Automatisiertes Testen asynchroner nichtdeterministischer Systeme mit Daten

vorgelegt von
Diplom-Informatiker
Dirk Seifert

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

Promotionsausschuss:

Vorsitzender:	
Berichter:	Prof. Dr. Stefan Jähnichen
Berichter:	Prof. Dr. Ina Schieferdecker
Berichter:	PD Dr. Thomas Santen

Tag der wissenschaftlichen Aussprache: XX. Juli 2007

Berlin 2007
D 83

Zusammenfassung

Eingebettete Systeme setzen sich aus Hardware- und Softwarekomponenten zusammen, die asynchron miteinander kommunizieren und nichtdeterministisches Verhalten aufweisen können. Dies macht eine umfassende und systematische Prüfung mit manuellen Techniken nahezu unmöglich. Auch automatisierte Prüftechniken können mit Asynchronität und Nichtdeterminismus bislang nur unzureichend umgehen. Der breite Einsatz von eingebetteten Systemen im täglichen Leben und das blinde Vertrauen in deren Funktionsfähigkeit erfordern jedoch eine zuverlässige und umfassende Qualitätssicherung.

Die vorliegende Arbeit beschäftigt sich mit dem automatisierten Testen asynchroner nichtdeterministischer Systeme mit Daten. Zur Beschreibung des reaktiven Verhaltens von Komponenten reaktiver technischer Systeme werden Zustandsmaschinen der *Unified Modeling Language* in reduzierter Form verwendet. Ausgehend von einer solchen Zustandsmaschine werden automatisiert Testfälle für einen Konformitätstest abgeleitet und gegen das System unter Test ausgeführt. Das korrekte, mögliche Verhalten wird für ausgewählte Eingaben »a priori« berechnet und in einem Testfall gespeichert. Die Speicherung ermöglicht eine wiederholte Ausführung der Testfälle nach Änderungen in der Entwicklung.

Der Hauptbeitrag der Arbeit liegt einerseits in der vollständigen Formalisierung der reduzierten Zustandsmaschinen, die die wesentlichen strukturellen Merkmale umfassen und komplex strukturierte Daten beinhalten, und andererseits in der Entwicklung eines praktikablen Ansatzes zum Testen reaktiver technischer Systeme mit Daten, deren Verhalten durch solche Zustandsmaschinen beschrieben werden kann. Die asynchrone Kommunikation und der inhärente Nichtdeterminismus in Zustandsmaschinen führen zu einem sehr komplexen Verhalten. Die präzise und widerspruchsfreie Interpretation dieses komplexen Verhaltens einer Zustandsmaschine ist Grundvoraussetzung für jegliche Art von Automatisierung. Die Bestimmung des korrekten, möglichen Verhaltens erfolgt in einer schrittweisen Simulation einer Zustandsmaschine. Da der Berechnungsaufwand mit zunehmender Länge der betrachteten Eingaben exponentiell ansteigt und deshalb eine Berechnung für typische Sequenzlängen unmöglich ist, werden Eingabesequenzen in Pakete unterteilt und für diese jeweils das korrekte, mögliche Verhalten bestimmt. Die Zusammenfassung von semantischen Zuständen nach einem solchen vereinfachten Simulationsschritt ermöglicht die Reduzierung des Berechnungsaufwands für den folgenden Simulationsschritt, wobei das korrekte, mögliche Verhalten approximiert wird. Da die Erkennungsrate der generierten Testfälle mit zunehmender Paketlänge steigt, ist der Generierungsalgorithmus entsprechend parametrisiert. Durch eine geeignete Wahl dieses Parameters kann hinsichtlich des Aufwands für die Testfallerzeugung und der Erkennungsrate der Testfälle ein Kompromiss erzielt werden.

Komplex strukturierte Daten treten sowohl in den Eingaben, als auch in den beobachteten Ausgaben auf. Die Behandlung von Daten in einem automatisierten Testansatz ist ein schwieriges und komplexes Problem. Zur Bestimmung geeigneter Eingabedaten werden unterschiedliche probabilistische Strategien benutzt, die durch weitere Analysen der Zustandsmaschine noch erweitert und verbessert werden können. Daneben erfolgt die Auswahl von relevanten Testfällen durch unterschiedliche probabilistische Selektionsstrategien. Dabei werden so genannte Nutzungsprofile zur Bestimmung von relevanten Eingabesequenzen benutzt und dadurch weitestgehend vermieden, dass unwahrscheinliche und für ein bestimmtes Testziel irrelevante Testfälle erzeugt werden. Die Trennung von Eingabeerzeugung und Testableitung

ermöglicht eine Testfallableitung auf Basis unterschiedlicher Teststrategien. Schließlich ermöglichen unterschiedliche Überdeckungskriterien qualitative Aussagen bezüglich der generierten Testfälle. Einige dieser Kriterien wurden auf Basis des semantischen Modells einer Zustandsmaschine definiert. Dadurch konnte die Aussagekraft erheblich gesteigert werden.

Praktisches Ergebnis der Arbeit ist die prototypische Werkzeugumgebung TEAGER. Diese setzt sich auf der einen Seite aus einer Umgebung zur Testfallerzeugung und -ausführung und auf der anderen Seite aus einer Umgebung zur Simulation von Zustandsmaschinen zusammen. Letztere ermöglicht eine realitätsnahe Ausführung von Zustandsmaschinen, d. h. insbesondere, dass nichtdeterministisches und zeitlich variables (probabilistisches) Ausführungsverhalten simuliert werden kann. Zum einen kann damit eine Zustandsmaschine, z. B. die Spezifikation, schon in einer frühen Entwicklungsphase analysiert werden und zum anderen kann der Ansatz selbst validiert werden. Mit der Werkzeugumgebung TEAGER habe ich die Anwendbarkeit des Ansatzes anhand kleinerer Beispiele und einer größeren Fallstudie demonstriert. Die Beispiele zeigen, dass durch die Verhaltensapproximation mit der Komplexität und der Zustandsexplosion innerhalb von Zustandsmaschinen umgegangen werden kann, wobei eine akzeptable Erkennungsrate erzielt und die Ablehnung korrekter Systeme vermieden werden kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Einleitung und Motivation

Eingebettete technische Systeme, wie sie z. B. als Assistenzsysteme in Kraftfahrzeugen vorkommen, gewinnen mit zunehmender Technisierung unseres Alltags eine immer größere Bedeutung in allen Bereichen unseres Lebens. Waren vor Jahren Assistenzsysteme in Kraftfahrzeugen nur in den gehobenen Klassen zu finden, so sind sie heutzutage bereits Standard und nicht mehr wegzudenken. Der breite Einsatz von eingebetteten technischen Systemen macht sie zu einem elementaren Bestandteil unseres täglichen Lebens und es wird mittlerweile erwartet, dass man sich auf ihre korrekte Funktionsweise blind verlassen kann. Das gilt nicht nur für Systeme, die in lebenswichtigen Bereichen zum Einsatz kommen und bei denen deshalb eine hohe Erwartung an ihre korrekte Funktionsweise gestellt wird, sondern auch für einfache technische Systeme wie z. B. ein Mobiltelefon. Aufgrund dieser Erwartungshaltung, der immer kürzer werdenden Entwicklungszeiten und des stetig steigenden Preisdrucks spielt Qualitätssicherung eine zunehmend an Bedeutung gewinnende Rolle bei der Entwicklung solcher Systeme.

Im Allgemeinen handelt es sich bei eingebetteten Systemen um Systeme, die aus Hardware- und Software-Komponenten bestehen und über Aktoren und Sensoren mit ihrer technischen

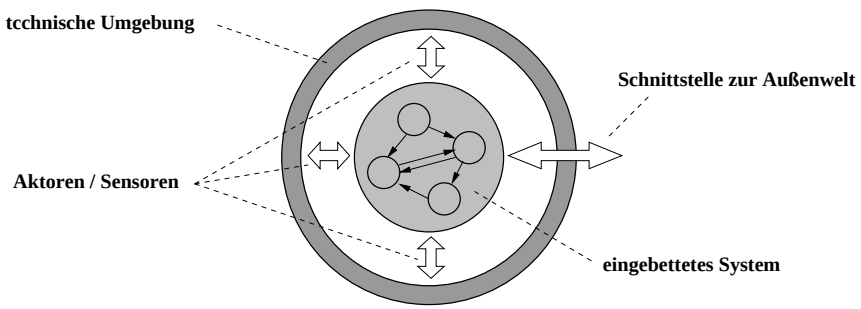


Abbildung 1.1: Schematische Darstellung eines eingebetteten technischen Systems.

Umgebung und der Außenwelt interagieren. Dies umfasst auch, dass über definierte Schnittstellen Informationen mit anderen Systemen ausgetauscht werden können. Abbildung 1.1 illustriert den allgemeinen Aufbau eines eingebetteten Systems. Im Gegensatz zu transformativen Systemen steht hier der reaktive Charakter im Vordergrund. Die Software dient dabei sowohl zur Steuerung des Systems selbst, als auch zur Berechnung von Steuersignalen als Reaktion auf aperiodisch auftretende Eingabeereignisse. Dabei werden, sofern benötigt, die ausgetauschten Informationen in analog/digital- bzw. digital/analog-Wandlern vorverarbeitet.

Der Erfolg von eingebetteten Systemen kann teilweise durch die Verbindung von spezifischer Hardware mit der Flexibilität von Software zur Steuerung begründet werden und es ist zu erwarten, dass der Bedarf an solchen Systemen zukünftig weiter steigen wird. Sehr häufig entstehen durch die Vernetzung einer Vielzahl von eigenständigen, eingebetteten Systemen auch vielschichtige Gesamtsysteme, deren Aufgabe in der Steuerung komplexer, zusammenhängender Prozesse besteht. Doch gerade durch diese Eigenschaften, d. h. da sie zumeist für spezielle Anwendungen entworfen werden und dedizierte Funktionen innerhalb eines Gesamtsystems übernehmen, entstehen sehr hohe - zum Teil auch nichtfunktionale - Anforderungen an solche Systeme. Die hohe Komplexität von eingebetteten Systemen in Verbindung mit den enorm hohen Anforderungen machen ihre Entwicklung sehr schwierig. Ein modellbasierter Ansatz ermöglicht eine Analyse der Systemeigenschaften bereits vor der Realisierung. Dies ist im Zusammenhang mit eingebetteten Systemen unverzichtbar, nicht nur um die Komplexität in den Griff zu bekommen und den hohen Anforderungen gerecht zu werden, sondern auch, da durch die häufig sehr individuellen Lösungen ein erheblicher Entwicklungsaufwand entsteht.

Die Konsequenzen fehlerhafter Software-Produkte zeigen eindrucksvoll, dass die systematische Entwicklung und insbesondere die Qualitätssicherung von Software noch nicht den Standard erreicht hat, den man sich wünschen würde. Dafür gibt es mehrere Gründe. Einerseits wird der Druck auf die Firmen, ihre Produkte möglichst schnell und kostengünstig auf den Markt zu bringen, immer größer. Als Folge werden die Entwicklungszyklen immer kürzer und gerade eine angemessene Qualitätssicherung oft vernachlässigt. Andererseits nehmen Umfang und Komplexität der Systeme immer schneller zu. Qualitätssicherung erfolgt auch nach jahrelanger Forschung häufig noch manuell, ad-hoc und unsystematisch. Die Folgekosten von Fehlern steigen drastisch, je später diese im Entwicklungsprozess aufgedeckt werden. Die Forschung auf dem Gebiet der modellbasierten Entwicklung und Qualitätssicherung soll dabei helfen, die bestehenden Probleme zukünftig zu bewältigen. In diesem Kontext stellt das automatisierte Testen einen wichtigen Bestandteil der modellbasierten Qualitätssicherung dar.

Die Arbeitsgruppe Softwaretechnik der Fakultät IV - Elektrotechnik und Informatik der Technischen Universität Berlin beschäftigt sich seit mehreren Jahren mit der Entwicklung von Methoden und Werkzeugen zur Softwareentwicklung, mit Qualitätssicherungstechniken und -maßnahmen sowie mit der softwaretechnischen Realisierung von IT-Sicherheitsanforderungen. Softwareentwicklung wird als das Zusammenspiel von Produktentwicklung und Qualitätssicherung verstanden, die durch geeignete Methodiken in einem systematischen Entwicklungsprozess integriert sind. Speziell die systematische Anwendung mathematisch fundierter Techniken und ihre Einbettung in Softwareentwicklungsprozesse bilden den Rahmen der vorliegenden Arbeit. In einer Kooperation mit den Fraunhofer Instituten FIRST in Berlin und IESE in Kaiserslautern wurde das Forschungsprojekt »QUASAR – Integrated Quality Assurance and Requirements Analysis for the Software Development in Automotive Systems« durchgeführt. Ziel des Projektes war es, auf Basis von Agenden und Referenzarchitekturen Techniken für

eine effiziente und methodische Unterstützung der Entwicklung von elektronischen Steuereinheiten zu entwickeln [81, 102]. Die vorliegende Arbeit knüpft an die dort begonnenen Arbeiten an und führt diese im Bereich der automatisierten Qualitätssicherung weiter.

1.2 Zielsetzung der Arbeit

Strenge Sicherheitsbestimmungen, die entweder vom Auftraggeber oder durch rechtliche Vorschriften gefordert werden, haben im Bereich der eingebetteten Systeme zu besonders hohen Qualitätsanforderungen an die beteiligten Ingenieursdisziplinen geführt. Dafür manuelle Techniken einzusetzen ist nicht nur kompliziert und zeitintensiv, sondern zudem extrem fehleranfällig. Die hohe Komplexität von eingebetteten Systemen erfordert jedoch Prozesse, die es zum einen ermöglichen Dinge möglichst früh auszuprobieren und zu testen, und sich zum anderen automatisieren lassen, um den prüfbaren Umfang zu erhöhen. Über 40 Prozent der im Betrieb festgestellten Fehler in Steuerungs- und Regelungssoftware sind auf Unzulänglichkeiten in der Analysephase oder der Systemspezifikation zurückzuführen. Eine umfassende Qualitätssicherung, die möglichst früh im Entwicklungsprozess einsetzt, kann daraus entstehende Kosten wesentlich reduzieren. Grafische Softwaremodelle begleiten die Entwicklungsprozesse sinnvoll und ermöglichen eine frühzeitige und detaillierte Analyse des zu entwickelnden Systems. Mit ihnen können Kernanforderungen identifiziert und Inkonsistenzen erkannt und beseitigt werden. Zudem können sie als Referenz für die Prüfung des entwickelten Systems benutzt werden. Sie haben darüber hinaus den Vorteil, dass sie von Menschen sehr schnell erfasst und verarbeitet werden können. Somit bilden sie eine ideale Arbeitsgrundlage während des gesamten Entwicklungsprozesses.

Die Schwierigkeit einer umfassenden automatisierten Qualitätssicherung für eingebettete technische Systeme ist darin begründet, dass solche System asynchron mit ihrer Umgebung kommunizieren und nichtdeterministisches Verhalten aufweisen können. Asynchrones Verhalten innerhalb von ereignisdiskreten Systemen bedeutet, dass Ereignisse nicht an bestimmte Zeiten gebunden sind. Das bedeutet insbesondere, dass die Ausgaben zu einer bestimmten Eingabe zeitlich versetzt erfolgen können. Somit können sich die Ausgaben zu einer Sequenz von Eingaben beliebig, d. h. unter Erhaltung der kausalen Ordnung, mit diesen verzahnen. Zustandsmaschinen sind ein adäquater Beschreibungsformalismus für solche Systeme. Asynchronität drückt sich in Zustandsmaschinen dadurch aus, dass die Eingaben bis zu ihrer weiteren Verarbeitung zwischengespeichert und anschließend atomar, d. h. einzeln und vollständig, verarbeitet werden. Der Ereignisspeicher ist nicht direkt sichtbar und induziert dadurch nichtdeterministisches Verhalten. Nichtdeterminismus ist außerdem impliziter Bestandteil des semantischen Modells von Zustandsmaschinen und kann vom Benutzer explizit modelliert werden. Dies erschwert nicht nur die Erstellung einer für eine automatisierte Qualitätssicherung geeigneten Spezifikation, sondern macht auch die Berechnung des zu erwartenden korrekten Verhaltens extrem kompliziert und aufwändig. Ein weiteres Problem ist, dass Zustandsmaschinen im UML-Standard [110] keine präzise und formale Semantik zugrunde liegt. Eine solche Semantik ist jedoch essentielle Voraussetzung für die Integration in automatisierte Prozesse.

Die vorliegende Arbeit verfolgt drei aufeinander aufbauende Ziele.

1. Zustandsmaschinen sollen als Notation für die Beschreibung des diskreten reaktiven Verhaltens von eingebetteten technischen Systemen verwendet werden. Aufgrund ihrer Ausdrucksmächtigkeit erlauben sie es, asynchrones und nichtdeterministisches Verhalten adäquat zu beschreiben. Zunächst soll ein Ausschnitt der Zustandsmaschinen identifiziert werden, der eine geeignete Ausgangsbasis für die Entwicklung einer Automatisierten Testfallableitung darstellt. Da der UML-Standard [110] keine präzise Semantik für Zustandsmaschinen angibt, muss für diese reduzierten Zustandsmaschinen eine formale Semantik entwickelt werden. Die so geschaffene semantische Fundierung soll gewährleisten, dass Konformität formal beschrieben und Testfälle automatisiert abgeleitet werden können.

2. Auf Basis der formalisierten Zustandsmaschinen soll ein Ansatz entwickelt werden, mit dem Konformität zwischen einer Spezifikation in Form einer Zustandsmaschine und einem System unter Test nachgewiesen werden kann. Dabei sollen die Beobachtungen, die an den eingehenden und ausgehenden Schnittstellen des Systems beobachtet werden, die Grundlage für die Definition von Konformität bilden. Ein abgeleiteter Testfall soll das mögliche korrekte Verhalten umfassen, um damit während der Ausführung eine Bewertung vornehmen zu können. Neben der theoretischen Fundierung soll auch die Praktikabilität berücksichtigt werden. Gerade die Asynchronität und der inhärente Nichtdeterminismus werden eine praktikable vollständige Verhaltensbestimmung unmöglich machen. Basierend auf dem theoretischen Modell und möglichen Reduktionsstrategien soll ein Algorithmus entwickelt werden, mit dem Testfälle für einen Konformitätstest abgeleitet werden können.

3. Um den entwickelten Ansatz zu evaluieren, sollen die theoretischen Resultate in einer prototypischen Realisierung umgesetzt werden. Diese soll vor allem die Machbarkeit und die Aussagefähigkeit des Ansatzes zeigen.

An diese Hauptziele der Arbeit grenzen drei weitere Themenkomplexe an, die bei der Bearbeitung berücksichtigt werden sollen.

4. Die Ableitung von Testfällen beruht i. A. auf einer Vorgabe von Eingaben. Da sich Eingabesequenzen für eingebettete Systeme zum einen beliebig zusammensetzen und zum anderen beliebig lang sein können, ist die Zahl der potenziellen Testfälle unbeschränkt. Daraus ergibt sich neben der Fragestellung »wie« Testfälle aus Zustandsmaschinen abgeleitet werden können auch sofort die Frage, »welche« Testfälle abgeleitet werden sollten. Es soll untersucht werden, wie die Testfallerzeugung so gesteuert werden kann, dass nur relevante und bezüglich eines definierten Testziels interessante Testfälle abgeleitet werden.

5. Die abgeleiteten Testfälle werden vornehmlich dafür benutzt, zu zeigen, dass im System unter Test ein gefordertes Qualitätsniveau erreicht wurde. Somit gibt es eine direkte Abhängigkeit der Qualität des Systems unter Test von der Qualität der generierten Testfälle. Ihre Qualität ist entscheidend für die Aussage, ob das System unter Test die an sie gestellten Anforderungen erfüllt oder nicht. Deshalb sollen geeignete Kriterien zur qualitativen Bewertung der abgeleiteten Testfälle definiert werden.

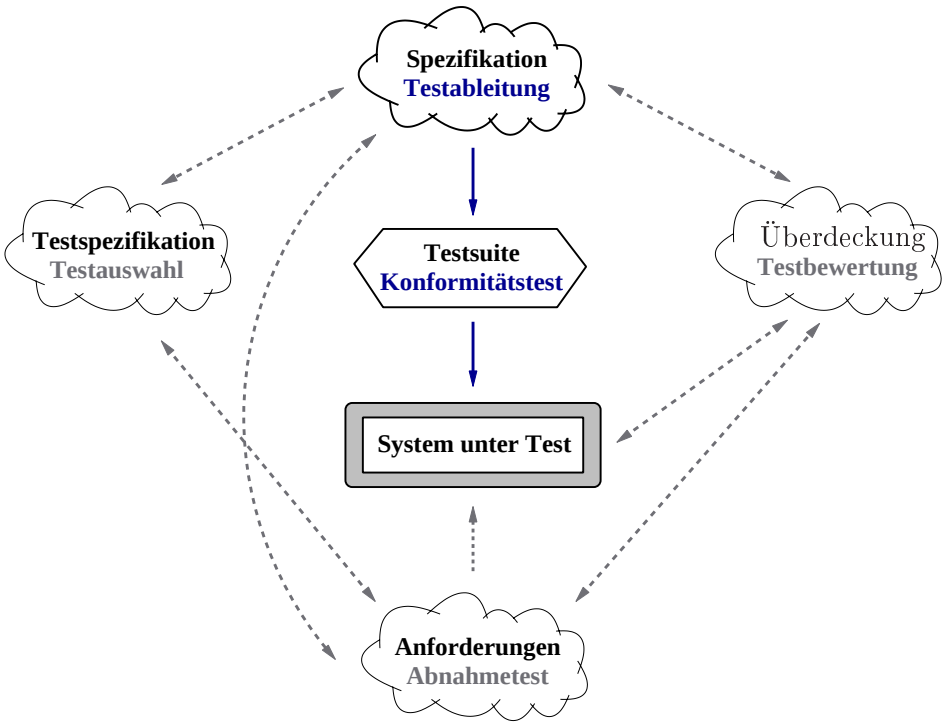


Abbildung 1.2: Zusammenhänge zwischen den Teilzielen.

6. Konformitätstests stellen nur einen Teil der i. A. benötigten Prüfungen dar. Ihr Ziel ist es primär, zu prüfen, ob das System unter Test »richtig entwickelt« wurde, also in seinem Verhalten konform zu seiner Spezifikation ist. Eine zweite wichtige zu beantwortende Frage ist, ob auch »das richtige System« entwickelt wurde. Diese Art der Prüfung wird hauptsächlich auf Basis der ermittelten Anforderungen durchgeführt. Deshalb sollen nach Möglichkeit beide Prüfziele in dem Testansatz zusammengeführt werden.

Die gezielte Testauswahl, die Bewertung der Qualität der erzeugten Testfälle und die Prüfung gegen die Anforderungen sind von besonderer Bedeutung für die Entwicklung eines in sich geschlossenen und umfassenden Testprozesses. Abbildung 1.2 illustriert die Zusammenhänge der genannten Teilziele 1 bis 6. Dabei sind mit durchgehenden blauen Pfeilen die Aktivitäten der Hauptziele der vorliegenden Arbeit dargestellt. Diese beinhalten die automatisierte Ableitung einer Testfsuite für den Konformitätstest gegen ein System unter Test. Mit gestrichelten grauen Pfeilen sind die Zusammenhänge der angrenzenden Aktivitäten dargestellt.

Zusammenfassend betrachtet bedeutet dies, dass mit der Arbeit ein konzeptioneller Rahmen für einen umfassenden, praktikablen und automatisierten Konformitätstest auf Basis reduzierter Zustandsmaschinen entwickelt werden soll.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in drei Teile unterteilt. Im ersten Teil werde ich die theoretischen Grundlagen erarbeiten. Dafür werde ich in Kapitel 2 Themengebiete und Grundlagen, auf

denen die Arbeit aufbaut, beschreiben. In Kapitel 3 werde ich die verwendeten Zustandsmaschinen an einem kleineren Beispiel einführen und deren Semantik formalisieren. Dies bildet die Grundlage, um in Kapitel 4 Konformität formal zu beschreiben und einen Ansatz zur Ableitung von Testfällen zu entwickeln. Der zweite Teil thematisiert die Entwicklung einer Werkzeugumgebung, die den vorgestellten Testansatz umsetzt und evaluiert. In Kapitel 5 werde ich die einzelnen Bestandteile der Werkzeugumgebung TEAGER und deren Merkmale vorstellen und diskutieren. In Kapitel 6 werde ich die Funktionsweise und die Anwendbarkeit der Werkzeugumgebung an einem kleineren Beispiel und einer größeren Fallstudie demonstrieren und die Ergebnisse der Evaluation diskutieren. Der dritte Teil soll die Arbeit abschließen. Dafür werde ich zunächst in Kapitel 7 verwandte Arbeiten diskutieren und mit meiner Arbeit vergleichen. Abschließend werde ich in Kapitel 8 weiterführende Themen der Arbeit besprechen und in Kapitel 9 die erzielten Ergebnisse der Arbeit umfassend diskutieren und offene Fragestellungen für weitergehende Forschung vorstellen.

Im Anhang A ist eine exemplarische Umsetzung eines semantischen Schritts einer Zustandsmaschine in der funktionalen Programmiersprache ML [76] wiedergegeben. Diese dient vornehmlich dazu, die entwickelte Formalisierung formal zu prüfen und deren generelle Anwendbarkeit exemplarisch zu zeigen. Im Anhang B sind die Grammatiken der Zustandsmaschinen und der Testfälle für die Werkzeugumgebung TEAGER inklusive eines kurzen Beispiels abgebildet. Anhang C enthält die Modelle der größeren Fallstudie. Im Anhang D sind die in der Arbeit verwendeten deutschen und ihre englischen Entsprechungen, vornehmlich die aus dem UML-Standard [110] tabellarisch wiedergegeben. Dies soll vor allem einen leichteren Bezug zur englischsprachigen Literatur ermöglichen.

Die vorliegende Arbeit bezieht sich vornehmlich auf den UML 2 Standard [110] und insbesondere auf die dortige *Superstructure* Spezifikation. Sollte ein anderer Bezug gemeint sein, so werde ich dies separat angeben. Mit der Notation in Kapitel 3 und Kapitel 4 lehne ich mich weitestgehend an die Notation Z an. Einen geeigneten Überblick über die Notation Z gibt die folgende Literatur: [100, 120, 49, 80].

Teil I

Theoretische Grundlagen

Kapitel 2

Grundlagen

In diesem Kapitel beschreibe ich die Gebiete der Softwaretechnik, in die sich die vorliegende Arbeit einbettet. In Abschnitt 2.1 werde ich zunächst einen vereinfachten Softwareentwicklungsprozess beschreiben und in Abschnitt 2.2 speziell auf die Thematik der Qualitätssicherung eingehen. In Abschnitt 2.3 werde ich mich detailliert mit Testen als dynamischer Prüftechnik befassen und an dessen Ende einen kurzen Überblick über unterschiedliche grafische Notationen für Zustandsautomaten geben. In Abschnitt 2.4 werde ich einen kurzen Überblick über die Modellierungssprache UML geben und ihren generellen Aufbau und die im Standard der UML 2 enthaltenen Diagrammtypen, sowie die für den Rahmen der Arbeit wichtigsten Diagrammtypen beschreiben. Abschließend werde ich in Abschnitt 2.5 den Problembereich der Arbeit beschreiben und den verfolgten Lösungsansatz skizzieren.

2.1 Softwareentwicklung

In der Geschichte der Softwareentwicklung ist schon früh erkannt worden, dass zur Beherrschung der steigenden Komplexität von Softwaresystemen ein präzise definierter und geeignet strukturierter Softwareentwicklungsprozess notwendig ist. Ein genau definiertes Vorgehen während der Entwicklung dient dabei der besseren Übersicht und insbesondere der besseren Planbarkeit von großen Softwareprojekten. Mit der Zeit ist eine Vielzahl von Vorgehensmodellen entstanden [5, 98, 99], die methodische Vorgehensweisen und Techniken zur Softwareentwicklung beschreiben. Prominente Vertreter unter diesen Entwicklungsmethoden sind das Wasserfallmodell [85], das V-Modell (XT) [8, 112, 113], die risikogetriebene Entwicklungsmethode des Spiralmodells [9] und neuere agile Entwicklungsmethoden [6, 18, 91]. In Deutschland ist insbesondere das V-Modell von erheblicher praktischer Bedeutung. Unabhängig von einer konkreten Entwicklungsmethode lässt sich ein Softwareentwicklungsprozess grob in vier aufeinanderfolgende Phasen einteilen [96], denen bestimmte Aktivitäten und Zwischenergebnisse zugeordnet werden können und die jeweils wohldefinierte Start- und Endpunkte und eindeutig definierte Ergebnisse haben.

In der ersten Phase, der Anforderungsanalyse und -spezifikation, werden die Kundenwünsche, die bestehenden Randbedingungen und die Ziele der Entwicklung ermittelt und strukturiert. Präzise bedeutet dies, dass die charakteristischen Eigenschaften, die vom System erwartet werden, insbesondere Funktionalität, Verhalten, Struktur und Invarianten, festgelegt und in einem Lastenheft bzw. einer Anforderungsspezifikation erfasst werden. In der zweiten Phase, dem Systementwurf und der Systemspezifikation, wird auf Basis der Anforderungsspezifikation das System entworfen. Dabei erfolgt eine genaue Definition der Systemarchitektur, sowie eine Unterteilung in Hardware- und Softwarekomponenten. Das Ergebnis wird in einem Pflichtenheft bzw. einer Systemspezifikation dokumentiert. In der dritten Phase, der Implementierung, wird die zu realisierende Software in einer geeigneten Programmiersprache umgesetzt und schrittweise in das aus Hard- und Softwarekomponenten bestehende Gesamtsystem integriert. In der anschließenden vierten Phase, der Auslieferung, dem Einsatz und der Wartung, wird das System beim Auftraggeber in Betrieb genommen und über einen längeren Zeitraum dessen einwandfreies Funktionieren sichergestellt bzw. Anpassungen an neue Anforderungen, häufig als Softwareevolution bezeichnet, vorgenommen.

Qualitätssicherung erfolgt heutzutage entwicklungsbegleitend und wird keiner eigenständigen Phase mehr zugeordnet. Ziel der Qualitätssicherung ist es, den erfolgreichen Ablauf der Entwicklung sicherzustellen und möglichst früh Unstimmigkeiten und Probleme zu identifizieren. Aus diesem Grund werden Prüfaktivitäten in jeder einzelnen Entwicklungsphase durchgeführt. Treten Probleme auf, so werden die Ursachen identifiziert und Korrekturen, auch an den Artefakten der vorangegangenen Phasen, vorgenommen. Als Artefakte bezeichnet man dabei die Gesamtheit aller Ergebnisse oder Dokumente, die im bisherigen Prozess erstellt wurden. Diese sind auch die Basis für die Qualitätssicherung und können auf Vollständigkeit, Konsistenz und Korrektheit bezüglich der Ergebnisse aus vorangegangenen Phasen geprüft werden. Da heutige Entwicklungen häufigen Änderungen in den Anforderungen unterworfen sind, fällt der Qualitätssicherung außerdem eine ständige Kontrolle der erstellten Artefakte zu. Für die klassische Softwareentwicklung und Qualitätssicherung sind insbesondere die Anforderungs- und die Systemspezifikation (Pflichtenheft/Lastenheft) von erheblicher Bedeutung. Auf Grund der immer kürzer werdenden Entwicklungszyklen in Verbindung mit immer weiter steigenden Anforderungen, die sich außerdem im Entwicklungsprozess häufig noch mehrfach ändern, hängt der erfolgreiche Abschluss eines Projekts maßgeblich von der Qualität der entwickelten Artefakte, also insbesondere der Anforderungs- und der Systemspezifikation, ab. Dies ist eben darin begründet, dass die Artefakte als Basis für einen Großteil der qualitätssichernden Maßnahmen dienen und anhand dieser Dokumente entschieden wird, was korrekt ist und was nicht. Unter diesem Aspekt und unter Berücksichtigung des zunehmenden Bedarfs an Automatisierung werden formale Vorgehensweisen immer bedeutender.

Im Rahmen der formalen Spezifikation von Software soll die ungenaue und mehrdeutige menschliche Sprache durch die präzisen Mittel der Mathematik mit klarer Syntax und Semantik ersetzt werden. Ungenauigkeiten und Mehrdeutigkeiten, die sowohl in der Spezifikation als auch in der Realisierung selbst zu Fehlern führen können, sollen so weitgehend vermieden werden oder zumindest so früh wie möglich entdeckt werden. Qualität wird damit »in das System hineinkonstruiert«. Wesentlicher Vorteil der formalen Spezifikation ist zudem, dass mathematische Definitionen eindeutig sind und somit von einem Anwender oder einer Maschine immer gleich interpretiert werden. Somit bilden sie eine ideale Basis für eine maschinengestützte Prüfung von Systemmodellen und Systemen. In Hinblick auf die angewandten Formalismen werden im Wesentlichen zwei Typen von Spezifikationen unterschieden [98]. *Algebraische Spe-*

zifikationen verwenden Funktionen, elementare oder abstrakte Datentypen und Axiome zur Beschreibung von Software. Dabei beschreiben die Funktionen das Verhalten und die Axiome zusätzlich einzuhaltende Konsistenzbedingungen. Die internen Zustände der Software werden nicht explizit beschrieben. Da sie die Funktion und nicht den Zustand der Software beschreiben, eignen sich algebraische Spezifikationen insbesondere zur Spezifikation von Schnittstellen. *Modellbasierte Spezifikationen* beschreiben primär die Zustände eines Systems. Diese werden vor allem über mathematische Mittel wie Mengen, Folgen und Relationen definiert und die Funktionen des Systems darüber angegeben, wie diese den Zustand des Systems ändern. Einzuhaltende Konsistenzbedingungen werden mittels Prädikaten, den sog. Invarianten, beschrieben. Verbreitete formale Spezifikationssprachen sind Z [49], B [2], CSP [17] und Petri-Netze [77]. Z und B eignen sich insbesondere zur Beschreibung von sequentiellen Systemen, wobei Z als prozedural und B als objektbasiert bezeichnet werden kann. CSP und Petri-Netze eignen sich insbesondere zur Beschreibung von nebenläufigen Systemen.

Während der Entwicklung werden häufig grafische Modelle des zu entwickelnden Systems benutzt. Insbesondere Ansätze, die die Strukturen und das Verhalten, d. h. die Systemzustände und die Übergänge zwischen diesen, mit Hilfe von Diagrammen darstellen, gewinnen zusehends an Bedeutung. Ziel ist es, die Entwicklung des Systems transparent und steuerbar zu machen. Dabei werden sowohl die Anforderungen an das System, in sog. Umgebungsmodellen, als auch die Funktionen des Systems, in Systemmodellen, spezifiziert. Der Vorteil der grafischen Modelle liegt zum einen in ihrer schnellen Erfassbarkeit durch einen Menschen. Dies erleichtert die Kommunikation und die Dokumentation während des Entwicklungsprozesses. Zum anderen ermöglichen es insbesondere ausführbare Modelle, abstrakte Entwürfe zu verstehen und zu analysieren. Softwareentwicklung, in deren Zentrum vornehmlich die Erstellung und Verfeinerung von (grafischen) Modellen steht, nennt man daher *modellbasierte Softwareentwicklung*. Die Verwendung von Modellen geht u. U. so weit, dass eine automatische Ableitung des Programmcodes aus den erstellten Modellen erfolgt. Der Begriff modellbasiert stammt ursprünglich aus der Regelungstechnik. Dort werden Modelle von regelungstechnischen Systemen in einer Experimentierumgebung auf einem Echtzeitrechner simuliert und auf Basis dieser Simulation die Funktionsweise des Systems überprüft und weiter verfeinert. Die zunehmend auftretende *modellgetriebene* Softwareentwicklung setzt ihren Fokus auf die Automatisierung in der Softwareentwicklung und wird für Vorgehensmodelle verwendet, die als zentralen Bestandteil die automatisierte Verfeinerung von Modellen über den gesamten Entwicklungsprozess beinhalten. Dabei erfolgt eine klare Trennung von Funktionalität und Technologie und ein Teil der Programmierung kann automatisiert werden. Zum Beispiel werden bei der *Model Driven Architecture* (MDA) [68] Modelle über Transformationen von rein logischen Ebenen zu plattformspezifischen Ebenen transformiert und letztlich Programmcode abgeleitet. Dabei wird stets zwischen technologieunabhängiger und technologieabhängiger Modellierung unterschieden.

Die vorliegende Arbeit läßt sich in die Thematik der Qualitätssicherung in einem modellbasierten Entwicklungsprozess einordnen. Ich gehe davon aus, dass die zu prüfende Software auf Basis von grafischen Struktur- und Verhaltensmodellen entworfen und implementiert wurde. Weiterhin gehe ich davon aus, dass das Verhalten der Software durch Zustandsmaschinen beschrieben wurde. Die Arbeit soll es ermöglichen, die implementierte Software gegen diese Zustandsmaschinen zu prüfen, um so auf die korrekte Funktionsweise schließen zu können.

2.2 Qualitätssicherung

Wird die Softwareentwicklung in verschiedene Phasen mit definierten Ergebnissen unterteilt, so kann eine gezielte und strukturierte Prüfung der Ergebnisse in jeder Phase, d. h. insbesondere auch nach der ersten Phase, erfolgen. Ziel der Qualitätssicherung ist es, die Vollständigkeit und Korrektheit der Ergebnisse jeder einzelnen Phase sicherzustellen. Dabei werden die Ergebnisse späterer Entwicklungsphasen zusätzlich auf Konsistenz zu den in den vorangegangenen Phasen erstellten Ergebnissen geprüft. Die Ergebnisse aus den vorherigen Phasen dienen somit als Grundlage für die Bewertung der Ergebnisse der Prüfungen in den späteren Phasen. Ähnlich zur modellbasierten Softwareentwicklung nimmt die modellbasierte Qualitätssicherung Modelle des zu entwickelnden Systems als Grundlage bzw. als zentrale Basis für ihre Aktivitäten. Auch bei der Qualitätssicherung kann der präzise formale Charakter einiger Modelle genutzt werden, um maschinengestützte Prüfungen vorzunehmen.

Bezüglich der Qualitätssicherung während der Softwareentwicklung können zwei zentrale Fragestellungen unterschieden werden [99, 118]. Zum einen gilt es durch die Qualitätssicherung zu beantworten, ob »das richtige System« entwickelt wurde, es also die festgelegte Aufgabe tatsächlich löst bzw. es für deren Lösung geeignet ist. Zu diesem Zweck wird das System gegen die Anforderungen der realen Welt geprüft. Ziel ist es, nachzuweisen, dass das System oder ein Modell des Systems das von ihm geforderte Verhalten aufweist bzw. tatsächlich spezifiziert. Dieser Nachweis wird auch als Validierung bezeichnet. Zum anderen gilt es durch die Qualitätssicherung zu beantworten, ob das System »richtig entwickelt« wurde, also ob die Spezifikation korrekt und vollständig umgesetzt wird. Dieser Nachweis wird auch als Verifikation bezeichnet. Für die Beantwortung der beiden zentralen Fragestellungen sind unterschiedliche Vorgehensweisen und Techniken entwickelt worden. Mit diesen sog. V & V Techniken wird die externe und die interne Korrektheit von Modellen nachgewiesen. Eine grobe Klassifizierung der V & V Techniken kann hinsichtlich ihrer Grundlage und ihres Charakters getroffen werden. Dabei meine ich mit Grundlage, ob die Prüfungen auf einem Modell des Systems oder auf der tatsächlichen Realisierung ausgeführt werden. Mit Charakter meine ich, ob sich mit der Prüfung gesicherte Aussagen treffen lassen oder ob es sich um ein stichprobenartiges Verfahren handelt, bei dem nur auf Grundlage von einzelnen Experimenten Rückschlüsse auf das Gesamtsystem gezogen werden. Mit einigen Einschränkungen lassen sich die Prüfungen (und damit auch die Beantwortung der Fragen) nicht nur auf das System, sondern auch in jeder Entwicklungsphase auf die dort erstellten Artefakte anwenden. Beispielhaft werde ich im Folgenden drei verschiedene V & V Techniken vorstellen [41].

Beweistechniken umfassen den formalen Nachweis von Konsistenz und Korrektheit auf einem (mathematischen) Modell des Systems. Dabei werden mit semantisch fundierten Methoden und Techniken Aussagen über ein Systemverhalten bewiesen. Zu den wichtigsten Vertretern der Beweistechniken zählen das Modellprüfen und das Theorembeweisen. Beim Modellprüfen werden temporallogische Formeln, die zu prüfende Eigenschaften des System ausdrücken, auf einem Zustandsmodell des Systems zumeist automatisch nachgewiesen, wobei der gesamte Zustandsraum durchsucht wird. Problematisch ist bislang das Problem der Zustandsexplosion. Symbolische Techniken, Abstraktions- und Dekompositionstechniken bieten Ansätze, dieses Problem zu lösen. Für das Modellprüfen existieren zahlreiche Werkzeuge wie z. B. die Modellprüfer FDR [84, 26] und SMV [66, 65] oder I-Logix's Statemate (Statemate Model Checker) [101]. Beim Theorembeweisen werden auf Grundlage eines in einer logischen

Sprache formulierten Beweissystems Systemeigenschaften zumeist interaktiv nachgewiesen. Werkzeuge zum Theorembeweisen sind z. B. Isabelle [74, 70] und Coq [20]. Beweistechniken bieten den Vorteil, dass sie in frühen Phasen eines Entwicklungsprozesses angewendet werden können, da für ihre Anwendung keine Realisierung benötigt wird, und dass sie vollständige und gesicherte Aussagen bezüglich der geprüften Eigenschaften ermöglichen.

Bei der *Simulation* wird ein Modell des Systems zur Ausführung gebracht und Experimente durchgeführt, um z. B. Konsistenz, Korrektheit oder das Zeitverhalten gegen eine Referenz zu prüfen. Simulation findet besonders dann Anwendung, wenn keine geschlossenen Lösungen existieren oder die Erstellung eines Modells für Verifikationszwecke extrem aufwändig ist (eingekaufte oder physikalische Komponenten). Da auch Simulation auf einem Modell des Systems beruht, kann sie ebenfalls früh im Entwicklungsprozess eingesetzt werden. Die Simulation eines Systems erfordert jedoch mehr Ressourcen als die Ausführung des realen Systems. Somit geht man, sofern eine Realisierung des Systems vorhanden ist, zum Testen des Systems über.

Testen bezeichnet das Ausführen von Experimenten auf dem realen System bzw. der Implementierung, um diese gegen die Anforderungen oder die Spezifikation zu prüfen und bezüglich der gewählten Eingaben konkrete Rückschlüsse auf das gesamte System vorzunehmen. Dabei wird die Implementierung mit realen Daten stichprobenartig zur Ausführung gebracht und die Korrektheit auf Basis der während des Experiments gemachten Beobachtungen bewertet. Die Entscheidung, ob eine Beobachtung korrekt ist, wird dabei auf Basis der Spezifikation getroffen. Der große Vorteil von Testen liegt in der Ausführung der Experimente auf dem realen System und in dem möglichen, breiten Anwendungsspektrum. So wird beim Testen das Zusammenspiel von tatsächlicher Hard- und Software geprüft und unterliegt fast keinen praktischen Einschränkungen. Abweichungen, die gefunden werden, können auf Fehler in der Umsetzung oder auf Fehler im Modell hinweisen.

Die vorgestellten Techniken ergänzen sich gegenseitig und nur ihre gemeinsame Anwendung ermöglicht eine qualitativ hochwertige Entwicklung und eine umfassende Prüfung eines Softwaresystems mit verlässlichen Aussagen. Dabei ist zu beobachten, dass die Grenzen zwischen den unterschiedlichen Techniken aufgrund der immer besser werdenden Algorithmen und der größeren Rechenleistung, die einem heutzutage zur Verfügung steht, weiter miteinander verfließen. Im nächsten Abschnitt werde ich detailliert auf den für die Arbeit grundlegenden Bereich des Testens eingehen.

2.3 Testen

Testen ist eine dynamische Prüftechnik und bezeichnet allgemein den Prozess, ein System unter Test mit der Absicht auszuführen, Fehlverhalten aufzudecken [69, 61]. Grundvoraussetzung für die Durchführung von Tests ist folglich, dass ein System unter Test, oder zumindest eine Implementierung der Software, zur Verfügung steht und dieses mit den einzelnen Tests zur Ausführung gebracht werden kann. Das Aufdecken von Fehlverhalten dient primär dazu, die Qualität des entwickelten Systems zu verbessern. Dies wird dadurch erreicht, dass nach der eigentlichen Testphase, dem *Debugging*, das aufgedeckte Fehlverhalten analysiert und die Ursache für dieses Fehlverhalten, der eigentliche Fehler, lokalisiert und beseitigt wird. Das Aufzeigen von Fehlverhalten ist eine analytische Qualitätssicherungsmaßname, da mit ihr geprüft (analysiert) wird, ob ein festgesetztes Qualitätsziel erreicht wurde. Im Gegensatz

zu den analytischen Qualitätssicherungsmaßnamen versuchen die konstruktiven Maßnamen bereits während der Entwicklung des Systems unter Test ein hohes Maß an Qualität zu erreichen. Die konstruktiven Maßnamen zielen somit auf den Prozess der Softwareerstellung, dass heisst z. B. auf die Entwicklungsmethodik, Festlegung von Richtlinien und Verwendung von geeigneten Entwicklungswerkzeugen. In der Praxis werden die dynamischen Prüftechniken mit statischen Prüftechniken, z. B. mit Inspektions- und Reviewtechniken, kombiniert. Diese haben den Vorteil, dass sie bereits in einer sehr frühen Phase zum Einsatz kommen können, da es bei diesen Techniken nicht erforderlich ist, dass das System unter Test implementiert ist und zur Ausführung gebracht werden kann. Nachteilig ist, dass sie zum Teil sehr aufwändig sind. Somit werden sie zumeist auf spezielle und besonders wichtige Artefakte angewendet.

Das Ausführen von Testfällen hat den Vorteil, dass das reale System unter Test zur Ausführung gebracht wird und somit bezüglich des tatsächlichen Verhaltens argumentiert weden kann. Die Ausführung der Tests kann auf unterschiedlichen Abstraktionsebenen geschehen, wobei durch den geeigneten Einsatz von Adaptern sehr häufig mit nahezu den gleichen Testfällen getestet werden kann. Bei der so genannten *Software in the Loop* (SIL) wird die Software losgelöst von ihren Hardwarekomponenten in einer Simulationsumgebung ausgeführt. Bei der sog. *Hardware in the Loop* (HIL) wird die Software mit ihren technischen Komponenten auf der Hardware in einem Umgebungs-Simulator ausgeführt. In beiden Szenarien kann mit Testfällen das Ausführungsverhalten überprüft werden. Hinsichtlich der Zielsetzung des Testens gibt es viele unterschiedliche Aspekte des Systems unter Test, die überprüft werden können. Den höchsten Stellenwert haben dabei der Überprüfung der Funktionalität bezüglich Vollständigkeit und Korrektheit (Konformitätstests) und die Überprüfung der Robustheit, d. h. wie es auf untypische Situationen oder Eingaben reagiert (Robustheitstest). Weitere Ziele können z. B. der Nachweis von Sicherheitseigenschaften, Integrität, Performanz, Zuverlässigkeit, oder Bedienbarkeit sein.

Ähnlich zum Prozess der Softwareentwicklung, unterteilt man auch die Aktivitäten des Testens auf unterschiedliche Phasen auf. Nach dem ANSI/IEEE-Standard 829 und dem ISO-Standard 12207 lassen sich die Testaktivitäten grob in sieben Phasen unterteilen: die *Testplanung*, der *Testentwurf*, die *Testfallspezifikation*, die *Testprozedurerstellung*, dem *Testaufbau*, der *Testausführung* und der *Testauswertung* [96]. Während der Testplanung erfolgt die zeitliche und ressourcenorientierte Planung der Aufgaben und Ziele der durchzuführenden Tests, dies ist somit vor allem eine organisatorische Maßnahme. Während des Testentwurfs erfolgt die konzeptionelle Planung des Tests. Hier wird besonders darauf eingegangen, in welcher Reihenfolge einzelne Systemteile getestet werden sollen. Die konzeptuelle Planung erfolgt dabei wie im Softwareentwicklungsprozess, um frühes Testen zu ermöglichen. Während der Testfallspezifikation werden die einzelnen Testfälle und explizit die Vor- und Nachbedingen eines Tests beschrieben. Die erstellten Testfälle werden während der Testprozedurerstellung um Testskripte erweitert. Die Testskripte bilden die technische Basis für die Testfälle und ermöglichen es, die erstellten Testfälle später auf dem zu testenden System auszuführen. Der Testaufbau zielt auf die eigentliche Vorbereitung der Testausführung ab. Hier werden benötigte Werkzeuge, konkrete Testdaten, sowie die fertige Testumgebung samt Hardware und Software bereitgestellt. Die Testausführung bezeichnet den eigentlich Test des Systems, also die Durchführung der Experimente, mit entsprechender Protokollierung. Die Bewertung der Testdurchführung und die Auswertung der Protokolle erfolgt in der abschließenden Phase der Testauswertung. Dabei wird aus den ermittelten Testergebnissen ein nachvollziehbarer und verständlicher Testbericht erzeugt.

Die Art der Tests und der Duchrführungszeitpunkt orientieren sich am Stand der Entwicklung. Dabei geht man davon aus, das moderne Software nicht mehr am Stück getestet werden kann. Zum einen aus reinen Komplexitätsbetrachtungen und zum anderen aus der Tatsache heraus, dass man möglichst früh mit der Qualitätssicherung beginnen möchte und nicht darauf warten kann, dass das gesamte System vollständig entwickelt wurde. Im Allgemeinen unterteilt man den Testansatz grob in vier aufeinander aufbauende Phasen: *Klassen- oder Modultest*, *Integrationstest*, *Systemtest* und *Abnahmetest*. Sind einzelne Teile der zu entwickelnden Software fertiggestellt, so können diese separat, d. h. losgelöst von den übrigen Funktionalitäten des Gesamtsystems, geprüft werden. Nach einer detaillierten Einzelprüfung erfolgt die Integration von einzelnen Teilen zu einem Gesamtsystem. Hier wird insbesondere das Zusammenspiel der einzelnen Teile geprüft. Der Systemtest bezeichnet die Prüfung des vollständig integrierten Gesamtsystems über die Benutzerschnittstelle und endet mit dem Abnahmetest, der beim Auftraggeber in der realen Umgebung und mit realen Daten durchgeführt wird. Wesentliches Ziel des Abnahmetests ist es zu prüfen, ob das System tatsächlich die gestellten Anforderungen erfüllt. Ist die Software im Einsatz kann es in der Wartungsphase weiterhin zu Änderungen an der Software kommen. Die dann eingesetzten Prüftechniken richten sich dann vornehmlich an den durchgeführten Änderungen und werden sehr häufig ebenfalls als Regressionstests bezeichnet. Regressionstests sind auch in früheren Phasen nützlich. Die Software unterliegt auch während des Entwicklungsprozesses ständigen Änderungen. Zum Beispiel werden Fehler, die durch die Prüfaktivitäten gefunden werden, auch beseitigt. Außerdem wird die Software ständig weiterentwickelt und verbessert, unter anderem als Folge der frühen Prüfaktivitäten. Aus diesem Grund werden Testfälle häufig wiederholt ausgeführt. Diese, auch als *Regressionstest* bezeichnete Testaktivität hat vornehmlich zur Aufgabe, Änderungen und Erweiterungen zu überprüfen und zu zeigen, dass die verbliebenen Systemteile weiterhin ordnungsgemäß funktionieren. Dabei muss beachtet werden, dass u. U. auch die Testfälle bei Änderungen des Funktionsumfangs oder der Spezifikation angepasst werden müssen.

Ein Testfall stellt die für die Durchführung eines Experiments mit dem System unter Test benötigten Informationen bereit. Dabei umfassen Testfälle i. A. mehr als nur die eigentlichen Eingaben für das Experiment. Wichtige Bestandteile der Beschreibung eines Testfalls sind: die Vorbedingungen, die vor der Testausführung hergestellt werden müssen, die Eingaben, die zur Durchführung des Testfalls notwendig sind, die erwarteten Ausgaben (die Sollwerte) des Systems unter Test auf diese Eingaben, die erwarteten Nachbedingungen, die als Ergebnis der Durchführung des Testfalls erzielt werden und z. B. Prüfanweisungen, d.h. Anweisungen, wie die Eingaben an das System unter Test zu übergeben sind und wie Sollwerte abzulesen sind. Zusätzlich beinhaltet ein Testfall aber auch noch eine Reihe von organisatorischen Informationen. So muss z. B. definiert sein, auf welche Version der Software sich der Testfall bezieht, oder welche Hardware- und Softwarekomponenten benötigt werden (Test-Setup). Zudem enthält ein Testfall auch eine Intention und eine Bewertung der Priorität, um auf Basis dieser Informationen den Testprozess möglichst effizient gestalten zu können. Die genaue Zusammensetzung eines Testfalls variiert jedoch stark und wird zumeist durch den konkreten Projektkontext bestimmt.

Die Bestimmung der Sollwerte stellt i. d. R. die größte Herausforderung bei der Ableitung von Testfällen dar. Aus diesem Grund wurde in der Literatur auch der Begriff *Testorakel* geprägt. Die Bestimmung der Eingaben kann auf Basis der Spezifikation (funktionsorientierte Testtechniken oder auch *Black-Box*-Tests) oder auf Basis des Programmcodes (strukturorientierte Testtechniken oder *White-Box*-Tests) erfolgen. Grob lassen sich die beiden Varianten

so beschreiben, dass bei den funktionsorientierten Testtechniken die Ableitung der Testfälle auf Basis der in der Spezifikation beschriebenen Funktionalität erfolgt. Bei den strukturorientierten Testtechniken erfolgt die Ableitung der Testfälle auf Basis der Ablaufstruktur des Programmcodes oder auf Basis des analysierten Datenflusses. Eine detaillierte Klassifikation und Beschreibung von unterschiedlichen Testtechniken kann in [61, 99] nachgelesen werden. Die Bestimmung der Erwartungswerte erfolgt bei beiden Varianten auf Basis der Spezifikation.

Einzelne Testfälle werden in einer Testsuite zusammengefasst. Bezüglich einer Testsuite können diverse Eigenschaften bestimmt werden. Man nennt eine Testsuite *vollständig* [105], wenn man auf Basis dieser Testsuite für jedes System unter Test entscheiden kann, ob das System unter Test eine korrekte Realisierung der Spezifikation ist oder nicht. Das Problem an einer vollständigen Testsuite ist, dass man i. d. R. eine unendliche Anzahl von Testfällen benötigt. Weicht man von der Vollständigkeit ab und geht zu einer endlichen Testsuite über, so kann man zwei Arten von Testsuiten unterscheiden. Eine Testsuite nennt man *exhaustive*, wenn sie jedes korrekte System unter Test auch als ein solches erkennt. Bei einer solchen Testsuite kann es jedoch passieren, dass inkorrekte Systeme unter Test nicht erkannt werden. In einem solchen Fall fällt die Testsuite das Urteil *false positive*. Im Gegensatz dazu bezeichnet man eine Testsuite als *sound*, falls sie nur ein negatives Urteil fällt, wenn das System unter Test auch inkorrekt ist. Hier ist problematisch, dass nicht jedes korrekte System unter Test auch als ein solches erkannt wird. Wichtig ist in beiden Fällen, das keine *false negative* Urteile abgegeben werden, da dadurch die Auswertung der Testdurchführung erheblich erschwert werden würde. Da man sich nun auf eine endliche Anzahl von Testfällen beschränkt ist man i. A. daran interessiert, dass die einzelnen Testfälle eine möglichst hohe Wahrscheinlichkeit haben, ein Fehlverhalten aufzudecken. Anders formuliert könnte man sagen, dass man daran interessiert ist, mit möglichst wenigen Testfällen eine möglichst viel Fehlverhalten aufzudecken. Dies ist der Grund, warum ich einen Teil meiner Arbeit der Problematik der Testfallauswahl gewidmet habe.

Neben der Beschränkung auf eine feste Anzahl von Testfällen kann man die Entscheidung, wann man mit dem Testen aufhören will, auch anhand von sog. Testendekriterien bestimmen. Häufigstes Kriterium ist dabei der Grad der Überdeckung der Spezifikation oder des zu testenden Programmcodes hinsichtlich Funktion, Struktur und Daten (z. B. Nachrichten oder Ausnahmebehandlungen). Ein anderes Kriterium ist z. B. die Fehlerrate, d. h. die Anzahl der Fehler, die pro ausgeführter Testaktivität gefunden werden. Die Testdurchführung erfolgt heute weitgehend durch vollautomatische Testwerkzeuge. Abhängig vom Zielsystem kommen hier Unit-Test-Werkzeuge, Testwerkzeuge für grafische Benutzeroberflächen, Lasttestsysteme, Hardware-in-the-loop-Prüfstände oder andere Werkzeuge zum Einsatz.

2.3.1 Testen in dieser Arbeit

In der vorliegenden Arbeit konzentriere ich mich auf den dynamischen Test, d. h. Testen unter dem Gesichtspunkt, dass das System unter Test mit beispielhaften Daten zu Ausführung gebracht wird. Dabei liegt der Fokus auf der Überprüfung des korrekten Verhaltens eines Systems, d. h. dem Konformitätstest. Es wird mit unterschiedlichen Eingabesequenzen ein Verhaltensmodell simuliert und die dabei erzeugten Systemläufe dienen als Grundlage für die Berechnung des korrekten Verhaltens (Testorakel). Da die Menge der möglichen Eingabesequenzen eines solchen Modells i. A. unendlich groß ist, werden relevante Sequenzen bezüglich

einer Testspezifikation ausgewählt. Die so erzeugten Testfälle werden anschließend auf dem realen System ausgeführt und die Ausgaben der Implementierung werden so mit den Ausgaben des Verhaltensmodells bei gleichen Eingaben verglichen. Sowohl die angesprochenen Softeware-in-the-Loop Tests, als auch die angesprochenen Hardware-in-the-Loop Tests sind mit dem in dieser Arbeit entwickelten Ansatz denkbar. Die Testfälle haben dabei speziell die Aufgabe, das System mit Eingabedaten zu versehen und die Reaktionen des Systems zu bewerten. Die hier betrachtete Klasse der Eingaben sind sporadisch auftretende Ereignisse, so wie sie z. B. von einem Nutzer des Systems erzeugt werden könnten. Es werden keine zyklischen Eingaben betrachtet, so wie sie z. B. von Hardwaresensoren erzeugt werden könnten.

Bezüglich der automatisierten Testfallableitung verfolge ich einen modellbasierten Ansatz. Dies ist dadurch begründet, dass bei den betrachteten Systemen neben den Eingabewerten auch der interne Zustand Einfluss auf die Ausgaben bzw. das Systemverhalten hat. In solchen Fällen muss in einem Testfall bzw. bei der Testfallableitung die Historie des Systems berücksichtigt werden. Da bei einem Black-Box Ansatz das System unter Test lediglich an den Kontrollpunkten stimuliert und an den Beobachtungspunkten beobachtet werden kann (*Points of Control and Observation*, PCO), kann die Testbewertung ausschließlich auf Basis der Beobachtungen erfolgen. Somit lässt sich die von mir betrachtete Vorgehensweise als funktionsorientiertes bzw. spezifikationsbasiertes dynamisches Prüfverfahren bezeichnen [61]. Entscheidend ist, dass die Testfälle auf Basis der Spezifikation und nicht auf Basis des Systems unter Test abgeleitet werden. Damit die Testfälle automatisiert abgeleitet werden können muss die Spezifikation in einer formalen Form vorliegen.

Folglich ist die Ausgangsbasis für die Testfallerzeugung ein Modell des Systems unter Test. Dieses Modell ist durch eine Zustandsmaschine, die das diskrete Systemverhalten beschreibt, gegeben. Alexander Pretschner diskutiert in [16] die unterschiedlichen Möglichkeiten, wie Modelle für die Ableitung von Testfällen und für die Ableitung von Code benutzt werden können und welche Abhängigkeiten bzw. Probleme sich daraus ergeben. So ist es z. B. offensichtlich, dass die Nutzung eines einziges Modells für beide Aspekte bedeuten würde, die Implementierung gegen sich selbst zu testen. Was getestet werden würde, wäre der Code- bzw. der Testfallgenerator oder die Annahmen, die bezüglich der Umgebung getroffen wurden. Dasselbe gilt, wenn der Code per Hand erzeugt wurde, jedoch das Modell für die Testfallerzeugung aus diesem Code abgeleitet wurde. Sinnvolle Tests können erst dann durchgeführt werden, wenn Spezifikation und Modell der Implementierung relativ unabhängig voneinander erstellt werden. Je unabhängiger diese Modelle sind, desto größer ist i. A. auch die Wahrscheinlichkeit, dass Fehlverhalten aufgedeckt werden kann. Dieses Szenario ermöglicht es, dass die Codeableitung und die Testableitung und -auswertung automatisch erfolgen können. Anwendung findet dies sehr häufig in der Automobilindustrie, wo die Spezifikation üblicherweise vom Zulieferer erstellt wird und das Testmodell vom Auftraggeber. Dadurch sind beide Modelle unabhängig voneinander. Im Allgemeinen hängt somit der Grad der Automatisierung und die Aussagekräftigkeit des Testansatzes eng mit der Verbindung von Testmodell und Spezifikation in Verbindung mit der automatischen Ableitung von Testfällen und Code zusammen.

Für die modellbasierte Beschreibung des Systemverhaltens stehen in der Informatik eine Vielzahl an unterschiedlichen Formalismen zur Verfügung. Automaten sind Notationen für dynamisches Verhalten und basieren auf unterschiedlichen mathematischen Modellen. Zentral sind in all diesen Formalismen die Zustände (Konfigurationen) und die Übergänge zwischen diesen Zuständen (Transitionen). Die Zustände kapseln dabei unterschiedliche Informationen

über den internen Status eines Systems. Mit den Übergängen wird das dynamische Verhalten als Reaktion auf Eingaben beschrieben. Das bedeutet, dass auf Basis des aktuellen Zustands und einer Eingabe die Reaktion des Systems bestimmt und ein Übergang in den nächsten Zustand vorgenommen wird. Je nach verwendetem Formalismus können dabei (unterschiedliche) Aktionen ausgeführt werden. Im Folgenden werde ich einige Formalismen zur Beschreibung des Systemverhaltens kurz vorstellen, um eine Einordnung der in dieser Arbeit verwendeten Zustandsmaschinen zu ermöglichen.

2.3.2 Endliche Automaten

Die einfachste Form verwendeter grafischer Formalismen sind endliche Automaten (*finite state machines*). Endliche Automaten sind mathematische Modelle des Systems und setzen sich aus einer festen, endlichen Anzahl von Zuständen zusammen. Diese beschreiben den Zustand des Systems, der sich aus den bisherigen (diskreten) Eingaben ergeben hat. Auf Basis des aktuellen Zustands und einer Eingabe erfolgt die Bestimmung des Folgestatus. Dabei unterscheidet man häufig zwei verschiedene Formen [60]. Zum einen die deterministischen Automaten und zum anderen die nichtdeterministischen Automaten. Bei letzteren können sich aus einem Zustand für eine bestimmte Eingabe mehrere Folgezustände ergeben, wodurch der Folgezustand nicht mehr eindeutig bestimmt werden kann. Endliche Automaten ohne Ausgaben werden auch Medwedjew-Automaten genannt. Das Konzept der endlichen Automaten lässt sich zu endlichen Automaten mit (diskreten) Ausgaben erweitern. Diese werden als Moore- oder Mealy-Automaten bezeichnet. Bei Moore-Automaten sind die Ausgaben eine Funktion des Zustands, hängen somit nicht direkt von den Eingaben ab. Bei Mealy-Automaten sind die Ausgaben eine Funktion von Zustand und Eingaben. Es können also direkte Abhängigkeiten zwischen Ein- und Ausgaben modelliert werden. Die Verallgemeinerung der Endlichen Automaten auf unendlich große Zustands- und Aktionsmengen nennt man Transitionssysteme (*transition systems*).

Problematisch bei allen endlichen Automaten ist, dass sie bei größeren Modellen extrem unübersichtlich werden. Sie besitzen keine explizite Parallelität, da zu einem Zeitpunkt immer nur ein Zustand aktiviert sein kann. Die Modellierung nebenläufigen Verhaltens erfordert daher eine exponentiell ansteigende Anzahl von Zuständen. Weiterhin ist die Wiederverwendung von Modellen und die Zusammenfassung von verschieden Modellen aufwändig, da dies im schlimmsten Fall exponentielle Komplexität hat.

2.3.3 Statecharts

Die Erweiterung der endlichen Automaten um Hierarchie, Parallelität und globale Kommunikation nennt man Statecharts. Statecharts wurden 1987 von David Harel für die Beschreibung von ereignisgesteuerten, kontrolldominierten Systemen vorgestellt [32] und stellen eine Vereinheitlichung einer Vielzahl von Formalismen für Transitionssysteme dar.

Statecharts erlauben es, die Komplexität großer Systeme mittels endlicher Automaten handhabbar zu machen. Dabei ermöglicht die Einführung von Hierarchie, Zustände durch weitere Zustandsautomaten zu verfeinern und die parallele Komposition von Zustandsautomaten ermöglicht es, sehr einfach nebenläufiges Verhalten auszudrücken und eine exponen-

tielle Zustandsexplosion, zumindest grafisch, zu vermeiden. Globale Kommunikation erfolgt über interne Nachrichten, die im gesamten Statechart sichtbar sind. Aus diesem Grund werden Statecharts mitunter auch als kommunizierende endliche Automaten bezeichnet. Ein- und Ausgaben und die internen Nachrichten werden konzeptionell unter dem Begriff Ereignisse zusammengefasst. Neben der Erzeugung von Ereignissen gibt es auch die Möglichkeit, einen dem Statechart unterlegten Datenraum zu lesen und zu manipulieren. Das Erzeugen von Ereignissen und das Lesen und Manipulieren eines Datenraums kann in den Aktionen des Statecharts verwendet werden. Aktionen können an Transitionen oder innerhalb von Zuständen annotiert werden. Aktionen, die an Zuständen annotiert sind, können wahlweise beim Betreten, beim Verlassen oder kontinuierlich, d. h. während sich das System in diesem Zustand befindet, ausgeführt werden. An Transitionen werden neben einer Menge von Aktionen auch das auslösende Ereignis und optional eine Schaltbedingung annotiert. Mittels temporallogischer Ausdrücke kann außerdem zeitliches Verhalten spezifiziert werden. Eine Schaltbedingung dient zu feingranularen Beschreibung, unter welchen Bedingungen die Transition schalten soll. In den Schaltbedingungen kann ebenfalls der Datenraum ausgelesen werden. Die Synchronisation innerhalb eines Statecharts erfolgt entweder durch gemeinsame Ereignisse, durch gemeinsame Daten oder durch Statusüberwachung. Es ist möglich, dass Transitionen die Grenzen von Zuständen überschreiten. Solche Transitionen werden Inter-Level-Transitionen oder auch Multi-Level-Transitionen genannt. Die von Harel eingeführten History-Konnektoren speichern beim Verlassen eines Zustandsautomaten den letzten aktiven Zustand. Wird dieser Zustandsautomat erneut betreten, dann wird initial der gespeicherte Zustand aktiviert.

Statecharts sind ein weit verbreiteter Formalismus zur Beschreibung von (synchronen) Systemen. Für die klassischen Statecharts wurden diverse Semantiken und Werkzeuge [32, 36, 117, 64] entwickelt. Eine kanonische formale Semantik existiert jedoch nicht.

2.3.4 Zustandsmaschinen

Die UML-Zustandsmaschinen stellen eine objekt-orientierte Erweiterung der Harel-Statecharts dar. Ebenso wie Statecharts, sind auch Zustandsmaschinen hierarchische Graphen, deren Knoten einfache oder zusammengesetzte Zustände darstellen. Mit Hilfe der zusammengesetzten Knoten ist es möglich, Zustandsmaschinen beliebig sequentiell oder parallel zu komponieren. Die Kanten des Zustandsgraphen repräsentieren auch hier die Transitionen zwischen den Zuständen und die Aktionen setzen sich aus Manipulationen eines Datenraums, dem Verschicken neuer Ereignisse oder dem Aufruf von Operationen zusammen.

Die Semantik der UML-Zustandsmaschinen basiert auf der sog. Statemate-Semantik [36, 34], wobei einige Modifikationen vorgenommen wurden, um sie in den objekt-orientierten Kontext einzupassen. Eine Zustandsmaschine zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann, und gibt an, aufgrund welcher Ereignisse Zustandsänderungen stattfinden. Innerhalb der UML werden Zustandsmaschinen entweder zur Beschreibung des diskreten Verhaltens eines Systems oder einer Systemkomponente (*behavioral state machine*) oder zur Beschreibung der zulässigen Nutzung einer Systemschnittstelle oder des Lebenszyklus eines Objektes verwendet (*protocol state machine*).

Im Unterschied zu den Statecharts von David Harel erfolgt die Verarbeitung der Ereignisse bei Zustandsmaschinen asynchron. Dies bedeutet, dass immer genau ein Ereignis vollständig

verarbeitet wird. Die empfangenen Ereignisse werden dafür bis zu ihrer Verarbeitung in einem Ereignisspeicher zwischengespeichert.

2.4 Unified Modeling Language

Bei der Entwicklung komplexer Softwaresysteme spielt die geeignete Erfassung und Definition der Anforderungen eine zentrale Rolle. Dabei müssen heutzutage neben den benötigten strukturellen Eigenschaften auch Anforderungen an das Verhalten beschrieben werden. Ablaufszenarien, die erwartete oder verbotene Abläufe des Systems beschreiben, sind eine weit verbreitete Methode für die Beschreibung solchen Verhaltens. Zustandsmaschinen sind eine weitere Methode für die Beschreibung von Verhalten. Bekannte Beispiele für Entwurfsnotationen die beide Ansätze in sich vereinen, sind die *Unified Modeling Language* (UML) [110] mit ihren Sequenzdiagrammen und Zustandsmaschinen sowie die *Specification and Description Language* [51] mit den *Message Sequence Charts* [50].

Die in dieser Arbeit verwendeten Zustandsmaschinen stellen eine Teilmenge der UML dar. Aus diesem Grund werde ich im Folgenden einen kurzen Überblick über die Konzepte und Diagrammtypen der UML geben. Eine ausführliche Beschreibung kann in [110, 111, 11] nachgelesen werden.

Als Ende der 80er Jahre die objektorientierten Programmiersprachen immer weitere Verbreitung fanden, wurden parallel auch viele verschiedene Konzepte für die Beschreibung objektorientierter Systeme entwickelt. Die Menge an unterschiedlichen Notationen und Methoden behinderte jedoch den erfolgreichen Einsatz in der industriellen Praxis. Grady Booch, Ivar Jacobson und James Rumbaugh führten deshalb im Unternehmen *Rational Software* Anfang der 90er Jahre verschiedene Notationssysteme (darunter OOSE, RDD, OMT, OBA, OODA, SOMA, MOSES, OPEN/OML) zu einem einheitlichen Konzept, der UML, zusammen. Seit 1997 wird die UML von der Object Management Group als Standard gepflegt und weiterentwickelt. Sie wurde 2005 durch eine grundlegend überarbeitete Version, der UML 2 [110], ersetzt und liegt seit 2007 in der Version 2.1.1 [111] vor. Die UML ist mittlerweile in der industriellen Entwicklung etabliert.

Die UML ist eine grafische Notation, die auf dem Prinzip der Objektorientierung basiert und aus einer Menge von Diagrammen zur Visualisierung, Spezifikation, Erstellung und Dokumentation von Struktur und Verhalten eines objektorientierten verteilten Systems besteht. Mit ihr werden die in der Analysephase identifizierten Artefakte spezifiziert und zueinander in Beziehung gesetzt und dann in der Entwurfsphase die statische Struktur und das dynamische Verhalten des Systems entworfen. Dabei werden vornehmlich Klassen und ihre Beziehungen, sowie die Ablaufstrukturen innerhalb eines Objektes bzw. die Kommunikation zwischen verschiedenen Objekten beschrieben. Die entworfenen Diagramme bilden in der Implementierungsphase sehr häufig die wesentliche Grundlage für die Realisierung.

Die Version UML 2 [110, 111] besteht aus zwei Teilspezifikationen, der *Infrastructure Specification* und der *Superstructure Specification*. In der *Infrastructure Specification* wird die Spracharchitektur und der Spezifikationsansatz beschrieben und sie dient als Fundament für den zweiten Teil. In der *Superstructure Specification* werden darauf aufbauend die Sprachkonzepte auf Nutzungsebene beschrieben. Sie beschreibt den eigentlichen (anwendbaren) Spezifi-

kationsformalismus. Im Sinne einer Sprache werden Bezeichner für die verwendeten Begriffe und mögliche Beziehungen zwischen diesen Begriffen, sowie grafische Notationen für diese Begriffe und für Modelle von statischen Strukturen und von dynamischen Abläufen, die man mit diesen Begriffen formulieren kann, festgelegt. *UML Profiles* passen die UML an spezifische Anwendungsgebiete oder Technologien an und die der UML zugeordneten Spezifikationen *Object Constraint Language* und *Diagram Interchange* ergänzen die UML.

Die UML enthält insgesamt dreizehn Diagrammtypen, die sich auf sechs Strukturdiagramme und auf sieben Verhaltensdiagramme aufteilen. Zu den Strukturdiagrammen zählen die *Class Diagrams*, die *Composite Structure Diagrams*, die *Component Diagrams*, die *Deployment Diagrams*, die *Object Diagrams* und die *Package Diagrams*. Zu den Verhaltensdiagrammen zählen die *Activity Diagrams*, die *Interaction Diagrams*, die *Use Case Diagrams* und die *State Machine Diagrams*, wobei sich die *Interaction Diagrams* in *Sequence Diagrams*, *Communication Diagrams*, *Interaction Overview Diagrams* und *Timing Diagrams* aufteilen.

Die statische Struktur eines Systems wird hauptsächlich durch Klassendiagramme und Objektdiagramme beschrieben. Die Basiselemente dieser Diagrammtypen sind Klassen und Relationen bzw. Objekte und Links. Eine Klasse besitzt eine Menge von Attributen, deren Belegung den Zustand ihrer Objekte ausmacht. Das Verhalten einer Klasse wird durch eine Menge von Methoden und Signalen, die auf Instanzen durch Aufruf- und Sendeaktionen aufgerufen werden können, beschrieben. Klassendiagramme spezifizieren dabei Typen und Schnittstellen des Problembereichs, während Objektdiagramme Instanzen dieser Klassen spezifizieren und somit als Instanzen von Klassendiagrammen angesehen werden können. Kompositionsstrukturdiagramme zeigen die innere Struktur von z. B. Klassen oder Systemteilen. Komponentendiagramme spezifizieren die Struktur der Systemkomponenten und ihre Beziehung untereinander und die Deployment Diagramme zeigen, wie die Komponenten auf entsprechender Hardware verteilt werden. Schließlich erlaubt das Paketdiagramm Modellelemente zur besseren Übersicht zu strukturieren.

Das dynamische Verhalten des Systems wird mit den Verhaltensdiagrammen beschrieben. Use Case Diagramme zeigen die Beziehungen zwischen Aktoren, d. h. Benutzern des Systems oder anderen Systemen, und den Kernfunktionen des Systems. Sequenzdiagramme beschreiben die Interaktionen, d. h. den Austausch von Nachrichten, zwischen unterschiedlichen Objekten. Dabei können die Verbindungen zwischen den Objekten in Kommunikationsdiagrammen (früher Kollaborationsdiagrammen) dargestellt werden. Das dynamische Verhalten von Objekten bzw. die Interaktion zwischen verschiedenen Objekten wird mit Zustandsmaschinen auf Basis der internen Struktur und der Reaktion auf Ereignisse beschrieben. Mit Hilfe von Zustandsmaschinen lassen sich alle möglichen Sequenzen von Zuständen und Aktionen beschreiben, die ein Objekt im Laufe seines Lebenszyklus infolge von auftretenden internen und externen Ereignissen durchläuft bzw. ausführt. Aktivitätsdiagramme beschreiben die Vernetzung von elementaren Aktionen und deren Verbindungen mit Kontroll- und Datenflüssen. Zumeist wird so der Ablauf eines Anwendungsfalls beschrieben. Zeitdiagramme werden zur Visualisierung komplexer zeitlicher Zusammenhänge benutzt. Eine Übersicht über alle Interaktionen bzw. ein Gesamtbild der Interaktionen des Systems kann schließlich mit Interaktionsübersichtsdiagrammen beschrieben werden.

2.5 Problembereich

Die Unified Modeling Language findet in der modernen Softwareentwicklung breite Anwendung und hat sich insbesondere in der industriellen Praxis etabliert. Dabei wird das zu entwickelnde System mit Hilfe unterschiedlicher Diagrammtypen auf verschiedenen Abstraktionsebenen und unter unterschiedlichen Gesichtspunkten beschrieben. Die einzelnen Modelle dienen nicht nur der Dokumentation des zu entwickelnden Systems und als Grundlage für Diskussionen innerhalb des Entwicklungsteams, sie tragen auch wesentlich zum Systemverständnis der Entwickler bei. Im Kontext der modell-getriebenen Softwareentwicklung werden die Modelle mittlerweile so stark in den Entwicklungsprozess integriert, dass sogar Teile des Programmcodes automatisch aus den Modellen abgeleitet werden.

Obwohl die Modelle der UML immer stärker in das Zentrum moderner Softwareentwicklungsprozesse rücken, fehlt es an einer rigorosen Einbeziehung in die Qualitätssicherung. Die Anwendung der Modelle der UML in der Qualitätssicherung wäre dabei in zweierlei Hinsicht vorteilhaft. Zum einen könnten sie als Basis für eine Automatisierung der Qualitätssicherung dienen, z.B. indem Testfälle zur Prüfung des Systems automatisch aus den Modellen abgeleitet und gegen das System ausgeführt werden. Zum anderen würden die Modelle bei einer verstärkten Einbeziehung in die Qualitätssicherung ständig auf Konsistenz überprüft und aktualisiert und würden somit jederzeit den aktuellen Stand des zu entwickelnden Systems widerspiegeln. Dieser Aspekt ist nicht zu unterschätzen, da aus vielen Entwicklungsprojekten bekannt ist, dass mit voranschreitender Projektlaufzeit die Schere zwischen Dokumentation und Entwicklung immer weiter auseinander geht.

Zustandsmaschinen werden im Softwareentwicklungsprozess besonders häufig eingesetzt und sind im Hinblick auf die Qualitätssicherung ein besonders interessantes Modell der UML, da sie das vollständige dynamische Verhalten eines Systems oder einer Systemkomponente beschreiben. Mit UML-Zustandsmaschinen wurde das Konzept der Automaten aufgegriffen. Sie bilden damit eine geeignete Basis für die Ableitung von Testfällen zur Überprüfung asynchroner, reaktiver Systeme, die im Fokus dieser Arbeit stehen.

In der vorliegenden Arbeit wird die Qualitätssicherung auf Basis von UML-Zustandsmaschinen thematisiert. Zustandsmaschinen sind insbesondere geeignet, dass dynamische Verhalten des Systems oder einzelner, spezieller Komponenten präzise zu beschreiben. Sie bilden daher eine ideale Ausgangsbasis für die Überprüfung des entwickelten Systems. Was dies bisher verhindert ist, dass ihnen keine vollständig präzise Semantik zugrunde liegt und folglich die Modelle nur unzureichend in der Qualitätssicherung eingesetzt werden können. Die Komplexität von modernen Systemen, insbesondere die Komplexität eingebetteter, reaktiver Systeme erfordert es jedoch, dass für eine umfassende und systematische Prüfung des Systems unter Test eine automatisierte Vorgehensweise entwickelt wird. Dafür ist eine präzise Semantik zwingende Voraussetzung. Aus dieser Motivation ergeben sich für die vorliegende Arbeit folgende Aufgaben:

1. Die Semantik der Zustandsmaschinen muss präzisiert werden, um Zustandsmaschinen als Grundlage für eine automatisierte Ableitung von Testfällen benutzen zu können. Die Formalisierung ist zum einen notwendige Voraussetzung für die Automatisierung und zum anderen hilft sie dabei, das tiefgehende Verständnis für Zustandsmaschinen bei den Softwareentwicklern zu erhöhen. (Kapitel 3 auf Seite 25).

2. Auf Basis der formalen Semantik von Zustandsmaschinen muss ein Testansatz entwickelt werden, der es ermöglicht, ein System unter Test gegen die Spezifikation, die in Form einer Zustandsmaschine vorliegt, umfassend und systematisch zu prüfen. Dabei liegt der Fokus auf der Prüfung von Konformität, d. h. speziell, dass bezüglich des Ein-/Ausgabeverhaltens geprüft werden soll, ob das System unter Test die in der Spezifikation gestellten Anforderungen erfüllt. (Kapitel 4 auf Seite 53, Abschnitte 4.1, 4.2 und 4.4).

3. Die betrachteten Systeme sind sehr komplex. Folglich ist der Aufwand extrem hoch, der für die Berechnung eines einzelnen Testfalls aufgebracht werden muss. Zudem ist im Allgemeinen die Anzahl der möglichen Testfälle unendlich groß, die zur vollständigen Überprüfung des Systems unter Test benötigt würden. Daher müssen neben dem Testansatz auch geeignete Strategien zur apriori Auswahl von Testfällen entwickelt werden. (Kapitel 4 auf Seite 53, Abschnitt 4.3).

4. Schließlich muss der entwickelte Testansatz praktisch evaluiert werden, um seine Praktikabilität und Effektivität beurteilen bzw. abschätzen zu können. (Kapitel 5 auf Seite 75 und Kapitel 6 auf Seite 89).

Aufgrund der weiten Verbreitung der UML und den enormen Vorteilen, die ihr Einsatz in der Qualitätssicherung in Aussicht stellt, besteht ein großes Anwendungspotential für die Ergebnisse der Arbeit. Die angesprochenen Aufgaben bilden die Arbeitspakete für die folgenden Kapitel.

Kapitel 3

Zustandsmaschinen mit Daten

In diesem Kapitel beschreibe ich die abstrakte Syntax und die Semantik des in der vorliegenden Arbeit formalisierten und in der Werkzeugumgebung TEAGER umgesetzten Ausschnitts von UML-Zustandsmaschinen. Die vollständige und präzise Formalisierung liefert die benötigten formalen Grundlagen, um auf Basis von Zustandsmaschinen automatisiert Testfälle für einen Konformitätstest ableiten zu können. In Abschnitt 3.1 werde ich zunächst begründen, worin die Vorteile einer formalen Semantik für meine Arbeit bestehen und aus welchen Gründen ich mich entschlossen habe, eine eigene Formalisierung anzugeben. Danach werde ich in Abschnitt 3.2 den von mir verwendeten Ausschnitt von UML-Zustandsmaschinen vorstellen. An einem Beispiel werde ich die Bestandteile und den generellen Aufbau informell beschreiben. In Abschnitt 3.3 werde ich dann die abstrakte Syntax von Zustandsmaschinen formal einführen und in Abschnitt 3.4 darauf basierend eine operationale Semantik angeben. Die Ausführung einer Zustandsmaschine erfolgt durch schrittweise Übergänge zwischen semantischen Zuständen. Aus diesem Grund werde ich zunächst die einzelnen Bestandteile eines semantischen Zustands beschreiben und dann definieren, wie der Übergang von einem semantischen Zustand zum nächsten semantischen Zustand erfolgt. Abschließend werde ich in Abschnitt 3.5 Anmerkungen zu möglichen Erweiterungen geben und in Abschnitt 3.6 die Ergebnisse des Kapitels zusammenfassen.

3.1 Einleitung

In der vorliegenden Arbeit konzentriere ich mich auf das Testen von Steuerungssoftware eingebetteter reaktiver Systeme. Unter einem eingebetteten System versteht man ein aus Hardware- und Softwarekomponenten bestehendes System, das in eine spezifische und zumeist feste Umgebung, auch Kontext genannt, eingebettet ist. Reaktiv nennt man ein solches System, wenn es in Abhängigkeit von Eingaben aus der Umgebung Veränderungen in dieser vornimmt. Die Eingaben werden auch Messgrößen genannt und beobachtet, die Ausgaben werden Stellgrößen genannt und kontrolliert [72]. Typische Eingaben sind Benutzerinteraktionen und Messdaten

von Sensoren, typische Ausgaben sind Signale zur Ansteuerung von Aktoren. Steuergeräte in Automobilen, z. B. ABS oder ESP, und Mobiltelefone sind typische Beispiele für eingebettete reaktive Systeme.

Bei der Entwicklung eingebetteter reaktiver Systeme werden zu deren Verhaltensbeschreibung sehr häufig grafische Notationen verwendet. Diese haben den Vorteil, dass sie von einem Menschen sehr schnell erfasst und somit in der Entwicklung als ideale Informationsquelle benutzt werden können. Zustandsmaschinen sind eine solche grafische Notation der UML, mit der das diskrete Verhalten von Systemkomponenten beschrieben werden kann. Ich habe Zustandsmaschinen als Beispielnotation ausgewählt, um zu untersuchen, wie auf Basis einer Zustandsmaschine automatisiert Testfälle für einen Konformitätstest abgeleitet werden können.

Damit aus Zustandsmaschinen automatisiert Testfälle abgeleitet werden können, muss ihnen eine formale Semantik zugrunde gelegt werden. Nur eine formale Semantik ermöglicht es, das Verhalten von Zustandsmaschinen zweifelsfrei zu bestimmen und dadurch korrekte Testfälle abzuleiten. Im UML-Standard [110] werden Zustandsmaschinen größtenteils natürlichsprachlich beschrieben, wobei die abstrakte Syntax noch recht präzise jedoch die operationale Semantik nur lose, d. h. ungenau und unvollständig, beschrieben wird. Zum Beispiel wurden in [27] UML-Zustandsmaschinen hinsichtlich vorhandener Unklarheiten untersucht und die daraus entstehenden Probleme beschrieben. Die Autoren führen 21 Unstimmigkeiten des Standards auf und diskutieren mögliche Lösungsansätze. Auch meine Erfahrungen mit dem UML-Standard und mit kommerziellen Werkzeugen zeigen, dass der Standard nicht präzise genug ist, um auf seiner Basis automatisiert arbeiten zu können.

Meine Literaturrecherche zur Semantik von Zustandsmaschinen lieferte eine Fülle an Arbeiten, von denen ich mir einige interessante herausgesucht und analysiert habe. Einen Überblick über diverse Semantiken kann man sich in [97] verschaffen. Die Untersuchung ergab, dass der größte Teil der bestehenden Arbeiten Zustandsmaschinen in sehr speziellen Anwendungsgebieten, wie z. B. dem Modellprüfen, verwenden. Die Art und Weise, wie einzelne Semantiken präsentiert werden, ist folglich sehr speziell und dadurch für meine Zwecke ungeeignet. Sehr häufig erfolgt eine Formalisierung der Semantik in der existierenden Logik des Beweisverfahrens, was eine allgemeine Anwendbarkeit zusätzlich erschwert. Selten wird der gesamte Umfang der Zustandsmaschinen betrachtet und trotz Beschränkungen kommt es vor, dass die angegebenen Definitionen unvollständig oder unpräzise sind. Nur wenige Arbeiten befassen sich mit der Integration von komplex strukturierten Daten.

Ziel meiner Arbeit ist es, dass Verhalten von Systemkomponenten mit Zustandsmaschinen zu beschreiben, um auf Basis der Zustandsmaschinen automatisiert Testfälle für einen Konformitätstest abzuleiten. Dabei sollen nicht nur die Eingaben an die Software, sondern auch - a priori - die Menge der möglichen korrekten Ausgaben bestimmt werden. Diese berechneten Ausgaben (Soll-Werte) dienen dann während der Testausführung zur Beurteilung der gemachten Beobachtungen (Ist-Werte). Daraus ergeben sich zwei wesentliche Anforderungen an eine Semantik für Zustandsmaschinen. Zum einen muss die Semantik zumindest den Teil der Zustandsmaschinen umfassen, bezüglich dessen ich die Testfallableitung untersuchen möchte. Zum anderen muss die Semantik den verwendeten Ausschnitt präzise und vollständig beschreiben. Da keine der untersuchten Arbeiten beide Anforderungen in geeigneter Weise erfüllt, habe ich mich entschlossen, einen Ausschnitt der UML-Zustandsmaschinen selbst zu formalisieren. Als Ausgangsbasis habe ich die Arbeiten [4, 62, 58] näher untersucht und mei-

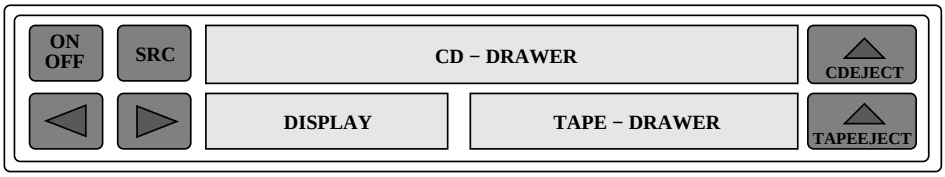


Abbildung 3.1: Vereinfachtes Bedienfeld der Musikanlage.

ne Formalisierung darauf gestützt. Der von mir betrachtete Ausschnitt umfasst den Kern der Zustandsmaschinen, d. h. alle wesentlichen strukturellen Ausdrucksmittel, sowie komplex strukturierte Daten. Insbesondere die Integration der Daten geht über die analysierten Arbeiten hinaus. Mit Zustandsmaschinen bezeichne ich im Folgenden den von mir betrachteten Ausschnitt aus den UML-Zustandsmaschinen.

3.2 Beispiel

In diesem Abschnitt werde ich Zustandsmaschinen an einem Beispiel informell einführen. Das verwendete Beispiel beschreibt eine einfache Musikanlage, so wie sie z. B. in einem Automobil eingebaut werden könnte. Möchte der Autofahrer Musik hören, dann hat er die Wahl, entweder Radio, eine Kassette oder eine CD zu hören. Der Wechsel in den Kassetten- bzw. in den CD-Modus ist nur möglich, wenn auch eine Kassette bzw. eine CD eingelegt ist. Abbildung 3.1 zeigt das vereinfachte Bedienfeld der Musikanlage. Die Musikanlage kann vom Autofahrer über einen `ON/OFF` Taster ein- und wieder ausgeschaltet werden. Mit einem `SRC` Taster kann der Autofahrer die unterschiedlichen Musikquellen anwählen. Weiterhin kann der Autofahrer in Abhängigkeit von der gewählten Musikquelle über zwei Taster ($\triangleleft$, $\triangleright$) zwischen vier Radiosendern wählen, die Kassette vor- bzw. zurückspulen oder den vorhergehenden bzw. nachfolgenden Titel einer CD anwählen. Mit den Tastern `TAPEEJECT` und `CDEJECT` kann der Autofahrer eine Kassette bzw. eine CD auswerfen, die zuvor in den `TAPE-DRAWER` bzw. in den `CD-DRAWER` eingelegt wurden. Auf dem `DISPLAY` werden bei eingeschalteter Musikanlage Informationen zum Status der Musikanlage angezeigt.

Im folgenden Abschnitt stelle ich zunächst eine Spezifikation des Beispiels vor, in der keine Daten zur Modellierung benutzt werden. Stattdessen wird das Verhalten der Zustandsmaschine vollständig ereignisorientiert modelliert. In Abschnitt 3.2.2 stelle ich eine zweite Spezifikation vor, in der Daten zur Modellierung verwendet werden. In diesen Abschnitten werde ich Zustandsmaschinen und ihre Funktionsweise informell einführen. Eine detaillierte Beschreibung von Zustandsmaschinen gebe ich dann in Abschnitt 3.3 auf Seite 33 und Abschnitt 3.4 auf Seite 39.

3.2.1 Datenlose Spezifikation

Aus den oben beschriebenen Anforderungen ist die Zustandsmaschine `CarAudioSystem` entworfen worden, die das diskrete Verhalten der Musikanlage beschreibt. Abbildung 3.2 zeigt diese Zustandsmaschine. In einem Entwicklungsprozess könnte sie die Spezifikation des Systems darstellen und somit als Basis für die Entwicklung der Steuerungssoftware dienen. Zur

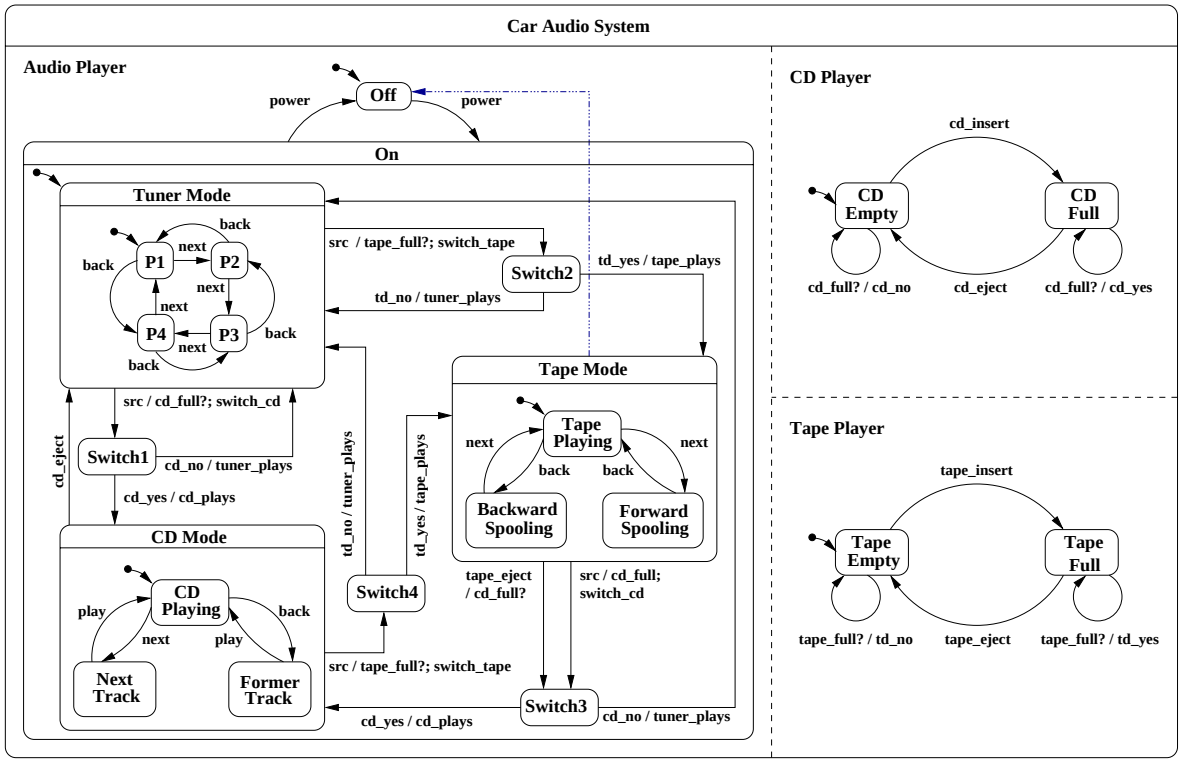


Abbildung 3.2: Datenlose Spezifikation der Musikanlage.

Qualitätssicherung können dann aus der Zustandsmaschine Testfälle abgeleitet werden und mit ihnen das konforme Verhalten der Software zur Spezifikation überprüft werden.[1]

Eine Zustandsmaschine ist ein Automat, dessen Zustände die Kontrollzustände des Systems darstellen. Zustände können mit Hilfe von Regionen hierarchisch und orthogonal strukturiert werden. Mittels Transitionen werden Übergänge zwischen den Zuständen beschrieben. Ein Zustandsübergang wird durch ein Ereignis ausgelöst und beim Übergang von einem Zustand in den nächsten Zustand wird der Effekt der Transition ausgeführt. Dabei wird der Quellzustand der Transition verlassen und der Zielzustand der Transition betreten, was auch mit Deaktivieren bzw. Aktivieren des Zustands bezeichnet wird. Eine Transition kann nur dann ausgeführt werden, wenn der Quellzustand der Transition aktiv ist. Jede Transition wird mit ihrem auslösenden Ereignis und optional mit einem Effekt beschriftet.

Die Zustandsmaschine in Abbildung 3.2 besteht aus dem orthogonalen Zustand `Car Audio System`, der durch die drei Regionen `Audio Player`, `CD Player` und `Tape Player` verfeinert wird. Die unterschiedlichen Regionen eines orthogonalen Zustands werden durch gestrichelte Linien voneinander abgetrennt. Allgemein gilt, dass ein orthogonaler Zustand durch mindestens zwei Regionen verfeinert werden muss. Die Region `Audio Player` enthält den Kontrollmechanismus der Musikanlage. Die beiden Regionen `CD Player` und `Tape Player` werden zur Speicherung benutzt, ob eine CD, respektive eine Kassette, in die Musikanlage eingelegt

[1] Ich gehe an dieser Stelle davon aus, dass die Software nicht automatisiert aus der Zustandsmaschine abgeleitet wurde. Probleme, die sich in diesem Zusammenhang ergeben würden, habe ich bereits in Abschnitt 2.3 diskutiert.

wurde. Wird ein Zustand, der durch mehrere Regionen verfeinert wird, aktiviert, dann werden auch alle ihn verfeinernden Regionen ebenfalls aktiviert - er befindet sich gleichzeitig in allen seinen Regionen. Da in jeder dieser Regionen zu einem Zeitpunkt genau ein Zustand aktiv ist, wird ein solcher Zustand auch als Und-Zustand bezeichnet.

In einer Region können nun wiederum Zustände enthalten sein. Die Region `CD Player` enthält zwei einfache Zustände: `CD Empty` und `CD Full`. Diese Zustände repräsentieren die beiden Kontrollzustände der Musikanlage, in denen keine oder eine CD eingelegt ist. Einfache Zustände sind Zustände, die nicht weiter verfeinert werden. Ein von einem ausgefüllten Punkt ausgehender Pfeil kennzeichnet innerhalb einer Region den initialen Zustand. Dieser wird aktiviert, sobald die Region aktiviert wird. In der Region `CD Player` sind vier Transitionen spezifiziert. Tritt das Ereignis ein, welches anzeigt das eine CD eingelegt wurde (`cd_insert`), wechselt die Zustandsmaschine vom Zustand `CD Empty` in den Zustand `CD Full`, wobei die Transition keinen Effekt hat, also keine Aktionen ausgeführt werden. Dabei wird der Zustand `CD Empty` deaktiviert und der Zustand `CD Full` aktiviert. Ähnliches passiert, wenn die CD wieder ausgeworfen wird. Mit Hilfe der beiden selbstbezüglichen Transitionen kann vom `Audio Player` abgefragt werden, ob eine CD eingelegt ist oder nicht. Über das Ereignis `cd_full?` wird entweder das Ereignis `cd_no` oder das Ereignis `cd_yes` als Effekt der Transition generiert. Dies kann im Folgenden vom `Audio Player` weiter verarbeitet werden. Das Verhalten der Region `Tape Player` ist in ihrer Funktionsweise identisch mit dem Verhalten der Region `CD Player`.

Die Region `Audio Player` wird durch zwei Zustände verfeinert. Der Zustand `Off`, der innerhalb der Region auch initial aktiv ist, kennzeichnet dabei den ausgeschalteten Zustand der Musikanlage. Wird vom Benutzer der `ON/OFF` Taster gedrückt und dadurch das Ereignis `power` ausgelöst, wird die Musikanlage eingeschaltet und die Zustandsmaschine wechselt in den Zustand `On`. Der Zustand `On` ist ein so genannter einfach zusammengesetzter Zustand. Einfach zusammengesetzte Zustände sind dadurch gekennzeichnet, dass sie durch genau eine Region verfeinert werden. Die Region des Zustands `On` enthält wiederum mehrere Zustände, von denen die Zustände `Switch1` bis `Switch4` einfache Zustände und die Zustände `Tuner Mode`, `CD Mode` und `Tape Mode` einfach zusammengesetzte Zustände sind. Einfach zusammengesetzte Zustände enthalten genau eine Region. Da in einer Region zu einem Zeitpunkt immer genau ein Zustand aktiv ist, werden einfach zusammengesetzte Zustände auch Oder-Zustände genannt.

Die modellierte Steuerung der Musikanlage spielt Radio, sobald sie eingeschaltet wird. Betätigt der Benutzer den `SRC` Taster, wird das Ereignis `src` generiert. Als Reaktion auf dieses Ereignis wechselt die Zustandsmaschine entweder in den Zustand `Switch1` oder in den Zustand `Switch2`. An dieser Stelle ist die Spezifikation - explizit - nichtdeterministisch, da für beide Transitionen die Quellzustände aktiv sind und das zu verarbeitende Ereignis beide Transitionen aktiviert. Somit könnten beide Transitionen ausgeführt werden. In einer Region darf aber nur genau ein Zustand aktiv sein. Daher ist es nicht erlaubt, dass beide Transitionen gleichzeitig ausgeführt werden. Stattdessen muss eine der beiden Transitionen ausgewählt werden. Nehmen wir beispielhaft an, dass die Transition ausgewählt wurde, die in den Zustand `Switch2` führt. Als Effekt der Transitionsausführung werden die Ereignisse `tape_full?` und `switch_tape` generiert. Letzteres führt zum Beispiel im `DISPLAY` zu einer Statusanzeige. Das Ereignis `tape_full?` wird dagegen benutzt, um den internen Zustand des `Tape Player`s abzufragen, d. h. zu prüfen, ob eine Kassette in die Musikanlage eingelegt wurde. Vom Zustand `Switch2` wechselt die Zustandsmaschine in Abhängigkeit von der Antwort, die in der Region `Tape Player` generiert wurde, entweder in den Zustand `Tape Mode` oder zurück in den

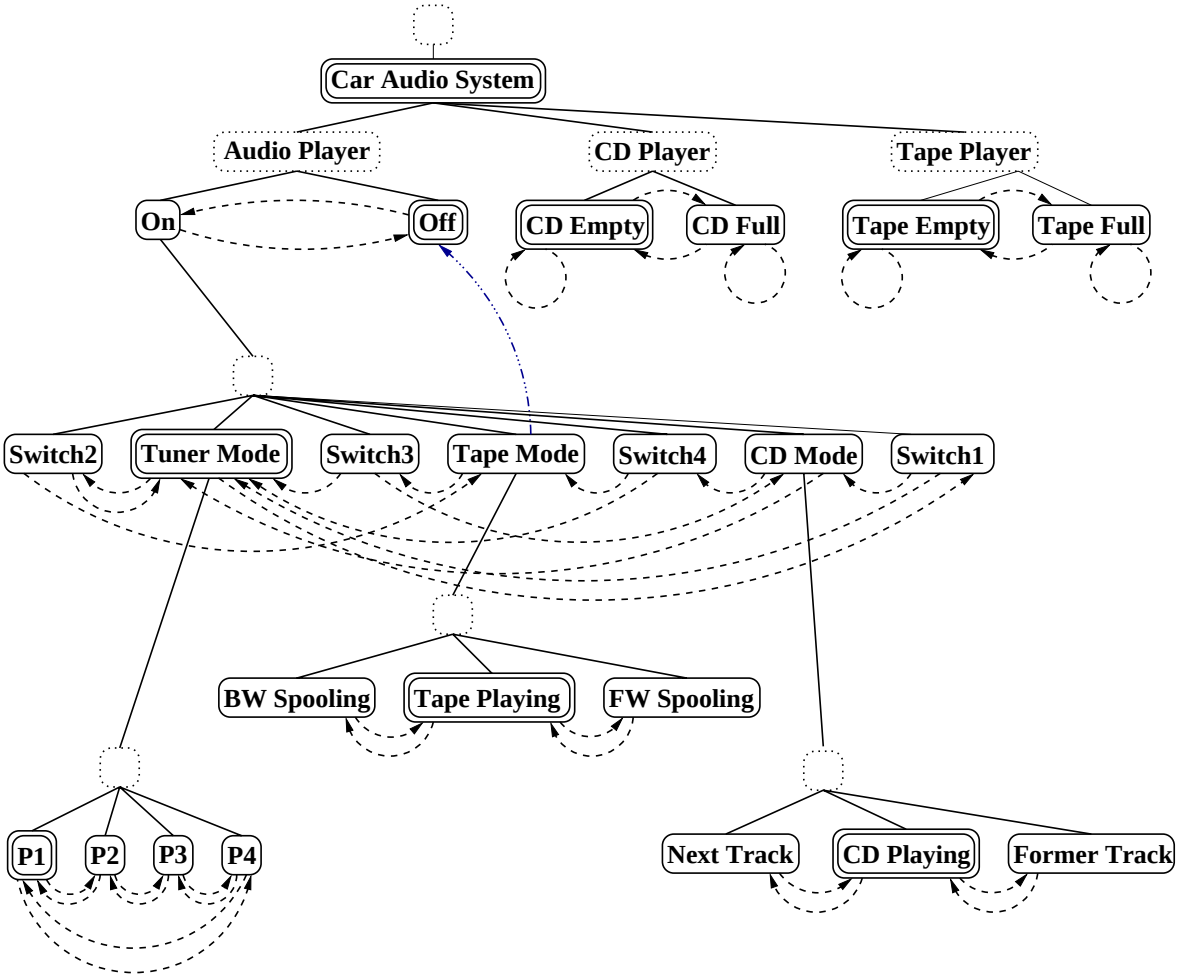


Abbildung 3.3: Baumstruktur der Zustandsmaschine der Musikanlage.

Zustand `TunerMode`. Dabei wird das Ereignis `tape_plays` bzw. `tuner_plays` generiert, die zu einer Aktualisierung des `DISPLAY`s führen könnten. Die Verarbeitung des Ereignisses `src` zeigt beispielhaft, wie in Zustandsmaschinen zwischen unterschiedlichen orthogonalen Regionen kommuniziert wird. Die weiteren Übergänge zwischen den Zuständen werden auf ähnliche Art und Weise verarbeitet und werden hier nicht im Detail beschrieben. Die Verfeinerungen der drei Modi spezifizieren die Reaktion auf die Ereignisse `play`, `next` und `back`.

Die relative Anordnung der Zustände und Regionen zueinander kann als Baumstruktur dargestellt werden. Abbildung 3.3 zeigt die Baumstruktur der Zustandsmaschine `CarAudioSystem`. Regionen sind in der Abbildung als gepunktete Knoten dargestellt und besitzen optional einen Namen. Zustände werden in einfach umrahmten Knoten dargestellt, wobei initiale Zustände doppelt umrahmt werden. Es gilt, dass der Wurzelknoten einer solchen Baumstruktur eine Region ist und dass sich Regionen und Zustände jeweils abwechseln. Weiterhin gilt, dass Zustände auf höheren Ebenen entweder einfach zusammengesetzte oder orthogonale Zustände sind. Dabei werden einfach zusammengesetzte Zustände durch genau eine Region und orthogonale Zustände durch mindestens zwei Regionen verfeinert. Die innersten Zustände einer Zustandsmaschine und somit auch die Blattknoten in einer solchen

Baumstruktur sind immer einfache Zustände. Weiterhin gilt, dass es innerhalb einer Region genau einen initialen Zustand gibt.

Die Transitionen der Zustandsmaschine `Car Audio System` sind in Abbildung 3.3 durch gestrichelte Pfeile dargestellt. Zur Übersichtlichkeit habe ich auf die Markierungen der Transitionen verzichtet. Durch die Darstellung der Zustandsmaschine in einer Baumstruktur ist gut erkennbar, welche Zustände beim Schalten einer Transition verlassen werden, und welche betreten werden müssen. Betroffen ist jeweils der Teilbaum der Baumstruktur, der den Quell- und den Zielzustand enthält. Die Transitionen des `Car Audio System` sind alle auf einer Hierarchieebene. Es ist jedoch auch möglich, Transitionen zu definieren, die die Grenzen von Zuständen überschreiten und damit unterschiedliche Hierarchieebenen betreffen. Solche Transitionen werden im englischem *multi level transitions* genannt. Zur Verdeutlichung habe ich in Abbildung 3.2 und 3.3 eine solche Transition hinzugefügt. Quellzustand dieser blau dargestellten Transition ist der Zustands `Tape Mode`. Zielzustands ist der Zustand `Off`. Mit Hilfe der Transition könnte z. B. modelliert werden, dass die Musikanlage am Ende einer Kassette automatisch ausgeschaltet wird. Auch hier kann man gut erkennen, welcher Teilbaum beim Schalten der Transition betroffen ist. Die Veranschaulichung einer Zustandsmaschine als Baumstruktur und die Bestimmung der Teilbäume, die durch das Schalten von Transitionen betroffen sind, helfen besonders in den Abschnitten 3.3 und 3.4, die formalen Definitionen einer Zustandsmaschine nachzuvollziehen.

Die Verarbeitung der Ereignisse in einer Zustandsmaschine ist atomar, d. h., dass mit der Verarbeitung des nächsten Ereignisses erst begonnen wird, wenn die Verarbeitung des vorherigen Ereignisses vollständig abgeschlossen ist. Ereignisse werden aus der Umgebung (externe Ereignisse) bzw. von der Zustandsmaschine selbst (interne Ereignisse) empfangen. Dabei werden die Ereignisse nicht sofort verarbeitet, sondern bis zu ihrer weiteren Verarbeitung in einem Ereignisspeicher abgelegt. Führt die Zustandsmaschine einen Schritt aus, dann entnimmt sie dafür ein Ereignis aus dem Ereignisspeicher, wählt die Transitionen aus, die in diesem Schritt schalten können und führt diese aus. Effekt der Ausführung eines solchen Schritts ist ein aktualisierter Datenraum und eine Sequenz von neu generierten Ereignissen, die entweder an die Zustandsmaschine oder an die Umgebung verschickt werden. Aufgrund der Zwischenspeicherung von Ereignissen bis zu ihrer Verarbeitung verhalten sich Zustandsmaschinen asynchron.

3.2.2 Datenbehaftete Spezifikation

Im diesem Abschnitt stelle ich eine Variante des Beispiels aus Abschnitt 3.2.1 vor, in dem komplex strukturierte Daten zur Spezifikation verwendet werden. Daten können in Zustandsmaschinen zum einen zur feingranularen Steuerung der Zustandsübergänge und zum anderen in komplex strukturierten Ereignissen verwendet werden. Jeder Zustandsmaschine liegt ein strukturierter Datenraum zugrunde, der während der schrittweisen Ausführung der Zustandsmaschine gelesen und manipuliert werden kann. Abbildung 3.4 zeigt eine zur Spezifikation der Musikanlage erstellte Zustandsmaschine, in der Daten benutzt werden.

Der Datenraum der Zustandsmaschine besteht aus den zwei booleschen Variablen `inCDFull` und `inTapeFull` und einer ganzzahligen Variable `trackCount`. In ihnen wird gespeichert, ob eine CD bzw. eine Kassette eingelegt wurde und wie viele Titel sich auf einer eingelegten CD befinden. Die booleschen Variablen werden im Zustand `On` benutzt, detailliert

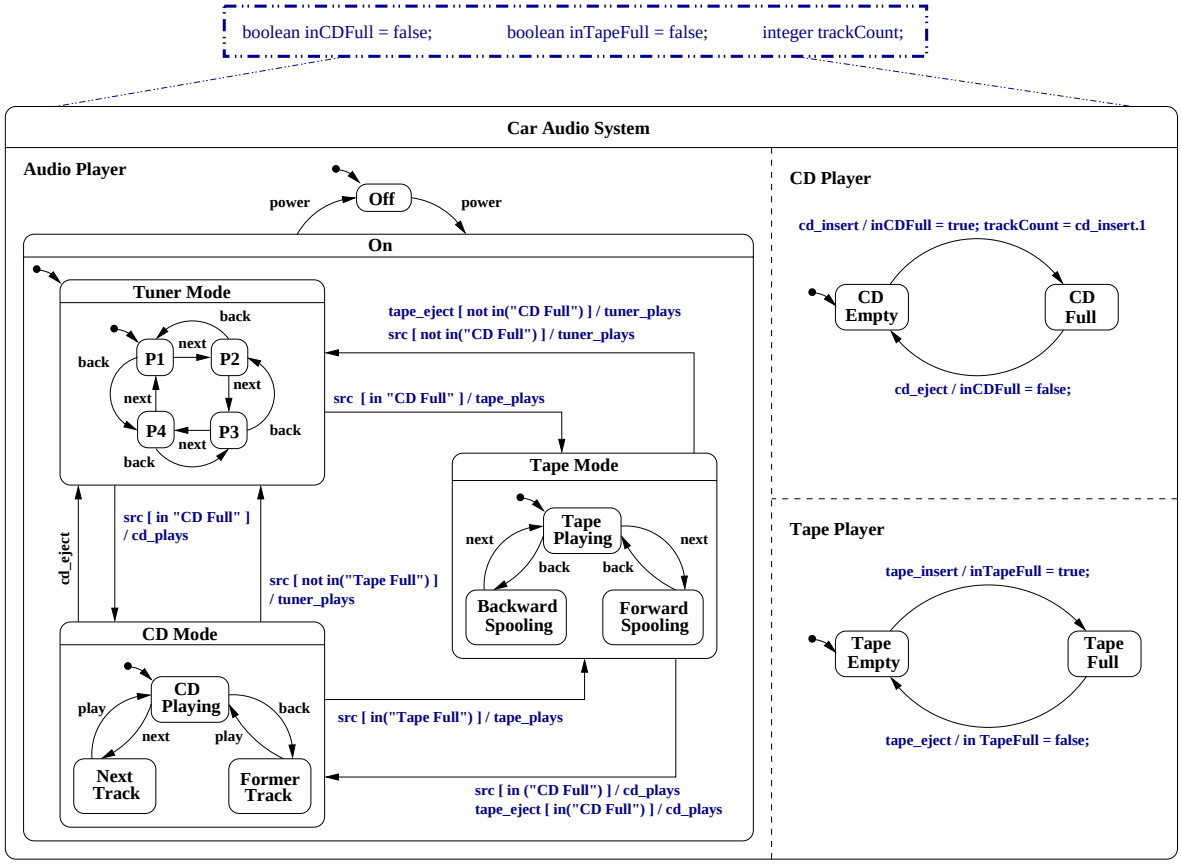


Abbildung 3.4: Datenbehaftete Spezifikation der Musikanlage

zu beschreiben, unter welcher Bedingungen die Übergänge zwischen den unterschiedlichen Musikquellen erfolgen sollen. Eine solche optionale Bedingung wird Übergangsbedingung (*guard*) der Transition genannt. Ein Wechsel zwischen den Musikquellen wird genau dann vorgenommen, wenn das Ereignis `src` anliegt und auch eine CD bzw. eine Kassette in die Musikanlage eingelegt wurde. Dabei wird die Bestimmung des aktuellen Zustands nicht mehr ereignisorientiert wie im vorherigen Beispiel vorgenommen, sondern datenorientiert mit Hilfe der beiden booleschen Variablen.

In den Regionen `CD Player` und `Tape Player` ist dargestellt, wie in Abhängigkeit von Ereignissen der Datenraum manipuliert werden kann. Ereignisse können ebenfalls komplex strukturiert sein und somit Daten beinhalten. Diese Daten können dann in der Übergangsbedingung und im Effekt einer Transition benutzt werden. So wird im Abbildung 3.4 durch die Hardware, die das Einlegen einer CD detektiert, gleichzeitig die Anzahl der auf der CD vorhandenen Titel bestimmt und dem Ereignis `cd_insert` mitgegeben. Man könnte sich z. B. vorstellen, dass die Anzahl der Titel im Display angezeigt werden soll. In der Transition vom Zustand `CD Empty` zum Zustand `CD Full` in der Region `CD Player` führt das auslösende Ereignis einen ganzzahligen Wert mit, der die Anzahl der Titel einer eingelegten CD repräsentiert. Im Effekt der Transition wird dieser Wert an die Variable `trackCount` gebunden und kann so z. B. für die Anzeige der Titelanzahl benutzt werden.

Die Ausdruckssprache für das Lesen, die Manipulation und die Auswertung von Daten wird durch die unterliegende Programmiersprache - in meinem Fall JAVA [53] - definiert. Das Programmausdruck 3.1 zeigt, wie die im Abbildung 3.4 verwendeten Daten und die in den Übergangsbedingungen verwendete Zugriffsfunktion `in` definiert werden könnten.

```
9   private boolean inCDFull = false;
    private boolean inTapeFull = false;
11  private int     trackCount;
    private boolean useData = true;
13
    private boolean in(String state) {
15    if (state.equals("CD Full")) {
        return inCDFull;
17    }
      if (state.equals("Tape Full")) {
19      return inTapeFull;
      }
21    return false;
    }
```

Programmausdruck 3.1: Auszug aus der generierten Java Klasse der Musikanlage

Bemerkung: Beispiel 3.4 ist eine Beispielhafte Umsetzung der Anforderungen unter Verwendung von Daten. Das Beispiel soll anzudeuten, in welcher Art und Weise es möglich ist, Daten zu benutzen und zu manipulieren. So ist z. B. formal gesehen eine Übergangsbedingung ein Prädikat und somit könnte man statt des Methodenaufrufs in der Übergangsbedingung `[in("CDFull")]` einfach die boolesche Variable selbst verwenden (`[inCDFull]`). Das Beispiel zeigt aber eine Ausformulierung des `in`-Prädikats, so wie es z. B. aus den Harel Statecharts [32, 36] bekannt ist. Ebenfalls ließen sich die Regionen `CDPlayer` und `TapePlayer` vereinfachen. Die Speicherung in Zuständen, ob eine CD bzw. eine Kassette eingelegt wurde, ist nicht mehr nötig.

Eine Grammatik für die Eingabe von Zustandsmaschinen in die Werkzeugumgebung TEAGER, die vollständigen Beispiele der Musikanlage und die aus dem datenorientierten Beispiel automatisch generierte Java-Klasse kann im Anhang B nachgelesen werden. In den folgenden Abschnitten werde ich Zustandsmaschinen formal einführen. Dazu wird in Abschnitt 3.4 zunächst die abstrakte Syntax der Zustandsmaschinen beschrieben und in Abschnitt 3.4 darauf aufbauend die operationale Semantik.

3.3 Abstrakte Syntax

Mit Zustandsmaschinen kann das diskrete, reaktive Verhalten eines Systems oder einer Systemkomponente beschrieben werden. Dabei ist jeder Zustandsmaschine ein Datenraum mit endlich vielen Partitionen zugeordnet. Die Partitionen können während der Ausführung einer Zustandsmaschine gelesen und geändert werden.

3.3.1 Datenräume

Definition 1 (Datenraum)
Sei K eine endliche Indexmenge mit n Elementen und sei $(P_k)_{k:K}$ eine Familie von Mengen P_k. Ein Datenraum D ist das kartesische Produkt der Mengen P_k.

$$D == P_1 \times \ldots \times P_n \tag{3.1}$$

Eine Menge P_k eines solchen Datenraums bezeichne ich als eine Partition des Datenraums und ein Element $d \in D$ bezeichne ich als eine Datenraumbelegung. □

3.3.2 Regionen und Zustände

Eine Zustandsmaschine ist aus hierarchisch und orthogonal angeordneten Knoten und Zustandsübergängen zwischen diesen Knoten aufgebaut. Die Menge der Knoten setzt sich aus einer Menge von Regionen und einer Menge von Zuständen zusammen.

Definition 2 (Knotenmenge)
Im Folgenden sei R eine endliche Menge von Regionen und S eine endliche Menge von Zuständen. Die Menge N der Knoten ergibt sich aus der disjunkten Vereinigung von R und S.

$$N == R \uplus S \tag{3.2}$$

□

Nach ihrer Verwendung innerhalb einer Zustandsmaschine unterscheidet man drei Arten von Zuständen. Zustände, die nicht verfeinert werden, werden als einfache Zustände (*simple states*) bezeichnet. Zustände, die verfeinert werden, werden als zusammengesetzte Zustände (*composite states*) bezeichnet. Zusammengesetzte Zustände werden weiter in einfach zusammengesetzte Zustände (*simple composite states*) und orthogonal zusammengesetzte Zustände (*orthogonal states*) unterschieden.

Bemerkung: In der Literatur werden einfach verfeinerte Zustände auch als *xor*-Zustände und orthogonal verfeinerte Zustände auch als *and*-Zustände bezeichnet. Diese Benennung ist durch die Anzahl der gleichzeitig aktiven »direkten« Unterzustände begründet. Wie in Abbildung 3.3 zu sehen ist, enthalten verfeinerte Zustände Regionen. Innerhalb einer Region muss genau ein Unterzustand aktiv sein. Einfach verfeinerte Zustände enthalten genau eine Region und somit ist genau einer der möglichen Unterzustände aktiv (⇒ xor). Orthogonal verfeinerte Zustände enthalten mindestens zwei Regionen. Somit sind mindestens zwei Zustände, jeweils einer in jeder Region, gleichzeitig aktiv (⇒ and).

Zur einfachen Unterscheidung der unterschiedlichen Knotenarten definiere ich folgende Funktion.

Definition 3 (Typfunktion für Knoten)
Sei N eine Menge von Knoten. Die Funktion *type* ordnet jedem Knoten eine Knotenart aus der Menge $\{region, simple, simple\ composite, orthogonal\}$ zu.

$$type : N \rightarrow \{region, simple, simple\ composite, orthogonal\} \tag{3.3}$$

□

Bemerkung: Im Unterschied zu dieser Formalisierung stellen Regionen und Zustände im UML Standard [110] eigenständige Klassen dar, und die verschiedenen Zustandsarten werden mittels boolescher Attribute der Zustandsklasse unterschieden [110, siehe S. 509 und S. 532]. Das Attribut *isSimple* gibt dabei an, ob es sich um einen einfachen Zustand handelt. Das Attribut *isComposite* gibt an, ob ein Zustand verfeinert wird. Das Attribut *isOrthogonal* gibt schließlich an, ob ein Zustand orthogonal verfeinert wird. Die Verwendung der Funktion *type* vermeidet, im Gegensatz zur Verwendung der Attribute, unnötige Redundanz, da alle Wertkombinationen, die keinen gültigen Knotentyp ergeben, unzulässig sind. Der Zusammenhang zwischen diesen drei Attributen und der Funktion *type* für Zustände $s \in S$ ist wie folgt gegeben:

$$type\,(s) = \begin{cases} simple & \text{iff} \quad s.isSimple \\ simple\ composite & \text{iff} \neg\, s.isSimple \wedge s.isComposite \wedge \neg\, s.isOrthogonal \\ orthogonal & \text{iff} \neg\, s.isSimple \wedge s.isComposite \wedge \quad s.isOrthogonal \end{cases} \tag{3.4}$$

Die Anordnung der Knoten ist als Baumstruktur (N, H) gegeben und wird als *Knotenhierarchie* bezeichnet. Dabei gibt die nicht-leere Relation $H : \mathbb{P}(N \times N)$ die relative Anordnung der Knoten zueinander wieder und wird als *Hierarchierelation* bezeichnet. Ein Tupel $n \mapsto n' : H$ gibt an, dass der Knoten n durch den Knoten n' verfeinert wird, d. h., dass der Knoten n' ein Unterknoten von n ist. Hinsichtlich der Knotenhierarchie definiere ich folgende Funktionen.

Definition 4 (Mengen der Unterknoten)
Sei (N, H) eine Knotenhierarchie. Die Funktionen $subnodes : N \to \mathbb{P}\, N$ und deren Abschlüsse zur Bestimmung der Mengen der Unterknoten eines Knotens $n : N$ sind wie folgt definiert.

$$subnodes\,(n) == \{n' : N \mid n \mapsto n' \in H\} \tag{3.5}$$

$$subnodes^{+}\,(n) == \{n' : N \mid n \mapsto n' \in H^{+}\} \tag{3.6}$$

$$subnodes^{*}\,(n) == \{n\} \cup subnodes^{+}\,(n) \tag{3.7}$$

Hierbei bezeichnet H^{+} den transitiven Abschluss von H. □

Die Funktion *subnodes* liefert alle *direkten Unterknoten* zu einem Knoten. Die Funktion $subnodes^{+}$ liefert alle transitiven und die Funktion $subnodes^{*}$ liefert alle transitiv-reflexiven Unterknoten eines Knotens. Ist ein Knoten $n_2 \in subnodes^{*}\,(n_1)$, dann sagt man, dass n_1 ein Oberknoten von n_2 und dass n_2 ein Unterknoten von n_1 ist. Aufgrund der Reflexivität von $subnodes^{*}$ gilt, dass ein Knoten n sowohl Oberknoten als auch Unterknoten von sich selbst ist. Ist ein Knoten $n_2 \in subnodes^{+}\,(n_1)$, dann sagt man, dass n_1 ein *strikter Oberknoten* und dass n_2 ein *strikter Unterknoten* von n_1 ist. In der Literatur findet man anstelle von Oberknoten/-zustand und Unterknoten/-zustand auch die Bezeichnungen Elternknoten (*parent*) und Kindknoten (*child*) bzw. Vorfahr (*ancestor*) und Nachkomme (*descendant*).

Eine Region darf nur durch Zustände verfeinert werden. Ein einfach komponierter Zustand muss durch genau eine Region, ein orthogonal verfeinerter Zustand muss durch mindestens zwei Regionen verfeinert werden. Daraus folgt, dass sich in der Baumstruktur Regionen und Zustände abwechseln und die Blätter des Baums einfache Zustände sind. Es folgt auch, dass jeder zusammengesetzte Zustand durch mindestens einen Zustand (präzise durch eine Region, die mindestens einen Zustand enthält) verfeinert wird (siehe Abbildung 3.3). Wird ein Zustand

aktiviert, wenn sein Oberknoten aktiviert wird, bezeichnen wir diesen Zustand auch als einen *initialen Zustand*. Die Menge I umfasst alle initialen Zustände. Für eine Region gilt, dass genau einer ihrer direkten Unterknoten in der Menge der initialen Zustände I enthalten ist.

Definition 5 (Wohlgeformte Knotenhierarchie)
Sei (N, H) eine Knotenhierarchie und sei $I \subseteq S$ eine Menge von initialen Zuständen. Eine wohlgeformte Knotenhierarchie erfüllt die folgenden Bedingungen.

$$\exists_1 n : N \bullet n \notin \mathsf{ran}\, H \land n \in \mathsf{dom}\, H \land \forall n' : N \setminus \{n\} \bullet \exists_1 n'' : N \bullet n'' \mapsto n' \in H \tag{3.8}$$

$$\forall n : N \mid type(n) = region \bullet (\forall n' : subnodes(n) \bullet type(n') \neq region) \land (\exists_1 n' : subnodes(n) \bullet n' \in I) \tag{3.9}$$

$$\forall n : N \mid type(n) = simple \bullet subnodes(n) = \varnothing \tag{3.10}$$

$$\forall n : N \mid type(n) = simple\ composite \bullet (\#subnodes(n) = 1) \land (\forall n' : subnodes(n) \bullet type(n') = region) \tag{3.11}$$

$$\forall n : N \mid type(n) = orthogonal \bullet (\#subnodes(n) > 1) \land (\forall n' : subnodes(n) \bullet type(n') = region) \tag{3.12}$$

□

Anforderung (3.8) stellt sicher, dass eine wohlgeformte Knotenhierarchie die Knoten der Zustandsmaschine in einer Baumstruktur anordnet, d. h., dass es einen ausgezeichneten Wurzelknoten gibt, dass alle anderen Knoten genau einen direkten Oberknoten haben und die Struktur keine Zyklen enthält. Die Hierarchierelation ist somit linkseindeutig und, mit Ausnahme des Wurzelknotens, rechtstotal. Anforderungen (3.9) bis (3.12) sind spezielle strukturelle Forderungen an eine Knotenhierarchie und charakterisieren die einzelnen Verfeinerungsvarianten, so wie ich sie in Abschnitt 3.2 beschrieben habe. Aus ihnen geht auch hervor, dass die Blätter einer Knotenhierarchie einfache Zustände sind ($\forall n : N \mid \nexists n' : N \bullet n \mapsto n' : H \bullet type(n) = simple$).

Definition 6 (Wurzelknoten einer Knotenhierarchie)
Aufgrund der Baumstruktur einer wohlgeformten Knotenhierarchie (N, H) existiert ein eindeutiger Knoten in N, der den Wurzelknoten der Baumstruktur darstellt. Ich bezeichne im Folgenden einen solchen Knoten einer Knotenhierarchie (N, H) mit *root*. Ferner gilt für Zustandsmaschinen, dass *root* eine Region ist.

$$root == \mu n : N \bullet n \notin \mathsf{ran}\, H \tag{3.13}$$

Für den Knoten *root* gilt, dass er eine Region ist: $type(root) = region$. □

Die Umkehrung der Hierarchierelation ist eine partielle Funktion, die jedem Knoten, mit Ausnahme des Wurzelknotens *root*, seinen direkten Oberknoten zuordnet. Diese Funktion nenne ich Containerfunktion ($container = H^{-1}$).

Definition 7 (Containerfunktion)
Sei (N, H) eine wohlgeformte Knotenhierarchie. Die partielle Funktion $container : N \nrightarrow N$ ordnet jedem Knoten $n : N \setminus \{root\}$, seinen direkten Oberknoten $n' : N$ zu.

$$container(n) == \mu\, n' : N \mid n' \mapsto n \in H \tag{3.14}$$

Den Knoten n' nenne ich Container von n. □

3.3.3 Zustandsübergänge

Die Übergänge zwischen den Zuständen einer Zustandsmaschine werden durch Transitionen beschrieben. Eine Transition ist eine gerichtete Beziehung zwischen einem Quellzustand (*source*) und einem Zielzustand (*target*) und ist mit einer Markierung (*label*) versehen. Eine Markierung setzt sich aus einem auslösenden Ereignis (*trigger*), einer optionalen Übergangsbedingung (*guard*) und einem optionalen Effekt (*effect*) zusammen.

Wie in der Einleitung beschrieben können Ereignisse Datenwerte mitführen. Daher setzt sich eine Menge E^ν von Ereignisinstanzen zu einem Ereignisnamen ν aus dem Ereignisnamen ν und einer endlichen Anzahl beliebiger (Parameter-) Mengen zusammen[2]. Die Menge E aller Ereignisinstanzen ergibt sich aus der disjunkten Vereinigung aller Mengen von Ereignisinstanzen, die bezüglich einer Menge von Ereignisnamen gebildet werden können. Ein Element einer solchen Menge bezeiche ich im Folgenden als Ereignisinstanz und die Abstraktion von einer konkreten Wertausprägung bezeichne ich als Ereignis. Sofern es der Kontext zulässt, benutze ich die Ausdrücke *Ereignisname* und *Ereignis* synonym.

Definition 8 (Mende der Ereignisinstanzen)
Sei E_ν eine Menge von Ereignisnamen, sei zu jedem Ereignisnamen $\nu : E_\nu$ die Menge K_ν eine endliche Indexmenge mit m_ν Elementen und sei $(P_k^\nu)_{k:K_\nu}$ eine Familie von Mengen P_k^ν. Die Menge E der Ereignisinstanzen zu einer Menge E_ν von Ereignisnamen ist wie folgt definiert:

$$E == \mid_{\nu:E_\nu} \nu \langle\langle P_1^\nu \times \ldots \times P_{m_\nu}^\nu \rangle\rangle \tag{3.15}$$

□

Ein kleines Beispiel soll im Folgenden die Bildung der Menge der Ereignisinstanzen veranschaulichen.

Beispiel 1 (Menge der Ereignisinstanzen)
Gegeben sind zwei Ereignisnamen a und b. Weiterhin ist die Anzahl der Parametermengen für a gleich 2 und für b gleich 1. Dann ergibt sich daraus die Menge aller Ereignisinstanzen wie folgt.

Menge der Ereignisnamen	$E_\nu = \{a, b\}$
Anzahl der Parametermengen	$m_a = 2 \wedge m_b = 1$
Parametermengen	$P_1^a = \mathbb{N} \wedge P_2^a = \mathbb{B}$
	$P_1^b = \mathbb{B}$
Mengen der Ereignisinstanzen	$E^a = a \langle\langle \mathbb{N}, \mathbb{B} \rangle\rangle = \{a(0, true), a(0, false), a(1, true), \ldots\}$
	$E^b = b \langle\langle \mathbb{B} \rangle\rangle = \{b(true), b(false)\}$
alle Ereignisinstanzen	$E = E^a \mid E^b = E^a \uplus E^b = \{a(0, true), \ldots, b(true), b(false)\}$

□

[2] In der Anwendung in einer Zustandsmaschine beschreibt $\nu(x, \ldots, z)$ die Menge E^ν und führt für jede Instanz einer Parametermenge eine Variable ein.

Die Übergangsbedingung einer Transition dient zur feingranularen Beschreibung der Umstände, unter denen eine Transition aktiviert werden soll. Sie ist eine boolesche Funktion, die unter Berücksichtigung einer Ereignisinstanz und einer Datenraumbelegung ausgewertet wird. Da die Ausdruckssprache für Übergangsbedingungen durch die verwendete Programmiersprache festgelegt wird [110], verzichte ich an dieser Stelle auf deren Definition. Beachtet werden muss, dass eine wohlgeformte Übergangsbedingung keine Seiteneffekte bezüglich des Datenraums einer Zustandsmaschine erzeugt.

Definition 9 (Menge der Übergangsbedingungen)
Sei D ein Datenraum. Die Menge G der Übergangsbedingungen ist eine Menge von booleschen Funktionen, die jede Datenraumbelegung auf einen Wahrheitswert abbilden.

$$G == D \to \mathbb{B} \tag{3.16}$$

□

Der Effekt einer Transition ist eine Sequenz von Aktionen, die beim Schalten der Transition ausgeführt werden. Eine Aktion ist entweder das Erzeugen eines neuen Ereignisses oder das Aktualisieren eines Datenraums (*event* bzw. *update*, siehe Definition 10). Erzeugung eines neuen Ereignisses bedeutet, dass in Abhängigkeit von einer Ereignisinstanz (dem auslösenden Ereignis) und einer Datenraumbelegung eine neue Ereignisinstanz als Resultat zurückgegeben wird. Aktualisierung eines Datenraums bedeutet, dass in Abhängigkeit von einer Ereignisinstanz (dem auslösenden Ereignis) und einer Datenraumbelegung eine neue Datenraumbelegung zurückgegeben wird.

Definition 10 (Menge der Aktionen)
Sei D ein Datenraum und sei E eine Menge von Ereignisinstanzen. Die Menge A der Aktionen ist eine Menge von partiellen Funktionen, die entweder neue Ereignisinstanzen oder neue Datenraumbelegungen liefern.

$$A == event\langle\langle D \nrightarrow E \rangle\rangle \mid update\langle\langle D \nrightarrow D \rangle\rangle \tag{3.17}$$

□

Definition 11 (Menge der Transitionen)
Sei (N, H) eine wohlgeformte Knotenhierarchie, sei E eine Menge von Ereignisinstanzen, sei G eine Menge von Übergangsbedingungen, sei A eine Menge von Aktionen und sei $S \subseteq N$ die Menge der Knoten, die keine Regionen sind. Dann ergibt sich die Struktur einer Transition wie folgt:

$$T \subseteq S \times (E \nrightarrow G \times \operatorname{seq} A) \times S \tag{3.18}$$

□

Zur besseren Lesbarkeit verwende ich für Transitionen die Projektionen *source*, *label* und *target*. Zu einer Transition $t : T$ liefert die Projektion $source == t.1$ den Quellzustand, die Projektion $label == t.2$ liefert die Markierung und die Projektion $target == t.3$ liefert den Zielzustand. Die Markierung einer Transition ist eine Funktion, deren Parameter das auslösende Ereignis der Transition ist. Die Anwendung der Funktion auf eine Ereignisinstanz liefert ein geordnetes Paar, welches sich aus einer Übergangsbedingung und einem Effekt, d. h.

einer Sequenz von Aktionen, zusammensetzt. Für ein solches Paar p liefert die Projektion $guard == p.1$ die Übergangsbedingung und die Projektion $effect == p.2$ den Effekt.

Eine Transition kann jedoch nur durch Ereignisinstanzen ausgelöst werden, die innerhalb der entsprechenden Zustandsmaschine bekannt sind. Daher unterscheide ich in der Menge E_ν der Ereignisnamen eine Menge $E_\nu^{SM} \subseteq E_\nu$ von Ereignisnamen, die einer Zustandsmaschine SM zugeordnet sind.

Definition 12 (Wohlgeformte Transitionsmenge)
Sei S eine Menge von Zuständen, sei E eine Menge von Ereignisinstanzen zu einer Menge E_ν von Ereignisnamen, sei zu jedem Ereignisnamen $\nu : E_\nu^{SM}$ die Menge K_ν eine endliche Indexmenge mit m_ν Elementen und sei $(P_k^\nu)_{k:K_\nu}$ eine Familie von Mengen P_k^ν, sei E_ν^{SM} eine Menge von Ereignisnamen zu einer Zustandsmaschine SM, sei G eine Menge von Übergangsbedingungen und sei A eine Menge von Aktionen. Für die Menge der Transitionen $T \subseteq S \times (E \nrightarrow G \times \operatorname{seq} A) \times S$ müssen die folgenden Anforderungen erfüllt sein.

$$\forall\, t : T \bullet \exists\, \nu : E_\nu^{SM} \bullet \operatorname{dom} label(t) = \nu \langle\!\langle P_1^\nu \times \ldots \times P_{m_\nu}^\nu \rangle\!\rangle \tag{3.19}$$

$$\begin{aligned} \forall\, t : T;\ e : E;\ d : D \bullet\ & e \in \operatorname{dom} label(t) \wedge guard(label(t)(e))(d) \\ & \Rightarrow \forall\, a : effect(label(t)(e)) \bullet d \in \operatorname{dom} a \end{aligned} \tag{3.20}$$

□

Zur besseren Lesbarkeit notiere ich eine Transition folgendermaßen: $s_1 \xrightarrow{e\,[g]\,/\,a} s_2$. In Definition 12 besagt Anforderung 3.19, dass eine Transition, d. h. präzise ausgedrückt die Markierungsfunktion, für alle Ereignisinstanzen zu einem Ereignisnamen definiert ist. Dabei ist die Menge der zu betrachtenden Ereignisnamen auf die Menge der Ereignisnamen der Zustandsmaschine SM beschränkt. Anforderung 3.20 besagt, dass für den Fall, dass die Markierungsfunktion auf eine Ereignisinstanz angewendet werden kann und die Übergangsbedingung angewendet auf eine Datenraumbelegung erfüllt ist, alle Aktionen des Effekts, d. h. präzise ausgedrückt die entsprechenden Funktionen, ebenfalls (für diese Datenraumbelegung) definiert sind.

Abschließend definiere ich auf Basis der bisherigen Definitionen den strukturellen Aufbau einer Zustandsmaschine.

Definition 13 (Struktur einer Zustandsmaschine)
Sei (N, H) eine wohlgeformte Knotenhierarchie, sei I eine Menge von initialen Zuständen, sei E eine Menge von Ereignisinstanzen zu einer Menge E_ν von Ereignisnamen, sei T eine Menge von Transitionen, sei D eine Menge von Datenraumbelegungen. Eine Zustandsmaschine SM ist eine aus diesen Komponenten bestehende Struktur.

$$SM == ((N, H), I, E, T, D) \tag{3.21}$$

Wobei die Menge I der initialen Zustände eine Teilmenge der Menge S der Zustände ist. □

3.4 Semantik

Das Verhalten einer Zustandsmaschine wird durch das Traversieren eines Zustandsgraphen, der sich aus der Knotenhierarchie und den spezifizierten Transitionen ergibt, modelliert. Das

Ablaufen eines Zustandsgraphen wird dabei durch Ereignisinstanzen, die von der Zustandsmaschine empfangen und für die weitere Verarbeitung gespeichert werden, ausgelöst. Während des Traversierens werden eine Reihe von Transitionen aktiviert. Wird eine aktivierte Transition für einen Zustandsübergang ausgewählt, so werden die Aktionen der Transition ausgeführt. Im Folgenden beschreibe ich eine operationale Semantik für Zustandsmaschinen. Ausgehend von einem semantischen Zustand (*status*) und einer Menge von aktivierten Transitionen definiere ich, wie eine Zustandsmaschine von einem wohlgeformten semantischen Zustand in den nächsten wohlgeformten semantischen Zustand überführt wird. Diesen Übergang bezeichne ich als *semantischen Schritt*.

3.4.1 Status

Während der Ausführung einer Zustandsmaschine kann ein Zustand aktiv oder inaktiv sein. Aufgrund der hierarchischen und orthogonalen Anordnung von Zuständen ist es möglich, dass sich eine Zustandsmaschine in mehreren Zuständen gleichzeitig befindet, also mehrere Zustände aktiv sind. Die Menge der zu einem Zeitpunkt aktiven Zustände wird als Zustandskonfiguration (*active state configuration*) oder kurz als Konfiguration bezeichnet.

Definition 14 (Menge der Zustandskonfigurationen)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine. Die Menge aller wohlgeformten Zustandskonfigurationen $C : \mathbb{P}\,\mathbb{P}\,N$ ist wie folgt definiert:

$$\begin{aligned} C == \{c : \mathbb{P}\,N \mid (\exists_1\, s' : \mathit{subnodes}\,(\mathit{root}) \bullet s' \in c) \land \\ (\forall\, s : c \mid \mathit{type}\,(s) \neq \mathit{simple} \bullet \forall\, r : \mathit{subnodes}\,(s) \bullet \exists_1\, s'' : \mathit{subnodes}\,(r) \bullet s'' \in c)\} \end{aligned} \tag{3.22}$$

□

Eine Konfiguration enthält nur Zustände. Da der Wurzelknoten eine Region ist, kann ich nicht fordern, dass der Wurzelknoten in jeder Konfiguration enthalten sein muss. Stattdessen fordere ich, dass genau ein direkter Unterknoten enthalten ist. Dies fungiert als Anker der Definition. Für alle anderen Zustände einer Konfiguration, die keine einfachen Zustände sind, gilt, dass aus jeder der sie verfeinernden Regionen genau ein Zustand in einer Konfiguration enthalten ist. Eine Konfiguration ist zuzüglich der benötigten Regionen ein Teilbaum der Knotenhierarchie. Ich nenne eine Konfiguration c_{start} *Startkonfiguration*, falls alle Zustände in c_{start} auch initiale Zustände sind ($c_{start} \subseteq I$). Zu beachten ist, dass eine Startkonfiguration i. d. R. verschieden von der Menge der initialen Zustände ist. Dies ist genau dann der Fall, wenn ein zusammengesetzter Zustand, der laut Definition mindestens einen initialen Zustand enthält, initial nicht aktiv ist.

Wie bereits angesprochen findet Kommunikation innerhalb der Zustandsmaschine und zwischen unterschiedlichen Systemkomponenten über Ereignisse statt, die während der Ausführung von Transitionen generiert oder aus der Umgebung empfangen werden. Ereignisse, die von der Zustandsmaschine empfangen werden, werden bis zur weiteren Verarbeitung zwischengespeichert. Im UML-Standard [110] ist nicht spezifiziert, in welcher Reihenfolge Ereignisse verarbeitet werden. Stattdessen wird es dem Anwender überlassen, eine Verarbeitungsvorschrift zu definieren. Diese Wahlmöglichkeit wird als semantischer Variationspunkt (*semantic variation point*) bezeichnet und ermöglicht es, unterschiedliche Strategien unter Beibehaltung der angegebenen Semantik zu definieren.

Um dieser Anforderung Rechnung zu tragen, definiere ich induktiv eine abstrakte Menge von Ereignisspeichern (*event pools*). Ein Ereignisspeicher kann entweder leer sein oder sich aus einem Ereignisspeicher und einer Ereignisinstanz zusammensetzen.

Definition 15 (Ereignisspeicher)
Sei E eine Menge von Ereignisinstanzen. Die Menge Q der *Ereignisspeicher* über der Menge der Ereignisinstanzen E ist wie folgt definiert.

$$Q == \langle\rangle \mid add\langle\langle Q \times E \rangle\rangle \tag{3.23}$$

□

Zur besseren Lesbarkeit verwende ich im Folgenden die Funktion $\oplus : Q \times \operatorname{seq} E \rightarrow Q$, die eine Sequenz von Ereignisinstanzen zum Ereignisspeicher hinzufügt. Weiterhin verwende ich die partielle Funktion $\ominus : Q \nrightarrow Q \times E$, die eine Ereignisinstanz aus einem nicht-leeren Ereignisspeicher entfernt. Mit der Funktion $\ominus$ abstrahiere ich von der Reihenfolge, in der Ereignisinstanzen zu einem Ereignisspeicher hinzugefügt wurden.

Bemerkung: In den meisten Anwendungen wird man eine *priorisierte Warteschlange* für die Speicherung und Verarbeitung von Ereignissen verwenden. Hier benutze ich jedoch eine abstrakte Beschreibung, welche alle im UML-Standard gestellten Anforderungen erfüllt, jedoch mögliche Realisierungen nicht unnötig beschränkt.

Auf Basis der Definitionen der Menge der Zustandskonfigurationen, der Menge der Ereignisspeicher und der Menge der Datenräume kann ich nun einen semantischen Zustand einer Zustandsmaschine beschreiben.

Definition 16 (Semantischer Zustand)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine, sei C eine Menge von Konfigurationen und sei Q eine Menge von Ereignisspeichern. Die Menge Z von semantischen Zuständen ist das kartesische Produkt über den Konfigurationen, den Ereignisspeichern und den Datenraumbelegungen.

$$Z == C \times Q \times D \tag{3.24}$$

Ein Element $z : Z$ bezeichne ich als *Status*, geschrieben $z = [\![c, q, d]\!]$. Ein *initialer Status* $z_{initial} = [\![c_{initial}, q, d]\!]$ besteht aus einer initialen Konfiguration, einem Ereignisspeicher und einer Datenraumbelegung. □

3.4.2 Transitionsauswahl

In einer Zustandsmaschine werden Ereignisinstanzen eine nach der anderen in einem so genannten *run-to-completion* Schritt verarbeitet. Das bedeutet, dass die nächste Ereignisinstanz aus dem Ereignisspeicher erst dann verarbeitet werden kann, wenn die Verarbeitung der vorherigen Ereignisinstanz vollständig abgeschlossen ist. Während eines Schritts vollführt eine Zustandsmaschine einen Übergang von einem Status zum nächsten Status. Wird eine Ereignisinstanz zur Verarbeitung ausgewählt, können eine oder mehrere Transitionen aktiviert werden. Konnte keine Transition aktiviert werden, wird die Ereignisinstanz verworfen und der

aktuelle Schritt wird beendet. Dies wird auch als Stottern (*stuttering*) der Zustandsmaschine bezeichnet. Im anderen Fall schalten die ausgewählten Transitionen. Im Folgenden beschreibe ich, wie die Menge der schaltenden Transitionen ausgewählt wird.

Eine Transition ist aktiviert, falls der Quellzustand der Transition in der aktuellen Konfiguration enthalten ist, die aktuelle Ereignisinstanz mit dem auslösenden Ereignis der Transition übereinstimmt und die Übergangsbedingung der Transition erfüllt ist.

Definition 17 (Aktivierte Transition)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine und sei $[\![c, q, d]\!] : Z$ ein Status. Die boolesche Funktion $enabled : T \times C \times E \times D \to \mathbb{B}$ bestimmt, ob eine Transition t im Status $[\![c, q, d]\!]$ für eine Ereignisinstanz e aktiviert ist.

$$enabled\,(t, c, e, d) \Leftrightarrow (source\,(t) \in c) \wedge (e \in \mathsf{dom}\, label\,(t)) \wedge (guard\,(label\,(t)(e))(d)) \quad (3.25)$$

□

Die Ausführung bzw. das Schalten einer Transition ist so definiert, dass zuerst der Quellzustand verlassen wird, dann die Aktionen der Transition ausgeführt werden und schließlich der Zielzustand der Transition betreten wird. Sind in einem Schritt mehrere Transitionen aktiviert, kann es sein, dass zwei oder mehr Transitionen in Konflikt zueinander stehen. Ein Konflikt tritt genau dann auf, wenn Transitionen gleiche Zustände verlassen und bedeutet, dass das Ausführen dieser Transitionen zu keinem wohlgeformten Status führt. Um die Menge der Zustände angeben zu können, die durch eine Transition verlassen werden, benötige ich noch folgende Definitionen.

Transitionen können zwischen unterschiedlichen Ebenen der Knotenhierarchie verlaufen. Solche Transitionen werden *multi-level* Transitionen genannt. Um dies in den folgenden Definitionen berücksichtigen zu können, muss der sog. Wirkungsbereich einer Transition bestimmt werden. Mit Hilfe des Wirkungsbereichs kann ich dann bestimmen, welche Zustände durch eine Transition verlassen und welche Zustände durch eine Transition neu betreten werden. Der Wirkungsbereich einer Transition ist durch den kleinsten gemeinsamen Oberknoten (*least common ancestor*) von Quellzustand und Zielzustand bestimmt und ist entweder eine Region oder ein orthogonal verfeinerter Zustand.

Definition 18 (Wirkungsbereich einer Transition)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine. Die Funktion $lca : T \to N$ ordnet jeder Transition $t : T$ den kleinsten gemeinsamen Oberknoten ihres Quell- und Zielzustands zu.

$$\begin{aligned} lca\,(t) == \mu\ n : N \mid\ & (source\,(t) \in subnodes^+\,(n)) \wedge \\ & (target\,(t) \in subnodes^+\,(n)) \wedge \\ & (\nexists n' : N \mid n' \in subnodes^+\,(n) \bullet \\ & \quad source\,(t) \in subnodes^+\,(n') \wedge target\,(t) \in subnodes^+\,(n')) \end{aligned} \quad (3.26)$$

□

Auf Basis des Wirkungsbereichs einer Transition kann ich nun deren *main source* und *main target* bestimmen. Diese geben die Oberzustände an, die durch die Transition verlassen bzw. betreten werden.

Definition 19 (Hauptquell- und Hauptzielzustand einer Transition)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine. Die Funktionen $mainSource : T \to N$ und $mainTarget : T \to N$ sind wie folgt definiert:

$$mainSource\,(t) ==$$
$$\begin{cases} \mu\ s : subnodes\,(lca\,(t)) \mid source\,(t) \in subnodes^*\,(s) & \text{if } type\,(lca\,(t)) = region \\ lca\,(t) & \text{if } type\,(lca\,(t)) = orthogonal \end{cases} \tag{3.27}$$

$$mainTarget\,(t) ==$$
$$\begin{cases} \mu\ s : subnodes\,(lca\,(t)) \mid target(t) \in subnodes^*\,(s) & \text{if } type\,(lca\,(t)) = region \\ lca\,(t) & \text{if } type\,(lca\,(t)) = orthogonal \end{cases} \tag{3.28}$$

Wobei gilt, dass $\forall\, n : \mathsf{ran}\ lca \bullet type\,(n) = region \lor type\,(n) = orthogonal$. □

Nun kann ich die Menge der Zustände definieren, die durch eine Transition $t : T$ verlassen werden. Diese Menge enthält alle Unterzustände von $mainSource\,(t)$, die derzeitig aktiv sind, d. h. in der aktuellen Konfiguration enthalten sind.

Definition 20 (Menge der Zustände, die durch eine Transition verlassen werden)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine und sei $[\![c, q, d]\!] : Z$ ein Status. Die Funktion $exits : T \to \mathbb{P}\, S$ liefert die Menge von Zuständen, die durch eine Transition $t : T$ verlassen werden.

$$exits\,(t) == subnodes^*\,(mainSource\,(t)) \cap c \tag{3.29}$$

□

Die Definition von Konflikten beruht auf den Zustandsmengen, die durch Transitionen verlassen werden. Zwei Transitionen stehen in Konflikt miteinander, falls sie gemeinsame Zustände verlassen.

Definition 21 (Konfliktrelation)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine. Die Relation $\nparallel : T \times T$ gibt an, ob zwei Transitionen $t_1, t_2 : T$ in Konflikt, geschrieben $t_1 \nparallel t_2$, zueinander stehen.

$$\nparallel == \{(t_1, t_2) : T \times T \mid exits\,(t_1) \cap exits\,(t_2) \neq \varnothing\} \tag{3.30}$$

Zusätzlich definiere ich die Relation $\parallel == \neg \nparallel$ und nenne zwei Transitionen verträglich, geschrieben $t_1 \parallel t_2$, falls sie keine gemeinsamen Zustände verlassen. □

Das Auflösen von Konflikten erfolgt in zwei Schritten. Zuerst werden die am höchsten priorisierten Transitionen ermittelt und ausgewählt. Bestehen in dieser Menge weiterhin Konflikte, dann werden Teilmengen derart bestimmt, dass diese konfliktfrei und maximal bezüglich ihrer Kardinalität sind.

Das Prioritätsschema basiert auf der relativen Position der Quellzustände der Transitionen in der Knotenhierarchie. Eine Transition, deren Quellzustand ein strikter Unterzustand des Quellzustands einer konkurrierenden Transition ist, ist höher priorisiert als die Andere.

Definition 22 (Prioritätsrelation)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine. Die Relation $\prec : T \times T$ gibt für zwei Transitionen $t_1, t_2 : T$ an, ob t_1 *Priorität* über t_2, geschrieben $t_1 \prec t_2$, hat.

$$\prec == \{(t_1, t_2) : T \times T \mid source\,(t_1) \in subnodes^+\,(source\,(t_2))\} \tag{3.31}$$

□

Mit Hilfe der Relation $\prec$ können jedoch nicht alle Konflikte aufgelöst werden. So hat zum Beispiel von zwei Transitionen $t_1, t_2 : T$, die vom gleichen Quellzustand ausgehen, keine Priorität über die andere, d. h. sie stehen in keiner Prioritätsbeziehung $(t1 \not\prec t2 \ \wedge \ t_2 \not\prec t_1)$. In solchen Fällen werden konfliktfreie Teilmengen $T_{\|} \subseteq T$ derart gebildet, dass diese die folgenden Eigenschaften erfüllen:

Definition 23 (Menge der schaltenden Transitionen)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine, sei $q : Q$ ein Ereignisspeicher, sei $e : E$ eine Ereignisinstanz aus dem Ereignisspeicher q $((q', e) = \ominus(q))$ und sei $[\![c, q, d]\!] : Z$ ein Status. Eine Menge $T_{\|} \subseteq T$ von schaltenden Transitionen erfüllt die folgenden Anforderungen:

$$\forall\, t : T_{\|} \ \bullet \ enabled\,(t, c, e, d) \tag{3.32}$$

$$\forall\, t_1, t_2 : T_{\|} \mid t_1 \neq t_2 \ \bullet \ t_1 \parallel t_2 \tag{3.33}$$

$$\nexists t' : T \setminus T_{\|} \mid enabled\,(t', e, c, d) \ \bullet \ \forall\, t : T_{\|} \bullet \ t \parallel t' \vee t' \prec t \tag{3.34}$$

□

Definition 23 stellt drei Anforderungen an eine Menge schaltender Transitionen. Erstens müssen alle Transitionen in einer solchen Menge aktiviert sein. Zweitens müssen alle Transitionen in dieser Menge paarweise konfliktfrei sein und drittens darf es keine Transition außerhalb dieser Menge geben, die aktiviert und konfliktfrei mit allen Transitionen innerhalb dieser Menge ist, oder die höher priorisiert ist als eine Transition innerhalb dieser Menge. Somit sind in einer solchen Menge alle Transitionen enthalten, die aktiviert sind und die höchste Priorität haben. Sollte es zu einem Zeitpunkt mehr als eine solche Menge geben, dann bezeichnet man die Zustandsmaschine als nichtdeterministisch, da man eine dieser Mengen für die Ausführung beliebig wählen kann. Diese Wahlmöglichkeit stellt einen weiteren semantischen Variationspunkt dar, der es ermöglicht, eine spezifische Auswahlstrategie festzulegen. Der Algoritmus für die Berechnung der Mengen schaltender Transitionen wird im UML-Standard [110] Transitionsselektionsalgorithmus (*transition selection algorithm*) genannt.

Abschließend definiere ich auf Basis von $mainTarget\,(t)$ die Menge der Zustände, die durch eine Transition $t : T$ neu betreten werden. Die Menge ist so definiert, dass alle Unterzustände von $mainTarget\,(t)$ in der Menge enthalten sind, die den Zielzustand der Transition enthalten oder durch die Transition implizit betreten werden.

Definition 24 (Menge der Zustände, die durch eine Transition betreten werden)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine. Die rekursive Funktion $enters : N \times S \to \mathbb{P}\, S$ liefert, angewendet auf $mainTarget\,(t)$ und $target(t)$ einer Transition $t : T$, die Menge der

Zustände, die durch t betreten werden.

$$enters\,(n, target) == \begin{cases} \{n\} & \text{if } type\,(n) = simple \\ \{n\} \cup \bigcup\limits_{n' \in subnodes\,(n)} enters\,(n', target) & \text{if } type\,(n) = orthogonal \vee \\ & \quad simple\ composite \\ enters\,(n'', target) & \text{if } type\,(n) = region \end{cases} \tag{3.35}$$

Wobei $n'' = \mu\, n''' : subnodes\,(n) \mid (target \in subnodes^*\,(n''')) \vee (n''' \in I \wedge target \notin subnodes^+\,(n))$. □

Die Funktion *enters* berechnet ausgehend von $mainTarget\,(t)$ die Menge der Zustände, die nach dem Schalten der Transition betreten werden. Ist der übergebene Knoten ein einfacher Zustand, dann wird nur dieser Zustand betreten. Ist der übergebene Knoten ein orthogonal oder ein einfach verfeinerter Zustand, dann werden neben dem Zustand selbst alle Regionen des Zustands betreten ($subnodes\,(n)$). Ist der übergebene Knoten eine Region, dann unterscheide ich zwei Alternativen. Entweder wird der direkte Unterzustand n''', der den Zielzustand enthält betreten – $target \in subnodes^*\,(n''')$, oder es wird der direkte Unterzustand n''' betreten, der ein initialer Zustand ist. Letzteres ist der Fall, wenn der Zielzustand kein Unterknoten des aktuellen Knotens ist – $n''' \in I \wedge target \notin subnodes^+\,(n)$. Die erste Alternative tritt auf, wenn man sich in der Knotenhierarchie in Richtung Zielzustand bewegt. Die zweite Alternative tritt auf, wenn man sich in der Knotenhierarchie nach unten bewegen, jedoch den Zielzustand bereits erreicht hat und noch an keinem Blattknoten angekommen ist. Dies ist genau dann der Fall, wenn der Zielzustand verfeinert wurde oder eine Region eines orthogonalen Zustands *implizit* betreten wird.

3.4.3 Transitionsausführung – *Run-To-Completion*

Als semantischen Schritt bezeichne ich den Übergang von einem wohlgeformten Status zum nächsten wohlgeformten Status. In Bezug auf einen solchen semantischen Schritt unterscheidet man verschiedene zwei Situationen. Ist keine Ereignisinstanz im Ereignisspeicher der Zustandsmaschine enthalten, dann bleibt die aktuelle Konfiguration und die aktuelle Datenraumbelegung unverändert und es werden lediglich Ereignisse, die aus der Umgebung empfangen wurden, zum Ereignisspeicher hinzugefügt. Kann dagegen eine Ereignisinstanz aus dem Ereignisspeicher entfernt werden, dann wird die Menge der schaltenden Transitionen bestimmt und danach die darin enthaltenden Transitionen zur Ausführung gebracht. Die Ausführung erfolgt nach der bereits besprochenen *run-to-completion* Semantik. Während der Verarbeitung werden durch die Aktionen der Transitionen die Datenraumbelegung aktualisiert und neue Ereignisinstanzen erzeugt. Die neu erzeugten Ereignisinstanzen, die für die Zustandsmaschine bestimmt sind, werden zum Ereignisspeicher hinzugefügt. Alle neu erzeugten Ereignisinstanzen, die für die Umgebung bestimmt sind, werden als Ergebnis des semantischen Schritts zurückgeliefert. Zuletzt werden alle Ereignisinstanzen, die aus der Umgebung empfangen wurden, zum aktuellen Ereignisspeicher hinzugefügt.

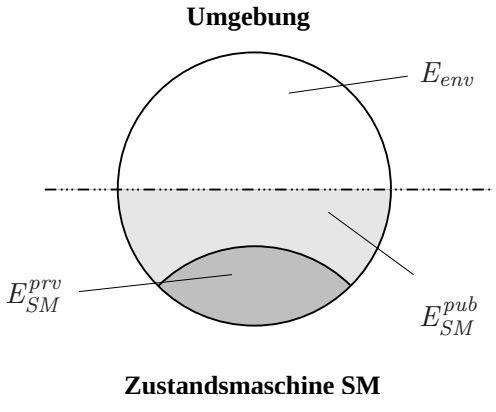


Abbildung 3.5: Zusammensetzung der Mengen von Ereignisinstanzen

Um präzise angeben zu können, welche Ereignisinstanzen aus der Umgebung an eine Zustandsmaschine gesendet werden können, unterscheide ich zwei verschiedene Arten von Ereignisinstanzen einer Zustandsmaschine. Zum einen gibt es Ereignisinstanzen, die ausschließlich in der Zustandsmaschine selbst erzeugt werden und für die interne Kommunikation benutzt werden können. Diese Menge der privaten Ereignisinstanzen einer Zustandsmaschine bezeichne ich mit $E_{SM}^{prv} \subseteq E_{SM}$. Zum anderen gibt es Ereignisinstanzen, die in der Zustandsmaschine selbst oder in der Umgebung erzeugt werden können. Diese Menge der öffentlichen Ereignisinstanzen einer Zustandsmaschine bezeiche ich mit $E_{SM}^{pub} \subseteq E$. Sie repräsentieren somit die öffentliche Schnittstelle der Zustandsmaschine. Die Menge der Ereignisinstanzen der Umgebung bezeichne ich mit $E_{env} \subseteq E$. Das sind die Ereignisinstanzen, die die Zustandsmaschine dafür benutzt, um mit der Umgebung zu kommunizieren. Sie repräsentieren somit die öffentliche Schnittstelle der Umgebung. Bezüglich dieser drei Mengen gilt folgende Beziehung: $E_{SM}^{prv} \uplus E_{SM}^{pub} \uplus E_{env} = E$. In Abbildung 3.5 sind die Mengen der Ereignisinstanzen dargestellt.

Im Folgenden notiere ich einen semantischen Schritt $(\llbracket c, q, d \rrbracket, E_{in}, E_{out}, \llbracket c', q', d' \rrbracket) : Z \times \operatorname{seq} E_{SM}^{pub} \times \operatorname{seq} E_{env} \times Z$ zur besseren Lesbarkeit folgendermaßen: $\llbracket c, q, d \rrbracket \xrightarrow{E_{in}, E_{out}} \llbracket c', q', d' \rrbracket$.

Definition 25 (Semantischer Schritt)
Sei $((N, H), I, E, T, D)$ eine Zustandsmaschine, sei $\llbracket c, q, d \rrbracket : Z$ ein Status, sei $T_{\|} \subseteq T$ eine Menge schaltender Transitionen und sei $E_{in} : \operatorname{seq} E_{SM}^{pub}$ eine Sequenz von eingehenden öffentlichen Ereignisinstanzen. Ein semantischer Schritt $\llbracket c, q, d \rrbracket \xrightarrow{E_{in}, E_{out}} \llbracket c', q', d' \rrbracket : Z \times \operatorname{seq} E_{SM}^{pub} \times \operatorname{seq} E_{env} \times Z$ ist durch die folgenden Regeln definiert.

$$\frac{\begin{array}{c} q = <> \\ q' = \oplus(q, E_{in}) \end{array}}{[\![c, q, d]\!] \xrightarrow{E_{in}, \langle\rangle} [\![c, q', d]\!]} \quad (3.36)$$

$$\frac{\begin{array}{c} q \in \mathsf{ran}\, add \\ (q'', e) = \ominus(q) \\ c' = (c \setminus \bigcup_{\forall\, t : T_{\|}} exits\,(t)) \cup \bigcup_{\forall\, t : T_{\|}} enters\,(t)) \\ A_{seq} \in perm(\{t : T_{\|} \bullet effect\,(label\,(t)(e))\}) \\ (d', E_{gen}) = performAll(\frown\!/\, A_{seq})(d) \\ (E_{int} = E_{gen} \upharpoonright E_{SM}) \wedge (E_{out} = E_{gen} \upharpoonright E_{env}) \\ q' = (q'' \oplus E_{int}) \oplus E_{in} \end{array}}{[\![c, q, d]\!] \xrightarrow{E_{in}, E_{out}} [\![c', q', d']\!]} \quad (3.37)$$

Wobei ich mit $E_{out} : \mathrm{seq}\, E_{env}$ eine Sequenz von ausgehenden und mit $E_{int} : \mathrm{seq}\, E_{SM}$ eine Sequenz von internen Ereignissen bezeichne. □

Die Ereignisse in E_{out} werden von der Zustandsmaschine an ihre Umgebung geschickt, d. h. an die entsprechenden Systemkomponenten. Die Funktion $perm : \mathbb{P}\,\mathrm{seq}\, X \rightarrow \mathbb{P}\,\mathrm{seq}(\mathrm{seq}\, X)$ liefert für eine Menge von Sequenzen die Menge aller möglichen Permutationen dieser Sequenzen. Die Funktion $\frown\!/ : \mathrm{seq}(\mathrm{seq}\, X) \rightarrow \mathrm{seq}\, X$ glättet eine Sequenz von Sequenzen, d. h., dass die Teilsequenzen zur einer einzigen Sequenz konkateniert werden. Die Funktion *performAll* : $\mathrm{seq}\, A \rightarrow D \rightarrow (D, \mathrm{seq}\, E)$ liefert, angewendet auf eine Datenraumbelegung, für eine Sequenz von Aktionen ein Tupel bestehend aus einer neuen Datenraumbelegung und einer Sequenz von erzeugten Ereignisinstanzen. Mit dieser Funktion wird der eigentliche Effekt des semantischen Schritts berechnet, d. h. der aktualisierte Datenraum und die Ereignisse, die während des Übergangs erzeugt werden. Eine beispielhafte Implementierung der Funktion *performAll* ist im Anhang A wiedergegeben.

In Definition 25 sind zwei weitere semantische Variationpunkte enthalten. Der erste Variationspunkt betrifft die Reihenfolge, in der die einzelnen Transitionen ausgeführt werden. Diese ist im UML-Standard [110] nicht vorgegeben. Der Transitionsselektionsalgorithmus selektiert lediglich eine *Menge* von Transitionen. Der zweite Variationspunkt betrifft die Behandlung von Ereignissen, die in der Umgebung erzeugt wurden. Es ist kein Zeitpunkt angegeben, an dem diese Ereignisinstanzen zu einem Ereignisspeicher hinzugefügt werden sollen. In der vorliegenden Formalisierung sehe ich einen semantischen Schritt als atomar an. Somit werden die Ereignisse aus der Umgebung erst nach Beendigung eines Schritts zu einem Ereignisspeicher hinzugefügt.

Bemerkung: Aus vielen Arbeiten zur Semantik von Statecharts ist bekannt [36, 40], dass es im Zusammenhang mit Daten bei der parallelen Ausführung von Transitionen zu Schreibkonflikten (*racing*) kommen kann, da die Ausführung von Transitionen gleichzeitig erfolgt. Ein Konflikt tritt genau dann auf, wenn zwei Transitionen gleichzeitig auf eine Variable eines Datenraums zugreifen wollen. Es gibt eine Fülle von Strategien, wie mit dieser Situation semantisch umgegangen werden kann. So wird zu Beispiel in einer *interleaving* Semantik das Problem in einem Nichtdeterminismus aufgelöst. Dabei wird angenommen, dass jede Transition den Konflikt gewinnen kann, der Wert der Variablen also entweder durch die eine oder durch die andere Transition bestimmt ist und es folglich mehrere mögliche semantische Zustände gibt.

Zu einem Schreibkonflikt kann es in meiner Semantik nicht kommen. Können mehrere Transitionen ausgeführt werden, führt jede mögliche Reihenfolge zu einen gültigen semantischen Schritt. In einem konkreten semantischen Schritt wird eine beliebige Transitionssequenz ausgewählt. Die Reihenfolge ist somit beliebig, aber fest. Weiterhin setzt sich der Effekt einer Transition aus einer Sequenz von Aktionen zusammen und führt somit ebenfalls zu keinen Konflikt. Die Sequenzialisierung der Transitionsausführung und der Aktionen steht in Einklang mit der UML. Dies ist eine naheliegende Herangehensweise, wenn man bedenkt, dass die Systemkomponenten, die durch Zustandsmaschinen beschrieben werden, letztendlich implementiert werden sollen.

3.5 Mögliche Erweiterungen

Wie bereits in Abschnitt 3.1 beschrieben, betrachte ich einen für die Untersuchung einer automatisierten Testfallerzeugung relevanten Ausschnitt der UML-Zustandsmaschinen. Es ist nicht das primäre Ziel dieser Arbeit, den vollen syntaktischen Umfang der Ausdrucksmittel von UML-Zustandsmaschinen zu unterstützen. Trotzdem möchte ich im Folgenden kurz beschreiben, welche Erweiterungen für zukünftige Arbeiten interessant sind. Die meisten im Folgenden beschriebenen Erweiterungen haben keine direkte Auswirkung auf die im nächsten Kapitel beschriebenen Testfallerzeugung, sondern erweitern die Ausdrucksmöglichkeiten meines Ausschnitts der UML-Zustandsmaschinen.

3.5.1 Zustände mit besonderer semantischer Bedeutung

Neben den bisher beschriebenen Zuständen sieht der UML-Standard noch eine Reihe von Zuständen mit besonderer semantischer Bedeutung vor, den sog. Pseudozuständen (*Pseudo States*). Diese Zustände ermöglichen es, spezielles Verhalten zu modellieren. Zu den Pseudozuständen gehören der *Initial*-Pseudozustand, der tiefe und flache *History*-Pseudozustand, die *Join*- und *Fork*-Pseudozustände, die *Junction*- und *Choice*-Pseudozustände und die *Entry*-, *Exit*- und *Terminate*-Pseudozustände. Beispielhaft werde ich im Folgenden kurz die Verwendung von *Initial*-Pseudozuständen und von *Join*- und *Fork*-Pseudozuständen erläutern. Die in der Werkzeugumgebung TEAGER implementierte Semantik ist bereits für die Verwendung von Pseudozuständen vorbereitet.

Initial-Pseudozustände sind Zustände, von denen eine einzige Transition ausgeht und am initialen Zustand einer Region endet. Diese Transition besitzen weder ein auslösendes Ereignis, noch eine Übergangsbedingung. Optional kann ein Effekt spezifiziert werden. Der Effekt kann z. B. zur Initialisierung des Datenraums benutzt werden. Dies ist ein Aspekt, der besonders bei der Testfallerzeugung mit Daten relevant wird.

Die *Join*- und *Fork*-Pseudozustände werden insbesondere im Zusammenhang mit Zusammengesetzten Transitionen (*Compound Transitions*) verwendet. Mit einem *Join*-Pseudozustand können mehrere Transitionen, die von unterschiedlichen orthogonalen Regionen ausgehen, zusammengeführt werden. Diese Transitionen dürfen jedoch werder ein auslösendes Ereignis, noch eine Übergangsbedingung aufweisen. Im Gegensatz dazu können *Fork*-Pseudozustände dazu verwendet werden, eine eingehende Transitionen in mehrere ausgehende Transitionen aufzuteilen. Die ausgehenden Transitionen dürfen weder ein auslösendes Ereig-

nis, noch eine Übergangsbedingung ausweisen und die Zielzustände sind jeweils Zustände in unterschiedlichen orthogonalen Regionen.

3.5.2 Aktionen außerhalb von Transitionen

Der UML-Standard sieht weiterhin vor, Aktionen auch außerhalb von Transitionen zu spezifizieren. Diese Aktionen können entweder beim Betreten (*entry*), beim Verlassen (*exit*) oder während sich die Zustandsmaschine in einem Zustand befindet (*doActivity*) ausgeführt werden. Bezüglich der Semantik von Zustandsmaschinen muss in diesem Zusammenhang besonders berücksichtigt werden, in welcher Reihenfolge Zustände betreten und verlassen werden. Zum jetzigen Zeitpunkt, d. h. ohne solche Aktionen, hat diese Reihenfolge keine Auswirkungen auf die operationale Semantik.

3.5.3 Erweiterung des Ereignisspeichers

Bezüglich eines Zustands besteht für eine Teilmenge der Ereignisinstanzen die Möglichkeit zu spezifizieren, dass diese, sofern sie keine Transition aktivieren, zurückgestellt und nicht gelöscht werden sollen (*deferred events*). Zurückgestellte Ereignisse bleiben im Ereignisspeicher so lange erhalten, bis sie entweder eine Transition aktivieren oder in einem Zustand nicht als zurückzustellen markiert sind und folglich gelöscht werden. Die Behandlung von zurückstellbaren Ereignissen erweitert die Funktionalität eines Ereignisspeichers und ist aus diesem Grund besonders zu berücksichtigen.

Eine weitere Erweiterung des Ereignisspeichers erfolgt auch durch die Einführung von Transitionen ohne explizites auslösendes Ereignis. Diese Transitionen gehen von einem Zustand aus und können optional eine Übergangsbedingung und einen Effekt aufweisen. Die Transitionen werden durch sog. Vervollständigungsereignisse (*completion events*) ausgelöst. Diese Ereignisse werden erzeugt, wenn die Eintrittsaktionen und die Zustandsaktionen eines Zustands abgeschlossen sind. Vervollständigungsereignisse werden vor allen anderen Ereignissen aus einem Ereignisspeicher verarbeitet. Sie sind somit den anderen Ereignissen gegenüber priorisiert. Dies muss bei der Ereignisverarbeitung gesondert berücksichtigt werden. In den meisten praktischen Anwendungen wird man als Ereignisspeicher eine priorisierte Warteschlange (*prioritized fifo queue*) verwenden.

3.6 Zusammenfassung

In diesem Kapitel habe ich die von mir verwendete abstrakte Syntax und operationale Semantik eines Ausschnitts der UML-Zustandsmaschinen vorgestellt. Dieser Ausschnitt enthält die wesentlichen syntaktischen Ausdrucksmittel und komplex strukturierte Daten. Im ersten Abschnitt habe ich begründet, dass eine automatisierte Ableitung von Testfällen aus Zustandsmaschinen nur möglich ist, wenn diesen eine vollständige und präzise Semantik zugrunde liegt. Da die Literaturrecherche keine für meine Zwecke verwendbare Semantik lieferte, habe ich eine eigene Formalisierung von Zustandsmaschinen vorgenommen.

In Abschnitt 3.2 habe ich zunächst eine informelle Einführung in Zustandsmaschinen oh-

ne Daten und darauf aufbauend in Zustandsmaschinen mit Daten gegeben. In Abschnitt 3.3 habe ich dann die verwendete abstrakte Syntax formal eingeführt und in Abschnitt 3.4 eine operationale Semantik auf Basis dieser abstrakten Syntax definiert. Ergebnis dieses Kapitels ist die formale Definition eines semantischen Schritts einer Zustandsmaschine (siehe Definition 25). Ein semantischer Schritt umfasst die Bestimmung der in einem Schritt auszuführenden Transitionen und die Berechnung des Effekts, der sich aus der Ausführung der Transitionen ergibt. Im Gegensatz zu vielen Formalisierungen die man in der Literatur findet [4, 62, 58, ?], beschreibt die hier angegebene Formalisierung den betrachteten Ausschnitt vollständig. Das bedeutet im Detail, dass alle benötigten Teile formal eingeführt und deren Verhalten präzise beschrieben wurde. Zur Überprüfung der Definitionen wurden zusätzlich die wichtigsten Definitionen in ML implementiert (siehe Anhang A).

Daten sind ein sehr mächtiges Konzept. Zum einen kann man mit komplexen Daten in den Übergangsbedingungen sehr detailliert beschreiben, unter welchen Bedingungen ein Zustandsübergang erfolgen soll. Zum anderen können über die Ereignisse komplexe Daten innerhalb der Zustandsmaschine und ihrer Umgebung ausgetauscht werden. In der vorliegenden Formalisierung habe ich formal beschrieben, wie komplex strukturierte Daten und deren Auswertung und Manipulation in Zustandsmaschinen integriert sind. Eine solche Beschreibung ist in der Literatur derzeitig nicht zu finden. In Zustandsmaschinen treten diese Daten an zwei Stellen auf. Zum einem kann jede Zustandsmaschine einen komplex strukturierten Datenraum, der in den Übergangsbedingungen und den Aktionen einer Transition gelesen und in den Aktionen einer Transition manipuliert werden kann. Zum anderen können Ereignisse komplex strukturiert sein, so dass sie Daten mitführen, die in den Übergangsbedingungen und in den Aktionen gelesen werden können. Dabei habe ich von einer konkreten Ausdruckssprache abstrahiert, da diese durch die bei einer konkreten Modellierung benutzten Programmiersprache festgelegt wird.

Die Markierung einer Transition habe ich in der Semantik durch eine Funktion beschrieben. Dies ermöglicht eine elegante Behandlung der unterschiedlichen Datenwerte, da die Datenraumbelegung und die Wertbelegung eines Ereignisses als Parameter der Markierungsfunktion auftreten. Die Markierungsfunktion bildet eine Ereignisinstanz auf ein Paar, bestehend aus der Übergangsbedingung und dem Effekt der Transition, ab. Wendet man die Übergangsbedingung auf eine Datenraumbelegung an, so kann man bestimmen, ob die Übergangsbedingung erfüllt ist. Für den Fall, dass die Übergansbedingung erfüllt ist, kann der Effekt der Transition aus der sequentiellen Auswertung der einzelnen Aktionen bestimmt werden. Der Effekt einer Transition setzt sich aus der aktualisierten Datenraumbelegung und der Menge neu erzeugter Ereignisinstanzen zusammen.

Ich habe bewusst nicht den vollen Umfang der UML-Zustandsmaschinen formalisiert. Der Ausschnitt wurde jedoch so gewählt, dass er den wesentlichen Kern der UML-Zustandsmaschinen umfasst und eine detaillierte Untersuchung einer automatisierten Testfallerzeugung aus Zustandsmaschinen erlaubt. Weiterhin habe ich in der vorgestellten Semantik insbesondere darauf geachtet, dass diese weitestgehend konform zum UML-Standard [110] sind und nur Lücken sinnvoll geschlossen werden. Dies hat vorallem zwei Vorteile. Zum einen bleiben die Definitionen bezüglich des Standards und anderer Arbeiten vergleichbar und zum anderen kann ich so eine Kernsemantik angeben, auf deren Basis in zukünftigen Arbeiten der betrachtete Ausschnitt vergrößert werden kann. Mögliche, lohnenswerte Erweiterungen habe ich im Abschnitt 3.5 beschrieben.

Die vorgestellte Semantik bildet die Basis für den automatisierten Konformitätstest, den ich im nächsten Kapitel beschreiben werde. In diesem Kapitel wird man sehen, dass gerade die Asynchronität und der in Zustandsmaschinen enthaltene Nichtdeterminismus eine besondere Herausforderung für die automatisierte Testfallerzeugung sind.

Kapitel 4

Testen auf Basis von Zustandsmaschinen

Die vorliegende Arbeit beschäftigt sich mit der Qualitätssicherung von eingebetteten reaktiven Systemen. Im Mittelpunkt der Arbeit steht der Aspekt zu prüfen, ob das Verhalten des zu testenden Systems konform zum spezifizierten Verhalten ist. Die Prüfung erfolgt durch die Ausführung von automatisch aus der Spezifikation abgeleiteten Testfällen. Dabei ist es das Ziel, durch die Ausführung der Testfälle möglichst viel Fehlverhalten möglichst effizient aufzuzeigen. Ein rigoroser Nachweis, dass das zu testende System sich konform zur Spezifikation verhält kann durch die Ausführung von Testfällen im Allgemeinen nicht erbracht werden. Das zu testende System nenne ich im Folgenden *System unter Test* oder kurz *SUT*.

In Abschnitt 4.1 werde ich zunächst die für die Arbeit angenommene Ausgangssituation für die Durchführung von Tests schildern. Anschließend werde ich erläutern, wie auf Basis einer Testhypothese Konformität formal definiert werden kann. Dafür werde ich in Abschnitt 4.2 auf Grundlage des im vorigen Kapitel definierten semantischen Schritts ein Beobachtungsmodell für Zustandsmaschinen beschreiben. Dieses Beobachtungsmodell stellt alle benötigten Informationen zur Verfügung, um die Bedeutung von Konformität zwischen einem zu testenden System und einer Spezifikation, die das Verhalten des zu realisierenden Systems beschreibt, formal zu definieren. Auf Basis der Definition von Konformität beschreibe ich anschließend, wie eine automatisierte Ableitung von Testfällen für einen Konformitätstest aus der Spezifikation erfolgen kann und welches Fehlverhalten mit Hilfe der generierten Testfälle aufgezeigt werden kann. Ziel der Testableitung und -ausführung ist möglichst effizient zu prüfen, ob das zu testende System eine gültige Realisierung der gegebenen Spezifikation ist. In Abschnitt 4.3 widme ich mich daher der Problematik, dass die Ableitung der Testfälle auf Basis der Spezifikation und der definierten Implementierungsrelation extrem aufwändig für Zustandsmaschinen ist. Zur Lösung des Problems stelle ich ein Konzept vor, wie Testfälle miteinander kombiniert werden können, um das System unter Test auch mit langen Eingabesequenzen effizient testen zu können. Die Reduktion des Berechnungsaufwands wird durch eine Approximation des Verhaltens erreicht, in der einzelne Testfälle miteinander kombiniert werden. In Abschnitt 4.4

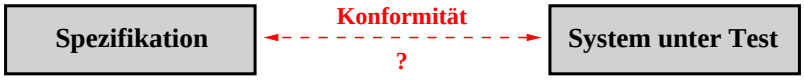


Abbildung 4.1: Ausgangssituation für die Durchführung von Tests.

werde ich beschreiben, wie die abgeleiteten Testfälle gegen ein System unter Test ausgeführt werden und die Ausführung bewertet wird. In Abschnitt 4.5 werde ich das Kapitel mit einer Zusammenfassung abschließen.

4.1 Testhypothese

Sehr häufig ist die Ausgangssituation für die Durchführung von Tests dadurch gekennzeichnet, dass die gewünschte Beziehung zwischen der Spezifikation und dem System unter Test nicht präzise definiert ist. Dies hat insbesondere zur Folge, dass nicht präzise definiert ist, unter welchen Bedingungen ein System unter Test eine korrekte Implementierung einer Spezifikation darstellt und wie ein eventuelles Abweichen von dieser Spezifikation durch die Ausführung von Testfällen aufgezeigt werden kann. Einerseits liegt die Spezifikation nicht zwingend in einer formalen, präzisen Form vor. Häufiger wird es der Fall sein, dass die Spezifikation in natürlicher Sprache formuliert ist. Dies gilt insbesondere für Lasten- und Pflichtenhefte, so wie sie in der industriellen Praxis eingesetzt werden. Das Problem an natürlichsprachlichen Dokumenten ist, dass sie zuweilen unpräzise und mehrdeutig sind. Eine detaillierte Ausformulierung von problematischen Formulierungen würde dieses Problem u. U. beheben, ist jedoch sehr aufwändig und kann die Lesbarkeit beeinflussen. Andererseits handelt es sich bei den hier betrachteten Systeme unter Test um eingebettete Systeme, die sich sowohl aus Hardware- als auch aus Softwarekomponenten zusammensetzen. Ein formaler Umgang mit solchen Systemen ist ohne eine entsprechende Modellbildung nicht möglich. Abbildung 4.1 illustriert diese Ausgangssituation.

Wie ich bereits in Abschnitt 1.2 auf Seite 3 dargestellt habe, ist eine präzise und eindeutige Spezifikation jedoch zwingende Voraussetzung für eine umfassende, systematische und maschinengestützte Prüfung des Systems unter Test. Da weder Spezifikation noch System unter Test in einer formal handhabbaren Form vorliegen, stehen unter dem Aspekt der Qualitätssicherung und speziell unter dem Aspekt des Nachweises der Konformität zwischen dem System unter Test und der Spezifikation die folgenden zwei Fragen zur Diskussion:

- Wie lässt sich Konformität in diesem Zusammenhang formal fassen?
- Wie kann auf einer solchen formalen Basis ein systematischer und automatisierter Konformitätsnachweis erbracht werden?

Eine Möglichkeit diese Problematik und die daraus resultierenden Fragestellungen anzugehen ist ein häufig in der Literatur [1, 115, 105] beschriebenes Gedankenexperiment, das Abbildung 4.2 auf der nächsten Seite veranschaulicht. In diesem Gedankenexperiment wird zunächst angenommen, dass die vorhandene natürlichsprachliche Spezifikation formalisiert und damit eine präzise und widerspruchsfreie Basis für den Konformitätstest erzeugt werden kann

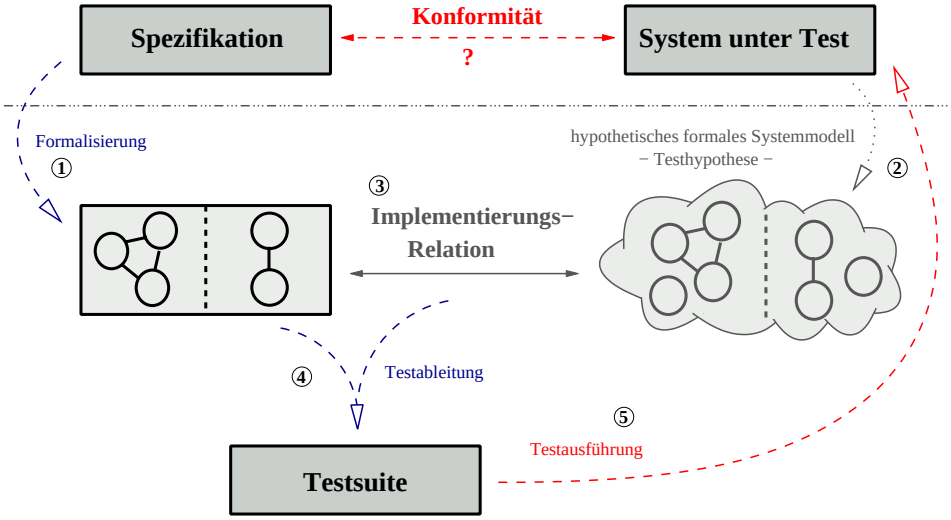


Abbildung 4.2: Zusammenhänge, die sich aus der *Testhypothese* ergeben.

(Punkt 1). Eine solche Formalisierung der Spezifikation ist in der industriellen Praxis sicherlich nicht immer üblich, kann aber vorausgesetzt werden, ohne dass dadurch die Anwendbarkeit des hier beschriebenen Ansatzes unnötig eingeschränkt wird. Die gleiche Vorgehensweise kann jedoch nicht auf das System unter Test angewendet werden. Bei diesem handelt es sich im Allgemeinen um ein reales physisches Objekt, dessen detaillierte innere Struktur und dessen Verhalten meist nicht bekannt sind. Allgemein kann man das System unter Test nur als eine mit ihrer Umgebung interagierende *Black-Box* betrachten.

Somit ist die Repräsentation des Systems unter Test zunächst nicht geeignet, um auf deren Basis formale Aussagen zu treffen. Das Gedankenexperiment besagt nun folgendes: Formal kann in diesem Zusammenhang mit einem System unter Test nur umgegangen werden, wenn angenommen wird, dass die relevanten Eigenschaften des Systems unter Test durch ein formales Modell beschrieben werden können (Punkt 2). Es ist nicht nötig, dass ein solches Modell auch real existiert, sondern es genügt die Annahme, dass ein solches Modell existiert. Aus diesem Grund wird das Modell auch als *hypothetisches* Modell bezeichnet und die Annahme, dass die relevanten Eigenschaften des Systems unter Test durch ein formales Modell beschrieben werden könne, wird als Testhypothese [1, 115, 105] bezeichnet.

Ein solches hypothetisches Modell erlaubt es nun, mit dem System unter Test formal umzugehen und auf Basis der formalisierten Spezifikation und des Modells des Systems unter Test eine formale Relation zwischen diesen beiden Modellen zu definieren (Punkt 3). Dies ist insbesondere dann möglich, wenn die formalisierte Spezifikation und das hypothetische Modell des Systems unter Test im gleichen Formalismus vorliegen. Zum Beispiel, wenn die Spezifikation in Form einer Zustandsmaschine formalisiert werden kann und angenommen werden kann, dass die relevanten Eigenschaften des Systems unter Test ebenfalls durch eine Zustandsmaschine beschrieben werden können. Eine solche Relation wird in der Literatur als *Implementierungsrelation* bezeichnet [14, 105]. Um die Lesbarkeit zu erhöhen, verwende ich im Folgenden für die formalisierte Form einer natürlichsprachlichen Spezifikation einfach den Begriff Spezifikation.

Im Rahmen dieser Arbeit verwende ich die Idee der Testhypothese, um durch das Ausführen von Testfällen untersuchen zu können, ob das unbekannte formale Modell des Systems unter Test konform zur Spezifikation des Systems ist, also in einer Implementierungsrelation zur Spezifikation steht. Für die Überprüfung werden Testfälle auf Basis einer Zustandsmaschine und einer konkreten Implementierungsrelation abgeleitet (Punkt 4) und diese gegen das reale System unter Test ausgeführt (Punkt 5). Folglich ist es durch die Annahme der Testhypothese möglich, die Frage der Konformität auf die Ausführung und Bewertung von Testfällen zurückzuführen.

Bemerkung: Die beschriebene Vorgehensweise, auf Basis einer formalen Spezifikation und eines hypothetischen Modells des Systems unter Test Konformität zu definieren, kann universell angewendet werden. Anstatt Zustandsmaschinen zu verwenden, hätte man z. B. auch algebraische Spezifikationen verwenden können. Beachtet werden muss, dass die relevanten Eigenschaften des Systems unter Test im gewählten Formalismus spezifiziert werden können müssen oder, andersherum ausgedrückt, dass der gewählte Formalismus die ausdrückbaren Eigenschaften des Systems unter Test beschränkt. Die Festlegung der relevanten Eigenschaften schränkt weiterhin die Art des Fehlverhaltens ein, das auf Basis der Implementierungsrelation identifiziert werden kann.

4.2 Implementierungsrelationen

In der Literatur sind verschiedene Implementierungsrelationen, zumeist für synchrone Systeme, beschrieben. So zum Beispiel schwache und starke Bisimulation [67], *Failure-Equivalence* und *-Preorder* [43] und *Testing Equivalence* und *-Preorder* [23, 21]. Die Unterschiede der einzelnen Relationen werde ich im Folgenden nicht erläutern, sondern nur die Grundlagen der in dieser Arbeit umgesetzten Implementierungsrelation beschreiben. Eine ausführliche Beschreibung verschiedener Relationen kann in [105] nachgelesen werden.

Ausgangspunkt für die meisten Implementierungsrelationen ist die Bisimulation. Dieser zustandsbasierten Betrachtung von Systemen liegt eine Äquivalenzrelation zugrunde, in der die Zustände miteinander in Beziehung gesetzt werden, die sich »gleich verhalten«. Das bedeutet für zwei Zustände, dass der eine Zustand die Übergänge des anderen simulieren kann und umgekehrt. In diesem Sinne können sie von einem außenstehenden Beobachter nicht voneinander unterschieden werden. Im Gegensatz zur zustandsbasierten Betrachtung bei Bisimulation verwende ich in dieser Arbeit einen pfadbasierten Ansatz, d. h. zur Definition von Konformität werden Systemläufe herangezogen. Dabei liegt, wie im vorherigen Abschnitt beschrieben, die Spezifikation in Form einer Zustandsmaschine vor und es wird angenommen, dass die relevanten Eigenschaften des Systems unter Test ebenfalls durch eine Zustandsmaschine beschrieben werden können. Bezüglich einer Menge von Zustandsmaschinen SM ist eine Implementierungsrelation eine zweistellige Relation: $imp : SM \times SM$.

Der in dieser Arbeit beschriebene Ansatz setzt voraus, dass das zu testende System über Ein- und Ausgaben mit seiner Umgebung kommuniziert und sich korrekt verhält, wenn es für alle möglichen Eingaben aus der Umgebung die »richtigen« Ausgaben zurückliefert. Somit können mögliche *Ausgabefehler* des Systems unter Test identifiziert werden. Um in einer Implementierungsrelation das Ein-/Ausgabeverhalten geeignet zu betrachten, werde ich deshalb

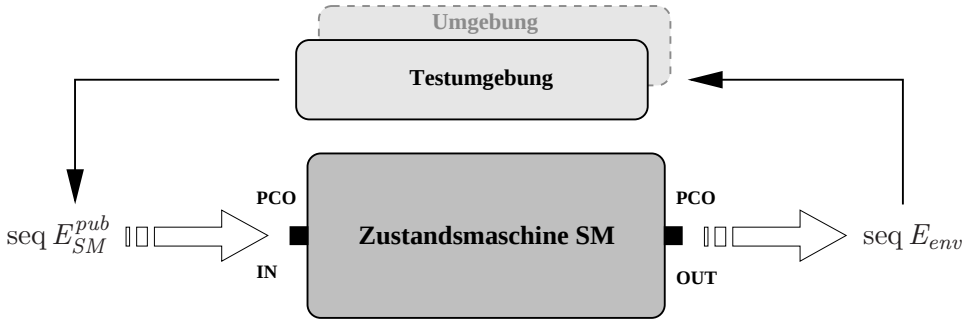


Abbildung 4.3: Abstrakte Testarchitektur.

ein Beobachtungsmodell für Zustandsmaschinen definieren. Dieses Modell bildet die Grundlage für die Definition einer geeigneten Implementierungsrelation und stellt im Wesentlichen eine Abstraktion des semantischen Modells dar, das auf Basis der semantischen Schritte einer Zustandsmaschine gebildet werden kann. Um das Beobachtungsmodell definieren zu können, werde ich zunächst beschreiben, wie einzelne semantische Schritte einer Zustandsmaschine zu einem Systemlauf zusammengesetzt werden. Auf Basis der Systemläufe definiere ich dann die Menge der beobachtbaren Systemläufe. Diese bilden die Basis, um die für die Testfallableitung benötigte Menge der korrekten Beobachtungen zu einer konkreten Eingabe zu bestimmen.

4.2.1 Beobachtungsmodell für Zustandsmaschinen

In der Definition 25 eines semantischen Schritts auf Seite 47 habe ich beschrieben, wie eine Zustandsmaschine von einem semantischen Zustand in den nächsten semantischen Zustand übergeht. Dabei wird in Abhängigkeit von der aktuellen Konfiguration, dem aktuellen Ereignisspeicher und der aktuellen Datenraumbelegung eine wohlgeformte Menge von aktivierten Transitionen bestimmt und diese während des semantischen Schritts ausgeführt. Als Folge des semantischen Schritts erhält man eine neue Konfiguration, einen neuen Ereignisspeicher und einen aktualisierten Datenraum. Zudem empfängt die Zustandsmaschine eine Sequenz von Ereignisinstanzen (E_{in}) aus der Umgebung und liefert als Ergebnis des semantischen Schritts eine Sequenz von Ereignisinstanzen (E_{out}), die während der Ausführung erzeugt wurden.

Abbildung 4.3 zeigt die abstrakte Testarchitektur, wie ich sie in der vorliegenden Arbeit verwende. Die Testarchitektur verdeutlicht die in Abschnitt 4.1 beschriebenen Aspekte bezüglich des Systems unter Test. Das System unter Test wird hier als eine *Black-Box* betrachtet, die über eine *eingehende* und eine *ausgehende* Schnittstelle mit ihrer Umgebung kommuniziert. An der eingehenden Schnittstelle können die Ereignisse beobachtet werden, die aus der Umgebung an das System geschickt werden (seq E_{SM}^{pub}). An der ausgehenden Schnittstelle können die Ereignisse beobachtet werden, die das System an seine Umgebung schickt (seq E_{env}). Während des Testens wird die reale Umgebung durch eine Testumgebung ersetzt. Dabei wird die eingehende Schnittstelle zur Stimulation bzw. Kontrolle des Systems unter Test benutzt und die ausgehende Schnittstelle zur Beobachtung der Reaktionen. Die beiden Sequenzen von Ereignisinstanzen, die während eines semantischen Schritts empfangen (E_{in}) bzw. neu erzeugt werden (E_{out}), entsprechen genau den Sequenzen von Ereignisinstanzen, die an der eingehenden bzw. an der ausgehenden Schnittstelle des Systems unter Test beobachtet werden können,

und stellen somit die Verbindung zwischen dem semantischen Modell der Zustandsmaschine und der abstrakten Testarchitektur her. Auf Basis der Beobachtungen an der ausgehenden Schnittstelle erfolgt die Bewertung der Testausführung. Die beiden Schnittstellen ermöglichen es, das System unter Test zu testen und bilden die Voraussetzung für die Anwendung des in dieser Arbeit beschriebenen Ansatzes.

Kombiniert man mehrere semantische Schritte einer Zustandsmaschine, so erhält man einen möglichen Systemlauf der Zustandsmaschine. Die Menge der möglichen Systemläufe repräsentiert das semantische Modell der Zustandsmaschine. Die Zustände eines solchen Modells repräsentieren die semantischen Zustände der Zustandsmaschine und die Übergänge zwischen den Zuständen repräsentieren die verschiedenen möglichen semantischen Schritte. Das semantische Modell umfasst sämtliche semantische Informationen über die Zustandsmaschine. Zu diesen gehört z. B. der jeweilige Inhalt des internen Ereignisspeichers, die jeweilige aktuelle Datenraumbelegung, die jeweilige aktuelle Zustandskonfiguration sowie die Sequenzen von eingehenden und ausgehenden Ereignisinstanzen. Beachtet werden sollte in diesem Zusammenhang, dass die Verarbeitung und Erzeugung von internen Ereignisinstanzen aus den semantischen Zuständen inferiert werden kann. Die Pfade eines solchen Modells werden als Systemläufe bezeichnet.

Definition 26 (Systemlauf)
Sei Z die Menge der semantischen Zustände zu einer Zustandsmaschine, seien $[\![c_1, q_1, d_1]\!], \ldots, [\![c_n, q_n, d_n]\!] : Z$ semantische Zustände und seien $in_1, \ldots, in_{n-1} : \operatorname{seq} E_{SM}^{pub}$ Sequenzen von eingehenden und $out_1, \ldots, out_{n-1} : \operatorname{seq} E_{env}$ Sequenzen von ausgehenden Ereignisinstanzen. Ein Systemlauf bezeichnet eine endliche Komposition von semantischen Schritten einer Zustandsmaschine.

$$[\![c_1, q_1, d_1]\!] \xrightarrow{in_1, out_1} [\![c_2, q_2, d_2]\!] \xrightarrow{in_2, out_2} \ldots \xrightarrow{in_{n-1}, out_{n-1}} [\![c_n, q_n, d_n]\!] \tag{4.1}$$

□

Die möglichen Systemläufe des semantischen Modells bilden die Basis für die Erstellung der Testfälle. Im Allgemeinen enthalten sie jedoch mehr Informationen, als für eine sinnvolle Ableitung von Testfällen verwendet werden können. Wie bereits beschrieben, wird das System unter Test als eine *Black-Box* mit einer eingehenden und einer ausgehenden Schnittstelle betrachtet. Es können folglich nur Informationen an diesen Schnittstellen beobachtet werden. Aus diesem Grund bilde ich zur Vereinfachung eine Abstraktion auf dem semantischen Modell und nenne ein solches Modell *Beobachtungsmodell* einer Zustandsmaschine. Das Beobachtungsmodell enthält nur die Informationen, die an der eingehenden bzw. an der ausgehenden Schnittstelle des Systems unter Test beobachtet werden können. Die Pfade eines Beobachtungsmodells erfassen somit lediglich die beobachtbaren Aspekte eines Systemslaufs. Ich bezeichne mit O die Menge aller beobachtbaren Ereignisinstanzen, bestehend aus den öffentlichen Ereignisinstanzen der Zustandsmaschine und den Ereignisinstanzen der Umgebung: $O == E_{SM}^{pub} \uplus E_{env}$ (siehe Abbildung 3.5 auf Seite 46).

Definition 27 (Beobachtbarer Systemlauf)
Sei $[\![c_1, q_1, d_1]\!] \xrightarrow{in_1, out_1} \ldots \xrightarrow{in_{n-1}, out_{n-1}} [\![c_n, q_n, d_n]\!]$ ein Systemlauf. Dann ergibt sich der beobachtbare Systemlauf zu diesem Systemlauf aus der Konkatenation der Sequenzen der eingehenden und ausgehenden Ereignisinstanzen:

$$in_1 \frown out_1 \frown \ldots \frown in_{n-1} \frown out_{n-1} \tag{4.2}$$

□

Durch die Reduktion des semantischen Modells auf das Wesentliche bilden das Beobachtungsmodell einer Zustandsmaschine und die daraus resultierenden beobachtbaren Systemläufe eine praktikable Basis für die Ableitung von Testfällen für einen Konformitätstest. Zur Vereinfachung der nachfolgenden Definitionen führe ich zusätzlich die folgenden abkürzenden Notationen ein.

Definition 28 (Übergangsrelationen)
Sei Z die Menge der semantischen Zustände zu einer Zustandsmaschine, seien $z, z' : Z$ semantische Zustände und seien $\alpha, \delta : \text{seq } O$ Sequenzen von Ereignisinstanzen.

$$z \xrightarrow{\alpha} z' == \exists\, i, o : \text{seq } O \bullet \alpha = i \frown o \land z \xrightarrow{i,o} z' \tag{4.3}$$

$$\begin{aligned} z \overset{\delta}{\Longrightarrow} z' == \exists\, \alpha_1, \ldots, \alpha_{n-1};\ z_1, \ldots z_n : Z \bullet\ & \delta = \alpha_1 \frown \cdots \frown \alpha_{n-1} \\ & \land\ z = z_1 \land z_1 \xrightarrow{\alpha_1} z_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{n-1}} z_n \land z_n = z' \end{aligned} \tag{4.4}$$

$$z \overset{\delta}{\Longrightarrow} == \exists\, z'' : Z \mid z \overset{\delta}{\Longrightarrow} z'' \tag{4.5}$$

□

Mit Hilfe dieser Abkürzungen und auf Basis der Definition 27 eines beobachtbaren Systemlaufs kann nun die Menge der beobachtbaren Systemläufe definiert werden, die ausgehend von einem Status z in einer Zustandsmaschine möglich sind.

Definition 29 (Menge der beobachtbaren Systemläufe)
Sei $z : Z$ ein Status der Zustandsmaschine S. Die Menge $otraces(z)$ umfasst die Menge der beobachtbaren Systemläufe, die ausgehend vom Status z möglich sind.

$$otraces(z) == \{\delta : \text{seq } O \mid z \overset{\delta}{\Longrightarrow} \} \tag{4.6}$$

Im Allgemeinen wird man für den semantischen Zustand z einen initialen Zustand verwenden. Für den Fall, das von einem konkreten *initialen* Zustand der Zustandsmaschine S abstrahiert werden kann, schreibe ich abkürzend $otraces(S)$ für die Menge der Pfade der Zustandsmaschine S, ausgehen von einem initialen Zustand. □

In Folgenden verwende ich die Menge aller beobachtbaren Systemläufe einer Zustandsmaschine als Grundlage für die Definition einer Implementierungsrelation. Somit dient diese Menge ebenfalls als Grundlage für die Ableitung von Testfällen, die zur Überprüfung der Konformität herangezogen werden.

4.2.2 Implementierungsrelation für Zustandsmaschinen

Eine naheliegende Implementierungsrelation im Rahmen eines pfadbasierten Ansatzes ist die gegenseitige Pfadinklusion. Bezogen auf diese Arbeit würde dies die gegenseitige Inklusion der beobachtbaren Systemläufe bedeuten. Diese Äquivalenz fordert, dass das beobachtbare Verhalten einer Spezifikation $S : SM$, vollständig und nur dieses, von einem System unter Test $I : SM$ umgesetzt werden muss:

$$I =_{otr} S \Leftrightarrow otraces(I) = otraces(S) \tag{4.7}$$

Die Forderung, dass die jeweiligen Mengen der beobachtbaren Systemläufe äquivalent sein müssen, ist für den praktischen Einsatz im Allgemeinen zu streng. Dafür gibt es unterschiedliche Gründe. Speziell für diese Arbeit möchte ich auf die Problematik von Nichtdeterminismus eingehen. Das folgende Beispiel soll dabei helfen, diesen Sachverhalt zu verdeutlichen:

> *Die Spezifikation eines Getränkeautomaten besagt, dass nach Einwurf einer Münze der Automat entweder Tee oder Kaffee zubereiten soll. Der Benutzer des Automaten hat keine Möglichkeit, ein bestimmtes Getränks auszuwählen.*

Die Frage die sich im Allgemeinen stellt ist, wie man mit spezifiziertem Nichtdeterminismus umgehen möchte. Man kann Nichtdeterminismus z. B. als Spezifikationsmittel begreifen. In einem solchen Fall darf die Implementierung diesen Nichtdeterminismus einschränken. Daraus folgt, dass die Menge der beobachtbaren Systemläufe kleiner sein darf, als die Menge der spezifizierten beobachtbaren Systemläufe. Man kann Nichtdeterminismus aber auch als Variabilitätsforderung begreifen. In einem solchen Fall darf der Nichtdeterminismus nicht durch die Implementierung begrenzt werden, sondern diese muss alle möglichen Verhalten anbieten. In diesem Fall müssen die Mengen von beobachtbaren Systemläufen äquivalent sein.[1]

Möchte man beim Konformitätstest beide Arten zulassen, dann muss die ursprüngliche Forderung der gegenseitigen Pfadinklusion in Äquivalenz (4.7) abgeschwächt werden. Die Abschwächung besteht darin, dass nun lediglich gefordert wird, dass das beobachtbare Verhalten einer Implementierung $I : SM$ im beobachtbaren Verhalten der Spezifikation $S : SM$ enthalten sein muss:

$$I \leq_{otr} S \Leftrightarrow otraces(I) \subseteq otraces(S) \tag{4.8}$$

Aus der Äquivalenz (4.8) folgt, dass eine korrekte Implementierung nur Verhalten aufzeigen darf, dass auch in der Spezifikation begründet werden kann. Die Implementierungsrelation $\leq_{otr}$ ist eine Quasiordnung, d. h., dass die Relation reflexiv und transitiv ist. Die Reflexivität sichert in diesem Zusammenhang zu, dass jede Spezifikation auch eine korrekte Implementierung von sich selbst ist. Die Transitivität sichert zu, dass auch über mehrere Schritte hinweg Konformität geschlussfolgert werden kann. Die Implementierungsrelation ist nicht antisymmetrisch, d. h., dass mit Hilfe der Implementierungsrelation nicht auf die Identität geschlossen werden kann.

Die Definition der Implementierungsrelation in Äquivalenz (4.8) suggeriert, dass für den Konformitätscheck die vollständigen Mengen der beobachtbaren Systemläufe bestimmt werden müssen. Eine andere, äquivalente Charakterisierung, aus der die Vorgehensweise zur Durchführung von Konformitätstests eleganter abgeleitet werden kann, wurde von De Nicola und Hennessy vorgeschlagen [23, 21]. In dieser Charakterisierung wird über die Beobachtungen *obs* argumentiert, die ein beliebiger externer Beobachter $o : \mathcal{O}$ bezüglich der Implementierung und der Spezifikation machen kann:

$$I \leq_{otr} S \Leftrightarrow \forall\, o : \mathcal{O} \bullet obs(I, o) \sqsubseteq obs(S, o) \tag{4.9}$$

[1] Bei der Ausführung und Bewertung der Testfälle in der Werkzeugumgebung TEAGER können diese Varianten anhand von unterschiedlichen Bewertungsstrategien geprüft werden.

Die Äquivalenz (4.9) besagt, dass eine Implementierung I eine Spezifikation S korrekt implementiert, falls die Beobachtungen *obs*, die beliebige externe Beobachter $o : \mathcal{O}$ bezüglich der Implementierung und der Spezifikation machen können, in einer hier nicht weiter spezifizierten Beziehung $\sqsubseteq$ zueinander stehen. Durch die Variation der Beobachter o, der möglichen Beobachtungen *obs* und der Relation $\sqsubseteq$ können verschiedene Ausprägungen einer solchen Charakterisierung einer Implementierungsrelation definiert werden.

Um die Implementierungsrelation aus Äquivalenz (4.9) für den Konformitätstest in dieser Arbeit anwenden zu können, müssen demzufolge die Menge der Beobachter $\mathcal{O}$ und die möglichen Beobachtungen *obs*, die die Beobachter machen können, definiert werden. Die Art der Tests, die ich mit dem System unter Test durchführen kann, ist, dass ich die Ausgaben des Systems unter Test für eine konkrete Eingabe beobachte und diese anschließend mit den erwarteten Beobachtungen vergleiche. Eine Eingabe an das System unter Test ist dabei eine (endliche) Sequenz von Ereignisinstanzen. Die Beobachtungen bestehen ebenfalls aus einer (endlichen) Sequenz von Ereignisinstanzen. Die Beziehung zwischen einer konkreten Eingabe und den möglichen Beobachtungen zu dieser Eingabe kann über die beobachtbaren Systemläufe aus Definition 27 auf Seite 58 hergestellt werden: Ein beobachtbarer Systemlauf setzt sich aus den eingehenden und aus den ausgehenden Sequenzen von Ereignisinstanzen zusammen. D. h., dass man die beobachtbaren Systemläufe dahingehend filtern kann, dass die darin enthaltenen eingehenden Sequenzen von Ereignisinstanzen mit der betrachteten Eingabesequenz übereinstimmen. Filtert man nun aus dieser Teilmenge die Sequenzen der ausgehenden Ereignisinstanzen heraus, dann erhält man die Menge der beobachtbaren Ausgabesequenzen zu einer konkreten Eingabesequenz:

Definition 30 (Menge der Ausgabesequenzen für eine gegebene Eingabe)
Sei S eine Zustandsmaschine und sei die Eingabesequenz $\sigma : \operatorname{seq} E_{SM}^{pub}$ eine Sequenz von öffentlichen Ereignisinstanzen der Zustandsmaschine S. Die Menge aller Ausgabesequenzen der Zustandsmaschine S bezüglich der Eingabesequenz σ ist wie folgt definiert:

$$out(S, \sigma) == \{\delta : otraces(S) \mid \sigma = \delta \upharpoonright E_{SM}^{pub} \bullet \delta \upharpoonright E_{env}\} \tag{4.10}$$

□

Die Eingabesequenz σ bezeichne ich im Folgenden auch als *Burst*. Auf Basis der Definitionen 30 kann nun die Implementierungsrelation aus Äquivalenz (4.9) für die vorliegende Arbeit konkretisiert werden. Die Menge der möglichen Beobachter wird durch die Menge der möglichen Eingabesequenzen $\sigma : \operatorname{seq} E_S^{pub}$ einer Zustandsmaschine bestimmt. Die möglichen Beobachtungen, die ein Beobachter, d. h. eine feste Eingabesequenz σ, bezüglich einer Zustandsmaschine S machen kann ist durch die Menge der entsprechenden Ausgabesequenzen $out(S, \sigma)$ bestimmt. Motiviert durch die unterschiedlichen Interpretationsmöglichkeiten von Nichtdeterminismus, verwende ich für die Relation $\sqsubseteq$ die Mengeninklusion $\subseteq$. Die in dieser Arbeit verwendete Implementierungsrelation $\leq_{out}$ für Zustandsmaschinen ist wie folgt definiert:

Definition 31 (Implementierungsrelation für Zustandsmaschinen)
Seien I und S zwei Zustandsmaschinen. Die Zustandsmachine I ist *konform* zur Zustandsmachine S genau dann, wenn die Ausgaben, die die Implementierung I bezüglich jeder Eingabe

$\sigma : \operatorname{seq} E_S^{pub}$ produziert, auch von der Spezifikation S produziert werden können.

$$I \leq_{out} S \Leftrightarrow \forall \sigma : \operatorname{seq} E_S^{pub} \bullet out(I, \sigma) \subseteq out(S, \sigma) \tag{4.11}$$

□

Die Implementierungsrelation $\leq_{out}$ besagt, dass eine Implementierung dann eine korrekte Implementierung der Spezifikation S ist, wenn sie zu jeder Eingabesequenz $\sigma : \operatorname{seq} E_S^{pub}$ nur Ausgabesequenzen produziert, die auch von der Spezifikation produziert werden können. Folglich weicht ein System unter Test von seiner Spezifikation ab, falls für eine Eingabesequenz von Ereignisinstanzen die beobachteten Ausgaben nicht bezüglich der Spezifikation begründet werden können. Da in dieser Arbeit die Spezifikation und das System unter Test Zustandsmaschinen sind, ist es möglich, beliebige Eingabesequenzen zu betrachten. Zustandsmaschinen besitzen die Eigenschaft, dass sie zu jedem beliebigen Zeitpunkt jede beliebige Eingabe verarbeiten können und nicht für bestimmte Eingaben blockieren (siehe Kapitel 3; insbesondere Definition 25 auf Seite 47). Dies wird in der Literatur als eingabebereit (*input enabled*) beschrieben.

Bevor ich im Folgenden beschreibe, wie Testfälle unter Berücksichtigung von Definition 31 abgeleitet und ausgeführt werden können, möchte ich noch ein Thema diskutieren, dass insbesondere in der praktischen Anwendung der Implementierungsrelation relevant wird.

4.2.3 Beschränkung der Testschnittstellen

Wie in Abbildung 4.3 auf Seite 57 dargestellt ist, besitzt das System unter Test eine eingehende und eine ausgehende Schnittstelle, die für die Durchführung von Tests benutzt werden können. Definition 31 besagt weiter, dass für alle Eingabesequenzen an der eingehenden Schnittstelle die Menge der beobachteten Ausgabesequenzen an der ausgehenden Schnittstelle in der Menge der möglichen Ausgabesequenzen der Spezifikation enthalten sein müssen. Für beide Schnittstellen, d. h. für die möglichen Beobachtungen an diesen Schnittstellen, wird man in der praktischen Anwendung der Definition 31 weitere Einschränkungen zulassen.

In der industriellen Praxis werden sehr häufig partielle Spezifikationen verwendet. Partiell bedeutet, dass die Spezifikation nicht für alle möglichen Eingaben der Implementierung festlegt, wie sich die Implementierung für diese Eingaben verhalten soll. Im Allgemeinen bedeutet dies, dass die Menge der eingehenden Ereignisinstanzen der Spezifikation eine Teilmenge der eingehenden Ereignisinstanzen der Implementierung ist. Aus diesem Grund werden in Definition 31 nur Eingabesequenzen bezüglich der Spezifikation betrachtet. In der praktischen Anwendung der Testfallableitung kann man diesbezüglich noch einen Schritt weiter gehen und die Menge der betrachteten Ereignisinstanzen (beliebig) weiter einschränken. Somit ist eine Beschränkung der möglichen Eingabesequenzen und eine zielgerichtete Ableitung von Testfällen möglich. Bezüglich der eingehenden Schnittstelle ergeben sich zwischen den betrachteten Eingaben einer Implementierung I, einer Spezifikation S und einer Testsuite T folgende Zusammenhänge: $\operatorname{seq} E_I^{In} \subseteq \operatorname{seq} E_S^{In} \subseteq \operatorname{seq} E_T^{In}$. Abbildung 4.4 auf der nächsten Seite illustriert diese Zusammenhänge.

Zum anderen kann es der Fall sein, dass das System unter Test mehr Ausgaben an seine Umgebung abgeben kann, als bezüglich der Spezifikation betrachtet werden. So kann z. B. ein

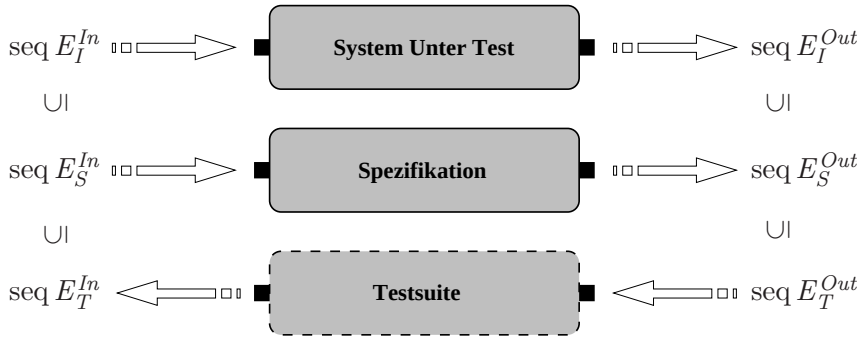


Abbildung 4.4: Zusammenhänge zwischen den Testschnittstellen.

System unter Test ein periodisches Ereignis versenden, mit dem es signalisiert, dass es noch einwandfrei funktioniert. Ein solches Verhalten muss nicht zwingend in der Spezifikation enthalten sein. Somit ist die ausgehende Schnittstelle des Systems unter Test umfangreicher, als in der Spezifikation angenommen. Tritt ein solches zusätzliches Ereignis während der Testausführung auf, dann würde dies zu einer Abweichung von den erwarteten Beobachtungen führen. Aus diesem Grund schränkt man die Schnittstelle zur Bewertung der Antwort des Systems unter Test i. A. (beliebig) weiter ein. Realisiert wird die Einschränkung dadurch, dass die Ausgaben der Implementierung nur bezüglich der ausgehenden Schnittstelle geprüft werden und alle übrigen Ausgaben herausgefiltert werden. Somit ergeben sich zwischen den betrachteten Ausgaben einer Implementierung I, einer Spezifikation S und einer Testsuite T folgende Zusammenhänge: $\text{seq}\, E_I^{Out} \supseteq \text{seq}\, E_S^{Out} \supseteq \text{seq}\, E_T^{Out}$

Auf Basis der Definition 31 kann man nun eine Testsuite erzeugen, die *sound* und *exhaustive* [105] ist (Eigenschaften einer Testsuite siehe Abschnitt 2.3 auf Seite 13). Präzise bedeutet dies, dass eine solche Testsuite in der Lage wäre, exakt zwischen korrekten und inkorrekten Systemen unter Test zu unterscheiden. Dies ist jedoch nur ein theoretisches Resultat, da eine solche Testsuite i. A. unendlich groß wäre. In Definition 31 kann man dies leicht an der Quantifizierung über allen möglichen Eingabesequenzen sehen. Für eine praktische Anwendung muss insbesondere beachtet werden, dass der Berechnungsaufwand exponentiell mit der Länge der Eingabesequenzen steigt und es somit praktisch nicht möglich ist, für beliebig lange Eingabesequenzen die möglichen Ausgabesequenzen zu bestimmen. Aus diesem Grund beschränke ich mich in dieser Arbeit auf eine Testsuite, die *sound* ist und mit begrenztem Berechnungsaufwand bestimmt werden kann. Ziel von Abschnitt 4.3 wird es sein, zu beschreiben, wie eine solche Testsuite abgeleitet werden kann. Die Testsuite soll dabei nicht nur *sound* sein, sondern zudem eine hohe Wahrscheinlichkeit besitzen, fehlerhafte Systeme unter Test zu erkennen. Um dies zu erreichen, werden die Eingabesequenzen bezüglich einer Testspezifikation ausgewählt. Um den Berechnungsaufwand möglichst gering zu halten, werden Eingabesequenzen einer festen Länge betrachtet und so die Exploration des semantischen Zustandsraums beschränkt. Durch das Zusammenfassen von semantischen Zuständen werden dann Testfälle miteinander kombiniert, um dennoch möglichst lange Eingabesequenzen testen zu können. Dabei führt jedoch die Zusammenfassung von semantischen Zuständen zu einer Approximation des korrekten Verhaltens.

Teil II

Praktische Evaluation

Kapitel 5

Werkzeugumgebung TEAGER

Zur Evaluation des in Kapitel 3 und Kapitel 4 entwickelten Konformitätstest auf Basis von Zustandsmaschinen habe ich die prototypische Werkzeugumgebung TEAGER implementiert. TEAGER ist ein Akronym, das zur Cebit 2003 entstanden ist und steht für *Test Execution and Generation Environment for Reactive Systems*. Die Werkzeugumgebung ermöglicht es, Testfälle gemäß der Definition für Konformität (Definition 31) und der in Abschnitt 4.3 besprochenen Umsetzung abzuleiten und anschließend gegen ein System unter Test auszuführen. Dabei erfolgt mit Hilfe unterschiedlicher Selektionsstrategien eine Auswahl zu erzeugender Testfälle anhand der Festlegung von Eingabesequenzen. Die Ausführung der Testfälle kann mit Hilfe probabilistisch gesteuerter Triggergeschwindigkeiten zur Ansteuerung der Implementierung unter Test erfolgen. Dadurch soll verhindert werden, dass bestimmte Fehler aufgrund eines Festen Zeitmodells unentdeckt bleiben.

Neben den Komponenten zur Testfallableitung und -ausführung habe ich zusätzlich einen Simulator für Zustandsmaschinen implementiert. Der Simulator setzt Feinheiten der entwickelten Semantik um und erlaubt insbesondere durch eine probabilistisch gesteuerte Ausführungszeit von Transitionen ein realistisches Ausführungsverhalten einer Zustandsmaschine zu simulieren. Durch die Simulation kann das Verhalten einer Zustandsmaschine experimentell analysiert und so die Eignung der Zustandsmaschine für die Ableitung von Testfällen beurteilt werden. Weiterhin kann der Simulator aber auch als System unter Test benutzt werden und somit können auf einfache Art und Weise unterschiedliche Testszenarien zur Verfügung gestellt werden. Zwei Fallstudien, die ich zur Erprobung der Werkzeugumgebung benutzt habe, sowie die Interpretation der experimentellen Ergebnisse werde ich in Kapitel 6 vorstellen.

In Abschnitt 5.1 werde ich die abstrakte Architektur der Werkzeugumgebung TEAGER vorstellen und dabei die generellen Abläufe während der Benutzung darstellen. In den darauf folgenden Abschnitten 5.2 bis 5.5 werde ich die einzelnen Bestandteile der Werkzeugumgebung detailliert erklären. Dafür werde ich anhand der unterschiedlichen Ansichten, d. h. anhand der unterschiedlichen Reiter in der grafischen Benutzeroberfläche, deren Aufgabe und Benutzung erläutern. In Abschnitt 5.2, 5.3 und 5.4 werde ich die einzelnen Bestandteile des `Test Case`

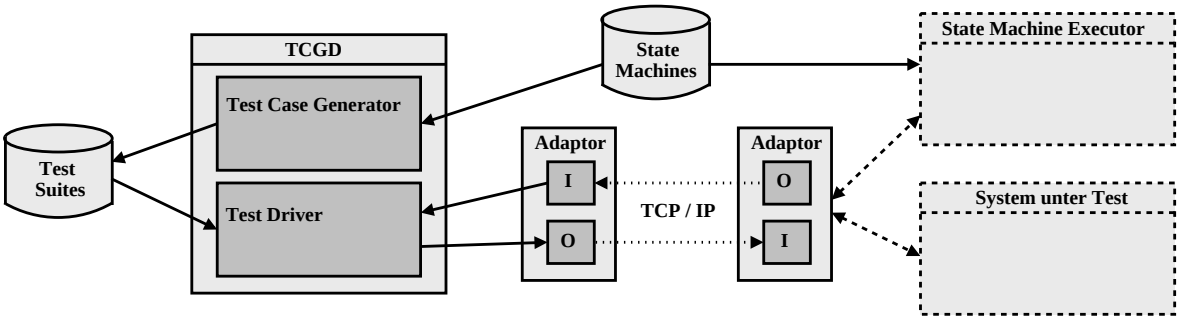


Abbildung 5.1: Abstrakte Architektur der Werkzeugumgebung TEAGER.

`Generator and Driver`s vorstellen und in Abschnitt 5.5 den `State Machine Simulator`. Abschließend werde ich in Abschnitt 5.6 das vorliegende Kapitel kurz zusammenfassen.

5.1 Architektur der Werkzeugumgebung TEAGER

Die Werkzeugumgebung TEAGER wurde in JAVA implementiert und hat derzeitig eine Größe von ca. 18 Tsd. Programmzeilen. Für die Programmierung wurde die Entwicklungsumgebung ECLIPSE inklusive des JUNIT-Plugins für die Qualitätssicherung der entwickelten Software verwendet. TEAGER setzt sich aus drei wesentlichen Komponenten zusammen: erstens aus der Komponente `Test Case Generator`, für die automatisierte Ableitung von Testfällen, zweitens aus der Komponente `Test Case Executor`, zur Ausführung dieser Testfälle gegen ein System unter Test und drittens aus der Komponente `State Machine Simulator`, die eine realistische Simulation einer Zustandsmaschine erlaubt. In Abbildung 5.1 ist die abstrakte Systemarchitektur von TEAGER wiedergegeben.

Für die Ableitung und Ausführung von Testfällen wurde der `Test Case Generator and Driver` (`TCGD`) entwickelt. Dieser setzt sich aus dem `Test Case Generator` und dem `Test Case Executor` zusammen. Über spezielle Adapter wird das System unter Test mit dem `Test Case Executor` verbunden. Die Kommunikation zwischen beiden Adaptern findet dabei auf Basis einer TCP/IP-Verbindung statt, wobei die Daten beidseitig gepuffert werden. Dabei werden die Ausgaben des `Test Case Executor` als die Eingaben des Systems unter Test und die Ausgaben des Systems unter Test als die Eingaben des `Test Case Executor` betrachtet. Die Verwendung der Adapter ermöglicht es, das System unter Test von der Werkzeugumgebung zu entkoppeln und, sofern gewünscht, auf einem anderen System auszuführen. Für die Ansteuerung des Systems unter Test über den Adapter wird ein von mir implementiertes, einfaches Protokoll verwendet. Das System unter Test wird über den Adapter angeschlossen, damit es getestet werden, kann muss es jedoch zusätzlich eine Testschnittstelle implementieren. Diese Schnittstelle umfasst die Funktionalitäten: `start()`, `stop()`, `initalize()`, `isInitialized()` und `receive(String message)`. Die Funktion `receive` bildet die Schnittstelle, um Eingaben an das System unter Test, die vom Adapter empfangen wurden, an das System unter Test zu übermitteln. Auf der Gegenseite kann das System unter Test die Funktion `send(String message)` im Adapter aufrufen, um Ausgaben an den Testserver zu versenden. Mit den Funktionen `start` und `stop` wird das System unter Test gestartet bzw. gestoppt. Die Funktionen `initalize` und `isInitialized` werden dafür benutzt, das System unter Test vor jedem Test-

fall in einen definierten Zustand zu überführen. Dabei gibt es unterschiedliche Strategien, wie eine neuerliche Initialisierung erfolgen kann. Bei Hardwarekomponenten kann die Initialisierung aus einem einfachen Rücksetzsignal oder aus einer Folge von Anweisungen, die das System in einen definierten Zustand überführen, bestehen (z. B. bei einem elektrisch verstellbaren Sitz in einem Kraftfahrzeug). Bei Softwarekomponenten können die einzelnen Objekte zerstört und neu erzeugt werden oder spezielle Funktionen zur Initialisierung implementiert werden. Wie die Initialisierung konkret umgesetzt wird, hängt vom konkreten System unter Test ab und ist daher offen gelassen.

Alternativ zu einem realen System unter Test kann über das Adapterkonzept auch der `State Machine Simulator` angeschlossen werden. Dies könnte z. B. Anwendung zur Modellprüfung finden. D. h. insbesondere, dass der Simulator für eine detaillierte Analyse des spezifizierten Verhaltens, sowie zur Analyse, ob aus der Zustandsmaschine geeignet Testfälle abgeleitet werden können (Stichwort: *Design for Testability*), verwendet werden könnte. Dabei können z. B. speziell zur Analyse des beschriebenen Verhaltens die in der Entwurfsphase aufgestellten Sequenzdiagramme in Verbindung mit einer manuellen Beurteilung der Beobachtungen zu Anwendung kommen. Ich habe in der Arbeit den Simulator insbesondere als Ersatz für ein reales System unter Test genutzt. Mit Hilfe des Simulators war mir möglich, auf einfache Art und Weise unterschiedliche Testszenarien zu realisieren und so die Testfallableitung und -ausführung zu testen. Im Folgenden werde ich den generellen Ablauf bei der Benutzung der Werkzeugumgebung beschreiben und danach im Detail die einzelnen Komponenten und deren Aufbau und Funktionsweise erläutern.

Ablauf

Die Zustandsmaschinen werden auf dem lokalen Dateisystem gespeichert und können für die Ableitung der Testfälle als Spezifikation geladen werden. Das Eingabeformat ist dabei ein textuelles Format. Die Grammatik für dieses Format, inklusive eines kleinen Beispiels, kann im Anhang B nachgelesen werden. Für die Testfallerzeugung wird eine Zustandsmaschine in den `Test Case Generator` eingelesen. Über Parameter, die im nächsten Abschnitt erläutert werden, kann die Ableitung konfiguriert werden und anschließend werden auf Basis einer Simulation der Zustandsmaschine Testfälle abgeleitet. Die erzeugten Testfälle (⇒ Dateien) werden dabei in einer Testsuite (⇒ Verzeichnis) auf dem lokalen Dateisystem abgelegt. Eine gespeicherte Testsuite wird dann vom `Test Case Executor` vom lokalen Dateisystem eingelesen und für die Ausführung die einzelnen Testfälle ausgewählt. Die Ausführung der Testfälle wird durch den `Test Case Executor` gesteuert. Besteht eine Verbindung zwischen dem `Test Case Executor` und einem System unter Test, dann wird das System unter Test durch den `Test Case Executor` aufgefordert sich zu initialisieren. Nachdem die Initialisierung vom System unter Test bestätigt wurde, wird mit der Ausführung des ersten Testfalls begonnen. War die Ausführung eines Testfalls erfolgreich bzw. ist ein Fehler aufgetreten, dann wird das System unter Test erneut aufgefordert sich zu initialisieren. Danach wird mit dem nächsten Testfall fortgefahren. Während der Ausführung eines Testfalls werden vom `Test Case Executor` Eingaben (Ereignisse) an das System unter Test gesandt und anhand der vom System unter Test empfangenen Ausgaben (Ereignisse) die Konformität mit den zuvor berechneten möglichen korrekten Ausgaben beurteilt. Der Ablauf der Testausführung ist im Zusammenspiel mit dem `State Machine Simulator` identisch.

Im Folgenden werde ich den Aufbau und die Funktionsweise des `Test Case Generators`, des `Test Case Executors`, und des `State Machine Simulators` im Detail vorstellen. Der `Test Case Generator` und der `Test Case Executor` sind in einer Komponente, dem `Test Case Generator and Driver (TCGD)`, zusammengefasst. Der `State Machine Simulator` ist eine eigenständige Komponente in der Werkzeugumgebung, die auch unabhängig vom `TCGD` ausgeführt werden kann. Für die Beschreibung werde ich mich an den unterschiedlichen Ansichten (in der grafischen Benutzeroberfläche Reiter) orientieren und anhand dieser die unterschiedlichen Funktionen beschreiben.

5.2 Konfigurationsansicht

Zum Programmstart müssen dem `TCGD` zwei Parameter übergeben werden. Mit dem ersten Parameter wird dem Programm das Arbeitsverzeichnis übergeben und mit dem zweiten Parameter wird dem Programm eine Konfigurationsdatei übergeben, die die gesicherte Einstellung der Parameter für die Testfallerzeugung und die Testfallausführung enthält. Das Arbeitsverzeichnis wird im `TCGD` als Präfix für alle weiteren Pfadangaben benutzt. D. h., dass alle weiteren Pfadangaben relativ zu diesem Arbeitsverzeichnis angegeben werden. In Programmausdruck 5.1 ist eine beispielhafte Konfigurationsdatei abgebildet. Die beiden ersten Parameter `start_view` und `max_files` können nicht über die grafische Benutzeroberfläche des `TCGD` konfiguriert werden. Der Parameter `start_view` legt fest, welche der unterschiedlichen Ansichten (Reiter) nach dem Programmstart angezeigt werden soll. Der Parameter `max_files` legt die maximale Anzahl von Dateien bzw. von Testfällen fest, die während der Testfallerzeugung generiert werden können. Der Parameter dient als Schutzmarke und soll verhindern, dass der Rechner dadurch blockiert wird, dass die Anzahl der erzeugten Testfälle (Dateien) in einer Testsuite (Verzeichnis) ungewollt extrem hoch wird. Alle weiteren Parameter können auch zur Laufzeit über die grafische Benutzeroberfläche eingestellt werden und werden im Folgenden erläutert.

```
# TCGD configuration file
2
# -- General Options ------------------------------------
4 start_view=1
max_files=500
6
# -- Specification Options ------------------------------
8 specification=../beispiele/cas-sm.spec

10 # -- Generation Options ---------------------------------
test_suite=../testsuiten/test
12 generation_rounds=25
simulation_length=15
14 burst_size=15
step_bound=1000
16 trigger_distribution=APRIORI
preamble=power back back next back back next cdin src back next
18
# -- Execution Options ----------------------------------
```

```
20 sut_port=1234
   execution_rounds=2
22 pass_condition=WEAK_MAY
   trigger_rate=25
24 trigger_rate_deviation=2.5
   observation_sampling_rate=25
26 timeout=200

28 # -- Public Event Weights ------------------------------
   this.tpej=1
30 this.back=1
   this.next=1
32 this.cdin=1
   this.power=1
34 this.tpin=1
   this.src=1
36 this.cdej=1
   this.play=1
38
   # -- State Machine Simulation Options ------------------
40 selectionMethod=uniformly_distributed
   transitionDuration=50
42 durationDeviation=1.5
```

Programmausdruck 5.1: Beispielkonfiguration des `TCGD`s.

Der folgende Aufruf startet den `TCGD`:

```
java TCGD -base /home/seifert/work/progs/TEAGER -conf src/tcgd/tcgd.conf.
```

In Abbildung 5.2 ist die Konfigurationsansicht des `TCGD` dargestellt. Die Ansicht unterteilt sich dabei in drei Teile: Unter `Specification Options` können alle Einstellungen im Zusammenhang mit der Spezifikation vorgenommen werden. Unter `Generation Options` können alle Einstellungen im Zusammenhang mit der automatischen Ableitung von Testfällen vorgenommen werden und unter `Execution Options` können alle Einstellungen im Zusammenhang mit der automatischen Ausführung von Testfällen vorgenommen werden.

Specification Options

Mit dem Parameter `Specification` kann die Zustandsmaschine, die im Folgenden als Spezifikation verwendet werden soll, festgelegt werden. Wird eine Zustandsmaschine ausgewählt, so wird diese in den Editor geladen (siehe Spezifikationsansicht), die syntaktische Korrektheit geprüft und eventuell aufgetretene Fehler im Editor angezeigt, und die bezüglich der Datenanteile benötigte Java-Klasse erzeugt und übersetzt.

Generation Options

Mit dem Parameter `Test Suite` kann das Verzeichnis zur Speicherung der generierten Testfälle festgelegt werden. Die Ableitung der einzelnen Testfälle erfolgt derart, dass jeder einzelne

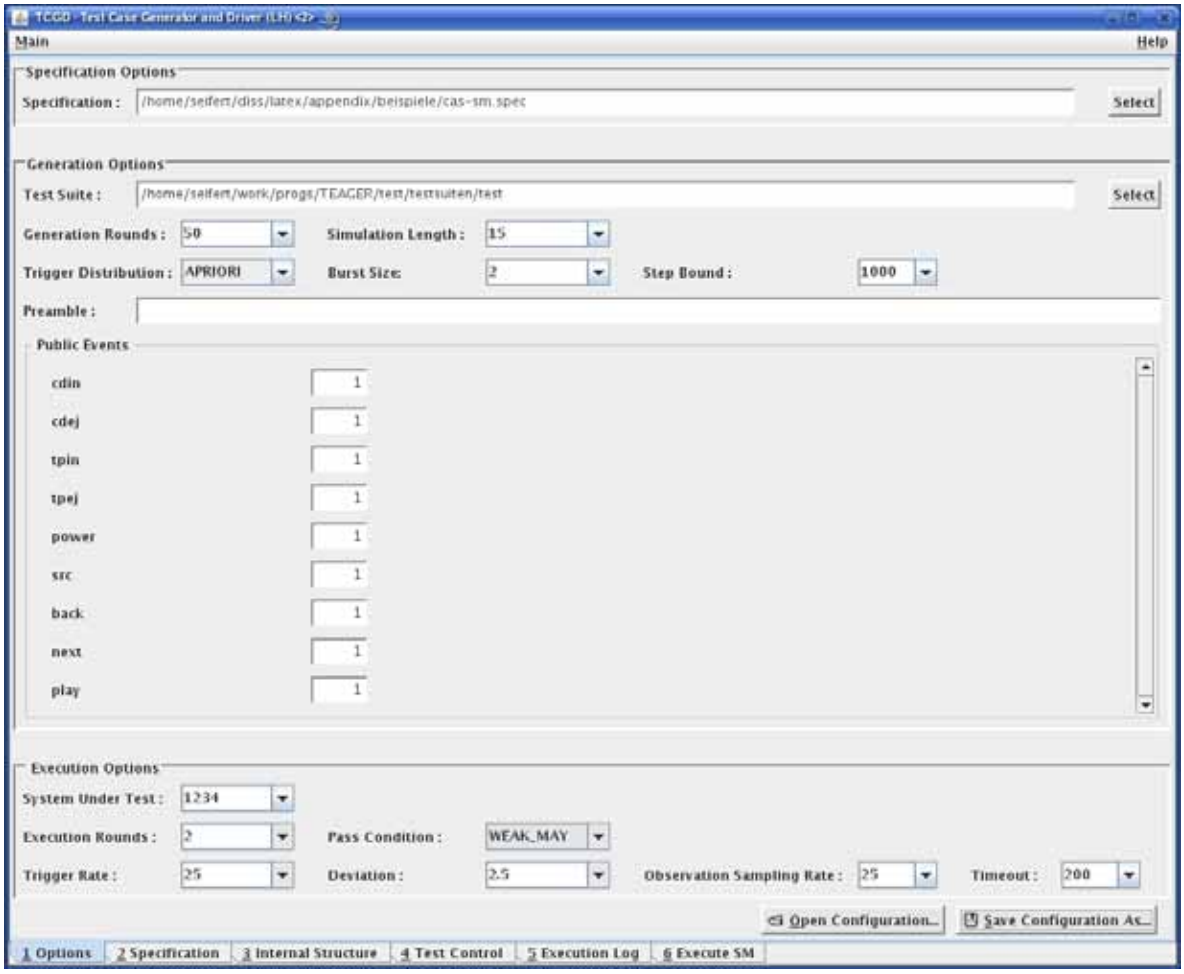


Abbildung 5.2: Konfigurationsansicht des `TCGD`.

Testfall in einer separaten Datei gespeichert wird. Die einzelne Speicherung erleichtert die Bearbeitung und Auswahl von Testfällen und somit den Umgang mit sehr großen Testsuiten.

Mit dem Parameter `Generation Rounds` wird die Anzahl der zu erzeugenden Testfälle festgelegt.

Mit dem Parameter `Simulation Length` wird die gewünschte Gesamtanzahl der Ereignisse, mit denen das System unter Test innerhalb eines Testfalls stimuliert werden soll, festgelegt. Dabei bestimmt sich die tatsächliche Anzahl aus einem vielfachen der festgelegten Burstlänge (siehe `Burst Size`).

Mit dem Parameter `Trigger Distribution` wird festgelegt, wie die Ereignisse, mit denen das System unter Test stimuliert werden soll, aus der Menge der möglichen Ereignisse ausgewählt werden. Dabei sind unterschiedliche Auswahlstrategien vorgegeben. Die Einstellung `UNIFORM` legt fest, dass die Ereignisse zufällig, gleichverteilt ausgewählt werden. Das heißt, dass jedes Ereignis mit gleicher Wahrscheinlichkeit ausgewählt werden kann. Bei der Einstellung `APRIORI` wird für die Auswahl des jeweils nächsten Ereignisses eine gegebene Wahrscheinlichkeitsverteilung angenommen. Diese kann unter dem Konfigurationspunkt `Public Events`

festgelegt werden. Bei der Einstellung `DEPENDANT` wird initial dieselbe Wahrscheinlichkeitsverteilung angenommen. Im Unterschied zur Einstellung `APRIORI` verändert sich diese Verteilung jedoch mit jeder Wahl eines Ereignisses. Das heißt im Detail, dass bei der Wahl eines Ereignisses die Gewichtung dieses Ereignisses um Eins reduziert wird. Somit erfolgt zwar immer noch eine zufällige Auswahl des jeweils nächsten Ereignisses, es wird jedoch sichergestellt, dass jedes Ereignis schließlich auch mit der entsprechenden Häufigkeit auftritt. Seien z. B. zwei Ereignisse a und b mit einer Gewichtung von $4 : 1$ gegeben. Dies bedeutet, dass im ersten Schritt mit einer Wahrscheinlichkeit 80% ein a und mit einer Wahrscheinlichkeit von 20% ein b ausgewählt wird. Nehmen wir nun beispielhaft an, dass ein a ausgewählt wurde. Dann verringert sich die Gewichtung von a auf 3. Bei der nächsten Auswahl wird mit einer Wahrscheinlichkeit von 75% ein a und mit einer Wahrscheinlichkeit von 25%ein b gewählt werden. Sollten die Gewichtungen aller Ereignisse gleich Null sein, dann wird wieder die initiale Gewichtung angenommen. Bei der Einstellung `MARKOV` wird ein spezielles Markovmodell ausgewählt. Dieses Modell beschreibt ein spezielles Nutzungsprofil für die Fallstudie `Sun Blind Control`. In einer zukünftigen Erweiterung der Werkzeugumgebung soll dies für beliebige Nutzungsprofile ausgebaut werden.

Mit dem Parameter `Burst Size` wird festgelegt, welche Länge ein einzelnes Eingabepaket haben soll. Für ein einzelnes Eingabepaket wird für einen Testfall das vollständige und korrekte Verhalten berechnet und als Orakel für die Testbewertung benutzt. Mehrere solcher Eingabepakete mit ihren jeweiligen möglichen Bobachtungen können zu einem längeren Testfall kombiniert werden. Dabei werden so viele Pakete miteinander kombiniert, bis die Anzahl von Ereignissen, die durch den Parameter `Simulation Length` festgelegt wurde, in ganzen Paketgrößen erreicht wurde. Zu beachten ist, dass nach jedem Ereignispaket ein Synchronisationspunkt eingefügt wird und die gemachten Beobachtungen überprüft werden.

Mit dem Parameter `Step Bound` wird festgelegt, wie viele Berechnungsschritte maximal ausgeführt werden, um einen stabilen Simulationszustand zu erreichen. Dabei bezeichnet stabiler Simulationszustand einen semantischen Zustand der Zustandsmaschine, in der diese nicht mehr selbständig fortfahren kann, also keine weiteren Ereignisinstanzen mehr verarbeitet werden können. Das bedeutet, dass im stabilen Simulationszustand das vollständige Verhalten der Zustandsmaschine berechnet wurde. Wird die festgelegte Anzahl von Schritten überschritten, ohne dass ein stabiler Simulationszustand erreicht wurde, dann wird die Simulation abgebrochen und entsprechend markiert. Begrenzt wird dadurch insbesondere die Verarbeitung von *internen* Schleifen. Der Zähler wird bei jedem neuen *externen* Ereignis zurückgesetzt. Eine Simulation, die abgebrochen wurde, führt zu Beobachtungen, die im Testfall mit dem Urteil *inconclusive* bewertet werden. Das heißt, dass die Beobachtung zwar korrekt sind, aber aufgrund der abgebrochenen Simulation das korrekte Verhalten nicht vollständig bestimmt werden konnte und alle weiteren Beobachtungen nicht mehr mit Sicherheit beurteilt werden können.

Mit dem Parameter `Preamble` kann eine initiale Ereignissequenz festgelegt werden. Die Ereignissequenz wird bei der Ereignisauswahl zuerst vollständig abgearbeitet und erst danach werden neue Ereignisse nach der Methode, die mit dem Parameter `Trigger Distribution` festgelegt wurde, bestimmt. Eine initiale Ereignissequenz dient vornehmlich dazu, das System unter Test in einen bestimmten Startzustand für den (eigentlichen) Test zu überführen. Weiterhin kann sie aber auch dazu benutzt werden, um das System unter Test nur mit einer bestimmten Abfolge von Ereignissen zu testen. Eine solche Sequenz stellt somit ein sehr

spezielles Nutzungsprofil des Systems unter Test dar.

Mit dem Parameter `Public Events` kann für jedes öffentliche Ereignis eine spezifische Gewichtung festgelegt werden (siehe `APRIORI` und `DEPENDANT` Auswahlstrategie). Dabei ist hier insbesondere anzumerken, dass ein Wert von 0 bedeutet, dass dieses Ereignis nicht mehr ausgewählt wird und somit die eingehende öffentliche Schnittstelle beschränkt wird (siehe Kapitel 4).

Execution Options

Mit dem Parameter `System under Test` wird festgelegt, über welche Portnummer die TCP/IP Verbindung zum System unter Test hergestellt wird. Auf Seiten des Systems unter Test muss neben dieser Portnummer zusätzlich der Servername, auf dem der `Test Case Executor` ausgeführt wird, festgelegt werden.

Mit dem Parameter `Execution Rounds` wird festgelegt, wie oft die Ausführung eines Testfalls wiederholt wird. Die Wiederholung eines Testfalls ist dann sinnvoll, wenn sich das System unter Test nichtdeterministisch verhalten kann und adäquat getestet werden soll.

Mit dem Parameter `Pass Condition` wird die anzuwendende Bewertungsstrategie für die Testausführung festgelegt. Die Einstellung `MUST` legt fest, dass jeder einzelne Testfall inklusive aller Wiederholungen erfolgreich ausgeführt werden muss (Bewertung mit dem Urteil *pass*). Die Einstellung `WEAK_MAY` legt fest, dass für eine erfolgreiche Testausführung kein Testfall fehlschlagen darf (Bewertung mit dem Urteil *fail*)). Die Einstellung `STRONG_MAY` legt fest, dass jeder Testfall zusätzlich mindestens einmal erfolgreich ausgeführt werden muss (Bewertung mit dem Urteil *pass*). Dabei wird jeder Testfall höchstens bis zu einer festgelegten Obergrenze wiederholt. Mit der Einstellung `STRONG_MAY` ist es möglich, Testfälle, die zu keinem `pass` Urteil führen zu identifizieren und dabei den Aufwand für die Erkennung solcher Testfälle relativ gering zu halten. Diese Variante wurde speziell für nichtdeterministische Systeme unter Test hinzugefügt. Eine weitere Steigerung wäre es, dass jede mögliche Beobachtung mindestens einmal auftreten muss.

Mit dem Parameter `Trigger Rate` wird festgelegt, mit welcher Rate (im Mittel) die einzelnen Ereignisse eines Testfalls an das System unter Test geschickt werden. Ein Testfall besteht aus einer alternierenden Folge von Ereignissequenzen, mit denen das System unter Test stimuliert wird, und Akzeptanzgraphen, mit deren Hilfe die Ausgaben des Systems unter Test bewertet werden.

Mit dem Parameter `Deviation` wird die Standardabweichung für den Parameter `Trigger Rate` festgelegt. Um ein möglichst realistisches Testszenario zu erhalten, wird für die Auswahl der unterschiedlichen Triggerraten eine Normalverteilung über den mit dem Parameter `Trigger Rate` festgelegten Mittelwert angenommen. Über die Variation der mit dem Parameter `Deviation` festgelegten Standardabweichung können dann die einzelnen Triggerraten verschieden stark variiert werden. Dabei werden etwa 70% der Werte aus dem Bereich Mittelwert $\pm$ Standardabweichung gewählt. Durch die Variation wird verhindert, dass ein festes Zeitmodell für den Test benutzt wird.

Mit dem Parameter `Observation Sampling Rate` wird festgelegt, mit welcher Rate der Puffer für die empfangenen Ausgaben des Systems unter Test auf neue Ereignisse geprüft

werden soll.

Mit dem Parameter `Timeout` wird schließlich die Obergrenze festgelegt, bis zu der auf eine Ausgabe des Systems unter Test gewartet werden soll (sofern nicht im Testfall für ein bestimmtes Ereignis anders angegeben). Die Festlegung einer Obergrenze betrifft insbesondere zwei Punkte. Zum einen wird dadurch angenommen, dass eine Ausgabe des Systems unter Test bis zu diesem Zeitpunkt erfolgt, und zum andern wird angenommen, dass, wenn innerhalb dieser Zeitspanne keine Ausgabe erfolgt ist, auch zukünftig keine weitere Ausgabe auftreten wird. Durch beide Punkte wird eine Testhypothese getroffen.

Sämtliche Einstellungen können in einer Konfigurationsdatei gespeichert bzw. aus einer Konfigurationsdatei ausgelesen werden.

5.3 Spezifikationsansicht

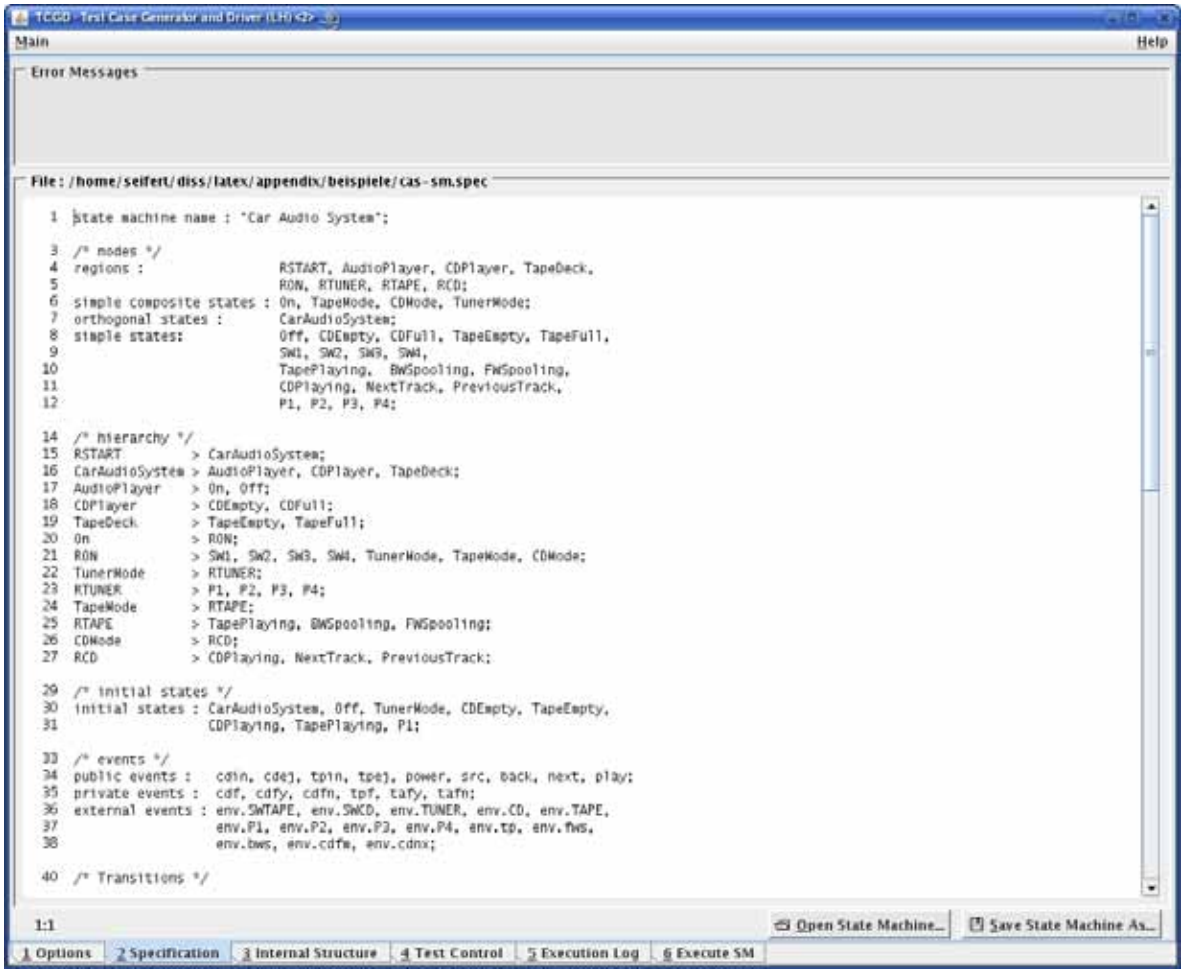
Die Spezifikationsansicht stellt primär einen Editor für die Bearbeitung einer Zustandsmaschine zur Verfügung. In Abbildung 5.3 ist diese Ansicht abgebildet. Wird eine Zustandsmaschine nach der Bearbeitung gespeichert, so erfolgt eine Überprüfung der Zustandsmaschine auf eventuelle Fehler. Sollten Fehler aufgetreten sein, dann werden die entsprechenden Fehlermeldungen im oberen Teil (`Error Messages`) der Ansicht eingeblendet. Gleichzeitig wird für alle datenbehafteten Bestandteile der Zustandsmaschine, z. B. Übergangsbedingungen, eine Java-Datei erzeugt und im Hintergrund übersetzt. Das Resultat der Übersetzung wird auf der Konsole ausgegeben und eventuell aufgetretene Fehlermeldungen werden in einer Datei gespeichert. Die generierte Datei dient während der Ausführung zur Speicherung und Auswertung der spezifischen Datenwerte. Mit den Schaltflächen am unteren rechten Rand der Ansicht kann die Zustandsmaschine unter einem neuen Namen gespeichert werden oder eine neue Zustandsmaschine geladen werden.

Neben der Spezifikationsansicht werden in der Ansicht `Internal Structure` die interne Struktur der Zustandmaschine sowie einige interne Datenstrukturen des `TCGD` und deren Belegungen abgebildet. Diese Ansicht dient primär zur Entwicklung der Werkzeugumgebung und hat keine weitere Bedeutung für die Ableitung und Ausführung von Testfällen.

5.4 Testansicht

Die Testansicht ist die Hauptansicht des `TCGD`s. In Abbildung 5.4 ist diese Ansicht abgebildet. In der Testansicht können Testfälle erzeugt, geladen, gespeichert, ausgewählt und ausgeführt werden. Im oberen Teil der Ansicht (`Logging`) werden alle Statusinformationen angezeigt und im unteren Teil werden die einzelnen Testfälle in einer Tabelle aufgelistet.

Die Testfälle werden aus dem Testsuiteverzeichnis ausgelesen. Dabei werden alle Dateien, die auf `.tc` enden eingelesen und auf Syntaxfehler überprüft. In jeder Zeile der Tabelle steht ein Testfall. In der ersten Spalte wird die Nummer des Testfalls angezeigt. In der zweiten Spalte kann ausgewählt werden, ob der Testfall ausgeführt werden soll oder nicht. In der dritten Spalte wird der Dateiname angezeigt. Der Dateiname wird um einen Stern (`*`) erweitert, falls der Testfall ein `inconclusive` Urteil enthält und um zwei Sterne (`**`) erweitert, falls der




```
state machine name : "Car Audio System";

/* modes */
regions :                RSTART, AudioPlayer, CDPlayer, TapeDeck,
                         RON, RTUNER, RTAPE, RCD;
simple composite states : On, TapeMode, CDMode, TunerMode;
orthogonal states :      CarAudioSystem;
simple states:           Off, CDEmpty, CDFull, TapeEmpty, TapeFull,
                         SW1, SW2, SW3, SW4,
                         TapePlaying,  BWSpooling, FWSpooling,
                         CDPlaying, NextTrack, PreviousTrack,
                         P1, P2, P3, P4;

/* hierarchy */
RSTART          > CarAudioSystem;
CarAudioSystem  > AudioPlayer, CDPlayer, TapeDeck;
AudioPlayer     > On, Off;
CDPlayer        > CDEmpty, CDFull;
TapeDeck        > TapeEmpty, TapeFull;
On              > RON;
RON             > SW1, SW2, SW3, SW4, TunerMode, TapeMode, CDMode;
TunerMode       > RTUNER;
RTUNER          > P1, P2, P3, P4;
TapeMode        > RTAPE;
RTAPE           > TapePlaying, BWSpooling, FWSpooling;
CDMode          > RCD;
RCD             > CDPlaying, NextTrack, PreviousTrack;

/* initial states */
initial states : CarAudioSystem, Off, TunerMode, CDEmpty, TapeEmpty,
                 CDPlaying, TapePlaying, P1;

/* events */
public events :   cdin, cdej, tpin, tpej, power, src, back, next, play;
private events :  cdf, cdfy, cdfn, tpf, tafy, tafn;
external events : env.SWTAPE, env.SWCD, env.TUNER, env.CD, env.TAPE,
                  env.P1, env.P2, env.P3, env.P4, env.tp, env.fws,
                  env.bws, env.cdfm, env.cdnx;

/* Transitions */
```

Abbildung 5.3: Spezifikationsansicht des `TCGD`.

Testfall ausschließlich `inconclusive` Urteile ausgehend vom Startknoten enthält. Durch einen Doppelklick auf den Namen wird der eigentliche Testfall zusätzlich in einem Fenster angezeigt. In der vierten Spalte werden während der Ausführung des Tests die Eingaben angezeigt, die an das System unter Test geschickt wurden. In der fünften Spalte werden die empfangenen Ausgaben des Systems unter Test angezeigt. Sollte eine Ausgabe von den Erwartungswerten abweichen und somit der Testfall fehlschlagen, so werden in der gleichen Spalte nach einem #-Zeichen alle möglichen, korrekten Ausgaben angezeigt. In der sechsten Spalte werden die Einzelergebnisse und in der siebenten Spalte farblich das Gesamtergebnis der Testausführung angezeigt. Dabei bedeutet eine grüne Markierung, dass die Testausführung erfolgreich, eine rote Markierung, dass die Testausführung nicht erfolgreich, eine gelbe Markierung, dass ein Testfall mit einem *inconclusive* Urteil bewertet wurde (die genaue farbliche Markierung hängt von der gewählten Bewertungsstrategie ab) und eine graue Markierung, dass der Testfall (noch) nicht ausgeführt wurde. Im unteren Teil wird zudem das Gesamtergebnis der Ausführen aller Testfälle in Zahlen angezeigt.

Auf der unteren linken Seite können zudem mit den Schaltflächen die Testfälle selektiert bzw. deselektiert werden. Die Schaltfläche `Select Failed` ermöglicht dabei die Selektion der

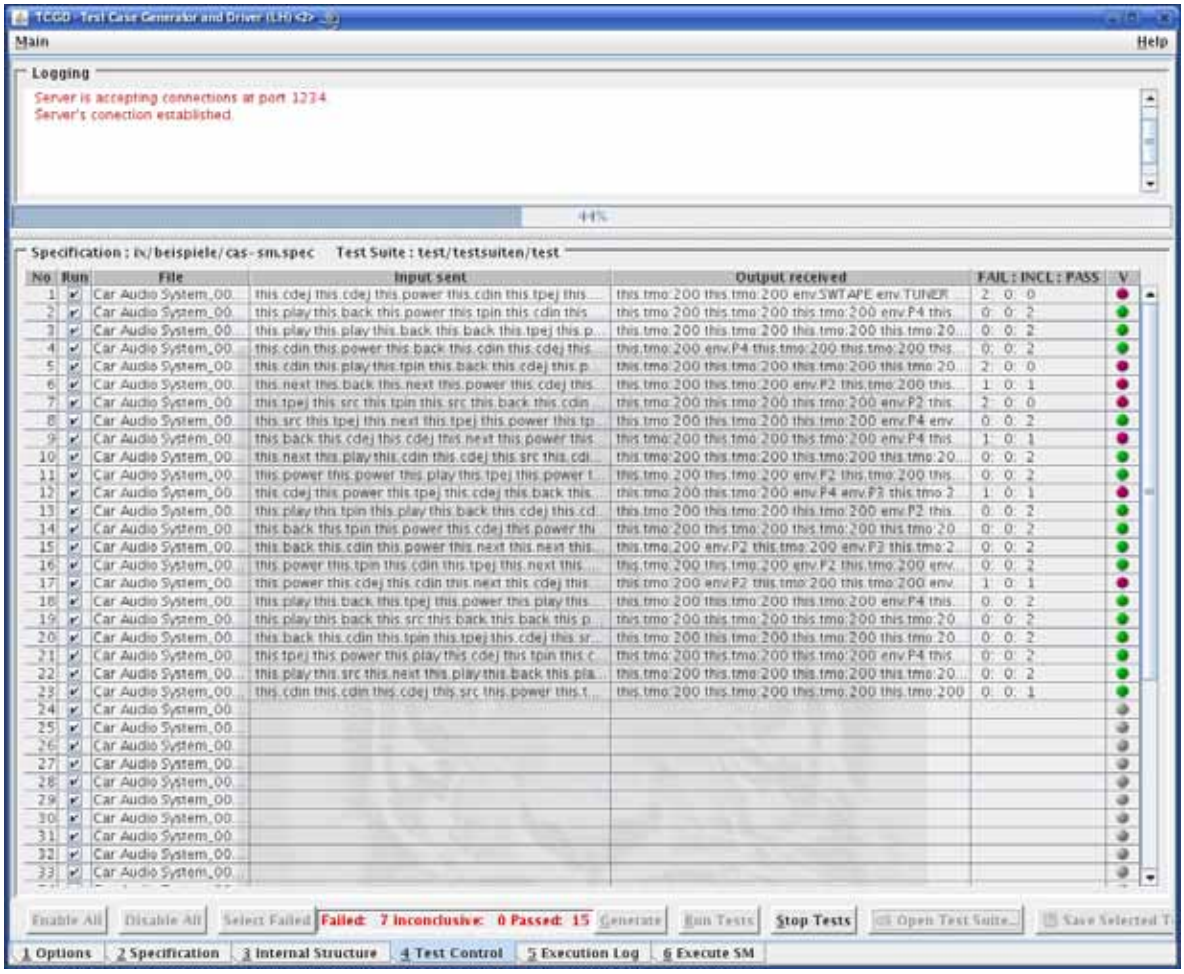


Abbildung 5.4: Testansicht des `TCGD`.

Testfälle, die bei der letzten Ausführung fehlgeschlagen sind. Somit können diese Testfälle auf einfache Art und Weise in einer separaten Testsuite (Verzeichnis) gespeichert und z. B. für Debuggingzwecke separat ausgeführt werden. Alternativ dazu können einzelne Testfälle auch mit den jeweiligen Checkboxen ausgewählt bzw. abgewählt werden.

Auf der unteren rechten Seite stehen schließlich drei Schaltflächen für die Erzeugung und Ausführung der Testfälle zur Verfügung. Mit der Schaltfläche `Generate` können auf Basis der im Editor geladenen Zustandsmaschine neue Testfälle erzeugt werden. Dafür wird zuvor abgefragt, ob alle Testfälle in der eingestellten Testsuite (Verzeichnis) gelöscht werden sollen. Wird dies positiv beantwortet, dann werden neue Testfälle erzeugt und die Tabelle mit den Testfällen entsprechend aktualisiert. Mit dem Knopf `Run` wird die Ausführung der Tests gestartet. Dafür wird zunächst der Adapter gestartet und auf eine Verbindung mit dem System unter Test am konfigurierten Port gewartet. Nachdem eine Verbindung hergestellt wurde, wird für jeden einzelnen Testfall das System unter Test zur Initialisierung aufgefordert und wenn dies mit einem `INIT_OK` beantwortet wurde, mit der Testausführung, entsprechend den eingestellten Parametern, begonnen. Ist die Ausführung eines Testfalls erfolgreich bzw. schlägt ein Testfall fehl, so wird ein neuer Testfall bzw. der selbe Testfall noch einmal gestartet. Mit Hilfe

der Schaltfläche `Stop` kann die Testausführung jederzeit abgebrochen werden.

In der zusätzlichen Ansicht `Execution Log` werden Statusinformationen angezeigt und die Kommunikation zwischen dem `Test Case Executor` und dem System unter Test aufgezeichnet. Diese werden für eine mögliche spätere Analyse in eine Datei geschrieben. Die letzte Ansicht `Execute SM` wird während des *On-the-fly* Tests auf Basis von Umgebungsmodellen und Anforderungen benutzt. Dieser Teil der Arbeit befindet sich derzeitig noch in der Entwicklung, so dass ich hier auf eine detaillierte Beschreibung der Umsetzung in der Werkzeugumgebung verzichte. Stattdessen sei auf Abschnitt **??** verwiesen.

5.5 Simulator für Zustandsmaschinen

Der `State Machine Simulator` wird zur Simulation einer Zustandsmaschine verwendet. Der Aufruf des Simulators erfolgt in gleicher Art und Weise wie beim `TCGD`:

```
java SME -base /home/seifert/work/progs/TEAGER -conf src/sme/sme.conf.
```

Analog zum `TCGD` können auch im `State Machine Simulator` die Einstellungen der unterschiedlichen Parameter in einer Konfigurationsdatei gespeichert und aus dieser wieder ausgelesen werden. Der `State Machine Simulator` setzt sich aus drei unterschiedlichen Ansichten zusammen. Abbildung 5.5 zeigt die Hauptansicht des `State Machine Simulator`s, die während der Simulation einer Zustandsmaschine benutzt wird. Im oberen Teil werden während der Simulation Status- und Logginginformationen über Ein- und Ausgaben der Zustandsmaschine ausgegeben. Im unteren Teil (`Execution Configuration`) kann der Simulator konfiguriert werden.

Mit dem Parameter `State Machine` wird die zu simulierende Zustandsmaschine festgelegt. Die Zustandsmaschine wird auch hier in den Editor geladen. Für diesen Editor und für die zweite Ansicht des `State Machine Simulator`s (`State Machine`) gelten die gleichen Anmerkungen wie für den Editor und die entsprechende Ansicht des `TCGD`s.

Mit dem Parameter `Test Server` wird der Testserver festgelegt. Der Wert besteht entweder aus dem Namen oder der IP-Adresse des Servers, gefolgt vom zu benutzenden Port. Drei mögliche Voreinstellungen können in der Konfigurationsdatei des `State Machine Simulator`s getroffen werden.

Mit dem Parameter `Firing Transition Set Selection` wird festgelegt, wie innerhalb des Simulators zwischen unterschiedlichen möglichen Mengen von aktivierten Transitionen gewählt werden soll. Dabei bedeutet die Einstellung `uniformly_distributed`, dass zwischen den möglichen Mengen eines Schritts gleichverteilt zufällig gewählt wird. Die Einstellung `deterministic_choice` bedeutet, dass bei der ersten Wahlmöglichkeit wieder zufällig gleichverteilt gewählt wird, aber in der Folge die gleiche Auswahl wiederholt getroffen wird. Somit kann ein deterministisches System simuliert werden, von dem jedoch initial die Wahl der Transitionsmenge nicht bekannt ist.

Mit dem Parameter `Transition Duration` wird festgelegt, wie lange die Ausführung einer Transition im Mittel dauern soll.

Mit dem Parameter `Deviation` wird wiederum die Varianz der Ausführungsdauer fest-

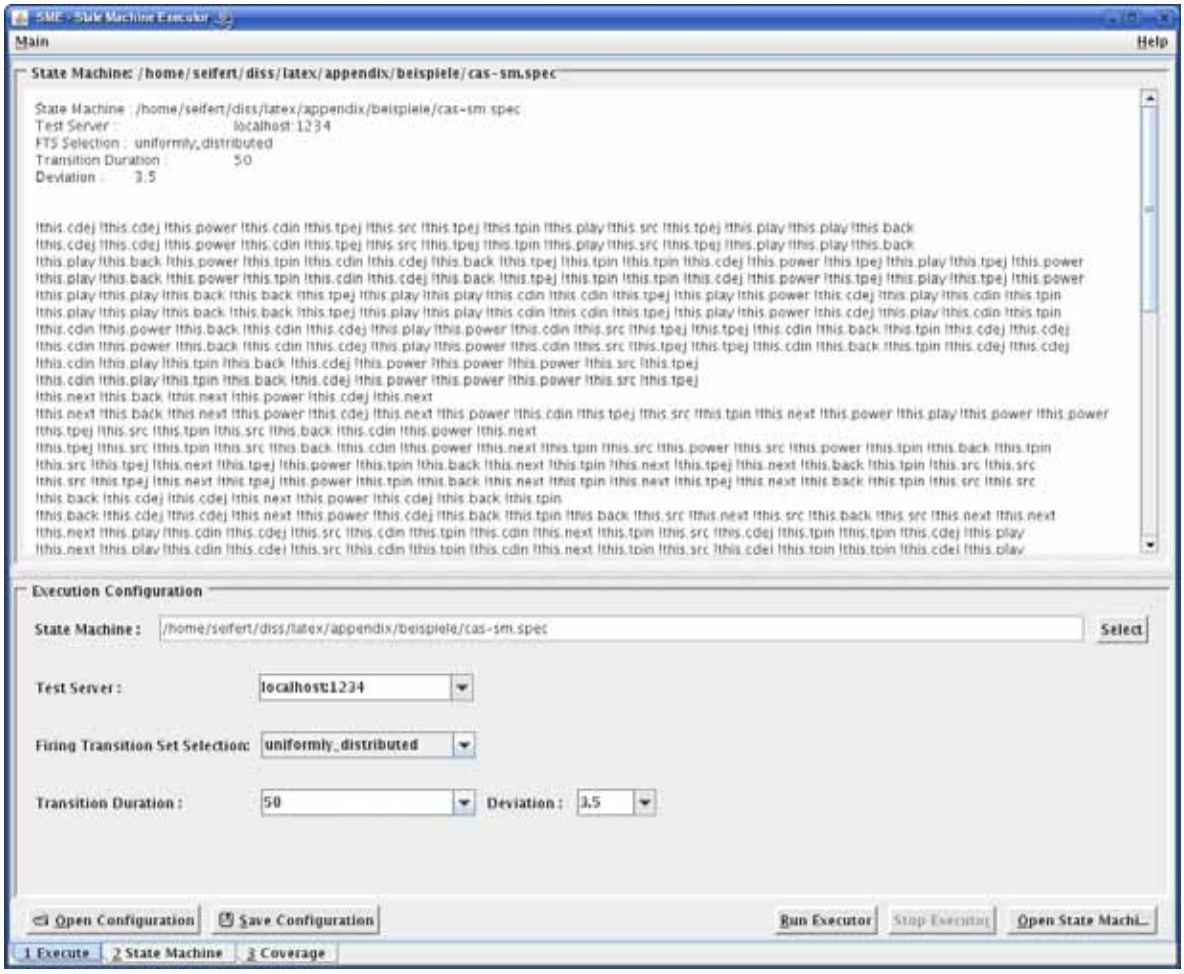


Abbildung 5.5: Ansicht des `State Machine Simulator`s.

gelegt. Hier gelten dieselben Aussagen wie die bezüglich de Triggergeschwindigkeit des `TCGD` getroffenen.

In der letzten Ansicht `Coverage` werden Informationen zur Überdeckung der Zustandsmaschine durch die Tests ausgegeben. Dieser Teil der Arbeit befindet sich ebenfalls in der Entwicklung, so dass ich auf eine detaillierte Beschreibung der Umsetzung in der Werkzeugumgebung verzichte. Für eine inhaltliche Diskussion sei auf Abschnitt **??** verwiesen.

5.6 Zusammenfassung

In diesem Kapitel habe ich die von mir implementierte prototypische Werkzeugumgebung TEAGER beschrieben, die die in Kapitel 3 und Kapitel 4 vorgestellten Konzepte umsetzt. Im folgenden Kapitel werde ich diesbezüglich einige Experimente und Erfahrungen, die ich auf Basis zweier Fallstudien (`Car Audio System` und `Sun Blind Control`) gemacht habe, vorstellen. Die dort gemachten Ergebnisse zeigen, dass mit der Werkzeugumgebung TEAGER die theoretischen Resultate der Kapitel 3 und 4 auch praktikabel umgesetzt werden konnten.

Neben der eigentlichen Umsetzung der Testfallerzeugung und -ausführung wurde zusätzlich ein Simulator für Zustandsmaschinen auf Basis der Semantik in Kapitel 3 implementiert. Mit diesem kann das Ausführungsverhalten einer Zustandsmaschine analysiert werden und so geprüft werden, ob ein System unter Test, das auf Basis dieser Zustandsmaschine implementiert wird, geeignet getestet werden kann. D. h. präzise ausgedrückt, ob die für einen adäquaten Test benötigten Schnittstellen und Informationen bereitgestellt werden. Viel wichtiger für diese Arbeit ist jedoch die Tatsache, dass mit Hilfe des Simulators der entwickelte Ansatz auf elegante Art und Weise getestet werden konnte. Der Simulator ermöglicht es, dieselbe Zustandsmaschine nicht nur als Spezifikation, sondern auch als System unter Test zu verwenden und somit die Testfallerzeugung und -ausführung auf einfache Art und Weise zu testen. Im Gegensatz zur vollständigen Simulation, die bei der Testfallerzeugung angewendet wird, zeigt der Simulator nur einen möglichen (wenn auch u. U. nichtdeterministischen) Ablauf der Zustandsmaschine. Zudem ermöglicht insbesondere das probabilistische Ausführungsverhalten des `Test Case Executor` und des `State Machine Simulator`, den vorgestellten Ansatz adäquat zu testen. Neben der Simulation der Spezifikation wurden JUnit-Testfälle zum Test der Werkzeugumgebung geschrieben. Im Wesentlichen umfassen diese Testfälle den Bereich der Semantik von Zustandsmaschinen und die Berechnung der Akzeptanzgrafen. JUnit-Testfälle wurden aber auch dazu benutzt, Die Testfallausführung und -bewertung automatisiert zu prüfen und bei Veränderungen Regressionstests durchzuführen. Dafür wurden sowohl die Testfallerzeugung und -ausführung, als auch der Simulator von ihren grafischen Benutzeroberflächen entkoppelt und die entsprechende Funktionalität in JUnit-Testfällen implementiert. Somit war eine automatische Ausführung von Tests für die Überprüfung der Testausführung möglich.

Neben der Möglichkeit zur Analyse einer Zustandsmaschine und der Bereitstellung eines Testbettes für den entwickelten Testansatz kann aber auch eine Testsituation dargestellt werden, in der Spezifikation und System unter Test verschieden sind. Es ist problemlos möglich, eine von der Spezifikation verschiedene Zustandsmaschine als System unter Test zu betrachten und so zu testen, ob diese eine gültige Realisierung im Sinne der verwendeten Implementierungsrelation ist. Dieser Aspekt wird insbesondere in zwei laufenden Diplomarbeiten [89, 78] genutzt (siehe Abschnitt **??**), in der die Überdeckung und Fehlerfindungsrate der Testfälle, d. h. die Qualität der erzeugten Testfälle, untersucht werden.

Zu guter Letzt ist es durch das Adapterkonzept und durch die Kommunikationsmöglichkeit über eine TCP/IP-Verbindung zwischen dem `Test Case Executor` und dem System unter Test auf einfache Art und Weise möglich, ein reales System unter Test zu testen. Die Schnittstelle, die durch das System unter Test implementiert werden muss, ist sehr klein und einfach gehalten, so dass keine wesentlichen Veränderungen an der Software vorgenommen werden müssen. Das zu testende System muss sich lediglich starten, stoppen und initialisieren lassen und eine Möglichkeit bieten, Ein- und Ausgaben zu versenden bzw. zu beobachten.

Kapitel 6

Fallstudien

Im folgenden Kapitel werde ich anhand von zwei Fallstudien die Werkzeugumgebung TEAGER evaluieren und anhand der Ergebnisse und Erfahrungen versuchen, Rückschlüsse auf die Eignung des hier vorgestellten Ansatzes für eine automatisierte Ableitung von Testfällen aus Zustandsmaschinen zu ziehen. Da die Werkzeugumgebung primär zur Evaluierung und Weiterentwicklung des Ansatzes dient, sollen die empirischen Untersuchungen Erfahrungswerte liefern, die Rückschlüsse auf eine allgemeine Anwendbarkeit des Testansatzes zulassen. Dazu ist es besonders interessant zu untersuchen, wie sich die Berechnungszeiten insbesondere für lange Eingabesequenzen verhalten und inwieweit die Kombination von Testfällen dem Problem der steigenden Komplexität entgegenwirken kann. Die Fragen nach der Überdeckung der unterschiedlichen Modelle durch eine Testsuite, sowie die Eignung zur Aufdeckung von Fehlverhalten werden in aktuellen Diplomarbeiten behandelt [89, 78]. Das erste Beispiel einer Musikanlage wurde bereits zur Beschreibung von Zustandsmaschinen verwendet. Das zweite, größere Beispiel einer Jalousiesteuerung wurde einer Lehrveranstaltung an der Universität Duisburg-Essen entnommen. Der Vorteil dieses Beispiels liegt darin, dass für das Beispiel ein vollständiger Entwurf und eine Implementierung in JAVA existieren. Beide wurden nicht von mir selbst entwickelt und stellen somit eine gute Grundlage für eine Evaluation dar. Speziell der Aspekt, dass es sich dabei um ein »reales Stück Software« handelt, brachte neue Erkenntnisse und Ideen für zukünftige Weiterentwicklungen.

In Abschnitt 6.1 werden zwei verschiedene Experimente mit der Musikanlage durchgeführt. Im ersten Experiment (siehe Abschnitt 6.1.1) wird gezeigt, wie sich die Berechnungszeiten in Abhängigkeit von der Burstlänge verhalten. Dabei wird in einer Variante eine zufällige Auswahl der Ereignisse getroffen und in einer zweiten Variante jeweils zwei feste Ereignissequenzen vorgegeben. Zusätzlich wird für die Testfälle der zweiten Variante die durchschnittliche Anzahl der Knoten des Akzeptanzgraphen aufgeführt. Zuletzt werden die Zeiten, die für die Ausführung der Testfälle gegen den Simulator für Zustandsmaschinen benötigt werden, dargestellt. Im zweiten Experiment (siehe Abschnitt 6.1.2) wird die Kombination von Testfällen untersucht. Dafür wird eine Eingabesequenz auf unterschiedlich lange Bursts aufgeteilt und dies den Zeiten für die Erzeugung und Ausführung der Testfälle gegenübergestellt.

In Abschnitt 6.2 werden beide Experimente mit der Fallstudie einer automatischen Jalousiesteuerung durchgeführt, um die Unabhängigkeit der zuvor ermittelten Ergebnisse vom spezifischen Beispiel zu prüfen. Neben diesen Experimenten zur Aufwandsanalyse werde ich auch einzelne Experimente mit Fehlerinjektion durchführen, um einen Eindruck von der Eignung zur Aufdeckung von Fehlern zu erhalten. Ausführliche Ergebnisse zu dieser Fragestellung sind aus den Diplomarbeiten zu erwarten. Zusätzlich werden die Testfälle gegen eine Beispielimplementierung in JAVA ausgeführt. Ich werde am Anfang des Abschnitts einen groben Überblick über die Fallstudie geben, um so einen Eindruck vom Umfang und Inhalt der gesamten Fallstudie zu vermitteln.

Alle Experimente wurden auf einem Pentium IV 2.6 GHz mit 512 MB RAM und dem Java-Laufzeitparameter `-Xmx256m` durchgeführt.

6.1 Musikanlage

Das Beispiel der Musikanlage wurde in dieser Arbeit bereits als durchgehendes Beispiel verwendet. Eine Beschreibung kann in Abschnitt 3.2 nachgelesen werden. Im Anhang B sind zudem die unterschiedlichen Eingabedateien, mit und ohne Verwendung von Daten, angegeben. Für die folgende Untersuchung wurde die Variante ohne Daten gewählt.

6.1.1 Berechnungsaufwand für unterschiedliche Burstlängen

Die erste Untersuchung, die ich mit der Fallstudie der Musikanlage durchgeführt habe, analysiert wie sich die Zeiten zur Ableitung von Testfällen mit steigender Länge des betrachteten Burst verhalten. Dabei wurde eine Versuchsreihe mit einer zufälligen Auswahl der Ereignisse und drei Versuchsreichen mit einer gegebenen, festen Sequenz von Eingaben durchgeführt. Jede Versuchsreihe wurde fünf mal wiederholt und von allen Zeiten der Mittelwert gebildet. Im Anschluss an die Ableitung der Testfälle wurden die Testfälle gegen den Simulator für Zustandsmaschinen mit der selben Zustandsmaschine ausgeführt und die entsprechenden Zeiten ermittelt. Die Experimente wurden mit den folgenden Parametereinstellungen durchgeführt, wobei die zwei unterschiedlichen Präambeln jeweils bei den entsprechenden Messungen P1 und P2 verwendet wurden:

Trigger Distribution:	`UNIFORM`
Step Bound:	`1000`
Preamble1:	`power back back next back back next cdin src next play next play cdej next next tpin src next back next back src back back back src power`
Preamble2:	`power tpin src back next back next back next next back next cdin src next play cdej next next tpej tpin src back next tpej next next next`
Execution Rounds:	2

Trigger Rate:	`25 ms`
Deviation:	`2.5`
Pass Condition:	`MUST`
Observation Sampling Rate:	`25 ms`
Timeout:	`1000 ms`
Test Server:	`localhost:1234`
Firing Transition Set Selection:	`uniformly_distributed`
Transition Duration:	`50 ms`
Deviation:	`5.0`

Die Ergebnisse der unterschiedlichen Messungen sind in Tabelle 6.1 wiedergegeben. In der ersten Spalte (**L**) ist die Länge des berechneten Bursts angegeben. In der zweiten Spalte (**U**) ist die Zeit angegeben, die für die Berechnung der Testfälle bei Auswahl der Eingaben mit gleichverteilter Wahrscheinlichkeit (Selektionsstrategie: `UNIFORM`) benötigt wurde. In den Spalten drei und vier (**P1** und **P2**) sind die Zeiten angegeben, die für die Berechnung der Testfälle bei gegebener Eingabesequenz (Parameter: `Preamble`) benötigt wurden. In der fünften Spalte ist die maximale `Knotenzahl` des Akzeptanzgraphen des Experiments **P2** angegeben. In der letzten Spalte (**A**) ist die Zeit angegeben, die für die Ausführung der erzeugten Testfälle des Experiments **P2** benötigt wurde.

L	U	P1	K(P1)	A(P1)	P2	K(P2)	A(P2)
5	0.962	0.700	5	77.555	1.853	11	83.529
8	2.400	1.546	7	83.610	12.623	23	94.224
10	6.205	4.264	13	96.994	35.984	36	100.151
13	21.342	17.184	20	105.234	130.305	57	106.078
15	59.454	37.056	52	111.273	306.594	148	117.780
18	161.624	137.369	152	119.800	940.592	226	124.425
20	276.278	448.499	257	125.766	1846.647	322	132.324
23	1051.945	1209.814	697	133.586	5312.930	1368	140.227
25	2164.834	2994.951	1041	139.969	10770.116	2818	145.561
28	7355.017	13498.36	3772	139.978	35997.721	6132	155.000

Tabelle 6.1: Messergebnisse der ersten Untersuchung.

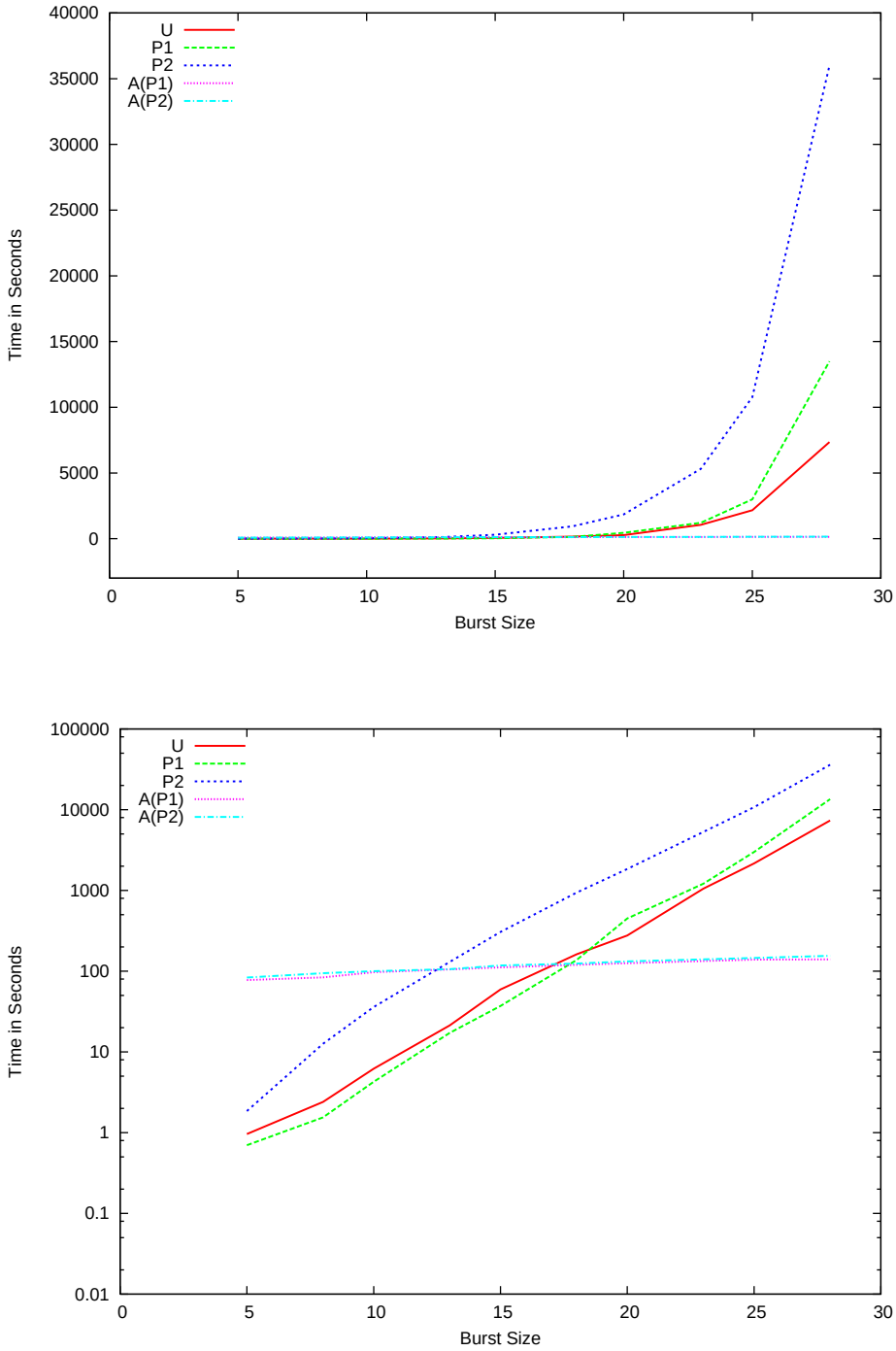


Abbildung 6.1: Messergebnisse der ersten Untersuchung.

Auswertung

Die Testfälle konnten auf dem oben angegebenen Rechner bis zu einer Länge von 28 berechnet werden. Bei einer Länge von 29 oder darüber reicht der zur Verfügung stehende Heapspeicher nicht mehr aus. Bei den Zeiten in der Spalte **U** muss beachtet werden, dass die Auswahl der Eingaben zufällig durchgeführt wurde. Die Musikanlage zeigt aber erst wirkliches Verhalten, wenn man sie als erstes einschaltet. So ist zu erklären, dass die Berechnungszeiten relativ stark von der berechneten Kurve abweichen. Um dieses Problem zu umgehen, habe ich in den folgenden Experimenten drei unterschiedliche, gegebene Eingabesequenzen verwendet. Die Auswertung der unterschiedlichen Berechnungszeiten zeigt deutlich, dass der Aufwand für die Berechnung mit zunehmender Länge des Burst exponentiell ansteigt (siehe Abbildung **??**). Dieses Ergebnis spiegelt die aus der Theorie erwarteten Ergebnisse wieder. Das Ergebnis könnte durch geeignete Optimierungen vermutlich noch verbessert werden, die gemessenen Zeiten zeigen aber, dass sich dadurch keine Lösung für den exponentiellen Aufwand zur Berechnung des Verhaltens ergeben würde. Weiterhin zeigt der Vergleich zwischen den Zeiten zur Ableitung der Testfälle und der Ausführungszeit, dass die Berechnung der Testfälle aufwändiger ist als die eigentliche Ausführung. Dies bestätigt die Herangehensweise, Testfälle zu speichern und so wiederholt ausführen zu können. Bei einem *On-the-fly* Ansatz wäre der exponentielle Faktor nicht so groß, jedoch müssten die Berechnungen für jeden neuerlichen Test wiederholt werden.

6.1.2 Berechnungsaufwand für unterschiedliche Kombinationsvarianten von Testfällen

In der zweiten Untersuchung werde ich die Auswirkung unterschiedlicher Burstlängen auf die Berechnung und Ausführungszeiten bei gleicher Gesamtanzahl an Eingaben untersuchen. Es soll gezeigt werden, welche Auswirkung die Begrenzung der vollständigen Verhaltensberechnung auf die unterschiedlichen Zeiten haben. Für die zweite Untersuchung wurden folgende Parametereinstellungen geändert:

Execution Rounds: 5

L	B	P1	A(P1)	P2)	A(P2)
24	24	1.014	3472.938	1.014	3456.648
24	12	x	y	1.977	1861.701
24	8	x	y	3.275	1332.252
24	6	x	y	6.209	1060.673
24	4	x	y	31.051	787.968
24	3	x	y	41.294	650.514
24	2	x	y	257.238	508.335
24	1	x	y	7426.259	357.494

Tabelle 6.2: Messergebnisse der zweiten Untersuchung.

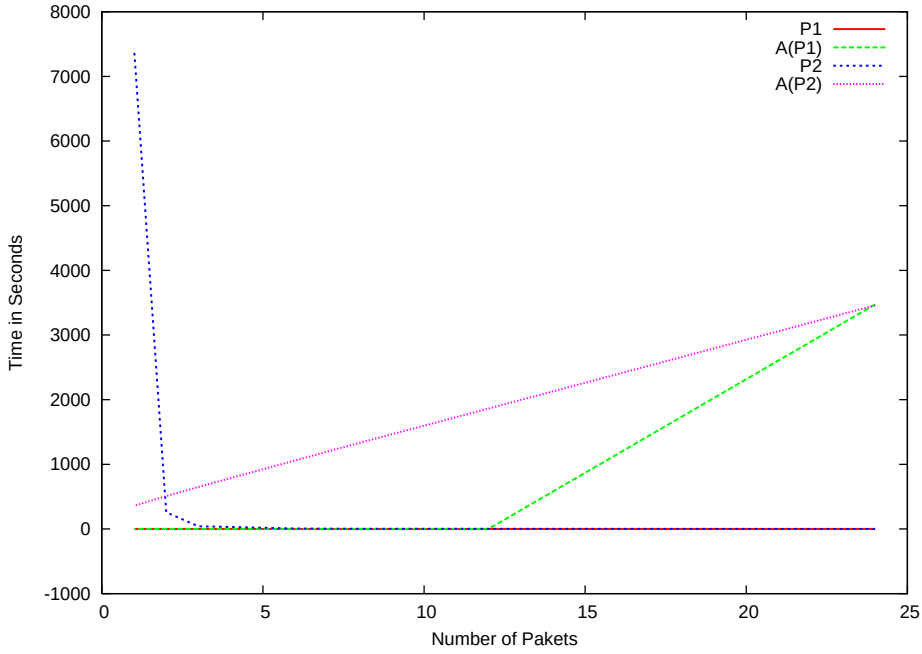


Abbildung 6.2: Messergebnisse der zweiten Untersuchung.

Auswertung

6.2 Jalousiesteuerung

Die Fallstudie `Sun Blind Control` ist eine automatische Jalousiesteuerung, wie sie zum Beispiel in modernen Bürogebäuden zum Einsatz kommen könnte und wird vom Fachgebiet *Software Engineering* der Universität Duisburg-Essen in der Lehrveranstaltung *Embedded Systems* eingesetzt [39]. Anhand der Fallstudie wird der dort entwickelte Entwicklungsprozess *DePES* für eingebettete Systeme dargestellt. Während der Lehrveranstaltung werden alle Phasen von der Anforderungsanalyse bis hin zur Implementierung und dem Test durchlaufen.

Ich habe die Fallstudie speziell benutzt, um an einem größeren und realistischen Beispiel den entwickelten Ansatz zu erproben. Gleichzeitig sollte auch untersucht werden, ob aus praktischen Gesichtspunkten heraus Veränderungen oder Erweiterungen an dem Ansatz vorgenommen werden müssen. Im Unterschied zum `Car Audio System` gibt es zu dieser Fallstudie eine Referenzimplementierug, gegen die im Folgenden auch getestet wurde.

6.2.1 Beschreibung der Fallstudie

Im Zuge von Umbauarbeiten sollen in einem Gebäude Fenster mit automatisch gesteuerten Jalousien eingebaut werden. Eine Jalousie besteht dabei aus einer Vielzahl von Lamellen, die, um die Intensität der Sonneneinstrahlung zu regulieren, in einem Winkel von ± 80 Grad verstellt werden können. Gleichzeitig soll die Jalousie in Abhängigkeit von Sonneneinstrahlung und Windstärke gesteuert werden. Scheint die Sonne, so soll die Jalousie herunter gefahren werden. Im Gegensatz dazu soll sie bei zu starkem Wind nach oben gefahren werden. Für einen maximalen Komfort soll eine manuelle Steuerung der Jalousie möglich sein.

Anforderungen

In der Analysephase werden zunächst die Anforderungen ermittelt. Anforderungen sind optative Aussagen, die beschreiben, wie sich die Umgebung verhalten soll, wenn die zu entwickelnde Maschine integriert wurde.

R1 Scheint die Sonne für mehr als eine Minute und ist kein starker Wind, dann wird die Jalousie nach unten gefahren.

R2 Scheint für mehr als fünf Minuten keine Sonne, wird die Jalousie nach oben gefahren.

R3 Die Jalousie soll nicht zerstört werden: Tritt starker Wind auf, wird die Jalousie nach oben gefahren.

R4 Falls kein starker Wind ist, die Lamellen nicht blockiert sind und der Benutzer die Lamellen in positiver Drehrichtung justiert, dann verstellen sich die Lamellen in positiver Drehrichtung. Falls kein starker Wind ist, die Lamellen nicht blockiert sind und der Benutzer die Lamellen in negativer Drehrichtung justiert, dann verstellen sich die Lamellen in negativer Drehrichtung.

R5 Wenn der Benutzer die Jalousie manuell öffnet, dann fährt die Jalousie nach oben.

R6 Wenn der Benutzer die Jalousie manuell schließt und kein starker Wind ist, dann fährt die Jalousie nach unten.

R7 Die Jalousie soll nicht zerstört werden: Wenn die Jalousie in ihrer untersten Position bzw. in ihrer obersten Position ist, dann stoppt die Jalousie.

R8 Wurde die Jalousie vom Benutzer bedient, so soll die automatisierte Steuerung innerhalb der nächsten vier Stunden auf den Wechsel von Sonneneinstrahlung nicht reagieren.

Domänenwissen

Als Domänenwissen werden indikative Aussagen über die Umgebung angesehen. Solche Aussagen sind gültig, unabhängig davon, wie die zu entwickelnde Maschine gebaut wird. Bezüglich des Domänenwissen werden Fakten und Annahmen unterschieden. Letztere beschreiben Fakten, die nicht immer garantiert werden können, die aber angenommen werden müssen, damit die zu entwickelnde Maschine gebaut werden kann.

F1 Ein Sonnensensor misst die Sonnenintensität und liefert die Intensität der Sonnenstrahlung als einen lux-Wert.

F2 Die Intensität der Sonnenstrahlung eines durchschnittlichen Sonnentages beträgt zwischen 32000 und 100000 lux.

F3 Die Rechtsdrehung des Motors bewirkt, dass die Jalousie nach unten fährt.

F4 Die Linksdrehung des Motors bewirkt, dass die Jalousie nach oben fährt.

F5 Die Lamellen können zwischen -80 Grad und 80 Grad eingestellt werden. Werden sie darüber hinaus verändert, dann tritt ein Phänomen auf, das signalisiert, dass die Lamellen blockiert sind.

F6 Starker Wind hat eine Geschwindigkeit von mehr als 80 km/h. Die Windgeschwindigkeit wird mit einem Windsensor gemessen.

F7 Die Jalousie wird durch starken Wind zerstört.

F8 Die Jalousie wird zerstört, wenn sie nicht innerhalb von 0,2 Sekunden, nachdem sie vollständig nach oben bzw. nach unten gefahren wurde, angehalten wird.

F9 Die Jalousie wird innerhalb von 20 Sekunden nach oben bzw. nach unten gefahren.

F10 Wenn die Jalousie vollständig nach oben bzw. nach unten gefahren wurde, dann blockiert der Motor. Ein blockierter Motor ist nach 0,2 Sekunden zerstört.

F11 Wenn der Motor gestoppt wird, dann stoppt auch die Jalousie.

F12 Die Lamellen besitzen ein Interface, so dass sie direkt an einen Mikrocontroller angeschlossen werden können.

F13 Der Motor kann seine Drehrichtung unmittelbar ändern.

A1 Der Benutzer drückt einen Hoch-Knopf drei Sekunden lang, wenn die Jalousie nach oben gefahren werden soll.

A2 Der Benutzer drückt einen Runter-Knopf drei Sekunden lang, wenn die Jalousie nach unten gefahren werden soll.

A3 Der Benutzer drückt den Hoch- bzw. den Runter-Knopf weniger als drei Sekunden lang, um die Stellung der Lamellen zu verändern.

A4 Starker Wind wirkt erst nach 30 Sekunden zerstörerisch.

Spezifikation

Die Spezifikation bilden die implementierbaren Anforderungen. Sie dienen als Grundlage für die Entwicklung der Modelle der zu entwickelnden Maschine und damit auch für die Ableitung der Testfälle.

S1 = R1

S2 = R2

S3 = R3

S4a Falls kein starker Wind ist, die Lamellen nicht blockiert sind und der Benutzer die Lamellen in positiver Drehrichtung justiert, dann werden die Lamellen in positiver Drehrichtung verstellt.

S4b Falls kein starker Wind ist, die Lamellen nicht blockiert sind und der Benutzer die Lamellen in negativer Drehrichtung justiert, dann werden die Lamellen in negativer Drehrichtung verstellt.

S5 = R5

S6 = R6

S7 = R7

S8 Die Jalousie wird für die nächsten vier Stunden nicht nach oben bzw. nach unten gefahren, wenn der Benutzer manuelle Einstellungen vorgenommen hat.

Im weiteren Entwicklungsprozess wird u. a. die Systemarchitektur und die Softwarearchitektur entworfen. Beide Modelle sind in Abbildung C.2 auf Seite 131 dargestellt. Für die in Software realisierte Controllerkomponente wurde zur Verhaltensbeschreibung eine Zustandsmaschine verwendet. Diese Modelle sind in Abbildung C.3 auf Seite 132 dargestellt. Diese Zustandsmaschine habe ich als Ausgangsbasis für die Ableitung von Testfällen benutzt. Die in TEAGER verwendete textuelle Repräsentation ist im Programmausdruck C.1 auf Seite 133 abgedruckt.

6.2.2 Erfahrungen aus der Praktischen Anwendung

Im folgenden möchte ich kurz über die Erfahrungen, die ich mit der Anwendung des externen Beispiels der Jalousiesteuerung gemacht habe, berichten. Besondere Herausforderung war dabei einerseits, die Software für die Testausführung in die Testumgebung einzubinden und andererseits, dass diese Spezifikation Zeitschranken enthält, die von meinem Ansatz bisher nicht berücksichtigt werden konnten.

Gegeben war eine JAVA-Implementierung, die die in Abbildung C.1 im Anhang C auf Seite 129 abgebildete Softwarearchitektur umsetzt. Die dort zusätzlich implementierte Hardwaresimulation habe ich durch den Adapter der Werkzeugumgebung TEAGER ersetzt. D. h., dass die gegebene Implementierung so erweitert wurde, dass die geforderte Schnittstelle

für den Test implementiert wurde. Die Schnittstelle umfasst dabei die Funktionen `start`, `stop`, `initialize`, `isInitialized` und `receive`. Die beiden interessanten Funktionen sind `initialize` und `receive`. Zusätzlich muss natürlich noch die Weiterleitung der Ausgaben realisiert werden. Letzteres war in der implementierten Hardwaresimulation relativ einfach zu ändern. An den Stellen, wo die Ausgaben vom Controller für die Simulation empfangen wurden, musste lediglich die Ausgabe an den Adapter und damit an den Testserver weitergeleitet werden. Die entgegengesetzte Richtung war ähnlich zu realisieren. Die Simulation der Hardware, d. h. der Benutzerknöpfe, des Windsensors und des Sonnensensors, wurden so angepasst, dass die vom Adapter empfangenen Eingaben an den Controller entsprechend weitergeleitet werden. Die größte Schwierigkeit war, in den bestehenden Java-Code die Initialisierungsanweisungen einzubauen. Hier musste die vollständige Objekterzeugungsreihenfolge analysiert werden und zusätzlich alle Initialisierungen von Variablen identifiziert werden. Um Objekterzeugung und (Re-)Initialisierung geeignet miteinander zu verbinden, habe ich in jeder Klasse eine Methode zum setzen der Variablenwerte eingefügt. Diese wird nun sowohl vom Konstruktor der Klasse, als auch von der zusätzlich hinzugefügten Initialisierungsmethode benutzt. Alternativ hätte man gerade in einer Softwaresimulation das gesamte System neu erzeugen können, was mir jedoch nicht als eine geeignete Strategie erschien. Das Beispiel der Initialisierung zeigt deutlich, dass Aspekte des Design for Testability währen des Entwurfs und der Implementierung von Softwaresystemen beachtet werden müssen. An dieser Stelle sei darauf hingewiesen, dass die (Re)-Initialisierung insbesondere beim *Hardware in the Loop* Test extrem aufwändig sein kann. Das Beispiel eines elektrisch verstellbaren Sitz verdeutlicht dies recht deutlich. Hier müsste der Sitz wieder in seine Ausgangsposition gefahren werden. Dies ist vielleicht noch nicht extrem teurer, aber zumindest Zeitaufwändig, was den Testprozess zumindest verzögert. Aus diesen Grund versucht man i. A. die Testsequenzen eines Testfalls so lang wie möglich (aber natürlich nur so lang wie nötig) zu machen. Dies ist einer der Gründe, warum ich einen Teil meiner Arbeit für die geeignete Kombination von Testfällen verwendet habe.

Der zweite interessante Punkt ist durch die geeignete Behandlung von Timeouts aufgeworfen worden. Ich verwende eine Testhypothese die besagt, dass alle Ausgaben bzw. das Ausbleiben einer Ausgabe bis zu einer oberen Schranke erfolgen müssen. Würde man dies 1:1 auf die Jalousiesteuerung anwenden, dann würde diese obere Schranke bei mindestens vier Stunden und einer Sekunde liegen, da Aufgrund von Anforderung *R8* das Ausbleiben von Ausgaben innerhalb dieser Zeitspanne beobachtet werden müsste. Dies ist für die Ausführung von vielen und langen Testfällen ungünstig. Zur Lösung des Problems habe ich mir überlegt, dass man für jedes Ereignis spezifizieren können möchte, innerhalb welchen Intervalls dieses Ereignis auftreten soll. Damit ist sowohl eine untere, als auch eine obere Schranke gemeint. Ich habe einen Testfall so erweitert, dass dies entsprechend spezifiziert werden kann. Erwartet ein Testfall eine Beobachtung o, so kann diese folgendermaßen manuell erweitert werden: $o : X : Y$. Dabei bezeichnet X die untere und Y die obere Grenze des Intervalls, in dem die Ausgabe o beobachtet werden soll. Realisiert wird dies durch den geeigneten Start von unterschiedlichen Timern während der Testausführung. Dabei können verschiedene Ausgaben in unterschiedlichen Intervallen miteinander kombiniert werden.

Die explizite Zuweisung einer unteren und oberen Zeitschranke zu einzelnen Ereignissen löst nicht nur das Problem der Behandlung des Timeouts. Es wird gleichzeitig ein konzeptioneller Rahmen geschaffen, mit dem die Einhaltung von Zeitschranken sinnvoll geprüft werden kann. Unter der Voraussetzung, dass der Testumgebung präzise Timer zur Verfügung gestellt werden,

könnten damit sogar Systeme, die Echtzeitbedingungen einhalten müssen, adäquat getestet werden.

Im Folgenden werde ich auch auf die Fallstudie der Jalousiesteuerung die ersten beiden Untersuchungen anwenden, um zu zeigen, dass die Ergebnisse unabhängig vom ersten Beispiel sind. Im Anschluss daran, werde ich einige Untersuchungen durchführen, in denen ich in die Implementierung Fehler einstreue und analysiere, ob diese Fehler durch die unterschiedlichen Testsuiten gefunden werden.

6.2.3 Berechnungsaufwand für unterschiedliche Burstlängen

Für die dritte Untersuchung wurden folgende Parametereinstellungen verändert:

Execution Rounds: 2

L	U	P1	K(P1)	A(P1)	P2	K(P2)	A(P2)
5	x	x	x	x	x	x	x
8	x	x	x	x	x	x	x
10	x	x	x	x	x	x	x
13	x	x	x	x	x	x	x
15	x	x	x	x	x	x	x
18	x	x	x	x	x	x	x
20	x	x	x	x	x	x	x
23	x	x	x	x	x	x	x
25	x	x	x	x	x	x	x
28	x	x	x	x	x	x	x

Tabelle 6.3: Messergebnisse der dritten Untersuchung.

Auswertung

6.2.4 Berechnungsaufwand für unterschiedliche Kombinationsvarianten von Testfällen

In der vierten Untersuchung werden wiederum die zeitlichen Auswirkungen bei der Kombination von Testfallen bestimmt. Für die vierte Untersuchung wurden folgende Parametereinstellungen verändert:

Execution Rounds: 5

L	B	P1	A(P1)	P2)	A(P2)
24	24	x	y	x	y
24	12	x	y	x	y
24	8	x	y	x	y
24	6	x	y	x	y
24	4	x	y	x	y
24	3	x	y	x	y
24	2	x	y	x	y
24	1	x	y	x	y

Tabelle 6.4: Messergebnisse der vierten Untersuchung.

Auswertung

6.2.5 Fehlerinjektionen

Für die fünfte Untersuchung wurden folgende Parametereinstellungen verändert:

Execution Rounds: 5

L	K	P	A	P	A
24	1	x	y	x	y
24	2	x	y	x	y
24	3	x	y	x	y
24	4	x	y	x	y
24	6	x	y	x	y
24	8	x	y	x	y
24	12	x	y	x	y
24	24	x	y	x	y

Tabelle 6.5: Messergebnisse der fünften Untersuchung.

Auswertung

6.3 Zusammenfassung

Anhang A

Semantischer Schritt

Zum besseren Verständnis der Definition eines semantischen Schritts (siehe Definition 25, Seite 47) aus Kapitel 3 ist im Folgenden eine beispielhafte Implementierung in der funktionalen Sprache ML angegeben. Anhand eines einfachen Beispiel wird gezeigt, wie die Schrittfunktion auf eine konkrete Zustandsmaschine angewendet werden kann. Neben der Veranschaulichung dient die prototypische Umsetzung auch der Bestätigung der korrekten Typisierung der angegebenen Definitionen eines semantischen Schritts.

Um das Beispiel möglichst anschaulich zu halten, habe ich den implementierten semantischen Schritt dahingehend vereinfacht, dass der Ereignisspeicher und die aktuelle Konfiguration der Zustandsmaschine nicht berücksichtigt werden. Das auslösende Ereignis der Transition wird nicht aus dem aktuellen Ereignisspeicher entnommen, sondern als freier Parameter verwendet. Es wird folglich im angegebenen *partiellen* semantischen Schritt nicht geprüft, ob die Transition aktiviert ist, so wie es in Definition 17 auf Seite 42 definiert wurde. Fehlende Funktionalität können jedoch leicht hinzugefügt werden.

Im Programmausdruck A.1 sind die allgemeinen Definitionen für einen semantischen Schritt angegeben. Abbildung A.1 zeigt eine einfache Zustandsmaschine, die aus zwei Zuständen und einer Transition zwischen diesen Zuständen besteht. Die Umsetzung der Zustandsmaschine aus Abbildung A.1 in ML ist in Programmausdruck A.2 angegeben, an dessen Ende die Transition beispielhaft ausgeführt wird.

A.1 Implementierung

In Zustandsmaschinen sind der Datenraum, d. h. die Bestandteile, aus denen sich der Datenraum zusammensetzt und die Menge der Ereignisse, variabel. Aus diesem Grund lasse ich diese beiden Komponenten undefiniert und fordere später, dass eine konkrete Zustandsmaschine die Signatur `SM_STRUCTURE` implementiert.

Die abstrakte Zustandsmaschine habe ich als Funktor umgesetzt, dessen Signatur in Programmausdruck A.1 (`signature STATEMACHINE`) angegeben ist. Basierend auf den variablen Komponenten `D` und `E` sind dort auch die noch fehlenden Komponenten `G`, `A` und `T` einer Zustandsmaschine definiert. Die Funktion `partialStep` ermöglicht eine Transition auszuführen und so den Effekt der Transition zu bestimmen.

Eine Transition ausführen heisst, dass die Aktionen der Transition unter Berücksichtigung der auslösenden Ereignisinstanz und der aktuellen Datenraumbelegung, d. h. unter Berücksichtigung der Wertbelegungen, in einer Sequenz ausgeführt werden. Resultat der Ausführung ist die aktualisierte Datenraumbelegung und eine Sequenz von Ereignisinstanzen, die während der Ausführung neu erzeugt wurden.

Die Funktion `performAction` realisiert die Ausführung einer einzelnen Aktion. Dabei werden zwei Fälle unterschieden. Besteht die Aktion aus einer Funktion, die eine neue Ereignisinstanz erzeugt (`event`), so wird diese auf die aktuelle Datenraumbelegung angewendet und die so erzeugte Ereignisinstanz zur Sequenz der erzeugten Ereignisinstanzen hinzugefügt. Besteht die Aktion aus einer Funktion, die einen Datenraum aktualisiert (`update`), dann wird die Funktion auf die aktuelle Datenraumbelegung angewendet und so die neue (aktuelle) Datenraumbelegung bestimmt. Beachtet werden sollte, dass die Aktionen einer Transition sich sowohl auf die aktuelle Datenraumbelegung, als auch auf die Datenwerte der auslösenden Ereignisinstanz beziehen können. Der vollständige Effekt einer Transition wird durch die sequentielle Komposition der einzelnen Aktionen (Funktionen) bestimmt. In der Funktion `performAllActions` wird zunächst die Funktion `performAction` auf alle Aktionen angewendet und danach werden die Funktionen miteinander kombiniert (Funktionskomposition). Das Resultat wird auf die aktuelle Datenraumbelegung angewendet. Als Ergebnis erhält man die aktualisierte Datenraumbelegung und alle während des Schritts erzeugten Ereignisinstanzen.

Die bedingte Ausführung einer Transition ist durch die Funktion `partialStep` gegeben. In der Funktion `partialStep` wird die Übergangsbedingung ausgewertet und die Funktion performAllActions nur ausgeführt, falls die Bedingung erfüllt ist.

```
signature SM_STRUCTURE =
  sig
  type D; (* type of data space *)
  type E; (* type of events     *)
end;

signature STATEMACHINE =
  sig
  type D; (* type of data space *)
  type E; (* type of events     *)

  (* type of guards: predicate over an event (from transition) and *)
  (* a data space assignment                                       *)
  datatype G = guard of D -> bool;

  (* type of actions: function that creates an event or updates a data *)
  (* space wrt an event (from transition) and a data space assignment  *)
  datatype A = event of D -> E | update of D -> D;
```

```
20    (* type of transitions (incomplete) *)
      datatype T = transition of E -> G * A list;
22
      (* example step implementation                              *)
24    (* not considering the configuration and the event pool *)
      val partialStep : T -> E * D -> D * E list;
26  end;

28  functor SM(structure sm_struc : SM_STRUCTURE) : STATEMACHINE =
      struct
30    type D = sm_struc.D;
      type E = sm_struc.E;
32
      datatype G = guard of D -> bool;
34    datatype A = event of D -> E | update of D -> D;
      datatype T = transition of E -> G * A list;
36
      (* perform one single action *)
38    (* input:  data space assignment and list of (generated) events     *)
      (* yields: new data space assignment and new event list             *)
40    (* perform single action: append a new generated event to the event *)
      (* list or update the data space; function f holds the action       *)
42    fun performAction(event(f)) (d,es) = (d, es @ [f(d)])
        | performAction(update(f))(d,es) = (f(d), es)
44
      (* fold left: function f will be the composition function o and the *)
46    (* neutral element will be the id function                          *)
      (* fold(o,[f, g, h])(n) = h o (g o ( f o n ))                       *)
48    fun fold(f,n)([]) = n
        | fold(f,n)(x::xs) = fold(f,f(x,n))(xs);
50
      (* sequential composition of all actions                        *)
52    (* perform all actions: apply perfomAction to every action and *)
      (* then (sequentially) compose all actions                     *)
54    fun performAllActions(actions)(d) =
            fold(o, fn x => x)(map(performAction)(actions))(d);
56
      (* combination of guard checking and performing all actions          *)
58    (* yields a new data space assignment and a list of all generated    *)
      (* events if the guard is satisfied or the old data space assignment *)
60    (* and an empty list otherwise *)
      fun partialStep(transition(t))(e,d) =
62      let
          val (guard(g), actions) = t(e);
64      in
          if g(d) then
66          performAllActions(actions)(d,[])
        else (d, [])
68      end;
  end;
```

Programmausdruck A.1: Beispielimplementierung der Definition eines semantischen Schritts

A.2 Beispiel

Eine konkrete Zustandsmaschine definiert zunächst, wie sich der Datenraum zusammensetzt und welche Ereignisse verarbeitet werden können. Im Beispiel aus Abbildung A.1 besteht der Datenraum aus zwei Komponenten. Einem ganzzahligen Wert, den ich mit `p1` referenziere, und einem booleschen Wert, den ich mit `p2` referenziere.

Die Transition vom Zustand `A` zum Zustand `B` wird durch eine Ereignisinstanz mit dem Namen `a` ausgelöst. Ereignisinstanzen zum Ereignisnamen `a` sind strukturiert und setzen sich aus einem ganzzahligen und einem booleschen Wert zusammen. Die konkreten Wertbelegungen referenziere ich mit `x` und `y`.

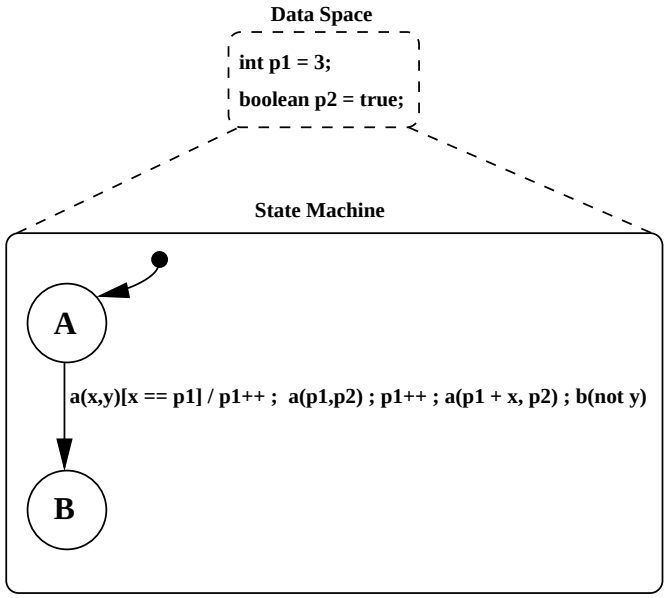


Abbildung A.1: Zustandsmaschine zum Programmausdruck A.2

Ist die Übergangsbedingung `x == p1` erfüllt, dann wird der Effekt der Transition ausgeführt. Im Beispiel besteht der Effekt der Transition aus fünf Aktionen. In der ersten Aktion wird `p1` erhöht. In der zweiten Aktion wird eine neue Ereignisinstanz mit dem Namen `a` erzeugt, welche die aktuellen Wertbelegungen des Datenraums (`p1`, `p2`) umfasst. In der dritten Aktion wird wieder `p1` inkrementiert und anschließend, in der vierten Aktion, eine neue Ereignisinstanz mit dem Namen `a` erzeugt. Hier wird jedoch `x` zu `p1` addiert. Der boolesche Wert der Ereignisinstanz ist mit dem Wert `p2` der aktuellen Datenraumbelegung identisch. Abschließend wird in der fünften Aktion eine neue Ereignisinstanz mit dem Namen `b` erzeugt, welche den negierten booleschen Wert des Datenraums enthält. Somit wird während der Ausführung der Transition der Datenraum zweimal aktualisiert und es werden drei neue Ereignisinstanzen erzeugt.

In Listing A.2 werden der Datenraum und die beiden Ereignisse `a` und `b` in der Struktur `sm1_struc` definiert. Diese wird als Parameter an den Funktor `SM` übergeben, um so die konkrete Zustandsmaschine `sm1` zu erhalten. Danach wird mit Hilfe der Zustandsmaschine `sm1` und der Struktur `sm1_struc` die Transition definiert und ausgeführt.

$$\mathbf{partialStep(t1)(a(3,true),(3,true))} \Rightarrow \mathbf{((5,true),[a(4,true),a(8,true),b(false)])}$$

```
4  use "statemachine.ml";
6  (* example structure of a state machine *)
   structure sm1_struc =
8    struct
10   (* the data space is constituted of an int value and a boolean value; *)
     (* for example: dataspace(11, true)
*)
12   datatype D = dataspace of int * bool;
14   (* event a carries an int and a boolean value and event b carries *)
     (*  a boolean value; for example: a(5,false) and b(true)          *)
16   datatype E = a of int * bool | b of bool;
   end;
18
   (* instantiate state machine 1 *)
20 structure sm1 = SM(structure sm_struc = sm1_struc);
22 (* transition t1: *)
   (* a(x,y)[x == p1] / p1++; a(p1, p2); p1++; a(p1+x, p2); b(not y) *)
24 val t1 = sm1.transition(fn sm1_struc.a(x,y) =>
     (* guard *)
26   (sm1.guard(fn sm1_struc.dataspace(p1,p2) => x = p1),
28    (* effect for (dataspace(p1, p2) and a(x,y)) *)
      [
30
       sm1.update(fn sm1_struc.dataspace(p1, p2) =>
32       sm1_struc.dataspace(p1+1, p2)),                          (* p1++ *)
34     sm1.event (fn sm1_struc.dataspace(p1,p2)  =>
         sm1_struc.a(p1, p2)),                                 (* a(p1, p2) *)
36
       sm1.update(fn sm1_struc.dataspace(p1, p2) =>
38       sm1_struc.dataspace(p1+1, p2)),                          (* p1++ *)
40     sm1.event (fn sm1_struc.dataspace(p1,p2)  =>
         sm1_struc.a(p1 + x, p2)),                          (* a(p1 + x, p2) *)
42
       sm1.event (fn sm1_struc.dataspace(p1,p2)  =>
44       sm1_struc.b(not y))                                    (* b(not y) *)
      ]
46   ));
48 sm1.partialStep(t1)(sm1_struc.a(3, true), sm1_struc.dataspace(3, true));
50 (* partialStep applied to t1, a(3,true) and (3,true)                  *)
   (* yields: (dataspace (5, true), [a (4, true), a (8, true), b false]) *)
```

Programmausdruck A.2: Beispiel für die Ausführung eines semantischen Schritts

Anhang B

Grammatiken

Im Folgenden stelle ich vereinfachte Grammatiken für Zustandsmaschinen und Testfälle dar. Grammatiken dieser Art, angereichert und angepasst, verwende ich innerhalb der Werkzeugumgebung TEAGER, um aus ihnen mit Hilfe des Parser-Generators ANTLR [73] einen Lexer und einen Parser automatisch zu erzeugen.

Im Abschnitt B.1 zeigt der Programmausdruck B.1 die Grammatik für Zustandsmaschinen mit Daten. Der folgende Programmausdruck B.2 zeigt ein Beispiel für eine Zustandsmaschine, in der keine Daten verwendet werden. Programmausdruck B.3 zeigt dann das gleiche Beispiel unter Verwendung von Daten. Hier wurde speziell die Möglichkeit genutzt, mit Hilfe von Datenvariablen den Zustand des Systems zu speichern und diesen über eine Methode abzufragen. Dadurch kann insbesondere auf die *Switch*-Zustände verzichtet werden. Zuletzt zeigt Programmausdruck B.4 die Java-Klasse, die aus den Datenanteilen der Zustandsmaschine generiert wird.

In Abschnitt B.2 zeigt Programmausdruck B.5 die Grammatik für Testfälle, so wie sie für die Stapelverarbeitung gespeichert werden. Programmausdruck B.2.2 zeigt einen beispielhaften Testfall.

B.1 Zustandsmaschinen

Die Spezifikation einer Zustandsmaschine beginnt mit der Angabe des Namens für die Zustandsmaschine. Der Name wird als Präfix für den Dateinamen der generierten Testfälle benutzt. Daher sollte er möglichst konform zur Namensgebung für Dateien auf dem verwendeten Betriebssystem sein. Danach folgt eine Auflistung der Regionen, der einfach zusammengesetzten Zustände, der orthogonal zusammengesetzten Zustände und der einfachen Zustände. Daran schließt sich die Definition der Zustandshierarchie an. Für jede Region und jeden zusammengesetzten Zustand wird angegeben, welche direkten Unterknoten in diesem enthalten sind. Dabei muss mit dem Wurzelknoten, der eine Region ist, begonnen werden. Nach der Definition der Knotenhierarchie werden alle initialen Knoten aufgeführt.

Anschließend werden die öffentlichen, privaten und externen Ereignisnamen aufgelistet. Bevor die Definitionen der einzelnen Transitionen die Spezifikation abschließen, kann noch ein JAVA-Fragment angegeben werden, welches an den Anfang der erzeugten Java-Klasse gestellt wird. Java-Fragmente werden in spitzen Klammern angegeben.

B.1.1 Grammatik für eine Zustandsmaschine

```
Start →
    state machine name : (* string literal *) ;

    regions :                 State { , State } ;
  [ simple composite states : State { , State } ; ]
  [ orthogonal states :       State { , State } ; ]
    simple states :           State { , State } ;

    Hierarchy { Hierarchy }

    initial states : State { , State } ;

  [ public events :    Event { , Event } ; ]
  [ private events :   Event { , Event } ; ]
  [ external events :  Event { , Event } ; ]

    class Java

  { State -> Event [ "[" Java "]" ] [ / Effect ] -> State }

Hierarchy → State > State { , State } ;

State → Identifier

Event → Identifier [ . Identifier ]

Effect → [ Java ] [  Event { , Event } ]

Identifier → ( a..z | A..Z ) { ( a..z | A..Z | 0..9 | _ ) }

Java → < (* java programming language expression *) >

Comment → // ... | /* ... */
```

Programmausdruck B.1: Grammatik für Zustandsmaschinen

B.1.2 Beispiel für eine Zustandsmaschine ohne Daten

Im Folgenden ist die Zustandsmaschine aus Abschnitt 3.2 abgebildet. Bei diesem Beispiel wurde auf die explizite Verwendung von Daten verzichtet und der interne Zustand des CD-Spielers und des Kassettenspielers wird mit Hilfe von Ereignissen bestimmt.

```
state machine name : "Car Audio System";

/* nodes */
regions :                    RSTART, AudioPlayer, CDPlayer, TapeDeck,
                             RON, RTUNER, RTAPE, RCD;
simple composite states :    On, TapeMode, CDMode, TunerMode;
orthogonal states :          CarAudioSystem;
simple states:               Off, CDEmpty, CDFull, TapeEmpty, TapeFull,
                             SW1, SW2, SW3, SW4,
                             TapePlaying, BWSpooling, FWSpooling,
                             CDPlaying, NextTrack, PreviousTrack,
                             P1, P2, P3, P4;

/* hierarchy */
RSTART          > CarAudioSystem;
CarAudioSystem  > AudioPlayer, CDPlayer, TapeDeck;
AudioPlayer     > On, Off;
CDPlayer        > CDEmpty, CDFull;
TapeDeck        > TapeEmpty, TapeFull;
On              > RON;
RON             > SW1, SW2, SW3, SW4, TunerMode, TapeMode, CDMode;
TunerMode       > RTUNER;
RTUNER          > P1, P2, P3, P4;
TapeMode        > RTAPE;
RTAPE           > TapePlaying, BWSpooling, FWSpooling;
CDMode          > RCD;
RCD             > CDPlaying, NextTrack, PreviousTrack;

/* initial states */
initial states : CarAudioSystem, Off, TunerMode, CDEmpty, TapeEmpty,
                 CDPlaying, TapePlaying, P1;

/* events */
public events :   cdin, cdej, tpin, tpej, power, src, back, next, play;
private events :  cdf, cdfy, cdfn, tpf, tafy, tafn;
external events : env.SWTAPE, env.SWCD, env.TUNER, env.CD, env.TAPE,
                  env.P1, env.P2, env.P3, env.P4, env.tp, env.fws,
                  env.bws, env.cdfm, env.cdnx;

/* Transitions */

/* CDPlayer */
CDEmpty -> cdin /      -> CDFull
CDEmpty -> cdf / cdfn -> CDEmpty
CDFull  -> cdej /      -> CDEmpty
CDFull  -> cdf / cdfy -> CDFull

/* TapeDeck */
TapeEmpty -> tpin /      -> TapeFull
TapeEmpty -> tpf / tafn -> TapeEmpty
TapeFull  -> tpej /      -> TapeEmpty
TapeFull  -> tpf / tafy -> TapeFull
```

```
53
   /* AudioPlayer */
55 Off -> power / -> On
   On  -> power / -> Off
57
   /* On */
59 TunerMode -> src   / cdf; env.SWCD    -> SW1
   SW1       -> cdfy  / env.CD           -> CDMode
61 SW1       -> cdfn  / env.TUNER        -> TunerMode

63 TunerMode -> src   / tpf; env.SWTAPE -> SW2
   SW2       -> tafy  / env.TAPE         -> TapeMode
65 SW2       -> tafn  / env.TUNER        -> TunerMode

67 CDMode    -> src   / tpf; env.SWTAPE -> SW4
   CDMode    -> cdej  /                  -> TunerMode
69 SW4       -> tafy  / env.TAPE         -> TapeMode
   SW4       -> tafn  / env.TUNER        -> TunerMode
71
   TapeMode  -> src   / cdf; env.SWCD    -> SW3
73 TapeMode  -> tpej  / cdf              -> SW3
   SW3       -> cdfy  / env.CD           -> CDMode
75 SW3       -> cdfn  / env.TUNER        -> TunerMode

77 /* TapeMode */
   TapePlaying -> back / env.bws        -> BWSpooling
79 BWSpooling  -> next / env.tp         -> TapePlaying
   TapePlaying -> next / env.fws        -> FWSpooling
81 FWSpooling  -> back / env.tp         -> TapePlaying

83 /* CDMode */
   CDPlaying     -> back / env.cdfm     -> PreviousTrack
85 CDPlaying     -> next / env.cdnx     -> NextTrack
   PreviousTrack -> play                -> CDPlaying
87 NextTrack     -> play                -> CDPlaying

89 /* TunerMode */
   P1 -> next / env.P2 -> P2
91 P2 -> next / env.P3 -> P3
   P3 -> next / env.P4 -> P4
93 P4 -> next / env.P1 -> P1
   P1 -> back / env.P4 -> P4
95 P4 -> back / env.P3 -> P3
   P3 -> back / env.P2 -> P2
97 P2 -> back / env.P1 -> P1
```

Programmausdruck B.2: Musikanlage ohne Daten

B.1.3 Beispiel für eine Zustandsmaschine mit Daten

Die folgende Zustandsmaschine benutzt explizit Datenvariablen, um den internen Zustand des CD-Spielers und des Kassettenspielers zu speichern. Im initialen Java-Codefragment ist ein Prädikat definiert, mit dem dieser gespeicherte interne Zustand ausgewertet werden kann.

```
state machine name : "Car Audio System";

/* nodes */
regions :                    RSTART, AudioPlayer, CDPlayer, TapeDeck,
                             RON, RTUNER, RTAPE, RCD;
simple composite states :    On, TapeMode, CDMode, TunerMode;
orthogonal states :          CarAudioSystem;
simple states:               Off, CDEmpty, CDFull, TapeEmpty, TapeFull,
                             SW1, SW2, SW3, SW4,
                             TapePlaying, BWSpooling, FWSpooling,
                             CDPlaying, NextTrack, PreviousTrack,
                             P1, P2, P3, P4;

/* hierarchy */
RSTART          > CarAudioSystem;
CarAudioSystem  > AudioPlayer, CDPlayer, TapeDeck;
AudioPlayer     > On, Off;
CDPlayer        > CDEmpty, CDFull;
TapeDeck        > TapeEmpty, TapeFull;
On              > RON;
RON             > SW1, SW2, SW3, SW4, TunerMode, TapeMode, CDMode;
TunerMode       > RTUNER;
RTUNER          > P1, P2, P3, P4;
TapeMode        > RTAPE;
RTAPE           > TapePlaying, BWSpooling, FWSpooling;
CDMode          > RCD;
RCD             > CDPlaying, NextTrack, PreviousTrack;

/* initial states */
initial states :  CarAudioSystem, Off, TunerMode, CDEmpty, TapeEmpty,
                  CDPlaying, TapePlaying, P1;

/* events */
public events :    cdin, cdej, tpin, tpej, power, src, back, next, play;
external events :  env.TUNER, env.CD, env.TAPE,
                   env.P1, env.P2, env.P3, env.P4, env.tp, env.fws,
                   env.bws, env.cdfm, env.cdnx;

/* java class declaraionts */

class <
  private boolean inCDFull = false;
  private boolean inTapeFull = false;
  private int     trackCount;
  private boolean useData = true;
```

```
47    private boolean in(String state) {
        if (state.equals("CD Full")) {
49        return inCDFull;
        }
51      if (state.equals("Tape Full")) {
          return inTapeFull;
53      }
        return false;
55    }
  >
57
  /* Transitions */
59
  /* CDPlayer */
61 CDEmpty -> cdin / <inCDFull = true;>  -> CDFull
  CDFull  -> cdej / <inCDFull = false;> -> CDEmpty
63
  /* TapeDeck */
65 TapeEmpty -> tpin / <inTapeFull = true;>  -> TapeFull
  TapeFull  -> tpej / <inTapeFull = false;> -> TapeEmpty
67
  /* AudioPlayer */
69 Off -> power / -> On
  On  -> power / -> Off
71
  /* On */
73 TunerMode -> src [<in("CD Full")>]    / env.CD    -> CDMode
  TunerMode -> src [<in("Tape Full")>] / env.TAPE  -> TapeMode
75
  CDMode    -> src [<in("Tape Full")>] / env.TAPE  -> TapeMode
77 CDMode    -> cdej                    / env.TUNER -> TunerMode

79 TapeMode  -> src [<in("CD Full")>]    / env.CD    -> CDMode
  TapeMode  -> tpej                    / env.TUNER -> TunerMode
81
  /* TapeMode */
83 TapePlaying -> back / env.bws       -> BWSpooling
  BWSpooling  -> next / env.tp        -> TapePlaying
85 TapePlaying -> next / env.fws       -> FWSpooling
  FWSpooling  -> back / env.tp        -> TapePlaying
87
  /* CDMode */
89 CDPlaying     -> back / env.cdfm    -> PreviousTrack
  CDPlaying     -> next / env.cdnx    -> NextTrack
91 PreviousTrack -> play               -> CDPlaying
  NextTrack     -> play               -> CDPlaying
93
  /* TunerMode */
95 P1 -> next / env.P2 -> P2
  P2 -> next / env.P3 -> P3
97 P3 -> next / env.P4 -> P4
  P4 -> next / env.P1 -> P1
99 P1 -> back / env.P4 -> P4
```

```
P4 -> back / env.P3 -> P3
P3 -> back / env.P2 -> P2
P2 -> back / env.P1 -> P1
```

Programmausdruck B.3: Musikanlage mit Daten

B.1.4 Generierte Java-Klasse

Als letztes gebe ich die Java-Klasse an, die aus den Java-Codefragmenten, die in der Spezifikation der Zustandsmaschine B.3 angegeben wurden, erzeugt wurde.

```
// auto-generated class

import statemachine.structure.data.Environment;
import statemachine.structure.data.OTFTimer;
import java.util.Random;

public class CarAudioSystem extends Environment {

  private boolean inCDFull = false;
  private boolean inTapeFull = false;
  private int     trackCount;
  private boolean useData = true;

  private boolean in(String state) {
    if (state.equals("CD Full")) {
      return inCDFull;
    }
    if (state.equals("Tape Full")) {
      return inTapeFull;
    }
    return false;
  }


        public void updateT4() {
                if (useData) {
                        inTapeFull = false;
                }
        }

        public void updateT3() {
                if (useData) {
                        inTapeFull = true;
                }
        }

        public void updateT2() {
                if (useData) {
                        inCDFull = false;
```

```
40                  }
            }
42
            public void updateT1() {
44                  if (useData) {
                          inCDFull = true;
46                  }
            }
48
            public boolean guardT9() {
50                  if (useData) {
                            return in("Tape Full");
52                  }
                    return true;
54          }

56          public boolean guardT8() {
                    if (useData) {
58                          return in("Tape Full");
                    }
60                  return true;
            }
62
            public boolean guardT7() {
64                  if (useData) {
                            return in("CD Full");
66                  }
                    return true;
68          }

70          public boolean guardT11() {
                    if (useData) {
72                          return in("CD Full");
                    }
74                  return true;
            }
76

78  }
```

Programmausdruck B.4: Generierte Java Klasse

B.2 Testfälle

Im Folgenden ist die Grammatik und ein Beispiel für einen Testfall angegeben. In Testfällen wird zunächst ein Name für den Testfall angegeben. Im Weiteren folgen die Ereignisse, die der Testfall an das System unter Test sendet und die Ereignisse, die der Testfall vom System unter Test empfängt und gleichzeitig als relevant betrachtet werden. Ereignisse, die dort nicht aufgeführt sind, werden während des Testens als nicht relevant herausgefiltert. Auf diese Art und Weise wird die Beschränkung der ausgehenden Schnittstelle in einem Testfall realisiert. Der

eigentliche Testfall besteht aus einer Aufzählung von Ereignissen, die an das System unter Test geschickt werden sollen, oder aus einer textuellen Repräsentation eines Akzeptanzgraphen.

B.2.1 Grammatik für einen Testfall

```
Start → test case name : (* string literal *) ;

    trigger events  : Event { , Event } ;
  [ external events : Event { , Event } ; ]

  { TestEntry }

Event → Identifier [ . Identifier ]

TestEntry → Event | { Number : "{" Observation "}" }

Identifier → ( a..z | A..Z ) { ( a..z | A..Z | 0..9 | _ ) }

Number → 1..9 { 0..9 } | 0

Observation → Event [ : Number : Number ] -> NextEntry

NextEntry → #  | @  | Number

Comment → // ... | /* ... */
```

Programmausdruck B.5: Grammatik für Testfälle

B.2.2 Beispiel für einen Testfall ohne Daten

```
/*
         TCGD - automatically generated test case.
*/

test case name : "Car Audio System_0001";

trigger events : this.cdin, this.cdej, this.tpin, this.tpej, this.power, thi
s.src, this.back, this.next, this.play;
external events : env.SWTAPE, env.SWCD, env.TUNER, env.CD, env.TAPE, env.P1,
 env.P2, env.P3, env.P4, env.tp, env.fws, env.bws, env.cdfm, env.cdnx;

this.power

this.back

this.back

this.next

this.back
```

```
this.back

this.next

this.cdin

this.src

this.next


{
  9 :             this.tmo ->    #
  8 :             this.tmo ->    #           env.cdnx ->    9
  7 :               env.CD ->    8
  6 :             env.SWCD ->    7           env.SWTAPE ->   10
  5 :               env.P3 ->    6
  4 :               env.P2 ->    5
 12 :             this.tmo ->    #
  3 :               env.P3 ->    4
 11 :               env.P2 ->   12           this.tmo ->    #
  2 :               env.P4 ->    3
 10 :            env.TUNER ->   11
  1 :               env.P3 ->    2
  0 :               env.P4 ->    1
}
```

Anhang C

Modelle der Jalousiesteuerung

Im Folgenden sind die Modelle zur Fallstudie der automatischen Jalousiesteuerung aus Abschnitt 6.2 abgebildet.

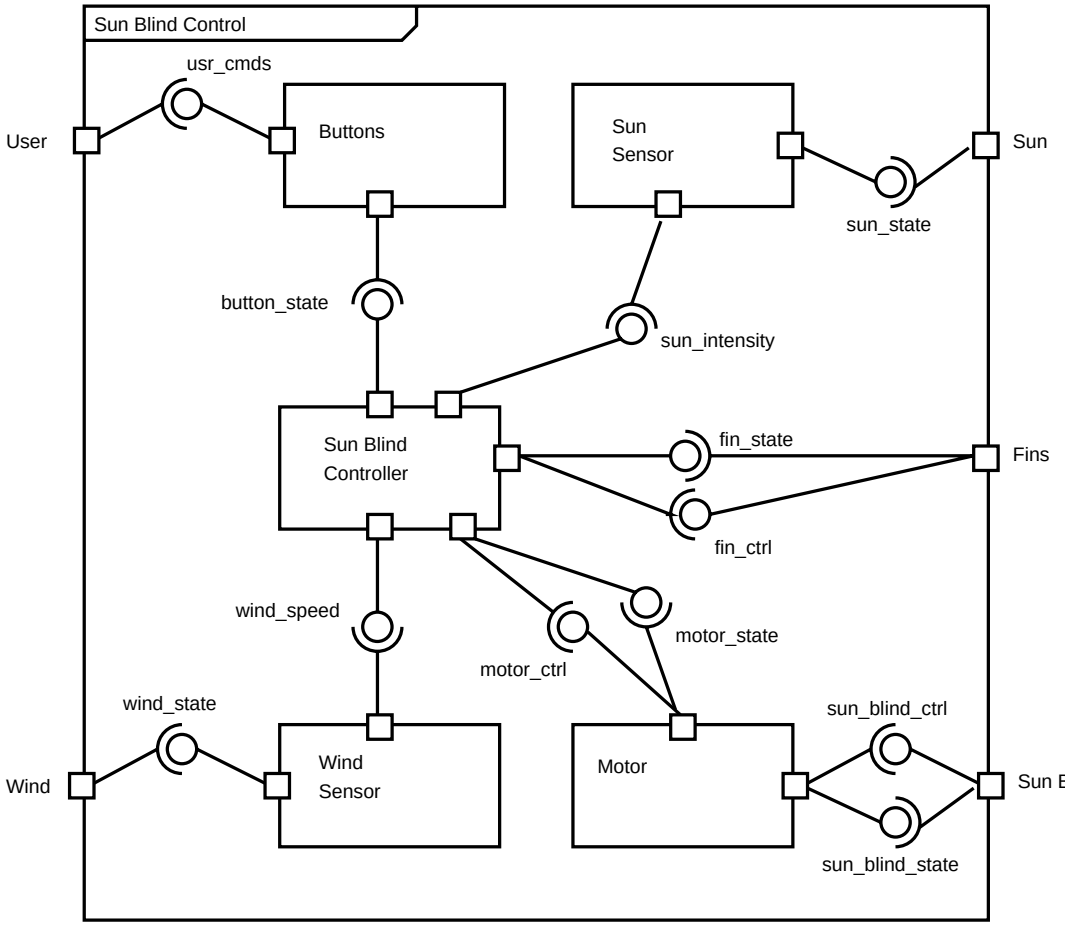


Abbildung C.1: Systemarchitektur der Jalousiesteuerung.

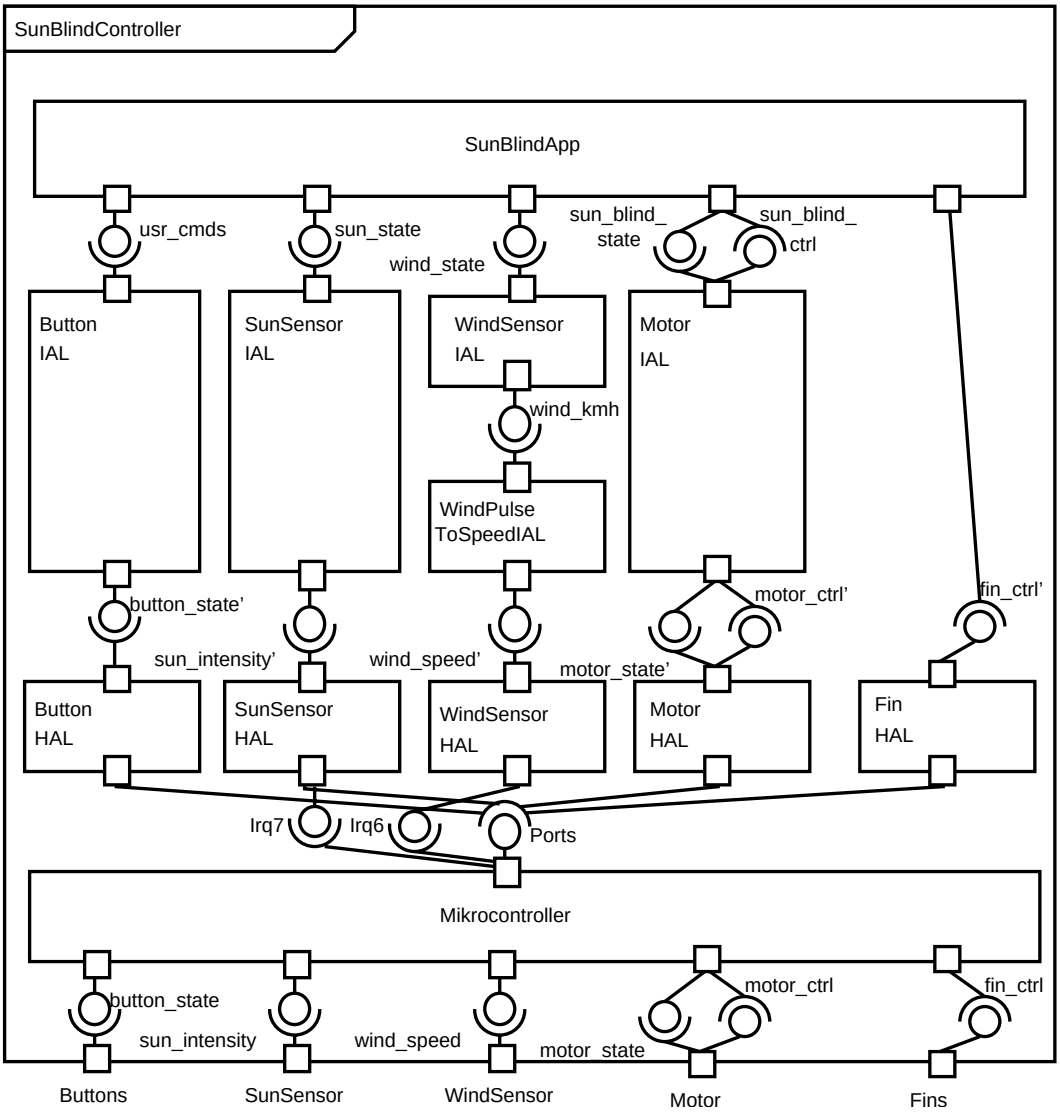
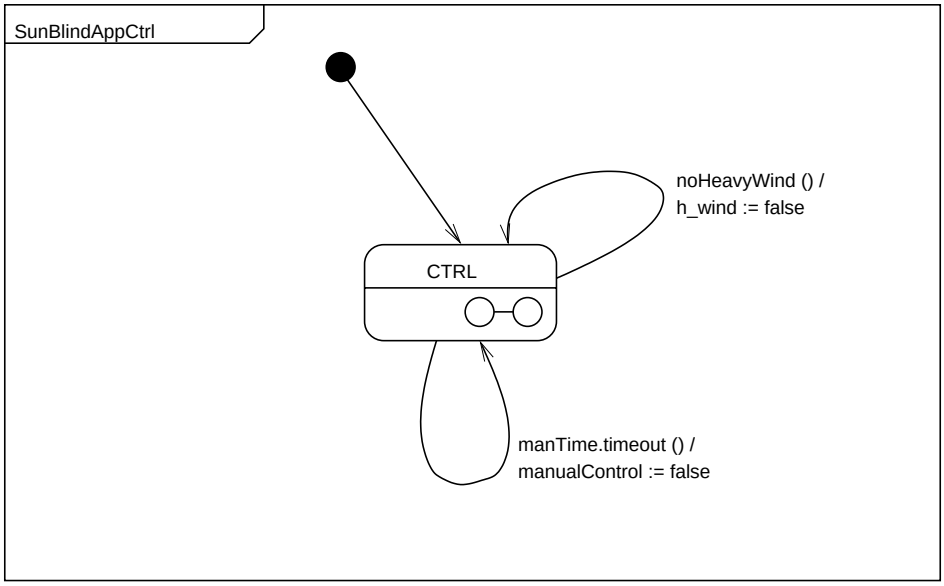


Abbildung C.2: Softwarearchitektur der Jalousiesteuerung.



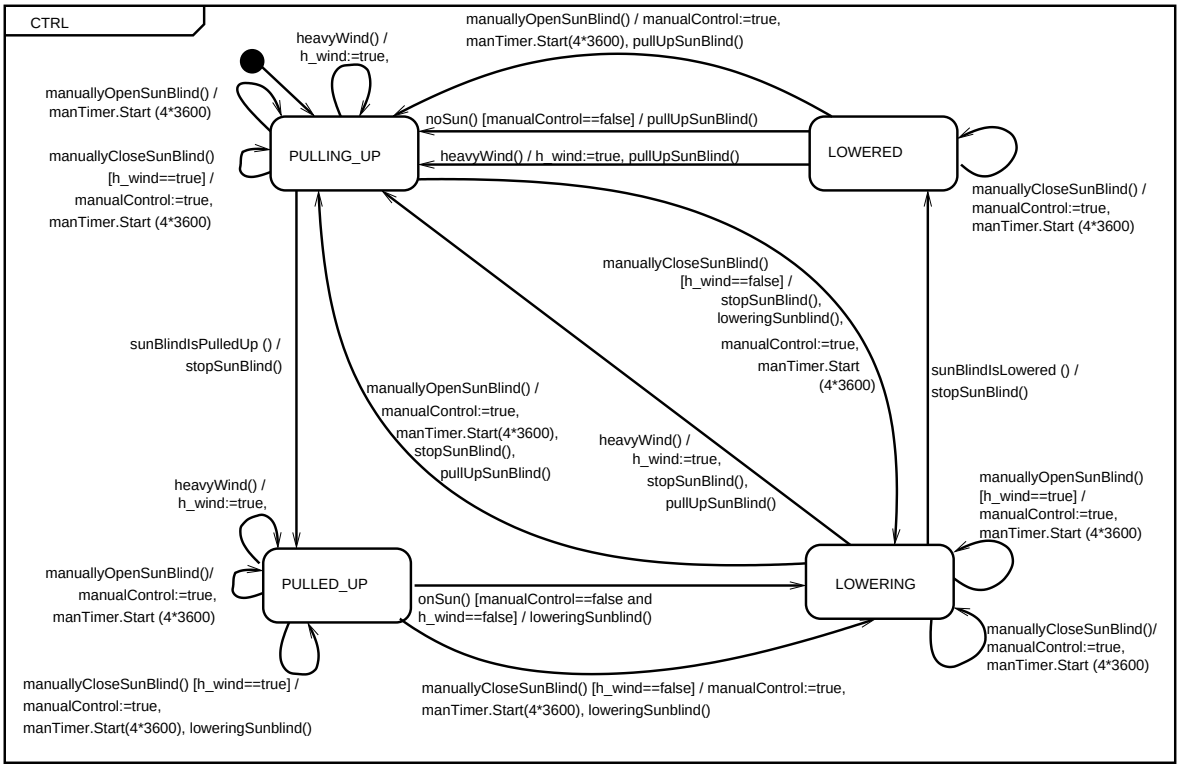


Abbildung C.3: Zustandsmaschine des Jalousie-Controllers.

```
// Sun Blind Control Example - batch approach

state machine name : "SBCbatch";

regions :            R_SBC,
                     R_CTRL, R_SUNDETECTION, R_WIND;
orthogonal states :  SunBlindAppCtrl;
simple states:       PULLING_UP, PULLED_UP, LOWERING, LOWERED,
                     NO_SUN, WF_SUN, SUN, WF_NO_SUN, Wind;

R_SBC            > SunBlindAppCtrl;
SunBlindAppCtrl > R_CTRL, R_SUNDETECTION, R_WIND;

R_CTRL           > PULLING_UP, PULLED_UP, LOWERING, LOWERED;
R_SUNDETECTION   > NO_SUN, WF_SUN, SUN, WF_NO_SUN;
R_WIND           > Wind;

initial states : SunBlindAppCtrl, PULLING_UP, NO_SUN, Wind;

public events : manuallyOpenSunBlind, adjFinsPos,
                manuallyCloseSunBlind, adjFinsNeg,
                heavyWind, noHeavyWind,
                sunBlindIsPulledUp, sunBlindIsLowered,
                timeout1, timeout2,
                onSunShine, noSunShine;

private events : onSun, noSun;

external events : motor.stopSunBlind, motor.loweringSunBlind,
                  motor.pullUpSunBlind,fins.rotateFinsWithPositiveDegree_,
                  fins.rotateFinsWithNegativeDegree_;

class <
  boolean useData = false;

  boolean manualControl = false;
  boolean heavyWind = false;
>

// Fehler 1 (auskommeniert -> konform)
//CTRL -> noHeavyWind / [h_wind = false;] -> CTRL
//CTRL -> timeout1     / <manualControl = false;> -> CTRL
Wind -> noHeavyWind / <heavyWind=false;> -> Wind

PULLING_UP -> heavyWind / <heavyWind=true;> -> PULLING_UP

PULLING_UP -> adjFinsNeg [<!heavyWind>]
/ <manualControl=true;>
                     motor.stopSunBlind; motor.loweringSunBlind -> LOWERING

PULLING_UP -> manuallyCloseSunBlind [<!heavyWind>] / <manualControl=true;>
                     motor.stopSunBlind; motor.loweringSunBlind -> LOWERING
```

```
52
PULLING_UP -> sunBlindIsPulledUp / motor.stopSunBlind -> PULLED_UP
54
PULLING_UP -> adjFinsPos[<heavyWind>] / fins.rotateFinsWithPositiveDegree_
56                                                          -> PULLING_UP

58
LOWERED -> manuallyOpenSunBlind / <manualControl=true;>
60                                          motor.pullUpSunBlind -> PULLING_UP

62 LOWERED -> adjFinsPos [<!heavyWind>] / fins.rotateFinsWithPositiveDegree_
                                                            -> LOWERED
64
LOWERED -> adjFinsNeg [<!heavyWind>] / fins.rotateFinsWithNegativeDegree_
66                                                          -> LOWERED

68 LOWERED -> heavyWind / <heavyWind=true;> motor.pullUpSunBlind
                                                            -> PULLING_UP
70
LOWERED -> noSun [<!manualControl && !heavyWind>] / motor.pullUpSunBlind
72                                                          -> PULLING_UP

74
LOWERING -> heavyWind / <heavyWind=true;> motor.stopSunBlind;
76                                          motor.pullUpSunBlind -> PULLING_UP

78 LOWERING -> sunBlindIsLowered / motor.stopSunBlind -> LOWERED

80 LOWERING -> adjFinsNeg [<!heavyWind>] / fins.rotateFinsWithNegativeDegree_
                                                            -> LOWERING
82
LOWERING -> manuallyOpenSunBlind / <manualControl=true;>
84                  motor.stopSunBlind; motor.pullUpSunBlind -> PULLING_UP

86 LOWERING -> adjFinsPos [<!heavyWind>] / <manualControl=true;>
                    motor.stopSunBlind; motor.pullUpSunBlind -> PULLING_UP
88

90 PULLED_UP -> onSun[<!manualControl && !heavyWind>] /motor.loweringSunBlind
                                                            -> LOWERING
92
PULLED_UP -> manuallyCloseSunBlind [<!heavyWind>] / <manualControl=true;>
94                                          motor.loweringSunBlind -> LOWERING

96 PULLED_UP -> adjFinsPos[<heavyWind>] / fins.rotateFinsWithPositiveDegree_
                                                            -> PULLED_UP
98
PULLED_UP -> adjFinsNeg[<!heavyWind>] / fins.rotateFinsWithNegativeDegree_
100                                                         -> PULLED_UP

102 PULLED_UP -> heavyWind / <heavyWind=true;> -> PULLED_UP

104 // Sun Detection
```

```
NO_SUN -> onSunShine -> WF_SUN
106
WF_SUN -> noSunShine -> NO_SUN
108
WF_SUN -> timeout2 / onSun -> SUN
110
SUN -> noSunShine -> WF_NO_SUN
112
WF_NO_SUN -> onSunShine -> SUN
114
WF_NO_SUN -> timeout2 / noSun -> NO_SUN
```

Programmausdruck C.1: Zustandsmaschine des Jalousie-Controllers (TEAGER)

Anhang D

Deutsch-Englische Begriffswelt

Im Folgenden sind die von mir im Kapitel 3 verwendeten deutschen Fachbegriffe und ihre in der englischsprachigen Literatur, insbesondere die in der UML, verwendeten Entsprechungen mit einer kurzen Erläuterung ihrer Verwendung in dieser Arbeit aufgelistet.

Aktion *action*
: Allgemein wird als Aktion die kleinste Einheit von Verhalten beschrieben, das bei der Ausführung einer Transition ausgeführt werden kann. Eine Aktion besteht entweder aus einer Aktualisierung des Datenraums oder aus der Erzeugung einer neuen Ereignisinstanz.

Effekt einer Transiton *effect*
: Als Effekt einer Transition wird die Gesamtheit aller Aktionen einer Transition bezeichnet. Wobei die Aktionen einer Transition sequentiell geordnet sind.

Einfacher Zustand *simple state*
: Ein einfacher Zustand einer Zustandsmaschine ist ein Zustand, der durch keine weiteren Zustände verfeinert wird. Einfache Zustände sind jeweils die innersten Zustände einer Zustandsmaschine.

Einfach zusammengesetzter Zustand *simple composite state*
: Ein einfach zusammengesetzter Zustand einer Zustandsmaschine ist ein Zustand, der durch genau eine Region verfeinert wird.

Ereignis, Ereignisinstanz, auslösendes Ereignis *event, event occurence*
: Als Ereignis wird im Allgemeinen ein Phänomen beschreiben, das, wenn es auftritt und verarbeitet wird, eine Transition auslösen kann. Daneben wird anstelle von Ereignis auch Ereignisname benutzt. Dabei bezeichnet Ereignis den Typ bzw. den Ereignisnamen und Ereignisinstanz bezeichnet eine konkrete Ausprägung. Hier wird dadurch speziell zwischen einem abstrakten Ereignis und einem konkreten, mit Daten instanziierten, Ereignis

unterschieden. Als auslösendes Ereignis (*trigger*) einer Transition wird das Ereignis bezeichnet, das die Transition aktiviert. Das auslösende Ereignis repräsentiert damit in gewisser Hinsicht den Typ aller auslösenden Ereignisinstanzen der Transition.

Hauptquellzustand, Hauptzielzustand *main source, main target*
Durch die Ausführung einer Transition wird der Hauptquellzustand verlassen und der Hauptzielzustand betreten. Beide Zustände können zusammengesetzte Zustände sein. Die Zustände markieren die Wurzel der Teilbäume, die von der Transition bezüglich des Verlassens bzw. Betretens betroffen sind. Auf Basis dieser Bäume werden jeweils die Mengen der Zustände, die verlassen bzw. betreten werden, bestimmt.

Initialer Zustand *initial state*
Als initialer Zustand wird der Zustand einer Region bezeichnet, der aktiviert wird, wenn die Region aktiviert wird, ohne dass ein anderer Zustand der Region explizit aktiviert wird.

Kleinster gemeinsamer Oberknoten *least common ancestor*
Als kleinster gemeinsamer Oberknoten einer Transition wird der tiefste Knoten in der Hierarchierelation bezeichnet, der sowohl den Quellzustand, als auch den Zielzustand einer Transition enthält. Der kleinste gemeinsame Oberknoten kennzeichnet somit den Wirkungsbereich einer Transition.

Konkurrierende Transitionen *conflicting transitions*
Transitionen, deren Schnittmenge der Zustände die sie jeweils verlassen, nicht leer ist, stehen in Konflikt zueinander. Der Konflikt rührt daher, dass ihre gleichzeitige Ausführung zu keiner wohlgeformten Folgekonfiguration führen würde.

Knoten *node, vertex*
Als Knoten wird die Abstraktion von Zustand oder Region in einer Zustandsmaschine bezeichnet.

Markierung *label*
Als Markierung wird die Beschriftung einer Transition bezeichnet. Eine Markierung setzt sich aus einem auslösenden Ereignis und optional aus einer Übergangsbedingung und einem Effekt zusammen.

Orthogonaler Zustand *orthogonal state*
Ein orthogonal zusammengesetzter Zustand einer Zustandsmaschine ist ein Zustand, der durch mindestens zwei Regionen verfeinert wird.

Quellzustand *source state*
Mit Quellzustand wird der Zustand bezeichnet, von dem eine Transition direkt ausgeht und der verlassen wird, wenn die Transition ausgeführt wird.

Region *region*
Als Region wird ein Knoten bezeichnet, der einen einfach zusammengesetzten bzw. einen orthogonal zusammengesetzten Zustand verfeinert. Eine Region dient primär als Container für Zustände, die einen Zustand verfeinern.

Run-to-completion Verarbeitung *Run-to-completion processing*
Als Run-to-completion Verarbeitung wird eine Verarbeitungsvorschrift bezeichnet, in der die Verarbeitung einer einzigen Ereignisinstanz erst vollständig abgeschlossen sein muss, bevor mit der Verarbeitung der nächsten Ereignisinstanz begonnen werden kann. Die Ereignisinstanzen, die zur Verarbeitung anstehen, werden in einem Ereignisspeicher zwischengespeichert.

Schalten einer Transition *fire*
Das Ausführen einer Transition wird als Schalten der Transition bezeichnet. Dabei werden alle Zustände deaktiviert, die von der Transition verlassen werden, die einzelnen Aktionen der Transition ausgeführt und die Zustände, die durch die Transition betreten werden, aktiviert.

Semantischer Zustand *Statuts*
Als semantischer Zustand wird ein Zustand des semantischen Modells einer Zustandsmaschine bezeichnet. Dieser semantische Zustand ist durch die aktiven Zustände der Zustandsmaschine, durch den aktuellen Inhalt des Ereignisspeichers und durch die aktuelle Datenraumbelegung gekennzeichnet.

Transition *transition*
Als Transition wird die gerichtete Beziehung zwischen einem Quellzustand und einem Zielzustand bezeichnet. Eine Transition ist mit einer Markierung beschriftet, die dabei die Reaktion der Zustandsmaschine auf das Auftreten eines spezifizierten Ereignisses beschreibt.

Transitionsselektionsalgorithmus *transition selection algorithm*
Als Reaktion auf empfangene Ereignisse muss eine maximal mächtige Menge von Transitionen bestimmt werden, die konfliktfrei sind und somit innerhalb eines semantischen Schritts ausgeführt werden können. Dabei gilt für alle Transitionen in einer solchen Menge, dass sie aktiviert sein müssen, dass sie nicht in Konflikt zueinander stehen und dass es keine Transition außerhalb dieser Menge geben darf, die höher priorisiert ist, als eine Transition innerhalb dieser Menge. Der Algorithmus zur Bestimmung einer solchen Menge wird Transitionsselektionsalgorithmus genannt.

Übergangsbedingung einer Transition *guard*
Optional kann für jede Transition eine Übergangsbedingung spezifiziert werden. Die Bedingung ermöglicht eine feingranulare Beschreibung der Situation, in der die Transition schalten soll.

Unterzustand *substate*
Jeder Zustand, der in einer Region eines zusammengesetzten Zustands enthalten ist, wird als Unterzustand bezeichnet.

Zielzustand *target state*
Als Zielzustand wird der Zustand bezeichnet, der erreicht wird, wenn eine Transition ausgeführt wird.

Zusammengesetzter Zustand *composite state*
Als zusammengesetzte Zustände werden alle Zustände bezeichnet, die entweder durch

eine (einfach zusammengesetzter Zustand), oder durch mindestens zwei Regionen (orthogonal zusammengesetzter Zustand) verfeinert werden.

Zustand *state*
Ein Zustand modelliert eine Situation der spezifizierten Komponente, in der (meistens implizit) eine bestimmte invariante Bedingung gilt. Häufig wird auch von einem Kontrollzustand des Systems oder der Komponente gesprochen.

Zustandskonfiguration *active state configuration*
Auf Grund der hierarchischen und der orthogonalen Komposition von Zuständen können mehrere Zustände einer Zustandsmaschine gleichzeitig aktiv sein. Die Menge aller zu einem bestimmten Zeitpunkt aktiven Zustände wird Zustandskonfiguration genannt.

Zustandsmaschine *state maschine*
Eine Zustandsmaschine spezifiziert das Verhalten eines Systems oder einer Systemkomponente. Dabei werden Sequenzen von Zuständen, die im Laufe des Lebenszyklus durchlaufen werden, und die Übergänge zwischen diesen als Reaktion auf empfangene Ereignisse, sowie die Reaktion und die ausgeführten Aktionen beschrieben.

Literaturverzeichnis

[1] S. Abramsky and T. S. E. Maibaum, editors. *Testing Against Formal Specifications: a Theoretical View*, volume 494 of *Lecturer Notes in Computer Science*. Springer-Verlag, 1991.

[2] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[3] Asml 2. Microsoft Research - Foundations of Software Engineering Group, 2005. `research.microsoft.com/fse/asml`.

[4] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. Interactive Verification of UML State Machines. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Engineering Methods (ICFEM'04)*, volume 3308 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2004.

[5] H. Balzert. *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, Heidelberg, 1998.

[6] K. Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.

[7] G. Bernot, M.-C. Gaudel, and B. Marre. Software Testing based on formal Specifications: a Theory and a Tool. *IEE Software Engineering Journal*, 6, 1991.

[8] B. W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. *EURO IFIP'79*, pages 711–719, 1979.

[9] B. W. Boehm. The Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, May 1988.

[10] H. Borck. Erweiterung eines Frameworks zum Testen auf Basis von UML Zustandsmaschinen um Daten und deren Auswertung. Diplomarbeit, Fachgebiet Softwaretechnik, Technische Universität Berlin, 2007. In Bearbeitung.

[11] M. Born and E. H. und Olaf Kath. *Softwareentwicklung mit UML 2*. Addison Wesley, 2004.

[12] R. Bosch GmbH. CAN — Controller Area Network Specification Version 2.0, Stuttgart, Germany, 1991.

[13] E. Brinksma. A Theory for the Derivation of Tests. In *Protocol Specification, Testing and Verification*. North-Holland, 1988.

[14] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A Formal Approach to Confomance Testing. In L. M. J. de Meer and W. Effelsberg, editors, *Workshop on Protocal Test Systems*, pages 349–363, 1990.

[15] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. *Lecture Notes in Computer Science*, pages 187–??, 2001.

[16] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[17] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[18] A. Cockburn. *Agile Software Development – The Cooperative Game*. Addison-Wesley, 2006.

[19] Conformiq Test Generator. Conformiq Software Ltd., 2007. `www.conformiq.com`.

[20] T. Coquand and G. Huet. The Coq Proof Assistant. INRIA Rocquencourt, 2007. `coq.inria.fr`.

[21] R. De Nicola. Extensional Equivalences for Transition Systems. *Acta Informatica*, 24(2):211–237, 1987.

[22] R. De Nicola and M. C. B. Hennessy. Testing Equivalence for Processes. In J. Díaz, editor, *Automata, Languages and Programming, 10th Colloquium*, volume 154 of *Lecture Notes in Computer Science*, pages 548–560, Barcelona, Spain, 1983. Springer-Verlag.

[23] R. De Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, pages 83–133, 1984.

[24] R. De Nicola and R. Segala. A Process Algebraic View of Input/Output Automata. *Theoretical Computer Science*, 138(2):391–423, 1995.

[25] R. Eshuis and R. Wieringa. Requirements Level Semantics for UML Statecharts. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems*. Kluwer Academic Publishers, 2000.

[26] FDR. Failures-Divergence Refinement - Model Checker. Formal Systems (Europe) Ltd, 2007. `www.fsel.com`.

[27] H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[28] C. Gaston and D. Seifert. Evaluating coverage based testing. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 293–322. Springer, 2004.

[29] C. Gaston and D. Seifert. Evalutating Coverage Based Testing. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 293–322. Springer-Verlag, 2005.

[30] M.-C. Gaudel. Testing Can Be Formal, Too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.

[31] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. *ACM International Symposium on Software Testing and Analysis*, 2002.

[32] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[33] D. Harel. Some Thoughts on Statecharts, 13 Years Later. In *International Conference on Computer Aided Verification (GAV'97)*, 1997.

[34] D. Harel and E. Gery. Executable Object Modeling with Statecharts. In *18th International Conference on Software Engineering*, pages 246–257. IEEE Press, 1996.

[35] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.

[36] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[37] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering*. Addison Wesley, 2001.

[38] M. Heisel. Assumption. In *?*, 2005.

[39] M. Heisel and D. Hartebur. Sun Blind Control: Fallstudie in der Lehrveranstaltung *Embedded Systems*. Fachgebiet Softwaretechnik, Universität Duisbur-Essen, 2006. `swe.uni-due.de/en/education/ws0607/embsys/index.php`.

[40] S. Helke and F. Kammüller. Verification of Statecharts Including Data Spaces. In D. Basin and B. Wolff, editors, *TPHOLs 2003: Emerging Trends Proceedings*, Technischer Report 189, pages 177–190. Albert-Ludwigs-Universität Freiburg, 2003.

[41] S. Helke, A. Nordwig, T. Santen, and D. Sokenou. Scaling-Up von V & V-Techniken durch Integration und Abstraktion. Rigorose Entwicklung software-intensiver Systeme. In M. Wirsing, M. Gogolla, H.-J. Kreowski, T. Nipkow, and W. Reif, editors, *Tagungsbericht GI 2000*, pages 11–20. Institut für Informatik, Ludwig-Maximilians-Universität München, 2000. Bericht 0005.

[42] F. Herbreteau, F. Cassez, A. Finkel, O. Roux, and G. Sutre. Verification of Embedded Reactive Fiffo Systems. In *Latin American Theoretical Informatics Conference (LATIN'02)*, LNCS 2286, pages 400–414. Springer, 2002.

[43] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[44] P. Hofstedt, D. Seifert, and E. Godehardt. A Framework for Cooperating Constraint Solvers - A Prototypic Implementation. In *Workshop on Cooperative Solvers in Constraint Programming (CoSolv01)*, 2001.

[45] G. J. Holzmann. *Disign and Validation of Computer Protocols.* Prentice Hall, 1991.

[46] G. J. Holzmann. *The SPIN Model Checker.* Pearson Education, 2003.

[47] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic Test Generation from Statecharts using Model Checking. In *Workshop on Formal Approaches to Testing of Software*, pages 15–30, 2001.

[48] F. Houdek. Beispiel-Systemspezifikation: Türsteuergerät. DaimlerChrysler AG, Foschung und Technologie, Ulm, Germany, 2001.

[49] International Organisation for Standardization. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, 2000. Reference number: ISO/IEC 13568:2002.

[50] International Telecommunication Union. Message Sequence Chart (MSC). SDL Forum Society, 2007. `www.sdl-forum.org/MSC`.

[51] International Telecommunication Union. Specification and Description Language (SDL). SDL Forum Society, 2007. `www.sdl-forum.org/SDL`.

[52] M. Jackson. *Problem Frames : Analysing and Structuring Software Development Problems.* ACM Press, 2001.

[53] Java se development kit 6. Sun Microsystems, 2007. `java.sun.com`.

[54] H. Jiang, D. Lin, and X. Xie. The Formal Semantics of UML State Machine. *Journal of Software*, 13(12):2244–2250, 2002.

[55] A. Kerbrat, T. Jeron, and R. Groz. Automated Test Generation from SDL Specifications. In *9th SDL Forum*, pages 135–151, 1999.

[56] S. Kripke. Semantical Considerations on Modal Logic. *Acta Philosphica Fennica*, 16:83–94, 1963. Es gibt noch eine gleichlautende Veroeffentlichung von 71, die im Oxford University Press erschienen ist.

[57] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[58] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, page 465. Kluwer, 1999.

[59] D. Latella and M. Massink. On Testing and Conformance Relations for UML Statechart Diagrams Behaviours. In *Proc. International Symposium on Software Testing and Analysis*, ACM, 2002.

[60] H. Liebig. *Rechnerorganisation.* Springer Verlag, 2003. 3. Auflage.

[61] P. Liggesmeyer. *Software-Qualität. Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, 2002.

[62] J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *The Unified Modeling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer, 1999.

[63] J. Lilius and I. P. Paltor. The Semantics of UML State Machines. Technical Report TUCS Tecnical Report No 273, Turku Centre for Computer Science, Finland, 1999.

[64] Matlab & Simulink. The MathWorks Inc., 2007. `www.mathworks.com`.

[65] K. McMillan. SMV Home Page. `www.kenmcmil.com/smv.html`.

[66] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[67] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[68] Model Driven Architecture. MDA Guide. Object Management Group, 2003. Version 1.0.1, mg/2003, `www.uml.org/mda`.

[69] G. J. Myers. *Methodisches Testen von Programmen.* Oldenbourg Wissenschaftsverlag, 1979.

[70] T. Nipkow and L. C. Paulson. Isabelle Home Page. `isabelle.in.tum.de`.

[71] S. Owre, J. Rushby, , and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science (LNCS)*, pages 748–752. Springer-Verlag, 1992.

[72] D. L. Parnas and J. Madey. Functional Documents for Computer Systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[73] T. Parr. ANTLR – ANother Tool for Language Recognition. `www.antlr.org`, 2007. Version 2.7.6.

[74] L. C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF.* Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.

[75] L. C. Paulson. The Foundation of a Generic Theorem Prover, 1987.

[76] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[77] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[78] M. Pietsch. Bewertung von Überdeckungsmaßen für automatisch generierte Testfälle auf Basis von UML-Zustandsmaschinen. Diplomarbeit, Fachgebiet Softwaretechnik, Technische Universität Berlin, 2007. In Vorbereitung.

[79] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Theoretical Aspects of Computer Software (TACS'91)*, pages 244–264. Springer, 1991.

[80] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, 1991.

[81] Integrated Quality Assurance and Requirements Analysis for the Software Development in Automotive Systems (QUASAR), Förderkennzeichen VFG0004A, 1999. `www.first.fhg.de/quasar`.

[82] Reactis Tester. Reactive Systems Inc., 2007. `www.reactive-systems.com`.

[83] Rhapsody. I-Logix Inc., 2005. `www.ilogix.com`.

[84] A. W. Roscoe. Model-Checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 353–378. Prentice Hall International, 1994.

[85] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *WESCON Technical Papers, v. 14*, Los Angeles, 1970. WESCON.

[86] T. Santen and D. Seifert. Executing UML State Machines. Forschungsberichte der Fakultät IV - Elektrotechnik und Informatik ,2006-4, Technische Universität Berlin, 2006. ISSN 1436-9915.

[87] T. Santen and D. Seifert. TEAGER - Test Automation for UML State Machines. In B. Biel, M. Book, and V. Gruhn, editors, *Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 79 of *Lecture Notes in Informatics*, pages 73–84. Gesellschaft für Informatik, 2006.

[88] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001.

[89] D. Schien. Coverage Criteria to Evaluate and Control a Testprocess based on UML State Machines. Diplomarbeit, Fachgebiet Softwaretechnik, Technische Universität Berlin, 2007. In Bearbeitung.

[90] S. Schneider. *The B-Method: An Introduction.* Palgrave, 2002.

[91] K. Schwaber. *Agile Project Management with Scrum.* Microsoft Press, 2004.

[92] D. Seifert. TEAGER – Test Execution and Generation Environment for Reaktive Systems. Fachgebiet Softwaretechnik, Technische Universität Berlin, 2007. swt.cs.tu-berlin.de/~seifert/teager.html.

[93] D. Seifert. *Testen asynchroner nichtdeterministischer Systeme mit Daten.* PhD thesis, Fakultät IV – Elektrotechnik und Informatik, Technische Universität Berlin, Verteidigung ist für Juli 2007 geplant.

[94] D. Seifert, S. Helke, and T. Santen. Conformance Testing for Statecharts. Technical Report 2003/1, Technical University of Berlin, 2003.

[95] D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts. In M. Broy and A. V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, (PSI'03),*, volume 2890 of *Lecture Notes in Computer Science*, pages 462–468. Springer-Verlag, 2003.

[96] H. M. Sneed and M. Winter. *Testen objektorientierter Software.* Carl Hanser Verlag, 2002.

[97] Software Technology Laboratory. STL: UML Semantics Project. School of Computing, Queen's University, 2007. http://www.cs.queensu.ca/~stl/internal/uml2/ bibtex/ref_umlstatemachines.htm.

[98] I. Sommerville. *Software Engineering.* Addison-Wesley, 7th edition, 2004.

[99] A. Spillner and T. Linz. *Basiswissen Softwaretest.* dpunkt.verlag, 2005. 3., überarbeitete und aktualisierte Auflage.

[100] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[101] Statemate Model Checker. Systems and Software Modeling Business Unit, 2007. `www.ilogix.com`.

[102] C. Sühl. Referenzarchitektur für Komponenten zur Steuerung un d Kontrolle technischer Prozesse. Fraunhofer FIRST, March 2002.

[103] C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[104] Torx. University of Twente, 2007. `fmt.cs.utwente.nl/tools/torx`.

[105] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software–Concepts and Tools*, 17(3):103–120, 1996.

[106] J. Tretmans. Test Generation with Inputs, Outputs, and Repetitive Quiescence. In T. Margaria and B. Steffen, editors, *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[107] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In *10th International Conference on Concurrency Theory (CONCUR'99)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[108] J. Tretmans and A. Belinfante. Automatic Testing with Formal Methods. In *7th European Int. Conference on Software Testing, Analysis & Review.* EuroStar Conferences, 1999.

[109] J. Tretmans and L. Verhaard. A Queue Model Relating Synchronous and Asynchronous Communication. In *12th International Symposium on Protocol Specification, Testing and Verification*, pages 131–145, 1992.

[110] UML2. Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, 2005. Version 2.0, formal/05-07-04, `www.uml.org/uml`.

[111] UML2. Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, 2007. Version 2.1.1, formal/07-02-03, `www.uml.org/uml`.

[112] Das neue V-Modell XT Release 1.2 - Der Entwicklungsstandard für IT-Systeme des Bundes. Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung, 2007. `www.v-modell-xt.de`.

[113] Das V-Modell. Industrieanlagen-Betriebsgesellschaft mbH, 2007. `v-modell.iabg.de`.

[114] F. W. Vaandrager. On the relationship between process algebra and input/output automata. In *LICS: IEEE Symposium on Logic in Computer Science*, 1991.

[115] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On Asynchronous Testing. In *5th International Workshop on Protocol Test Systems*, volume C-8 of *IFIP Transactions*. North-Holland, 1992.

[116] M. Von der Beeck. A Comparison of Statechart Variants. In *Formal Techniques in RealTime and FaultTolerant Systems*, pages 128–148. Springer-Verlag, 1994.

[117] M. Von der Beeck. A Comparison of Statecharts Variants. *Lecture Notes in Computer Science*, 863:128–148, 1994.

[118] E. Wallmüller. *Software-Qualitätssicherung*. Hanser Verlag, 1990.

[119] K. Wallnau, S. A. Hissam, and R. C. Seacord, editors. *Building Systems from Commercial Components*. Addison-Wesley, 2002.

[120] J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof*. Prentice-Hall, 1996.