RWTH Aachen

Department of Computer Science
Technical Report

Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation

Daniel Klünder

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2009-04

RWTH Aachen · Department of Computer Science · February 2009

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von
Diplom-Ingenieur
Daniel Klünder
aus
Boppard / Rhein-Hunsrück

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. Ursula Goltz

Tag der mündlichen Prüfung: 03.02.2009

Daniel Klünder
Lehrstuhl Informatik 11
kluender@cs.rwth-aachen.de

Aachener Informatik Bericht AIB-2009-04

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstraße 55
52074 Aachen
GERMANY

ISSN 0935-3232

We should be careful to build a
world we actually want to live in.

(Richard Gold: The Plenitude)

Zusammenfassung

Funktionalität und Qualität eingebetteter Systeme werden in immer stärkerem Maße durch Software bestimmt. Aufgrund ihrer immer stärkeren Verbreitung und Vernetzung kommt es zu einer Komplexitätssteigerung, deren Beherrschung zu einem zentralen Wettbewerbsfaktor wird. Die klassischen Methoden des Software-Engineering stoßen bei der Entwicklung eingebetteter Systeme jedoch an ihre Grenzen. In dieser Arbeit werden deswegen Entwurfsmodelle aus dem Software-Engineering um Eigenschaften erweitert, die typisch für den Entwurf eingebetteter Systeme sind.

Zur Verhaltensmodellierung werden abstrakte Zustandsmaschinen (ASMs) eingesetzt und um die abstrakte Darstellung von Regelalgorithmen durch Nutzung der nichtdeterministischen Wahl erweitert. Zur Verifikation sicherheitskritischer Systeme wird ein Model-Checker entwickelt, der erstmals in der Lage ist, ASMs direkt zu unterstützen. Dazu wird der ASM-Simulator CoreASM in den Model-Checker [mc]square eingebaut. Vorteile des so implementierten Model-Checkers sind die komplette Unterstützung der ASM-Syntax des Simulators, die einfache Erweiterbarkeit sowie die Möglichkeit, Gegenbeispiele und Zeugen direkt im ASM-Zustandsraum anzeigen lassen zu können. Nachteilig wirkt sich aus, dass die Laufzeiteigenschaften des Simulators nicht für seine Verwendung in einem Model-Checker optimiert sind, sondern auf Verständlichkeit und Erweiterbarkeit hin ausgelegt sind.

Zur Strukturmodellierung wird die Business Object Notation (BON) eingesetzt und zur Kombination mit ASMs um die Repräsentation nebenläufiger Klassen erweitert. Die vorgeschlagene Formulierung von Plattformanforderungen in Vorbedingungen eignet sich nur für kurze Methoden, während die Darstellung von Zeitanforderungen in Nachbedingungen auch für lange Methoden nützlich ist.

Um neben den ASMs auch die zeitbehafteten, nebenläufigen BON-Modelle semiautomatisch in eine Implementierung übersetzen zu können, wird eine Laufzeitumgebung entwickelt, die *simple concurrent object-oriented programming* (SCOOP) auf einem Echtzeitbetriebssystem implementiert und um die Überprüfung der Zeitanforderungen in den Nachbedingungen erweitert. Wegen der Unvorhersagbarkeit von Ausführungszeiten eignet sie sich nur für weiche Echtzeitsysteme und erhöht die Wiederverwendbarkeit des Systems auf Kosten seiner Leistung.

Zur Evaluation des Ansatzes wird ein Abstandsregeltempomat, der zwei Regler, die Interaktion mit der Umwelt über Sensoren und Aktoren, Zeitanforderungen und sicherheitskritische Funktionen beinhaltet, erfolgreich implementiert.

Abstract

The ever increasing presence of information technology along with the pervasion of hardware and software into everyday life boosts the need for engineering methods and tools for the development of high quality embedded systems. This trend is further amplified by the rising complexity of such systems and their usage for safety critical tasks. However, classical software engineering methods lack support for the peculiarities of embedded systems. This thesis therefore extends these methods with support for verification and requirements that arise through interaction with the physical world.

Abstract state machines (ASMs) are used for functional modeling and extended with an abstract notion for feedback control algorithms by utilizing non-deterministic choose. For their verification this thesis presents a novel approach to model checking ASMs that directly supports them by utilizing their notion of run. The simulator CoreASM is adapted to branch into all possible successor states and integrated into the model checker [mc]square. On the one hand, this enables the approach to present counterexamples and witnesses directly as sequences of ASM states, to be easily extendable, and at the same time supports the whole CoreASM syntax. On the other hand, it suffers from the simulator's design that was built with the goals of comprehensiveness and extendibility in mind and hence is not optimized for performance in a model checker.

The Business Object Notation (BON) is used for structural modeling and extended with the representation of concurrent classes. Requirements that arise through the execution of the embedded system on a physical platform are represented in preconditions while those that arise through reactions to the physical environment are modeled in postconditions. The former is only useful for short methods.

While ASMs can easily be transformed into an implementation, timed and concurrent BON models need an extra runtime environment for a semi-automatic translation. Therefore, simple concurrent object-oriented programming (SCOOP) is implemented on a real-time operating system and extended with the runtime checking of time assertions in postconditions. Because of the unpredictability of runtimes it is only practical for soft real-time systems and improves the system's reusability at the expense of its resource usage.

For evaluating the approach an adaptive cruise control which includes two closed loop controllers, interaction with the environment via sensors and actuators, time requirements, and safety critical functions is successfully implemented.

Danksagung

Ohne Hilfe und Unterstützung wäre diese Dissertation nicht möglich gewesen. Daher danke ich Herrn Prof. Dr.-Ing. Stefan Kowalewski für die Möglichkeit, meine Dissertation zu schreiben, Frau Prof. Dr. Ursula Goltz für ihre Bereitschaft, ein Gutachten meiner Dissertation zu verfassen, Herrn Prof. Dr.-Ing. Nagl, Herrn Prof. Dr. Seidl und Herrn Prof. Rossmanith für ihre Teilnahme an der Prüfungskommission, meinen Kollegen vom Lehrstuhl Informatik 11 für die gute Arbeitsatmosphäre, Bastian Schlich, Dirk Wilking und Falk Salewski für den gemeinsamen Weg, den Studenten Konrad Gerhards, Rafael Pabst, Ralf Mitsching, Joan Torres, Alexander Michailidis, Raymond Martens, Max Petel, Jing Da, Jin Sun, Dimitri Kameni, Jörg Beckers, Zheng Hu, Shun Yang und Mudassir Rasool für ihren Einsatz und der Gemeinschaft der ASM-Forscher für immer freundlichen Rat.

Meiner Familie danke ich für stete Unterstützung.

Nora danke ich für wundervolle Jahre. Ihr soll dieses Buch gewidmet sein.

Daniel Klünder
Stuttgart, Februar 2009

Inhaltsverzeichnis

1 Einleitung

Motorsteuerung, Antiblockiersystem, Abstandsregeltempomat, elektronisches Stabilitätsprogramm und viele weitere – in kaum einem Anwendungsfeld ist die Verbreitung eingebetteter Softwaresysteme in unserem Alltag ähnlich deutlich wie im Automobil. Die Entwicklungen in dieser Domäne stehen dabei beispielhaft für viele weitere Anwendungsfelder eingebetteter Systeme: Durch ihre immer stärkere Verbreitung und Vernetzung vor allem im Bereich Komfort und Fahrerassistenz wird ein Komplexitätsgrad erreicht, der dem von Desktop- und Enterprise-Software in nichts mehr nachsteht. Heutzutage, etwa 30 Jahre nach Auslieferung der ersten noch wenige Kilobyte großen Software zur Motorsteuerung, ist die Anzahl der Quelltextzeilen nahezu exponentiell auf bis zu mehrere 10 Millionen angewachsen [BKPS07]. Ein aktuelles Fahrzeug des Premiumsegments implementiert beispielsweise etwa 270 kundensichtbare Funktionen, die auf über 70 Steuergeräte verteilt sind und sich zu insgesamt mehreren hundert Megabyte Binärcode addieren und schon bald die Grenze zu einem Gigabyte überschreiten werden [SZ06].

In Zukunft wird sich diese Entwicklung eher verstärken denn abschwächen. Studien gehen heute davon aus, dass 90 Prozent der zukünftigen Innovationen im Automobil auf der Elektronikentwicklung und davon wiederum 80 Prozent auf der Softwareentwicklung basieren werden [LHFB02]. Gleichzeitig verkürzen sich die Entwicklungszeiten durch die immer schnellere Abfolge von Fahrzeuggenerationen. Entsprechend entwickeln sich die Elektronikkosten zum Hauptkostentreiber in der Entwicklung eines Automobils. Für das Jahr 2015 werden für sie bis zu 35 Prozent Anteil an den Gesamtentwicklungskosten prognostiziert [EK07].

Damit wird die Softwareentwicklung zu einem wettbewerbsentscheidenden Faktor und zu einer Kernkompetenz für die Automobilindustrie. Gründe für den vermehrten Einsatz von Software zur Unterstützung oder als Ersatz für bisher mechanische Systeme sind verschärfte Abgas- und Sicherheitsbestimmungen, Individualisierungsbedürfnisse der Kunden und ganz allgemein die Realisierungsmöglichkeiten von Systemen, die ohne Software kaum vorstellbar wären.

Methoden und Werkzeuge zur Erzeugung und Ausführung von Programmen wie Entwicklungsumgebungen oder Kompilierer gibt es mittlerweile zur Genüge. Auch die Entwicklung und Verwendung immer leistungsfähigerer Prozessoren und Hardware wird als selbstverständlich angesehen. Durch die oben beschriebene Komplexitätssteigerung, insbesondere durch den hohen Vernetzungsgrad vieler Systeme, wird es jedoch zur großen Herausforderung, korrekte, sichere, verlässliche und vertrauens-

würdige Systeme herzustellen. Dies wird in der Informatik durch das Gebiet des ingenieursmäßigen Entwurfs komplexer Softwaresysteme, des sogenannten Software-Engineerings, unterstützt, welches Methoden und Prozesse bereitstellt, um Software klar strukturiert zu entwerfen und unerwünschte Nebeneffekte zu vermeiden.

1.1 Motivation

Im Software-Engineering wird beim Übergang von umgangsprachlichen, informellen Anforderungen zum Architekturentwurf wie in Abbildung 1.1 dargestellt zwischen funktionalen und qualitativen Anforderungen an ein Produkt unterschieden. Neben der reinen Funktionalität strebt das Software-Engineering danach, qualitative Anforderungen zu erfüllen, wie z. B. Zuverlässigkeit, Integrierbarkeit, Wartbarkeit, Testbarkeit oder Änderbarkeit. Qualitative Anforderungen sind erfahrungsgemäß schwieriger zu analysieren aber entscheidend für den Markterfolg von softwareintensiven Systemen. Dabei werden Qualitätsmerkmale, die von den Anwendern eines Systems wahrgenommen werden können, auch als externe Faktoren bezeichnet [MRW+77, Mey97]. Darunter fallen alle in der Norm ISO 9126:2001 [ISO01] aufgeführten Merkmale, wie z. B. Zuverlässigkeit, Effizienz oder Änderbarkeit. Als interne Faktoren werden solche bezeichnet, die nur für Softwareexperten sichtbar sind, wie z. B. Lesbarkeit oder Modularität, die also Zugang zu Quelltexten oder Entwurfsdokumenten voraussetzten.

Beim Entwurf komplexer Softwaresysteme ist die modellbasierte Softwareentwicklung [Bru95], also der frühe und durchgehende Einsatz von Modellen, heute Stand der Technik. Diese Modelle werden in Verhaltens- und Strukturmodelle unterteilt. Die Erkenntnis, dass die Architektur eines Systems, also seine Grobstrukturen, einen Haupteinfluss auf die externen Qualitäten hat, hat sich heute durchgesetzt [SG96]. Beim Entwurf einer Architektur helfen Heuristiken in Form von Architekturmustern, die bestimmte Qualitäten unterstützen. Dazu kann ein Architekt aus einer Menge an dokumentierten Mustern, die zum Teil domänenspezifische Lösungen enthalten, wählen. Die Kombination mehrerer Muster zu einer Gesamtarchitektur, die die qualitativen Anforderungen erfüllt, ist wegen der wechselseitigen Beeinflussung der Qualitäten durch die Muster dennoch ein Prozess, dessen Erfolg in großem Maße von der Erfahrung des Architekten abhängt.

Da Fehler, die in den frühen Phasen des Lebenszyklus eines großen Softwareprojekts erkannt werden, nur etwa ein hundertstel der Kosten von Fehlern verursachen, die erst beim Betrieb erkannt werden [Nag90], wird eine frühe Validierung der Modelle angestrebt. Im Automobil sind auch sicherheitskritische Funktionen, wie z. B. der Airbag, in Software implementiert. Beim Entwurf solcher Systeme müssen Sicherheitsnormen wie die IEC 61508 [IEC05] und von ihr abgeleitete beachtet werden. Diese verlangen den semiformalen Entwurf sowohl der Struktur als auch

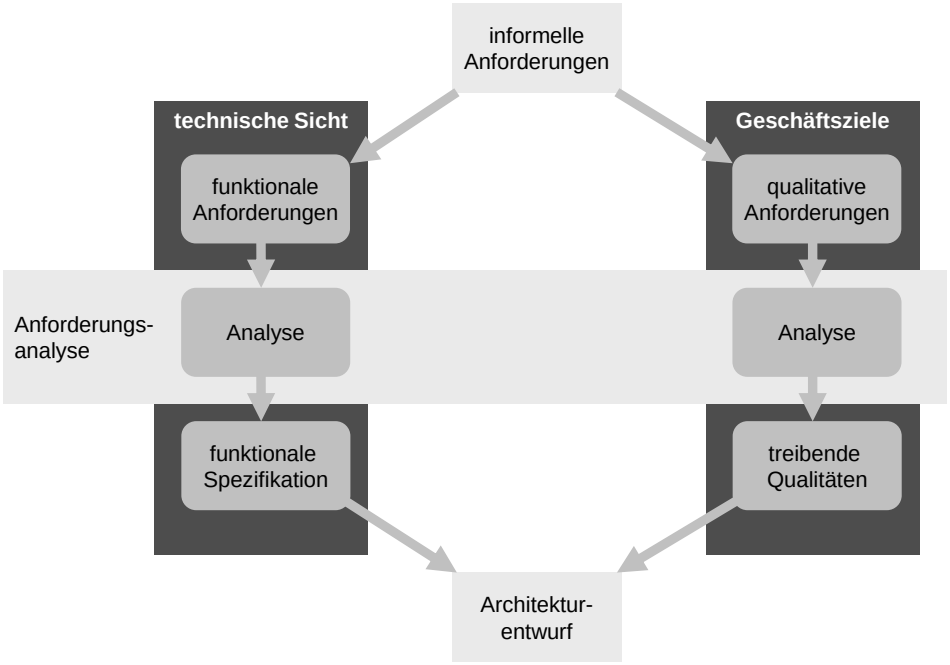

Abbildung 1.1: Unterscheidung funktionaler und qualitativer Anforderungen beim Software-Entwurf.

des Verhaltens von Software für sicherheitskritische, eingebettete Systeme. Mit Hilfe dieser Modelle können die Systeme nicht nur validiert sondern auch formal verifiziert werden, um fatale Ereignisse wie die Ariane 5 Katastrophe [L+96] oder die Entstehung hoher Kosten wie beim Toyota Prius [Gar05] zu vermeiden. Die Beobachtung, dass 41,8% der Wartungskosten, die ihrerseits etwa zwei Drittel des Gesamtaufwands eines Lebenszyklus ausmachen, auf Änderungen der Anforderungen zurückzuführen sind [LS80], verlangt von den eingesetzten Modellen gleichzeitig Flexibilität gegenüber Anforderungsänderungen.

In der Automobilindustrie gibt es erste Ansätze, Methoden des Software-Engineerings als modellbasierte Softwareentwicklung umzusetzten [SZ06]. Die klassischen Methoden des Software-Engineering stoßen laut Henzinger und Sifakis bei der Entwicklung eingebetteter Systeme jedoch an ihre Grenzen [HS06]: Ein eingebettetes System ist ein Ingenieursprodukt, welches durch Interaktion mit der umgebenden Welt physikalischen Einschränkungen unterworfen ist. Henzinger und Sifakis unterscheiden dabei Einschränkungen durch Reaktion auf die Umgebung, wie z. B. Durchsatz oder Echtzeitbedingungen, und Einschränkungen durch die Ausführung auf einer physikalischen Plattform, wie z. B. Schranken der verfügbaren Energie oder des verfügbaren Speicherplatzes. Daher fordern sie, beim Entwurf eingebetteter Sys-

teme sowohl Software, als auch Hardware und Umwelt des Systems zu modellieren, wobei alle drei physikalischen Einschränkungen unterliegen. Der Autor stimmt mit ihnen überein, dass die Trennung von Berechnungen in der Software und der Physik in Ausführungsplattform und Umwelt nicht funktioniert. Auf Seiten des Softwareentwurfs entstehen Schwierigkeiten bei der Beherrschung von Nebenläufigkeit und bei der Integration der oben genannten physikalischen Einschränkungen in den Entwurf.

1.2 Zielsetzung

Diese Arbeit verfolgt das Ziel, Modelle zum Entwurf eingebetteter Software bereitzustellen, die möglichst früh im Softwareentwicklungsprozess, d. h. bereits bei der Anforderungsmodellierung, einsetzbar sind, aber auch darüber hinaus bis hin zur Implementierung verfeinert werden können. Dabei wird kein bestimmter Entwurfsprozess vorausgesetzt; insbesondere soll das beschriebene Vorgehen auch für iterative Herangehensweisen geeignet sein. Ein möglichst nahtloser Übergang zwischen dem ersten Softwareentwurf und detailierteren Modellen bis hin zur Implementierung soll in beide Richtungen möglich sein, um einen durchgängigen Entwicklungsprozess zu ermöglichen. Dadurch soll eine Alterung von Entwurfsdokumenten beispielsweise bei Anforderungsänderungen oder Wartungsarbeiten durch möglichst automatische Transformierbarkeit der Änderungen in alle übrigen Dokumente verhindert werden.

Um die dabei übliche und oben beschriebene Unterscheidung von funktionalen und qualitativen Anforderungen zu unterstützen, sollen Verhalten und Struktur des Systems in separaten Modellen abgelegt werden können. Da eingebettete Systeme häufig für sicherheitskritische Aufgaben verwendet werden, sollen die Verhaltensmodelle möglichst früh gegen die Anforderungen verifiziert werden können. Die Strukturmodelle sollen die Darstellung verschiedener Sichten auf das System erlauben.

Da eingebettete Systeme mit der physischen Welt interagieren, soll die von Henzinger und Sifakis geforderte Unterstützung von Plattform- und Echtzeiteigenschaften [HS06] in den Modellen möglich sein. Dazu sollen geeignete Modelle, die sich bei der klassischen Softwareentwicklung für Desktop- oder Enterprisesysteme bewährt haben, um diese Eigenschaften erweitert werden. Die Arbeit soll dabei nicht auf einem abstrakten Modellniveau stehen bleiben, sondern die Implementierbarkeit der Modelle aufzeigen. Dazu soll eine Laufzeitumgebung bereitgestellt werden, die einen nahtlosen Übergang in die Implementierung eines Fahrerassistenzsystems erlaubt, das viele typische Eigenschaften eingebetteter Systeme in sich vereint.

1.3 Beitrag

Die Beiträge zum Erreichen der oben genannten Zielsetzung sind:

Verhaltensmodelllierung mit abstrakten Zustandsmaschinen Das Verhalten des Systems wird mit Hilfe von abstrakten Zustandsmaschinen (engl. abstract state machines - ASMs) [BS03a] modelliert. Diese werden um die frühe und daher abstrakte Darstellung bzw. Spezifikation eines Reglers — eines der typischen Aufgabengebiete eingebetteter Systeme — erweitert. Dies ist nach Wissen des Autors eine bisher noch nicht publizierte Verwendung der nicht-deterministischen Wahl in ASMs. Weiterhin beschreibt und implementiert diese Arbeit das direkte Model-Checking von ASMs durch Speicherung aller Zustände auf allen möglichen Pfaden in einem angepassten ASM-Simulator. Aufbauend auf einem bestehenden Model-Checker für einen Assemblersimulator hat dieses Vorgehen gegenüber anderen Arbeiten, die die Verifikation durch Übersetzung von ASMs in die Eingabesprache bestehender Model-Checker erreichen, den Vorteil, die komplette Syntax der ASMs zu unterstützen, sowie Gegenbeispiele und Zeugen direkt im ASM-Zustandsraum anzeigen zu können.

Strukturmodellierung mit der Business Object Notation Mit Hilfe der Business Object Notation (BON) [Wal98, WN95] wird die Struktur des Systems modelliert. Wegen der direkten Unterstützung von Design-by-Contract (DBC) können in den Invarianten, Vor- und Nachbedingungen Plattform- und Zeitanforderungen spezifiziert werden, die sowohl zur Entwurfszeit genutzt werden, als auch während der Laufzeit zum Testen und zur Überwachung des Systems bestehen können. Insbesondere die Formulierung der Zeitanforderungen als Nachbedingungen ermöglicht eine anforderungsnahe Ausdrucksmöglichkeit, die in späteren Schritten semiautomatisch bis zur Implementierung transformiert werden kann.

Laufzeitumgebung Um die Implementierungsfähigkeit des gewählten Ansatzes zu validieren wird das Nebenläufigkeitskonzept *simple concurrent object-oriented programming* (SCOOP) [Mey97] um Zeitanforderungen in der Nachbedingung erweitert. Insbesondere liefert diese Arbeit die erste dem Autor bekannte SCOOP-Implementierung auf einem Echtzeitbetriebssystem und beschreibt die notwendigen Erweiterungen des Konzepts, namentlich die Einbindung von Prioritäten und Fristen sowie die Einbindung von Interrupts, um eingebettete Systeme implementieren zu können.

Fallstudie Zur Validierung wird ein Fallstudie durchgeführt, die eine Bewertung der oben genannten Beiträge erlaubt.

1.4 Vorgehensweise und Gliederung

Der restliche Text dieser Dissertation ist folgendermaßen gegliedert: im nächsten Kapitel werden zunächst die theoretischen Grundlagen der oben beschriebenen

Beiträge beschrieben. Anschließend werden beide Pfade aus Abbildung 1.1 verfolgt: Kapitel 3 beschreibt den Pfad der funktionalen Anforderungen und der daraus resultierenden Verhaltensmodellierung und deren Verifikation mit Hilfe von ASMs. Die Strukturmodellierung der qualitativen Anforderungen in BON unter Einbindung von Plattform- und Echtzeiteigenschaften ist Inhalt von Kapitel 4. Die nötigen Schritte, um beide Modelle semiautomatisch in ausführbaren, plattformabhängigen Quelltext zu transformieren sowie die dazu benötigte Laufzeitumgebung, eine zeitbehaftete Variante von SCOOP, werden in Kapitel 5 beschrieben. Anhand des Fallbeispiels eines Abstandsregeltempomaten werden die genannten Modellierungen und ihre Transformation in eine Implementierung anschließend in Kapitel 6 bewertet, bevor Kapitel 7 eine Zusammenfassung gibt und einen Ausblick wagt.

1.5 Eigene Vorarbeiten

Einige Teile dieser Dissertation basieren auf Arbeiten, die bereits in früheren Publikationen veröffentlicht wurden: ein Überblick über das in dieser Arbeit verwirklichte Dissertationsvorhaben wurde auf einem Doktorandensymposium präsentiert [Klü08]. Die direkte Verifikation von ASMs durch Ausführen aller möglichen Simulationspfade [BKKS08] und eine Einordnung der Verwendung von ASMs in frühen Phasen des Software-Entwurfs [KK07] sind frühere Publikationen zu ASMs. Die Modellierung des Fallbeispiels Abstandsregeltempomat und erste Ansätze zur Verwendung der Business Object Notation finden sich in [KKK08]. Erste Ideen zur Verwendung von Vor- und Nachbedingung zur Formulierung von Plattformeinschränkungen sind ebenso veröffentlicht [KLK08], wie Papiere zum Einsatz von Heuristiken beim Entwurf von Software-Architekturen [Klü06a, Klü06b].

Auf verwandte Arbeiten anderer Autoren wird in dieser Dissertation jeweils separat in den einzelnen Kapiteln eingegangen.

2 Grundlagen

Dieses Kapitel führt die Grundlagen dieser Dissertation ein. Dazu gibt der nächste Abschnitt eine Einführung in ASMs und Abschnitt 2.2 beschreibt die Idee des Model-Checking. Die statischen und dynamischen Diagramme sowie Eigenschaften der BON sind Gegenstand von Abschnitt 2.3, der auch auf *Design-by-Contract* eingeht. Zum Abschluss des Kapitels gibt Abschnitt 2.4 eine Einführung in SCOOP.

2.1 Abstrakte Zustandsmaschinen

Abstrakte Zustandsmaschinen (engl. Abstract State Machines - ASMs) wurden 1984 von Yuri Gurevich als *dynamische Strukturen*, später als *Evolving Algebras* und schließlich als ASMs bezeichnet, erdacht, um die Church-Turing-These [Tur36] zu verfeinern [Gur84, Gur85, Gur88b, Gur88a, Gur91, Gur94], was für die sogenannte sequentielle ASM-These gelungen ist [Gur00]. Die ausgedehnte Modellierung der Semantik mehrerer Programmiersprachen [Bör90a, Bör92, Bör90b, BD90, Bör94, Bla92, SSB01, EGG$^+$01] und ihrer Erweiterungen [BB92, BR94a] führte zu der Erkenntnis der praktischen Anwendbarkeit von ASMs. Die Modellierung und Verifikation mehrerer Praxisbeispiele aus den Bereichen Hardwarearchitektur [BD95a, BDGR94, BM97a, BGM95, BGM94], virtuelle Maschinen [BR94b, BG94, BG95], Protokolle [BGR95, GM95, Hug95] und Steuerungssoftware [BM97b, Mea97, BBD$^+$96, GH96, BRS00] führte schließlich zur endgültigen formalen Definition [Gur95] und ihrer Überprüfung.

Durch die Anwendung von ASMs hat sich auch eine ASM-Methode etabliert, die für den Entwurf und die Analyse von Software auf hohem Abstraktionsniveau verwendet werden kann [Bör99, BS03b]. Im Folgenden werden zunächst sogenannte Basis-ASMs dargestellt und weitere Klassen von ASMs genannt. Abschnitt 2.1.2 geht auf die Verwendung von ASMs beim Softwareentwurf ein.

2.1.1 Grundlagen

Im Folgenden wird eine Einführung in ASMs gegeben, die für das weitere Verständnis ausreicht und sich an der Definition in der klassischen Einführung, dem Lipari Guide [Gur95], orientiert. Auf die formal mathematische Darstellung wird hier aus Platzgründen verzichtet. Für sie sei auf Börger und Stärk [BS03b] und die komplette Einführung von Gurevich [Gur95] verwiesen.

Diese Einführung folgt der Einteilung von ASMs nach Börger und Stärk [BS03b], die sich an praktischen anstatt historischen Gesichtspunkten orientiert. Dazu werden ASMs in Ein- und Mehragentensysteme unterteilt, wobei ein Agent immer eine ASM ausführt. Mehragentensysteme werden weiterhin nach Art der Interaktion der Agenten, namentlich synchrone und asynchrone, unterschieden. Als Basis-ASMs werden Einagentensysteme bezeichnet. Diese verfügen bereits über Modellierungsmöglichkeiten für potentiell unbeschränkten Nichtdeterminismus und Parallelität mit Hilfe der unten erläuterten Regelelemente **choose** und **forall**.

Intuitiv können Basis-ASMs als Pseudocode auf abstrakten Daten verstanden werden. Dazu werden Algebren mit Transitionssystemen verknüpft, um die folgenden Definitionen zu erhalten:

Abstrakte Zustandsmaschine Eine abstrakte Zustandsmaschine setzt sich aus einer Signatur Σ, einer Menge von initialen Zuständen, einer Menge von Regeldeklarationen und einem ausgezeichneten Regelnamen für die Startregel der Maschine zusammen.

Regeln Die Transitionsregeln einer ASM erzeugen die Aktualisierungen der Zustandsübergänge. Sie sind eine endliche Menge von Vorschriften der Form **if** *Condition* **then** *Updates*. Die Bedingung *Condition* ist eine Formel der Aussagenlogik, die als wahr oder unwahr bewertet werden kann. *Updates* ist eine Menge von Zuweisungen von Werten zu Lokationen.

Signatur Eine Signatur Σ ist eine endliche Menge von Funktionsnamen. Jeder Funktionsname hat eine Stelligkeit n, mit $n \geq 0$. Ein 0-stelliger Funktionsname ist eine Konstante. Jede Signatur enthält außerdem die statischen Konstanten $undef$, $true$ und $false$. Zum Beispiel enthält die Signatur Σ_{bool} der booleschen Algebra zwei Konstanten 0 und 1, einen einstelligen Funktionsnamen $'-'$ und zwei zweistellige Funktionsnamen $'+'$ und $'*'$.

Zustand Ein Zustand S einer Signatur Σ besteht aus einer nicht-leeren Menge U, genannt Superuniversum, welche die Elemente des Zustands enthält und aus Interpretationen der Funktionsnamen von Σ über U besteht.

Funktionen Jeder n-stellige Funktionsname f von Σ hat eine Interpretation f^S, welche eine Funktion von U^n nach U ist. Ist c eine Konstante von Σ, dann ist ihre Interpretation c^S ein Element aus U.

Die Konstanten $undef$, $true$ und $false$ sind paarweise verschiedene Elemente von U. Funktionen sind totale Funktionen und haben den voreingestellten Wert $undef$, wodurch ein unbestimmtes Objekt repräsentiert wird. Eine Relation ist eine Funktion, die immer die Werte $true$ oder $false$ hat. Der Standardwert einer Relation ist $false$.

Universen Bei der Modellierung wird das Superuniversum in kleinere Universen unterteilt. Dadurch ähneln Universen den Variablentypen in Programmiersprachen. Ein Element des Super-Universums $e \in U$ ist Element des Universums U_1, wenn $U_1^S(e) = true$ gilt. Ein Element kann in mehreren Universen enthalten sein.

Hintergründe Äquivalent zu vordefinierten Datentypen in höheren Programmiersprachen mit speziellen Elementen und vordefinierten Funktionen gibt es bei ASMs sogenannte Hintergründe (engl. Backgrounds). Ein Hintergrund kann als statisches Universum betrachtet werden, dessen Elemente implizit Teil des Zustands sind. Zusätzlich können vordefinierte Funktionen im Zustand enthalten sein.

Lokation Eine Lokation des Zustands ist ein Paar $(f, (e_1, ..., e_n))$. Dabei ist f ein n-stelliger Funktionsname und $e_1, ..., e_n$ sind Elemente von U. Der Inhalt des Speicherplatzes ist der Wert von $f^S(e_1, ..., e_n)$, die Elemente des Speicherplatzes sind die Elemente der Menge $e_1, ..., e_n$. Ein Zustand ist somit eine Funktion, die Speicherplätze auf ihren Inhalt abbildet. Um Modularisierung und schrittweise Verfeinerungen der ASM zu erleichtern, werden dynamische Lokationen – solche, die sich zur Laufzeit der ASM ändern können – weiter unterteilt nach Schreib- und Lesezugriff der ASM bzw. ihrer Umgebung. Als kontrollierte Lokationen werden solche bezeichnet, die nur von der laufenden ASM aktualisiert und gelesen werden. Beobachtete Lokationen sind solche, die von der ASM nur gelesen, nicht aber geschrieben werden. Diese dienen als Eingabe der ASM und werden von ihrer Umgebung, also anderen ASMs oder der Umwelt, aktualisiert und tauchen in der ASM in Aktualisierung nicht links vom Zuweisungsoperator auf. Lokationen, die Interaktion mit der Umgebung ausdrücken, also sowohl von der ASM als auch von ihrer Umgebung geschrieben und gelesen werden können, werden als geteilte Lokationen bezeichnet. Wegen des beiderseitigen Schreibzugriffs ist für ihre spätere Implementierung im Allgemeinen ein Protokoll nötig. Schließlich gibt es noch Ausgabelokationen, die von der ASM nur geschrieben, nicht aber gelesen werden, die also nur auf der linken Seite des Zuweisungsoperators in den Regeln der ASM auftauchen.

Aktualisierung Eine Aktualisierung des Zustands ist ein Paar (l, v), mit einem Speicherplatz l und einem Wert des Universums $v \in U$. Bei einer Aktualisierung wird der Inhalt des Speicherplatzes l auf den Wert von v gesetzt.

Konsistenz Durch Ausführen der Transitionsregeln einer ASM wird eine Menge von Aktualisierungen erzeugt, die auch leer sein kann. Die Transitionsregeln sind die Ausführungsvorschriften bzw. die Programme der abstrakten Zustandsmaschine. Da die Regeln im Allgemeinen parallel ausgeführt wer-

den kann es vorkommen, dass mehrere Aktualisierungen für den gleichen Speicherplatz existieren. Eine Menge von Aktualisierungen $Updates$ wird als konsistent bezeichnet, wenn für jeden Speicherplatz l, mit $(l, v) \in Updates$ und $(l, w) \in Updates$, $v = w$ gilt. Andernfalls ist diese Menge von Aktualisierungen inkonsistent.

Zustandsübergang Eine konsistente Menge von Aktualisierungen führt zu einem Zustandsübergang. Dabei wird jede Aktualisierung auf den aktuellen Zustand angewandt. Inkonsistente Mengen von Aktualisierungen führen nicht zu einem Zustandsübergang.

Neben diesen Grunddefinitionen gibt es einige Regelelemente für ASMs, die sich im Laufe der Zeit etabliert haben. Tabelle 2.1 zeigt beispielsweise die Standardregeln des ASM-Simulators CoreASM [FGG07], der im Rahmen dieser Arbeit benutzt wird. Hier sei insbesondere auf das Element **choose** verwiesen, das hilft Nichtdeterminismus auszudrücken. Mit diesem lassen sich beispielsweise Details von Scheduling-Algorithmen abstrahieren. Kapitel 3.1 beschreibt den Einsatz dieses Regelelementes zur abstrakten Spezifikation von Reglern.

Regelsyntax	Semantik
skip	Mache nichts.
$f(t_1, ..., t_n) := t$	Aktualisiere den Wert der Lokation $f(t_1, ..., t_n)$ auf t.
P **par** Q	P und Q werden parallel ausgewertet.
if φ **then** P **else** Q	Wenn φ wahr ist, werte P aus, sonst Q.
let $x = t$ **in** P	Weise x den Wert von t zu und werte P aus.
forall x **with** φ **do** P	Werte P parallel für jedes x aus, das φ erfüllt.
choose x **with** φ **do** P	Wähle ein x, das φ erfüllt und werte dafür P aus.
P **seq** Q	Werte P und Q sequentiell aus.
$r(t_1, ..., t_n)$	Rufe Transitionsregel r mit den Parametern t_i auf.

Tabelle 2.1: Regelelemente von CoreASM [FGG07]. P und Q sind weitere Transitionsregeln. Die verschiedenen t_i repräsentieren Terme. Ein Term ist induktiv definiert: Variablen (nullstellige Funktionen) sind Terme. Ist f eine n-stellige Funktion und sind $t_1, ..., t_n$ Terme, dann ist auch $f(t_1, ..., t_n)$ ein Term. x steht für eine Variable und φ für einen booleschen Term.

Neben Basis-ASMs sind im Rahmen dieser Arbeit noch die Klassen der Mehragentensysteme von Interesse. Diese werden, wie oben erwähnt, nochmals unterteilt in synchrone und asynchrone oder auch verteilte Mehragenten-ASMs.

Mehragenten-ASMs Mehragenten-ASMs haben ein spezielles endliches Universum von Agenten, die die virtuellen Maschinen des Programms repräsentieren. Jeder Agent $a \in Agents$ hat eine zugehörige dynamische Funktion $program$, die sein Verhalten definiert. Das Verhalten ist durch den Wert des Speicherplatzes $program(a)$ festgelegt. Letztendlich zeigt $program(a)$ auf die Transitionsregeln, die Agent a im aktuellen Zustand ausführt. Jeder Agent besitzt eine spezielle statische Funktion $self$, die für jeden Agenten auf sich selbst verweist.

Synchroner Zustandsübergang Jeder Agent einer synchronen Mehragenten-ASM führt parallel zu den anderen seine eigene Basis-ASM aus. Die Synchronisation der Zustandsübergänge der einzelnen Agenten findet über eine implizite, globale Systemzeit statt. Als globaler Zustand wird die Vereinigung aller Signaturen der einzelnen Komponenten bezeichnet.

Asynchroner Zustandsübergang Bei Ausführung einer asynchronen Multiagenten-ASM führt jeder Agent seine eigene Basis-ASM aus. Die Zustandsübergänge der Agenten müssen dabei folgende Bedingungen erfüllen:

- jeder Übergang hat nur endlich viele Vorgänger
- die Zustandsübergänge eines Agenten sind linear geordnet
- alle Linearisierungen jedes endlichen Anfangssegments der Zustandsübergänge aller ASMs einer asynchronen Multiagenten-ASM müssen zu dem selben Endzustand führen

2.1.2 ASMs im Softwareentwurf

Neben ihrer Anwendung in der theoretischen Informatik lassen sich ASMs auch im Softwareentwicklungsprozess nutzen [BM97b, BPS00]. Auf Basis der Abstraktionsmechanismen von ASMs lässt sich eine Entwurfsmethode entwickeln, die die folgenden drei Herausforderungen des Systementwurfs meistert [Bör99]:

- Formalisierung der Anforderungen
- iterative, nachverfolgbare Verfeinerung bis hin zur Implementierung
- komponentenbasiertes Vorgehen

Dazu werden wie in Abbildung 2.1 dargestellt die informellen, natürlichsprachlichen Anforderungen in einer präzisen, eindeutigen, konsistenten, vollständigen und minimalen Systembeschreibung festgehalten. Dieses erste ASM-Modell des Systems wird auch als Grundmodell [Bör04] bezeichnet. Der Abstraktionsgrad dieses Grundmodells ist frei wählbar, d. h. Universen und Regeln können in diesem Schritt so gewählt werden, dass sie eine möglichst natürliche Modellierung des Systems erlauben.

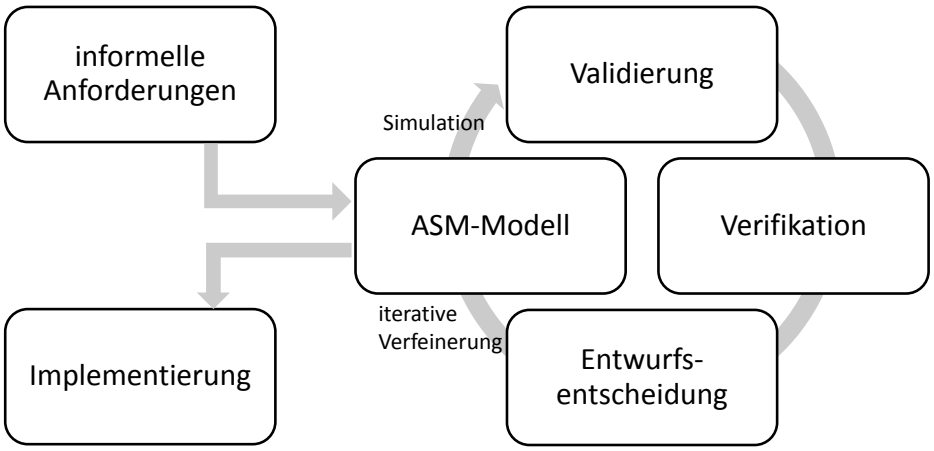

Abbildung 2.1: Softwareentwurf mit ASMs nach [Bör04].

ASMs bieten bei der Modellierung folgende Abstraktionsmechanismen an: Durch die freie Wahlmöglichkeit bei der Abstraktion von Daten als Universen muss beim Entwurf nicht in von einer Programmiersprache angebotenen Datentypen gedacht werden. Die implizit simultane Ausführung von Aktualisierungen, der synchrone Parallelismus, erlaubt die Abstraktion von sequentiellem Vorgehen, was beispielsweise beim Vertauschen zweier Lokationen hilfreich ist. Die ausdrückliche Formulierung von Nichtdeterminismus in Verbindung mit Bedingungen an diesen hilft Anforderungen an eine solche Wahl für spätere Verfeinerungen zu formulieren. Schließlich erlauben ASMs die Kompositionierung des Entwurfs über Module mit Im- und Exportbeziehungen.

Da für ASMs ein Ablaufverhalten definiert ist, können ASM-Modelle beginnend bei der abstraktesten Darstellung im Grundmodell simuliert werden. Mit Hilfe von Testfällen lässt sich ihr Verhalten so validieren. Des Weiteren kann die Simulation zur Verifikation der Modelle genutzt werden. Diese im Rahmen dieser Arbeit erstmals ausgenutzte Möglichkeit wird in Kapitel 3 näher beschrieben.

Ausgehend vom Grundmodell können ASMs immer weiter verfeinert werden. Dabei kann formal nachgewiesen werden, dass verfeinerte Modelle noch immer dem abstrakten Modell genügen [Bör03, Sch01a]. Die einzelnen Verfeinerungsschritte reflektieren dabei wichtige Entwurfsentscheidungen und können systematisch und nachverfolgbar dokumentiert werden. Schließlich kann die Verfeinerung iterativ bis zur Implementierung durchgeführt werden. Teilweise kann dies automatisch erfolgen, beispielsweise durch eine Kompilierung einer ASM zu C++ Quelltext [Sch01c].

Das in Abbildung 2.1 dargestellte und oben beschriebene Vorgehen bei der Verwendung von ASMs soll lediglich die Möglichkeiten bei ihrer Verwendung wiedergeben,

aber keinen Entwicklungsprozess vorgeben. ASMs lassen sich in viele verbreitete Entwicklungsprozesse wie beispielsweise das V-Modell integrieren [BD95b].

2.2 Model-Checking

Beim Model-Checking [CGP99, BBF+01, MP95, Sch04] wird ein Systemmodell M automatisch gegen eine formale Spezifikation φ verifiziert. Dazu werden alle erreichbaren Zustände des Modells durchsucht und hinsichtlich der Spezifikation überprüft, d. h. es wird überprüft, ob M ein Modell von φ ist ($M \models \varphi$). Die Methode wurde unabhängig voneinander von Queille und Sikakis [QS82] und Clarke und Emerson [EC81b] vorgeschlagen.

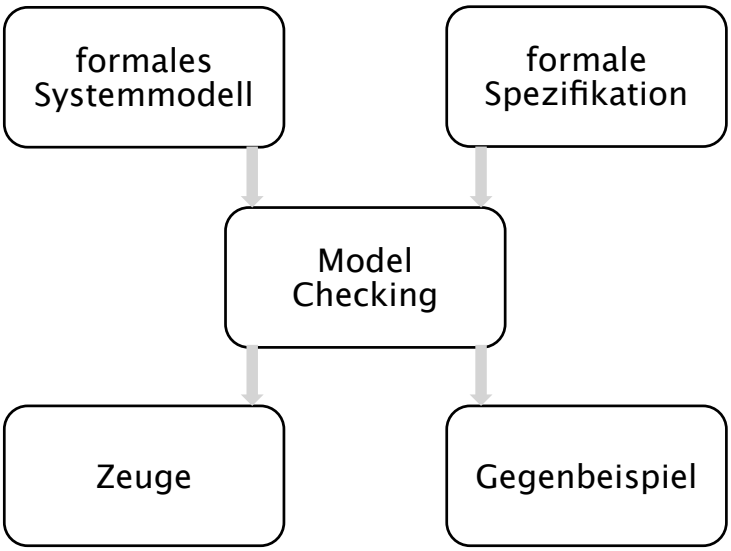

Abbildung 2.2: Allgemeiner Ablauf des Model-Checkings nach [MP95, Sch08].

Abbildung 2.2 veranschaulicht das Vorgehen. Sowohl die Systembeschreibung als auch die Spezifikation müssen in formaler Form vorliegen. Der Model-Checker sucht den erreichbaren Teil des Zustandsraums des Systemmodells vollständig ab und prüft, ob alle Zustände die Spezifikation erfüllen. Als Ergebnis der Suche liefert der Model-Checker die Antwort, ob das Systemmodell die formale Spezifikation erfüllt oder nicht sowie ein Gegenbeispiel oder einen Zeugen aus dem Zustandsraum. Bei zu großen Zustandsräumen kann es wegen Speicherbeschränkungen auch zu einem Abbruch der Suche kommen, so dass kein Ergebnis geliefert werden kann.

Im Allgemeinen muss die für das Model-Checking nötige formale Beschreibung

des Systems zunächst durch Modellierung aus einer Systembeschreibung generiert werden. Diese Beschreibung liegt dann beispielsweise als Quelltext vor und wird in eine Kripkestruktur oder einen endlichen Automaten überführt. Diese Modellierung erfolgt meist von Hand. Wegen des damit verbundenen Aufwands und der Fehleranfälligkeit dieses Vorgehens ist es jedoch erstrebenswert, es zu automatisieren. Im Rahmen dieser Dissertation werden Systembeschreibungen, die als ASM vorliegen, durch Simulation automatisch in eine Kripkestruktur überführt.

Die ebenfalls benötigte formale Spezifikation ergibt sich durch eine Formalisierung der Anforderungen, die in temporärer Logik ausgedrückt wird. Im Rahmen dieser Arbeit wird die Computation Tree Logic (CTL) benutzt, die im Folgenden kurz vorgestellt wird.

Temporale Logiken [Pnu81] erlauben im Gegensatz zur Aussagenlogik, die nur Schlussfolgerungen innerhalb einzelner Zustände zulässt, auch Aussagen über Abfolgen von Zuständen. Beispiele für temporale Logiken sind die Linear Time Logic (LTL) [Pnu81] und die Computation Tree Logic (CTL) [EC81b, EC81a]. Im Folgenden wird kurz auf CTL eingegangen. Eine ausführliche Darstellung findet sich bei Clarke et al. [CGP99].

Zur Formulierung der logischen Aussagen gibt es in CTL neben den üblichen booleschen Junktoren temporale Operatoren und Pfadquantoren. Tabelle 2.2 listet die temporalen Operatoren der CTL für Spezifikationen φ und ψ auf.

Operator	Bedeutung
G φ	es gilt immer φ (**G**lobally φ)
F φ	es gilt irgendwann φ (**F**uture φ)
X φ	im nächsten Zustand gilt φ (Ne**X**t φ)
φ **U** ψ	es gilt solange φ bis ψ gilt (φ **U**ntil ψ)

Tabelle 2.2: Temporale Operatoren in CTL

Mit ihnen lassen sich Aussagen über einzelne Pfade treffen. Um des weiteren Aussagen über den Zustandsraum treffen zu können, werden zusätzlich die in Tabelle 2.3 aufgeführten Pfadquantoren benutzt.

Operator	Bedeutung
A φ	es gilt auf allen Pfaden φ (**A**lways φ)
E φ	es gibt einen Pfad, auf dem φ gilt (**E**xist φ)

Tabelle 2.3: Pfadquantoren der CTL

Durch spezielle Kombinationen von temporalen Operatoren und Pfadquantoren

entstehen CTL-Operatoren. Diese dürfen jedoch nur aus einem Pfadquantor (E oder A) gefolgt von einem temporalen Operator (G, F, X, U), welcher sich dann auf den betrachteten Pfad bzw. auf die betrachteten Pfade bezieht, bestehen. Damit ergeben sich insgesamt acht verschiedene Möglichkeiten, die in Tabelle 2.4 aufgeführt sind. CTL-Formeln werden induktiv aus CTL-Operatoren, den Aussagevariablen und booleschen Junktoren aufgebaut.

CTL-Operator	Bedeutung
AG φ	auf allen Pfaden gilt immer φ
EG φ	es gibt einen Pfad auf dem immer φ gilt
AF φ	auf allen Pfaden gilt irgendwann φ
EF φ	es gibt einen Pfad auf dem irgendwann φ gilt
AX φ	auf allen Pfaden gilt im nächsten Zustand φ
EX φ	es gibt einen Pfad auf dem im nächsten Zustand φ gilt
A (φ **U** ψ)	auf allen Pfaden gilt φ bis ψ gilt
E (φ **U** ψ)	es gibt einen Pfad auf dem φ gilt bis ψ gilt

Tabelle 2.4: CTL-Operatoren

2.3 Business Object Notation

Die Business Object Notation (BON) [Wal98, WN95] stellt Konzepte, Regeln und Notationen für den objektorientierten Entwurf und die Analyse von Software bereit. Dabei wird ein besonderes Augenmerk auf die Wiederverwendbarkeit, Erweiterbarkeit und Wartbarkeit des Entwurfs und seiner Implementierung gelegt. Grundlegendes Prinzip ist der nahtlose Übergang von Anforderungen zum Entwurf und weiter zur Implementierung und damit die Kommunikation aller am Entwicklungsprozess Beteiligter sowie die Verfolgbarkeit von Anforderungen durch den Prozess. Durch die Möglichkeit der Rückkopplung von der Implementierung in die Notation wird verhindert, dass das Entwurfsdokument nicht mehr die Realität widerspiegelt und damit die Konsistenz der Architekturbeschreibung mit der Implementierung aber auch die Konsistenz unterschiedlicher Sichten auf die Architektur garantiert. Um diese Rückkopplung zu ermöglichen, beschränkt sich die Notation auf solche Aspekte, die später auch in Programmiersprachen, insbesondere in Eiffel [Mey92a, Mey88], implementiert werden können [Pes97]. Zur Semantikdarstellung auf hohem Abstraktionsniveau wird in BON Design-by-Contract (DBC) [Mey92b] unterstützt, welches zusätzlich den Vorteil von klaren Schnittstellenvereinbarungen mit sich bringt.

Im Folgenden wird eine kurze Einführung in die BON gegeben, die sich an der kurzen Einführung von Waldén [Wal98] orientiert. Detailliertere Informationen gibt

es im Buch zur BON [WN95]. Der nächste Abschnitt geht auf DBC ein, bevor die statischen Diagramme der BON vorgestellt werden und im Abschnitt 2.3.3 die dynamischen erläutert werden. Auf die BON-Methode wird hier nicht weiter eingegangen, da sie im Rahmen dieser Arbeit keine Rolle spielt.

2.3.1 Design-by-contract

Die Erkenntnis, dass Zusicherungen (engl. assertions), die während der Laufzeit eines Systems überprüft werden, eine vorteilhafte Softwaretechnik sind hat sich heute durchgesetzt [CR06]. Eiffel [Mey92a, Mey88] ist die erste objektorientierte Programmiersprache, die Zusicherungen eingeführt hat. Für die Nutzung von Zusicherungen in objektorientierten Programmen prägte Meyer den Ausdruck Design-by-contract (DBC) [Mey92b, Mey02, MJ97]. Es wird als permantenter, defensiver Mechanismus zur Fehlerfindung eingesetzt und wurde vor allem durch die Arbeiten von Floyd [Flo67], Hoare [Hoa69], und Dijkstra [Dij75] bekannt.

Insbesondere definierte Hoare das Beweisschema $\{P\}S\{Q\}$, wobei S eine Komposition von Anweisungen, P die Vorbedingung und Q die Nachbedingung von S ist. Wenn die Umgebung von S die Bedingung P im Zustand, der genau vor Ausführung von S eintritt gewährleistet, und S ausgeführt wird und terminiert, dann muss S die Bedingung Q im Zustand direkt nach Ausführung gewährleisten.

Zusicherungen in anderen Sprachen wie Alphard [WLS76], Euclid [PHL+77], und Anna [LVH85] beeinflussten DBC in Eiffel ebenso wie die Spezifikationssprache von Liskov und Guttag [LG86] und Z [Spi92]. In Eiffel selbst können Zusicherungen als Vor- und Nachbedingungen von Routinen und als Invarianten für Schleifen und Klassen formuliert werden. Die Invarianten zusammen mit den Vor- und Nachbedingungen werden als Verträge bezeichnet, die Rechte und Pflichten für zwei Vertragspartner festlegen. Im Fall eines Routinenaufrufs durch eine Klasse bedeutet dies, dass die aufrufende Klasse die Einhaltung der Vorbedingung garantieren muss woraufhin die aufgerufene Klasse bei Beendigung der Routine die Einhaltung der Nachbedingung garantiert. Also bindet die Vorbedingung die dienstnutzende Klasse und die Nachbedingung die dienstanbietende. Im Gegensatz zu Vor- und Nachbedingungen beschreiben Invarianten globale Eigenschaften von Klasseninstanzen, die bei Instanziierung sowie vor und nach jedem externen Routinenaufruf gelten müssen, nicht jedoch während des Aufrufs.

Diese Verträge werden dann auch bei Vererbungen eingehalten: die Invarianten aller Klassen von denen eine Klasse Eigenschaften erbt, gelten auch für diese selbst. Die Vorbedingungen überschriebener Routinen dürfen nur durch eine gleichstarke oder schwächere Vorbedingung und die Nachbedingungen durch eine gleiche oder stärkere ersetzt werden.

DBC kann in mehreren Phasen des Entwicklungsprozesses eingesetzt werden: zur Entwurfszeit dient es der Schnittstellenspezifikation über die reine Typisierung

hinaus und trägt damit unter anderem zur klaren Aufgabenverteilung bei. Diese hilft auch bei der Dokumentation und der Fehlersuche während des Testens. Zur Laufzeit können die Invarianten sowie Vor- und Nachbedingungen schließlich abgeschaltet werden oder mit Ausnahmebehandlungen zur Behandlung unerwarteter Programmzustände kombiniert werden. Neben Eiffel gibt es auch in weiteren Programmiersprachen Unterstützung für DBC, z. B. in den Sprachen der .Net-Umgebung [AS01] oder Java [KP98].

2.3.2 Statisches Modell

Statische BON-Modelle zeigen die Klassen des Softwaresystems, ihre Schnittstellen und Beziehungen zueinander sowie ihre Gruppierung zu Clustern. Dazu werden in frühen Entwurfsstadien erste statische Beziehungen in tabellarischer Form dargestellt. Diese wird im nächsten Abschnitt beschrieben, bevor die darauf folgenden Abschnitte auf die Elemente der grafischen Darstellung und ihre Kombination eingehen.

Tabellarische Darstellung

Inspiriert von Beck und Cunningham [BC89] definiert die BON drei Typen von Tabellen, um informell statische Informationen über ein System darzustellen: in einer SystemTabelle werden die obersten Cluster des Systems beschrieben, ClusterTabellen listen die beinhalteten Klassen und Cluster auf und KlassenTabellen beschreiben die Schnittstellen von einzelnen Klassen. Die Tabellen sind dabei nicht typisiert, da sie früh im Entwicklungsprozess eingesetzt werden und vor allem der Kommunikation mit Software-Laien dienen.

Tabelle 2.5 zeigt beispielhaft eine KlassenTabelle. In den oberen zwei Zeilen wird die Art der Tabelle sowie der Name der Klasse, eine Sequenznummer, eine Kurzbeschreibung und eine Reihe von frei wählbaren Schlüsselworten benannt. Darunter folgt die Klassenbeschreibung, die in drei Blöcke aufgeteilt ist. Anfragen sind Operationen, die Informationen zurückliefern, ohne den Systemzustand zu verändern, während Kommandos umgekehrt keine Informationen liefern dafür aber den Systemzustand ändern. Die saubere Trennung der Beiden verhindert Seiteneffekte und ermöglicht die Verwendung von DBC. Als Beschränkungen sind schließlich generelle Konsistenzbedingungen aufgeführt.

KlassenTabellen sollen das spätere Erstellen von typisierten Schnittstellenbeschreibungen vereinfachen, indem sie Informationen der Problemdomäne aufnehmen und diese übersichtlich darstellen. Die System- und ClusterTabellen stellen auf ähnliche Art und Weise Informationen ihrer jeweiligen Ebene zur Verfügung und werden hier aus Platzgründen nicht weiter beschrieben.

Klasse	AUFZUG	**Teil:** 1/1
Objekttyp Modelliert einen motorbetriebenen Aufzug		**Indizierung** Version: 1.1
Anfrage	aktuelles Stockwerk? weitere Nutzeranfragen? bewegt sich der Aufzug? sind die Türen geöffnet? geht es nach oben? geht es nach unten? ist der Aufzug im Leerlauf?	
Kommandos	öffne Türen! schließe Türen! bearbeite Nutzeranfrage	
Einschränkungen	Kann nicht gleichzeitig hoch und runter fahren. Falls gestoppt und keine weiteren Nutzeranfragen ist der Aufzug im Leerlauf. Bewegt sich nicht wenn die Türen geöffnet sind. Bewegt sich im Leerlauf nicht.	

Tabelle 2.5: Beispiel für eine informelle KlassenTabelle nach [Wal98].

Klassen

In statischen Diagrammen werden Klassen in komprimierter Form als Ellipsen dargestellt, die den Klassennamen und unter Umständen eine der in Abbildung 2.3 dargestellten Annotationen enthält. Neben dieser komprimierten Form können Klassen auch ausführlicher mit ihren weiter unten beschriebenen Verträgen und typisierten Schnittstellen beschrieben werden. Die in der komprimierten Form enthaltenen Annotationen können die folgenden Informationen enthalten: Wiederverwendete Klassen stammen aus älteren Systemen. Persistente Klassen sind solche, deren Instanzen extern gespeichert werden müssen. Verzögerte (engl. deferred) Klassen beinhalten zumindest einige Operationen, die von Nachfahren dieser Klasse implementiert werden, während effektive Klassen alle Operationen selbst implementieren. Klassen mit der Annotation Schnittstelle beinhalten externe Aufrufe zur Umgebung des Systems. Die Wurzelklasse ist diejenige, die beim Systemstart oder beim Start eines nebenläufigen Fadens instanziiert wird.

Cluster

Klassen können in der BON rekursiv in Cluster eingeteilt werden, die grafisch durch gestrichelte Rechtecke mit abgerundeten Ecken und dem Namen des Clusters als Beschriftung dargestellt werden. Cluster fassen verwandte Klassen und möglicherweise andere Cluster entsprechend einem gewählten Gesichtspunkt zusammen. Dessen Wahl hängt von den besonderen Merkmalen, die betont werden sollen ab und erlaubt

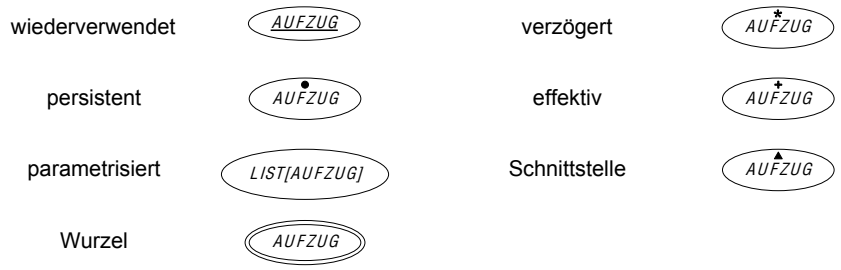

Abbildung 2.3: Komprimierte Darstellung von Klassen mit allen möglichen Annotationen nach [Wal98].

dadurch die Darstellung unterschiedlicher Architektursichten. Solche Sichten können z. B. die Verteilung auf Hardwareplattformen, unterschiedliche Abstraktionsniveaus oder funktionale Subsysteme darstellen.

Statische Beziehungen

In statischen BON-Diagrammen gibt es zwei Typen von Beziehungen: Vererbung und Dienstnutzung. Einfache Pfeile stellen Vererbung dar und Pfeile mit doppelter Linie die Beziehung vom Dienstnutzer zum Anbieter. Letztere wird weiter unterteilt in Assoziation und Aggregation, die jeweils mit Mutliplizität dargestellt werden können. Abbildung 2.4 zeigt Beispiele für alle diese statischen Beziehungen. Andere Beziehungen sind in der BON bewusst ausgelassen, da nur die von Implementierungssprachen unterstützten Beziehungen dargestellt werden sollen.

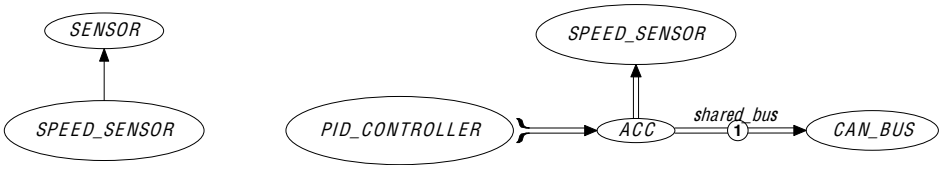

Abbildung 2.4: Vererbung und Dienstnutzung sind die in der BON darstellbaren statischen Beziehungen (nach [Wal98]). Im linken Teil ist die Vererbung dargestellt, rechts die verschiedenen Dienstnutzungsbeziehungen: Aggregation (PID_CONTROLLER ist Teil von ACC), Klient (ACC ruft SPEED_SENSOR auf) und geteilte Assoziation (ein geteilter CAN_BUS)

Verträge

Die Verträge des DBC [Mey92b, Mey02, MJ97] können in BON-Diagrammen ohne weitere Hilfsmittel dargestellt werden. Dazu können die Vor- und Nachbedingungen jeder einzelnen Klassenmethode und die Klasseninvariante beschrieben werden. Vor- und Nachbedingungen sind durch ein Frage- bzw. Ausrufezeichen markiert und werden nach der betreffenden Operation aufgeführt. Die Invariante wird in einem extra dafür vorgesehenen Abschnitt der ausführlichen Klassendarstellung beschrieben. Abbildung 6.2 zeigt ein Beispiel einer ausführlichen Klassendarstellung, in der neben einigen Vor- und Nachbedingungen sowie einer Invariante auch die unten beschriebenen typisierten Schnittstellen zu sehen sind.

Statische Strukturdiagramme

Im Gegensatz zur eher informellen Darstellung in den oben beschriebenen Tabellen, enthalten die grafischen Darstellungen der Klassen in der BON typisierte Schnittstellen. Mit diesen Schnittstellen und durch Beschreiben ihrer Verträge, statischen Beziehungen und Cluster können unterschiedlich detaillierte statische Strukturdiagramme erstellt werden. In Abbildung 6.2 auf Seite 89 ist ein Beispiel dargestellt, das alle diese Elemente beinhaltet. Je nach Notwendigkeit können Klassen ebenso wie Cluster in ihrer komprimierten Form dargestellt werden.

2.3.3 Dynamisches Modell

Mit den dynamischen BON-Modellen sollen die groben Möglichkeiten zur Erfüllung der Spezifikation mit Hilfe der Klassenfunktionen festgehalten werden. Dazu werden Funktionsaufrufe zwischen Klasseninstanzen oder Objekten bei typischen Anwendungsszenarien in dynamischen Diagrammen festgehalten. Diese bestehen aus einer Menge kommunizierender Objekte, die sich gegenseitig Nachrichten schicken und wieder nach Bedarf gruppiert oder komprimiert dargestellt werden können. Im Unterschied zu UML-Sequenzdiagrammen können die Objekte freier im Raum verteilt werden und ermöglichen nach Ansicht der Erfinder dadurch eine etwas übersichtlichere Darstellung auch komplexerer Szenarien.

Tabellarische Darstellung

Ähnlich wie beim statischen Modell gibt es auch im dynamischen drei Typen von Tabellen zum Erfassen dynamischer Beziehungen in frühen Entwurfsstadien. Ereignistabellen listen eingehende, externe und ausgehende, interne Ereignisse auf. Szenarien können informell in Tabellen festgehalten werden und Generierungstabellen beschreiben die Klassen, die für die Instanziierung der Problemdomäne des Systems zuständig sind. Tabelle 2.6 zeigt beispielhaft eine Szenarientabelle, die drei

interessante und mögliche Systemverwendungen auflistet. Jedes Szenario besteht aus einem Kurztitel zur Referenzierung und einer genaueren Beschreibung.

Szenarien	ACC_Control	**Teil:** 1/3
Kommentar Repräsentative Szenarien, die typisches Systemverhalten beschreiben		**Indizierung** Version: 1.3
Hindernis vor dem Fahrzeug: Vor dem Fahrzeug taucht ein langsamer fahrendes Hindernis auf und verschwindet nach einiger Zeit wieder.		
Stop & Go: Das Fahrzeug fährt auf ein Stauende auf und dieser löst sich nach einiger Zeit wieder auf.		
Neuer Abstandssensor: Der Sensor zum Erkennen von Distanzen wird durch einen mit anderer Technologie ersetzt.		

Tabelle 2.6: Beispiel für eine informelle Szenariotabelle.

Dynamische Diagramme

In Abbildung 6.3 auf Seite 90 ist ein Szenario exemplarisch in detaillierterer Form grafisch dargestellt. Zu sehen sind rechteckig dargestellte Objekte und Objektmengen, die sich gegenseitig Nachrichten schicken. Jedes Objekt ist mit dem typisierenden Klassennamen beschriftet. Ein gestrichelter Pfeil stellt eine Nachrichtenbeziehung dar und ist mit einer Sequenznummer beschriftet, die die Reihenfolge der Aufrufe bzw. Nachrichten bezeichnet. In der frei platzierbaren Szenariobox werden diese Nummern um erläuternden Text erweitert. Mit gepunkteten Boxen können mehrere Objekte zu Gruppen zusammengefasst werden, die die Nachrichten von bzw. an diese Objekte gemeinsam darstellen.

2.4 SCOOP

Simple Concurrent Object-Oriented Programming (SCOOP) [Mey97] ist ein Konzept zur Erweiterung der Programmiersprache Eiffel um ein einzelnes Schlüsselwort, das die Implementierung von verteilten und nebenläufigen Systemen ermöglicht. Es basiert auf Arbeiten von Meyer [Mey90, Mey93] und beinhaltet Beiträge von Caromel [Car93]. Daneben existiert auch eine Variante von Jalloul [Jal94, JP91]. Im Folgenden wird eine kurze Beschreibung gegeben, die sich an der Zusammenfassung

von Meyer orientiert [Mey97]. Eine ausführliche Herleitung und Diskussion des Konzeptes ist im selben Buch zu finden.

Zur Beschreibung nebenläufiger Systeme wird die fundamentale Grundlage der Objektorientierung, der Aufruf einer Routine eines anderen Objektes, wie folgt erweitert: Bei einem sequentiellen Ablauf bedeutet der Aufruf $x.f(a)$ innerhalb eines Objektes O_1, dass die Routine f eines Objektes O_2, dass an x gebunden ist, mit dem Argument a ausgeführt werden soll. Anstatt eines einzelnen Prozessors, der die Operationen auf allen Objekten ausführt, können bei nebenläufigen und verteilten Systemen zwei verschiedene Prozessoren für die Abarbeitung der Objekte O_1 und O_2 zuständig sein. Dann kann das Objekt O_1 nach dem Aufruf weiterarbeiten und muss nicht auf dessen Beendigung warten, da diese nun von einem anderen Prozessor erledigt wird.

Der Effekt eines Aufrufs hängt dann also davon ab, ob die beiden Objekte auf dem selben Prozessor oder unterschiedlichen ablaufen. Um dies im Quelltext sichtbar zu machen wird das Schlüsselwort **separate** eingeführt. Um auszudrücken, dass x im obigen Beispiel auf einem anderen Prozessor abläuft, wird anstelle von x: *SOME_ TYPE* jetzt x: **separate** *SOME_ TYPE* geschrieben, so dass alle Aufrufe an x parallel zum Rest der Berechnung ausgeführt werden. Durch diese Deklaration führt jede Instanziierung von x zu einem neuen Prozessor, der alle zukünftigen Aufrufe an x abarbeitet.

Dabei wird im Quelltext nicht angegeben welcher Prozessor benutzt werden soll. Durch die Deklaration wird lediglich ausgedrückt, dass die zwei Objekte von unterschiedlichen Prozessoren behandelt werden sollen, da dies die Semantik des Systems beeinflusst. Die tatsächliche Zuweisung eines Prozessors geschieht erst bei der Kompilierung oder zur Laufzeit des Systems. Auch die genaue Art des Prozessors wird im Quelltext nicht festgelegt.

In SCOOP können sowohl Hardware, wie CPU-Kerne oder Rechenknoten in einem Netzwerk, als auch Software, wie Prozesse eines nebenläufigen Betriebssystems, einen Prozessor darstellen. Aus Sicht des zu implementierenden Systems ist ein Prozessor ein abstraktes Konzept, dass die Ausführung einer nebenläufigen Anwendung erlaubt. Dadurch kann diese Anwendung auf vielen unterschiedlichen Architekturen, z. B. in Zeitscheiben auf einem Rechner oder in einem verteilten Netzwerk mehrerer Rechner, ausgeführt werden, ohne dass ihr Quelltext geändert werden muss. Die Verteilung der separaten Objekte auf ihre Prozessoren wird in einer eigenen Datei spezifiziert, die dann bei der Kompilierung oder bei der Ausführung des Systems für die Verteilung genutzt wird.

Zur Synchronisierung von Objekten, die auf unterschiedlichen Prozessoren laufen, gelten folgende Konventionen:

- Zur Resynchronisierung nach einem Aufruf $x.f(a)$ wartet das aufrufende Objekt nur wenn und falls es nötig ist, d. h. erst wenn es Informationen vom

dienstanbietenden Objekt mit *value:=x.some_ query* abfragt [Car93]. Dieser automatische Mechanismus wird auch *Warten nach Notwendigkeit (engl. wait by necessity)* genannt.

- Um exklusiven Zugriff auf ein separates Objekt zu erlangen, genügt es, eine mit ihm verknüpfte Entität als Argument in dem entsprechenden Aufruf zu verwenden.
- DBC wird zur Synchronisierung von Objekten genutzt: Enthält eine Vorbedingung ein separates Argument, dann wird die dazugehörige Routine erst dann ausgeführt, wenn die Vorbedingung erfüllt ist, sonst wird gewartet. Damit werden die Vorbedingungen anstatt als Korrektheitsbedingungen als Wartebedingungen genutzt.
- Um Klasseninvarianten zu erhalten, darf ein Prozessor, der für ein Objekt zuständig ist, zu jedem Zeitpunkt höchstens eine Routine ausführen.
- Wenn die Unterbrechung einer solchen Routine durch einen wichtigeren Klienten nötig ist, wird eine Ausnahme im entsprechenden Klienten ausgelöst, die dieser dann behandeln kann.

SCOOP ermöglicht auf die oben beschriebene Art und Weise die Implementierung von nebenläufigen und verteilten Systemen bei gleichzeitiger Nutzung aller objektorientierten Eigenschaften wie Mehrfachvererbung oder DBC. Im Rahmen dieser Arbeit wird das Konzept um Zeiteigenschaften erweitert und erstmals auf einem Echtzeitbetriebssystem implementiert.

3 Verhaltensmodellierung und -verifikation mit abstrakten Zustandsmaschinen

ASMs eignen sich, wie andere Modellierungssprachen auch, zur Verhaltensbeschreibung von Softwaresystemen. Sie sind eine effektive, formale Methode zur Spezifikation im industriellen Kontext [BPS00]. Ihre Abstraktionsmöglichkeiten bei gleichzeitiger, früher Ausführbarkeit durch Simulation und explizite Darstellung von Nichtdeterminismus weckten beim Autor das Interesse, sie zum Entwurf eingebetteter Software zu nutzen. Insbesondere ihre Eignung zur Modellierung von Hardware-Software-Systemen [Bör95, BG95, Gau95, BM97a, DH98] und die Nutzung des expliziten Nichtdeterminismus zur Darstellung von Reglern auf hohem Abstraktionsniveau lässt sie dazu geeignet erscheinen. Nicht zuletzt ermöglicht die formale Semantik der ASMs deren automatische Verifikation, was insbesondere für sicherheitskritische, eingebettete Systeme von Interesse ist.

Im Folgenden Abschnitt wird zunächst auf die Verhaltensmodellierung eingebetteter Systeme mit ASMs eingegangen, hierbei insbesondere auf Vorarbeiten und deren Erweiterung zur Darstellung von Reglern. Abschnitt 3.2 beschreibt anschließend einen neuen Ansatz zur Verifikation von ASMs durch Model-Checking, der die Ausführbarkeit von ASMs mit Simulatoren ausnutzt.

3.1 Modellierung eingebetteter Systeme mit ASMs

Da bereits mehrere Arbeiten zur Modellierung eingebetteter Systeme mit ASMs veröffentlicht sind, geht der folgende Abschnitt zunächst auf diese ein. Anschließend werden in Kapitel 3.1.2 mögliche Unzulänglichkeiten dieser Modellierung und mögliche Erweiterungen von ASMs beschrieben. Zunächst werden Arbeiten präsentiert, die wie in Kapitel 1 motiviert, die von Henzinger und Sifakis [HS06] geforderte Integration von Zeit- und Plattforminformationen ermöglichen. Anschließend wird ein theoretischer Ansatz diskutiert, der die Modellierung von hybriden Systemen, also solchen die sowohl diskrete als auch kontinuierliche Größen enthalten, als ASMs ermöglicht. Abschließend wird auf die Darstellung von Reglern mit Hilfe des **choose**-Konstrukts eingegangen. Letzteres ist keine Erweiterung der ASM-Sprache, stellt jedoch eine wichtige Anwendung dar, da Regelung eine typische Aufgabe von

eingebetteten Systemen ist. Abschließend geht Kapitel 3.2.1 auf die Simulation von ASMs ein, die zur Validierung beim ingenieurmäßigen Entwurf eine wichtige Rolle spielt, im Rahmen dieser Arbeit aber auch zur Verifikation eingesetzt wird.

3.1.1 Verwandte Arbeiten

Die hier vorgestellten, verwandten Arbeiten zur Modellierung eingebetteter Systeme mit ASMs sind allesamt öffentliche Fallstudien, die häufig zum Vergleich unterschiedlicher, formaler Methoden durchgeführt wurden. Hier wird aber das Hauptaugenmerk auf die Eigenschaften der modellierten Systeme und ihre Darstellung als ASMs gelegt, nicht auf den Vergleich zu anderen Methoden.

Die ASM-Lösung [BM97b, Mea97] zur Robotersteuerung in einer Produktionsanlage [Lin95] nutzt eine synchrone Mehragenten-ASM, um die von einer metallverarbeitenden Fabrik inspirierte Fallstudie zu bearbeiten. Die Anlage besteht aus zwei Transportbändern, einem Positionierungstisch, einem Roboter, einer Presse und einem Kran. Metallplatten, die über ein Transportband angeliefert werden müssen mit Hilfe des Tisches und des Roboters in die Presse gelegt und nach ihrer Bearbeitung über das andere Band und den Kran abtransportiert werden. In [BM97b] werden alle geforderten Korrektheits-, Sicherheits-, Leistungs- und Lebendigkeitsbedingungen nachgewiesen, indem sie auf dem abstrakten Modell unter gewissen Bedingungen bewiesen werden und gezeigt wird, dass diese Bedingungen für verfeinerte Modelle gelten. Nach mehreren Verfeinerungsschritten wird das implementierungsnächste ASM-Modell schließlich zu C++ Quelltext verfeinert [Mea97], der dann auf einem Simulator des Systems ausgeführt wird. Diese Fallstudie inspirierte mehrere Ansätze zur mechanischen Verifikation [Win97, GR00] und zur automatischen Generierung von C++ Quelltext aus einer ASM [Sch01b].

Eine ähnliche aber kleinere Fallstudie beschäftigt sich mit einem Dampfkessel [ABL96]. Das System und die dazugehörige ASM-Lösung [BBD+96] ähneln der Robotersteuerung sehr, weshalb hier nicht näher auf sie eingegangen wird.

Die Steuerung einer Schranke an einem Bahnübergang [HM96] wird auch von einem synchronen Mehragenten-ASM gesteuert [GH96], beinhaltet jedoch im Gegensatz zur Robotersteuerung zusätzliche Echtzeitbedingungen. Mehrere Schienen können von Zügen in beiden Richtungen befahren werden. Jedes Schienenpaar hat dabei vier Sensoren, die das Betreten oder Verlassen des Bahnübergangs registrieren. Diese Signale werden von der Steuerung ausgewertet, um die Entscheidung zu treffen, die Schranke zu öffnen bzw. zu schließen. Dabei muss die Schranke geschlossen sein, wenn sich ein Zug im Bahnübergang befindet und geöffnet sein, wenn dem nicht so ist. Beim Entwurf der Steuerung müssen die Zeiten zum Öffnen und Schließen der Schranke sowie die minimalen und maximalen Fahrtzeiten der Züge in den unterschiedlichen Streckenabschnitten beachtet werden. In der ASM-Lösungt [GH96] werden dazu Echtzeit-ASMs eingeführt, die zwischen diskreten, geordneten Zeitpunkten entweder

interne Zustandsübergänge oder externe ausführen.

Die Fallstudie zur Lichtsteuerung [BHPR99] in einem Universitätsgebäude wird in [BRS00] mit einer asynchronen Mehragenten-ASM gelöst. Neben dem Grundmodell zur formalen Modellierung der Anforderungen ist auch eine ausführbare Verfeinerung gegeben. Das verteilte System steuert die Beleuchtung in Räumen und Gängen und wird dabei von einzelnen Benutzern in den Räumen, Bewegungssensoren, der Helligkeit in der Umgebung und einer zentralen Handsteuerung beim Hausmeister beeinflusst. Je nach Informationslage müssen einzelne Lichter oder Lichtreihen ein- und ausgeschaltet oder gedimmt werden.

3.1.2 Erweiterungen

Alle oben genannten Fallstudien beschreiben die Modellierung eingebetteter Systeme und werden teilweise bis hin zu einem ausführbaren Modell oder Quelltext verfeinert. Bei allen Fallstudien werden Sicherheits- und Lebendigkeitseigenschaften formal nachgewiesen, was insbesondere bei den sicherheitskritschen Beispielen von Bedeutung ist. Weitere typische Eigenschaften eingebetteter Systeme treten in den Fallstudien jedoch nicht auf: Die Regelung von dynamischen Systemen ist eine der häufigsten Aufgaben eingebetteter Systeme. In den genannten Beispielen werden jedoch nur Steuerungen ohne Rückkopplung aus der Umgebung untersucht. Die untersuchten Systeme sind abgesehen vom Bahnübergang rein diskrete Systeme. Es kommen keine kontinuierlichen Variablen vor. Beim Bahnübergang werden Echtzeiteigenschaften untersucht, diese werden aber zuvor diskretisiert. Zeit- und Plattformeinschränkungen, die durch die Umgebung des Systems verursacht werden, werden nicht beachtet. Schließlich fällt auf, dass die ASMs bei ihrer Verwendung wenig Strukturmodellierung erlauben, insbesondere können Module und ihre Kommunikation nur mit Hilfe von globalen Variablen dargestellt werden. Nichtfunktionale Anforderungen und ihr Ausdruck in der Architektur des Systems werden in den Fallstudien nicht beachtet.

Dieser Abschnitt beschreibt einige Möglichkeiten zur Erweiterung von ASMs, um die genannten Anforderungen eingebetteter Systeme darzustellen. Auf die Strukturmodellierung wird in Kapitel 4 näher eingegangen.

Integration von Zeit- und Plattformeigenschaften

Ouimet und Lundqvist erweitern ASMs um Spracheigenschaften zur Integration von Plattform- und Zeiteigenschaften [OL07b]. Neben der funktionalen Spezifikation eines Systems mit ASMs können zeitliches Verhalten und Ressourcenverbrauch durch Annotation der ASM-Regeln beschrieben werden. Die Maschinen können automatisch auf Konsistenz und Vollständigkeit geprüft werden [OL07a]. Das zeitliche Verhalten kann durch eine Übersetzung der ASMs nach Uppaal [LPY97] verifiziert

werden [OL07c]. Anwendungsdomäne der Timed ASM genannten Sprache und des zugehörigen Werkzeugs sind reaktive Echtzeitsysteme. So wird besipielsweise ein Drosselklappenregler modelliert [OBL07]. Wie weiter oben diskutiert sind die Integration von Plattform- und Zeiteigenschaften entscheidende Beiträge, um eingebettete Systeme modellieren zu können.

Im Ansatz von Ouimet und Lundquvist werden der Ressourcenverbrauch und die Zeit einzelnen Regeln zugeschrieben und bei der Simulation und Verifikation des Systems als Werte für den ungünstigsten Fall angenommen und dann nach Ausführungsreihenfolge der Regeln addiert. Aussagen zur Ausführungszeit im schlechtesten Fall sind jedoch meist erst zu einem späten Zeitpunkt im Entwicklungsprozess, also bei verfeinerten ASMs einsetzbar. Zur abstrakten Darstellung in frühen Entwurfsstadien, insbesondere beim Grundmodell, eigenen sie sich nicht, da diese Informationen zu diesem Zeitpunkt meist noch nicht verfügbar sind. Des weiteren entspricht der ASM-Teil der Sprache nicht der allgemeinen Definition und eignet sich nicht zur Modellierung asynchroner Mehragentensysteme.

Hybride Systeme

Hybride Systeme beinhalten sowohl kontinuierliche als auch diskrete Werte und kommen bei eingebetteten Systemen aufgrund deren Interaktion mit der Umgebung häufig vor. Rust präsentiert einen Weg wie auch der kontinuierliche Anteil eines solchen Systems durch eine Erweiterung der diskreten ASM-Modellierung dargestellt werden kann [Rus00, Rus03]. Dazu werden hyperreelle Zahlen als Zeit- und Wertdomäne der kontinuierlich änderbaren Werte genutzt. Punktweise definierte hyperrelle Funktionen können in kontinuierliche, reelle Funktionen übersetzt werden und mit einer ASM mit geänderter Ausführungssemantik simuliert werden.

Dem theoretischen Ansatz mangelt es noch an Fallstudien und Werkzeugunterstützung. Auch gibt es noch einige ungelöste Probleme, wie beispielsweise für stückweise lineare hybride Systeme.

Abstrakte Modellierung von Regelalgorithmen

Zum Ausgleich von Störeinflüssen ist bei der Steuerung dynamischer Systeme häufig eine Rückkopplung nötig. Diese typische Aufgabe eingebetteter Systeme ist in den in Abschnitt 3.1.2 beschriebenen Fallstudien jedoch ausgeklammert, da dort nur Steuerungssysteme ohne Rückkopplung beschrieben werden. Hier wird deswegen eine Möglichkeit zur Spezifikation solcher Regler präsentiert, die ohne Erweiterung der ASM-Syntax auskommt und in dieser expliziten Form nach Wissen des Autors noch nicht veröffentlicht ist.

Wie in Tabelle 2.1 beschrieben, kann Nichtdeterminismus in einer ASM mit Hilfe des Ausdrucks **choose** x **with** φ **do** P modelliert werden. Die Regel P wird

dann mit einem beliebigen x ausgeführt, das φ erfüllt. Gibt es kein solches x, wird nichts gemacht. Dieses Regelelement der ASMs wird benutzt, um von Details oder verschiedenen Implementierungsmöglichkeiten bei der Auswahl eines Elements zu abstrahieren. Typische Beispiele sind die abstrahierte Darstellung der Auswahl eines Prozesses bei Scheduling-Algorithmen oder die Wahl zweier zu vertauschender Elemente einer Menge bei Sortieralgorithmen. Bei Verfeinerungsschritten, spätestens beim Übergang zur Implementierung, werden diese Konstrukte durch explizite, deterministische Anweisungen ersetzt. Sie werden also hauptsächlich in frühen, abstrakten Entwurfsphasen verwendet, um bei der weiteren Entwicklung Entscheidungsfreiheiten zu lassen.

Abstrakt kann ein Regler ähnlich zum **choose**-Konstrukt beschrieben werden. Er wählt einen Stellwert aus, um den geschlossenen Regelkreis bestehend aus Regler und Strecke in einer gewünschten Weise zu beeinflussen. Die Anforderungen an diesen Stellwert werden zum einen durch die physikalischen Grenzen dieses Wertes in der Stelleinrichtung vorgegeben. Diese lassen sich direkt als Bedingung φ in einer **choose**-Regel formulieren. Zum Anderen ergeben sich die Anforderungen an diesen Stellwert auch aus dem gewünschten Verhalten des geschlossenen Regelkreises. Typische Beispiele solcher Anforderungen sind Stabilität, schneller Ausgleich oder minimales Überschwingen [Lun05].

Diese Anforderungen beschreiben aber solche an die Kombination aus Regler und Strecke. Daher werden zur Simulation des Gesamtsystems also auch Informationen über den Zustand dieser Umwelt benötigt. Wenn sie als eigener ASM-Agent mitmodelliert wird, könnten einige der benötigten Informationen zwar verfügbar sein, im Allgemeinen sind sie es aber nicht. Oftmals wird die Umwelt im ASM-Modell gar nicht erfasst, obwohl das natürlich möglich wäre. Wird sie doch modelliert, dann als zustandsbasiertes System, das keine Informationen über das dynamisch-kontinuierliche Verhalten liefert. Diese sind jedoch im Allgemeinen nötig, um bei einer Simulation des Systems die Bedingungen an den zu wählenden Stellwert auszuwerten.

Ein Regler lässt sich also mit der **choose**-Regel in einer ASM erfassen. Dies ist als Anforderungsspezifikation zu verstehen, die zwar die üblichen Anforderungen formal erfassen kann, sich jedoch nicht ausführen lässt. Diese abstrakteste Form eines Reglers kann auch innerhalb des ASM-Rahmenwerks noch verfeinert werden. z. B. könnte in einem Verfeinerungsschritt entschieden werden, den Regler als PI-Regler zu implementieren. Dieser lässt sich dann mit zwei **choose**-Anweisungen modellieren, die jeweils für die Wahl des proportionalen bzw. integralen Verstärkungsfaktors zuständig sind. Zur endgültigen Verfeinerung in der Implementierung, die dann natürlich ohne Nichtdeterminismus auskommen muss, werden aber spezialisierte Werkzeuge aus der Regelungstechnik benötigt, in denen neben dem Regler auch die Strecke modelliert und simuliert werden kann.

3.2 Model-Checking von abstrakten Zustandsmaschinen

Neben der Modellierung und Simulation eigenen sich ASMs auch zur Verifikation eines Systems, da sie es formal darstellen. Mit der steigenden Komplexität eingebetteter Systeme steigt auch der Bedarf an formaler Verifikation insbesondere bei sicherheitskritischen Systemen wie beispielsweise dem Airbag. Wie bereits in der Einleitung erwähnt, wird diese auch in relevanten Sicherheitsnormen wie der IEC 61508 [IEC05] für Systeme gewisser Kritikalität vorgeschrieben. Sie hilft aber auch die Entstehung hoher Kosten wie beim Toyota Prius [Gar05] zu vermeiden.

Vollständiges Testen von Softwaresystemen ist wegen Zeit- und Ressourcenbeschränkungen häufig nicht möglich. Die Verifikation soll aus eben diesen Gründen meist automatisiert erfolgen. Daher beschreibt dieser Abschnitt eine Möglichkeit zum Model-Checking von ASMs. Die ASM wird dabei als formales Systemmodell gemäß Abbildung 2.2 benutzt und gegen eine formale Spezifikation, die als CTL-Formel gegeben ist, verifiziert.

Wegen der Verfeinerungsbeziehungen zwischen verschieden detaillierten ASM-Modellen kann diese automatische Verifikation wie in Abbildung 2.1 angedeutet nach jeder Entwurfsentscheidung stattfinden. So kann bereits die abstrakteste Darstellung, das Grundmodell, verifiziert werden, was eine frühe Fehleridentifizierung mit der dazugehörigen Kostenersparnis ermöglicht. Anschließend kann auch jede verfeinerte ASM auf dieselben und je nach Anforderungen auch neue Eigenschaften hin untersucht werden. Somit kann sichergestellt werden, dass einmal verifizierte Eigenschaften auch über Entwurfsentscheidungen hinweg gültig sind.

Hier wird ein neuartiger Ansatz zum Model-Checking von ASMs beschrieben, der ausnutzt, dass ASMs nicht nur logische Formeln sind sondern auch ausgeführt werden können [BS03b]. Dazu wird der ASM-Simulator CoreASM [Far07a] so angepasst, dass er bei mehreren Möglichkeiten alle möglichen Nachfolgezustände erstellt anstatt einen zufälligen zu wählen. Ein so aufgebauter Zustandsraum kann dann von einem Model-Checker durchsucht werden, wobei die Wahrheitswerte der atomaren Aussagen der zu verifizierenden Formel vom Simulator erfragt werden.

Frühere Arbeiten zum Model-Checking von ASMs nutzen deren formale Darstellung dazu aus, sie automatisch in eine weitere formale Darstellung zu übersetzen, die von existierenden Model-Checkern verstanden wird. Dies hat den Vorteil, dass die Eingabe für den Model-Checker automatisch aus Dokumenten erstellt werden kann, die im Rahmen eines Systementwurfs mit ASMs bereits vorhanden sind. Diese Eingabe muss also nicht von Hand erstellt werden, was Übersetzungsfehler vermeiden hilft. Der große Nachteil dieses Ansatzes ist jedoch der Verlust von Ausdrucksfähigkeit bei der Übersetzung. Wenn die ASMs in der Eingabesprache des Model-Checkers nicht vollständig darstellbar sind, können nicht alle ihre Eigenschaften verifiziert werden. Auch können Gegenbeispiele und Zeugen nicht im ASM-Zustandsraum sondern nur in dem des Model Checkers dargestellt werden.

Im hier beschriebenen Ansatz werden die ASMs dagegen erstmals direkt als Eingabesprache für einen Model-Checker verwendet. Dazu wird mit dem Simulator CoreASM der gesamte Zustandsraum aufgebaut und mit dem Model Chekcer [mc]square [SK06, Sch08] durchsucht. Durch diese direkte Unterstützung von ASMs wird der Audrucksverlust durch Übersetzung vermieden. Hauptvorteile dieses Vorgehens sind die Möglichkeit der Präsentation von Zeugen und Gegenbeispielen im ASM-Zustandsraum, die direkte Unterstützung aller Modellierungs- und Simulationsfähigkeiten von CoreASM und die einfache Erweiterbarkeit des Ansatzes um neue Funktionen von CoreASM. Nachteilig wirkt sich aus, dass die Datenstrukturen und Laufzeit des verwendeten Simulators nicht im Hinblick auf ihre Nutzung zum Model-Checking entwickelt wurden.

Im Folgenden wird kurz auf den zum Model-Checking verwendeten Simulator für ASMs eingegangen. Abschnitt 3.2.2 beschreibt den Model-Checker [mc]square mit einem Fokus auf die Eigenschaften, die seinen Einsatz zur Verifikation von ASMs ermöglichen. Die Interaktion zwischen [mc]square und CoreASM und die nötigen Änderungen an Letzterem werden in Kapitel 3.2.3 beschrieben, bevor abschließend in Abschnitt 3.2.4 noch einmal detailliert auf Unterschiede zu anderen Model-Checkern für ASMs eingegangen wird.

3.2.1 CoreASM

Neben ihrer formalen Definition ist es die Ausführbarkeit von ASMs, die das hier beschriebene Vorgehen zum Model-Checking erst ermöglicht. Mit den ersten Anwendungsversuchen in der Industrie entstanden auch die ersten Interpreter und Simulatoren für ASMs. Hier wird ein kurzer Überblick, über verfügbare Werkzeuge gegeben. Anschließend wird der im Rahmen dieser Arbeit verwendete Simulator CoreASM genauer beschrieben. Dazu wird zunächst auf die Eigenschaften eingegangen, die zu seiner Auswahl führten. Dann werden seine Architektur, der Aufbau eines Zustandes, der Ablauf der Simulation und der Zustandsübergang beschrieben.

Andere Simulatoren

Aus Platzgründen wird hier nur auf Werkzeuge zur Ausführung von ASMs eingegangen, die ASMs selbst interpretieren. Für eine genauere Beschreibung anderer Ansätze, wie die Kodierung in einer Programmiersprache [CP02], oder die Kompilierung in eine ausführbare Form [Sch01b] sei auf [BS03b] verwiesen.

Der erste veröffentlichte Interpreter für ASMs von Kappel [Kap93, Kap90] wurde im Rahmen der Standardisierung von Prolog entwickelt. Neben einem Interpreter für Basis-ASMs mit externen Funktionen liefert die ASM Workbench [Del99, DC01] auch weitere Grundfunktionalitäten wie einen Parser, abstrakte Syntaxbäume, Typüberprüfung und eine Übersetzung der ASMs in endliche Automaten, die mit dem

Model-Checker SMV [CES86] verifiziert werden können [DW00]. AsmGofer [Sch02] bietet erstmals eine grafische Schnittstelle zur Erstellung von Turbo-ASMs .

Neuere Ansätze zur Simulation von ASMs sind AsmL [GRS05], das auf der .NET-Plattform läuft und Xasm [Anl00], das mit Hilfe externer Funktionen eine komponentenbasierte Modularisierung von ASMs ermöglicht. Die in 3.1.2 bereits erwähnten Timed ASM [OL07b] können ASMs mit Zeit- und Plattformeigenschaften simulieren. Asmeta [SGG+05] formuliert ASMs als UML-Metamodell und kann diese dann auch ausführen.

Eigenschaften

CoreASM [FGG07] ist ein ASM-Werkzeug, das sich im Unterschied zu den oben genannten Arbeiten sehr nah an das mathematische Modell der ASMs hält und dessen Sprache [Far07a] auf Verständlichkeit ausgelegt ist. Dabei werden alle in 2.1 beschriebenen Arten von ASMs unterstützt.

Entscheidend für die Verwendung im Rahmen dieser Arbeit ist jedoch seine auf Erweiterbarkeit und Modifizierbarkeit ausgelegte Architektur. Durch die Verwendung eines Mikrokerns, in dem nur minimale Funktionalität implementiert ist und die konsequente Erweiterung dieses Kerns mit Plugins können die Änderungen des Simulators wie sie hier benötigt werden durch das Ersetzen von Plugins verwirklicht werden. So sind im Kern hauptsächlich der Zuweisungsoperator und Lokationen implementiert, selbst die Kontrollstruktur **if**...**then**...**else** ist in einem Plugin beschrieben. Wegen der großen Bedeutung der Architektur von CoreASM wird diese im Folgenden Abschnitt detailliert beschrieben.

Als grafische Oberfläche mit Syntaxhervorhebung und Projektverwaltung dient die Entwicklungsumgebung Eclipse. Dadurch wird ein strukturierter Entwicklungsprozess innerhalb einer einheitlichen Oberfläche, von der Spezifikation bis zur Implementierung unterstützt.

Architektur

Hier werden nur die für diese Arbeit wichtigen Eigenschaften der Architektur von CoreASM erläutert. Für eine ausführlichere Beschreibung sei auf die Arbeiten von Farahbod et al. [FGG05, FGGM06, Far07b, Far07c] verwiesen.

Wie in Abbildung3.1 in einem statischen BON-Diagramm dargestellt, besteht CoreASM aus den vier Komponenten Parser, Interpreter, Scheduler und abstrakter Speicher, durch deren Interaktion der Lauf einer ASM simuliert wird. Dazu liest der Parser eine ASM-Spezifikation ein und liefert einen beschrifteten abstrakten Syntaxbaum für jede Transitionsregel an den Interpreter. Dieser wertet die Regeln aus und generiert dabei eine Menge von Aktualisierungen. Diese werden auf einmal auf die Lokationen, die vom abstrakten Speicher verwaltet werden, angewandt.

Werden zum Auswerten der Regeln Werte bestimmter Lokationen benötigt, werden diese vom abstrakten Speicher an den Interpreter geliefert. Bei Mehragenten-ASMs wählt der Scheduler die Menge von Agenten, die im nächsten Schritt ausgeführt werden und koordiniert deren Auswertung.

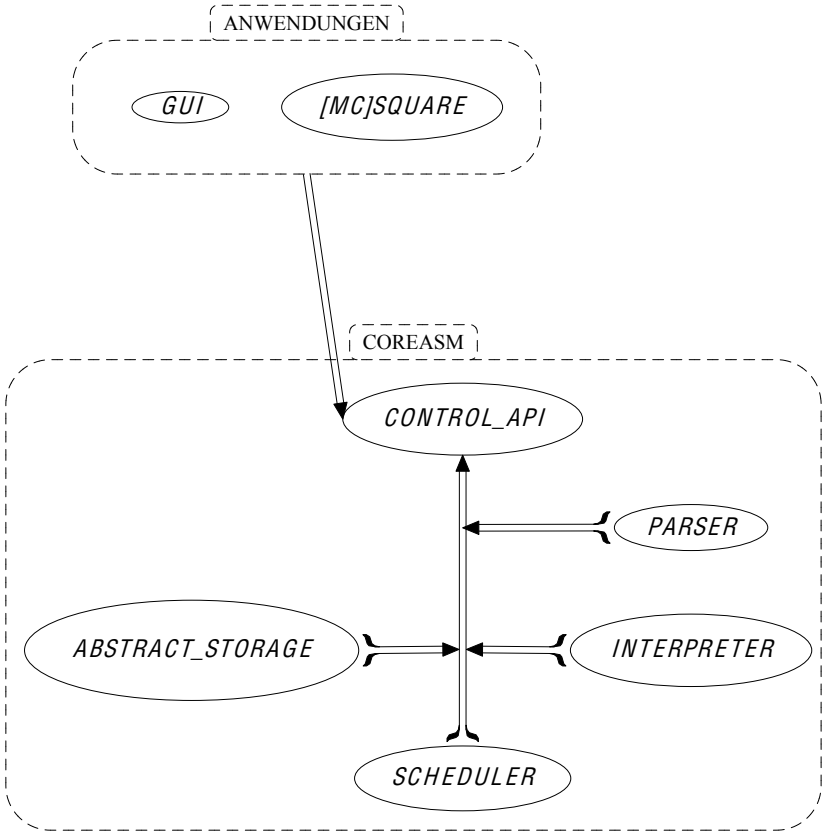

Abbildung 3.1: Statisches BON-Diagramm der Interaktion von CoreASM mit unterschiedlichen Anwendungen.

Die vier Komponenten können von außen über eine gemeinsame Schnittstelle, genannt *control API* angesprochen werden. Sie stellt Operationen, wie das Laden einer ASM-Spezifikation oder das Ausführen eines Zustandsübergangs zur Verfügung. Verschiedene externe Anwendungen können diese Schnittstelle nutzen, um während der Simulation mit CoreASM zu interagieren. Ein einfaches Beispiel für die Nutzung dieser Schnittstelle ist beispielsweise die Steuerung eines Simulationslaufes über die grafische Schnittstelle. Auch der Model-Checker nutzt diese Schnittstelle, um den Ablauf der Simulation zu steuern.

Die Grundfunktionen von Parser, Interpreter, Scheduler und abstraktem Speicher sind in einer Mikrokern genannten Minimalversion von CoreASM implementiert. Ein Rahmenwerk ermöglicht die Erweiterung und Veränderung dieser Funktionalität mit Hilfe von Plugins. Im Folgenden wird kurz auf die Inhalte von Kern, Rahmenwerk und Plugins eingegangen.

Kern Zur Speicherung des Zustands einer ASM enthält der Kern die Namen der Elemente, ihrer Universen und Funktionen mit dazugehörigem Wertebereich. Die Universen werden zusätzlich durch ihre charakteristischen Funktionen repräsentiert. Des weiteren enthält der Kern die vordefinierten Werte $true$, $false$, $undef$ und $self$. Das eigentliche Programm der ASM wird im Kern durch alle Regelnamen und ihre Wertebereiche repräsentiert. Dabei sind tatsächlich nur die Wertebereiche, nicht aber die entsprechenden Operationen enthalten. So sind wie oben angesprochen z. B. die booleschen Werte im Kern enthalten, nicht aber die booleschen Operationen, die durch ein Plugin zur Verfügung gestellt werden. Die einzigen Sprachelemente, die im Kern implementiert sind, sind der Zuweisungsoperator und eine Regel zum Importieren von Plugins.

Rahmenwerk Das Rahmenwerk für die Plugins besteht in CoreASM aus abstrakten Schnittstellen für die drei unten beschriebenen Arten von Plugins, welche zur Kommunikation mit ihnen verwendet wird. Durch die Implementierung einer dieser Schnittstellen kann die Funktionalität des Kerns verändert werden. Dies geschieht normalerweise als Erweiterung des Funktionsumfangs um Neues, kann aber auch in einer Änderung des alten Verhaltens bestehen. CoreASM bietet dabei in allen vier oben genannten Komponenten Ansatzpunkte für Plugins: die Grammatikregeln des Parsers, die semantischen Regeln des Interpreters, die Datentypen und -operationen des abstrakten Speichers sowie die Algorithmen des Schedulers können allesamt erweitert oder verändert werden.

Plugins Plugins werden in drei Kategorien eingeteilt:

1. Grundlagen-Plugins implementieren neue Datentypen. Dies betrifft die folgenden Komponenten des Programms:
   - Erweiterung des Parsers: Definition der Syntax von Funktionen, Operatoren, Literalen, um mit den Elementen des Datentyps zu arbeiten.
   - Erweiterung des Speichers: Kodierung und Dekodierung von Funktionen, um die Elemente des Datentyps im Speicher zu repräsentieren.
   - Erweiterung des Interpreters: Unterstützung der Semantik aller Operatoren, die im Datentyp definiert sind.

2. Regel-Plugins implementieren spezielle Regeln. Durch Ausführung der Regeln entsteht eine (möglicherweise leere) Menge von Aktualisierungen. Regel-Plugins erweitern die folgenden Komponenten des Programms:
   - Erweiterung des Parsers: Definition der Syntax der einzelnen Regeln.
   - Erweiterung des Interpreters: Festlegung der Semantik der verschiedenen Regeln.
3. Scheduling-Plugins werden verwendet um neue Scheduling-Algorithmen für verteilte ASMs zu implementieren. Diese ersetzen dann den aktuellen Scheduler des Programms:

CoreASM selbst liefert eine große Menge an Plugins mit, die die Ausführung aller in 2.1 beschriebenen Klassen von ASMs ermöglichen. Zusätzlich gibt es eine ganze Reihe von Plugins, die im Rahmen spezieller Projekte entstanden sind und jetzt zur Verfügung stehen, wie z. B. zur wie Einbindung von Java-Quelltext oder zur Ausgabe von Datenpunkten in einem Koordinatensystem.

ASM-Zustände in CoreASM

CoreASM soll möglichst nah am mathematischen Modell der ASMs bleiben. Dies spiegelt sich auch im Aufbau eines Zustands, der vom abstrakten Speicher verwaltet und aktualisiert wird, wieder. Er besteht aus Universen, Funktionen und Regeln.

Ein Universum besteht aus einer Menge von Abbildungen zwischen dem Namen des jeweiligen Universums und den enthaltenen Elementen, genauer deren Speicherplätzen. Das Universum *agents*, das die einzelnen virtuellen Maschinen von Mehragenten-ASMs enthält, ist beispielsweise bereits vordefiniert. Funktionen sind Abbildungen zwischen Mengen von Elementen der Universen. Die vordefinierte Funktion *program* verknüpft z. B. die Elemente des Universums *agents* jeweils mit ihren auszuführenden Regeln. Regeln bestehen schließlich aus ihrem Namen und dem zugehörigen abstrakten Syntaxbaum. Sie bilden ein eigenes Universum, das ebenfalls im Zustand gespeichert wird. Alle Elemente des Zustands gehören dabei einer gemeinsamen Basisklasse an. Zu dieser gehören auch die Datenelemente, die die Werte an bestimmten Funktionslokationen repräsentieren.

Simulationsablauf

Die Simulation wird beispielsweise von der grafischen Benutzeroberfläche aus über die Schnittstelle von CoreASM von Außen angestoßen. Dabei werden die folgenden Schritte durchlaufen, wobei nach jedem einzelnen Schritt über die Schnittstelle Informationen über den Zustand der Simulation abgefragt werden können:

1. Initialisierung der Engine

   a) Initialisieren des Kerns
   b) Laden des Plugin-Bibliothek Katalogs
   c) Laden und aktivieren der Plugins der Standard-Bibliothek

2. Laden einer CoreASM-Spezifikation
   a) Parsen des Spezifikationskopfs
   b) Laden weiterer Plugins, die im Spezifikationskopf deklariert sind
   c) Parsen des Spezifikationsrumpfs
   d) Initialisieren des Speichers
   e) Aufsetzen des initialen Zustands

3. Ausführung der Spezifikation
   a) Ausführen eines Zustandsübergangs, wie unten beschrieben
   b) Wiederholen des vorigen Schritts, bis eine Abbruchbedingung für die Simulation eintritt

Insbesondere kann nach jedem Zustandsübergang der abstrakte Speicher ausgelesen werden. Dadurch ist der aktuelle Zustand der ASM während der Simulation für externe Anwendungen zur Weiterverarbeitung verfügbar. Die Abbruchbedingung für die Simulation kann über die Benutzerschnittstelle gesetzt werden, wobei verschiedene Bedingungen zur Verfügung stehen. Die Simulation kann nach einer festen Anzahl von Zustandsübergängen gestoppt werden, wenn bei einem Zustandsübergang keine Aktualisierungen erzeugt wurden oder wenn sich der Zustand beim Übergang nicht geändert hat. Je nach Einstellung der Abbruchbedingung und der simulierten ASM sind dann natürlich unendliche Simulationsläufe möglich.

Zustandsübergang

Bei einem Zustandsübergang werden in CoreASM die folgenden Schritte ausgeführt:

1. Von außen wird über die Schnittstelle der Befehl zum Zustandsübergang an den Scheduler gesendet.

2. Der Scheduler erfragt vom Speicher die gesamte Menge von aktiven Agenten (aus der speziell ausgezeichneten Menge *agents*).

3. Der Scheduler wählt je nach implementiertem Algorithmus eine Teilmenge der Agenten, deren Regeln im nächsten Schritt ausgeführt werden sollen.

4. Der Scheduler wählt aus der zuvor gewählten Menge einen Agenten aus und weist ihm die spezielle Variable $self$ im Speicher zu.

5. Der Scheduler ruft den Interpreter auf, um das Programm des aktuellen Agenten auszuführen.

6. Der Interpreter wertet das Programm des aktuellen Agenten aus, indem er auf $program(self)$ im aktuellen Zustand zugreift und die entsprechende Transitionsregeln ausführt.

7. Nach der Auswertung benachrichtigt der Interpreter den Scheduler, dass die Interpretation beendet ist und liefert diesem die durch die Ausführung der Regeln entstandene Menge an Aktualisierungen.

8. Der Scheduler wählt solange einen weiteren Agenten aus der selektierten Teilmenge der Agenten aus und wiederholt den Aufruf des Interpreters, bis es keine weiteren Agenten mehr in der Menge gibt. Dann ruft der Scheduler den Speicher auf, um alle gesammelten Aktualisierungen gleichzeitig auf den aktuellen Zustand anzuwenden.

9. Der Speicher benachrichtigt den Scheduler, ob die Menge von Aktualisierungen Konflikte enthält, oder ob der Zustandsübergang erfolgreich war. Liegen Konflikte in der Menge vor, wählt der Scheduler eine andere Teilmenge von aktiven Agenten aus und startet mit dieser Menge wieder bei Schritt 4. Gibt es keine mögliche weitere Teilmenge mehr, scheitert der Zustandsübergang.

3.2.2 [mc]square

[mc]square ist ein Akronym für Model-Checking Mikrocontroller und ist ein diskreter CTL Model-Checker, der Assemblerprogramme von eingebetteten Systemen verifizieren kann [SSK07, SK06, Sch08]. Er unterstützt dabei verschiedene Mikrocontroller und nutzt zum Aufbau und zur Durchsuchung des Zustandsraums einen Algorithmus von Vergauwen und Lewi [VL93], der später von Heljanko [Hel97] angepasst wurde. Als formale Systemmodelle werden Assemblerprogramme im Binärformat „Executable and Linking Format“ (ELF) akzeptiert und gegen eine formale Spezifikation, die Aussagen über Register, Variablen usw. enthalten kann, verifiziert. Zusätzlich überprüft [mc]square die Programme auf Kollisionen und Überläufe im Kellerspeicher sowie unerwünschte Nutzung wie z. B. Schreibzugriff auf reservierte Register.

Dabei nutzt [mc]square Simulatoren der unterstützten Mikorcontroller, um Zustände zu erzeugen und Informationen über atomare Aussagen zu erhalten. Dazu liefert ein Simulator eine serialisierte Repräsentation aller möglichen Nachfolger eines gegebenen Zustands und die Wahrheitswerte der atomaren Aussagen der zu verifizierenden Formel in dem gegebenen Zustand. Dem bekannten Problem der Zustandsexplosion wird auf zwei Arten entgegengewirkt: Um die Größe des Zustandsraums im Speicher zu verkleinern werden die serialiserten Zustände komprimiert

und um die Anzahl der zu speichernden Zustände zu verringern, werden mehrere Abstraktionstechniken verwendet.

Zusätzlich kann der Nutzer entscheiden, ob zur Speicherung des Zustandsraums der Hauptspeicher oder die Festplatte verwendet werden soll. Dadurch können erheblich größere Zustandsräume bearbeitet werden, denn anstatt den Zustand komplett im Arbeitsspeicher zu halten, wird nur ein Verweis auf den Speicherort auf der Festplatte im Arbeitsspeicher gehalten. Die Anzahl der speicherbaren Zustände wird bei dieser Technik jedoch weiterhin von der Größe des Arbeitsspeichers beschränkt. Der Vorteil dieser Methode liegt in der Möglichkeit, sehr große Zustandsräume zu erzeugen, sie ist jedoch langsamer.

All diese Ansätze werden, abgesehen von den Abstraktionstechniken, hier auch für die Verifikation von ASMs verwendet. Letztere werden im Rahmen dieser Arbeit nicht angewendet, da die implementierten Techniken nicht direkt auf ASMs übertragbar sind. Dennoch liegt hier ein mögliches Gebiet für zukünftige Verbesserungen.

Model-Checking von Assembleranweisungen ist zwangsläufig hardwareabhängig. Daher ist die einfache Erweiterbarkeit um neue Mikrocontrollersimulatoren mit den von ihnen unterstützten, speziellen Assemblerbefehlen eines der wichtigsten Entwurfsprinzipien von [mc]square. Da CoreASM die Simulation von ASMs unterstützt wird es hier als quasi virtueller Mikrocontroller in [mc]square integriert. Die nächsten Abschnitte geben eine kurze Zusammenfassung von [SK07] und beschreiben den Ablauf des Model-Checking in [mc]square sowie seine Architektur, insbesondere deren Erweiterbarkeit, die den Einbau von CoreASM erst ermöglicht.

Model-Checking in [ms]square

In Abbildung 3.2 ist der Ablauf in [mc]square dargestellt: Als Eingaben werden eine CTL-Formel und ein Assemblerprogramm (ELF), das zusätzlich vom dazugehörigen C-Programm begleitet werden kann, verwendet. Zunächst wird die Formel von einem Parser eingelesen, auf syntaktische Korrektheit überprüft, analysiert und zur weiteren Verwendung durch den Model-Checker vorverarbeitet. Dabei wird die Formel so umgeformt, dass sie nur noch CTL-Operatoren der Form AX φ, A(φ U ψ) und E(φ U ψ) sowie die booleschen Operatoren and or und not enthält. Zudem wird sie in ihre atomaren Aussagen zerlegt. Ein Disassembler übersetzt die binär kodierte ELF-Datei in ein menschenlesbares Assemblerprogramm. Anschließend wird der Model-Checker gestartet. Dieser fragt im Zustandsraum nach dem Initialzustand bzw. Nachfolgern eines bestimmten Zustands wenn diese zum Überprüfen der Formel benötigt werden. Wenn ein benötigter Zustand noch nicht im Zustandsraum enthalten ist, wird der Simulator genutzt, diesen zu erzeugen. Dazu werden die Effekte des Ausführens der aktuellen Anweisung des Assemblerprogramms im aktuellen Zustand simuliert. Zusätzlich liefert der Simulator die Wahrheitswerte der atomaren Aussagen der Formel in einem Zustand. Wird ein Zustand gefunden, in dem die Formel nicht

erfüllt ist, wird der Algorithmus abgebrochen.

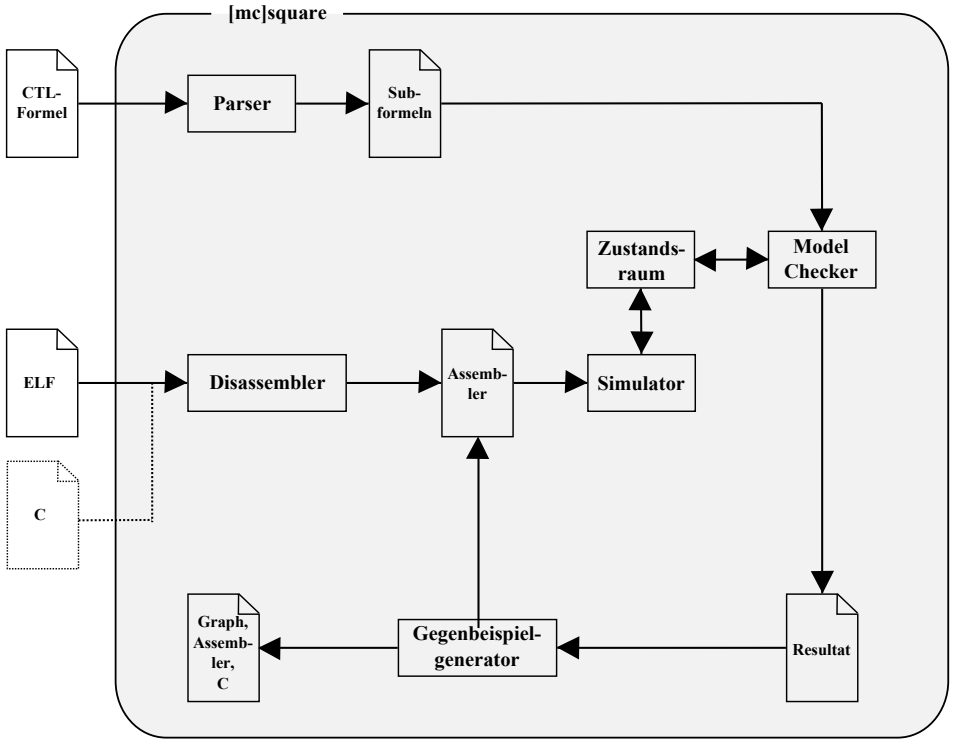

Abbildung 3.2: Ablauf des Model-Checking in [mc]square zur Verifikation von Assembelerprogrammen (nach [Sch08]).

Anschließend kann ein Gegenbeispiel oder Zeuge der geprüften Formel erzeugt werden. Dazu wird ein vom Model-Checker gelieferter Pfad durch den Zustandsraum als Graph, im Assemblerprogramm und falls gegeben im C-Programm dargestellt. Die Zustände des Pfades können einzeln durchgegangen werden, wobei in jedem Zustand Informationen über den Mikrocontroller wie z. B. Register- und Speicherinhalte oder nächste auszuführende Befehle angezeigt werden.

Architektur

Abbildung 3.3 zeigt ein statisches BON-Diagramm der für diese Arbeit wichtigen Klassen von [mc]square. Die Klassen ModelChecker, StateSpace und Simulator entsprechen den jeweiligen Komponenten aus Abbildung 3.2. Eine der wichtigsten, qualitativen Anforderungen, die diese Architektur beeinflussen, ist die Leichtigkeit, mit der das Werkzeug um die Unterstützung neuer Mikrocontroller erweitert werden kann. Grundlegende Strategie zum Erreichen dieses Ziels ist die Kapselung aller

hardwareabhängigen Informationen im Simulator.

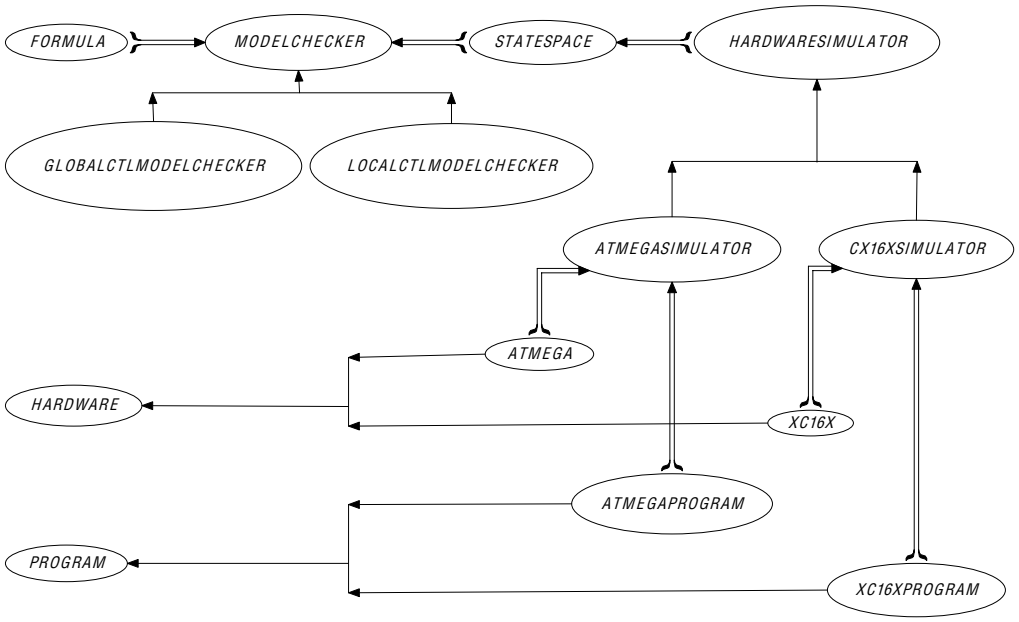

Abbildung 3.3: Architektur von [mc]square als statisches BON-Diagramm. Hier sind nur Klassen abgebildet, die wichtig für das Verständnis der Erweiterbarkeit um neue Simulatoren sind. Einige Unterklassen sowie ganze Pakete wie z. B. die grafische Oberfläche sind ausgelassen (nach [Sch08]).

Eine Instanz des Model-Checkers nutzt die vom Benutzer eingegebene Formel sowie den Zustandsraum, der zu Beginn lediglich den vom Hardwaresimulator gelieferten Initialzustand enthält. Um den Zustandsraum hardwareunabhängig zu halten, werden die gesamten hardwareabhängigen Daten des Zustands in einem Byte-Feld gespeichert. Dieses kann nur vom Simulator interpretiert werden und wird von ihm genutzt um den tatsächlichen Zustand des Mikrocontrollers, wie Prozessorzustand, Speicherinhalt usw., herzustellen. Zusätzliche hardwareunabhängige Informationen eines Zustands wie sein Name, Verweise auf seine Nachfolger und Wahrheitswerte der Formel werden im Zustand festgehalten.

Im Hardwaresimulator werden ein Assemblerprogramm und ein Mikrocontroller kombiniert. Sie repräsentieren die Assembleranweisungen und die Eigenschaften der Hardware, wie z. B. deren Register. Sollen die Nachfolger eines Zustands erzeugt werden, wird die einzelne, im Befehlszähler benannte Anweisung, auf dem Mikrocontroller simuliert und als Feld von Byte-Feldern, von denen jedes einen möglichen Nachfolger repräsentiert, an den Zustandsraum zurückgegeben. Im Simulator wird

auch Nichtdeterminismus, der z. B. entsteht wenn Eingaben aus der Umgebung des Mikrocontrollers eingelesen werden, behandelt indem eine Überapproximation des realen Verhaltens erstellt wird.

Um neue Mikrocontroller zu unterstützen, muss ein Simulator mit den oben beschriebenen Eigenschaften implementiert werden. Der Aufwand dieser Implementierung hängt stark von der Mikrocontrollerfamilie und deren Funktionsumfang ab. Zur Zeit werden mehrere Mikrocontroller der Atmel ATmega-Familie sowie der Infineon XC167 unterstützt.

3.2.3 Verifikation von CoreASM-Modellen mit [mc]square

Im Rahmen dieser Arbeit wird CoreASM als neuer Hardwaresimulator (vgl. Abbildung 3.3) in [mc]square eingebaut und repräsentiert dann einen ASM-Mikrocontroller. Die in einem Simulator gekapselten Hardware und Programm werden dann durch die abstrakte Maschine und die darauf laufende ASM-Spezifikation repräsentiert. Dieser Ansatz bietet den Vorteil, die gesamte Funktionalität von CoreASM zu unterstützen und direkt auf zukünftige Erweiterungen von CoreASM anwendbar zu sein. So werden beispielsweise alle in 2.1 beschriebenen ASM-Klassen unterstützt.

ASMs, die auf mindestens einem Pfad ihres Zustandsraums einen unendlichen Lauf beinhalten, können theoretisch mit diesem Ansatz für valide Formeln nicht verifiziert werden. Existiert ein Gegenbeispiel auf einem Pfad, der beim Durchsuchen des Zustandsraums nicht hinter dem unendlichen liegt, würde dieses natürlich dennoch gefunden. ASMs mit unendlichen Läufen existieren aber nicht im Simulator, da hier alle Datentypen eine endlich große Wertemenge haben. Auch ist zu beachten, dass die Dauer eines Laufs stark von dem in CoreASM eingestellten Abbruchkriterium für die Simulation abhängt (vgl. Kapitel 3.2.1). Steht genügend Arbeitsspeicher zur Verfügung, können alle in CoreASM simulierbaren ASMs mit dem hier entwickelten Ansatz verifiziert werden.

Die folgenden Abschnitte beschreiben die nötigen Änderungen an CoreASM und die Erweiterung von [mc]square um den neuen Simulator.

Änderungen an CoreASM

Um CoreASM als Simulator in [mc]square zu verwenden, müssen die folgenden Änderungen an Ersterem vorgenommen werden: Zunächst müssen die ASM-Zustände in ein Byte-Feld umgewandelt und normiert werden. Da CoreASM als Simulator nur einen möglichen und nicht alle Nachfolger eines ASM-Zustands erzeugt, sind Änderungen und Erweiterung daran nötig. Namentlich sind dies die Erweiterung des Schedulers, der nichtdeterministischen Auswahl und des Interpreters. Diese wurden durch die wohl überlegte Architektur von CoreASM vereinfacht, insbesondere durch die Möglichkeit, das Verhalten durch den Austausch von Plugins zu verändern. Im

Folgenden werden alle Änderungen und Erweiterungen genauer beschrieben.

Serialisierung und Deserialisierung Wie oben erwähnt, müssen alle hardwareabhängigen Informationen eines Zustands vom Simulator als Byte-Feld repräsentiert werden. Im Fall von CoreASM bedeutet dies, dass alle Universen, Funktionen und Regeln, die im abstrakten Speicher liegen, in ein solches Feld umgewandelt und auch wieder zurücktransformiert werden müssen. Eine wichtige Anforderung an die Serialisierung ist, dass sie auch mit wenig Aufwand an Erweiterungen von CoreASM anpassbar ist.

Der abstrakte Speicher von CoreASM wurde mit dem Ziel entworfen, durch Nähe zur mathematischen ASM-Definition möglichst verständlich und mit Hilfe von Plugins einfach erweiterbar zu sein. Die Elemente des abstrakten Speichers sind stark objektorientiert strukturiert, letztlich sind sie jedoch alle als Zeichenketten abgespeichert.

Zur Transformation dieser Daten in ein Byte-Feld und zur Rücktransformation wird hier die native Unterstützung der Programmiersprache Java zur Serialisierung und Deserialisierung von Objekten verwendet. Diese Technik wird oft benutzt, um Objekte in einen persistenten Speicher abzulegen und bietet den Vorteil, nur geringe Änderungen am Quelltext der Klassen des abstrakten Speichers zu erfordern. Dazu müssen lediglich alle Klassen, die in das Byte-Feld aufgenommen werden sollen, als serialisierbar gekennzeichnet werden. Aufgrund der stark verzweigten Struktur des abstrakten Speichers von CoreASM ist dieser Schritt trotz des geringen Umfangs der Änderungen zeitaufwändig.

Der geringe Umfang der Änderungen am Quelltext bietet den entscheidenden Vorteil, auch auf zukünftige Erweiterungen des abstrakten Speichers zuzutreffen, die ebenfalls in das Byte-Feld aufgenommen werden sollen. Somit ist der Ansatz auch für zukünftige Versionen oder ungewöhnliche Erweiterungen einfach einsetzbar: Die Plugins, die den abstrakten Speicher erweitern, müssen lediglich die zum Zustand gehörigen Klassen als serialisierbar kennzeichnen. Der Speicherbedarf eines Zustands ist ein erwarteter Nachteil, hat sich jedoch in den in Kapitel 6 vorgestellten Vergleichen als überraschend klein herausgestellt. Die in CoreASM für alle Elemente des abstrakten Zustands verwendeten Zeichenketten sind weder besonders speicherschonend noch ist es die native Serialiserung der Objekte. Die Kompression des erzeugten Byte-Feldes nutzt die große Informationsredundanz innerhalb eines Zustandes jedoch sehr effizient zur Verkleinerung aus.

In Zukunft könnten in allen zu serialisierenden Klassen eigene Serialisierungsmethoden implementiert werden, die nur die zur Speicherung eines Zustands notwendigen Informationen in das Byte-Feld einfügen. Mit Hilfe einer Legende, die die sich häufig wiederholenden Zeichenketten auf Datentypen mit geringerem Speicherverbrauch abbildet, könnte der Speicherbedarf signifikant gesenkt werden. Dieser Ansatz würde

dennoch allen zukünftigen Erweiterungen des Speichermodells die Freiheit lassen, entweder die native, einfache Serialisierung oder eine effizientere aber aufwendiger zu implementierende, eigene zu verwenden.

Gleichheit von Zuständen Ein vor der Imlementierung des hier beschriebenen Ansatzes unerwartetes Problem betrifft die Gleichheit zweier Zustände. Jeder neu erzeugte Zustand wird von [mc]square zunächst auf Existenz im Zustandsraum überprüft. Dazu ist ein Vergleich von Zuständen nötig, der jedoch nur auf der Byte-Repräsentation eines Zustands stattfinden kann, da er im hardwareunabhängigen Teil stattfindet. Daher ist die Reihenfolge, in der einzelne Elemente des Zustands in diesem enthalten sind, von Bedeutung.

Intern besteht ein Zustand in CoreASM aus mehreren ungeordneten Abbildungen. In dieser Datenstruktur der Programmiersprache Java werden Daten mit einem zugehörigen Schlüssel abgespeichert. In einer solchen Abbildung werden beispielsweise den Lokationsnamen der ASM als Schlüssel ihre Werte zugeordnet. Zur Verwendung innerhalb von CoreASM reicht die Möglichkeit, mit dem Schlüssel Werte aus den Abbildungen abzufragen bzw. neue Werte einzufügen.

Da [mc]square aber die Bedeutung der einzelnen Bytes in dem Byte-Feld nicht kennt und somit nicht auf die einzelnen Elemente der Abbildungen zugreifen kann, ist hier die Reihenfolge der Schlüssel-Wert-Paare in der Abbildung wichtig. Dies führt dazu, dass ein- und derselbe Zustand von [mc]square als scheinbar neuer im Zustandsraum abgelegt wird, wenn z. B. zwei Lokationen in der Abbildung ihren Platz getauscht haben, was bei der Simulation auftreten kann. Um dies zu verhindern wird in CoreASM eine Ordnungsrelation auf den Zuständen eingeführt, die vor jeder Serialisierung durchgeführt wird. Jetzt werden genau die Zustände von [mc]square als identisch erkannt, die es auch tatsächlich sind.

Aussagen über Elemente des Zustands Die durch die CTL-Formeln gegebenen Spezifikationen enthalten Aussagen über einzelne Elemente des Zustands, wie z. B. den Wert einer Lokation. Diese atomaren Aussagen können nur in CoreASM überprüft werden, da sie den hardwareabhängigen Teil eines Zustands betreffen. Dazu wird ein Zustand als Byte-Feld übergeben und wie oben beschrieben deserialisiert. Anschließend können alle Elemente des Zustands, also seine Universen, Regeln und Funktionen mit ihren aktuellen Elementen abgefragt werden. Dies Abfrage kann direkt über die in Abbildung 3.1 gezeigte Schnittstelle erfolgen und bedarf keiner Änderungen an CoreASM. Die verifizierbaren Formeln können also Aussagen über alle Elemente des Zustands machen, wie z. B. die Aktivität eines bestimmten Agenten, die Werte bestimmter Lokationen oder die Existenz bestimmter Elemente in einem Universum.

Zustandsübergang Wie in Abschnitt 3.2.1 beschrieben, wählt der Scheduler für Mehragenten-ASMs für jeden Zustandsübergang zufällig eine Teilmenge von aktiven Agenten aus. Diese Agenten führen dann ihre jeweiligen Regeln aus und generieren eine Menge von Aktualisierungen, die bei Konsistenz im Zustandsübergang ausgeführt werden. Bei Inkonsistenz wird eine andere Teilmenge von aktiven Agenten ausgewählt. Für das Model-Checking muss diese zufällige Wahl durch die Iteration über alle möglichen Teilmengen ersetzt werden.

Bei der Implementierung wurde die eigentliche Iteration jedoch in den Simulator in [mc]square ausgelagert, um die Änderungen an CoreASM klein zu halten. Insbesondere kann so der Programmablauf in CoreASM bestehen bleiben, es muss lediglich das Plugin für das Scheduling ausgetauscht werden. Dies soll die Kompatibilität zu zukünftigen Ausgaben von CoreASM sicherstellen. Daher wird bei jedem Zustandsübergang weiterhin nur ein Nachfolgezustand erzeugt, die Menge der aktiven Agenten wird jedoch von [mc]square aus vorgegeben.

Der neue Scheduler liefert zu Beginn eines jeden Schritts die Menge aller n Agenten an [mc]square. Dieses erstellt dann die 2^n möglichen Teilmengen aktiver Agenten und liefert eine davon an den Scheduler zurück. Dieser sorgt für die Ausführung all dieser Agenten, die Aggregation ihrer Aktualisierungen und bei Konsistenz die Zuweisung aller Aktualisierungen. Bei Inkonsistenz schlägt der Zustandsübergang fehl. Der möglicherweise leere, neue Zustand wird anschließend an den Simulator in [mc]square zurückgegeben. Dieser sorgt für das Zurücksetzen auf den alten Ausgangszustand und lässt den Scheduler mit der nächsten Teilmenge aktiver Agenten einen weiteren Nachfolgezustand erzeugen. Dies geschieht so lange, bis alle möglichen Teilmengen von Agenten abgearbeitet sind. Inkonsistente Aktualisierungsmengen führen zu einem Fehlschlag beim Zustandsübergang und somit nicht zu einem neuen Zustand.

Durch dieses Vorgehen müssen die Funktionen des alten Schedulers nur an wenigen Stellen geändert werden: Die Auswahl der Agenten wird ausgelagert und durch die festgelegte Auswahl von Außen ersetzt. Um zu unterscheiden, ob im aktuellen Zustand nur ein weiterer Übergang mit einer neuen Menge aktiver Agenten ausgeführt werden muss, oder zunächst alle Agenten an [mc]square übergeben werden müssen, sind zusätzlich kleine Änderungen an der Steuerung des Programmablaufs in CoreASM nötig.

Nichtdeterministische Wahl Die nichtdeterministische Wahl mit der **choose**-Regel wählt in CoreASM ähnlich wie der Scheduler einen von mehreren möglichen Werten für den Zustandsübergang aus. Wie oben muss diese Wahl durch eine Iteration über alle Möglichkeiten ersetzt werden.

Dazu wird das entsprechende Plugin durch ein neues ersetzt, das auf der gleichen Idee basiert wie der neue Scheduler. Auch hier ist die Beibehaltung des alten Programmablaufs der Hauptgrund für diese Vorgehensweise. Da die **choose**-Regel

aber mehrmals in einem Schritt aufgerufen werden kann, gibt es einige Unterschiede zum oben beschriebenen Vorgehen.

Trifft der Interpreter beim Ausführen eines abstrakten Syntaxbaums auf das Schlüsselwort *chosse*, wird das entsprechende Plugin aufgerufen. Dieses hat dabei jedoch keine Information darüber, welcher Agent welche Regel aufgerufen hat, da es lediglich vom Interpreter angestoßen wird. Um über alle Möglichkeiten der Auswahl iterieren zu könne, muss daher auch der Interpreter geändert werden.

Die neue Implementierung des Interpreters verändert sein Verhalten beim Aufruf des **choose**-Plugins. Dieses gibt bei Aufruf die Anzahl der Elemente der Auswahlmenge an den Interpreter, der gleichzeitig vom Scheduler die Information, welcher Agent die aktuelle Regel ausführt, erhält. Beide Informationen werden für jeden Aufruf des Plugins gespeichert und über die Schnittstelle an [mc]square gereicht. Dieses erstellt daraus wiederum alle möglichen Kombinationen und gibt sie nacheinander für jeden Zustandsübergang an den Interpreter zurück. Der Interpreter gibt für jeden Agenten die aktuelle Iterationsnummer an das Plugin, das dann das entsprechende Element der Menge auswählt.

Plugin Rahmenwerk Bei der Änderung des **choose**-Plugins wurde gleichzeitig das Plugin-Rahmenwerk erweitert. Ein neues Interface muss von den deterministischen Versionen von nichtdeterministischen Plugins implementiert werden, um kompatibel mit [mc]square zu sein. Dadurch können auch für zukünftige Plugins, die eine nichtdeterministische Auswahl enthalten, einfacher deterministische Versionen geschrieben werden.

Erweiterung von [mc]square

Die Implementierungen der Formel, des Model-Checkers und des Zustandsraums in [mc]square müssen zur Integration von CoreASM nicht abgeändert werden. Es wird lediglich ein neuer Simulator eingeführt, der die Interaktion mit CoreASM steuert. Wie oben bereits erwähnt betrifft dies vor allem die Iterationen über alle Möglichkeiten einer Auswahl. Zusätzlich wird hier beschrieben, wie die CTL-Formeln für die Verifikation von ASM-Modellen aussehen. Für deren Verarbeitungen waren zusätzlich einige kleinere Änderungen am Parser von [mc]square notwendig.

Simulator Der Zustandsraumaufbau erfolgt in einer neuen Simulatorkomponente von [mc]square, die über die in Abbildung 3.1 gezeigte Schnittstelle auf CoreASM zugreift und dessen Ablauf steuert. Beim Start des Model-Checking wird dazu das ASM-Programm geladen und initialisiert. Der dabei erzeugte Anfangszustand wird auf die atomaren Aussagen der zu verifizierenden Formel hin überprüft, serialisiert und im Zustandsraum abgelegt.

Zusätzlich liefert der Simulator eine Methode für einen als Byte-Feld übergebenen, speziellen Zustand alle Nachfolger zu erzeugen. Durch Wiederholen dieses Schritts für alle erzeugten Zustände wird der gesamte Zustandsraum aufgebaut. Nach der Deserialisierung des gegebenen Zustands wird zuerst die Menge aller Agenten von CoreASM erfragt. Über jeder möglichen Teilmenge von Agenten wird dann jeweils die Menge aller möglichen Kombinationen ermittelt, die durch die Verwendung von **choose** entstehen. Für jede mögliche Auswahl des **choose**-Plugins und für jede mögliche Teilmenge aktiver Agenten wird ein Zustandsübergang in CoreASM durchgeführt. Dieser wird wieder auf die atomaren Aussagen der zu verifizierenden Formel hin überprüft, serialisiert und im Zustandsraum abgelegt. Die Überprüfung, ob ein Zustand bereits im Zustandsraum enthalten ist und seine verlustfreie Kompression, wird von [mc]square übernommen.

Aufbau der CTL-Formeln Um Aussagen über den Zustandsraum zu treffen, werden in [mc]square CTL-Formeln verwendet. Sie sind wie in Kapitel 2.2 beschrieben aus temporalen Operatoren, Pfadquantoren, booleschen Junktoren und Aussagenvariablen aufgebaut. Der Parser zerlegt sie an Hand ihrer festgelegten Syntax in ihre atomaren Bestandteile. Dabei kann die Syntax der temporalen Operatoren, Pfadquantoren und booleschen Junktoren vom existierenden Parser direkt übernommen werden. Die Syntax der Aussagenvariablen wird wie hier beschrieben an CoreASM angepasst.

Wie oben beschrieben besteht der abstrakte Speicher von CoreASM aus mehreren Abbildungen, die die Universen, Funktionen und Regeln eines Zustands enthalten. Da die Regeln einer ASM in CoreASM konstant sind, wird im Folgenden die Syntax der Aussagenvariablen zu Funktionen und Universen betrachtet.

Die Elemente jedes Universums sind in jedem Zustand eindeutig durch ihren Namen bestimmt. Für Universen wird daher die folgende Syntaxschreibweise verwendet:

$$Element(name)$$

Diese Aussage ist erfüllt, wenn das Element eines Universums mit dem entsprechenden Namen im Zustand aktiv ist. *Element* ist ein Schlüsselwort des Parsers, während *name* ein Bezeichner ist und aus beliebigen Zahlen und Zeichen aufgebaut sein kann.

Jeder Funktionswert ist eindeutig durch den Namen der Funktion und durch den Wert des Definitionsbereichs bestimmt. Um einen Funktionswert mit einem beliebigen Wert vergleichen zu können, wird die folgende Syntaxschreibweise verwendet. Es werden beliebige n-stellige Funktionen unterstützt. Der Operator *sign* steht für die üblichen Vergeichsoperatoren.

$$Funktionsname(name_1, \ldots, name_n)\ sign\ Wert$$

Um allgemein den Wert einer n-stelligen Funktion zu vergleichen, werden die verschiedenen Domänen durch Kommas getrennt. Im Fall einer 0-stelligen Funktion, also einer Konstante wird der Parameter *name* durch einen Punkt ersetzt.

3.2.4 Verwandte Arbeiten

Es gibt bereits mehrere Veröffentlichungen zur Verifikation von ASMs. So sind bei allen in Kapitel 3.1.1 genannten Fallstudien immer auch Nachweise bestimmter Sicherheits-, Lebendigkeits- oder anderer Eigenschaften der Systeme erfolgt. Aufgrund ihrer formalen Darstellung erlauben ASMs händische Beweise, hier soll jedoch auf die Verwendung von Werkzeugen zur Verifikation eingegangen werden.

Schellhorn und Ahrendt präsentieren die formale Verifikation von ASMs, die bei der Standardisierung der Programmiersprache Prolog erstellt wurden, mit dem *Karlsruhe Interactive Verifier* (KIV) [Ahr95, SA97, Sch99]. Dabei werden die ASMs schrittweise zu Warren Abstract Machines [War83, BR94b] verfeinert. KIV ist ein interaktives Beweissystem, d. h., die Verifikation erfolgt benutzergesteuert. Giese, Kempe und Schönegge nutzen das KIV-System zur Verifikation von ASM-Spezifikationen am Beispiel einer DLX-Pipelining Architektur [GKS97].

Dold beschreibt eine Repräsentation von ASMs im *Prototype Verification System* (PVS) [Dol98], die zur Verifikation mit dem interaktiven Beweissystem genutzt werden kann. Dieses Vorgehen wird im Rahmen des Verifix Projekts (Verifizierende Compiler) zur mechanischen Verifikation von Compiler Back-Ends ebenso verwendet [DGVZ98], wie von Gargantini und Riccobene zur Verifikation der in Kapitel 3.1.1 beschriebenen Robotersteuerung [GR00].

Diese Arbeiten nutzen alle Systeme, in denen die Beweise durch Interaktion mit dem Benutzer geschehen. Diese Werkzeuge arbeiten also nur teilweise automatisiert. Daneben gibt es mehrere Ansätze zur vollautomatische Verifikation von ASMs mit Model-Checking.

Del Castillo und Winter übersetzen ASM-Spezifikationen der ASM-Workbench [Del99] in endliche Automaten [Win97, DW00], die mit dem Model-Checker *Symbolic Model Verifier* (SMV) [McM92] verifiziert werden. SMV ist im Gegensatz zu [mc]-square ein symbolischer Model-Checker. Die Übersetzung unterstützt die meisten Standard ASM-Regeln mit Ausnahme von **import** und **choose**, sowie beliebige n-stellige Funktionen. Als Beispiele werden verschiedene Eigenschaften eines Protokolls zur Speicherverwaltung in verteilten Systemen und die bekannte Produktionsanlage verifiziert [Win00]. Winter beschreibt eine Erweiterung, die auch die Verwendung von abstrakten Datentypen ermöglicht [Win01] und eine dazugehörige Fallstudie [GTW03]. Sowohl das Werkzeug zur Übersetzung in endliche Automaten, als auch die ASM-Workbench werden nicht mehr weiterentwickelt.

Im Rahmen des Projekts ISILEIT [PDIJG04] entwickelt Kardos einen Model-Checker [Kar05] für AsmL [GRS05]. Der Model-Checker arbeitet direkt auf den

AsmL-Spezifikationen, ohne sie in eine andere Eingabesprache zu übersetzen. Er wurde aber bisher nur für einfache Spezifikationen eingesetzt und ist nicht verfügbar. *Spec Explorer* [CGN+05] ist ein modellbasiertes Testwerkzeug für AsmL, das in Ansätzen spezielle vordefinierte Klassen von temporalen Eigenschaften verifizieren kann.

Tang und Ternovska stellen Model-Checking von ASMs durch *Answer Set Programming* (ASP) vor [TT05, Tan06]. ASP ist ein deklaratives, logisches Programmierparadigma, welches zur Lösung von kombinatorischen Suchproblemen entwickelt wurde. Zum Model-Checking werden ASMs zusammen mit einer temporalen Eigenschaft in ein Logikprogramm übersetzt und dessen Antwortmenge wird berechnet. Die Übersetzung unterstützt die ASM-Regeln der ASM-Workbench inklusive **import** und n-stelliger Funktionen.

Gargantini und Riccobenne übersetzen ASMs, die als AsmGofer [Sch02] und Asmeta-Spezifikationen [SGG+05] gegeben sind, nach Promela [GRR03], der Eingabesprache des Model-Checkers Spin [Hol97]. Die Übersetzung ist jedoch starken Einschränkungen unterworfen. So werden nur ASMs mit einem Agenten, nur eine Teilmenge der ASM-Sprachregeln und auch nur 0-stellige Funktionen unterstützt. Als einzige Regeln werden Zuweisungen und Bedingungen unterstützt.

Ouimet und Lungqvist verifizieren Timed ASM-Modelle [OL07b] mit *Uppaal* [BDL], indem sie sie in Zeitautomaten übersetzen. Vasilyev beschreibt einen Ansatz zur Verifikation von reaktiven Echtzeit-ASMs [Vas07]. Das Projekt befindet sich jedoch noch in der Entwicklungsphase und ein Werkzeug ist nicht verfügbar.

Eine frühere Version von CoreASM bietet ebenfalls die Möglichkeit der Übersetzung nach Promela [FGM07, Ma07]. Sie geht dabei in mehreren Punkten über die Arbeit von Garantini und Riccobenne hinaus. So werden alle Standard ASM-Regeln, n-stellige Funktionen und verteilte ASMs unterstützt. Die Übersetzung ist in der aktuellen Version von CoreASM jedoch nicht mehr enthalten, da sie nur mit großem Aufwand an Erweiterungen von CoreASM anpassbar ist.

Der hier vorgestellte Ansatz unterscheidet sich von den anderen oben genannten Model-Checkern dadurch, dass die ASMs nicht in eine andere Sprache übersetzt werden müssen, sondern ihre Simulation dazu genutzt wird, den Zustandsraum aufzubauen. Die nötige Serialisierung zur Verifikation in [mc]square findet ohne Informationsverlust statt. Damit werden ASMs direkt und ohne möglichen Verlust an Ausdrucksstärke bei der Übersetzung unterstützt. Die Vorteile dieses Vorgehens liegen in der Möglichkeit, Gegenbeispiele und Zeugen im ASM-Zustandsraum anzeigen zu können und in der direkten Unterstützung aller Modellierungs- und Simulationseigenschaften von CoreASM, wie n-stelliger Funktionen, verteilter Mehragenten-ASMs und erweiterter Regeln. Die verifizierbaren Formeln können entsprechend Aussagen zu allen Elementen eines Zustands in CoreASM treffen. Zusätzlich kann der Ansatz zukünftige Erweiterungen von CoreASM direkt und mit wenig Aufwand integrieren. Nachteilig wirkt sich aus, dass die Datenstrukturen und der Programmablauf in

CoreASM auf Verständlichkeit und Erweiterbarkeit ausgelegt sind und ihre Laufzeit und Speicherbedarf daher nicht für eine Verwendung in einem Model-Checker optimiert sind.

4 Strukturmodellierung mit Business Object Notation

Wie in Abbildung 1.1 dargestellt spielen neben den funktionalen Anforderungen, die wie in Kapitel 3 beschrieben mit ASMs modelliert und verifiziert werden können, auch die qualitativen eine wichtige Rolle beim Entwurf eines Systems. Diese beeinflussen insbesondere die Architektur eines Systems, also seine Grobstrukturen [SG96]. Neben der Modellierung der funktionalen Anforderungen und ihrer Verfeinerung zur Implementierung, die wie oben beschrieben mit ASMs möglich ist, müssen zur Unterstützung der qualitativen Anforderungen auch Modelle der Systemstruktur erstellt werden.

Der Übergang von den informellen, qualitativen Anforderungen zur Architektur des Systems ist trotz Heuristiken in Form von Architekturmustern ein komplexer Vorgang, dessen Erfolg in hohem Maße von der Erfahrung des Architekten abhängt. Dies liegt an der Schwierigkeit der wechselseitigen Beeinflussung der betroffenen Qualitäten bei der Kombination mehrerer Muster zur Gesamtarchitektur. Neben verschiedenen Modellierungssprachen, die die Struktur eines Systems beschreiben, haben sich deshalb auch Prozesse für den Entwurf und die Evaluation dieser Struktur etabliert, wie z. B. *Attribute-Driven Design* [BKBI02], *Architecture Tradeoff Analysis Method* [KKB+98] und weitere szenariobasierte Evaluationsmethoden [DN02].

Neben der BON gibt es auch andere Modellierungssprachen für die Struktur eines Systems, insbesondere einige Diagrammarten der *Unified Modelling Language* (UML) sind eine weit verbreitete Alternative. Für den Einsatz der BON sprechen die vier wichtigsten Anforderungen, die zur Verwendung in der in Kapitel 6 vorgestellten Fallstudie gestellt wurden: Erstens soll die Strukturmodellierung mit der funktionalen Modellierung der ASMs kombinierbar sein und benötigt daher keine eigenen Diagrammarten zur Verhaltensmodellierung. Wie in Kapitel 1 erläutert spielen beim Entwurf eingebetteter Software Zeit- und Plattformeigenschaften eine wichtige Rolle [HS06]. Zweitens sollen diese deshalb in den Modellen von Anfang an darstellbar sein. Grundlegende Idee des hier vorgestellten Ansatzes ist die Formulierung dieser Eigenschaften in den Vor- und Nachbedingungen des DBC, welches deshalb in den Modellen darstellbar sein soll. Drittens müssen mit den Modellen alle drei unten vorgestellten Arten von Architektursichten dargestellt werden können. Viertens sollen die Modelle leicht, schnell und einfach zu erlernen und einzusetzen sein.

In Ansätzen ist die Strukturmodellierung beispielsweise durch die Darstellung von Schnittstellen auch mit ASMs möglich [Anl00], sie erfüllt jedoch nicht alle oben beschriebenen Anforderungen. Insbesondere lassen sich mit ASMs nicht alle drei Architektursichten darstellen: Sie können am ehesten für die unten vorgestellte Komponentensicht auf die Architektur eines Systems eingesetzt werden, Modulsichten und Verteilungssichten lassen sich mit ihnen aber nicht einfach darstellen. Beim Entwurf der Fallstudie fehlte den Entwicklern bei der alleinigen Nutzung von ASMs zur Systemmodellierung auch eine informellere Darstellung, die in frühen Phasen des Entwurfs auch widersprüchliche Darstellungen aufnehmen kann und in der die wichtigen Strukturinformationen schnell und einfach dargestellt werden können.

Die BON erfüllt dagegen alle vier Anforderungen: Sie lässt sich wie unten gezeigt mit den ASMs kombinieren, untersützt DBC, eignet sich zur Darstellung aller drei Typen von Architektursichten und ist eine sogenannte leichtgewichtige Methode. Sie wurde auch schon zum Entwurf eingebetteter Systeme wie z. B. einer Prozesssteuerung in der Produktion eingesetzt [WN95]. Die Entscheidung für den Einsatz von BON und gegen UML hat die folgenden Gründe:

Da die BON keine eigene Verhaltensmodellierung bietet, kann sie einfach mit der funktionalen Modellierung der ASMs kombiniert werden. Statische und dynamische Modelle liefern Aussagen zur Verteilung von Funktionalität auf Module und deren Interaktion, machen jedoch keine Aussage zum Entwurf oder zur Implementierung dieser Funktionalität. ASMs sind somit die ideale Ergänzung dieser Strukturinformationen und treten nicht in Konkurrenz zu anderen Verhaltensmodellen, wie sie z. B. in der UML vorkommen. Es gibt auch bereits Ansätze die BON mit einer anderen formalen Funktionsmodellierung nämlich Z zu kombinieren [PO98].

Die BON bietet mit nur zwei Arten der grafischen Darstellung die Möglichkeit alle Arten von Architektursichten dazustellen. Diese beiden grafischen Darstellungen sind leicht zu erlernen und lassen sich vollständig auf wenigen Seiten beschreiben [Wal98]. Im Gegensatz dazu existieren in der UML 2 sechs verschieden Diagrammarten zur Strukturmodellierung.

Die Unterstützung von DBC ist in der BON eingebaut und muss nicht über eine zusätzliche Sprache wie die *Object Constraint Language* bei der UML nachgereicht werden. BON wird vor allem im Bereich der Programmiersprache Eiffel eingesetzt und wurde auch für diese Verwendung entwickelt. Eiffel ist die Sprache für die DBC entwickelt wurde und eine der wenigen Sprachen, die es ohne die Verwendung zusätzlicher Werkzeuge, unterstützen.

Die folgenden Unterkapitel beschreiben genau, wie die ersten drei der oben genannten Anforderungen mit Hilfe der BON umgesetzt werden: Abschnitt 4.1 beschreibt die Kombination von BON mit ASMs. Anschließend geht Abschnitt 4.2 darauf ein, wie mit Hilfe des DBC die Anforderungen an eingebettete Software in die Strukturmodellierung eingebunden werden können, die sich aus seiner Interaktion mit der Umgebung, also seiner Umwelt und seiner Plattform, ergeben. Anschließend

geht Abschnitt 4.3 auf die Darstellung unterschiedlicher Sichten ein bevor in 4.4 verwandte Arbeiten vorgestellt werden.

4.1 Kombination mit abstrakten Zustandsmaschinen

Hier wird eine Möglichkeit beschrieben, ASMs in die BON zu integrieren. Dadurch soll kein Vorgehen beim Entwurf vorgeschrieben werden. So kann die Integration ebenso von der in BON modellierten Struktur starten, die dann nach und nach durch funktionale ASM-Modelle erweitert wird, wie auch ausgehend von ASMs die Struktur aufgebaut werden kann. Mischformen zwischen diesen beiden Extremen sind ebenso denkbar. Im Rahmen der Fallstudie aus Kapitel 6 werden z. B. zunächst die Anforderungen an den Hauptalgorithmus als ASM modelliert und anschließend eine Struktur, die die qualitativen Anforderungen erfüllt, um diese ASM aufgebaut. Die Aufgaben weiterer Teile dieses Strukturmodells werden anschließend mit weiteren ASMs beschrieben.

Durch die Unterscheidung zwischen funktionalen und qualitativen Anforderungen und deren Entsprechungen in Verhaltens- bzw. Strukturmodellen im Entwurf wird deutlich, dass die BON- und ASM-Modelle unterschiedliche Aspekte eines Systems darstellen. Dennoch sind, wie in Abschnitt 4.1.1 beschrieben, bei ihrer Kombination einige Schwierigkeiten zu überwinden. Ein wichtiges Kriterium für eine erfolgreiche Implementierung des Systems ist die Konsistenz der unterschiedlichen Modelle. d. h., dass die Darstellungen in der BON und die ASMs sich nicht widersprechen dürfen. Wie Abschnitt 4.1.2 beschreibt ist dies im Fall statischer BON-Diagramme einfach zu überprüfen, während die Übereinstimmung dynamischer BON-Diagramme mit möglicherweise mehreren als ASM modellierten Objekten schwerer ist.

4.1.1 Vorgehen

Um ASMs in die BON zu integrieren, müssen Entsprechungen zwischen den beiden festgelegt werden. Größte Schwierigkeit ist dabei die fehlende ASM-Entsprechung der Unterscheidung von Entwurfs- und Laufzeit in der BON. In statischen Diagrammen werden Klassen, also Quelltextdokumente und ihre Beziehungen zur Entwurfszeit dargestellt. Dynamische Diagramme stellen dagegen Laufzeitobjekte und ihre Interaktionen dar. Zwischen den Objekten und Klassen gibt es aber natürlich die Beziehung, das jedes Objekt vom Typ einer Klasse ist. Von einer Klasse können dabei aber durchaus mehrere Objekte instantiiert werden. Bei den ASMs sind die Maschinen, also die Universen, Regeln und Funktionen sowohl Entwurfs- als auch Laufzeitmodule. Die abstrakten Maschinen, auf denen sie ausgeführt werden entsprechen nicht den Objekten der dynamischen BON-Diagramme sondern eher den physikalischen Maschinen auf denen sie laufen.

Hier werden einzelne ASM-Maschinen als Entsprechungen von Klassen im Strukturmodell verwendet. Ebenso wie Klassen existieren sie als Quelltext und können über Verfeinerungen auch automatisiert in diesen umgewandelt werden [Sch01b, Sch01c]. Die einzelnen Regeln der Maschine entsprechen dann den Klassenmethoden und werden in BON-Diagrammen als Kommentare referenziert. Abbildung 6.2 auf Seite 89 stellt dies beispielhaft für die `acc_main_rule` dar, die mit demselben Regelnamen auch im ASM-Modell existiert. Wie in der Abbildung zu sehen ist müssen keineswegs alle Methoden durch eine ASM-Regel beschrieben werden. Solche Methoden wie beispielsweise die Methode `speed_error` sind im abstrakten ASM-Modell nicht enthalten und können entweder in einem Verfeinerungsschritt spezifiziert werden oder in der Implementierung dazukommen.

Die Vor- und Nachbedingungen solcher Methoden können teilweise als formale Spezifikationen verstanden werden, die mit Hilfe des in Kapitel 3 vorgestellten Model-Checkers verifiziert werden können. So kann etwa auch überprüft werden, ob sich die ASM-Regel `acc_main_rule` aus Abbildung 6.2 nur im erlaubten Geschwindigkeitsintervall aktivieren lässt. Ebenso müssen die Klasseninvarianten dann für eine ASM gelten und können zum Teil auch verifiziert werden.

Den Laufzeitobjekten der dynamischen BON-Modelle entsprechen die Entitäten, die zur Laufzeit die Programme der Maschinen ausführen. Dies sind bei den ASMs die Agenten, die in einem speziellen Universum der gesamten ASM definiert werden. Wenn eine ASM-Maschine einer Klasse entspricht werden alle nichttrivialen Systeme aus mehreren Agenten bestehen.

Mehragenten-ASMs werden in synchrone und asynchrone unterteilt. Mit der BON sind jedoch nur sequentielle Systeme darstellbar. Diese entsprechen synchronen Mehragenten-ASMs, da Objektaufrufe und -Antworten hier synchron stattfinden. Klassen oder Objekte, die asynchron kommunizieren bzw. ausgeführt werden kommen in verteilten Systemen vor. Diese sind in der BON nicht enthalten, da auch die von der BON avisierte Implementierungssprache Eiffel in ihrer aktuellen Version das in Kapitel 2.4 vorgestellte Verteilungs- und Nebenläufigkeitskonzept SCOOP noch nicht enthält.

Wie in der Vorstellung von SCOOP erläutert, wird die Sprache Eiffel für nebenläufige und verteilte Systeme nur um das eine Schlüsselwort **separate** erweitert. Dieses zeigt an, dass eine Klasse auf einem anderen Prozessor ausgeführt werden soll, unabhängig davon ob dieser Prozessor als eigener Rechenknoten eines verteilten Systems oder als nebenläufiger Prozess eines Betriebssystems implementiert ist. SCOOP schließt somit die Implementierung nebenläufiger Software ein, die häufig bei eingebetteten Systemen vorkommt. Die Kommunikation zwischen separaten Klassen findet asynchron statt. Diese Erweiterung entspricht somit genau der in BON fehlenden Klasse der asynchronen Mehragenten-ASMs. Daher wird die BON hier um die Kennzeichnung solcher separaten Klassen erweitert. Dazu wird zusätzlich zu den in Abbildung 2.3 auf Seite 19 gezeigten komprimierten Klassendarstellungen,

die von Abbildung 4.1 eingeführt.

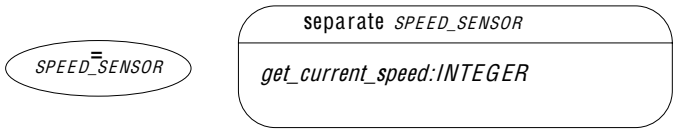

Abbildung 4.1: Komprimierte und ausführliche Darstellung einer separaten SCOOP-Klasse in BON.

Damit sind Entsprechungen zu allen Typen von ASMs in der BON darstellbar. Aus Sicht der BON sind zwar Entsprechungen von Klassen und Objekten gefunden, jedoch nicht für einige andere Elemente: Die Dienstbeziehungen Aggregation und Klient zwischen Klassen gelten auch für ASMs, Vererbung von Funktionalität gibt es bei ASMs aber nicht. Für erbende Klassen ist es empfehlenswert nur die neu definierten oder überschreibenden Methoden einer Unterklasse als ASM-Regeln zu beschreiben, wie es auch im späteren Quelltext der Fall ist. Soll die ASM simuliert werden, müssen die quasi ererbten ASM-Regeln aber kopiert werden. Typisierte Schnittstellen der Klassen entsprechen den beobachteten, geteilten und Ausgabelokationen der ASMs. Die Invarianten, Vor- und Nachbedingungen des DBC haben keine Entsprechungen bei den ASMs, eigenen sich aber wie oben erwähnt hervorragend als Anhaltspunkte für die Verifikation. Für die Cluster der dynamischen und statischen Diagramme der BON gibt es keine ASM-Entsprechung. Sie dienen aber in der BON zur Betonung von Zusammenhängen in unterschiedlichen Sichten auf die Struktur des Systems und haben auch keine Entsprechung in der Implementierungssprache Eiffel.

4.1.2 Konsistenz

Wird ein System in mehreren, verschiedenen Repräsentationen dargestellt, ist die Konsistenz, d. h. Widerspruchsfreiheit dieser Repräsentationen sofern nicht explizit erlaubt und gekennzeichnet erforderlich, um dieses System implementieren zu können. Die Situation ist ähnlich der oftmals geforderten Widerspruchsfreiheit von Anforderungen mit dem Unterschied, dass Systemmodelle später im Entwicklungsprozess eingesetzt werden und die Widerspruchsfreiheit mit der zeitlichen Nähe zur Implementierung immer mehr an Bedeutung gewinnt. Konsistenz wird wie in Kapitel 4.3 beschrieben für unterschiedliche Sichten auf das System, die in einer Modellierungssprache formuliert sind, gefordert ebenso wie für unterschiedliche Diagrammarten einer Sprache wie etwa der BON oder der UML.

Hier wird diskutiert, wie die Konsistenz zwischen einer Systemdarstellung in möglicherweise mehreren BON-Diagrammen und einer ASM sichergestellt werden kann. Dabei wird davon ausgegangen, dass die BON-Diagramme und die ASM selbst konsistent sind. Für ASMs ist diese Konsistenz aufgrund ihrer formalen

Definition implizit gegeben während sie für BON-Diagramme wie in Kapitel 4.3 beschrieben sichergestellt werden muss. Sofern Konsistenz nicht implizit gegeben ist, ist ihre automatisierte Überprüfung wünschenswert. Hier wird nur auf die händische Überprüfung eingegangen, die zum Teil jedoch auch in einem Werkzeug automatisiert werden könnte.

Damit eine in einem statischen BON-Diagramm beschriebene Klasse mit der sie modellierenden ASM-Maschine konsistent ist, müssen die folgenden Bedingungen überprüft werden: Jede ASM-Regel muss dem Namen nach mit einer Klassenmethode übereinstimmen. Ist die Schnittstelle der Methode genauer typisiert, müssen ihre Ein- und Ausgabeparameter den beobachteten bzw. Ausgabelokationen der ASM entsprechen. Geteilte Lokationen der ASM können sowohl Ein- als auch Ausgabe der Methode sein. Aggregations- und Klientenbeziehungen zwischen Klassen müssen auch zwischen den entsprechenden ASMs unterschiedlicher Agenten bestehen. Die Invarianten, Vor- und Nachbedingungen einer Klasse bzw. ihrer Methoden beschreiben Bedingungen an die Semantik derselben. Sie können deshalb nur durch Ausführung der ASM überprüft werden. Teilweise kann die Einhaltung der Verträge durch Simulation mit Testfällen validiert oder mit Hilfe eines Model-Checkers für ASMs verifiziert werden. Es sind jedoch auch nicht verifizierbare Ausdrücke in den Verträgen denkbar und genauso kann es von Interesse sein Eigenschaften der ASMs zu verifizieren, die nicht in den Verträgen enthalten sind.

Für dynamische Diagramme der BON ist die Konsistenzprüfung mit ASMs nicht durch Überprüfen von Äquivalenzen sicherzustellen. Hierbei ist zu beachten, dass ein dynamisches Diagramm lediglich ein wichtiges Anwendungsszenario des Systems darstellt. Ist ein dynamisches Diagramm konsistent mit den statischen und diese wiederum mit der ASM, ist sichergestell, dass jedem Objekt des dynamischen Diagramms eine ASM entspricht. Das Durchspielen eines durch das Diagramm dargestellten Szenarios im Kopf oder mit einem ASM-Simulator kann dann durchgeführt jedoch kaum automatisiert werden, ähnelt es doch der Durchführung eines natürlichsprachlich spezifizierten Testfalls. Ist das Durchlaufen des Szenarios mit der ASM möglich, dann ist sie konsistent mit dem dynamischen BON-Diagramm. Es liegt ähnlich der Konsistenz unterschiedlicher Sichten auf die Architektur eines Systems in der Verantwortung des Architekten dies sicherzustellen.

4.2 Modellierung physikalischer Anforderungen

Wie in Kapitel 1 beschrieben unterscheiden sich eingebettete Systeme grundlegend von anderen Softwaresystemen, da sie mit der sie umgebenden Welt interagieren [HS06]. Henzinger und Sifakis unterscheiden dabei Einschränkungen des Systems, die durch Reaktionen auf die Umgebung und solche, die durch die Ausführung auf einer Plattform entstehen. Typische Beispiele für Erstere sind Echtzeitbedingungen

bzw. beschränkte Ressourcen wie Energie oder Speicherplatz für Letztere. Sie werden hier zusammenfassend als physikalische Anforderungen bezeichnet.

Diese Eigenschaften des Systems bzw. die Anforderungen an das System oder vom System an seine Umgebung müssen bei eingebetteten Systemen beachtet werden. Da diese Eigenschaften entscheidenden Anteil an der Funktionstüchtigkeit und am Erfolg des Systems haben, müssen sie möglichst früh im Entwurfsprozess mitmodelliert werden. Dies geschieht hier durch ihre Einbindung in die Verträge des DBC in der BON.

Dies ist insofern sinnvoll, als dass dadurch sowohl die Anforderungen des Systems an seine Umgebung ebenso wie die von ihm zu garantierenden Eigenschaften beschrieben werden können. So können beispielsweise in den Vorbedingungen Anforderungen an die Plattform gestellt werden und in den Nachbedingungen Zeitschranken für die Ausführung gefordert werden. Durch die Integration von DBC in die BON können solche Eigenschaften vom ersten Modell an eingebunden werden. Dabei ist zu beachten, dass die damit erweiterten BON-Modelle früh im Entwurfsprozess zum Einsatz kommen und insofern eher Anforderungen an das System formulieren als tatsächliche Ausführungszeiten zu garantieren.

Damit sind die für eingebettete Systeme durch die physikalische Anforderungen zu Stande kommenden Einschränkungen Teil des Strukturmodells in der BON und die ihnen zu Grunde liegenden Anforderungen müssten konsequenter Weise als qualitative Anforderungen verstanden werden. Während dies für Plattformeigenschaften, die z. B. auch in Qualitätsbäumen vorkommen, üblich ist, ist die Zuordnung von Zeitanforderungen umstritten. Wegen der in Abschnitt 4.1.1 dargestellten Entsprechungen, ist die Bedeutung der Verträge für die in den ASMs modellierten, funktionalen Anforderungen jedoch eindeutig und bei Bedarf auch als solche zu verstehen.

Mit Hilfe der durch DBC beschriebenen Schnittstellen können Unstimmigkeiten bereits beim Zusammenbau von Komponenten, gerade bei ihrer Wiederverwendung entdeckt werden. Daher wird die Verwendung von DBC vor allem für wiederverwendbare Komponenten empfohlen und erhöht die Verlässlichkeit des konstruierten Systems [MJ97]. Die im Vertrag festgehaltenen Vor- und Nachbedingungen enthalten sowohl für Methoden als auch für ihre Nutzer Rechte und Pflichten. So kann eine Methode immer davon ausgehen, dass ihre Klienten die Vorbedingung einhalten, unabhängig davon, ob sie in dem System eingesetzt wird, für welches sie ursprünglich geplant war oder ob sie in einem neueren System wiederverwendet wird. Dann muss sie aber auch die Nachbedingung garantieren. Aus Sicht des Klienten ist die Pflicht das Erfüllen der Vorbedingung, während er sich auf das Einhalten der Nachbedingung durch die Methode verlassen darf. Enthalten die Verträge physikalische Anforderung, dann werden auch die Plattform und die Umwelt des Systems zu seinen Vertragspartnern mit entsprechenden Rechten und Pflichten.

Beugnard et al. [BJPW99] teilen Verträge in vier Kategorien mit steigender Anzahl zu verhandelnder Eigenschaften ein:

syntaktische Verträge definieren die Operationen einer Komponente mit Ein- und Ausgabeparametern sowie mögliche Ausnahmen, die während der Ausführung behandelt werden müssen. Typische Beispiele sind die Datentypisierungssysteme gängiger Programmiersprachen.

Verhaltensverträge beschreiben zusätzlich ein abstrahiertes Komponentenverhalten durch Zusicherungen. Typisches Beispiel ist das klassische DBC, wie es in der Sprache Eiffel implementiert ist.

Synchronisierungsverträge sind wichtig, wenn eine Komponente mehrere Dienstnutzer bedient. Die Verträge garantieren den Dienstnutzern, dass die angeforderten Dienste unabhängig von anderen Dienstanfragen bearbeitet werden.

QoS-Verträge (quality of service) quantifizieren das mit den anderen Verträgen festgelegte Verhalten. Dies kann sowohl statisch als auch dynamisch zur Laufzeit geschehen.

Ein Vertrag muss Folgendes zur Verfügung stellen [Col01]:

Formalismus zur Spezifikation Zur Spezifikation wird der von Walden und Nerson eingeführte Formalismus der BON verwendet [WN95]. Diese Sprache für Zusicherungen orientiert sich an den in Eiffel verfügbaren Schlüsselwörtern und ist dadurch direkt in die Implementierung abbildbar.

Ersetzungsregel Wie beim DBC üblich, gilt, dass Vorbedingungen bei Vererbung nur abgeschwächt und Nachbedingungen nur verschärft werden dürfen. In der Implementierung wird dies von der Eiffel-Laufzeitumgebung sichergestellt.

Technik zur Überwachung Dies kann vor oder zur Laufzeit des Systems geschehen. Hier wird in der Implementierung die in Eiffel eingebaute Überwachung des DBC während der Laufzeit verwendet. Um die Einhaltung der Verträge überprüfen zu können, müssen die in ihnen enthaltenen Eigenschaften zur Laufzeit quantifizierbar und messbar sein.

In den folgenden beiden Abschnitten wird genauer auf die Formulierung der beiden von Henzinger und Sifakis unterschiedenen Anforderungsarten mit Hilfe des DBC und ihre Kategoriezugehörigkeit nach Beugnard et al.eingegangen.

4.2.1 Modellierung von Plattformanforderungen

Plattformeigenschaften, die beim Entwurf eingebetteter Systeme beachtet werden müssen, entstehen meist durch Ressourcenbeschränkungen des Systems. Typische Beispiele sind etwa begrenzter Platz im Programm- und Datenspeicher sowie im

persistenten Speicher, begrenzte zur Verfügung stehende Energie oder begrenzte Leistungsfähigkeit der CPU. Werden sie in die Verträge einer Klassen mit aufgenommen, dann wird die Plattform auf der die Klasse ausgeführt wird zum Vertragspartner dieser Klasse. Die Vorbedingung muss beim Aufruf einer Methode dann nicht mehr nur vom aufrufenden Klienten sondern auch von der Plattform sichergestellt werden. Ebenso hat die Klasse nicht mehr nur Pflichten gegenüber ihrem Klienten sondern ebenso gegenüber der Plattform.

Verträge, die Aussagen über die Plattform der Komponente enthalten, gehören meist zur vierten oben beschriebenen Kategorie, da Aussagen zum Ressourcenverbrauch üblicherweise zu den Qualitäten eines Systems gerechnet werden. Sie beschreiben sowohl erwartetes als auch garantiertes Verhalten in Bezug auf die Plattform. So kann eine Komponente fordern, einen gewissen Anteil geteilter Ressourcen zugeteilt zu bekommen oder sich zu einem beschränkten Ressourcenverbrauch verpflichten.

Plattformverträge werden je nach Entwicklungszeitpunkt unterschiedlich eingesetzt werden. Beim Entwurf helfen sie vor allem bei der Verteilung wiederverwendeter Komponenten auf Hardware-Plattformen. Für neu entwickelte Komponenten können darin gemessene oder abgeschätzte Ressourcenanforderungen in der Vorbedingung gefordert bzw. Ressourcenverbrauch in der Nachbedingung zugesichert werden.

Beim Testen des Systems werden die Verträge als Unterstützung bei der Fehlersuche verwendet. Ist eine der Bedingungen verletzt, wird die Ausführung unterbrochen und diese Bedingung benannt. Da in den Verträgen festgeschrieben ist, wer für die Einhaltung der Bedingung verantwortlich ist, können Fehler schnell gefunden werden.

Zur Laufzeit können die Verträge entweder ausgeschaltet werden oder als permanenter, defensiver Mechanismus zur Erhöhung der Verlässlichkeit des Systems mitlaufen. Wird eine Bedingung verletzt, dann wird eine Ausnahmebehandlung in der zuständigen Komponente aufgerufen. So könnte beispielsweise eine Komponente, die mehr Speicherplatz verbraucht, als zugesichert nicht benötigten Speicherplatz freigeben.

Um die in den Verträgen enthaltenen Plattformeigenschaften zu messen, wird meist eine weitere Software-Komponente verwendet werden müssen. Diese übernimmt im Sinne des DBC alle Rechte und Pflichten der Plattform. Eine Komponente, die den Speicher der Plattform repräsentiert könnte dessen Belegung z. B. messen und auch verwalten. Wird die den Speicher betreffende Vorbedingung einer Komponente verletzt müsste die Speicherkomponente neuen freigeben und könnte gleichzeitig zur Überprüfung von den Speicher betreffenden Nachbedingungen eingesetzt werden.

Welche Plattformeigenschaften in den Verträgen angesprochen werden können hängt von deren Verfügbarkeit auf der Software-Ebene ab. Im Allgemeinen werden sie über spezielle Routinenaufrufe abgefragt werden, die dann einfach in den Vorbedingungen verwendet werden können. Dabei ist aber darauf zu achten, dass diese

Routinen Anfragen und keine Kommandos, die den Systemzustand ändern, sind.

Viele Anforderungen an die Plattform werden umgangssprachlich vermutlich als Invariante formuliert. Dabei ist jedoch zu beachten, dass Invarianten in Eiffel während der Ausführung einer Klassenmethode eines objektorientierten Systems durchaus verletzt werden dürfen. Invarianten entsprechen eher einer Vor- und Nachbedingung für alle Methoden einer Klasse inklusive des Konstruktors.

4.2.2 Modellierung von Umweltanforderungen

Anforderungen, die durch die Interaktion eines eingebetteten Systems mit seiner Umwelt entstehen, beinhalten häufig zwei unterschiedliche Typen von Anforderungen für die Vor- und die Nachbedingung. Die in den Vorbedingungen beschriebenen Anforderungen der Komponente an die Umwelt beschreiben die Umgebung und verlangen beispielsweise den Empfang bestimmter Funksignale, bei lokalisierten Anwendungen einen bestimmten Aufenthaltsort oder eine bestimmte Energiequelle. In Nachbedingungen, die die Anforderungen der Umwelt an das System beinhalten werden typischerweise Zeitanforderungen, wie z. B. eine bestimmte Antwort- oder Zykluszeit enthalten.

Diese Verträge passen am ehesten in die zweite oben beschriebene Kategorie, auch wenn die Zuordnung nicht eindeutig ist. Für die Verwendung in den unterschiedlichen Phasen eines Systementwurfs gelten die gleichen Aussagen wie für die Plattformeigenschaften. Anzumerken ist, dass es auch zur Laufzeit sinnvoll sein kann beim Überschreiten einer Zeitschranke eine Ausnahmebehandlung der betroffenen Komponente zu starten, die z. B. einen Fehlerzähler implementieren könnte, der die Komponente beim Überschreiten einer gewissen Fehleranzahl neu startet.

Im Rahmen dieser Arbeit werden Zeitanforderungen in der Laufzeitumgebung mit einer Stoppuhr überprüft. Mögliche Anforderungen, die in die Verträge in den BON-Diagrammen eingefügt werden können, sind minimale und maximale Ausführungsdauer sowie periodischer Aufruf.

Der Hauptvorteil bei der Verwendung von Verträgen für Plattform- und Umweltanforderungen ist die klare Schnittstelle mit ihren präzisen Auflagen für die Klasse und ihre Plattform bzw. Umwelt. Zwar könnten die Bedingungen ebenso beim Betreten oder Verlassen einer Methode überprüft werden, dadurch ist jedoch keine Zuständigkeit bei Verletzung der Bedingungen gegeben. Ebenso wird das mehrmalige Überprüfen der Bedingungen verhindert und ein Laufzeitmechanismus geboten, der insbesondere für wiederverwendete Komponenten wichtig ist.

4.3 Darstellung unterschiedlicher Sichten

Die Vielfältigkeit von Strukturen, aus denen Software aufgebaut ist, wurde bereits von Parnas erkannt [Par74]. Die Architektur eines nichttrivialen Systems ist mul-

tidimensional und damit zu komplex für eine einzelne, vollständige Darstellung. Analog zu Gebäudearchitekturen entwickelten Perry und Wolf daher die Idee unterschiedlicher Sichten auf eine Softwarearchitektur [PW92]. Dies führte zu mehreren Vorschlägen für Sichten, die zur Dokumentation einer Softwarearchitektur notwendig sind: Kruchtens 4+1 Sichten [Kru95], die vier Siemens-Sichten nach [HNS00], der hier verfolgte Ansatz *views and beyond* von Clements et al. [CBB+03] und schließlich ein IEEE-Standard [IEE00].

Eine Sicht schränkt die gezeigten Typen von Elementen und ihre Relationen ein, um einen spezifischen Aspekt des Systems zu beschreiben. Eine Architekturdokumentation besteht aus allen relevanten Sichten auf das System und den Informationen, die für alle diese Sichten gelten [CBB+03]. In den oben erwähnten Ansätzen von Kruchten [Kru95] und Hofmeister et al. [HNS00] werden die zu verwendenden Sichten direkt festgelegt. Beispielsweise schlägt Kruchten die folgenden vor:

Logische Sicht beschreibt Verhaltensanforderungen

Prozesssicht unterstützt Nebenläufigkeit und verteilt Objekte auf Ausführungsfäden

Entwicklungssicht unterteilt die Software in Module

Physikalische Sicht verteilt Software-Elemente auf Ausführungsknoten

+1 Sicht zeigt wichtige Anwendungsfälle und Szenarien

Wie an diesen Sichten oder am Beispiel speziell entworfener Sichten wie der Entscheidungssicht [CNPC06] zu sehen ist, hängt die Relevanz einzelner Sichten sehr stark vom betroffenen Anspruchsträger ab. Im IEEE-Standard wird deshalb empfohlen, eigene Sichten zu entwerfen, die den Bedürfnissen einzelner Gruppen von Anspruchsträgern gerecht werden. Clements et al. schlagen vor, die Sichten an der Software-Architekur und den Interessen der Stakeholder zu orientieren und beschreiben einen Prozess, mit dem systematisch entschieden wird, welche Sichten dokumentiert werden müssen [CBB+03].

Zu diesem Zweck werden Sichten in die folgenden drei Kategorien eingeteilt:

Modulsichten beschreiben, wie ein System aus Quelltexteinheiten zusammengesetzt ist und welche Beziehungen zwischen diesen bestehen

Komponentensichten zeigen das System als Menge von Laufzeitelementen, ihre Beziehungen und Interaktionen

Verteilungssichten setzen die Software in Beziehung zu Elementen, die keine Software sind

Als Faustregel empfehlen Clements et al., zur Dokumentation mindestens eine Sicht aus jeder Kategorie zu verwenden, ohne außer Acht zu lassen, dass jede Sicht nicht nur Vorteile sondern auch Kosten für ihre Erstellung und Wartung mit sich bringt. Wiederkehrende Entwurfsformen in jeder der drei Kategorien werden als Architekturstile oder -muster bezeichnet und in Katalogen gesammelt, wie z. B. von Booch [Boo06].

Wie für die Widerspruchsfreiheit der Kombination aus ASMs und BON-Diagrammen müssen unterschiedliche Sichten auf ein System auf ihre Konsistenz hin überprüft werden. Während bei frühen Systementwürfen Inkonsistenzen auch erwünschte Repräsentationen von Entwurfsalternativen sein können, müssen diese spätestens für die Implementierung beseitigt werden. Es gibt Ansätze die Konsistenzprüfung durch Prozesse und Werkzeuge zu unterstützen, wie den zur Prüfung von BON-Diagrammen nach Gao [Gao04]. Bei semiformalen Darstellungen wie der BON liegt die Konsistenzsicherung jedoch meist in der Verantwortung des Architekten.

In den folgenden drei Abschnitten wird beschrieben wie alle drei Kategorien von Sichten mit der BON erstellt werden können. Dabei wird das Zusammenführen von Objekten oder Klassen zu Clustern sowohl in den statischen als auch dynamischen Diagrammen der BON ausgenutzt, um die für die jeweilige Sichtkategorie wichtigen Aspekte des Systems zu betonen.

4.3.1 Modulsichten

Modulsichten lassen sich mit den statischen Diagrammen der BON erstellen. Sie zeigen die geforderten Quelltexteinheiten, namentlich Klassen, und die Beziehungen zwischen ihnen, namentlich Vererbung, Aggregation und Dienstnutzung. Bei komplexen Systemen ist es jedoch meist nicht sinnvoll alle Klassen mit ihren kompletten Schnittstellen sowie alle Beziehungen zwischen diesen Klassen darzustellen. Daher bietet die BON die Möglichkeit, Klassen in vollständig komprimierter Form oder nicht mit ihrer vollen Schnittstelle zu zeigen. Ist eine Klasse für die dargestellte Sicht nicht wichtig, kann sie auch ganz ausgelassen werden.

Das Zusammenfassen von Klassen zu Clustern ermöglicht die Darstellung spezieller Aspekte des Systems. Dabei gelten die Beziehungen eines Clusters zu anderen Clustern oder zu einzelnen Klassen für alle in ihm enthaltenen Klassen. Da in der Modulsicht nur Quelltextelemente vorkommen dürfen, muss bei der Erstellung der Sichten darauf geachtet werden, dass die Cluster nur ebensolche Einheiten darstellen und nicht etwa Klassen nach einem Kriterium zusammenfassen, dass nichts mit dem Quelltext des Systems zu tun hat.

Eine häufig verwendetes Architekturmuster zu Verbesserung der Portierbarkeit, dass z. B. in einer Modulsicht dargestellt werden könnte, ist die Schichtenarchitektur. In einer Modulsicht auf diese Schichten würden die zu einer Schicht gehörenden Klassen in Clustern zusammengefasst und die Beziehungen zwischen den Schichten

mit Hilfe der verfügbaren Schnittstellen beschrieben.

4.3.2 Komponentensichten

Komponentensichten werden in der BON mit den dynamischen Diagrammen repräsentiert. In ihnen sind die verlangten Laufzeitelemente als Objekte enthalten. Die einzig mögliche Beziehung ist das Versenden von Nachrichten zwischen den Objekten. Zur Betonung eines Systemaspekts in einer einzelnen Sicht dieser Kategorie können in der BON auch Objekte beliebig zu Clustern zusammengefasst werden. Ähnlich wie bei den Modulsichten, muss bei der Erstellung der Komponentensichten darauf geachtet werden, dass die Cluster Elemente nur nach Laufzeitkriterien zusammenfassen.

Beispiel für eine Komponentensicht, die ebenfalls ein verbreitetes Architekturmuster umsetzt, ist eine Dienstanbieter-Nutzer-Sicht (engl. Client-Server-View). In ihr werden der zur Laufzeit tatsächlich existierende Dienstanbieter und seine Nutzer mit den nötigen Aufrufen gezeigt.

4.3.3 Verteilungssichten

Verteilungssichten können in der BON sowohl mit statischen als auch mit dynamischen Diagrammen repräsentiert werden. Dies liegt daran, dass der Software-Anteil, dessen Beziehungen zu Elementen, die keine Software sind, in einzelnen Verteilungssichten dargestellt werden soll, sowohl ein Laufzeitelement als auch ein Quelltextdokument sein kann. Im Unterschied zu den Modul- und Komponentensichten stellen Cluster von Klassen bzw. Objekten in Verteilungsdiagrammen aber Elemente dar, die keine Software sind.

Eine häufig verwendete Verteilungssicht, die insbesondere für verteilte, eingebettete Systeme wichtig ist, stellt die Aufteilung von Laufzeitelementen auf Hardware-Plattformen dar. Dies lässt sich in einem dynamischen BON-Diagramm darstellen, in dem die auf einer Plattform laufenden Objekte zu Clustern zusammengefasst werden, die dann den Namen dieser Plattform tragen. Nachrichtenbeziehungen zwischen Clustern repräsentieren dann z. B. Busnachrichten zwischen den Plattformen.

Als Beispiel für eine Verteilungssicht, die in einem statischen BON-Diagramm dargestellt wird sei hier die Verteilung von zu entwickelnden Modulen auf Programmierer genannt. Im statischen BON-Diagramm werden die Klassen, die von einem Programmierer implementiert werden sollen, zu einem Cluster mit dessen Namen zusammengefasst. An Beziehungen zwischen Klassen unterschiedlicher Cluster lassen sich notwendige Absprachen zwischen verschiedenen Entwicklern identifizieren.

4.4 Verwandte Arbeiten

Neben der Verhaltensmodellierung wird auch zur Strukturmodellierung eingebetteter Systeme häufig UML eingesetzt, wie z. B. in [FMS00, Mar02]. Zur Verwendung der BON in dieser Domäne ist dem Autor nur das Beispiel der Prozesssteuerung aus dem BON-Buch bekannt [WN95]. Neben diesem Beispiel werden an einigen Stellen industrielle Einsätze z. B. zur Druckersteuerung erwähnt, die jedoch nicht veröffentlicht sind.

Weitere Alternativen zur Strukturmodellierung sind domänenspezifische Sprachen wie die *Architecture Analysis and Description Language* [Fei06], eine Architekturbeschreibungssprache, die auf MetaH [BEJV96] aufbaut. Sie bietet zusätzlich eine Werkzeugumgebung zur Analyse von Verlässlichkeit und Scheduling des Systems.

Die Kombination der BON mit einer formalen Sprache wird auch von Paige und Ostroff präsentiert [PO98], allerdings unter der Verwendung der Sprache Z. ASMs selbst sind schon häufiger in Zusammenhang mit UML verwendet worden. Neben dem ASM Metamodell [SGG+05] gibt es Arbeiten zur formalen Semantik von UML-Aktivitätsdiagrammen [BCR00a] und Zustandsdiagrammen [CHS00, BCR00b].

Die Verwendung von DBC zur Beschreibung von Zeit- und Plattformeigenschaften ist verbreitet. Nienaltowski und Arslan verwenden Reaktionsbeschränkungen in Verträgen [NA03]. Auf die Verwendung dieser Verträge in frühen Entwurfsphasen eines Systems gehen sie aber nicht ein. Li et al. nutzen die Verträge in Kombination mit einem Komponentenmodell für eingebettete Systeme, dass sich an der *common object request broker architecture* (CORBA) orientiert [LLW05]. Urting et al. präsentieren Zeitverträge in verschiedenen Sichten [UBHB01]. Im Unterschied zu diesen Arbeiten werden in dieser Arbeit sowohl Plattform- als auch Umweltanforderungen in Verträgen formuliert und sowohl zur Entwurfszeit als auch wie in Kapitel 5 beschrieben zur Laufzeit des Systems genutzt.

Eine andere Sicht auf die skizzierte Verwendung der Verträge liefert die Erkenntnis, dass mit ihnen ein kleiner Teil der Semantik einer Klasse und ihrer Methoden in ihren Schnittstellen beschrieben wird. Datentypisierungssysteme und die mit ihnen beschriebenen syntaktischen Schnittstellen von Methoden sind eine der erfolgreichsten Praktiken zur Fehlervermeidung beim Softwareentwurf [Lee00]. Ähnlich zu DBC gibt es erste Ansätze, die ebenso das Verhalten von Komponenten als Schnittstelle beschreiben, beispielsweise mit Schnittstellenautomaten [AH01a, AH05, AH01b], die auch zeitliches Verhalten oder Ressourcenverbrauch der Komponente beschreiben können [AHS02, CAHS03].

Die Verwendung unterschiedlicher Sichten ist Stand der Technik [IEE00]. Die Möglichkeit der Darstellung wird in [WN95] nur erwähnt, jedoch nicht so detailliert beschrieben wie in Kapitel 4.3.

5 Echtzeiterweiterung von SCOOP

In diesem Kapitel wird eine Laufzeitumgebung beschrieben, die zur Implementierung eines Systems geeignet ist, dessen funktionale Anforderungen in einer ASM und dessen Struktur, die sich aus seinen qualitativen Anforderungen ergibt, in der BON modelliert ist. Der Übergang von einer ASM zu Quelltext einer Programmiersprache ist dabei als letzter Verfeinerungsschritt bei der Systementwicklung zu sehen, der wegen der syntaktischen Nähe der ASMs zu vielen Programmiersprachen, wegen der sie häufig als Pseudocode mit abstrakten Daten bezeichnet werden, zu einem großen Teil automatisiert erfolgen kann [Sch01b, Sch01c, Sch02].

Auch der Übergang von der BON zur Struktur des Systems soll möglichst automatisiert erfolgen. Dazu muss die BON die für die Anwendungsdomäne wichtigen Abstraktionsmechanismen der Implementierungssprache unterstützen. Durch die Abstraktion wird dann der Entwurf auf hohem Niveau unterstützt; die Einschränkung auf eine Zielsprache für die Implementierung erlaubt den nahtlosen Übergang. Die durchgängige Verwendung desselben Paradigmas durch den kompletten Lebenszyklus eines Softwareproduktes, erleichtert die Kommunikation zwischen den am Produkt beteiligten Interessengruppen, erleichtert die Ausbildung neuer Entwickler, ermöglicht eine ganzheitliche Sicht auf den Lebenszyklus und erhöht die Verfolgbarkeit von Anforderungen. Ein wirklich nahtloser Übergang ermöglicht dabei die Transition in beide Richtungen. Dies erlaubt eine Rückkopplung von einer eventuell geänderten Implementierung in die abstraktere Architekturdarstellung und verhindert dabei das Altern der Entwurfsdokumente. Dadurch wird neben einer tayloristischen Abwärtsstrukturierung auch ein wohl häufiger praktiziertes, iteratives Vorgehen unterstützt. Um die Abstraktionsmechanismen der BON komplett zu nutzen bietet sich Eiffel als Implementierungssprache an. Daher nutzt die hier beschriebene Laufzeitumgebung Eiffel als Sprache und erweitert diese um eine zeitbehaftete SCOOP-Umsetzung.

Im Gegensatz dazu ist ein heute verbreitetes Vorgehen bei eingebetteten Echtzeitsystemen die Implementierung in prozeduralen Sprachen unter Verwendung eines Echtzeitbetriebssystems. Für die Verwendung einer solchen Implementierung mit der BON müssten dann Entsprechungen der Implementierungssprache und der BON-Abstraktionen gefunden werden. Dazu könnten die Klassen statischer BON-Modelle durch Dateien ersetzt werden. Ähnlich wie Klassen existieren sie zur Entwurfszeit und stellen das grundlegendste Softwaremodul dar. Die einzig mögliche statische Beziehung zwischen zwei Dateien ist das Einbinden auf Quelltextebene also die Beziehung Klient-Anbieter; Vererbung ist nicht möglich. Genau wie Klassen bieten

Dateien Befehle und Rückfragen als Funktionen an, deren Semantik durch Vor- und Nachbedingungen verdeutlicht werden können. Eine prozedurale Bedeutung der Objekte der dynamischen BON-Diagramme könnten die nebenläufigen Prozesse eines Betriebssystems sein. Sie stellen Laufzeitobjekte dar, die durch gegenseitiges Aufrufen interagieren können. Allerdings gibt es keine eindeutige Beziehung zwischen den Elementen der statischen und dynamischen BON-Diagramme, wie sie bei Klassen und Objekten herrscht.

Im Gegensatz zum oben beschriebenen, verbreiteten Vorgehen eignet sich die Verwendung von Eiffel wegen fehlender Vorhersagbarkeit nur für Systeme mit weichen Echtzeitanforderungen. Sie unterstützt aber alle Eigenschaften der BON und der in Kapitel 4 beschriebenen Erweiterung, insbesondere Nebenläufigkeit und die Formulierung von Zeitanforderungen in Nachbedingungen. Ein großer Teil des Systemmodells kann automatisch in die Implementierung übersetzt werden. Die Einbindung der ASMs ist ebenfalls automatisierbar; sie selbst können analog zum Vorgehen von Schmid [Sch01b, Sch01c, Sch02] nach Eiffel statt C++ übersetzt werden. DBC wird in Eiffel direkt unterstützt, so dass die Plattformbedingungen aus den BON-Diagrammen einfach in den Quelltext übernommen werden können. Die in den Nachbedingungen formulierten Zeitanforderungen sind jedoch kein Teil des SCOOP-Modells. Hier wird deswegen beschrieben wie sie in SCOOP eingebaut werden können.

Dazu beschreibt Abschnitt 5.1 die Anforderungen an Echtzeitsysteme und Abschnitt 5.2 geht auf die daraus resultierenden Probleme und ihre möglichen Lösungen ein. Schließlich wird in Abschnitt 5.3 die implementierte Lösung gezeigt, bevor Abschnitt 5.4 verwandte Arbeiten präsentiert.

5.1 Anforderungen

Aus Platzgründen kann hier keine vollständige Analyse der Anforderungen von Echtzeitsystemen erfolgen, wie sie z. B. in [BW97] zu finden ist. In Übereinstimmung mit [WPJB06] werden hier die folgenden Anforderungen an ein Echtzeitsystem angenommen:

- eine Möglichkeit, Nebenläufigkeit auszudrücken
- eine Abbildung zwischen den Zeitanforderungen der Anwendung und Attributen des Nebenläufigkeitkonzepts, wie z. B. Fristen und Perioden
- eine Abbildung zwischen der Ausführungsreihenfolge der Anwendung und Attributen des Nebenläufigkeitkonzepts, wie z. B. Prioritäten
- die Unterstützung von Kommunikation und Synchronisation zwischen nebenläufigen Fäden der Anwendung, die die Inversion der Ausführungsreihenfolge

vermeidet, wie z. B. Monitore und Prioritätenvererbung

- ein Fehlerschutzmechanismus wie z. B. Ausnahmebehandlung
- eine Technik, um nebenläufige Fäden asynchron zu benachrichtigen

Des weiteren werden folgende typische Anwendungsanforderungen eingebetteter Systeme angenommen:

- Zugriff auf Gerätetreiber und Behandlung von Interrupts
- Erfassung von verpassten Fristen und die Möglichkeit, darauf zu reagieren
- Erfassung des Überschreitens der schlimmstenfalls möglichen Ausführungszeit (engl. worst case execution time - WCET) und die Möglichkeit, darauf zu reagieren
- Möglichkeit, während der Laufzeit die oben genannten Echtzeitattribute zu ändern

5.2 Probleme

Bei der Implementierung des SCOOP-Modells treten einige zum Teil absichtliche Unterspezifikationen auf, die auch in [WPJB06] untersucht und in den nächsten Unterkapiteln näher beschrieben werden: Wartezeiten, Prioritätsinversion, Verklemmungen, asynchrone Ausnahmebehandlung, Fehlertoleranz und Terminierung.

5.2.1 Wartezeiten

Wie in Abschnitt 2.4 beschrieben werden im SCOOP die Vorbedingungen als Wartebedingungen genutzt. Bei sequentiellem Programmablauf müssen sie als Teil des DBC vom aufrufenden Klienten sichergestellt werden. Wird die Bedingung verletzt, wird eine Ausnahmebehandlung im Klienten ausgelöst. In einem SCOOP-System wartet der Klient bei Verletzung der Vorbedingung eines nebenläufigen Objekts dagegen ab bis sie z. B. durch Interaktion mit einem dritten nebenläufigen Klienten erfüllt ist. Dies kann wie unten beschrieben auch zu einer Verklemmung führen. Die Nachbedingungen erhalten in einem SCOOP-System keine neue Bedeutung.

Zur Wartezeit bis zur Wiederholung des Aufrufs bei Verletzung der Vorbedingung eines Objektes durch den Dienstaufruf eines nebenläufigen Klienten wird in SCOOP keine Aussage gemacht. Werden mehrere Aufrufe wegen der selben Bedingung blockiert, ist nicht vorgeschrieben in welcher Reihenfolge sie neu überprüft werden sollen. Je nach Anwendungsdomäne kann dies und die Wartezeit vom Entwickler der SCOOP-Implementierung gewählt werden.

5.2.2 Prioritätsinversion

Da die Aufrufe nebenläufiger Objekte nicht sofort ausgeführt werden, müssen sie in einer Warteschlange des Dienstanbieters gespeichert werden. Es gibt keine einfache Möglichkeit diese Warteschlange mit Prioritäten zu versehen, ohne bei mehreren wartenden Aufrufen Prioritätsinversionen und Wettlaufsituationen zu ermöglichen.

Weitere Wettlaufsituationen können entstehen, wenn in einem System mit vielen nebenläufigen Objekten für einen Aufruf mehrere Objekte reserviert werden müssen, eines davon jedoch nicht verfügbar ist und dann alle vorher reservierten Objekte wieder abgegeben werden.

5.2.3 Verklemmungen

Wie oben erwähnt kann es bei Ketten von Dienstaufrufen, die alle das selbe separate Argument beinhalten, zu Verklemmungen kommen. Ruft z. B. ein Objekt die Routine eines nebenläufigen mit einem weiteren nebenläufigen als Attribut auf, dann benötigen die zwei ersten exklusiven Zugriff auf das dritte. Kann das erste ohne das Ergebnis des Routineaufrufs nicht weiterarbeiten, kommt es zur Verklemmung. Dieses Problem wird dadurch erschwert, dass solche Ketten von Dienstaufrufen mit dem selben Argument häufig in objektorientierten Programmen vorkommen. Brooke et al. schlagen deshalb vor, die reservierten Objekte bei eigenen Dienstaufrufen mit an das aufgerufene Objekt zu geben [BPJ07].

Verklemmungen können u. U. entdeckt werden, wenn festgestellt werden kann, dass alle auszuführenden Aufrufe blockiert sind. Partielle Verklemmungen können so aber im Allgemeinen nicht entdeckt werden.

5.2.4 Asynchrone Ausnahmebehandlung

Asynchroner Aufruf von Ausnahmebehandlungen kann in SCOOP-Systemen vorkommen, wenn ein dienstaufrufendes Objekt während der Abarbeitung in einem nebenläufigen Objekt weiterläuft, wie es durch das *Warten nach Notwendigkeit* erlaubt ist. Dies führt zu Problemen, wenn eine aufgerufene Routine fehlschlägt und eine Ausnahme erzeugt, z. B. wenn die Vorbedingung verletzt wird. Bei sequentiellem Programmablauf wird diese Ausnahme an den Dienstnutzer bzw. dessen Aufrufer weitergeleitet bis entweder eine Ausnahmebehandlung gefunden wird oder die Weiterleitung im Wurzelobjekt angekommen ist. In diesem Fall wird das Programm von der Laufzeitumgebung gestoppt.

In SCOOP-Systemen sind Dienstanbieter und -nutzer aber nebenläufig. Daher kann der Aufrufer bereits beendet sein, wenn die Ausnahme auftritt. Dann gibt es kein offensichtliches Ziel für die Ausnahme. Andere Objekte im selben Subsystem haben nicht notwendigerweise eine Aufrufbeziehung zu dem betroffenen Dienstanbieter.

Noch weitere Auswirkungen haben asynchrone Ausnahmen wenn der in Kapitel 2.4 nicht vorgestellte Duellmechanismus implementiert ist. Dabei können nebenläufige Objekte bei der Verarbeitung eines Routineaufrufs von einem höherprioren Objekt unterbrochen werden. In SCOOP ist nicht vorgegeben wie asynchronen Ausnahmen behandelt werden sollen.

Ausnahmebehandlung ist für verteilte oder nebenläufige Systeme generell ein schwieriges Problem. Wegen der komplexen Laufzeitstruktur und der fehlenden Trennung von Reservierung und Synchronisierung ist es für SCOOP-Systeme aber besonders komplex. Ein Lösungsvorschlag von Brooke und Paige [BP07] erhält die Semantik von Ausnahmen für sequentielle Systeme.

5.2.5 Fehlertoleranz

Bei Ausfällen des Kommunikationssystems kann es bei Verwendung von SCOOP für ein verteiltes System zu weiteren Problemen kommen. Fehlertoleranz gegen solche Ausfälle oder Redundanzmechanismen sind in SCOOP bisher nicht vorgesehen.

5.2.6 Terminierung

Ein sequentielles Eiffel-Programm ist beendet, wenn die Konstruktorprozedur des Wurzelobjekts abgearbeitet ist. In einem SCOOP-System muss dagegen zusätzlich überprüft werden, ob alle separaten Objekte alle gepufferten Aufrufe abgearbeitet haben. Dazu muss zu einem Zeitpunkt festgestellt werden, ob alle Objekte eine leere Warteschlange haben. Diese Überprüfung ist zwar möglich, jedoch für große Systeme schwierig effizient durchzuführen.

5.3 Implementierung

Die oben genannten Probleme des SCOOP-Modells und Anforderungen von Echtzeitsystemen werden ähnlich zu den in Kapitel 5.4 beschriebenen, verwandten Arbeiten auch in der hier vorgestellten Implementierung nicht gänzlich erfüllt bzw. gelöst. Dennoch ist es mit ihrer Hilfe möglich, weiche Echtzeitsysteme, die auf einem Mikrocontroller laufen, in Eiffel unter Verwendung des Schlüsselwortes **separate** und mit Zeitaussagen in den Nachbedingungen zu programmieren.

Dazu wird die in Abbildung 5.1 dargestellte Schichtenarchitektur von SCOOP verwendet: Ein Präprozessor transformiert ein Eiffel-Programm, das das Schlüsselwort **separate** verwendet und in den Nachbedingungen Zeitaussagen enthält, in eines ohne diese Eigenschaften, das aber stattdessen Klassen einer Echtzeitbibliothek enthält. Diese Bibliothek stellt nebenläufige Prozesse, Semaphoren und eine Stoppuhr zur Verfügung. Diese Arbeit ist die erste, die diese Bibliothek aufbauend auf einem Echtzeitsystem implementiert. Während in anderen Arbeiten Ausführungsfäden

nach dem Posix-Standard [Gal95] oder nebenläufige Ausführungseinheiten der .NET-Umgebung verwendet werden, baut die Bibliothek hier auf dem Betriebssystem FreeRTOS[1] auf.

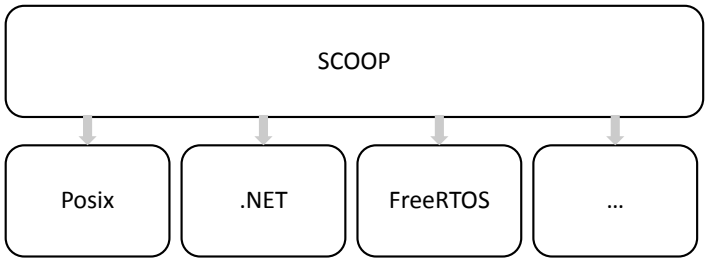

Abbildung 5.1: Schichtenarchitektur von SCOOP nach [NAM03].

Das mit dem Präprozessor und der Bibliothek entstandene Eiffel-Programm kann mit einem Kompilierer in Ansi-C übersetzt werden, benötigt zur Übersetzung in ausführbaren Maschinencode aber noch eine Eiffel-Laufzeitumgebung für die Zielplattform. Im Rahmen dieser Arbeit ist zusätzlich zu einem SCOOP-Präprozessor und einer FreeRTOS-Bibliothek eine solche Laufzeitumgebung für den Mikrocontroller Atmel AT90CAN128 entstanden. Die folgenden Unterkapitel beschreiben die einzelnen Teile der echtzeitfähigen SCOOP-Umgebung – Bibliothek, Eiffel-Laufzeitumgebung und Präprozessor – genauer.

5.3.1 Bibliothek

In der Bibliothek werden die Funktionen des Betriebssystems FreeRTOS zur Verfügung gestellt. In ihr sind in erster Linie die nebenläufigen Ausführungsfäden, Prozesse genannt, verfügbar. Sie können dynamisch zur Laufzeit des Systems generiert werden und werden dann vom Scheduler in den Ablaufplan aufgenommen. Für das hier beschriebene Vorgehen ist die Fähigkeit zur dynamischen Erzeugung nötig, da vor Laufzeit des Systems nicht bekannt ist, wie viele Prozesse benötigt werden. Statische Betriebssystem etwa nach dem OSEK-Standard [Lem01] können daher nicht verwendet werden.

Bei der Erzeugung eines neuen Prozesses müssen die Echtzeitparameter Frist und Priorität angegeben werden. Die Idee bei der automatischen Generierung des Quelltextes mit dem Präprozessor ist das Einlesen dieser Parameter aus einer Konfigurationsdatei. Damit bleibt die eigentliche Implementierung frei von diesen möglicherweise plattformabhängigen Details und ist auf anderen Plattformen ohne Änderung des Quelltextes wiederverwendbar. Die Verknüpfung zwischen einer

[1] `http://www.freertos.org`

separate-Klasse und ihren Attributen in der Konfigurationsdatei kann über die Zeitanforderung in der Nachbedingung geschehen.

Für die SCOOP-Umsetzung werden zusätzlich Verfahren benötigt, die verhindern, dass nebenläufige Prozesse gleichzeitig auf Daten zugreifen. Dazu werden die Mutexe von FreeRTOS mit Funktionen zum Belegen und Freigeben von Ressourcen in die Bibliothek mit aufgenommen.

Zusätzlich zu den Prozessen liefert FreeRTOS auch die Möglichkeit, auf Interrupts der Plattform zu reagieren. Mit Hilfe dieser Eigenschaft können mit der Bibliothek Eiffel-Klassen geschrieben werden, die auf einen bestimmten Interrupt reagieren.

Ebenso liefert FreeRTOS eine Abstraktion des an die Plattform angeschlossenen CAN-Buses. Mit deren Hilfe können Eiffel-Klassen CAN-Nachrichten empfangen und versenden.

Zur Überprüfung der in den Nachbedingungen angegebenen Zeitbedingungen wird eine Stoppuhr benötigt. Diese ist ebenfalls in der Bibliothek verfügbar, liefert aber nur eine untere Schranke für die tatsächlich verstrichene Zeit. Daher lässt sich diese Art der Zeitüberwachung nur für weiche Echtzeitsysteme verwenden.

5.3.2 Eiffel-Laufzeitumgebung

Um Eiffel-Programme, ob mit oder ohne SCOOP, auf dem Mikrocontroller Atmel ATMega 128 CAN ausführen zu können, muss dort eine Eiffel-Laufzeitumgebung verfügbar sein. Im Rahmen dieser Arbeit wird dazu die Laufzeitumgebung des GNU Eiffel-Kompilierers [ZCC97] so angepasst, dass sie mit dem Kompilierer für den Mikrocontroller in Maschinensprache übersetzt werden kann. Die mit dem Eiffel-Kompilierer ausgelieferte Laufzeitumgebung ist zwar in ANSI-C geschrieben und soll daher mit den meisten C-Kompilierern ohne Änderungen übersetzbar sein, im Fall des Mikrocontrollers fehlen jedoch einige verwendete Bibliotheksfunktionen. Die angepasste Laufzeitumgebung hat keine automatische Speicherbereinigung, wie sie beispielsweise in [LH83, Hen98] für Echtzeitsysteme vorgeschlagen wird.

5.3.3 Präprozessor

Der Präprozessor erledigt die eigentliche Übersetzung eines SCOOP-Programm in ein Eiffel-Programm, das die oben beschriebene Bibliothek verwendet. Der hier beschriebene Präprozessor ähnelt dem *SCOOP-to-Eiffel Code Generator* (SECG) von Fuks et al. [FOP04], erweitert diesen aber um die Überprüfung der zeitbehafteten Nachbedingungen und setzt SCOOP erstmalig auf einem Echtzeitbetriebssystem um.

Grundlegende Idee des Präprozessors ist das Hinzufügen von Puffern und Mutexen zu separaten Klassen, um offene Anfragen von Klienten zu speichern. Zusätzlich werden ähnliche Änderungen und Hinzufügen von Mutexen für separate Argumente

durchgeführt, um exklusiven Zugriff auf sie zu ermöglichen. Jede separate Klasse erbt bei der Übersetzung von der Prozessklasse der FreeRTOS-Bibliothek und erhält einen Puffer zur Speicherung von Klientenanfragen. Die Prozesse werden vom Scheduler des Betriebssystems aufgerufen und jeder von ihnen entfernt eventuell vorhandene Anfragen aus seinem Puffer und führt sie aus.

Dazu erhält der Präprozessor eine Liste aller Klassen, d. h. Dateien der Anwendung und eine Markierung der Wurzelklasse. Diese Angaben sind bei Eiffel-Projekten meist in einer speziellen Projektdatei enthalten. Mit diesen Informationen werden die einzelnen Dateien durchgearbeitet und der folgende Quelltext generiert:

- Im Gegensatz zum SECG wird die Wurzelklasse nicht gesondert behandelt. Der Präprozessor geht davon aus, dass das eingebettete System seine periodischen Aufgaben bis zum Ausschalten durch den Nutzer erfüllen muss und daher keine Detektion der Terminierung nötig ist.

- Alle separaten Klassen und die Wurzelklasse erben von der Prozessklasse der FreeRTOS-Bibliothek und sind somit im späteren System nebenläufige Ausführungsfäden.

- Alle separaten Klassen erhalten einen FIFO-Puffer zur Speicherung von Klientenanfragen und einen Mutex zur Sicherung des exklusiven Zugriffs auf diesen Puffer sowie einen Zähler für die aktuell offene Anzahl an Klientenanfragen, der mit einem weiteren Mutex abgesichert ist. Zwei weitere Attribute speichern den aktuell abzuarbeitenden Aufruf sowie seine Argumente als Zeichenketten. Nebenläufige Klientenanfragen an die separate Klasse werden im Puffer gespeichert.

- Jede separate Klasse erhält eine Routine, die nach vorheriger Belegung des Mutex überprüft, ob der Zähler für die Klientenanfragen auf null steht.

- Eine weitere Routine fügt eine Klientenanfrage in den Puffer ein. Dazu werden die Mutexe belegt, der Zähler um eins erhöht und die Klientenanfrage als Zeichenkette codiert in den Puffer eingetragen.

- Die gegenteilige Aufgabe, das Entfernen und Delegieren einer Anfrage aus dem Puffer wird von einer weiteren Routine erledigt. Sie wird in der Hauptroutine der Klasse, die regelmäßig vom Scheduler des Betriebssystems aufgerufen wird, ausgeführt.

Die Deklaration separater Entitäten wie x: **separate** *XTYPE* wird ebenso mit einem Mutex versehen wie separate Argumente und das Schlüsselwort **separate** wird entfernt. Die Mutexe ermöglichen den exklusiven Zugriff auf die separaten Argumente bzw. Entitäten.

Die Konstruktoren der separaten Klassen müssen um die Initialisierung der Puffer, Mutexe und Zähler erweitert werden. Die Pufferung von Klientenanfragen steht erst nach Beendigung des Konstruktors zur Verfügung.

Schließlich müssen noch alle Routinenaufrufe an separate Klassen umgeschrieben werden. Anstelle des alten Routinenaufrufs tritt der Aufruf der Routine, die eine Anfrage aus dem Puffer entfernt und an die entsprechende, dienstanbietende Routine weiterleitet, mit dem als Zeichenkette codierten eigentlichen Routinenaufrufs und seiner Argumente. Zu separaten Argumenten eines Routinenaufrufs müssen dabei auch deren Mutexe mit an die dienstanbietende Klasse übergeben werden.

Damit die gepufferten Klientenanfragen auch abgearbeitet werden, wird jede separate Klasse noch um eine Routine execute erweitert. Dies ist die Routine, die regelmäßig vom Scheduler aufgerufen wird. Sie entfernt eine Klientenanfrage aus dem Puffer, dekodiert diese und führt sie aus.

Neben den oben genannten Änderungen, die so ähnlich auch im SECG vorkommen, übernimmt der Präprozessor hier zusätzlich die Aufgabe, die Zeitanforderungen in den Nachbedingungen umzuschreiben. Beim Übergang von Eiffel nach SCOOP ist es nötig, die Bedeutung der Vorbedingungen zu ändern. Sie werden zu Wartebedingungen und stellen somit einen Synchronisierungsmechanismus zur Verfügung. Ähnlich zu [WPJB06] wird hier die Umdeutung der Nachbedingungen vorgeschlagen. Sie werden zur Detektion von Überschreitungen von Fristen genutzt.

Dazu werden die in den Nachbedingungen enthaltenen Zeitanforderungen mit Hilfe der Stoppuhr der FreeRTOS-Bibliothek überprüft. Diese läuft beim Starten einer Routine los und kann beim Betreten der Nachbedingung gestoppt werden, um die Einhaltung der Zeitanforderung zu überprüfen. Verletzungen werden wie für alle Nachbedingungen mit einer Ausnahmebehandlung abgefangen.

5.4 Verwandte Arbeiten

Automotive open system architecture (AUTOSAR) [HBS+06] ist der kommende Industriestandard für den Enwurf von Fahrzeugsoftware, der sich in einigen Bereichen stark an der UML orientiert, ohne jedoch die Implementierung in einer objektorientierten Sprache anzustreben. Er beschreibt eine Laufzeitumgebung, die einen virtuellen Bus zur Verfügung stellt, um Komponenten zur Entwurfszeit in einem verteilten System einfach verschieben zu können.

Es gibt eine unvollständige SCOOP-Implementierung, die wie der hier gezeigte Ansatz den SmartEiffel-Kompilierer [ZCC97] nutzt [CW02] jedoch direkt in dessen Laufzeitumgebung arbeitet. Eine Präprozessorimplementierung von Fuks et al. [FOP04], an der sich die hier vorgestellte orientiert, arbeitet auf einem anderen Kompilierer mit der dort verfügbaren Prozessbibliothek. Die neueste Implementierung von Nienaltowski und Arslan [NA03] nutzt ebenfalls einen Präprozessor, um

ein SCOOP-Programm unter der Verwendung von Prozessklassen umzuschreiben und läuft auf der .NET-Plattform. Keine dieser drei SCOOP-Implementierungen ist für eingebettete Systeme geeignet: Die verwendeten Eiffel-Laufzeitumgebungen sind nicht für Mikrocontroller verfügbar, die genutzten Prozesse sind nicht echtzeitfähig und es ist kein Zugriff auf Interrupts möglich.

Weitere Ansätze, die Nebenläufigkeit in objektorientierte Sprachen integrieren, werden von [Agh90, Phi00, WKH92] zusammengefasst. Darunter sind auch einige Alternativen zu SCOOP für nebenläufiges Eiffel [Car93, KB93, Mey93]. Ada 95 [Bur98] erweitert eine nebenläufige Sprache zu einer objektorientierten.

Wellings et al. unterscheiden zwei Ansätze, um Nebenläufigkeit und Objektorientierung zu vereinigen [WPJB06]: Auf der einen Seite gibt es Ansätze mit Koexistenz von Objekten und Ausführungsfäden mit unterschiedlich starker Integration der beiden Konzepte wie z. B. Java [Lea00, OW04]. Auf der anderen Seite Ansätze mit voller Integration von Objekten und Ausführungsfäden, bei denen das objektorientierte Modell so abgeändert wird, dass es Nebenläufigkeit beinhaltet wie POOL-T [Ame87]. Die Kommunikation und Synchronisierung nebenläufiger Aktivitäten werden bei letzteren Ansätzen so konstruiert, dass sie die sogenannte Vererbungsanomalie vermeiden [MY93].

Sprachen, die echtzeitfähiges, objektorientiertes Programmieren ermöglichen verwenden meist den Koexistenzansatz wie beispielsweise Ada 95 oder Java [BBF+06]. Dies liegt vermutlich daran, dass bei diesem Ansatz die Identifikation von Ausführungsfäden und die Zuordnung von Echtzeitattributen zu diesen Fäden einfacher ist [WPJB06]. Eine nicht veröffentlichte Arbeit zur echtzeitfähigen Erweiterung von SCOOP namens RECOOP ist Grundlage des hier vorgestellten Konzepts [WPJB06], liegt aber nicht implementiert vor.

6 Fallstudie

Um den Entwurf eingebetteter Software mit der Kombination aus ASMs und BON sowie die echtzeitfähige SCOOP-Laufzeitumgebung zu evaluieren, wird hier eine Fallstudie aus dem Automobilbereich präsentiert. Ein Abstandsregeltempomat (eng. adaptive cruise control - ACC) ist ein Fahrerassistenzsystem zur Längsregelung eines Autos. Er hält die Geschwindigkeit des Fahrzeugs ähnlich wie ein Tempomat konstant, kann aber zusätzlich Hindernisse auf der Fahrbahn, wie etwa langsamere Autos, erkennen und einen sicheren Abstand zu ihnen wahren. Verschwindet das Hindernis, wird das Fahrzeug wieder auf die alte Geschwindigkeit beschleunigt.

Dieses Beispiel enthält viele typische Aufgaben eingebetteter Systeme wie die Interaktion mit der Umwelt über Sensoren und Aktoren oder die Regelung einer Ausgangsgröße. Das System enthält Zeitanforderungen und ist sicherheitskritisch, da es bei Fehlfunktionen zu Unfällen führen kann. Das Auslesen der Sensorwerte sowie die Ansteuerung der Aktoren sollen für ein Einparksystem wiederverwendbar sein. Das implementierte ACC-System läuft auf einem Atmel AT90CAN128 Mikrocontroller, der über eine CAN-Schnittstelle an ein Netzwerk von Mikrocontrollern auf einem Modellauto angeschlossen ist.

Die Beschreibung der Fallstudie ist wie die vorangehenden Kapitel dieser Arbeit strukturiert: Zunächst stellt Abschnitt 6.1 die Modellierung und Verifikation der funktionalen Anforderungen mit ASMs vor. Zusätzlich werden einige weitere Fallstudien zum Vergleich mit den in Kapitel 3.2.4 vorgestellten anderen ASM Model-Checkern beschrieben. Die sich aus den qualtiativen Anforderungen ergebenden BON-Strukturmodelle sowie die Einbindung der ASMs und die Darstellung der physikalischen Anforderungen sind Gegenstand von Kapitel 6.2. Dort wird ebenso auf die Darstellung unterschiedlicher Sichten eingegangen. Abschließend geht Kapitel 6.3 durch den Vergleich dreier unterschiedlicher Implementierungen der Fallstudie auf die echtzeitfähige SCOOP-Umgebung ein, bevor Abschnitt 6.4 eine Gesamtbewertung gibt.

6.1 Verhaltensmodellierung und -verifikation

Die funktionalen Anforderungen werden zunächst in einem Grundmodell des Systems erfasst. Hier ist insbesondere der Teil des Systems von Interesse, der das ACC steuert, der also aus den Eingangswerten wie Benutzerinteraktion und Sensorwerten die Ausgaben Benutzeranzeige und Aktorenansteuerung erzeugt. Dazu können die

Anforderungen zunächst natürlichsprachlich erfasst und dann in eine ASM übersetzt werden. Aufgrund ihrer Pseudocode-ähnlichen Syntax sind ASMs aber für viele am Entwurfsprozess Beteiligte sofort verständlich und können insbesondere von technisch Ausgebildeten sofort zur Modellierung der Anforderungen genutzt werden.

Das ASM-Grundmodell kann in Simulationen ausgetestet und validiert werden, um das Verständnis des Systems zu erhöhen. Erkenntnisse aus diesen Simulationen und Entwurfsentscheidungen führen zu verfeinerten Modellen. Abschnitt 6.1.1 stellt das simulierbare ASM-Grundmodell sowie einen Verfeinerungsschritt vor.

Auf allen Verfeinerungsstufen im Entwicklungsprozess können die ASM-Modelle auf Grund ihrer formalen Semantik verifiziert werden. Das Model-Checking mit Hilfe des in Kapitel 3.2 vorgestellten Werkzeugs wird in Abschnitt 6.1.2 vorgestellt und mit anderen Werkzeugen verglichen. Abschließend wird die Verwendung von ASMs zum funktionalen Entwurf in Abschnitt 6.1.3 bewertet.

6.1.1 Modellierung der funktionalen Anforderungen mit ASMs

Hier wird zunächst das Grundmodell des ACC-Systems vorgestellt, das gleichzeitig die funktionalen Anforderungen widerspiegelt. Anschließend wird ein Verfeinerungsschritt beschrieben, der aufgrund einer Entwurfsentscheidung vollzogen wird.

Grundmodell

In Listing 6.1 ist ein Ausschnitt des ASM-Grundmodells des ACC-Systems zu sehen. Dargestellt ist die Hauptregel der ASM ohne Funktionsdeklarationen und Hilfsregeln in der Sprache des verwendeten Simulators CoreASM.

Listing 6.1: Hauptregel des ACC-Grundmodells.

```
rule acc_main_rule =
 seq
   readui
 seq
   readsensors
 par
   if (currentui = Break) then
     currentmode := Ready
   else if (currentmode = SpeedControl) then
   par
     if (currentdistance/currentspeed < safedistance) then
       currentmode := DistControl
     else
       choose setPWM
   endpar
   else if (currentmode = DistControl) then
```

```
    par
      if (currentdistance/currentspeed >= safedistance) then
        currentmode := SpeedControl
      else
        choose setPWM
    endpar
    else if (currentmode = Ready and currentui = SwitchOn) then
    par
      if not(currentspeed < MaxSpeed) and (MinSpeed <
          currentspeed) then
        print("you can not switch ACC on at this speed")
      else
      par
        CruiseControlSpeed := currentspeed
        if (currentdistance/currentspeed >= safedistance) then
          currentmode := SpeedControl
        else
          currentmode := DistControl
      endpar
  endpar
```

Die Funktionen *currentui*, *currentspeed* und *currentdistance* sind beobachtete Lokationen, die von der ASM nur gelesen werden können. Sie stellen die Eingangswerte der ASM dar, nämlich die Nutzereingaben Aktivieren und Bremsen sowie die Sensorwerte zur Messung der eigenen Geschwindigkeit sowie der Distanz zum nächsten erkannten Hindernis.

Die statischen Funktionen *MinSpeed* und *MaxSpeed* definieren das Geschwindigkeitsintervall, in dem das ACC-System aktiviert werden darf. Die Obergrenze wird meist aus Sicherheitsüberlegungen deutlich unter die Höchstgeschwindigkeit des Fahrzeugs gelegt. Im Fall des Modellautos entspricht sie aber der erreichbaren Höchstgeschwindigkeit. Liegt die Untergrenze bei null, kann das System beispielsweise in Stausituationen automatisch bis zum Stillstand abbremsen und auch wieder anfahren. Beim Modellauto ist diese Funktionalität nicht implementiert, da das ACC-System nur das Gaspedal, nicht jedoch die Bremse ansteuert und somit kein Abbremsen bis zum Stillstand garantieren kann. Deshalb liegt die Untergrenze hier bei einer niedrigen Geschwindigkeit des Fahrzeugs.

Die kontrollierte Funktion *currentmode*, die auch als Zustandsvariable bezeichnet wird, beschreibt das Verhalten des ACC-Systems in den drei verschiedenen Zuständen Bereit, Geschwindigkeits- und Distanzregelung. Im Initialzustand Bereit greift das System nicht in die Steuerung des Fahrzeugs ein. Startet der Fahrer das System wird je nach Fahrsituation in einen der anderen Zustände gewechselt. Durch Bremsen kann der Fahrer das System jederzeit deaktivieren und in den Startzustand zurückkehren. Je nachdem ob ein Hindernis auf der Fahrbahn erkannt wird oder nicht, wechselt

das aktivierte System zwischen Geschwindigkeits- und Distanzregelung.

Die kontrollierte Funktion *safedistance* ist der als sicher erachtete Zeitabstand zu einem vorausfahrenden Fahrzeug. Ist der aktuelle Zeitabstand, der sich mit den Sensorwerten *currentdistance*/*currentspeed* berechnen lässt, kleiner als dieser Wert wird in den Zustand Distanzregelung gewechselt, ist er größer wird die Geschwindigkeit des Autos wie bei einem Tempomaten konstant gehalten. Der Wert dieser sicheren Distanz richtet sich nach den physikalischen Randbedingungen des Modellautos wie Sensorreichweite in Bezug auf erreichbare Geschwindigkeitswerte und Bremsbeschleunigung.

In der kontrollierten Lokation *CruiseControlSpeed* wird die Führungsgröße für die Geschwindigkeitsregelung abgelegt. Im gezeigten Grundmodell entspricht sie der Fahrzeuggeschwindigkeit bei Aktivierung des Systems. In einer späteren Verfeinerung des Systems wird dieser Wert bei aktivem ACC für den Nutzer änderbar.

Die Ausgabelokation *chooseP WM* repräsentiert den Stellwert, der an die Aktoren des Autos gesendet wird, um die Geschwindigkeit des Fahrzeugs zu beeinflussen. Die nichtdeterministische Wahl dieses Wertes in den Zeilen 14 und 21 wird in späteren Verfeinerungen des Grundmodells zu zwei Reglern, die die Geschwindigkeit bzw. den Abstand zum vorausfahrenden Fahrzeug konstant halten. Auf dem Abstraktionsniveau des Grundmodells ist noch keine Entscheidung gefallen, ob dazu nur das Gaspedal oder auch die Bremse angesteuert werden müssen.

readsensors und *readui* sind Hilfsregeln, die das Einlesen der Sensorwerte bzw. Benutzereingaben modellieren und hier aus Platzgründen nicht gezeigt werden. Sie modellieren sehr einfach die Umgebung, die benötigt wird, um das ACC-System simulieren zu können. Dazu beschreiben sie die beobachteten Lokationen des Systems und stellen ihm so seine Eingangswerte zur Verfügung. Für die Simulation werden diese Werte durch den Entwickler von Hand eingegeben. Alternativ können sie auch aus einer Datei eingelesen oder zufällig gewählt werden. Mit zufällig gewählten Werten kann das System auch im Model-Checker verifiziert werden. Natürlich sind auch komplexere Modelle der Umwelt möglich, aber der Erstellungsaufwand verspricht im Fall des ACC-Systems keinen weiteren Erkenntnisgewinn. Die genannten Umgebungsmodelle sind einfach zu erstellen und erlauben die Simulation zur Validierung der funktionalen Anforderungen und die Verifikation der ASM. Quantitativ genauere Modelle wie sie für den Reglerentwurf nötig sind, sind mit ASMs nicht gut zu erstellen und werden gewöhnlich mit spezialisierten Werkzeugen entworfen.

An Hand der beiden oben genannten Hilfsregeln ist eine weitere Besonderheit der ASMs zu erkennen: Da Zustandsaktualisierungen nicht direkt sondern gesammelt beim Zustandsübergang auf einmal ausgeführt werden, müssen die Eingabelokationen in einem eigenen Zustandsübergang beschrieben werden, weil ihre Werte sonst im aktuellen Zustand nicht verfügbar sind. Dazu bietet CoreASM das Schlüsselwort **seq**, das eine sequentielle Ausführung der Befehle erzwingt.

Verfeinerung

Das oben präsentierte Grundmodell enthält noch nicht alle funktionalen Anforderungen, sondern nur die, die der Entwickler auf diesem, frei wählbaren Abstraktionsgrad darstellen kann und will. Es dient der Kommunikation unter den Entwicklern und zur Auffindung von Ungenauigkeiten oder Inkonsistenzen. Weitere Anforderungen, die noch nicht enthalten sind, durch Entwurfsentscheidungen entstehen oder neu dazu kommen, werden in Verfeinerungen dieses Grundmodells eingearbeitet. Im Folgenden werden einige der durchgeführten Verfeinerungen beschrieben. Es gibt noch keine Möglichkeit, die Verfeinerungsbeziehung zweier ASMs automatisiert und formal zu beweisen. Der Beweis von Hand ist teilweise möglich jedoch häufig aufwändig. Im Rahmen der Fallstudie wurde deshalb auf einen Beweis verzichtet und die Verfeinerungsbeziehung mit Hilfe von Simulationen validiert.

Um die Stabilität des Systems zu erhöhen, soll eine Plausibilisierung der Messwerte erfolgen. Dazu werden die Werte auf die physikalisch gegebenen Grenzen der Sensoren hin überprüft. Bei fehlerhaften Werten wird ein Zähler erhöht, der das System bei Erreichen eines Grenzwertes ausschaltet und in einen Fehlerzustand versetzt, der nur durch Neustart verlassen werden kann. Diese Verfeinerung beinhaltet das vollständige Verhalten des Grundmodells und fügt lediglich einen neuen Fehlerzustand in das System ein und erweitert die Regel um Übergänge in diesen zu ermöglichen.

Eine Verfeinerung, die in tatsächlichen ACC-Systemen enthalten ist, ist die Möglichkeit der Einflussnahme auf die Führungsgrößen der Regler durch den Benutzer. Der Fahrer kann die Führungsgrößen für die Geschwindigkeit *CruiseControlSpeed* und für die Distanz *safedistance* innerhalb gewisser Grenzen verändern. Dazu werden in Listing 6.1 Befehle zum Erhöhen bzw. Erniedrigen dieser Werte innerhalb der erlaubten Grenzen bei entsprechenden Benutzereingaben eingefügt.

Durch die Modellierung von Entwurfsentscheidungen entfernt sich die ASM von der reinen Anforderungsmodellieung immer weiter zur Implementierung. Im Rahmen des ACC-Systems ist dies beispielsweise die Entscheidung für die Verwendung eines PI-Reglers zur Geschwindigkeitsregelung. Dazu wird Zeile 14 des Grundmodells durch Listing 6.2 ersetzt.

Listing 6.2: Verfeinerter Geschwindigkeitsregler.

```
speederror := CruiseControlSpeed - currentspeed
choose a in PTHROTTLE with (a < MaxThrottle and MinThrottle < a)
    do
  pTerm := a
choose b in ITHROTTLE do
  iTerm := b
setPWM := pTerm + iTerm
```

Der berechnete Fehler wird in der ASM nicht weiterverwendet und ist daher eigentlich unnötig. Er dient hier lediglich dem Hinweis auf die zu verwendenden Lokationen beim weiteren Reglerentwurf. Beim P-Anteil ist beispielhaft dargestellt wie eine physikalische Beschränkung der Stelleinrichtung in die Bedingung mit aufgenommen werden kann. Andere Anforderungen wie etwa die gewünschte Geschwindigkeit des Ausgleichs bei Abweichung können wie in Kapitel 3.1.2 beschrieben mit in die Bedingungen aufgenommen werden, sind hier jedoch ausgelassen, da sie nicht mitsimuliert werden können. Die Möglichkeit, sie zumindest auf dem Papier als Anforderungen an den Regler in das Modell aufzunehmen, ist der Grund für die zufällige Wahl des P- bzw. I-Anteils des Reglers.

Die **choose**-Regel in CoreASM kann nicht über dem Universum *NUMBERS* ausgeführt werden, da es keine Aufzählung ist. Für die Simulation werden hier behelfsweise die beiden Universen *PTHROTTLE* und *ITHROTTLE* definiert, die einige mögliche Stellwerte enthalten. Da die Universen jedoch als Aufzählungen definiert werden müssen und die möglichen Stellwerte alle vom Rechner darstellbaren Zahlen eines Intervalls enthalten müssten, können sie nur bei größtem Aufwand alle möglichen Werte enthalten.

Eine rein deterministische Variante in der Art $pTerm := pGain * speederror$ hat zusätzlich den oben beschriebenen Nachteil, komplett in **seq**-Befehle eingefasst werden zu müssen, da die berechneten Werte in einem Zustand noch nicht für weitere Berechnungen zur Verfügung stehen. Aus dem gleichen Grund stellt CoreASM das verwendete zweizeilige **choose** zur Verfügung. Für die weitere Verfeinerung des Reglers eignet sich die ASM-Modellierung kaum. Spezialisierte Werkzeuge, die das dynamische Verhalten von Regler und Strecke modellieren und simulieren können, sind dazu die bessere Wahl.

6.1.2 Verifikation

Neben der Aufdeckung von Ungenauigkeiten in den Anforderungen durch deren Formalisierung und die Simulation der ASMs, können durch die formale Verifikation weitere, schwieriger aufzudeckende Fehler gefunden werden. Dazu wird hier zunächst das Model-Checking des oben beschriebenen ASM-Modells erläutert. Zum Vergleich mit den anderen in Kapitel 3.2.4 genannten Model-Checkern kann die Fallstudie jedoch nicht verwendet werden. Die Werkzeuge sind größtenteils nicht verfügbar bzw. wegen des Betriebssystems auf dem zum Model-Checking verwendeten Rechner nicht lauffähig. Daher vergleicht der darauf folgende Abschnitt den im Rahmen dieser Arbeit entwickelten Model-Checker mit Fallstudien aus der Literatur. Auch wenn die dort veröffentlichten Werte wegen unterschiedlicher Hardware nicht direkt mit den gemessenen vergleichbar sind, sind dennoch qualitative Aussagen zur Leistungsfähigkeit des Ansatzes möglich.

Formale Verifikation des ACC-Verhaltensmodells

Wie oben bereits erwähnt müssen die Hilfsregeln *readsensors* und *readui* für das Model-Checking umgeschrieben werden. Da die Eingaben des Fahrers bzw. die gemessenen Sensorwerte während des Aufbaus des Zustandsraums nicht von Hand eingegeben werden können, werden sie durch **choose**-Regeln ersetzt. Da die Fahrereingaben lediglich drei verschiedene Werte annehmen kann ist dies für sie in CoreASM direkt darstellbar. Wie bei der Verfeinerung des Reglers können die Sensorwerte aber wieder nicht über dem Universum *NUMBERS* ausgewählt werden. Wie bei der Reglerverfeinerung werden daher Hilfsuniversen mit typischen Sensorwerten definiert über denen die nichtdeterministische Auswahl dann stattfindet.

Dies stellt eine Unterapproximation des tatsächlichen Verhaltens der Umgebung des ACC-Systems dar. Wenn der Entwickler aber sicherstellt, dass Sensorwerte aus allen vom System unterschiedenen Intervallen vorkommen, wird dennoch das komplette Verhalten des Systems simuliert. Mit dem so abgeänderten System wurden im Rahmen der Fallstudie die folgenden Eigenschaften des Systems verifiziert:

Ein Bremsen durch den Fahrer führt immer zur Deaktivierung des Systems. Dazu wird folgende Formel verifiziert:

$$\text{AG } (\text{currentui}(.) = \text{break} \Rightarrow \text{AX currentmode}(.) = \text{Ready})$$

Wie an den Zeilen 7 bis 8 in Listing 6.1 zu sehen ist, ist diese Formel für das System erfüllt. Das Vorziehen dieses Übergangs in den inaktiven Zustand auf die ersten Zeilen außerhalb der Beschreibung für die Distanz- und Geschwindigkeitsregelung, das die Validität der Formel auch durch scharfes Hinsehen verdeutlicht, liegt an einem Fehler in einer früheren Version des Grundmodells. Dort war der Übergang in die Inaktivität noch jeweils in der Distanz- und Geschwindigkeitsregelung implementiert aber wegen eines Kopierfehlers nicht in allen Fällen gewährleistet. Ebenso kann die Gegenaussage verifiziert werden, dass das System ohne Bremsen durch den Fahrer nach Aktivierung nicht mehr in den inaktiven Zustand wechselt.

Eine Aussage, die wie in Kapitel 4.1 beschrieben auch in der Vorbedingung der entsprechenden Klasse im BON-Diagramm festgehalten ist, betrifft die Aktivierung des Systems außerhalb des erlaubten Intervalls. Dazu wird geprüft, ob das System bei Aktivierung durch den Fahrer außerhalb des Geschwindigkeitsintervalls $MinSpeed \ldots MaxSpeed$ den Zustand *Ready* verlässt. Dies ist für das Grundmodell erfüllt. Wenn sich der Entwickler durch die Verifikation der feinsten ASM sicher ist, dass diese Eigenschaft auch auf die Implementierung zutrifft, kann die Überprüfung der entsprechenden Vorbedingung zur Laufzeit deaktiviert werden. Diese Verwendung der Vorbedingung entspricht genau genommen nicht der Idee des DBC. Durch die Vorbedingung wird dem Regler zugesichert, dass die Geschwindigkeitswerte im erlaubten Intervall liegen, so dass er sie nicht mehr selbst überprüfen muss. Im

Rahmen der Fallstudie empfanden die Entwickler die Dokumentation der Verifikation auf diese Art jedoch als hilfreich.

Die beobachteten Laufzeiten und Speicherbedarf liegen für valide Formeln in einem ähnlichen Bereich wie für die unten dargestellten Vergleiche zu anderen Model-Checkern. Der komplette Zustandsraumaufbau liegt für das ASM-Grundmodell abhängig von den gewählten, möglichen Sensorwerten in der Größenordnung mehrerer Stunden bis hin zu einem Tag. Dabei wird überraschend wenig Speicherplatz verbraucht. Je nach überprüfter Formel liegt er bei einigen Kilobyte pro Zustand. Gründe für diese Leistungswerte des Model-Checkers werden in Abschnitt 6.1.3 diskutiert. Genauere Zahlen zum Vergleich mit anderen Werkzeugen finden sich im nächsten Abschnitt.

Vergleich mit anderen Model-Checkern

Wie oben erwähnt müssen zum Vergleich mit anderen ASM-Model-Checkern Fallstudien aus der Literatur verwendet werden. Dazu werden in den beiden folgenden Abschnitten ein Protokoll, das Speicherkohärenz in einem verteilten System sicherstellen soll und ein Algorithmus zur Terminierungsdetektion verifiziert.

Speicherkohärenz Der *Stanford Flash Multiprozessor* ist eine Architektur für verteilte Prozessoren mit geteiltem Speicher [KOH+94]. Dieser wird von einem Kohärenzprotokoll koordiniert, das sowohl von Winter [Win01], als auch von Ma [Ma07] benutzt wurde, um die Fähigkeiten ihrer ASM-Model-Checker zu demonstrieren. Der Speicher ist über alle Knoten des Prozessornetzwerks verteilt und in Linien aufgeteilt, die einzelnen Knoten zugeordnet sind. Jeder Prozessor kann lokale Kopien jeder Datei haben und das Kohärenzprotokoll stellt sicher, dass alle Dateien aktuell sind.

Wir nutzen die CoreASM-Spezifikation dieses Protokolls von Ma [Ma07], um die folgenden zwei Eigenschaften für ein Netzwerk mit zwei Prozessoren und einer Speicherlinie zu verifizieren:

- Sicherheitsbedingung (S):

  AG !(CCState(a1,l1)=exclusive & CCState(a2,l1)=exclusive)

- Lebendigkeitsbedingung (L):

  AG AF (CurPhase(a1,l1)=ready & CurPhase(a2,l1)=ready)

Die Sicherheitsbedingung sagt aus, dass beide Knoten niemals zur selben Zeit Zugriff auf eine einzelne Speicherlinie haben dürfen. Die Lebendigkeitsbedingung stellt sicher, dass alle Anfragen auch irgendwann bearbeitet werden. Keine dieser beiden

Eigenschaften wird von dem Protokoll erfüllt. In Tabelle 6.1 sind die Laufzeiten der Verifikation der entsprechenden ASM mit den Werkzeugen CoreASM2Promela und ASM2SMV wie von Ma veröffentlicht [Ma07] dargestellt. Tabelle 6.2 zeigt die Werte bei Verifikation mit CoreASM.

Eigenschaft	**Sicherheit**	**Lebendigkeit**
CoreASM2Promela [Ma07]	43s	7s
ASM2SMV [Win01]	438s	921s

Tabelle 6.1: Laufzeiten der Verifikation des *Flash Cache Coherence*-Protokolls mit CoreASM2Promela und ASM2SMV nach Ma [Ma07].

Eigenschaft	**Sicherheit**	**Lebendigkeit**
Anzahl der gespeicherten Zustände	873	173
Anzahl der Transitionen	2240	247
Anzahl der erzeugten Zustände	3011	323
Laufzeit	339s	35s
Speicherverbrauch	3051 KB	13 KB
Gegenbeispiel		
Anzahl der Zustände	91	7
Anzahl der Transitionen	140	7

Tabelle 6.2: Verifikation des *Flash Cache Coherence*-Protokolls mit [mc]square.

Abbildung 6.1 zeigt einen Ausschnitt der grafischen Repräsentation des Gegenbeispiels für die Lebendigkeitsbedingung. Diese Möglichkeit unterscheidet [mc]square von anderen Werkzeugen zum Model-Checking von ASMs, da es das Gegenbeispiel im ASM-Zustandsraum anzeigen kann.

Obwohl die Testläufe, die zu den Ergebnissen in den Tabellen 6.1 und 6.2 geführt haben, auf unterschiedlichen Maschinen durchgeführt wurden, kann die Schlussfolgerung gezogen werden, dass [mc]square ein vergleichbares Laufzeitergebnis wie CoreASM2Promela und ASM2SMV liefert. Dies liegt an der Verwendung eines lokalen Model-Checking-Algorithmuses, der den Zustandsraumaufbau bei Existenz eines Gegenbeispiels abbricht.

Terminierungsdetektion Ein *Distributed Termination Detection*-Algorithmus erkennt die Terminierung eines verteilten Programms, das auf mehreren Netzwerknoten

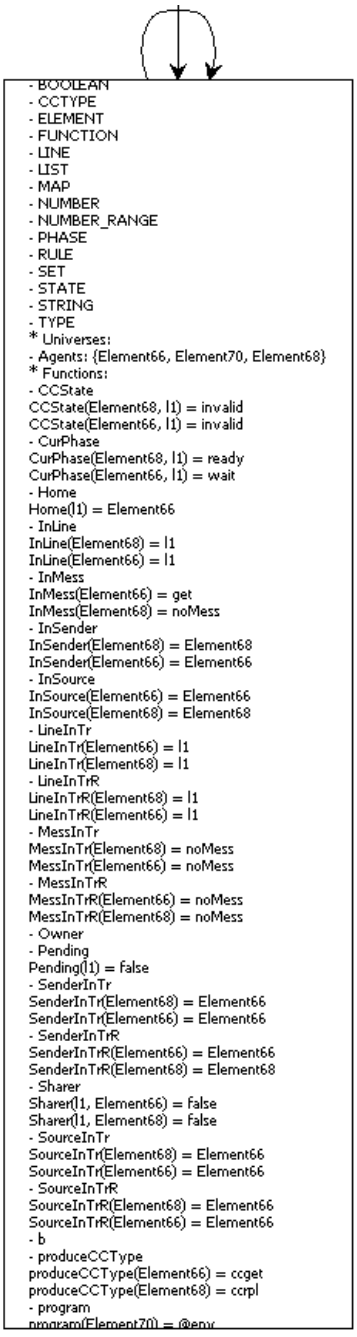

Abbildung 6.1: Ausschnitt der grafischen Repräsentation des Gegenbeispiels für die Lebendigkeitsbedingung in [mc]square.

läuft. Jeder Knoten kann dabei im aktiven oder passiven Zustand sein. Aktive Knoten können Nachrichten an andere Knoten schicken, um Teile ihrer Berechnungen an diese zu transferieren. Nach dem Empfang einer solchen Nachricht wird ein passiver Knoten aktiviert. Nach der Beendigung einer Berechnung wechseln die Knoten wieder vom aktiven in den passiven Zustand zurück. Der Systemzustand, in dem alle Knoten passiv sind und keine Nachrichten mehr unterwegs sind, ist derjenige, der vom Algorithmus erkannt werden soll.

Eschbach modelliert einen solchen Algorithmus von Dijkstra et al. [DFG83] als ASM und beweist einige seiner Eigenschaften von Hand [Esc99]. Eine CoreASM-Version dieses Algorithmus wird von Ma [Ma07] genutzt, um eine Übersetzung von CoreASM nach Promela zu evaluieren. Dazu begrenzt er die maximale Anzahl der Nachrichten, die jeder Knoten senden kann. Hier wird die von Ma veröffentlichte ASM-Spezifikation genutzt, um die folgende Eigenschaft zu verifizieren:

$$\text{AG (termination(.)=true} \Rightarrow \text{AF terminationDetected(.)=true)}$$

Sie sagt aus, dass jede Terminierung auch irgendwann entdeckt wird. In Tabelle 6.3 sind die Ergebnisse der Verifikation für zwei Knoten und maximal einer Nachricht pro Knoten dargestellt. Die Verifikation der Übersetzung von CoreASM nach Promela dauert nur eine Sekunde [Ma07], die von der ASM-Workbench nach SMV knapp fünf Sekunden [Win01].

Anzahl der gespeicherten Zustände	89208
Anzahl der Transitionen	430787
Anzahl der erzeugten Zustände	440952
Laufzeit	15 Stunden, 14 Minuten
Speicherverbrauch	261857 kByte

Tabelle 6.3: Verifikation des *Distributed Termination Detection*-Algorithmus mit [mc]square.

Offensichtlich liegt die Laufzeit von über 15 Stunden weit über der anderer Ansätze zum Model-Checking von ASMs. Dennoch zeigt der Ansatz seine Anwendbarkeit und stellt seine Funktionstüchtigkeit unter Beweis. Die Anzahl der gespeicherten Zustände und insbesondere der Speicherverbrauch sind im Vergleich zu anderen Model-Checking Problemen bemerkenswert klein, was an den Abstraktionsfähigkeiten der ASMs liegt.

6.1.3 Bewertung

Die Modellierung der funktionalen Anforderungen in einem ASM-Grundmodell helfen diese zu präzisieren. Die Formalisierung ermöglicht ein eindeutiges, konsistentes

und dennoch knappes, erstes Modell des Systems aufzustellen. Neben der Klärung der Vorstellungen eines einzelnen Entwicklers ist eine ASM ein gutes Kommunikationsmittel, das Missverständnissen zwischen Entwicklern vorzubeugen hilft. So wurde den Entwicklern erst durch die ASM-Modellierung klar, dass die Distanzregelung einen zeitlichen Abstand als Führungsgröße benötigt und keinen räumlichen.

Die Abstraktionsmechanismen der ASMs ermöglichen ihren frühen Einsatz, lassen den Entwicklern aber gleichzeitig die Freiheit bei der Wahl des Abstraktionsgrades. Insbesondere die Verwendung von abstrakten Daten vereinfacht die frühe Modellierung im Grundmodell. Die implizite Parallelität hilft zwar bei der Abstraktion von nicht benötigter Sequenzialität, ist aber auch hinderlich. Dies trifft insbesondere auf komplexere Berechnungen zu, die in mehreren Schritten ausgeführt werden, wie es etwa bei der Verfeinerung des Reglers auftritt. Die Implementierung der **choose**-Regel in CoreASM ermöglicht die nichtdeterministische Wahl nur für Aufzählungen. Im Rahmen der Fallstudie wäre die Wahl über dem *NUMBERS*-Universum mehrmals nützlich gewesen, aber nicht möglich, so dass eine unschöne Problemumgehung gewählt werden musste.

Aufgrund ihrer Nähe zu vielen Programmiersprachen ist die ASM-Sprache für technisch Ausgebildete meist leicht zu erlernen. In einigen Fällen kann auch eine hier nicht verwendete grafische Notation verwendet werden. Die Möglichkeit der Simulation abstrakter Darstellungen des Systems in frühen Phasen des Entwurfsprozesses war für das Verständnis der Anforderungen sehr hilfreich und wurde häufig genutzt.

Das verwendete Werkzeug CoreASM hat auch einige Nachteile. Neben den bereits oben angesprochenen Unzulänglichkeiten der **choose**-Regel ist insbesondere die Fehlerbeseitigung aufgrund schwer verständlicher Fehlermeldungen schwierig. Die Darstellung der Zustände bei einem Simulationslauf ist nur sehr mühsam zu überblicken. Besser wäre eine konfigurierbare Darstellung, die nur die für den Entwickler interessanten Lokationen anzeigt.

Mit Hilfe von Verfeinerungen können ASMs den Übergang von den Anforderungen zur Implementierung vereinfachen. In beliebig vielen Schritten können einzelne Entwurfsentscheidungen modelliert werden. Dabei kann jede einzelne Stufe simuliert werden, so dass jedes neue Modell sofort validiert und verifiziert werden kann. Der letzte Verfeinerungsschritt vom ASM-Modell in die Implementierungssprache ist meist einfach, da die ASM-Syntax derjenigen vieler Programmiersprachen ähnelt.

Insgesamt lohnt sich der Aufwand einer ASM-Modellierung insbesondere für größere, sicherheitskritische Teilsysteme. Eine der Stärken des Ansatzes ist aus Sicht des Autors gerade diese Möglichkeit der formalen Modellierung und Verifikation von Teilsystemen. Die Schnittstellenvereinbarungen unterstützen dieses Anliegen durch klare Aufgabenverteilung mit Hilfe von Typisierung sowie Vor- und Nachbedingungen.

Die Möglichkeit der frühen Verifikation der Modelle ist eine der Stärken der ASM-Modelle. Im Rahmen der Fallstudie konnte so ein vorher unerkannter Fehler

im ASM-Grundmodell gefunden werden. Allerdings war die Formalisierung der Anforderungen als CTL-Formel schwierig.

Die Verifikation mit [mc]square ist funktionstüchtig und liefert für invalide Formeln Laufzeitleistungen, die mit anderen Werkzeugen vergleichbar sind. Die Möglichkeit der Darstellung von Gegenbeispielen im ASM-Zustandsraum unterscheidet [mc]-square von allen anderen ASM-Model-Checkern. Sie vereinfacht das Nachvollziehen von Fehlern oder macht es in manchen Fällen überhaupt erst möglich.

Für valide Formeln liegen die Laufzeiten dagegen um etwa zwei Größenordnungen über denen anderer Ansätze. Dies liegt in erster Linie am verwendeten Simulator. Dies zeigt sich am Fehlen von Laufzeitunterschieden bei der Verwendung unterschiedlicher Kompressionstechniken für die Zustände. Bei Verwendung einer aufwändigen zip-Kompression zeigen sich für ASMs keine Laufzeitunterschiede im Vergleich zur Verwendung einer einfachen Lauflängenkodierung. Wird [mc]square zur Verifikation von Assemblercode verwendet ergeben sich in der gleichen Situation Laufzeitunterschiede um einen Faktor von etwa zehn [SK06]. Da die Kompression den Prozessor stark belastet deutet dies darauf hin, dass dieser beim Model-Checking von ASMs häufig im Leerlauf ist und deshalb genug Zeit für eine zusätzliche Kompression bleibt. Die Vermutung liegt nahe, dass dies an vielen Kontextwechseln bei der Simulation in CoreASM liegt, die auch an der stark objektorientierten Struktur des Simulators zu erkennen sind. Diese Struktur ist aus den Anforderungen der Verständlichkeit und Erweiterbarkeit an CoreASM entstanden und ist daher nicht für den Einsatz in einem Model-Checker optimiert.

Im Gegensatz zur Laufzeit ist der Speicherbedarf bei der Verifikation überraschend gering. Leider gibt es hierfür keine veröffentlichten Vergleichswerte der anderen Werkzeuge. Obwohl ein Zustand in CoreASM aus vielen Zeichenketten aufgebaut ist, die nicht speichereffizient sind und die verwendete Serialisierung optimiert werden kann, liegt der Speicherbedarf für einen Zustand in Abhängigkeit von der ASM bei einigen wenigen Kilobyte. Dies liegt an der starken Redundanz innerhalb der Zeichenketten eines Zustands die durch die Kompression der Zustände effizient entfernt werden kann.

Um die Laufzeitergebnisse zu verbessern, bieten sich in erster Linie Optimierungen am Simulator an. In einer ähnlichen Situation wurde für die Verifikation von Assemblercode in [mc]square mit einem optimierten Simulator ein Zeitgewinn um einen Faktor 100 erzielt [SK06]. Die Implementierung eines eigenen Simulators erscheint wegen der breiten Nutzung des vorhandenen Werkzeugs jedoch nicht sinnvoll.

Die Einbindung von CoreASM in [mc]square profitiert enorm von den durchdachten Architekturen der beiden kombinierten Werkzeuge. Die Plugin-Architektur von CoreASM ermöglicht seine gute Anpassbarkeit beispielsweise durch das veränderte **choose**-Plugin. [mc]square zeigt eine Erweiterbarkeit, die weit über den ursprünglich angedachten Rahmen hinausgeht.

6.2 Strukturmodellierung

Wie bereits in Kapitel 1 erwähnt, beeinflussen die qualitativen Anforderungen an ein System insbesondere seine Architektur [SG96]. Dokumentierte Architekturmuster und ihr Einfluss auf einzelne Systemqualitäten helfen zwar beim Entwurf, die Kombination mehrerer Muster zu einer Gesamtarchitektur ist wegen der wechselseitigen Beeinflussung der Muster dennoch ein Prozess, der in großem Maße von der Erfahrung des Architekten abhängt.

Zur Erfassung der eigentlichen Anforderungen eignen sich Szenarien, die in der BON mit Hilfe von Tabellen erfasst werden. So zeigt Tabelle 2.6 auf Seite 21 beispielsweise ein Szenario zur qualitativen Anforderung der Erweiterbarkeit des Systems durch einen neuen Abstandssensor. Mit Hilfe weiterer dynamischer und statischer Tabellen können erste Strukturentwürfe festgehalten und diskutiert werden. Im Rahmen der Fallstudie wurden jedoch nur die SzenarioTabellen von den Entwicklern als hilfreich eingeschätzt. Im Allgemeinen sind die Tabellen nur in sehr frühen Entwurfsphasen nützlich.

Neben der oben angesprochenen Erweiterbarkeit haben auch noch qualitative Anforderungen aus den Bereichen Wiederverwendbarkeit, Änderbarkeit, Wartbarkeit, Testbarkeit und Sicherheit einen Einfluss auf die gewählte Struktur des ACC-Systems. Da hier jedoch die Modellierungsfähigkeiten der BON bewertet werden, wird nicht weiter auf die Herleitung der Architektur eingegangen. Stattdessen werden Beispiele für die Darstellung aller in Kapitel 4.3 genannten drei Arten von Architektursichten präsentiert, bevor die Strukturmodellierung eingebetteter Systeme mit der BON bewertet wird. Dabei wird hier nur der fertige, implementierte Entwurf dargestellt. Dieser ist jedoch erst durch Abwägung möglicher Alternativen und genauerer Analyse anderer Entwurfsmöglichkeiten entstanden.

6.2.1 Modulsichten

Abbildung 6.2 zeigt ein statisches BON-Diagramm des Systems. Zu sehen sind Klassen zur Abfrage von Sensorwerten, die noch zusätzlich zu einer Gruppe zusammengefasst sind, zur Abrage von Nutzereingaben (`COM_REMOTE`) sowie zur Ansteuerung der Aktuatoren (`SET_PWM`). Diese vier sind in komprimierter Form dargestellt, während die Hauptdatei ACC mit typisierter Schnittstelle abgebildet ist.

Die Funktion `set_speed_setpoint` erwartet einen Parameter vom Typ `INTEGER`, die Funktionen `speed_error` und `distance` liefern einen Wert eben diesen Typs zurück. Eine Besonderheit stellt die Funktion `acc_main_rule` dar. Diese ist mit Vor- und Nachbedingungen, erkennbar am `?` bzw. `!`, dargestellt. Vorbedingungen müssen vom Klienten der Funktion erfüllt werden. In diesem Fall müssen die Werte des Geschwindigkeitssensors in einem physikalisch plausiblen Intervall liegen, wodurch ein funktionierender Geschwindigkeitssensor sichergestellt werden soll. Bei Einhaltung

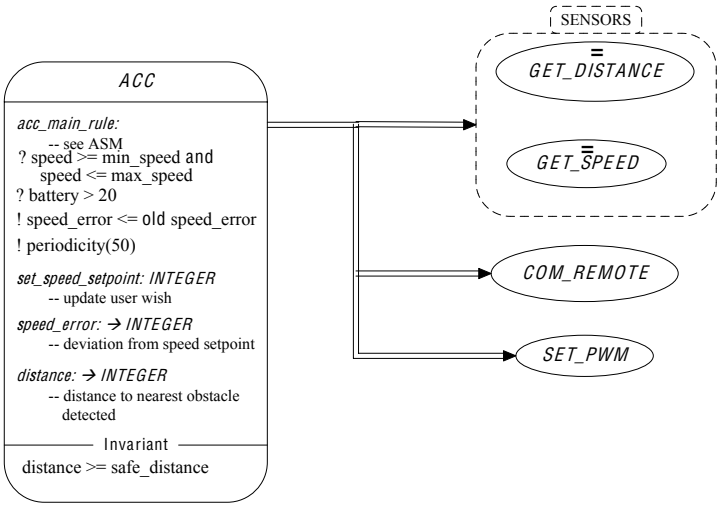

Abbildung 6.2: Beispiel für eine Modulsicht auf das ACC-System mit einem statischen BON-Diagramm.

der Vorbedingung garantiert die Funktion die Nachbedingung, also in diesem Fall, dass der Regler die Regelabweichung nicht verschlechtert. Die `acc_main_rule` stellt auch ein Beispiel für den Verweis auf ein zugehöriges Funktionsmodell in einer ASM dar. Zusätzlich ist noch eine Invariante dargestellt. Diese muss für die beinhaltende Datei immer gelten. Im Beispiel darf der Zeitabstand zum nächsten erkannten Hindernis nie kleiner als ein als sicher betrachteter, von der aktuellen Geschwindigkeit abhängiger Abstand werden.

Zusätzlich ist eine physikalische Anforderung an die Plattform, namentlich die Batteriespannung, in den Vorbedingungen des Hauptreglers formuliert. Das System soll nicht aktiviert werden, wenn nicht genügend Spannung zur Verfügung steht. Hier ist bereits ein Nachteil der Formulierung von Plattformanforderungen in Vorbedingungen zu erkennen: Da die Bedingung in der Implementierung nur vor dem Eintritt in die Routine überprüft wird, gilt sie nur zu deren Beginn. Da die Plattform jedoch parallel zur Software weiterläuft, also quasi eine nebenläufige Ausführungseinheit darstellt, kann eine zu Routinenbeginn erfüllte Vorbedingung während der Laufzeit der Routine negiert werden. Um eine regelmäßige Überprüfung der Anforderung zu gewährleisten ist es daher empfehlenswert, sie für kurze, periodisch aufgerufene Routinen zu verwenden. Im hier beschriebenen Beispiel ist dies der Fall, da die entsprechende Routine periodisch aufgerufen wird, wie an der als Nachbedingung notierten Zeitanforderung zu sehen ist.

Wie zu erkennen ist, müssen in den Diagrammen nicht alle vorhandenen Klassen

dargestellt werden. So fehlt beispielsweise die Klasse `CAN_COM` in der Abbildung, da deren Funktionalität von einer Bibliothek geliefert wird und sie somit nicht implementiert werden muss. Auch das Zusammenfassen beliebiger Klassen zu Clustern, wie in diesem Fall für die Sensorklassen, ermöglicht das Verbessern der Übersichtlichkeit von Diagrammen.

Anhand des Gleichheitszeichens ist zu erkennen, dass die beiden Sensorklassen nebenläufig implementiert sind. Diese Struktur wurde im Fallbeispiel gewählt, weil die betreffenden Klassen ohne Änderungen am Quelltext für einen Parkassistenten wiederverwendbar sein sollen. Für den ACC-Regler ist zusätzlich eine Zeitanforderung für eine regelmäßige Ausführung in der Nachbedingung der `acc_main_rule` angegeben. Diese schreibt die periodische Ausführung in einem bestimmten Intervall vor, welches sich aus den physikalischen Rahmenbedingungen des Systems wie der maximalen Geschwindigkeit und Bremsverzögerung ergibt. Hier sei nochmal betont, dass es sich um eine Anforderung und nicht um ein Echtzeitattribut wie z. B. die WCET der Methode handelt.

6.2.2 Komponentensichten

In Abbildung 6.3 ist ein einfaches Szenario durch die beteiligten Objekte mit ihren Aufrufbeziehungen dargestellt. Im Gegensatz zu statischen Diagrammen wird hier der Ablauf der Aufrufe in typischen Situationen gezeigt. Im Beispiel wird das ACC-Objekt von einem mehrerer `CAN_COM`-Objekte aktiviert. Der Aufruf der Sensorengruppe im zweiten Schritt ist gleichbedeutend mit dem Aufruf jedes einzelnen Gruppenobjekts. Schließlich können Aufrufbeziehungen wie im dritten Schritt zu sehen auch gegenseitig sein. Dabei steht die Indexnummer näher zum aufgerufenen Objekt.

Komponentensichten werden vor allem zum Durchspielen von Szenarien verwendet. Sie werden im Fallbeispiel ohne Änderungen, wie in der normalen BON verwendet.

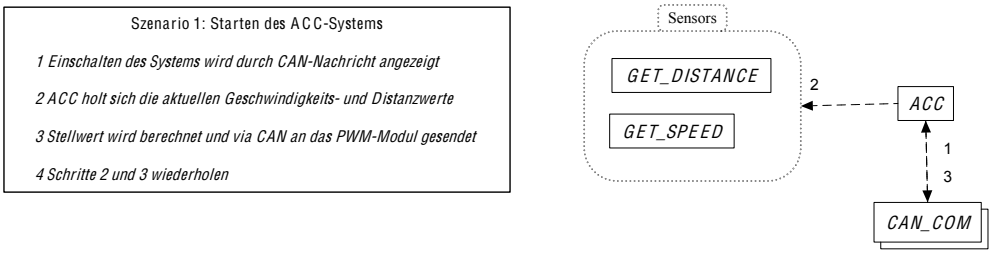

Abbildung 6.3: Beispiel für eine Komponentensicht auf das ACC-System mit einem dynamischen BON-Diagramm.

6.2.3 Verteilungssichten

Abbildung 6.4 zeigt ein dynamisches BON-Diagramm, das erst gegen Ende des Entwurfs erstellt wurde. Die dargestellte Gruppierung zeigt die Verteilung der Softwaremodule auf verschiedene Hardwareplattformen wie BML_1 usw. Hier sind auch einige Module, wie der VOTER dargestellt, die nicht direkt zum ACC-System gehören, aber für das Verständnis des kompletten Fahrzeugs wichtig sind. Zusätzlich wird von der Möglichkeit, Klienten-Anbieter Beziehungen zu benennen, Gebrauch gemacht, um die Verwendung des CAN-Busses darzustellen.

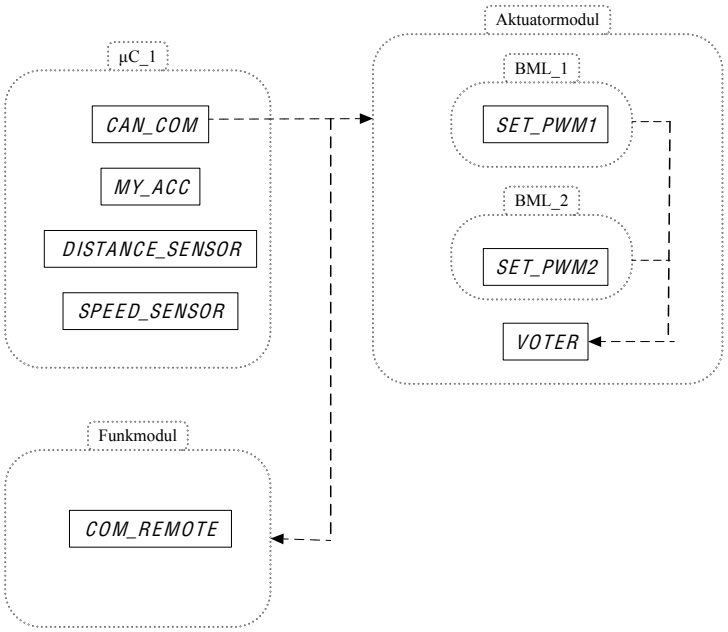

Abbildung 6.4: Beispiel für eine Verteilungssicht auf das ACC-System mit einem statischen BON-Diagramm.

Für eingebettete Systeme werden Verteilungssichten wie in der Abbildung dargestellt vor allem für die Verteilung von Software-Moulen auf Hardware-Knoten benutzt. Wie im Beispiel können dann auch Verknüpfungen dieser Knoten mit Bussen oder ihre Energiversorgung dargestellt werden.

6.2.4 Bewertung

Wie in [CBB+03] vorgeschlagen, wird das ACC-System hier aus drei Architektursichten dargestellt, die jeweils zu einem der drei Typen von Sichten gehören: eine

Modulsicht, die die Beziehungen der Quelltextmodule abbildet, eine Komponentensicht zur Darstellung des Verhaltens von Laufzeitobjekten und eine Verteilungssicht, die das Verhältnis zwischen Umwelt und System zeigt. Die Darstellung der drei Arten von Sichten ist problemlos möglich und wird insbesondere durch die Freiheit bei der Wahl der Cluster vereinfacht.

Insgesamt hatten die Entwickler beim Entwurf des ACC-Systems den Eindruck, dass die Darstellungsform die notwendige Freiheit, die beim Architekturentwurf nötig ist, lässt ohne dabei nicht implementierbare Abstraktionsformen zu verwenden. Auch scheint die Verwendung mit allen gängigen Prozessmodellen und sogar agilen Methoden problemlos möglich zu sein. Die Unterstützung unterschiedlicher Architektursichten durch das einfache Konzept der Gruppierung und die schnelle Erlernbarkeit der Modellierung sind weitere Pluspunkte. Die Integration von ASMs in die vorgegebene Struktur ist durch die orthogonalen Modellierungseigenschaften der BON unkompliziert.

Ein unvermeidlicher Nachteil bei der Verwendung einer informellen Darstellungsform wie der BON, ist die mögliche Inkonsistenz der verschiedenen Sichten. Dies zu vermeiden liegt in der Verantwortung der Entwickler und kann nicht automatisch sichergestellt werden. Durch den nahtlosen Übergang von der BON in die Programmiersprache Eiffel können solche Inkonsistenzen jedoch zumindest in späteren Entwurfsphasen erkannt und behoben werden. Durch die Möglichkeit der Rücktransformation von Implementierungsänderungen in die BON-Darstellung kann dann zusätzlich das Altern der Strukturmodelle vermieden werden.

Die direkte Unterstützung von DBC hilft nicht nur bei der Anreicherung der Schnittstellen mit semantischen Informationen sondern wird in der Fallstudie auch zur Modellierung physikalischer Anforderungen verwendet. Die Darstellung von Plattformanforderungen in Vorbedingungen ist wie oben erläutert wegen der impliziten Nebenläufigkeit der Plattform nur für kurze Routinen sinnvoll einsetzbar. Die Formulierung von Zeitanforderungen in den Nachbedingungen ist dagegen sinnvoll und meist leicht aus den physikalischen Randbedingungen des Systems abzuleiten.

Wie für Zusicherungen, die keine physikalischen Anforderungen darstellen gilt, dass der Hauptvorteil in der klaren Schnittstelle der Routine mit präzisen Pflichten für Dienstanbieter, -aufrufer und Umgebung liegt. Ein Programmierer kann die Bedingungen zwar auch beim Betreten oder Verlassen einer Routine testen, aber es gibt keine verbindliche Regel, wie eine Verletzung der Bedingung behandelt werden soll. Durch DBC wird das mehrmalige Überprüfen der Parameter vermieden und die Überprüfung der Zusicherungen zur Laufzeit inklusive einer Schuldzuweisung beim Verletzen der Bedingungen ermöglicht. Diese Vorteile sind insbesondere bei der Verwendung von Bibliotheken relevant.

6.3 Implementierung

Um die Leistungsfähigkeit der Laufzeitumgebung zu testen wurde das ACC-System ausgehend vom feinsten ASM-Modell und den BON-Sichten auf seine Struktur auf drei verschiedene Arten implementiert.

Die erste Implementierung nutzt die echtzeitfähige SCOOP-Implementierung. d. h. das System wird unter Verwendung des Schlüsselwortes **separate** in Eiffel programmiert. Aus den Zeitanforderungen in den Nachbedingungen werden von Hand Echtzeitparameter wie Fristen oder Prioritäten abgeleitet und in einer Konfigurationsdatei abgelegt. Aus dem Eiffel-Programm und der Konfigurationsdatei wird mit Hilfe des erstellten Präprozessors automatisch ein **separate**-loses Eiffel-Programm erzeugt, das entsprechende Aufführungsfäden mit den entsprechenden Fristen und Prioritäten mit Hilfe der erstellten Bibliothek verwirklicht. Dieses Eiffel-Programm kann mit zwei Kompilierungen in Binärcode für die verwendete Mikrocontrollerplattform übersetzt werden. Dieser Binärcode wird auf dem Mikrocontroller unter dem Echtzeitbetriebssystem FreeRTOS und der angepassten Eiffel-Laufzeitumgebung ausgeführt.

Die zweite Implementierung ist ein normales Eiffel-Programm, das ohne SCOOP-System auskommt. Hier werden die Fäden direkt mit Hilfe der erstellten Bibliothek programmiert und können dann wie oben zunächst mit einem Eiffel-Kompilierer nach ANSI-C und anschließend mit einem C-Kompilierer in Maschinencode übersetzt werden. Diese Implementierung nutzt also ebenfalls das Echtzeitbetriebssystem FreeRTOS und die angepasste Eiffel-Laufzeitumgebung.

Die dritte Implementierung wird direkt in der Sprache C programmiert. Um mit den anderen beiden Implementierungen vergleichbar zu sein, nutzt sie ebenfalls das Echtzeitbetriebssystem um nebenläufige Ausführungsfäden umzusetzen.

6.3.1 Vergleich

Um die Leistungen der drei Implementierungen des ACC-Systems zu vergleichen, werden die folgenden Werte gemessen:

Größe ist die Größe der Binärdatei, die auf den Mikrocontroller geladen wird und damit der statische Speicherbedarf des Systems.

belegter Speicher gibt die Größe des RAM-Speichers auf dem Mikrocontroller zur Laufzeit an, die nach einem Durchlauf des ACC-Reglers von dem Programm belegt ist und stellt somit ein Maß für den dynamischen Speicherbedarf des Systems dar.

Startzeit ist die Zeit vom Starten des Mikrocontrollers bis zur erstmaligen Ausführung des ACC-Reglers. Sie ist somit ein Maß für die Initialisierungsdauer des Systems.

Laufzeit ist die Dauer eines Durchlaufs des ACC-Reglers bei neuen Sensormessungen. Sie ist hier als Maß für den Laufzeitbedarf des Systems zu verstehen.

Die Messung findet in einem Simulator des verwendeten Mikrocontrollers statt, der zur Entwicklungsumgebung gehört. Wegen der fehlenden Buskommunikation in der Simulation müssen zur Ermittlung des belegten Speichers sowie der Laufzeit die Implementierungen leicht abgeändert werden. Dazu werden die Module, die eigentlich Werte über den CAN-Bus empfangen so geändert, dass sie Beispielwerte an den ACC-Regler liefern. Daher gelten die jeweiligen Messungen auch nur für das ACC-Reglermodul. Tabelle 6.4 fasst die Ergebnisse der Messungen zusammen.

	C	Eiffel	SCOOP
Größe	30 kB	42 kB	74 kB
belegter Speicher	8,8 kB	13,1 kB	24,3 kB
Startzeit	4,0 ms	4,5 ms	11,2 ms
Laufzeit	16,4 ms	13,1 ms	41,8 ms

Tabelle 6.4: SCOOP Evaluation

6.3.2 Bewertung

Wie in Tabelle 6.4 zu sehen ist, ist der statische Speicherbedarf der SCOOP-Implementierung mehr als doppelt so groß wie der der reinen C-Implementierung. Auf die Eiffel-Laufzeitumgebung entfallen dabei etwa die 12 Kilobyte Unterschied zwichen C- und Eiffel-Implementierung. Die restlichen Mehrkosten kommen von den vielen vom Präprozessor erzeugten Ausführungsfäden, die jeweils einen Puffer und zwei Mutexe nutzen.

Ähnlich verhält es sich mit dem dynamischen Speicherbedarf. Auch hier ist der geringe Unterschied zwischen Eiffel und C überraschend, während die SCOOP-Implementierung aus den selben Gründen wie oben hier sogar den dreifachen Bedarf der C-Implementierung hat.

Die Startzeiten für die C- und Eiffel-Implementierung sind fast gleich, was für eine besonders effiziente Eiffel-Laufzeitumgebung spricht. Der Mehrbedarf der SCOOP-Implementierung ergibt sich aus der Initialisierung von deutlich mehr Objekten als in den anderen beiden Fällen.

Die Laufzeiten der drei Implementierungen überrascht mit einer Eiffel-Implementierung, die schneller als die C-Implementierung ist. Dies liegt vermutlich am verwendeten Eiffel-Kompilierer, der stark optimierten C-Code erzeugt. Dieser ist in dieser Kategorie besser als der handgeschriebene C-Code, der von einem erfahrenen

Programmierer implementiert wurde. Die SCOOP-Implementierung ist auch in dieser Kategorie mit Abstand am schlechtesten.

Wie an den Werten der Tabelle zu erkennen ist, ist SCOOP zur Zeit nicht in der Praxis einsetzbar. Dies gilt insbesondere für ressourcenkritische, eingebettete Systeme. Im Gegenteil zu den beiden anderen Implementierungen bietet die SCOOP-Variante jedoch den Vorteil der besseren Wiederverwendbarkeit. So können z. B. die Sensorklassen des ACC-Systems ohne Änderung einer einzigen Quelltextzeile für einen Parkassistenten wiederverwendet werden. Dies liegt daran, dass die zu ändernden Echtzeitattribute nicht Teil der Implementierung sondern in einer Konfigurationsdatei abgelegt sind. So erfüllt die SCOOP-Laufzeitumgebung auch als einzige die eigentliche Motivation zu ihrer Untersuchung im Rahmen dieser Arbeit: Die einfache und direkte Erzeugung aus BON-Diagrammen mit Zeitanforderungen in den Nachbedingungen. Die Einhaltung dieser Nachbedingungen wird zur Laufzeit durch die Mechanismen des DBC überwacht und im Fehlerfall mit einer Ausnahmebehandlung aufgefangen.

6.4 Abschließende Bewertung

Neben den Bewertungen der einzelnen Aspekte der Fallstudie in den einzelnen obigen Unterkapiteln wird hier eine Bewertung zum kombinierten Entwurf mit BON und ASMs sowie die Implementierung in der Laufzeitumgebung gegeben. Diese erfolgt unter Berücksichtigung der in Abschnitt 1.2 beschriebenen Zielsetzung.

Die Modellierung mit ASMs und BON ist früh im Entwicklungsprozess, namentlich bei der Modellierung der Anforderungen einsetzbar, lässt sich aber durch Verfeinerungen auch bis zur Implementierung weiterverwenden.

Die Trennung von Struktur- und Verhaltensmodellierung, entsprechend der Unterscheidung von funktionalen und qualitativen Anforderungen, erschien beim Entwurf des ACC-Systems sinnvoll. Insbesondere die Kombination von formaler Modellierung des Verhaltens mit der weniger strikten Darstellung der Struktur erleichterte den Entwurf des Systems. Mit der formalen Verhaltensmodellierung als ASM ergeben sich die Vorteile der Eindeutigkeit der Anforderungen aber insbesondere auch die Möglichkeit der Ausführung durch Simulation. Die für sicherheitskritische Systeme empfohlene, formale Verifikation lässt sich sofort auf den bereits vorhandenen Dokumenten des Entwicklungsprozesses ausführen und zwar über alle Verfeinerungsschritte des Entwurfs hinweg.

Der Aufwand einer ASM-Modellierung erscheint insbesondere für größere, sicherheitskritische Teilsysteme sinnvoll. Eine der Stärken des Ansatzes ist aus Sicht des Autors gerade die Möglichkeit der formalen Modellierung und Verifikation von Teilsystemen. Die Schnittstellenvereinbarungen unterstützen dieses Anliegen durch klare Aufgabenverteilung mit Hilfe von Typisierung sowie Vor- und Nachbedingungen.

Die weniger strikte Modellierung der Struktur in der BON lässt den Entwicklern dagegen die benötigten Freiheiten beim Architekturentwurf und lässt sich durch die Ausnutzung der Cluster für die Darstellung aller drei empfohlener Architektursichten nutzen. Sie unterstützt zusätzlich die geforderten physikalischen Anforderungen mit Hilfe der Darstellung des DBC. Die frühe Integration dieser speziellen Anforderungen an eingebettete Systeme in die Modelle spiegelt ihre Wichtigkeit für diese Art von Systemen wider.

Der nahtlose Übergang von den Anforderungen bis hin zur Implementierung ist möglich. Dagegen ist der umgekehrte Weg, die Rückkopplung von einer eventuell geänderten Implementierung in die Modelle nicht immer möglich. Während dies für ASMs und die statischen BON-Diagramme möglich erscheint, ist es für dynamische BON-Diagramme nicht automatisiert möglich sondern nur durch händische Übersetzung. Dennoch verhindert die mögliche Rückkopplung von ASMs und statischen BON-Diagrammen das Altern der Entwurfsdokumente in einem für die Fallstudie ausreichendem Maß. Ob dies auch für komplexere Projekte gilt muss aber angezweifelt werden.

Gerade wegen ihrer unterschiedlichen Aufgaben lassen sich die beiden Modellierungsarten BON und ASMs gut kombinieren. Ähnlich zur *Model Driven Architecture* ermöglichen beide Darstellungen eine zumindest semi-automatische Transformation in eine Implementierungssprache. Während die einzelnen ASM-Verfeinerungen Entwurfsentscheidungen widerspiegeln und daher von Hand durchgeführt werden müssen, kann die letzte Übersetzung in eine Programmiersprache teilweise automatisiert erfolgen. Die BON stellt von Anfang an die Abstraktionsmechanismen der Sprache Eiffel zur Verfügung und kann daher problemlos in diese transformiert werden. Mit Hilfe der Laufzeitumgebung wird eine solche automatische Transformation auch für nebenläufige Systeme mit Zeitanforderungen ermöglicht.

Gegen eine solche Implementierung eingebetteter Systeme in der Praxis spricht sicherlich der erhöhte Ressourcenverbrauch. Die damit einhergehenden steigenden Stückkosten müssen aber gegen andere externe Qualitätsmerkmale, vor allem Wartbarkeit und Wiederverwendbarkeit abgewägt werden.

Insgesamt konnte in der Fallstudie die Zielsetzung aus Kapitel 1.2 erreicht werden.

7 Zusammenfassung und Ausblick

In den folgenden zwei Abschnitten wird zunächst eine Zusammenfassung dieser Arbeit gegeben, bevor ein Ausblick auf zukünftige Arbeiten gewagt wird.

7.1 Zusammenfassung

Funktionalität und Qualität eingebetteter Systeme werden heute in immer stärkerem Maße durch Software bestimmt. Aufgrund der immer stärkeren Verbreitung und Vernetzung kommt es zu einer Komplexitätssteigerung, deren Beherrschung immer mehr zu einem zentralen Wettbewerbsfaktor wird. Die große Bedeutung der Software spiegelt sich dabei im steigenden Anteil an den Gesamtkosten für Produktion und Entwicklung wider.

Der ingenieursmäßige Entwurf komplexer Softwaresysteme, das Software-Engineering, ist die Disziplin, die sich mit der Beherrschung der Komplexität beim Entwurf von Systemen beschäftigt. Die klassischen Methoden des Software-Engineering stoßen bei der Entwicklung eingebetteter Systeme an ihre Grenzen. Die Interaktion eingebetteter Systeme mit ihrer Umwelt führt zu physikalischen Anforderungen an die Plattform und an das Zeitverhalten der Systeme, die beim Entwurf des Systems nicht vernachlässigt werden dürfen.

In dieser Arbeit werden deswegen Entwurfsmodelle aus dem Software-Engineering um Eigenschaften erweitert, die typischerweise beim Entwurf eingebetteter Systeme auftreten. Dazu wird die übliche Unterscheidung zwischen funktionalen und qualitativen Anforderungen und ihre Repräsentation in Verhaltens- bzw. Strukturmodellen des Systems übernommen. Dabei konzentriert sich die Arbeit auf die frühen Phasen des Entwurfs, also den Übergang von den Anforderungen zum ersten Systemmodell und dessen schrittweise Verfeinerung bis hin zur Implementierung. Weitere Aktivitäten wie Testen oder Wartung werden nicht betrachtet.

Zur Verhaltensmodellierung werden ASMs eingesetzt, da sie schon mehrfach erfolgreich zur Modellierung eingebetteter Systeme genutzt wurden. Zur abstrakten Darstellung von Regelalgorithmen wird erstmals in dieser Arbeit der Einsatz der nichtdeterministischen Wahl vorgeschlagen. Weitere Erweiterungsmöglichkeiten wie die Integration von Plattform- und Zeiteigenschaften oder die Modellierung hybrider Systeme werden kurz diskutiert.

Die formale Darstellung als ASM ermöglicht die automatische Verifikation des Systems, was insbesondere für sicherheitskritische, eingebettete Systeme empfeh-

lenswert ist. In dieser Arbeit wird dazu ein Model-Checker entwickelt, der erstmals in der Lage ist, ASMs direkt zu unterstützen. Im Gegensatz dazu wurden ASMs in früheren Arbeiten in die Eingabesprachen von existierenden Model-Checkern übersetzt. Hier wird stattdessen der ASM-Simulator CoreASM in den Model-Checker [mc]square eingebaut. Letzterer ist ein Model-Checker für Assemblerprogramme, der den Zustandsraum mit Hilfe der Simulation der unterstützten Mikrocontroller aufbaut. Um CoreASM als virtuellen Mikrocontroller in [mc]square einzubauen, müssen seine Zustände serialisiert, deserialisiert und normiert werden, atomare Aussagen überprüfbar sein, sowie Änderungen am Scheduler, der nichtdeterministischen Wahl und dem Plugin-Rahmenwerk vorgenommen werden. Vorteile des so implementierten Model-Checkers sind die komplette Unterstützung der ASM-Syntax des Simulators, die einfache Erweiterbarkeit um zukünftige Plugins des Simulators sowie die Möglichkeit, Gegenbeispiele und Zeugen direkt im ASM-Zustandsraum anzeigen lassen zu können. Nachteilig wirkt sich aus, dass die Laufzeiteigenschaften des Simulators nicht für seine Verwendung in einem Model-Checker optimiert sind, sondern auf Verständlichkeit und Erweiterbarkeit hin ausgelegt sind.

Zur Strukturmodellierung wird die BON eingesetzt, weil sie DBC direkt unterstützt und dieses zur Integration der physikalischen Anforderungen genutzt werden soll. Zunächst werden das Vorgehen zur Integration von ASMs in die BON-Diagramme diskutiert und Bedingungen für die Konsistenz der Kombination aufgestellt. Um auch Mehragenten-ASMs mit ihrer asynchronen Kommunikation darstellen zu können, wird eine BON-Repräsentation nebenläufiger Klassen nach dem SCOOP-Modell eingeführt. Die physikalischen Anforderungen, die für den Entwurf eingebetteter Systeme wichtig sind werden weiter unterschieden in solche die durch das Ablaufen des Systems auf einer Plattform entstehen und solche die durch Interaktion mit der Umgebung zu Stande kommen.

Typische Beispiele für Plattformanforderungen entstehen meist durch Ressourcenbeschränkgungen des Systems wie z. B. begrenzter Platz im Programm- und Datenspeicher. Hier wird vorgeschlagen die Anforderungen des Systems an seine Plattform in die Vorbedingungen des DBC aufzunehmen, da diese vor Ausführung des Moduls sichergestellt werden. Insbesondere bedeutet dies, dass die Plattform selbst zum Vertragspartner des Software-Moduls wird und die Einhaltung der Vorbedingung sicherstellen muss. Wird eine solche Vorbedingung verletzt, wird eine Ausnahmebehandlung in der Hardware bzw. einem die Hardware repräsentierenden Software-Modul aufgerufen, die auf diese Verletzung reagieren kann, indem sie z. B. mehr benötigte Ressourcen zur Verfügung stellt.

Anforderungen, die aus der Interaktion eingebetteter Systeme mit ihrer Umwelt entstehen, sind meist Echtzeitanforderungen. Sie werden hier in die Nachbedingungen des DBC aufgenommen, da sie nach Ausführung des entsprechenden Moduls überprüft werden. Dies bedeutet, dass das Modul selbst dafür verantwortlich ist, sicherzustellen, dass diese Anforderungen erfüllt werden. Bei einer Verletzung einer

solchen Nachbedingung wird eine Ausnahmebehandlung im selben Software-Modul aufgerufen, die auf diese Verletzung reagieren kann, indem sie z. B. einen Fehlerzähler erhöht oder sich selbst neu startet.

Zum Abschluss der Strukturmodellierung mit der BON wird auf die Darstellung unterschiedlicher Architektursichten eingegangen. Das Zusammenfassen beliebiger Klassen und Objekte zu Clustern wird dazu ausgenutzt, drei in der Literatur empfohlene Arten von Sichten zu repräsentieren: Modul-, Komponenten- und Verteilungssichten. Die ersten beiden können nur mit den statischen bzw. dynamischen BON-Diagrammen dargestellt werden, letztere mit beiden. Im Unterschied zu den ersten beiden Sichten werden die Klassen bzw. Objekte dabei aber unter Gesichtspunkten zusammengefasst, die nichts mit Software-Elementen zu tun haben.

Während die Übersetzung von ASMs in eine Implementierung meist einfach ist und teilweise automatisiert werden kann, ist dies für die zeitbehafteten, nebenläufigen BON-Modelle nicht möglich. Daher wird eine Laufzeitumgebung vorgestellt, die eine solche automatische Transformation ermöglicht. Dazu wird in dieser Arbeit erstmals ein SCOOP-System auf einem Echtzeitbetriebssystem implementiert. Des weiteren wird SCOOP um die Überprüfung der Zeitanforderungen in den Nachbedingungen des DBC erweitert. Wegen der Unvorhersagbarkeit von Ausführungszeiten eignet es sich nur für weiche Echtzeitsysteme.

Mit Hilfe dieser Laufzeitumgebung können nebenläufige Systeme unter Verwendung des Schlüsselwortes **separate** in Eiffel programmiert werden. Aus den Zeitanforderungen in den Nachbedingungen werden von Hand Echtzeitparameter wie Fristen oder Prioritäten abgeleitet und in einer Konfigurationsdatei abgelegt. Aus dem Eiffel-Programm und der Konfigurationsdatei wird mit Hilfe eines Präprozessors automatisch ein **separate**-loses Eiffel-Programm erzeugt, das entsprechende Ausführungsfäden mit den entsprechenden Fristen und Prioritäten mit Hilfe einer erstellten Bibliothek verwirklicht. Dieses Eiffel-Programm kann mit zwei Kompilierungen in Binärcode für die verwendete Mikrocontrollerplattform übersetzt werden. Dieser Binärcode wird auf dem Mikrocontroller unter dem Echtzeitbetriebssystem FreeRTOS und einer angepassten Eiffel-Laufzeitumgebung ausgeführt.

Zur Evaluation des Ansatzes wird ein Abstandsregeltempomat auf einem Modellauto implementiert. Dieser beinhaltet typische Eigenschaften eingebetteter Systeme, wie mehrere Regler mit Rückkopplung, die Interaktion mit der Umwelt über Sensoren und Aktoren, Zeitanforderungen und sicherheitskritische Funktionen.

Die Formalisierung der funktionalen Anforderungen in einem ASM-Grundmodell bietet die typischen Vorteile der formalen Darstellung wie Eindeutigkeit und Konsistenz und vereinfacht die Kommunikation unter mehreren Entwicklern. Insbesondere die Möglichkeit der Simulation hilft bei der Präzisierung von Anforderungen. Die eingeführte, abstrakte Modellierung von Reglern ist bei fehlenden Umgebungsmodellen nur auf dem Papier jedoch nicht in der Simulation möglich.

Das Model-Checking ist funktionstüchtig und hilft bei der Fehlersuche. Hierzu ist

insbesondere die Darstellung von Gegenbeispielen im ASM-Zustandsraum von großer Hilfe. Im Vergleich mit anderen Werkzeugen zeigen sich aber Laufzeitnachteile mit einem Faktor von etwa 100. Dies Nachteile liegen in erster Linie am verwendeten Simulator, der nicht zur Verwendung in einem Model-Checker sondern mit den Zielen der Verständlichkeit und Erweiterbarkeit entworfen wurde.

Die Darstellung unterschiedlicher Sichten auf die Struktur des Systems mit der BON funktioniert ebenso wie die Kombination mit ASMs und die Darstellung nebenläufiger Klassen. Die physikalischen Anforderungen können in die Verträge des DBC aufgenommen werden. Im Fall der Plattformeigenschaften zeigt sich jedoch, dass diese nur bedingt als Vorbedingung verstanden werden können. Da sich Plattformeigenschaften auch während der Laufzeit eines Moduls ändern können, reicht es nur bei kurzen Routinen aus, diese lediglich vor der Ausführung der Routine zu überprüfen.

Die implementierte Laufzeitumgebung ermöglicht den automatischen Übergang aus zeitbehafteten, nebenläufigen BON-Modellen, hat aber gegenüber anderen Implementierungen erhebliche Ressourcennachteile. Bei einer Implementierung muss abgewägt werden, ob die verbesserte Wartbarkeit und Wiederverwendbarkeit des Systems diesen Nachteil aufwiegt.

Insgesamt werden in dieser Arbeit Verhaltens- und Strukturmodelle aus dem klassischen Software-Engineering um die für den Entwurf eingebetteter Software wichtigen Eigenschaften formale Verifikation und Modellierung physikalischer Anforderungen erweitert. Für den Industrieeinsatz sind die Laufzeiten des Model-Checking bzw. der semi-automatisch erzeugten Implementierung in der SCOOP-Laufzeitumgebung zu hoch.

7.2 Ausblick

Zukünftige Arbeiten sollten aus Sicht des Autors an den im Rahmen dieser Arbeit deutlich gewordenen Problemstellen ansetzen:

Die Verbesserung der Laufzeit beim Model-Checking von ASMs ist wie bereits diskutiert hauptsächlich durch eine Optimierung des Simulators CoreASM möglich. Erste Arbeiten auf diesem Gebiet beschäftigen sich mit der Verteilung der Simulation von Mehragenten-ASMs auf unterschiedliche Kerne einer CPU. Die Erweiterung von [mc]square zur Ermöglichung von verteiltem Model-Checking in einem Rechnernetzwerk wird ebenfalls bearbeitet und könnte auch weiteren Laufzeitgewinn einbringen.

Ebenso kann der Speicherbedarf beim Model-Checking verkleinert werden. Dazu könnten die sich häufig wiederholenden Zeichenketten eines CoreASM-Zustands mit Hilfe einer Legende auf effizientere Datenstrukturen abgebildet werden. Dann müsste die Legende nur noch ein einziges mal für den gesamten Zustandsraum und für jeden

einzelnen Zustand nur die Verweise darauf gespeichert werden. Dieses Vorgehen ist vermutlich nicht effizienter als die zur Zeit verwendete zip-Komprimierung, könnte jedoch in Zukunft eine Rolle spielen, wenn sich bei Verwendung einfacherer Kompressionsalgorithmen Laufzeitvorteile ergeben.

Um die Anzahl der zu speichernden Zustände zu verkleinern werden in [mc]square statische Analysen eingesetzt. Wegen der unterschiedlichen verwendeten Systemmodelle sind sie nicht direkt auf ASMs übertragbar. Insbesondere die Verwendung des sogenannten verzögerten Nichtdeterminismus erscheint jedoch interessant. Dabei wird eine Aufsplittung des Zustandsraums solange wie möglich verzögert, so dass die Zustände teilweise symbolisiert werden.

Die Modellierung hybrider Systeme ist durch Einführung hyperreller Zahlen in ASMs wie in Kapitel 3.1.2 erwähnt möglich. Die Implementierung eines entsprechenden Plugins für CoreASM könnte so die Simulation hybrider Systeme ermöglichen. Mit einem auf diese Weise erweiterten Simulator könnten auch in [mc]square Zeiteigenschaften dieser Systeme verifiziert werden, da sie dann Teil des Zustandsraums sind.

Bei der Integration von Plattformeigenschaften in die BON wäre eine Art Methodeninvariante wünschenswert. Anstatt wie bei der verwendeten Vorbedingung die Plattformanforderungen nur vor Eintritt in eine Methode zu überprüfen müsste diese regelmäßig während der Abarbeitung getestet werden. Ein solcher Automatismus würde die Formulierung der Anforderungen ebenso vereinfachen wie im hier vorgestellten Fall, wäre dabei aber nicht auf kurze Methoden beschränkt.

Schließlich ist eine effizientere Implementierung der zeitbehafteten SCOOP-Laufzeitumgebung wünschenswert, wie sie beispielsweise in [WPJB06] beschrieben wird.

Literaturverzeichnis

[ABL96] Abrial, JR ; Boerger, E. ; Langmaack, H.: *LNCS*. Bd. 1165: *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. Springer-Verlag, 1996

[Agh90] Agha, G.: Concurrent object-oriented programming. In: *Communications of the ACM* 33 (1990), Nr. 9, S. 125–141

[AH01a] Alfaro, L. de ; Henzinger, T.A.: Interface automata. In: *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, ACM Press, 2001, S. 109–120

[AH01b] Alfaro, L. de ; Henzinger, T.A.: Interface theories for component-based design. In: *EMSOFT: Embedded Software*. Springer, 2001 (Lecture Notes in Computer Science 2211), S. 148–165

[AH05] Alfaro, L. de ; Henzinger, T.A.: Interface-based design. In: Broy, M. (Hrsg.) ; Grünbauer, J. (Hrsg.) ; Harel, D. (Hrsg.) ; Hoare, C.A.R. (Hrsg.): *Engineering Theories of Software-intensive Systems*, Springer, 2005 (NATO Science Series: Mathematics, Physics, and Chemistry 195), S. 83–104

[Ahr95] Ahrendt, W.: *Von Prolog zur WAM. Verifikation der Prozedurübersetzung mit KIV*. Germany, Universität Karlsruhe, Diplomarbeit, 1995

[AHS02] Alfaro, L. de ; Henzinger, T.A. ; Stoelinga, M.: Timed interfaces. In: *EMSOFT: Embedded Software*. Springer, 2002 (Lecture Notes in Computer Science 2491), S. 108–122

[Ame87] America, P.: Pool-T: a parallel object-oriented language. In: *Computer Systems Series* (1987), S. 199–220

[Anl00] Anlauff, M.: XASM – An Extensible, Component-Based Abstract State Machines Language. In: Gurevich, Y. (Hrsg.) ; Kutter, P. (Hrsg.) ; Odersky, M. (Hrsg.) ; Thiele, L. (Hrsg.): *Abstract State Machines: Theory and Applications* Bd. 1912, Springer-Verlag, 2000 (LNCS), S. 69–90

[AS01] Arnout, Karine ; Simon, Raphaël: The .NET Contract Wizard: Adding Design by Contract to Languages Other than Eiffel. In: *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39).* Washington, DC, USA : IEEE Computer Society, 2001, 14

[BB92] Beierle, C. ; Börger, E.: Correctness Proof for the WAM With Types. In: Börger, E. (Hrsg.) ; Jäger, G. (Hrsg.) ; Kleine Büning, H. (Hrsg.) ; Richter, M. M. (Hrsg.): *Computer Science Logic* Bd. 626. Springer, 1992, S. 15–34

[BBD+96] Beierle, C. ; Börger, E. ; Durdanovic, I. ; Glässer, U. ; Riccobene, E.: Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In: Abrial, J.-R. (Hrsg.) ; Börger, E. (Hrsg.) ; Langmaack, H. (Hrsg.): *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control.* Springer, 1996 (LNCS 1165), S. 62–78

[BBF+01] Berard, B. ; Bidoit, M. ; Finkel, A. ; Laroussinie, F. ; Petit, A. ; Petrucci, L. ; Schnoebelen, P.: *Systems and Software Verification: Model-Checking Techniques and Tools. Vol. 7 of ISBN: 3-540-41523-8.* Springer, 2001

[BBF+06] BOLLELLA, G. ; BROSGOL, B. ; FURR, S. ; HARDIN, D. ; DIBBLE, P. ; GOSLING, J. ; TURNBULL, M. ; Belliardi, R.: *The real-time specification for Java.* Addison-Wesley, 2006

[BC89] Beck, C. ; Cunningham, W: A Laboratory For Teaching OO Thinking. In: *The Conference on Object-Oriented Programming Systems, Languages and Applications. ACM Press, October* (1989), S. 1–6

[BCR00a] Börger, E. ; Cavarra, A. ; Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In: Rus, Teodor (Hrsg.): *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings* Bd. 1816, Springer-Verlag, 2000 (LNCS), S. 293–308

[BCR00b] Börger, E. ; Cavarra, A. ; Riccobene, E.: Modeling the DYnamics of UML State Machines. In: Y. Gurevich and P. Kutter and M. Odersky and L. Thiele (Hrsg.): *Abstract State Machines: Theory and Applications* Bd. 1912, Springer-Verlag, 2000 (LNCS), S. 223–241

[BD90] Börger, E. ; Dässler, K.: Prolog: DIN Papers for Discussion / National Physical Laboratory. Middlesex, England, 1990 (58). – ISO/IEC JTCI SC22 WG17 Prolog Standardization Document

[BD95a] Börger, E. ; Del Castillo, G.: A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study). In: Werner, B. (Hrsg.): *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, 1995, S. 145–148

[BD95b] Bröhl, A.P. ; Dröschel, W.: *Das V-Modell: der Standard für die Softwareentwicklung mit Praxisleitfaden.* Oldenbourg, 1995

[BDGR94] Börger, E. ; Del Castillo, G. ; Glavan, P. ; Rosenzweig, D.: Towards a Mathematical Specification of the APE100 Architecture: the APESE Model. In: Pehrson, B. (Hrsg.) ; Simon, I. (Hrsg.): *IFIP 13th World Computer Congress* Bd. I: Technology/Foundations. Elsevier, Amsterdam, the Netherlands, 1994, S. 396–401

[BDL] Behrmann, Gerd ; David, Alexandre ; Larsen, Kim G. ; Department of Computer Science, Aalborg University, Denmark (Hrsg.): *A Tutorial on Uppaal.* Updated 17th November 2004. Department of Computer Science, Aalborg University, Denmark

[BEJV96] Binns, P. ; Englehart, M. ; Jackson, M. ; Vestal, S.: Domain Specific Software Architectures for Guidance, Navigation and Control. In: *Int. Journal of Software Engineering and Knowledge Engineering* 6 (1996), June, Nr. 2, S. 201–227

[BG94] Börger, E. ; Gläßer, U.: A Formal Specification of the PVM Architecture. In: Pehrson, B. (Hrsg.) ; Simon, I. (Hrsg.): *IFIP 13th World Computer Congress* Bd. I: Technology/Foundations. Elsevier, Amsterdam, the Netherlands, 1994, S. 402–409

[BG95] Börger, E. ; Gläßer, U.: Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In: Gurevich, Y. (Hrsg.) ; Börger, E. (Hrsg.): *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4).* University of Aarhus, Denmark, Juli 1995, S. 128–153

[BGM94] Börger, E. ; Gläßer, U. ; Müller, W.: The Semantics of Behavioral VHDL'93 Descriptions. In: *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94.* Los Alamitos, California : IEEE CS Press, 1994, S. 500–505

[BGM95] Börger, E. ; Glässer, U. ; Müller, W.: Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In: Delgado Kloos, C. (Hrsg.) ; Breuer, P. T. (Hrsg.): *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995, S. 107–139

[BGR95] Börger, E. ; Gurevich, Y. ; Rosenzweig, D.: The Bakery Algorithm: Yet Another Specification and Verification. In: Börger, E. (Hrsg.): *Specification and Validation Methods*. Oxford University Press, 1995, S. 231–243

[BHPR99] Börger, E. ; Hörger, B. ; Parnas, D. ; Rombach, D.: *Dagstuhl Seminar No. 99241: Requirements Capture, Documentation, and Validation*. Juni 1999

[BJPW99] Beugnard, A. ; Jézéquel, J.M. ; Plouzeau, N. ; Watkins, D.: Making Components Contract Aware. (1999)

[BKBI02] Bass, L. ; Klein, M. ; Bachmann, F. ; INST, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE E.: Quality Attribute Design Primitives and the Attribute Driven Design Method. In: *LECTURE NOTES IN COMPUTER SCIENCE* (2002), S. 169–186

[BKKS08] In: Beckers, J. ; Klünder, D. ; Kowalewski, S. ; Schlich, B.: *Lecture Notes in Computer Science*. Bd. 5238: *Direct Support for Model Checking Abstract State Machines by Utilizing Simulation*. Springer, 2008, S. 112–124

[BKPS07] Broy, M. ; Krüger, I.H. ; Pretschner, A. ; Salzmann, C.: Engineering Automotive Software. In: *Proceedings of the IEEE* 95 (2007), Nr. 2, S. 356–373

[Bla92] Blakley, B.: *A Smalltalk Evolving Algebra and its Uses*. Ann Arbor, Michigan, University of Michigan, Diss., 1992

[BM97a] Börger, E. ; Mazzanti, S.: A Practical Method for Rigorously Controllable Hardware Design. In: Bowen, J. P. (Hrsg.) ; Hinchey, M. B. (Hrsg.) ; Till, D. (Hrsg.): *ZUM'97: The Z Formal Specification Notation* Bd. 1212. Springer, 1997, S. 151–187

[BM97b] Börger, E. ; Mearelli, L.: Integrating ASMs into the Software Development Life Cycle. In: *Journal of Universal Computer Science* 3 (1997), Nr. 5, S. 603–665

[Boo06] Booch, G.: On architecture. In: *IEEE Software* 23 (2006), March / April, Nr. 2

[Bör90a] Börger, E.: A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In: Börger, E. (Hrsg.) ; Kleine Büning, H. (Hrsg.) ; Richter, M. M. (Hrsg.) ; Schönfeld, W. (Hrsg.): *CSL'89. 3rd Workshop on Computer Science Logic* Bd. 440, Springer, 1990 (LNCS), S. 36–64

[Bör90b] Börger, E.: A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In: Rovan, B. (Hrsg.): *Mathematical Foundations of Computer Science* Bd. 452. Springer, 1990, S. 1–14

[Bör92] Börger, E.: A Logical Operational Semantics for Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output. In: Moschovakis, Y. (Hrsg.): *Logic From Computer Science* Bd. 21. Springer, 1992, S. 17–50

[Bör94] Börger, E.: Logic Programming: The Evolving Algebra Approach. In: Pehrson, B. (Hrsg.) ; Simon, I. (Hrsg.): *IFIP 13th World Computer Congress* Bd. I: Technology/Foundations. Elsevier, Amsterdam, the Netherlands, 1994, S. 391–395

[Bör95] Börger, E.: Why Use Evolving Algebras for Hardware and Software Engineering? In: Bartosek, M. (Hrsg.) ; Staudek, J. (Hrsg.) ; Wiederman, J. (Hrsg.): *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics* Bd. 1012, Springer, 1995 (LNCS), S. 236–271

[Bör99] Börger, E.: High Level System Design and Analysis using Abstract State Machines. In: Hutter, D. (Hrsg.) ; Stephan, W. (Hrsg.) ; Traverso, P. (Hrsg.) ; Ullmann, M. (Hrsg.): *Current Trends in Applied Formal Methods (FM-Trends 98)*. Springer-Verlag, 1999 (LNCS 1641), S. 1–43

[Bör03] Börger, E.: The ASM Refinement Method. In: *Formal Aspects of Computing* 15 (2003), S. 237–257

[Bör04] Börger, E.: The ASM Ground Model Method as a Foundation of Requirements Engineering. In: Dershowitz, N. (Hrsg.): *Verification: Theory and Practice* Bd. 2772. Springer-Verlag, 2004, S. 146–161

[BP07] Brooke, P.J. ; Paige, R.F.: Exceptions in Concurrent Eiffel. In: *Journal of Object Technology, November/December* (2007)

[BPJ07] Brooke, P.J. ; Paige, R.F. ; Jacob, J.L.: A CSP model of Eiffel's SCOOP. In: *Formal Aspects of Computing* 19 (2007), Nr. 4, S. 487–512

[BPS00] Börger, E. ; Päppinghaus, P. ; Schmid, J.: Report on a Practical Application of ASMs in Software Design. In: Gurevich, Y. (Hrsg.) ; Kutter, P. (Hrsg.) ; Odersky, M. (Hrsg.) ; Thiele, L. (Hrsg.): *Abstract State Machines: Theory and Applications* Bd. 1912, Springer-Verlag, 2000 (LNCS), S. 361–366

[BR94a] Börger, E. ; Riccobene, E.: Logic + Control Revisited: An Abstract Interpreter for Gödel Programs. In: Levi, G. (Hrsg.): *Advances in Logic Programming Theory.* Oxford University Press, 1994

[BR94b] Börger, E. ; Rosenzweig, D.: The WAM – Definition and Compiler Correctness. In: Beierle, C. (Hrsg.) ; Plümer, L. (Hrsg.): *Logic Programming: Formal Methods and Practical Applications.* North-Holland, 1994 (Studies in Computer Science and Artificial Intelligence), Kapitel 2, S. 20–90

[BRS00] Borger, E. ; Riccobene, E. ; Schmid, J.: Capturing Requirements by Abstract State Machines: The Light Control Case Study. In: *Journal of Universal Computer Science* 6 (2000), Nr. 7, S. 597–620

[Bru95] Bruno, G.: *Model-based Software Engineering.* Chapman & Hall, 1995

[BS03a] Börger, E. ; Stärk, R.: *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003

[BS03b] Börger, E. ; Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003

[Bur98] Burns, A.: *Concurrency in Ada.* Cambridge University Press, 1998

[BW97] Burns, A. ; Wellings, A.J.: *Real-time Systems and Programming Languages.* Addison-Wesley Longman, 1997

[CAHS03] Chakrabarti, A. ; Alfaro, L. de ; Henzinger, T.A. ; Stoelinga, M.: Resource interfaces. In: *EMSOFT: Embedded Software.* Springer, 2003 (Lecture Notes in Computer Science 2855), S. 117–133

[Car93] Caromel, D.: Toward a method of object-oriented concurrent programming: Concurrent object-oriented programming. In: *Communications of the ACM* 36 (1993), Nr. 9, S. 90–102

[CBB+03] Clements, P. ; Bachmann, F. ; Bass, L. ; Garlan, D. ; Ivers, J. ; Little, R. ; Nord, R. ; Stafford, J. ; Longman (Hrsg.): *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2003

[CES86] CLARKE, EM ; EMERSON, EA ; SISTLA, AP: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In: *ACM Transactions on Programming Languages and Systems* 8 (1986), Nr. 2, S. 244–263

[CGN+05] Campbell, C. ; Grieskamp, W. ; Nachmanson, L. ; Schulte, W. ; Tillmann, N. ; Veanes, M.: Testing Concurrent Object-Oriented Systems with Spec Explorer. In: *LECTURE NOTES IN COMPUTER SCIENCE* 3582 (2005), S. 542

[CGP99] Clarke, E. M. ; Grumberg, O. ; Peled, D. A.: *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999

[CHS00] Compton, K. ; Huggins, J. ; Shen, W.: A Semantic Model for the State Machine in the Unified Modeling Language. In: G. Reggio and A. Knapp and B. Rumpe and B. Selic and R. Wieringa (Hrsg.) ; Ludwig-Maximilians-Universität München (Veranst.): *Proceedings of UML 2000 Workshop* Ludwig-Maximilians-Universität München, 2000 (Institut für Informatik Bericht 0006), S. 25–31

[CNPC06] Capilla, Rafael ; Nava, Francisco ; Pérez, Sandra ; C. Dueñas, J.: A web-based tool for managing architectural design decisions. In: *SIGSOFT Softw. Eng. Notes* 31 (2006), Nr. 5, S. 4. – DOI http://doi.acm.org/10.1145/1163514.1178644. – ISSN 0163–5948

[Col01] Collet, P.: Functional and Non-Functional Contracts Support for Component-Oriented Programming. In: *First OOPSLA Workshop on Language Mechanisms for Programming Software Components, OOPSLA 2001* (2001)

[CP02] Castillo, GD ; Pappinghaus, P.: Designing software for internet telephony: experiences in an industrial development process. In: *Theory and Applications of Abstract State Machines, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science* (2002)

[CR06] Clarke, Lori A. ; Rosenblum, David S.: A historical perspective on runtime assertion checking in software development. In: *SIGSOFT Softw. Eng. Notes* 31 (2006), Nr. 3, S. 25–37. – DOI http://doi.acm.org/10.1145/1127878.1127900. – ISSN 0163–5948

[CW02] Compton, M. ; Walker, R.: A run-time system for SCOOP. In: *Journal of Object Technology* 1 (2002), Nr. 3, S. 119–157

[DC01] Del Castillo, G.: *The ASM-Workbench. A tool environment for computer aided analysis and validation of ASM models*, PhD thesis, University of Paderborn, Diss., 2001

[Del99] Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines. In: Hutter, D. (Hrsg.) ; Stephan, W. (Hrsg.) ; Traverso, P. (Hrsg.) ; Ullmann, M. (Hrsg.): *Applied Formal Methods — FM-Trends 98* Bd. 1641, Springer-Verlag, 1999 (LNCS), S. 311–325

[DFG83] Dijkstra, E.W. ; Feijen, W.H.J. ; Gasteren, A.J.M.: Derivation of a Termination Detection Algorithm for Distributed Computations. In: *Information Processing Letters* 16 (1983), Nr. 5, S. 217–219

[DGVZ98] Dold, Axel ; Gaul, Thilo ; Vialard, Vincent ; Zimmermann, Wolf: ASM-Based Mechanized Verification of Compiler Back-Ends / University of Karlsruhe, Institute for Program Structures and Data Organisation. 1998. – Forschungsbericht

[DH98] Del Castillo, G. ; Hardt, W.: Towards a Unified Analysis Methodology of HW/SW Systems based on Abstract State Machines: Modelling of Instruction Sets. In: *Proceedings of the GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, Paderborn, 1998

[Dij75] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. In: *Communications of the ACM* 18 (1975), Nr. 8, S. 453–457

[DN02] Dobrica, Liliana ; Niemelä, Eila: A Survey on Software Architecture Analysis Methods. In: *IEEE Transactions on Software Engineering* 28 (2002), Juli, Nr. 7, S. 638–653

[Dol98] Dold, A.: A Formal Representation of Abstract State Machines Using PVS / Universität Ulm. 1998 (Ulm/6.2). – Verifix Technical Report

[DW00] Del Castillo, G. ; Winter, K.: Model Checking Support for the ASM High-Level Language. In: Graf, S. (Hrsg.) ; Schwartzbach, M. (Hrsg.): *Proceedings of the 6th International Conference TACAS 2000* Bd. 1785, Springer-Verlag, 2000 (LNCS), S. 331–346

[EC81a] Emerson, E. A. ; Clarke, E. M.: Characterizing properties of parallel programs using fixpoints. In: *7th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science* 85 (1981), S. 169–181

[EC81b] Emerson, EA ; Clarke, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Workshop on Logic of Programs. LNCS* 131 (1981)

[EGG+01] Eschbach, R. ; Glässer, U. ; Gotzhein, R. ; Löwis, M. von ; Prinz, A.: Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models. In: *Journal of Universal Computer Science* 7 (2001), Nr. 11, S. 1024–1049

[EK07] Erl, H. P. ; Kirstan, S.: Kosten- / Nutzenanalyse der modellbasierten Softwareentwicklung im Automobil / Arthur D. Little. Leopoldstr. 11a, 80802 München, Januar 2007. – Studie

[Esc99] Eschbach, R.: A Termination Detection Algorithm: Specification and Verification. In: Wing, J. (Hrsg.) ; Woodcock, J. (Hrsg.) ; Davies, J. (Hrsg.): *Proceedings of FM'99*, Springer-Verlag, 1999 (LNCS 1709), S. 1720–1737

[Far07a] Farahbod, Roozbeh ; www.coreasm.org (Hrsg.): *CoreASM Language User Manual*. Engine version 1.0.3. Software Technology Lab, School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, BC, Canada V5A 1S6: www.coreasm.org, 2007

[Far07b] Farahbod, Roozbeh: *Executing ASM Specifications with CoreASM, Part 1: Introduction to the CoreASM engine*. Presentation, Lipari Summer School, 2007

[Far07c] Farahbod, Roozbeh: *Executing ASM Specifications with CoreASM, Part 2: Extensibility and Applications*. Presentation, Lipari Summer School, 2007

[Fei06] Feiler, P.: *AADL and Simulink*. Short paper draft available at `http://la.sei.cmu.edu/aadl-wiki/index.php/AADL_and_Simulink`., Sept. 2006

[FGG05] Farahbod, Roozbeh ; Gervasi, Vincenzo ; Glässer, Uwe: Design and Specification of the CoreASM Execution Engine / Simon Fraser University. 2005 (SFU-CMPT-TR-2005-02). – Forschungsbericht

[FGG07] Farahbod, Roozbeh ; Gervasi, Vincenzo ; Glässer, Uwe: CoreASM: An Extensible ASM Execution Engine. In: *Fundamenta Informaticae* 77 (2007), S. 71–103

[FGGM06] Farahbod, R. ; Gervasi, V. ; Glässer, U. ; Memon, M.: Design and Specification of the CoreASM Execution Engine, Part 1: The Kernel / Simon Fraser University. 2006 (SFU-CMPT-TR-2006-09). – Forschungsbericht

[FGM07] Farahbod, R. ; Glässer, U. ; Ma, G.: Model Checking CoreASM Specifications. In: *ASM 2007* Proc. 14th Int. Workshop on Abstract State Machines, June 2007, Grimstad, Norway, 2007

[Flo67] Floyd, R.W.: Assigning meanings to programs. In: *Mathematical Aspects of Computer Science* 19 (1967), Nr. 19-32, S. 1

[FMS00] Fernandes, J.M. ; Machado, R.J. ; Santos, H.D.: Modeling industrial embedded systems with UML. In: *Proceedings of the eighth international workshop on Hardware/software codesign* ACM New York, NY, USA, 2000, S. 18–22

[FOP04] Fuks, O. ; Ostroff, J.S. ; Paige, R.F.: SECG: The SCOOP-to-Eiffel Code Generator. In: *Journal of Object Technology* 3 (2004), Nr. 10, S. 143–160

[Gal95] Gallmeister, B.O.: *POSIX. 4: programming for the real world*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1995

[Gao04] Gao, Y.: *Multi-VIEW Consistency Checking OF BON Software Description Diagrams*, York University, Diss., 2004

[Gar05] Garfinkel, S.: History's worst software bugs. In: *Wired News, Nov* (2005)

[Gau95] Gaul, T.: An Abstract State Machine specification of the DEC-Alpha Processor Family / University of Karlsruhe. 1995 ([Verifix/UKA/4]). – Verifix Working Paper

[GH96] Gurevich, Y. ; Huggins, J.: The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In: *Proceedings of CSL'95 (Computer Science Logic)* Bd. 1092, Springer, 1996 (LNCS), S. 266–290

[GKS97] Giese, M. ; Kempe, D. ; Schönegge, A.: KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur / Universität Karlsruhe. 1997 (16/97). – Interner Bericht

[GM95] Gurevich, Y. ; Mani, R.: Group Membership Protocol: Specification and Verification. In: Börger, E. (Hrsg.): *Specification and Validation Methods*. Oxford University Press, 1995, S. 295–328

[GR00] Gargantini, A. ; Riccobene, E.: Encoding Abstract State Machines in PVS. In: Gurevich, Y. (Hrsg.) ; Kutter, P. (Hrsg.) ; Odersky, M. (Hrsg.) ; Thiele, L. (Hrsg.): *Abstract State Machines: Theory and Applications* Bd. 1912, Springer-Verlag, 2000 (LNCS), S. 303–322

[GRR03] Gargantini, Angelo ; Riccobene, Elvinia ; Rinzivillo, Salvatore: Using Spin to Generate Tests from ASM Specifications. In: Börger, Egon (Hrsg.) ; Gargantini, Angelo (Hrsg.) ; Riccobene, Elvinia (Hrsg.): *Abstract State Machines* Bd. 2589, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–00624–9, S. 263–277

[GRS05] Gurevich, Y. ; Rossman, B. ; Schulte, W.: Semantic essence of AsmL. In: *Theoretical Computer Science* 343 (2005), Nr. 3, S. 370–412

[GTW03] Gawanmeh, A. ; Tahar, S. ; Winter, K.: Interfacing ASMs with the MDG Tool. In: E. Borger, A. G. (Hrsg.) ; Recobene, E. (Hrsg.): *Abstract State Machines - Advances in Theory and Applications*, Springer Verlag, 2003, 278-292

[Gur84] Gurevich, Y.: Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs / EECS Department, University of Michigan. 1984 (CRL-TR-36-84). – Technical Report

[Gur85] Gurevich, Y.: A New Thesis. In: *Abstracts, American Mathematical Society* (1985), August, S. 317

[Gur88a] Gurevich, Y.: Algorithms in the World of Bounded Resources. In: Herken, R. (Hrsg.): *The Universal Turing Machine – A Half-Century Story.* Oxford University Press, 1988, S. 407–416

[Gur88b] Gurevich, Y.: Logic and the Challenge of Computer Science. In: Börger, E. (Hrsg.): *Current Trends in Theoretical Computer Science.* Computer Science Press, 1988, S. 1–57

[Gur91] Gurevich, Y.: Evolving Algebras. A Tutorial Introduction. In: *Bulletin of EATCS* 43 (1991), S. 264–284

[Gur94] Gurevich, Y.: Evolving Algebras. In: Pehrson, B. (Hrsg.) ; Simon, I. (Hrsg.): *IFIP 13th World Computer Congress* Bd. I: Technology/Foundations. Elsevier, Amsterdam, the Netherlands, 1994, S. 423–427

[Gur95] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Börger, E. (Hrsg.): *Specification and Validation Methods.* Oxford University Press, 1995, S. 9–36

[Gur00] Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms. In: *ACM Transactions on Computational Logic* 1 (2000), Juli, Nr. 1, S. 77–111

[HBS+06] Heinecke, H. ; Bielefeld, J. ; Schnelle, K.P. ; Maldener, N. ; Fennel, H. ; Weis, O. ; Weber, T. ; Ruh, J. ; Lundh, L. ; Sandén, T. u. a.: AUTOSAR–Current results and preparations for exploitation. In: *7th EUROFORUM conferenceSoftware in the vehicle* (2006)

[Hel97] Heljanko, K.: Model checking the branching time temporal logic CTL / Helsinki University of Technology, Digital Systems Laboratory. Espoo, Finland, May 1997 (A45). – Research Report

[Hen98] Henriksson, R.: *Scheduling garbage collection in embedded systems.* Schweden, Department of Computer Science, Lund Institute of Technology, Diss., 1998

[HM96] Heitmeyer, C. ; Mandrioli, D.: *Trends in Software.* Bd. 5: *Formal Methods for Real-Time Computing.* John Wiley & Son Ltd, 1996

[HNS00] Hofmeister, C. ; Nord, R. ; Soni, D.: *Applied Software Architecture.* Addison-Wesley, 2000

[Hoa69] Hoare, C. A. R.: An axiomatic basis for computer programming. In: *Commun. ACM* 12 (1969), Nr. 10, 576–580. – DOI http://doi.acm.org/10.1145/363235.363259. – ISSN 0001–0782

[Hol97] Holzmann, Gerard J.: *The model checker SPIN.* Addison-Wesley, 1997

[HS06] Henzinger, T.A. ; Sifakis, J.: The embedded systems design challenge. In: *Proceedings of the 14 thInternational Symposium on Formal Methods (FM'06)* (2006), S. 1–15

[Hug95] Huggins, J.: Kermit: Specification and Verification. In: Börger, E. (Hrsg.): *Specification and Validation Methods.* Oxford University Press, 1995, S. 247–293

[IEC05] IEC: *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 0: Functional safety and IEC 61508.* IEC/TR 61508-0, Januar 2005

[IEE00] IEEE: *Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE Standard No. 1471-2000, 2000

[ISO01] ISO / IEC: *Software Engineering – Product Quality – Part1: Quality Model.* ISO Standard 9126-1, 2001

[Jal94] Jalloul, G.: *Concurrent object-oriented systems: a disciplined approach*, University of Technology, Sydney, Australia, Diss., 1994

[JP91] Jalloul, G. ; Potter, J.: Models for Concurrent Eiffel. In: *Technology of Object-oriented Languages and Systems: Proceedings of the Sixth International Conference, Tools, Sydney, 1992* (1991)

[Kap90] Kappel, A.M.: Implementation of Dynamic Algebras with an Application to Prolog. In: *Diplom thesis, Computer Science Dept., Universitat Dortmund, Germany* (1990)

[Kap93] Kappel, A. M.: Executable Specifications Based on Dynamic Algebras. In: Voronkov, A. (Hrsg.): *Logic Programming and Automated Reasoning* Bd. 698. Springer, 1993, S. 229–240

[Kar05] Kardos, M.: An approach to model checking asml specifications. In: *Proceedings of the 12th International Workshop on Abstract State Machines.*, 2005

[KB93] Karaorman, M. ; Bruno, J.: Introducing concurrency to a sequential language. In: *Communications of the ACM* 36 (1993), Nr. 9, S. 103–115

[KK07] Klünder, D. ; Kowalewski, S.: Scenario-Based Evaluation of Abstract State Machines. In: *Proceedings of the ASM'07 - The 14th International ASM Workshop*, 2007

[KKB+98] Kazman, R. ; Klein, M. ; Barbacci, M. ; Longstaff, T. ; Lipson, H. ; Carriere, J.: The architecture tradeoff analysis method. In: *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*, 1998, S. 68–78

[KKK08] Klünder, D. ; Kameni, D. ; Kowalewski, S.: Nahtloser Entwurf sicherheitskritischer, eingebetteter Softwaresysteme mit abstrakten Zustandsmaschinen und Business Object Notation. In: Jumar, U. (Hrsg.) ; Schnieder, E. (Hrsg.) ; Diedrich, C. (Hrsg.) ; Institut für Automation und Kommunikation e. V. Magdeburg (ifak) (Veranst.): *Entwurf komplexer Automatisierungssysteme (EKA 2008)* Institut für Automation und Kommunikation e. V. Magdeburg (ifak), 2008. – ISBN 978–3–940961–01–3

[KLK08] Klünder, D. ; Li, J. ; Kowalewski, S.: Design-by-Contract for Embedded Systems. In: *International Symposium on Software Variability: a Programmers' Perspective (SVPP)*, 2008

[Klü06a] Klünder, Daniel: An Approach to Resolving Contradicitions in Software Architecture Design. In: Hofmeister, Christine (Hrsg.) ; Crnkovic, Ivica (Hrsg.) ; Reussner, Ralf (Hrsg.) ; Becker, Steffen (Hrsg.): *Perspectives in Software Architecture Quality. Short papers of the 2nd International Conference on the Quality of Software Architectures (QoSA 2006)*, Universität Karlsruhe, Fakultät für Informatik, June 27-29 2006. – ISSN 1432 – 7864, 59-63

[Klü06b] Klünder, Daniel: TRIZ for Software Architecture. In: Duflou, Joost R. (Hrsg.) ; D'hondt, Joris (Hrsg.) ; Devoldere, Tom (Hrsg.) ; Dewulf, Simon (Hrsg.) ; Dekeyser, Nele (Hrsg.): *Proceedings of the ETRIA TRIZ Future Conference 2006: Creativity, Quality and Efficiency - Building an Innovation Culture* Bd. 1: Scientific Contributions, 2006. – ISBN 90–77071–05–9, S. 93–98

[Klü08] Klünder, D.: Modellierung und Analyse von Fahrzeugsoftware mit Abstract State Machines. In: Kühne, T. (Hrsg.) ; Reisig, W. (Hrsg.) ; Steinmann, F. (Hrsg.) ; Gesellschaft für Informatik (Veranst.): *Modellierung 2008* Gesellschaft für Informatik, 2008 (Lecture Notes in Informatics). – ISBN 978–3–88579–221–5, S. 225–228

[KOH+94] Kuskin, J. ; Ofelt, D. ; Heinrich, M. ; Heinlein, J. ; Simoni, R. ; Gharachorloo, K. ; Chapin, J. ; Nakahira, D. ; Baxter, J. ; Horowitz, M. u. a.: The Stanford FLASH multiprocessor. In: *Computer Architecture, 1994. Proceedings the 21st Annual International Symposium on* (1994), S. 302–313

[KP98] Kramer, R. ; Partners, C.T.: iContract-the Java TM design by Contract TM tool. In: *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings* (1998), S. 295–307

[Kru95] Kruchten, P. B.: The 4+1 View Model of architecture. In: *IEEE Software* 12 (1995), November, Nr. 6, S. 42–50

[L+96] Lions, J.L. u. a.: Ariane 5 Flight 501 Failure. In: *Report by the Inquiry Board, Paris* 19 (1996)

[Lea00] Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Professional, 2000

[Lee00] Lee, Edward A.: What's Ahead for Embedded Software? In: *Computer* 33 (2000), Nr. 9, S. 18–26. – ISSN 0018–9162

[Lem01] Lemieux, J.: *Programming in the OSEK/VDX Environment*. CMP, 2001

[LG86] Liskov, B. ; Guttag, J.: *Abstraction and specification in program development.* The MIT Press, 1986

[LH83] Lieberman, H. ; Hewitt, C.: A real-time garbage collector based on the lifetimes of objects. In: *Communications of the ACM* 26 (1983), Nr. 6, S. 419–429

[LHFB02] Lederer, D. ; Heling, G. ; Fetzer, J. ; Beck, T.: Der Schlüssel zum Erfolg: Durchgängige Systems-Engineering-Prozesse - eine vordringliche Aufgabe. In: *Elektronik Automotive* (2002), Nr. 3

[Lin95] Lindner, T.: Task Description. In: *Lecture Notes In Computer Science; Vol. 891* (1995), S. 7–19

[LLW05] Li, Shuyu ; Li, XiaoJiang ; Wu, Jian: Components and Contracts for Embedded Software. In: *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems.* Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2308–0, 19–24

[LPY97] Larsen, K.G. ; Pettersson, P. ; Yi, W.: Uppaal in a nutshell. In: *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1997), Nr. 1, S. 134–152

[LS80] Lientz, B.P. ; Swanson, E.B.: *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, 1980

[Lun05] Lunze, J.: *Regelungstechnik 1.* Springer, 2005

[LVH85] Luckham, DC ; Von Henke, FW: An Overview of Anna, a Specification Language for Ada. In: *Software, IEEE* 2 (1985), Nr. 2, S. 9–22

[Ma07] Ma, George: *Model Checking Support for CoreASM: Model Checking Distributed Abstract State Machines Using SPIN*, Simon Fraser University, Diplomarbeit, 2007

[Mar02] Martin, G.: UML for Embedded Systems Specification and Design: Motivation and Overview. In: *Proceedings of the conference on Design, automation and test in Europe* IEEE Computer Society Washington, DC, USA, 2002, S. 773

[McM92] McMillan, K.L.: The SMV system / CMU. 1992 (Technical Report CMU-CS-92-131). – Forschungsbericht

[Mea97] Mearelli, L.: Refining an ASM Specification of the Production Cell to C^{++} Code. In: *Journal of Universal Computer Science* 3 (1997), Nr. 5, S. 666–688

[Mey88] Meyer, B.: Eiffel: a language and environment for software engineering. In: *J. Syst. Softw.* 8 (1988), Nr. 3, S. 199–246. – DOI http://dx.doi.org/10.1016/0164–1212(88)90022–2. – ISSN 0164–1212

[Mey90] Meyer, B.: Sequential and concurrent object-oriented programming. In: *Proceedings of TOOLS* 90 (1990)

[Mey92a] Meyer, B.: *Eiffel: The Language.* Prentice Hall, 1992

[Mey92b] Meyer, Bertrand: Applying "Design by Contract". In: *Computer* 25 (1992), Nr. 10, S. 40–51. – DOI http://dx.doi.org/10.1109/2.161279. – ISSN 0018–9162

[Mey93] Meyer, B.: Systematic concurrent object-oriented programming. In: *Communications of the ACM* 36 (1993), Nr. 9, S. 56–80

[Mey97] Meyer, B.: *Object-oriented Software Construction.* 2. Edition. Prentice Hall, 1997

[Mey02] Meyer, B.: *Design by Contract.* Prentice Hall, 2002

[MJ97] Meyer, B. ; Jézéquel, JM: Design by Contract: The Lessons of Ariane. In: *IEEE Computer* 30 (1997), Nr. 2, S. 129–130

[MP95] Manna, Z. ; Pnueli, A.: *Temporal Verification of Reactive Systems: Safety.* Springer, 1995

[MRW+77] McCall, J.A. ; Richards, P.K. ; Walters, G.F. ; States, United ; Division, Electronic S. ; Force, A. ; Center, Rome Air D. ; Command, Systems: *Factors in Software Quality.* NTIS, 1977

[MY93] Matsuoka, S. ; Yonezawa, A.: Analysis of inheritance anomaly in object-oriented concurrent programming languages. (1993)

[NA03] Nienaltowski, P. ; Arslan, V.: SCOOPLI: a library for concurrent object-oriented programming on .NET. In: *Journal of .NET Technologies* (2003)

[Nag90] Nagl, M.: *Softwaretechnik: Methodisches Programmieren im grossen.* Springer, 1990

[NAM03] Nienaltowski, P. ; Arslan, V. ; Meyer, B.: Concurrent object-oriented programming on .NET. In: *Software, IEE Proceedings-[see also Software Engineering, IEE Proceedings]* Bd. 150, 2003, S. 308–314

[OBL07] Ouimet, M. ; Berteau, G. ; Lundqvist, K.: Modeling an Electronic Throttle Controller Using the Timed Abstract State Machine Language and Toolset. In: *LECTURE NOTES IN COMPUTER SCIENCE* 4364 (2007), S. 32

[OL07a] Ouimet, M. ; Lundqvist, K.: Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. In: *Electronic Notes in Theoretical Computer Science* 190 (2007), Nr. 2, S. 85–97

[OL07b] Ouimet, M. ; Lundqvist, K.: The timed abstract state machine language: Abstract state machines for real-time system engineering. In: *Proceedings ASM*, 2007

[OL07c] Ouimet, Martin ; Lundqvist, Kristina: Verifying Execution Time using the TASM Toolset and UPPAAL / MIT, Embedded Systems Laboratory. 2007 (ESL-TIK-00212). – Forschungsbericht

[OW04] Oaks, S. ; Wong, H.: *Java Threads*. O'Reilly Media, Inc., 2004

[Par74] Parnas, D. L.: On a Buzzword: Hierarchical Structure. In: *Proc. of the IFIP Congress* (1974), S. 336–339

[PDIJG04] Prof. Dr.-Ing. Jürgen Gausemeier, Prof. Dr. rer. nat. Wilhelm S. Priv.-Doz. Dr. rer. nat. Uwe Glässer G. Priv.-Doz. Dr. rer. nat. Uwe Glässer: Abschlussbericht für die Phase 3 des Forschungsvorhabens: Integrative Spezifikation von verteilten Leitsystemen der flexibel automatisierten Fertigung (ISILEIT) / Universität Paderborn. 2004 (GA 456/7-3 ISILEIT). – Abschlussbericht

[Pes97] Pescio, C.: The View from the Eiffel Tower Bertrand Meyer-scholar, entrepreneur, and business manager-offers insight into his clear and practical approach to software development. In: *SOFTWARE DEVELOPMENT-SAN FRANCISCO-* 5 (1997), S. 51–56

[Phi00] Philippsen, M.: A survey of concurrent object-oriented languages. In: *Concurrency - Practice and Experience* 12 (2000), Nr. 10, S. 917–980

[PHL+77] Popek, GJ ; Horning, JJ ; Lampson, BW ; Mitchell, JG ; London, RL: Notes on the design of Euclid. In: *Proceedings of an ACM conference*

on Language design for reliable software table of contents (1977), S. 11–18

[Pnu81] Pnueli, A.: A temporal logic of concurrent programs. In: *Theoretical Computer Science* 13 (1981), Nr. 1, S. 45–60

[PO98] Paige, R.F. ; Ostroff, J.S.: From Z to BON/Eiffel. In: *13th IEEE international Conference on Automated Software Engineering (ASE)*, 1998

[PW92] Perry, Dewayne E. ; Wolf, Alexander L.: Foundations for the study of software architecture. In: *SIGSOFT Softw. Eng. Notes* 17 (1992), Nr. 4, S. 40–52. – DOI http://doi.acm.org/10.1145/141874.141884. – ISSN 0163–5948

[QS82] Queille, J. ; Sifakis, J.: Specification and verification of concurrent systemsin CESAR. In: *Fifth International Symposium on Programming, Lecture Notes in Computer Science* 137 (1982), S. 337–351

[Rus00] Rust, H.: Hybrid Abstract State Machines: Using the Hyperreals for Describing Continuous Changes in a Discrete Notation. In: *Abstract State Machines–ASM* (2000), S. 341–356

[Rus03] Rust, H.: A Non-standard Approach to Operational Semantics for Timed Systems. In: *LECTURE NOTES IN COMPUTER SCIENCE* (2003), S. 423–424

[SA97] Schellhorn, G. ; Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. In: *Journal of Universal Computer Science* 3 (1997), Nr. 4, S. 377–413

[Sch99] Schellhorn, Gerhard: *Verifikation abstrakter Zustandsmaschinen*, Universität Ulm, Diss., 1999

[Sch01a] Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation. In: *Journal of Universal Computer Science* 7 (2001), Nr. 11, S. 952–979

[Sch01b] Schmid, J.: Compiling Abstract State Machines to C++. In: *Journal of Universal Computer Science* 7 (2001), Nr. 11, S. 1068–1087

[Sch01c] Schmid, J.: Compiling Abstract State Machines to C++ (Extended Abstract). In: Moreno-Díaz, R. (Hrsg.) ; Quesada-Arencibia, A. (Hrsg.) ; Universidad de Las Palmas de Gran Canaria (Veranst.): *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*. Canary Islands, Spain, Februar 2001, S. 298–300

[Sch02] Schmid, J.: *Refinement and Implementation Techniques for Abstract State Machines*, PhD thesis, University of Ulm, Germany, 2002, Diss., 2002

[Sch04] Schneider, K.: *Verification of Reactive Systems: Formal Methods and Algorithms.* Springer, 2004

[Sch08] Schlich, Bastian: Model Checking of Software for Microcontrollers / RWTH Aachen University. Version: June 2008. `http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf`. 2008 (AIB-2008-14). – Dissertation Thesis. – ISSN 0935–3232

[SG96] Shaw, M. ; Garlan, D.: *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996

[SGG+05] Scandurra, P. ; Gargantini, A. ; Genovese, C. ; Genovese, T. ; Riccobene, E.: A Concrete Syntax derived from the Abstract State Machine Metamodel. In: *12th International Workshop on Abstract State Machines (ASM'05)* (2005), S. 8–11

[SK06] Schlich, Bastian ; Kowalewski, Stefan: [mc]square: A model checker for microcontroller code. In: *On-Site Proc. 2nd Int'l Symp. Leveraging Applications of Formal Methods, Verification and Validation (IEEE-ISoLA 2006)*, IEEE, 2006. – To appear

[SK07] Schlich, Bastian ; Kowalewski, Stefan: An extendable architecture for model checking hardware-specific automotive microcontroller code. In: *Proc. 6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, GZVB, 2007. – ISBN 978–3–937655–09–3, S. 202–212

[Spi92] Spivey, JM: *The Z notation: a reference manual.* Prentice Hall, 1992

[SSB01] Stärk, R. ; Schmid, J. ; Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001

[SSK07] Schlich, B. ; Salewski, F. ; Kowalewski, S.: Applying Model Checking to an Automotive Microcontroller Application. In: *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)*, IEEE, 2007. – ISBN 1–4244–0840–7, S. 209–216

[SZ06] Schäuffele, J. ; Zurawka, T.: *Automotive Software Engineering.* Vieweg Verlag, 2006

[Tan06] Tang, Calvin Kai F.: *Model Checking Abstract State Machines with Answer Set Programming*, Simon Fraser University, Diplomarbeit, 2006

[TT05] Tang, Calvin Kai F. ; Ternovska, Eugenia: Model Checking Abstract State Machines with Answer Set Programming. In: *Abstract State Machines*, 2005, S. 397–416

[Tur36] Turing, A.M.: On Computable Numbers: With an Application to the Entscheidungsproblem. (1936)

[UBHB01] Urting, D. ; Baelen, S. v. ; Holvoet, T. ; Berbers, Y.: Embedded software development: Components and contracts. In: Gonzalez, T. F. (Hrsg.): *IASTED international conference: parallel and distributed computing and systems (IASTED-PDCS)*, 2001, S. 685–690

[Vas07] Vasilyev, P.: Simulator-Model Checker for Reactive Real-Time Abstract State Machines. In: *ASM 2007* Proc. 14th Int. Workshop on Abstract State Machines, June 2007, Grimstad, Norway, 2007

[VL93] Vergauwen, B. ; Lewi, J.: A linear local model checking algorithm for CTL. In: *Proc. 4th Int. Conf. Concurrency Theory (CONCUR)* Bd. 715, Springer, 1993 (LNCS), S. 447–461. – ISBN 3-540-57208-2

[Wal98] *Kapitel* 10 Business Object Notation. In: Waldén, K.: *Handbook of Object Technology*. CRC Press, 1998

[War83] Warren, D.H.D.: An Abstract Prolog Instruction Set, Technical note 309 / Artificial Intelligence Center, SRI International. 1983. – Forschungsbericht

[Win97] Winter, K.: Model Checking for Abstract State Machines. In: *Journal of Universal Computer Science* 3 (1997), Nr. 5, S. 689–701

[Win00] Winter, K.: Towards a Methodology for Model Checking ASM: Lessons Learned from the FLASH Case Study. In: Gurevich, Y. (Hrsg.) ; Kutter, P. (Hrsg.) ; Odersky, M. (Hrsg.) ; Thiele, L. (Hrsg.): *Abstract State Machines: Theory and Applications* Bd. 1912, Springer-Verlag, 2000 (LNCS), S. 341–360

[Win01] Winter, K.: *Model Checking Abstract State Machines*, Technical University of Berlin, Germany, Diss., 2001

[WKH92] Wyatt, BB ; Kavi, K. ; Hufnagel, S.: Parallelism in object-oriented languages: a survey. In: *Software, IEEE* 9 (1992), Nr. 6, S. 56–66

[WLS76] Wulf, W.A. ; London, R.L. ; Shaw, M.: An Introduction to the Construction and Verification of Alphard Programs. In: *IEEE Transactions on Software Engineering* 2 (1976), Nr. 4, S. 253–265

[WN95] Waldén, K. ; Nerson, J.-M.: *Seamless Object-Oriented Sofware Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, 1995

[WPJB06] Wellings, A. ; Paige, R. ; Jacob, J. ; Burns, A.: *RECOOP: Real-Time Concurrent Programming in Eiffel*. Juni 2006. – Keynote at The First Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)

[ZCC97] Zendra, O. ; Colnet, D. ; Collin, S.: Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler. In: *ACM SIGPLAN Notices* 32 (1997), Nr. 10, S. 125–141

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications
2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code
2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter:: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2009-03	Alexander Nyßen: Model-Based Construction of Embedded & Real-Time Software - A Methodology for Small Devices

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

Lebenslauf

Name	Daniel Klünder
Geburtsdatum	29.03.1978
Geburtsort	Boppard / Rhein-Hunsrück

2004 – 2008	Wissenschaftlicher Angestellter am Lehrstuhl Informatik 11 an der RWTH Aachen University
1998 – 2004	Studium der Elektrotechnik und Informationstechnik an der RWTH Aachen University
1997 – 1998	Zivildienst
1988 – 1997	Bischöfliches Cusanus Gymnasium Koblenz
1984 – 1988	Grundschule Schenkendorf, Koblenz