Zeichen für *don't-care* und „-" als Zeichen für *not-used*, wobei die Werte für Belegungen der Propositions- und Tracevariablen p_0,T_l_0,T_y,T_z in dieser Reihenfolge stehen.

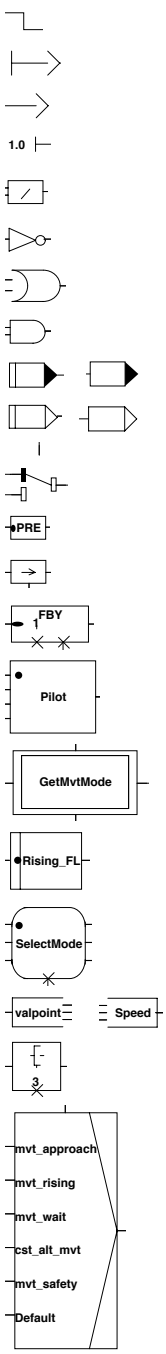
```
CODE
 WHILE true DO
   @[T_l0$ , 0]  @[T_y$ , 0]  @[T_z$ , 0]
   DCASE
     [] true :
       @[z_written$ , false]
       DCASE
         [] c1 :
           @[T_l0$ , 1]
         [] ¬(c1) :
           @[T_l0$ , 2]
       DESAC
       @[T_y$ , 1]
       DCASE
         [] (in=0) :
           @[z_written$ , true]
           @[T_z$ , 2]
         [] ¬(in=0) :
           DCASE
             [] (in=1) :
               @[z_written$ , true]
               @[T_z$ , 3]
             [] ¬(in=1) :
               DCASE
                 [] (in=2) :
                   @[z_written$ , true]
                   @[T_z$ , 4]
               DESAC
           DESAC
       DESAC
       @[out$ , (p0)]
       DCASE
         [] z_written$=false :
           @[T_z$ , 1]
       DESAC
   DESAC
   DCASE
     [] ((T_l0$=1)∧(T_y$=1)∧(T_z$=1)) :
       @[Sp0$ , 1]
     [] ((T_l0$=2)∧(T_y$=1)∧(T_z$=1)) :
       @[Sp0$ , 3]
     [] ((T_l0$=1)∧(T_y$=1)∧(T_z$=2))∨((T_l0$=1)∧(T_y$=1)∧(T_z$=3)) :
       @[Sp0$ , 5]
     [] ((T_l0$=2)∧(T_y$=1)∧(T_z$=2)) :
       @[Sp0$ , 7]
     [] ((T_l0$=1)∧(T_y$=1)∧(T_z$=4)) :
       @[Sp0$ , 9]
     [] ((T_l0$=2)∧(T_y$=1)∧(T_z$=3))∨((T_l0$=2)∧(T_y$=1)∧(T_z$=4)) :
       @[Sp0$ , 11]
   DESAC
   DCASE
     [] p0=0∧Sp0$ /= 0 :
       @[Sp0$ , Sp0$+1]
   DESAC
   DCASE
     [] ((T_z$=1)) :
       @[z_ST$ , 1]
     [] ((T_z$=2))∨((T_l0$=1)∧(T_y$=1)∧(T_z$=3)) :
       @[z_ST$ , 2]
     [] ((T_z$=4))∨((T_l0$=2)∧(T_y$=1)∧(T_z$=3)) :
       @[z_ST$ , 3]
   DESAC
 OD
END
```

Abbildung A.2: Zuweisungen zu Super-Trace-Variablen in Abhängigkeit der Propositions- und Trace-Variablen für das Beispiel in Abbildung A.1

```
DCASE
  [] y>=0.0:
    @[_src_0$, y]
    @[_positive_0$, true]
  [] not(y>=0.0):
    @[_src_0$, -(y)]
    @[_positive_0$, false]
DESAC
DCASE
  [] _src_1$≥16384.0:
    @[_tgrt_1$, _tgrt_0$+16384]
    @[_src_2$, _src_1$-16384.0]
  [] not(_src_1$≥16384.0):
    @[_tgrt_1$, _tgrt_0$]
    @[_src_2$, _src_1$]
DESAC
...
DCASE
  [] _src_13$≥4.0:
    @[_tgrt_13$, _tgrt_12$+4]
    @[_src_14$, _src_13$-4.0]
  [] not(_src_13$≥4.0):
    @[_tgrt_13$, _tgrt_12$]
    @[_src_14$, _src_13$]
DESAC
DCASE
  [] _src_14$≥2.0:
    @[_tgrt_14$, _tgrt_13$+2]
    @[_src_15$, _src_14$-2.0]
  [] not(_src_14$≥2.0):
    @[_tgrt_14$, _tgrt_13$]
    @[_src_15$, _src_14$]
DESAC
DCASE
  [] _src_15$≥1.0:
    @[_tgrt_15$, _tgrt_14$+1]
    @[_src_16$, _src_15$-1.0]
  [] not(_src_15$≥1.0):
    @[_tgrt_15$, _tgrt_14$]
    @[_src_16$, _src_15$]
DESAC
DCASE
  [] _positive_0$:
    @[_rfinal_0$, _tgrt_15$]
  [] not _positive_0$:
    @[_rfinal_0$, -(_tgrt_15$)]
DESAC
@[x$, _rfinal_0$]
```

Abbildung A.3: Explizite Konversion von $\mathbb{R}$ nach $\mathbb{Z}$ für den Wertebereich $\mathcal{D} = [-32768; 32768]$

A.3 SCADE-Operatoren



Verbindung (*connection*)

Eingangssignal des dargestellen Operators (*input*)

Ausgangssignal des dargestellten Operators (*output*)

Konstante (*constant*)

Vordefinierter Operator, auch +,-,*,/,>,<,=,≤,≥ (*predefined operator*)

Inverter (*negation-operator*)

Disjunktion (*or-operator*)

Konjunktion (*and-operator*)

Globale Variable, Lese-,Schreibzugriff

Lokale Variable, Lese-,Schreibzugriff

Auswahl, (*if-then-else-operator*)

Verzögerungsoperator (*pre-operator*)

Initialisierung (*init-operator*)

n Schritte zurückliegender Wert, untere Konnektoren sind Parameter *n* (immer konstant, hier 1) und Initialwerte (*followed-by-operator*)

Benutzerdefinierter Operator (*operator,node*)

Operator (innen) läuft mit anderer Clock (obiger Konnektor) und Ausgänge werden mit vorgegebenen Werten initialisiert (untere Konnektoren) (*condact-operator*)

Externer Operator in C (*extern operator*)

Zustandsautomat (*state machine*)

Zerlegung,Strukturierung (*break-down-operator, structuring operator*)

Projektion (*projection-operator*)

Auswahl (*case-operator*)

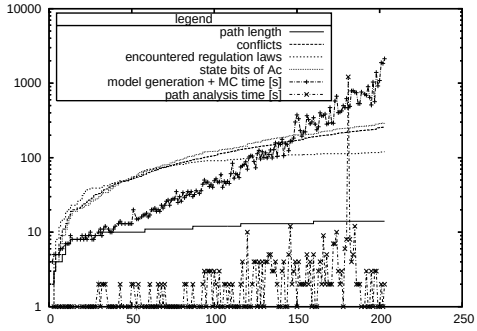
Abbildung A.4: Auswahl einiger SCADE-Operatoren und deren Funktion. Alle Operatoren haben ihre Eingänge links und Ausgänge rechts

A.3.1 Bibliotheksfunktionen für *Pilot*-Modell

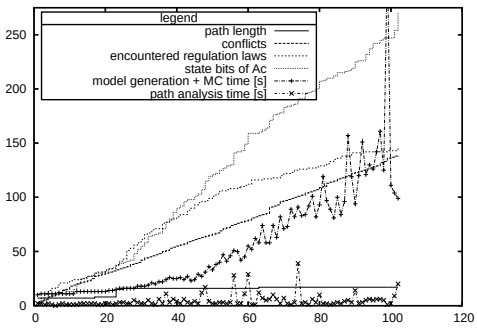
```
type
    real3 = [real , real , real] ;

function SQRT(x : real)
        returns (result : real) ;

node COUNTER(init : int ; incr : int ;
        reset : bool)
        returns (cpt : int) ;
let
  cpt = (init) -> (if reset then (init) else ((pre (cpt) + incr))) ;
tel ;

node MODULE(x : real ; y : real ;
        z : real)
        returns (norme : real) ;
var
  _L1 : real ;  _L2 : real ;  _L3 : real ;  _L4 : real ;  _L5 : real ;
  _L6 : real ;  _L8 : real ;  _L9 : real ;
let
  _L1 = x ;
  _L2 = y ;
  _L3 = z ;
  _L4 = _L1 * _L1 ;
  _L5 = _L2 * _L2 ;
  _L6 = _L3 * _L3 ;
  norme = _L8 ;
  _L8 = SQRT(_L9) ;
  _L9 = _L4 + _L5 + _L6 ;
tel ;

node Vec3Diff(A1 : real3 ; A2 : real3)
        returns (O : real3) ;
var
  _L1 : real3 ;  _L13 : real ;  _L14 : real ;  _L15 : real ;  _L2 : real3 ;  _L3 :
      real3 ;
  _L4 : real ;   _L5 : real ;   _L6 : real ;    _L7 : real ;  _L8 : real ;    _L9 :
      real ;
let
  _L1 = A1 ;
  _L2 = A2 ;
  O = _L3 ;
  _L3 = _TO_real3(_L13 , _L14 , _L15) ;
  _L4 , _L5 , _L6 = _FROM_real3(_L1) ;
  _L7 , _L8 , _L9 = _FROM_real3(_L2) ;
  _L13 = _L4 - _L7 ;
  _L14 = _L5 - _L8 ;
  _L15 = _L6 - _L9 ;
tel ;

node Vec3Mult(A : real3 ; k : real)
        returns (O : real3) ;
var
  _L1 : real3 ;  _L2 : real ;  _L3 : real ;  _L4 : real ;  _L5 : real ;
  _L6 : real3 ;  _L7 : real ;  _L8 : real ;  _L9 : real ;
let
  _L1 = A ;
  _L2 = k ;
  _L3 = _L7 * _L2 ;
  _L4 = _L8 * _L2 ;
  _L5 = _L9 * _L2 ;
  O = _L6 ;
  _L6 = _TO_real3(_L3 , _L4 , _L5) ;
  _L7 , _L8 , _L9 = _FROM_real3(_L1) ;
tel ;
node Vec3Norm(I : real3)
        returns (O : real) ;
var
  _L1 : real3 ;  _L13 : real ;  _L14 : real ;  _L15 : real ;  _L16 : real ;
let
  _L1 = I ;
  O = _L13 ;
  _L13 = MODULE(_L14 , _L15 , _L16) ;
  _L14 , _L15 , _L16 = _FROM_real3(_L1) ;
tel ;

node Vec3Prod(A1 : real3 ; A2 : real3)
        returns (O : real3) ;
```

```
var
  _L1 : real3 ;  _L10 : real ;  _L11 : real ;  _L12 : real ;  _L2 : real3 ;  _L3 :
      real ;
  _L4 : real ;   _L5 : real ;   _L6 : real3 ;  _L7 : real ;  _L8 : real ;   _L9 :
      real ;
let
  _L1 = A1 ;
  _L2 = A2 ;
  _L3 = _L7 * _L10 ;
  _L4 = _L8 * _L11 ;
  _L5 = _L9 * _L12 ;
  O = _L6 ;
  _L6 = _TO_real3(_L3 , _L4 , _L5) ;
  _L7 , _L8 , _L9 = _FROM_real3(_L1) ;
  _L10 , _L11 , _L12 = _FROM_real3(_L2) ;
tel ;

node Vec3Sum(A1 : real3 ; A2 : real3)
       returns (O : real3) ;
var
  _L1 : real3 ;  _L10 : real ;  _L11 : real ;  _L12 : real ;  _L13 : real ;  _L14 :
      real ;
  _L15 : real ;  _L2 : real3 ;   _L6 : real3 ;  _L7 : real ;   _L8 : real ;   _L9 :
      real ;
let
  _L1 = A1 ;
  _L2 = A2 ;
  O = _L6 ;
  _L6 = _TO_real3(_L13 , _L14 , _L15) ;
  _L7 , _L8 , _L9 = _FROM_real3(_L1) ;
  _L10 , _L11 , _L12 = _FROM_real3(_L2) ;
  _L13 = _L7 + _L10 ;
  _L14 = _L8 + _L11 ;
  _L15 = _L9 + _L12 ;
tel ;
```

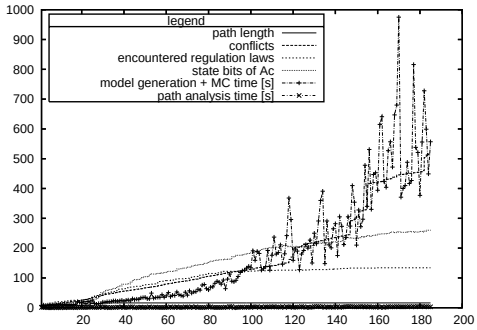
A.4 Entwicklungen der Abstraktionsverfeinerung



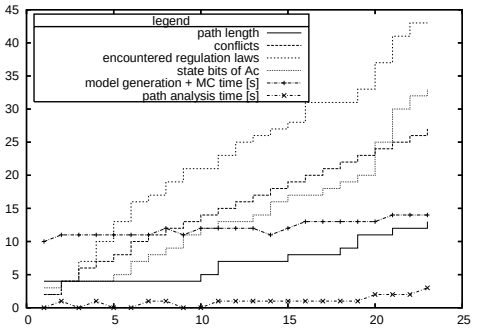
(a) Beweisaufgabe φ_{10}^{APS}, log. Skalierung



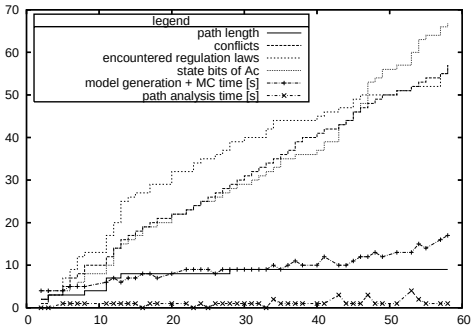
(b) Beweisaufgabe φ_{2}^{APS}



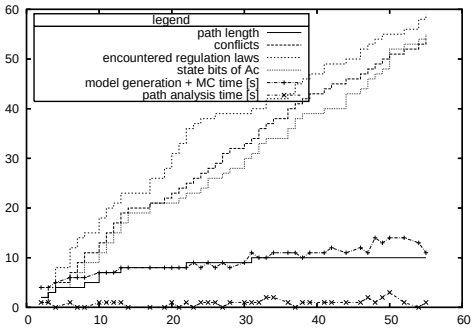
(c) Beweisaufgabe φ_{15}^{APS}



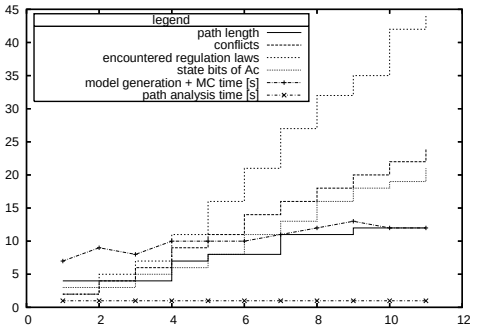
(d) Beweisaufgabe φ_{3}^{APS}



(e) Beweisaufgabe φ_{7}^{APS}

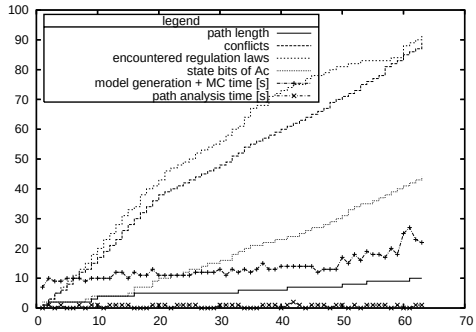


(f) Beweisaufgabe φ_{9}^{APS}

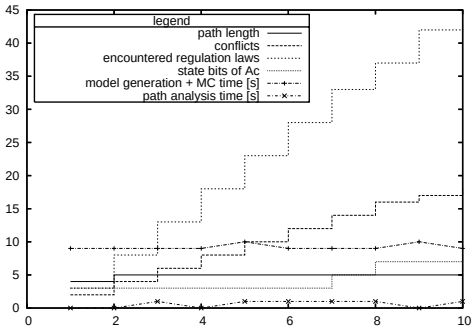


(g) Beweisaufgabe φ_{13}^{APS}

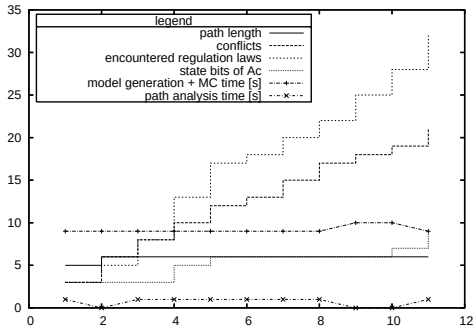
Abbildung A.5: Verläufe von Verifikationsprozessen verschiedener Beweise auf dem Autopilotsystem



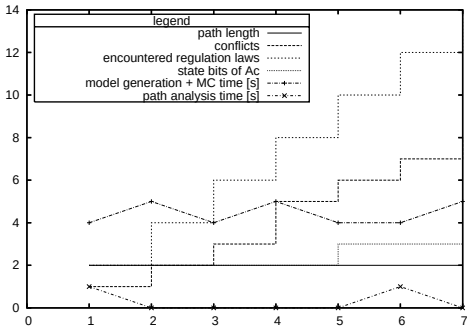
(a) Beweisaufgabe φ_1^{FHS}



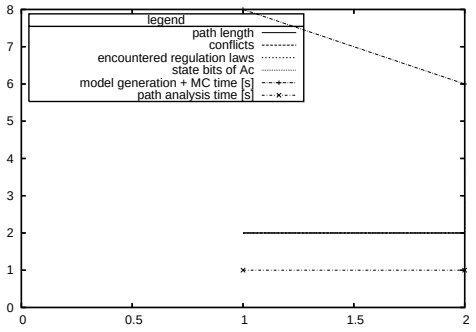
(b) Beweisaufgabe φ_5^{FHS}



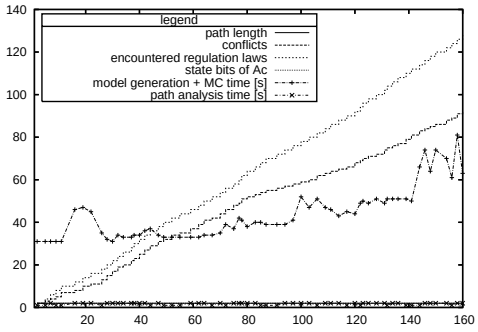
(c) Beweisaufgabe φ_7^{FHS}



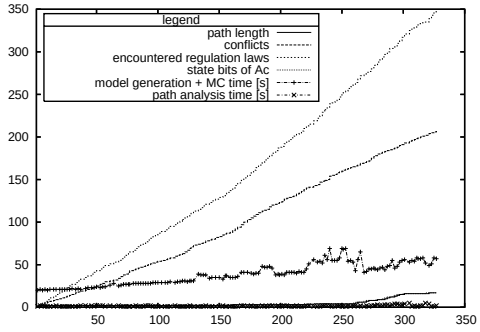
(d) Beweisaufgabe φ_2^{FCS}



(e) Beweisaufgabe φ_3^{FCS}



(f) Beweisaufgabe φ_5^{FCS}



(g) Beweisaufgabe φ_6^{FCS}

Abbildung A.6: Verläufe von Verifikationsprozessen verschiedener Beweise auf dem Fensterheber System und dem Fuelrate Controller System

Literaturverzeichnis

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. of CADE'02*, 2002.

[ADI02] R. Alur, T. Dang, and F. Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In *Fifth International Workshop on Hybrid Systems: Computation and Control*, volume 2289 of *LNCS*, pages 35–48. Springer-Verlag, 2002.

[ADI03] R. Alur, T. Dang, and F. Ivancic. Counter-example guided predicate abstraction of hybrid systems. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2003.

[ADI06] Rajeev Alur, Thao Dang, and Franjo Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *Trans. on Embedded Computing Sys.*, 5(1):152–199, 2006.

[ADM02] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid systems. In *CAV*, pages 365–370, 2002.

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1996.

[BB91] Gerard Berry and Albert Beneviste. The synchronous approach to reactive and real-time systems. In *Another Look at Real Time Programming, Proceedings of the IEEE,79*, pages 1270–1282, 1991.

[BCC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS 99*, LNCS. Springer, 1999.

[BDG+98] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-order CTL model checking. In *FSTTCS 98*, LNCS, 1998.

[BDL+01] P. Baufreton, F. Dupont, T. Lesergent, M. Segelken, H. Brinkmann, O. Strichman, and K. Winkelmann. Safeair: Advanced design tools for aircraft systems and airborne software. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*. IEEE Computer Society, July 2001.

[BDL+02] P. Baufreton, F. Dupont, R. Leviathan, M. Segelken, and K. Winkelmann. Constructing correct systems in the safeair project. In *SCI 2002*, volume VII. Multiconference on Systemics, Cybernetics and Informatics, 2002.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer, 1998.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002. Copenhagen, Denmark.

[BEN04] K. Berkelaar, M. Eikland, and P. Notebaert. Open source (mixed-integer) linear programming system. Eindhoven University of Technology, May 2004.

[Bie99] Tom Bienmüller. smi2fsm/smi2blifmv, version 0.8. Technical report, Carl von Ossietzky Universität Oldenburg, October 1999.

[Bie01] Tom Bienmüller. smi2fsm/smi2blifmv, version 1.0. Technical report, Carl von Ossietzky Universität/-OFFIS Oldenburg, 2001.

[Bie03] Tom Bienmüller. *Reducing Complexity for the Verification of Statemate Designs*. PhD thesis, Carl von Ossietzky Universität/OFFIS Oldenburg, 2003.

[BLL+96] J. Bengtsson, K. Larsen, F. Larsson, P. Petterson, and W. Yi. Uppaal: a tool suit for automatic verification of realtime systems. In *Hybrid Systems III, no. 1066*, volume 1066 of *LNCS*, pages 232–243. Springer, 1996.

[Boe86] Barry Boehm. A sprial model of program development and enhancement. *Software Engineering Notes*, 11(4):14–24, 1986.

[Bri99] Henning Brinkmann. Verifikation eines hybriden Steuersystems mit Hilfe erweiterter Abstraktionsmethoden. Master's thesis, Carl von Ossietzky Universität Oldenburg, February 1999.

[Bro99] Udo Brockmeyer. *Verifikation von* Statemate *Designs*. PhD thesis, Carl von Ossietzky Universität Oldenburg, December 1999.

[Bry92] Randal E. Bryant. Symbolic boolean Manipulation with ordered Binary-Decision Diagrams. *ACM Comp. Surveys*, (24):293–318, 1992.

[BT00] Oleg Botchkarev and Stavros Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 73–88, London, UK, 2000. Springer-Verlag.

[CFH+03] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *ACM Transactions on Programming Languages and Systems*, number 16, pages 1512–1542, September 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*, volume ISBN 0-262-03270-8. MIT Press,Cambridge, Massachusets, London, England, 1999.

[DDH+06] W. Damm, S. Disch, H. Hungar, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Automatic verification of hybrid systems with large discrete state space. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis*, volume 4218 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[DDH+07] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact state set representation in the verification of linear hybrid systems with large discrete state space. Reports of SFB/TR 14 AVACS 21, SFB/TR 14 AVACS, June 2007. ISSN: 1860-9821, http://www.avacs.org.

[DFK+03] Marcel Dhiflaoui, Stefan Funke, Carsten Kwappik, Kurt Mehlhorn, Michael Seel, Elmar Schömer, Ralph Schulte, and Dennis Weber. Certifying and repairing solutions to large LPs: How good are LP-solvers? In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 255–256, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[DGV99] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pages 249–260. Springer-Verlag, 1999.

[DJ90] G. Das and G. Joseph. The complexity of minimum convex nested polyhedra. In *Proc. 2nd Canadian Conference on Computational Geometry*, pages 296–301, 1990.

[DMO+07] Werner Damm, Alfred Mikschl, Jens Oehlerking, Ernst-Rüdiger Olderog, Jun Pang, André Platzer, Marc Segelken, and Boris Wirtz. Automating verification of cooperation, control, and design in traffic applications. In Cliff Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *LNCS*, pages 115–169. Springer, 2007.

[dMRRS03] L. de Moura, H. Ruess, J. Rushby, and N. Shankar. Embedded deduction with ics. In Brad Martin, editor, *HCSS'03—High Confidence Software and Systems Conference*, Baltimore, MD, 1-3 April 2003.

[dMRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of the 15th Computer-Aided Verification conference (CAV'03)*, volume 2725 of *LNCS*, 2003.

[DSS+03] W. Damm, C. Schulte, M. Segelken, H. Wittke, U. Higgen, and M. Eckrich. Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. *Lecuture Notes in Informatics*, P-34:340–345, May 2003.

[ESt97] Bundesministerium des Inneren EStdIT. V-Model, Development Standard for IT-Systems of the federal Republic of Germany, 1997.

[FCJK05] Ansgar Fehnker, Edmund M. Clarke, Sumit Kumar Jha, and Bruce H. Krogh. Refining abstractions of hybrid systems using counterexample fragments. In *HSCC*, pages 242–257, 2005.

[FH05] M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Electronic Notes in Theoretical Computer Science*, 133:119–137, 2005.

[Fre05] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems past HyTech. In *HSCC*, pages 258–273, 2005.

[FS01] R.Bloem F. Somenzi. Wring: an LTL to buechi translator. Tool can be download from http://vlsi.colorado.edu/ rbloem/wring.html, June 2001.

[GGB+03] T. Gaudre, H. Guillermo, P. Baufreton, D. Goshen, J. Cruz, F. Dupont, R. Leviathan, M. Segelken, K. Winkelmann, and N. Halbwachs. A methodology and a tool set designed to develop aeronautics, automotive and safety critical embedded control-systems. In *Convergence 2003*, 2003.

[GKvV95] J. F. Groote, J. W. C. Koorn, and S. F. M. van Vlijmen. The safety guaranteeing system at station hoorn-kersenboogerd. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 57–68, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Gra] Mark Grand. *Java Language Reference*. O'Reilly & Associates Inc., 2 edition.

[Gro96a] The VIS Group. VIS : A System for Verification and Synthesis. In *8th international Conference on Computer Aided Verification*, number 1102 in LNCS, 1996. `http://www-cad.eecs.Berkeley.EDU/~vis`.

[Gro96b] The VIS Group. VIS: A System for Verification and Synthesis. In *FMCAD*, 1996.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HH94] Thomas A. Henzinger and Pei-Hsin Ho. Hytech: The cornell hybrid technology tool. In *Hybrid Systems*, pages 265–293, 1994.

[HHMWT00] Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HY-TECH: Hybrid systems analysis using interval numerical methods. In *HSCC*, pages 130–144, 2000.

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1990.

[Hyd97] Daniel C. Hyde. *CSCI 320 Computer Architecture Handbook on Verilog HDL*. Computer Science Department, Bucknell University, Lewisburg, PA 17837, August 1997.

[Iee00] Ieee. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*. IEEE, 2000.

[KR90] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Carl Hanser Verlag und Prentice-Hall International, 2. edition, 1990. ANSI C.

[Kuk96] Yuji Kukimoto. BLIF-MV format description. web page: http://vlsi.colorado.edu/~vis/doc/blifmv/blifmv/blifmv.html, November 1996.

[Mak03] Andrew Makhorin. Glpk (gnu linear programming kit). http://www.gnu.org/software/glpk/glpk.html, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia, 2003.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, june 1992. Springer-Verlag.

[OS03] Sam Owre and N. Shankar. Writing PVS proof strategies. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number CP-2003-212448 in NASA Conference Publication, pages 1–15, Hampton, VA, September 2003. NASA Langley Research Center.

[R.E86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[RGA+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *CAV*, volume 1102, pages 428–432. Springer Verlag, 1996.

[RTC92] RTCA. RTCA/DO-178B, Software Consideration in Airborne Systems and Equipment Certification, RTCA-Requirements and Technical Concepts for Aviation, 1992.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *CAV*, number 1855 in LNCS, pages 248–263. Springer Verlag, July 2000.

[Seg07] Marc Segelken. Abstraction and Counterexample-guided Construction of ω-Automata for Model Checking of Step-discrete linear Hybrid Models. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *LNCS*, pages 433–448. Springer-Verlag, July 2007.

[SFHK04] O. Stursberg, A. Fehnker, Z. Han, and B. H. Krogh. Verification of a cruise control system using counterexample-guided search. In *Control Engineering Practice*. Elsevier Science, 2004.

[SK00] B.I. Silva and B.H. Krogh. Formal verification of hybrid systems using checkmate: a case study. In *Proceedings of the American Control Conference*, pages 1679–1683, 2000.

[TD95] S. Tucker Taft and Robert A. Duff. *Ada 95 Reference Manual*. Number 1246. Springer-Verlag, 1995.

[The03] The MathWorks, Inc., 3 Apple Hill Drive Natick, MA 01760-2098. *Stateflow and Stateflow Coder, User's Guide*, 5 edition, january 2003.

[VER98] VERILOG SA. *SCADE Data Flow Editor - User's Manual*, version 2.1 edition, June 1998.

[WBBL02] Bernd Westphal, Tom Bienmueller, Juergen Bohn, and Rainer Lochmann. The world's ugliest language a.k.a smi. syntax and semantics. Technical report, OFFIS Oldenburg, University of Oldenburg, 2002.

[Wit99] Gunnar Wittich. *Ein problemorientierter Ansatz zum Nachweis von Realzeiteigenschaften eingebetteter Systeme*. PhD thesis, Carl von Ossietzky Universität Oldenburg, August 1999.

[Wit06] Hartmut Wittke. *An Environment for Compositional Specification Verification of Complex Embedded Systems*. PhD thesis, Carl von Ossietzky Universität/OFFIS Oldenburg, 2006.

[WW99] Steven A. Wolfman and Daniel S. Weld. The LPSAT engine & its application to resource planning. In *IJCAI*, pages 310–317, 1999.

Lebenslauf

6.11.1971	geboren in Bremen
1978-1992	Schulausbildung an der Grundschule Lemwerder / Eschofschule Lemwerder / Gymnasium a. d. Willmsstrasse in Delmenhorst. Erlangung des Abiturabschlusses
1992-1993	Grundwehrdienst in Varel und Delmenhorst
1993	Beginn des Informatikstudiums an der Carl von Ossietzky Universität Oldenburg
1996	Vordiplom in Informatik
2000	Diplom in Informatik mit Nebenfach Physik. Thema der Diplomarbeit: „Bewertung von Lastverteilungsalgorithmen und Bindungsmodellen für parallele Implementierungen funktional-logischer Programmiersprachen unter besonderer Berücksichtigung verschiedener Zielarchitekturen"
2000 - 2007	Wissenschaftlicher Mitarbeiter beim Institut OFFIS e. V. in Oldenburg im Bereich Sicherheitskritische Systeme
2007	Promotion