Inkrementelle Entwicklung von Verhaltensmodellen zum Test von reaktiven Systemen

Wolfgang Prenninger

Institut für Informatik
Technische Universität München
Boltzmannstraße 3, D–85748 Garching

Institut für Informatik
der Technischen Universität München
Lehrstuhl für Software & Systems Engineering

Inkrementelle Entwicklung von Verhaltensmodellen zum Test von reaktiven Systemen

Wolfgang Ludwig Johann Prenninger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Bernd Brügge, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Univ.-Prof. Dr. Helmut Veith

Die Dissertation wurde am 24.02.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.06.2005 angenommen.

Kurzfassung

Testen bezeichnet Aktivitäten mit dem Ziel, die Übereinstimmung oder die Abweichung des Soll- und Istverhaltens eines Systems nachzuweisen. Das Sollverhalten wird durch Testfälle codiert, die neben den Eingaben für das zu testende System auch die erwarteten Ausgaben spezifizieren. Nach Applikation der Eingaben an das zu testende System werden die Ausgaben des Systems mit den erwarteten Ausgaben des Testfalls verglichen.

Eine viel versprechende Testmethodik ist das modellbasierte Testen, das Nachteile klassischer Methoden zur Definition von Testfällen überwindet und ein strukturiertes, nachvollziehbares und automatisches Ableiten von Testfällen unterstützt. Die Idee des modellbasierten Testens ist, das Sollverhalten explizit durch ein abstraktes Verhaltensmodell, das so genannte Testmodell, zu spezifizieren und aus diesem Testmodell Testfälle für das zu testende System abzuleiten.

Eine Reihe von erfolgreichen Fallstudien im industriellen Kontext belegen, dass aus technischer Sicht bereits ausreichend leistungsfähige Testfallgeneratoren zur automatischen Ableitung von Testfällen zur Verfügung stehen. Methodische Probleme bei der Erstellung von Testmodellen beeinträchtigen jedoch den effizienten Einsatz der modellbasierten Testmethodik. Bisher ist unzureichend geklärt, auf welchem Abstraktionsniveau und mit welchen Abstraktionen Testmodelle erstellt werden sollen. Weiterhin erreichen Testmodelle trotz der verwendeten Vereinfachungen eine hohe Komplexität. Dies erfordert einen strukturierten Entwicklungsprozess, der eine korrekte Modellierung gegenüber den Anforderungen des zu testenden Systems sicherstellt. Überdies sind keine Arbeiten bekannt, die die Güte von modellbasierten Tests gegenüber traditionell entwickelten Tests vergleichen.

Diese Defizite behandeln wir in dieser Arbeit: Wir stellen einen inkrementellen Entwicklungsprozess zur Erstellung von Testmodellen bereit, der durch sein strukturiertes Vorgehen die Qualität gegenüber den Anforderungs- und Spezifikationsdokumenten sichert. Wir identifizieren geeignete Abstraktionstechniken, die integriert in den Entwicklungsprozess die Bildung von Testmodellen erleichtern. Weiterhin unterstützen wir den Reviewprozess von Testmodellen durch ein geeignetes Transformationsverfahren von Verhaltensmodellen. Abschließend demonstrieren wir die Konzepte anhand einer industriellen Fallstudie und vergleichen im Rahmen dieser Fallstudie modellbasierte Tests mit traditionell erstellten Tests bezüglich ihrer Abdeckung und Fehleraufdeckungsrate.

Danksagung

An erster Stelle möchte ich mich bei Professor Manfred Broy für die Schaffung einer einzigartigen und sehr anregenden Atmosphäre in seiner Forschergruppe, sowie für seine Förderung und die fachliche Betreuung meiner Arbeit bedanken. Herrn Professor Helmut Veith gebührt mein Dank für das Interesse an dieser Arbeit und für die Übernahme des Zweitgutachtens.

Alexander Pretschner bin ich zu großen Dank verpflichtet. Die intensiven Diskussionen im Rahmen unserer gemeinsamen Projektarbeit und auch nach seiner Zeit am Lehrstuhl waren stets kritisch und sehr hilfreich. Sie bildeten die Grundlagen dieser Arbeit. Ohne seine Unterstützung und den stetigen Ermutigungen wäre das Entstehen dieser Arbeit nicht möglich gewesen.

Für das kritische Lesen und Kommentieren einer vollständigen Version oder Teile dieser Arbeit danke ich herzlich Alexander Pretschner, Stefan Wagner und Frank Marschall. Für die zahlreichen und befruchtenden Diskussionen möchte ich Bernhard Schätz, Stefan Wagner, Jan Romberg, Peter Braun, Katharina Spies, Alexander Wißpeintner, sowie den ehemaligen Lehrstuhlmitgliedern Robert Sandner, Thomas Stauner, Jan Philipps, Ingolf Krüger und Bernhard Rumpe meinen Dank ausdrücken.

Für die ausgezeichnete und stets hilfsbereite Unterstützung bezüglich technischer Fragen bedanke ich mich beim Administrationsteam des Lehrstuhls und besonders bei Franz Huber. Für den reibungslosen organisatorischen Betrieb am Lehrstuhl gebührt bester Dank dem Sekretariat. Hier möchte ich Silke Müller hervorheben, die schwierigen Situationen stets mit professioneller und heiterer Gelassenheit begegnet und diese auf unkomplizierte Weise zügig löst.

Ganz besonders bin ich meiner Familie und meinen Freunden dankbar für ihre Geduld, ihre Aufmunterungen und ihren Beistand. Ich möchte aber diese Arbeit meinem Vater Wilhelm widmen, der kurz vor ihrer Fertigstellung verstorben ist und ihren Fortschritt bis zum Schluss verfolgte. Sein künstlerisches Wirken und seine Persönlichkeit prägten mich sehr. Seine Lebenserfahrung und seine Unterstützung werden mir sehr fehlen.

Inhaltsverzeichnis

1. Einleitung

Die rasante Entwicklung von informationstechnischen Systemen und ihre rasche Ausbreitung in alle Lebensbereiche erfordern fortwährend leistungsstärkere Methoden zur Systemerstellung und deren Qualitätssicherung. Diese Entwicklung lässt sich gerade im Automobilbau sehr deutlich verfolgen. Der Einzug von informationstechnischen Systemen im Automobil begann in den 70er Jahren mit der Einführung von Steuergeräten für die Benzineinspritzung mit dem Ziel, den Kraftstoffverbrauch zu reduzieren und gleichzeitig die Leistung zu steigern. Den nächsten Meilenstein stellte in den 80er Jahren die Einführung von programmierbaren Steuergeräten dar. Diese Geräte kamen z.B. für Systeme wie das Antiblockiersystem (ABS) oder die Antischlupfregelung (ASC) zum Einsatz. Bis zu diesem Zeitpunkt waren die Systeme weitgehend unabhängig voneinander und arbeiteten getrennt. In den 90er Jahren setzte verstärkt die Vernetzung der Systeme ein, die bis heute stetig zunimmt. Die Vernetzung ist mittlerweile einer der größten Komplexitätstreiber im Automobil. Heutige Premiumfahrzeuge enthalten bis zu 70 Steuergeräte, die über fünf verschiedene Bussysteme kommunizieren. Die Beobachtung, dass in Pannenstatistiken Softwarefehler als Ursache für liegen gebliebene Fahrzeuge stetig zunehmen, ist ein Indiz für die Notwendigkeit von leistungsfähigeren Methoden zu Qualitätssicherung.

Die Qualitätssicherung erfordert zum einen die präzise Spezifikation der erwünschten Funktionalität des Systems und zum anderen den Nachweis der Übereinstimmung zwischen dem fertig gestellten System und seiner Spezifikation. Eine in der Praxis weit verbreitete Form der Qualitätssicherung ist das Testen. Hier codieren Testfälle partiell das Sollverhalten des zu erstellenden Systems, indem sie neben den Eingaben für das System auch die erwarteten Ausgaben des Systems spezifizieren. Je nachdem ob nach Applikation der Eingaben an das System die Ausgaben des Systems –das Istverhalten– mit den erwartenden Ausgaben –dem Sollverhalten– übereinstimmt oder nicht, wird die Übereinstimmung bzw. das Abweichen des Istverhaltens gegenüber dem Sollverhalten nachgewiesen.

Eine viel versprechende Form der Testmethodik zur effektiven Qualitätssicherung ist das modellbasierte Testen, das die strukturierte, reproduzierbare und automatische Ableitung von Testfällen unterstützt. Basis für diese Methodik ist die Erstellung eines abstrakten und ausführbaren Modells und die Eingliederung in einen übergeordneten Entwicklungsprozess. Das Modell codiert vereinfacht das Sollverhalten des konkreten Systems. Mit Hilfe dieser Abstraktion wird die Überprüfung der Korrektheit des Modells durch manuelle Reviews von Simulationsläufen, des Modells selbst und ggf. durch formale Verifikation wesentlich erleichtert bzw. erst ermöglicht. Das Modell dient schließlich als

Quelle für Modellabläufe, die nach geeigneter Transformation die Testfälle für den Test des konkreten Systems darstellen.

1.1. Motivation und Ziele

Prenninger u.a. [PERH05] untersuchten acht aussagekräftige Fallstudien [DBG01, SA99, FKL99, PPS+03, CJRZ01, KVZ98, BFV+99, FHP02], um zu klären, auf welche Art und Weise modellbasiertes Testen im industriellen Kontext eingesetzt wird und welche Probleme und Schwierigkeiten dabei auftreten. Es wurden folgende Gemeinsamkeiten im Bezug auf Motivation und Prozess identifiziert:

- Die Anwendung von modellbasiertem Testen wird durch die Tatsache motiviert, dass die Komplexität von Informatiksystemen in der heutigen Zeit rapide zunimmt. Momentan werden die meisten Systeme in unstrukturierter und schwer nachvollziehbarer Weise getestet, indem Testingenieure auf Basis ihres impliziten Wissens über das System von Hand einzelne Testfälle entwickeln. Dieses Vorgehen und die steigende Komplexität verstärken zunehmend die Schwierigkeit, eine ausreichende Testabdeckung und Testtiefe auf effiziente Weise zu erreichen. Ein weiterer Punkt ist die stark anwachsende Größe der Testsuiten. Zum Beispiel beschreiben Fournier u.a. [FKL99], dass die Testsuite für die 32-Bit PowerPC-Architektur 87.000 Testfälle und für die 64-bit Architektur 150.000 Testfälle umfasst. Allgemein ausgedrückt stellt also die Überprüfung des Systems gegenüber seinen Anforderungen eine zunehmende Herausforderung dar und erfordert innovative Techniken wie zum Beispiel das modellbasierte Testen, um der steigenden Komplexität Herr zu werden.
- Alle Fallstudien folgen einem gemeinsamen abstrakten Prozess. Der Prozess basiert auf einem abstrakten Testmodell des Systemverhaltens, das lediglich die wesentlichen Aspekte des Systems auf abstrakte Weise berücksichtigt. Durch die so erreichte Komplexitätsreduktion im Testmodell wird dessen Überprüfung wesentlich erleichtert und kann gegenüber der direkten Überprüfung des konkreten Systems effizienter gestaltet werden. Auf Basis des Testmodells und der Selektionskriterien für Testfälle –die so genannten Testfallspezifikationen– werden Testfälle erzeugt, konkretisiert, ausgeführt und ausgewertet. Auf diese Weise wird das Verhalten des konkreten Systems gegenüber dem abstrakten Testmodell verifiziert. Dieser Ansatz führt zu einem strukturierten, reproduzierbaren und nachvollziehbaren Testprozess, der eine umfangreiche und messbare Testabdeckung und eine große Testtiefe ermöglicht.

Dagegen wurden bei der Untersuchung der Fallstudien Probleme und Schwierigkeiten bei der Testmodellerstellung und Defizite bei der kritischen Bewertung des Verfahrens festgestellt:

- Modellbasiertes Testen ist im Begriff, in der Industrie im Bereich der Prozessor-

verifikation weitläufig eingesetzt zu werden. Dies beruht auf der Tatsache, dass in dieser Domäne ein gut verstandener Entwicklungsprozess mit klar definierten Abstraktionsebenen (z.B. von VHDL zu RTL) existiert. Dies ermöglicht sogar eine teilweise automatisierte Ableitung von Testmodellen. In den anderen Fallstudien wurden mangels klar definierter Abstraktionsebenen die Testmodelle ad hoc und von Hand erzeugt und im Anschluss erfolgreich zum modellbasierten Testen eingesetzt. Folglich fehlt in vielen Domänen noch das Know-how, welche Abstraktionen bei der Entwicklung von Testmodellen eingesetzt und auf welchem Abstraktionsniveau sie entwickelt werden sollen.

- Testmodelle erreichen trotz ihrer Abstraktion bzw. ihrer Vereinfachung gegenüber der Realität eine sehr hohe Komplexität. Das wirft das Problem auf, wie sie entwickelt werden und wie die Validität gegenüber ihren Anforderungen sichergestellt wird.

- Keine der Fallstudien präsentiert Ergebnisse und kritische Bewertungen im Vergleich zu anderen Ansätzen. Sie enthalten keine Aussagen über ihre Effektivität und Kosten im Vergleich zu traditionellen Testmethoden oder anderen Qualitätssicherungsmethoden.

Insgesamt zeigten die Studien, dass sich bereits ein gemeinsamer modellbasierter Prozess durchgesetzt hat und die modellbasierte Technologie zur Generierung von Testfällen inzwischen ausgereift genug ist, im industriellen Kontext eingesetzt zu werden. Für den breiten Einsatz in der Praxis müssen jedoch allgemeine und domänenspezifische Abstraktionsprinzipien und strukturierte Entwicklungsmethoden für Testmodelle entwickelt werden, um den effektiven Einsatz des modellbasierten Testens zu unterstützen. Ferner ist eine strenge Untersuchung des modellbasierten Testens im Vergleich zu traditionellen Qualitätssicherungsmaßnahmen erforderlich.

Auf Basis dieser Beobachtungen identifizieren wir die Ziele dieser Arbeit:

- Bereitstellung eines einfachen und strukturierten Prozesses zur Entwicklung von ausführbaren Testmodellen, mit dessen Unterstützung der Entwickler schrittweise die Anforderungen an das zu testende System in das Testmodell integriert und sich dabei kontrolliert dem vollständigem Sollverhalten inkrementell annähert. Ferner muss der Entwicklungsprozess bis zur Komplexität von industriellen Systemen oder zumindest Teilsystemen skalieren.

- Identifikation von charakteristischen Abstraktionsmustern und -techniken, die es erlauben, für den Test unwesentliche Details wegzulassen, und die die Konzentration auf die wesentlichen Aspekte des zu testenden Systems unterstützen. Dabei ist wichtig, dass diese Abstraktionstechniken die wesentlichen Eigenschaften des zu testenden Systems im Testmodell erhalten und sie in den obigen Entwicklungsprozess integriert werden.

- Bereitstellung von Methoden und Techniken, die die Überprüfung des Testmodells gegenüber den Anforderungen des zu testenden Systems unterstützen und erleich-

tern.

- Vergleiche anzustellen, die die Qualität des modellbasierten Testens gegenüber traditionellen Testmethoden kritisch untersucht.

1.2. Beitrag

Die Hauptmotivation dieser Arbeit basiert auf der Beobachtung, dass für das modellbasierte Testen von Systemen die Korrektheit des abstrakten Verhaltensmodells eine zentrale Voraussetzung ist. Mit anderen Worten: Es ist von entscheidender Bedeutung, das Vertrauen zu gewinnen, dass das Ein-/Ausgabeverhalten des Modells korrekt gegenüber den informellen Anforderungen des Systems spezifiziert wurde.

In diesem Kontext liefern wir in dieser Arbeit Beiträge von methodischer und technischer Natur, die den Zielen aus Abschnitt 1.1 entsprechen:

- Wir definieren einen inkrementellen Entwicklungsprozess, welcher sich auf formale Grundlagen abstützt. Der Zweck dieses Entwicklungsprozesses ist einerseits, das inkrementelle Vorgehen auf formaler Basis zu begreifen, und anderseits, die Grundlagen zu schaffen, für diesen in Zukunft Werkzeugunterstützung entwickeln zu können. Im Gegensatz zu klassischen formalen Entwicklungsprozessen, die im wesentlichen auf Reduktion von nichtdeterministischem Verhalten beruhen, legt unser Ansatz den Schwerpunkt auf die natürliche Vorgehensweise, schrittweise das funktionale Verhalten zu erweitern anstatt zu reduzieren. Dieses Vorgehen unterstützt die intellektuelle Beherrschung des Verhaltensmodells und dies erhöht das Vertrauen, valide Modelle zu spezifizieren. Ein nützlicher Nebeneffekt dieser Vorgehensweise ist, dass systematisch Auslassungen, Mehrdeutigkeiten und Widersprüche in Spezifikationsdokumenten aufgedeckt werden und diese durch das Modell vervollständigt werden (vgl. Kapitel 5)
- Technische Unterstützung leisten wir, indem wir die Qualitätssicherung durch Reviews von Verhaltensmodellen erleichtern. Wir geben ein Verfahren an, das eine Zustandsmaschine (EFSM) zusammen mit einer gegebenen Partitionierung des Datenzustandsraums automatisch eine abstrahierte Sicht –einen Kontrollzustandsgraphen– berechnet. In diesem Graphen entspricht ein Kontrollzustand einer Partition und die Transitionen zwischen den Kontrollzuständen werden automatisch berechnet. Mit diesem Verfahren erhält der Entwickler die Möglichkeit, gezielt Aspekte einer Zustandsmaschine zu untersuchen, indem er mittels einer geeigneten Partitionierung die abstrahierte Sicht auf die Zustandsmaschine –den Kontrollzustandsgraphen– inspiziert (vgl. Kapitel 6). Ferner kann dieses Verfahren zum Refactoring von Zustandsmaschinen eingesetzt werden.
- Die Wahl eines geeigneten Abstraktionsniveaus ist entscheidend für die Beherrschbarkeit und Verwendbarkeit von Verhaltensmodellen. Dazu stellen wir in der Arbeit die Abstraktionsprinzipien funktionale, temporale, strukturelle, Daten- und

Kommunikationsabstraktion vor und zeigen, wie diese zur Modellbildung und im inkrementellen Entwicklungsprozess eingesetzt werden (vgl. Kapitel 3).

- Wir weisen die Anwendbarkeit der eingeführten Techniken anhand einer industriellen Fallstudie nach. Dabei liefern wir erste Zahlen, wie modellbasierte Tests gegenüber traditionell entwickelte Tests abschneiden und diskutieren kritisch den Nutzen der Modellbasierung und ihr Potential der Automatisierung (vgl. Abschnitt 7.5).

1.3. Umfeld

In diesem Abschnitt charakterisieren wir das Umfeld der Arbeit und erläutern die wesentlichen Entscheidungen für die Entwicklung unseres inkrementellen Ansatzes.

Methoden der Softwareentwicklung In den vergangenen Jahrzehnten entstanden eine Reihe von Softwareentwicklungsmethoden in unterschiedlichsten Fassetten, von denen wir hier einige herausgreifen wollen. Eher allgemeine Methoden stellen zum Beispiel das V-Modell [DW00], das die Aktivitäten und Produkte beschreibt, die während der Entwicklung von Software durchzuführen bzw. zu erstellen sind, die Cleanroom-Methode [PTLP98] mit Schwerpunkt auf inkrementeller Entwicklung unter ständiger und strenger Qualitätssicherung oder die dokumentenbasierte Methode von Denert [Den93, Den92] dar. Eine weitere Gruppe bilden Methoden, die auf objektorientierten Beschreibungstechniken wie z.B. der UML [OMG02] basieren. Hier nennen wir als Beispiele den Rational Unified Prozess (RUP) [Kru00], der eher auf die Entwicklung von Geschäftsanwendungen abzielt, und den Rapid Object-Oriented Process for Embedded Systems (ROPES) [Dou98], der für die Entwicklung von eingebetteten Echtzeitsystemen zugeschnitten ist. Abschließend identifizieren wir die Gruppe der so genannten agilen Methoden, die eine schnelle, kostengünstige und dokumentationsreduzierte Softwareentwicklung propagieren. Hier wären Extreme Programming (XP) [Bec00], Scrum [Sch04] und Crystal [CH04] als wichtigste Vertreter zu nennen.

Wie wir bereits angedeutet haben, hängen die Eignung und der Zweck der Entwicklungsmethoden von unterschiedlichen Faktoren wie Anwendungsdomäne, Projektgröße, Qualitätsanforderungen, Kosten etc. ab. Die Methoden bieten je nach Anwendungsgebiet eine Fülle von Richtlinien für den Entwurf und zur Entwicklung von Softwaresystemen.

Wir verstehen die Entwicklung von Testmodellen als eine Aktivität der Anforderungsanalyse, die in einen übergeordneten Entwicklungsprozess eingeordnet ist. Für die Entwicklung des Modells selbst entnehmen wir Prinzipien der inkrementellen Vorgehensweise aus der Cleanroom-Methode und den agilen Methoden und stützen diese auf formalen Grundlagen ab.

Entwicklungswerkzeuge Je nach Anwendungsgebiet und Entwicklungsphase gibt es eine Reihe von Entwicklungswerkzeugen, von denen wir eine Auswahl kurz vorstellen. Im Bereich des Anforderungsmanagements stellen DOORS [Tel02] und Rational Requisite [Sof04a] zwei wichtige Vertreter dar. Diese versprechen die Rückverfolgbarkeit und Strukturierung von Anforderungen mittels Hyperlinkstrukturen. Allzweckwerkzeuge auf Basis der UML für den Softwareentwurf sind Rational Suite [Sof04b], Telelogic TAU UML Suite [Tau02], Together [Bor04] und ArgoUML [Col04], sowie die Varianten für Echtzeitsysteme Rational Rose real-time [Ros02] und ARTiSAN Real-time Studio [Too01]. Speziell für die Entwicklung von eingebetteten Systeme sind die Werkzeuge ASCET-SD [ASC02], Stateflow [Bea02] und Betterstate [Bet02], die die Entwicklung von zeitdiskreten Systemen mit Hilfe von Strukturdiagrammen und einer Variante von statecharts [Har87] unterstützen, Matlab [Mat02b] und MatrixX [Mat02a], die die Entwicklung von sowohl zeitdiskreten als auch kontinuierlichen Systemen mit Hilfe von Blockdiagrammen aus der Regelungstechnik unterstützen, und VCC [VCC02] zu nennen, das sich auf die Analyse von Laufzeitverhalten und Leistungsaspekten von eingebetteten Systemen in Bezug auf konkreter Hardware spezialisiert hat.

Den aktuellen Stand der Technik von Werkzeugen zur Modellierung eingebetteten Systemen geben Schätz u.a. [SRS+03]. Ferner vergleichen Schätz u.a. [SHH+03] ausführlich diese Werkzeuge anhand einer Fallstudie aus der Automobilindustrie. Braun u.a. [BBC+02] geben einen Überblick über aktuelle und zukünftige Werkzeuge im Bezug auf die Entwicklung von eingebetteten Systemen in der Automobilindustrie.

Wir wählen zur Entwicklung von Testmodellen das Werkzeug AutoFocus [HSE97], das auf der formalen Modellierungssprache Focus [BS01] basiert. AutoFocus spezifiziert die Struktur von eingebetteten Systemen mit Hilfe von Systemstrukturdiagrammen (SSDs) und das Verhalten mit einer eingeschränkten Variante von statecharts (STD). AutoFocus vereint aus unserer Sicht die folgenden Vorteile für die Entwicklung von Testmodellen:

- AutoFocus basiert auf einer einfachen, verständlichen und formalen Semantik. Die Semantik der meisten Werkzeuge ist nur semiformal oder implizit durch Codegeneratoren gegeben. Werkzeuge mit einfacher Semantik erhöhen die Wahrscheinlichkeit, dass sie vom Nutzer richtig verstanden, korrekt angewendet werden, und das Vertrauen erhöhen, das Richtige zu modellieren (Validität).
- Das Werkzeug unterstützt die Testfallgenerierung [PLP01, Pre01]. In den meisten Werkzeugen wird –wenn überhaupt– die Durchführung von Tests unterstützt.
- Ferner wird die formale Verifikation von Modellen durch die Anbindung von Modelcheckern und Theorembeweisern [PS99] unterstützt.
- AutoFocus ist für Rapid-Prototyping konzipiert [HS97]. Damit vereint AutoFocus die Vorteile von Rapid-Prototyping durch Modellierung von Testmodellen in der Anforderungsanalyse mit der Analysierbarkeit von formalen Modellen und der Weiterverwendung des Prototyps zur Qualitätssicherung einer Implementierung mittels modellbasierten Testens.

Der inkrementelle Entwicklungsprozess, den wir für die Entwicklung von Testmodellen einführen (vgl. Abschnitt 2.3.2 und Kapitel 5), erfordert Anpassungen des Werkzeugs AutoFocus. Wir führen zur Entwicklung von Verhaltensmodellen eine leichter handhabbare Darstellungsform von STDs in Tabellenform ein (vgl. Abschnitt 4.5). Ferner modifizieren wir leicht die Semantik bzw. das Ausführungsmodell von AutoFocus, indem wir keine Idlevervollständigung der AutoFocus-STDs vorsehen und damit partielle Verhaltensmodelle zulassen (vgl. Abschnitt 7.2.5 und Kapitel 4).

Formale Modellierungssprachen Formale Modellierungssprachen umfassen eine Reihe von Methoden und Formalismen, die sich auf mathematische Grundlagen stützen und dadurch die Systementwicklung auf präzise Weise unterstützen. Als Beispiele für formale Ansätze nennen wir TLA [Lam94], eine Logik zur Spezifikation und Beweisführung über nebenläufige und reaktive Systeme, Unity [CM88], das aus einer abstrakten Programmiersprache mit Unterstützung von Nebenläufigkeit und einem Kalkül zur Beweisführung über diese Programme besteht, sowie algebraische Ansätze wie CCS [Mil89] und CSP [Hoa85], die zur Beschreibung von kommunizierenden Systemen entwickelt wurden und zugehörige Beweiswerkzeuge bereitstellen. Diese Sprachen sind meist nur auf ausgewählte Teilaspekte formaler Methoden spezialisiert und unterscheiden sich teilweise wesentlich in ihren zugrunde liegenden Kommunikations-, Ausführungs- und Zeitparadigmen etc. Die praktische Verwendbarkeit formaler Methoden ist nach wie vor Gegenstand von regen Diskussionen. Zur Zeit läuft ein deutschlandweites Projekt mit Namen Verisoft [PBR04], um die Einsetzbarkeit und die Grenzen von formalen Beweistechniken in der Breite und im industriellen Kontext zu untersuchen. Rushby u.a. [TSR03] propagieren eher den Ansatz der "unsichtbaren formalen Methoden", d.h. formale Methoden in Entwicklungswerkzeugen auf einfache Weise für gezielte Probleme transparent für den Nutzer einsetzbar zu machen.

Wir wählen Focus [BS01] und das zugehörige Werkzeug AutoFocus als Grundlage, um unsere Modellierungssprache und -methodik abzustützen. Focus vereint für unsere Bedürfnisse wichtige Konzepte und bietet folgende Vorteile:

- Focus bietet ein einheitliches mathematisches Grundmodell, auf das verschiedene Sichten eines Systems abgestützt werden können.
- Der klare Schnittstellenbegriff ermöglicht die deutliche Trennung zwischen einzelnen Komponenten und ihren Kommunikationsbeziehungen und führt zu einer klaren Unterscheidung zwischen System und Umgebung.
- Eine Folge des Schnittstellenbegriffs ist die Kompositionalität, die eine modulare Systementwicklung gestattet.
- Focus bietet einen geeigneten Beobachtungsbegriff auf das Schnittstellenverhalten von Systemen, der für die Domäne des modellbasierten Testens gut geeignet ist.
- Ferner ist für zeitsynchrones Focus Werkzeugunterstützung durch das Werkzeug AutoFocus vorhanden.

Faktoren der Systemerstellung Das Ziel bei der Systemerstellung ist, ein System unter Einbeziehung von begrenzten Mitteln und gemäß den Anforderungen des Auftraggebers zu entwickeln. Die konkurrierenden Faktoren bei der Systemerstellung im Rahmen eines Projekts sind (1) die Projektlaufzeit, (2) die Projektkosten und (3) die Qualität, die sowohl die Formulierung der gewünschten Systemfunktionalität als auch die zugehörigen Qualitätssicherungsmaßnahmen einschließt. In diesem Umfeld liefert die Arbeit einen Beitrag, die Qualitätssicherung mittels Testen durch die Methodik des modellbasierten Testens zu verbessern. Erste Zahlen bezüglich Fehleraufdeckungsrate und Bedingungsabdeckung wurden in einem Pilotprojekt ermittelt und deuten auf eine Verbesserung gegenüber traditionellen Testmethoden hin (vgl. Abschnitt 7.5). Der modellbasierte Testprozess ist in einen übergeordneten Entwicklungsprozess eingegliedert. Durch den Mehraufwand der Testmodellentwicklung in der Anforderungsanalyse steigen die Kosten und der Zeitaufwand in dieser Phase. Ob dieser erhöhte Aufwand durch die erwarteten Einsparungen in den späteren Projektphasen ausgeglichen wird oder gar zu Aufwandseinsparungen führt, kann mit dieser Arbeit nicht beantwortet werden. Die dazu erforderlichen umfassenden empirischen Untersuchungen würden den Rahmen dieser Dissertation sprengen.

1.4. Aufbau der Arbeit

Kapitel 2, Grundlagen, führt grundlegende Begriffe und Vorgehensweisen ein, die den Rahmen dieser Arbeit bilden. Abschnitt 2.1 beginnt mit allgemeinen Begriffen wie Abstraktion und Modell, reaktive bzw. transformative Systeme, Verifikation und Validation sowie Fehlverhalten und Fehlerursache. Abschnitt 2.2 beschäftigt sich ausführlich mit dem Begriff des modellbasierten Testens und führt die zugehörige Terminologie ein. Die Basis des modellbasierten Testens bildet das Testmodell und die zugehörigen Testfallspezifikationen. Die Anforderungen und Eigenschaften von Testmodellen werden ausführlicher diskutiert, da deren inkrementelle Entwicklung im Zentrum dieser Arbeit stehen. Abschnitt 2.3 charakterisiert die Eigenschaften eines inkrementellen Entwicklungsprozesses und zeigt dessen Vorteile anhand der Cleanroom-Methode auf. Wir übertragen die wesentlichen Merkmale der Cleanroom-Methode auf einen inkrementellen Entwicklungsprozess zur Erstellung von Testmodellen. Dieser zerfällt in die drei Phasen *Planung*, *Spezifikation der Schnittstelle* und *inkrementelle Entwicklung des Verhaltensmodells.*

Kapitel 3, Abstraktionen in Testmodellen, erläutert zunächst in Abschnitt 3.1 wesentliche Merkmale von Abstraktionen bei der Modellbildung und führt die Abstraktionsklassifikation *Abstraktion durch Kapselung von Details* und *Abstraktion durch Weglassen von Details* ein. Bei der ersteren kann der Informationsverlust durch Compiler, Linker o.ä. ausgeglichen werden, bei der zweiteren nicht. Abschnitt 3.2 beinhaltet den ersten Beitrag dieser Arbeit. Es werden die Abstraktionsarten, die bei der Entwicklung von Testmodellen verwendet werden, klassifiziert und mit Beispielen aus der Literatur belegt. Wir unterscheiden die Abstraktionsarten *funktionale Abstraktion*, *Datenabstrak-*

tion, *Kommunikationsabstraktion*, *temporale Abstraktion* und *strukturelle Abstraktion*. Abschnitt 3.3 diskutiert die Grenzen beim Einsatz von Abstraktion bei der Entwicklung von Testmodellen. Abschließend erläutert Abschnitt 3.4 an welcher Stelle welche Abstraktionsprinzipien im inkrementellen Entwicklungsprozess von Testmodellen eingesetzt werden.

Kapitel 4, Formale Grundlagen, legt das formale Fundament für Kapitel 5, um den inkrementellen Entwicklungsprozess für Testmodelle formal fassen und beschreiben zu können. Abschnitt 4.1 führt grundlegende Begriffe und Definitionen zur Modellierungssprache FOCUS ein und definiert den zentralen Begriff des Modellverhaltens, der alle Ein-/Ausgabeabläufe an der Schnittstelle eines Modells umfasst, und später der Menge aller Testfälle entspricht, die ein Testmodell potenziell erzeugen kann. Abschnitt 4.2, 4.3 und 4.4 führen Zustandsmaschinen zur operationellen Beschreibung von Modellverhalten ein, definiert die Komposition auf Modellverhalten und Zustandsmaschinen bzw. führt Regelsysteme ein, die Zustandsmaschinen mit einer endlichen Regelmenge beschreiben. Abschnitt 4.5 und 4.6 beschreiben, wie Tabellen bzw. Kontrollzustandsgraphen zur Repräsentation von Regelsystemen verwendet werden. Tabellen bieten eine detaillierte Sicht und konzentrieren sich auf die Verhaltensregeln. Kontrollzustandsgraphen dagegen bieten eine abstrahierte Sicht und konzentrieren sich auf den abstrakten Kontrollfluss. Abschnitt 4.7 diskutiert die Vor- und Nachteile der verschiedenen Darstellungsformen im methodischen Einsatz, sowie verwandte Arbeiten.

Kapitel 5, Inkrementelle Entwicklung von Testmodellen, liefert einen der Kernbeiträge dieser Arbeit. Nach einer kurzen Wiederholung der Anforderungen an einen inkrementellen Entwicklungsprozess für Testmodelle in Abschnitt 5.1 wird in Abschnitt 5.2 ein formaler Inkrementbegriff auf Ebene des Modellverhaltens definiert. Es wird unterschieden zwischen der *Inkrementbildung unter Schnittstellenerweiterung* und der *Inkrementbildung unter Verhaltenserweiterung*. Die erstere dient zur Vorbereitung der zweiteren. In Abschnitt 5.3 wird der zugehörige inkrementelle Entwicklungsprozess erläutert. Seine wesentliche Eigenschaft ist, dass die vom Entwickler spezifizierten existenziellen Eigenschaften über den Entwicklungsprozess erhalten bleiben. In Abschnitt 5.4 wird der vorliegende Prozess mit Entwicklungsprozessen verglichen, die auf klassischer Verhaltensverfeinerung beruhen. Abschnitt 5.5 ist das Kernstück dieses Kapitels und zeigt, welche Operationen auf Regelsystemen zur Inkrementbildung verwendet werden. Dabei wird zwischen Operationen zum *Vorbereiten einer Verhaltenserweiterung*, zur *Korrektur des Verhaltens* und zur *Verhaltenserweiterung des Modells* unterschieden.

Kapitel 6, Transformation von Regelsystemen, stellt einen weiteren Beitrag der Arbeit dar und zeigt ein Verfahren zum Refactoring von Regelsystemen. Refactoringoperationen sind bei der Inkrementbildung eine wichtige Klasse von Operationen zur Vorbereitung einer Verhaltenserweiterung. Abschnitt 6.1 führt die formale Definition des Transformationsverfahrens ein und zeigt, dass das Modellverhalten unter dieser Transformation

erhalten bleibt. Abschnitt 6.2 erläutert, wie die Transformation methodisch im inkrementellen Entwicklungsprozess auf verschiedenste Weise eingesetzt werden kann. Abschließend zeigt Abschnitt 6.3 auf, wie das Verfahren in einem Entwicklungswerkzeug umgesetzt werden kann.

Kapitel 7, Anwendung in der Praxis, demonstriert alle eingeführten Entwicklungstechniken an einer industriellen Fallstudie und zeigt dessen Vorteile auf. Abschnitt 7.1 führt die Fallstudie, den Netzwerkcontroller im MOST-Netzwerk ein. Abschnitt 7.3 beschreibt das Entwicklungswerkzeug AutoFocus, mit dem das Testmodell zum Netzwerkcontroller entwickelt wird. In Abschnitt 7.3 wird detailliert der inkrementelle Verhaltensaufbau des Testmodells gemäß dem Prozess aus Kapitel 5 erläutert. In Abschnitt 7.4 wird das Transformationsverfahren aus Kapitel 6 auf die Fallstudie angewendet und dessen Nutzen aufgezeigt. Abschnitt 7.5 beinhaltet ein weiteres Kernstück der Arbeit, indem erste Zahlen zum Vergleich von modellbasierten zu traditionell erstellten Tests angegeben werden.

Kapitel 8, Zusammenfassung und Ausblick, schließt die Arbeit mit einer Zusammenfassung aller Ergebnisse und gibt einen Ausblick auf zukünftige Arbeiten.

Anhang A, Beweise, enthält die Beweise zu den Propositionen aus den Kapiteln 4, 5 und 6.

Anhang B, Fallstudie, enthält ergänzendes Material zu der industriellen Fallstudie aus Kapitel 7.

2. Grundlagen

In diesem Kapitel führen wir die grundlegenden Begriffe und Vorgehensweisen ein, die den Rahmen dieser Arbeit bilden und zum Großteil in den folgenden Kapiteln vertieft werden. Abschnitt 2.1 diskutiert kurz die zentralen Begriffe wie Abstraktion, Modell, reaktive Systeme, Verifikation und Validation sowie Fehler. Anschließend wird in Abschnitt 2.2 die Terminologie im Bereich Testen und das Prinzip des modellbasierten Testens erläutert. Die Basis des modellbasierten Testens bilden Testmodelle, deren Eigenschaften und Anforderungen der Gegenstand von Abschnitt 2.2.1 sind. Abschnitt 2.3 skizziert das Vorgehen bei der inkrementellen Entwicklung von Testmodellen und Abschnitt 2.4 schließt mit einer kurzen Zusammenfassung des Kapitels.

2.1. Grundlegende Begriffe

In diesem Abschnitt führen wir eine Reihe grundlegender Begriffe ein, welche wir im Laufe der Arbeit häufig verwenden. In den einzelnen Abschnitten verweisen wir auf weiterführende Literatur.

Abstraktion und Modell Modellbildung ist wohl die wichtigste Technik, die es in der Informatik bzw. in der Wissenschaft allgemein gibt. In Anlehnung an die Definition von Stachowiak [Sta73] identifizieren wir drei wesentliche Merkmale von Modellen:

Bezug zur Realität Ein Modell bezieht sich auf einen Gegenstand der Realität, der bereits existiert oder existieren wird.

Abstraktion Ein Modell stellt den modellierten Gegenstand vereinfacht dar, indem von unwesentlichen Details abstrahiert wird.

Zweck Ein Modell ist an einen Zweck gebunden.

Im Allgemeinen legt der Zweck des Modells fest, welche Abstraktionsprinzipien bzw. Vereinfachungen bei der Modellbildung gegenüber der Realität vorgenommen werden können. Zum Beispiel wird durch den Zweck die Erhaltung ausgewählter Systemeigenschaften im Modell gegenüber der Realität motiviert. Folglich dürfen bei der Modellbildung lediglich Abstraktionstechniken angewendet werden, die den Erhalt von wesentlichen Eigenschaften sicherstellen. In dieser Arbeit betrachten wir Verhaltensmodelle von reaktiven Systemen, die verwendet werden, um das Schnittstellenverhalten von realen Systemen zu verifizieren. In diesem Zusammenhang steht der Erhalt der kausalen

Abhängigkeit von Nachrichten bzw. Ereignissen vom realen System und seinem Modell im Vordergrund.

Reaktive Systeme Wir unterscheiden reaktive Systeme von transformativen Systemen. Transformative Systeme starten mit der Eingabe eines Eingabedatums und terminieren ihre Berechnung mit der Ausgabe eines Ausgabedatums. Reaktive Systeme dagegen terminieren in allgemeinen nicht und bleiben reaktiv, indem ein stetiger Nachrichtenempfang aus der Umgebung des Systems möglich ist. Der Empfang von Nachrichten verändert u.U. den internen Zustand und führt ggf. zu einer Ausgabe, die wiederum die Umgebung des Systems beeinflusst. Die Reaktion von einer Eingabe hängt meist von dem internen Zustand und der Vorgeschichte der Eingabe ab. Zusammenfassend können wir reaktive Systeme als Verallgemeinerung von transformativen Systemen darstellen: transformative System bilden ein Eingabedatum unter Terminierung auf ein Ausgabedatum ab. Reaktive Systeme dagegen bilden potentiell unendlich lange Sequenzen von Eingabedaten auf Sequenzen von Ausgabedaten ab. Sie können weiter in eingebettete Systeme und Geschäftsanwendungen unterschieden werden, wobei der Übergang zwischen beiden fließend ist. Beispiele für eingebettete Systeme sind etwa die Airbagsteuerung oder die Motorsteuerung im Automobil, für Geschäftsanwendungen das Buchungssystem einer Fluggesellschaft oder ein Webserver.

Wir modellieren in dieser Arbeit reaktive Systeme mit Hilfe des Werkzeugs AutoFocus [HSE97], wobei wir für die Beschreibung des Modellverhaltens so genannte Regelsysteme (vgl. Kapitel 4) einführen.

Verifikation und Validation In dieser Arbeit verwenden wir die Begriffe Verifikation und Validation gemäß etablierter Standards. Die IEEE definiert im Standard 1012-1998 [IEE98] Verifikation als "confirmation by examination and provisions of objective evidence that specified requirements have been fullfilled" und Validation als "confirmation by examination and provisions of objective evidence that the paticular requirement for a specific intended use are fullfilled". Ähnlich zur IEEE versteht Balzert [Bal96] unter Verifikation die Überprüfung der Übereinstimmung zwischen einem Software-Produkt und seiner Spezifikation und unter Validation die Eignung bzw. Wert eines Produkts bezogen auf den Einsatzzweck. Ein Produkt wird also gegenüber seiner Spezifikation verifiziert und gegenüber den Anforderungen eines Nutzers auf einen Einsatzzweck hin validiert. Insbesondere wird offen gelassen, welche Techniken dazu verwendet werden. Zum Beispiel wird oft unter Verifikation der formale Nachweis der Korrektheit des Produkts gegenüber seiner formalen Spezifikation verstanden. Die Definition schließt aber auch nicht formale Techniken ein, bei denen z.B. durch Inspektionen oder Reviews entschieden wird, ob ein Produkt seiner Spezifikation genügt.

Der Schwerpunkt dieser Arbeit liegt auf der Bereitstellung von Entwicklungstechniken und -methodiken von so genannten Testmodellen (siehe Abschnitt 2.2.1). Diese Modelle charakterisieren eindeutig das Ein-/Ausgabeverhalten eines zu testenden Systems.

Einerseits werden die Testmodelle gegen die meist informellen und unvollständigen Spezifikationsdokumente des Systems verifiziert. Andererseits werden Vervollständigungen und Ergänzungen der Spezifikation, die bei der Entwicklung des Testmodells stattfinden, auf ihre Adäquatheit hin validiert.

Fehler In der Literatur unterscheiden u.a. Breitling [Bre01] und Lyu [Lyu96] zwischen Fehlverhalten, Fehlerursache und Fehlerzustand. Ein Fehlverhalten oder ein Versagen (engl. *failure*) des Systems liegt vor, wenn das Istverhalten des Systems nicht mit seinem Sollverhalten übereinstimmt. Dieses kann also nur zur Laufzeit des Systems beobachtet werden. Es wird von einem Fehlerzustand (engl. *error*) gesprochen, wenn der Zustand des Systems von seinem gewünschten Zustand abweicht. Ein Fehlerzustand führt nicht zwangsweise zu einem Fehlverhalten. Die Fehlerursache (engl. *fault*) ist der Grund, der zum Auftreten des Fehlerzustands oder Fehlverhaltens führt. Musa u.a. [MIO87, Mus99] verwenden eine einfachere Terminologie und unterscheiden lediglich zwischen Fehlverhalten (engl. *failure*) und Fehlerursache (engl. *fault*). Für unsere Arbeit genügt diese einfachere Unterscheidung.

Qualitätssicherungsmaßnahmen wie zum Beispiel Inspektionen und Reviews können direkt Fehlerursachen aufdecken, ohne dass zuvor ein Fehlverhalten beobachtet wurde. Das Testen dagegen kann lediglich Fehlverhalten des Systems gegenüber seinem Sollverhalten spezifiziert durch den Test aufdecken. Erst nach dem Aufdecken wird die Fehlerursache mit Hilfe von Analyse- bzw. Diagnosetechniken identifiziert. Alle Aktivitäten nach der Aufdeckung von Fehlverhalten, wie Fehlerdiagnose oder Fehlerbehebung sind nicht Gegenstand dieser Arbeit. Da sich die Arbeit mit dem modellbasierten Black-Box-Test von reaktiven Systemen beschäftigt, verwenden wir meist vereinfacht den Begriff Fehler anstatt Fehlverhalten.

2.2. Modellbasiertes Testen

In diesem Abschnitt führen wir den Begriff Testen, das Prinzip des modellbasierten Testens und alle benötigten Begriffe in diesem Kontext ein. Unsere Ontologie stützt sich dabei auf die Arbeiten von Pretschner, Leucker und Lötzbeyer [Pre03b, BJK+05, Löt03].

Testen Unter Testen verstehen wir Aktivitäten mit dem Ziel, die Übereinstimmung oder Abweichung von Ist- und Sollverhalten eines Systems nachzuweisen. Dabei bezeichnen wir die Übereinstimmung und Abweichung als Konformität zwischen Spezifikation und Implementierung bzw. als Fehler in der Implementierung. Diese weit verbreitete Definition ist eine Verallgemeinerung der Definition von Myers [Mye89], der unter Testen die Tätigkeit der Programmausführung mit dem Zweck Fehler zu finden versteht. Das Sollverhalten ist häufig in Form von informellen und unvollständigen Spezifikationsdokumenten gegeben. Die Schwierigkeit besteht nun darin auf Basis dieser Dokumente strukturierte und nachvollziehbare Verfahren zu entwickeln, die das Testen auf effiziente

Weise unterstützen. Man unterscheidet beim systematischen Test, wie u.a. von Wegener [Weg01] ausführlich beschrieben, folgende Aktivitäten: Die *Testplanung* legt das zu testende System, dessen Systemgrenze, dessen Umgebung, das Testziel und das Testendekriterium fest. Die *Testorganisation* befasst sich im wesentlichen die Verwaltung von Testfällen und zu testender Systeme. Die *Testdokumentation* dient zur Sicherstellung der Nachvollziehbarkeit und Reproduzierbarkeit aller Aktivitäten. Bei der *Testfallermittlung* werden auf Basis eines vergebenen Testziels Testfälle abgeleitet. Diese Aktivität ist die wichtigste, weil damit die Art und Umfang der Prüfung des zu testenden Systems festgelegt wird. Bei der *Testausführung* werden die Testfälle mit dem zu testenden System zur Aufführung gebracht, indem mit dem Eingabeteil der Testfälle das System stimuliert wird. Beim *Monitoring* wird das Verhalten des zu testenden Systems bei der Testausführung aufgezeichnet und dient als Grundlage für den Vergleich mit dem Sollverhalten. Der Vergleich zwischen Ist- und Sollverhalten findet bei der *Testauswertung* statt und deckt damit Unterschiede und Übereinstimmung zwischen Ist- und Sollverhalten auf.

Prinzip des Modellbasierten Testens Das Grundprinzip des modellbasierten Testen ist der Vergleich des Schnittstellenverhaltens bzw. Ein-/Ausgabeverhaltens eines als valide betrachteten Verhaltensmodells mit dem einem *System unter Test* (SUT). Dabei wird die Konformität zwischen dem Sollverhalten, codiert durch das Verhaltensmodell, und dem Istverhalten, repräsentiert durch das System unter Test, überprüft. Um dies zu bewerkstelligen, wird eine endliche Menge –die so genannte *Testsuite*– von Ein-/Ausgabeabläufen endlicher Länge des Modells –die so genannten *Testfälle*– ausgewählt. Das Auswahlkriterium wird charakterisiert durch ein *Testziel* oder durch eine *Testfallspezifikation*. Der Eingabeteil eines Testfalls –die so genannten *Testdaten*– wird dem SUT injiziert und die Ausgabe des SUT wird mit der erwarteten Ausgabe gegeben durch den Ausgabeteil des Testfalls verglichen. Stimmt Ausgabe der SUT mit der erwarteten Ausgabe überein, dann ist das SUT konform mit dem Modell bezüglich dem ausgeführten Testfall. Andernfalls liegt Nichtkonformität bzw. ein Fehler im Modell oder im SUT vor.

Das vorgestellte Prinzip des modellbasierte Testens gilt, wenn das Verhaltensmodell des SUT zur Ableitung von Testfällen deterministisch ist. Bei nichtdeterministischen Verhaltensmodellen muss die einfache Ablaufstruktur von Testfällen zu komplexen Baumstrukturen verallgemeinert werden, bei denen jede Verzweigung einer möglichen nichtdeterministischen Reaktion entspricht. Da wir uns in dieser Arbeit auf deterministische Verhaltensmodelle beschränken, werden Testfälle und Ein-/Ausgabeabläufe endlicher Länge als Synonym betrachtet.

Prozess des modellbasierten Testens Prenninger u.a. [PERH05] untersuchten eine Reihe von Fallstudien im industriellen Kontext aus den Bereichen des Prozessortests, Smart-Card-Tests, Protokolltests und Test von Betriebssystemteilen und Teilen des Java-Standards. Dabei stellte sich heraus, dass allen Fallstudien ein gemeinsamer modellbasierter Testprozess unterliegt (siehe Abbildung 2.1).

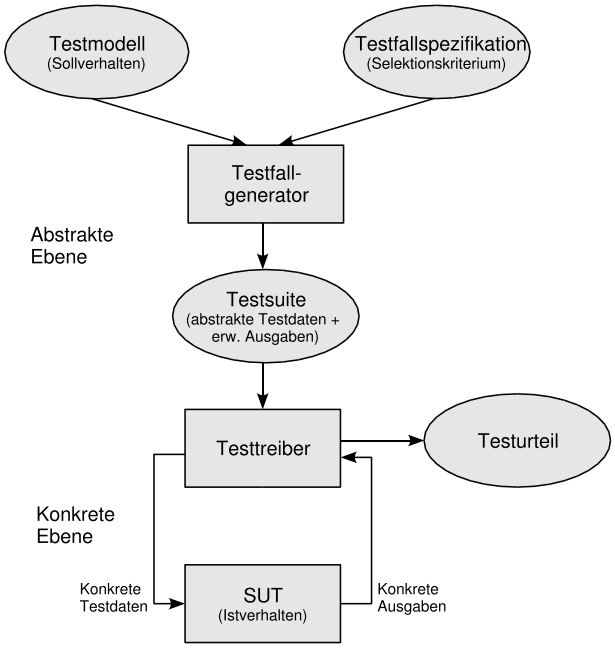


Abbildung 2.1.: Prozess des modellbasierten Testens

Grundlage für den modellbasierten Test ist ein abstraktes Verhaltensmodell des SUT. Dieses Verhaltensmodell bezeichnen wir im folgenden als *Testmodell*. Dieses Testmodell ist ausführbar bzw. simulierbar und codiert in abstrakter Weise das Sollverhalten an der Schnittstelle des SUT. Das Testmodell codiert im Allgemeinen eine unendlich große Menge von Testfällen unendlicher Länge. Da das Testen des SUT-Verhaltens in endlicher Zeit erfolgen muss, werden Testfallspezifikationen verwendet, um eine endliche Menge von Testfälle endlicher Länge zu selektieren. Eine geeignete Auswahl an "guten" Testfällen zu treffen, ist eine anspruchsvolle Aufgabe. Der *Testfallgenerator* generiert aus dem Testmodell und einer Testfallspezifikation eine *Testsuite*. Die Testsuite besteht aus einen oder mehreren Ein-/Ausgabeabläufen (Testfällen) des Testmodells und erfüllt die Testfallspezifikation. Der *Testtreiber* erhält als Eingabe einen Testfall oder eine ganze Testsuite und hat die Aufgabe, diese gegen das SUT auszuführen und den Test auszuwerten. Da die erzeugten Testfälle ebenso wie die Testmodell abstrakt gegenüber dem SUT sind, muss der Testtreiber die Abstraktionslücke zwischen dem Testfall und dem SUT überbrücken. Dies geschieht, indem vor oder bei Ausführung des Testfalls die abstrakten *Testdaten* des Testfalls zu konkreten Eingaben für das SUT konkretisiert werden und diesem injiziert werden. Zur Auswertung des Testfalls werden hingegen die Ausgaben des SUT abstrahiert mit den erwarteten Ausgaben des Testfalls verglichen. Abschließend gibt der Testtreiber das Ergebnis des Tests in Form eines *Testurteils* aus. Das Testurteil wird zu *passiert* (engl. *pass*) ausgewertet, falls das Verhalten des Testmodells mit dem des SUT bezüglich eines Testfalls bzw. einer Testsuite übereinstimmt. Andernfalls wird es zu *fehlgeschlagen* (engl. *fail*) ausgewertet. Wird das Testmodell als fehlerfreie Spezifi-

kation des SUT angesehen, dann wurde ein Fehlverhalten in dem SUT aufgedeckt. Wird dagegen das Testmodell als abstrakte Implementierung redundant gegenüber des SUT aufgefasst, dann wird nach Analyse der Verhaltensabweichung der Fehler entweder im Testmodell oder in dem SUT identifiziert. Eine ausführliche Diskussion über den Einsatz von Verhaltensmodellen im Entwicklungsprozess zum Test oder zur Codeerzeugung von reaktiven Systemen erfolgte in der Dissertation von Pretschner [Pre03b].

Der Schwerpunkt dieser Arbeit liegt auf der Entwicklung von Testmodellen. Dabei leisten wir einen Beitrag, indem wir die Abstraktionstechniken, die zur Testmodellbildung eingesetzt werden, klassifizieren, diese in den Entwicklungsprozess integrieren (Kapitel 3) und einen inkrementellen Entwicklungsprozess entwickeln, der schrittweise unscharfe und unvollständige funktionale Anforderungen an das SUT eindeutig und vervollständigt in ein ausführbares Testmodell integriert (Kapitel 5).

Zusammenfassung der eingeführten Begriffe

- **System unter Test (SUT)** ist das System, dessen Schnittstellenverhalten mittels Testen verifiziert werden soll.
- **Testmodell** ist ein ausführbares bzw. simulierbares Verhaltensmodell, dass das Schnittstellenverhalten des SUT (oder Teile davon) abstrakt modelliert.
- **Testfall** ist eine Struktur, die einen Teil des intendierten Ein-/Ausgabeverhaltens codiert und in endlicher Zeit ausgeführt werden kann.
- **Testdaten** bezeichnen den Eingabeteil eines Testfalls.
- **Testsuite** ist eine endliche Menge von Testfällen endlicher Länge.
- **Testziel** ist eine u.U. informelle Beschreibung einer Testsuite.
- **Testfallspezifikation** ist eine formale Spezifikation einer Testsuite und operationalisierbar in Hinblick auf einen Testfallgenerator, d.h. ein Testfallgenerator kann die Testfallspezifikation zusammen mit dem Testmodell zur Ausführung bringen, um eine Testsuite zu erzeugen.
- **Testfallgenerator** ist ein Programm, das aus einem Testmodell und einer Testfallspezifikation eine Testsuite berechnet.
- **Testtreiber** ist ein Programm, das einen Testfall oder eine Testsuite gegen das SUT ausführt und nach Ausführung des Tests ein Testurteil berechnet. Dabei muss der Testtreiber die Abstraktionslücke zwischen dem Testmodell und dem SUT überbrücken.
- **Testurteil** ist das Ergebnis des Vergleichs zwischen erwarteten und aufgetretenen Verhalten. Ein Testurteil wird zu *passiert* (engl. *pass*) ausgewertet, falls das Verhalten konform zueinander ist. Andernfalls wird es zu *fehlgeschlagen* (engl. *fail*) ausgewertet.

2.2.1. Testmodelle

Der Schwerpunkt der Arbeit liegt auf einer Methodik zur Entwicklung von Testmodellen. Testmodelle haben den Zweck, den Validations- und Verifikationsprozess des zu entwickelnden Systems zu unterstützen. Einerseits dient das Testmodell zur Formalisierung, Vervollständigung und Validation der Anforderungen des Systems. Anderseits dient das Testmodell als Spezifikation und wird verwendet, um die Systemverhalten einer Implementierung mittels Test zu verifizieren. In diesem Zusammenhang können Testmodelle als ausführbare Anforderungsspezifikationen und abstrakte Prototypen angesehen werden, die später im Entwicklungsprozess zur Qualitätssicherung mittels Test weiterverwendet werden. Wir betrachten in dieser Arbeit aufführbare Testmodelle, da der Review von Modellabläufen ein wesentliches Hilfsmittel ist, die Korrektheit des Modells gegenüber funktionaler Anforderungen zu überprüfen. Ferner erleichtert die Ausführbarkeit die automatische Erzeugung von Testfällen.

In diesem Abschnitt geben wir einen Überblick, welche grundlegenden Eigenschaften Testmodelle haben und welchen methodischen Nutzen sie im Entwicklungsprozess von reaktiven Systemen bringen.

Anforderungen an das Testmodell Neben der technischen Anforderung, dass ein Testmodell operationalisierbar ist, d.h. dass es mit Hilfe eines Testfallgenerators (siehe Pretschner und Lötzbeyer [Pre03b, Löt03]) zur Testfallgenerierung genutzt werden kann, ergeben sich eine Reihe von methodischen Anforderungen an ein Testmodell. Da die aus dem Testmodell erzeugten Testfälle die Grundlage für die Verifikation des SUT bilden, ist es von größter Wichtigkeit, dass das Testmodell das Schnittstellenverhalten korrekt und vollständig gegenüber den Nutzeranforderungen bzw. den Spezifikationsdokumenten modelliert. Diesem Umstand tragen wir Rechnung, indem wir die intellektuelle Beherrschbarkeit von Testmodellen erleichtern.

Wir verwenden eine Beschreibungstechnik mit einfacher Semantik, die einerseits geeignet ist reaktive Systeme in operationeller Weise zu modellieren und anderseits einen geeigneten Beobachtungsbegriff bietet, um das Schnittstellenverhalten eines Systems zu testen. Diese Festlegung kommt der Beobachtung entgegen, dass Sprachen mit einer einfachen Semantik vom Nutzer leichter und richtig verstanden und demzufolge richtig eingesetzt werden. Dementsprechend führen wir in Kapitel 4 Beschreibungstechniken ein, die in das einfache Ausführungsmodell von AutoFocus eingebettet bzw. an zeitsynchrones Focus angelehnt sind.

Weiterhin fordern wir, dass Testmodelle im folgendem Sinne deterministisch sind: identische Eingabesequenzen liefern genau eine Ausgabesequenz auf Modellebene. Einerseits werden wir damit der Forderung von Heimdahl u.a. [HL96, THM99] gerecht, dass es bei sicherheitskritischen Systemen nicht wünschenswert ist, unterspezifiziertes Verhalten offen zu lassen, und dass bereits früh zur Spezifikationszeit die Nutzer, Anforderungsanalytiker, Reviewer etc. die Entscheidungen über akzeptables Verhalten treffen anstatt

dies später im Entwicklungsprozess weniger qualifiziertem Personal zu überlassen. Andererseits stellt diese Einschränkung einen wesentlichen Beitrag dar, die intellektuelle Beherrschung des Testmodells zu erleichtern: Es ist wesentlich einfacher zu überprüfen, ob ein spezifisches deterministisches Verhalten korrekt ist, als nachzuweisen, ob alle möglichen nichtdeterministischen Verhalten adäquat sind. Das bedeutet jedoch nicht, dass nichtdeterministisches Verhalten des SUT nicht getestet werden kann. Durch den Testtreiber kann mit einem abstrakten Testfall eine ganze Menge von Systemabläufen des SUT getestet werden, indem z.B. die Testdaten in unterschiedlichen Zeitabständen an das SUT gesendet werden und spezielle Vergleichsoperatoren verwendet werden, um die nichtdeterministischen Ausgaben des SUT mit der deterministischen Ausgabe des Testmodells zu vergleichen.

Vollständigkeit des Testmodells Am Ende seiner Entwicklung repräsentiert das Testmodell eine abstrakte Referenzimplementierung des Sollverhaltens, die alle oder einen Teil der funktionalen Anforderungen an das Schnittstellenverhalten des zu testenden Systems eindeutig und vollständig codiert. Vollständigkeit bezieht sich jedoch lediglich auf den betrachteten Teil der Anforderungen und führt im Allgemeinen nicht zur so genannten Eingabevollständigkeit auf Modellebene.

Testmodelle haben den Zweck, die Menge aller potenziellen Testfälle in Form von Ein-/Ausgabeabläufen zu codieren. Folglich soll die durch das Testmodell induzierte Ablaufmenge keine Abläufe enthalten, für die es auf Ebene des SUT keine entsprechenden gibt. Die Wahl der Schnittstelle des Testmodells auf Modellebene erlaubt jedoch im Allgemeinen Eingabesequenzen, für die es auf SUT-Ebene keine entsprechenden gibt. In diesem Fall ist es auf Modellebene keine Ausgabesequenz zu dieser Eingabe vorgesehen, d.h. Testmodelle sind im Allgemeinen nicht eingabevollständig. Zur Veranschaulichung skizzieren wir folgendes Beispiel: auf Testmodellebene gibt es je einen Eingabekanal für Daten- und Kontrollnachrichten, um diese auf Modellebene zwecks Verständlichkeit und Analysierbarkeit des Modells konzeptuell zu unterscheiden. Dadurch ist es auf Modellebene möglich, dass Daten- und Kontrollnachrichten gleichzeitig empfangen werden. Auf SUT-Ebene ist jedoch aufgrund der technischen Infrastruktur nur ein sequenzieller Empfang möglich. In dieser Situation gibt es also für den Test keine sinnvolle Ausgabe. Folglich existiert diese auf Modellebene nicht und das Testmodell ist damit in diesem Fall nicht eingabevollständig.

Dagegen wird von Entwicklungsmodellen, die zur Verifikationen von Systemeigenschaften oder zur Codegenerierung dienen, die Eingabevollständigkeit häufig gefordert, da ein System in der Realität immer auf irgendeine Weise reagieren muss. Beispiele hierfür sind I/O Automaten [LT89], klassisches FOCUS [BS01] und das Werkzeug AUTOFOCUS [HSE97]. Diese Ansätze ergänzen implizit das unvollständig spezifizierte Verhalten durch Chaosvervollständigung (I/O Automaten und klassisches FOCUS) oder Idlevervollständigung (AUTOFOCUS). Diese Vervollständigungsmechanismen sind jedoch für die Testmodellmodellierung nicht sinnvoll, da sie implizit Modellverhalten erzeugen, das in vielen Fällen für den Test des SUT nicht geeignet ist und in diesem Fall deren Erzeugung bei

der Testfallgenerierung mühsam durch entsprechende Testfallspezifikationen verhindert werden muss.

Testmodell und Umgebungsmodell Häufig wird parallel zum Testmodell ein Umgebungsmodell entwickelt. Das Umgebungsmodell modelliert einen für das Testmodell relevanten Teil des Verhaltens der Umgebung des SUT (vgl. Abbildung 2.2). Das Umwelt-

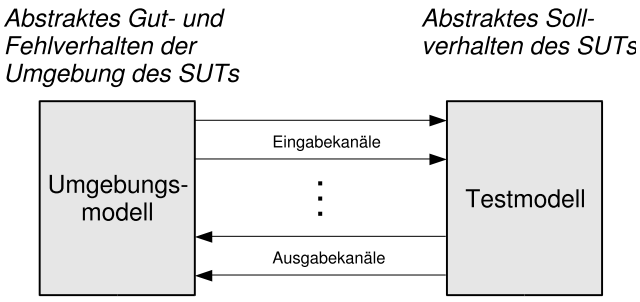


Abbildung 2.2.: Umgebungsmodell und Testmodell

modell beschreibt also das Verhalten von z.B. weiteren Geräten in einem Netzwerk, Teile von Betriebssystemen, Dienste einer Kommunikationsinfrastruktur usw. Der Zweck von Umweltmodellen ist häufig einer oder mehrere der folgenden:

- Das Umgebungsmodell codiert Annahmen an das Umgebungsverhalten oder charakteristisches Umgebungsverhalten, indem es sowohl Gut- als auch Fehlverhalten der Umgebung simuliert. Zur Modellierung der Umgebung wird vor allem funktionale Abstraktion (vgl. Abschnitt 3.2.1) und strukturelle Abstraktion (vgl. Abschnitt 3.2.5) eingesetzt.
- Das Umgebungsmodell dient als Treiber, um das Testmodell bei der Simulation zur Ausführung zu bringen. Dieses Vorgehen erleichtert die Validation bzw. Verifikation des Modellverhaltens, da die Erzeugung von Simulationsabläufen des Testmodells vereinfacht wird. Diese Art von Umgebungsmodell wird häufig nur zu Entwicklungszeit des Testmodells verwendet, d.h. bei der Generierung von Testfällen aus dem Testmodell wird dieses nicht in Betracht gezogen.
- Mit dem Umgebungsmodell wird das Verhalten des Testmodells eingeschränkt, d.h. das Testmodell zeigt unter Einschränkung des Umgebungsverhaltens nur einen Teil seines Verhaltens in Form von Kommunikationsabläufen. Die so erzeugten Abläufe dienen z.B. zur Prüfung der Korrektheit ausgewählter Funktionen des Testmodells oder zur Erzeugung von Testfällen die einerseits zum Test des SUT oder zum Regressionstest einer Weiterentwicklung des Testmodells dienen.

Methodischer Einsatz von Testmodellen Testmodelle haben einerseits den Zweck, Spezifikationsdokumente zu validieren, zu vervollständigen und zu ergänzen anderseits

dienen sie als abstrakte Referenzimplementierung, gegen die das Schnittstellenverhalten des SUT getestet wird.

Ausgangspunkt für die Entwicklung eines Testmodells sind meist informelle Spezifikationsdokumente, die ggf. Ablauf-, Sequenzdiagramme und Tabellen mit informeller Semantik enthalten. Beispiele hierfür sind die Fallstudien [PPS+03, CJRZ01, FHP02]. Bei der Entwicklung des Testmodells werden schrittweise die Anforderungen an das Schnittstellenverhalten des SUT im Testmodell codiert. Bei jedem Entwicklungsschritt wird beispielsweise eine weitere Anforderung an eine Funktionalität oder ein Verhaltensmuster in die operationelle Sprache des Testmodells umgesetzt und mit dem bisher modellierten Verhalten integriert. Bei diesem Vorgang werden schrittweise Auslassungen, Mehrdeutigkeiten und Widersprüche in den Spezifikationsdokumenten aufgedeckt. Ggf. nach Absprache mit Entscheidungsträgern wird das Verhalten in solchen Fällen eindeutig festgelegt und damit die Lücken der Spezifikationsdokumente in Laufe der Modellierung des Testmodells geschlossen. Es wird also der Teil der Spezifikationsdokumente, die das SUT betreffen, durch das Testmodell eindeutig gemacht und geschärft. Am Ende seiner Entwicklung repräsentiert das Testmodell eine abstrakte Referenzimplementierung des SUT, die gegenüber den Spezifikationsdokumenten korrekt und gegenüber den Nutzungsanforderungen valide ist und damit das Sollverhalten des SUT kodiert.

Im nächsten Schritt werden Testfallspezifikationen festgelegt und mit Hilfe der modellbasierten Testfallgenerierung eine oder mehrere Testsuiten erzeugt. Diese werden durch den Testtreiber mit dem SUT zur Ausführung gebracht und weisen damit die Übereinstimmungen bzw. Unterschiede zwischen dem Schnittstellenverhalten der Implementierung –dem SUT– und der Referenz –dem Testmodell– nach.

Nutzen von ausführbaren Testmodellen Abschließend identifizieren wir den Nutzen beim Einsatz von Testmodellen in Form von ausführbaren Verhaltensmodellen. Ausführbare Verhaltensmodelle ermöglichen eine gezielte und schrittweise Simulation des Modells in unterschiedlichen Betriebssituationen des Systems und die Erzeugung von charakteristischen Systemabläufen. Dadurch

- helfen sie dem Entwickler in der Anforderungsanalyse, unzureichend verstandene Bereiche der Spezifikationsdokumente einzuschätzen und zu untersuchen,
- verbessern sie die Kommunikation zwischen den unterschiedlichen Parteien, die an der Entwicklung beteiligt sind,
- erlauben sie die Untersuchung und Einschätzung von Design- und Spezifikationsvarianten und
- ermöglichen sie den direkten Vergleich zwischen Spezifikation und Implementierung mittels modellbasierten Testen.

2.2.2. Testfallspezifikationen

Im Allgemeinen codieren Testmodelle unendlich viele und unendlich lange Abläufe. Testen ist jedoch ein endlicher Prozess. Deshalb muss eine Auswahl an Abläufen getroffen und deren Länge eingeschränkt werden. Dies gilt insbesondere für eingebettete Systeme, da hier ein Test häufig 10 Sekunden oder mehr benötigt. Zu diesem Zweck werden Selektionskriterien für Abläufe –die so genannten Testfallspezifikationen– definiert. Testfallspezifikationen sind operational in dem Sinne, dass ein Testfallgenerator zusammen mit dem Testmodell und der Testfallspezifikation eine endliche Anzahl von Testfällen endlicher Länge erzeugt. Grundlage für die Spezifikation von Testfallspezifikationen bilden häufig übergeordnete Testziele. Testziele beschreiben ggf. auch informell, was getestet werden soll. Zum Beispiel schreibt ein Testziel vor, dass die Implementierung einer bestimmten Anforderung mittels Test auf dem konkreten System verifiziert werden soll oder dass die erzeugten Testfälle einem ausgewähltem Abdeckungskriterium genügen müssen. Der Unterschied zwischen Testziel und Testfallspezifikation ist vergleichbar mit der Beziehung zwischen Anforderung und Spezifikation. Insgesamt lassen sich folgende unterschiedliche Aufgaben, Zwecke oder Rollen von Testfallspezifikationen identifizieren:

- Testfallspezifikationen legen fest, was getestet wird, d.h. welche Teile des Testmodellverhaltens gegen das des SUT verglichen werden. Sie definieren also, welche Testfälle aus der potentiell verfügbaren Testfallmenge zu einer Testsuite ausgewählt werden.
- Testfallspezifikationen erlauben anstatt der mühsamen Ableitung einzelner Testfälle die Spezifikationen ganzer Testfallmengen mit gewünschten Eigenschaften.
- Testfallspezifikationen beziehen sich auf eine oder mehrere Anforderungen an das System.
- Testfallgenerierung kann als Suchproblem aufgefasst werden. Diesbezüglich haben Testfallspezifikationen die Aufgabe den Suchraum einzuschränken.
- Die Qualität einer Testsuite kann über die Testfallspezifikation definiert werden [GG75, ZHM97].

In der Literatur [BJK+05, PPS+03, Pre03b] werden die drei Arten funktionale, strukturelle und stochastische Testfallspezifikation unterschieden. In vielen Ansätzen werden sie zur Testfallerzeugung miteinander kombiniert. Wir erläutern sie in den folgenden Absätzen genauer.

Funktionale Testfallspezifikationen Funktionale Testfallspezifikationen beziehen sich auf die Anforderungen des Systems, indem mit ausgewählten Umgebungsverhalten oder Nutzungsfällen das Verhalten des Testmodells auf bestimmte Funktionalitäten eingeschränkt wird. Typische Techniken, um Testfallspezifikationen in Hinblick auf einzelne oder kombinierte Funktionalitäten zu definieren, sind die folgenden.

- Die Testfallspezifikation schreibt das Erreichen eines bestimmten Zustands vor, da dieser mit der gewünschten Funktionalität verknüpft ist oder die Erfahrung gezeigt hat, dass in diesem Bereich häufig Fehler auftreten [FKL99]. Dieses Vorgehen ist zum Beispiel geeignet zur Realisierung von Testzielen wie "Ist der Füllstand x erreicht, dann muss die Pumpe abgeschaltet werden". In diesem Fall wird also mit der Testfallspezifikation festgelegt, dass ein entsprechender Zustand im Testmodell erreicht werden muss, um eine gewünschte Reaktion hervorzurufen. Der Testfallgenerator hat dann die Aufgabe des Testfallgenerators Abläufe mit dieser Eigenschaft zu finden.
- Ähnlich verhält es sich, wenn Testfälle erzeugt werden müssen, die sich auf Ein-/Ausgabesequenzen von Spezifikationsdokumenten beziehen [PPS+03]. Zum Beispiel wird zum Test des Verhaltens beim Abbruch einer Transaktion spezifiziert, dass die Testfälle eine Nachricht zum Eröffnen und später eine zum Abbruch einer Transaktion als Teilsequenz enthalten müssen.
- Eine weitere Technik ist, ausgewählte Sequenzen aus den Spezifikationsdokumenten zu einem Umgebungsmodell zu vervollständigen. Durch Komposition des Umgebungsmodells mit dem Testmodell wird gezielt die Funktionalität des Testmodells auf eine gewünschte eingeschränkt, d.h. Testfälle werden unter Einschränkung des Umgebungsmodells generiert [PPS+03].

Strukturelle Testfallspezifikationen Strukturelle Testfallspezifikationen beziehen sich auf die Struktur des Testmodells oder auch auf die Struktur eines zugehörigen Umgebungsmodells. Das Spektrum der eingesetzten Überdeckungskriterien reicht von Anweisungsüberdeckung [FKL99], Zustandsüberdeckung [DBG01], bis hin zu komplexen Überdeckungskriterien wie MC/DC [Pre03a, PPS+03] oder Pfadüberdeckung [SA99, KVZ98]. Obwohl bisher kein schlüssiger Nachweis über die Fähigkeit von strukturellen Kriterien zum Auffinden von Fehlern erbracht werden konnte, haben sie den Vorteil, dass sie messbar sind und auf diesem Wege die Qualität einer Testsuite definiert werden kann.

Stochastische Testfallspezifikationen Stochastische Testfallspezifikationen selektieren Testfälle anhand einer zugrunde liegenden stochastischen Verteilung oder erzeugen gemäß einer Verteilung zufällige Eingaben und benutzen das Testmodell, um die zugehörigen erwarteten Ausgaben zu berechnen. Es ist Gegenstand reger Diskussionen wie Zufallstesten im Vergleich zu anderen Testmethoden abschneidet [DN84, Nta98, Ham94, HT90, Gut99].

Abschließend merken wir an, dass der Zufall entweder implizit durch die zugrunde liegenden Such- bzw. Generierungsstrategie oder durch eine explizite zufällige Auswahl von Testfällen aus einer Menge, die zuvor generiert wurde, bei praktisch allen Testfallgenerierungsansätzen eine Rolle spielt. Ferner wird bei der Generierung häufig die Länge der zu erzeugenden Testfälle explizit eingeschränkt oder es werden Testfälle minimaler Länge

ausgewählt. Die Annahme, dass dieses Vorgehen für die Fehleraufdeckungsrate eine untergeordnete Rolle spielt, ist willkürlich, aber sie erleichtert die manuelle Diagnose von Fehlerursachen, falls ein Fehlverhalten aufgedeckt wurde.

2.3. Inkrementelle Entwicklung

In diesem Abschnitt gehen wir von der Cleanroom-Methode aus, bei der die inkrementelle Entwicklung von Systemen im Vordergrund steht. Wir übernehmen wesentliche Elemente dieser Methode und entwickeln eine einfache inkrementelle Entwicklungsmethodik für Testmodelle.

2.3.1. Inkrementelle Entwicklung in der Cleanroom-Methode

Die inkrementelle Entwicklung von Softwaresystemen wurde erstmals 1970 von Mills [Mil71] vorgeschlagen, gewann aber erst in den achtziger Jahren bei der Entwicklung und Anwendung der Cleanroom-Methode [PTLP98] an Bedeutung. Hier wird die inkrementelle Entwicklung von Softwaresystemen als ein Top-down Ansatz aufgefasst, in dem die Funktionen eines Softwaresystems schrittweise entwickelt und getestet werden. Durch Anwendung der inkrementellen Entwicklungsmethodik ergeben sich im wesentlichen folgende Vorteile:

Sichtbarkeit des Fortschritts Im inkrementellen Entwicklungsprozess implementiert ein Inkrement eine Teilmenge von Funktionen des Gesamtsystems. Bei der Entwicklung des nächsten Inkrements werden weitere Funktionen zum System hinzugefügt und integriert. Jedes Inkrement enthält also die Funktionalität aller zuvor entwickelten Inkremente und einige neue. Dadurch wächst die Funktionalität des Systems auf kontrollierte Weise an, bis das System vollständig realisiert ist. Zu Beginn der Entwicklung ist z.B. das erste Inkrement ein Minimalsystem, bei dem man zuversichtlich ist, dass insgesamt 10% aller Funktionen korrekt gegenüber den Nutzungsanforderungen realisiert und lauffähig sind, anstatt zu vermuten, dass von allen Funktionen 10% realisiert sind.

Intellektuelle Beherrschbarkeit Der inkrementelle Entwicklungsprozess zerlegt die Entwicklung des Gesamtsystem in überschaubare Teilschritte. Nach Fertigstellung jedes Inkrements wird die Qualität der realisierten Funktionen gegenüber den Nutzungsanforderungen bzw. der Spezifikation geprüft und sichergestellt. Dadurch wird die intellektuelle Beherrschbarkeit der einzelnen Teilschritte und des Systems gewährleistet.

Kontinuierliche Qualitätskontrolle Zu jedem Inkrement wird folgender Zyklus durchlaufen: (1) Einbeziehen und ggf. Vervollständigung der Spezifikation, (2) Entwicklung der neuen Funktionalität und (3) Verifikation der neuen Funktionen und Testen des gesamten Systems, das bisher entwickelt wurde.

Kontinuierliches Feedback durch den Nutzer Jedes Inkrement ist ablauffähig und realisiert einen Teil der Funktionen des Gesamtsystems. Dadurch kann das System frühzeitig vom Nutzer des Systems ausgeführt werden. Der Nutzer erhält also früh die Möglichkeit Feedback zum Softwaresystem zu geben. Dies vermeidet die Entwicklung falscher Systemfunktionalität, die nicht den Anforderungen des Nutzer genügt, und ermöglicht dadurch das frühzeitige Einarbeiten von erforderlichen Korrekturen der Nutzeranforderungen.

2.3.2. Inkrementelle Entwicklung von Testmodellen

Testmodelle codieren das Sollverhalten eines SUT mit Hilfe von operationellen Verhaltensmodellen. Wir nutzen die Vorteile der inkrementellen Entwicklung, indem wir ihre

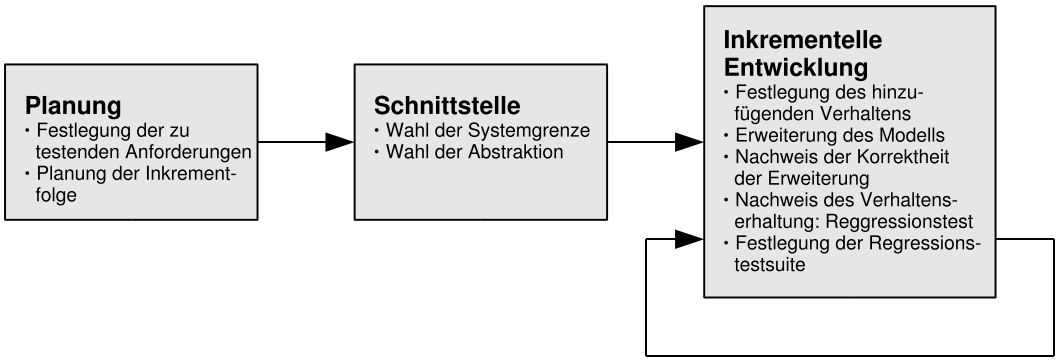


Abbildung 2.3.: Inkrementeller Entwicklungsprozess von Testmodellen

Prinzipien für den Entwicklungsprozess von simulierbaren Verhaltensmodellen anpassen. Der Prozess ist schematisch in Abbildung 2.3 dargestellt und zerfällt in drei wesentliche Phasen:

Planung In dieser Phase wird festgelegt, welche Anforderungen an Funktionalitäten des SUT mit Hilfe eines Testmodells verifiziert werden sollen (vgl. funktionale Abstraktion in Abschnitt 3.2). Ferner wird ein Plan erstellt, in welcher Reihenfolge diese Anforderungen von Inkrement zu Inkrement im Modell realisiert werden. Die Planung der Reihenfolge wird von einem oder mehreren der folgenden Faktoren beeinflusst:

- Der Nutzer des SUT wünscht, dass bestimmte Funktionalitäten vor anderen getestet werden. In diesem Fall sind die entsprechenden Anforderungen Kandidaten für eine frühe Integration in das Testmodell.
- Zu Beginn der Testmodellbildung sind die verschiedenen Anforderungen an das System unterschiedlich präzise formuliert bzw. unterschiedlich gut verstanden. Durch die inkrementelle Entwicklung können Domänenexperten durch

die Ausführung der Inkremente früh Rückkopplung zum Modellverhalten geben. Die unterschiedliche Klarheit der Anforderungen kann die Planung auf zwei unterschiedliche Weisen beeinflussen: (1) Unklare Anforderungen werden in einem frühen Inkrement modelliert, so dass sie präzisiert werden können. (2) Alternativ werden unklare Anforderungen in späten Inkrementen modelliert, um bis dahin Fragen diese Anforderungen betreffend klären zu können.

- Funktionalitäten mit höher Nutzungswahrscheinlichkeit werden vorzugsweise in frühen Inkrementen modelliert. Dadurch werden sie häufiger überprüft und das Vertrauen in ihre korrekte Modellierung wird erhöht, da sie jedesmal nach Fertigstellung eines Inkrements im Testprozess überprüft und ggf. ihre zugehörigen Regressionstests erweitert werden.

- Die Minimalfunktion wird zu Beginn der Entwicklung des Modells realisiert und dann schrittweise um Erweiterungen ergänzt. Dieses Vorgehen wird häufig bei Modellen für eingebettete Software verwendet. Hier steht z.B. zu Beginn die Modellierung des Aufstartverhaltens bevor weitere Funktionalitäten modelliert werden.

Spezifikation der Schnittstelle In dieser Phase wird die syntaktische Schnittstelle des Testmodells spezifiziert, d.h. welche Nachrichten an das Testmodell gesendet bzw. vom Testmodell empfangen werden und wie diese in einzelne Schnittstellen strukturiert werden. Dies ist bereits ein sehr kritischer Schritt, da hier einerseits die Systemgrenze des Testmodells genau festgelegt wird und bei der Schnittstellenbildung die Abstraktionsformen Datenabstraktion und Kommunikationsabstraktion eingesetzt werden (vgl. Abschnitt 3.2). Diese Abstraktionsbildung wird ggf. die gewählte funktionale Abstraktion aus dem vorangegangen Schritt beeinflusst. Zur automatischen Ableitung von prototypischen Strukturen und Schnittstellen aus Spezifikationssequenzen, die mit UML-SDs oder MSCs spezifiziert sind, können ggf. vom Verfasser mitentwickelte Verfahren [KPSB02, KPS01, KPS02] verwendet werden.

Inkrementelle Entwicklung des Verhaltensmodells Ein Entwicklungsschritt von einem Inkrement zum folgenden wird in folgende Teilschritte untergliedert:

- Spezifikation der endlichen Ablaufmenge, die im Verhaltensmodell in diesem Schritt realisiert wird,

- Erweiterung des Verhaltensmodells, um die spezifizierte Ablaufmenge zusätzlich zum bereits vorhandenen Verhalten zu realisieren,

- Nachweis, dass das spezifizierte Verhalten durch das Verhaltensmodell realisiert wird,

- Regressionstest zum Nachweis, dass das gewünschte Verhalten der vorangegangenen Inkremente erhalten wurde und

- Erweiterung der Regressionstestsuite um Abläufe, damit das neu hinzufügte

Verhalten im nächsten Inkrement des Modell verifiziert werden kann.

Analog zu der inkrementellen Entwicklungsmethodik aus Cleanroom ergeben sich bei der inkrementellen Entwicklung von Testmodellen folgende Vorteile:

- Durch die Sichtbarkeit des Entwicklungsfortschritts und durch das kontrollierte Wachstum der modellierten Funktionalität im Verhaltsmodell erhält man das Vertrauen, dass das Testmodell das Sollverhalten des SUT korrekt realisiert.
- Durch die Zerlegung in überschaubare Entwicklungsschritte und die Konzentration auf die Entwicklung und Integration weniger Funktionalitäten in das bestehende Modell wird die intellektuelle Beherrschbarkeit des Verhaltensmodells und das kontrollierte Annähern an das gewünschte und eindeutige Sollverhalten des Systems gewährleistet.
- Die kontinuierliche Qualitätskontrolle am Ende jedes Inkrements sichert den Erhalt des gewünschten Verhaltens der vorangegangen Inkremente und erhöht dadurch wiederum das Vertrauen in die Korrektheit des Modells.
- Durch den schrittweisen Aufbau und Integration von Teilverhalten in das Testmodell werden frühzeitig Konflikte zu bereits spezifiziertem Modellverhalten und Spezifikationslücken aufgedeckt. Diese können z.B. durch Simulationsläufe des Testmodells demonstriert werden und mit Hilfe von Experten- oder Nutzerfeedback aufgelöst bzw. geschlossen werden. Das Resultat ist, dass durch die Modellbildung die Spezifikationsdokumente validiert und vervollständigt werden.

Ein formale Behandlung der Inkrementbildung von Verhaltensmodellen erfolgt in Kapitel 5, die Demonstration anhand einer industriellen Fallstudie in Kapitel 7.

2.4. Zusammenfassung

Der Zweck dieses Kapitels ist die Einführung und Klärung der zentralen Begriffe für diese Arbeit, die wir hier zusammenfassen:

- Modellbildung und Modelle werden charakterisiert durch drei Merkmale: (1) Bezug zur Realität, (2) Abstraktion von unwesentlichen Details und (3) Zweckgebundenheit.
- Reaktive Systeme unterscheiden sich von transformativen Systemen, indem sie im Allgemeinen nicht terminieren und Zeit als weitere Dimension besitzen. Sie reagieren zur Laufzeit auf Eingaben bzw. Sequenzen von Eingaben, indem sie abhängig von ihrem inneren Zustand Ausgaben bzw. Sequenzen von Ausgaben berechnen.
- Verifikation bezeichnet die Überprüfung der Übereinstimmung zwischen einem Produkt und seiner Spezifikation (“Wird ein korrektes Produkt entwickelt?”), Validation dagegen die Überprüfung der Eignung bzw. den Wert eines Produktes gegenüber seinem Einsatzzweck (“Wird das richtige Produkt entwickelt?”).

- Wir unterscheiden zwischen Fehlverhalten und Fehlerursache. Fehlverhalten resultiert aus der Beobachtung, dass das Istverhalten eines Systems von seinem Sollverhalten abweicht. Fehlerursache ist der Grund für das Auftreten eines Fehlverhalten. In dieser Arbeit bezeichnen wir Fehlverhalten meist vereinfacht als Fehler.

- Testen umfasst die Aktivitäten zum Nachweis der Übereinstimmung bzw. Abweichung von Ist- und Sollverhaltens eines Systems. Diese umfassen die Aktivitäten Testfallermittlung, Testdurchführung, Monitoring und Testauswertung, sowie vorbereitende und begleitende Aktivitäten der Testplanung, Testorganisation und Testdokumentation.

- Die Grundlage für das modellbasierte Testen bildet das Testmodell, das das abstrakte Sollverhalten des zu testenden Systems (SUT) modelliert. Zusammen mit der Testfallspezifikation –einem Selektionskriterium für Testfälle– erzeugt der Testfallgenerator eine Testsuite (Menge von Testfällen). Diese Testfälle befinden sich auf dem gleichen Abstraktionsniveau wie das Testmodell. Ein Testtreiber bringt Testfälle mit dem SUT zur Ausführung, indem er den Eingabeteil der Testfälle –die Testdaten– konkretisiert und diese an das SUT anlegt. Ferner zeichnet der Testtreiber die Ausgaben des SUT auf und vergleicht diese nach Abstraktion mit den erwartenden Ausgaben der Testfälle. Der Vergleich führt zur Testauswertung und zur Bildung des Testurteils.

- Testmodelle haben in zweierlei Hinsicht Bedeutung im Entwicklungsprozess eines Systems: (1) Die Modellentwicklung liefert einen wertvollen Beitrag zur Analyse und Vervollständigung von Spezifikationsdokumenten und damit zum Verständnis des Systems. (2) Das modellbasierte Testen auf Basis des Testmodells sichert die Konformität zwischen Spezifikation und Implementierung.

- Testfallspezifikationen sind operationell mit dem Testmodell ausführbar und steuern die Testfallgenerierung, indem sie Eigenschaften der zu erzeugenden Testfälle festlegen und den Suchraum einschränken. Wir unterscheiden funktionale, strukturelle und stochastische Testfallspezifikation, die sich auf funktionale Anforderungen des Systems, auf die Struktur des Modells bzw. die zufällige Erzeugung von Testfällen beziehen.

- Testmodelle werden inkrementell entwickelt, d.h. in das Modellverhalten werden schrittweise die Anforderungen an das Systemverhalten integriert, vervollständigt und eindeutig ausführbar gemacht. Dieses Vorgehen sichert das strukturierte und schrittweise Annähern an das vollständige Sollverhalten des SUT, unterstützt die intellektuelle Beherrschbarkeit und erhöht das Vertrauen in die Korrektheit des Modells. Die inkrementelle Entwicklung zerfällt in drei wesentliche Phasen: (1) die Planung, welche Anforderungen an das Verhalten des Systems mittels Testen verifiziert und in welcher Reihenfolge diese in das Modell integriert werden, (2) die Spezifikation der Schnittstelle, die die Wahl der Systemgrenze und die Wahl des Abstraktionsgrads einschließt und (3) der inkrementelle Verhaltensaufbau, der die Integration einer Anforderung durch Erweiterung des Modells, den Nachweis der

Korrektheit der Erweiterung und den Erhalt von Modellverhalten zyklisch wiederholt.

3. Abstraktionen in Testmodellen

Abstraktion ist ein wesentliches Merkmal bei der Modellbildung (vgl. Abschnitt 2.1). Abstraktionen in einem Modell sind immer mit Informationsverlust gegenüber der Realität behaftet. Folglich ist das offensichtliche Problem bei allen Modellierungsaktivitäten, welche Informationen im Modellen weggelassen und vernachlässigt werden dürfen und welche nicht. Dies hängt ab von der Domäne, in der die Modellbildung eingesetzt wird, bzw. allgemeiner, welcher Einsatzzweck mit dem Modell verfolgt wird. Die Wahl der geeigneten Abstraktion eines Modells ist nach wie vor eine Kunst im Bereich der Softwarekonstruktion. Der Zweck dieses Kapitels ist, einen ersten Schritt in Richtung dem ingenieurmäßigen Erstellen von Testmodellen zu gehen, indem wir dem Entwickler von Testmodellen eine Klassifikation von einsetzbaren Abstraktionstechniken an die Hand geben, deren Einsatzzweck erläutern, deren Grenzen aufzeigen und deren bevorzugten Einsatzort im inkrementellen Entwicklungsprozess aufzeigen. Die vorgestellte Klassifikation der Abstraktionstechniken stützt sich zu einem auf die Recherche von Veröffentlichungen, welche den Einsatz von modellbasierten Testen anhand einer Fallstudie demonstrieren, und zum anderen auf die Erfahrungen, welche der Autor in Rahmen eines industriellen Pilotprojekts zum modellbasierten Testen [PPW+05] sammelte. Grundlegende Ideen und Teile dieses Kapitels wurden vom Autor in [PP04, PERH05] bereits veröffentlicht.

Abschnitt 3.1 führt eine übergeordnete Klassifikation von Abstraktionen und weitere Eigenschaften dieser ein. Abschnitt 3.2 diskutiert die Abstraktionstechniken bei Modellierung von Testmodellen und belegt diese mit Beispielen aus der Literatur. Darauf aufbauend werden in Abschnitt 3.3 die Grenzen der Abstraktion bei der Testmodellbildung untersucht. Abschnitt 3.4 beschäftigt sich mit dem Einsatz der Abstraktionstechniken im inkrementellen Entwicklungsprozess von Testmodellen und Abschnitt 3.5 schließt das Kapitel mit einer Zusammenfassung.

3.1. Abstraktionsarten bei der Modellbildung

In Abschnitt 2.1 stellten wir fest, dass Abstraktion ein grundlegendes Merkmal bei der Modellbildung ist. Abstraktion bedeutet die Vereinfachung der Realität und geht damit immer mit Informationsverlust einher. In dieser Arbeit unterscheiden wir zwei grundsätzliche Abstraktionsarten: (1) Abstraktion, die Komplexität und Details der Realität kapselt, d.h. Informationen kapselt, die später automatisch mit einem Compiler, Linker o.ä. wieder hinzugefügt werden können, und (2) Abstraktion, die Informationen der Realität tatsächlich weglassen, d.h. dieser Informationsverlust kann nicht automatisch mittels

Compiler etc. ausgeglichen werden. Diese Abstraktionen und zwei weitere Eigenschaften von Abstraktionen –Domänenspezifität und Zweckgebundenheit– diskutieren wir in diesem Abschnitt.

Abstraktion: Kapselung von Details Die erfolgreichsten Abstraktionstechniken basieren heutzutage auf Compilern und Bibliotheken, die die automatische Übersetzung eines Programms bzw. Modells zu seiner Realisierung ermöglichen. Beispiele hierfür sind:

- In Programmiersprachen abstrahieren Prozeduren von konkreten stack frames.
- Die Swing-API [Sun04b] abstrahiert von einer hohen Anzahl von Befehlen, die früher benötigt wurde, um Benutzeroberflächen zu programmieren.
- Das ISO-OSI Referenzmodell [Tan00] abstrahiert von Kommunikationsdetails. Eine Instruktion auf hoher Ebene wird atomar behandelt und entspricht einer komplexen Interaktion auf niedrigeren Ebenen, die meist unterbrechbar ist.
- Corba [OMG00] und J2EE [Sun04a] sind weitere Beispiele für Abstraktion von Kommunikationsdetails, die durch entsprechende Compiler eingefügt werden.

In diesem Zusammenhang können wir Modellierungssprachen als natürliche Erweiterung von Programmiersprachen auffassen, bei denen die grundlegende Idee ist, zu abstrahieren, indem Details und Komplexität mit Hilfe von Bibliotheken und Sprachkonstrukten gekapselt werden. Diesen Vorgehen liegt im allgemeinen ein Art Makromechanismus zugrunde, welcher beim Kompilieren oder zur Laufzeit aufgelöst wird. Im Rahmen des modellbasierten Testens kann die Abstraktionslücke zwischen Testmodell und SUT, der Abstraktion durch Kapselung von Details zugrunde liegt, mit Hilfe des Testtreibers aufgelöst werden.

Abstraktion: Weglassen von Details Wenn bei Modellierung substanzielle Details weggelassen werden, d.h. kein Makromechanismus die fehlende Information automatisch einfügen kann, dann wird das Modell wesentlich vereinfacht und dies trägt damit zur leichteren Verständlichkeit des Modells bei. Mit anderen Worten ausgedrückt, es gibt eine inhärente Komplexität in Systemen, bei deren Abstraktion ein substanzieller Informationsverlust auftritt. Dies ist eine Variation der bedeutenden Bemerkung von Brooks [Bro86], dass Komplexität eher eine essenzielle als eine zufällige Eigenschaft ist. Zum Beispiel konzentriert sich das Modell einer smart card von Philipps u.a. [PPS+03] auf den Protokollfluss und abstrahiert vom Inhalt der Dateien und von der Wirkung der Dateioperationen. Diese Information ist im Modell nicht enthalten, um das Modell überschaubar und wartbar zu halten. Ohne weitere Informationen kann ein statischer Compiler bzw. der Testtreiber die abstrakten Dateien nicht zu einer konkreten Datei instanziieren und zusätzlich die zugehörigen dynamischen Veränderungen umsetzen. Ein weiteres Beispiel für fehlende Informationen ist, dass einige Modelle von konkretem Zeitverhalten abstrahieren. Die statische Übersetzung durch einen Compiler bzw. Testtreiber kann im Allgemeinen nicht das hoch diverse Zeitverhalten einer konkreten Realisierung

instanziieren. Für das modellbasierte Testen hat dies zur Folge, dass ein Testtreiber den Informationsverlust, der durch die Abstraktionsform "Weglassen von Details" entsteht, nicht bzw. nicht vollständig ausgleichen kann.

Abstraktion ist domänenspezifisch Die Abstraktionslücke, die automatisch durch einen Makroexpansionsmechanismus aufgelöst werden kann, ist mehr oder weniger spezifisch für eine Domäne. Während das Konzept von Prozeduren allen Programmiersprachen gemein ist, ist die Swing-API beschränkt auf die Programmierung von Benutzeroberflächen und die MDA [ORM01] bezieht sich mehr auf Geschäftsanwendungen als auf eingebettete Echtzeitsysteme. Diese domänenspezifischen Abstraktionstechniken zu finden und zu entwickeln, zählt sicherlich zu einer der wichtigen Herausforderungen in der Informatik.

Abstraktion ist zweckgebunden Zusammenfassend treten Abstraktionen mit Informationsverlust in zwei Formen auf. Sie sind Vereinfachungen, wo fehlende Information automatisch eingefügt werden kann, oder sie sind Vereinfachungen, wo Information vorsätzlich fehlt, um das Modell einfach zu halten. Vereinfachungen neigen dazu domänenspezifisch zu sein, und das zentrale Ziel bei der modellbasierten Entwicklung liegt darin, geeignete Abstraktionen zu finden, welche bei allen Systemen einer Domäne oder Produktlinie anwendbar sind.

Gemäß dem Modellbegriff aus Abschnitt 2.1, ergibt Abstraktion nur Sinn, wenn ein bestimmter Zweck damit verfolgt wird. Verschiedene Abstraktionen werden für verschiedene Zwecke verwendet. Sie werden eingesetzt, um (1) ein System oder dessen zugrunde liegenden Anforderungen besser zu verstehen bzw. zu analysieren, (2) um ein System zu spezifizieren, (3) um auf Teile eines Systems gekapselt zuzugreifen (z.B. durch Bibliotheksaufrufe oder durch Binden von externen Komponenten oder Diensten), (4) um die Kommunikation zwischen Entwicklern zu fördern, (5) um Code zu generieren und (6) um Systeme zu testen.

3.2. Klassifikation von Abstraktionen in Testmodellen

In diesem Abschnitt identifizieren und dokumentieren wir die verschiedenen Abstraktionsklassen, die verwendet werden, um Testmodelle zu entwickeln. Die Grundlage dieser Übersicht bilden die Erfahrungen des Autors in einem industriellen Pilotprojekt [PPW+05] und die Untersuchung von Fallstudien im industriellen Kontext aus den Domänen Prozessortest [DBG01, SA99, FKL99], Test von smart cards [PPS+03, CJRZ01], Test von Protokollimplementierungen [KVZ98, BFV+99] und Test von Betriebssystemteilen bzw. Teilen von Implementierungen eines Programmiersprachenstandards [FHP02]. Wir unterschieden die Klassen funktionale Abstraktion, Datenabstraktion, Kommunikationsabstraktion, temporale Abstraktion und strukturelle Abstraktion, die Gegenstand von Abschnitt 3.2.1, 3.2.2, 3.2.3, 3.2.4 bzw. 3.2.5 sind.

In Tabelle 3.1 geben wir an, welche der Abstraktionsklassen bei welcher Fallstudie angewendet wurde. Der Tabelleneintrag *ja* bedeutet, dass das Abstraktionsprinzip bei der Fallstudie eingesetzt wurde, *nein* bedeutet das Gegenteil und das Symbol *?*, dass in der Veröffentlichung kein Hinweis vorhanden ist, ob das Prinzip eingesetzt wurde oder nicht. Insgesamt zeigt die Untersuchung, dass bei fast allen Fallstudien explizit dokumentiert ist, dass die Prinzipien funktionale Abstraktion und Datenabstraktion zum Einsatz kamen.

Fallstudie	Funktionale Abstraktion	Daten-abstraktion	Temporale Abstraktion	Kommunika-tionsabstraktion	Strukturelle Abstraktion
[PPW+05]	ja	ja	ja	ja	ja
[DBG01]	ja	ja	nein	ja	?
[SA99]	ja	ja	ja	ja	?
[FKL99]	ja	?	?	?	?
[PPS+03]	ja	ja	ja	ja	?
[CJRZ01]	?	?	?	?	?
[KVZ98]	ja	?	?	ja	?
[BFV+99]	?	ja	?	?	?
[FHP02]	ja	ja	?	?	?

Tabelle 3.1.: Eingesetzte Abstraktionsklassen

In den folgenden Abschnitten werden die fünf Abstraktionsklassen erläutert, mit Beispielen aus der Literatur belegt und –soweit möglich– durch die Einteilung aus Abschnitt 3.1 Kapselung bzw. Weglassen von Details charakterisiert.

3.2.1. Funktionale Abstraktion

Der Zweck von funktionaler Abstraktion ist, sich auch auf die "wesentliche" Funktionalität bzw. Teile des Ein-/Ausgabeverhaltens des SUT zu konzentrieren, die mit Hilfe des Testmodells verifiziert werden soll. Dies führt zur Vereinfachung von komplexen Sachverhalten im Testmodell, welche für den Test nicht benötigt werden. Das bedeutet nicht notwendigerweise, dass Spezialfälle oder Ausnahmeverhalten im Testmodell weggelassen werden, falls diese getestet werden müssen. Gemäß der Klassifikation in Abschnitt 3.1 unterscheiden wir zwei Arten von funktionaler Abstraktion:

Kapseln von Funktionen In diesem Fall wird eine Funktion im Modell durch einen Namen symbolisiert, ohne die Funktion selbst auf Modellebene zu modellieren. Erst durch den Testtreiber wird dieser symbolische Name durch eine Implementierung der Funktion ersetzt.

Weglassen von Funktionen Diese Form der Abstraktion wird auf unterschiedlichen Granularitätsebenen angewandt: zum einen werden ganze Systemteile, die in den Spezifikationsdokumenten vorgesehen sind und eine oder mehrere Funktionen beschreiben, im Modell vollständig weggelassen. Dies führt zu einer starken Vereinfachung

bzw. starken Reduktion von Komplexität im Testmodell. Zum anderen ist im Testmodell nicht spezifiziert, wie in einer Situation auf manche Ereignisse reagiert wird (fehlende Eingabevollständigkeit auf Modellebene).

In manchen Fällen kann funktionale Abstraktion durch Weglassen von Funktionen den modellbasierten Testprozess wie folgt unterstützen: Falls die Funktionalität der SUT in unabhängige Teile zerlegt werden kann, dann kann eine starke Komplexitätsreduktion dadurch erreicht werden, indem für unabhängige Teilfunktionalitäten der SUT separate Testmodelle erstellt werden, um jede Funktionalität unabhängig von der anderen zu testen. Ein offensichtlicher Nachteil bei diesem Vorgehen ist, dass durch das unabhängige Testen der Teilfunktionalitäten keine "feature interaction" zwischen diesen Funktionalitäten aufgedeckt werden kann.

Beispiele für funktionale Abstraktion In einer Fallstudie von Philipps u.a. [PPS+03] dient das Testmodell zum Test des Protokollverhaltens einer smart card gegenüber ihrer Umgebung, einem Terminal. Insbesondere steht nicht der Test der komplexen kryptographischen Funktionen einer smart card im Vordergrund. Deshalb wurde im Testmodell von einer Realisierung der kryptographischen Funktionen einer smart card abstrahiert. Diese Funktionen werden auf Modellebene gekapselt, indem sie lediglich durch symbolische Namen im Testmodell dargestellt werden. Zum Beispiel existieren im Modell nur die Namen `encrypt(data)` für das Aufrufen einer Verschlüsselungsfunktion und `encryptedData` als Antwort auf die Funktion. Auf Testtreiberebene werden jene Namen durch Aufrufe entsprechender kryptographischer Algorithmen bzw. deren Rückgabewerte ersetzt.

In der Fallstudie von Pretschner u.a. [PPW+05] wird ein Netzwerkcontroller getestet. Für diesen sind in den Spezifikationsdokumenten drei Hauptfunktionalitäten beschrieben. Im Testmodell werden jedoch nur zwei von diesen modelliert und die dritte weggelassen. Die Konsequenz nur jene zwei testen zu können, wurde in Kauf genommen, da die dritte Funktionalität für die stabile Funktion des Netzwerks nicht relevant ist.

Farchi u.a. [FHP02] verwenden funktionale Abstraktion auf zwei Arten. Zum einen werden separate Testmodelle zum Test von unterschiedlichen Funktionen des POSIX-Standards modelliert. Das erste Modell wurde für den Test des byte locking interfaces `fcntl` entwickelt. Diese Schnittstelle ermöglicht die Kontrolle über geöffnete Dateien, um den Zugriff von Prozessen auf diese Dateien zu steuern. Zum anderen wird die Funktionalität der `fcntl`-Schnittstelle auf Modellebene im Vergleich zum POSIX-Standard funktional reduziert, indem im Modell nur ein einmaliges Erweitern der Dateien erlaubt ist. Ferner wird in dieser Veröffentlichung ein weiteres Modell erwähnt, das zum Test der `fork`-Operation in POSIX entwickelt wurde.

Weiteres Beispiel für funktionale Abstraktion ist, das Verhalten im Testmodell lediglich partiell zu spezifizieren, d.h. im Modell wird in bestimmten Zuständen für bestimmte Eingabewerte kein Verhalten festgelegt oder im Modell wird vollkommen von einer Ausnahmebehandlung abstrahiert.

3.2.2. Datenabstraktion

Mit Hilfe der Datenabstraktion werden konkrete Datentypen auf logische Datentypen im Testmodell abgebildet, um im Modell eine kompakte Darstellung der Daten zu erhalten oder eine Reduktion der Datenkomplexität zu erreichen. Eines der bekanntesten Beispiele für Datenabstraktion ist, Binärzahlen auf konkreter Ebene durch ganze Zahlen auf abstrakter Ebene zu repräsentieren. In diesem Fall liegt eine Kapselung von Realisierungsdetails (ohne substanziellen Informationsverlust) vor, da im Modell lediglich die Repräsentation der Datenwerte verändert wurde oder formal ausgedrückt eine Bijektion zwischen den Werten des konkreten Datentyps und den Werten des zugehörigen abstrakten Datentyps existiert. Eine oft verwendete Datenabstraktion unter Weglassen von Realisierungsdetails (mit substantiellen Informationsverlust) ist die Äquivalenzklassenbildung auf konkreten Datenwerten. Dabei werden disjunkte Mengen von konkreten Datenwerten auf abstrakter Modellebene jeweils einem abstrakten Datenwert zugeordnet.

Häufig auftretende Muster bei der Anwendung von Datenabstraktion sind folgende:

- Abstrakte Datenwerte werden im Testmodell als Operanden in Operationen verwendet, um das Sollverhalten der SUT zu spezifizieren. Dann ist das Testziel, nachzuweisen, ob die Operationen der SUT auf den konkreten Datenwerten korrekt gegenüber den abstrakten Operationen auf abstrakten Datenwerten im Testmodell implementiert wurden.
- Datenabstraktion wird durch funktionale Abstraktion motiviert. In diesem Fall werden z.B. Äquivalenzklassen von Nachrichten gebildet, die zu unterscheidbaren Verhalten des SUT führen, indem Nachrichtenbestandteile weggelassen oder zusammengefasst werden oder die Menge aller Nachrichten des Systems wird auf die Menge reduziert, die für die modellierten Systemfunktionen relevant sind. Beim Einsatz dieser Form der Datenabstraktion wird die implizite Annahme getroffen, dass eine analoge Klassifikation beim SUT-Verhalten vorliegt. Die Rechtfertigung dieser Annahme muss ggf. zuvor überprüft werden.

Beispiele für Datenabstraktion Dushina u.a. [DBG01] testeten modellbasiert eine Store Data Unit (SDU) eines digitalen Signalprozessors (DSP). Das Verhalten des SDU hängt im wesentlichen von dem Füllstand der Datenpuffer innerhalb der SDU ab. Im Testmodell wurde der Füllstand eines Puffers mit den abstrakten Datenwerten `empty`, `valid`, `quasifull` und `full` repräsentiert, die jeweils eine Menge von konkreten Füllwerten modellieren. Beim Test der realen SDU wurde festgestellt, dass diese nicht mit voller Leistung arbeitet, da der Füllstand nur den Status `quasifull` erreicht, d.h. die volle Kapazität der Puffer nicht ausgenutzt wird.

Philipps u.a. [PPS+03] verwenden Datenabstraktion, um im Testmodell einer smart card vollständig vom Inhalt ihrer Dateien zu abstrahieren. Die Dateien werden im Modell lediglich durch symbolische Namen repräsentiert.

Ein weiteres Beispiel für die Anwendung von Datenabstraktion ist das Einführen von Äquivalenzklassen auf den Datentypen der Operanden von Operationen des SUT (siehe [SA99, PPS+03]).

Hudak u.a. [HCDG+02] beschreiben ein Modell eines Netzwerksystems, in dem Datenabstraktion durch eine vorausgegangene funktionale Abstraktion motiviert wurde. Auf konkreter Ebene besteht ein Netzwerkpaket aus Zieladresse, Absenderadresse, Paketkörper, Prüfsumme usw. Der Zweck des Modells ist den routing-Mechanismus einer Netwerkkomponente zu prüfen, daher werden im Modell nur die routing-Funktionen der Komponente realisiert (funktionale Abstraktion). Dies führt dazu, dass das Modell mit Hilfe von Datenabstraktion weiter vereinfacht werden kann, indem der Inhalt eines Pakets auf die für das routing benötigte Bestandteile Zieladresse und Prüfsumme reduziert wird und alle anderen im Modell weggelassen werden.

3.2.3. Kommunikationsabstraktion

Die wohl bekannteste Anwendung von Kommunikationsabstraktion ist das ISO-OSI Referenz Modell. Hier wird eine komplexe Interaktion auf konkreter Ebene zu genau einer Operation auf abstrakter Ebene zusammengefasst. Auf abstrakter Ebene wird die Operation als atomar betrachtet, obwohl im Allgemeinen auf konkreter Ebene die zugehörige Interaktion von Operationen mit anderen Operationen verschränkt ablaufen kann. Mit dieser Klasse von Abstraktion können Handshaking-Operationen oder voneinander kausal abhängige Operationen zu einer Operation abstrahiert werden. Es handelt sich hier um die Abstraktionsform “Kapseln von Details”.

Beispiele für Kommunikationsabstraktion Bei Hardware-Verifikation [BCG+00] und Prozessortest [DBG01] werden eine konkrete Aggregation von Pinwerten, mehrere aufeinander folgende Signale auf unterschiedlichen Bussen oder wiederkehrende Prozessoranweisungen zu einer symbolischen Operation im Modell zusammengefasst.

Kahlouche u.a. [KVZ98] fassen im Modell zum Testen einer Protokollimplementierung kausal abhängige Operationen, die zu einer Transaktion gehören, zu einer atomaren Operation zusammen, obwohl die Transaktion auf konkreter Ebene unterbrochen werden kann.

Kommunikationsabstraktion wird auch mit Datenabstraktion kombiniert: im Testmodell einer smart card [PPS+03] werden die konkreten Byte-String-Kommandos einer smart card, die durch eine Folge von Hexadezimalzahlen repräsentiert werden, als symbolische und menschenlesbare Nachrichten dargestellt.

3.2.4. Temporale Abstraktion

Der Zweck von temporaler Abstraktion ist, vom genauen Zeitpunkt beim Auftritt von Ereignissen zu abstrahieren. Dabei wird angenommen, dass nur die Reihenfolge von Ereignissen bzw. Nachrichten relevant ist, nicht jedoch das genaue zeitliche Auftreten. Dagegen betrachten wir Abstraktion, bei denen die Reihenfolge der Ereignisse irrelevant ist, als Kommunikationsabstraktion. Wir unterscheiden folgende Formen von temporaler Abstraktion:

- Das Testmodell und das SUT verwenden unterschiedliche Granularitäten diskreter Zeit. Dabei ist die Zeitgranularität auf abstrakter Ebene größer als auf konkreter Ebene.

- Das Testmodell abstrahiert von physikalischer Zeit. Zum Beispiel hängt die konkrete Implementierung von einem Timer der Dauer 200 ms ab. Im Testmodell wird dieser abstrahiert zu zwei symbolischen Ereignissen, eines zum Start des Timers und eines für den Ablauf des Timers. Damit wird im Modell durch Einführen zweier diskreter Ereignisse von der physikalischen Dauer des Timers abstrahiert. Dies ist auch dann möglich, wenn sich die Dauer des konkreten Timers zur Laufzeit verändert.

- Im Testmodell wird von zeitlichen Nichtdeterminismus abstrahiert, d.h. auf Modellebene werden alle Ein-/Ausgabesequenzen des SUT, bei denen die Nachrichten in unterschiedlichen Zeitabständen auftreten, aber die Nachrichtenreihenfolge identisch ist, durch eine Sequenz repräsentiert.

Prinzipiell könnte temporale Abstraktion zu Quality-of-Service Abstraktion verallgemeinert werden. Die Anwendung von Abstraktion auf andere Qualitätsmerkmale außer Zeit ist prinzipiell denkbar, wurde aber zum aktuellen Stand der Literatur im Bereich des modellbasierten Testens noch nicht verwendet.

Beispiele für temporale Abstraktion Die erste Form der temporalen Abstraktion, unterschiedliche Granularitäten von diskreter Zeit zu verwenden, wird häufig im Bereich der Hardwareverifikation verwendet [DBG01, SA99, BCG+00, Mel88]. In diesem Bereich werden einem Uhrenzyklus auf abstrakter Ebene mehrere Zyklen auf konkreter Ebene zugeordnet. Diese Art von temporaler Abstraktion wird häufig mit Kommunikationsabstraktion oder funktionaler Abstraktion kombiniert.

Im Testmodell eines Netzwerkcontrollers [PPW+05] wird z.B. der Timer *t_Answer*, der die Zeitdauer festlegt, wie lange der Netzwerkkontroller auf die Antwort eines Gerätes warten soll, abstrahiert durch die diskreten Ereignisse `startTAnswer` und `timeoutTAnswer`. Diese symbolisieren den Start bzw. den Ablauf des Timers.

3.2.5. Strukturelle Abstraktion

Mit Hilfe von struktureller Abstraktion werden Strukturen im realen System im Modell vereinfacht oder umstrukturiert mit dem Ziel die Verständlichkeit und die Analysierbarkeit der Modellierung zu erhöhen bzw. zu erleichtern. Strukturelle Abstraktion zählt zu der Abstraktionsgruppe "Kapselung von Details". Diese Abstraktionsform wird häufig durch andere Abstraktionsformen motiviert, deren Vereinfachungen auch eine Vereinfachung der Modellstrukturen ermöglichen. Ferner kann strukturelle Abstraktion immer dann eingesetzt werden, wenn Testmodelle ausschließlich zum Testen des Schnittstellenverhaltens des SUT verwendet werden, da in diesem Fall die interne Struktur des SUT auf Modellebene keine Rolle spielt.

Beispiel für strukturelle Abstraktion Im Modell eines Netzwerkcontrollers [PPW+05] werden physikalische Geräte in der Umgebung des Controllers nicht wie der Controller selbst durch eine Komponente in der Modellierungssprache AutoFocus [HSE97], sondern durch eine rekursive Datenstruktur modelliert. Eine Komponente simuliert das Verhalten dieser Geräte, indem sie auf dieser Datenstruktur operiert. Dadurch wird neben der starken Vereinfachung der Modellstruktur erreicht, dass das Modell einfach parametrisiert werden kann, um eine variable Anzahl von Geräten in der Umgebung zu simulieren.

Hudak u.a. [HCDG+02] beschreiben eine Klimaanlagensteuerung in einem hohen Gebäude. Hier werden Sensoren unabhängig von ihrer Lage im Gebäude und ihrem Zusammenschluss im Modell als Variablen repräsentiert und zur Erleichterung der Analyse nach Art der Sensoren in unterschiedliche Gruppen zusammengefasst.

3.3. Grenzen beim Einsatz von Abstraktion

In diesem Abschnitt gehen wir auf die Grenzen und Konsequenzen ein, die der Einsatz von Abstraktionen in Testmodellen beim Testen des SUT mit sich bringt. Es ist entscheidend, dass der Modellierer sich der Grenzen bewusst ist, und deshalb den richtigen Kompromiss zwischen Abstraktion und Präzision findet, wenn er ein geeignetes Testmodell, das die kritischen Aspekte des SUT modelliert, entwickelt. Wie wir in Abschnitt 3.1 ausgeführt haben, gibt es eine inhärente Komplexität in realen Systemen. Wenn ein Teil davon im Modell abstrahiert wird (Abstraktion durch "Weglassen von Details", d.h. mit substanziellem Informationsverlust), dann gibt es keine Möglichkeit, in dem SUT Fehler zu entdecken, die diese Details betreffen. Im folgendem erläutern wir einige Einschränkungen und Konsequenzen, die beim Einsatz von Abstraktionen in Testmodellen auftreten können.

Oft leiden Modelle unter einer unterschiedlich stark ausgeprägten und impliziten "Happy world"-Annahme, die durch den Einsatz von funktionaler Abstraktion zustande kommt.

Ein typisches Beispiel ist, dass im Modell angenommen wird, dass Parameter von Eingabenachrichten oder Eingabeoperationen sich in erlaubten Grenzen befinden, eine erlaubte Länge oder Stelligkeit besitzen. Mit Hilfe solcher Modelle ist es nicht möglich, das SUT-Verhalten für jenen Fall zu testen, wenn dieses eine Nachricht mit illegalen Parametern empfängt. Zum Beispiel in der Domäne von smart cards bestehen eine Operation und ihre Operanden auf konkreter Ebene aus einer Folge von Bytes. Im Modell dagegen werden Operationen und seine Operanden durch einen menschenlesbaren Namen symbolisiert. Mit dieser Art von Modellen, kann das Verhalten der smart card nicht direkt getestet werden, wenn sie eine Bytefolge empfängt, die ein Byte zu kurz bzw. zu lang ist. Es liegt beim Testingenieur zu entscheiden, ob eine solch illegale Eingabe auf Modellebene oder auf Ebene des Testtreibers behandelt wird.

Der intensive Einsatz von Datenabstraktion kann zu Informationsverlust führen, der für die Testfallgenerierung nicht überwunden werden kann. Zum Beispiel abstrahiert das smart card Modell von Philipps u.a. [PPS+03] vollständig von Dateiinhalten und symbolisiert Dateien und Dateioperationen ohne Realisierung im Modell. Folglich können der Inhalt der Dateien und ihre Veränderung während der Laufzeit mit diesem Modell nicht getestet werden. Es können nur statische Eigenschaften wie z.B. die Länge der Datei überprüft werden.

Es ist schwierig so genannte “feature interaction” aufzudecken, wenn funktionale Abstraktion zur Entwicklung separater Testmodelle verwendet wird, mit dem Zweck unterschiedliche Funktionalitäten zu testen. Zum Beispiel benutzen Farchi u.a. [FHP02] zwei separate Modelle, um unterschiedliche Operationen des POSIX-Standards zu testen. Diese Modelle helfen, die korrekte Funktion von Einzeloperationen zu verifizieren, aber das Aufdecken unerwünschten Verhaltens, das ggf. durch “feature interaction” dieser Operationen verursacht wurde, ist nicht möglich.

Probleme können auftreten, wenn temporale Abstraktion intensiv im Modell eingesetzt wird. Offensichtlich kann der rigorose Einsatz von temporaler Abstraktion im Bereich von Echtzeitsystemen das Auffinden von Fehlern beeinträchtigen, die durch empfindliches, ineinander verzahntes Zeitverhalten von verschiedenen Systemkomponenten entstehen. Aus diesem Grund verzichteten Dushina u.a. [DBG01] beim Test eines Prozessors bewusst auf temporale Abstraktion, um einerseits die Nachvollziehbarkeit der generierten Testfälle zu vereinfachen und andererseits das Leistungsverhalten des SUT testen zu können. Deshalb entspricht in dieser Fallstudie einem Uhrenzyklus im Modell genau ein Zyklus im realen Prozessordesign.

3.4. Einsatz der Abstraktion im Entwicklungsprozess

In diesem Abschnitt erläutern wir, wie und an welcher Stelle die Abstraktionsprinzipien aus Abschnitt 3.2 im inkrementellen Entwicklungsprozess von Testmodellen eingesetzt werden. Abbildung 3.1 zeigt den Überblick und die Struktur dieses Abschnitts gemäß der drei Phasen aus Abschnitt 2.3.2: Abschnitt 3.4.1 beschreibt den Einsatz der Abstrak-

tionsformen in der Planung der Entwicklung des Testmodells, Abschnitt 3.4.2 erläutert den Einsatz der Abstraktionsformen bei der Definition von Schnittstellen im Testmodell und Abschnitt 3.4.3 geht auf den Einsatz der Abstraktionsformen während der inkrementellen Entwicklung des Verhaltensmodells ein.

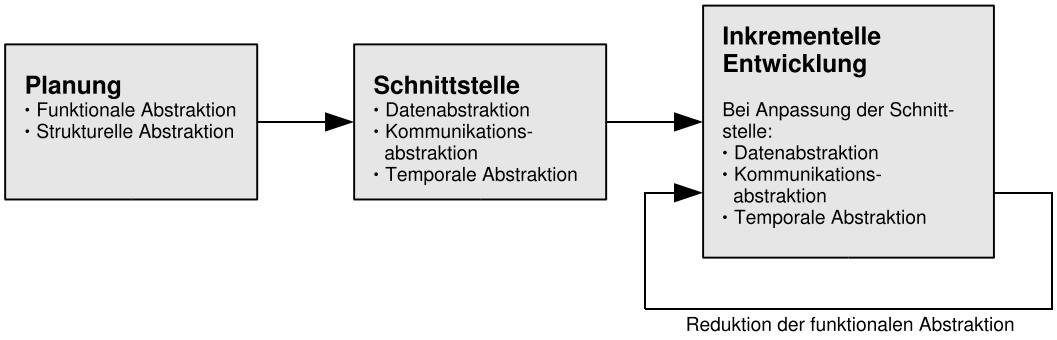


Abbildung 3.1.: Abstraktionen im inkrementellen Testmodellentwicklungsprozess

3.4.1. Abstraktionen in der Planung

Bei der Planung der Entwicklung des Testmodells wird funktionale Abstraktion und strukturelle Abstraktion eingesetzt und durch folgende Fragen gegliedert:

- Welche Systemteile, Funktionalitäten bzw. Anforderungen des SUT sollen mittels Testmodell verifiziert werden? Durch diese Frage wird systematisch festgelegt, was im Modell modelliert werden muss und was nicht, d.h. wie das zu entwickelnde Modell durch Weglassen von Systemteilen und einzelner Funktionalitäten vereinfacht werden kann bzw. welche Systemteile und Funktionalitäten modelliert werden müssen (vgl. funktionale Abstraktion durch Weglassen von Funktionen, Abschnitt 3.2.1). Die Auswahl der zu modellierenden Funktionalitäten ist besonders kritisch, da durch sie festgelegt wird, welche Systemteile später überhaupt mit dem Testmodell verifiziert werden können. Sie wird durch unterschiedliche Faktoren beeinflusst, wie der Nutzungswahrscheinlichkeit der verschiedenen Funktionen, dem Einfluss der Funktionen auf Zuverlässigkeit, der Performanz oder Sicherheit im Gesamtsystem etc.
- Welche Funktionalität wird auf Ebene des Testmodells und welche wird auf Ebene des Testtreibers realisiert? Durch diese Frage wird entschieden, welche Funktionalität im Modell modelliert werden muss und welche im Modell gekapselt werden kann, indem sie im Testtreiber untergebracht wird (vgl. funktionale Abstraktion durch Kapseln von Funktionen, Abschnitt 3.2.1).
- Wie werden Systemteile und Funktionalitäten des SUT und ggf. Teile der Umgebung des SUT im Modell strukturiert? Durch diese Frage wird festgelegt, wie das Modell in strukturelle und funktionale Komponenten zerlegt wird. Die Struktu-

rierung weicht ggf. von den Strukturen vorgegeben in Spezifikationsdokumenten ab, um eine einfachere Modellierung und Analyse des Systems auf Modellebene zu ermöglichen (vgl. strukturelle Abstraktion, Abschnitt 3.2.5).

- In welcher Reihenfolge werden die Funktionalitäten bzw. deren Anforderungen im Testmodell modelliert? Mittels dieser Frage wird geplant, in welcher Reihenfolge das Verhalten des Testmodells inkrementell aufgebaut wird, damit am Ende der Entwicklung das Testmodell die zu modellierende Funktionalität vollständig modelliert. Häufig wird mit der Entwicklung von Anforderungen begonnen, die typisch für die Anwendung bzw. für das Normalverhalten sind, und beendet mit Anforderungen, die Spezialfälle oder Ausnahmebehandlung mit einbeziehen (vgl. Abschnitt 2.3.2). Insgesamt spiegelt die Planung der Entwicklung wieder, wie zu Beginn der Entwicklung die starke funktionale Abstraktion durch Weglassen von Funktionen schrittweise durch die inkrementelle Weiterentwicklung des Testmodells abgebaut wird.

3.4.2. Abstraktion bei der Spezifikation der Schnittstellen

Bei der Spezifikation der syntaktischen Schnittstellen im Testmodell werden die Abstraktionsformen Datenabstraktion, Kommunikationsabstraktion und temporale Abstraktion eingesetzt. Diese spielen bei der Definition der Datentypen der syntaktischen Schnittstellen im Testmodell eine wesentliche Rolle und werden durch folgende Fragen gegliedert:

- Welche Nachrichten bzw. Ereignisse im realen System sind für das Testmodell relevant und wie werden sie in Datentypen für das Modell zusammengefasst? Aufgrund der funktionalen Abstraktion, die in der Entwicklungsplanung erfolgte, kann die Menge der Nachrichten und Ereignisse, die im realen System auftreten, im Allgemeinen für das Testmodell erheblich reduziert werden (vgl. Datenabstraktion, Abschnitt 3.2.2).

- Können Nachrichten, Operationen oder deren Parameter in Äquivalenzklassen zusammenfasst werden? Durch die funktionale Abstraktion in der Entwicklungsplanung können häufig Äquivalenzklassen von konkreten Nachrichten, Operationen und deren Parameter identifiziert werden, die gleiches Verhalten implizieren. Diese Äquivalenzklassen werden jeweils durch einen abstrakten Datenwert im Testmodell repräsentiert (vgl. Datenabstraktion, Abschnitt 3.2.2).

- Bei welchen Nachrichten und Ereignissen im realen System spielt die Reihenfolge ihres Auftretens keine Rolle bzw. kann eine Folge von Nachrichten und Ereignissen als eine atomare Transaktionen aufgefasst werden? In diesem Fall können jene Nachrichten und Ereignisse im Testmodell zu einer Nachricht bzw. einem Ereignis zusammengefasst werden (vgl. Kommunikationsabstraktion, Abschnitt 3.2.3).

- Inwieweit spielt Zeitverhalten im realen System eine Rolle? Kann bzw. darf temporale Abstraktion verwendet werden, um im Testmodell von Zeitverhalten zu

abstrahieren (vgl. temporale Abstraktion, Abschnitt 3.2.4)?

3.4.3. Abstraktion beim inkrementellen Verhaltensaufbau

Bei Beginn der inkrementellen Entwicklung des Verhaltensmodells wird zunächst das Minimalverhalten des SUT modelliert, d.h. das erste Inkrement des Verhaltensmodells stellt eine starke funktionale Abstraktion durch "Weglassen von Details" dar. Im Laufe der inkrementellen Weiterentwicklung gemäß der Entwicklungsplanung, die u.a. festlegt welche Funktionalitäten bzw. deren Anforderungen in das Verhaltensmodell von Inkrement zu Inkrement integriert und modelliert werden, wird die funktionale Abstraktion durch "Weglassen von Details" schrittweise reduziert. Dieser Vorgang erfordert unter Umständen, dass Schnittstellen und deren Datentypen angepasst werden müssen, und damit –wie in Abschnitt 3.4.2 beschrieben– wiederum die Abstraktionsformen Datenabstraktion, Kommunikationsabstraktion und temporale Abstraktion eingesetzt werden.

3.5. Zusammenfassung

Inhalt dieses Kapitels ist eine Klassifikation von Abstraktionstechniken und deren Anwendung bei der Entwicklung von Testmodellen.

- Wir legen dar, dass Abstraktionen bei der Modellbildung zweckgebunden und domänenspezifisch sind und sich in durch zwei Hauptklassen Abstraktion durch *Kapselung von Details* und Abstraktion durch *Weglassen von Details* charakterisieren lassen. Abstraktion durch Kapselung von Details hat die Eigenschaft, dass dieser eine Makromechanismus zugrundeliegt, der das automatische und vollständige Hinzufügen der gekapselten Information erlaubt. Bei Abstraktion durch Weglassen von Details ist dies nicht möglich, da die notwendigen Informationen im Modell fehlen, um diesen Automatismus zu gestatten.
- Bei der Modellierung von Testmodellen kommen fünf wesentliche Abstraktionsformen zum Einsatz:
  - funktionale Abstraktion, durch deren Einsatz die Funktionalität des zu testenden Systems im Testmodell reduziert wird oder Teile der Funktionalität im Testmodell gekapselt werden,
  - Datenabstraktion, die die Datenkomplexität im Testmodell verringert, indem konkrete Datentypen äquivalent repräsentiert oder durch Äquivalenzklassenbildung reduziert werden,
  - Kommunikationsabstraktion, bei der komplexe Interaktionen auf konkreter Ebene zu einer atomaren abstrakten Operation zusammengefasst werden,
  - temporale Abstraktion, durch die entweder im Testmodell ein gröberes Zeitmodell verwendet wird oder von physikalischen Zeitdauern abstrahiert wird,

  - strukturelle Abstraktion, die zur Restrukturierung des Modells gegenüber des zu testenden Systems dient.

  Es ist dabei charakteristisch, dass sich diese Abstraktionsformen häufig gegenseitig bedingen und in Kombination verwendet werden.

- Dem Einsatz der Abstraktionstechniken bei der Modellierung von Testmodellen sind Grenzen gesetzt und es obliegt dem Modellierer, einen geeigneten Kompromiss zwischen Abstraktion und Präzision zu finden, so dass das Testmodell genügend Information enthält, um die entscheidenden Aspekte des zu testenden Systems verifizieren zu können.
- Es zeichnet sich ab, dass bei den drei Phasen der inkrementellen Entwicklung von Testmodellen mit unterschiedlicher Gewichtung obige Abstraktionstechniken zum Einsatz kommen:
  - bei der Planung wird hauptsächlich funktionale und strukturelle Abstraktion verwendet,
  - bei der Spezifikation der Testmodellschnittstelle kommt verstärkt Datenabstraktion, Kommunikationsabstraktion und temporale Abstraktion zum Einsatz und
  - bei dem inkrementellen Verhaltensaufbau des Testmodells wird schrittweise die funktionale Abstraktion im Modell reduziert und ggf. Datenabstraktion, Kommunikationsabstraktion und temporale Abstraktion eingesetzt, falls die Schnittstelle während dem Entwicklungsprozess angepasst werden muss.

4. Formale Grundlagen

In diesem Kapitel beschreiben wir die formalen Grundlagen für die folgenden Teile dieser Arbeit. In Abschnitt 2.2.1 haben wir beschrieben, welchen Anforderungen eine Modellierungssprachen für die Entwicklung von Testmodellen genügen muss. Dazu führen wir eine Modellierungssprache auf Basis von FOCUS [BS01] ein. Der Beobachtungsbegriff von FOCUS, Systeme durch die Menge ihrer Ein-/Ausgabeabläufe zu beschreiben, bildet eine geeignete Grundlage, die erzeugbaren Testfälle eines Testmodells auszudrücken. Hierzu passen wir entsprechend unseren Anforderungen zeitsynchrones FOCUS an, und erhalten dadurch neben einer einfach verständlichen Modellierungssprache den Vorteil der Werkzeugunterstützung durch AUTOFOCUS [HSE97] mit zugehörigem Testfallgenerator [Pre01, PLP01]. Abbildung 4.1 zeigt die Struktur unseres Ansatzes, der auch die Struktur des Kapitels widerspiegelt. Die Basis bildet das Modellverhal-

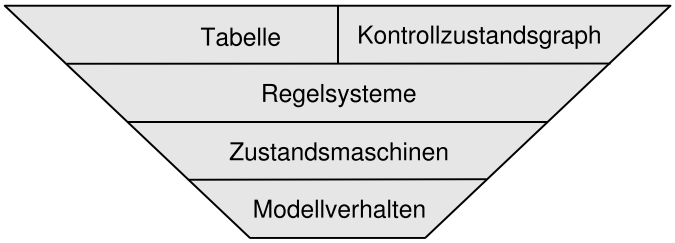


Abbildung 4.1.: Struktur der Modellkonzepte

ten (Abschnitt 4.1). Dieses beschreibt das SUT-Verhalten an dessen Schnittstelle mit Hilfe von Ein-/Ausgabesequenzen. Das Modellverhalten entspricht also der Menge aller Testfälle, die potenziell mit dem Testmodell erzeugt werden können. Dem deskriptiven Beschreibungscharakter des Modellverhaltens stellen wir Zustandsmaschinen zur Seite, die eine operationelle Beschreibung des Modellverhaltens ermöglichen (Abschnitt 4.2). Im Allgemeinen sind sowohl das Modellverhalten –die Menge aller Abläufe– als auch die operationelle Beschreibung durch eine Zustandsmaschine –Menge aller Transitionen auf einen Datenraum– nicht endlich. Diesem Umstand begegnen wir, indem wir Regelsysteme zur endlichen Beschreibung von Zustandsmaschinen einführen (Abschnitt 4.4). Die wesentliche Idee der Regelsysteme ist, eine Menge von Transitionen auf dem Datenraum mit Hilfe von Prädikaten bestimmter Form zu spezifizieren. Um die Benutzbarkeit von Regelsystemen einfacher zu gestalten, führen wir zwei unterschiedliche syntaktische Repräsentationen von Regelsystemen ein. Die eine basiert auf Tabellen (Abschnitt 4.5), die eine strukturierte und detaillierte Übersicht über alle Verhaltensregeln des Regelsystems bieten und eine einfache und direkte Manipulation des Regelsystems während der

Entwicklung des Testmodells ermöglichen. Die zweite stellt Regelsysteme in Form eines Graphen dar (Abschnitt 4.6). Diese Darstellungsform hat den Vorteil, dass sie eine abstrahierte Sicht auf das Regelsystem bietet und die schnelle Erfassung eines abstrakten Kontrollflusses des Regelsystems erleichtert. Später in der Arbeit werden wir je nach Einsatzzweck beide Darstellungsformen von Regelsystemen verwenden, um Testmodelle inkrementell zu entwickeln. Das Ende des Kapitels schließen wir mit einer Diskussion und einer Zusammenfassung ab (Abschnitt 4.7 bzw. 4.8).

4.1. Formale Modellierungssprache

In diesem Abschnitt legen wir die formalen Grundlagen für die folgenden Abschnitte. Zunächst führen den zentralen Strombegriff aus FOCUS, einfache Operationen darauf, sowie die grundlegenden Begriffe Variable, Belegung und Formel ein. Der Strombegriff bildet die Basis zur Einführung des Begriffs des Modellverhaltens. Das Modellverhalten beschreibt die Beziehungen zwischen den Ein- und Ausgabeströmen an der Schnittstelle des betreffenden Modells. Diese bilden die Ein-/Ausgabeabläufe des Modells und entsprechen der Menge aller erzeugbaren Testfälle.

4.1.1. Mathematische Grundlagen

Die Grundlage zur Definition des Modellverhaltens von Modellen und zur Beschreibung der Kommunikation zwischen Modellen bzw. ihrer Umgebung bildet der Strombegriff aus FOCUS [BS01]. Ströme sind Sequenzen von Nachrichten, die die Kommunikationshistorie auf einem Kommunikationskanal beschreiben. Wir definieren Ströme und grundlegende Operationen auf diesen in der folgenden Definition:

Definition 4.1 (Ströme und ihre Operatoren). *Sei M eine nichtleere Menge. Ein Strom über M ist eine endliche oder unendliche Sequenz aus Elementen von M. Die Menge der endlichen und unendlichen Ströme bezeichnen wir mit M^* bzw. mit M^∞. Die Menge aller Ströme definieren wir als $M^\omega := M^* \cup M^\infty$*

Der Ausdruck $\langle x_1, \ldots, x_n \rangle \in M^*$ bezeichnet den endlichen Strom der Länge n bestehende aus den Elementen $x_1, \ldots, x_n \in M$. Der leere Strom wird mit dem Ausdruck $\langle\rangle$ dargestellt.

Für einen Strom $s \in M^\omega$ bezeichnet $\#s$ seine Länge. Ist s unendlich, dann gilt $\#s = \infty$.

Für einen Strom $s \in M^\omega$ bezeichnet $\mathsf{dom}.s := \{n \in \mathbb{N} | \, n \leq \#s\}$ seine Domäne, für $n \in \mathsf{dom}.s$ bezeichnet $s.n$ das n-te Element in Strom s und $s \downarrow n$ ist das Präfix von Strom s bis einschließlich dem n-ten Element $s.n$.

Für zwei Ströme s und t bezeichnet $s \frown t$ die Konkatenation von s und t. Ist s ein unendlicher Strom, dann gilt $s \frown t = s$.

Ein Strom s ist ein Präfix von Strom t, wenn s zu t verlängert werden kann. In Zeichen:

$$s \sqsubseteq t :\Leftrightarrow \exists r \in M^{\omega} \text{ mit } s \frown r = t$$

Wir haben uns hier auf die Einführung der Stromoperatoren beschränkt, die im Verlauf der Arbeit verwendet werden. Weitere Operatoren und deren Verwendung finden sich in [BS01].

Wir werden später Modelle mit Hilfe von Formeln beschreiben. Dazu führen wir jetzt die benötigten Begriffe ein.

Definition 4.2 (Variable, Belegung, Formel). *Die Menge aller Variablen wird mit VAR bezeichnet. Jeder Variablen $v \in VAR$ wird mit Hilfe der Abbildung $type : VAR \to TYPE$ ein Typ aus der Menge aller Typen $TYPE$ zugeordnet. Ein Typ ist eine Menge von Werten, die eine Variable annehmen kann.*

Eine Belegung $\alpha : VAR \to \bigcup_{T \in TYPE} T$ weist jeder Variablen v einen Wert $\alpha.v$ ihres Typs zu. Die Menge aller Belegungen wird mit VAL bezeichnet.

Eine Belegung α heißt typkorrekt, wenn α jeder Variablen $v \in VAR$ einen Wert gemäß seines Variablentyps $type(v)$ zuordnet. In Zeichen:

$$\forall v \in VAR \text{ gilt } \alpha.v \in type(v)$$

Die Menge aller typkorrekten Belegungen einer Variablenmenge $V \subset VAR$ wird mit B_V bezeichnet. Für zwei disjunkte Variablenmengen $V, W \subset VAR$ und zwei Belegungen $\alpha \in B_V$ und $\beta \in B_W$ bezeichnet $\alpha \oplus \beta$ die Summe der beiden Belegungen

$$(\alpha \oplus \beta).v = \begin{cases} \alpha.v & \text{falls } v \in V \\ \beta.v & \text{falls } v \in W \end{cases}$$

Zwei Belegungen $\alpha \in B_U$, $\beta \in B_V$ über den Variablenmengen $U, V \subset VAR$ stimmen auf einer Variablenmenge $W \subset U \cap V$ überein, wenn sie allen Variablen aus W jeweils den gleichen Wert zuordnen. In Zeichen:

$$\alpha \overset{W}{=} \beta :\Leftrightarrow \forall w \in W \text{ gilt } \alpha.w = \beta.w$$

Eine Formel Φ ist ein wohlgeformter logischer Ausdruck über Variablen aus VAR. Ist eine Formel gültig unter der Belegung α, dann wird dies in Zeichen wie folgt ausgedrückt:

$$\alpha \models \Phi$$

Für eine Variablenmenge V bezeichnet V' die Variablenmenge, die jede Variable aus V mit einem Strich versieht:

$$V' := \{v' \mid v \in V\}$$

Dabei bezeichnen v und v' zwei unterschiedliche Variablen. Für eine Belegung α wird die Belegung α' definiert, indem sie einer Variablen v' den gleichen Wert zuweist wie die Belegung α der Variablen v:

$$\forall v \in V \text{ gilt } \alpha'.v' = \alpha.v$$

Einer Formel, die sowohl ungestrichene als auch gestrichene Variablen enthält, wird ein Wahrheitswert zugewiesen, indem eine Belegung α und eine gestrichene Belegung β' verwendet wird. Die Aussage

$$\alpha, \beta' \models \Phi$$

gilt, falls die Formel Φ wahr wird, wenn die ungestrichenen Variablen mit α und die gestrichenen Variablen mit β' ausgewertet werden.

Die Menge der ungebundenen Variablen in einer Formel Φ wird mit $\mathsf{free}.\Phi$ *bezeichnet.*

4.1.2. Modellverhalten

In unserer Modellierungssprache kommunizieren Modelle ausschließlich über gerichtete Kanäle mit ihrer Umgebung. Folglich definieren wir zunächst den Kanalbegriff und dessen Eigenschaften:

Definition 4.3 (Kanäle). *Kanäle C sind eine Teilmenge der Variablen VAR. Der Typ $type(c)$ eines Kanals $c \in C$ beschreibt alle möglichen Nachrichten, die über den Kanal gesendet bzw. empfangen werden können.*

Jeder Kanaltyp $type(c)$ enthält den ausgezeichneten Wert ϵ. Der Wert ϵ drückt aus, dass ein Kanal mit keiner Nachricht belegt ist.

Wir unterscheiden explizit den Kanalwert ϵ von den übrigen Werten $t \in type(c) \setminus \{\epsilon\}$. Dadurch ermöglichen wir methodisch, dass ein Modell auf das nicht Vorhandensein einer Nachricht auf einem Kanal reagieren kann bzw. explizit keine Ausgabe sendet.

Wir beschreiben das Verhalten eines Modells, indem wir die Eingabe des Modells mit der Ausgabe des Modells an seine Umgebung in Beziehung setzen. Deshalb unterscheiden wir explizit Ein- und Ausgabekanäle eines Modells, die zu der Schnittstelle des Modells zusammengefasst werden:

Definition 4.4. *Die Schnittstelle eines Modells S wird durch das Paar (I, O) definiert, wobei I und O endliche Teilmengen der Kanalmenge C sind und diese die Eingabe- bzw. Ausgabekanäle von S beschreiben.*

In unserer Modellierungssprache gehen wir davon aus, dass Modelle von einer globalen Uhr getrieben werden. Diese wird durch die natürlichen Zahlen $\mathbb{N}$ beschrieben. Ein Modell vollzieht einen Schritt, indem es die Werte auf den Eingabekanälen einliest, die Ausgabe berechnet, gegebenenfalls seinen internen Zustand ändert und zum nächsten

Schritt die Ausgabe auf seine Ausgabekanäle schreibt.

Definition 4.5 (Kanalhistorien, Modellverhalten). *Zunächst werden die Menge aller Kanalhistorien $\vec{C}$ definiert: Die Menge aller Kanalhistorien zu Kanal $c \in C$ sind gegeben durch die Ströme $type(c)^\omega$. Entsprechend werden die Menge aller Kanalhistorien einer Kanalmenge $C = \{c_1, \ldots, c_n\}$ definiert durch*

$$\vec{C} := \left(\prod_{c \in C} type(c) \right)^\omega$$

Die Projektion einer Historie $h \in \vec{C}$ auf einen Kanal $c \in C$ wird mit $h.c$ bezeichnet. Da die Ströme $h.c$ der Kanäle $c \in C$ einer Kanalhistorie $h \in \vec{C}$ alle die gleiche Länge besitzen, kann der Längenoperator eindeutig auf Kanalhistorien $h \in \vec{C}$ geliftet werden:

$$\#h := \#h.c \text{ für ein beliebiges } c \in C$$

Analog wie bei den Variablenbelegungen definieren wir für disjunkte Kanalmengen C und D die Summe $h_C \oplus h_D$ zweier Kanalhistorien $h_C \in \vec{C}$ und $h_D \in \vec{D}$, wobei wir $\#h_C = \#h_D$ voraussetzen:

$$(h_C \oplus h_D).c = \begin{cases} h_C.c & \text{falls } c \in C \\ h_D.c & \text{falls } c \in D \end{cases}$$

Das Verhalten eines Modells S mit Schnittstelle (I, O) wird durch eine Relation $R_S \subset \vec{I} \times \vec{O}$ beschrieben. Dabei wird der Eingabebereich von S durch $\mathsf{in}.S := \{i \in \vec{I} | (i, o) \in R_S\}$ definiert, und die Relation R_S hat folgende Eigenschaften:

1. *R_S ist abgeschlossen bezüglich ihrer Eingabe, d.h.*

$$\forall (i, o) \in R_S \text{ gilt } \#i = \#o. \tag{4.1}$$

2. *R_S ist streng kausal, d.h.*

$$\begin{aligned} \forall i, \tilde{i} \in \mathsf{in}.S \, \forall n < min(\#i, \#\tilde{i}) \text{ mit } i\downarrow_n = \tilde{i}\downarrow_n \text{ gilt} \\ \{o\downarrow_{n+1} \mid (i, o) \in R_S\} = \{\tilde{o}\downarrow_{n+1} \mid (\tilde{i}, \tilde{o}) \in R_S\}. \end{aligned} \tag{4.2}$$

3. *R_S ist präfixabgeschlossen, d.h.*

$$\forall (i, o) \in R_S \, \forall 1 \leq n \leq \#i \text{ gilt } (i\downarrow_n, o\downarrow_n) \in R_S. \tag{4.3}$$

Um eine kompakte Darstellung des Modellverhaltens zu ermöglichen, werden wir auch die Funktionsdarstellung $F_S : \mathsf{in}.S \to \wp(\vec{O}), i \mapsto \{o \in \vec{O} | (i, o) \in R_S\}$ alternativ zu der Relationsdarstellung von R_S verwenden.

Wir fordern die Eigenschaften (4.1), (4.2) und (4.3) für eine Relation R_S, um die Realisierbarkeit von Modellen sicherzustellen: Zu allererst fordern wird, dass ein Modell zu jedem Verhalten seines Eingabebereichs $i \in \mathsf{in}.S$ ein oder mehrere Verhalten $(i, o) \in R_S$ zeigt, wobei die Ausgabehistorie o die gleiche Länge wie die zugehörige Eingabehistorie i hat. Bezüglich der getakteten Arbeitsweise von Modellen bedeutet dies, dass die letzte Ausgabe $o.\#o$ die Reaktion auf Eingabe $i.(\#i - 1)$ ist und damit die Eingabe $i.\#i$ im vorliegenden Ablauf (i, o) nicht mehr verarbeitet wurde. Als nächstes fordern wir, dass die Ausgabe eines Modells zu einem Zeitpunkt nur von den Eingaben bis zu diesem Zeitpunkt abhängen darf. Und als drittes fordern wir, dass für jedes Verhalten $(i, o) \in R_S$ die zugehörigen Präfixe zum Verhalten des Modells gehören.

Für den Rest der Arbeit bzw. in der Praxis ist der Nachweis der drei Eigenschaften nicht erforderlich, da wir Zustandsmaschinen zur Beschreibung des Modellverhaltens verwenden und deren Semantik die drei Eigenschaften sicherstellt.

Ferner ist zu anzumerken, dass wir später in der Arbeit die Relation R_S als die Menge aller Testfälle eines Testmodells S identifizieren und damit ein Verhalten $(i, o) \in R_S$ einem Testfall mit Testdaten i und erwarteten Ausgaben o entspricht. Abschließend definieren wir unter welchen Voraussetzungen das Modellverhalten eines Testmodells deterministisch bzw. eingabevollständig ist:

Definition 4.6 (Determinismus und Eingabevollständigkeit). *Ein Modell S heißt deterministisch genau dann, wenn für jede Eingabe aus dem Eingabebereich genau eine Ausgabe existiert:*

$$\forall (i, o), (\tilde{i}, \tilde{o}) \in R_S \text{ mit } i = \tilde{i} \text{ gilt } o = \tilde{o}. \tag{4.4}$$

Ein Modell S heißt eingabevollständig genau dann, wenn der Eingabebereich von S alle Ströme über die Eingabekanäle umfasst:

$$\mathsf{in}.S = \vec{I} \tag{4.5}$$

4.2. Beschreibung von Modellverhalten mit Zustandsmaschinen

In diesem Abschnitt führen wir Zustandsmaschinen zur operationellen Beschreibung von Modellverhalten ein. Zustandsmaschinen arbeiten gemäß der globalen Uhr in unserer Modellierungssprache. In einem Schritt lesen sie die Eingaben von den Eingabekanälen und berechnen in Abhängigkeit ihres lokalen Zustandes die Werte für die Ausgabekanäle und für den lokalen Folgezustand. Diese werden zum nächsten Schritt gesetzt. Dabei wird die Eingabe, die Ausgabe und der Zustand einer Zustandsmaschine durch die Belegung seiner Eingabekanäle, Ausgabekanäle bzw. lokalen Variablen beschrieben. Es folgt die formale Definition von Zustandsmaschinen:

Definition 4.7 (Zustandsmaschine). *Eine Zustandsmaschine Z wird definiert durch das Tupel*

$$Z = (I, O, L, Init, T).$$

Hierbei bezeichnen I, $O \subset C$ die endlichen Mengen der Ein- bzw. Ausgabekanäle, $L \subset VAR$ die endliche Menge der lokalen Variablen, $Init$ das Prädikat zur Beschreibung der Menge von Startzuständen und T die Transitionsrelation der Zustandsmaschine Z.

Die Menge aller Variablen einer Zustandsmaschine wird definiert durch $V := I \cup O \cup L$.

Die Zustandsmaschine startet in einem Zustand $\alpha \in B_V$ mit $\alpha \models Init$ und $Init$ hat die Eigenschaft nur den Wert der Variablen $L \cup O$ einzuschränken, d.h.

$$\forall \alpha, \beta \in B_V \text{ mit } \alpha \models Init \text{ und } \alpha \overset{L \cup O}{=} \beta \text{ gilt } \beta \models Init \tag{4.6}$$

Die Transitionsrelation $T \in B_V \times B_V$ hat folgende Eigenschaft: für alle Transitionen $(\alpha, \beta) \in T$ schränkt der Ausgangszustand α nur die Variablenwerte von den lokalen Variablen L und den Eingabekanälen I ein. Dagegen schränkt der Folgezustand β lediglich die Variablenwerte von den lokalen Variablen L und den Ausgabekanälen O ein:

$$\forall (\alpha, \beta) \in T \, \forall \gamma, \delta \in B_V \text{ mit } \gamma \overset{L \cup I}{=} \alpha \text{ und } \delta \overset{L \cup O}{=} \beta \text{ gilt } (\gamma, \delta) \in T \tag{4.7}$$

Um den Zusammenhang zwischen einer Zustandsmaschine und seinem Modellverhalten herzustellen, führen wir zunächst den Begriff des Ablaufs einer Zustandsmaschine ein:

Definition 4.8 (Ablauf einer Zustandsmaschine). *Sei eine Zustandsmaschine $Z = (I, O, L, Init, T)$ gegeben. Ein Ablauf ist ein Strom $\eta = \langle \alpha_1, \alpha_2, \alpha_3, \ldots \rangle \in B_V^\omega$ von Belegungen mit den Eigenschaften:*

1. *Die Zustandsmaschine startet in einem Startzustand, d.h. es gilt*

$$\eta.1 \models Init \text{ und}$$

2. *zwei aufeinander folgende Zustände in einem Ablauf bilden eine Transition der Zustandsmaschine, d.h.*

$$\forall k \in \mathsf{dom}.\eta \setminus \{1\} \text{ gilt } (\alpha_{k-1}, \alpha_k) \in T.$$

Die Menge aller Abläufe einer Zustandsmaschine Z bezeichnen wir mit $\langle\langle Z \rangle\rangle$.

Nun haben wir alle benötigten Begriffe eingeführt, um das Modellverhalten einer Zustandsmaschine definieren zu können. Wir erhalten eine Ein-/Ausgabebeziehung, indem wir einen Ablauf auf die Ein-/Ausgabekanäle projizieren:

Definition 4.9 (Modellverhalten einer Zustandsmaschine). *Das Modellverhalten R_Z einer Zustandsmaschine $Z = (I, O, L, Init, T)$ ist definiert durch*

$$R_Z := \{(i,o) \in \vec{I} \times \vec{O} \,|\, \exists \eta \in \langle\langle Z \rangle\rangle \text{ mit } i.\iota.n = \eta.n.\iota \wedge o.\theta.n = \eta.n.\theta \text{ für } \iota \in I,\, \theta \in O,\, n \in \mathsf{dom}.\eta\,\}. \tag{4.8}$$

Analog wie beim Modellverhalten eines Modells bezeichnen wir den Eingabebereich der Zustandsmaschine Z mit $\mathsf{in}.Z := \{i \,|\, (i,o) \in R_Z\}$.

Mit Hilfe der folgenden Proposition zeigen wir, dass das oben definierte Modellverhalten einer Zustandsmaschine konsistent mit dem Verhaltensbegriff aus Abschnitt 4.1.2 ist:

Proposition 4.1. *Die Relation R_Z ist wohldefiniert, d.h. sie ist ein Modellverhalten gemäß Definition 4.5.*

Ferner geben wir Kriterien an, unter welchen Voraussetzungen eine Zustandsmaschine deterministisch bzw. eingabevollständig ist:

Definition 4.10 (Determinismus und Eingabevollständigkeit). *Eine Zustandsmaschine $Z = (I, O, L, Init, T)$ heißt deterministisch, wenn die Startbedingung Init und die Transitionsrelation T folgende Eigenschaften besitzen:*

1. *Das Prädikat Init legt die Belegung der Variablen L und O eindeutig fest, d.h.*

$$\forall \alpha, \beta \in B_V \text{ mit } \alpha \models Init \text{ und } \beta \models Init \text{ gilt } \alpha \overset{L \cup O}{=} \beta. \tag{4.9}$$

2. *Im Folgezustand einer Transition wird die Belegung der Variablen L und O eindeutig durch der Belegung der Variablen L und I festgelegt, d.h.*

$$\forall (\alpha_1, \beta_1), (\alpha_2, \beta_2) \in T \text{ mit } \alpha_1 \overset{L \cup I}{=} \alpha_2 \text{ gilt } \beta_1 \overset{L \cup O}{=} \beta_2 \tag{4.10}$$

Eine Zustandsmaschine $Z = (I, O, L, Init, T)$ heißt eingabevollständig, wenn ihre Transitionsrelation T folgende Eigenschaft hat:

$$\forall \alpha \in B_V \, \exists \beta \in B_V \text{ mit } (\alpha, \beta) \in T. \tag{4.11}$$

Die Verträglichkeit von Definition 4.10 mit Definition 4.6 zeigen wir durch die folgende Proposition:

Proposition 4.2 (Verträglichkeit). *Eine Zustandsmaschine Z mit den Eigenschaften (4.9) und (4.10) induziert ein Modellverhalten R_Z mit der Eigenschaft (4.5).*

Eine Zustandsmaschine Z mit der Eigenschaft (4.11) induziert ein Modellverhalten R_Z mit der Eigenschaft (4.5).

4.3. Komposition

In diesem Abschnitt definieren wir die Komposition von Modellverhalten bzw. von Zustandsmaschinen und zeigen, dass diese beiden Kompositionsbegriffe miteinander verträglich sind.

Komposition wird verwendet, um komplexe Modelle aus Teilmodellen zusammenzusetzen. Die Zerlegung eines komplexen Modells in Teilmodelle führt zur Komplexitätsreduktion, da die Komplexität des Gesamtmodells auf mehrere Teilmodelle verteilt wird und diese relativ unabhängig voneinander entwickelt werden können. Die Komposition beliebig vieler Teilmodelle kann zur Komposition zweier Modelle reduziert werden, da die Komposition von n Modellen der $n-1$-maligen Komposition von jeweils zwei Modellen entspricht.

Wir betrachten also ab jetzt die Komposition von zwei Modellen. Zwei Modelle werden miteinander komponiert, indem Ein- und Ausgabekanäle der beiden Modelle miteinander verschaltet werden. Die Verschaltung gibt an, welche Ausgaben des einen Modells die Eingaben des anderen Modells enthält. In unserer Modellierungssprache wird ein Ausgabekanal des einen Modells mit einem Eingabekanal des anderen Modells verschaltet, wenn beide den gleichen Namen haben. Ferner muss aus Konsistenzgründen der Datentyp der beiden Kanäle identisch sein. Um eine eindeutige Zuordnung der Verschaltung von Kanälen zu ermöglichen, fordern wir Disjunktheit der Ausgabe- bzw. Eingabekanalmengen zwischen den beiden Modellen. Wir fassen in der folgenden Definition formal zusammen, wann zwei Modelle bezüglich ihrer Schnittstelle zueinander komponiert werden können, d.h. zueinander kompatibel sind, und geben in diesem Fall die Schnittstelle des komponierten Modells an.

Definition 4.11 (Kompatibilität und Komposition von Modellen). *Seien S und T Modelle mit den syntaktischen Schnittstellen (I_S, O_S) bzw. (I_T, O_T) und der Typisierung $type_S$ bzw. $type_T$. Die Modelle S und T heißen kompatibel, wenn folgendes gilt:*

1. *Die Eingabekanäle der beiden Modelle sind disjunkt, d.h. $I_S \cap I_T = \emptyset$.*
2. *Die Ausgabekanäle der beiden Modelle sind disjunkt, d.h. $O_S \cap O_T = \emptyset$.*
3. *Zu verschaltende Kanäle besitzen den gleichen Typ, d.h.*
   $\forall c \in (I_S \cup I_T) \cap (O_S \cup O_T)$ gilt $type_S(c) = type_T(c)$.

Sind die Modelle S und T kompatibel, dann wird das komponierte Modell mit $S \otimes T$ bezeichnet, werden die gleichnamigen Kanäle $C := (I_S \cup I_T) \cap (O_S \cup O_T)$ miteinander verschaltet und das komponierte Modell $S \otimes T$ erhält die syntaktische Schnittstelle (I, O) mit

$$I := (I_S \cup I_T) \setminus (O_S \cup O_T)$$
$$O := (O_S \cup O_T) \setminus (I_S \cup I_T).$$

Nach der syntaktischen Komposition zweier Modelle anhand ihrer Schnittstelle wenden wir uns ihrer semantischen Bedeutung zu. Das Modellverhalten eines komponierten Modells entsteht auf natürliche Weise aus dem Modellverhalten der beiden Teilmodelle: Abläufe aus dem Modellverhalten der beiden Modelle, die auf den verschalteten Kanälen identisch sind, werden miteinander verknüpft. Diese Festlegung fassen wir formal in der folgenden Definition und zeigen in der darauf folgenden Proposition, dass durch diese Verknüpfung wieder ein Modellverhalten gemäß Definition 4.5 entsteht.

Definition 4.12 (Komposition von Modellverhalten). *Seien S und T Modelle mit den syntaktischen Schnittstellen (I_S, O_S) bzw. (I_T, O_T), die zueinander kompatibel sind, und $R_S \subset \vec{I_S} \times \vec{O_S}$ bzw. $R_T \subset \vec{I_T} \times \vec{O_T}$ ihr Modellverhalten. Dann definiert für die Relation*

$$\begin{aligned} R_{S\otimes T} := \{(i,o) \in \vec{I} \times \vec{O} \mid \exists (i_S, o_S) \in R_S, (i_T, o_T) \in R_T \text{ mit } \#i_S = \#i_T \wedge \\ i = (i_S \oplus i_T)|_I \wedge o = (o_S \oplus o_T)|_O \wedge (i_S \oplus i_T)|_C = (o_S \oplus o_T)|_C\} \end{aligned} \tag{4.12}$$

das Modellverhalten des komponierten Modells $S \otimes T$ mit verschalteten Kanälen C und syntaktischer Schnittstelle (I, O) aus Definition 4.11.

Proposition 4.3 (Wohldefiniertheit der Komposition). *Das Modellverhalten $R_{S\otimes T}$ (4.12) aus Definition 4.12 ist wohldefiniert bezüglich Definition 4.5.*

Nachdem wir den Kompositionsbegriff auf dem deskriptiven Modellverhalten eingeführt haben, definieren wir einen entsprechenden Kompositionsbegriff auf operationellen Zustandsmaschinen und zeigen, dass dieser Kompositionsbegriff verträglich mit dem Begriff auf dem Modellverhalten ist.

Proposition 4.4 (Komposition von Zustandsmaschinen).
Seien $Y = (I_Y, O_Y, L_Y, Init_Y, T_Y)$ und $Z = (I_Z, O_Z, L_Z, Init_Z, T_Z)$ Zustandsmaschinen mit kompatiblen Schnittstellen gemäß Definition 4.11. Wir definieren die Zustandsmaschine $(Y \otimes Z) := (I, O, L, Init, T)$ mit

$$\begin{aligned} I &:= (I_Y \cup I_Z) \setminus (O_Y \cup O_Z) \\ O &:= (O_Y \cup O_Z) \setminus (I_Y \cup I_Z) \\ L &:= L_Y \cup L_Z \cup C \\ Init &:= Init_Y \wedge Init_Z \\ T &:= \{(\alpha, \beta) \in B_V \mid \exists (\alpha_Y, \beta_Y) \in T_Y, (\alpha_Z, \beta_Z) \in T_Z \text{ mit} \\ &\qquad \alpha \overset{L_Y \cup C}{=} \alpha_Y \wedge \alpha \overset{L_Z \cup C}{=} \alpha_Z \wedge \\ &\qquad \beta \overset{L_Y \cup C}{=} \beta_Y \wedge \beta \overset{L_Z \cup C}{=} \beta_Z\}, \end{aligned} \tag{4.13}$$

wobei $C := (I_Y \cup I_Z) \cap (O_Y \cup O_Z)$ *und* $V := I \cup O \cup L$ *gilt. Dann ist die Zustandsmaschine* $(Y \otimes Z)$ *gemäß Definition 4.7 wohldefiniert und es gilt*

$$R_{(Y \otimes Z)} = R_Y \otimes R_Z.$$

Zwei Zustandsmaschinen werden also komponiert, indem sie einen gemeinsamen Datenraum bilden und die Transitionen der Zustandsmaschinen über die verschalteten Kanäle verknüpft bzw. synchronisiert werden. Dabei werden die verschalteten Kanäle zu lokalen Variablen der komponierten Zustandsmaschine.

4.4. Beschreibung von Zustandsmaschinen durch Regelsysteme

Im Allgemeinen werden Zustandsmaschinen durch unendlich viele Transitionen beschrieben. Um eine endliche Darstellung von Zustandsmaschinen zu ermöglichen, führen wir Regelsysteme zur kompakten Beschreibung von Zustandsmaschinen ein. Die wesentliche Idee besteht darin, mit Hilfe einer endlichen Menge von Verhaltensregeln, die potenziell unendlich vielen Transitionen auf dem Datenraum B_V der Zustandsmaschine zu beschreiben. Eine Verhaltensregel besteht aus vier Teilen, einer Vorbedingung, einem Eingabemuster, einer Ausgabe und einer Zuweisung. Das Eingabemuster beschreibt eine Bedingung, welche die Belegung der Eingabekanäle erfüllen muss, damit die Regel schalten kann. Wenn die Eingabe zu den Mustern passt, dann werden ggf. so genannte regellokale Variablen gebunden, die in der Vorbedingung, der Ausgabe und der Zuweisung verwendet werden können. Die Vorbedingung beschreibt, welche Bedingung die lokalen Variablen und die regellokalen Variablen erfüllen müssen, damit die Regel ausgeführt werden kann. Die Ausgabe und Zuweisung ordnet bei Ausführung der Regel den Ausgabekanälen bzw. den lokalen Variablen Werte zu. Sind die Eingabemuster und die Vorbedingung einer Regel erfüllt, dann wird sie als schaltbereit bezeichnet. Sind im einem Zustand mehrere Regeln schaltbereit, dann wird eine nichtdeterministisch ausgewählt und ausgeführt. Es folgt die formale Definition eines Regelsystems:

Definition 4.13 (Regelsystem). *Ein Regelsystem* G *wird definiert durch das Tupel*

$$G = (I, O, L, Init, R).$$

Hierbei gelten für I*,* O*,* L *und Init die gleichen Bedingungen wie für Zustandsmaschinen (siehe Definition 4.7).*

R *ist eine endliche Menge von Verhaltensregeln. Jede Regel* $r \in R$ *hat die Form* $r = (pre, input, output, assign)$*, wobei pre, input, output und assign Prädikate sind und folgende Eigenschaften haben:*

1. *Das Prädikat input ist eine Konjunktion von Gleichungen, durch die ausgedrückt wird, welchen Mustern die Belegungen der Eingabekanäle genügen müssen, d.h.*

input hat die Form

$$input \equiv \bigwedge_{i \in J} i \cong d_i,$$

wobei $J \subset I$ gilt und $\cong$ die matching-Gleichheit zwischen dem Eingabemuster d_i von Typ $type(i)$ und der aktuellen Eingabe i bezeichnet ($i \in J$). Die ungebundenen Variablen im Eingabemuster heißen regellokal. Die regellokalen Variablen einer Regel sind definiert durch

$$loc := \bigcup_{i \in J} \mathsf{free}.d_i$$

und sind disjunkt von den Variablen des Regelsystems, d.h. $loc \cap V = \emptyset$.

2. *Das Prädikat pre definiert die Vorbedingung über den lokalen Variablen L und den regellokalen Variablen loc, d.h. $\mathsf{free}.pre \subset L \cup loc$.*

3. *Das Prädikat output ist eine Konjunktion von Gleichungen, die die Belegung der Ausgabekanäle im nächsten Zustand in Abhängigkeit der lokalen und regellokalen Variablen definiert, d.h. output hat die Form*

$$output \equiv \bigwedge_{o \in O} o' = e_o$$

wobei die Terme e_o nach der Ersetzung ihrer ungebundenen Variablen durch ihre aktuelle Belegung Terme von $type(o)$ ergeben und $\mathsf{free}.e_o \subset L \cup loc$ gilt ($o \in O$).

4. *Das Prädikat assign ist eine Konjunktion von Gleichungen, die die Belegung der lokalen Variablen im nächsten Zustand in Abhängigkeit der lokalen und regellokalen Variablen definiert. assign hat die Form*

$$assign \equiv \bigwedge_{l \in L} l' = f_l.$$

wobei die Terme f_l nach der Ersetzung ihrer ungebundenen Variablen durch ihre aktuelle Belegung Terme vom Typ $type(l)$ ergeben und $\mathsf{free}.f_l \subset L \cup loc$ gilt ($l \in L$).

Wir bezeichnen die Vorbedingung, die Eingabemuster, die Ausgabe, die Zuweisung und regellokale Variablen einer Regel $r \in R$ mit pre_r, $input_r$, $output_r$, $assign_r$ bzw. loc_r.

Eine Regel $r \in R$ heißt schaltbereit unter Belegung $\alpha \in B_V$ genau dann, wenn eine Belegung $\tilde{\alpha} \in B_{loc_r}$ mit $\alpha \oplus \tilde{\alpha} \models pre_r \wedge input_r$ existiert. Wir definieren den Schaltbereich $\mathsf{en}.r \subset B_V$ einer Regel $r \in R$ durch

$$\mathsf{en}.r := \{\alpha | \, \exists \tilde{\alpha} \in B_{loc_r} \text{ mit } \alpha \oplus \tilde{\alpha} \models \ pre_r \wedge input_r\}.$$

Die folgende Proposition stellt den Zusammenhang zu Zustandsmaschinen her, indem wir der Regelmenge eines Regelsystems eine Transitionsrelation einer Zustandsmaschine zuordnen:

Proposition 4.5 (Zustandsmaschine eines Regelsystems).
Sei $G = (I, O, L, Init, R)$ ein Regelsystem und $T_R \subset B_V \times B_V$ die Relation definiert durch

$$T_R := \{(\alpha, \beta) | \exists r \in R, \tilde{\alpha} \in B_{loc_r} \text{ mit } \alpha \oplus \tilde{\alpha} \models pre_r \wedge input_r \text{ und } \alpha \oplus \tilde{\alpha}, \beta' \models output_r \wedge assign_r\}. \tag{4.14}$$

Dann ist $Z_G := (I, O, L, Init, T_R)$ eine Zustandsmaschine gemäß Definition 4.7.

Durch die Abbildung eines Regelsystems zu einer Zustandsmaschine wird einem Regelsystem das Modellverhalten R_{Z_G} zugeordnet. Die folgende Proposition charakterisiert, unter welchen Voraussetzungen ein Regelsystem ein deterministisches bzw. eingabevollständiges Modellverhalten induziert.

Proposition 4.6 (Determinismus und Eingabevollständigkeit).
Ein Regelsystem $G = (I, O, L, Init, R)$ heißt deterministisch, wenn die Startbedingung Init und die Regelmenge R folgende Eigenschaften besitzen:

1. *Das Prädikat Init legt die Belegung der Variablen L und O eindeutig fest, d.h.*

$$\forall \alpha, \beta \in B_V \text{ mit } \alpha \models Init \text{ und } \beta \models Init \text{ gilt } \alpha \overset{L \cup O}{=} \beta. \tag{4.15}$$

2. *Die Schaltbereiche zweier Regeln sind paarweise disjunkt, d.h.*

$$\forall r, s \in R \text{ gilt } \mathsf{en}.r \cap \mathsf{en}.s = \emptyset. \tag{4.16}$$

Diese Festlegung ist konsistent mit Definition 4.10, da ein Regelsystem G mit den Eigenschaften (4.15) und (4.16) eine Zustandsmaschine Z_G mit den Eigenschaften (4.9) und (4.10) induziert.

Ein Regelsystem $G = (I, O, L, Init, R)$ heißt eingabevollständig, wenn die Regelmenge R folgende Eigenschaft hat:

$$\forall \alpha \in B_V \, \exists r \in R \text{ mit } \alpha \in \mathsf{en}.r. \tag{4.17}$$

Diese Festlegung ist wiederum konsistent mit Definition 4.10, da ein Regelsystem G mit der Eigenschaft (4.17) eine Zustandsmaschine Z_G mit der Eigenschaft (4.11) induziert.

4.5. Tabellarische Darstellung

In diesem Abschnitt führen wir eine syntaktische Darstellung von Regelsystemen in Form von Tabellen zur Beschreibung des Verhaltens eines Modells ein. Tabellen dienen zur Spezifikation von Verhaltensmodellen und bieten durch ihren strukturierten Aufbau und die explizite Darstellung aller Verhaltensregeln eine detaillierte Übersicht über ein Verhaltensmodell. Die explizite Darstellung aller Regeln frei von graphischem und syntaktischem Zucker ermöglicht eine zügige Navigation im Modell und einen schnellen Zugriff auf die Regelbestandteile. Dadurch unterstützt die tabellarische Darstellungsform eines Regelsystems die zügige Entwicklung eines Verhaltensmodells.

4.5.1. Notation der tabellarischen Darstellung

Gemäß unserer Modellierungssprache wird das Verhalten eines Modells durch eine globale Uhr getrieben. Ein Modell führt einen Schritt durch, indem es in Abhängigkeit der Belegung seiner Eingabekanäle und seiner lokalen Variablen die Belegung der lokalen Variablen und der Ausgabekanäle berechnet. Die grundlegende Idee ist, das Verhalten eines Modells operationell mit Hilfe einer Tabelle zu beschreiben. Jede Zeile entspricht einer Verhaltensregel, die in Abhängigkeit von Eingabe und lokalem Datenzustand ausgeführt wird. Eine Regel bzw. ein Zeileneintrag der Tabelle ist unterteilt in fünf Spalten:

- *Name* bezeichnet den Namen der Verhaltensregel.
- *Pre* beschreibt die Vorbedingung der Verhaltensregel. Die Vorbedingung ist ein Prädikat über dem lokalen Datenzustand und gegebenenfalls regellokalen Variablen.
- *Input* beschreibt das Eingabemuster, welches von den Eingabekanälen gelesen wird.
- *Output* beschreibt die Ausgabe, welche nach Schalten der Regel auf die Aufgabekanäle geschrieben wird.
- *Assign* beschreibt durch Zuweisungen, welcher lokaler Datenzustand nach Schalten der Regel eingenommen wird.

Die Notation, die wir zur Beschreibung der Vorbedingungen, Eingabe, Ausgabe und Zuweisung verwenden, entspricht der von AutoFocus, die wir hier kurz wiederholen. Für eine ausführliche Einführung verweisen wir auf [LPS+02]. Das Eingabemuster *Input* wird notiert als eine Liste von Ausdrücken von:

$$i_1?d_1; \ldots; i_n?d_n$$

wobei $i_k \in I$ jeweils ein Eingabekanal und d_k ein Muster passend zum Typ $type(i_k)$ ist $(1 \leq k \leq n)$. Eine Regel kann nur schalten, wenn die aktuelle Belegung der Eingabekanäle zu den definierten Eingabemustern passt. In diesem Fall werden ggf. vorhandene freie Variablen der Ausdrücke d_k durch Patternmatching zu späteren Verwendung in der Regel gebunden. Diese Variablen heißen regellokal. Weiterhin muss zur Schaltbereitschaft der Regel deren Vorbedingung *Pre* erfüllt sein. Diese ist ein Prädikat über lokale Variablen und regellokale Variablen. Schaltet eine Regel, dann produziert sie Ausgaben entsprechend der Liste von Ausdrücken, die durch *Output* beschrieben werden:

$$o_1!e_1; \ldots; o_m!e_m$$

wobei $o_k \in O$ jeweils einen Ausgabekanal bezeichnet und e_k ein Ausdruck über regellokalen und lokalen Variablen vom Typ $type(o_k)$ ist $(1 \leq k \leq m)$. Weiterhin wird beim Schalten der Regel der neue lokale Zustand berechnet. Dieser wird mit Hilfe einer Liste von Ausdrücken, die in *Assign* notiert werden, berechnet und zugewiesen:

$$l_1 = f_1; \ldots; l_p = f_p$$

wobei $l_k \in L$ jeweils eine lokale Variable bezeichnet und f_k ein Ausdruck über regellokale und lokale Variablen vom Typ $type(l_k)$ ist $(1 \leq k \leq p)$.

Eine Tabelle zusammen mit der Definition der lokalen Variablen inklusive ihrer Initialisierung und der Definition ihrer Schnittstelle ist eine eindeutige und formale Darstellung eines Regelsystems. Nach dem folgendem Beispiel geben wir die Semantik der tabellarischen Darstellung als Regelsystem an.

Beispiel 4.1 (Verhalten eines Stapels). *Wir beschreiben das reaktive Verhalten eines Stapels mit Hilfe der Tabelle 4.1. Zunächst definieren wir einige Datentypen und einfache Funktionen darauf, die für die Verhaltensbeschreibung nötig sind. Dazu verwenden wir die funktionale Programmiersprache* QuestF *[BLS00] von* AutoFocus *[HSE97]. Für eine kurze Einführung in* AutoFocus *verweisen wir auf Abschnitt 7.2 in dieser Arbeit, für eine ausführliche auf das* AutoFocus*-Tutorial [LPS+02]:*

```
data dInput = push(dData) | get | pop;
data dData  = item1 | item2 | item3;
data dList  = empty | List(dData, dList);

isE: dList -> Bool;
fun isE(empty) = True
  | isE(List(_,_)) = False;

ft: dList -> dData;
fun ft(List(head,_)) = head;

rt: dList -> dList;
fun rt(List(_,tail)) = tail;
```

Wir deklarieren den Eingabekanal `e`*, den Ausgabekanal* `a` *und die lokale Variable* `st` *des Stapels. Diese Information wird in* AutoFocus *aus Strukturdiagrammen (vgl. Abschnitt 7.2.1) abgeleitet oder alternativ –wie hier– textuell angegeben:*

```
inputchannel  dInput e;
outputchannel dData  a = epsilon;
localvariable dList  st = empty;
```

Das Verhalten des Stapels wird nun mit folgender Tabelle beschrieben:

Name	Pre	Input	Output	Assign
pushItem	True	e?push(DATA)		st=List(DATA,st)
getItem	not(isE(st))	e?get	a!ft(st)	
popItem	not(isE(st))	e?pop		st=rt(st)
idle	True	e?ϵ		

Tabelle 4.1.: Verhalten eines Stapels: tabellarische Darstellung

Die Regel **pushItem** *dient zum Ablegen eines Elements auf den Stapel. Liegt im Eingabekanal* **e** *das Muster* **push(DATA)** *vor, dann wird der Wert des Parameters der Nachricht* **push** *an die regellokale Variable* **DATA** *gebunden und mittels der Zuweisung in der Spalte* **Assign** *in der lokalen Variablen* **st** *abgelegt. Die Regeln* **getItem** *und* **popItem** *beschreiben, wie mittels der Nachrichten* **get** *und* **pop** *das oberste Element des Stapel ausgelesen bzw. entfernt werden kann. Abschließend definiert die Regel* **idle** *wie sich der Stapel verhält, wenn keine Nachricht auf Kanal* **e** *anliegt.*

Mit diesen Verhaltensregeln können Abläufe produziert werden, die zum Test einer entsprechenden Stapelimplementierung verwendet werden können. Man beachte, dass dieses Verhaltensmodell nicht eingabevollständig ist, da keine Reaktion auf den leeren Stapel für die Operationen **get** *und* **pop** *vorgesehen ist. Entsprechende Abläufe können also mit diesem Verhaltensmodell nicht erzeugt werden und folglich nicht getestet werden.*

4.5.2. Semantik der tabellarischen Darstellung

Die Semantik einer Tabelle geben wir an, indem wir einer Tabelle ein Regelsystem $G = (I, O, L, Init, R)$ zuordnen:

- Die Variablen Mengen I, O und L werden eindeutig durch den Deklarationsteil der Tabelle definiert.
- Das Prädikat *Init* folgt ebenfalls aus dem Deklarationsteil der Tabelle und hat demzufolge die Form einer Konjunktion von Gleichungen:

  $$Init \equiv \bigwedge_{v \in O \cup L} v = t_v$$

  wobei t_v ein Term vom Typ $type(v)$ $(v \in O \cup L)$ ist.
- Jede Zeile der Tabelle wird in eine Regel des Regelsystem überführt: eine Zeile

  $$name : \ pre : \ i_1?d_1; \ldots; i_n?d_n : \ o_1?e_1; \ldots; o_m?e_m : \ l_1?f_1; \ldots; l_p?f_p$$

  wird in die Regel

  $$(pre, \bigwedge_{k=1}^{n} i_k \cong d_k, \bigwedge_{k=1}^{m} o'_k = e_k \wedge \bigwedge_{o \in \tilde{O}} o' = \epsilon, \bigwedge_{k=1}^{p} l'_k = f_k \wedge \bigwedge_{l \in \tilde{L}} l' = l)$$

  übersetzt, wobei die Ausgabekanäle $\tilde{O} := O \backslash \{o_1, \ldots, o_m\}$ und die lokalen Variablen $\tilde{L} := L \backslash \{l_1, \ldots, l_p\}$, deren Wert in der Tabellenzeile nicht explizit angegeben wurde, mit Grundwert ϵ versehen werden bzw. unverändert bleiben.

Diese Übersetzung verdeutlichen wir mit folgendem Beispiel:

Beispiel 4.2 (Stapel als Regelsystem). *Gemäß obiger Vorschrift ergibt die tabellarische Beschreibung des Stapels aus Beispiel 4.1 das Regelsystem* $G = (I, O, L, Init, R)$ *mit* $I = \{e\}$, $O = \{a\}$, $L = \{st\}$, $Init \equiv a = \epsilon \wedge st = empty$ *und*

$$
\begin{array}{lllll}
R = \{ (& True & , \; e \cong push(DATA) & , \; a' = \epsilon & , \; st' = List(DATA, st) \;), \\
(& \neg isE(st) & , \; e \cong get & , \; a' = ft(st) & , \; st' = st \;), \\
(& \neg isE(st) & , \; e \cong pop & , \; a' = \epsilon & , \; st' = rt(st) \;), \\
(& True & , \; e \cong \epsilon & , \; a' = \epsilon & , \; st' = st \;) \; \}
\end{array}
$$

Abbildung 4.2 zeigt einige Abläufe über die Schnittstelle des Stapels in tabellarischer Form. Tabelle 4.2(a) zeigt einen trivialen Ablauf des Stapels, bei dem immer die Regel

Kanal e	Kanal a
ϵ	ϵ
ϵ	ϵ
ϵ	ϵ
ϵ	ϵ

(a)

Kanal e	Kanal a
push(item1)	ϵ
get	ϵ
pop	item1
ϵ	ϵ

(b)

Kanal e	Kanal a
push(item1)	ϵ
get	ϵ
pop	item1
pop	ϵ

(c)

Tabelle 4.2.: Beispielabläufe zum Stapel

`idle` *schaltet.*

In Ablauf 4.2(b) liegt zum ersten Schritt die Nachricht `push(item1)` *an und entsprechend schaltet die Regel* `pushItem`*. Gemäß der Regel wird das Element* `item1` *in der lokalen Variablen abgelegt und es findet keine Ausgabe statt. Im zweiten Schritt wird mit Nachricht* `get` *das oberste Element des Stapels ausgelesen. Dabei schaltet Regel* `getItem`*, deren Ausgabe* `item1` *einen Schritt verzögert, d.h. im viertem Schritt erscheint. Im vierten Schritt wird mit Nachricht* `pop` *die Regel* `popItem` *ausgelöst, welche das oberste Element des Stapels entfernt. Dessen zugehörige leere Ausgabe erscheint im fünften Schritt.*

Ablauf 4.2(c) ist in den ersten 4 Schritten identisch mit Ablauf 4.2(b). Im fünften Schritt liegt erneut die Nachricht `pop` *an. Zu diesem Zeitpunkt ist jedoch die variable* `st` *mit dem Wert* `empty` *belegt, d.h. der Stapel ist leer. Entsprechend dem Regeln des Stapels (vgl. Tabelle 4.1) gibt es also keine Regel, die diese Nachricht verarbeiten kann. Folglich gehört Ablauf 4.2(c) nicht zu den spezifizierten Abläufen des Stapels.*

4.6. Graphische Darstellung

In diesem Abschnitt führen wir eine syntaktische Darstellung von Regelsystemen in Form eines Graphen zur Beschreibung eines Verhaltensmodells ein. Diese Darstellungsform entspricht den *State Transition Diagrams (STDs)* von AutoFocus (vgl. Abschnitt 7.2.2). Die Graphendarstellung bietet eine abstrahierte Sicht auf ein Regelsystem, indem sie Realisierungsdetails von Verhaltensregeln graphisch verbirgt und den abstrakten Kon-

trollfluss anhand von ausgewählten Kontrollzuständen darstellt. Dadurch ermöglicht diese Darstellungsform das zügige Erfassen des Kontrollflusses und unterstützt bei der Simulation des Verhaltensmodells das schnelle Einschätzen des aktuellen Modellzustands.

4.6.1. Notation der graphischen Darstellung

Eine weit verbreitete Darstellungsform von Zustandsmaschinen sind gerichtete Graphen. Kontrollzustände werden als Knoten und Zustandsübergänge als Kanten repräsentiert. Vor allem für kleine Modelle sind Kontrollzustandsgraphen eine kompakte und intuitive Möglichkeit das Verhalten eines Modells zu modellieren. Die Variante von Kontrollzustandsgraphen, die wir verwenden, entspricht den State Transition Diagrams (STD) von AutoFocus. Der Startkontrollzustand wird mit einem schwarz gefüllten Kreis markiert. Die Transition und die Zustandsübergänge sind wie die Verhaltensregeln der Tabellendarstellung aufgebaut. Wie bei den Tabellen kann ein Zustandsübergang als Verhaltensregel eines Regelsystems aufgefasst werden.

Beispiel 4.3 (Der Stapel als graphischer Kontrollzustandsgraph). *Wir stellen in Abbildung 4.2 den Stapel aus Beispiel 4.1 als Kontrollzustandsgraph dar. Von dem Zustandsübergängen wird nur der Name angezeigt, Vorbedingung, Eingabemuster, Ausgabe und Zuweisung verbleiben verborgen und können Tabelle 4.1 entnommen werden.*

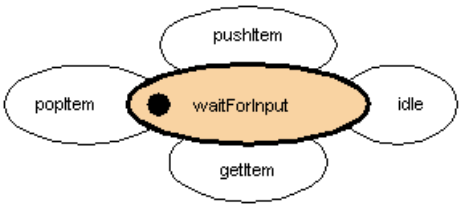


Abbildung 4.2.: Verhalten des Stapels in graphischer Darstellung

4.6.2. Interpretation der graphischen Darstellung

Ein Kontrollzustandsgraph kann einfach in die tabellarische Darstellung aus Abschnitt 4.5 überführt werden, indem die Kontrollzustände im Kontrollzustandsgraphen als Werte einer lokalen Variable *cstate* in der Tabelle aufgefasst werden. Die Übergänge des Kontrollzustandsgraphen werden zu Verhaltensregeln in der Tabelle, indem Vorbedingung und Zuweisung eines Zustandsübergangs gemäß seinem Quell- bzw. Zielkontrollzustand erweitert werden. Wir verdeutlichen dies am Beispiel des Stapels:

Beispiel 4.4 (Übergang Kontrollzustandsgraph – Tabelle). *Für die Kontrollzustände des Kontrollzustandsgraphen des Stapels aus Beispiel 4.3 wird der Datentyp* `dStates` *eingeführt:*

```
data dStates = waitForInput;
```

Die Deklaration von Eingabekanälen, Ausgabekanälen und lokalen Variablen wird um die lokale Variable `cstate` *erweitert:*

```
inputchannel  dInput  e;
outputchannel dData   a = epsilon;
localvariable dList   st = empty;
              dStates cstate = waitForInput;
```

Gemäß der Kontrollzustandsübergangsrelation im Graphen werden die Zustandsübergänge zu Verhaltensregeln erweitert und ergeben folgende Tabelle:

Name	Pre	Input	Output	Assign
pushItem	cstate==waitForInput	e?push(DATA)		st=List(DATA,st); cstate=waitForInput
getItem	cstate==waitForInput && not(isE(st))	e?get	a!ft(st)	cstate=waitForInput
popItem	cstate==waitForInput && not(isE(st))	e?pop		st=rt(st); cstate=waitForInput
idle	cstate==waitForInput	e?ϵ		cstate=waitForInput

Tabelle 4.3.: Kontrollzustandsgraph als Tabelle

4.7. Diskussion

In diesem Abschnitt diskutieren wir die methodischen Vor- und Nachteile bei der Verwendung der tabellarischen bzw. graphischen Darstellung von Regelsystemen. Im Anschluss setzen wir uns mit verwandten Arbeiten auseinander.

4.7.1. Diskussion der Darstellungsformen

Erfahrungen mit der graphischen Darstellung In AutoFocus wird derzeit die Modellierung von Modellverhalten mit Hilfe von graphischen Kontrollzustandsgraphen (STDs) unterstützt, deren Knoten die Kontrollzustände und deren Kanten die Übergänge zwischen den Kontrollzuständen repräsentieren (vgl. Abbildung 4.2). Bisher wurden zwei Fallstudien [PPS+03, PPW+05] von industrieller Komplexität mit AutoFocus und dem Zweck modelliert, die Modelle zum modellbasierten Test entsprechender im Industrieeinsatz befindlicher Implementierungen zu verwenden. In beiden Projekten wurden

inkrementell sehr komplexe STDs entwickelt, bei denen ein großer Teil der Komplexität in den funktionalen QuestF-Programmen zur Modellierung der Vorbedingungen und Zuweisungen der STD-Transitionen liegt. Dabei wurde von den Entwicklern unabhängig voneinander die Erfahrung gemacht, dass die Entwicklung von STDs in graphischer Darstellung folgende Nachteile hat:

- Zu Beginn der Entwicklung ist das komplexe Modellverhalten noch nicht verstanden. Folglich sind die benötigten Kontrollzustände unbekannt bzw. ist eine geeignete Wahl der Kontrollzustände noch nicht möglich. Da aber die Kontrollzustände ein zentrales Modellierungsmittel der STDs sind und da die Verhaltensentwicklung eng mit einem Lernprozess über das Modellverhalten selbst verknüpft ist, führt dies gerade in der Anfangsphase der Entwicklung zu aufwendigen Änderungen in den STDs, die die zügige Weiterentwicklung stark behindern.
- In der Regel sind Kontrollzustandsübergänge mit sehr komplexen Vorbedingungen und Zuweisungen ausgestattet. Um die Übersichtlichkeit der STDs so gut wie möglich aufrechtzuerhalten, wird die Komplexität eines Übergangs durch seinen Namen in der graphischen Darstellung überdeckt. Dieser Umstand führt dazu, dass der Entwickler bei der großen Anzahl der Übergänge schnell den Überblick über deren genauen Inhalt verliert. Folglich werden bei der inkrementellen Weiterentwicklung viele Fehler eingebaut, die später mühsam entfernt werden müssen.

Vorteile der tabellarischen Darstellung Diese Schwierigkeiten bei der Modellierung am Beginn der Modellbildung führte zum Einsatz von Tabellen, wie in Abschnitt 4.5 beschrieben. In der Tabelle modelliert eine Zeile, die als Verhaltensregel bezeichnet wird, eine Menge von Transitionen auf einem Datenzustandsraum, der durch die Belegungen der lokalen Variablen aufgespannt wird. Die Zeilen entsprechen also den Kontrollzustandsübergängen in den STDs. Dies führt zu einer Konzentration und schrittweisen Modellierung von bekannten Übergängen durch Verhaltensregeln anstatt der unbekannten Kontrollzustände und überwindet damit die oben angeführten Schwierigkeiten. Ferner wird durch die tabellarische Darstellung erreicht, dass alle Übergänge explizit, vollständig und übersichtlich angeordnet dargestellt werden. Daraus ergeben sich folgende Vorteile für den Modellierungsprozess:

- Informelle Regeln der Form "Tritt in der Situation x das Ereignis y auf, dann muss die Aktionen z ausgeführt oder angestoßen werden" können direkt in Verhaltensregeln in der Tabelle umgesetzt werden. Die Situation x wird geeignet in den Datenraum abgebildet, das Eintreten des Ereignisses y mündet in ein entsprechendes Eingabemuster und die Aktionen z werden in den Ausgaben bzw. in der Zuweisung umgesetzt.
- Die explizite und vollständige Darstellung aller Regeln ermöglicht eine schnelle und einfache Navigation in der Modellierung und gestattet die schnelle Identifikation und das Durchführen von Transformationen, wie z.B. das Zusammenfassen von Regeln mit ähnlichen Vorbedingungen bzw. Eingabemustern. Ferner führt sie zu

einer Konzentration auf die Logik der Regeln anstatt auf deren graphischen Darstellung, d.h. der Entwickler verwendet seine Zeit ausschließlich auf die korrekte Modellierung der Regeln ansatt einen großen Teil davon während der Entwicklung auf das graphische Arrangieren von Kontrollzuständen und Transitionen.

- Der Überblick, der durch die Tabelle geschaffen wird, erleichtert Modellierungstätigkeiten wie
  - das Auffinden von Konflikten in Form von Nichtdeterminismus zwischen den Verhaltensregeln, d.h. der Entwickler kann schnell durch den Vergleich von Vorbedingungen und den zugehörigen Eingabemustern beurteilen, ob sich deren Eingabebereiche überschneiden,
  - das Identifizieren und Schließen von Lücken im Eingabebereich einer Gruppe von Regeln, d.h. der Entwickler deckt leicht durch den Vergleich ihrer Vorbedingungen und Eingabemuster logische Beziehungen auf, für noch keine Reaktion im Verhaltensmodell spezifiziert wurde, und
  - das Aufspüren von sinnvollen Kontrollzuständen, d.h. der Entwickler identifiziert ähnliche Muster in den Vorbedingungen der Regeln, die häufig geeignete Kandidaten zur Partitionierung des Datenraums in Kontrollzustände sind[1]. Wir verweisen dazu auf Kapitel 6, das ein Verfahren zur Transformation von Tabellen in Kontrollzustandsgraphen beschreibt.

Nachteile der tabellarischen Darstellung Die detaillierte und die rein textbasierte Darstellung in einer Tabelle führt dazu, dass ein abstrakter Kontrollfluss im Verhaltensmodell schwer erkennbar ist. Dies gilt insbesondere während der Simulation des Verhaltensmodell, da in diesem Fall der aktuelle Zustand des Modells mühsam durch den Review der Belegung aller lokalen Variablen erfasst werden muss. Dieser Nachteil wird durch die Transformation in einen Kontrollzustandsgraphen überwunden, da dieser eine abstrahierte Sicht auf den Kontrollfluss im Datenzustandsraum bietet (vgl. Kapitel 6).

Vorteile der graphischen Darstellung Durch das Verfahren, das wir in Kapitel 6 einführen, können tabellarische Regelsysteme werkzeuggestützt in Kontrollzustandsgraphen transformiert werden. Die graphische Darstellung in Form von Kontrollzustandsgraphen bietet gegenüber der detaillierten Darstellung in Form einer Tabelle folgende Vorteile:

- Die graphische Darstellung bietet durch die Abstraktion des lokalen Datenraums in Kontrollzustände und das Verdecken von Realisierungsdetails der einzelnen Übergänge eine abstrahierte und vereinfachte Sicht auf das Verhaltensmodell. Damit stehen im Vergleich zur Tabelle nicht die Übergänge, sondern die abstrakten Kontrollzustände und der grobe Kontrollfluss im Vordergrund. Die dadurch

[1] Ein Kontrollzustand entspricht einer Partition des lokalen Datenzustandsraums.

erreichte abstrahierte Sicht erleichtert das intuitive Verständnis und eine grobe Inspizierung des Modells.

- Zur Simulationszeit bietet die abstrahierte Sicht den Vorteil, dass zu jedem Zeitpunkt auf Modellebene der aktuelle Zustand abstrakt und schnell eingeschätzt werden und zur Validation mit entsprechenden Situationen der Realität verglichen werden kann. Ferner vereinfacht bzw. ermöglicht erst die Simulation der abstrahierten Sicht den gemeinsamen Review durch Modellentwickler und Entscheidungsträger, die nicht mit den Modellierungstechniken und den Modellierungsdetails vertraut sind.

Nachteile der graphischen Darstellung Die graphische Darstellung hat neben den in Paragraph "Erfahrungen mit der graphischen Darstellung" beschriebenen den Nachteil, dass komplexe Kontrollzustandsgraphen häufig viele Übergänge ohne Kontrollzustandswechsel enthalten ("Gänseblümcheneffekt"). Diese Übergänge sind nicht hilfreich für das Verständnis des Verhaltensmodells. Für Beispiele verweisen wir auf die Graphen in Abbildung 7.6 und 7.8(b).

Fazit Tabellen eignen sich durch ihre detaillierte Übersicht besser für das zügige Modellieren und Auffinden von Unstimmigkeiten im Modell während der Modellierung. Graphische Kontrollzustandsgraphen bieten durch ihre abstrahierte Sicht auf das Verhaltensmodell Vorzüge bei der Simulation, der Präsentation des Modells und beim gemeinsamen Review mit Nicht-Modellierungsspezialisten.

4.7.2. Verwandte Arbeiten

Zustandsmaschinen in FOCUS Die von uns eingeführte Variante von Zustandsmaschinen wurde inspiriert durch Breitling [Bre01] und Philipps [BP99], jedoch vereinfacht und für unsere Zwecke angepasst. Im Gegensatz zu unseren Arbeiten werden dort Zustandsmaschinen als eingabevollständig betrachtet und kommunizieren über ein asynchrones Ausführungsmodell. Wir definierten eine einfachere Variante, um einerseits die Sprache einfacher und verständlicher zu halten und andererseits durch das Werkzeug AUTOFOCUS nahezu direkt eine Werkzeugunterstützung zur Verfügung zu haben.

Tabellen Bekannt wurden Tabellen zur Spezifikation von Systemen durch Parnas u.a. [JPZ97, SZP96, PM91] und ihre Weiterentwicklung in der Methodik "Software Cost Reduction (SCR)" nach Heitmeyer u.a. [HLK95, HJL96], für die auch Werkzeugunterstützung verfügbar ist [HBGL95]. In diesem Ansatz existieren drei verschiedene Tabellentypen "Condition Tables", "Event Tables" und "Mode Transition Tables". Konzeptionell entspricht ein Mode einem Kontrollzustand in AUTOFOCUS. Folglich beschreibt eine Mode Transition Table genau die Übergänge, die zu einem Kontrollzustandswechsel führen. "Condition Tables" und "Event Tables" beschreiben in Abhängigkeit von dem

aktuellen Mode und Bedingungen an lokale Variablen bzw. dem Auftreten eines Ereignisse den Zustandswechsel einer Ausgabevariablen bzw. einer lokalen Variablen, wobei der Mode nicht verändert wird. Die Unterscheidung von verschiedenen Tabellenformen für ein Teilsystem und die daraus resultierende hohe Anzahl von Tabellen bei komplexeren Systemen führt zu unnötiger Komplexität, die durch die Beschreibungstechnik hervorgerufen wird, und zu Schwierigkeiten, die Tabellen konsistent zueinander zu halten. Gerade die Einfachheit und leichte Verständlichkeit ist einer der Vorteile unserer Tabellenform aus Abschnitt 4.5.

In Varianten von FOCUS [BRSS97, BS01] werden ebenfalls Tabellen benutzt, um das Verhalten von Komponenten zu spezifizieren. In dieser Tabellendarstellung wird im Vergleich zu unserer Tabellenform für jeden Kanal und jede lokale Variable eine eigene Spalte eingeführt. Mit dieser Tabellenvariante wurde auch in einer industriellen Fallstudie, die in Kapitel 7 beschrieben wird, experimentiert. Das Ergebnis war, dass die Tabellen aufgrund der vielen Ein-/Ausgabekanäle und lokalen Variablen sehr breit und unübersichtlich wurden und dass viele der Zellen der Tabelle leer sind, da meistens für die Spezifikation einer Verhaltensregel nur ein Teil der Kanäle bzw. Variablen benötigt wird. Dies und die Verwandtschaft zu AUTOFOCUS-STDs ist der Grund, warum wir die Tabellenform aus Abschnitt 4.5 gewählt haben.

4.8. Zusammenfassung

Der Zweck dieses Kapitels ist, die formalen Grundlagen für die Folgenden zu schaffen, indem wir eine Modellierungssprache einführen, die zur Entwicklung von Testmodellen geeignet ist. Der Beobachtungsbegriff unserer formalen Beschreibungstechnik beschreibt Modelle anhand der Menge ihrer Ein-/Ausgabeabläufe, die den potenziell erzeugbaren Testfälle eines Testmodells entsprechen. Wir fassen abschließend die grundlegenden Konzepte und deren Eigenschaften wie folgt zusammen:

- Unsere Modellsprache wird durch eine globale Uhr getrieben. Ein Modell S ist mit einer syntaktischen Schnittstelle (I, O) versehen und wird an dieser durch ihr Verhalten charakterisiert. Das Modellverhalten ist eine Relation $R_S \subset \vec{I} \times \vec{O}$, welche getaktete Eingabesequenzen $i \in \vec{I}$ mit getakteten Ausgabesequenzen $o \in \vec{O}$ in Beziehung setzt. Das Modellverhalten R_S wird als Menge aller Testfälle identifiziert, die durch ein Testmodell beschrieben werden. Testfallspezifikationen dienen dann zur Auswahl von Teilmengen des Modellverhaltens.
- Ein Modellverhalten hat die Eigenschaften, dass es abgeschlossen gegenüber seiner Eingabe, streng kausal und präfixabgeschlossen ist. Diese Eigenschaften sind trivialerweise erfüllt, wenn operationelle Techniken, wie zum Beispiel Zustandsmaschinen, zur Spezifikation von Modellverhalten verwendet werden.
- Ein Modellverhalten heißt deterministisch und eingabevollständig, wenn es jeder Eingabesequenz höchstens eine bzw. mindestens eine Ausgabesequenz zuordnet.

- Wir beschreiben das Modellverhalten mit Hilfe von Zustandsmaschinen. Zustandsmaschinen sind mit einem lokalen Datenzustand ausgestattet und berechnen Schritt für Schritt in Abhängigkeit von der Eingabe und ihrem lokalen Datenzustand eine Ausgabe und einen lokalen Folgedatenzustand. Ein solcher Schritt wird als Transition einer Zustandsmaschine bezeichnet. Das Modellverhalten einer Zustandsmaschine erhalten wir, indem wir die Abläufe einer Zustandsmaschine, d.h. Folgen von Transitionen bzw. von Belegungen ihrer lokalen Variablen, ihrer Ein- und ihrer Ausgabekanäle, auf Sequenzen ihrer Eingabe mit zugehörigen Ausgaben projizieren.
- Da sowohl die Transitionsmenge als auch das Modellverhalten einer Zustandsmaschine im allgemeinen unendlich viele Elemente enthält, führen wir Regelsysteme ein, die eine ggf. unendliche Menge von Transitionen einer Zustandsmaschine mit einer endlichen Menge von Regeln beschreiben. Eine Regel beschreibt dabei eine ganze Menge von Transitionen und besteht aus Prädikaten in spezieller Form, die Quell- und Zielbelegung von lokalen Variablen, Ein- und Ausgabekanälen beschreiben. Eine Regel zerfällt in die Bestandteile Vorbedingung, Eingabemuster, Ausgabe und Zuweisung. Eine Regel kann schalten, wenn die Belegung zu ihrem Eingabemuster passt und die Vorbedingung erfüllt ist. In diesem Fall wird die Belegung der Ausgabekanäle und der lokalen Variablen anhand Ausgabe bzw. Zuweisung der Regel berechnet. Einem Regelsystem ordnen wir eine Zustandsmaschine zu, indem zu jeder Regel die entsprechende Transitionsmenge berechnet wird.
- Syntaktische Repräsentationen von Regelsystemen sind Tabellen und Kontrollzustandsgraphen. In der tabellarischen Darstellung entspricht jeder Zeile bzw. in der graphischen jeder Kontrollzustandsübergang einer Regel des Regelsystems. Der Übergang zwischen den unterschiedlichen Darstellungsformen und dessen methodischer Nutzen ist Gegenstand von Kapitel 6.
- Die tabellarische hat Vorteile gegenüber der graphischen Darstellung aufgrund ihrer detaillierten Übersicht über alle Verhaltensregeln bei der Modellbildung und dem Aufdecken von Unstimmigkeiten beim Zusammenspiel der Regeln. Die graphische Darstellung dagegen eignet sich besser durch ihre abstrahierte Sicht auf das Verhaltensmodell für die Simulation, die Präsentation und den gemeinsamen Review mit Nicht-Modellierungsspezialisten.

5. Inkrementelle Entwicklung von Testmodellen

Auf Basis der formalen Grundlagen, die wir im vorangegangen Kapitel gelegt haben, führen wir in diesem Kapitel einen Inkrementbegriff auf Verhaltensmodellen und einen zugehörigen Entwicklungsprozess ein, der einen strukturierten, kontrollierten und schrittweisen Aufbau von Modellverhalten vorsieht. Diese Begriffe wurden durch Projekterfahrungen [PPW+05, PSAK04] motiviert, bei denen Verhaltensmodelle von industrieller Komplexität inkrementell entwickelt wurden. Sie charakterisieren formal, wie ein Modellentwickler beim inkrementellen Verhaltensaufbau vorgeht und welche Aktivitäten im Entwicklungsprozess wiederkehrend und zyklisch durchgeführt werden. Die formale Beschreibung hat einerseits den Zweck genau zu verstehen, was inkrementelles Vorgehen bei der Entwicklung von Verhaltensmodellen bedeutet, und bildet andererseits die Basis, in Zukunft Werkzeugunterstützung für den vorgeschlagenen Prozess entwickeln zu können. Denkbar wäre zum Beispiel, Wizards in das Werkzeug AutoFocus zu integrieren, die den Entwickler beim Aufbau von Modellinkrementen und dem Durchführen von komplexeren Operationen auf Verhaltensmodellen unterstützen.

Der vorgeschlagene Entwicklungsprozess weicht bewusst von Prozessen ab, die auf klassischer Verhaltensverfeinerung beruhen. Die Erfahrung bei der Entwicklung von komplexen Systemen zeigt, dass der Entwicklungsprozess mit einem intensiven Lernprozess über das Sollverhalten des zu entwickelnden Systems einhergeht, d.h. dieses erst während der Entwicklung und nicht zu Beginn der Entwicklung verstanden wird. Dies hat zur Folge, dass während der Entwicklung häufig Korrekturen und Änderungen am Verhalten erforderlich sind. Genau dieses wird durch die strengen Regeln bei auf klassischer Verhaltensverfeinerung beruhender Prozesse unzureichend unterstützt, da hier spezifiziertes Verhalten bei jeder Weiterentwicklung erhalten wird und folglich Korrekturen sehr eingeschränkt zugelassen werden. Dieses ist eine Folge der zentralen Eigenschaft dieser Prozesse, dass alle universellen Eigenschaften während der Entwicklung erhalten bleiben. Unser Ansatz stellt jedoch die erforderliche Flexibilität zur Verfügung, umfassende Korrekturen am Modellverhalten während der Entwicklung zuzulassen. Der Preis für diese Flexibilität ist, dass im Entwicklungsprozess lediglich vom Modellentwickler ausgewählte existenzielle Eigenschaften des Modellverhaltens erhalten werden. Diese Eigenschaft ist jedoch vollkommen ausreichend, das Vertrauen in die Korrektheit eines Modells zu gewinnen und zu erhalten, welches die Voraussetzung von Verhaltensmodellen für den Einsatz zum modellbasierten Testen ist. Der vorgestellte Inkrementbegriff und der zugehörige Entwicklungsprozess hat weiterhin den Vorteil, dass er einfach gehalten ist und

auf einfachen Beschreibungsmitteln beruht. Dies erlaubt eine Integration in Werkzeuge wie AutoFocus und erhöht die Wahrscheinlichkeit, dass er für Entwickler anwendbar und zudem richtig angewendet wird.

Die Struktur dieses Kapitels ähnelt der Struktur des vorangegangenen. Wir führen zuerst den Inkrementbegriff anhand des deskriptiven und abstrakteren Modellverhaltens ein und stellen im Anschluss Operationen auf Basis der operationellen Beschreibungstechnik der Regelsysteme vor, die ihre Verwendung zur Entwicklung der Modellinkremente finden: Abschnitt 5.1 wiederholt kurz die Anforderungen an einen inkrementellen Entwicklungsprozess von Testmodellen. Abschnitt 5.2 erarbeitet die formale Definition des Inkrementbegriffs aus Sicht des Modellverhaltens und Abschnitt 5.3 erläutert den zugehörigen Entwicklungsprozess, der sich zur Integration in ein Modellierungswerkzeug eignet. Abschnitt 5.4 gibt einen kritischen Vergleich zu den Prozessen, die auf klassischer Verfeinerung beruhen. Der zentrale Abschnitt 5.5 diskutiert ausführlich die Operationen, die bei der inkrementellen Entwicklung von Modellen auf Basis von Regelsystemen eingesetzt werden, und verdeutlicht diese anhand eines Beispiels. Das Kapitel schließt mit einer Zusammenfassung in Abschnitt 5.6.

5.1. Anforderungen

Die intellektuelle Beherrschbarkeit des Modells während seiner Entwicklung muss gewährleistet werden und es muss zu jedem Zeitpunkt im Entwicklungsprozess das Vertrauen bestehen, dass das Modell einen Teil des Systemverhaltens korrekt gegenüber der Spezifikation bzw. Nutzeranforderungen realisiert. Vor diesem Hintergrund ergeben sich zwei wesentliche Anforderungen an den Entwicklungsprozess von Verhaltensmodellen:

- Der Prozess unterstützt das schrittweise hinzufügen von Funktionalität bzw. von Verhalten zum Modell. Dabei muss eine Möglichkeit bereitgestellt werden, nachweisen zu können, dass bei Weiterentwicklung des Modells das bereits spezifizierte Verhalten erhalten bleibt.
- Der Modellierungsprozess des zu testenden Systems ist eng verbunden mit einem Lernprozess über die Anforderungen an dessen Systemverhalten. Deshalb muss der Prozess Möglichkeiten bereitstellen, Verhalten, das sich während der Entwicklung als unzureichend herausstellt, korrigieren zu können.

Wie wir in Abschnitt 2.2.1 und 5.4 diskutieren, beschränken wir uns auf einen Entwicklungsprozess für deterministische Modelle.

5.2. Formale Definition eines Inkrements

In diesem Abschnitt führen wir die formale Definition eines Inkrementbegriffs ein, der den Anforderungen aus Abschnitt 5.1 genügt.

Im Laufe der inkrementellen Entwicklung eines Testmodells wird die Funktionalität in Form von Modellverhalten von einem Inkrement zum nächsten erweitert. Um die Erweiterung transparent zu gestalten und ihre intellektuelle Beherrschbarkeit zu erleichtern, unterscheiden wir auf Ebene des Modellverhaltens zwei Entwicklungsschritte:

Erweiterung der Schnittstelle Die funktionale Erweiterung des Modells erfordert u.U. die Erweiterung der Schnittstelle des Modells, da die vorhandenen Ein- und Ausgabekanäle bzw. die damit definierten Ein- und Ausgabenachrichten nicht ausreichend sind, das hinzuzufügende Verhalten zu modellieren. Bei diesem Schritt wird das Ein-/Ausgabeverhalten im Wesentlichen nicht verändert. Er dient als Vorbereitung für den folgenden Schritt.

Erweiterung des Verhaltens Im diesem Schritt wird die eigentliche Erweiterung des Modellverhaltens vorgenommen, indem dem Modell neue Verhaltensregeln hinzugefügt werden. Die Verhaltenserweiterung kann in manchen Fällen neben der Erweiterung der Schnittstelle weitere vorbereitende Modifikationen am vorhandenen Verhaltensmodell erfordern. Zum Beispiel müssen Korrekturen an vorhandenen Verhaltensregeln durchgeführt werden, da sie im Konflikt mit den hinzufügenden Verhaltensregeln stehen, oder es werden Transformationen auf dem Verhaltensmodell vorgenommen, die seine innere Struktur verändern, aber das an der Schnittstelle beobachtbare Modellverhalten unverändert lassen. Die letztere Form der Modifikation wird in der Literatur in Anlehnung an Fowler [FBB+99] mit Refactoring bezeichnet und dient zur Vorbereitung, eine Erweiterung zu erleichtern bzw. überhaupt zu ermöglichen.

Welche Operationen für obiges Vorgehen auf Basis von Regelsystemen verwendet werden und welche Auswirkungen diese auf ihr Modellverhalten haben, werden wir ausführlich in Abschnitt 5.5 untersuchen und diskutieren. Ein Verfahren zum automatisierbaren Refactoring von Regelsystemen erarbeiten wir in Kapitel 6.

5.2.1. Inkrementbildung unter Schnittstellenerweiterung

Bei der inkrementellen Modellentwicklung reichen für eine anstehende funktionale Erweiterung des Modellverhaltens u.U. die vorhandenen Schnittstellendefinitionen nicht aus, um das hinzufügende Verhalten adäquat modellieren zu können. Deshalb ist die Schnittstellenerweiterung des Modells im Entwicklungsprozess ein wichtiger Schritt zur Vorbereitung einer Verhaltenserweiterung. Dieser Schritt darf jedoch keine gravierenden Auswirkungen auf das bereits modellierte Modellverhalten haben, damit die Transparenz und die Kontrollierbarkeit der Modellveränderungen sichergestellt werden. Wir unterscheiden vier Möglichkeiten die Schnittstelle eines Modells zu erweitern:

- Die Schnittstelle wird um einen *Eingabekanal* mit beliebigen Typ erweitert.
- Dem *Typ* eines bereits vorhandenen Eingabekanals werden neue Nachrichten hinzugefügt.

- Die Schnittstelle wird um einen *Ausgabekanal* mit beliebigen Typ erweitert.
- Dem *Typ* eines bereits vorhandenen Ausgabekanals werden neue Nachrichten hinzugefügt.

Entsprechend definieren wir die Erweiterung einer Schnittstelle formal, wie folgt:

Definition 5.1 (Erweiterung der Schnittstelle). *Ist (I, O) eine syntaktische Schnittstelle mit Typisierung type, dann ist $(\hat{I}, \hat{O})$ mit Typisierung $\widehat{type}$ eine erweiterte Schnittstelle von (I, O), wenn folgendes gilt:*

1. $I \subset \hat{I}$
2. $O \subset \hat{O}$
3. $\forall i \in I$ *gilt* $type(i) \subset \widehat{type}(i)$
4. $\forall o \in O$ *gilt* $type(o) \subset \widehat{type}(o)$

Darauf aufbauend bringen wir in der folgenden Definition zum Ausdruck, dass bei Inkrementbildung unter Schnittstellenerweiterung das Modellverhalten nicht wesentlich verändert wird. Dies erreichen wir durch die Forderung, dass das Modellverhalten des Ausgangsmodell vollständig in das Modellverhalten des erweiterten Modells eingebettet ist:

Definition 5.2 (Inkrementbildung unter Schnittstellenerweiterung). *Sei S ein Modell mit syntaktischer Schnittstelle (I, O) und $\hat{S}$ ein Modell mit syntaktischer Schnittstelle $(\hat{I}, \hat{O})$. Ist $(\hat{I}, \hat{O})$ eine erweiterte Schnittstelle von (I, O), dann heißt $\hat{S}$ Inkrement von S unter Schnittstellenerweiterung, wenn gilt:*

$$\forall (i, o) \in R_S \, \exists (\hat{i}, \hat{o}) \in R_{\hat{S}} \text{ mit } i.\iota = \hat{i}.\iota \text{ und } o.\theta = \hat{o}.\theta \text{ für } \iota \in I, \theta \in O \tag{5.1}$$

Wie oben bereits erwähnt dient die Inkrementbildung unter Schnittstellenerweiterung zur Vorbereitung der Inkrementbildung unter Verhaltenserweiterung, die wir im folgenden Abschnitt einführen.

5.2.2. Inkrementbildung unter Verhaltenserweiterung

Im Folgenden legen wir fest, welche Eigenschaften des Modells im inkrementellen Modellierungsprozess erhalten bleiben. Wir unterscheiden existenzielle und universelle Eigenschaften eines Modells. Existenzielle Eigenschaften treffen Aussagen darüber, welches Verhalten das Modell zeigen muss. Universelle Eigenschaften fordern dagegen, dass Aussagen für das Gesamtverhalten des Modells gelten müssen. Wir definieren existenzielle und universelle Eigenschaften wie folgt:

Definition 5.3 (Existenzielle und universelle Eigenschaften). *Sei S ein Modell mit Schnittstelle (I, O) und Modellverhalten R_S. Eine Menge von Abläufen $P \subset \vec{I} \times \vec{O}$ heißt existenzielle Eigenschaft von S, wenn*

$$P \subset R_S$$

gilt. P heißt heißt universelle Eigenschaft, wenn

$$R_S \subset P$$

gilt.

Entwicklungsprozesse, die auf klassischer Verfeinerung beruhen, führen zur Erhaltung von universellen Eigenschaften. Diese Prozesse schränken jedoch die durchführbaren Schritte während der Entwicklung stark ein und werden dadurch unflexibel (vgl. Abschnitt 5.4). Deshalb wählen wir einen Entwicklungsprozess, der sich auf den Erhalt von existenziellen Eigenschaften stützt. Die existenziellen Eigenschaften, die während der Entwicklung des Modells erhalten werden, werden vom Entwickler ausgewählt und festgelegt.

Wird ein Inkrement eines Modells zum nächsten Inkrement erweitert, dann stehen die folgenden zwei Punkte im Vordergrund: (1) Das Inkrement erhält im wesentlichen das Verhalten des Vorgängermodells und (2) das Inkrement modelliert das hinzuzufügende Verhalten korrekt. Das Verhalten, das bei der Inkrementbildung erhalten werden soll, wird zuvor in Form einer existenziellen Eigenschaft spezifiziert. Ebenso geben wir in Form einer existenziellen Eigenschaft an, welches Verhalten durch das Folgeinkrement zusätzlich realisiert werden muss. Dies führt zur folgenden intuitiven Definition:

Definition 5.4 (Inkrementbildung unter Verhaltenserweiterung). *Seien S und $\hat{S}$ Modelle mit syntaktischer Schnittstelle (I, O). Modell $\hat{S}$ heißt Inkrement unter Verhaltenserweiterung von Modell S mit der Erhaltung der Eigenschaft P und der Realisierung der Eigenschaft Q, wenn folgendes gilt:*

1. *P ist existenzielle Eigenschaft von S und $\hat{S}$, d.h. $P \subset R_S$ und $P \subset R_{\hat{S}}$.*
2. *$\hat{S}$ erweitert S um die existentielle Eigenschaft Q, d.h. $Q \subset R_{\hat{S}}$ und $Q \not\subset R_S$.*

Da wir bei der Inkrementbildung nicht den Erhalt des vollständigen Verhaltens fordern, ermöglichen wir ein kontrolliertes Vorgehen, das die Modifikation von bereits modellierten Verhalten zulässt. Dadurch können wir im Entwicklungsprozess flexibel auf geänderte Anforderungen reagieren, die z.B. während der Modellentwicklung durch die Analyse und dem Lernprozess über das Systemverhalten entstehen.

Die Definition 5.4 zeigt ihren Nutzen erst in Zusammenspiel mit deterministischen Modellen. Bei deterministischen Modellen führt der Nachweis der Eigenschaften P und Q dazu, dass im Modellverhalten kein Verhalten enthalten sein kann, das zu P bzw. Q im Widerspruch steht. Formal zeigen wir dies mit folgender Proposition:

Proposition 5.1. *Sei Modell $\hat{S}$ mit syntaktischer Schnittstelle (I, O) ein Inkrement unter Verhaltenserweiterung von Modell S mit dem Erhalt der Eigenschaft P und der Realisierung der Eigenschaft Q. Sind $\hat{S}$ und S deterministische Modelle, dann gilt für Modell S*

$$\forall (i, o) \in P \, \forall (\tilde{i}, \tilde{o}) \in R_S \text{ mit } i \sqsubseteq \tilde{i} \text{ gilt } o \sqsubseteq \tilde{o} \tag{5.2}$$

und analog für Modell $\hat{S}$

$$\forall (i, o) \in P \cup Q \, \forall (\hat{i}, \hat{o}) \in R_{\hat{S}} \text{ mit } i \sqsubseteq \hat{i} \text{ gilt } o \sqsubseteq \hat{o}. \tag{5.3}$$

Aus einem anderen Blickwinkel betrachtet, können wir einen inkrementellen Entwicklungsschritt als Modifikation am Modellverhalten im Sinne von Breitling [Bre01] auffassen. Breitling benutzt Modifikationen, um Fehler im Modellverhalten auszudrücken. Wir dagegen verwenden den Begriff *Modifikation des Modells*, um einen Entwicklungsschritt im inkrementellen Prozess darzustellen, der einerseits Fehler korrigiert und andererseits neues Verhalten hinzufügt. In unserer Sicht ist also ein Modell $\hat{S}$ ein Inkrement von Modell S, wenn es Verhalten E, N gibt, wobei das Verhalten N nichttrivial ist, so dass

$$R_{\hat{S}} = (R_S \setminus E) \cup N$$

gilt. Dabei repräsentiert $R_S \setminus E$ das Verhalten, das in Modell $\hat{S}$ von Modell S erhalten wird. Die Menge E enthält also die Abläufe von Modell S, die sich bei der Bildung des Inkrements $\hat{S}$ als fehlerhaft herausstellten bzw. verändert werden mussten. Die Menge N dagegen enthält die Abläufe, die das hinzugefügte Verhalten bzw. das veränderte Verhalten von Modell S modellieren. Abbildung 5.1 verdeutlicht graphisch den Zusammenhang

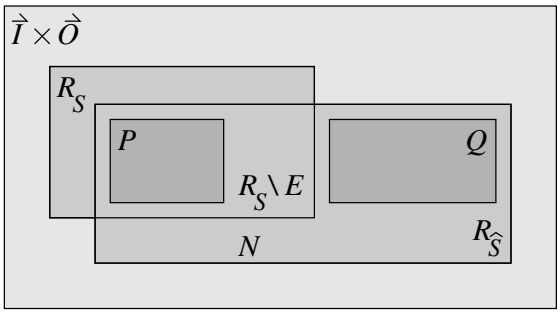


Abbildung 5.1.: Inkrementbildung als Modifikation

zwischen den Modellverhalten R_S und $R_{\hat{S}}$, den Modifikationsmengen E und N und den existenziellen Eigenschaften P und Q. Beim Übergang von Modellinkrement S zu $\hat{S}$ wird also die Weiterentwicklung eingeschränkt, indem nach den Veränderungen E im Modell S die existenzielle Eigenschaft P erhalten werden muss, d.h. es gilt $P \subset R_S \setminus E$, und die geforderte existenzielle Eigenschaft Q realisiert werden muss, d.h. es gilt $Q \subset R_{\hat{S}}$. Aus methodischer Sicht werden also die im Allgemeinen unendlich viele Abläufe umfassenden Modifikationsmengen E und N durch die existenziellen Eigenschaften P und Q approximiert.

5.2.3. Diskussion und verwandte Ansätze

Eigenschaften Nach Dams u.a. [DGG97] werden vier Arten von Eigenschaften unterschieden: universelle/existenzielle Eigenschaften (engl. *universal/existential properties*) und Sicherheits-/Lebendigkeitseigenschaften (engl. *safety/liveness properties*). Universelle und existenzielle Eigenschaften kennzeichnen sich durch ihre Gültigkeit auf allen bzw. einer Auswahl von Abläufen. Die Verletzung von Sicherheits- und Lebendigkeitseigenschaften kann nach endlicher Zeit bzw. nur durch Untersuchung von vollständigen ggf. unendlichen Abläufen beobachtet werden. Alpern und Schneider [AS85] zeigten, dass universelle Eigenschaften sich in eine Sicherheits- und Lebendigkeitseigenschaft zerlegen lassen. Der Erhalt von universellen Eigenschaften und universellen Sicherheitseigenschaften wurde in vielen klassischen Ansätzen ausführlich untersucht. Dagegen spielt der Erhalt von universellen Eigenschaften bei der Entwicklung von Testmodellen eine untergeordnete Rolle: die unscharfen Anforderungen zu Beginn der Entwicklung, der Lernprozess und sich ändernde Anforderungen während der Entwicklung erfordern ständig Korrekturen am Modellverhalten. Folglich können universelle Eigenschaften in der vollen Allgemeinheit nicht erhalten bzw. nur mit sehr hohen Aufwand erhalten werden. Ähnliche Erfahrungen wurden z.B. von Keidar u.a. [KKLS00] berichtet. Sie entwickelten aus theoretischer Sicht ein elegantes Verfahren, um Beweise für die Gültigkeit von universellen Eigenschaften von IO-Automaten [LT89] während der Entwicklung unter speziellen Operationen inkrementell wieder zu verwenden. Der Einsatz dieses Verfahrens zeigte in der Praxis jedoch, dass bei der Entwicklung eines Nachfolgeautomaten, die eine nicht erlaubte Veränderung des Ausgangsautomaten erfordert, immer mit hohen Aufwand verbunden ist. Um diesen und ähnlichen Schwierigkeiten aus dem Weg zu gehen und eine zügige aber dennoch qualitativ hochwertige Entwicklung von Testmodellen zu unterstützen, stellen wir im inkrementellen Entwicklungsprozess von Testmodellen den Erhalt von ausgewählten existenziellen Eigenschaften in den Vordergrund und verzichten im Allgemeinen auf den Erhalt von universellen Eigenschaften. Während der Modellentwicklung erhält also der Entwickler die verantwortungsvolle Aufgabe, zu erhaltende existenzielle Eigenschaften festzulegen und deren Erhalt sicherzustellen. Nichtsdestotrotz betrachten wir z.B. den Einsatz von Modelcheckern zum Nachweis von ausgewählten universellen Eigenschaften als sinnvoll, um die Korrektheit des Testmodells am Ende der Entwicklung gegenüber diesen Eigenschaften zu verifizieren. Dieser Nachweis ist u.U. ein wertvoller Beitrag, um das Vertrauen in die Korrektheit des Testmodells zu erhöhen. Ihr direkter Nutzen für den Test des SUT selbst ist jedoch eingeschränkt: da beim Black-Box-Testen nur endlich viele Abläufe endlicher Länge verwendet werden können, ist eine Verifikation von universellen Eigenschaften auf der Implementierung mittels Testen nicht bzw. nur näherungsweise möglich (vgl. Pretschner [Pre03b], Kap. 3).

Zustandsmaschinen in FOCUS Breitling und Philipps [BP99] entwickelten eine Zustandsmaschinenvariante mit asynchronen Ausführungsmodell und stellen diesen Black-Box-Spezifikationen gegenüber. Sie geben eine Reihe komplexer Beweisverpflichtungen an, um spezielle Sicherheits- bzw. Lebendigkeitseigenschaften auf der Black-Box-Sicht

einer Zustandsmaschine nachweisen zu können. Es stellte sich heraus, dass bereits für kleine Modelle die Komplexität der Beweisverpflichtungen sehr groß ist. Die hohe Komplexität des Ansatzes verhindern die ansprechenden Ergebnisse der Arbeit auf die Entwicklung von Testmodellen zu übertragen.

Rumpe [Rum96] und Scholz [Sch98] entwickelten die statecharts-Varianten buchstabierende Automaten bzw. μ-Charts, um die Spezifikation von verteilten objektorientierten bzw. reaktiven Systemen zu unterstützen. Sie gaben zu den statechart-Varianten eine formale Semantik und eine Entwicklungsmethodik an, die Operationen auf den jeweiligen Automatensprache bereitstellt, so dass das Black-Box-Verhalten der Automaten in einer Verfeinerungsbeziehung (= Ablaufinklusion) zueinander stehen. Beide Ansätze schränken die Entwicklung von Automaten stark ein (vgl. auch Abschnitt 5.4) und scheinen für den industriellen Einsatz nicht zu skalieren, da dies bisher weder durch entsprechende Fallstudien noch durch die Realisierung der Ansätze in Werkzeuge nachgewiesen werden konnte.

5.3. Inkrementeller Entwicklungsprozess

In Abschnitt 2.3.2 haben wir den Prozess der inkrementellen Entwicklung von Testmodellen informell skizziert. In diesem Abschnitt verfeinern wir diesen auf Basis des formalen Inkrementbegriffs aus Abschnitt 5.2. Der Zweck der Formalisierung ist einerseits, den inkrementellen Entwicklungsprozess formal zu fassen und zu verstehen, und andererseits die Grundlage für die Realisierung einer Werkzeugunterstützung zu schaffen. Zum Beispiel wäre die Integration eines Wizards in das Werkzeug AutoFocus denkbar, der den Entwickler bei der Ausführung von wiederkehrenden Modellierungstätigkeiten im inkrementellen Prozess und bei der Prüfung ihrer Konsistenz unterstützt.

Schritte bei der Inkrementbildung Nach der Planung liegt –ggf. nur informell dokumentiert– ein Folge von Anforderungen an Funktionalitäten $F_1, \ldots, F_n$ vor, die der Reihe nach im Modell realisiert bzw. in Form von Verhaltenserweiterungen hinzugefügt werden. Bei der Spezifikation der syntaktischen Schnittstelle (I, O) werden die Eingabekanäle I, die Ausgabekanäle O und deren Typ festgelegt. Im Anschluss erfolgt der eigentliche inkrementelle Aufbau des Modellverhaltens. Gemäß der fünf Schritte aus Abschnitt 2.3.2 wird beim Aufbau des k-ten Inkrements Folgendes abgearbeitet:

- Aus den Anforderungen der Funktionalität F_k wird eine endliche Ablaufmenge über der Schnittstelle (I, O) in Form einer existenziellen Eigenschaft Q_k spezifiziert, welche das im aktuellen Inkrement zu realisierende Verhalten festlegt. In der Praxis leitet der Entwickler aus den informellen Anforderungen für das Entwicklungswerkzeug lesbare Abläufe ab, die später direkt zum Nachweis ihrer Realisierung im Modell dienen.
- Das Verhaltensmodell S_{k-1} wird vom Entwickler zu Modell S_k derart erweitert, so

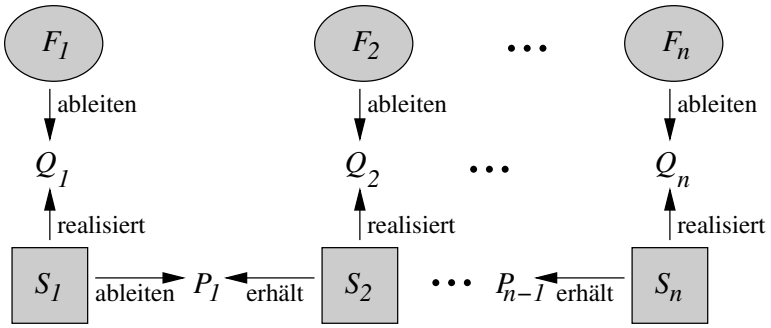


Abbildung 5.2.: Inkrementeller Entwicklungsprozess

dass das Verhalten Q_k zusätzlich im Modell realisiert wird. Dazu muss er ggf. die Schnittstelle des Modells erweitern und weitere Operationen auf dem Verhaltensmodell durchführen. Diese Operationen und etwaig mögliche Werkzeugunterstützung diskutieren wir in Abschnitt 5.5.

- Mit Hilfe des Entwicklungswerkzeugs wird nachgewiesen, dass das Modell die Eigenschaft Q_k erfüllt, d.h. das Werkzeug testet die Abläufe aus Q_k gegen das Modell S_k.
- Weiterhin wird mit Hilfe des Werkzeugs nachgewiesen, dass das Verhalten des Vorgängerinkrements, spezifiziert durch die existenzielle Eigenschaft P_{k-1}, erhalten wurde[1], d.h. das Werkzeug führt mit den Abläufen aus P_{k-1} einen Regressionstest auf dem Modell S_k durch. Wurde bei der Inkrementbildung von S_k zu S_{k-1} die Schnittstelle erweitert, dann muss zuvor ggf. werkzeuggestützt die Regressionstestsuite P_{k-1} auf die Schnittstelle von S_k geeignet geliftet werden (vgl. Abschnitt 5.5.1).
- Der Entwickler erweitert die existenzielle Eigenschaft P_{k-1} zu der Eigenschaft P_k, welche das Verhalten spezifiziert, das bei der Bildung des nächsten Inkrements erhalten werden muss. Im einfachsten Fall ergänzt er von Hand die Abläufe aus P_{k-1} um die Abläufe aus Q_k und um weitere, die sich bei der Inkrementbildung oder der Simulation des Modells als erforderlich herausgestellt haben. Ein Mittel, um eine umfassende Regressionstestsuite P_k zu erhalten, ist, aus den existenziellen Eigenschaften Q_k funktionale Testfallspezifikationen abzuleiten und diese mit dem Modell S_k zur Testfallgenerierung zu nutzen. Die erzeugten Testfälle bilden zusammen mit denen aus P_{k-1} die Regressionstestsuite P_k für das folgende Modellinkrement S_{k+1}.

In Abbildung 5.2 fassen wir den Sachverhalt aus methodischer Sicht schematisch zusammen. Die Ellipsen stellen die funktionalen Anforderungen F_k an das Modellverhalten dar, die gemäß der Planung der Reihe nach im Modell modelliert werden. Aus diesen werden

[1] Wird das erste Inkrement erstellt, dann gibt es kein zu erhaltendes Verhalten, d.h. P ist in diesem Fall leer.

zu realisierende Eigenschaften abgeleitet, die vom jeweiligen Inkrement des Modells S_k realisiert werden. Schließlich werden aus den Modellen die Eigenschaften P_k ableitet, die vom Folgeinkrement S_{k+1} erhalten werden. Am Ende der Entwicklung liegt ein Verhaltensmodell S_n vor, das alle Anforderungen an das Verhalten bzw. an die Funktionalitäten des zu testenden Systems realisiert und miteinander integriert hat.

Eigenschaften des Entwicklungsprozesses Insgesamt identifizieren wir im inkrementellen Entwicklungsprozess für alle k folgende Eigenschaften (wobei wir kurzzeitig annehmen, dass sich die Modellschnittstelle im Laufe des Prozesses nicht verändert):

1. Die existenzielle Eigenschaft Q_k wird in Modellinkrement S_k integriert und die existenzielle Eigenschaft P_{k-1} vom Vorgängerinkrement erhalten, d.h. es gilt

$$Q_k \subset R_{S_k} \supset P_{k-1}. \tag{5.4}$$

2. Die zu erhaltenden Eigenschaften bzw. Regressionstestsuiten P_k werden sukzessive mindestens um die zu realisierenden Eigenschaften erweitert, d.h. es gilt

$$Q_k \subset P_k. \tag{5.5}$$

3. Die zu erhaltenden existenziellen Eigenschaften bzw. Regressionstestsuiten wachsen im Laufe des Entwicklungsprozesses an, d.h. es gilt

$$P_{k-1} \subset P_k. \tag{5.6}$$

Aus (5.4) und (5.6) folgt, dass die existenziellen Eigenschaften über den ganzen Entwicklungsprozess erhalten werden, d.h. es gilt für alle k

$$P_1 \subset P_2 \subset \ldots \subset P_{k-1} \subset R_{S_k}.$$

Dieser Vorgang ist ein wesentlicher Beitrag, um das Vertrauen in die Korrektheit des Testmodells gegenüber seiner Anforderungen zu erhalten. Ferner spiegelt er den Lernprozess während der Modellentwicklung wieder, d.h. zu dem Zeitpunkt, zu dem die Anforderungen F_k in das Modellinkrement S_k integriert werden und durch die intensive Auseinandersetzung mit ihnen entsprechend umfassend verstanden werden, werden adäquate Abläufe in die Regressionstestsuite aufgenommen. Ferner erlaubt dieses Vorgehen jederzeit, jenes Verhalten zu korrigieren, welches über die Regressionstestsuiten hinaus in den Modellinkrementen spezifiziert ist, jedoch noch nicht intensiv untersucht bzw. ausstreichend verstanden wurde.

Zur Verdeutlichung stellen wir in der Abbildung 5.3 exemplarisch anhand von drei Inkrementen die Inklusionsbeziehungen zwischen Modellverhalten R_{S_k}, zu modellierenden Verhalten Q_k und zu erhaltenden Verhalten P_k beim inkrementellen Verhaltensaufbau dar. In dieser Abbildung wurde der Einfachheit halber angenommen, dass sich die Schnittstelle des Modells im Laufe der Entwicklung nicht verändert.

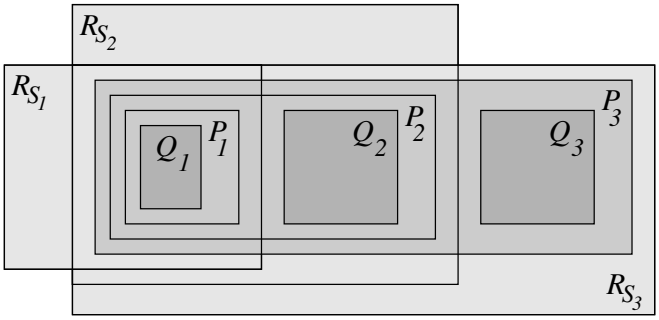


Abbildung 5.3.: Inklusionsbeziehungen beim inkrementellen Verhaltensaufbau

Schnittstellenerweiterung bei der Inkrementbildung Falls beim Übergang von Modellinkrement S_{k-1} zu S_k eine Erweiterung der Schnittstellen stattgefunden hat (vgl. Definition 5.2), dann muss die zugehörige Regressionstestsuite P_{k-1} auf die erweiterte Schnittstelle geliftet werden (vgl. Abschnitt 5.5.1). Folglich werden obige Eigenschaften (5.4) und (5.6) zu $Q_k \subset R_{S_k} \supset \mathit{lift}(P_{k-1})$ bzw. $\mathit{lift}(P_{k-1}) \subset P_k$ verallgemeinert, wobei *lift* die Funktion zum Liften von Abläufen auf die Schnittstelle von Modellinkrement S_k bezeichnet. Für Details bezüglich der Funktion *lift* verweisen wir auf Abschnitt 5.5.1.

Abschließende Bemerkung Der entscheidende und kreative Prozess während der Inkrementbildung findet im zweiten Schritt bei der Integration von neuem zu dem bereits modellierten Verhalten statt. Hier werden Auslassungen, Unvollständigkeiten bzw. Mehrdeutigkeiten in den Anforderungen oder Konflikte zu dem bereits spezifizierten Verhalten aufgedeckt. Nach deren genauen Analyse und Untersuchung von unterschiedlichen Auflösungsvarianten, die ggf. die Konsultation von Domänenexperten oder Entscheidungsträgern erfordert, werden diese Lücken geschlossen bzw. die Konflikte aufgelöst.

Wir modellieren das Verhalten des zu testenden Systems operationell mit einem Regelsystem (Kapitel 4.4). Welche Operationen auf dem Modell welche Auswirkungen auf das Modellverhalten haben, untersuchen und erläutern wir in Abschnitt 5.5. Ferner demonstrieren wir diesen Prozess in Kapitel 7 anhand einer realen Fallstudie aus der Industrie.

5.4. Vergleich zum klassischen Verfeinerungsprozess

In diesem Abschnitt vergleichen wir den vorgestellten inkrementellen Entwicklungsprozess mit Prozessen, die auf klassischer Verhaltensverfeinerung beruhen. Wir wählen FOCUS mit Verhaltensverfeinerung als Repräsentant, um klassische Entwicklungsprozesse mit unserem inkrementellen Ansatz zu vergleichen. Da unser Ansatz eine Variante von FOCUS darstellt, bezeichnen wir zur Unterscheidung zu unserem Ansatz FOCUS mit Verhaltensverfeinerung als klassisches FOCUS. Zunächst fassen wir für den Vergleich alle

relevanten Eigenschaften von klassischem FOCUS kurz zusammen. Für eine ausführliche Beschreibung der Grundlagen und Varianten von FOCUS verweisen wir auf [BS01].

Eingabevollständigkeit Im klassischen FOCUS wird ein Modell S mit Schnittstelle (I, O) durch eine Relation $R_S \subset \vec{I} \times \vec{O}$ mit vollständigem Eingabebereich $\mathsf{in}.S = \vec{I}$ beschrieben, d.h. Modelle sind hier per Definition eingabevollständig. Deshalb wird meist die kompaktere funktionale Schreibweise in Form einer totalen Funktion $f_S : \vec{I} \to \wp(\vec{O})$ verwendet.

Nichtdeterminismus Nichtdeterminismus in einem Modell S wird als Unterspezifikation aufgefasst, d.h. einer Eingabesequenz $i \in \vec{I}$ wird u.U. eine Menge von Ausgabesequenzen $f_S.i \subset \vec{O}$ zugeordnet.

Verfeinerung Der Entwicklungsprozess beruht auf klassischer Verfeinerung, d.h. Ablaufinklusion bzw. Reduktion von Nichtdeterminismus: ein Modell $\hat{S}$ ist eine Verfeinerung bzw. Weiterentwicklung von S, wenn $R_S \supset R_{\hat{S}}$ gilt.

Erhalt von Eigenschaften Die wesentliche Eigenschaft in diesem Prozess ist, dass er die universellen Eigenschaften eines Modells erhält: ist $S_1, \ldots, S_n$ eine Folge von Modellen, die in Verfeinerungsbeziehung stehen, und erfüllt das Ausgangssystem S_1 die universelle Eigenschaft P, dann gilt

$$P \supset R_{S_1} \supset R_{S_2} \supset \ldots \supset R_{S_n},$$

d.h. nach jedem Entwicklungsschritt unter Verhaltensverfeinerung erfüllt das Modell alle universellen Eigenschaften der Vorgängersysteme.

Auf Grundlage dieser kurzen Einführung erläutern und begründen wir die Wahl und Festlegungen unseres Ansatzes inklusive dem zugehörigen Entwicklungsprozess.

Partialität des Systemmodells Wir betrachten im Gegensatz zu klassischen FOCUS partielles Modellverhalten $R_S \subset \vec{I} \times \vec{O}$ mit $\mathsf{in}.S \subset \vec{I}$. Dies hat den Vorteil, dass streng zwischen spezifiziertem und nicht spezifiziertem Verhalten unterschieden werden kann, denn nicht spezifiziertes Verhalten ist im Modellverhalten R_S nicht enthalten. Im klassischen FOCUS dagegen kann spezifiziertes und unterspezifiziertes Verhalten nicht unterscheiden werden. Diese Unterscheidungsmöglichkeit in unserem Ansatz erleichtert und trägt wesentlich zur Verständlichkeit und Beherrschbarkeit des Modells während der Entwicklung bei. Ferner ist die Partialität von Testmodellen ausreichend, da sie im Gegensatz zu Entwicklungs- bzw. Spezifikationsmodellen den Zweck haben, Testfälle für den von Natur aus unvollständigen Prozess des Testens zu erzeugen. Ein weiterer Grund ist, dass erzwungene Eingabevollständigkeit auf Modellebene, die zum Beispiel durch Chaos- oder Idlevervollständigung erreicht wird, für die Testfallgenerierung hinderlich sein kann. In diesem Fall würden auf Modellebene zu Eingabesequenzen Abläufe erzeugt, für die es auf konkreter Ebene keine sinnvolle Entsprechung gibt. Die Erzeugung solcher sinnlosen Testsequenzen müsste dann mühsam durch geeignete Testfallspezifikationen verhindert werden.

Deterministische Modelle Wir betrachten in unserem Entwicklungsprozess ausschließlich die Entwicklung von deterministischen Modellen, d.h. allen $i \in \mathsf{in}.S$ wird genau ein $o \in \vec{O}$ zugeordnet. Dies führt zu folgenden Vorteilen:

- Die Simulationsfähigkeit der Verhaltensmodelle kann durch Werkzeuge einfach unterstützt werden. Simulationsfähigkeit ist unerlässlich, um den Lernprozess während dem Entwicklungsprozess und die intellektuelle Beherrschbarkeit von Verhaltensmodellen zu unterstützen. Dagegen können Entwicklungsprozesse, die auf klassischer Verfeinerung beruhen, nicht oder sehr schwer mit Werkzeugen simuliert werden, da das Modellverhalten—zumindest in der Anfangsphase der Entwicklung—hochgradig nichtdeterministisch ist.
- Es ist wesentlich einfacher zu validieren, ob ein spezifisches deterministisches Verhalten korrekt ist, als zu überprüfen, ob alle möglichen nichtdeterministischen Verhalten akzeptabel sind. Dieses trägt wiederum zur Verständlichkeit und einfacheren Beherrschbarkeit der Verhaltensmodelle bei.
- Nach Heimdahl [HL96] ist es nicht wünschenswert in Spezifikationsmodellen Nichtdeterminismus in Form von Unterspezifikation zuzulassen, da er erfahrungsgemäß häufig eine Quelle für Spezifikationsfehler darstellt.

Die Beschränkung auf deterministische Testmodelle führt jedoch zu folgenden Nachteil: Im Testmodell kann eine Überspezifikation gegenüber dem zu testenden System auftreten, d.h. das Modell schreibt bezüglich Determinismus ein präziseres Verhalten vor, als es für die Realität nötig wäre. Das Auflösen der Überspezifikation muss in einem solchen Fall bei der Testausführung durch den Testtreiber bewerkstelligt werden. Zum Beispiel wurde im Testmodell eines Netzwerkcontrollers [PPW+05] die Reihenfolge von Parametern bestimmter Nachrichten deterministisch festgelegt. In der Realität ist jedoch jede Permutation dieser Parameter zulässig. Diese Überspezifikation wird durch den Testtreiber aufgelöst, indem er zur Laufzeit der Tests mit Hilfe einer Prozedur prüft, ob die Parameter der SUT-Nachrichten eine Permutation der erwarteten Nachrichten aus dem Testmodell sind.

Erhalt von Eigenschaften und Flexibilität des Entwicklungsprozesses Da der Entwicklungsprozess im klassischen Focus auf Verfeinerung basiert, herrscht eine strenge Monotonie im Prozess und man erhält dadurch die starke Eigenschaft, dass universelle Eigenschaften erhalten werden. Diese strenge Monotonie erschwert jedoch erheblich, dass Erkenntnisse gewonnen aus dem Lernprozess während der Modellentwicklung flexibel in den Modellierungsprozess und damit Korrekturen von bereits modellierten Verhalten eingebracht werden können. Um den Entwicklungsprozess flexibler und leichter beherrschbar zu gestalten, verzichten wir auf die strenge Monotonie und damit auf den Erhalt von universellen Eigenschaften und wählen einen Entwicklungsprozess, der auf natürliche Weise schrittweise das Verhalten mittels deterministischer Modelle aufbaut, anstatt Unterspezifikation von nichtdeterministischen Modellen zu reduzieren. Dies hat zur Folge,

- dass während der Modellentwicklung flexibel auf Änderungen reagiert werden kann,
- dass durch dieses Vorgehen zuvor spezifizierte existenzielle Eigenschaften erhalten werden (universelle dagegen im allgemeinen nicht),
- dass Werkzeugunterstützung für diesen Entwicklungsprozess leichter realisierbar ist und
- dass der Prozess leichter beherrschbar und begreifbar wird und damit es wahrscheinlicher ist, von Nutzern richtig angewendet zu werden.

5.5. Operationale Entwicklung der Inkremente

Der inkrementelle Prozess sieht den schrittweisen Aufbau von Modellverhalten vor, welches die Anforderungen des Systems während der Entwicklung in ein eindeutiges und ausführbares Verhaltensmodell integriert und diese damit selbst ergänzt und validiert. Wir modellieren die Inkremente im unseren Entwicklungsprozess operationell mit Regelsystemen, die wir in Abschnitt 4.4 eingeführt haben. Ausgehend von einem Minimalmodell, welches nur eine oder wenige Anforderungen des Systems realisiert, werden schrittweise weitere Anforderungen in Form von Verhaltensregeln dem Modell hinzugefügt. Dabei werden die Anforderungen im Modell konkretisiert und miteinander integriert. Das strukturierte Vorgehen ermöglicht, dass in den Anforderungen des Systems systematisch Konflikte, Lücken und Mehrdeutigkeiten aufgedeckt und behoben werden.

Für die Integration von Anforderungen des Systems in das Testmodell unterscheiden wir häufig wiederkehrende Modellierungstätigkeiten in Form von Operationen auf Regelsystemen und teilen diese in drei Klassen ein:

- Vorbereiten einer Verhaltenserweiterung,
- Korrektur des Verhaltens und
- Verhaltenserweiterung.

Diese Klassifikation bildet den Grundstock für eine mögliche Unterstützung des inkrementellen Entwicklungsprozesses innerhalb eines Modellierungswerkzeugs. Dieses Werkzeug verfügt zum Beispiel über einen Modellierungswizard, der anhand einer gewählten Klasse dem Entwickler verschiedene Operationen vorschlägt, die zum Erreichen eines Modellierungsziels führen. Ferner gibt die Klassifikation die Gliederung dieses Abschnitts vor, die jeweils durch die folgenden Fragen strukturiert sind:

- Was ist die Motivation der Modelländerung?
- Welche Operationen sind geeignet, die angestrebte Modelländerung umzusetzen?
- Welche formale Eigenschaften bzw. Auswirkungen haben diese Operationen auf das Modellverhalten?

Am Ende des Abschnitts werden wir eine repräsentative Auswahl der vorgestellten Operationen an einem Beispiel verdeutlichen. Ausführlicher werden wie sie anhand einer industriellen Fallstudie in Kapitel 7 demonstrieren.

5.5.1. Vorbereiten einer Verhaltenserweiterung

Motivation In vielen Fällen reicht die Schnittstelle oder der lokale Datenraum nicht aus, um eine geforderte Verhaltenserweiterung durch alleiniges Hinzufügen oder Verallgemeinern einer Regel bewerkstelligen zu können (siehe Abschnitt 5.5.3). Die Schnittstelle muss beispielsweise erweitert werden, falls Nachrichten zum Zurücksetzen des Modells eingeführt werden oder das Modell auf Fehlermeldungen seiner Umgebung reagieren muss. Der lokale Datenraum wird dazu verwendet, relevante Informationen aus Eingangssignalen dauerhaft abzuspeichern. Dieser Datenraum muss beispielsweise erweitert werden, falls Information von Fehlermeldungen abgelegt werden müssen, um adäquates Verhalten zur Fehlerbehebung modellieren zu können. Ferner werden zur Vorbereitung einer Verhaltenserweiterung so genannte Refactoring-Operationen auf Modellen verwendet, die das beobachtbare Modellverhalten nicht verändern. Dieses umfangreiche Thema werden wir unten kurz diskutieren und auf weiterführende Literatur verweisen.

Operationen Wir unterscheiden zur Vorbereitung einer Verhaltenserweiterung folgende Operationen:

- Erweiterung der Eingabeschnittstelle, d.h. durch Hinzufügen eines Eingabekanals oder durch Erweitern des Typs eines vorhandenen Eingabekanals
- Erweiterung der Ausgabeschnittstelle, d.h. durch Hinzufügen eines Ausgabekanals oder durch Erweitern des Typs eines vorhandenen Eingabekanals
- Erweiterung des lokalen Datenraums durch Hinzufügen einer lokaler Variablen oder durch Erweitern des Typs einer vorhandenen lokalen Variable

Bemerkung: Falls eine Erweiterung stattfindet, indem ein vorhandener Datentyp erweitert wird, dann müssen ggf. Prädikats- und Funktionsdefinitionen, die auf den erweiterten Typ operieren, überarbeitet werden, um die Typkorrektheit des Modells wiederherzustellen. Auf diese notwendigen Tätigkeiten werden wir hier nicht näher eingehen, da sie nicht im Fokus der Modellierungstätigkeit stehen.

In den folgenden Absätzen erläutern und untersuchen wir die obigen Operationen.

Erweiterung der Eingabeschnittstelle Wird die Schnittstelle eines Regelsystems um einen Eingabekanal erweitert und bleibt die Regelmenge unverändert, dann findet implizit eine Erweiterung des Modellverhaltens statt. Diese kommt dadurch zustande, da die Eingabemuster der Regeln keine Bedingungen an den neuen Eingabekanal stellen und folglich das Regelsystem völlig unabhängig von der Belegung des neuen Eingabekanals

operiert. Mit anderen Worten: der Schaltbereich jeder Regel vergrößert sich und damit findet eine Erweiterung des Modellverhaltens statt. Dieses neu entstandene Verhalten ist im Allgemeinen nicht erwünscht und deshalb ist nach Hinzufügen eines Eingabekanals, jede Regel zu prüfen, ob nicht zusätzliche Bedingungen an den neuen Eingabekanal spezifiziert werden müssen (z.B. die Bedingung: der neue Eingabekanal ist unbelegt). Dies ist eine Korrektur des Modellverhaltens im Sinne vom Abschnitt 5.5.2.

Wird der Typ eines vorhandenen Eingabekanals eines Regelsystems erweitert und bleibt die Regelmenge unverändert, dann findet im Allgemeinen ebenfalls implizit eine Verhaltenserweiterung statt. Dies ist insbesondere dann der Fall, wenn es Verhaltensregeln gibt, die keine Bedingungen an den Eingabekanal stellen, dessen Typ erweitert wurde. Bei diesen Regeln vergrößert sich der Schaltbereich, was zu einer Erweiterung des Modellverhaltens führt. Deshalb sind nach Erweiterung des Typs diese Regeln zu überprüfen, ob zusätzliche Bedingungen an den betreffenden Eingabekanal erforderlich sind (siehe auch Abschnitt 5.5.2).

Die Eigenschaften der Erweiterung der Eingabeschnittstelle eines Regelsystems fassen wir in der folgenden Proposition zusammen:

Proposition 5.2 (Erweiterung Eingabeschnittstelle). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (\hat{I}, O, L, Init, R)$ das Regelsystem, das durch eine Erweiterung der Eingabeschnittstelle im Sinne von Definition 5.1 entstanden ist, d.h. es gilt $I \subset \hat{I}$ und $\forall i \in I : type(i) \subset \widehat{type}(i)$. Dann hat das Regelsystem $\hat{G}$ folgende Eigenschaften:*

1. *$\hat{G}$ ist deterministisch.*
2. *$\hat{G}$ ist eine Inkrementbildung unter Schnittstellenerweiterung im Sinne von Definition 5.2, d.h. $\forall (i,o) \in R_{Z_G} \exists (\hat{i},\hat{o}) \in R_{Z_{\hat{G}}}$ mit $o = \hat{o}$ und $i.\iota = \hat{i}.\iota$ für $\iota \in I$.*
3. *Es gibt eine Abbildung $lift : R_{Z_G} \to R_{Z_{\hat{G}}}$ mit $i.\iota = \hat{i}.\iota$ und $o = \hat{o}$, wobei $\iota \in I$, $(i,o) \in R_{Z_G}$ und $(\hat{i},\hat{o}) = lift(i,o)$ gilt.*
4. *Das Modellverhalten von $\hat{G}$ ist gegenüber dem von G durch ein Modellverhalten N über Schnittstelle $(\hat{I}, O)$ im folgenden Sinn erweitert:*

$$R_{Z_{\hat{G}}} = lift(R_{Z_G}) \cup N.$$

Die Abbildung *lift* aus vorangegangener Proposition wird zum Beispiel genutzt, um eine Regressionstestsuite $P \subset R_{Z_G}$ des Regelsystems G auf die Schnittstelle des Regelsystems $\hat{G}$ zu liften. Da die Regeln des Regelsystems $\hat{G}$, die identisch mit denen von G sind, die Belegungen der hinzugefügten Eingabekanäle nicht einschränken, dürfen diese beliebig belegt sein. Das Liften der Regressionstestsuite P kann also werkzeuggestützt erfolgen, indem der Entwickler eine gewünschte Standardbelegung für die hinzugefügten Eingabekanäle auswählt und dann das Werkzeug jeden Ablauf aus P durch Ergänzung entsprechender Kanalbelegungen erweitert.

Erweiterung der Ausgabeschnittstelle Wird die Schnittstelle eines Regelsystems um einen Ausgabekanal oder der Typ eines vorhandenen Ausgabekanals erweitert und bleibt die Regelmenge unverändert, dann ändert sich das Modellverhalten im Gegensatz zu Erweiterung der Eingabeschnittstelle im Wesentlichen nicht. Der Grund hierfür ist, dass die unveränderten Regeln des erweiterten Regelsystems keinen Bezug auf den neuen Ausgabekanal bzw. die neuen Ausgabewerte des erweiterten Typs nehmen und damit die neuen Ausgabewerte niemals gesendet werden. Diesen Sachverhalt fassen wir in der folgenden Proposition zusammen:

Proposition 5.3 (Erweiterung Ausgabeschnittstelle). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (I, \hat{O}, L, Init, R)$ das Regelsystem, das durch eine Erweiterung der Ausgabeschnittstelle im Sinne von Definition 5.1 entstanden ist, d.h. es gilt $O \subset \hat{O}$ und $\forall o \in O : type(o) \subset \widehat{type}(o)$. Dann hat das Regelsystem $\hat{G}$ folgende Eigenschaften:*

1. *$\hat{G}$ ist deterministisch.*
2. *$\hat{G}$ ist eine Inkrementbildung unter Schnittstellenerweiterung im Sinne von Definition 5.2, d.h. $\forall (i,o) \in R_{Z_G} \; \exists (\hat{i}, \hat{o}) \in R_{Z_{\hat{G}}}$ mit $i = \hat{i}$ und $o.\theta = \hat{o}.\theta$ für $\theta \in O$.*
3. *Es gibt eine Abbildung $lift : R_{Z_G} \to R_{Z_{\hat{G}}}$ mit $i = \hat{i}$, $o.\theta = \hat{o}.\theta$ und $\hat{o}.\hat{\theta} = \epsilon^{\#\hat{o}}$, wobei $\theta \in O$, $\hat{\theta} \in \hat{O} \setminus O$, $(i,o) \in R_{Z_G}$ und $(\hat{i}, \hat{o}) = lift(i,o)$ gilt.*
4. *Das Modellverhalten von $\hat{G}$ hat sich gegenüber dem von G im folgenden Sinn nicht verändert:*

$$R_{Z_{\hat{G}}} = lift(R_{Z_G}).$$

Wie auch bei der Erweiterung der Eingabeschnittstelle kann die Abbildung *lift* beim Übergang vom Regelsystem G zu $\hat{G}$ zum werkzeuggestützten Liften einer Regressionstestsuite verwendet werden. Der Unterschied besteht lediglich darin, dass die Belegung der hinzugefügten Ausgabekanäle beim Liften nicht frei gewählt werden kann, sondern gemäß der Standardbelegung von nicht spezifizierten Ausgabebelegungen mit der leeren Nachricht belegt werden.

Erweiterung des lokalen Datenraums Wird der lokale Datenraum eines Regelsystems erweitert, indem eine weitere lokale Variable mit geeigneten Initialwert hinzugefügt wird oder der Datentyp einer vorhandenen Variable erweitert wird, dann hat dies keine Auswirkungen auf das Modellverhalten, da die Verhaltensregeln des Modells die neue Variable bzw. die neuen Variablenwerte nicht verwenden. Dies zeigen wir formal mit folgender Proposition:

Proposition 5.4 (Erweiterung lokaler Datenraum). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (I, O, \hat{L}, \widehat{Init}, R)$ das Regelsystem, das durch eine Erweiterung des lokalen Datenraums entstanden ist, d.h. es gilt $L \subset \hat{L}$, $\forall l \in L$:*

$type(l) \subset \widehat{type}(l)$ und $\widehat{Init} \equiv Init \bigwedge_{l \in \hat{L} \setminus L} l = f_l$ für geeignete $f_l \in \widehat{type}(l)$. Dann hat das Regelsystem $\hat{G}$ folgende Eigenschaften:

1. *$\hat{G}$ ist deterministisch.*
2. *Das Modellverhalten von $\hat{G}$ hat sich gegenüber dem von G nicht verändert, d.h.*

$$R_{Z_{\hat{G}}} = R_{Z_G}.$$

Anmerkung Refactoring Operationen auf Modellen, die wie in der vorangegangen Proposition das beobachtbare Modellverhalten nicht verändern, werden in der Literatur in Anlehnung an Fowler [FBB+99] als Refactoring bezeichnet. Das Ziel beim Durchführen von Refactoring-Operationen ist meist, die Struktur eines Modells dahingehend zu verbessern, um eine Erweiterung des Modells zu erleichtern oder überhaupt zu ermöglichen. Bezüglich dem Thema Refactoring beschränken wir uns in dieser Arbeit auf die obige einfache Erweiterungsoperation und auf das Verfahren aus Kapitel 6 zur Transformation von Regelsystemen. Wir verzichten auf die Einführung und die Diskussion weiterer Refactoring-Operationen, da diese nicht im Zentrum dieser Arbeit liegen und ihren Rahmen weit sprengen würde. Wir verweisen stattdessen auf die Dissertation von Wißpeintner [Wiß05].

5.5.2. Korrektur des Verhaltens

Motivation Die Korrektur, d.h. Reduktion oder Modifikation, von bereits spezifiziertem Modellverhalten kann aus folgenden Gründen erforderlich sein:

- Nach Erweiterung der Eingabeschnittstelle des Modells wird das Modellverhalten implizit erweitert (siehe Abschnitt 5.5.1). Teile dieser Verhaltenserweiterung sind oft unerwünscht, so dass diese durch Einschränkung von Verhaltensregeln korrigiert werden müssen.
- Bei dem Erweitern vom Modellverhalten treten u.U. Konflikte zwischen Verhaltensregeln auf, d.h. Anforderungen an das Modellverhalten widersprechen sich. Nach Klärung, wie der Konflikt aufgelöst wird, muss das Modell entsprechend korrigiert werden.
- Bei Qualitätssicherungsmaßnahmen des Modells, z.B. Reviews des Modell oder von Modellabläufen, Regressionstest des Modells etc., wird festgestellt, dass das Modell Teile der Anforderungen nicht adäquat modelliert bzw. Fehler im Modellverhalten vorhanden sind. Dies erfordert Korrekturen im Modellverhalten.

Operationen Zur Korrektur von Modellverhalten verwenden wir folgende Operationen:

- Einschränken einer Verhaltensregel

- Modifikation der Ausgabe einer Verhaltensregel
- Modifikation der Zuweisung einer Verhaltensregel

Diese Operationen erläutern und untersuchen wir in den folgenden Abschnitten.

Einschränken einer Verhaltensregel Das Modellverhalten wird reduziert, indem die Vorbedingung oder das Eingabemuster einer Verhaltensregel eingeschränkt wird, d.h. der Schaltbereich der Regel wird reduziert. Diese Operation wird verwendet, um unerwünschte Abläufe aus dem Modell zu entfernen, das z.B. durch Erweiterung der Eingabeschnittstelle entstanden ist oder aufgrund eines Konflikts zwischen Verhaltensregeln entfernt werden muss. Folgende Proposition fasst den Sachverhalt bei Einschränkung einer Verhaltensregel zusammen:

Proposition 5.5 (Einschränkung Verhaltensregel). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (I, O, L, Init, \hat{R})$ das Regelsystem, das aus G entstanden ist, indem eine Regel eingeschränkt wurde, d.h. $\hat{R} = R \setminus \{r\} \cup \{\hat{r}\}$, wobei $\mathsf{en}.r \supset \mathsf{en}.\hat{r}$, $output_r \equiv output_{\hat{r}}$ und $assign_r \equiv assign_{\hat{r}}$ gilt. Dann hat das Regelsystem $\hat{G}$ folgende Eigenschaften:*

1. *$\hat{G}$ ist deterministisch.*
2. *Das Verhalten von $\hat{G}$ wird reduziert, d.h. es gibt ein Verhalten E über der Schnittstelle (I, O) mit*
$$R_{Z_{\hat{G}}} = R_{Z_G} \setminus E.$$

Modifikation der Ausgabe einer Regel Werden z.B. durch Qualitätssicherungsmaßnahmen des Modellverhaltens Abläufe mit nicht korrekter Ausgabe aufgedeckt, dann ist oft die Modifikation der Ausgabe einer Verhaltensregel erforderlich. Im Modellverhalten bewirkt die Modifikation der Ausgabe einer Verhaltensregel, dass alle Abläufe, in denen diese Regel schaltet, ersetzt werden durch Abläufe, die die veränderte Ausgabe der Regel zeigen. Wir fassen dies formal mit folgender Proposition zusammen:

Proposition 5.6 (Modifikation der Ausgabe). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (I, O, L, Init, \hat{R})$ das Regelsystem, das aus G entstanden ist, indem die Ausgabe einer Regel modifiziert wurde, d.h. $\hat{R} = R \setminus \{r\} \cup \{\hat{r}\}$, wobei $pre_r \equiv pre_{\hat{r}}$, $input_r \equiv input_{\hat{r}}$ und $assign_r \equiv assign_{\hat{r}}$ gilt. Dann hat das Regelsystem $\hat{G}$ folgende Eigenschaften:*

1. *$\hat{G}$ ist deterministisch.*
2. *Es gibt eine Abbildung $lift : R_{Z_G} \to R_{Z_{\hat{G}}}$ mit $(i, \hat{o}) = lift(i, o)$ für $(i, o) \in R_{Z_G}$.*
3. *Das Verhalten von $\hat{G}$ wird modifiziert, d.h. es gibt Verhalten E und N über der Schnittstelle (I, O) mit*
$$R_{Z_{\hat{G}}} = (R_{Z_G} \setminus E) \cup N.$$

Die Abbildung *lift* wird wie bei der Schnittstellenerweiterung genutzt, um werkzeuggestützt durch die Abbildung einer Regressionstestsuite des Regelsystems G diese für das Regelsystem $\hat{G}$ wieder verwenden zu können. Dazu werden die Abläufe, bei denen die veränderte Regel schaltet, an den betreffenden Stellen die alte Ausgabe durch veränderte Ausgabe ersetzt.

Das Verhalten $R_Z \setminus E$ enthält intuitiv die Abläufe, bei denen die veränderte Regel niemals schaltete, und E enthält die Abläufe, bei denen die veränderte Regel mindestens einmal schaltete, d.h. für jeden Ablauf aus E gibt es einen Ablauf aus N mit gleicher Eingabesequenz, aber mit veränderter Ausgabesequenz.

Modifikation der Zuweisung einer Regel Wird z.B. durch Qualitätssicherungsmaßnahmen des Modellverhaltens fehlerhaftes Modellverhalten aufgedeckt oder wurden aufgrund einer Erweiterung des lokalen Datenraums weitere lokale Variablen im Modell eingeführt, dann kann dies u.U. die Modifikation der Zuweisung einer Verhaltensregel erfordern. Die Modifikation der Zuweisung einer Regel kann starke Auswirkungen auf das Modellverhalten haben und diese sind meist schwer abschätzen, da die Berechnung des Zielzustands im lokalen Datenraum verändert wird. Deshalb ist es erforderlich nach einer solchen Modifikation, das Modellverhalten z.B. mittels Regressionstest zu überprüfen, ob gefordertes Modellverhalten durch die Modifikation nicht zerstört wurde. Mit anderen Worten die Auswirkungen einer Modifikation der Zuweisung einer Verhaltensregel sind schwer kontrollierbar, da oft nicht direkt ersichtlich ist, welche Abläufe im Modellverhalten wegfallen bzw. hinzukommen. Dies ist auch der Grund, weshalb die Aussagen der folgenden Proposition relativ schwach ausfallen.

Proposition 5.7 (Modifikation der Zuweisung). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (I, O, L, Init, \hat{R})$ das Regelsystem, das aus G entstanden ist, indem die Zuweisung einer Regel modifiziert wurde, d.h. $\hat{R} = R \setminus \{r\} \cup \{\hat{r}\}$, wobei $pre_r \equiv pre_{\hat{r}}$, $input_r \equiv input_{\hat{r}}$ und $output_r \equiv output_{\hat{r}}$ gilt. Dann hat das Regelsystem $\hat{G}$ folgende Eigenschaften:*

1. *$\hat{G}$ ist deterministisch.*
2. *Das Verhalten von $\hat{G}$ wird modifiziert, d.h. es gibt Verhalten E und N über der Schnittstelle (I, O) mit*

$$R_{Z_{\hat{G}}} = (R_{Z_G} \setminus E) \cup N.$$

5.5.3. Verhaltenserweiterung des Modells

Motivation Bei der Weiterentwicklung von einem Modellinkrement zum Folgeinkrement werden weitere Anforderungen an das Systemverhalten in das Modell integriert. Um dies zu bewerkstelligen, muss das Modell um adäquates Verhalten erweitert werden, das die hinzufügenden Anforderungen bzw. deren entsprechenden Funktionalitäten modelliert.

Operationen Wir gehen davon aus, dass die Schnittstelle und der lokale Datenraum des Modells ausreichend sind, um das hinzuzufügende Verhalten modellieren zu können. Falls das nicht der Fall ist, muss das Modell zuvor entsprechend vorbereitet werden (siehe Abschnitt 5.5.1). Es stehen folgende Operationen zur Verfügung, um das Modellverhalten zu erweitern:

- Hinzufügen einer Verhaltensregel
- Verallgemeinern einer vorhandenen Verhaltensregel

Die beiden Operationen werden in den folgenden Abschnitten näher erläutert.

Hinzufügen einer Verhaltensregel Das Verhalten des Modells wird erweitert, indem eine oder mehrere Verhaltensregeln dem Regelsystem hinzugefügt werden. Es genügt den Spezialfall, genau eine Verhaltensregel hinzuzufügen, zu betrachten, da der allgemeine Fall, mehrere Regeln hinzuzufügen, lediglich ein wiederholtes Durchführen des Spezialfalls ist. Die zu realisierende Anforderung wird im Modell abgebildet, indem Vorbedingung, Eingabemuster, Ausgabe und Zuweisung einer Verhaltensregel formuliert und ggf. dazu benötigte Prädikat- und Funktionsdefinitionen dem Modell hinzugefügt werden. Da Testmodelle die Anforderung besitzen, deterministisch zu sein, gehen wir davon aus, dass das Modell vor dem Hinzufügen der Regel deterministisch ist. Nach dem Hinzufügen der Regel unterscheiden wir folgende Fälle:

- Das um die Verhaltensregel erweiterte Modell ist deterministisch. In diesem Fall ließ sich die Anforderung ohne Konflikt in das Modell integrieren.
- Das um die Verhaltensregel erweiterte Modell ist nichtdeterministisch. In diesem Fall liegt ein Konflikt von Anforderungen vor, d.h. es gibt eine Verhaltensregel im Modell, dessen Schaltbereich mit dem Schaltbereich der hinzugefügten Regel nicht disjunkt ist. Es wurde also ein Konflikt in den Anforderungen aufgedeckt. Nun muss entschieden werden, wie dieser Anforderungskonflikt aufgelöst wird. Im Modell spiegelt sich dies wieder, indem entweder Modellverhalten bezüglich der Konflikt auslösenden Regel korrigiert wird (siehe Abschnitt 5.5.2) oder die Vorbedingung und das Eingabemuster der hinzugefügten Regel überarbeitet wird, um den Konflikt zu vermeiden.

Verallgemeinern einer Verhaltensregel Das Modellverhalten wird erweitert, indem die Vorbedingung oder das Eingabemuster einer Verhaltensregel verallgemeinert werden, d.h. der Schaltbereich einer Regel wird vergrößert. Dieses Vorgehen findet häufig Anwendung, wenn im Modell zunächst ein einfach zu modellierender Spezialfall realisiert wird und nach Prüfung seiner Korrektheit dieser verallgemeinert wird. Bei der Verallgemeinerung ist darauf zu achten, ob das Modell dadurch nicht nichtdeterministisch wird, in diesem Fall ist die Verallgemeinerung nicht zulässig.

Die in diesem Abschnitt diskutierten Operationen und ihre Eigenschaften fassen wir in

der folgenden Proposition formal zusammen:

Proposition 5.8 (Verhaltenserweiterung). *Sei $G = (I, O, L, Init, R)$ ein deterministisches Regelsystem und $\hat{G} = (I, O, L, Init, \hat{R})$ das Regelsystem, das aus G entstanden ist, indem entweder*

- *die Regelmenge um eine Regel erweitert wurde, d.h. $\hat{R} = R \cup \{\hat{r}\}$, oder*
- *eine Regel verallgemeinert wurde, d.h. $\hat{R} = R \setminus \{r\} \cup \{\hat{r}\}$, wobei $\mathsf{en}.r \subset \mathsf{en}.\hat{r}$, $output_r \equiv output_{\hat{r}}$ und $assign_r \equiv assign_{\hat{r}}$ gilt.*

Dann gibt es ein Modellverhalten N über Schnittstelle (I, O) mit

$$R_{Z_{\hat{G}}} = R_{Z_G} \cup N.$$

Das hinzugefügte Modellverhalten N zeichnet sich dadurch aus, dass in jedem Ablauf aus N die hinzugefügte Regel bzw. die verallgemeinerte Regel mindestens einmal geschaltet hat.

5.5.4. Anwendung am Beispiel

In diesem Abschnitt veranschaulichen wir die eingeführten Operationen an einem Beispiel von geringer Komplexität. Wir wenden mindestens je eine Operation aus den Abschnitten 5.5.1, 5.5.2 und 5.5.3 am Modell des Stapels aus Abschnitt 4.5, Beispiel 4.1 an.

Ziel Das Modell des Stapels aus Beispiel 4.1 soll erweitert werden, dass mittels einer Resetnachricht der Stapel zur Laufzeit in seinen Grundzustand zurückgesetzt werden kann. Um diese Anforderung auf Modellebene umzusetzen muss die Eingabeschnittstelle des Modells erweitert werden, die den Empfang einer Resetnachricht ermöglicht. Prinzipiell gibt es zwei Möglichkeiten dies zu erreichen, den Typ des bereits vorhandenen Eingabekanals um eine Resetnachricht zu erweitern oder einen neuen Eingabekanal einzuführen, auf dem eine Resetnachricht gesendet werden kann. Wir entscheiden uns für die letztere Möglichkeit, um konzeptuell anhand der Schnittstelle zwischen gewöhnlichen Stapeloperationen und Stapelkontrolloperationen zu unterscheiden.

Beispiel 5.1 (Erweiterung der Eingabeschnittstelle). *Wir erweitern die Eingabeschnittstelle des Stapelmodells aus Beispiel 4.1, indem wir einen Eingabekanal* `r` *vom Typ*

```
data dReset = reset;
```

hinzufügen. Die Schnittstelle des Modells lautet nun wie folgt:

```
inputchannel  dInput e;
```

```
              dReset r;
outputchannel dData  a = epsilon;
localvariable dList  st = empty;
```

Wenn die Regelmenge aus Tabelle 4.1 unverändert bleibt, zeigt das Modell gemäß Proposition 5.2 zusätzliches Verhalten, da die Eingabemuster der Verhaltensregeln die Belegung des Eingabekanals `r` *nicht einschränken. Tabelle 5.1 zeigt zwei Beispielabläufe, die aufgrund dieses Effekts jetzt zum Modellverhalten gehören.*

Kanal e	Kanal r	Kanal a
ϵ	reset	ϵ
ϵ	ϵ	ϵ
ϵ	reset	ϵ

(a)

Kanal e	Kanal r	Kanal a
push(item1)	reset	ϵ
get	reset	ϵ
pop	reset	item1

(b)

Tabelle 5.1.: Beispielabläufe

Obwohl das Verhalten dargestellt in Tabelle 5.1(a) aus Schnittstellensicht zufällig gegenüber dem Verhalten, das wir intuitiv von einem Stapel mit Resetfunktion in dieser Situation erwarten, korrekt ist, ist das Verhalten dargestellt in Tabelle 5.1(b) nicht das, was wir bei dieser Eingabefolge erwarten. Um dieses Verhalten zu vermeiden, müssen wir das Modellverhalten korrigieren, indem wir die Eingabemuster der Verhaltensregeln einschränken.

Beispiel 5.2 (Korrektur von Verhalten). *Wir korrigieren das Modellverhalten des Stapels aus Beispiel 5.1, indem wir die Eingabemuster der Regeln aus Tabelle 4.1 verstärken (vgl. Proposition 5.5). Dazu modellieren wir durch Hinzufügen des Eingabemusters* `r?`ϵ *zu jeder Regel, dass bei keiner Regel der Eingabekanal r mit einer Nachricht belegt sein darf. Das korrigierte Regelsystem ist in Tabelle 5.2 dargestellt, wobei die Änderungen gegenüber dem vorherigen Regelsystem fett gedruckt sind.*

Name	Pre	Input	Output	Assign
pushItem	True	e?push(DATA); **r?**ϵ		st=List(DATA,st)
getItem	not(isE(st))	e?get; **r?**ϵ	a!ft(st)	
popItem	not(isE(st))	e?pop; **r?**ϵ		st=rt(st)
idle	True	e?ϵ; **r?**ϵ		

Tabelle 5.2.: Korrigierte Verhaltensregeln des Stapelmodells

Durch diese Korrektur erreichen wir, dass unerwünschtes Verhalten, das durch die Schnittstellenerweiterung implizit hinzugefügt wurde, wieder entfernt wird, d.h. dass z.B. der Ablauf aus Tabelle 5.1(b) im Modellverhalten des Stapels gemäß Tabelle 5.2 nicht mehr enthalten ist.

Im nächsten Schritt vollziehen wir die eigentliche Verhaltenserweiterung, indem wir eine Verhaltensregel hinzufügen, die die Reaktion beim Empfang einer Resetnachricht modelliert.

Beispiel 5.3. *Wir erweitern das Modellverhalten des Stapelmodells (vgl. Proposition 5.8), indem wir eine Regel mit Namen* `reset` *hinzufügen, die bei Empfang der Nachricht* `reset` *auf Kanal* `r` *die lokale Variable* `st` *auf* `empty` *zurücksetzt. Es ergibt sich das Regelsystem in Tabelle 5.3, wobei die hinzugefügte Verhaltensregel fett gedruckt ist. Nun enthält das*

Name	Pre	Input	Output	Assign
pushItem	True	e?push(DATA); r?ϵ		st=List(DATA,st)
getItem	not(isE(st))	e?get; r?ϵ	a!ft(st)	
popItem	not(isE(st))	e?pop; r?ϵ		st=rt(st)
idle	True	e?ϵ; r?ϵ		
reset	**True**	**r?reset**		**st=empty**

Tabelle 5.3.: Erweitertes Modell des Stapels

Modellverhalten des Stapels die im obigem Ziel geforderten Abläufe. Tabelle 5.4 zeigt zwei Beispiele.

Kanal e	Kanal r	Kanal a
push(item1)	ϵ	ϵ
get	ϵ	ϵ
ϵ	reset	item1

(a)

Kanal e	Kanal r	Kanal a
push(item1)	ϵ	ϵ
get	reset	ϵ
ϵ	ϵ	ϵ

(b)

Tabelle 5.4.: Beispielabläufe

Durch die Wahl des Eingabemusters der Regel `reset` wurde erreicht, dass das Verarbeiten einer Resetnachricht Vorrang vor allen anderen hat, d.h. Nachrichten, die gleichzeitig mit einer Resetnachricht anliegen, werden ignoriert. Tabelle 5.4(b) zeigt dazu einen Beispielablauf, bei dem die Ausgabe zur `get`-Nachricht unterdrückt wird, da gleichzeitig eine Resetnachricht anliegt und diese vorrangig verarbeitet wird. Durch diese Festlegung wurde die Anforderung, die im obigen Ziel formuliert wurde, präzisiert und die Auslassung, was bei gleichzeitigem Ankommen von Nachrichten geschehen soll, behoben.

5.6. Zusammenfassung

Dieses Kapitel hat zum Gegenstand die Definition eines Inkrementbegriffs auf Verhaltensmodellen und die Einführung eines zugehörigen Entwicklungsprozesses. Zweck dieser

Begriffsbildung ist einerseits die Erfahrungen, die sich bei der inkrementellen Entwicklung von komplexen Testmodellen bewährt haben, formal zu fassen und zu verstehen und andererseits die Grundlage zu schaffen, für diesen Entwicklungsansatz Werkzeugunterstützung entwickeln zu können. Wir fassen die wesentlichen Aussagen des Kapitels wie folgt zusammen:

- Aus Sicht des Modellverhaltens unterscheiden wir zwei Formen der Inkrementbildung:

  Inkrementbildung unter Schnittstellenerweiterung Diese behandelt den Fall, dass die aktuelle Schnittstelle eines Modells nicht ausreicht, um im nächsten Entwicklungsschritt hinzuzufügende Anforderungen an das Modellverhalten modellieren zu können. Die wesentliche Forderung bei der Inkrementbildung unter Schnittstellenerweiterung ist, dass das Modellverhalten des Ausgangsmodells vollständig in das Modellverhalten des Modells mit der erweiterten Schnittstelle eingebettet ist.

  Inkrementbildung unter Verhaltenserweiterung Dieser Begriff setzt das Modellverhalten zwischen Ausgangsmodell und Folgemodell unter der Vorraussetzung in Beziehung, dass beide Modelle die gleiche Schnittstelle besitzen. Dieser Entwicklungsschritt legt die Spezifikation einer existenziellen Eigenschaft fest, welches Verhalten das Folgemodell gegenüber dem Ausgangsmodell erhalten muss, und eine weitere existenzielle Eigenschaft, um welches Verhalten das Ausgangsmodell gegenüber dem Folgemodell mindestens erweitert wird. Die existenziellen Eigenschaften werden in der Praxis durch endliche Mengen von Modellabläufen mit endlicher Länge definiert. Die Gültigkeit der Eigenschaften wird durch Test gegenüber dem Verhaltensmodell nachgewiesen.

  Eine alternative Charakterisierung besagt, dass das Ausgangsmodell und das Folgemodell in einer Modifikationsbeziehung zueinander stehen, d.h. dass es ein Modellverhalten gibt, das im Ausgangsmodell korrigiert wurde, und eines, das dem Ausgangsmodell hinzugefügt wurde.

- Aus diesem Inkrementbegriff ergibt sich ein zugehöriger inkrementeller Entwicklungsprozess. Dieser Prozess bildet eine Folge von Modellen, bei der zwei aufeinander folgende Modelle durch Inkrementbildung unter Verhaltenserweiterung verknüpft sind. Die Inkrementbildung schließt das Ableiten und das Prüfen der Gültigkeit von zu erhaltenden und hinzufügenden Eigenschaften mit ein. Bei Bedarf geht eine Inkrementbildung unter Schnittstellenerweiterung voraus. Eine wesentliche Eigenschaft des Prozesses ist, dass die vom Entwickler festgelegten existenziellen Eigenschaften über den gesamten Prozess anwachsen und erhalten bleiben, jedoch Verhalten außerhalb dieser Eigenschaften jederzeit korrigiert und verändert werden darf. Der Erhalt der existenziellen Eigenschaften ist ein wesentlicher Beitrag, das Vertrauen in die Korrektheit des Testmodells gegenüber den funktionalen Anforderungen des SUT zu erhalten.

- Wir entwickeln die Modellinkremente operationell mit Regelsystemen. Eine Inkre-

mentbildung hat zum Ziel, eine Anforderung an eine Funktionalität bzw. an ein Verhalten in das Modell zu integrieren. Zur Umsetzung unterscheiden wir drei Gruppen von Operationen:

Vorbereiten einer Verhaltenserweiterung Reicht die syntaktische Schnittstelle oder der lokale Datenraum des Regelsystems nicht aus, um die zu modellierende Anforderung in das Regelsystem integrieren zu können, müssen diese entsprechend erweitert werden. Die Erweiterung der Ausgabeschnittstelle bzw. des lokalen Datenraums haben keine Auswirkungen auf das Modellverhalten, wohingegen die Erweiterung der Eingabeschnittstelle mit einer impliziten Erweiterung des Modellverhaltens einhergeht. Diese muss ggf. im folgenden Schritt korrigiert werden.

Korrektur des Verhaltens Die Erweiterung der Eingabeschnittstelle, das Auftreten eines Konflikts zwischen Verhaltensregeln oder die Identifikation von fehlerhaften Abläufen nach Durchführen von Qualitätssicherungsmaßnahmen erfordert unter Umständen Korrekturen am Regelsystem. Wir unterscheiden die Operationen das Einschränken einer Schaltbedingung und die Modifikation der Ausgabe bzw. der Zuweisung einer Verhaltensregel. Das Einschränken einer Verhaltensregel bewirkt, dass das Modellverhalten des Regelsystems reduziert wird. Die Modifikation der Ausgabe führt zu einer Ersetzung auf einem Teil des Modellverhaltens und die Modifikation der Zuweisung einer Verhaltensregel führt u.U. zu schwer kontrollierbaren Veränderungen im Modellverhalten.

Verhaltenserweiterung des Modells Sind alle Vorbereitungen bzw. Korrekturen erledigt, dann wird die eigentliche Erweiterung des Modellverhaltens vorgenommen. Dazu stehen die Operationen Hinzufügen einer oder mehrerer Verhaltensregeln und die Verallgemeinerung einer vorhandenen Verhaltensregel zur Verfügung.

6. Transformation von Verhaltensmodellen

In diesem Kapitel führen wir ein Verfahren zur Transformation von Verhaltensmodellen ein und zeigen ihre methodische Anwendung. Transformationen auf Modellen oder Programmcode, die das beobachtbare Verhalten des Modells bzw. des Programms nicht verändern, werden in der Literatur mit *Refactoring* [MT04, FBB+99] bezeichnet. Unser Transformationsverfahren berechnet ausgehend von einem Verhaltensmodell in Form eines Regelsystems und einer gewünschten Kontrollzustandsstruktur die zugehörigen Übergänge zwischen den Kontrollzuständen. Das Ergebnis der Berechnung wird graphisch als Kontrollzustandsgraph dargestellt. Im Kontext unserer Arbeit ist dieses Verfahren methodisch bei der inkrementellen Entwicklung von Testmodellen in zweierlei Hinsicht von Bedeutung: es wird einerseits bei der Inkrementbildung eingesetzt, um durch Restrukturierung des Verhaltensmodells eine Verhaltenserweiterung vorzubereiten (vgl. Abschnitt 5.5.1), und andererseits wird das Verhaltensmodell gemäß unterschiedlicher Kontrollzustandsstrukturen restrukturiert, um bei einem Review eines komplexen Verhaltensmodells gezielt aufschlussreiche Aspekte untersuchen zu können (vgl. Abschnitt 7.4). Zur prototypischen Integration des Transformationsverfahrens in das Entwicklungswerkzeug AutoFocus streben wir eine Anbindung und Implementierung mit Hilfe der funktional-logischen Sprache Curry [Han04] an und folgen damit der Vision "invisible formal methods" von Rushby [TSR03]: um formale Methoden für den ingenieurmäßigen Einsatz verwendbar zu machen, müssen diese in Entwicklungswerkzeugen gezielt zur Lösung von speziellen und wiederkehrenden Problemen transparent für den Werkzeuganwender integriert werden.

Das Kapitel ist wie folgt strukturiert: Abschnitt 6.1 beschreibt die Grundidee des Transformationsverfahrens und zeigt formal, dass sich durch die Transformation das Modellverhalten nicht verändert. In Abschnitt 6.2 zeigen wir anhand von einfachen Beispielen die methodische Anwendung des Verfahrens. Abschnitt 6.3 diskutiert erste Erfahrungen, die beim Einsatz der funktional-logischen Sprache Curry zur Umsetzung des Transformationsverfahrens für AutoFocus-Kontrollzustandsgraphen gemacht wurden. Abschnitt 6.4 schließt das Kapitel mit einer Zusammenfassung.

6.1. Formale Beschreibung

In diesem Abschnitt erarbeiten wir ein Verfahren, wie ein Regelsystem in tabellarischer Darstellung in einen verhaltensäquivalenten Kontrollzustandsgraphen bzw. ein Kontrollzustandsgraph in einen verhaltensäquivalenten Kontrollzustandsgraphen mit ande-

rer Kontrollzustandsstruktur überführt werden kann. Abbildung 6.1 zeigt schematisch das Prinzip des Verfahrens im Überblick. Ausgangspunkt ist ein Regelsystem mit sei-

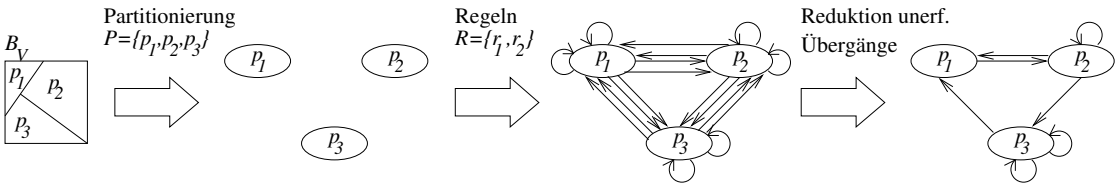


Abbildung 6.1.: Transformationsverfahren auf Verhaltensmodellen

ner Variablenmenge V und seinen Verhaltensregeln R. Zunächst wird der lokale Datenraum des Regelsystems B_V partitioniert. Jeder Partition entspricht einem Kontrollzustand im Kontrollzustandsgraphen (1. Schritt). Auf Basis der Regelmenge R werden alle Übergänge, die zwischen den Kontrollzuständen möglich sind, erzeugt, d.h. bei n Kontrollzuständen und m Regeln ergeben sich $n^2 \cdot m$ potenzielle Übergänge (2. Schritt). Abschließend werden alle Übergänge, die eine unerfüllbare Vorbedingung besitzen, aus dem Graphen entfernt (3. Schritt). Das Resultat ist ein Kontrollzustandsgraph, der die gewählte Partitionierung des Datenraums B_V als Kontrollzustände besitzt und dessen Übergänge den Verhaltensregeln R entsprechen.

Die folgenden Abschnitte diskutieren das Verfahren ausführlicher und stellen es auf eine formale Grundlage.

Transformation einer Tabelle zum Kontrollzustandsgraphen In der tabellarischen Darstellung beschreibt eine Zeile bzw. eine Verhaltensregel eine Menge von Transitionen auf dem Datenzustandsraum des Regelsystems und stellt damit eine übergangszentrierte Sicht auf das Verhaltensmodell dar. Im Allgemeinen enthält der Datenzustandsraum eines Regelsystems unendlich viele Datenzustände. Um hier das Konzept des Kontrollzustands einzuführen, partitionieren wir den Datenzustandsraum in endlich viele und disjunkte Teile, wobei wir mit jeder Partition genau einen Kontrollzustand identifizieren. Wir beschreiben eine Partition, die potenziell unendlich viele Datenzustände enthält, durch ein Prädikat über den lokalen Variablen des Regelsystems. Diesen Schritt fassen wir formal mit der folgenden Definition:

Definition 6.1. *Seien $L \subset V \subset VAR$ endliche Variablenmengen und sei P eine endliche Menge von Prädikaten über den Variablen L. Die Prädikatenmenge P partitioniert den Datenraum B_V, genau dann wenn*

1. *P überdeckt B_V, d.h.*

$$B_V = \bigcup_{p \in P} \{\alpha \mid \alpha \models p\}$$

2. *die Prädikate aus P sind paarweise nicht erfüllbar, d.h.*

$$\forall p, q \in P \text{ mit } p \neq q \text{ gilt } \{\alpha \mid \alpha \models p\} \cap \{\alpha \mid \alpha \models q\} = \emptyset$$

Nach der Partitionierung des Datenraums ergibt sich das Problem, welche Übergänge zwischen den Partitionen respektive Kontrollzustände existieren. Dazu müssen wir die Regeln des Regelsystems transformieren, so dass sie genau den Übergängen zwischen den gewählten Kontrollzuständen entsprechen. Dies erreichen wir, indem wir eine Regel durch eine Menge von Regeln ersetzen, die den potenziellen Übergängen zwischen den Kontrollzuständen entsprechen. Bei der Ersetzung wird die Vorbedingung der Regel r jeweils um zwei kontrollzustandsbeschreibende Prädikate p und q' erweitert, so dass eine neu entstandene Regel einem potenziellen Übergang zwischen den Kontrollzuständen p und q' entspricht. Durch diese Operation entstehen aus m Regeln des Regelsystems $n^2 \cdot m$ potenzielle Übergänge zwischen den Kontrollzuständen, wenn n Prädikate jeweils einen Kontrollzustand definieren. Diese Transformation des Regelsystems erzeugt ein Regelsystem, dass das gleiche Modellverhalten wie das ursprüngliche zeigt und der gewählten Partitionierung entspricht. Wir fassen die Transformation formal und zeigen den Erhalt des Modellverhaltens mit der folgenden Proposition:

Proposition 6.1. *Sei $G = (I, O, L, Init, R)$ ein Regelsystem gemäß Definition 4.13 und P eine endliche Menge von Prädikaten über den lokalen Variablen L, die den Datenraum der Zustandsmaschine B_V partitioniert. Dann ist $\tilde{G} = (I, O, L, Init, \tilde{R})$ mit*

$$\begin{aligned}\tilde{R} := \{(pre_{\tilde{r}}, input_r, output_r, assign_r) \mid\ & (pre_r, input_r, output_r, assign_r) \in R \wedge \\ & p, q \in P \wedge \\ & pre_{\tilde{r}} \equiv pre_r \wedge p \wedge q'[f_l/l']_{l \in L} \wedge \\ & assign_r \equiv \bigwedge_{l \in L} l' = f_l\},\end{aligned}$$

ein wohldefiniertes Regelsystem und G und $\tilde{G}$ induzieren die gleiche Zustandsmaschine, d.h. $Z_G = Z_{\tilde{G}}$. Insbesondere zeigen G und $\tilde{G}$ das gleiche Modellverhalten, d.h. $R_{Z_G} = R_{Z_{\tilde{G}}}$. (Dabei bezeichnet in obiger Definition von $\tilde{R}$ der Ausdruck $q'[f_l/l']_{l \in L}$ für alle $l \in L$ die Ersetzung der Variable l' durch Term f_l im Prädikat q'.)

Die Prädikate aus der Menge P entsprechen im Kontrollzustandsgraphen den Kontrollzuständen und die Regeln aus $\tilde{R}$ den potenziellen Übergängen zwischen den Kontrollzuständen. Die Übergänge sind in dem Sinne potenziell, da bei geeigneter Wahl der Prädikatmenge P viele der Regeln aus $\tilde{R}$ eine unerfüllbare Vorbedingung haben und diese folglich weggelassen werden können. Durch diese Transformation wird die detaillierte und übergangszentrierte Darstellung des Regelsystems in Form einer Tabelle in eine abstrahierte graphische und kontrollzustandszentrierte Darstellung in Form eines Kontrollzustandsgraphen überführt. Für die Anwendung des Transformationsverfahrens auf ein einfaches Beispiel verweisen wir auf Abschnitt 6.2 bzw. für die Anwendung auf ein komplexes auf Abschnitt 7.4.

Transformation eines Kontrollzustandsgraphen In diesem Abschnitt skizzieren wir, wie ein Kontrollzustandsgraph in einen verhaltensäquivalenten Kontrollzustandsgraphen

mit veränderter Kontrollzustandsstruktur überführt werden kann. Ein Verhaltensmodell in graphischer Kontrollzustandsgraphendarstellung wird als Regelsystem in tabellarischer Form interpretiert, indem die Kontrollzustände als Werte einer lokalen Variable des Regelsystems und die Kontrollzustandsübergänge als Regeln aufgefasst werden (vgl. Abschnitt 4.3). Auf Grundlage des resultierenden tabellarischen Regelsystems wird eine Partitionierung auf den lokalen Datenraum gewählt und gemäß dieser Partitionierung, die den neuen Kontrollzuständen entspricht, wie oben beschrieben der gewünschte Zielkontrollzustandsgraph erzeugt.

Verwandte Arbeiten Ähnliche Verfahren werden im Bereich des Modelcheckings zur Abstraktion von Transitionssystemen verwendet [CGL94, CGP99]. Hier wird das Verhalten eines Programms als ein Transitionssystem aufgefasst. Ziel ist es, universelle Eigenschaften, die den Anforderungen des Programms entsprechen, auf dem Transitionssystem mittels Modelchecking zu zeigen. Um dem Problem der Zustandsexplosion bei komplexen Systemen zu begegnen, wird der Zustandsraum durch Äquivalenzklassenbildung reduziert und ein entsprechendes abstraktes Transitionssystem berechnet. Bei der Abstraktion wird auf den Erhalt von gewünschten universellen Eigenschaften im folgenden Sinne geachtet: gilt die universelle Eigenschaft im abstrakten System, dann gilt sie auch im Originalsystem. Gilt sie dagegen im abstrakten System nicht, dann kann keine Aussage getroffen werden, ob sie im Originalsystem gilt oder nicht. Dams u.a. [DGG97] verallgemeinerten diese Abstraktionsansätze und untersuchten sie auch in Hinblick auf existenzielle Eigenschaften. Der Zweck unseres Transformationsverfahrens ist jedoch nicht, Abstraktion mit substanziellen Informationsverlust zu erreichen, sondern das Verhaltensmodell, ohne Informationsverlust zu restrukturieren, um den Review mittels einer abstrahierten Sicht auf das Verhaltensmodell zu erleichtern oder das Modell durch Refactoring für die Weiterentwicklung vorzubereiten.

Philipps und Rumpe [PR03, PR01] beschreiben eine Reihe von einfachen Operationen auf Strukturdiagrammen und Zustandsübergangsdiagrammen wie z.B. das Hinzufügen und Entfernen von Zuständen und Übergängen, die Refactoring- und Verfeinerungsregeln genügen. Ihr Beobachtungsbegriff ist wie bei unserem Ansatz das Modellverhalten an der Schnittstelle. Insgesamt beschreiben Sie also Refactoring- und Verfeinerungsoperationen, die vom Entwickler lokal in Modell angewendet werden können, wohingegen unser Transformationsansatz Auswirkungen auf das gesamte Verhaltensmodell hat.

6.2. Methodische Anwendung

In diesem Abschnitt wenden wir das obige Verfahren zur Transformation auf einfache Beispielmodelle mit einer lokalen Variablen an und zeigen, welchen methodischen Nutzen aus der transformierten Sicht des Verhaltensmodells gezogen werden kann. Für die Anwendung des Verfahrens auf komplexe Verhaltensmodelle mit mehreren lokalen Variablen verweisen wir auf Abschnitt 7.4.

6.2.1. Transformation von Tabelle zum Kontrollzustandsgraphen

Wie wir in Abschnitt 4.7 diskutiert haben, eignet sich die detaillierte tabellarische Darstellung besser für die zügige Modellierung und die abstrahierte Sicht in der Darstellung eines Kontrollzustandsgraphen zeigt Vorteile bei der Simulation und beim gemeinsamen Review mit Nichtmodellierungsspezialisten. Wir zeigen die Transformation bzw. das Refactoring einer Tabelle in einen Kontrollzustandsgraphen anhand des folgenden Beispiels:

Beispiel 6.1 (Transformation Tabelle). *Wir transformieren das Verhaltensmodell des Stapels aus Beispiel 4.1 von der tabellarischen Darstellung (s. Tab. 4.1) in einen Kontrollzustandsgraphen. Wir wählen als charakteristische Kontrollzustände den leeren und den gefüllten Stapel, indem wir die Prädikate $P = \{isEmpty, isFilled\}$ mit*

$$isEmpty \equiv isE(st)$$
$$isFilled \equiv not(isE(st))$$

definieren. Offensichtlich partitionieren diese Prädikate den Datenraum des Verhaltensmodells und folglich ist die Transformation aus Proposition 6.1 anwendbar. Nach der Transformation des Verhaltensmodells erhalten wir die Regeln in Tabelle 6.1, wobei in den Spalten `From` und `To` der Quell- bzw. Zielkontrollzustand eingetragen ist und zur Verdeutlichung die Regeln mit unerfüllbaren Vorbedingungen durchgestrichen sind. Der resultierende Kontrollzustandsgraph ist in Abbildung 6.2 dargestellt. Dieser zeigt die Kontrollzustände `isEmpty` und `isFilled` sowie die erfüllbaren Regeln aus Tabelle 6.1 als Kontrollzustandsübergänge zwischen den beiden Kontrollzuständen.

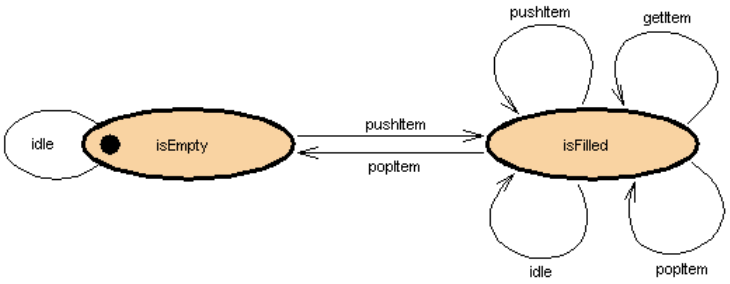


Abbildung 6.2.: Kontrollzustandsgraph des Stapels nach der Transformation

Die Transformation einer Tabelle zum Kontrollzustandsgraphen kann gezielt genutzt werden, um den Review eines Verhaltensmodells zu unterstützen. Durch geeignete Wahl unterschiedlicher Partitionen des lokalen Datenraums des Verhaltensmodells entstehen verschiedene Kontrollzustandsgraphen. Durch die abstrahierte Sicht auf das Verhaltensmodell wird die Überprüfung erleichtert, ob alle erwarteten Kontrollzustandsübergänge vorhanden sind, einige von den erwarteten Übergängen fehlen oder mehr als die erwarteten entstehen. Wir demonstrieren dies an einem einfachen Beispiel:

From	To	Name	Pre	Input	Output	Assign
~~isEmpty~~	~~isEmpty~~	~~push-Item~~	~~isE(st) && isE(List(DATA,st))~~	~~e?push(DATA)~~		~~st=List(DATA,st)~~
isEmpty	isFilled	push-Item	isE(st) && not(isE(List(DATA,st)))	e?push(DATA)		st=List(DATA,st)
~~isFilled~~	~~isEmpty~~	~~push-Item~~	~~not(isE(st)) && isE(List(DATA,st))~~	~~e?push(DATA)~~		~~st=List(DATA,st)~~
isFilled	isFilled	push-Item	not(isE(st)) && not(isE(List(DATA,st)))	e?push(DATA)		st=List(DATA,st)
~~isEmpty~~	~~isEmpty~~	~~get-Item~~	~~not(isE(st)) && isE(st) && isE(st)~~	~~e?get~~	~~a!ft(st)~~	
~~isEmpty~~	~~isFilled~~	~~get-Item~~	~~not(isE(st)) && isE(st) && not(isE(st))~~	~~e?get~~	~~a!ft(st)~~	
~~isFilled~~	~~isEmpty~~	~~get-Item~~	~~not(isE(st)) && not(isE(st) && isE(st))~~	~~e?get~~	~~a!ft(st)~~	
isFilled	isFilled	get-Item	not(isE(st)) && not(isE(st)) && not(isE(st))	e?get	a!ft(st)	
~~isEmpty~~	~~isEmpty~~	~~pop-Item~~	~~not(isE(st)) && isE(st) && isE(rt(st))~~	~~e?pop~~		~~st=rt(st)~~
~~isEmpty~~	~~isFilled~~	~~pop-Item~~	~~not(isE(st)) && isE(st) && not(isE(rt(st)))~~	~~e?pop~~		~~st=rt(st)~~
isFilled	isEmpty	pop-Item	not(isE(st)) && not(isE(st)) && isE(rt(st))	e?pop		st=rt(st)
isFilled	isFilled	pop-Item	not(isE(st)) && not(isE(st)) && not(isE(rt(st)))	e?pop		st=rt(st)
isEmpty	isEmpty	idle	isE(st) && isE(st)	e?ϵ		
~~isEmpty~~	~~isFilled~~	~~idle~~	~~isE(st) && not(isE(st))~~	~~e?ϵ~~		
~~isFilled~~	~~isEmpty~~	~~idle~~	~~not(isE(st)) && isE(st)~~	~~e?ϵ~~		
isFilled	isFilled	idle	not(isE(st)) && not(isE(st))	e?ϵ		

Tabelle 6.1.: Verhalten des Stapels nach der Transformation

Beispiel 6.2 (Reviewunterstützung). *Wir legen diesmal das Modell eines Stapels zugrunde, das in Tabelle 6.2 abgebildet ist. Nach Transformation gemäß der Partition*

Name	Pre	Input	Output	Assign
pushItem	True	e?push(DATA)		st=List(DATA,st)
getItem	not(isE(st))	e?get	a!ft(st)	**st=rt(st)**
popItem	not(isE(st))	e?pop		st=rt(st)
idle	True	e?ϵ		

Tabelle 6.2.: Verhalten des Stapels zu Beispiel 6.2

$P = \{isEmpty, isFilled\}$ mit $isEmpty \equiv isE(st)$ und $isFilled \equiv not(isE(st))$ erhalten wir den Kontrollzustandsgraphen in Abbildung 6.3. Bei der Untersuchung des Graphen

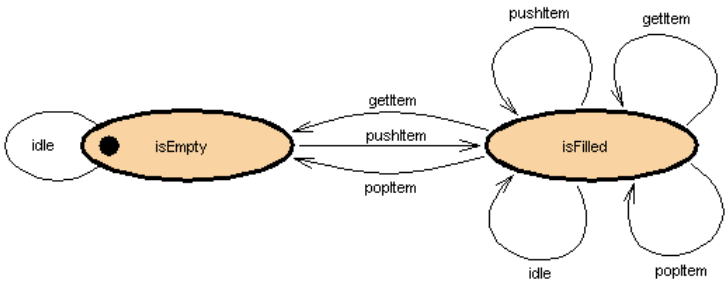


Abbildung 6.3.: Kontrollzustandsgraph des Stapels zu Beispiel 6.2

stellt sich heraus, dass dieser fälschlicherweise den Kontrollzustandsübergang `getItem` *von* `isFilled` *nach* `isEmpty` *zu viel enthält, da durch die* `get`*-Operation der Füllstand des Stapels nicht verändert werden soll. Auf diese Weise wird ein Fehler in der Zuweisung der Regel* `getItem` *aufgedeckt.*

In diesem einfachen Beispiel hat der Review des Kontrollzustandsgraphen 6.3 das Auffinden des Fehlers in der Regel `getItem` gegenüber dem direkten Review der Tabelle 6.2 nicht wesentlich erleichtert. Im Gegensatz zu obigem Beispielmodell besitzen Modelle von Systemen industrieller Komplexität mehrere lokale Datenvariablen und wesentlich komplexere Vorbedingungen und Zuweisungen. Durch die Anwendung der Transformation können gezielt unwichtige Details ausgeblendet werden und der Kontrollfluss bezüglich einer gewünschten Kontrollzustandsstruktur studiert werden. Für die Anwendung des Transformationsverfahrens auf eine komplexe Fallstudie verweisen wir auf Abschnitt 7.4. Dort wird am Beispiel deutlich, welchen Nutzen die Bildung einer abstrahierten Sicht auf ein komplexes Verhaltensmodell beim Review desselbigen hat.

6.2.2. Transformation von Kontrollzustandsgraphen

Während des Entwicklungsprozesses von Verhaltensmodellen ist es häufig wünschenswert, die Kontrollzustandsstruktur eines Kontrollzustandsgraphen unter Erhalt des Modellverhaltens zu verändern, d.h. ein Refactoring eines Kontrollzustandsgraphen durchzuführen. Die Motivation dazu ist häufig, die Struktur des Modells zu verbessern, um eine Erweiterung des Modells vorzubereiten oder überhaupt zu ermöglichen. Für eine ausführliche Abhandlung von Refactorings auf Modellebene mit Schwerpunkt auf Architekturtransformationen verweisen wir auf die Arbeit von Wißpeintner [Wiß05]. Im nächsten Beispiel zeigen wir, wie das Transformationsverfahren aus Abschnitt 6.1 auch zum Refactoring eines Kontrollzustandsgraphen eingesetzt werden kann.

Beispiel 6.3 (Refactoring von Kontrollzustandsgraphen). *Wir transformieren den Kontrollzustandsgraphen in Abbildung 6.2 aus Beispiel 6.1. Die Beschreibung der Übergänge des Kontrollzustandsgraphen des Stapels können aus Tabelle 6.1 entnommen werden. Wir nehmen an, dass für die folgenden Entwicklungsschritte neben den Stati* `isEmpty` *und* `isFilled` *auch entscheidend ist, ob der Stapel einen kritischen Füllstand erreicht hat. Wir partitionieren also den Datenraum des Stapels mit den Prädikaten* $P = \{isEmpty, isFilled, isCritical\}$*, wobei*

$$\begin{aligned} isEmpty &\equiv isE(st) \\ isFilled &\equiv not(isE(st)) \;\&\&\; length(st) \leq c \\ isCritical &\equiv not(isE(st)) \;\&\&\; length(st) > c \end{aligned}$$

gilt. Dabei berechnet die Funktion `length` *die Länge der Liste* `st` *und* `c` *bezeichnet eine beliebige ganzzahlige Konstante mit* `c>0`*. Nach Anwendung der Transformation erhalten wir den Kontrollzustandsgraphen in Abbildung 6.4. Durch die Transformation des Verhaltensmodells wurde die Weiterentwicklung des Modells vorbereitet, um das Verhalten im Kontrollzustand* `isCritical` *gezielt modifizieren zu können.*

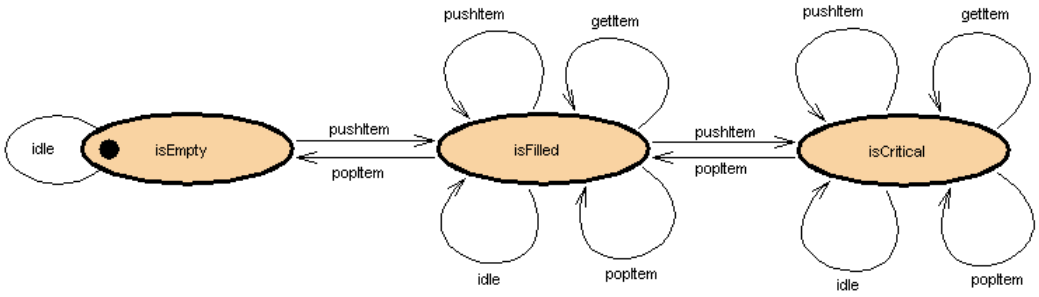


Abbildung 6.4.: Kontrollzustandsgraph des Stapels nach Transformation von Abb. 6.2

6.3. Werkzeugunterstützung

In diesem Abschnitt skizzieren und diskutieren wir die mögliche prototypische Implementierung des Verfahrens aus Abschnitt 6.1 in das Werkzeug AutoFocus (vgl. Abschnitt 7.2) zur Transformation von Kontrollzustandsgraphen (STDs).

6.3.1. Beschreibung der Werkzeugunterstützung

Das Hauptproblem bei der technischen Umsetzung des Transformationsverfahrens ist die Überprüfung, ob die konstruierten Vorbedingungen der neuen Kontrollzustandsübergänge erfüllbar sind, d.h. ob es eine Belegung der lokalen Variablen gibt, so dass die konstruierte Vorbedingung zu `True` ausgewertet wird. Die Vorbedingungen können also als existenzquantifizierte funktionale Terme aufgefasst werden. Die Erfüllbarkeit dieser Terme ist aber im allgemeinen Fall nicht entscheidbar. Die Nichtentscheidbarkeit spielt jedoch für die Praxis eine untergeordnete Rolle, da die Vorbedingungen durch ihren Praxisbezug wohlstrukturiert sind, so dass sie von Rewriting Systemen effizient vereinfacht werden können.

Wir verwenden die funktional-logische Curry [Han04] zur Überprüfung der Erfüllbarkeit der konstruierten Vorbedingungen. Curry bietet den Vorteil, dass sich die Datentyp- und Funktionsdefinitionen aus AutoFocus leicht nach Curry übertragen lassen und Curry über einen leistungsfähigen *narrowing*-Algorithmus verfügt. Wir legen folgende Entscheidungsprozedur fest: kann Curry eine Lösung d.h. eine Belegung finden, so dass eine konstruierte Vorbedingung zu `True` ausgewertet wird, dann ist Vorbedingung erfüllbar und der zugehörige Übergang wird weiterverwendet. Im anderen Fall ist sie unerfüllbar und der zugehörige Übergang wird verworfen.

Zur Beschreibung der technischen Umsetzung des Transformationsverfahrens nehmen wir vereinfachend an, dass die Information des Kontrollzustandsgraphen vollständig in seinen Übergängen codiert ist, d.h. er kann in einer Tabelle von der Form, wie in Abschnitt 4.5 beschrieben, dargestellt werden[1]. Es folgt eine informelle Beschreibung einer möglichen technischen Umsetzung des Transformationsverfahren aus Proposition 6.1 mit Hilfe der Programmiersprache Curry:

1. Ausgangspunkt ist ein AutoFocus-STD interpretiert als Regelsystem mit Regelmenge R, eine Partitionierung des lokalen Datenraums in Form der Prädikatenmenge P und die zugehörigen Datentyp- bzw. Funktionsdefinitionen DTD formuliert in funktionalen Sprache QuestF [BLS00].

2. Es werden die Datentyp- und Funktionsdefinitionen DTD in Curry-Datentypen bzw. Curry-Funktionsdefinition übersetzt.

[1] Dies ist ohne Beschränkung der Allgemeinheit möglich, da die Kontrollzustände als Werte einer lokalen Variablen aufgefasst werden können (vgl. Abschnitt 4.6).

3. Es werden für alle Regeln $r \in R$ die Vorbedingungen pre_r und die rechten Seiten der Zuweisungen $assign_r \equiv \bigwedge_{l \in L} l' = f_l$ nach Curry übersetzt. Ferner werden die Vorbedingungen der potenziellen Übergänge $\tilde{r} \in \tilde{R}$ gemäß zweier Partitionen $p, q \in P$ und der Vorschrift $pre_{\tilde{r}} \equiv pre_r \wedge p \wedge q'[f_l/l']_{l \in L}$ aus Proposition 6.1 konstruiert.

4. In Curry werden Lösungen zur Erfüllung der konstruierten Vorbedingungen $pre_{\tilde{r}}$ der potenziellen Übergänge $\tilde{r} \in \tilde{R}$ gesucht. Existiert eine Lösung, dann wird die Vorbedingung ggf. vereinfacht und im zugehörigen Übergang $\tilde{r}$ ersetzt. Im anderen Fall wird der Übergang $\tilde{r}$ aus $\tilde{R}$ entfernt.

5. Das Resultat der Berechnung sind die erfüllbaren Übergänge $\tilde{R}$ gemäß der Partitionierung P.

6. Aus den verbleibenden Übergängen wird der zugehörige Kontrollzustandsgraph in AutoFocus erzeugt.

Wir zeigen zur Verdeutlichung die Umsetzung des Transformationsverfahrens am folgenden Beispiel.

Beispiel 6.4 (Transformationsberechnung mit Curry). *Wir zeigen die Berechnung in Curry anhand der bekannten Transformation aus Beispiel 6.1. Der Transformation liegen die Regeln der Tabelle 4.1, die Partitionierung $P = \{isEmpty, isFilled\}$ mit $isEmpty \equiv isE(st)$ bzw. $isFilled \equiv not(isE(st))$ und die Datentyp- und Funktionsdefinitionen aus Beispiel 4.1 zugrunde. Wie erwartet, entsteht nach der Übersetzung des Modells in Curry und der automatischen Überprüfung der Vorbedingungen auf ihre Erfüllbarkeit der Kontrollzustandsgraph aus Abbildung 6.2, wobei die resultierenden Übergänge in Tabelle 6.3 berechnet wurden.*

From	To	Name	Pre	Input	Output	Assign
isEmpty	isFilled	pushItem	isE(st)	e?push(DATA)		st=List(DATA,st)
isFilled	isFilled	pushItem	not(isE(st))	e?push(DATA)		st=List(DATA,st)
isFilled	isFilled	getItem	not(isE(st))	e?get	a!ft(st)	
isFilled	isEmpty	popItem	not(isE(st)) && isE(rt(st))	e?pop		st=rt(st)
isFilled	isFilled	popItem	not(isE(st)) && not(isE(rt(st)))	e?pop		st=rt(st)
isEmpty	isEmpty	idle	isE(st)	e?ϵ		
isFilled	isFilled	idle	not(isE(st))	e?ϵ		

Tabelle 6.3.: Vereinfachtes Verhalten des Stapels nach der Transformation mit Curry

6.3.2. Diskussion

Bisher wurden lediglich AutoFocus-Kontrollzustandsgraphen (STDs) aus den Beispielen 6.1, 6.2 und 6.3, sowie der industriellen Fallstudie aus Kapitel 7 von Hand in ein Curry-Programm überführt, um die Automatisierbarkeit des Transformationsverfahrens zu evaluieren. Dazu wurden neben den Übergängen des STDs alle zugehörigen Datentypen und Funktionsdefinitionen in ein Curry-Programm übersetzt. Nach Bildung der potenziellen Übergänge wurden deren Vorbedingungen automatisch auf Erfüllbarkeit überprüft. Die erfüllbaren Übergänge werden im Anschluss wiederum von Hand nach AutoFocus übertragen. Insgesamt lieferte die Berechnung insbesondere auch für das komplexe Beispiel aus Abschnitt 7.4 die erwarteten Ergebnisse. Aufgrund dieses Erfolgs ist geplant mit Hilfe von Studentenprojekten, eine prototypische Integration des Verfahrens in AutoFocus entwickeln zu lassen. Insgesamt führte die Evaluation zu folgenden Beobachtungen:

- Gibt es im Modell lokale Variablen mit unendlich großen Datentypen, wie es z.B. bei unseren Fallstudien in Form rekursiven Listen der Fall ist, dann muss deren maximale Instanziierungstiefe eingeschränkt werden, da sonst die Erfüllbarkeitsprüfung bei unerfüllbaren Vorbedingungen u.U. nicht terminiert. In der Praxis stellt dies aber im Allgemeinen kein Hindernis dar. Zum Beispiel entspricht der Listenlänge der lokalen Variablen in der Fallstudie aus Kapitel 7 die Anzahl der Geräte in der Umgebung des modellierten Controllers. Für die Prüfung der Erfüllbarkeit beschränkten wir in der entsprechenden Curryübersetzung die maximale Instanziierungstiefe der Variablen auf fünf.
- Bei der Überführung der berechneten Übergänge nach AutoFocus ergibt das Problem des automatischen Layout des resultierenden STDs. Bei den vorliegenden Beispielen wurde das Layout von Hand erstellt. Für die prototypische Integration des Verfahrens in AutoFocus schlagen wir vor, Standardlayoutverfahren [BETT94] zu verwenden.
- Durch die Transformation können u.U. Übergänge erzeugt werden, deren Vorbedingungen zwar erfüllbar sind, aber keiner der zugehörigen Datenzustände im Modell jemals erreicht werden kann. In diesem Sinne ist also eine Erzeugung von "totem Code" möglich. Für das resultierende Modellverhalten spielt das zwar keine Rolle, aber aus methodischer Sicht entstehen unnötige Übergänge, die ggf. mühsam identifiziert und entfernt werden müssen.

6.4. Zusammenfassung

In diesem Kapitel stellten wir ein Verfahren zur strukturellen Transformation von Verhaltensmodellen vor, bei dem das Modellverhalten nicht verändert wird. Wir fassen den Inhalt des Kapitels wie folgt zusammen:

- Die Grundidee des Verfahrens ist, den Datenraum eines Verhaltensmodells zu partitionieren. Jeder Partition entspricht einem Kontrollzustand des Verhaltensmodells. Die Übergänge bezüglich dieser Kontrollzustandspartitionierung werden konstruiert, indem je ein Übergang im ursprünglichen Verhaltensmodell durch eine Menge von potenziellen Übergängen zwischen den Partitionen ersetzt wird. Dabei unterscheiden sich die neu konstruierten Übergänge lediglich in den Vorbedingungen von den ursprünglichen. Das Transformationsverfahren kann sowohl zur Überführung einer Tabelle in einen Kontrollzustandsgraphen als auch zur Umstrukturierung eines Kontrollzustandsgraphen verwendet werden.
- Das Transformationsverfahren kann methodisch einerseits zur Bildung einer abstrahierten Sicht auf das Verhaltensmodell und andererseits zum Refactoring eines Verhaltensmodells eingesetzt werden. Durch die Bildung einer abstrahierten Sicht auf ein Verhaltensmodell wird bei geeigneter Wahl der Kontrollzustandspartitionierung der Review des Modells und die Verfolgung des abstrakten Kontrollflusses bei der Simulation des Modells erleichtert. Wird das Verfahren zum Refactoring von Verhaltensmodellen verwendet, dann kann es durch erfolgte Umstrukturierungen anstehende Erweiterungen bzw. Veränderungen am Modell gezielt vorbereiten.
- Bisher wurde lediglich die Automatisierbarkeit des Verfahrens untersucht, indem AutoFocus-Kontrollzustandsgraphen (STDs) von Hand in die funktional-logische Curry übersetzt wurde und die in Curry implementierten Verfahren benutzt wurden, um automatisch die konstruierten Vorbedingungen zu vereinfachen und auf ihre Erfüllbarkeit zu prüfen. Aufgrund der viel versprechenden Ergebnisse dieser Untersuchung planen wir in Studentenprojekten, eine vollständige Anbindung von AutoFocus an Curry zu entwickeln, so dass ein Nutzer von AutoFocus das Verfahren zur Transformation von STDs transparent im Bezug auf Curry einsetzen kann.

7. Anwendung in der Praxis

In diesem Kapitel demonstrieren wir die Konzepte, die wir bisher in der Arbeit erarbeitet haben, anhand einer industriellen Fallstudie und zeigen ihren Nutzen. Gegenstand der Modellierung und später des modellbasierten Testens ist ein Netzwerkcontroller für ein Infotainmentnetzwerk im Automobilbereich. Dieser Netzwerkcontroller zeichnet sich durch seine hohe Komplexität aus, da er viele Spezialfälle in unterschiedlichsten Kontexten erkennen und richtig behandeln muss, um eine möglichst störungsfreie Funktion des Netzwerks gewährleisten zu können. Der erste Teil des Kapitels beschäftigt sich mit der inkrementellen Entwicklung eines Testmodells für den Netzwerkcontroller und zeigt den Nutzen der Modellbildung in der Anforderungsanalyse. Während der Modellbildung wurden drei grobe Spezifikationsfehler und 7 größere Lücken in den Spezifikationsdokumenten des Netzwerkcontrollers aufgedeckt. Der zweite Teil des Kapitels beschäftigt sich mit dem modellbasierten Testen des Netzwerkcontrollers. Dieser Abschnitt liefert in der Literatur den ersten uns bekannten Vergleich von modellbasierten und traditionell von Hand erzeugten Tests. Die Vergleichskriterien sind die Fehleraufdeckungsrate, die Bedingungs-/Entscheidungsüberdeckung auf dem Modell und der Implementierung. Da die Zahlen nur auf Basis einer Fallstudie ermittelt wurden, liefern sie lediglich Indizien für allgemeine Zusammenhänge. Um allgemein gültige statisch belastbare Schlüsse ziehen zu können, müssen in Zukunft noch viele weitere Studien vergleichbarer Art durchgeführt werden. Die wichtigsten Ergebnisse der vorliegenden Studie fassen wir bereits hier kurz zusammen: (1) Tests, die ohne Modell abgeleitet wurden, decken weniger Fehler auf als modellbasierte Tests. Die Anzahl der aufgedeckten *Programmierfehler* ist in etwa gleich, aber die Anzahl der aufgedeckten *Lastenheftfehler*[1] ist bis zu sechs mal höher. (2) Automatisch und modellbasiert generierte Tests decken genauso viele Fehler auf wie von Hand erzeugte modellbasierte Tests. Die Letzteren führen zu einer höheren Abdeckung auf dem Modell als die Ersteren, aber zu einer niedrigeren auf der Implementierung. (3) Wir konnten lediglich eine moderate Korrelation zwischen der Bedingungs-/Entscheidungsabdeckung auf dem Modell und der Implementierung nachweisen. (4) Obwohl nur eine moderate Korrelation zwischen Modell und Implementierung bezüglich Bedingungs-/Entscheidungsabdeckung existiert, zeichnet sich eine deutliche Korrelation zwischen der Abdeckung auf dem Modell bzw. der Implementierung und der Fehleraufdeckungsrate ab. Es wurde aber auch festgestellt, dass eine höhere Abdeckung sowohl auf dem Modell als auch auf der Implementierung nicht zwangsweise zu einer höheren Fehleraufdeckungsrate führt.

Das Kapitel ist wie folgt gliedert: Abschnitt 7.1 führt die Fallstudie und Abschnitt 7.2 das

[1] Fehler, die eine Veränderung in den Anforderungs- bzw. Spezifikationsdokumenten erfordern würde.

Werkzeug AutoFocus ein. Das Ausführungsmodell von AutoFocus ist sehr ähnlich zu dem von Regelsystemen und folglich lässt sich der Entwicklungsprozess von Testmodellen anhand von Regelsystemen leicht auf das Werkzeug AutoFocus abbilden. Abschnitt 7.3 beschreibt die inkrementelle Entwicklung des Testmodells der Fallstudie. Diese beschränkt sich auf den wesentlichen Teil der Fallstudie, da eine vollständige Beschreibung den Rahmen dieser Arbeit sprengen würde. Abschnitt 7.4 wendet die Transformationstechnik aus Kapitel 6 auf die Fallstudie an und zeigt deren Nutzen für die Qualitätssicherung des Modells. Abschnitt 7.5 beschreibt das modellbasierte Testen eines realen Netzwerkcontrollers und liefert erste Zahlen zum Vergleich von modellbasierten und traditionellen Testmethoden. Das Kapitel schließt mit einer Zusammenfassung in Abschnitt 7.6.

7.1. Fallstudie: MOST NetworkMaster

Die Ergebnisse dieser Arbeit werden anhand einer Fallstudie MOST NetworkMaster demonstriert. MOST (Media Oriented Systems Transport) ist ein optisches Ringnetzwerk optimiert zur Vernetzung von Multimediaanwendungen in Automobilen. Das MOST-Netzwerk stellt synchrone und asynchrone Kanäle zur Übertragung von synchronen Daten wie z.B. Audio oder Video bzw. asynchronen Daten wie z.B. Graphiken zur Verfügung. Die gesamte Bandbreite des Rings beträgt ca. 20 MBit.

Im folgenden führen die Grundlagen über den Aufbau eines MOST-Netzwerks ein, die zum Verständnis der Fallstudie notwendig sind. Für eine ausführliche Spezifikation von MOST verweisen wir auf öffentliche Spezifikationsdokumente [MOS99, MOS02b]. Ein MOST-Netzwerk besteht aus mehreren Geräten, die über ein Glasfaserkabel zu einem Ring vernetzt sind. Ein Gerät (engl. *device*) enthält eine oder mehrere funktionale Komponenten wie z.B. CD-Player, Tuner etc., die als Funktionsblöcke (kurz: *FBlock*) bezeichnet werden. Ein Funktionsblock enthält eine Menge von Funktionen, z.B. besitzt das Gerät CD-Player den Funktionsblock `AudioDiskPlayer`, der die Funktionen `Play`, `nextTrack`, `Stop` etc. zur Verfügung stellt. Ferner enthält jedes Gerät einen speziellen Funktionsblock namens `NetBlock`. Dieser Funktionsblocks wird benutzt, um Informationen über alle anderen Funktionsblöcke eines Geräts abfragen zu können. Geräte können die Verbindung zum Netzwerk aufbauen und unterbrechen, indem sie ihren Bypass öffnen bzw. schließen. Geräte besitzen zur Adressierung im Netzwerk eine physikalische Adresse, die aus der Position des Geräts im MOST-Netzwerk berechnet wird[2], und einer logischen Adresse, die vom Gerät zur Laufzeit verändert werden kann. Eine MOST-Nachricht ist wie folgt aufgebaut:

```
SrcAdr.TrgAdr.FBlockID.InstID.FktID.OPType(Parameter)
```

Diese besteht aus den Bestandteilen:

[2]Das Gerät, das einen speziellen Funktionsblock den sog. `TimingMaster` enthält, hat Position Null.

- *SrcAdr* ist die Absenderadresse der Nachricht. Diese ist die logische Adresse des sendenden Geräts.

- *TrgAdr* ist die Empfängeradresse der Nachricht. Diese ist in der Regel die logische Adresse des Empfängers. Bei speziellen MOST-Nachrichten z.B. Anfragen an den Funktionsblock `NetBlock` wird der Empfänger über seine physikalische Adresse adressiert.

- *FBlockID* ist eine eindeutige Bezeichnung für einen Funktionsblock (z.B. `AudioDiskPlayer`).

- *InstID* Falls es mehrere Funktionsblöcke gleicher Art im MOST-Netzwerk gibt, werden diese anhand der InstID eindeutig unterschieden. Gibt es z.B. zwei CD-Player im Netzwerk, dann werden diese durch `AudioDiskPlayer.01` und `AudioDiskPlayer.02` eindeutig voneinander unterschieden. Das Paar *FBlockID.InstID* wird auch als funktionale Adresse bezeichnet.

- *FktID* ist eine eindeutige Bezeichnung einer Funktion eines Funktionsblocks. Zum Beispiel wird die Stop-Funktion eines CD-Players mit `AudioDiskPlayer.01.Stop` aufgerufen.

- *OPType(Parameter)* steht für eine Operation, die bei einigen Funktionen angegeben werden müssen und optional über Parameter verfügen. Ein Beispiel ist die Funktion `Track` zum Setzen eines Titels `AudioDiskPlayer.01.Track.Set(5)`.

Der Aufbau der meisten Funktionsblöcke und deren Funktionen sind im MOST-Funktionenkatalog [MOS02a] standardisiert.

Im Rahmen dieser Arbeit betrachten wir die Entwicklung eines Testmodells für den MOST NetworkMaster. Es gibt nur einen NetworkMaster im MOST-Netzwerk, welcher in Form eines Funktionsblock mit Namen `NetworkMaster` im einem Gerät implementiert ist. Der NetworkMaster hat im wesentlichen drei Aufgaben:

- Aufbau und Verwalten der zentralen Registrierung (engl. *registry*): die Registrierung enthält alle im MOST Netzwerk verfügbaren Funktionsblöcke und ihre zugehörigen physikalischen und logischen Adressen.

- Auflösen von funktionalen Adressen, d.h. Geräte können beim NetworkMaster abfragen, unter welcher logischen Adresse ein gewünschter Funktionsblock ansprechbar ist.

- Überwachen der Geräte, d.h. der NetworkMaster überprüft von Zeit zu Zeit, ob alle Geräte im Netzwerk noch ansprechbar sind.

7.2. Die Beschreibungstechnik AutoFocus

In diesem Abschnitt führen wir kurz die Modellierungssprache AutoFocus [HSE97] ein. Für eine ausführliche Einführung verweisen wir auf das AutoFocus-Tutorial [LPS+02]. Wir verwenden AutoFocus zum modellieren von Testmodellen in Form von Regelsystemen. Dies ist mit kleineren Einschränkungen problemlos möglich, da Regelsysteme in Anlehnung an AutoFocus entwickelt wurden und der semantische Unterschied zwischen AutoFocus und Regelsystemen leicht umgangen werden kann (vgl. Abschnitt 7.2.5).

7.2.1. Struktur

Modelle in AutoFocus werden aus hierarchischen *Komponenten* aufgebaut, die über gerichtete und getypte Kanäle kommunizieren. Die Struktur wird mit so genannten *Systemstrukturdiagrammen (SSDs)* beschrieben. Komponenten werde durch Rechtecke und Kanäle durch Pfeile dargestellt. Die Kanäle zeigen den Datenfluss zwischen den Komponenten. Die Verbindung zwischen einer Komponente und einem Kanal wird als *Port* bezeichnet. Ein Eingabeport wird als weißgefüllter und ein Ausgabeport als schwarzgefüllter Kreis dargestellt. Jeder Ausgabeport kann mit einen oder mehreren Ausgabekanälen verknüpft sein, ein Eingabekanal dagegen höchstens mit einem Eingabekanal.

Komponenten auf untersten Hierarchiestufe können beliebig viele lokale Variablen definieren. Variablen sind ebenso wie Kanäle getypt und sind nur sichtbar für die Komponente, die sie definiert. In diesem Sinne kapseln Komponenten ihren Datenzustand. Wenn Information von Komponenten gemeinsam genutzt werden soll, dann muss diese über Kanäle ausgetauscht werden, d.h. Komponenten auf unterster Hierarchiestufe haben keine Kenntnis über die Variablen anderer Komponenten. Abschließend sei bemerkt, dass Hierarchie lediglich genutzt wird, um SSDs graphisch zu strukturieren. Folglich können Komponenten, die nicht auf unterster Hierarchiestufe stehen, keine lokalen Variablen definieren.

7.2.2. Verhalten

Das Verhalten wird durch so genannte *Zustandsübergangsdiagramme (STDs)* definiert. Diese entsprechen der graphischen Darstellung von Regelsystemen durch Kontrollzustandsgraphen (vgl. Abschnitt 4.6). Analog zu lokalen Variablen können nur Komponenten auf untersten Hierarchiestufe einen STD enthalten. STDs bestehen aus Kontrollzuständen, dargestellt durch Ovale, und Kontrollzustandsübergänge, dargestellt durch Pfeile. Der Startzustand wird mit einem schwarz gefüllten Kreis markiert. Während Kanäle den Datenfluss zwischen den Komponenten repräsentieren, repräsentieren die Kontrollzustandsübergänge den Kontrollfluss innerhalb einer Komponente. Der Zustand der Komponente besteht aus dem aktuell aktiven Kontrollzustand und der aktuellen Belegung der lokalen Variablen. Die Entscheidung, welche Projektion oder Kombination

von Projektionen des Datenzustandsraum geeignet für eine Modellierung ist, ist nicht trivial. Zu diesem Thema stellt Kapitel 6 ein Verfahren vor, wie aus einem Regelsystem unterschiedliche Repräsentationen als STD berechnet werden kann bzw. wie ein STD zu einem anderem transformiert werden, dem eine andere Projektion des Datenraums zugrunde liegt.

STDs werden zeitsynchron ausgeführt. Es existiert eine globale Uhr, die periodisch getriggert wird, was zum Begriff eines *Ticks* führt. Auf unterster Hierarchieebene der Komponenten, d.h. alle STDs, führen ihre Berechnungen gleichzeitig zwischen zwei Ticks aus.

7.2.3. Datentypen und Funktionen

Datentypen Lokale Variablen, Kanäle sowie deren angeschlossenen Ports sind getypt. Typisierung geschieht auf Basis einer funktionalen Sprache QuestF [BLS00]. Datentypen und zugehörige Funktionen werden in so genannten *Datentypdefinitionen (DTDs)* definiert. Datentypdefinitionen haben die Form

```
data d = c1 | ... | cN;
```

wobei `d` den Datentyp bezeichnet und die `c`'s Terme sind, die aus Datentypen und *Typkonstruktoren* aufgebaut sind. Typkonstruktoren treten nur auf der rechten Seite einer Datentypdefinitionsgleichung auf, Datentypen dagegen sowohl auf der linken als auch auf der rechten. Das Kopfsymbol von jedem `c` muss ein Typkonstruktor sein. Das Symbol '`|`' besagt, dass Werte des Datentyps `d` durch eines der `c`'s gegeben sind, d.h. das Symbol '`|`' wird benutzt um Vereinigungstypen zu definieren. Weiterhin können die `c`'s genutzt werden, um strukturierte Datentypen zu definieren, die u.U. rekursiv sind. Zum Beispiel definiert der Datentyp `data dList = empty | List(Int,dList);` eine rekursive definierte Liste von ganzen Zahlen. Ferner ist es möglich, den äußersten Typkonstruktor eines Terms zu überprüfen: Falls z.B. ein `c` ein Term `t(a1,...,aM)` ist, dann gibt die Funktion `is_t` den Wahrheitswert `True` zurück, wenn der äußerste Typkonstruktor der Arguments `t` ist, andernfalls `False`.

Funktionen Funktionen können auf obigen Datentypen definiert werden und sind ein wichtiger Bestandteil für die Definition von STD-Transitionen (siehe nächster Abschnitt). Auf Basis von Funktionsdefinitionen können funktionale Ausdrücke aufgebaut werden. Funktionen werden durch eine Menge von Gleichungen definiert. Eine Funktion `f` mit `N` Parametern wird mit `M` Definitionsgleichungen definiert:

```
fun f(p11,...,p1N) = rs1
   |f(p21,...,p2N) = rs2
          ...
   |f(pM1,...,pMN) = rsM;
```

wobei die `rs`'s selbst funktionale Ausdrücke sind. Die `p`'s sind die Formalparameter der Funktion und sind aus Variablen und Typkonstruktoren aufgebaut. Die Variablen auf der rechten Seite sind eine Teilmenge von denen auf der linken Seite. Wenn zur Laufzeit ein Satz von Argumenten mit einem Satz von Formalparametern matchen, dann werden die Variablen in den Formalparametern an die entsprechenden Werte der Aktualparameter gebunden und damit die rechte Seite ausgewertet, die das Ergebnis der Funktion darstellt. Wenn mehr als eine der Formalparametersätze mit den Aktualparametern matchen, dann wird die erste passende Definitionsgleichung ausgewählt. Analog zur Typdefinition können Funktionen rekursiv definiert werden. Zum Beispiel hängt die Funktion

```
fun append(empty,L) = L
   |append(List(HEAD,TAIL),L) = List(HEAD,append(TAIL,L));
```

zwei ganzzahlige Listen aneinander.

Weitere einfache Beispiele für Datentypdefinitionen und Funktionen sind in Beispiel 4.1 bzw. komplexere in Anhang B.1 zu finden.

7.2.4. Kommunikation und Berechnung

Kommunikation In einem AutoFocus-Modell ist ein Schritt wie folgt aufgebaut: Nach einem Tick lesen alle STDs ihre Eingaben an den Eingabeports. Dann führen sie ihre Berechnungen gemäß den Kontrollzustandsübergängen ausgehend vom aktuellen Kontrollzustand aus. Die Auswahl der Übergangs hängt vom aktuellen Kontrollzustand, dem aktuellen Datenzustand und der aktuellen Eingabe an den Ports ab. Nach Berechnung werden die Werte auf die Ausgabeports geschrieben. Beim Auftreten des nächsten Ticks werden die Werte der Ausgabeports auf die mit Kanälen verbundenen Eingabeports und die Kontrolle auf dem Übergang verweisenden Kontrollzustand transferiert. Dieser Transfer benötigt keine Zeit. Jetzt wiederholt sich dieser Vorgang des Ports Auslesens, Berechnens usw.

Berechnung Der Aufbau, die Auswahl und die Auswertung von Kontrollzustandsübergängen sind analog zu dem der Verhaltensregeln eines Regelsystems (vgl. Abschnitt 4.4), die wir hier nochmals kurz zusammenfassen. Ein Kontrollzustandsübergang ist wie folgt aufgebaut:

```
Vorbedingung : i1?d1;...;iN?dN : o1!e1;...;oM!eM : l1=f1;...;lP=fP
```

wobei die `i`'s eine Teilmenge der Eingabeports, die `o`'s eine Teilmenge der Ausgabeports und die `l`'s eine Teilmenge der lokalen Variablen der Komponente sind. Die vier Abschnitte eines Kontrollzustandsübergangs werden analog zu Regelsystemen mit Vorbedingung, Eingabemuster, Ausgabe und Zuweisung bezeichnet. Tritt ein Eingabeport nicht in dem Eingabemuster auf, dann ist dessen Wert nicht relevant für den Zustandstandsübergang. Tritt ein Ausgabeport nicht in der Ausgabe auf, dann wird die leere

Nachricht auf diesem geschrieben. Tritt eine lokale Variable nicht in der Zuweisung auf, dann hat sie nach dem Tick den gleichen Wert als vor dem Tick.

Die Eingabeterme `d` müssen den Typ des entsprechenden Ports besitzen und enthalten ggf. übergangslokale Variablen[3] und lokale Variablen. Übergangslokale Variablen werden zur Laufzeit gebunden, wenn die Werte der Eingabeports zu dem Eingabemuster passt. Eine Ausnahme bildet das Muster `i?`. Dieses zeigt an, dass der Eingabeport `i` mit keiner Nachricht, d.h. mit der leeren Nachricht belegt sein muss.

Die Vorbedingung ist ein funktionaler Ausdruck über lokalen Variablen und übergangslokalen Variablen vom Typ `Bool`. Wenn die aktuelle Belegung der Eingabeports zu dem Eingabemuster passt und die Vorbedingung zu `True` ausgewertet wird, dann ist der Kontrollzustandsübergang schaltbereit. Falls mehrere Übergänge schaltbereit sind, dann wird ein beliebiger ausgewählt und ausgeführt oder der Nutzer wählt zur Laufzeit einen davon aus. Wenn kein Übergang schalten kann, dann wird ein Leerschritt ausgeführt, d.h. die Komponente verbleibt im aktuellen Kontrollzustand bzw. Datenzustand und gibt auf allen Ausgabeports die leere Nachricht aus. Diese implizite Vervollständigung eines STDs führt zur Eingabevollständigkeit des STD und wird *Idlevervollständigung* genannt.

Schaltet ein Kontrollzustandsübergang, dann wird der Kontrollzustand gemäß dem Pfeilende des Übergangs aktualisiert und die Ausgabe und die Zuweisung ausgewertet:

- Ein Ausdruck `o!O` in der Ausgabe bedeutet, dass der Wert des Ausdrucks `O` auf dem Ausgabeport `o` geschrieben wird, wobei der Ausdruck `O` ein funktionaler Ausdruck vom Typ des Ausgabeports `o` ist, der übergangslokale und lokale Variablen enthalten kann.
- Eine Gleichung `l=f` in der Zuweisung bedeutet, dass die lokale Variable `l` mit dem Wert des Ausdrucks `f` aktualisiert wird. Der Ausdruck `f` ist wiederum ein funktionaler Ausdruck, der übergangslokale und lokale Variablen enthalten kann.

7.2.5. Vergleich zu Regelsystemen

Da sich unser Modellierungsansatz, den wir Kapitel 4 eingeführt haben und in der tabellarischen Beschreibungstechnik der Regelsysteme mündet, stark an die Konzepte von AutoFocus bzw. zeitsynchronen Focus anlehnt, gibt es keine gravierenden Unterschiede bzw. Brüche zwischen beiden Beschreibungstechniken. Im Folgenden identifizieren wir zwei Unterschiede und geben an, wie Regelsysteme nach AutoFocus übertragen und in AutoFocus simuliert werden:

Idlevervollständigung Regelsysteme besitzen in Vergleich zu STDs in AutoFocus-Komponenten keine Idlevervollständigung. Dieser semantische Unterschied führt dazu,

[3] Diese entsprechen den regellokalen Variablen bei Regelsystemen.

dass das Modellverhalten von Regelsystemen partiell sein kann. Dies führt zu folgenden methodischen bzw. technischen Konsequenzen:

- Durch die Idlevervollständigung der STDs ist aus Sicht des Modellverhaltens keine Unterscheidung möglich, welches Verhalten explizit modelliert und implizit vorhanden ist. Häufig ist dieses implizite Verhalten unerwünscht und dessen Auftreten u.U. schwer kontrollierbar. Bei Regelsystemen findet dagegen ein klare Trennung statt, indem kein implizites Verhalten im Modellverhalten enthalten ist. Dieser Umstand erleichtert dem Entwickler wesentlich die Prüfung der Korrektheit des Modellverhaltens, das Auffinden von Fehlern etc. Insgesamt trägt es zur intellektuellen Beherrschbarkeit der Modelle bei.

- Modellierungsoperationen wie Hinzufügen oder Verändern von Übergangen führen bei STDs zu einer allgemeinen Modifikation am Modellverhalten, d.h. es wird sowohl Verhalten hinzugefügt als auch entfernt. Bei Regelsystemen dagegen kann bei den meisten Operationen ein klare Unterscheidung getroffen werden, ob diese zu einer Erweiterung oder Reduktion des Modellverhaltens führt. Dies hat zur Folge, dass Operationen auf Regelsystemen im Entwicklungsprozess gezielt eingesetzt werden können, um das Modellverhalten zu verändern (vgl. Abschnitt 5.5).

- Durch die Eingabevollständigkeit von STDs führt die Komposition der Komponenten in AutoFocus wiederum zu eingabevollständigen Komponenten bzw. Modellen. Bei der Komposition von Regelsystemen mit partiellen Modellverhalten nimmt im Allgemeinen der Eingabebereich des komponierten Modells im Vergleich zu den ursprünglichen Modellen ab.

- Für die Testfallgenerierung enthält das Modellverhalten des Regelsystems nur die Testfälle, die relevant für das SUT sind, da durch nicht vorhandene implizite Vervollständigung kein unerwünschtes Verhalten entstehen kann. Dagegen muss häufig bei der Generierung von Testfällen aus AutoFocus-Modellen durch geeignete Wahl von Testfallspezifikationen vermieden werden, dass implizites und unerwünschtes Modellverhalten generiert wird.

Trotz diesem semantischen Unterschied kann AutoFocus zur Modellierung verwendet werden. Es ist lediglich darauf zu achten, dass z.B. bei der Simulation der Modelle die Idlevervollständigung nicht ausgeführt wird.

Portkonzept Bei der Definition von Regelsystemen haben wir kein Portkonzept eingeführt, um den Formalismus einfacher und schlanker zu halten. Auf Werkzeugebene erleichtert das Portkonzept die Handhabung von Komponenten, da dadurch jede Komponente seinen eigenen Namensraum bezüglich seiner Schnittstelle erhält. Dies erleichtert wesentlich die Komposition und die Wiederverwendung von Komponenten, denn es wären oft aufwendige Umbenennungen von Kanalnamen notwendig, wenn kein Portkonzept vorhanden wäre und die Komposition rein auf dem Übereinstimmen von Kanalnamen beruhen würde. Das fehlen des Portkonzepts bei Regelsystemen ist nur von

syntaktischer Natur und hat keine Auswirkungen auf die Ausdrucksmächtigkeit oder dergleichen.

Regelsysteme in AutoFocus Regelsysteme können einfach mit AutoFocus zur Ausführung bzw. zur Simulation gebracht werden, indem sie mit Hilfe des Transformationsverfahrens aus Kapitel 6 in einen verhaltensäquivalenten STD transformiert und in AutoFocus überführt werden. In einfachsten Fall wird die triviale Partitionierung des Datenzustandsraums *True* gewählt, so dass ein STD mit einem Kontrollzustand entsteht und alle Verhaltensregeln des Regelsystems zu Übergängen von diesem Kontrollzustand auf sich selbst werden. Bei der Simulation des übertragenen Regelsystems muss lediglich beachtet werden, das die in AutoFocus vorgesehene Idlevervollständigung (siehe oben) nicht ausgeführt wird.

7.3. Inkremente der Fallstudie

In diesem Abschnitt demonstrieren wir die inkrementelle Entwicklung eines Testmodells anhand der industriellen Fallstudie des MOST-NetworkMasters. Der Abschnitt ist gemäß dem Vorgehen aus Abschnitt 2.3 gegliedert:

- Planung der Inkrementbildung (Abschnitt 7.3.1)
- Festlegung der initialen Schnittstelle des Testmodells (Abschnitt 7.3.2)
- Inkrementelle Realisierung des Modellverhaltens gemäß der Planung (Abschnitte 7.3.3 bis 7.3.6)

In den Abschnitten 7.3.3 bis 7.3.6 beschreiben wir sehr detailliert das Vorgehen bei der Bildung der ersten vier Modellinkremente. Wir wählten den hohen Detaillierungsgrad, um einen unerfahrenen Modellentwickler die Möglichkeit zu geben, das inkrementelle Vorgehen anhand eines Beispiels von industrieller Komplexität nachzuvollziehen. Diese Abschnitte können ohne weiteres übersprungen werden, ohne das Verstehen der wesentlichen Kapitelinhalte zu beeinträchtigen.

7.3.1. Planung der Entwicklung

Zu Beginn der inkrementellen Entwicklung steht die Planung, welche und in welcher Reihenfolge die Anforderungen an das Systemverhalten im Testmodell realisiert werden. Wir beschränken die Modellierung im Testmodell auf zwei von drei Hauptfunktionalitäten des NetworkMasters (funktionale Abstraktion, vgl. Abschnitt 3.2.1). Wir modellieren "den Aufbau und das Verwalten der zentralen Registrierung" und "das Auflösen von funktionalen Adressen" und lassen aus Kostengründen "die Überwachung der Geräte" weg, da diese Funktionalität nicht entscheidend für die Funktion des Netzwerks ist, sondern lediglich zur Identifikation von häufig ausfallenden Geräten bzw. Fehlerdiagnose im

Netzwerk dient. Die Entwicklung der verbleibenden zwei Funktionalitäten spalten wir in einzelne Anforderungen an das Verhalten des NetworkMasters auf.

Anordnungsprinzip der Anforderungen Wir ordnen in der Planung die funktionalen Anforderungen an das NetworkMaster-Verhalten nach folgendem Prinzip an: Zu Beginn der Modellierung sind die Einschränkungen an das Umgebungsverhalten maximal, d.h. es wird der Idealfall modelliert, dann werden von Inkrement zu Inkrement sukzessiv die Umgebungsannahmen verringert, d.h. es wird mehr und mehr Fehlverhalten der Umgebung zugelassen. Folglich nimmt während der Entwicklung die modellierte Funktionalität zu, währenddessen die Annahmen an das Umgebungsverhalten abnehmen. Für unsere Fallstudie bedeutet dies konkret, dass im ersten Inkrement das Startverhalten des NetworkMasters modelliert wird, wenn sich die Geräte in der Umgebung fehlerfrei verhalten, und in den Folgeinkrementen das NetworkMaster-Verhalten modelliert wird, wenn Fehlverhalten in der Umgebung auftritt. Fehlverhalten bedeutet zum Beispiel, dass Geräte in der Umgebung fehlerhafte Nachrichten schicken oder auf Anfragen nicht antworten. Dabei werden Spezialfälle, die der Erfahrung nach seltener auftreten, später modelliert. Dieses Vorgehen hat den Vorteil, dass Verhalten mit hoher Nutzungswahrscheinlichkeit früh modelliert und häufig durch die wiederkehrenden Regressionstests überprüft wird. Dadurch steigt die Wahrscheinlichkeit und das Vertrauen, dass das Kernverhalten des zu entwickelnden Systems korrekt modelliert wird.

Planung Das Resultat ist vorerst ein Entwicklungsplan in Form einer Liste, in der wir kurz und informell die zu realisierenden Anforderungen beschreiben. Eine ausführliche Beschreibung der einzelnen Anforderungen erfolgt jeweils in dem Abschnitt, in dem sie in das Testmodell integriert werden.

1. Verhalten des NetworkMasters nach Einschalten des MOST-Netzwerks unter der Annahme, dass alle Geräte in der Umgebung des NetworkMasters auf Anfragen des NetworkMasters mit einer Statusmeldung antworten.
2. Verhalten des NetworkMasters, falls Geräte in der Umgebung des NetworkMasters auf Anfragen des NetworkMasters mit einer inkonsistenten Statusmeldung antworten.
3. Verhalten des NetworkMasters, falls Geräte in der Umgebung des NetworkMasters auf Anfragen des NetworkMasters nicht antworten.
4. Verhalten des NetworkMasters, falls Geräte in der Umgebung des NetworkMasters während der Laufzeit vom MOST-Netzwerk trennen oder verbinden.
5. Verhalten des NetworkMasters, falls Geräte in der Umgebung des NetworkMasters auf Anfragen des NetworkMasters mit einer Errormeldung statt mit einer Statusmeldung antworten.
6. Verhalten des NetworkMasters, falls Geräte in der Umgebung des NetworkMasters

auf Anfragen des NetworkMasters wiederholt mit einer inkonsistenten Statusmeldung antworten.

7. Verhalten des NetworkMasters, falls Geräte in der Umgebung des NetworkMasters ohne Anfrage des NetworkMasters spontan eine Statusmeldung an den NetworkMaster schicken.

8. Verhalten des NetworkMasters, falls dieser von Geräten aus seiner Umgebung Anfragen zum Auflösen von funktionalen Adressen erhält.

In dieser Arbeit werden wir in den Abschnitten 7.3.2 bis 7.3.6 die inkrementelle Entwicklungsmethodik demonstrieren, indem wir uns auf die Modellierung der ersten vier Inkremente beschränken. Die vollständige Modellierung zu beschreiben, würde den Rahmen dieser Arbeit sprengen.

7.3.2. Festlegen der Schnittstelle

Bevor wir mit der Modellierung des Modellverhaltens beginnen, muss die Schnittstelle des Testmodells festgelegt werden. Zunächst benötigen wir für das Testmodell des NetworkMasters einen Eingabekanal `fromEnv` und einen Ausgabekanal `fromNM` zum Empfangen bzw. zum Senden von MOST-Nachrichten. Zur klaren Strukturierung der Schnittstelle auf Modellebene unterscheiden wir explizit zwischen den MOST-Nachrichten, die vom NetworkMaster empfangen bzw. gesendet werden. Dazu modellieren wir die Datentypen `dMsgEnv`, `dMsgNM` und weitere, die zur Definition von diesen benötigt werden:

```
data dMsgEnv = MsgEnv(dAddr, dAddr, dInstID, dFktOPTypeEnv);
data dMsgNM  = MsgNM(dAddr, dAddr, dInstID, dFktOPTypeNM);

data dAddr   = phy0x0400 | phy0x0401 | phy0x0402 | phy0x0403 |
               log0x0100 | log0x0101 | log0x0102 | log0x0103 |
               bc0x03C8;
data dInstID = inst0x00 | inst0x01 | inst0x02 | inst0x03;

data dFktOPTypeEnv = FBlockIDsStatus(dFBlockIDList);
data dFktOPTypeNM  = FBlockIDsGet | ConfigurationStatus(dStatus);

data dStatus       = Ok;
data dFBlockIDList = noFB | FBList(dFBlockID,dInstID,dFBlockIDList);
data dFBlockID     = FBlock1 | FBlock2 | FBlock3;
```

Bei der Definition dieser Datentypen kommt das Prinzip der Datenabstraktion (siehe Abschnitt 3.2.2) wiederholt zu Einsatz:

- Der Inhalt der Konstruktoren `MsgEnv` und `MsgNM` der Datentypen `dMsgEnv` bzw. `dMsgNM` vereinfachen und abstrahieren vom Aufbau einer realen MOST-Nachricht (siehe Abschnitt 7.1), indem der Nachrichtenbestandteil `FBlockID` weggelassen

wurde und die Bestandteile `FktID` und `OPType` zu einem zusammengefasst wurden. Dies ist sinnvoll, da für das NetworkMastermodell nur wenige MOST-Nachrichten benötigt werden und sich diese anhand der Funktionsoptypekombination eindeutig unterscheiden lassen.

- Im MOST-Netzwerk steht der Adressraum *0x0000* bis *0xFFFF* zur Verfügung, welcher in verschiedene Teilbereiche gegliedert ist. Der Adressbereich für logische Adressen *0x0100-0x013F* und physikalische Adressen *0x0400-0x43F* wird im Modell abstrahiert und im Datentyp `dAddr` auf die Symbole `log0x0100-log0x0103` bzw. `phy0x0100-phy0x0103` reduziert. Andere Adressbereiche wie z.B. Gruppenadressen werden dagegen im Modell nicht modelliert, da sie für den NetworkMaster nicht relevant sind. Die Broadcastadresse *0x03C8*, die vom NetworkMaster zum Verteilen von Konfigurationsnachrichten dient, wird im Modell durch das Symbol `bc0x03C8` dargestellt.
- Der InstID-Bereich *0x00-0xFF* wird analog zu den logischen und physikalischen Adressen im Modell reduziert und zu den Symbolen `inst0x00-inst0x03` abstrahiert.
- Einerseits empfängt der NetworkMaster auf Modellebene lediglich MOST-Nachrichten mit der Funktionsoptypekombination *FBlockIDs.Status*, mit der Geräte ihre verfügbaren Funktionsblöcke beim NetworkMaster melden. Dieser Funktionsoptype wird im Datentyp `dFktOPTypeEnv` modelliert. Andererseits sendet der NetworkMaster MOST-Nachrichten mit den Funktionsoptypekombinationen *FBlockIDs.Get* und *Configuration.Status* zum Abfragen der Funktionsblöcke eines Geräts bzw. zum Broadcasten des MOST-Netzwerkstatus. Diese werden mit dem Datentyp `dFktOPTypeNM` modelliert. Der Datentyp `dStatus` modelliert die verschiedenen Netzwerkstati, die durch den NetworkMaster via Broadcast gesendet werden. Zu Beginn der Entwicklung benötigen wir nur den Status *Ok*.
- Die Geräte melden ihre verfügbaren Funktionsblöcke mittels einer Funktionsblockliste in einer Statusnachricht an den NetworkMaster. Diese Liste wird mittels dem rekursiven Datentyp `dFBlockIDList` modelliert, indem die Konstante `noFB` die leere Liste und der Konstruktor `FBList` die Liste bestehend aus *FBlockID*, *InstID* und der Restliste modelliert. Wir abstrahieren von der Vielzahl der Funktionsblöcke in einem MOST-Netzwerk, indem wir uns im Modell auf drei symbolisch unterscheidbare `FBlock1-3` beschränken.

Weiterhin benötigen wir den Eingabekanal `net` mit Datentyp `dNet`, der das An- und Abschalten des Netzwerks modelliert:

```
data dNet  =  NetOn | NetOff;
```

Insgesamt besitzt das NetworkMastermodell zu Beginn der Entwicklung folgende Schnittstelle:

```
inputchannel  dNet    net;
              dMsgEnv fromEnv;
```

```
outputchannel dMsgNM  fromNM;
```

7.3.3. Erstes Inkrement: Startverhalten

In diesem Abschnitt modellieren wir das erste Inkrement gemäß der Planung aus Abschnitt 7.3.1. Bei der Modellierung des ersten Inkrements werden wir folgende Operationen auf Regelsystemen verwenden:

- Erweiterung des lokalen Datenraums und
- Hinzufügen von Verhaltensregeln.

Folgende informelle Anforderung an das Verhalten des NetworkMasters ist Gegenstand des ersten Inkrements des Testmodells:

Anforderung "Systemstart" Das *NetOn*-Ereignis löst den Systemstart des MOST-Netzwerks aus. Die Netzwerkhardware jedes Gerät mit geöffneten Bypass berechnet seine physikalische Adresse in Abhängigkeit seiner Position im Netzwerk. Beim Systemstart ermittelt der NetworkMaster die Systemkonfiguration, indem er von jedem Gerät im Netzwerk die Funktionsblöcke abfragt. Die Abfrage erfolgt mittels einer physikalisch adressierten *FBlockIDs.Get*-Nachricht an den *NetBlock* des jeweiligen Geräts[4], z.B. wird das zweite Gerät mit der Nachricht

```
0x0100.0x0402.NetBlock.0x02.FBlockIDs.Get
```

abgefragt. Das befragte Gerät antwortet mit einer *FBlockIDs.Status*-Nachricht und meldet seine verfügbaren Funktionsblöcke in einer *FBlockIDList*, z.B. antwortet das zweite Gerät auf obige Anfrage mit

```
0x0102.0x0100.NetBlock.0x02.FBlockIDs.Status(AudioDiskPlayer.0x01).
```

Der NetworkMaster speichert alle Funktionsblöcke und die zugehörigen physikalischen Positionen und die logischen Adressen[5] in der zentralen Registrierung ab. Nach erfolgreichen Abfragen aller Geräte sendet der NetworkMaster via Broadcast *Configuration.Status(Ok)* und gibt damit die Erlaubnis an die Geräte, im Netzwerk frei zu kommunizieren. Das Abfragen der Systemkonfiguration wird auch mit *System Configuration Check (SCC)* bezeichnet. Ein Beispielablauf eines Systemstarts ist in Form eines informellen Sequenzdiagramms, wie sie häufig in Spezifikationsdokumenten verwendet werden, in Abbildung 7.1 dargestellt.

Vorbereitungen Um diese Anforderung im Modell durch Verhaltensregeln modellieren zu können, müssen wir zunächst geeignete lokale Variablen und zugehörige Datentypen

[4] Die physikalische Adresse eines Geräts ist eindeutig durch seine Position im MOST-Ring festgelegt.
[5] Die logische Adresse eines Geräts wird vom Gerät selbst bestimmt und bleibt solange gültig bis ein Netzwerkreset ausgelöst wird.

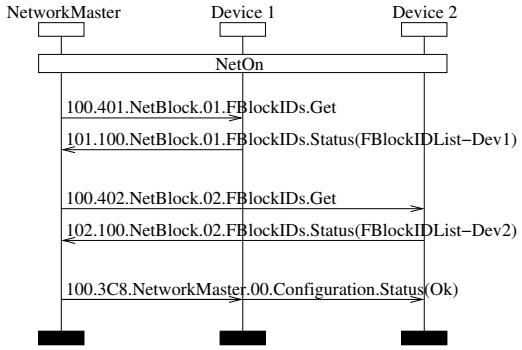


Abbildung 7.1.: Systemstart

definieren. Wir definieren die lokalen Variablen `mode`, `dL`, `reg` und `wA` mit dem Datentyp `dMode`, `dDeviceList` bzw. `dRegistry`:

```
data dMode     = off | init | cfgOk;
data dDevList  = noDL | DevList(dInstID,dRequestStatus,dDevList);
data dRequestStatus = notRequested | requested | answered;
data dRegistry = noR | Registry(dInstID,dAddr,dFBlockIDList,dRegistry);
```

Die Variablen und ihre Datentypen spannen den Datenraum des Testmodells auf und modellieren Folgendes:

- Die Werte der Variablen `mode` modellieren die möglichen Modi des NetworkMasters: (1) der NetworkMaster ist abgeschaltet, (2) führt einen SCC nach NetOn durch oder (3) hat *Configuration Status Ok* erteilt.
- In der Variablen `dL` speichert der NetworkMaster, welche Geräte noch nicht abgefragt, bereits abgefragt wurden bzw. bereits geantwortet haben. Ihr rekursiver Datentyp `dDevList` speichert deshalb zu der physikalischen Position eines Geräts im MOST-Netzwerk[6] den Abfragestatus `notRequested`, `requested` bzw. `requested` des Geräts[7] ab.
- Die Registrierung des NetworkMasters wird ebenfalls durch einen rekursiven Datentyp modelliert, welche pro Eintrag die physikalische Position, die logische Adresse sowie die Funktionsblöcke eines Geräts enthält[8].
- In der lokalen Variablen `wA` wird die physikalische Position von dem Gerät gespeichert, von dem der NetworkMaster eine *FBlockIDs.Status*-Nachricht erwartet.

Insgesamt definieren wir vier lokale Variablen und initialisieren sie wie folgt:

```
localvariable dMode mode    = off;
              dDevList dL   = noDL;
```

[6] erster Parameter des Konstruktors `DevList`
[7] zweiter Parameter des Konstruktors `DevList`
[8] erster, zweiter bzw. dritter Parameter des Konstruktors `Registry`

```
dRegistry reg = noR;
dInstID wA    = inst0x00;
```

Verhaltenserweiterung Jetzt sind alle Vorbereitungen getroffen, um obige Anforderung an das Verhalten des NetworkMasters modellieren zu können. Tabelle 7.1 zeigt die Modellierung als Regelsystem. Das Regelsystem besteht aus fünf Verhaltensregeln, die nacheinander dem Regelsystem hinzugefügt werden und damit schrittweise das Verhalten des Modells erweitern (siehe auch Abschnitt 5.5.3). Im Folgenden erläutern wir deren Aufbau und Funktionsweise. Die dabei benötigten und verwendeten Prädikate und Funktionen beschreiben lediglich informell. Die präzise Implementierung dieser Funktionen kann im Anhang B.1.2 nachgeschlagen werden.

NetOn Die Regel mit Namen *NetOn* modelliert das Verhalten des NetworkMasters, wenn das MOST-Netzwerk mittels des NetOn-Ereignisses eingeschaltet wird. Auf Modellebene fordern wir dazu das Eingabemuster, dass auf Eingabekanal `net` die Nachricht `NetOn` und auf Kanal `fromEnv` die leere Nachricht anliegt. Ferner kann die NetOn-Nachricht nur vorarbeitet werden, wenn das NetworkMastermodell im mode `off` (Prädikat `is_off(mode)` in der Vorbedingung) befindet. Dadurch treffen wir auf Modellebene die Annahme, dass ein NetOn-Ereignis nur dann auftreten darf, wenn der NetworkMaster abschaltet ist und er gleichzeitig keine MOST-Nachricht empfängt. Wir schränken dadurch die Menge der modellierten Abläufe bewusst ein. Diese Annahmen sind gerechtfertigt, da es physikalisch nicht möglich ist, ein NetOn-Ereignis bei eingeschaltetem MOST-Netzwerk zu erzeugen, bzw. es bei abgeschalteten Netzwerk nicht möglich ist, Nachrichten zu versenden. Wird die NetOn-Nachricht auf Modellebene empfangen, dann wird keine Ausgabe gesendet und es werden die lokalen Variablen initialisiert: der Modus `mode` wird auf `init` gesetzt, die Variable `dL` wird mittels Funktion `buildDevList` initialisiert, die in Abhängigkeit der Geräteanzahl in der Umgebung des NetworkMaster die Geräteliste mit zugehörigen Abfragestatus festlegt, die Registrierung `reg` wird auf leer gesetzt und die Variable `wA` wird auf `inst0x00` gesetzt, das bedeutet, dass der NetworkMaster keine *FBlockIDs.Status*-Nachricht von einem Gerät erwartet.

NetOff Die Regel *Netoff* modelliert das Abschalten des MOST-Netzwerk. Die zugehörige Nachricht `NetOff` auf Kanal `net` wird vom Modell nur unter der Vorbedingung akzeptiert, wenn der NetworkMaster nicht abgeschaltet ist (Prädikat `not(is_off(mode))`). Im Zuweisungsteil der Regel wird die Modusvariable entsprechend auf `off` gesetzt und die Variablen `dL`, `reg` und `wA` auf ihre Grundwerte `noDL`, `noR` bzw. `inst0x00` zurückgesetzt.

sndFBGet Die Regeln *sndFBGet*, *recFBStatus* und *recFBStatusSndOk* modellieren den SCC, den der NetworkMaster bei Systemstart durchführt. Die Regel *sndFBGet* regelt das Senden einer *FBlockIDs.Get*-Nachricht. Die Nachricht wird unter der Vorbedingung gesendet, wenn sich das Modell im Modus `init` befindet, es ein Gerät gibt, das noch nicht abgefragt wurde (Prädikat `isNotReq(dL)`) und der Network-

Name	Pre	Input	Output	Assign
NetOn	is_off(mode)	net?NetOn; fromEnv?		mode=init; dL=buildDevList(2); reg=noR; wA=inst0x00
NetOff	not(is_off(mode))	net?NetOff; fromEnv?		mode=off; dL=noDL; reg=noR; wA=inst0x00
sndFBGet	is_init(mode) && isNot-Req(dL) && wA == inst0x00	net?; fromEnv?	fromNM!MsgNM(log0x0100, inst2phy(getNotReq(dL)), getNotReq(dL), FBlockIDsGet)	dL=setNextReq(dL); wA=getNotReq(dL)
recFBStatus	is_init(mode) && isNot-Req(dL) && wA == INST	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL))		dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
recFBStatusSndOk	is_init(mode) && not(isNotReq(dL)) && wA == INST	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL))	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(Ok))	mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00

Tabelle 7.1.: Erstes Inkrement

Master auf keine *FBlockIDs.Status*-Nachricht wartet (Bedingung `wA==inst0x00`). Ferner wird gefordert, dass keine Nachricht an den Eingabekanälen anliegt (Eingabemuster mit leeren Nachrichten), die der NetworkMaster ggf. verarbeiten müsste. Wenn diese Bedingungen erfüllt sind, dann wird im Modell eine *FBlockIDs.Get*-Nachricht gesendet, indem der Ausgabekanal `fromNM` mit einer Nachricht belegt wird, die die logische Adresse des NetworkMaster `log0x0100`, die berechnete physikalische Adresse des Empfängergeräts `inst2phy(getNotReq(dL))`, die berechnete Position des Empfängergeräts `getNotReq(dL)` und den Funktionsoptype `FBlockIDsGet` enthält. In der Zuweisung wird das abgefragte Gerät markiert (`dL=setNextReq(dL)`) und die physikalische Position des selben Geräts in der Variablen `wA` gespeichert (`wA=getNotReq(dL)`) und damit das NetworkMastermodell in einen Zustand versetzt, auf die *FBlockIDs.Status*-Antwort des betreffenden Geräts zu warten.

recFBStatus Die Regel *FBStatus* regelt den Empfang einer *FBlockIDs.Status*-Nachricht von einem angefragten Gerät während eines SCCs. Dementsprechend fordert die Vorbedingung, dass sich der NetworkMaster im Modus `init` befindet, es noch weitere Geräte gibt, die abgefragt werden müssen (Prädikat `isNotReq(dL)`) und die physikalische Position des Absendergeräts mit der übereinstimmt, von welcher der NetworkMaster eine Statusnachricht erwartet (Bedingung `wA==INST`). Zusätzlich fordert das Eingabemuster eine Nachricht auf Kanal `fromEnv` mit Zieladresse `log0x0100` und Funktionsoptype `FBlockIDsStatus`. Dabei werden die logische Absenderadresse, die Absenderposition und die übermittelten Funktionsblöcke an die regellokalen Variablen `LOG`, `INST` und `FBL` gebunden, die in der Vorbedingung und Zuweisung verwendet werden. In der Zuweisung wird das antwortende Gerät in der Geräteliste markiert (`sL=setAns(INST, dL)`), die Daten des Geräts in die Registrierung eingetragen `reg=Registry(INST, LOG, FBL, reg)` und das Modell in den Zustand versetzt auf keine Statusnachricht zu warten (`wA=inst0x00`).

recFBStatusSndOk Die Regel *recFBStatusSndOk* ist ähnlich zu Regel *recFBStatus* und sendet die Nachricht *Configuration.Status(Ok)*, wenn das letzte Gerät geantwortet hat. Dementsprechend unterscheidet sich die Regel von Regel *recFBStatus* nur in der Vorbedingung und Ausgabe. Die Vorbedingung prüft, ob keine weiteren Geräte abzufragen sind (Bedingung `not(isNotReq(dL))`), und in der Ausgabe wird der Kanal `fromNM` mit einer Nachricht belegt, dessen Zieladresse die Broadcastadresse `bc0x03C8` und der Funktionsoptype `ConfigurationStatus(Ok)` ist.

Das modellierte Regelsystem ist deterministisch, da sich die Vorbedingung offensichtlich gegenseitig ausschließen (vgl. Tabelle 7.1).

Validation&Verifikation Nach der Spezifikation der Verhaltensregeln müssen wir nachweisen, dass das geforderte Verhalten aus obiger Anforderung “Systemstart” korrekt im Modell realisiert wurde. Wir erhalten durch Simulation des Modells oder durch Testfallgenerierung aus dem Modell den Ablauf aus Abbildung 7.2. Der Vergleich des Ablaufs

aus der Anforderungsspezifikation (Abb. 7.1) mit dem Modellablauf (Abb. 7.2) zeigt, dass das geforderte Verhalten korrekt im Modell implementiert wurde.

input channels		output channels
net	fromEnv	fromNM
NetOn		
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
NetOff		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))

Abbildung 7.2.: Beispielablauf des ersten Inkrements

Ein Regressionstest wird nach der Modellierung des ersten Inkrements nicht durchgeführt, da es kein Vorgängerinkrement gibt.

Regressionstestsuite In den folgenden Inkrementen soll nachgewiesen werden, dass sich das NetworkMastermodell beim Systemstart korrekt verhält, wenn alle Geräte im Netzwerk auf Anfragen des NetworkMaster mit Statusmeldungen antworten. Dazu definieren wir die Regressionstestsuite P_1, die neben dem Ablauf aus Abbildung 7.2 weitere enthält (siehe Anhang B.2.1).

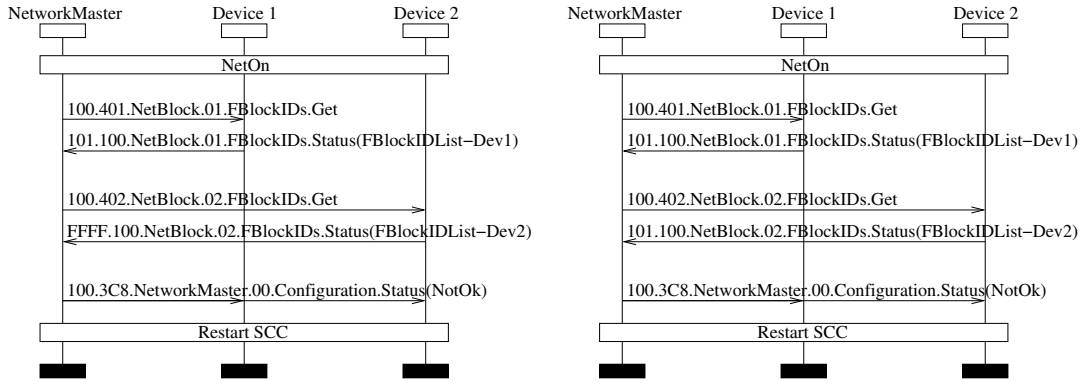
7.3.4. Zweites Inkrement: Systemstatus NotOk

In diesem Abschnitt modellieren wir das zweite Inkrement auf Basis des ersten gemäß der Planung aus Abschnitt 7.3.1. Dabei kommen folgende Operationen auf Regelsystemen bzw. Abstraktionsformen zum Einsatz:

- Erweiterung der Eingabe- und Ausgabeschnittstelle unter Einsatz von Datenabstraktion,
- Einschränken von Verhaltensregeln und
- Hinzufügen von Verhaltensregeln.

Das Modell wird durch die Modellierung der folgenden informellen Anforderung erweitert:

Anforderung "Systemstatus NotOk" Antwortet ein Gerät mit einer ungültigen Statusnachricht auf eine Anfrage des NetworkMaster, dann löst der NetworkMaster durch Senden der Broadcastnachricht *Configuration.Status(NotOk)* einen Netzwerkreset aus, löscht die Registrierung und führt einen SCC durch. Nach erfolgreichen Durchführen des SCCs sendet der NetworkMaster die Broadcastnachricht *Configuration.Status(Ok)*. Eine Statusmeldung is ungültig, wenn die Absenderadresse nicht im logischen Adressraum *0x0100-0x013F* liegt oder ein Adresskonflikt vorliegt, d.h. die eineindeutige Zuordnung zwischen physikalischer Geräteposition und dessen logischen Adresse im MOST-Netzwerk verletzt wird. Die Abbildung 7.3 zeigt zwei Beispielabläufe, die zum Reset des Netzwerks führen.



(a) Ungültige logische Adresse (b) Adresskonflikt zwischen zwei Geräten

Abbildung 7.3.: Systemstatus NotOk

Vorbereitungen Um diese Anforderung mit weiteren Verhaltensregeln in das NetworkMastermodell integrieren zu können, müssen wir dieses zunächst vorbereiten, indem wir die Schnittstelle des Modells erweitern. Damit im Modell der NetworkMaster die Broadcastnachricht *Configuration.Status(NotOk)* senden kann, wird der Datentyp `dStatus` um den Wert `NotOk` erweitert. Dieser Datentyp lautet nun:

```
data dStatus = Ok | NotOk;
```

Diese Erweiterung der Schnittstelle hat keine Auswirkungen auf das Modellverhalten (vgl. Abschnitt 5.5.1). Ferner erweitern wir die Schnittstelle, damit das Empfangen einer Statusmeldung mit ungültiger Absenderadresse modelliert werden kann. Dazu erweitern wir den Datentyp `dAddr` um den Wert `logInvalid` und wenden dabei das Prinzip der Datenabstraktion an (vgl. Abschnitt 3.2.2)

```
data dAddr   = phy0x0400 | phy0x0401 | phy0x0402 | phy0x0403 |
               log0x0100 | log0x0101 | log0x0102 | log0x0103 |
               logInvalid | bc0x03C8;
```

Dabei werden alle Adressen, die außerhalb des logischen Adressbereichs liegen, im Modell zusammengefasst und durch den Wert `logInvalid` repräsentiert. Durch diese Operationen wurde sowohl die Ein- als Ausgabeschnittstelle erweitert. Die Erweiterung der Eingabeschnittstelle führt zu einer impliziten Erweiterung des Modellverhaltens (vgl. Abschnitt 5.5.1. In unserem Fall ist diese Erweiterung unerwünscht und wird im folgenden Abschnitt korrigiert.

Korrektur Das Modellverhalten muss in zweierlei Hinsicht korrigiert werden: (1) Die Verhaltensregeln des ersten Inkrements modellieren unter anderem Abläufe, in denen einen Adresskonflikt zwischen Geräten zugelassen ist. (2) Durch die Schnittstellenerweiterung wurde das Modellverhalten implizit erweitert und ermöglicht nun, dass sich Geräte mit ungültigen Adressen bei dem NetworkMaster anmelden. Beide Verhalten stehen im Widerspruch zu Anforderung "Systemstatus NotOk" und müssen deshalb aus dem Modellverhalten entfernt werden. Dieses Erreichen wir, indem die Verhaltensregel `recStatus` durch Hinzufügen des Prädikats `isAnsOk` einschränken (siehe Tabelle 7.2 und vgl. Abschnitt 5.5.2). Das Prädikat `isAnsOk` stützt sich auf die Prädikate `isAddrValid` und `isAddrDuplicate`

```
fun isAnsOk(INST,LOG,REG) =
        isAddrValid(LOG) && not(isAddrDuplicate(INST,LOG,REG));
```

und prüft, ob einerseits die Absenderadresse der Statusnachricht eine logische Adresse ist und ob andererseits die physikalische Position und die Absenderadresse keinen Adresskonflikt mit den Registrierungseinträgen verursacht.

Verhaltenserweiterung Nun wurden alle Vorbereitungen getroffen, um das Auslösen von *Configuration.Status(NotOk)* gemäß Anforderung "Systemstatus NotOk" modellieren zu können. Wir fügen dem Verhaltensmodell folgende Regel hinzu:

sndNotOk Das Eingabemuster zum Empfang einer Statusnachricht ist identisch zu dem Eingabemuster der Regel *recFBStatus*. Wir legen die Vorbedingung der Regel fest, indem wir fordern, dass sich das NetworkMastermodell im Modus `init` befindet, dass die physikalische Position der Nachricht mit der erwarteten übereinstimmt (`wa=INST`) und dass die Nachrichtenparameter physikalische Position, Absenderadresse unzulässig sind (`not(isAnsOk(INST, LOG, reg))`). Systemstatus NotOk entspricht einem Netzwerkreset, deshalb werden in der Zuweisung der Regel die lokalen Variablen `mode`, `dL`, `reg`, `wA` zurück gesetzt auf `init`, `setNotReq(dL)`, `noR` bzw. `inst0x00`. Dadurch wird das Modell in den Zustand versetzt, der einem SCC entspricht, wie er nach einem Systemstart durchgeführt wird.

Mit Hinzufügen der Regel *sndNotOk* ist die Verhaltenserweiterung im zweiten Inkrement abgeschlossen und das Regelsystem ist offensichtlich deterministisch. Tabelle 7.2 zeigt das Ergebnis, wobei die Änderungen in den Verhaltensregeln gegenüber dem ersten Inkrement fett gedruckt sind.

Name	Pre	Input	Output	Assign
NetOn	is_off(mode)	net?NetOn; fromEnv?		mode=init; dL=buildDevList(2); reg=noR; wA=inst0x00
NetOff	not(is_off(mode))	net?NetOff; fromEnv?		mode=off; dL=noDL; reg=noR; wA=inst0x00
sndFBGet	is_init(mode) && isNot-Req(dL) && wA == inst0x00	net?; fromEnv?	fromNM!MsgNM(log0x0100, inst2phy(getNotReq(dL)), getNotReq(dL), FBlockIDsGet)	dL=setNextReq(dL); wA=getNotReq(dL)
recFBStatus	is_init(mode) && isNot-Req(dL) && wA == INST **&& isAnsOk(INST, LOG, reg)**	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL))		dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
recFBStatusSndOk	is_init(mode) && not(isNotReq(dL)) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL))	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(Ok))	mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
sndNotOk	**is_init(mode) && wA == INST && not(isAnsOk(INST, LOG, reg))**	**net?; fro-mEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDs-Status(FBL))**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(NotOk))**	**mode=init; dL=setNotReq(dL); reg=noR; wA=inst0x00**

Tabelle 7.2.: Zweites Inkrement

Validation&Verifikation Nach Abschluss der Verhaltenserweiterung weisen wir nach, dass die Anforderung "Systemstatus NotOk" im NetworkMastermodell realisiert wurde. Wir erhalten durch Simulation oder Testfallerzeugung aus dem Modell die Abläufe in den Abbildungen B.3 und B.4 (Anhang B.2.2) und zeigen durch Vergleich, dass das gewünschte Verhalten ein Teil des Modellverhaltens ist.

Da bei der Entwicklung dieses Inkrements eine Reduktion des Modellverhaltens durch Einschränken einer Verhaltensregel stattfand, weisen wir mit Hilfe der Regressionstestsuite P_1 nach, dass das entscheidende Verhalten aus dem erstem Inkrement erhalten wurde. Wir testen also die Regressionstestsuite P_1 gegen das zweite Modellinkrement, indem wir die Eingaben der Abläufe aus P_1 in das Modell injizieren und die Modellausgaben mit den Ausgaben des betreffenden Ablaufs vergleichen[9].

Regressionstestsuite Wir erweitern die Regressionstestsuite P_1 um die Abläufe aus den Abbildungen B.3 und B.4 zu Testsuite P_2 und um weitere (siehe Anhang B.2.2). Mit dieser Testsuite kann nachgewiesen werden, dass die nachfolgenden Inkremente ungültige Statusnachrichten während des Systemstarts korrekt erkennen.

7.3.5. Drittes Inkrement: Nicht antwortende Geräte

In diesem Abschnitt modellieren wir das dritte Inkrement gemäß der Planung aus Abschnitt 7.3.1. Hierfür werden wir folgende Operationen auf Regelsystemen verwenden:

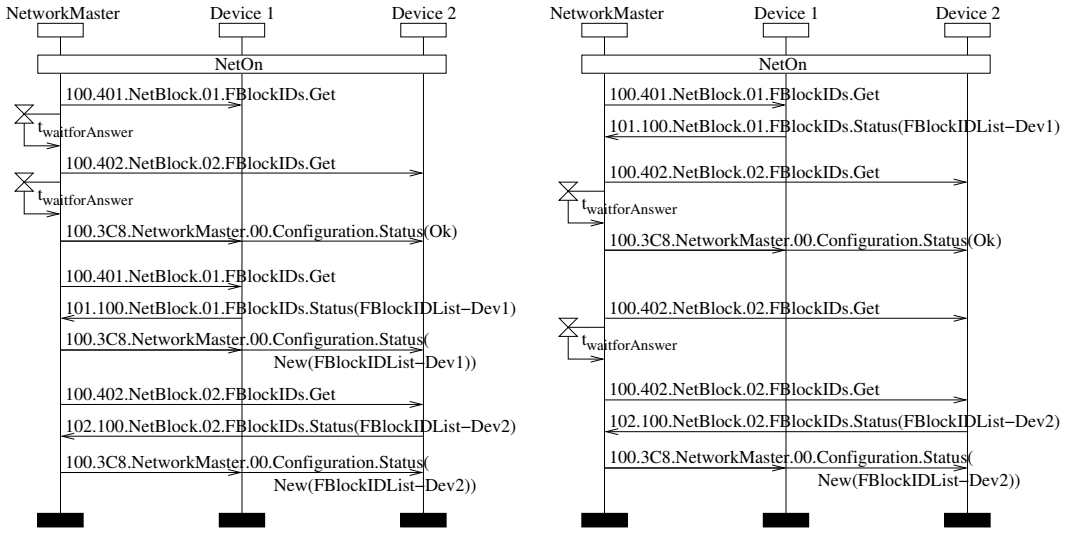
- Erweiterung der Ein- und Ausgabeschnittstelle,
- Erweiterung des lokalen Datenraums,
- Einschränken von Verhaltensregeln,
- Modifikation der Ausgabe einer Regel,
- Hinzufügen von Verhaltensregeln und
- Verallgemeinern einer Verhaltensregel.

Im Modell wird folgende informelle Anforderung integriert:

Anforderung "Nicht antwortende Geräte" Der NetworkMaster muss nach Absenden einer *FBlockIDs.Get*-Anfrage die Zeitdauer $t_{WaitForAnswer}$ auf eine Antwort des angefragten Gerätes warten. Antwortet ein angefragtes Gerät nicht, dann wird das nächste Gerät angefragt bzw. der betreffende SCC beendet. Befindet sich das Netzwerk im Systemstatus Ok, dann werden die Geräte, die noch nicht geantwortet haben, zyklisch solange angefragt, bis diese geantwortet haben. Antwortet nun ein Gerät mit einer Statusmeldung, dann sendet der NetworkMaster die neu gemeldeten Funktionsblöcke *FBlockIDList*

[9] Das automatische Testen von Abläufen gegen ein Modell wird vom Werkzeug AutoFocus unterstützt.

mit einer Broadcastnachricht *Configuration.Status(New(FBlockIDList))* an das MOST-Netzwerk. Die informellen Sequenzdiagramme in Abbildung 7.4 zeigen zwei Beispielabläufe.



(a) Beide Geräte antworten nicht

(b) Ein Gerät antwortet wiederholt nicht

Abbildung 7.4.: Nicht antwortende Geräte

Vorbereitungen Zur Integration dieser Anforderung in das NetworkMastermodell wenden wir das Prinzip der temporalen Abstraktion an (vgl. Abschnitt 3.2.4). Wir modellieren das Warten der Zeitdauer $t_{WaitForAnswer}$, indem wir im Modell zwei abstrakte Nachrichten einführen, eine zum Starten eines Timers und eine, die den Ablauf des Timers symbolisiert. Dazu erweitern wir die Schnittstelle des Modells um den Ausgabekanal `t` vom Typ `dTimer` und um den Eingabekanal `to` vom Typ `dTimeout`.

```
data dTimer   = tAnswer;
data dTimeout = timeoutAnswer;
```

Nach Abschnitt 5.5.1 verändert das Hinzufügen eines Ausgabekanals das Modellverhalten im Wesentlichen nicht. Das Hinzufügen eines Eingabekanals jedoch erweitert jedoch implizit das Modellverhalten. Die erforderlichen Korrekturen werden im nächsten Absatz diskutiert. Ferner erweitern wir die Ausgabeschnittstelle, um das Senden einer *Configuration.Status(New(FBlockIDList))* zu ermöglichen, indem wir den Datentyp `dStatus` wie folgt erweitern:

```
data dStatus = Ok | NotOk | New(dFBlockIDList);
```

Um das zyklische Abfragen von nicht antwortenden Geräten zu modellieren, führen wir einen weiteren Modus ein, indem wir den Datentyp `dMode` um den Wert `delayed` erwei-

tern. Der Datentyp `dMode` ist nun wie folgt definiert:

```
data dMode      = off | init | cfgOk | delayed;
```

Korrektur Da die Schnittstelle des Modells erweitert wurde, sind folgende Korrekturen an den Verhaltensregeln notwendig, bevor wir weitere Regeln dem Modell hinzufügen. Gemäß der Anforderung "Nicht antwortende Geräte" starten wir symbolisch auf Modellebene einen Timer, wenn der NetworkMaster ein Gerät mit einer *FBlockIDs.Get*-Nachricht abfragt. Dazu erweitern wir die Ausgabe der Regel `sndFBGet` um den Ausdruck `t!tAnswer` (vgl. Tabelle 7.3). Durch die Erweiterung der Eingabeschnittstelle wird das Modellverhalten implizit erweitert. Diesen Effekt machen wir rückgängig, indem wir das Eingabemuster jeder Verhaltensregel um den Ausdruck `to?` erweitern und damit den Schaltbereich der Regeln einschränken. Dadurch wird verhindert, dass sinnlose Verhaltensabläufe entstehen, bei denen zu ungeeigneten Zeitpunkten eine Timeout-Nachricht auftritt.

Verhaltenserweiterung Im Modell modellieren wir ein nicht antwortendes Gerät, indem die Umgebung keine Statusnachricht schickt, sondern die Nachricht `timeoutAnswer` auf dem Eingabekanal `to` empfängt. Dazu fügen wir dem Modell die folgenden zwei Regeln hinzu:

recTOAnswer, recTOAnswerSndOk Diese Regeln sind ähnlich wie die bereits vorhandenen Regeln *recFBStatus* und *recFBStatusSndOk* aufgebaut. Die Regeln *recTOAnswer* und *recTOAnswerSndOk* unterscheiden sich von der Regeln *recFBStatus* bzw. *recFBStatusSndOk*, indem in der Vorbedingung das Prädikat `isAnsOk` entfällt, im Eingabemuster statt einer Statusnachricht der Timeout `to?timeoutAnswer` empfangen wird und in der Zuweisung das Abspeichern der Daten einer Statusnachricht in den Variablen `dL` und `reg` entfällt.

Nun wenden wir uns der Anforderung, nicht antwortende Geräte zyklisch abzufragen, zu. Dazu verallgemeinern wir bzw. fügen wir die folgenden Regeln hinzu:

swDelay Diese Regel ist schaltbereit, wenn sich das Modell in Modus `cfgOk` befindet und noch nicht alle Geräte geantwortet haben (Prädikat `not(haveAllAns(dL))`. Ist das der Fall, dann wird mittels Zuweisung in den Modus `delayed` gewechselt und der Gerätestatus aller nicht antwortenden Knoten, die in der lokalen Variablen `dL` mit `requested` gekennzeichnet sind, auf `notRequested` gesetzt (`dL=setReq2-NotReq(dL)`).

sndFBGet Um Geräte im Modus `delayed` abzufragen verallgemeinern wir die Vorbedingung der Regel *sndFBGet* durch Disjunktion mit dem Prädikat `is_delayed(mode)` und fügen die Regeln *recNewFBStatus* und *recNewFBStatusCfgOk* hinzu.

recNewFBStatus, recNewFBStatusCfgOk Diese Regeln unterscheiden sich von den bereits vorhandenen Regeln *recFBStatus* bzw. *recFBStatusSndOk* lediglich dadurch,

dass in der Vorbedingung `is_init(mode)` durch `is_delayed(mode)` ersetzt wurde und dass in der Ausgabe die empfangenen Funktionsblöcke `FBL` in einer Broadcastnachricht mit Funktionsoptype `ConfigurationStatus(New(FBL))` über Kanal `fromNM` gesendet werden.

recTOAnswer, recTOAnswerCfgOk Das Verhalten bei nicht antwortenden Geräten im Modus `delayed` modellieren wir, indem wir die Vorbedingung der Regel *recTOAnswer* durch Disjunktion mit dem Prädikat `is_delayed(mode)` verallgemeinern und die Regel *recTOAnswerCfgOk* hinzufügen. Diese Regel unterscheidet sich von Regel *recTOAnswerSndOk* lediglich dadurch, dass das Senden der Broadcastnachricht *Configuration.Status(Ok)* auf Kanal `fromNM` entfällt.

sndNotOk Abschließend werden die Anforderungen vervollständigt, dass ein Gerät auch im Modus `delayed` den Systemstatus NotOk auslösen kann, indem wir die Vorbedingung der Regel *sndNotOk* durch Disjunktion mit dem Prädikat `is_delayed(mode)` verallgemeinern.

Mit diesen umfangreichen Erweiterungen haben wir schließlich die Anforderung "Nicht antwortende Geräte" in das NetworkMastermodell integriert. Es ist wiederum leicht zu sehen, dass das Regelsystem deterministisch ist, weil sich die neu hinzugefügten Regeln einerseits untereinander gegenseitig ausschließen und andererseits mit den bereits vorhandenen sich nicht überschneiden können, da die vorhandenen Regeln eines der Prädikate `is_off`, `is_init` oder `is_cfgOk` und die hinzugefügten Regeln das Prädikat `is_delayed` in der Vorbedingung haben und diese Prädikate nicht gleichzeitig gelten können. Das resultierende Regelsystem ist in Tabelle 7.3 dargestellt, wobei alle Erweiterungen gegenüber den zweiten Inkrement fett gedruckt sind.

Name	Pre	Input	Output	Assign
NetOn	is_off(mode)	net?NetOn; fromEnv?; **to?**		mode=init; dL=buildDevList(2); reg=noR; wA=inst0x00
NetOff	not(is_off(mode))	net?NetOff; fromEnv?; **to?**		mode=off; dL=noDL; reg=noR; wA=inst0x00
sndFBGet	(is_init(mode)|| **is_delayed(mode)**) && isNotReq(dL) && wA == inst0x00	net?; fromEnv?; **to?**	fromNM!MsgNM(log0x0100, inst2phy(getNotReq(dL)), getNotReq(dL), FBlockIDsGet); **t!tAnswer**	dL=setNextReq(dL); wA=getNotReq(dL)
recFBStatus	is_init(mode) && isNot-Req(dL) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); **to?**		dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
recFBStatusSndOk	is_init(mode) && not(isNotReq(dL)) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); **to?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(Ok))	mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
sndNotOk	(is_init(mode) || **is_delayed(mode)**) && wA == INST && not(isAnsOk(INST, LOG, reg))	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); **to?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(NotOk))	mode=init; dL=setNotReq(dL); reg=noR; wA=inst0x00
recTOAnswer	**(is_init(mode) || is_delayed(mode)) && isNotReq(dL) && wA != inst0x00**	**net?; fromEnv?; to?timeoutAnswer**		**wA=inst0x00**

Name	Pre	Input	Output	Assign
recTOAnswer-SndOk	**is_init(mode) && not(isNotReq(dL)) && wA != inst0x00**	**net?; fromEnv?; to?timeoutAnswer**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(Ok))**	**mode=cfgOk; wA=inst0x00**
swDelay	**is_cfgOk(mode) && not(haveAllAns(dL))**	**net?; fromEnv?; to?**		**mode=delayed; dL=setReq2Not-Req(dL)**
recNewFB-Status	**is_delayed(mode) && is-NotReq(dL) && wA == INST && isAnsOk(INST, LOG, reg)**	**net?; from-Env?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); to?**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(New(FBL)))**	**dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00**
recNewFB-StatusCfgOk	**is_delayed(mode) && not(isNotReq(dL)) && wA == INST && isAn-sOk(INST, LOG, reg)**	**net?; from-Env?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); to?**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(New(FBL)))**	**mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00**
recTOAnswer-CfgOk	**is_delayed(mode) && not(isNotReq(dL)) && wA != inst0x00**	**net?; fromEnv?; to?timeoutAnswer**		**mode=cfgOk; wA=inst0x00**

Tabelle 7.3.: Drittes Inkrement

Validation&Verifikation Nach Abschluss der Verhaltenerweiterung erzeugen wir durch Simulation oder durch Testfallgenerierung z.B. die Abläufe aus Abbildung B.11 und B.12 (siehe Anhang B.2.3). Der Vergleich mit den Sequenzdiagrammen aus Abbildung 7.4 zeigt, dass diese ein Teil des Modellverhaltens sind und damit die Anforderung "Nicht antwortende Knoten" korrekt im Modell realisiert wurde.

Da bei der Bildung des dritten Inkrements eine umfangreich Schnittstellenerweiterung stattgefunden hat und die Ausgabe einer Verhaltensregel verändert wurde, muss die Regressionstestsuite P_2 auf die erweiterte Schnittstelle des dritten Inkrements geliftet werden. Dies erreichen wir dadurch, indem wir bei jedem Schritt bei einem Ablauf aus P_2, bei dem eine `FBlockIDsGet`-Nachricht auftritt, die Timernachricht `tAnswer` ergänzen. Zum Beispiel wird ein Schritt, der die Ausgabe

```
fromNM!MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet);
```

enthält, um die Ausgabe

```
t!tAnswer;
```

erweitert. Durch diese Operation bilden wir die Testsuite P_2 auf die Testsuite P_2^{lift} ab (siehe Anhang B.2.3). Diese testen wir erfolgreich gegen das dritte Inkrement des NetworkMastermodells und weisen dadurch nach, dass das wesentliche Verhalten des zweiten Inkrements erhalten wurde.

Regressionstestsuite Wir erweitern die Regressionstestsuite P_2^{lift} um die Abläufe aus Abbildung B.11, B.12 und um weitere Abläufe (siehe Anhang B.2.3), so dass jede der hinzugefügten Regeln mindestens einmal geschaltet hat und erhalten dadurch Regressionstestsuite P_3.

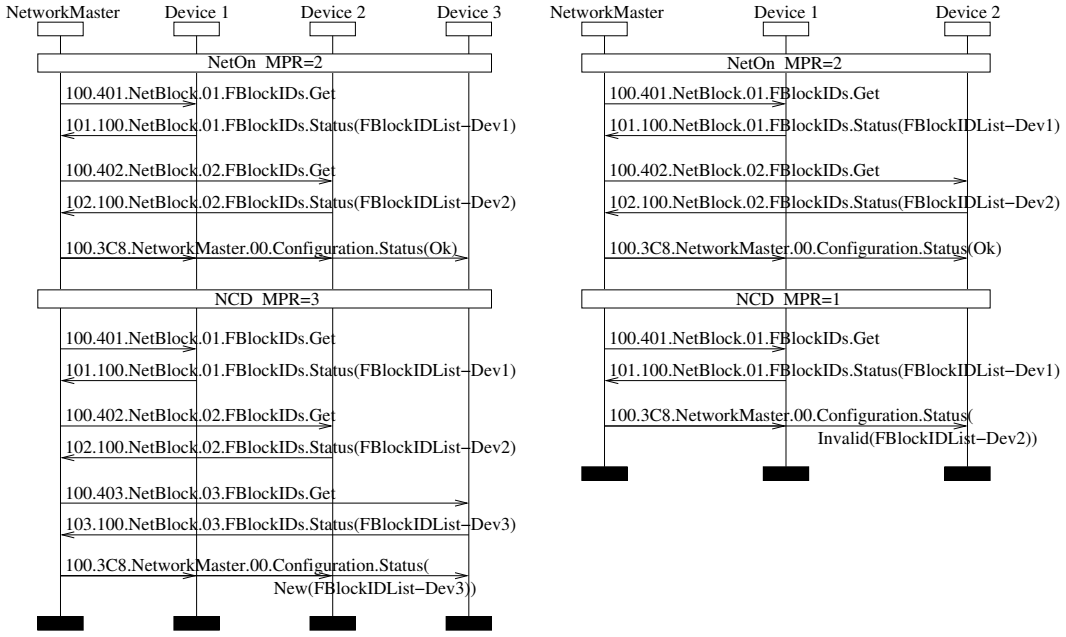
7.3.6. Viertes Inkrement: Network Change Delayed

In diesem Abschnitt modellieren wir das vierte Inkrement gemäß der Planung aus Abschnitt 7.3.1. Dabei kommen folgende Operationen auf Regelsystemen bzw. Abstraktionsprinzipien zum Einsatz:

- Erweiterung der Ein- und Ausgabeschnittstelle unter Verwendung von Kommunikationsabstraktion,
- Erweiterung des lokalen Datenraums,
- Einschränken von Verhaltensregeln,
- Modifikation der Zuweisung einer Regel,
- Hinzufügen von Verhaltensregeln und
- Verallgemeinern einer Verhaltensregel.

Im Modell wird folgende informelle Anforderung integriert:

Anforderung "Network Change Delayed" Verbinden oder trennen sich ein oder mehrere Geräte vom MOST-Netzwerk, dann wird ein *Network Change Delayed (NCD)* Ereignis ausgelöst. Dabei verändert sich das Hardwareregister mit Namen *Maximum Position Register (MPR)* in jedem Gerät, in dem die Anzahl mit dem Netzwerk verbundenen Geräte gespeichert ist. Nach einem NCD wird ein Ereignis in der Netzwerkhardware ausgelöst und der NetworkMaster erhält den aktuellen Wert des MPR. In Abhängigkeit dieses Werts führt der NetworkMaster einen SCC aus und sendet am Ende des SCCs via Broadcast, welche Funktionsblöcke nicht mehr verfügbar bzw. welche neuen Funktionsblöcke im Netzwerk verfügbar sind. Die Abbildung 7.5 zeigt je ein Beispiel, in dem sich nach einem NCD der Wert des MPR erhöht bzw. erniedrigt hat.



(a) MPR+ Ereignis (b) MPR- Ereignis

Abbildung 7.5.: Network Change Delayed

Vorbereitungen Zur Integration dieser Anforderung ist die Erweiterung der Schnittstelle erforderlich. Wir fügen der Eingabeschnittstelle den Kanal `n` vom Typ

```
data dNCD = nMPR(Int);
```

Ein Empfang einer Nachricht `nMPR(newMPR)` auf Kanal `n` modelliert das Auftreten eines NCD-Ereignisses, wobei der Parameter der Nachricht `newMPR` den Wert des MPR

nach dem NCD enthält. Das Erweitern der Eingabeschnittstelle impliziert, wie schon bei früheren Inkrementen, die Korrektur von Verhaltensregeln (siehe nächsten Paragraph "Korrektur"). Ferner ist es erforderlich die Ausgabeschnittstelle des Modells zu erweitern, um das Senden von Veränderungen in der Registrierung via Broadcastnachrichten zu ermöglichen. Dazu erweitern den Datentyp `dStatus` wie folgt:

```
data dStatus       = Ok | NotOk | New(dFBlockIDList) |
                     Invalid(dFBlockIDList) |
                     InvalidNew(dFBlockIDList,dFBlockIDList);
```

Dabei haben wir das Prinzip der Kommunikationsabstraktion (vgl. Abschnitt 3.2.3) angewandt: falls der NetworkMaster sowohl nicht verfügbare Funktionsblöcke *invFBL* durch eine *Configuration.Status(Invalid(invFBL))*-Nachricht abmelden als auch neu verfügbare Funktionsblöcke *newFBL* durch *Configuration.Status(New(newFBL))* anmelden muss, dann fassen wir dies auf Modellebene als eine Transaktion auf und fassen diese zwei Nachrichten zu einer zusammen `ConfigurationStatus(InvalidNew(invFBL,newFBL))`.

Die Integration obiger Anforderung erfordert auch die Erweiterung des lokalen Datenraum des NetworkMastermodells. Wie benötigen für die Durchführung eines SCCs nach einem NCD-Ereignis einen weiteren Modus. Dazu erweitern wir den Datentyp `dModus` um den Wert `ncd`:

```
data dMode = off | init | cfgOk | delayed | ncd;
```

Ferner benötigen wir weitere lokale Variablen `mpr` und `oldFBs`. In Variable `mpr` wird der aktuelle Wert des MPR gespeichert und in `oldFBs` werden beim Auftreten eines NCD-Ereignisses die Funktionsblöcke gespeichert, die vor dem Ereignis im Netzwerk verfügbar waren, um am Ende des folgenden SCC den Unterschied berechnen zu können. Die Variablen `mpr` und `oldFBs` sind vom Typ `Int` bzw. `dFBlockIDList` und werden wie folgt initialisiert:

```
Int           mpr    = 2;
dFBlockIDList oldFBs = noFB;
```

Korrekturen Die Einführung eines weiteren Eingabekanals erfordert eine Einschränkung der Eingabemuster aller Regeln, indem wir diese um den Ausdruck `n?` erweitern. Damit machen wir die implizite Verhaltenserweiterung durch die Erweiterung der Eingabeschnittstelle wieder rückgängig. Ferner können wir in der Zuweisung der Regel *NetOn* den Funktionsaufruf `buildDevList(2)` durch `buildDevList(mpr)` ersetzen, da durch die neu eingeführte Variable `mpr` die Anzahl der Geräte in der Umgebung des NetworkMasters im Modell zur Verfügung steht. Obwohl wir diese Operation auf einer Zuweisung einer Regel ausführen, hat diese keine Auswirkung auf das Modellverhalten, da die Variable `mpr` mit dem Wert `2` initialisiert wurde (vgl. Abschnitt 5.5.2).

Verhaltenserweiterung Wir integrieren die Anforderung "Network Change Delayed" in das Modell, indem wir die Verhaltensregeln *NCD*, *recFBStatusEndNCD* und *recTOAnswerEndNCD* hinzufügen und die Regeln *sndFBGet*, *recFBStatus*, *sndNotOK* und *recTOAnswer* verallgemeinern:

NCD Die Regel *NCD* modelliert den Start eines SCCs nach einem NCD-Ereignis. Dementsprechend wird durch dessen Eingabemuster auf Kanal `n` die Nachricht `nMPR(newMPR)` empfangen und dabei die regellokale Variable `newMPR` gebunden. In der Vorbedingung wird gefordert, dass sich das Modell im Modus *init* befindet und dass sich die Werte der Variablen `mpr` und `newMPR` unterscheiden, d.h. das sich nach dem NCD-Ereignis entweder mehr oder weniger Geräte im Netzwerk befinden. Unter diesen Bedingungen wird in der Zuweisung der Modus auf `ncd` gesetzt, die Geräteliste neu initialisiert `dL=buildDevList(newMPR)`, die Registrierung gelöscht `reg=noR`, die Variable `wA` zurückgesetzt `wA=inst0x00`, die Variable `mpr` auf den neuen Wert gesetzt `mpr=newMPR` und alle gespeicherten Funktionsblöcke der Registrierung zur späteren Verwendung gespeichert `oldFBs=allFBs(reg)`. Dazu verwenden wir die Funktion `allFBs`, die eine FBlockListe aller Funktionsblöcke der übergebenen Registrierung zurückgibt.

recFBStatusEndNCD, recTOAnswerEndNCD Diese sind ähnlich zu den Regeln *recFBStatusSndOk* bzw. *recTOAnswerSndOk*. Sie unterscheiden sich dadurch, dass in der Vorbedingung das Prädikat `is_cfgOk(mode)` durch `is_ncd(mode)` ersetzt wurde und dass in der Ausgabe keine `ConfigurationStatus(OK)`-Nachricht, sondern nicht mehr vorhandene bzw. neu hinzugekommene Funktionsblöcke durch eine `ConfigurationStatus(New(...))`, `ConfigurationStatus(Invalid(...))` oder `ConfigurationStatus(InvalidNew(...))`-Nachricht an das Netzwerk gemeldet werden. Dies wird mit Hilfe der Funktion `regChange` berechnet.

sndFBGet, recFBStatus, sndNotOk, recTOAnswer Das Abfragen von Geräten, das Empfangen von gültigen und ungültigen Statusnachrichten und das Behandeln von nicht antwortenden Geräten während eines SCCs nach einem NCD-Ereignis wird modelliert, indem wir die Vorbedingung der Regeln *sndFBGet*, *recFBStatus* , *sndNotOk* und *recTOAnswer* durch Disjunktion mit dem Prädikat `is_ncd(mode)` verallgemeinern.

Name	Pre	Input	Output	Assign
NetOn	is_off(mode)	net?NetOn; fromEnv?; to?; **n?**		mode=init; dL=buildDevList(**mpr**); reg=noR; wA=inst0x00
NetOff	not(is_off(mode))	net?NetOff; fromEnv?; to?; **n?**		mode=off; dL=noDL; reg=noR; wA=inst0x00
sndFBGet	(is_init(mode) || is_delayed(mode) ||**is_ncd(mode)**) && isNotReq(dL) && (wA == inst0x00))	net?; fromEnv?; to?; **n?**	fromNM!MsgNM(log0x0100, inst2phy(getNotReq(dL)), getNotReq(dL), FBlockIDsGet); t!tAnswer	dL=setNextReq(dL); wA=getNotReq(dL)
recFBStatus	(is_init(mode) || **is_ncd(mode)**) && isNotReq(dL) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL)); to?; **n?**		dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
recFBStatusSndOk	is_init(mode) && not(isNotReq(dL)) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL)); to?; **n?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
sndNotOk	(is_init(mode) || is_delayed(mode) || **is_ncd(mode)**) && wA == INST && not(isAnsOk(INST, LOG, reg))	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsStatus(FBL)); to?; **n?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	mode=init; dL=setNotReq(dL); reg=noR; wA=inst0x00
recTOAnswer	(is_init(mode) || is_delayed(mode) || **is_ncd(mode)**) && isNotReq(dL) && wA != inst0x00	net?; fromEnv?; to?timeoutAnswer; **n?**		wA=inst0x00

Name	Pre	Input	Output	Assign
recTOAnswer-SndOk	is_init(mode) && not(isNotReq(dL) && wA != inst0x00	net?; fromEnv?; to?timeoutAnswer; **n?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(Ok))	mode=cfgOk; wA=inst0x00
swDelay	is_cfgOk(mode) && not(haveAllAns(dL))	net?; fromEnv?; to?; **n?**		mode=delayed; dL=setReq2NotReq(dL)
recNewFBStatus	is_delayed(mode) && isNot-Req(dL) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); to?; **n?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(New(FBL)))	dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
recNewFBStatus-CfgOk	is_delayed(mode) && not(isNotReq(dL)) && wA == INST && isAnsOk(INST, LOG, reg)	net?; fromEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDsSta-tus(FBL)); to?; **n?**	fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(New(FBL)))	mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00
recTOAnswer-CfgOk	is_delayed(mode) && not(isNotReq(dL)) && wA != inst0x00	net?; fromEnv?; to?timeoutAnswer; **n?**		mode=cfgOk; wA=inst0x00
NCD	**(is_cfgOk(mode) || is_delayed(mode)) && mpr != newMPR**	**net?; fromEnv?; to?; n?nMPR(newMPR)**		**mode=ncd; dL=build-DevList(new-MPR); reg=noR; wA=inst0x00; mpr=newMPR; oldFBs=allFBs(reg)**
recFBStatus-EndNCD	**is_ncd(mode) && not(isNotReq(dL)) && wA == INST && isAn-sOk(INST, LOG, reg)**	**net?; fro-mEnv?MsgEnv(LOG, log0x0100, INST, FBlockIDs-Status(FBL)); to?; n?**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, regChange(oldFBs, Registry(INST, LOG, FBL, reg)))**	**mode=cfgOk; dL=setAns(INST, dL); reg=Registry(INST, LOG, FBL, reg); wA=inst0x00**

Name	Pre	Input	Output	Assign
recTOAnswer-EndNCD	**(is_ncd(mode) && not(isNotReq(dL)) && wA != inst0x00**	**net?; fromEnv?; to?timeoutAnswer; n?**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, regChange(oldFBs, reg))**	**mode=cfgOk; wA=inst0x00**
NCDSCC	**(is_init(mode) || is_ncd(mode) && mpr != newMPR**	**net?; fromEnv?; to?; n?nMPR(newMPR)**		**dL=buildDe-vList(new-MPR); reg=noR; wA=inst0x00; mpr=newMPR**
NCDNotOk	**(is_init(mode) || is_cfgOk(mode) || is_delayed(mode) || is_ncd(mode) && mpr == newMPR**	**net?; fromEnv?; to?; n?nMPR(newMPR)**	**fromNM!MsgNM(log0x0100, bc0x03C8, inst0x00, Configuration-Status(NotOk))**	**mode=init; dL=setNotReq(dL); reg=noR; wA=inst0x00**

Tabelle 7.4.: Viertes Inkrement

Spezifikationslücken Bei der Integration der Anforderung "Network Change Delayed" werden im Zusammenspiel mit den bereits modellierten Anforderungen folgende Spezifikationslücken aufgedeckt, da entsprechende Verhaltensregeln für diese Situationen fehlen:

- Wie wird ein NCD-Ereignis behandelt, wenn es während des SCCs der Systemstarts (Modus `init`), beim Abfragen von nicht antwortenden Geräten (Modus `delayed`) oder bei einem SCC nach einem NCD-Ereignisses (Modus `ncd`) auftritt?
- Wie wird ein NCD-Ereignis behandelt, bei dem sich der Wert des MPRs nicht verändert?

Diese Lücken werden geschlossen, indem wir die Regeln *NCDSCC* und *NCDNotOk* hinzufügen und die Regel *NCD* verallgemeinern.

NCDSCC Tritt ein NCD-Ereignis während eines SCCs beim Systemstart bzw. nach einem NCD-Ereignis auf, dann soll der entsprechende SCC neu gestartet werden. Dazu fügen wir die *NCDSCC* hinzu. Diese spezifiziert das Eingabemuster, dass ein NCD-Ereignis aufgetreten ist, und die Vorbedingung, dass sich das Modell im Modus `init` oder `ncd` befindet und dass sich der Wert des MPRs verändert hat (`mpr != newMPR`). Die Zuweisung ist identisch mit der der Regel *NCD* mit Unterschied, dass der Modus nicht verändert wird. Dadurch wird erreicht, dass das Modell den Modus nicht wechselt, sondern lediglich den betreffenden SCC neu startet.

NCDNotOk Den Sonderfall, dass ein NCD-Ereignis auftritt und sich dabei das MPR nicht verändert, behandeln wir, indem wir in den Systemstatus NotOk übergehen, da in diesem Fall unklar ist, was im Netzwerk geschehen ist. Dazu fügen wir die Regel *NCDNotOk* hinzu, in deren Eingabemuster ein NCD-Ereignis empfangen wird (`n?nMPR(newMPR)`) und deren Vorbedingung einen der Modi `init`, `cfgOk`, `delayed` und `ncd` und die Nichtveränderung des MPRs (`mpr == newMPR`) fordert. Die Ausgabe und Zuweisung sind identisch mit denen der Regel *sndNotOk*.

NCD Falls ein NCD-Ereignis auftritt während, der NetworkMaster nicht antwortende Knoten abfragt, dann soll er sich genauso verhalten, als wäre es im Modus `cfgOk` aufgetreten. Diese Festlegung setzen wir im Modell, indem wir die Vorbedingung der Regel *NCD* durch Disjunktion mit dem Prädikat `is_delayed(mode)` verallgemeinern.

Das resultierende Regelsystem ist in Tabelle 7.4 dargestellt, wobei die Änderungen gegenüber dem Vorgängermodell fett gedruckt sind.

Validation&Verifikation Nach Abschluss der Verhaltenserweiterung erzeugen wir durch Simulation oder durch Testfallgenerierung z.B. die Abläufe aus Abbildung B.15 und B.16 (siehe Anhang B.2.4). Der Vergleich mit den Sequenzdiagrammen aus Abbildung 7.5 zeigt, dass diese ein Teil des Modellverhaltens sind und damit die Anforderung "Network Change Delayed" korrekt im Modell realisiert wurde.

Ferner validieren wir das Modellverhalten, das durch Schließen obiger Spezifikationslücken hinzugefügt wurde, durch entsprechende Abläufe. Die Abbildungen B.19 und B.20 (siehe Anhang B.2.4) zeigen Beispiele dazu.

Abschließend wird mit Hilfe der Regressionstestsuite P_3 nachgewiesen, dass das wesentliche Verhalten aus den vorangegangen Inkrementen erhalten wurde.

Regressionstestsuite Wir erweitern die Regressionstestsuite P_3 um die Abläufe aus Abbildungen B.15, B.16, B.19 und B.20 und um weitere (siehe Anhang B.2.4). Wir erhalten damit die Regressionstestsuite P_4 für die nachfolgenden Inkremente.

7.4. Abstraktion zum Kontrollzustandsgraphen

In diesem Abschnitt wenden wir das Transformationsverfahren aus Kapitel 6 auf das vierte Modellinkrement des NetworkMasters an. Wir erhalten unterschiedliche abstrahierte Sichten auf das Verhaltensmodell und können gezielt ausgewählte Aspekte des Modells untersuchen.

Für die erste Transformation wählen die fünf Modi des NetworkMasters als Kontrollzustände. Folglich definieren wir zur Partitionierung des lokalen Datenraums die Prädikatenmenge $P_1 = \{off, init, cfgOk, delayed, ncd\}$ mit

$$\begin{aligned} off &\equiv is\_off(mode) \\ init &\equiv is\_init(mode) \\ cfgOk &\equiv is\_cfgOk(mode) \\ delayed &\equiv is\_delayed(mode) \\ ncd &\equiv is\_ncd(mode). \end{aligned}$$

Nach Durchführung der Transformation erhalten wir aus Tabelle 7.4 den STD in Abbildung 7.6. Anhand dieses Graphen lässt sich gezielt untersuchen unter welchen Regeln der Tabelle 7.4 ein Moduswechsel des NetworkMasters stattfindet. Der Graph zeigt zum Beispiel, dass der NetworkMaster, solange er Geräte in seiner Umgebung abfragt, jeweils in dem Modus `init`, `delayed` bzw. `ncd` verbleibt (Übergänge `sndFBGet`, `recTOAnswer` etc.) und dass in den Modi `off` und `cfgOk` kein Abfragen von Geräten stattfindet. Eine weitere Beobachtung ist, wenn während des Abfragens der Geräte ein Netzwerkreset *Configuration.Status(NotOk)* ausgelöst wird, dass dann der NetworkMaster in den Modus `init` zurückkehrt (Übergänge mit den Namen `sndNotOk` und `NCDNotOk`). Bei der Bildung des dritten und des vierten Modellinkrements (siehe Abschnitt 7.3.5 bzw. 7.3.6) wurden die Anforderungen vervollständigt, dass nicht nur im Modus `init`, sondern auch in den Modi `delayed` und `ncd` ein Netzwerkreset ausgelöst werden kann. Hätte diese Vervollständigung nicht stattgefunden, wäre ein Fehlen entsprechender Übergänge beim Review des STDs 7.6 aufgefallen und die Vervollständigung würde zu diesem Zeitpunkt nachgeholt werden.

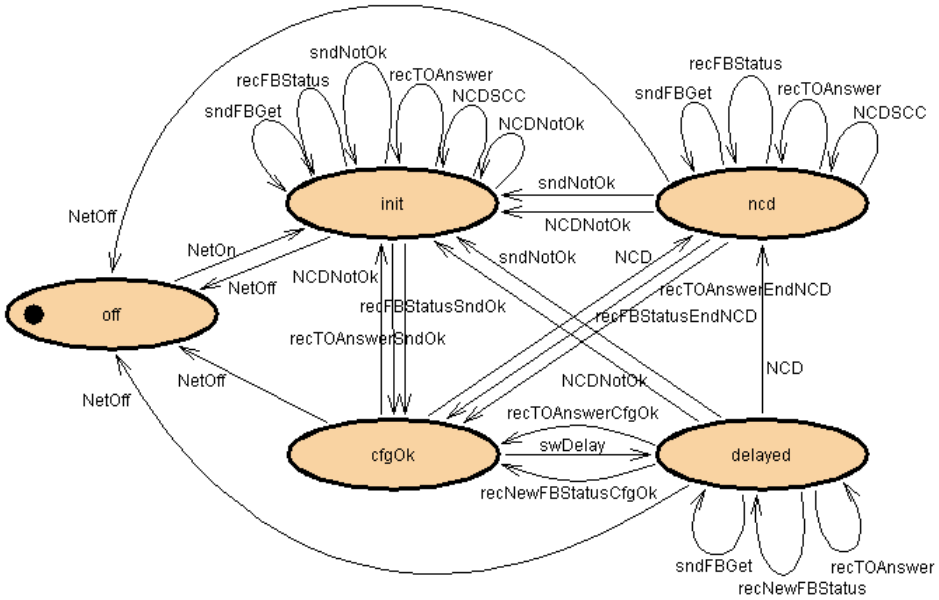


Abbildung 7.6.: Kontrollzustandsgraph mit Partitionierung P_1

Für eine weitere Untersuchung des vierten Modellinkrements wählen wir für die Transformation eine andere Partitionierung des Datenraums bzw. andere Kontrollzustände. Wir wollen nun zwischen Kontrollzuständen unterscheiden, dass der NetworkMaster eines der Geräte abfragt, auf die Antwort eines Geräts wartet, alle Geräte abgefragt hat bzw. abgeschaltet ist. Dazu wählen wir die Partitionierung $P_2 = \{requesting-Devices, waitForFBStatus, cfgOk, off\}$ mit

$$\begin{aligned} requestingDevices &\equiv (is\_init(mode) \,||\, is\_delayed(mode) \,||\, is\_ncd(mode)) \,\&\& \\ &\quad wA == inst0x00 \\ waitForFBStatus &\equiv (is\_init(mode) \,||\, is\_delayed(mode) \,||\, is\_ncd(mode)) \,\&\& \\ &\quad wA\,! = inst0x00 \\ cfgOk &\equiv is\_cfgOk(mode) \\ off &\equiv is\_off(mode) \end{aligned}$$

Nach der Transformation der Tabelle 7.4 erhalten wir den STD in Abbildung 7.7. Bei der Betrachtung des STDs stellen wir fest, dass der NetworkMaster ausschließlich unter Senden einer *FBlockIDs.Get*-Nachricht von Zustand `requestingDevices` in den Zustand `waitForFBStatus` wechseln kann und in diesen zurückkehrt, falls ein NCD auftritt, ein Gerät antwortet oder sein Timer abgelaufen ist und noch weitere Geräte abzufragen sind. Erst wenn das letzte Gerät geantwortet hat bzw. dessen Timer abgelaufen ist, dann wechselt der NetworkMaster von Zustand `waitForFBStatus` in Zustand `cfgOk` (Übergänge, die im Namen auf `SndOk`, `CfgOk` oder `EndNCD` enden). Der NetworkMaster kann also die

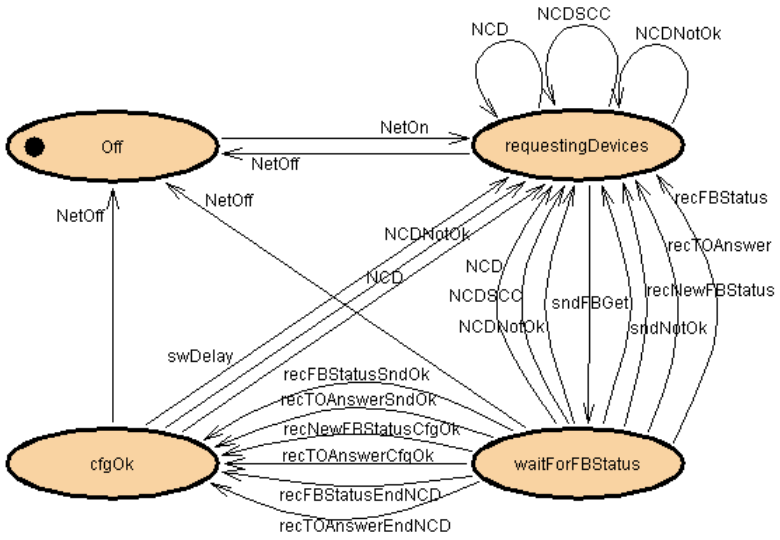


Abbildung 7.7.: Kontrollzustandsgraph mit Partitionierung P_2

Freigabe der Kommunikation auf dem Netzwerk (Kontrollzustand `cfgOk`) nur über den Zustand `waitForFBStatus` erreichen und nicht direkt vom Zustand `requestingDevices` aus. Eine weitere Beobachtung ist, dass ein Übergang von `cfgOk` zu `requestingDevices` nur stattfinden kann, wenn ein NCD-Ereignis auftritt oder einige Geräte noch nicht geantwortet haben. Gäbe es weitere Übergänge läge ein Fehler in der Verhaltensmodell vor.

Fazit Die beiden Beispiele zeigen, dass sich mit der Bildung von abstrahierten Sichten gezielt Symmetrien im Modell finden und analysieren lassen. Auf diese Weise kann bei einem Review leicht festgestellt werden, ob aus Symmetriegründen bestimmte Übergänge vorhanden sein müssen bzw. nicht vorhanden sein dürfen. Weiterhin legt die abstrahierte Sicht Zusammenhänge frei, die u.U. dem Modellierer während der Entwicklung in der detaillierten tabellarischen Sicht verborgen bleiben. Dadurch kann die Bildung unterschiedlicher abstrahierter Sichten auf das ein und dasselbe Modell wesentlich zum Lernprozess über das Modell und seinem Verhalten beitragen.

Insgesamt zeigt die Transformation der tabellarischen Darstellung in verschiedene Kontrollzustandsgraphen (STDs), dass (1) durch die Untersuchung von abstrahierten Sichten auf das Verhaltensmodell neue Zusammenhänge, Symmetrien usw. aufdeckt werden, dass (2) dies zum Erkenntnisgewinn über das Verhaltensmodell beträgt und dass (3) dies das Identifizieren von fehlenden bzw. fälschlicherweise vorhandenen Übergängen erleichtert.

7.5. Modellbasierter Test

In diesem Abschnitt verwenden wir eine Weiterentwicklung des NetworkMaster-Modells aus Abschnitt 7.1 bis 7.4 zum modellbasierten Test einer NetworkMaster-Implementierung. Dabei vergleichen wir Testsuiten von unterschiedlichem Ursprung in Hinblick auf ihre Fehleraufdeckungsrate, Bedingungs-/Entscheidungsabdeckung auf dem Modell und der Implementierung und untersuchen die folgenden Fragen:

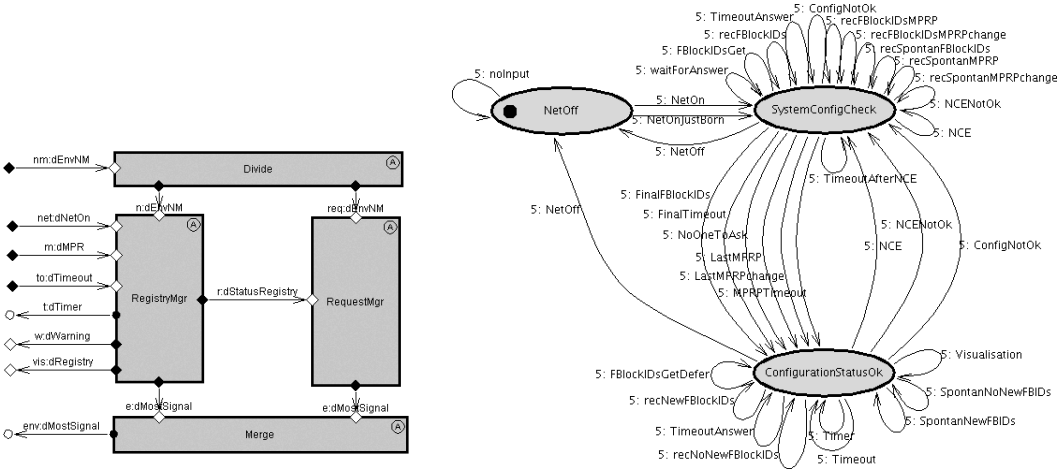
- Wie schneidet anhand der obigen Kriterien die Qualität der modellbasierten Testfälle im Vergleich zu traditionell von Hand erzeugten Testfällen ab?
- Wie schneiden von Hand erzeugte Testfälle mit oder ohne Einbeziehung des Modells im Vergleich zu automatisch generierten Testfällen ab? D.h. ist die Automatisierung bei der Testfallgenerierung hilfreich?
- Gibt es einen Zusammenhang zwischen dem Modell und der Implementierung im Bezug auf ihre Bedingungs-/Entscheidungsabdeckung?
- Wie hängen die Bedingungs-/Entscheidungsabdeckung und die Fehleraufdeckungsrate zusammen?

Ferner sei bemerkt, dass die wesentlichen Ergebnisse dieses Abschnitts in enger Zusammenarbeit mit Pretschner erarbeitet wurden und zusammen mit weiteren Autoren auf der ICSE [PPW+05] zur Veröffentlichung angenommen wurden.

7.5.1. Testmodell und SUT

Testmodell Abbildung 7.8(a) zeigt die funktionale Dekomposition des NetworkMasters in vier AutoFocus-Komponenten. Wie wir in Abschnitt 7.3.1 beschrieben haben, modelliert das Testmodell des NetworkMasters lediglich zwei seiner drei Hauptfunktionalitäten. Die erste –der Aufbau und die Verwaltung der zentralen Registrierung– wird durch die Komponente `RegistryMgr` modelliert. Die zweite –das Auflösen von funktionalen Adressen– wird durch die Komponente `RequestMgr` modelliert. Die Komponenten `Divide` und `Merge` werden aus technischen Gründen benötigt und haben die Aufgabe ankommende MOST-Nachrichten an die betreffende Komponente weiterzuleiten bzw. ausgehende auf einen Kanal zusammenzufassen.

Abbildung 7.8(b) zeigt den STD der Komponente `RegistryMgr`, der der komplexeste im Testmodell ist. Er ist entstanden aus der Transformation eines Regelsystems. Das zugrunde liegende Regelsystem ist eine Weiterentwicklung des Regelsystems aus Abschnitt 7.3.6 und modelliert die Anforderungen eins bis sieben aus Abschnitt 7.3.1. Die Anforderungen acht wird von Komponente `RequestMgr` modelliert. Dem STD aus Abbildung 7.8(b) liegt die folgende Partitionierung des Datenraums in Kontrollzustände



(a) SSD des NetworkMasters

(b) STD der Komponente `RegistryMgr`

Abbildung 7.8.: Testmodell des NetworkMasters

zugrunde:

$$Netoff \equiv is\_off(mode)$$
$$SystemConfigCheck \equiv is\_init(mode) \;||\; is\_ncd(mode)$$
$$ConfigurationStatusOK \equiv is\_cfgOk(mode) \;||\; is\_delayed(mode)$$

Diese Partitionierung ist ein Kompromiss, die Redundanz von gleichen Übergängen zwischen den Kontrollzuständen in Grenzen zu halten, aber dennoch wesentliche Modi des NetworkMasters zu unterscheiden. Der Kontrollzustand *Netoff* modelliert den NetworkMaster im ausgeschalteten Zustand, im Kontrollzustand *SystemConfigCheck* fassen wir die Modi *init* und *ncd* zusammen, d.h. in diesen Kontrollzustand führt der NetworkMaster einen SCC durch und setzt bzw. überprüft den Inhalt der zentralen Registrierung, und im Kontrollzustand *ConfigurationStatusOK* fassen wir die Modi *cfgOk* und *delayed* zusammen, d.h. in diesem Kontrollzustand ist das MOST-Netzwerk lauffähig und alle Netzwerkgeräte dürfen frei kommunizieren, und der NetworkMaster fragt ggf. Geräte ab, die noch auf keine Anfragen geantwortet haben.

Bei der Modellierung des Testmodells wurden die Abstraktionsprinzipien funktionale Abstraktion, Datenabstraktion, Kommunikationsabstraktion und temporale Abstraktion angewandt (vgl. Abschnitt 3.2 und 7.3). Das Testmodell besteht aus 5 AutoFocus-Komponenten mit 21 Kanälen und 34 Ports, 4 STDs mit insgesamt 6 Kontrollzuständen und 45 Übergängen, 6 lokalen Variablen, 26 Datentypen mit 60 Konstruktoren und 94 Funktionen.

SUT Das SUT ist eine Softwaresimulation des NetworkMasters im Betastadium, die an ein reales MOST-Netzwerk angeschlossen ist. Obwohl es vorgesehen ist, NetworkMaster-Implementierungen von Zulieferern der Automobilhersteller entwickeln zu lassen, werden vom Hersteller selbst Software-in-the-loop-Simulationen benötigt, um die Integration mit anderen Geräten im Netzwerk simulieren zu können. Die NetworkMaster-Simulation, deren Schnittstelle identisch zu einem Hardware-NetworkMaster ist, wurde von einer dritten Partei entwickelt.

Um die abstrakten Testfälle aus dem Testmodell auf das konkrete SUT anwenden zu können, schrieben wir einen Übersetzer, der die Testfälle in 4CS-Code übersetzt. 4CS [GAD02] ist eine Testumgebung, um MOST-Geräte zu testen. Der Übersetzer zusammen mit den 4CS-Testumgebung bildet den Testtreiber (vgl. Abschnitt 2.2). Wir benutzen so genannte Optolyser, um Geräte in der Umgebung des NetworkMasters zu simulieren. Optolyser sind frei programmierbare Geräte, mit denen man nahezu jedes beliebiges Gerät im MOST-Netzwerk simulieren kann. In unserem Fall werden die Optolyser von der 4CS-Testumgebung gesteuert, das genau das Verhalten in der Umgebung des NetworkMasters vorgibt, das durch den Testfall vorgeschrieben ist. Auf diese Weise stimulieren wir das SUT. Im 4CS-Code vergleichen wir die Ausgabe des SUTs mit der erwarteten Ausgabe. An Ende jedes Testfalls wird die zentrale Registrierung des NetworkMasters abgefragt und mit der erwarteten aus dem Testmodell verglichen.

Wir verzichten darauf detailliert den Vorgang der Instanziierung von abstrakten Testfällen zu konkreten zu beschreiben. Dies ist die Aufgabe des obigen Übersetzers und wir deuten die Überbrückung der Abstraktionslücke zwischen der Ebene des Testmodells und der der NetworkMaster-Implementierung wie folgt an:

- Bezüglich Datenabstraktion werden Repräsentanten einer Äquivalenzklassen von konkreten Datenwerten, z.B. der Repräsentant aller ungültigen logischen Adressen, durch einen oder unterschiedliche Werte ersetzt.

- Bezüglich Kommunikationsabstraktion werden zusammengefasste Nachrichten, z.B. aufeinanderfolgende *Configuration.Status*-Nachrichten, zu mehreren Einzelnachrichten expandiert.

- Bezüglich temporaler Abstraktion wird das Ablaufen eines Timers durch einen *wait*-Befehl mit entsprechender Zeitdauer ersetzt.

- Die Ausgabe des Testmodells wird in ausführbaren Entscheidungscode übersetzt. Zum Beispiel wird für eine Nachricht in der Ausgabe, die eine Liste von Daten als Parameter enthält, übersetzt in ein Codefragment, das entscheidet, ob die konkrete Liste aus der Ausgabe der Implementierung eine Permutation der erwarteten ist.

Diese Instanziierungs- und Abstraktionsmechanismen können mit Hilfe einer Konfigurationsdatei des Übersetzers festgelegt werden.

7.5.2. Testfallspezifikationen

Wir definierten funktionale Testfallspezifikationen (vgl. Abschnitt 2.2.2), um zu generierende Mengen von Testfällen zu spezifizieren. Jede Testfallspezifikation bezieht sich auf eine Teilfunktionalität des NetworkMasters bzw. auf Teile seines Verhaltens, das er in speziellen Situation zeigt. Wir bildeten zunächst informelle Testziele, die Klassen von Testfallspezifikationen beschreiben.

TS1 Startet der NetworkMaster das Netzwerk zu normaler Funktion auf, wenn alle Geräte in seiner Umgebung korrekt auf seine *FBlockIDs.Get*-Anfragen antworten, so dass alle Geräte im Netzwerk frei kommunizieren können?

TS2 Antwortet der NetworkMaster korrekt auf Registrierungsanfragen *CentralRegistry.Get* und unterstützt er die verschiedenen Varianten?

TS3 Wie verhält sich der NetworkMaster, wenn eines oder mehrere Geräte nicht auf *FBlockIDs.Get*-Anfragen antworten?

TS4 Löst der NetworkMaster einen Netzwerkreset *Configuration.Status(NotOk)* aus, wenn er von einem der Geräte in der Umgebung eine ungültige *FBlockIDs.Status*-Nachricht empfängt?

TS5 Registriert der NetworkMaster spontane *FBlockIDs.Status*-Nachrichten, d.h. wenn er *FBlockIDs.Status*-Nachrichten ohne vorangegangene *FBlockIDs.Get*-Anfrage empfängt?

TS6 Rekonfiguriert der NetworkMaster das Netzwerk korrekt, wenn zur Laufzeit genau ein MPR+ oder MPR- Ereignis auftritt, d.h. wenn sich ein Gerät spontan mit dem Netzwerk verbindet bzw. vom Netzwerk trennt?

TS7 Rekonfiguriert der NetworkMaster das Netzwerk korrekt, wenn zur Laufzeit mehr als ein MPR+ bzw. MPR- Ereignis auftritt?

Wir implementierten diese Testziele, indem wir sie miteinander kombinierten und in 33 Testfallspezifikationen verfeinerten. Die Testfallspezifikationen sind in unserem Fall funktionale (vgl. Abschnitt 2.2.2) und werden spezifiziert, indem auf den Ein- und Ausgabekanälen des Testmodells bestimmte Nachrichten in bestimmter Reihenfolge auftreten bzw. nicht auftreten dürfen.

7.5.3. Testfallgenerierung

Testfallgenerierung wird durchgeführt, indem das Testmodell, d.h. das AutoFocus-Modell des NetworkMasters, in eine *Constraint Logic Programming Language (CLP)* übersetzt wird, die Testfallspezifikationen aus Abschnitt 7.5.2 in Form von Constraint-Mengen hinzugefügt werden und das resultierende CLP-Programm ausgeführt wird.

Wir fassen im folgenden die Übersetzung von AutoFocus-Modellen kurz zusammen: Jeder Kontrollzustandsübergang eines STDs einer AutoFocus-Komponente K auf unterster Hierarchiestufe wird in eine CLP-Formel

$$step^K(\vec{\sigma}_{src}, \vec{\iota}, \vec{o}, \vec{\sigma}_{dst}) \Leftarrow guard(\vec{\iota}, \vec{\sigma}_{src}) \wedge assgmt(\vec{o}, \vec{\sigma}_{dst})$$

übersetzt, wobei die Komponente K unter dem Eingabevektor $\vec{\iota}$ vom Zustand $\vec{\sigma}_{src}$ (Kontroll- und Datenzustand) in den Zustand $\vec{\sigma}_{dst}$ unter der Ausgabe von $\vec{o}$ übergeht, falls der Wächter *guard* erfüllt ist. Die unter Umständen vorhandenen Referenzen von Wächtern und Zuweisungen auf rekursive Datentypen und Funktionen können ebenso leicht übersetzt werden.

Liegt eine Komposition der Komponenten $\mathcal{C}$ vor, d.h. deren Ports wurden in einem SSD mit Kanälen verbunden, dann benötigen wir ein Treiberprädikat. Dieses Treiberprädikat ruft anschließend alle Prädikate der Zustandsmaschinen auf, die jedem Element der Menge der Komponenten $\mathcal{C}$ entspricht. Ferner wickelt dieses Prädikat die Kommunikation zwischen zwei Komponenten ab. Komponenten K, die sich nicht auf unterster Hierarchiestufe befinden, enthalten folglich Subkomponenten $K = \{k_1, \ldots, k_n\}$ und werden rekursiv in

$$step^K(\vec{\sigma}^K_{src}, \vec{\iota}^K, \vec{o}^K, \vec{\sigma}^K_{dst}) \Leftarrow \bigwedge_{j=1}^{n} step^{k_j}(\vec{\sigma}^{k_j}_{src}, \vec{\iota}^{k_j}, \vec{o}^{k_j}, \vec{\sigma}^{k_j}_{dst})$$

übersetzt, wobei interne Kanäle als lokale Variablen von K codiert und damit Bestandteile von σ^K_{src} bzw. σ^K_{dst} werden.

Das einfache Übersetzungsschema ist eine Konsequenz des einfachen zeitsynchronen Ausführungsmodells von AutoFocus. Die Ausführung des resultierenden CLP-Programms zählt sukzessive alle Abläufe des Modells unter den Einschränkungen der Testfallspezifikationen auf. Genauer gesagt wird das Modell symbolisch ausgeführt, d.h. anstatt jeden einzelnen Zustand des Modells bestehend aus Eingabe, Ausgabe und lokalen Zustand aller Komponenten zu berechnen, wird mit Mengen von Zuständen gearbeitet. Wenn zum Beispiel zu einem Zeitpunkt der Bestandteil einer Nachricht einer der Werte von `log0x0100`, `log0x0101` oder `log0x0103` ist, dann entsteht keine Aufspaltung in drei unterschiedliche Suchebäume, stattdessen wird die Berechnung lediglich mit einem weitergeführt. Dies hat Auswirkungen auf die Anzahl der generierten Testfälle und auf die Zustandsspeicherungsstrategie. Da wir in der Regel nicht daran interessiert sind, einen Zustand zweimal zu besuchen, müssen wir, anstatt jeden einzelnen Zustand in einer Hash-Tabelle zu speichern, die Inklusion von Zustandsmengen prüfen. Wir führen keine weiteren Details der Übersetzung nach CLP und über die Zustandsspeicherung aus. Wir verweisen dafür auf die ausführlichen Arbeiten von Pretschner [Pre03b, PSAK04].

7.5.4. Testsuiten

In diesem Abschnitt beschreiben wir sieben verschiedene Testsuiten, anhand denen wir untersuchen welche Art von Testsuite im Vergleich zu anderen bezüglich ihre Fehleraufdeckungsrate abschneidet. Wir untersuchen, wie (1) modellbasierte im Vergleich zu

handgeschriebenen Tests abschneiden, (2) wie automatisch generierte Tests gegenüber handgeschriebenen abschneiden und (3) welchen Nutzen explizit formulierte Testfallspezifikationen haben.

Test-suite	Automa-tisierung	Modell-basierung	Testfall-spezifikation
A	manuell	ja	ja
B	auto	ja	ja
C	auto	ja	nein
D	auto	nein	entfällt
E	manuell	nein	entfällt
F	manuell	ja	nein
G	manuell	nein	entfällt

Tabelle 7.5.: Testsuiten

Zufällig ohne Modell generierte Test dienen als Nullhypothese, um zu überprüfen, ob modellbasiertes Testen irgendwelche Vorteile zeigt. Die Länge aller Tests beträgt zwischen 8 und 25 Schritten im Testmodell, die durch eine Postambel von 3 bis 12 Schritten automatisch erweitert wird. Die Postambel ist notwendig, um den internen Zustand des SUT einschätzen zu können, da ein direkter Zugriff von außen nicht möglich ist. Wir beschreiben die sieben Testsuiten wie folgt:

[A] Eine Testsuite, die *von Hand* erstellt wurde, indem mit dem *Testmodell* Simulationsläufe *manuell* und *interaktiv* erstellt wurden. Diese Testsuite besteht aus 105 Testfällen.

[B] Mehrere Testsuiten, die *automatisch mit* Einbezug von *Testfallspezifikationen* aus Abschnitt 7.5.2 und dem *Testmodell* generiert wurden. Diese Testfälle wurden mit zusätzlichen Einschränkungen, welche den Testfallspezifikationen entsprechen, zufällig generiert. Die Anzahl der Testfälle pro Testsuite schwankt zwischen 40 und 1000.

[C] Mehrere Testsuiten, die *automatisch ohne* Einbezug der *Testfallspezifikationen* aber *mit dem Testmodell zufällig* generiert wurden, d.h. das Testmodell wurde ohne zusätzliche Einschränkungen zufällig ausgeführt.

[D] Mehrere Testsuiten, die *zufällig* aber *ohne Einbezug* der Logik des *Testmodells* generiert wurden. D.h. die Grundlage für die Generierung ist nur die syntaktische Schnittstelle des Testmodells und einige Grundsatzregeln, wie z.B. zu Beginn des Testfalls muss der NetworkMaster mit einer entsprechenden Nachricht erst eingeschaltet werden. Um dennoch erwartete Ausgaben zu den zufällig erzeugten Eingaben zu erhalten, legten wir diese Eingaben an das Testmodell an, um die erwarteten Ausgaben zu erhalten.

[E] Eine Testsuite, die *von Hand* erzeugt wurde und den *ursprünglichen Message Sequence Charts (MSCs) der Spezifikation* entsprechen. Diese Testsuite enthält 43

Testfälle.

[F] Eine Testsuite, die *von Hand* erzeugt wurde, die neben den ursprünglichen Spezifikations-MSCs aus Testsuite E weitere MSCs enthält. Diese MSCs entstanden bei Klärung von Unklarheiten, während der Entwicklung des Testmodells. Diese Testsuite enthält 51 Testfälle.

[G] Eine Testsuite, die *von Hand* mit traditionellen Mitteln erzeugt wurde, d.h. *ohne das Modell* mit einzubeziehen. Diese Testsuite enthält 61 Testfälle.

Die Tabelle 7.5 gibt den Überblick und fasst zusammen, welche Testsuiten von Hand erzeugt und welche automatisch wurden. Ferner gibt sie Auskunft, wenn eine Testsuite auf Basis des Testmodells erzeugt wurde, ob die Testfallspezifikationen aus Abschnitt 7.5.2 einbezogen wurden oder nicht. Der Unterschied zwischen Testsuiten E bzw. F und G besteht darin, dass E bzw. F auf Anforderungsdokumenten basieren, G dagegen auf Testdokumenten. Analog gilt dies auch für Testsuiten F und A. F ist Resultat von Aktivitäten in der Anforderungsanalyse und A von Testaktivitäten.

Generierung Die Generierung der automatisch erzeugten Testsuiten B, C und D wurde wie folgt ausgeführt:

Testsuiten von Typ B Wir übersetzten die 33 Testfallspezifikationen in CLP-Constraints und fügten diese jeweils einer CLP-Übersetzung des Testmodells hinzu. Die resultierenden CLP-Programme führten wir für Testfälle bis zu einer Länge von 25 Modellschritten aus. Die Berechnung wurde nach einer vorgegebenen Zeit abgebrochen oder kam früher zum Erliegen, falls aufgrund von Zustandsspeicherung und Testfallspezifikationen keine weiteren Testfälle mehr aufgezählt werden konnten. Dies führte zu Testsuiten, die jeweils einer Testfallspezifikation entsprechen. Während der Generierung wird die Wahl der Eingabenachrichten und der Kontrollzustandsübergänge per Zufall ausgewählt. Um die Probleme, die aufgrund der Tiefensuche entstehen, abzuschwächen, verwendeten wir unterschiedliche *Seeds*, d.h. unterschiedliche Startpunkte der Berechnung, die die zufällige Auswahl der Eingaben und der Übergänge beeinflussen. Wir wählten für die Berechnungen gemäß einer Testfallspezifikation bis zu 15 verschiedene Seeds aus und bestimmten nach Abschluss der Berechnungen per Zufall Testfälle aus den berechneten Testfallmengen. Mit anderen Worten: wir erzeugten die Testsuiten von Typ B, indem wir aus allen Testfällen, die den Testfallspezifikationen genügen, zufällig ein Menge von Testfällen auswählten.

Testsuiten von Typ C Da zu dieser Gruppe von Testsuiten der Einbezug von Testfallspezifikationen nicht vorgesehen ist, entfiel das Hinzufügen von weiteren CLP-Constraints zu der CLP-Übersetzung des Testmodells. Analog wie oben wurde das CLP-Programm mit unterschiedlichen Seeds zu Ausführung gebracht und nach einer vergebenen Zeit abgebrochen. Aus den resultierenden Testfallmengen wurden Testfälle per Zufall ausgewählt, um Testsuiten von Typ C in unterschiedlicher

Größe zu erhalten.

Testsuiten von Typ D Hier wurde nur die syntaktische Eingabeschnittstelle des Testmodells in ein CLP-Programm übersetzt, d.h. die vollständige Kontrolllogik der Zustandsübergänge des Testmodells wurde im CLP-Programm weglassen. Ferner wurden zusätzlich grundlegende Einschränkungen codiert, z.B. dass im ersten Schritt des Testfalls das Gerät eingeschaltet werden muss. Das resultierende Programm wurde mit unterschiedlichen Seeds ausgeführt, um beliebige Sequenzen von Eingabenachrichten zu erhalten, ohne die Kontrolllogik des Testmodells mit einzubeziehen. Im Anschluss wurden die Testdaten an das Testmodell angelegt, um zugehörige Ausgaben zu erhalten. Aus den so erzeugten Testfallmengen wurden wiederum Testfälle per Zufall ausgewählt, um schließlich Testsuiten von Typ D in unterschiedlicher Größe zu erhalten.

Von Hand erzeugte Testfälle wurden mit Ausnahme der Testsuite A ohne Kenntnis des Testmodells gebildet. Um jedoch die erwarteten Ausgaben für die Testfälle zu erhalten, legten wir interaktiv die Testdaten am Testmodell an. Das Testmodell für die manuelle Erzeugung von Testfällen zu benutzen, bedeutet hier also, dass das Wissen über das Testmodell und seiner Struktur beim Ableitungsprozess von Testfällen miteinbezogen wurde.

Zur Wahl der Testsuiten Die Fragen, die wir zu Beginn des Abschnitts 7.5 formuliert haben, präzisieren wir nun wie folgt:

1. Führt der Einsatz von Testmodellen zu besseren Testfällen? Diesbezüglich stellen die Testsuiten D, E und G die Nullhypothesen gegenüber den Testsuiten A, B, C und F dar, d.h. modellbasiertes Testen ist vorteilhaft gegenüber traditionellen Methoden, Testfälle von Hand oder zufällig zu erzeugen. Der Zweck der Testsuiten E und F ist, vergleichende Zahlen angeben zu können, wenn lediglich die Anforderungssequenzen zum Test betrachtet werden.

2. Rechtfertigt die Automatisierung den Aufwand? Im Vergleich zu Testsuiten B und C stellt Testsuite A die Nullhypothese dar, ob Automatisierung zu irgendwelchen Vorteilen führt. Testsuite B hat zum Ziel, den Prozess des automatisierten modellbasierten Testens als Ganzes einzuschätzen, und der Vergleich zwischen Testsuite C und B wird genutzt, um das Konzept der Testfallspezifikationen einzuschätzen.

7.5.5. Messungen und Interpretationen

Wir untersuchen die Testsuiten aus dem vorangegangenen Abschnitt nach folgenden Kriterien:

- Anzahl der aufgedeckten Fehler,
- ihrer Bedingungs-/Entscheidungsabdeckung auf dem Modell und

- ihrer Bedingungs-/Entscheidungsabdeckung auf dem Implementierungscode.

Aufgedeckte Fehler Bei der Entwicklung der Testmodells deckten wir drei schwerwiegende Spezifikationsfehler und sieben Auslassungen auf. Während dem Testen der NetworkMaster-Simulation fanden wir insgesamt 26 Fehler, davon waren zwei Fehler im Modell, d.h. es wurden Anforderungen falsch interpretiert und im Testmodell modelliert. 13 von 26 Fehlern waren Programmierfehler und 11 Lastenheftfehler, d.h. die Behebung von Lastenheftfehler würde die Korrektur, Vervollständigung oder Präzisierung von Lastenheftspezifikationen erfordern, wohingegen dies bei Programmierfehlern nicht erforderlich wäre. 15 von den insgesamt 24 Fehlern in der Implementierung wurden von einen Domänenexperten als schwerwiegend eingestuft, d.h. ihr Auftreten würde zur Laufzeit die Funktion des Gesamtsystems stark beeinträchtigen. Da die Fehlerdiagnose, d.h. die Zuordnung eines auftretenden Fehlverhalten zu einer Fehlerverhaltensklasse, nicht automatisch, sondern von Hand durchgeführt werden musste, war die Anzahl der ausgewerteten Tests eingeschränkt. Für die Testsuite B wählten wir per Zufall 4 mal 5 und einmal 10 Testfälle pro Testfallspezifikation aus. Dies führt zu insgesamt 4*5*33+1*10*33=990 Testfällen der 5 Testsuiten von Typ B. Für Testsuite C wählten wir 4 mal 150 Testfälle und 3 mal 150 für Testsuite D aus. Die Abbildung 7.9 zeigt die Anzahl der Fehler bzw. Fehlerklassen die jeweils von einer Testsuite aufgedeckt wurden. Dabei ist der erste Balken von den Testsuiten B der, der aus 10*33=330 Testfällen besteht.

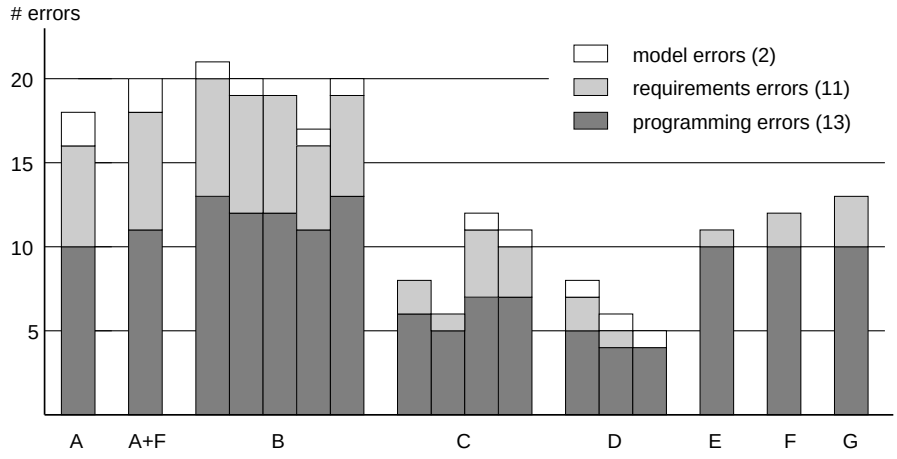


Abbildung 7.9.: Aufgedeckte Fehler

Eine wesentliche Beobachtung ist, dass die modellbasierten Testsuiten A und B und die von Hand erzeugten Testsuiten E, F und G alle nahezu gleich viele Programmierfehler aufdecken, wohingegen Lastenfehler hauptsächlich durch die modellbasierten Testsuiten gefunden werden. Die Ursache für diese Beobachtung ist nicht überraschend, da bei der Entwicklung des Testmodells eine gründliche Überprüfung aller Anforderungs- und Spezifikationsdokumente erfolgte und diese direkt in die Modellierung des Testmodells

einfloss. Weitere Beobachtungen sind, dass keine der Testsuiten alle 26 Fehler aufdeckt und dass keine Korrelation zwischen Testsuiten und der Schwere der Fehler existiert.

Testsuite A deckt geringfügig weniger Fehler auf als die Testsuiten B. Die automatisch generierten Testsuiten subsumieren alle Fehler der Testsuite A bis auf einen Modellierungsfehler. Für zwei Fehler, die durch Testsuite B aber nicht durch A gefunden wurden, wurden entsprechende Testfälle bei manuellen Erstellung der Testsuite A einfach vergessen. Die verbleibenden Fehler, die durch Testsuite B aber nicht durch A aufgedeckt wurden, sind Permutationen von anderen Testfällen, die auf einen Entwickler abstrus wirken. In diesem Fall gehen wir davon aus, dass solche Testfälle von einen menschlichen Testentwickler nicht gefunden würden und dass dies ein Folge der automatischen Testfallgenerierung mit zufälliger Auswahl ist. Wenn wir Testsuiten B und F vergleichen, dann findet Testsuite F einen Fehler, den Testsuite B nicht aufdeckt.

Zufällig erzeugte Testfälle ohne Einbeziehung von Testfallspezifikationen (Testsuiten C, die insgesamt 14 Fehler aufdecken) finden ungefähr gleich viele Fehler wie die von Hand und ohne Modell erstellten Testfälle (Testsuiten E,F und G). Insgesamt finden die letzteren mehr Programmierfehler als Lastenheftfehler. Fehler aufgedeckt von Testsuite C werden auch durch die Testsuiten A und B aufgedeckt, die hauptsächlich Abläufe enthalten, welche mit hoher Wahrscheinlichkeit ausgeführt werden.

Testsuiten D, welche beliebige Eingaben an die SUT anlegen, decken insgesamt sieben Fehler auf und damit die geringste Anzahl in Vergleich zu allen anderen Testsuiten. Alle ihre Fehler werden auch von Testsuite B aufgedeckt, aber es gibt drei Fehler, die nicht von Testsuite A gefunden werden. Diese sind Fälle, die aus Entwicklersicht als abwegig eingestuft würden.

Insgesamt finden Testsuiten erzeugt auf Basis von Testfallspezifikationen mehr Fehler und schneiden damit besser ab als Testsuiten die zufällig ohne Testfallspezifikationen erzeugt wurden.

Abdeckung auf dem Modell Wir verwenden Bedingungs-/Entscheidungsabdeckung als Abdeckungskriterium. Bedingungs-/Entscheidungsabdeckung misst die Ergebnisse der einzelnen Bedingungen in einer zusammengesetzten Bedingung und zusätzlich den Ausgang der gesamten Entscheidung. 100% Abdeckung setzt also voraus, dass jede atomare Bedingung mindestens einmal zu true und einmal zu false und die gesamte Entscheidung zu beiden Wahrheitswerten ausgewertet wurde.

Das Testmodell enthält insgesamt 1722 Bedingungen und Entscheidungen, die in den Kontrollzustandsübergängen und in den funktionalen Programmen auf den Übergängen auftreten. Die Implementierung dagegen enthält 916 Bedingungen und Entscheidungen. Abbildung 7.10 zeigt die Bedingungs-/Entscheidungsabdeckung auf dem Javacode, der aus dem Testmodell zu Simulationszwecken erzeugt wurde. Wir erzeugten Testsuiten unterschiedlicher Größe. Jeder Messpunkt in der Graphik zeigt das Mittel von 25 Experimenten gleicher Größe und des jeweiligen Testsuitetyps, d.h. wir wählten 25 mal

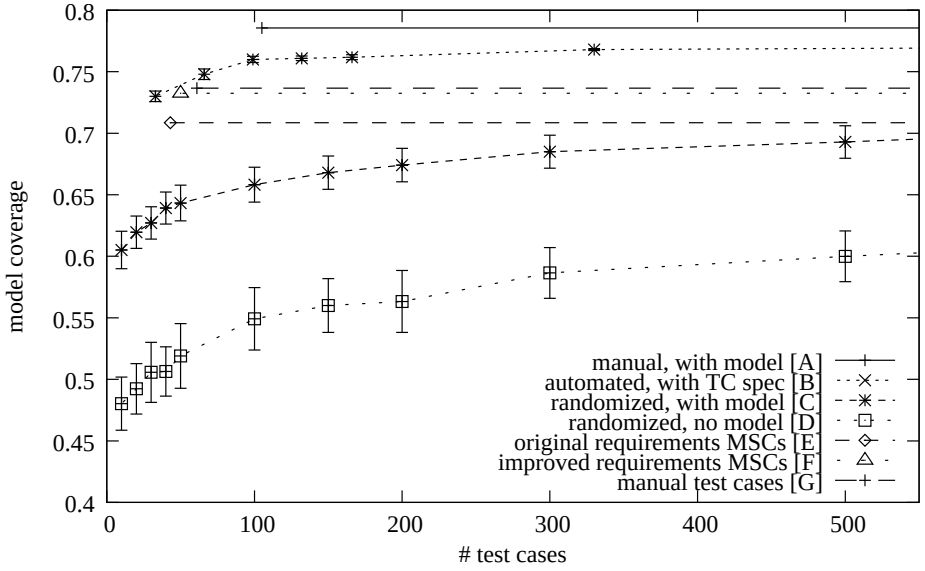


Abbildung 7.10.: Modellabdeckung

die betreffende Anzahl von Testfällen aus der ursprünglich erzeugten Testfallmenge. Die Grundmengen zur Auswahl enthielten zwischen 6'000 und 10'000 Testfälle. Die Fehlerbalken zeigen ein Konfidenzintervall von 98% für das betreffende Mittel unter der Annahme, dass die Auswahl normal verteilt ist.

Die erste Beobachtung ist, dass die Abdeckung von keiner Testsuite den Wert von 79% übersteigt. Der Grund ist die Behandlung von "Patternmatching" bei der Generierung des Javacodes, der triviale immer zu wahr auszuwertende Bedingungen enthält. Die zweite Beobachtung ist, dass bis auf die Testsuiten D, die ohne dem Modell automatisch erzeugt wurden, alle Konfidenzintervalle für ein gegebenes Mittel relativ klein sind, d.h. der Zufall hat bei der zufälligen Auswahl der Testfälle aus den zugrunde liegenden Testfallmengen eine geringe Auswirkung auf die Modellabdeckung. Ferner zeigen die Kurven zu den Testsuiten B, C und D die gleiche Grundform und erreichen früh eine Sättigung der Abdeckung, d.h. ab ca. 200 Testfällen führt die Erhöhung der Anzahl der Testfälle nur noch zu einem vergleichsweise geringem Anstieg der Abdeckung.

Testsuite A erreicht die höchste Abdeckung, die nicht einmal von der Testsuite B erreicht wird. Das gute Abschneiden der Testsuite A kann wie folgt erklärt werden: die gleiche Person[10], die das Testmodell und die Testfallspezifikationen entwickelte, erzeugte die Testsuite A und versuchte dabei intuitiv die Struktur und damit alle Bereiche des Testmodells abzudecken. Folglich konnte in unserer Studie die Automatisierung nicht die Abdeckung der von Hand erzeugten modellbasierten Tests erreichen. Testsuite A deckt jedoch nicht alle Bedingungen/Entscheidungen ab, die durch die Testsuiten B bis G abgedeckt werden, d.h. obwohl Testsuite A absolut die höchste Abdeckung erzielt, gibt es

[10] d.h. der Verfasser dieser Arbeit

14 weitere atomare Bedingungen, die von Testsuiten B bis G abgedeckt werden aber nicht von A. Weiterhin stellte sich heraus, dass die automatisch generierten Testfälle als Folge des Zufalls mehr mögliche Eingabenachrichten abdeckten als die von Hand erzeugten Testfälle.

Die Testsuiten F und G sind die nächst besten Testsuiten bezüglich der Bedingungs-/Entscheidungsabdeckung. Dieses erklären wir dadurch, dass die verbesserten Spezifikations-MSCs (Testsuite F) und die manuell ohne Modell erzeugten Tests (Testsuite G) die "wesentlichen" Abläufe des Modells enthalten. Die Testsuiten C, die auf Basis des Testmodells zufällig erzeugte Testfälle enthalten, erreichen bei ca. 500 Testfällen in etwa die gleiche Abdeckung wie die Testsuite E.

Der Vergleich der Testsuiten C und D mit Testsuite B zeigt, dass die Verwendung von Testfallspezifikationen eine höhere Abdeckung mit weniger Testfällen erzielt. Diese Beobachtung ist nicht überraschend, da gerade die Testfallspezifikationen die Aufgabe haben, das Testmodell in sinnvolle Teile zu schneiden. Wenn Testfälle für diese Teile generiert werden, die den verschiedenen Strukturelementen oder den speziellen Bedingungen im Testmodell entsprechen, dann steigt die Wahrscheinlichkeit, dass diese speziellen Bedingungen abgedeckt werden. Aus technischer Sicht wird also der Zustandsraum des Testmodells in kleinere Teile zerteilt und auf diesen Subräumen wird eine zufällige Testfallgenerierung durchgeführt. Je kleiner der Zustandsraum ist, desto wahrscheinlicher ist es, auch die meisten seiner Elemente per Zufall zu erreichen.

Abdeckung auf der Implementierung Aus technischen Gründen konnten wir mehrere Testsuiten nicht automatisch hintereinander gegen die Implementierung ablaufen lassen. Deshalb konnten wir nicht die gleiche Anzahl an Experimenten wie in vorangegangenen Abschnitt auf der Implementierung durchführen. Folglich ist es uns nicht möglich gewesen, die Abdeckung gegen die Testsuiten mit unterschiedlichen Größen aussagekräftig aufzutragen. Wir beschränken uns stattdessen in Abbildung 7.11, die Beziehung zwischen der Modell- und der Implementierungsabdeckung zu zeigen.

Aufgrund der Abstraktionen im Testmodell erreicht die Abdeckung auf der Implementierung ein Maximalwert von 75%. Wir schlossen einige C-Funktionen der Implementierung aus der Messung aus, für die es keine Entsprechung im Testmodell gibt. Dies war jedoch nur zum Teil möglich, da vieles mit für die Messung relevanten Teilen stark verknüpft war, und deshalb aus der Messung nicht entfernt werden durfte. Wir beobachten, dass Testsuiten B, C und D, die unter Einfluss des Zufalls erzeugt wurden, sich in verschiedenen Regionen konzentrieren. Dies ist auf den Einfluss des Zufalls zurückzuführen. Zum Vergleich: bei unseren Messungen auf dem Modell (Abb. 7.10) stellten wir mit einem Konfidenzintervall von 98% fest, dass Testsuiten von einem Typ nahezu konstante Abdeckungen besitzen.

Grob führen im Mittel die zufällig erzeugten Testsuiten C und D zur gleichen Abdeckung auf der Implementierung. Wie auch beim Modell zeigt sich, dass die Abdeckung auf der Implementierung der Testsuiten B tendenziell höher liegt als die der Testsuiten C bis

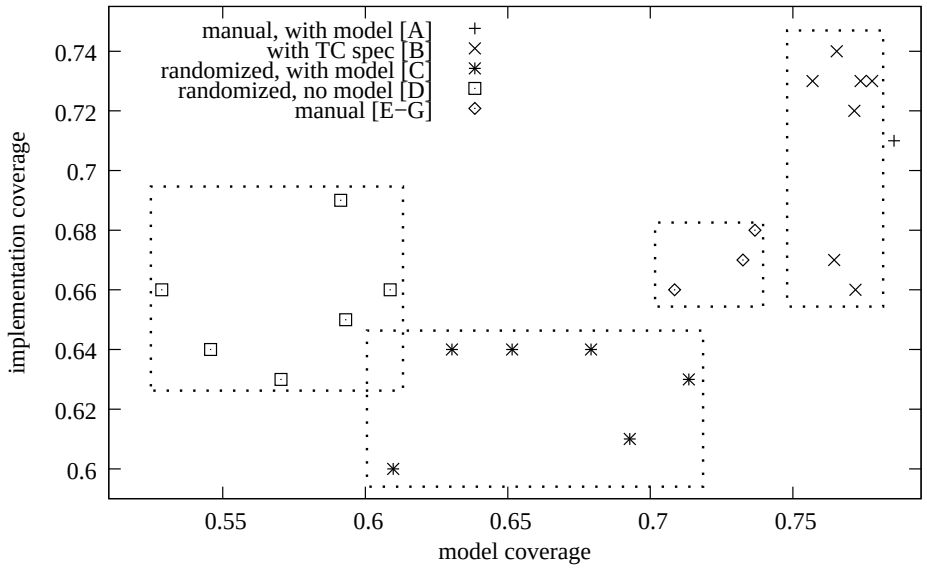
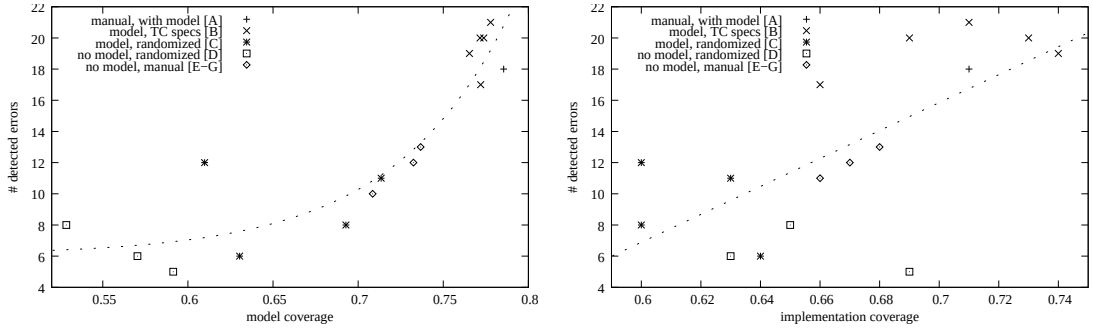


Abbildung 7.11.: Modell- und Implementierungsabdeckung

G. Insgesamt gibt es nur eine schwache Korrelation zwischen Modell- und Implementierungsabdeckung. Wir erwarteten eine stärkere Korrelation, da die wesentlichen Abläufe der Hauptfunktionalitäten von beiden identisch sind. Dies könnte jedoch nicht bestätigt werden. Weiterhin suggeriert die Graphik, dass eine relativ gute Korrelation zwischen den von Hand erzeugten Testsuiten A, E, F und G besteht. Die geringe Zahl an Proben verbietet jedoch eine statistische Analyse. Obwohl die manuell und modellbasierte Testsuite A auf dem Testmodell die höchste Abdeckung erzielte, insbesondere höhere als die Testsuiten B, ist dies bei der Implementierungsabdeckung nicht der Fall und liegt im Mittel der Abdeckungen der Testsuiten B. Dies ist wiederum auf die Tatsache zurückzuführen, dass die Implementierung in Zweige lief, die es auf Modellebene nicht gibt.

Abdeckung und Fehleraufdeckung Abschließend zeigen wir Beziehung zwischen Modell- und Implementierungsabdeckung und der Fehleraufdeckungsrate in den Abbildungen 7.12(a) bzw. 7.12(b).

Beide Abbildungen suggerieren eine positive Korrelation zwischen Bedingungs-/Entscheidungsüberdeckung und der Fehleraufdeckungsrate der einzelnen Testsuiten, wobei die Messpunkte im Falle der Implementierung stärker verteilt sind. Auffällig in Abbildung 7.12(b) ist, dass die Testsuiten D eine vergleichsweise hohe Abdeckung erzielen, aber eine geringe Anzahl von Fehlern aufdecken. Dies können wir wie oben erklären, dass in diesen Fällen Zweige der Implementierung abgedeckt werden, für die es auf Modellebene keine entsprechenden gibt. Weiterhin stellen wir fest, dass eine höhere Abdeckung nicht immer zu einer höheren Fehleraufdeckungsrate führt.



(a) Modellabdeckung und Fehleraufdeckung

(b) Implementierungsabdeckung und Fehleraufdeckung

Abbildung 7.12.: Abdeckung und Fehleraufdeckung

Zusammenfassung Insgesamt machen wir aufgrund der obigen Messungen folgende Beobachtungen:

- Der Einsatz von Testmodellen führt signifikant zu einer höheren Anzahl von aufgedeckten Lastenheftfehlern. Im groben hängt die Zahl der aufgedeckten Programmierfehler nicht vom Einsatz des Modells ab. Zufallstests unabhängig vom Einsatz des Modells decken weniger Fehler auf als alle anderen Testsuitetypen.
- Wenige manuell und modellbasierte Testfälle decken etwa gleich viele Fehler auf wie viele automatisch generierte modellbasierte Testfälle. Dies stellt den Nutzen der Automatisierung bei der Testfallerzeugung im Bereich des modellbasierten Testens in Frage.
- Bedingungs-/Entscheidungsabdeckung auf Modell und Implementierung korrelieren lediglich moderat miteinander.
- Bedingungs-/Entscheidungsabdeckung korreliert positiv mit der Fehleraufdeckungsrate, obwohl eine höhere Abdeckung nicht notwendigerweise zu einer höheren Fehleraufdeckungsrate führt.
- Die Tatsache, dass unterschiedliche Testsuiten verschiedene Fehler aufdecken, suggeriert, dass eine Kombination von verschiedenen Testsuiten sinnvoll ist.

7.5.6. Ergebnisse und Diskussion

Dass der Einsatz von ausführbaren Verhaltensmodellen in der Anforderungsanalyse und beim Aufdecken von Fehlern hilfreich ist, ist keine Überraschung, denn Verhaltensmodelle sind abstrakte Prototypen des SUT. Der Nutzen von Prototypen in der Anforderungsanalyse wurde bereits von Boehm u.a. [BGS84] festgestellt. Eine weitere bemerkenswerte

aber nicht überraschende Feststellung ist, dass die Anzahl der aufdeckten Programmierfehler unabhängig vom Einsatz von Modellen ist.

Dass der Nutzen von Automatisierung einer gründlichen Untersuchung bedarf, entspricht der Vermutung aus früheren Studien [PPS+03, PSAK04]. Es werden nach wie vor Entwickler benötigt, um Testfallspezifikationen zu formulieren. Es ist mittlerweile unumstritten, dass strukturelle Kriterien allein für die Testfallgenerierung nicht ausreichend sind. Zusätzlich mussten wir von Hand den generierten CLP-Code optimieren, um aufgrund der hohen Komplexität des Testmodells Testfälle überhaupt effizient generieren zu können. Wie alle anderen bekannten Technologien zur Testfallgenerierung ist auch unser Ansatz noch keine so genannte "Push-Button"-Technologie.

Unter diesem Hintergrund ist Automatisierung dennoch hilfreich, wenn wir Änderungen am Testmodell mit einbeziehen müssen. Unter der Voraussetzung, dass Testfallgenerierung den Stand einer "Push-Button"-Technologie erreicht hat, ist es natürlich wesentlich einfacher neue Testfälle zu generieren als diese wiederholt von Hand zu erstellen. Es ist möglicherweise einfach, 100 Testfälle in wenigen Stunden abzuleiten, aber dies wird wesentlich schwieriger für 1'000 Testfälle. Wenn diese erneut abgeleitet werden müssen, dann ist Automatisierung sicherlich hilfreich. Trotzdem muss die Anzahl der Testfälle in Grenzen gehalten werden, denn sie müssen nicht nur ausgeführt werden, sondern auch ausgewertet werden. Dies gilt insbesondere, wenn eine Verhaltensabweichung festgestellt wurde, denn dann muss manuell die Ursache dieser Abweichung untersucht und ggf. klassifiziert werden. Dies kann schnell zu unnötigen Aufwand führen, wenn z.B. 100 Testfälle durch ähnliches Fehlverhalten den gleichen Fehler aufdecken. Zusätzlich kann die Dauer für die Ausführung eines Testfalls die Anzahl der durchführbaren Tests einschränken. Zum Beispiel benötigt in dieser Studie aufgrund von Einschränkungen in der Hardware die "Software-in-the-loop"-Simulation des eingebetteten Systems 10 Sekunden pro Testfall. Weiterhin stellten wir fest, dass vollständig per Zufall erzeugte Testfälle schwer auszuwerten sind, da das erzeugte Verhalten von einem Entwickler schwer nachvollziehbar ist, weil es sehr stark von Standardverhalten abweicht.

Das Zählen von Fehlern von reaktiven Systemen ist nichttrivial. Wenn in unserer Studie zu einen bestimmten Zeitpunkt eine Abweichung zwischen Modell- und Implementierungsverhalten eintrat, dann unterschied es sich meistens auch für den Rest des Testfalls. Wir versuchten so viele Fehler wie möglich einen Testlauf zuzuordnen, sind aber manchmal im Zweifel: in unseren Statistiken deckten die meisten Testfälle nicht mehr als einen Fehler auf.

Es ist schwierig Rückschlüsse aus der moderaten Korrelation von Modell- und Implementierungsabdeckung zu ziehen. Der Einsatz von Abdeckungskriterien als Testfallspezifikation für die automatische Testfallgenerierung basiert auf der Annahme, dass eine hohe Abdeckung eine hohe Fehleraufdeckungsrate erzielt. In Bezug auf die anhaltenden Diskussionen über dieses Thema zeigen unsere Ergebnisse, dass die Ergebnisse bezüglich der Implementierungsabdeckung nicht direkt auf die Modellabdeckung übertragen werden können. (Zusätzliche Bemerkung: Modellabdeckung, wie wir sie definieren, hängt definitiv von dem zugrunde liegende Codegenerator ab.)

Insgesamt ist die Verallgemeinerung der Ergebnisse mit Vorsicht zu betreiben. Wenn wir von unterschiedlichen Gruppen entwickelte Testsuiten vergleichen, wie es bei uns mit den Testsuiten A bis G der Fall ist, müssen wir einbeziehen, dass verschiedene Personen in unterschiedlichen Umfeld mit unterschiedlichen Wissenstand über das System diese entwickelten. Obwohl wir es für möglich halten unsere Ergebnisse auf andere eingebettete Systeme mit geringem Anteil an geordneten Datentypen zu übertragen, können wir nicht entscheiden, ob ähnliches auch für Geschäftsanwendungen gilt. Ferner wissen wir nicht, ob sich unsere Erkenntnisse für Systeme verallgemeinern lassen, die sich in einem ausgereifteren Zustand befinden als unsere Betaversion des NetworkMasters. Abschließend können wir nicht entscheiden, ob andere Abdeckungsmaße insbesondere solche, die auf der Abdeckung des Datenflusses beruhen, ähnliche Charakteristiken zeigen. Im Bezug auf die Testfallgenerierungstechnologie glauben wir nicht, dass sich unser Ansatz wesentlich von anderen unterscheidet (siehe folgenden Abschnitt).

7.5.7. Verwandte Arbeiten

Ansätze der Testfallgenerierung auf Basis von strukturellen Kriterien mit Hilfe von Modelcheckern oder symbolischer Ausführung wurden von einer Reihe von Autoren sowohl für Modelle als auch für Implementierungen vorgeschlagen [HLSU02, Pre03a, AB99, HGW04, PPS+03]. Auf AutoFocus-Modelle können auch Modelchecking-Techniken wie SMV [McM92] oder SATO [Zha97] angewendet werden. Diese waren jedoch wegen den rekursiven Datenstrukturen in unserem Testmodell nicht anwendbar. Die hohe Komplexität des Testmodells verhinderte die Anwendung der Technologie für MC/DC [Pre03a]. Wir vermuten, dass auch bei Einschränkung der maximalen Tiefe der rekursiven Datenstrukturen ein Modelchecker mit der Komplexität des Testmodells überfordert wäre. Für einen Überblick auf andere modellbasierte Testfallgenerierungstechnologien verweisen wir auf [BJK+05, PSAK04].

In unserer Arbeit verwenden wir ein Abdeckungskriterium zum Messen der Testsuiten und nicht zu deren Generierung. Wir verwenden stattdessen die zufällige Generierung bzw. zufällige Auswahl von Testfällen [DN84, Gut99, HT90], die früher oder später in vielen Testfallgeneratoren verwendet wird und die auch implizit durch die unterschiedlichen Suchstrategien von Modelcheckern vorhanden ist. Wir dagegen schränken zusätzlich den Suchraum mit Hilfe von Testfallspezifikationen ein. Mit anderen Worten: wir generieren zufällig Testfälle für Teile des Testmodells und diese Teile entsprechen den wesentlichen Verhaltensweisen des zu testenden Systems. In diesem Sinne kombinieren wir funktionales Testen mit stochastischen Testen (vgl. Abschnitt 2.2.2).

Heimdahl u.a. [HGW04] stellte kürzlich fest, dass Testfälle, die auf Basis von symbolischen Modelcheckern und einem Abdeckungskriterium generiert wurden, im Bezug auf die Fehleraufdeckungsrate mit Vorsicht benutzt werden müssen. Viele bekannte Studien [FI98, FW93, HFGO94, Nta84, GW86] befassen sich mit der Fehleraufdeckungsrate von Abdeckungskriterien. Insgesamt sind diese jedoch nicht besonders aussagekräftig: Hutchins u.a. [HFGO94] verwenden Testsuiten, die sie von Hand mit der “category

partition"-Methode ableiten und anschließend erweitern, um ihre Abdeckung zu erhöhen. Die anderen Studien verwenden zufällig generierte Tests und verzichten auf die Fähigkeiten von menschlichen Entwicklern, Testfälle auszuwählen. All diese Studien untersuchen nicht den Zusammenhang zwischen automatisch generierten und von Hand erstellten Testfällen. Ihr Schwerpunkt liegt stattdessen auf den Vergleich der Tests anhand ihrer Abdeckung. Die Studien von Ntafos [Nta84] und von Hutchins u.a. [HFGO94] basieren auf Mutationstesten mit den zugehörigen kritischen Annahmen. Ebenso wie Frankl [FI98, FW93] verwenden wir nicht Mutationsanalysen, um die Effektivität der Tests zu messen, sondern die aufdeckte Anzahl von Fehlern.

Außer den Arbeiten von Heimdahl [HGW04] und Weyuker u.a. [WGS94] gibt es zumindest uns keine anderen bekannten Arbeiten, die die Beziehung zwischen der Modell- bzw. Spezifikationsabdeckungen und der Fehleraufdeckungsrate untersuchen.

7.6. Zusammenfassung

Der Zweck dieses Kapitels ist es, die Konzepte zur Entwicklung von Testmodellen und die Methodik des modellbasierten Testens auf eine industrielle Fallstudie anzuwenden und zu demonstrieren. Wir fassen das Wesentliche wie folgt zusammen:

- Wir zeigen am industriellen Beispiel wie die Abstraktionsform funktionale Abstraktion bei Planung der Modellentwicklung, die Formen Datenabstraktion, Kommunikationsabstraktion und temporale Abstraktionen bei der Bildung der Schnittstellen des Testmodells und bei dem inkrementellen Aufbau des Modellverhaltens ein Teil der funktionalen Abstraktion schrittweise reduziert wird.
- Wir demonstrieren wie zur Realisierung eines Inkrements des Testmodells, d.h. zur Integration einer weiteren Anforderung an das Modellverhalten Operationen zur *Vorbereitung einer Verhaltenserweiterung*, zur *Korrektur von Modellverhalten* und zur *Verhaltenserweiterung* auf Regelsystemen eingesetzt werden. Dieses Vorgehen unterstützt ein schrittweises und kontrolliertes Annähern an das Sollverhalten des zu testenden Systems.
- Weiterhin führt dieses Vorgehen zu einer gründlichen Vervollständigung der Anforderungs- bzw. Spezifikationsdokumente des betrachteten Systems. Bei unserer Fallstudie führte dies zur Aufdeckung von 3 groben Spezifikationsfehlern und zur Schließung von 7 größeren Spezifikationslücken. Da Testmodelle als abstrakte Prototypen angesehen werden können, bestätigt dies die weithin bekannte Erfahrung, dass Prototypen zur Klärung und Vervollständigung von Anforderungen hilfreich sind [BGS84]; jedoch mit dem großen Unterschied, dass Prototypen im Gegensatz zu Testmodellen im allgemeinen nicht weiterverwendet werden. Testmodelle hingegen dienen als Quelle zur Ableitung von modellbasierten Testfällen und ermöglichen damit die direkte Verifikation einer Implementierung gegen ihre Spezifikation.
- Auf Grundlage des entwickelten Testmodells, der Entwicklung von funktionalen

Testfallspezifikationen leiteten wir automatisch mit und ohne Testfallspezifikationen und von Hand Testfälle ab. Wir verglichen ihre Güte mit traditionell erstellten Testfällen, indem wir insgesamt sieben verschiedene Testsuitearten mit unterschiedlichen Ursprung anhand ihrer Fehleraufdeckungsrate in der Implementierung, ihrer Bedingungs-/Entscheidungsüberdeckung auf dem Modell und der Implementierung untersuchten. Die Ergebnisse des Vergleichs sind die Folgenden:

- Testsuiten, die ohne Einsatz von Modellen erzeugt wurden, decken weniger Fehler auf als modellbasierten Testsuiten.
- Programmierfehler werden unabhängig vom Modelleinsatz aufgedeckt. Modellbasierte Testsuiten decken bis zu sechsmal so viele Lastenheftfehler auf als nicht modellbasierte.
- Modellbasierte automatisch erzeugte Testsuiten finden nicht wesentlich mehr Fehler als modellbasierte manuell erzeugte Testsuiten, benötigen aber eine größere Anzahl von Testfällen bei gleicher Aufdeckungsrate.
- Wir maßen die Bedingungs-/Entscheidungsüberdeckung der Testsuiten sowohl auf dem Modell als auch auf der Implementierung und konnten lediglich eine moderate Korrelation feststellen.
- Sowohl die Abdeckung auf dem Modell als auch die auf der Implementierung zeigte eine deutliche Korrelation zur Fehleraufdeckungsrate, obwohl eine höhere Abdeckung nicht immer zu einer höheren Fehleraufdeckungsrate führte.

8. Zusammenfassung und Ausblick

In diesem abschließenden Kapitel fassen wir die Ergebnisse und Beiträge der vorliegenden Arbeit zusammen und identifizieren Problemstellungen und Herausforderungen für weiterführende Aufgabenstellungen.

8.1. Ergebnisse

Die Kernergebnisse dieser Arbeit sind

- die Bereitstellung eines inkrementellen Entwicklungsprozess für die Erstellung von Testmodellen,
- die Identifikation von geeigneten Abstraktionsprinzipien zur Bildung von Testmodellen,
- ein Transformationsverfahren zum Refactoring von Verhaltensmodellen,
- die exemplarische Anwendung der Techniken an einer industriellen Fallstudie und
- die Evaluation der Qualität von modellbasierten Tests gegenüber traditionell entwickelten Tests anhand der Fallstudie.

In den folgenden Absätzen wiederholen wir die wesentlichen Kernpunkte zu den Ergebnissen und verweisen auf die entsprechenden Abschnitte in der Arbeit.

Inkrementeller Entwicklungsprozess Testmodelle erreichen trotz ihrer Vereinfachungen und Abstraktionen eine hohe Komplexität. Ferner wird die Erstellung von Testmodellen als eine Aktivität im Bereich der Anforderungsanalyse angesehen. Die Grundlage für die Erstellung sind entsprechend meist informelle, unvollständige und manchmal in Teilen widersprüchliche Anforderungs- und Spezifikationsdokumente. Diese beiden Beobachtungen erfordern einen Entwicklungsprozess zur Erstellung von Testmodellen, der einerseits die intellektuelle Beherrschbarkeit der Komplexität von Testmodellen und andererseits die Korrektheit gegenüber den Anforderungs- und Spezifikationsdokumenten sichert. Wir schlagen deshalb einen inkrementellen Entwicklungsprozess zur Erstellung von Testmodellen vor (vgl. Kapitel 5). Dieser vereint folgende Eigenschaften:

- Im Prozess werden der Reihe nach die funktionalen Anforderungen an das zu testende System in einem ausführbaren Verhaltensmodell integriert. Dies führt zu einer gründlichen Analyse der Anforderungen und zu einer Aufdeckung der Lücken,

Mehrdeutigkeiten und Widersprüche in den Anforderungs- bzw. Spezifikationsdokumenten.

- Die intellektuelle Beherrschbarkeit und Korrektheit des Modells wird gewährleistet, indem während dem Entwicklungsprozess entsprechende existenzielle Eigenschaften über dem Modellverhalten erhalten werden. Diese werden nach jeder Inkrementbildung mit Hilfe von Regressionstests überprüft.
- Der Prozess bietet eine ausreichende Flexibilität, um den Lernprozess während der Modellentwicklung zu unterstützen. Der Prozess fordert lediglich den Erhalt von ausgewählten existenziellen Eigenschaften an das Modellverhalten. Diese Eigenschaften entsprechen den bereits untersuchten und verstandenen Anforderungen. Das übrige Modellverhalten darf beliebig angepasst, verändert und erweitert werden, so dass Erkenntnisse, die im Verlauf der Modellentwicklung entstehen, flexibel in das Modell eingearbeitet werden können.
- Der Prozess wurde durch seine Struktur und Einfachheit so angelegt, dass Werkzeugunterstützung realisierbar ist und dass die Wahrscheinlichkeit erhöht wird, von Entwicklern richtig angewendet zu werden.

Insgesamt dient die Testmodellbildung analog zum Rapid-Prototyping, die Eindeutigkeit und die Vervollständigung von Anforderungen bzw. der Spezifikation zu erreichen. Im nächsten Schritt bildet das Modell die Grundlage für das modellbasierte Testen und damit ein leistungsfähiges Mittel, direkt die Konformität zwischen der Spezifikation und seiner Implementierung nachzuweisen bzw. zu widerlegen.

Abstraktion bei der Testmodellbildung Im Allgemeinen sind Abstraktionen bei der Modellbildung zweckgebunden und domänenspezifisch. Wir unterscheiden in dieser Arbeit zwei Hauptklassen von Abstraktionen: *Kapselung von Details* und *Weglassen von Details* (vgl. Abschnitt 3.1). Bei der Kapselung von Details liegt ein Art von Makromechanismus zugrunde, der das automatische und vollständige Hinzufügen der gekapselten Information erlaubt. Bei dem Weglassen von Details fehlt substanzielle Information im Modell, so dass ein automatisches Hinzufügen der fehlenden Information nicht oder nur im geringen Maße möglich ist. Speziell für die Bildung von Testmodellen unterscheiden wir die folgenden fünf Abstraktionsprinzipien, die sich sowohl zum Kapseln als auch zum Weglassen von Details eingesetzt werden (vgl. Abschnitt 3.2):

- Funktionale Abstraktion wird eingesetzt, um die Funktionalität des zu testenden System im Testmodell zu reduzieren oder Teile der Funktionalität im Testmodell zu kapseln.
- Datenabstraktion dient zur Reduktion der Datenkomplexität im Testmodell, indem konkrete Datentypen äquivalent repräsentiert oder durch Äquivalenzklassenbildung reduziert werden.
- Kommunikationsabstraktion fasst komplexe konkrete Interaktionen zu einer ato-

maren Operation auf Testmodellebene zusammen.

- Temporale Abstraktion führt entweder im Testmodell ein gröberes Zeitmodell ein oder abstrahiert von physikalischen Zeitdauern.

- Strukturelle Abstraktion dient zur Restrukturierung des Modells gegenüber des zu testenden Systems.

Im modellbasierten Testprozess wird die Abstraktionslücke zwischen Testmodell und dem zu testenden System soweit wie nötig bzw. möglich durch den Testtreiber überbrückt. Durch die Abstraktion des Testmodells wird eine Verteilung der Komplexität auf das Testmodell und dem Testtreiber erreicht. Die dadurch erreichten Vereinfachungen im Testmodell erleichtern die Analysierbarkeit und die Prüfung der Validität gegenüber seiner Anforderungen. Ferner tragen sie wesentlich zur intellektuellen Beherrschbarkeit und zum Vertrauen in die Korrektheit des Testmodells bei.

Transformation von Verhaltensmodellen Bei der inkrementellen Entwicklung von Modellen bilden Refactoringoperationen eine wichtige Klasse. Diese Operationen verändern nicht das beobachtbare Modellverhalten und dienen zur Verbesserung der Modellstruktur, zur Vorbereitung einer Modellerweiterung oder zur Restrukturierung des Modells, um bestimmte Aspekte gezielt analysieren zu können. In Kapitel 6 wurde ein automatisierbares Refactoringverfahren vorgestellt, das auf der Transformation von Verhaltensmodellen beruht. Die Idee des Verfahrens ist, den Datenraum eines Verhaltensmodells in Äquivalenzklassen zu partitionieren, wobei jeder Partition einem Kontrollzustand des Verhaltensmodells entspricht. Die Übergänge zwischen den Kontrollzuständen werden berechnet, indem je ein Übergang im ursprünglichen Verhaltensmodell durch eine Menge von Übergängen zwischen allen Kontrollzuständen ersetzt wird und danach die Übergänge mit unerfüllbaren Vorbedingungen entfernt werden. Das Transformationverfahren kann sowohl zur Überführung einer Tabelle in unterschiedliche Kontrollzustandsgraphen als auch zur Restrukturierung von Kontrollzustandsgraphen verwendet werden. Das Verfahren ist methodisch in zweierlei Hinsicht einsetzbar:

- Es dient zur Bildung von verschiedenen abstrahierten Sichten auf ein Verhaltensmodell, indem unterschiedliche Partitionierung des Datenzustandsraums zur Transformation des Modells gewählt werden. Dadurch können für Reviews gewünschte Aspekte des Verhaltensmodells betont bzw. andere ausgeblendet werden, so dass diese gezielt und effizient untersucht bzw. überprüft werden können.

- Es dient zum Refactoring von Kontrollzustandsgraphen, um deren Weiterentwicklung zu erleichtern bzw. erst zu ermöglichen. Dazu wird eine geeignete Kontrollzustandspartitionierung gewählt und der Datenraum entsprechend partitioniert. Im Anschluss kann das Verhalten des Modells gezielt und lokal modifiziert werden, indem lediglich die Übergänge bei bestimmten Kontrollzuständen verändert werden.

Anwendung auf eine industrielle Fallstudie Wir wiesen die Praktikabilität unserer Ansätze anhand einer industriellen Fallstudie nach (vgl. Kapitel 7). Das zu testende System der Studie ist ein Netzwerkcontroller des Infotainmentnetzwerks MOST in Automobilen. Bei der inkrementellen Entwicklung gemäß unseres Ansatzes wurden insgesamt drei grobe Spezifikationsfehler, sieben größere Spezifikationslücken sowie viele kleinere Mehrdeutigkeiten in den Spezifikationsdokumenten aufgedeckt und behoben. Dies untermauert unsere früheren Erfahrungen, dass systematische Modellbildung einen großen Beitrag zur Verbesserung von Spezifikationsdokumenten leistet. Weiterhin wendeten wir das Verfahren zur Transformation von Verhaltensmodellen auf die Fallstudie an und bestätigten den Nutzen des Verfahrens für den Review von Verhaltensmodellen (vgl. Abschnitt 7.4). Durch die Bildung unterschiedlicher abstrakter Sichten auf das Modell werden verschiedene Symmetrien und zuvor verborgene Zusammenhänge im Modell aufgezeigt. Diese helfen einerseits, überflüssige oder fehlende Übergänge zu finden, und führen andererseits zu neuen Erkenntnissen über das Modell.

Modellbasierte im Vergleich zu traditionell entwickelten Tests Im Rahmen der Fallstudie des MOST-Netzwerkcontrollers verglichen wir modellbasierte Tests mit traditionell entwickelten Tests durch die Bildung von sieben Testsuites unterschiedlichen Ursprungs. Die Vergleichskriterien sind ihre Entscheidungs-/Bedingungsüberdeckung und ihre Fehleraufdeckungsrate. Die wesentlichen Ergebnisse der Untersuchung sind:

- Tests, die ohne Testmodell abgeleitet wurden, decken weniger Fehler auf als modellbasierte Tests. Die Anzahl der aufgedeckten Programmierfehler ist in etwa gleich, aber die Anzahl der aufdeckten Lastenheftfehler ist höher.
- Automatisch generierte modellbasierte Tests decken genauso viele Fehler auf wie von Hand erstellte modellbasierte Tests bei etwa gleicher Anzahl von Tests. Eine sechsfach höhere Anzahl von automatisch generierten Tests führt zu einer 11% höheren Fehleraufdeckungsrate. Keine der Testsuiten deckte alle Fehler auf. Ferner erreichen von Hand erstellte modellbasierte Tests eine höhere Abdeckung auf dem Modell als die automatisch generierten.
- Es existiert eine moderate Korrelation zwischen der Entscheidungs-/Bedingungsüberdeckung auf dem Modell und der Implementierung.
- Es existiert eine moderate Korrelation zwischen der Entscheidungs-/Bedingungsüberdeckung auf der Implementierung und der Fehleraufdeckungsrate und eine starke Korrelation zwischen der Entscheidungs-/Bedingungsüberdeckung auf dem Modell und der Fehleraufdeckungsrate. Außerdem stellten wir fest, dass eine höhere Abdeckung nicht zwangsweise zu einer höheren Fehleraufdeckungsrate führt.

Insgesamt ziehen wir drei Schlüsse aus der Untersuchung: Erstens, bezüglich der Fehleraufdeckungsrate zahlt sich die Nutzung von Modellen aus. Zweitens, obwohl in einigen Domänen eine starke Korrelation zwischen Entscheidungs-/Bedingungsüberdeckung auf der *Implementierung* und der Fehleraufdeckungsrate gefunden wurde, bedeutet das nicht

zwangsweise, dass sich diese positiven Ergebnisse übertragen lassen, wenn die gleichen Kriterien zur automatischen Testfallgenerierung aus *Modellen* verwendet werden. Drittens, falls die Anzahl der aufgeführten Tests eine Rolle spielt, dann muss der Nutzen von automatischer Testfallgenerierung erst noch gezeigt werden.

8.2. Zukünftige Arbeiten

In diesem Abschnitt identifizieren und beschreiben wir weiterführende Aufgabenfelder, die sich zur Fortführung dieser Arbeit anbieten. Sie befassen sich mit der Entwicklung von Werkzeugunterstützung und weiterer Evaluierung der erarbeiteten Konzepte, mit weiteren Studien zum modellbasierten Testen, mit dem modellbasierten Testen von nichtdeterministischen Systemen und mit der Vision modellbasiertes Testen für verteilte Systeme zu nutzen.

Werkzeugunterstützung und weitere Evaluierung Ein Aufgabe ist in Zukunft, für die erarbeiteten Konzepte Werkzeugunterstützung zu entwickeln und ihren Nutzen im Vergleich zu anderen Techniken mit Hilfe von weiteren Fallstudien zu evaluieren. Die dabei gewonnenen Erkenntnisse können dann wiederum zur Verbesserung der Techniken selbst beitragen.

Als erster Schritt ist geplant, das Transformationsverfahren aus Kapitel 6 zum Refactoring von Verhaltensmodellen vollständig in das Werkzeug AutoFocus zu integrieren. Nach den erfolgreichen Experimenten, die Programmiersprache Curry [Han04] zur Prüfung der Erfüllbarkeit von generierten Vorbedingungen zu verwenden, soll in Bälde eine automatische Überführung von AutoFocus-STDs nach Curry und eine entsprechende Rückführung der Berechnungsresultate nach AutoFocus realisiert werden (vgl. Abschnitt 6.3). Im Anschluss ist eine gründliche Evaluierung des Verfahrens mit Modellen von praxisrelevanter Komplexität erforderlich, um eine solide Abschätzung der Stärken und Schwächen des Transformationsverfahrens geben zu können.

Eine weitere Aufgabenstellung ist, die erarbeiteten Grundlagen aus Kapitel 5 zu nutzen, um Werkzeugunterstützung für den inkrementellen Entwicklungsprozess zur Erstellung von Testmodellen bereitzustellen. Hier ist die Entwicklung eines Wizards möglich, der dem Entwickler bei den fünf Phasen der Inkrementbildung unterstützt. Dieser gibt Unterstützung bei der Spezifikation von zu realisierenden existenziellen Modelleigenschaften, bei der Erstellung oder Generierung von Regressionstestsuiten und bei der automatischen Prüfung von existenziellen Eigenschaften bzw. der automatischen Durchführung von Regressionstests. Werkzeugunterstützung wäre auch bei der Ursachenlokalisation hilfreich, wenn eine Regressionstestsuite scheitert. Zum Beispiel wird beim Scheitern eines Regressionstests durch das Werkzeug der aktuelle Daten- und Kontrollzustand sowie der verursachende Übergang ausgegeben, um das Auffinden der Fehlerursache zu erleichtern. Wenn eine ausreichende Werkzeugunterstützung existiert, ist eine gründliche Untersuchung nötig, um seinen Nutzen z.B. gegenüber unstrukturierter Entwicklung zu

untersuchen. Ein mögliches Szenario wäre zum Beispiel, mehrere Teams von Studenten für ein System ein Testmodell mit und ohne Werkzeugunterstützung für die inkrementelle Entwicklung erstellen zu lassen. Dabei werden die Aufwände der einzelnen Teams und die aufdeckten Schwächen in den Spezifikationsdokumenten erfasst. Am Ende der Entwicklung wird die Qualität der Modelle mit Hilfe einer Referenztestsuite ermittelt. Die Auswertung dieses Experiments gäbe erste Anhaltspunkte über die Stärken und Schwächen beim Verwenden der Werkzeugunterstützung.

Weitere Studien zum modellbasierten Testen Die industrielle Fallstudie aus Kapitel 7 und die zugehörige Studie in Abschnitt 7.5 zeigen den Nutzen von ausführbaren Testmodellen für die Anforderungsanalyse und Testaktivitäten in Hinblick auf aufgedeckte Mängel und Fehler in den Spezifikationsdokumenten bzw. Implementierungen.

Weder in dieser Studie noch in einer von uns bekannten wurde die Wirtschaftlichkeit von modellbasierten Testen untersucht. Dazu wird ein Kostenmodell benötigt, das einem Fehler die verursachenden Kosten zuordnet. Im Kontext unserer Fallstudie, bei der sich viele Geräte im Feld befinden und damit hohe Wartungskosten einhergehen, würden wir erwarten, dass sich die Investition, Modelle zu erstellen, auszahlt. Momentan gibt es aber keine Zahlen, die diese Behauptung stützen.

Weiterhin wurde modellbasiertes Testen bisher nicht mit anderen Qualitätssicherungsmaßnahmen wie Reviews oder Inspektionen gründlich verglichen. Bei diesen Techniken hat sich herausgestellt, dass diese am effektivsten in Bezug auf Gesamtzahl der aufgedeckten Fehler und der Kosten beim Aufdecken und Beheben der Fehler sind [Fag02, Fag76]. Zusätzlich ist weitläufig akzeptiert, dass die Kombination von Qualitätssicherungsmaßnahmen die besten Ergebnisse liefert [Mye78].

Zusammenfassend identifizieren wir den Bedarf an weiteren Studien, die offene Fragen im Bereich des modellbasierten Testens aufgreifen:

- Es sollten weitere Studien ähnlich zu der aus Abschnitt 7.5 folgen, um zu untersuchen, ob sich die gefundenen Resultate verallgemeinern lassen oder nicht.
- Weitere Studien sollten mögliche Kostenmodelle miteinschließen. Obwohl wir überzeugt sind, dass sich die Erstellung von Modellen auszahlt, muss dies erst durch belastbare Zahlen nachgewiesen werden.
- Ferner könnten weitere Studien den Vergleich zwischen den Nutzen der Erstellung von ausführbaren Modellen inklusive zugehöriger Testfallgenerierung und Techniken wie Reviews und Inspektionen miteinbeziehen.
- Ein ambitioniertes Projekt wäre, eine Studie durchzuführen, bei der verschiedene Modellierungssprachen und verschiedene Testfallgeneratoren mit unterschiedlichen Generierungstechniken und Testfallspezifikationen für den modellbasierten Test ein und desselben Systems eingesetzt werden. Dies würde einen umfassenden Vergleich zulassen, welche Techniken in welchem Kontext ihre Vor- und Nachteile haben.

Modellbasiertes Testen nichtdeterministischer Systeme In dieser Arbeit geben wir eine Methodik an, deterministische Testmodelle systematisch zu entwickeln. Deterministische Modelle sind einerseits intellektuell wesentlich leichter beherrschbar und andererseits sind zugehörige Testfallgenerierungstechnologien bereits vorhanden. Deterministische Testmodelle können zum Test von nichtdeterministischen Systemen eingesetzt werden, indem geeignete Schnittstellen für das Testmodell gewählt werden oder der Nichtdeterminismus geeignet zu Determinismus im Modell abstrahiert wird. Der Preis für die erreichte Vereinfachung im Testmodell ist in der Regel, dass das Modell gegenüber dem zu testenden System überspezifiziert ist und dementsprechend der Testtreiber zur Ausführung der Testfälle komplexer wird. Dieser muss die Überspezifikation in den Testfällen in eine geeignete Entscheidungslogik überführen, so dass das nichtdeterministische Verhalten des Systems gegen den deterministischen Testfall verglichen werden kann. Welche Arten von Nichtdeterminismus und inwieweit dieser auf Testmodellebene sinnvoll abstrahiert werden kann, wurde bisher noch nicht eingehend untersucht. Bisher existieren lediglich Fallstudien [PPW+05], bei denen bewusst im Modell gegenüber der Realität überspezifiziert und die Überspezifikation durch den Testtreiber ausgeglichen wurde.

Ein anderer Weg nichtdeterministische Systeme zu testen, ist direkt nichtdeterministische Testmodelle und zugehörige Testfallgenerierung zu verwenden. Dies führt zu einer wesentlich höheren Komplexität bei der Testfallgenerierung und bei der Prüfung der Validität des Testmodells; erlaubt jedoch den Test einer größeren Systemklasse. Ein Testfall codiert in diesem Fall nicht einen Ablauf, sondern eine Menge von möglichen Abläufen und enthält eine Entscheidungslogik, wie auf nichtdeterministisches Verhalten des zu testenden Systems reagiert werden muss, um ein Testziel zu erreichen. Bisher sind wenige Ansätze zum modellbasierten Testen nichtdeterministischer Systeme und zugehöriger Testfallgenerierung bekannt. In der Fallstudie einer Chipkarte [CJRZ01] wird zum Beispiel für je ein Testziel einen Testfall in Form eines Ein-/Ausgabeautomaten generiert, der bei Ausführung mit dem zu testenden System prüft, ob sein Verhalten das Testziel erfüllt oder nicht.

Modellbasiertes Testen von verteilten Systemen Die bisher bekannten Fallstudien [PPW+05, DBG01, SA99, FKL99, PPS+03, CJRZ01, KVZ98, BFV+99, FHP02] aus dem industriellen Umfeld modellieren und testen lediglich monolithische Systeme oder Teilsysteme eines Gesamtsystems. Eine Vision für die Zukunft wäre, das logische Kommunikationsverhalten verteilter Systeme auf abstrakter Ebene mit vernetzten Verhaltensmodellen vollständig zu modellieren und diese zum modellbasierten Testen zu verwenden. Die Modellbildung des verteilten Gesamtsystems würde in den frühen Phasen ein umfassendes Verständnis und Analyse des Verhaltens sowie die Integration des Systems auf konzeptueller Ebene unterstützen. Teilmodelle werden dann zum Test entsprechender Teilsysteme und komponierte Teilmodelle zum Integrationstest entsprechender Systemteile verwendet. Ein weiteres Anwendungsszenario dieses Ansatzes ist bei einem bevorstehenden Austausch eines Teilsystems einen Kompatibilitätstest der Austausch-

komponente mit Hilfe des entsprechenden Teilmodells vorzunehmen. Auf diese Weise könnte ohne hohen Aufwand die Kompatibilität des Austauschsystems vor seiner Integration überprüft werden.

A. Beweise

A.1. Beweise zu Kapitel 4

Beweisskizze zu Proposition 4.1, Seite 50. Es ist zu zeigen, dass das Modellverhalten R_Z der Zustandsmaschine $Z = (I, O, L, \mathit{Init}, T)$ die drei Eigenschaften (4.1), (4.2) und (4.3) aus Definition 4.5 erfüllt:
zu (4.1) Die Eigenschaft (4.1) folgt direkt aus der Definition von R_Z, da für alle $(i, o) \in R_Z$ ein Strom von Belegungen $\eta \in \langle\!\langle Z \rangle\!\rangle$ mit $i.\iota.n = \eta.n.\iota \wedge o.\theta.n = \eta.n.\theta$ ($\iota \in I$, $\theta \in O$, $n \in \mathsf{dom}.\eta$) existiert und damit $\#\eta = \#i = \#o$ gilt.
zu (4.2) Mit den Eigenschaften des Prädikats *Init* (4.6) und der Transitionsmengen T (4.7) der Zustandsmaschine Z ergibt sich direkt mit Induktion über n die allgemeinere Eigenschaft:
$\forall i, \tilde{i} \in \mathsf{in}.Z, n < max(\#i, \#\tilde{i})$ mit $i\downarrow_n = \tilde{i}\downarrow_n$ gilt

$$\{\langle \eta.1, \ldots, \eta.(n+1)\rangle \mid \eta \in \langle\!\langle Z \rangle\!\rangle \wedge i\downarrow_n = \langle \eta.1|_I, \ldots, \eta.n|_I\rangle\} = \{\langle \eta.1, \ldots, \eta.(n+1)\rangle \mid \eta \in \langle\!\langle Z \rangle\!\rangle \wedge \tilde{i}\downarrow_n = \langle \eta.1|_I, \ldots, \eta.n|_I\rangle\}.$$

Und damit gilt durch Projektion die Spezialisierung $\{o\downarrow_{n+1} \mid (i, o) \in R_Z\} = \{\tilde{o}\downarrow_{n+1} \mid (\tilde{i}, \tilde{o}) \in R_Z\}$.
zu (4.3) Die Eigenschaft (4.3) folgt direkt aus der Definition von R_Z und der Eigenschaft der Ablaufmenge $\langle\!\langle Z \rangle\!\rangle$, dass diese für alle Abläufe $\eta \in \langle\!\langle Z \rangle\!\rangle$ auch die Teilabläufe $\tilde{\eta} \sqsubseteq \eta$ enthält. □

Beweisskizze zu Proposition 4.2, Seite 50. Sei $Z = (I, O, L, \mathit{Init}, T)$ eine Zustandsmaschine mit Eigenschaften (4.9) und (4.10). Es genügt zu zeigen, dass jede Eingabe $i \in \mathsf{in}.Z$ zu einem eindeutigen Ablauf $\eta \in \langle\!\langle Z \rangle\!\rangle$ führt, d.h.

$$\forall \eta, \tilde{\eta} \in \langle\!\langle Z \rangle\!\rangle,\ 1 \leq k \leq \#i,\ \iota \in I \text{ mit } \eta.k.\iota = i.\iota.k \text{ gilt } \eta = \tilde{\eta}$$

Induktion über $1 \leq k \leq \#i$:
$k = 1$: Es gilt $\eta.1.\iota = i.\iota.1 = \tilde{\eta}.1.\iota$ ($\iota \in I$) und wegen (4.9) gilt $\eta.1 \overset{L\cup O}{=} \tilde{\eta}.1$. Also gilt insgesamt

$$\eta.1 = \tilde{\eta}.1.$$

$k \to k+1$: Sei $\eta.k = \tilde{\eta}.k$ für ein k bereits bewiesen. Dann gilt wegen (4.10) $\eta.(k+1) \overset{L\cup O}{=} \tilde{\eta}.(k+1)$. Ferner gilt $\eta.(k+1).\iota = i.\iota.(k+1) = \tilde{\eta}.(k+1).\iota$ ($\iota \in I$). Also gilt insgesamt

$$\eta.(k+1) = \tilde{\eta}.(k+1).$$

Sei nun $Z = (I, O, L, Init, T)$ eine Zustandsmaschine mit der Eigenschaft (4.11). Es ist zu zeigen, dass für jede Eingabe $i \in \vec{I}$ mindestens ein Ablauf $\eta \in \langle\langle Z \rangle\rangle$ existiert, d.h.

$$\forall i \in \vec{I} \exists \eta \in \langle\langle Z \rangle\rangle \text{ mit } \eta.k.\iota = i.\iota.k, \text{ wobei } 1 \leq k \leq \#i, \iota \in I \text{ gilt.}$$

Induktion über $1 \leq k \leq \#i$:
$k = 1$: Wähle $\eta.1 \in B_V$ mit $\eta.1 \models Init$ und $\eta.1.\iota = i.\iota.1$ $(\iota \in I)$. Dann existiert wegen (4.11) und (4.7) eine Belegung $\beta \in B_V$ mit $(\eta.1, \beta) \in T$ bzw. $\beta.\iota = i.\iota.2$ $(\iota \in I)$. Folglich definieren wir $\eta.2 := \beta$.
$k \to k+1$: Sei $\langle \eta.1, \ldots, \eta.k \rangle \in \langle\langle Z \rangle\rangle$ mit $\eta.k.\iota = i.\iota.k$ $(\iota \in I)$ für ein k bereits konstruiert. Dann konstruieren wir ein $\eta.(k+1)$ mit $(\eta.k, \eta.(k+1)) \in T$ und $\eta.(k+1).\iota = i.\iota.(k+1)$ $(\iota \in I)$, das wegen (4.11) bzw. (4.7) existiert. □

Beweisskizze zu Proposition 4.3, Seite 52. Die drei Eigenschaften (4.1), (4.2) und (4.3) aus Definition 4.5 gelten für die Modellverhalten R_S und R_T. Durch die Definition der Relation $R_{S \otimes T}$ (4.12) werden diese Eigenschaften direkt auf $R_{S \otimes T}$ übertragen. Die Relation $R_{S \otimes T}$ ist also ein Modellverhalten über Schnittstelle (I, O) gemäß Definition 4.5. □

Beweisskizze zu Proposition 4.4, Seite 52. Zunächst skizzieren wir, dass $(Y \otimes Z) = (I, O, L, Init, T)$ wohldefiniert ist, d.h. für $Init$ gilt die Eigenschaft (4.6) und für T die Eigenschaft (4.7):
Da $L \cup O \subset L_Y \cup O_Y \cup L_Z \cup O_Z$ gilt und die Eigenschaft (4.6) für $Init_Y$ und $Init_Z$ gilt, überträgt sich die Eigenschaft (4.6) auf $Init = Init_Y \wedge Init_Z$.
Analog wird die Eigenschaft (4.7) von T_Y und T_Z auf T übertragen, da die Eigenschaft (4.7) für T_Y bzw. T_Z gilt und $L_Y \cup I_Y \subset L \cup I$ bzw. $L_Y \cup O_Y \subset L \cup O$ und $L_Z \cup I_Z \subset L \cup I$ bzw. $L_Z \cup O_Z \subset L \cup O$ gilt.

Wir skizzieren nun, dass die Komposition von Zustandsmaschinen verträglich mit der Komposition auf Modellverhalten ist, d.h. $R_{(Y \otimes Z)} = R_Y \otimes R_Z$.
"$\subset$" Sei $(i, o) \in R_{(Y \otimes Z)}$. Dann ist zu zeigen, dass Schnittstellenverhalten $(i_Y, o_Y) \in R_Y$ und $(i_Z, o_Z) \in R_Z$ existieren, die gemäß Vorschrift (4.12) zu (i, o) komponieren. Es folgt die Konstruktion von $(i_Y, o_Y) \in R_Y$ bzw. $(i_Z, o_Z) \in R_Z$: nach Definition 4.9 existiert zu (i, o) ein Ablauf $\eta = \langle \alpha_1, \alpha_2, \ldots \rangle \in \langle\langle Y \otimes Z \rangle\rangle$. Für die Transitionen $(\alpha_k, \alpha_{k+1}) \in T$ existieren gemäß Vorschrift (4.13) Transitionen $(\alpha_{k,Y}, \alpha_{k+1,Y}) \in T_Y$ und $(\alpha_{k,Z}, \alpha_{k+1,Z}) \in T_Z$ der Zustandsmaschine Y bzw. Z, die zu (α_k, α_{k+1}) komponieren. Folglich existieren die Abläufe $\eta_Y := \langle \alpha_{1,Y}, \alpha_{2,Y}, \ldots \rangle \in \langle\langle Y \rangle\rangle$ und $\eta_Z := \langle \alpha_{1,Z}, \alpha_{2,Z}, \ldots \rangle \in \langle\langle Z \rangle\rangle$. Die Projektion der Abläufe η_Y und η_Z auf ihr Schnittstellenverhalten (i_Y, o_Y) bzw. (i_Z, o_Z) leisten das Gewünschte.
"$\supset$" Sei nun $(i, o) \in R_Y \otimes R_Z$. Dann ist zu zeigen, dass ein Ablauf $\eta \in \langle\langle Y \otimes Z \rangle\rangle$ existiert, dessen Projektion auf das Schnittstellenverhalten (i, o) ist. Es folgt die Konstruktion von η: Zu (i, o) existieren nach Vorschrift (4.12) $(i_Y, o_Y) \in R_Y$ und $(i_Z, o_Z) \in R_Z$, die zu (i, o) komponieren. Nach Vorschrift (4.8) existieren Abläufe $\eta_Y = \langle \alpha_{1,Y}, \alpha_{2,Y}, \ldots \rangle \in \langle\langle Y \rangle\rangle$ bzw. $\eta_Z = \langle \alpha_{1,Z}, \alpha_{2,Z}, \ldots \rangle \in \langle\langle Z \rangle\rangle$. Die paarweise Komposition der Transitionen $(\alpha_{k,Y}, \alpha_{k+1,Y}) \in T_Y$ und $(\alpha_{k,Z}, \alpha_{k+1,Z}) \in T_Z$ zu $(\alpha_k, \alpha_{k+1}) \in T$ gemäß Vorschrift (4.13)

führt zu einem Ablauf $\eta := \langle \alpha_1, \alpha_2, \ldots \rangle \in \langle\!\langle Y \otimes Z \rangle\!\rangle$. Der Ablauf η leistet das Gewünschte. □

Beweisskizze zu Proposition 4.5, Seite 55. Wir skizzieren, dass die Definition von T_R gemäß Vorschrift (4.14) die Eigenschaft (4.7) aus Definition 4.7 erfüllt. Sei $(\alpha, \beta) \in T_R$. Da existiert gemäß Vorschrift (4.14) eine Regel $r \in R$ und eine Belegung $\tilde{\alpha} \in B_{loc_r}$ mit $\alpha \oplus \tilde{\alpha} \models \mathit{pre}_r \wedge \mathit{input}_r$ und $\alpha \oplus \tilde{\alpha}, \beta' \models \mathit{output}_r \wedge \mathit{assign}_r$. Seien zusätzlich $\gamma, \delta \in B_V$ mit $\gamma \overset{L \cup I}{=} \alpha$ und $\delta \overset{L \cup O}{=} \beta$. Dann gilt auch $\gamma \oplus \tilde{\alpha} \models \mathit{pre}_r \wedge \mathit{input}_r$, da weder pre_r noch input_r Variablen aus O enthalten. Analog gilt auch $\alpha \oplus \tilde{\alpha}, \delta' \models \mathit{output}_r \wedge \mathit{assign}_r$, da weder output_r noch assign_r Variablen aus I enthalten und es gilt $\gamma \oplus \tilde{\alpha}, \delta' \models \mathit{output}_r \wedge \mathit{assign}_r$, da auf den rechten Seiten von output_r und assign_r nur Variablen aus $L \cup \mathit{loc}_r$ auftreten. Insgesamt folgt also $(\gamma, \delta) \in T_R$. □

Beweisskizze zu Proposition 4.6, Seite 55 . Sei $G = (I, O, L, \mathit{Init}, R)$ ein Regelsystem mit den Eigenschaften (4.15) und (4.16). Aufgrund Proposition 4.2 genügt es zu zeigen, das die induzierte Transitionsrelation T_R aus Vorschrift (4.14) die Eigenschaft (4.10) hat. Sei also $(\alpha_1, \beta_1), (\alpha_2, \beta_2) \in T_R$ mit $\alpha_1 \overset{L \cup I}{=} \alpha_2$. Dann existiert eine Regel $r \in R$ mit $\alpha_1, \alpha_2 \in \mathsf{en}.r$, da die Vorbedingungen und die Eingabemuster der Regeln R nur von den Variablen L und I abhängen. Die Regel r ist sogar eindeutig bestimmt, da G die Eigenschaft (4.16) hat. Da die Ausgabe output_r und die Zuweisung assign_r einer Regel r die Form von Zuweisungen in Abhängigkeit von den Variablen $L \cup \mathit{loc}_r$ und damit indirekt von den Variablen $L \cup I$ besitzt, bildet eine Regel r in Abhängigkeit von der Belegung der Variablen $L \cup I$ eindeutig auf eine Belegung der Variablen $L' \cup O'$ ab. Also gilt insgesamt wegen $\alpha_1, \alpha_2 \in \mathsf{en}.r$ auch $\beta_1 \overset{L \cup O}{=} \beta_2$. □

A.2. Beweise zu Kapitel 5

Beweis zu Proposition 5.1, Seite 72. Wir zeigen die Eigenschaft (5.2) durch Widerspruchsannahme. Sei $(i, o) \in P$ und es existiere ein $(\tilde{i}, \tilde{o}) \in R_S$ mit $i \sqsubseteq \tilde{i}$ und $o \not\sqsubseteq \tilde{o}$. Da $P \subset R_S$ gilt und da das Modellverhalten R_S die Eigenschaft (4.1) hat, gilt auch $\#i = \#o$. Wir definieren $\bar{i} := \tilde{i} \downarrow_{\#i}$ und $\bar{o} := \tilde{o} \downarrow_{\#o}$, dann gilt $(\bar{i}, \bar{o}) \in R_S$, da R_S als Modellverhalten die Eigenschaft (4.3) hat. Insgesamt gibt es also ein $(i, o) \in P \subset R_S$ und ein $(\bar{i}, \bar{o}) \in R_S$ mit $i = \bar{i}$ und $o \neq \bar{o}$. Dies steht im Widerspruch, dass S deterministisch ist, d.h. die Eigenschaft (4.4) hat.

Wir zeigen analog die Eigenschaft (5.3) auf $R_{\hat{S}}$, indem wir im obigen Beweis P durch $P \cup Q$ und S durch $\hat{S}$ ersetzen. □

Beweis zu Proposition 5.2, Seite 82. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.2 vorausgesetzt.
zu 1. Das Regelsystem $\hat{G}$ besitzt die gleiche Startbedingung *Init* und die gleiche Regelmenge R wie das Regelsystem G. Also übertragen sich die Eigenschaften (4.15) und

(4.16) von G auf $\hat{G}$ und damit ist $\hat{G}$ deterministisch.
zu 2. Sei $(i,o) \in R_{Z_G}$. Dann existiert ein Ablauf $\eta = \langle \alpha_1, \alpha_2, \ldots \rangle \in \langle\!\langle Z_G \rangle\!\rangle$ in der von G induzierten Zustandsmaschine Z_G, so dass die Projektion von η auf die Schnittstelle (I,O) das Verhalten (i,o) ergibt (s. Def. 4.8, Prop. 4.5). Der Ablauf η induziert über die Transitionsmenge T_R (4.14) einen Strom von Regeln $\rho = \langle r_1, r_2, \ldots \rangle \in R^\omega$ mit $\#\rho + 1 = \#\eta$ und

$$\begin{aligned} \forall 1 \leq k < \#\eta \, \exists \tilde{\alpha}_k \in B_{loc_{r_k}} \text{ mit } \alpha_k \oplus \tilde{\alpha}_k \models pre_{r_k} \wedge input_{r_k} \text{ und} \\ \alpha_k \oplus \tilde{\alpha}_k, \alpha'_{k+1} \models output_{r_k} \wedge assign_{r_k}. \end{aligned} \tag{A.1}$$

Wir wählen nun für alle $k \in \mathsf{dom}.\eta$ eine beliebige Belegung $\beta_k \in B_{\hat{I} \setminus I}$ auf den neuen Eingabekanälen $\hat{I} \setminus I$. Dann gilt für alle $1 \leq k < \#\eta$ und den gleichen $\tilde{\alpha}_k$ wie in (A.1)

$$\begin{aligned} &\alpha_k \oplus \beta_k \oplus \tilde{\alpha}_k \models pre_{r_k} \wedge input_{r_k} \text{ und} \\ &\alpha_k \oplus \beta_k \oplus \tilde{\alpha}_k, \alpha'_{k+1} \oplus \beta'_{k+1} \models output_{r_k} \wedge assign_{r_k}, \end{aligned} \tag{A.2}$$

da die Eingabemuster $input_{r_k}$ die neuen Eingabekanäle $\hat{I} \setminus I$ nicht enthalten und die Ausgabe $output_{r_k}$ und die Zuweisung $assign_{r_k}$ die Eingabekanäle $\hat{I}$ nicht beeinflussen. Folglich haben wir einen Ablauf $\hat{\eta} = \langle \alpha_1 \oplus \beta_1, \alpha_2 \oplus \beta_2, \ldots \rangle \in \langle\!\langle Z_{\hat{G}} \rangle\!\rangle$ auf der Zustandsmaschine $Z_{\hat{G}}$ des Regelsystems $\hat{G}$ konstruiert, dessen Projektion auf die Schnittstelle $(\hat{I}, O)$ wir mit $(\hat{i}, \hat{o})$ bezeichnen. Nach Konstruktion gilt $(\hat{i}, \hat{o}) \in R_{Z_{\hat{G}}}$, $o = \hat{o}$ und $i.\iota = \hat{i}.\iota$ $(\iota \in I)$.
zu 3. Die Existenz von *lift* folgt direkt aus (2.). Nach (2.) ist die Wahl der Belegung der neuen Eingabekanäle beliebig. Wir liften z.B. einen Ablauf $(i,o) \in R_{Z_G}$ zu $(\hat{i}, \hat{o}) \in R_{Z_{\hat{G}}}$, indem wir $\hat{o} := o$ und

$$\hat{i}.\iota := \begin{cases} i.\iota & \text{für } \iota \in I \\ \epsilon^{\#i} & \text{für } \iota \in \hat{I} \setminus I \end{cases}$$

wählen.
zu 4. Wir definieren $N := R_{Z_{\hat{G}}} \setminus lift(R_{Z_G})$. Dabei ist N sicher nicht leer, falls $\hat{I} \setminus I \neq \emptyset$ gilt. Denn für einen Ablauf $(i,o) \in R_{Z_G}$ enthält $R_{Z_{\hat{G}}}$ nach der Konstruktion zu (2.) nicht nur $lift(i,o)$, sondern auch $(i \oplus \hat{i}, o)$ für ein beliebiges $\hat{i} \in \overrightarrow{\hat{I} \setminus I}$ mit $\#\hat{i} = \#i$. □

Beweis zu Proposition 5.3, Seite 83. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.3 vorausgesetzt.
zu 1. Dies folgt analog wie in (1.) im Beweis zu Proposition 5.1.
zu 2. Wir zeigen diesen Punkt ähnlich wie in (2.) im Beweis zu Proposition 5.1: sei $(i,o) \in R_{Z_G}$. Dann existiert ein Ablauf $\eta = \langle \alpha_1, \alpha_2, \ldots \rangle \in \langle\!\langle Z_G \rangle\!\rangle$ von der induzierten Zustandsmaschine Z_G, dessen Projektion auf die Schnittstelle (I,O) das Verhalten (i,o) ergibt. Der Ablauf η induziert über die Transitionsmenge T_R (4.14) einen Strom von Regeln $\rho = \langle r_1, r_2, \ldots \rangle \in R^\omega$ mit $\#\rho + 1 = \#\eta$ und der Eigenschaft (A.1). Ausgabekanäle, deren Wert durch keine Gleichung in der Ausgabe $output_r$ festgelegt wird, werden standardmäßig mit der leeren Nachricht ϵ belegt (vgl. Abschn. 4.5.2)). Dies gilt insbesondere für die neuen Ausgabekanäle $\hat{O} \setminus O$, so dass wir für alle $k \in \mathsf{dom}.\eta$ die

Belegungen $\beta_k \in B_{\hat{O}\setminus O}$ mit $\beta_k.\theta = \epsilon$ ($\theta \in \hat{O} \setminus O$) festlegen. Dann erfüllen die Belegungen $\alpha_k \oplus \beta_k$ für alle $1 \leq k < \#\eta$ die Eigenschaft (A.2). Wir haben also einen Ablauf $\hat{\eta} = \langle \alpha_1 \oplus \beta_1, \alpha_2 \oplus \beta_2, \ldots \rangle \in \langle\!\langle Z_{\hat{G}} \rangle\!\rangle$ auf der von $\hat{G}$ induzierten Zustandsmaschine $Z_{\hat{G}}$ konstruiert, dessen Projektion auf die Schnittstelle $(I, \hat{O})$ wir mit $(\hat{i}, \hat{o})$ bezeichnen. Nach Konstruktion gilt $(\hat{i}, \hat{o}) \in R_{Z_{\hat{G}}}$, $i = \hat{i}$ und $o.\theta = \hat{o}.\theta$ ($\theta \in O$).
zu 3. Die Existenz der Funktion *lift* folgt direkt aus (2.), wobei die neuen Ausgabekanäle $\hat{O} \setminus O$ standardmäßig mit der leeren Nachricht belegt werden. D.h. jedem Ablauf $(i, o) \in R_{Z_G}$ wird durch die Funktion *lift* der Ablauf $(\hat{i}, \hat{o}) \in R_{Z_{\hat{G}}}$ mit $\hat{i} = i$ und

$$\hat{o}.\theta = \begin{cases} o.\theta & \text{für } \theta \in O \\ \epsilon^{\#o} & \text{für } \theta \in \hat{O} \setminus O \end{cases}$$

zugeordnet.
zu 4. Dies folgt direkt aus (3.). □

Beweis zu Proposition 5.4, Seite 83. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.4 vorausgesetzt.
zu 1. Da die neuen lokalen Variablen $\hat{L} \setminus L$ durch die Definition der Startbedingung $\widehat{Init} \equiv Init \bigwedge_{l \in \hat{L}\setminus L} l = f_l$ ($f_l \in \widehat{type}(l)$) durch Zuweisungen eindeutig gebunden werden und die Startbedingung *Init* die Eigenschaft (4.15) hat, erfüllt auch $\widehat{Init}$ die Eigenschaft (4.15). Da beim Übergang von Regelsystem G zu Regelsystem $\hat{G}$ die Regelmenge R nicht verändert wurde, erfüllt auch $\hat{G}$ die Eigenschaft (4.16). Folglich ist $\hat{G}$ deterministisch.
zu 2. Da die neuen lokalen Variablen und die neuen Datenwerte durch die unveränderte Regelmenge R nicht referenziert werden und da das Modellverhalten durch Projektion vom lokalen Datenraum abstrahiert, ändert sich das Modellverhalten nicht, d.h. es gilt $R_{Z_{\hat{G}}} = R_{Z_G}$. □

Beweis zu Proposition 5.5, Seite 85. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.5 vorausgesetzt.
zu 1. Da beim Übergang von Regelsystem G zu Regelsystem $\hat{G}$ die Startbedingung *Init* nicht verändert wurde, erfüllt auch $\hat{G}$ die Eigenschaft (4.15). Durch das Ersetzen der Regel r durch $\hat{r}$ mit $\mathsf{en}.r \supset \mathsf{en}.\hat{r}$, $output_r \equiv output_{\hat{r}}$ und $assign_r \equiv assign_{\hat{r}}$ beim Übergang von der Regelmenge R zu der Regelmenge $\hat{R} = R \setminus \{r\} \cup \{\hat{r}\}$ bleibt die Eigenschaft (4.15) erhalten, denn es gilt (4.15) auf R und für alle $s \in \hat{R} \setminus \{\hat{r}\}$ gilt

$$\mathsf{en}.r \cap \mathsf{en}.s = \emptyset \overset{\mathsf{en}.r \supset \mathsf{en}.\hat{r}}{\Longrightarrow} \mathsf{en}.\hat{r} \cap \mathsf{en}.s = \emptyset.$$

Folglich ist $\hat{G}$ deterministisch.
zu 2. Wir zeigen $R_{Z_{\hat{G}}} \subset R_{Z_G}$. Sei $(i, o) \in R_{Z_{\hat{G}}}$. Dann existiert ein Ablauf $\eta \in \langle\!\langle Z_{\hat{G}} \rangle\!\rangle$, dessen Projektion auf die Schnittstelle (I, O) das Verhalten (i, o) ergibt. Ferner induziert der Ablauf η einen Strom von Regeln $\rho \in R^\omega$ mit $\#\rho + 1 = \#\eta$ und

$$\forall 1 \leq k < \#\eta \, \exists \tilde{\alpha}_k \in B_{loc_{\rho.k}} \text{ mit } \eta.k \oplus \tilde{\alpha}_k \models pre_{\rho.k} \wedge input_{\rho.k} \text{ und } \eta.k \oplus \tilde{\alpha}_k, \eta'.(k+1) \models output_{\rho.k} \wedge assign_{\rho.k}.$$

Durch die Voraussetzungen der ersetzenden Regel $\hat{r}$ gegenüber Regel r, gilt für den Strom ρ für alle $k \in \mathsf{dom}.\rho$

$$\begin{aligned} \mathsf{en}.(\rho.k) &\subset \mathsf{en}.(\rho[r/\hat{r}].k), \\ \mathit{output}_{\rho.k} &\equiv \mathit{output}_{\rho[r/\hat{r}].k} \text{ und} \\ \mathit{assign}_{\rho.k} &\equiv \mathit{assign}_{\rho[r/\hat{r}].k} \end{aligned}$$

Folglich ist η auch ein Ablauf von $\langle\langle Z_G \rangle\rangle$ und damit gilt für dessen Projektion $(i, o) \in R_{Z_G}$. □

Beweis zu Proposition 5.6, Seite 85. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.6 vorausgesetzt.
zu 1. Das Regelsystem $\hat{G}$ hat die gleiche Startbedingung *Init* wie Regelsystem G. Folglich überträgt sich die Eigenschaft (4.15) von G auf $\hat{G}$. Beim Übergang von G auf $\hat{G}$ wurden die Vorbedingungen und Eingabemuster der Regeln aus $\hat{R}$ gegenüber denen aus R nicht verändert. Folglich überträgt sich auch die Eigenschaft (4.16) von G auf $\hat{G}$. Insgesamt ist also das Regelsystem $\hat{G}$ deterministisch.
zu 2. Wir konstruieren die Funktion *lift* mit der gewünschten Eigenschaft $(i, \hat{o}) = \mathit{lift}(i, o)$ $((i, o) \in R_{Z_G})$, indem wir für jedes $(i, o) \in R_{Z_G}$ ein $(i, \hat{o}) \in R_{Z_{\hat{G}}}$ angeben. Sei also $(i, o) \in R_{Z_G}$. Dann konstruieren wir wie im Beweis zu Proposition 5.2 einen Ablauf $\eta = \langle \alpha_1, \alpha_2, \ldots \rangle \in \langle\langle Z_G \rangle\rangle$ und eine zugehörige Sequenz von Regeln $\rho = \langle r_1, r_2, \ldots \rangle \in R^\omega$, so dass (i, o) die Projektion von η auf die Schnittstelle (I, O) ist und für η und ρ die Eigenschaft (A.1) gilt. Folglich induziert die Regelsequenz $\hat{\rho} := \rho[\hat{r}/r]$ einen Ablauf $\hat{\eta} \in Z_{\hat{G}}$ mit $\alpha_k \overset{L \cup I}{=} \hat{\alpha}_k$ $(k \in \mathsf{dom}.\eta)$, da die Vorbedingungen, Eingabemuster und Zuweisungen beim Übergang von Regelmenge R zu $\hat{R}$ nicht verändert wurden. Folglich gilt für die Projektion $(\hat{i}, \hat{o})$ von $\hat{\eta}$ auf die Schnittstelle (I, O) $i = \hat{i}$. Also gilt insgesamt $(i, \hat{o}) \in R_{Z_{\hat{G}}}$.
zu 3. Hier gibt es nichts zu zeigen, da jede Veränderung an einem Modell unter Beibehaltung der Schnittstelle durch eine allgemeine Modifikation am Modellverhalten ausgedrückt werden kann (vgl. Abschnitt 5.2.2). Dennoch charakterisieren wir kurz die Ablaufmengen E und N: Die Ablaufmenge E enthält genau die Abläufe aus R_{Z_G}, bei denen die Regel r mindestens einmal geschaltet hat. Entsprechend werden die Abläufe aus E durch die Abläufe aus N ersetzt, bei denen anstatt der Regel r die Regel $\hat{r}$ schaltet. □

Beweis zu Proposition 5.7, Seite 86. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.6 vorausgesetzt.
zu 1. Das Regelsystem $\hat{G}$ hat die gleiche Startbedingung *Init* wie Regelsystem G. Folglich überträgt sich die Eigenschaft (4.15) von G auf $\hat{G}$. Beim Übergang von G auf $\hat{G}$ wurden die Vorbedingungen und die Eingabemuster der Regeln aus $\hat{R}$ gegenüber denen aus R nicht verändert. Folglich überträgt sich auch die Eigenschaft (4.16) von G auf $\hat{G}$. Insgesamt ist also das Regelsystem $\hat{G}$ ebenfalls deterministisch.
zu 3. Hier gibt es nichts zu zeigen, da jede Veränderung an einem Modell unter Beibehaltung der Schnittstelle durch eine allgemeine Modifikation am Modellverhalten ausgedrückt werden kann (vgl. Abschnitt 5.2.2). □

Beweis zu Proposition 5.8, Seite 88. Seien G und $\hat{G}$ Regelsysteme, wie in Proposition 5.6 vorausgesetzt.
1. Fall: Es gilt $\hat{R} = R \cup \{\hat{r}\}$.
Durch die Vorschriften (4.14) und (4.8) induziert das Regelsystem G mit den Verhaltensregeln R das Modellverhalten R_{Z_G}. Wegen $R \subset \hat{R}$ induziert $\hat{R}$ über die Vorschriften (4.14) und (4.8) das Modellverhalten $R_{Z_{\hat{G}}}$ mit $R_{Z_G} \subset R_{Z_{\hat{G}}}$. Folglich leistet $N := R_{Z_{\hat{G}}} \setminus R_{Z_G}$ das Gewünschte.
2. Fall: Es gilt $\hat{R} = R \setminus \{r\} \cup \{\hat{r}\}$ mit $\mathsf{en}.r \subset \mathsf{en}.\hat{r}$, $output_r \equiv output_{\hat{r}}$ und $assign_r \equiv assign_{\hat{r}}$.
Sei $(i, o) \in R_{Z_G}$. Dann induziert (i, o) wie im Beweis zu Proposition 5.2 einen Ablauf $\eta \in Z_G$ und eine Sequenz von Regeln $\rho \in R^\omega$, so dass (i, o) die Projektion von η auf die Schnittstelle (I, O) ist und η durch die Regelsequenz ρ entsteht (A.1). Dann gilt wegen den Voraussetzungen über r und $\hat{r}$ die Eigenschaft (A.1) auch für η und $\hat{\rho} := \rho[\hat{r}/r]$. Folglich gilt $(i, o) \in R_{Z_{\hat{G}}}$. Also gilt $R_{Z_G} \subset R_{Z_{\hat{G}}}$ und somit leistet $N := R_{Z_{\hat{G}}} \setminus R_{Z_G}$ das Gewünschte. □

A.3. Beweise zu Kapitel 6

Beweis zu Proposition 6.1, Seite 95. Da die Eingabekanäle I, die Ausgabekanäle O, die lokalen Variablen L und die Startbedingung *Init* von den Regelsystemen G und $\tilde{G}$ identisch sind, genügt es zu zeigen, dass die Regelmengen R und $\tilde{R}$ die gleiche Transitionsrelation einer Zustandsmaschine induzieren, d.h. $T_R = T_{\tilde{R}}$ gilt. Die Transitionsrelation T_R einer Regelmenge R wurde in Proposition 4.5 durch die Vorschrift (4.14) definiert:

$$T_R := \{(\alpha, \beta) \mid \exists r \in R,\, \tilde{\alpha} \in B_{loc_r} \text{ mit } \alpha \oplus \tilde{\alpha} \models pre_r \wedge input_r \text{ und } \alpha \oplus \tilde{\alpha}, \beta' \models output_r \wedge assign_r\}.$$

"$\subset$": Ist $(\alpha, \beta) \in T_R$, dann existiert eine Regel $r \in R$ und eine Belegung $\tilde{\alpha} \in B_{loc_r}$ mit

$$\alpha \oplus \tilde{\alpha} \models pre_r \wedge input_r \text{ und } \alpha \oplus \tilde{\alpha}, \beta' \models outp_r \wedge assign_r. \tag{A.3}$$

Da die Prädikate aus P den Zustandsraum B_V partitionieren, existiert ein eindeutiges Prädikat $p \in P$ bzw. ein eindeutiges Prädikat $q \in P$ mit

$$\alpha \models p \text{ und } \beta \models q. \tag{A.4}$$

Nach Definition 4.13 ist die Zuweisung $assign_r$ eine Konjunktion der Form $assign_r \equiv \bigwedge_{l \in K} l' = f_l$ und dann gilt wegen (A.3) und (A.4)

$$\alpha \oplus \tilde{\alpha}, \beta' \models \bigwedge_{l \in K} l' = f_l \text{ und } \beta' \models q'. \tag{A.5}$$

Daraus folgt

$$\alpha \oplus \tilde{\alpha}, \beta' \models q' \wedge \bigwedge_{l \in K} l' = f_l. \tag{A.6}$$

Da $q'[f_l/l']_{l\in L} \equiv q' \wedge \bigwedge_{l\in K} l' = f_l$ gilt und $q'[f_l/l']_{l\in L}$ frei von gestrichenen Variablen ist, folgt aus (A.6)

$$\alpha \oplus \tilde{\alpha} \models q'[f_l/l']_{l\in L}. \tag{A.7}$$

Mit $pre_{\tilde{r}} := pre_r \wedge p \wedge q'[f_l/l']_{l\in L}$ konstruieren wir also eine Regel

$$\tilde{r} := (pre_{\tilde{r}}, input_r, output_r, assign_r)$$

aus $\tilde{R}$, für die wegen (A.3), (A.4) und (A.7)

$$\alpha \oplus \tilde{\alpha} \models pre_{\tilde{r}} \wedge input_r \text{ und } \alpha \oplus \tilde{\alpha}, \beta' \models outp_r \wedge assign_r.$$

gilt. Folglich gilt $(\alpha, \beta) \in T_{\tilde{R}}$.

"$\supset$": Ist $(\alpha, \beta) \in T_{\tilde{R}}$, dann existiert nach der Definition von $\tilde{R}$ und $T_{\tilde{R}}$ eine Regel $r \in R$ bzw. eine Belegung $\tilde{\alpha} \in B_{loc_r}$ mit

$$\alpha \oplus \tilde{\alpha} \models pre_{\tilde{r}} \wedge input_r \text{ und } \alpha \oplus \tilde{\alpha}, \beta' \models outp_r \wedge assign_r,$$

wobei $pre_{\tilde{r}} \equiv pre_r \wedge p \wedge q'[f_l/l']_{l\in L}$ und $assign_r \equiv \bigwedge_{l\in K} l' = f_l$ gilt. Also gilt insbesondere

$$\alpha \oplus \tilde{\alpha} \models pre_r \wedge input_r \text{ und } \alpha \oplus \tilde{\alpha}, \beta' \models outp_r \wedge assign_r,$$

und damit gilt $(\alpha, \beta) \in T_R$. □

B. Fallstudie

Dieser Anhang enthält ergänzendes Material zu der Fallstudie, die wir ausführlich in Kapitel 7 beschrieben haben. Abschnitt B.1 enthält alle Datentyp und Funktionsdefinition, Abschnitt B.2 Abläufe und Regressionstestsuiten der einzelnen Inkremente.

B.1. DTDs der Fallstudie

B.1.1. Datentypen

Es folgen die Datentypdefinitionen des Modells aus Abschnitt 7.1. Wie die Datentypen im Laufe der inkrementellen Entwicklung erweitert wurden, ist in Form Kommentaren im Code dokumentiert.

```
/******************
types for channels
******************/

data dNet    = NetOn | NetOff;
data dMsgEnv = MsgEnv(dAddr, dAddr, dInstID, dFktOPTypeEnv);
data dMsgNM  = MsgNM(dAddr, dAddr, dInstID, dFktOPTypeNM);

//extended by value logInvalid for increment 2
data dAddr   = phy0x0400 | phy0x0401 | phy0x0402 | phy0x0403 |
               log0x0100 | log0x0101 | log0x0102 | log0x0103 |
               logInvalid | bc0x03C8;

data dInstID = inst0x00 | inst0x01 | inst0x02 | inst0x03;

data dFktOPTypeEnv = FBlockIDsStatus(dFBlockIDList);
data dFktOPTypeNM  = FBlockIDsGet | ConfigurationStatus(dStatus);

//extended by value NotOk for increment 2
//extended by value New(..) for increment 3
//extended by values Invalid(..) and InvalidNew(...,...) for increment 4
data dStatus       = Ok | NotOk | New(dFBlockIDList) |
                     Invalid(dFBlockIDList) |
```

```
                    InvalidNew(dFBlockIDList,dFBlockIDList);

data dFBlockIDList = noFB | FBList(dFBlockID,dInstID,dFBlockIDList);
data dFBlockID     = FBlock1 | FBlock2 | FBlock3;

//types for channels added for increment 3
data dTimer = tAnswer;
data dTimeout = timeoutAnswer;

//types for channels added for increment 4
data dNCD = nMPR(Int);


/************************
types for local variables
************************/

//extended by value delayed for increment 3
//extended by value ncd for increment 4
data dMode     = off | init | cfgOk | delayed | ncd;

data dDevList  = noDL | DevList(dInstID,dRequestStatus,dDevList);

data dRequestStatus = notRequested | requested | answered;

data dRegistry = noR | Registry(dInstID,dAddr,dFBlockIDList,dRegistry);
```

B.1.2. Prädikate und Funktionen

Es folgen die Prädikats- und Funktionsdefinitionen, die in den Verhaltensregeln der Fallstudie aus Abschnitt 7.1 verwendet werden. Welche Prädikate und Funktionen im Laufe der inkrementellen Entwicklung hinzugefügt wurden, ist in Form Kommentaren im Code dokumentiert.

```
import Most_Data;
/************************
functions for increment 1
************************/

buildDevList: Int -> dDevList;
fun buildDevList(0) = noDL
  | buildDevList(1) = DevList(inst0x01,notRequested,noDL)
  | buildDevList(2) =
```

```
    DevList(inst0x01,notRequested,
    DevList(inst0x02,notRequested,noDL))
  | buildDevList(3) =
    DevList(inst0x01,notRequested,
    DevList(inst0x02,notRequested,
    DevList(inst0x03,notRequested,noDL)))
  | buildDevList(_) = noDL;

isNotReq: dDevList -> Bool;
fun isNotReq(noDL) = False
  | isNotReq(DevList(_,notRequested,_)) = True
  | isNotReq(DevList(_,_,DLIST)) = isNotReq(DLIST);

getNotReq: dDevList -> dInstID;
fun getNotReq(noDL) = inst0x00
  | getNotReq(DevList(INST,notRequested,_)) = INST
  | getNotReq(DevList(_,_,DLIST)) = getNotReq(DLIST);

setNextReq: dDevList -> dDevList;
fun setNextReq(noDL) = noDL
  | setNextReq(DevList(INST,notRequested,DLIST)) =
    DevList(INST,requested,DLIST)
  | setNextReq(DevList(INST,RSTATUS,DLIST)) =
    DevList(INST,RSTATUS,setNextReq(DLIST));

setAns: dInstID -> dDevList -> dDevList;
fun setAns(_,noDL) = noDL
  | setAns(INST,DevList(I,RSTATUS,DLIST)) =
    if INST==I then DevList(INST,answered,DLIST)
    else DevList(I,RSTATUS,setAns(INST,DLIST)) fi;

inst2phy: dInst -> dAddr;
fun inst2phy(inst0x00) = phy0x0400
  | inst2phy(inst0x01) = phy0x0401
  | inst2phy(inst0x02) = phy0x0402
  | inst2phy(inst0x03) = phy0x0403
  | inst2phy(_) = phy0x0400;


/************************
functions for increment 2
************************/

setNotReq: DevList -> DevList;
```

```
fun setNotReq(noDL) = noDL
  | setNotReq(DevList(INST,_,DLIST)) =
    DevList(INST,notRequested,setNotReq(DLIST));

isAnsOk: dInst -> dAddr -> dRegistry -> Bool;
fun isAnsOk(INST,LOG,REG) =
    isAddrValid(LOG) && not(isAddrDuplicate(INST,LOG,REG));

isAddrValid: dAddr -> Bool;
fun isAddrValid(log0x0100) = True
  | isAddrValid(log0x0101) = True
  | isAddrValid(log0x0102) = True
  | isAddrValid(log0x0103) = True
  | isAddrValid(_) = False;

isAddrDuplicate: dInst -> dAddr -> dRegistry -> Bool;
fun isAddrDuplicate(_,_,noR) = False
  | isAddrDuplicate(INST,LOG,Registry(I,L,_,REG)) =
    if ((INST==I && LOG!=L)||(INST!=I && LOG==L)) then True
    else isAddrDuplicate(INST,LOG,REG) fi;


/************************
functions for increment 3
************************/

haveAllAns: DevList -> DevList;
fun haveAllAns(noDL) = True
  | haveAllAns(DevList(_,answered,DLIST)) = haveAllAns(DLIST)
  | haveAllAns(DevList(_,_,_)) = False;

setReq2NotReq: DevList -> DevList;
fun setReq2NotReq(noDL) = noDL
  | setReq2NotReq(DevList(INST,requested,DLIST)) =
    DevList(INST,notRequested,setReq2NotReq(DLIST))
  | setReq2NotReq(DevList(INST,RSTATUS,DLIST)) =
    DevList(INST,RSTATUS,setReq2NotReq(DLIST));


/************************
functions for increment 4
************************/

allFBs: dRegistry -> dFBlockIDList;
```

```
fun allFBs(REG)=buildFBs(REG,noFB);

buildFBs:dRegistry -> dFBlockIDList -> dFBlockIDList;
fun buildFBs(noR,RESULT) = RESULT
  | buildFBs(Registry(_,_,FBL,REG),OFBL) =
    buildFBs(REG,appendFBs(FBL,OFBL));

appendFBs: dFBlockIDList -> dFBlockIDList -> dFBlockIDList;
fun appendFBs(noFB,FLIST) = FLIST
  | appendFBs(FBList(FID,INST,FLIST),OFLIST) =
    appendFBs(FLIST,FBList(FID,INST,OFLIST));

regChange: dFBlockIDList -> dRegistry -> dFktOPTypeNM;
fun regChange(FBL,REG) =
  if (minusFBL(allFBs(REG),FBL)!=noFB &&
      minusFBL(FBL,allFBs(REG))!=noFB) then
      ConfigurationStatus(InvalidNew(minusFBL(FBL,allFBs(REG)),
      minusFBL(allFBs(REG),FBL)))
  else if (minusFBL(allFBs(REG),FBL)==noFB &&
           minusFBL(FBL,allFBs(REG))!=noFB) then
           ConfigurationStatus(Invalid(minusFBL(FBL,allFBs(REG))))
       else if (minusFBL(allFBs(REG),FBL)!=noFB &&
                minusFBL(FBL,allFBs(REG))==noFB) then
                ConfigurationStatus(New(minusFBL(allFBs(REG),FBL)))
            else ConfigurationStatus(Ok)
            fi
       fi
  fi;

minusFBL: dFBlockIDList -> dFBlockIDList -> dFBlockIDList ;
fun minusFBL(noFB,FBL)=noFB
  | minusFBL(FBL,noFB) = FBL
  | minusFBL(FBList(FB,INST,FLIST),OFLIST) =
    if isFBinFBList(FB,INST,OFLIST) then minusFBL(FLIST,OFLIST)
    else FBList(FB,INST,minusFBL(FLIST,OFLIST)) fi;

isFBinFBList: dFBlockID -> dInstID -> dFBlockList -> Bool;
fun isFBinFBList(_,_,noFB) = False
  | isFBinFBList(FB,INST,FBList(OFB,OINST,OFLIST)) =
    if (FB==OFB&&INST==OINST) then True
    else isFBinFBList(FB,INST,OFLIST) fi;
```

B.2. Abläufe und Regressionstestsuiten

In die diesem Abschnitt geben wir die Abläufe zu den Regressionstestsuiten P_1, P_2, P_3 und P_4, die sich auf die Modellinkremente aus den Abschnitten 7.3.3, 7.3.4, 7.3.5 bzw. 7.3.6 beziehen.

B.2.1. Regressionstestsuite zum ersten Modellinkrement

Die Abbildungen B.1 und B.2 zeigen die Abläufe der Regressionstestsuite P_1.

input channels		output channels
net	fromEnv	fromNM
NetOn		
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
NetOff		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))

Abbildung B.1.: Ablauf zum Startverhalten

input channels		output channels
net	fromEnv	fromNM
NetOn		
	MsgEnv(log0x0103, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0101, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
NetOff		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))

Abbildung B.2.: Ablauf zum Startverhalten

B.2.2. Regressionstestsuite zum zweiten Modellinkrement

Die Abbildungen B.3, B.4 und B.5 zeigen die Abläufe, die die Regressionstestsuite P_1 zu Regressionstestsuite P_2 erweitern.

input channels		output channels
net	fromEnv	fromNM
NetOn		
	MsgEnv(logInvalid, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
NetOff		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))

Abbildung B.3.: Ablauf mit Netzwerkreset

input channels		output channels
net	fromEnv	fromNM
NetOn		
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0101, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
NetOff		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))

Abbildung B.4.: Ablauf mit Netzwerkreset

input channels		output channels
net	fromEnv	fromNM
NetOn		
	MsgEnv(log0x0102, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)
NetOff		MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))

Abbildung B.5.: Ablauf mit Netzwerkreset

B.2.3. Regressionstestsuite zum dritten Modellinkrement

Bei Bildung der dritten Inkrements wurde die Ausgabeschnittstelle, um einen Kanal erweitert. Dementsprechend müssen die Abläufe aus Regressionstestsuite P_2 auf die Schnittstelle des dritten Modellinkrements geliftet werden. Die Abbildungen B.6, B.7, B.8, B.9 und B.10 zeigen die Abläufe der gelifteten Regressionstestsuite P_2^{lift}. Die Abbildungen B.11, B.12, B.13 und B.14 zeigen die Abläufe, die die Regressionstestsuite P_2^{lift} zu Regressionstestsuite P_3 erweitern.

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.6.: Ablauf zum Startverhalten (geliftet)

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
	MsgEnv(log0x0103, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0101, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.7.: Ablauf zum Startverhalten (geliftet)

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
	MsgEnv(logInvalid, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.8.: Ablauf mit Netzwerkreset (geliftet)

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0101, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.9.: Ablauf mit Netzwerkreset (geliftet)

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
	MsgEnv(log0x0102, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.10.: Ablauf mit Netzwerkreset (geliftet)

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
		timeout-Answer	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
		timeout-Answer	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(New(FBList(FBlock1, inst0x01, noFB))))	
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(New(FBList(FBlock1, inst0x02, noFB))))	

Abbildung B.11.: Nicht antwortende Geräte

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
		timeoutAnswer	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
		timeoutAnswer	MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(New(FBList(FBlock1, inst0x02, noFB))))	

Abbildung B.12.: Nicht antwortende Geräte

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
		timeout-Answer	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(New(FBList(FBlock1, inst0x01, noFB))))	

Abbildung B.13.: Nicht antwortende Geräte

input channels			output channels	
net	fromEnv	to	fromNM	t
NetOn				
		timeout-Answer	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(logInvalid, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))		MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff			MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.14.: Nicht antwortende Geräte

B.2.4. Regressionstestsuite zum vierten Modellinkrement

Abschließend stellen die Abbildungen B.15, B.16, B.17, B.18, B.19 und B.20 die Abläufe dar, die das Verhalten des NetworkMasters beim Auftreten eines NCD-Ereignisses in unterschiedlichen Situationen zeigen und die die Regressionstestsuite P_3 zu der Regressionstestsuite P_4 erweitern.

input channels				output channels	
net	fromEnv	to	n	fromNM	t
NetOn					
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			nMPR(1)	MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
NetOff				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Invalid(FBList(FBlock1, inst0x02, noFB))))	

Abbildung B.15.: Verhalten bei einem NCD-Ereignis

input channels				output channels	
net	fromEnv	to	n	fromNM	t
NetOn					
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			nMPR(3)	MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0103, log0x0100, inst0x03, FBlockIDsStatus(FBList(FBlock1, inst0x03, noFB)))			MsgNM(log0x0100, phy0x0403, inst0x03, FBlockIDsGet)	tAnswer
NetOff				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(New(FBList(FBlock1, inst0x03, noFB))))	

Abbildung B.16.: Verhalten bei einem NCD-Ereignis

input channels				output channels	
net	fromEnv	to	n	fromNM	t
NetOn					
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			nMPR(3)	MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
		timeout-Answer		MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0103, log0x0100, inst0x03, FBlockIDsStatus(FBList(FBlock1, inst0x03, noFB)))			MsgNM(log0x0100, phy0x0403, inst0x03, FBlockIDsGet)	tAnswer
				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(InvalidNew(FBList(FBlock1, inst0x02, noFB), FBList(FBlock1, inst0x03, noFB))))	
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff				MsgNM(log0x0100, bc-0x03C8, inst0x00, Configu-rationStatus(New(FBList(FBlock1,inst0x02,noFB))))	

Abbildung B.17.: Verhalten bei einem NCD-Ereignis

input channels				output channels	
net	fromEnv	to	n	fromNM	t
NetOn					
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			nMPR(1)	MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
	MsgEnv(logInvalid, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
NetOff				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.18.: Verhalten bei einem NCD-Ereignis

input channels				output channels	
net	fromEnv	to	n	fromNM	t
NetOn					
			nMPR(1)	MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
NetOff				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.19.: Verhalten bei einem NCD-Ereignis

input channels				output channels	
net	fromEnv	to	n	fromNM	t
NetOn					
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
			nMPR(2)	MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	
				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(NotOk))	
	MsgEnv(log0x0101, log0x0100, inst0x01, FBlockIDsStatus(FBList(FBlock1, inst0x01, noFB)))			MsgNM(log0x0100, phy0x0401, inst0x01, FBlockIDsGet)	tAnswer
	MsgEnv(log0x0102, log0x0100, inst0x02, FBlockIDsStatus(FBList(FBlock1, inst0x02, noFB)))			MsgNM(log0x0100, phy0x0402, inst0x02, FBlockIDsGet)	tAnswer
NetOff				MsgNM(log0x0100, bc0x03C8, inst0x00, ConfigurationStatus(Ok))	

Abbildung B.20.: Verhalten bei einem NCD-Ereignis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AB99] P. Ammann and P. Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. In *Proc. 18th Digital Avionics Systems Conference (DASC'99)*, volume 2, pages 10.A.6.1–10, October 1999.

[AS85] B. Alpern and F. B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, February 1985.

[ASC02] ASCET-SD product info. `http://www.etas.info/html/products/ec/ascetsd/en_products_ec_ascetsd_index.php`, 2002.

[Bal96] Helmut Balzert. *Lehrbuch der Software-Entwicklung, Software-Entwicklung.* Spektrum Akademischer Verlag, 1996.

[BBC+02] Peter Braun, Manfred Broy, Victoria Cengarle, Jan Philipps, Wolfgang Prenninger, Alexander Pretschner, Martin Rappl, and Robert Sandner. The Automotive CASE. In *DFG-Workshop Modelle, Werkzeuge, Infrastrukturen zur Unterstützung von Entwicklungsprozessen.* Wiley-VCH, 2002.

[BCG+00] D. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. Ip, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting. The transaction-based verification methodology. Technical report, Cadence Design Systems, Inc., August 2000.

[Bea02] Beacon for Simulink/Stateflow. `http://www.adi.com/products_be_bss.htm`, 2002.

[Bec00] Kent Beck. *Extreme Programming Explained.* Addison-Wesley, 2000.

[Bet02] Betterstate Product Info. `http://www.windriver.com/products/html/betterstate.html`, 2002.

[BETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4(5):235–282, 1994.

[BFV+99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation: A Simple Experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[BGS84] B. Boehm, T. Gray, and T. Seewaldt. Protoyping Versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering*, SE-10(3):290–303, 1984.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005. to appear.

[BLS00] Peter Braun, Heiko Lötzbeyer, and Oscar Slotosch. Quest Users Guide. `http://www4.in.tum.de/proj/quest/papers/UserGuide.pdf`, 2000.

[Bor04] Borland. Together. `http://www.borland.com/together/`, 2004.

[BP99] Max Breitling and Jan Philipps. Black Box Views of State Machines. Technical Report TUM-I9916, Technische Universität München, 1999.

[Bre01] Max Breitling. *Formale Fehlermodelierung für verteilte reaktive Systeme*. PhD thesis, Technische Universität München, 2001.

[Bro86] F. Brooks. No Silver Bullet. In *Proc. 10th IFIP World Computing Conference*, pages 1069–1076, 1986.

[BRSS97] Manfred Broy, Franz Regensburger, Bernhard Schätz, and Katharina Spies. The Steamboiler Specification - A Case Study in Focus. Technical Report TUM-I9714, Technische Universität München, 1997.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement*. Springer New York, 2001. ISBN 0-387-95073-7.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CH04] Alistair Cockburn and Jim Highsmith. Crystal Methodologies. `http://alistair.cockburn.us/crystal/crystal.html`, 2004.

[CJRZ01] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Automated Test and Oracle Generation for Smart-Card Applications. In *Proc. E-smart*, pages 58–70, 2001.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[Col04] CollabNet. ArgoUML. Project Hompage: `http://argouml.tigris.org/`, 2004.

[DBG01] Julia Dushina, Mike Benjamin, and Daniel Geist. Semi-Formal Test Generation with Genevieve. In *Proc. DAC*, 2001.

[Den92] Ernst Denert. *Software-Engineering*. Springer-Verlag, 1992.

[Den93] Ernst Denert. Dokumentenorientierte Software-Entwicklung. *Informatik-Spektrum*, 16(3):159–164, 1993.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[DN84] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE TSE*, SE-10(4):438–444, July 1984.

[Dou98] Bruce Powel Douglass. *Real-Time UML*. Object Technology Series. Addison Wesley, 1998.

[DW00] Wolfgang Dröschel and Manuela Wiemers. *Das V-Modell 97*. Oldenbourg Verlag, 2000.

[Fag76] M. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[Fag02] M. Fagan. Reviews and Inspections. In *Software Pioneers–Contributions to Software Engineering*, pages 562–573. Springer Verlag, 2002.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FHP02] Eitan Farchi, Alan Hartman, and Shlomit Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[FI98] P. Frankl and O. Iakounenko. Further Empirical Studies of Test Effectiveness. In *Proc.6th ACM SIGSOFT Intl. Symp. on the Foundations of Software Engineering*, 1998.

[FKL99] Lauret Fournier, Anatoly Koyfman, and Moshe Levinger. Developing an Architecture Validation Suite—Application to the PowerPC Architecture. In *Proc. 36th ACM Design Automation Conf.*, pages 189–194, 1999.

[FW93] P. Frankl and S. Weiss. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE TSE*, 19(8):774–787, 1993.

[GAD02] GADV. 4CS - Test-Suite für moderne Komfort-, Multimedia- und Unterhaltungs-Elektronik-Komponenten. `http://www.4cs.de/`, 2002.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[Gut99] W. Gutjahr. Partition testing versus random testing: the influence of uncertainty. *IEEE TSE*, 25(5):661–674, 1999.

[GW86] M. Girgis and M. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proc. IEEE/ACM workshop on software testing*, pages 64–73, July 1986.

[Ham94] R. Hamlet. Random Testing. In J. J. Marciniak, editor, *Encyclopedia of Software Testing*, volume 2. Addison-Wesley, 1994.

[Han04] Michael Hanus. Functional logic language curry. Language Hompage: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2004.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[HBGL95] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A Toolset for Specifying and Analyzing Requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

[HCDG+02] John Hudak, Santiago Comella-Dorda, David P. Gluch, Grace Lewis, and Chuck Weinstock. Model-Based Verification: Abstraction Guidelines. Technical Report CMU/SEI-2002-TN-011, Carnegie Mellon University, October 2002.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE'94*, pages 191–200, 1994.

[HGW04] M. Heimdahl, D. George, and R. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *Proc. 8th IEEE High Assurance in Systems Engineering Workshop*, February 2004.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[HL96] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and Consistency in Hierarchical State-Base Requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[HLK95] C. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 56. IEEE Computer Society, 1995.

[HLSU02] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proc. TACAS'02*, pages 327–341, 2002.

[Hoa85] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[HS97] Franz Huber and Bernhard Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch*, pages 343 – 352. GMD Verlag (St. Augustin), 1997.

[HSE97] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent Graphical Specification of Distributed Systems. In *Proc. 4th Intl. Symp. of Formal Methods Europe (FME'97)*, volume 1313 of *LNCS*, pages 122 – 141. Springer Verlag, 1997.

[HT90] D. Hamlet and R. Taylor. Partition Test Does Not Inspire Confidence. *IEEE TSE*, 16(12):1402–1411, December 1990.

[IEE98] IEEE. IEEE Standard for Software Verification and Validation, 1998.

[JPZ97] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. pages 184–196, 1997.

[KKLS00] Idit Keidar, Roger Khazan, Nancy A. Lynch, and Alexander A. Shvartsman. An inheritance-based technique for building simulation proofs incrementally. In *International Conference on Software Engineering*, pages 478–487, 2000.

[KPS01] Ingolf Krüger, Wolfgang Prenninger, and Robert Sandner. Architectural Design of a Broadcasting System using UML-RT. In Andreas Schürr, editor, *OMER-2 Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung*. Universität der Bundeswehr, München, 2001.

[KPS02] Ingolf Krüger, Wolfgang Prenninger, and Robert Sandner. Development of an Autonomous Transport System using UML-RT. Technical Report TUM-I0215, Technische Universität München, 2002.

[KPSB02] Ingolf Krüger, Wolfgang Prenninger, Robert Sandner, and Manfred Broy. From Scenarios to Hierarchical Broadcasting Software Architectures using UML-RT. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 2002.

[Kru00] Philippe Kruchten. *Rational Unified Process - An Introduction*. Addison-Wesley, 2000.

[KVZ98] Hakim Kahlouche, Cesar Viho, and Massimo Zendri. An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol. In A. Petrenko and N. Yevtushenko, editors, *IFIP TC6 11th International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1998.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Toplas*, 16(3):872–923, May 1994.

[LPS+02] Heiko Lötzbeyer, Wolfgang Prenninger, Oscar Slotosch, Ralf Steinbrüggen, Thomas Ströse, and Ferdinand Winhard. Einführung in die Systemmodellierung mit AutoFOCUS. `http://autofocus.in.tum.de/nelli/html/`, 2002.

[LT89] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Lyu96] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, 1996.

[Löt03] Heiko Lötzbeyer. *Modellbasierte Testfallermittlung für eingebettete Systeme in sicherheitskritischen Anwendungen*. PhD thesis, Technische Universität München, 2003.

[Mat02a] MATRIXx©Product Family. `http://www.mathworks.com/products/matrixx/index.shtml`, 2002.

[Mat02b] The MathWorks Product Family. `http://www.mathworks.com/products/prodoverview.shtml`, 2002.

[McM92] K.L. McMillan. The SMV system, Symbolic Model Checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[Mel88] Thomas Melham. Abstraction Mechanisms for Hardware Verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157, Boston, 1988. Kluwer Academic Publishers.

[Mil71] Harlan Mills. Top-Down Programming in Large Systems. *Debugging Techniques in Large Systems*, 1971.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[MIO87] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[MOS99] MOST Cooperation. MOST Specification Framework, rev. 1.1. `http://www.mostnet.de/downloads/Specifications/`, 1999.

[MOS02a] MOST Cooperation. MOST Function Catalog. `http://www.mostnet.de/downloads/Specifications/`, 2002.

[MOS02b] MOST Cooperation. MOST Specification, Rev. 2.2. `http://www.mostnet.de/downloads/Specifications/`, 2002.

[MT04] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[Mus99] J.D. Musa. *Software Reliability Engineering*. McGraw-Hill, 1999.

[Mye78] G. Myers. A Controlled Experiment in Program Testing and Codel Walkthroughs/Inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[Mye89] Glenford Myers. *Methodisches Testen von Programmen*. Oldenbourg Verlag, 1989. ISBN 3-486-21317-2.

[Nta84] Simon Ntafos. An evaluation of required element testing strategies. In *Proc. ICSE '84*, pages 250–256, 1984.

[Nta98] Simeon Ntafos. On Random and Partition Testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 42–48. ACM Press, 1998.

[OMG00] OMG. The Common Object Request Broker: Architecture and Specification, 2000.

[OMG02] OMG. OMG Unified Modeling Language Specification (Action Semantics), Januar 2002. Final Adopted Specification.

[ORM01] OMG Architecture Board ORMSC. Model Driven Architecture (MDA). OMG Document ormsc/2001-07-01, July 2001.

[PBR04] Wolfgang Paul, Manfred Broy, and Thomas Rieden. Verisoft. Project Hompage: `http://www.verisoft.de/index.html`, 2004.

[PERH05] Wolfgang Prenninger, Mohammad El-Ramly, and Marc Horstmann. Case Studies. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005. to appear.

[PLP01] Alexander Pretschner, Heiko Lötzbeyer, and Jan Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping*, pages 155–160, 2001.

[PM91] D. Parnas and J. Madey. Functional Documentation for Computer Systems Engineering (Version 2). Technical Report CRL Report 237, Department of Electrical and Computer Engineering, McMaster University, 1991.

[PP04] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In Mauro Pezze, editor, *Proceedings Test and Analysis of Component-based Systems (TACoS'04)*, 2004.

[PPS+03] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-Based test case generation for smart cards. In *In Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003.

[PPW+05] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas

Stauner. One Evaluation of Model-Based Testing and its Automation. In William Griswold and Bashar Nuseibeh, editors, *International Conference on Software Engineering (ICSE)*, 2005. to appear.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kenneth Baclavski and Haim Kilov, editors, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. *Refactoring of Programs and Specifications.*, pages 281 – 297. Kluwer Academic Publishers, 2003.

[Pre01] Alexander Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proc. Formal Approaches to Testing of Software*, pages 47–60, 2001.

[Pre03a] Alexander Pretschner. Compositional generation of MC/DC test suites. *Electronic Notes in Theoretical Computer Science*, 82(6):1–11, 2003.

[Pre03b] Alexander Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. PhD thesis, Technische Universität München, 2003.

[PS99] Jan Philipps and Oscar Slotosch. The Quest for Correct Systems: Model Checking of Diagramms and Datatypes. In *Asia Pacific Software Engineering Conference 1999*, 1999. to appear.

[PSAK04] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-Based Testing for Real—The Inhouse Card Case Study. *Software Tools for Technology Transfer*, 5(2–3):140–157, 2004.

[PTLP98] Stacy Prowell, Carmen Trammell, Richard Linger, and Jesse Poore. *Cleanroom Software Engineering*. Addison-Wesley, 1998.

[Ros02] Rational Rose Real-Time. `http://www.rational.com/product/rose/index.jsp`, 2002.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, Technische Universität München, 1996.

[SA99] Jian Shen and Jacob Abraham. An RTL Abstraction Technique for Processor Micorarchitecture Validation and Test Generation. *J. Electronic Testing: Theory&Application*, 16(1-2):67–81, February 1999.

[Sch98] Peter Scholz. *Design of Reactive Systems and their Distributed Implementation wirh Statecharts*. PhD thesis, Technische Universität München, 1998.

[Sch04] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[SHH+03] Bernhard Schätz, Tobias Hain, Frank Houdek, Wolfgang Prenninger, Martin Rappl, Jan Romberg, Oscar Slotosch, Martin Strecker, and Alexander Wißpeintner. Case tools for embedded systems. Technical Report TUM-I0309, Technische Universität München, 2003.

[Sof04a] IBM Rational Software. Rational RequisitePro. `http://www-306.ibm.com/software/awdtools/reqpro/`, 2004.

[Sof04b] IBM Rational Software. Rational Suite DevelopmentStudio. `http://www-306.ibm.com/software/awdtools/suite/dstudio/unix/`, 2004.

[SRS+03] Bernhard Schätz, Jan Romberg, Oscar Slotosch, Martin Strecker, Alexander Wißpeintner, Tobias Hain, Wolfgang Prenninger, Martin Rappl, and Katharina Spies. Modeling Embedded Software: State of the Art and Beyond. In *Proceedings of ICCSEA 16th International Conference on Software and Systems Engineering and their Applications*, 2003.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, 1973.

[Sun04a] Sun Microsystems, Inc. Java 2 platform, enterprise edition (J2EE). `http://java.sun.com/j2ee/`, 2004.

[Sun04b] Sun Microsystems, Inc. Java foundation classes (JFC/Swing). `http://java.sun.com/products/jfc/index.jsp`, 2004.

[SZP96] H. Shen, J. Zucker, and D.L. Parnas. Table transformation tools: Why and how. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 3, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[Tan00] Andrew Tanenbaum. *Computernetzwerke*. Pearson Studium, 2000. ISDN 3-8273-7011-6.

[Tau02] Telelogic Tau UML Suite. `http://www.telelogic.com/products/tau/uml/index.cfm`, 2002.

[Tel02] Telelogic. doors/ERS. http://www.telelogic.com/products/doorsers/index.cfm, 2002.

[THM99] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification Based Prototyping for Embedded Systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[Too01] ARTiSAN Software Tools. ARTiSAN Real-time Studio. `http://www.artisansw.com/products/professional_overview.asp`, 2001.

[TSR03] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, January 2003.

[VCC02] Cadence VCC Product Information. `http://www.cadence.com/products/vcc.html`, 2002.

[Weg01] Joachim Wegener. *Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen*. PhD thesis, Humboldt Univerität Berlin, 2001.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE TSE*, 20(5):353–363, May 1994.

[Wiß05] Alexander Wißpeintner. *Modellbasiertes Refactoring – Modelltransformationen für den inkrementellen Entwurf von reaktiven Systemen.* PhD thesis, Technische Universität München, 2005. to appear.

[Zha97] H. Zhang. SATO: An Efficient Propositional Prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July 13–17 1997. Springer.

[ZHM97] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.