Zum modellbasierten funktionalen Test reaktiver Systeme

Alexander Pretschner

Institut für Informatik
der Technischen Universität München

Zum modellbasierten funktionalen Test reaktiver Systeme

Walter Alexander Pretschner

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. Dr. Florian Matthes

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Univ.-Prof. Dr. Klaus D. Müller-Glaser, Universität Karlsruhe (TH)

Die Dissertation wurde am 13.3.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 4.7.2003 angenommen.

Kurzfassung

Testen bezeichnet Aktivitäten, die das Vertrauen in die Übereinstimmung des Verhaltens eines Systems mit seinem intendierten Verhalten erhöhen oder die Übereinstimmung widerlegen sollen. Das Sollverhalten liegt zumeist als informelle Spezifikation vor, anhand derer Tester eine ungefähre Vorstellung des intendierten Verhaltens bilden, ein mentales Modell, das zur Definition von Testfällen herangezogen wird. Diese in der Praxis verbreitete Herangehensweise an das Testen ist implizit, unstrukturiert, in Details unmotiviert und nicht reproduzierbar.

Idee des modellbasierten Testens ist es, mentale Modelle durch explizite Verhaltensmodelle zu ersetzen, um die Rigorosität des Qualitätssicherungsprozesses zu erhöhen. Explizite Verhaltensmodelle codieren das Sollverhalten und dienen als Referenz für das Istverhalten des zu testenden Systems. Aus dem Modell werden automatisiert Testfälle abgeleitet. Eingaben werden als Eingaben für die Implementierung verwendet; Ausgaben der Implementierung werden mit den Ausgaben des Modellablaufs verglichen. Das setzt die Überbrückung der unterschiedlichen Abstraktionsniveaus von Modell und Implementierung sowie die Validität des Modells voraus. Errechnete Testfälle dienen der manuellen Validierung des Modells und der automatischen Verifikation der Implementierung.

Die Berechnung von Testfällen beruht auf Testfallspezifikationen: auf expliziten Eigenschaften, auf zu vervollständigenden Szenarien, auf Überdeckungs- und auf stochastischen Kriterien. Häufig läßt sich das Problem der Testfallgenerierung aus Testfallspezifikationen als Suchproblem im Zustandsraum des Modells begreifen. Der präsentierte Testfallgenerator basiert auf der effizienten symbolischen Ausführung des Modells, die solange durchgeführt wird, bis der spezifizierte Zustand gefunden wird. Das Verfahren ist kompositional: Aus Testfällen für Teilsysteme werden Testfälle für komponierte Systeme und damit letztlich Integrationstests berechnet. Am Beispiel eines starken Codeüberdeckungskriteriums wird gezeigt, wie automatisch Testfälle erzeugt werden, die das Kriterium nicht nur auf Modul-, sondern sogar auf Systemebene erfüllen.

Praktische Anwendbarkeit erfordert die Einbettung in den übergeordneten Entwicklungsprozeß. Dementsprechend wird die Verzahnung der Entwicklung von Modell und Implementierung mit modellbasiertem Testen analysiert. Unter besonderer Berücksichtigung inkrementeller Entwicklungsprozesse bei sich stetig ändernden Anforderungen wird die Rolle von Verhaltensmodellen zur Überprüfung von Modelleigenschaften, als Spezifikation, als Grundlage automatisierter Codeerzeugung, zur kompositionalen Testfallgenerierung sowie als Abstraktion der Umwelt untersucht.

Die Tragfähigkeit des vorgestellten Verfahrens wird anhand einer industriellen Fallstudie, einer Chipkartenanwendung, demonstriert.

Danksagung

Dank gebührt zuvörderst Prof. Dr. Manfred Broy: für die Möglichkeit, in großer Freiheit das zu tun, was ich für interessant erachtete; für die Möglichkeit, dieses in der befruchtenden Atmosphäre einer durchaus heterogenen Forschergruppe zu tun; schließlich dafür, die Betreuung dieser Arbeit zu übernehmen. Prof. Dr. Klaus Müller-Glaser danke ich für die Übernahme des Zweitgutachtens.

Jan Philipps hat starken Einfluß auf meine informatikbezogene Entwicklung der letzten beinahe vier Jahre ausgeübt. Unzählige teils heftige Diskussionen habe ich stets als bereichernd empfunden. Meine Quintessenz vieler Gespräche findet sich in dieser Dissertation wieder.

Nicht nur Jan Philipps, sondern auch Dr. Thomas Stauner, Dr. Christian Salzmann und Jan Romberg haben Vorversionen dieser Arbeit oder Teile davon gelesen und angenehm kritisch kommentiert.

Kommunikation habe ich als wesentlichen Teil der Forschung kennengelernt. Nicht nur für häufig kontroverse Diskussionen und Einflußnahme vermutlich nicht immer in ihrem Sinn danke ich Dr. Christian Salzmann, Dr. Thomas Stauner, Prof. Dr. Tobias Nipkow, Dr. Bernhard Schätz, Dr. Oscar Slotosch, Dr. Max Breitling, Dr. Ingolf Krüger, Jan Romberg, Dr. Michael von der Beeck, Wolfgang Prenninger, Franz Huber, Peter Braun, Dr. Katharina Spies, Dr. Peter Hofmeister, Dr. Bernhard Rumpe und Christoph Kern.

Familie und Freunden gebührt der Dank, der selten explizit formuliert wird und das erfreulicherweise auch gar nicht muß.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Problemstellung . . . 3
1.2. Beitrag . . . 7
1.2.1. Methodik . . . 8
1.2.2. Technologie . . . 8
1.2.3. Pragmatik . . . 9
1.2.4. Hypothese und Annahmen . . . 9
1.2.5. Gewichtung . . . 10
1.3. Einordnung . . . 11
1.4. Gliederung . . . 11

2. Modellbasiertes Testen **15**
2.1. Fallstudie: Wireless Identity Module . . . 15
2.2. Testen . . . 20
2.2.1. Überblick . . . 20
2.2.2. Testaktivitäten . . . 25
2.2.3. Vorgehen . . . 27
2.2.4. Nichtdeterminismus . . . 30
2.2.5. Präzisierung der Terminologie . . . 33
2.3. Modelle und Modellbasierung . . . 39
2.3.1. Modelle . . . 41
2.3.2. Verschiedene Abstraktionsstufen . . . 50
2.3.3. Zusammenfassung Modellbasierung . . . 53
2.3.4. Inkrementalität, Refactorings und Regressionstests . . . 54
2.4. Modelle für Codeerzeugung, Verifikation und Validierung . . . 56
2.4.1. Codegenerierung aus einem Modell . . . 57
2.4.2. Modellextraktion aus Code . . . 59
2.4.3. Unabhängige oder simultane Entwicklung . . . 60
2.5. Andere QS-Maßnahmen . . . 62
2.6. Zusammenfassung . . . 65

3. Testfallspezifikation **69**
3.1. Eigenschaften und Testfallspezifikationen . . . 70
3.2. Funktionale Testfallspezifikationen . . . 72
3.2.1. Test von Szenarien . . . 72
3.2.2. Test universeller Eigenschaften . . . 76
3.2.3. Methodik . . . 80
3.3. Überdeckungsmaße als Testfallspezifikationen . . . 81

3.3.1. Kontroll- und Datenfluß 83
3.3.2. Abdeckung von Anforderungen 85
3.4. Stochastische Testfallspezifikationen 86
3.5. Formalismen für die Testfallspezifikation 88
3.6. Zusammenfassung 92

4. Testfallgenerierung 97
4.1. Vorgehen 98
4.2. Systemmodellierung mit AutoFocus 99
4.2.1. AutoFocus 99
4.2.2. Fallstudie 103
4.3. Übersetzung in Constraint-Logik-Programmierung 104
4.3.1. Constraint-Logik-Programmierung 105
4.3.2. Übersetzung von EFSMs 109
4.3.3. Übersetzung von Funktionen 116
4.3.4. Optimierungen 119
4.4. Testfallgenerierung mit symbolischer Ausführung 122
4.5. Instantiierung und Konkretisierung 126
4.6. Kompositionalität und Integrationstests 127
4.7. Generierung integrierter MC/DC-Testsuiten 131
4.8. Verwandte Arbeiten 142
4.9. Zusammenfassung 145

5. Effizienzsteigerung und Anwendung 149
5.1. Suchstrategien 150
5.2. Zustandsspeicherung und EF-Model Checking 156
5.3. Verwendung von Constraints 171
5.4. Experimente 172
5.4.1. Fallstudie 2: Inhouse-Karte 173
5.4.2. Suchverfahren 174
5.4.3. WIM: Funktionale und stochastische Spezifikationen . . . 180
5.4.4. WIM: MC/DC und Kompositionalität 182
5.4.5. Diskussion 184
5.5. Verwandte Arbeiten 185
5.6. Zusammenfassung 188

6. Ergebnisse und Ausblick 191
6.1. Ergebnisse 192
6.2. Zukünftige Arbeiten 196

A. Programmiersprachen: Abstraktionen und Domänenabhängigkeit 203
A.1. Abstraktionen und Programmiersprachen (2.3.1) 203
A.2. Domänenspezifische Programmiersprachen (2.3.1) 204

B. Beweise **207**
B.1. Transitionsordnungen (Algorithmus 5.2) 207
B.2. Charakterisierung der Inklusion (5.4) 208
B.3. Charakterisierung des Komplements (5.6) 211
B.4. Vollständigkeit der Zustandsspeicherung (5.10) 212

Abbildungsverzeichnis **213**

Tabellenverzeichnis **215**

Literaturverzeichnis **217**

1. Einleitung

Hersteller eines Systems sind daran interessiert, dieses System unter Berücksichtigung limitierter Ressourcen in Übereinstimmung mit den inhaltlichen Vorstellungen eines Auftraggebers zu entwickeln. Relevante Faktoren bei der Systemerstellung sind somit (1) Kosten, (2) Dauer und (3) Qualität: (3a) die präzise Formulierung der gewünschten Systemfunktionalität (Lasten- und Pflichtenheft) und (3b) die Übereinstimmung der Funktionalität des entwickelten und des erwünschten Systems.

Eine schwerwiegende Abweichung von Soll- und Istverhalten ist nach Möglichkeit zu vermeiden. Systeme, von deren einwandfreiem Verhalten Menschenleben abhängen können (Airbags, Bremssysteme), und solche, deren Fehlverhalten zu Imageschäden (Chipkarten) und kostspieligen Rückrufaktionen (Benzinpumpen in Automobilen) führen, illustrieren das eindringlich.

Inhalt dieser Arbeit ist ein systematischer Ansatz zur Bewertung des Soll-Verhaltens und der Übereinstimmung von Soll- und Ist-Verhalten eines Systems, dem modellbasierten Testen. Eingebettet in den übergreifenden Entwicklungsprozesses werden Modelle eines Systems erstellt. Dieses geschieht zu verschiedenen Zwecken vor, nach oder zeitgleich mit der Systementwicklung. Als vereinfachte Abbilder eines Systems eignen sich Modelle eher als die konkreten Systeme zur automatisierten Berechnung von Testfällen, d.h. Sequenzen von Stimuli und erwarteten Reaktionen. Diese Testfälle werden zur Überprüfung des Modells selbst – dem Soll-Verhalten – und nach geeigneten Transformationen zur Bewertung des Systems – dem Ist-Verhalten – eingesetzt. Ein Anspruch auf Vollständigkeit der Überprüfung wird nicht erhoben.

Der erste und zweite o.g. Faktor, Kosten und Zeit, werden im Gegensatz zur Spezifikation des Soll-Verhaltens als Teil des Requirements Engineering und im Gegensatz zum Nachweis der Übereinstimmung von Soll- und Ist-Verhalten in der vorliegenden Arbeit nicht umfassend studiert. Solche Untersuchungen bedürfen umfangreicher statistischer Erhebungen und empirischer Auswertungen, die den Rahmen dieser Dissertation sprengen würden.

Testen, Verifikation, Validierung Die praktische Relevanz des Testens mit bis zu 50% des Gesamtentwicklungsaufwands ist weithin anerkannt.[1] Die stiefmütterliche Behandlung des Themas in universitären Curricula mag – bei deutlichen Anzeichen einer Sensibilisierung – ein Grund für die in Industriekooperationen wahrgenommene unsystematische Behandlung dieses Themenkomplexes in der Industrie sein. Der verstärkte Einsatz von Software in allen Bereichen legt

[1] Vgl. z.B. Myers (1979), Beizer (1983), Balzert (1998). Fagan (2002, S. 569) vermutet, daß die Hälfte der „Testaktivitäten" der Fehlerkorrektur zuzurechnen seien.

den Schluß nahe, daß Qualitätssicherungsmaßnahmen aktiver (z.B. Testen, mathematische Verifikation, Inspektionen) und passiver (prozeßbezogener) Natur zunehmend wichtiger werden.

Testen, verstanden als Aktivität des Ausführens von Programmen zum Zweck (1) des Findens von Fehlern und (2) der Überprüfung der Übereinstimmung von Soll- und Ist-Verhalten kann nach Dijkstras berühmtem Diktum allein die Anwesenheit von Fehlern zeigen (Dijkstra, 1970, S. 7). Der scheinbare Widerspruch zwischen Testen als Aktivität der Überprüfung der Übereinstimmung und der allgemein akzeptierten Definition, Testen sei der Vorgang der Programmausführung zum Zweck der Fehlererkennung (Myers, 1979), wird in Abschnitt 2.2.1 aufgelöst.

Zweck der *Verifikation* ist die Überprüfung der Übereinstimmung zweier formaler Dokumente, z.B. in bezug auf das codierte Verhalten.[2] Dabei spielt es keine Rolle, ob diese Dokumente ausführbar sind. Ziel des Testens ist eine Approximation des Nachweises der Übereinstimmung einer Implementierung mit ihrer spezifizierten Funktionalität. Diese kann maschinell operationalisierbar abgelegt sein oder nicht – im ersten Fall ist Testen eine Aktivität der Verifikation, im zweiten eine Aktivität der *Validierung*. Der Unterschied besteht in der Formalisierung und Operationalisierbarkeit des Anforderungsdokuments. Ist dieses maschinell nutzbar, können Verdikte, d.h. Bewertungen der tatsächlichen Ausgabe, automatisch erzeugt werden. Ist es das nicht, wird dafür menschliche Intelligenz benötigt. Das formale Dokument, das das gewünschte Verhalten codiert, wird i.a. als *Spezifikation* bezeichnet. Unter dem zweiten formalen Dokument, dem tatsächlichen System, wird i.a. eine *Implementierung* verstanden. Testen dient somit der Überprüfung der Übereinstimmung, der *Conformance*, von Spezifikation und Implementierung. Testen kann aber auch gezielt zur Approximation des Nachweises bestimmter Eigenschaften eingesetzt werden.

Formale Verifikation mit Techniken wie z.B. Model Checking oder deduktivem Beweisen kann die Anwesenheit von Fehlern allein in bezug auf eine abstrakte Programmsemantik aufzeigen. Prinzipiell kann diese Semantik zwar beliebig detailliert sein. Sie ist aber ein mathematisches Konstrukt und damit ein im Normalfall vereinfachendes Abbild von Phänomenen der Realität. Die Realität der formalen Verifikation ist die mathematischer Objekte, und aus Verifikationsbemühungen resultierende Aussagen betreffen zunächst die Mathematik (was je nach Formalisierungsgrad der Anforderungsdokumente auch andere Techniken der Verifikation, z.B. das Testen, betrifft – getestet wird aber ein tatsächlich ausgeführtes Programm). Formale Verifikation ist als Ergänzung zum Testen, das auch in der Realität durchgeführt wird, durchaus wertvoll, da es logische Fehler oder Auslassungen in der Abstraktion eines System oder einem Algorithmus nachweisen kann. Gerade der Bereich eingebetteter Systeme, in dem bei der Verifikation von großen Teilen der Umwelt abstrahiert wird, zeigt aber die Notwendigkeit beider Aktivitäten, vgl. Fetzer (1988), Paulson u. a. (1989), DeMillo u. a. (1977), Peled u. a. (1999) und Dijkstra (1970, S.19-

[2] Verschiedene Definitionen der Termini „Verifikation“ und „Validierung“ in den Standards ISO 1059-1993, ISO/DIN 9000:2000, ANSI/IEEE 729-1983 und DO-178B werden von Huber u. a. (2001, Kap. 7.1) diskutiert und verglichen.

20). Verifikation kann die Korrektheit eines Protokolls zeigen. Rückschlüsse auf die Korrektheit einer Implementierung des Protokolls sind je nach Ausprägung der Umweltannahmen schwierig.

Viele Formen des Testens erfordern Redundanz in den Artefakten, die bei der Systementwicklung entstanden sind; ohne Redundanz gibt es keine Vergleichsmöglichkeit. Ausnahmen sind Verfahren, die nach syntaktisch identifizierbaren Fehlerquellen suchen. Beizer (1983, S.12) beschreibt Testen als das „Schaffen, Auswählen, Explorieren und Revidieren (mentaler) Modelle". Binder (1999, S. 111) formuliert das plakativer: „Testen muß modellbasiert sein", und er bezieht sich dabei auf explizite Modelle, die nicht nur als ungefähre Vorstellungen der Tester existieren. Die Notwendigkeit dieser Redundanz wird von Anwenderseite verkannt, wenn aus der Existenz eines zur Testfallgenerierung geeigneten Verhaltensmodells aus einer ökonomischen Perspektive nicht ganz zu Unrecht gefolgert wird, dann könne aus diesem Modell auch gleich Produktionscode erzeugt werden. Dieser Wunsch ist die Motivation für die Untersuchung verschiedener Szenarien der zeitlichen Abfolge von Modell- und Systementwicklung.

Die Kernfrage des modellbasierten Testens ist, ob es vom Aufwand her günstiger ist, Modelle zu Generierung von Testsequenzen einzusetzen, oder Testsequenzen direkt zu erstellen, wie dies etwa im Extreme Programming (Beck, 1999) propagiert und auch von entsprechenden Bibliotheken (z.B. JUnit) unterstützt wird. Die These ist, daß Modelle nicht nur hilfreich sind, sondern sogar unabdingbar, wenn keine andere vollständige operationalisierbare Beschreibung des Sollverhaltens vorliegt. Ohne empirische Untersuchungen läßt sich die Frage nach der Abschätzung des Aufwands nicht klären. Mit der vorliegenden Arbeit werden die Konzepte modellbasierten Testens vorgestellt, um einen Grundstein für solche Untersuchungen zu legen.

Zusammenfassend ist festzuhalten:

- Unter Verifikation wird der Nachweis der Übereinstimmung zweier formaler Dokumente verstanden. Verifiziert werden Eigenschaften von Modellen oder die Übereinstimmung einer Implementierung mit ihrer Spezifikation.
- Validierung bezeichnet die Überprüfung der Übereinstimmung eines formalen Dokuments mit den tatsächlichen Anforderungen, d.h. der durch den Auftraggeber gewünschten Funktionalität. Tatsächliche Anforderungen können sich von den in Lastenheften oder Spezifikationsdokumenten notierten Anforderungen unterscheiden.
- Testen bezeichnet die Menge der Aktivitäten, die letztlich der im Normalfall unvollständigen Verifikation der Übereinstimmung des Verhaltens einer Implementierung in ihrer realen oder simulierten Umwelt mit ihrer Spezifikation dienen.

1.1. Problemstellung

Das Problem, zu dessen Lösung diese Arbeit Beiträge in technologischer und methodischer Hinsicht leistet, ist das folgende:

Wie können abstrakte Verhaltensmodelle, z.B. gegeben durch erweiterte endliche Zustandsmaschinen, validiert werden (Eingabedatengenerierung für Modelle)? Wie und unter welchen Umständen können abstrakte Verhaltensmodelle eingesetzt werden, um Systeme in ihrer realen oder simulierten Umgebung zu testen (Testfallgenerierung für Modelle und Konkretisierung der Testfälle)? Welche Konsequenzen ergeben sich daraus für den Entwicklungsprozeß?

Bevor die Beiträge dieser Arbeit skizziert werden, soll eine Präzisierung der Problematik – und damit ihres Titels – erfolgen.

Systemklasse

Die betrachtete Systemklasse ist die ereignis- und zeitdiskreter reaktiver Systeme mit vergleichsweise einfachen zugrundeliegenden Datenmodellen. In dieser Dissertation werden ausschließlich Softwaretests betrachtet. Auf die speziellen Probleme beim Test von Hardware wird nicht eingegangen. Der Test weicher und harter Echtzeitanforderungen wird ebenfalls nicht betrachtet (vgl. den Ausblick in Abschnitt 6.2).

Reaktive Systeme Reaktive Systeme sind Systeme, die potentiell unendlich lange laufen und deren Funktionalität im Gegensatz zu transformativen Systemen dahingehend charakterisiert ist, daß nicht nur ein Eingabedatum zu einem Ausgabedatum führt, sondern Sequenzen von Eingabedaten zu Sequenzen von Ausgabedaten. Reaktive Systeme besitzen insbesondere einen internen Zustand, der durch das Anlegen unterschiedlicher Eingabesignale modifiziert werden kann. Die besondere Schwierigkeit im Vergleich mit transformativen Systemen liegt also darin, daß der Datenraum um eine weitere Dimension erweitert wird, die der Zeit nämlich. Reaktive Systeme können offen oder geschlossen sein. Beispiele für letztere sind Protokolle bzw. Protokollspezifikationen. Beispiele für erstere sind eingebettete Systeme wie Vorflügelsteuerungen oder Airbag-Steuersysteme, aber auch Komponenten von Businessinformationssystemen wie Webserver. Modelle reaktiver nebenläufiger Systeme werden im Rahmen dieser Arbeit mit dem Modellierungswerkzeug AutoFocus beschrieben (Huber u. a., 1997).

Unter Systemen wird, wenn nicht explizit anders vermerkt, das zu entwickelnde oder testende Artefakt und nicht seine Umwelt verstanden. Später wird präziser zwischen „Modell“ und „Maschine“ differenziert.

Datenmodellierung Businessinformationssysteme sind häufig ebenfalls reaktive Systeme. Von der unscharfen Domäne der eingebetteten Systeme lassen sie sich in bezug auf die Komplexität der erforderlichen Daten- und Verhaltensmodellierung abgrenzen. Eingebetteten Systemen scheint bei einfacheren Datentypen eine komplexere Funktionalität innezuwohnen. Der Test von Datenmodellen ist nicht Teil dieser Arbeit; einfache Summen- und Produkttypen werden als ausreichend vorausgesetzt. Insbesondere wird der Test objektorientierter oder -basierter Datenstrukturen nicht behandelt. Zur Verhaltensmo-

dellierung werden kommunizierende erweiterte endliche Zustandsmaschinen des Modellierungstools AutoFocus eingesetzt.

Diskrete Systeme Diese Arbeit fokussiert auf zeit- und wertdiskrete Systeme. Viele methodische und technologische Resultate lassen sich auch auf zeit- und wertkontinuierliche Systeme übertragen, die natürlich auf digitaler Hardware ebenfalls eine diskrete Natur aufweisen. Der Zustandsraum solcher Systeme ist aber üblicherweise noch größer als der diskreter Systeme. Im Ausblick in Abschnitt 6.2 wird skizziert, wie eine Testfallgenerierung für gemischt kontinuierlich-diskrete und Realzeitsysteme erfolgen kann.

Determinismus Die Technologie zur Testfallgenerierung basiert auf deterministischen Modellen. Der konzeptionelle, methodische und terminologische Rahmen hingegen ist so allgemein halten, daß er auch auf nichtdeterministische Systeme angewendet werden kann. Aus Gründen der intellektuellen Beherrschbarkeit gerade im Zusammenhang mit reproduzierbarer Ausführbarkeit von Modellen wird die Verwendung deterministischer Modelle propagiert, wann immer das möglich ist.

Nichtdeterminismus betrifft nicht nur das System selbst, sondern auch seine Umwelt. Wenn die Umgebung sich nichtdeterministisch verhält (was natürlich zugelassen werden muß), hat das Konsequenzen für die Notation von Testfällen. Die Schwierigkeit des Tests wird offenbar, wenn ein Teilsystem in seiner nichtdeterministischen Umgebung getestet werden soll und die Testfälle in das kombinierte System eingespielt werden. In diesem Fall kontrollieren die Testfälle also nur einen Teil der Umgebung. Es kann dann passieren, daß sich das System anders verhält, als dies durch den Testfall antizipiert wird, und dennoch ist das Verhalten korrekt. Konsequenz für die Notation von Testfällen ist, daß Testsequenzen, definiert als eine endliche Folge von Ein- und Ausgabepaaren, nicht ausreichend sind, sondern zusätzlich Fallunterscheidungen zulassen müssen. Das kann mit bewährten Notationen wie TTCN-3 oder Sequenzdiagrammen geschehen, oder mit äquivalenten Zustandsmaschinen. Für den Test müssen dann geeignete Abstraktionen gefunden werden, um den Nichtdeterminismus beispielsweise bei der Prozeßkommunikation handhaben zu können. Im Fall gemischt kontinuierlich-diskreter oder von Realzeit-Systemen ist häufig eine Abstraktion erforderlich, die das Modell nichtdeterministisch macht (Abschnitt 6.2).

Anwendungsdomäne Die vorgestellten Verfahren sind für eine Vielzahl von Anwendungsdomänen anwendbar. Die Wahl von AutoFocus als Modellierungstool legt eine Charakterisierung der Anwendungsdomäne auf (1) monolithische eingebettete Systeme wie Chipkarten – in denen der nebenläufige Komponentenbegriff im wesentlichen zur funktionalen Dekomposition verwendet wird – und (2) nebenläufige, taktsynchron kommunizierende sicherheitskritische Systeme wie X-by-wire-Anwendungen mit time-triggered-Architekturen (Kopetz und Bauer, 2001; Rushby, 2001) nahe.[3] Die präsentierten Verfahren

[3]Allgemeiner bieten sich alle zeitsynchronen Bussysteme wie z.B. auch der MOST-Bus (MOST Cooperation, 2001) an.

zur Testfallgenerierung eignen sich damit für alle Systeme, die adäquat mit AutoFocus modelliert werden können. Die potentielle Übertragung auf andere Modellierungssprachen wird diskutiert.

Testen

Zum Zweck der Validierung und Verifikation von ausführbaren Systemen und ihren ebenfalls ausführbaren Abstraktionen, d.h. Verhaltensmodellen, müssen im Fall der Validierung Sequenzen von Eingaben und im Fall des Tests als Aktivität der Verifikation Sequenzen von Ein-/Ausgabepaaren erzeugt werden. Wie können solche Sequenzen effektiv und effizient nicht nur für Module – atomare Strukturen, z.B. *modules* in Modula –, sondern auch das Gesamtsystem bestimmt werden? Die Frage nach der Charakterisierung „sinnvoller“ Mengen von Testsequenzen, d.h. die Frage nach der Testfallspezifikation, wird unter der Annahme diskutiert, daß das Testziel vorgegeben ist. Testziele, d.h. Beschreibungen einer zu überprüfenden Eigenschaft, sind funktionaler, struktureller oder stochastischer Natur; Testfallspezifikationen ihre operationalisierbare Formalisierung. Die außerordentlich schwierige und im Bereich des Testens zentrale Frage, was einen „guten Testfall“ i.a. auszeichnet – mit großer Wahrscheinlichkeit potentielle Fehler zu finden – wird also nicht umfassend beantwortet. Die Existenz einer allgemeinen Antwort auf diese Frage, die im Bereich der formalen Verifikation ebenfalls in Form der Frage nach zu verifizierenden Eigenschaften (Putativtheoreme im Sinn von Rushby (1995)) auftritt, ist unwahrscheinlich.

Funktionalität

Funktionalität ist eine Frage der Perspektive, wie die Diskussion um funktionale vs. nicht-funktionale Anforderungen demonstriert. Das zeigt das Beispiel der zeitlichen Anforderungen an ein System, die klassisch als nichtfunktional charakterisiert werden, im Bereich der Echtzeitsysteme aber natürlich eine funktionale Anforderung darstellen. Die zu testende Funktionalität beschränkt sich in dieser Arbeit auf die Ablauflogik des Systems, d.h. auf eine mehr oder weniger abstrakte Kausalität. Wenn zeitliche Anforderungen explizit in das Modell aufgenommen werden, werden sie als funktional interpretiert. Nicht unter die Funktionalität fallen Aspekte wie Wartbarkeit oder Benutzbarkeit, die nur sehr schwierig mit den Modellierungstechniken von AutoFocus formalisiert werden können. Lasttests werden ebenfalls nicht betrachtet.

Modellbasierung

Der verwendete Modellbegriff wurde oben bereits kurz skizziert. Modelle sind Abstraktionen eines Systems, die einem dedizierten Zweck dienen. Für die Beschreibung von Modellen ist eine Sprache notwendig. Im Rahmen dieser Arbeit werden dafür die Sprachen des Werkzeugs AutoFocus verwendet. So beschriebene Modelle werden zur Validierung des Modells selbst verwendet (Erzeugung von Eingabesequenzen) und für die Verifikation eines tatsächlichen Systems. Der Einfluß der Modellierungssprache und der Komplexität der zugrundeliegenden Semantik auf die Testfallgenerierung wird diskutiert. Aus pragmatischer

Sicht erfordert dies eine Diskussion des Wesens modellbasierter Entwicklung, die skizzenhaft erfolgt. Dabei werden die Schwierigkeiten erläutert, die sich aus der Integration verschiedener Abstraktionsebenen auf Modellierungs- und auf Testfallebene ergeben.

Entwicklungsprozeß

Der erfolgte Einsatz der im Rahmen dieser Arbeit entwickelten Werkzeuge und Methoden in industriellen Kooperationen legt nahe, daß ihr abteilungsweiter Einsatz mit tiefgreifenden organisatorischen, prozeßbezogenen und den Schulungsstand der Mitarbeiter betreffenden Änderungen einhergeht. Die politische Natur entsprechender Entscheidungen birgt Unwägbarkeiten, die ohne empirische Untersuchungen des Kosten-Nutzen-Verhältnisses nicht geeignet bewertet werden können. Unter Vernachlässigung dieses entscheidenden politischen Kriteriums müssen weitere Einflußfaktoren – wie beispielsweise die Verfügbarkeit industriell einsetzbarer CASE-Werkzeuge mit domänenspezifischen Beschreibungstechniken – untersucht werden. Obwohl die außerordentliche Relevanz dieser Faktoren nicht bestritten wird, kann eine solche Untersuchung hier naturgemäß nicht umfassend geschehen. Die Überzeugung, daß akademische Arbeiten im Software-Engineering an der Schnittstelle von Theorie und Praxis zielgerichtet auf langfristige Verbesserungen des Produkts oder des Prozesses hinauslaufen sollten, macht eine auch unvollständige Berücksichtigung solcher Faktoren allerdings notwendig. Das beinhaltet sowohl den Gesamtentwicklungsprozeß als auch, spezieller, den Vorgang der Modellierung.

1.2. Beitrag

Die Behandlung der o.g. Problemstellung gliedert sich grob in Beiträge und Erkenntnisse methodischer, technologischer und pragmatischer Art.

Ausgangspunkt der in dieser Arbeit angestellten Überlegungen ist die Beobachtung, daß Testingenieure

- nach der Lektüre von Spezifikationsdokumenten, sofern solche vorhanden sind oder
- durch Gespräche mit Entwicklern, wenn Spezifikationsdokumente nicht vorhanden sind,

eine ungefähre Vorstellung des Sollverhalten eines Systems entwickeln, ein mentales Modell also. Dieses mentale Modell dient dann als Grundlage für die Definition von Testfällen. Der implizite Charakter des Modells führt dazu, daß der Prozeß der Testdefinition i.a. nicht reproduzierbar, unstrukturiert und in seiner Qualität stark von der Person des Testers abhängig ist. Explizite, widerspruchsfreie, möglichst vollständige und ausführbare Modelle des Systems stellen eine Alternative dar, im Rahmen derer die o.g. Probleme abgemildert werden können.

Kern der Ergebnisse ist ein Werkzeug, das sich wie folgt zur Generierung von Testfällen eignet. Ein AutoFocus-Modell eines Systems wird in eine Constraint-Logik-Programmiersprache übersetzt, deren Prinzipien in Abschnitt 4.3.1 kurz

vorgestellt werden. Eine Testfallspezifikation funktionaler, struktureller oder stochastischer Natur, die in Form einer Zustandsmaschine, von Sequenzdiagrammen oder eingeschränkten temporalen Formeln gegeben ist, wird ebenfalls übersetzt. Die Konjunktion beider Systeme wird dann zum Zweck der weiteren Einschränkung des Suchraums mit Constraints angereichert, die Umweltannahmen oder explizite Suchraumbeschränkungen modellieren. Der entwickelte Werkzeugprototyp berechnet dann Testfälle, d.h. Repräsentationen von Mengen von Testsequenzen, die instantiiert werden und somit Sequenzen darstellen, die der Testfallspezifikation genügen. Die Testfälle sind eine gemäß der Testfallspezifikation ausgewählte Teilmenge des erwünschten Systemverhaltens. Diese Sequenzen können zur manuellen Validierung des Modells eingesetzt werden und nach geeigneten Konkretisierungen auch für den automatischen Test eines tatsächlichen Systems. Die Testfallgenerierung ist somit (a) im Requirements Engineering zur Validierung des Modells und (b) zu Verifikationszwecken der Implementierung einsetzbar.

1.2.1. Methodik

Die industrielle Anwendbarkeit neuer Technologien ist immer abhängig von deren Einbettung in den Entwicklungsprozeß und den verwendeten Beschreibungstechniken. Die Erzeugung von Ablaufsequenzen für reaktive Systeme wird dementsprechend in einen inkrementellen Entwicklungsprozeß mit simultanem Entwurf von Systeminkrementen und Testfällen eingebettet. Das beinhaltet sowohl Hilfestellung für Simulationen im Requirements Engineering als auch die Erzeugung von Testsequenzen für Systeme in ihrer tatsächlichen Umgebung. Der Zusammenhang von inkrementellem Design und Regressionstests wird verdeutlicht. Integrationstests werden kompositional aus Testfällen generiert, die für Subsysteme erzeugt wurden. Diskutiert und bewertet wird die Erstellung von Modellen vor, nach und während der Entwicklung von Code. Kriterien für den erfolgreichen Einsatz bestimmter Modellierungstechniken und -sprachen werden ebenfalls berücksichtigt.

1.2.2. Technologie

In technologischer Hinsicht lassen sich zwei Hauptergebnisse ausmachen, die sich auf Suchverfahren und die Verwendung von Constraints beziehen:

Suchverfahren Zum einen handelt es sich um die Integration gerichteter Suchverfahren aus der Künstlichen Intelligenz (best-first search) in Technologien, die Model-Checking und symbolische Ausführung verquicken (Pretschner, 2001; Pretschner u. a., 2003a). Die Integration gerichteter Suche in die Technologie des explicit-state Model-Checking erfolgte unabhängig und zeitgleich durch Edelkamp u. a. (2001) für den Model Checker Spin (Holzmann, 1997). Diese Suchverfahren sind insbesondere für die im Rahmen dieser Arbeit entwickelten Verfahren zur automatischen Erzeugung von Integrationstests notwendig.

Constraints Zum anderen handelt es sich um die Verwendung von Constraints für Zustandsraumexplorationen im Rahmen der symbolischen Ausführung. Viele Testziele lassen sich als existentielle Eigenschaften formulieren („finde Zustand σ“), und dementsprechend wird eine darauf zugeschnittene Model Checking-Technik vorgestellt. Gerechnet wird nicht mit einzelnen Daten, sondern mit Mengen von Daten. Constraints finden neben der symbolischen Ausführung auch bei der Spezifikation von Testfällen und dem Speichern von Zustandsmengen Verwendung. Für viele Systeme führt dies zu signifikanten Effizienzsteigerungen. Nutzen wird darüberhinaus aus der gewählten technologischen Infrastruktur (Constraint-Logikprogrammierung) für neuartige Verfahren zur Generierung von Integrationstests gezogen.

Formale Verifikationstechniken Wenn das vorgestellte Verfahren zur Generierung von Testfällen allein auf Modellebene angewendet wird, also als eine Aktivität des Requirements Engineering angesehen wird, dann ist es formalen Verifikationstechniken wie Model Checking oder deduktivem Beweisen durchaus vergleichbar. Der bewußte Verzicht auf Vollständigkeit – Eigenschaften des Modells werden nicht nachgewiesen, sondern der Nachweis wird durch Testfälle approximiert – führt dazu, daß auch sehr große Systeme überprüft werden können, was für die anderen Techniken z.Z. noch nicht möglich ist. Testen wird nicht als Ersatz für Model Checking oder formales Beweisen angesehen, sondern als komplementierende Aktivität (vgl. Sharygina und Peled (2001)).

1.2.3. Pragmatik

Zur Validierung des Ansatzes wurde ein in Industriekooperationen erfolgreich eingesetzter Werkzeugprototyp zur Erzeugung von Testfällen implementiert. Teil dieser Arbeit ist ein System, das in der Lage ist, aus einem Modell vollautomatisch nicht nur Testfälle für eine gegebene explizite, funktionale Testfallspezifikation zu errechnen, sondern auch Testfälle, die von Standards wie DO-178B (RTCA und EUROCAE, 1992) geforderten Überdeckungskriterien genügen und damit einen wesentlichen Teil der Vorbereitung des Zertifizierungsprozesses z.B. in der Avionik vereinfachen können.

1.2.4. Hypothese und Annahmen

Eine Arbeit im Bereich des Software-Engineering, an deren Ausgangspunkt explizit der Wunsch nach der Synthese theoretischer und praktischer Inhalte stand, bedarf eigentlich einer empirischen Validierung der vorgeschlagen Methoden und Materialien. Unter ökonomischen Gesichtspunkten müßten die präsentierten Techniken der Testfallgenerierung mit manueller Testfallerstellung und alternativen Techniken wie Reviews verglichen werden. Das leistet diese Arbeit nicht. Grund dafür ist nicht nur die fundamentale Schwierigkeit, die Qualität einer Testsuite zu bemessen, was für den Vergleich der vorgeschlagenen automatischen Testfallableitung mit manuellen Techniken notwendig wäre. Auch die Notwendigkeit der Existenz einer für Experimente geeigneten, ausreichend großen und zum Zweck der Vergleichbarkeit homogenen Grundmenge von zu

testenden Systemen und ihren Testfällen stellt eine prinzipielle Schwierigkeit dar. Insbesondere dann, wenn Metriken wie Kosten oder zeitlicher Aufwand zur Bewertung eines Ansatzes im Vergleich zu anderen Anwendung finden sollen, ergeben sich große Schwierigkeiten. Obwohl die Notwendigkeit solcher empirischer Analysen also durchaus eingesehen wird – und beim Verfasser insbesondere die Überzeugung herrscht, daß Arbeiten im ingenieurmäßigen Bereich des Software Engineering konstruktiv und mit einer pragmatischen zielgerichteten Ausrichtung zur Behebung eines Mißstandes erfolgen sollten –, wird im Rahmen dieser Arbeit darauf verzichtet. Schon der Einsatz systematisierter Qualitätssicherungsmaßnahmen, wie sie hier propagiert werden, ist in seinem positiven Einfluß auf die Produktqualität allgemein anerkannt. Die zentrale Hypothese dieser Arbeit wird also sorgfältig diskutiert, aber, abgesehen von Erfahrungen aus exemplarischen industriellen Kooperationen, nicht empirisch unterfüttert:

Die systematische Verwendung abstrakter Verhaltensmodelle für die Qualitätssicherung führt zu höherem Vertrauen in ihre Korrektheit. Das beinhaltet nicht nur die unscharf faßbare intellektuelle Durchdringung des Problems, sondern auch die konkrete Anwendung für die systematische Testfallerstellung. Systematisierung des Qualitätssicherungsprozesses als solche führt zu erhöhter Produktqualität. Der Systementwicklungsprozeß und die darin angewandten Formalismen beeinflussen den Testprozeß unmittelbar.

Die vorliegende Arbeit geht auf die Problematik der Verwendung von Modellen, also Abstraktionen, in der Systementwicklung ein. Das zentrale Problem nicht nur des modellbasierten Testens, sondern allgemein der modellbasierten Entwicklung, ist: Entwickler wünschen einerseits abstrakte Beschreibungen ihres Systems, andererseits muß das System aber auch in seiner tatsächlichen Umgebung lauffähig sein und damit außerordentlich konkret. Berücksichtigt werden müssen also die Integration verschiedener Abstraktionsebenen z.B. durch architektonische Schichtungen. Es wird anhand der Fallstudie gezeigt, daß das für die Testfallgenerierung für Chipkarten möglich ist. Hoffnung ist, daß die Integration verschiedener Abstraktionsebenen auch für andere Systemklassen möglich ist, insbesondere solche mit Realzeitanforderungen.

Anspruch und Nachweis Dementsprechend nimmt die Arbeit für sich in Anspruch, einen allgemeinen methodischen und technologischen Rahmen für den modellbasierten funktionalen Test reaktiver Systeme zu liefern. Das beinhaltet die Anwendbarkeit nicht nur für akademische Beispiele, sondern auch in der industriellen Praxis. Der formulierte Anspruch wird insoweit belegt, als (a) die Tragfähigkeit an der Fallstudie der Chipkarte demonstriert, (b) auf Spezifika der Fallstudie explizit hingewiesen und (c) mögliche Verallgemeinerungen dieser Besonderheiten analysiert werden.

1.2.5. Gewichtung

Nach Ansicht des Verfassers stellen

- in methodischer Hinsicht Abschnitt 2.4 über verschiedene Szenarien des modellbasierten Testens, Abschnitt 3.2.2 über den schematischen Test universeller Eigenschaften und die Abschnitte 4.6, 4.7 und 2.3.4 über kompositionale Testfallerzeugung sowie
- in technologischer Hinsicht Abschnitt 5.1 über die Einführung gerichteter Suchverfahren in Model-Checking-artige Technologien und Abschnitt 5.2 über Zustandsspeicherstrategien für die symbolische Ausführung

die wesentlichen Beiträge dieser Arbeit dar.

1.3. Einordnung

Thema dieser Arbeit ist das Testen. Wie sich herausstellen wird, beeinflußt das Testen viele Aktivitäten der Systementwicklung. Das betrifft

- das Requirements Engineering, denn in iterativen Prozessen ist die Modellerstellung sowie die Ableitung systemrelevanter zu testender Eigenschaften Teil dieser Aktivität sowie
- im Kontext der Modellbasierung auch Entwurfsmethodik, Design und Implementierung, denn für die Erstellung von Verhaltensmodellen bedarf es adäquater Sprachen. Aus in solchen Sprachen notierten Modellen werden dann Code und Testfälle manuell oder automatisiert abgeleitet.

In technischer Hinsicht ergeben sich für den präsentierten Testfallgenerator Überschneidungen mit der Technik des Model Checking, auf deren Vollständigkeit aber bewußt verzichtet wird. Die Implementierung erfolgt mit Techniken der funktionalen, logischen und Constraint-Programmierung. In Anwendungshinsicht schließlich ergibt sich durch die betrachtete Fallstudie der Chipkarte eine Überschneidung mit dem großen Gebiet der Prozessorverifikation und Prozessorvalidierung.

Grundkenntnisse der Konzepte und Terminologie des Testens, des Model Checking und der temporalen Logik werden vorausgesetzt. Die Vielzahl der berührten Gebiete würde bei einer ausführliche Einführung in jedes einzelne den Umfang dieser Dissertation sprengen. Die Literaturangaben verweisen auf relevante Referenzen.

1.4. Gliederung

Diese Dissertation besteht aus zwei Teilen, einem methodisch und einem technologisch orientierten. Der methodische Teil (Kap. 2 und 3) beschreibt die Grundidee des modellbasierten Testens. Das beinhaltet den terminologischen Rahmen, die Einbettung des modellbasierten Testens in den übergeordneten Entwicklungsprozeß, verschiedene Szenarien des modellbasierten Testens, eine Analyse verschiedener relevanter Modellformen incl. Anforderungen an ihre Beschreibung und die Frage, wie zu testende Eigenschaften abgeleitet und spezifiziert werden. Dieser in Teilen bewußt skizzenhaft gehaltene Abschnitt formu-

liert insbesondere Kriterien, die zu einem erfolgreichen Einsatz der vorgestellten Technologie notwendig erscheinen.

Der technologische Teil (Kap. 4 und 5) beschreibt einen Testfallgenerator, der für auf deterministischen erweiterten Zustandsmaschinen und funktionalen Programmen basierenden Modellen Testfälle berechnet. Es wird gezeigt, wie aus Testfällen für einzelne Komponenten Testfälle für integrierte Systeme generiert werden können. Effizienzsteigerungen durch Strategien zur Zustandsspeicherung und zur gerichteten Suche werden definiert und experimentell belegt.

Im einzelnen gliedert sich die Arbeit wie folgt:

Kapitel 2, Modellbasiertes Testen, beginnt in Abschnitt 2.1 mit einem Überblick über die im Rahmen dieser Arbeit durchgeführte Fallstudie, das wireless identity module (3GPP, 2001) einer Chipkarte. Diese Studie wurde in Zusammenarbeit mit den Firmen Validas AG und Giesecke&Devrient GmbH erstellt, die das Modell und die technologische Infrastruktur zur Testdurchführung zur Verfügung stellten. Anhand der Fallstudie werden im Verlauf der Arbeit alle wesentlichen Konzepte illustriert. Abschnitt 2.2 gibt eine Einführung in die Konzepte und Aktivitäten des Testens, liefert eine Präzisierung der Terminologie und beschreibt die zentralen Probleme. In Abschnitt 2.3 wird das für die Arbeit wesentliche Konzept des Modells erläutert. Das schließt eine Skizze des Verständnisses modellbasierter Entwicklung sowie die inkrementelle Entwicklung von Modellen und den Zusammenhang mit Regressionstests ein. Motivation hierfür ist die Notwendigkeit, Modelle zu erstellen, die zur Grundlage von Testaktivitäten gemacht werden können. Dieser Abschnitt beinhaltet auch eine allgemeine Diskussion der fundamentalen Schwierigkeit modellbasierter Ansätze, gleichzeitig abstrakt – zum Zweck der intellektuellen Beherrschbarkeit – und konkret – zum Zweck der Code- und Testfallerzeugung – zu sein. In Abschnitt 2.4 werden verschiedene Szenarien des modellbasierten Testens diskutiert. Untersucht wird die Reihenfolge der Erstellung von Modellen und dem tatsächlichen System, die gleichzeitig oder zeitlich nacheinander erfolgen kann.

Kapitel 3, Testfallspezifikation, behandelt das Problem der Suche nach geeigneten Testzielen und das Problem der Spezifikation von Testfällen. Dieses schwierige Problem, das direkt die Schwierigkeiten geeigneter Metriken für die Qualität einer Testsuite – und damit Testendekriterien – widerspiegelt, kann z.Z. vermutlich nicht umfassend und allgemein gelöst werden. Nach einer einführenden Diskussion der Problematik werden in Abschnitt 3.2 funktionale Testfallspezifikationen untersucht. Dabei wird unterschieden zwischen zu testenden universellen Eigenschaften wie Invarianten, die existentiell approximiert werden müssen – explizit oder durch Überdeckungsmaße auf Automaten, die Übersetzungen der zu testenden Eigenschaft sind – und Szenarien, die sich aus den Anforderungen ergeben. Abschnitt 3.3 diskutiert daten- und kontrollflußbezogene Überdeckungsmaße, die nicht nur zur Messung der Qualität einer Testsuite verwendet werden, sondern auch als Testfallspezifikation dienen können. Der Abschnitt beinhaltet außerdem einen Ansatz zur Erzeugung von

Testfällen, die spezielle Anforderungen abdecken. Idee dabei ist, alle Elemente eines AutoFocus-Modells mit Absätzen der Anforderungsdokumente, wie sie beispielsweise mit DOORS erstellt werden, zu annotieren, und Überdeckung für diese Annotationen zu verlangen und zu errechnen. Abschnitt 3.4 behandelt die stochastische Erzeugung von Testsuiten, die rein zufällig oder anhand von Nutzungsprofilen erzeugt werden können. Schließlich diskutiert Abschnitt 3.5 Beschreibungstechniken für die Spezifikation von Testfällen.

Kapitel 4, Prinzip der Testfallgenerierung mit AutoFocus, beschreibt zunächst das grobe Vorgehen der Testfallerzeugung und Testdurchführung in Abschnitt 4.1. Dabei wird von der in Abschnitt 2.4 getroffenen Unterscheidung verschiedener Szenarien abstrahiert. Abschnitt 4.2 beschreibt das Modellierungswerkzeug AutoFocus und seine Beschreibungstechniken anhand des für die Fallstudie relevanten AutoFocus-Modells des wireless identity module. Die Übersetzung von AutoFocus-Modellen in die Constraint-Logik-Programmierung wird in Abschnitt 4.3 erörtert. Das beinhaltet eine Einführung in Constraint-Logikprogrammierung, die Übersetzung von Zustandsmaschinen und die Übersetzung von Funktionsdefinitionen, den zwei wesentlichen Teilen der AutoFocus-Beschreibungsmittel für die Verhaltensspezifikation. Testfallgenerierung mit symbolischer Ausführung wird in Abschnitt 4.4 diskutiert. In Abschnitt 4.5 wird erläutert, wie die abstrakten Testfälle als Resultat einer symbolischen Ausführung mit klassischen Heuristiken wie der Grenzwertanalyse in ausführbare Sequenzen instantiiert werden. Ein allgemeines Verfahren zur Generierung von Integrationstests aus Unittests wird in Abschnitt 4.6 präsentiert und in Abschnitt 4.7 am Beispiel der Erzeugung von strukturellen (Modified Condition/Decision Coverage-) Testsuiten auf Systemebene instantiiert.

Kapitel 5, Effizienzsteigerung und Anwendung, beschreibt verschiedene Techniken zur Eindämmung des Problems der Zustandsexplosion. Da die Testfallgenerierung im wesentlichen als Suchproblem aufgefaßt wird, werden in Abschnitt 5.1 verschiedene Suchverfahren diskutiert. Diese beinhalten insbesondere verschiedene Formen der Tiefensuche, die randomisiert oder gerichtet u.U. mit Wettbewerbsparallelität durchgeführt wird. In Abschnitt 5.2 wird die Verwendung von Constraints zur Zustandsspeicherung vorgestellt. Die technische Motivation ist, daß die Suche eingeschränkt werden kann, wenn der mehrfache Besuch von Zuständen verboten wird. Im wesentlichen wird eine spezialisierte Instanz des Model Checking diskutiert, das für die Testfallgenerierung relevant ist. Auswirkungen auf die Qualität der erzeugten Testfälle werden diskutiert. Abschnitt 5.3 faßt die verschiedenen Einsatzgebiete von Constraints nocheinmal zusammen. Alle Techniken, Suchstrategien, Wettbewerbsparallelität und Zustandsspeicherung, werden anhand experimenteller Daten in Abschnitt 5.4 analysiert.

Kapitel 6 schließlich, Zusammenfassung und Ausblick, faßt die erarbeiteten Ergebnisse in Abschnitt 6.1 zusammen. Im folgenden Abschnitt 6.2 werden abschließend zukünftige Arbeiten mit einem Fokus auf nichtdeterministische

und gemischt diskret-kontinuierliche Systeme diskutiert.

Anhang A, Programmiersprachen: Abstraktionen und Domänenabhängigkeit, enthält historische Betrachtungen über die Entwicklung von Programmiersprachen, als deren Weiterführung die Verwendung modellbasierter Formalismen vom Verfasser angesehen wird.

Anhang B beinhaltet die Beweise von in der Arbeit aufgestellten Sätzen.

Referenzen

Verwandte Arbeiten werden in den einzelnen Kapiteln diskutiert. Teile dieser Arbeit wurden in Pretschner (2003, 2001); Pretschner und Philipps (2003, 2002, 2001); Pretschner u. a. (2003a,b, 2001a,b, 2000); Philipps u. a. (2003); Hahn u. a. (2003a,b); Braun u. a. (2003); Bender u. a. (2002); Schätz u. a. (2002a,b); Blotz u. a. (2002); Pretschner und Lötzbeyer (2001); Pretschner und Schätz (2001); Lötzbeyer und Pretschner (2000a,b); Wimmel u. a. (2000) veröffentlicht, sind zur Veröffentlichung akzeptiert oder befinden sich in der Begutachtungsphase.

2. Modellbasiertes Testen

Dieses Kapitel definiert den terminologischen und methodischen Rahmen der Arbeit. Abschnitt 2.1 präsentiert zunächst die Fallstudie, anhand derer alle Konzepte illustriert und erprobt werden. In Abschnitt 2.2 werden zentrale Begriffe des Testens definiert. Da ausführbare Verhaltensmodelle als Grundlage der Testfallgenerierung dienen werden, diskutiert Abschnitt 2.3 wesentliche Aspekte des Modellbegriffs und stellt den Bezug zwischen modellbasiertem Testen und modellbasierter Entwicklung her. Abschnitt 2.4 präsentiert drei Szenarien zur Abfolge der Entwicklung von Modell und tatsächlichem System. Eine Skizzierung verwandter Ansätze im Bereich der Qualitätssicherung und der zentralen Beobachtungen dieses Kapitels erfolgt in den Abschnitten 2.5 und 2.6.

2.1. Fallstudie: Wireless Identity Module

Die Ergebnisse dieser Arbeit werden anhand einer Chipkartenanwendung illustriert, die in einer gemeinsamen Studie (Philipps u. a., 2003) mit Validas AG und Giesecke&Devrient GmbH untersucht wurde. Modell der Chipkarte und Infrastruktur zur Testdurchführung wurden dem Verfasser zur Verfügung gestellt.

Chipkarten existieren als Speicherkarten oder als Prozessorkarten (Rankl und Effing, 1999). Erstere finden Verwendung z.B. als Krankenkassenkarten und werden hier nicht weiter berücksichtigt. Prozessorkarten (im folgenden: Chipkarten) sind vollständige Computer, meist auf Basis einer CISC-Architektur. Sie bestehen aus CPU (Architektur üblicherweise basierend auf Motorola 6805 oder Intel 8051), RAM (256-2048 Bytes), ROM (8-64 KB), EEPROM/Flash (2-32 KB) sowie ggf. einem numerischen Coprozessor z.B. für kryptographische Berechnungen. Takt und Spannung kommen durch Kontakt mit dem Terminal oder durch Induktion im Fall kontaktloser Karten von außen. Das ROM beinhaltet das Betriebssystem sowie eine oder mehrere Anwendungen; das EEPROM ist üblicherweise als Dateisystem organisiert.

Chipkarten sind im wesentlichen Kommandointerpreter, die bei Anlegen eines Kommandos eine Antwort berechnen und diese zur Verfügung stellen. Kommandos und Antworten werden in Form von Bytefolgen, sogenannten APDUs (Application Protocol Data Units) angegeben. Eingaben werden durch ein Terminal geliefert, das auch die Antworten entgegennimmt. Kommandos von Chipkarten umfassen u.a. Kommandos zur Dateiverwaltung, zur Authentisierung und für kryptographische Operationen. Chipkarten finden überwiegend als Sicherheitstokens zur Zugangskontrolle für Gebäude oder zur Authentisierung beispielsweise in mobilen Telephonen Verwendung.

Im Rahmen dieser Arbeit findet das WAP Identity Module (WIM (WAP Fo-

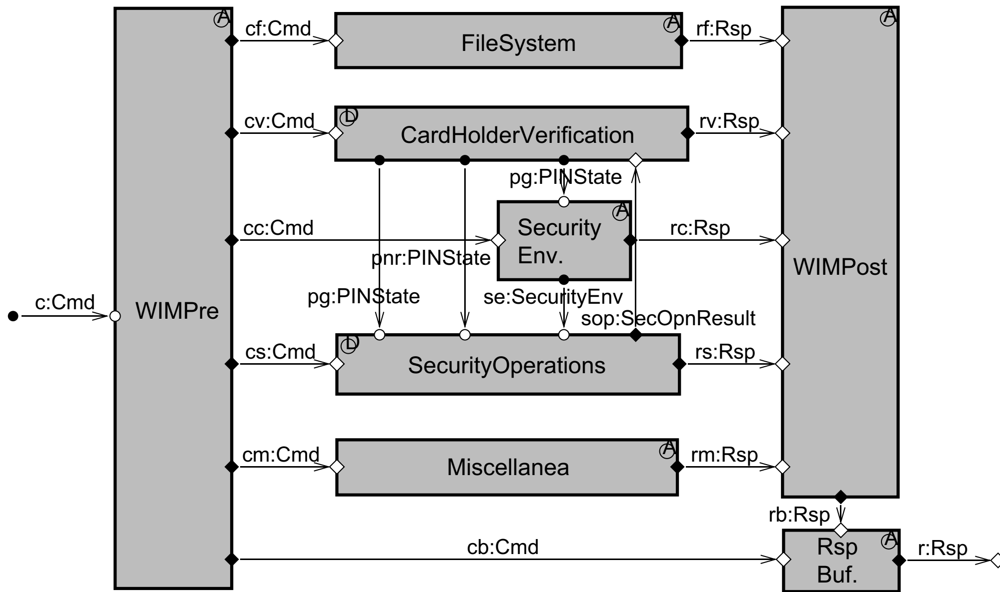


Abbildung 2.1.: Systemstrukturdiagramm der WIM

rum, 2001)) zur Illustration Verwendung. Die WIM dient der Authentisierung eines Benutzers mit PINs und PUKs (Personal Identification Numbers, Personal Unblocking Keys), implementiert kryptographische Funktionen (Ver- und Entschlüsselung, Berechnung digitaler Signaturen) und den sicherheitsbezogenen Teil des RSA-Handshakes zwischen mobilem Endgerät und einem Server. Ohne hier bereits umfassend auf die Beschreibungstechniken des Werkzeugs AutoFocus einzugehen, zeigt Abb. 2.1 die funktionale Dekomposition des Systems. Vierecke sind Komponenten, Pfeile repräsentieren Kanäle und damit den Datenfluß.

Kommandos werden über die linke Seite in das Modell geleitet. Die Komponente `WimPre` verteilt das Kommando auf den zugehörigen funktionalen Block. Die folgenden funktionalen Blöcke wurden identifiziert:

- Die Verwaltung des Dateisystems zum Lesen und Schreiben von Dateien übernimmt die Komponente `FileSystem`. Von dieser Komponente bearbeitete Kommandos sind `SelectFile`, `ReadBinary` und `UpdateBinary`. Das Dateisystem ist im Modell nur rudimentär implementiert, weil es zum Test der Sicherheitsfunktionalität der WIM nicht relevant ist. Unterschieden wird zwischen lesbaren und nicht öffentlich lesbaren Dateien, die z.B. Schlüssel enthalten.

- Der Authentisierung des Benutzers dient die Komponente `CardHolderVerification`. Es gibt zwei verschiedene PIN-Typen (personal identification number), die PIN-G (general) für transportbezogene Zwecke und die PIN-NR (non-repudiation) für digitale Signaturen. Verwaltet wird der Status der PINs (verifiziert, nicht verifiziert, abgeschaltet, blockiert) sowie die Anzahl noch verbleibender Versuche zur Verifizierung. Mithilfe

sog. PUKs (personal unblocking key) können blockierte PINs wieder freigeschaltet werden.

- Sicherheitsumgebungen werden in der Komponente `SecurityEnvironment` modelliert. Die Komponente codiert im wesentlichen drei Modi: keine Sicherheitsumgebung ist ausgewählt, die Umgebung für wireless transport level services (WTLS, verantwortlich für Kommunikation zwischen mobilem Endgerät und einem Server) ist ausgewählt, oder die Umgebung für die WIM (digitale Signaturen) ist ausgewählt. Behandelt werden für beide Umgebungen kryptographische Kommandos wie das Setzen der privaten und öffentlichen Schlüssel, die Berechnung des Master Secrets für den RSA-Handshake (im wesentlichen eine Implementierung des Needham-Schröder-Protokolls) zwischen mobilem Endgerät und einem Server, sowie das Setzen von Signaturlängen. Die entsprechenden Datenstrukturen sind drei sogenannte Templates: ein allgemeines kryptographisches Template, ein Template für kryptographische Prüfsummen und eins für digitale Signaturen.

- Kryptographische Operationen werden von der Komponente `SecurityOperations` durchgeführt, die in Abb. 2.2 verfeinert wird – jeweils eine Komponente zur Entschlüsselung, Verschlüsselung, Berechnung einer kryptographischen Prüfsumme, Verifikation und Berechnung einer digitalen Signatur.[1] Im Modell werden die kryptographischen Operationen nicht wirklich durchgeführt; die Ausgaben sind stattdessen alle notwendigen Informationen zur tatsächlichen Ausführung der kryptographischen Kommandos. Diese werden während der Testdurchführung auf der Testtreiberebene durchgeführt (siehe Beispiel 8 auf S. 51).

- Für andere Kommandos wie Zufallszahlengenerierung ist die Komponente `Miscellanea` zuständig. Zufallszahlen werden zur Verschlüsselung verwendet; Zweck ist es, symmetrische Schlüssel auszuhandeln und Nachrichten auch bei identischen Daten zu verfremden.

Die Antworten werden in der Komponente `WimPost` zusammengefaßt und, sofern erforderlich, in der Komponente `ResponseBuffer` gespeichert. Kommandos wie ein `CardReset` betreffen alle Komponenten. Von der Selektion verschiedener GSM-Kanäle wird abstrahiert.

Testfallgenerierung und -durchführung Aus diesem Modell wurden Testfälle erzeugt, die nicht nur zur Validierung des Modells, sondern auch zur Verifikation einer tatsächlichen Karte verwendet wurden. Diese Testfälle sind I/O-Traces des Modells und codieren das Sollverhalten der tatsächlichen Karte. Eingaben und Ausgaben (Kommandos und Antworten) der tatsächliche Karte sind Bytestrings. Da das Modell eine Abstraktion darstellt und Kommandos und Antworten abstrakte Strings darstellen, müssen diese bei der Testdurchführung in für

[1] Die Komponenten `SEDist` und `PINDist` verteilen die notwendigen Parameter der Sicherheitsumgebungen und der PINs auf die funktionalen Blöcke.

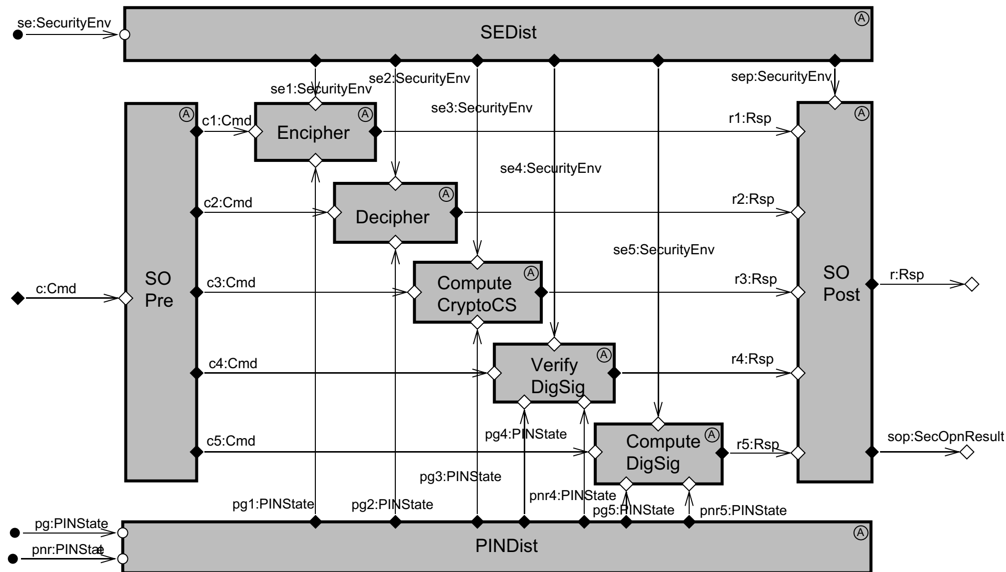


Abbildung 2.2.: Systemstrukturdiagramm für kryptographische Funktionen

die Karte verständliche Bytestrings umgewandelt werden. Kommandos werden konkretisiert:

- Das abstrakte Kommando `AskRandom(n)`, das n zufällige Bytes von der Karte anfordert, kann direkt in den entsprechenden Bytestring umgewandelt werden.
- Für das abstrakte Kommando `Verify(PinGRef,PinG)` zur Verifikation einer der im System auftretenden PINs werden zunächst die symbolischen Namen `PinG` und `PinGRef` durch tatsächliche PINs, d.h. vierstellige Zahlenfolgen, ersetzt, was anhand einer Konfigurationsdatei geschieht. Dann kann das so konkretisierte Kommando in eine von der Karte verständliche Bytefolge übersetzt werden.

Ausgaben des Modells können nicht konkretisiert werden, um mit den tatsächlichen Kartenantworten verglichen zu werden, weil diese Konkretisierung u.U. die falsche Kartenantwort „raten“ würde. Stattdessen müssen die Kartenantworten abstrahiert werden, um mit den Modellantworten verglichen werden zu können:

- Der abstrakte Wert `ResRandom` beschreibt einen Bytestring, der aus n zufälligen Bytes besteht. Er wird vom Modell als Antwort auf das Kommando `AskRandom(n)` ausgegeben. Da der Zufallszahlengenerator nicht getestet werden soll, wird nur die Länge der Zufallszahl überprüft, d.h. die n' von der Karte gelieferten zufälligen Bytes werden in ihre Länge n' abstrahiert, und dieser Wert wird mit der Zahl n verglichen.
- Kryptographische Operationen wurden im Modell nicht modelliert, um den Zustandsraum möglichst klein zu halten. Die abstrakte Antwort des Modells auf ein Kommando, das eine Verschlüsselung anstößt, ist deshalb

ein abstrakter Term, der alle für die Verschlüsselung notwendigen Parameter enthält. Diese sind insofern abstrakt, als beispielsweise für Schlüssel nicht lange Bytestrings im Modell auftauchen, sondern stattdessen abstrakte Werte wie etwa `KeyPub1`. Diese werden bei Testdurchführung wiederum unter Rückgriff auf die Konfigurationsdatei durch tatsächliche Schlüssel ersetzt, und die kryptographische Operation wird auf Ebene der Testdurchführung nachgerechnet. Das Resultat wird in einen Bytestring übersetzt, der mit der Antwort der Karte verglichen wird. In diesem Fall sind Antwort der Karte und Antwort des Modells bijektiv aufeinander abbildbar.

Beispiel 1 (Testdurchführung). *Zur Illustration der Testdurchführung auf Hardwareebene sei ein Testfall angegeben, der die Berechnung einer digitalen Signatur durchführt. Zuerst erfolgt die Verifikation mit einer der PINs. Nach Setzen der entsprechenden Sicherheitsumgebung wird der private Schlüssel gesetzt und dann das abstrakte Dokument* `BytesD` *signiert. Die Antwort wird dann mit dem letzten Kommando ausgelesen. Kommandos an die Karte sind mit einem vorgestellten* `c?` *und erwartete Kartenantworten mit einem vorgestellten* `r!` *gekennzeichnet.*

```
1:  c?Verify(pinNRRef, PinNR);r!H9000
2:  c?MSERestore(SE2Ref);r!H9000
3:  c?MSEDSTSetPrivateKey(EF_4452, Key1Ref);r!H9000
4:  c?PSOComputeDigSig(BytesD);r!H61XX
5:  c?GetResponse;r!ResComputeDigSig(EF_4452, Key1Ref, BytesD)
```

Bei der Testdurchführung werden die Kommandos in Bytefolgen konkretisiert und die Antworten der Karte mit den Antworten des Modells verglichen.

```
Resetting card...

ATR: 3B BF 11 00 C0 14 1F C3 80 73 12 41 11 59 49 37 32 39 53 31 30 30
     20 79

--- Test Prologue ---
Request: ManageChannelOpen; Expected response: H9000
Sending:    00 70 00 00 01; Receiving:  01 90 00; 1 result bytes: 01
Success.
Request: SelectApplication; Expected response: H9000
Sending:    01 A4 04 00 0C A0 00 00 00 63 57 41 50 2D 57 49 4D
Receiving:  90 00 ... Success.

--- Command #1 ---
Request: Verify(pinNRRef,PinNR); Expected response: H9000
Sending:    81 20 00 07 08 34 33 32 31 FF FF FF FF
Receiving:  90 00 ... Success.

--- Command #2 ---
Request: MSERestore(SE2Ref); Expected response: H9000
Sending:    81 22 F3 02 00; Receiving:  90 00 ... Success.
```

```
--- Command #3 ---
Request: MSEDSTSetPrivateKey(EF_4452,Key1Ref); Expected response: H9000
Sending:    81 22 41 B6 07 81 02 44 52 84 01 01
Receiving:  90 00 ... Success.

--- Command #4 ---
Request: PSOComputeDigSig(BytesD); Expected response: H61XX
Sending:    81 2A 9E 9A 28 64 64 64 64 64 64 64 64 64 64 64
            64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64
            64 64 64 64 64 64 64 64 64 64 64 64 64
Receiving:  61 80 ... Success.


--- Command #5 ---
Request: GetResponse
Expected response: ResComputeDigSig(EF_4452,Key1Ref,BytesD)
Sending:    01 C0 00 00 80
Receiving:  49 35 54 BF 2E C7 7E 52 D1 C9 90 AF 2C F2 F1 28
            D0 5C FE E1 00 A7 86 C4 AD 0C DC 15 53 DB C1 A6
            43 26 EB 83 8F A9 43 E3 5C 55 D1 D5 5E 55 61 F4
            3D 98 39 71 B7 42 14 BB 47 B3 77 A8 1B 34 CE ED
            99 8B A5 D0 B0 4C C6 FF 03 3F 47 5F F4 F0 C4 7B
            17 BB 6E 5F 07 D5 0B EE 4B BC 1D 4C 2B B7 D9 9B
            68 74 95 21 92 25 5B D0 67 2D 1B 4F FB 6B 9F 7D
            4F 57 A2 10 D4 31 67 81 46 33 45 65 FB 13 79 F8
            90 00
Signature matches ... Success.

--- Test Epilogue ---
Request: ManageChannelClose; Expected response: H9000
Sending:    00 70 80 01 00; Receiving:  90 00 ... Success.
```

2.2. Testen

Dieser Abschnitt definiert die wesentlichen Konzepte, Aktivitäten und Zielsetzungen des Testens. Abschnitt 2.2.1 gibt zunächst einen groben Überblick über die relevanten Konzepte, die in Abschnitt 2.2.2 in die verschiedenen Aktivitäten des Testprozesses eingebettet werden. In Abschnitt 2.2.3 wird das Vorgehen beim modellbasierten Testen geschildert. Da eine sinnvolle Definition von Testfällen u.a. davon abhängig ist, ob das zugrundeliegende System oder Modell deterministisch ist, wird auf die Problematik des Nichtdeterminismus in Abschnitt 2.2.4 eingegangen. Es folgt die Präzisierung der Terminologie in Abschnitt 2.2.5.

2.2.1. Überblick

Bevor im Abschnitt 2.2.5 eine Präzisierung ausgewählter Teile der Terminologie erfolgt, soll hier zunächst ein grober Überblick über die relevanten Termini im

modellbasierten Testen und ihren Zusammenhang erfolgen.

Die Benennung „System“ bezeichnet in dieser Arbeit sowohl Modelle als auch Implementierungen: Mit AutoFocus spezifizierte Modelle und auch Implementierungen solcher Modelle sind Systeme. In Anlehnung an Zave und Jackson (1997) werden im folgenden Implementierungen als „Maschinen“ bezeichnet.

Zielsetzung Zweck des Testens ist es, Vertrauen in die Übereinstimmung des Istverhaltens einer Maschine und dem Sollverhalten der Spezifikation zu erhöhen oder Fehler zu finden. Diese abstrakte Formulierung liefert dem Tester im Normalfall keine konkrete Hilfestellung, welche Eingaben zum Test sinnvollerweise an ein System angelegt werden sollen. Die Beschreibung einer sinnvollen Menge von Eingaben und/oder durch den Testfall hervorzurufenden Reaktionen erfolgt anhand einer *Testfallspezifikation*, d.h. einem Selektionskriterium für Teile des Sollverhaltens. Zusammen mit einem Modell eines tatsächlichen Systems berechnet ein *Testfallgenerator* daraus Ein- und erwünschte Ausgabesequenzen, eine Teilmenge des Sollverhaltens also. Diese Sequenzen sind *Testfälle.* In Anlehnung an Binder (1999) wird der Eingabeteil eines Testfalls ohne die erwarteten Ausgaben als *Testdaten* bezeichnet. Das Modell kann expliziter (Binder, 1999) oder impliziter, mentaler (Beizer, 1983) Natur und der Testfallgenerator ein Programm oder ein Mensch sein.

Das zentrale Problem des Testens ist die Frage, was einen „guten“ Testfall auszeichnet. Wenn Testdurchführungskosten nicht vernachlässigt werden können, ist eine erste Forderung die, daß die Menge von Testfällen möglichst klein sein soll. Wenn mit Testfällen ein Fehler gefunden wurde, dann muß nach der Ursache des Fehlers gesucht werden. Dementsprechend ergibt sich als zweite Forderung, daß Testfälle möglichst kurz sein sollen.

Diese Forderungen sagen nichts darüber aus, welche Eingaben an ein zu testendes System ein Test beinhalten sollte. Myers (1979) definiert Testen als die Tätigkeit der Programmausführung mit dem Zweck, Fehler zu finden. Daraus kann eine implizite Anforderung an die Qualität eines Testfalls abgelesen werden, nämlich die Fähigkeit, Fehler zu finden. Wenn eine Menge von Testfällen, eine *Testsuite*, keine Fehler findet, z.B. weil das zu testende Artefakt korrekt ist, ist diese Definition ungeeignet: Für solche Systeme würden keine „guten“ Testfälle existieren. Weniger angreifbar – jedoch in ihrer Allgemeinheit ebensowenig direkt verwendbar – ist die Forderung, jeder Testfall solle *mit hoher Wahrscheinlichkeit eine potentielle Klasse von Fehlern aufdecken.* Das setzt eine Klassifizierung von Fehlern für eine Systemklasse in einer abgegrenzten Domäne voraus. Beispiele für Klassifizierungen sind im folgenden angegeben.

- Lutz (1993) präsentiert eine Checkliste für den Entwurf von eingebetteten Systemen in der Raumfahrt, die auf in mehreren Systemen identifizierten Fehlern basiert. Inwieweit eine solche Liste für Testaktivitäten automatisierbar eingesetzt werden kann, ist zu untersuchen.

- Beizer (1983, Kap. 3.2) kategorisiert Fehler grob in Funktions-, System- und Datenfehler. Funktionsfehler betreffen Spezifikationsfehler, inkorrekte Funktionen, Testfehler und Validierungskriterien betreffende Fehler.

Systemfehler betreffen externe und interne Schnittstellen, Probleme der Hardware- bzw. Softwarearchitektur, mit dem Betriebssystem oder mit dem Ressourcenmanagement. Datenfehler schließlich betreffen alle Probleme mit der Datenmodellierung wie beispielsweise Typfehler. Diese Klassifizierung ist insofern hilfreich, als sie mögliche Fehlerquellen auflistet. Für die Ableitung konkreter Testfälle für ein System ist sie allerdings nicht direkt geeignet.

- Breitling (2001, S.14-16) klassifiziert Fehler nach Lokalisierung, Verursacher, Zeitpunkt des Auftretens, Dauer, Grund und Schwere – eine Klassifikation, die in ihrer Allgemeinheit dem Tester wiederum keine Hilfestellung leistet.

- Engler u. a. (2001) definieren Annahmen, die hinter bestimmten Anweisungen stehen. Die Dereferenzierung eines Zeigers beinhaltet die Annahme, daß der Zeiger ungleich Null ist. Wenn gezeigt werden kann, daß der Zeiger doch Null ist, gibt es einen Widerspruch zwischen Annahme und Tatsache. Das ist auch der Fall für eine mit einem Lock versehene kritische Region: Wenn der Lock entfernt wird, ist die Annahme, daß er vorher vorhanden war. Weitere Annahmen aus dem Betriebssystembereich werden formuliert, und es wird gezeigt, wie mit dieser Methode Fehler in frei verfügbarem Linux-Code gefunden wurden.

Fehler sind häufig auch ohne explizite Spezifikation einer konkreten Abweichung von Ist- und Sollverhalten zu finden:

- Ein bekanntes Beispiel dafür sind Typchecker: Typfehler sind häufig Indizien für Fehler in der zugrundeliegenden Programmlogik („Well-typed programs can't go wrong“).

- Prolog-Compiler geben Warnungen aus, wenn innerhalb eines Prädikats eine Variable nur einmal referenziert wird. Motivation ist die Erfahrung, daß (a) eine häufige Ursache dieses Umstands Schreibfehler sind und (b) die nicht explizite Verwendung anonymer Variablen auf Probleme in der Programmlogik zurückzuführen sein kann.

- Im Verifikationswerkzeug der Firma Polyspace wird eine große Klasse der in den Spezifikationsdokumenten einer Programmiersprache erörterten Laufzeitfehler untersucht. Mit abstrakter Interpretation (Cousot und Cousot, 1977) können nach Firmenangaben (PolyspaceTechnologies, 1999) modulo einer zu grob gewählten Abstraktion die folgenden Fehler gefunden werden: Dereferenzierung von Nullzeigern, Out-of-Bound-Feldzugriffe, Lesen nichtinitialisierter Variablen, Zugriffskonflikte auf geteilte Variablen, ungültige arithmetische Operationen wie Division durch Null oder Quadratwurzelberechnung einer negativen Zahl, illegale Typkonversionen, Über- oder Unterlauf auf Integers und Floats, unerreichbarer Code.

- Als weiteres Beispiel zeigen Xie und Engler (2002) für Betriebssystemcode eine Korrelation zwischen Programmierfehlern und „redundantem“

Code. Wenn Code idempotente Operationen (Zuweisungen einer Speicherzelle an sich selbst), Zuweisungen, auf die niemals zugegriffen wird, unerreichbaren Code oder redundante Konditionale (die immer zu wahr oder falsch evaluieren) enthält, dann ist es zu 45-100% wahrscheinlicher, daß die den Code enthaltende Datei Fehler enthält, als wenn die Datei zufällig ausgewählt wurde. Ein anderes Beispiel ist das in Abschnitt 4.7 diskutierte Überdeckungskriterium MC/DC. Darin führen redundante oder direkt voneinander abhängige Literale dazu, daß das Coveragekriterium nicht erfüllt werden kann (Beispiel 21 auf S. 134). Das kann als konzeptionelle Schwäche des Kriteriums aufgefaßt werden. Bisweilen sind solche Abhängigkeiten aber auch nicht beabsichtigt, dem Benutzer nicht bewußt und stellen eine mögliche Fehlerquelle dar.

Die Definition und Operationalisierung einer hohen Wahrscheinlichkeit, Fehler zu finden, – wie in der alternativen Definition auf S. 21 vorgeschlagen – ist natürlich ebenfalls ein schwieriges Unterfangen. Im Rahmen der Testaktivitäten (Abschnitt 2.2.2) ist das Testziel klar zu definieren, auch wenn explizite Aussagen bzgl. Fehlerklassen und Wahrscheinlichkeiten dem heutigen Stand der Kunst gemäß i.a. nicht durchgeführt werden können.

Myers Definition (S. 21) des übergreifenden Zwecks des Testens, Fehler zu finden, ist psychologisch und heuristisch motiviert:

- Zum einen zeigen Studien, daß der Wunsch, Übereinstimmung zu zeigen, im Vergleich mit einem expliziten Streben nach Fehlerentdeckung weniger Fehler aufdeckt. Deshalb sollten Tester und Programmierer unterschiedliche Personen sein, denn wenn beide Aktivitäten von derselben Person durchgeführt werden, herrscht der Wunsch nach Demonstration der Übereinstimmung vor. Diese Erkenntnis wird erstaunlicherweise von den Proponenten sog. agiler, nach eigener Aussage testzentrierter Entwicklungsprozesse wie dem Extreme Programming (XP, Beck (1999)) oder der Agilen Modellierung (Ambler, 2002) ignoriert. Hier sollen, abgemildert durch implizite, unsystematische Reviews ohne explizite Referenz im Rahmen des Pair Programming, ausschließlich Programmierer ihren (eigenen) Code testen. Beck (2003) formuliert im Kontext des Test-Driven Development (TDD) als Motivation, daß in testzentrierten Entwicklungen der Abstimmungsaufwand zwischen Testern und Programmierern zu hoch sei: Es werden ununterbrochen Tests durchgeführt, und der Programmierer kann nicht ständig auf die Tests warten. Das ist nachvollziehbar, schließt aber zusätzliche Tests durch Personen, die nicht den zu testenden Code geschrieben haben, nicht aus. In XP und TDD spielen darüberhinaus Regressionstests eine große Rolle, und für die ist es offenbar irrelevant, wer sie durchführt.

- Zum anderen gibt es empirische Evidenz, daß bestimmte Fehler an bestimmten Stellen des Programms gehäuft auftauchen und, da Programmierer den „Normalfall“ häufig korrekt implementieren, demzufolge diese Stellen mit erhöhter Fehlerwahrscheinlichkeit verstärkt getestet werden sollten (Grenzwerttesten; vgl. die Untersuchungen zum randomisierten

Testen von Duran und Ntafos (1984), Ntafos (1998) und Hamlet und Taylor (1990)). Die verstärkte Betrachtung bestimmter Stellen im Programm ist deshalb notwendig, weil ein erschöpfender Test von Programmen im Normalfall nicht durchführbar ist.

Softwarefehler können definiert werden als *software-related discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition* (ANSI/IEEE, 1983). Fehler stellen Abweichungen vom Sollverhalten dar (Breitling, 2001) und sind immer in bezug auf ein Referenzdokument (Spezifikation, Lastenheft, Pflichtenheft) definiert. Testen hat demnach das Ziel, Abweichungen eines zu testenden Systems von seiner Spezifikation aufzudecken. Im Rahmen dieser Arbeit wird deshalb kein prinzipieller Unterschied zwischen Testen als (a) Tätigkeit zur Fehlererkennung und (b) Tätigkeit zum Nachweis oder Widerlegung der Übereinstimmung von Implementierung und möglicherweise partieller Spezifikation gesehen. Tätigkeit (b) subsumiert Tätigkeit (a).

Die praktische Erfahrung zeigt, daß Fehler häufig dort auftreten, wo Unklarheiten über die erwünschte Funktionalität vorhanden sind. Fehler scheinen selten in in Spezifikationsdokumenten klar beschriebenen Szenarien aufzutreten: Wenn klare Spezifikationen existieren, sind sie den Entwicklern bekannt, und natürlich konzentrieren diese sich auf explizite Informationen. Fehler treten häufiger in nicht oder nicht präzise spezifizierten Situationen auf. Für die Definition des Zwecks des Testens ist das aber irrelevant: Sowohl spezifizierte, als auch nicht spezifizierte Programmabläufe müssen auf Übereinstimmung getestet werden.

Failure, Error, Fault In der Literatur wird unterschieden zwischen Systemversagen, Fehlerzustand und Fehlerursache (Breitling, 2001). Ein Systemversagen (Failure) liegt vor, wenn das Verhalten eines Systems nicht seinem Sollverhalten entspricht, kann also nur zur Laufzeit auftauchen. Ein Fehlerzustand (Error) wird verstanden als Abweichung des Zustands eines Systems von seinem erwünschten Zustand. Ein Fehlerzustand kann, muß aber nicht zu einem Systemversagen führen. Die Fehlerursache (fault) ist der eigentliche Grund für das fehlerhafte Verhalten des Systems.

Der Unterschied zwischen Fehlerzustand und Fehlerursache wird deutlich, wenn man die Maßnahmen der Behebung betrachtet. Ein Fehlerzustand kann zur Laufzeit korrigiert werden, etwa durch Redundanzmechanismen. Eine Fehlerursache ist i.a. viel schwieriger zu beseitigen, weil dazu beispielsweise das Design eines Systems oder der Programmcode geändert werden muß.

Qualitätssicherungsmaßnahmen wie Reviews (Abschnitt 2.5) können Fehlerursachen identifizieren, ohne daß ein Systemversagen festgestellt wurde. (Black-Box)-Tests der Maschine können nur ein Systemversagen identifizieren. In dieser Arbeit wird getestet, um Systemversagen festzustellen. Der Vorgang der Fehlerlokalisation und -behebung ist nicht Teil dieser Arbeit. Es findet eine Konzentration auf zur Entwicklungszeit behebbare Fehler statt, also nicht auf solche, die zur Laufzeit durch Redundanzmechanismen behoben werden müssen.

Verwendung von Testfällen Da das Modell eine Abstraktion der Maschine darstellt und im Rahmen dieser Arbeit Modelle als Grundlage der Definition von Testfällen dienen, müssen die Testfälle konkretisiert werden, um direkt auf die Maschine angewendet werden zu können. Ein Testtreiber übernimmt dann die Aufgabe, den Eingabeteil dieser Testfälle in eine tatsächliches Maschine einzuspielen. Ein Monitor vergleicht die tatsächlichen Ausgaben der Maschine (das Ist-Verhalten bzw. Abstraktionen davon) mit dem Sollverhalten, das durch den Ausgabeteil eines Testfalls codiert wird. Das Resultat des Vergleichs wird in Form eines Verdikts („stimmen überein“ oder „stimmen nicht überein“) ausgegeben.

Im wesentlichen wird also ein Teil des Verhaltens eines Modells mit dem entsprechenden Verhalten einer Maschine verglichen. Wenn sich Unstimmigkeiten zwischen Soll- und Istverhalten ergeben, kann das am Modell liegen, an der Maschine, oder an der Testdurchführungskomponente, die u.a. den Abstraktionsunterschied überbrückt.

Testobjekt Zu Beginn jeder Testaktivität muß das Testobjekt, also das zu testende (Teil)system und seine Umgebung festgelegt werden. Testobjekte können Maschinen oder Modelle davon sein. Maschinen können in der Entwicklungsumgebung auf Hostrechnern oder auf ihrer Einsatzplattform getestet werden. So sind verschiedene Hard- und Software-Plattformen mit entsprechenden (Cross)-Compilern festzulegen, und die Maschinenumgebung kann auf verschiedenen Stufen simuliert (Modellebene, Codeebene auf Host, Instruction-Level-Simulatoren) sein oder die tatsächliche Umwelt. Nach Prüfung der Testbarkeit (Vollständigkeit von Schnittstellenbeschreibungen, Präzisionsgrad von das Sollverhalten beschreibenden Dokumenten) muß außerdem die zeitliche Abfolge verschiedener Tests für verschiedene Testobjekte festgelegt werden. Der Test des Zeitverhaltens einer Maschine beispielsweise erscheint sinnvoller nach dem Test seiner logischen Funktionalität als davor, und das Zeitverhalten einzelner Komponenten kann aussagekräftig nur in einem entsprechenden (bereits getesteten) Kontext getestet werden. Ein Testendekriterium ist ebenfalls zu festzulegen.

Zusammenfassend ist festzuhalten:

- Testen dient dem Vergleich von Soll- und Istverhalten.
- Wenn Testfälle (Definition in Abschnitt 2.2.5) aus abstrakten Modellen generiert werden, müssen zum Test der entsprechenden Implementierung die Abstraktionsniveaus überbrückt werden.
- Vor dem Test ist genau zu spezifizieren, welche Aspekte eines Systems getestet werden sollen.

2.2.2. Testaktivitäten

In Anlehnung an Wegener (2001) lassen sich die folgenden zentralen Testaktivitäten identifizieren: Testplanung, Testorganisation, Testdokumentation, Testfallgenerierung, Testausführung, Monitoring und Testauswertung. Die Aktivität

der Fehlerbehebung ist nicht Teil des Testens. Die Beschreibung der Aktivitäten ist kursorisch; eine Präzisierung findet sich in der zitierten Dissertation. Hartmann (2001) schlägt für viele Testaktivitäten eine Beschreibung auf der Basis der UML vor.

Planung Die Testplanung legt Testobjekt, Kontext, Testziel und Testendekriterium fest.

Organisation Die Organisation legt im wesentlichen eine Verwaltung von Testfällen und Testobjekten fest. Das beinhaltet auch Mechanismen für den Regressionstest, d.h. die Verwendung von Testfällen für verschiedene Entwicklungsstufen, Varianten oder Revisionen eines Systems. Dadurch wird die Forderung nach der Reproduzierbarkeit der Testdurchführung und ihrer Resultate induziert.

Dokumentation Zum Zweck der Nachvollziehbarkeit und Reproduzierbarkeit müssen alle Aktivitäten und Ergebnisse des Tests dokumentiert werden.

Testfallgenerierung Nach Angabe eines Testziels, also der informellen Formulierung einer Testfallspezifikation, müssen aus letzterer auf nachvollziehbare Art und Weise Testfälle abgeleitet werden, die ihrer Spezifikation genügen. Die automatisierte Ableitung funktionaler Tests stellt einen wesentlichen Teil dieser Arbeit dar; sie wird in den Kapiteln 4 und 5 ausführlich behandelt. Die Ermittlung eines Testziels und der Testfallspezifikation stellt den wesentlichen intellektuellen Anspruch an den Tester dar.

Ausführung Nachdem Testfälle und Testobjekte festgelegt sind, muß der Eingabeteil der Testfälle – die Testdaten – an das System angelegt werden. Diese Aktivität wird als Testausführung bezeichnet.

Monitoring Die Reaktionen des zu testenden Systems auf die angelegten Testdaten muß zum Vergleich mit dem Sollverhalten aufgezeichnet werden. Zu beachten ist hier, daß der (Black-Box)-Test beinhaltet, daß bei Abwesenheit von watchdog-artigen Mechanismen allein die Ausgaben eines Systems für den Rückschluß auf seinen inneren Zustand zur Verfügung stehen.

Beispiel 2 (Zustand und Ausgabe). *Wenn eine Chipkarte die Information (die Ausgabe) liefert, eine PIN sei blockiert, kann nicht gezwungenermaßen darauf geschlossen werden, daß die PIN auch tatsächlich blockiert ist, also bestimmte Folgeoperationen nicht mehr möglich sind. Solche Situationen sind geeignet zu berücksichtigen, z.B. durch entsprechende Folgekommandos, sog. Postambeln.*

Das Monitoring kann beispielsweise auch die Messung von Codeabdeckungen beinhalten, was eine Instrumentierung der entsprechenden Codeteile notwendig macht. Diese Instrumentierung verändert ihrerseits das System, z.B. kann der Code größer werden, als seine Targetplattform das zuläßt.

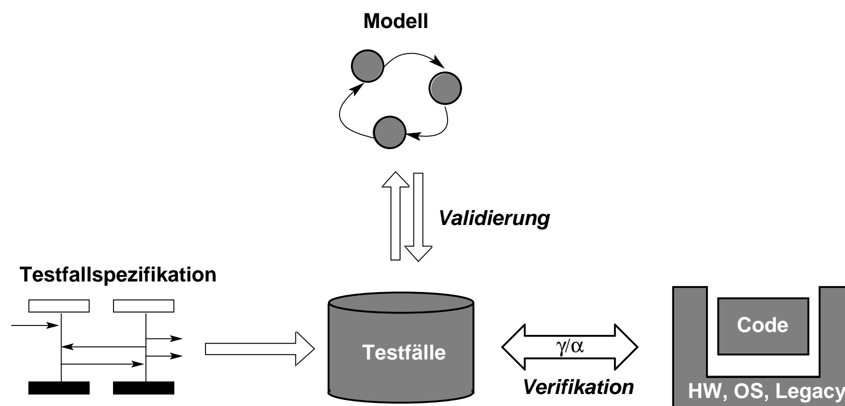


Abbildung 2.3.: Modellbasiertes Testen

Auswertung Die Testauswertung, d.h. der Vergleich von Ist- und Sollverhalten, ist nicht auf die Feststellung von Fehlern beschränkt. Gefundene Fehler müssen an die entsprechenden Entwickler weitergeleitet werden; ihre Behebung muß kontrolliert werden.

Im Idealfall dient sie darüberhinaus der Bewertung des Entwicklungsprozesses und der Identifizierung seiner Schwachstellen. In dieser Arbeit wird darauf nicht näher eingegangen. Testauswertung zur Prozeßverbesserung wird im Rahmen des Cleanroom Reference Models (Prowell u. a., 1999) propagiert: Wenn die Fehlerzahl auf einer bestimmten Stufe des Prozesses oder in einer bestimmten organisatorischen Einheit zu häufig auftreten, soll das Konsequenzen für den Entwicklungsprozeß zeitigen. Losgelöst vom Testen, findet sich diese explizite Forderung ebenfalls in den Inspektionen von Fagan (1976, 2002). Im Capability Maturity Model wird, als Verallgemeinerung, auf der höchsten Qualitätsstufe 5 quantitatives Feedback innerhalb des Prozesses zur stetigen Prozeßverbesserung gefordert.

Die Testauswertung kann auch eine Beurteilung beinhalten, ob die vorher gewählten Testendekriterien erfüllt sind.

Im folgenden bezeichnet der Terminus *Testdurchführung* die Aktivitäten der Testausführung und des Monitoring sowie den Vergleich des Ist- mit dem Sollverhalten.

2.2.3. Vorgehen

Die Grundidee des modellbasierten Testens ist in Abb. 2.3 skizziert. Aus einem Modell und einer Formalisierung der zu testenden Eigenschaft, der Testfallspezifikation, werden Sequenzen von Ein- und Ausgaben generiert. Die Ausgaben der resultierenden Traces werden manuell validiert, was nach mehreren Iterationen zu einem validen Modell führt, einem Modell also, das die Anforderungen des Auftraggebers erfüllt.[2] Testen stellt dabei nur eine mögliche Form der Qualitäts-

[2] Wenn zusätzliche Spezifikationen existieren, können die Ausgaben der Traces gegen diese Spezifikation verifiziert werden. Diese Spezifikationen müssen ihrerseits validiert werden,

sicherung dar. Aus diesem validen Modell können schließlich Testfälle für die zu testende Implementierung generiert werden. Die Idee ist, daß die Ausgaben des Modells mit denen der Implementierung verglichen werden. Das erfordert die Einführung einer Zwischenschicht, die die Abstraktionsniveaus von Modell und Implementierung überbrückt (Pfeil mit γ/α).

Testfallspezifikationen werden auf der Basis von Testzielen definiert und dienen dann der Erzeugung von Testfällen. Grob gesagt, bezeichnet ein Testziel eine informelle oder formale Eigenschaft der Maschine, deren Nachweis geführt oder approximiert werden soll:

- Eigenschaften der Art „Eine vorgegebene Sequenz von Eingaben führt schließlich zu einer bestimmten Ausgabe“ können bewiesen werden. Dazu wird ein entsprechender Trace für das Modell erzeugt und überprüft, ob er auch von der Maschine ausgeführt werden kann.

- Eigenschaften der Art „Für alle Traces[3] gilt in jedem Zustand ein bestimmtes Prädikat φ“ können hingegen nicht direkt getestet werden:

  - Erstens ist Testen eine endliche Tätigkeit. Die Testfälle sind endlich, und die Testfallmenge ist ebenfalls endlich. Wenn eine Invarianzeigenschaft wie die obige nun auf Modellebene gilt, erfüllt eine unendliche Menge von Traces oder eine Menge unendlicher Traces sie. Im Unterschied zu Modellen, auf denen Aussagen über unendliche Abläufe oder unendliche Mengen von Abläufen in endlicher Zeit getroffen werden können, muß zum Test der Maschine eine Einschränkung vorgenommen werden: Aus der Menge aller die Eigenschaft erfüllenden Traces muß eine endliche Menge endlicher Traces ausgewählt werden.
  - Da die Maschine zweitens als black-box angesehen wird, kann der interne Zustand i.a. nicht direkt beobachtet werden. Es müssen also Rückschlüsse aus dem Ein-/Ausgabeverhalten auf den internen Zustand gezogen werden. Eine Möglichkeit der Approximation besteht darin, die entsprechenden Ein-/Ausgaben des Modells mit denen der Maschine zu vergleichen und anhand weiterer Kommandos (Postambeln) zu überprüfen, ob φ tatsächlich in jedem Zustand Gültigkeit hatte. Das setzt voraus, daß die Zustandsstruktur des Modells dem der Maschine ähnlich ist.

Konsequenz ist, daß zu testende Eigenschaften u.U. modifiziert werden müssen, so daß die Menge ihrer Zeugen endlich ist und aus endlichen Traces besteht. Diese Modifikation findet Ausdruck in der Testfallspezifikation. Von Testfallspezifikationen wird darüberhinaus verlangt, daß sie operationalisierbar sind, daß also ein Testfallgenerator in der Lage ist, aus ihnen Testfälle, d.h. I/O-Traces, zu erzeugen.

was das Problem der Validierung verschiebt, aber nicht löst.

[3] Das Wort „Trace“ auf Modellebene bezieht sich nicht gezwungenermaßen ausschließlich auf den Ein-/Ausgabeteil, sondern kann auch den Zustand des Modells beinhalten. Die Lesart wird aus dem Kontext klar.

Beispiel 3 (Testziel, Testfallspezifikation, Testfall). *Im Fall der Chipkarte ist eine relevante Eigenschaft E die, daß eine Signatur nur dann berechnet werden kann, wenn die zugehörige PIN verifiziert ist. Bei intuitivem Verständnis der Zustandsprädikate lautet die zugehörige Formel*

$$E' \equiv AG(\mathit{CardResponse} = \mathit{SignatureComputed} \Rightarrow \mathit{PINStatus} = \mathit{Verified}).[4]$$

E und E′ sind das Testziel. *Eine zugehörige Testfallspezifikation beschreibt dann eine endliche Menge endlicher Kommandofolgen, die schließlich das Kommando zur Berechnung der digitalen Signatur enthalten.*

- *Das können z.B. alle Traces des zugehörigen Modells bis zur Länge* 5 *sein, die vom zugehörigen Testfallgenerator mit einem Kommando der Art „computeAllTracesUpToLength(5)" berechnet werden. Da diese Tracemenge sehr groß ist, wird in der Praxis ein weiteres Selektionskriterium angewendet, z.B. „zufällig ausgewählte 2% aller Traces der Länge* 5*". Die berechneten Traces sind dann die Testfälle.*

- *Da der Status der PIN in der tatsächlichen Karte nun noch nicht überprüft wurde, muß an jeden dieser Traces eine Kommandosequenz angehängt werden, die den Zustand der PIN überprüft. Dazu kann das Kommando „VerifyCheck" verwendet werden, das für eine gegebene PIN deren Status liefert. In der Testfallspezifikation muß also spezifiziert werden, daß an jeden erzeugten Trace der Länge* 5 *das Kommando „VerifyCheck" angehängt wird.*

Offenbar stellt die Einschränkung auf Traces der Länge 5 *eine Approximation der ursprünglichen Eigenschaft E′ dar, die für alle Traces spezifiziert wurde. Darüberhinaus ist nicht gewährleistet, daß das Kommando „VerifyCheck" auch tatsächlich den in der Karte vorhandenen Status der PIN ausgibt. Das kann durch weitere Kommandos überprüft werden, auf die die Karte in Abhängigkeit vom tatsächlichen internen PIN-Zustand unterschiedliche Antworten liefert. Dieses Beispiel wird in Beispiel 13 auf den Seiten 77 ff. weiter ausgeführt.*

Die Unterscheidung ist nicht trennscharf: Testziele können bereits Testfallspezifikationen oder sogar Testfälle sein.

Intuitiv ergibt sich eine ungefähre Analogie, wenn

- Testziele mit Lasten- oder Pflichtenheften,
- Testfallspezifikationen mit Spezifikationsdokumenten und
- Testfälle mit (Maschinen-)Implementierungen

verglichen werden. Der Übergang von Testziel zu Testfallspezifikation kann, wenn ersteres formalisiert vorliegt, als Verhaltensverfeinerung verstanden werden. Dasselbe gilt für den Übergang von Testfallspezifikationen zu Testfällen.

[4]Dabei wird der Einfachheit halber davon abstrahiert, daß in der WIM die Berechnung der Signatur die PIN wieder in einen nicht-verifizierten Zustand versetzt, vgl. Beispiel 13 auf S. 77 ff.

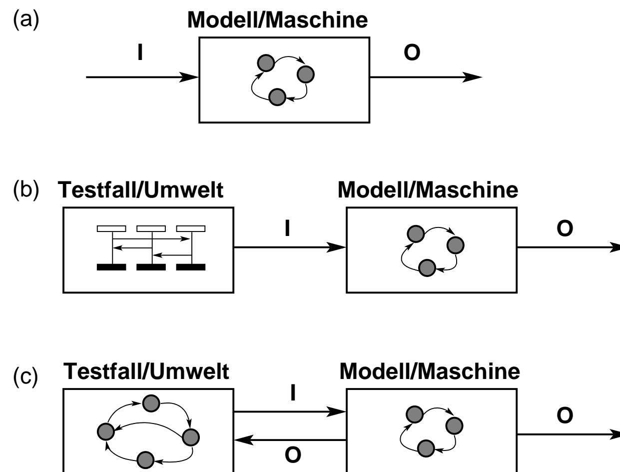


Abbildung 2.4.: Modelle, Umwelt und Testfälle

2.2.4. Nichtdeterminismus

Die Definition des zentralen Begriffs der *Testfälle* beruht auf der Beobachtung, daß sich der Test von deterministischen Systemen von dem nichtdeterministischer fundamental unterscheidet. Bevor im folgenden Abschnitt 2.2.5 die Terminologie präzisiert wird, wird deshalb die Problematik des Nichtdeterminismus diskutiert. Entscheidend ist, daß Testfälle bei Nichtdeterminismus auf Modell- oder Maschinenebene *Mengen* von Traces darstellen, wobei der Ausgabeteil intuitiv das intendierte Maschinenverhalten repräsentiert. Es wird skizziert, wie zum Zweck der Testfallgenerierung häufig auf Nichtdeterminismus auf Modellebene verzichtet werden kann, was für den in Kapitel 4 vorgestellten Testfallgenerator notwendig ist.

Motivation: Testfälle als Mengen von Traces

Modelle von Maschinen und ggf. Teilen ihrer Umwelt bilden Eingaben auf Ausgaben ab (Abb. 2.4 (a)). Bei der Testdurchführung werden durch Testfälle definierte Stimuli an ein System – ein Modell oder eine Maschine – angelegt. Testfälle ersetzen somit Teile der Umwelt des Systems (vgl. Spitzer (2001); Sax u. a. (2002)). Testfallgenerierung bedeutet also, daß ein spezifisches Umweltverhalten erzeugt wird.

Deterministische Systeme Wenn das System deterministisch ist, dann ist ein Testfall *ein Trace des Systems*. Der Eingabeteil des Testfalls ist dabei gleichsam Ausgabeteil der entsprechenden partiellen Umgebung (Abb. 2.4 (b)). Jeder „Schritt“ der Umgebung ist von vornherein festgelegt. Eine Rückkopplung mit dem zu testenden System findet nicht statt. Im wesentlichen stellt die betrachtete Chipkarte einen solchen Fall dar: Kommandos werden nacheinander an die Chipkarte gesendet, unabhängig davon, wie die Chipkarte auf sie reagiert.

Nichtdeterministische Systeme Ist das zu testende System hingegen nichtdeterministisch, kann nicht jeder Schritt der Umgebung von vornherein festgelegt werden. Vielmehr muß der Testfall – die partielle Umgebung – auf das Verhalten des Systems reagieren (Abb. 2.4 (c)). Dann muß der Testfall über eine eigene Ablauflogik verfügen. Ein Testfall repräsentiert in diesem Fall *eine Menge möglicher Traces*.

Als Beispiel sei ein Bussystem betrachtet, an das ein Netzwerk-Master und eine Menge von Geräten angeschlossen sind. Für den Test des Netzwerk-Masters werden aus einem Modell des Bussystems Testfälle abgeleitet. Diese spiegeln eine Einschränkung des Verhaltens der anderen angeschlossenen Geräte wider, also die Umwelt des Netzwerk-Masters. Bei der Testausführung werden alle diese Geräte durch das „Verhalten“ des Testfalls ersetzt. Netzwerk-Master und Bus selbst hingegen seien durch tatsächliche Hardware gegeben. Nun kann es bei der Testausführung passieren, daß Signale nicht rechtzeitig an den Netzwerk-Master gesendet werden können, etwa weil die Kommunikationsinfrastruktur im laufenden Betrieb zwischenzeitlich andere Kommunikationsaufgaben zu erfüllen hat: Netzwerk-Master und Kommunikations-Manager bilden ein nichtdeterministisches System. Konsequenz kann sein, daß der Netzwerk-Master wegen des erfolgten Timeouts Anfragen ggf. erneut an dasjenige Gerät sendet, dessen Antwort nicht rechtzeitig übermittelt worden ist. Das entsprechende wiederholte Signal ist im Testfall nicht vorgesehen. Das vom Testfall spezifizierte Verhalten und das tatsächliche Verhalten des Netzwerk-Masters differieren. Soll diese Situation ausgeschlossen werden, muß der Testfall auch potentiell auf die wiederholte Anfrage reagieren können. Er stellt damit eine Menge möglicher Abläufe des Systems dar.

„Qualitativer“ und „quantitativer“ Nichtdeterminismus Dieses Beispiel stellt einen gewissermaßen „qualitativen“ Nichtdeterminismus dar, der sich in qualitativ fundamental unterschiedlichem Systemverhalten niederschlägt. Es läßt sich auf viele andere Situationen übertragen, in denen Nichtdeterminismus auftritt, etwa wenn während des Betriebs Interaktion mit einem menschlichen Benutzer zugelassen wird.

Häufig tritt auch ein eher „quantitativer“ Nichtdeterminismus auf, der sich in einer Unschärfe in Zeit- und in Wertdimension niederschlägt. Häufig sind solche Schwankungen für das korrekte Verhalten irrelevant. In Anwesenheit kontinuierlicher Signale etwa ist es nicht sinnvoll, den Ausgabeteil eines Testfalls als das exakt intendierte Verhalten zu interpretieren. Vielmehr müssen im Normalfall leichte Schwankungen im Wertebereich akzeptiert werden, die durch „Schläuche“ um die Ausgaben definiert werden können (Gupta u. a., 1997; Hahn u. a., 2003a) oder die durch Verfahren der Signalverarbeitung im Nachhinein „herausgerechnet“ werden (Wiesbrock u. a., 2002). In zeitlicher Hinsicht sind definierte Toleranzen für das Senden oder Empfangen von Signalen häufig akzeptabel. Für die Chipkarte etwa ist das exakte zeitliche Verhalten irrelevant:

Beispiel 4 (Zeitlich bedingter Nichtdeterminismus). *Bei der drahtlosen Kommunikation einer Chipkarte mit einem Terminal kann die Dauer der Übertragung von Daten zwischen Karte und Terminal variieren. Außerdem benötigt*

eine Chipkarte unterschiedliche Zeiten für die Abarbeitung verschiedener Kommandos: Die Selektion einer Datei erfolgt in kürzerer Zeit als ein Reset der Karte oder die Berechnung einer digitalen Signatur. Im Modell wird von diesem Zusammenhang abstrahiert (s. Beispiel 6 auf S. 32).

Methodisch ergibt sich ein wesentlicher Unterschied zwischen den beiden Formen des Nichtdeterminismus. Akzeptable Toleranzen in Wert- und Zeitrichtung sollen in einem das Sollverhalten codierenden Verhaltensmodell nicht auftauchen. Wenn möglich, soll er auf der Ebene der Testdurchführung behandelt werden. Qualitativer Nichtdeterminismus hingegen ist durchaus für den Einsatz in Verhaltensmodellen vorgesehen. Wenn Modelle qualitativ deterministisch sind, sind Testfälle, die aus dem Modell berechnet werden, einzelne Traces. Sie repräsentieren bei Anwesenheit von quantitativem Nichtdeterminismus Mengen von Traces der Maschine. Wenn Modelle qualitativ nichtdeterministisch sind, sind Testfälle Repräsentationen von Mengen von Traces.

Beherrschung von Nichtdeterminismus

Solange die *Umgebung* (oder eine Abstraktion) soweit kontrolliert werden kann, daß sie deterministisch ist, spielt der „qualitative“ Nichtdeterminismus bei Verwendung deterministischer Modelle keine Rolle. Im obigen Beispiel des Netzwerk-Masters kann ggf. sichergestellt werden, daß die diskutierte Verzögerung der Kommunikation nicht auftritt. Dann ist die beschriebene Situation unmöglich.

Bei der Chipkarte gibt es außer dem deterministisch kontrollierbaren Terminal keine Umgebung. Von nichtdeterministischem Verhalten der Chipkarte selbst kann ebenfalls abstrahiert werden.

Beispiel 5 („Qualitativer“ Nichtdeterminismus in der Maschine). *Chipkarten mit Kryptofunktionalität sind mit Zufallsgeneratoren ausgestattet, die als nichtdeterministisch angesehen werden können. Im Modell ist dieser Nichtdeterminismus wegabstrahiert (s. Beispiel 8 auf S. 51).*

Durch die Abstraktion von Zufallszahlen durch einen symbolischen Wert *someRND* kann diese Form des Nichtdeterminismus ausgeschlossen werden – erst, wenn eine Instanz des Testfalls auf die Chipkarte angewendet wird, kann dieser Wert zur Laufzeit des Systems durch eine tatsächliche Zufallszahl von der Chipkarte ersetzt werden.

Nichtdeterminismus in zeitlicher Hinsicht kann häufig durch eine geschickte Partitionierung der Zeitachse behandelt werden. Wenn diese Lösung gewählt wird, ist das Modell durch die erfolgende Abstraktion wiederum deterministisch. Testfälle für Chipkarten sind dann Sequenzen von Ein-Ausgabepaaren ohne Zeit. Wenn ein solcher Testfall auf die Chipkarte angewendet wird, wird vor Absenden eines Kommandos durch den Treiber solange gewartet, bis eine Antwort der Karte auf das vorherige Kommando vorliegt. Das wird üblicherweise durch die Treibersoftware für das Terminal sichergestellt.

Beispiel 6 (Nichtdeterminismus, Fortsetzung Bsp. 4). *Ein* AutoFocus-*Sequenzdiagramm beschreibt genau einen nicht-zeitbehafteten Ablauf und kann als Testfall interpretiert werden. Wenn die entsprechende Sequenz zum Test*

einer tatsächlichen Chipkarte verwendet wird, muß der Treiber – neben der Konkretisierung von Modell-Kommandos in Karten-Bytefolgen – der Tatsache Rechnung tragen, daß verschiedene Kommandos verschiedene Ausführungszeiten besitzen. Ein Beispiel ist der Testfall

$$\begin{array}{ll} < & (CardReset, ATR), \\ & (SelectFile(\{X : X \neq EF_prKDF\}), H9000) \quad >, \end{array}$$

der auf ein CardReset *ein* Answer-To-Reset *erwartet und* H9000 *als Antwort auf die Selektion einer Datei, die nicht* EF_prKDF *ist. Der erste Befehl benötigt deutlich mehr Zeit als der zweite. Im Modell wird davon abstrahiert: Erst während der Testdurchführung werden die unterschiedlichen Zeiten implizit berücksichtigt, wenn vor Absenden eines Kommandos auf die Antwort auf das zuletzt gesendete Kommando gewartet wird.*

Nicht immer kann von Nichtdeterminismus abstrahiert werden, oder es ist nicht immer sinnvoll. Auf das Problem der Testfallgenerierung für nichtdeterministische Systeme wird deshalb in Abschnitt 6.2 zurückgekommen.

Zusammenfassung

Die Tatsache, daß Testfälle Mengen von Sequenzen repräsentieren, ist durch „qualitativen“ und „quantitativen“ Nichtdeterminismus in Wert- und Zeitdimension motiviert.

In dieser Arbeit wird aus Gründen der intellektuellen Beherrschbarkeit zwar eine System- und insbesondere Modellentwicklung auf der Basis ausführbarer Artefakte propagiert: Zum Zweck der eindeutig wiederholbaren Ausführung sollten Modelle von Maschinen dazu deterministisch sein.[5] Nichtdeterminismus in Umweltmodellen hingegen muß zugelassen werden, einfach deshalb, weil die Umgebung i.a. als stochastischer Prozeß aufgefaßt werden kann.

2.2.5. Präzisierung der Terminologie

Bei deterministisch kontrollierbarer oder zum Zweck der Testdurchführung adäquat deterministisch abstrahierbarer Umwelt und deterministischem Modell der Maschine ist die Intention hinter der folgenden Definition eines Testfalls die, daß er *ein Trace des Modells* ist. Dieser Trace repräsentiert ggf. eine *Menge von Traces der Maschine.*

Ist die Umwelt oder das Maschinenmodell nichtdeterministisch, besitzt ein Testfall eine interne Ablauflogik und repräsentiert so nicht nur Mengen von Traces der Maschine, sondern auch Mengen von Traces des Modells. Entscheidend ist, daß ein aus einem Modell abgeleiteter Testfall nach geeigneter Transformation in eine Maschine eingespielt werden kann. Daraus resultieren die folgenden Definitionen von Testfällen und Testsuiten:

[5] Der Test von gemischt diskret-kontinuierlichen Systemen erfordert aus Gründen der Komplexität bisweilen die Verwendung nichtdeterministischer Abstraktionen. Das wird in Abschnitt 6.2 betrachtet.

Definition 1 (Testfall). *Ein* Testfall *ist eine Charakterisierung einer endlichen Menge von Modelltraces endlicher Länge. Die Charakterisierung ist insofern operationalisierbar, als ein* Testtreiber *Eingaben möglicherweise unter Berücksichtigung vorheriger Systemausgaben in eine zu testende Maschine einspielen kann. Vermöge einer Abstraktions- und Konkretisierungsabbildung lassen sich die Ein- und Ausgaben der Modelltraces auf Maschinentraces abbilden.*

Ein Testfall wird als Charakterisierung einer Tracemenge definiert und nicht als eine Tracemenge selbst. Der Grund ist, daß ein Testfall *mögliche* Abläufe repräsentiert. Wenn im Testfall eine Aktion eines zu testenden nichtdeterministischen Systems vorgesehen ist, die niemals auftritt, dann werden die Traces, die die entsprechende Aktion enthalten, auch nicht durchgeführt.

Ausführungsbedingungen für einen Testfall werden als konstant vorausgesetzt. Gegen die entsprechende Erweiterung der Definitionen von Testfällen, Testzielen und Testfallspezifikationen spricht aber nichts; hier wurden sie aus Gründen der Übersichtlichkeit weggelassen. Bei Binder (1999) oder im DO-178B (RTCA und EUROCAE, 1992) sind sie explizit Teil der Definition.[6]

Definition 2 (Testsuite). *Eine* Testsuite *ist eine endliche Menge von Testfällen.*

Testfälle beinhalten Eingaben und intendierte Ausgaben. Wenn ein Modell validiert werden soll und keine Referenzspezifikation zur Verfügung steht, müssen die Ausgaben manuell untersucht werden. Es ist deshalb kontraintuitiv, die Ein-/Ausgabesequenzen auf Modellebene als Testfälle zu bezeichnen. Bei der Testfallgenerierung wird deshalb präziser von Testdaten gesprochen:

Definition 3 (Testdaten). Testdaten *sind die Projektion eines Testfalls auf die Eingaben.*

Definition 4 (Testtreiber). *Ein* Testtreiber *ist ein Programm, das Testdaten nach evtl. erforderlichen Transformationen in eine bestehende Maschine einspielen, die Maschinenausgaben protokollieren und mithilfe eines* Monitors *mit den durch den Testfall gegebenen intendierten Ausgaben vergleichen kann. Das kann eine Abstraktion der Maschinenausgaben erfordern.*

Der Testtreiber gleicht also die Abstraktionsniveaus von Testfall und zu testendem System ab. Aufgabe des Testtreibers ist es u.a., den in der Maschine auftretenden Nichtdeterminismus in Wert- und Zeitrichtung so weit wie möglich zu kapseln. Im Fall der Chipkarte beispielsweise wird nach Absenden eines Kommandos solange gewartet, bis die Ausgabe der Karte erfolgt ist; daraufhin wird das folgende Kommando abgesendet, usw. Der Testtreiber übernimmt darüberhinaus die Aufgabe, abstrakte Modellkommandos in Bytefolgen zu übersetzen und nimmt kryptographische Operationen vor, die nicht adäquat im Modell abgebildet werden können. Er kümmert sich auch um die Behandlung von Zufallszahlen.

[6] Der DO-178B definiert Testfälle (test cases) als „a set of inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement“.

Bisher wurde die Benennung „Konkretisierung von Testfällen“ gewählt. Das ist nicht ganz präzise: Eingaben an die Maschine, also die Testdaten, werden in der Tat *konkretisiert* – z.B. der symbolische Wert *askRandom(n)* durch eine Bytefolge. Die Maschinenausgaben hingegen müssen u.U. *abstrahiert* werden – z.B. eine zufällige Bytefolge durch den symbolischen Wert *someRND*.

Definition 5 (Testziel). *Ein Testziel ist die informelle oder formale Beschreibung einer zu überprüfenden Eigenschaft, aus der – wie im Fall von beispielsweise formalisierten Invarianzeigenschaften – u.U. nicht direkt Testfälle abgeleitet werden können.*

Testziele beschreiben eine Menge von Testfällen.[7] Beispiele sind

- die Übereinstimmung (Conformance) zweier Artefakte bzgl. eines nicht formalisierten Beobachtungsbegriffs, z.B. „die Implementierung der WIM einer Chipkarte soll einem gegebenen Standard genügen“,
- eine formalisierte oder nicht formalisierte Invarianzeigenschaft, z.B. „niemals kann eine PIN gleichzeitig abgeschaltet (disabled) und blockiert sein“. Wegen unendlicher Systemabläufe müssen Invarianzeigenschaften zunächst in Testfallspezifikationen übersetzt werden, um endliche Traces erzeugen zu können.
- ein Coveragekriterium, z.B. „Abdeckung aller Kontrollzustände des Protokolls zur Berechnung einer digitalen Signatur“.
- allgemein eine Anforderung, z.B. „Test sämtlicher APDUs mit sinnvollen Permutationen auf Byteebene“ oder
- ein umgangssprachlich spezifiziertes, unvollständiges oder semantisch nicht fundiertes Szenario, z.B. „das durch ein Sequenzdiagramm (WAP Forum, 2001, S. 36-37) beispielhaft angegebene Protokoll zum abgekürzten Handshake zwischen WIM, mobilem Endgerät und Server“.

Testziele sind struktureller („Bedingungscoverage, def-use-Coverage“), funktionaler („teste das Protokoll zur Berechnung einer digitalen Signatur“) oder stochastischer Natur („teste n zufällig oder bzgl. einer bekannten Eingabedatenverteilung ausgewählte Traces“). Operationalisierbare Varianten werden in Form von Testfallspezifikationen angegeben.

Definition 6 (Testfallspezifikation). *Eine* Testfallspezifikation *ist eine Charakterisierung einer endlichen Menge von Testfällen. Diese Charakterisierung ist insofern operationalisierbar, als ein Testfallgenerator daraus Testfälle erzeugen kann.*

[7] Im Kontext des Conformance Testing definiert die ITU-T Recommendation X.290 OSI Conformance Testing Methodology and Framework—General Framework einen „test purpose“ als „A prose description of a well-defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements as specified in the appropriate OSI specification“.

Der Unterschied zu Testfällen liegt darin, daß Testfälle in einer Form vorliegen müssen, die von einem Treiber in eine Maschine eingespielt werden, während Testfallspezifikationen in einer Form vorliegen müssen, aus denen ein Testfallgenerator Testfälle erzeugen kann. Bei gegebenem Testfallgenerator, der in der Lage ist, Traces zu finden, die zu einem spezifizierten Zustand oder einer spezifizierten Reaktion des Systems führen, sind Beispiele:

- Eine Menge von CTL-Formeln[8] $\bigcup_{\sigma' \in \varepsilon(\sigma)} \{EF\sigma'\}$ für alle in einer Umgebung ε eines Zustands σ liegenden Zustands, der eine Invarianzeigenschaft des Systems verletzt, um so den Test einer negierten Invarianzeigenschaft zu approximieren. Neben den Formeln muß ein Selektionskriterium angegeben werden, das aus der ggf. unendlichen Menge ggf. unendlicher Traces eine endliche Teilmenge endlicher Traces auswählt.

- $\bigcup_{\sigma} \{EF\sigma\}$ für alle Kontrollzustände σ beispielsweise einer Komponente (in Abschnitt 3.3 wird erläutert, wie für verschiedene Coveragemaße solche Spezifikationen automatisch berechnet werden können). Auch hier ist u.U. zusätzlich ein endliches Selektionskriterium notwendig.

- $\pi(T)$ für einen Permutationsoperator π auf einer gegebenen Menge T von Traces auf Byteebene.

- $EF(i = \mathit{set} \rightarrow (EF(o = \mathit{timeout})))$ für eine irgendwann (endlich) zu erfolgende Ausgabe *timeout* auf einem Kanal o, nachdem vorher auf einem Kanal i ein Signal *set* anlag. Ein endliches Selektionskriterium ist ggf. anzugeben.

- Ein Programm, eine Zustandsmaschine, oder ein mit einer präzisen Semantik versehenes Sequenzdiagramm zusammen mit einem Coveragekriterium. Ein Beispiel ist eine Zustandsmaschine, die verschiedene Permutationen des Protokolls zur Berechnung einer digitalen Signatur codiert (Abb. 3.1). Diese Maschine wird mit dem Modell der Karte komponiert, sie dient also als Treiber für das Kartenmodell. Der Treiber kann nichtdeterministisch für das Protokoll relevante Kommandos ausgeben. Mit einem endlichen Selektionskriterium können daraus dann beispielsweise alle Testfälle der Länge n abgeleitet werden.

- Eine Menge explizit angegebener (z.B. zufällig erzeugter) endlicher Traces, die nach Meinung des Entwicklers die Conformance zwischen System und Spezifikation nachweisen. Testfälle sind also detaillierte Testfallspezifikationen.

Hier wird die konkrete Operationalisierung einer Testfallspezifikationen nicht präjudiziert. Testfallspezifikationen können direkt zur Berechnung von Testfällen

[8] Modelle von CTL-Zustandsformeln sind Zustände in einer Kripkestruktur $\mathcal{K}$. Da Testfälle als Traces definiert sind, können sie nicht als Modelle von CTL-Zustandsformeln verwendet werden. Stattdessen wird nur der Initialzustand *init* und die dem existentiellen Pfadquantor folgende Pfadformel betrachtet, deren Modell – ein Pfad – einen Testfall charakterisiert. Für $EF\varphi$ wird also ein Pfad $\pi = (\mathit{init}, \sigma_1, \ldots, \sigma_n)$ für Zustände σ_i mit $\mathcal{K}, \pi \models F\varphi$ gesucht.

verwendet werden. Man kann sie auch als Selektionskriterium für eine vorhandene (große) Menge von Testfällen verwenden. Schließlich werden sie auch als Testendekriterium interpretiert (Zhu u. a., 1997).

Aus einer ökonomischen Perspektive ist eine möglichst kleine Menge möglichst kurzer Testfälle anzustreben. Es wird davon ausgegangen, daß solche Bestrebungen als Teil der Testfallspezifikation codiert sind; das ist das o.g. Selektionskriterium.

Offenbar gibt es für ein Testziel mehrere entsprechende Testfallspezifikationen. Wesentlich ist, daß ein Testfall seine Testfallspezifikation erfüllt, daß also die durch einen Testfall beschriebenen Traces eine Untermenge der durch die Testfallspezifikation definierten darstellt.

Definition 7 (Testfallgenerator). *Ein* Testfallgenerator *ist ein Verfahren oder ein Programm, das aus einer Testfallspezifikation und einem Modell eine Testsuite berechnet.*

Das Vorgehen bei der Testfallgenerierung gestaltet sich wie folgt. Ausgehend von einem Testziel, wird eine Testfallspezifikation abgeleitet (Kap. 3). Diese Testfallspezifikation ist in einem Format angegeben, das der Testfallgenerator zur Erzeugung von Testfällen verwenden kann (ein konkreter Testfallgenerator wird in den folgenden Kapiteln vorgestellt). Diese Testfälle werden dann vom Treiber in das System eingespielt. Der Vergleich von Ist- und Sollverhalten erfolgt durch den Monitor, der Verdikte bildet:

Definition 8 (Monitor, Verdikt). *Ein* Monitor *ist ein Programm, das das Verhalten eines Systems mit dem durch den Testfall spezifizierten intendierten Verhalten vergleicht. Das Resultat dieses Vergleichs heißt* Verdikt *und nimmt im Rahmen dieser Arbeit wegen der Konzentration auf vollständig spezifizierte deterministische Systeme die Werte* pass *und* fail *an.*

Semantische Überlappung Die Definitionen von Testziel, Testfallspezifikation und Testfall sind nicht orthogonal. Da eine Präzisierung des „Grades der Operationalisierbarkeit“ schwierig ist, wird auf disjunkte Definitionen verzichtet:

- Die Unterscheidung von Testziel und Testfallspezifikation basiert auf einem unterschiedlichen Grad von Formalisierung und Operationalisierbarkeit. Testziele sind eine Obermenge von Testfallspezifikationen. Aus Testzielen können i.a. nicht direkt Testfälle erzeugt werden. Sinn von Testfallspezifikationen ist es, Testfälle zu erzeugen.

- Aus Testfallspezifikationen sollen Testfälle abgeleitet werden können. Testfälle hingegen sollen auf Modelle oder Maschinen angewendet werden. Beide charakterisieren eine Menge von Ein-/Ausgabesequenzen. Eine Testfallspezifikation kann direkt als eine Ein-/Ausgabesequenz angegeben werden.

Black-Box- und White-Box-Tests

Zuletzt soll auf die in der Literatur übliche Unterscheidung zwischen White-Box- und Black-Box-Tests eingegangen werden. Black-Box-Tests werden ohne Kenntnis des Programmcodes entworfen. White-Box-Tests werden mit Kenntnis des Programmcodes entwickelt.

- Wenn Testdaten aus dem Modell generiert werden, die seiner manuellen Validierung dienen, dann handelt es sich um einen Whitebox-Test des Modells.

- Wenn aus einem Modell Testfälle für die Maschine abgeleitet werden, dann handelt es sich beim Test der Maschine um einen Black-Box-Test: Der Code der Implementierung wird für die Testfallgenerierung nicht betrachtet. Insbesondere können die Strukturen des Modells und des Codes sich stark unterscheiden.

- Wenn nicht nur Testfälle, sondern auch die Implementierung aus dem Modell generiert wird, dann ist die Unterscheidung zwischen Whitebox- und Blackbox-Test unscharf: Der Programmcode wird zwar nicht für die Ableitung von Testfällen verwendet, aber stattdessen ein Artefakt, aus dem direkt der Code erzeugt wird. Auf die Problematik, daß in dieser Situation die Implementierung gewissermaßen gegen sich selbst getestet wird, wird in Abschnitt 2.4 eingegangen.

Die Unterscheidung ist abgesehen von dem dritten Fall für diese Arbeit irrelevant und wird nicht weiter berücksichtigt.

Zusammenfassung

Die zentralen Beobachtungen dieses Abschnitts sind:

- Testfälle repräsentieren Mengen von Sequenzen. Wenn deterministische Modelle verwendet werden und die Umgebung bei der Testdurchführung deterministisch kontrolliert werden kann, dann entspricht ein Modelltrace (ein Testfall) wegen der erfolgten Abstraktion einer Menge von Maschinentraces. Wenn nichtdeterministische Maschinenmodelle Verwendung finden, dann muß der Testfall – z.B. in Form eines Automaten – mögliche Reaktionen auf nicht vorhersehbare Aktionen der Maschine codieren. Im Rahmen des beschriebenen Verfahrens zur Testfallgenerierung (Kap. 4) findet eine Konzentration auf deterministische Modelle und zu testende Systeme statt, weshalb dort ein Testfall ein Modelltrace ist.

- Es gibt verschiedene Formalisierungs- und Präzisionsgrade von zu testenden Eigenschaften, nämlich Testziele und Testfallspezifikationen. Wenn Testziele Invarianzeigenschaften oder andere alle Abläufe des Systems betreffende Eigenschaften darstellen, muß ein Selektionskriterium angegeben werden, das eine endliche Menge endlicher Testfälle auswählt.

- Die Definition eines Testfalls muß mit Referenz auf einen Testtreiber erfolgen, der die Abstraktionsstufen von Testfall und zu testendem System überbrückt.
- Eine Testfallspezifikation ist nur sinnvoll in Zusammenhang mit einem (automatischen oder manuellen) Verfahren, das aus dieser Testfallspezifikation Testfälle ableiten kann.

2.3. Modelle und Modellbasierung

Zum Test von Systemen werden im Rahmen dieser Arbeit Verhaltensmodelle eingesetzt. Daraus resultieren sofort die folgenden Beobachtungen:

- Die Modelle müssen mit einem Aufwand erstellt sein, den ihr Nutzen rechtfertigt. Wenn modellbasiertes Testen erfolgreich sein soll, dann müssen Sprachen und Methoden zur effizienten und problem- sowie domänenadäquaten Modellerstellung zur Verfügung stehen.
- Die Qualität der Ergebnisse modellbasierten Testens ist durch die Qualität der zugrundeliegenden Artefakte, also Modell und Testfallspezifikation, beschränkt.

Explizite grundsätzliche Diskussionen der Erstellung von Verhaltensmodellen sind rar (Schätz u. a., 2002a; Pretschner und Philipps, 2002; Keutzer u. a., 2000; Sgroi u. a., 2000), weshalb eine kurze Charakterisierung der Modellbasierung und ihre Abgrenzung von codezentrierten Ansätzen notwendig erscheint. Mit Modellbasierung wird dabei die Eigenschaft von Aktivitäten – Systemerstellung oder -test – verstanden, die sich durch den Einsatz von Modellen auszeichnen. Der Unterschied zwischen Modellen und Code ist dabei fließend: Auch Code kann als Modell angesehen werden, und analog können erweiterte Zustandsmaschinen durchaus als Code angesehen werden. Die nicht trennscharfe Unterscheidung liegt im Abstraktionsgrad des betrachteten Artefakts.

Die Überzeugung, daß Modellbasierung qualitativ durchaus eine Weiterentwicklung herkömmlicher codezentrierter Entwicklung darstellt, motiviert diesen Abschnitt, auch wenn die Entwicklung modellbasierter Sprachen nicht Teil dieser Arbeit ist.

Es erfolgt hier zunächst keine Einschränkung auf das modellbasierte Testen, weil die grundlegenden Konzepte der Modellerstellung sowohl auf die Systementwicklung als auch die Verifikation bzw. Validierung von Modellen anwendbar sind. Die Unterschiede werden erst später im Prozeß offenbar: Im ersten Fall soll aus dem Modell manuell oder automatisiert Code abgeleitet werden, im zweiten Testfälle. In der Tat wird die Verwendung ausführbarer Modelle gerade in hochdynamischen Prozessen mit sich ständig wandelnden Anforderungen aufgrund des höheren Abstraktionsgrads und der damit stärkeren intellektuellen Beherrschbarkeit als eine vielversprechende Lösung der aktuellen Probleme der Software-Entwicklung beispielsweise im Bereich der Automobilzulieferer gesehen.

Ein weiteres Einsatzgebiet von Verhaltensmodellen ist das Requirements Engineering und das Rapid oder Evolutionary Prototyping. Insbesondere im Bereich sich stetig ändernder Anforderungen haben sich Modelle hier als wertvolles Hilfsmittel erwiesen (Ward und Mellor, 1985; Hatley und Pirbhai, 1987). Die Idee ist, zum Zweck frühen Feedbacks durch den Auftraggeber simulierbare Artefakte zu schaffen, die nach Klärung etwaiger Mißverständnisse oder Mehrdeutigkeiten als Grundlage für die tatsächliche Systementwicklung dienen.

Die Motivation für die Verwendung von Modellen auch in der Systementwicklung liegt in der mit herkömmlichen codezentrierten Techniken kaum noch handhabbaren Komplexität heutiger Systeme. Es wird sich herausstellen, ob die Verwendung nicht nur von Strukturmodellen (ER- oder Klassendiagramme) und partiellen Verhaltensmodellen (erweiterte endliche Zustandsmaschinen, Statecharts, Sequenz- und Objektdiagramme in der UML), sondern auch von vollständigen ausführbaren Verhaltensmodellen zu erhöhter Effizienz des Entwicklungsprozesses und der Produktqualität führt.[9] Auch wenn dieses Ziel z.Z. nur in Ansätzen erreicht ist, ist eine Alternative zur modellbasierten Entwicklung, der wachsenden Komplexität Herr zu werden, nicht einfach vorstellbar. Grund zu Optimismus ergibt sich beispielsweise aus der industriellen Verwendung kontinuierlicher Modelle, die mit Matlab (Mathworks) oder ASCET-SD (ETAS) erstellt und mit geeigneten Codegeneratoren in Produktionscode übersetzt werden, sowie aus der Entwicklung und Anwendung moderner Sprachen wie Giotto (Henzinger u. a., 2001). Die Modellierung des Verhaltens und der Struktur eines Produkts legt die Frage nahe, ob zum Zweck erhöhter Rigorosität und damit Verfolgbarkeit nicht auch der Prozeß als solcher modellbasiert formalisiert werden könnte (Schätz u. a., 2002a). Diese Frage wird hier nicht weiter berücksichtigt.

Überblick Ziel dieses notwendigerweise abstrakt gehaltenen Teils ist eine Klärung des Begriffs der Modellbasierung. Abschnitt 2.3.1 umreißt den zentralen Begriff des Modells. Berücksichtigt werden Modelle als Abbilder von Problem und Lösung einerseits und System und Umwelt andererseits. Abstraktionen als eine Problemklasse charakterisierende ontologische Entitäten werden als Grundlage modellbasierter Entwicklung identifiziert. In Abschnitt 2.3.2 wird die Problematik verschiedener Granularitäten und Einsatzgebiete von Modellen umrissen. Abschnitt 2.3.3 faßt die Essenz der Modellbasierung zusammen. In Abschnitt 2.3.4 wird die Verzahnung von Testfallgenerierung und inkrementeller Modellentwicklung skizziert. Abschnitt 2.4 diskutiert verschiedene Szenarien der Reihenfolge der Entwicklung von Modell und Code. Dieser Abschnitt konzentriert sich auf Verhaltensmodelle. Datenmodelle werden nur am Rand berücksichtigt.

[9] Empirische Belege für die Behauptung, Klassendiagramme würden die Produktivität erhöhen, sind dem Verfasser nicht bekannt. Die weite Verbreitung von UML-Klassendiagrammen allerdings läßt diesen Schluß nicht von vornherein abwegig erscheinen – so, wie der Erfolg der Objektorientierung nahelegt, sie führe zu Effizienzgewinnen. Vgl. Tichy (1998), der genau das bestreitet.

2.3.1. Modelle

Modelle werden als vereinfachende Abbilder (z.B. Lem (1964, S. 301)) einer tatsächlichen oder einer (noch) nicht tatsächlich existierenden Realität verstanden.[10] Beispiele für die erste Form sind Keplers Gleichungen für die Bewegungen von Planeten, das Bohrsche Atommodell oder Landkarten. Diese Form von Modellen dient der Beschreibung der Realität und im Fall der ersten zwei Beispiele auch der Vorhersage von Ereignissen in dieser Realität. Die zweite Form von Modellen wird durch Architekturmodelle oder Simulationsmodelle von Fahrzeugen exemplifiziert. Sie dienen zum einen dazu, Eigenschaften eines Systems vor seiner tatsächlichen Existenz zu überprüfen – *Maßstabsmodelle* von Flugzeugen für den Windkanal; *Ideen- oder Konzeptmodelle* in der Architektur – oder herauszufinden. Zum anderen liegt ihr Zweck, z.B. im Fall von Blaupausen, darin, als Grundlage und Stütze für den Bau des tatsächlichen Systems zu dienen. Brockhaus (1989, Bd.12, S.151) definiert Modelle knapp als „Muster, Vorbild; Nachbildung oder Entwurf eines Gegenstandes“.

In diesem Sinne sind Modelle deskriptiver und konstruktiver bzw. analytischer und synthetischer Natur. Im Bereich der Softwareentwicklung spielen beide Formen von Modellen eine Rolle, und zwar in zweierlei Form. Unter Berücksichtigung der doppelten Verwendung von Verhaltensmodellen zur Maschinenkonstruktion und zum Testen ist der Zusammenhang von Problem und Lösung einerseits und der von Maschine und Umwelt andererseits zu klären.

Problem und Lösung

Neben anderen identifiziert Jackson (2001) als wesentliche Schwierigkeit beim Systementwurf und insb. im Requirements Engineering eine oftmals unzureichende Unterscheidung von Problem und Lösung. Diese klare Unterscheidung findet sich in Ansätzen zur Systemanalyse höchstens implizit (DeMarco, 1978; McMenamin und Palmer, 1984; Ward und Mellor, 1985; Hatley und Pirbhai, 1987). Zu oft würden Probleme in Begrifflichkeiten der Lösung und nicht der Problemdomäne analysiert. Wenn man ein Problem als einen zu erreichenden Zustand der Welt ansieht (Zave und Jackson, 1997) – z.B. die Versendung von Jahresabrechungen eines Versicherungsunternehmens – und Lösungen als Elemente der Realität, mit deren Hilfe dieser Zustand erreicht wird – z.B. ein System, das die Abrechnungen durchführt, Briefe verfaßt, druckt und verschickt –, dann werden im Rahmen dieser Arbeit Modelle von Lösungen betrachtet. Das bedeutet nicht, daß eine Konzentration auf das Problem in frühen Phasen der Entwicklung nicht als sinnvoll angesehen wird. Wenn sie nicht auf der Basis bestehenden Codes erstellt werden, sind die betrachteten Modelle also konstruktiver Art. Wenn Modelle zur Analyse bestehenden Codes eingesetzt werden, sind sie deskriptiver Art.

[10] Vgl. Coyne (1995, S. 219) und Jackson (2001, S. 12) zur Unterscheidung analytischer, analoger und ikonischer Modelle. Andere Modellbegriffe, wie beispielsweise der mathematische als eine Formel wahrmachende Interpretation, eines FEM- oder geometrischen Modells sowie Modellbegriffe aus anderen Wissenschaften wie z.B. Modelle des kognitiven Apparats oder der menschlichen Psyche, werden ignoriert.

Die Unterscheidung ist insofern fundamental, als aus Modellen des Problems im Normalfall weder Code noch Testfälle für die Maschine abgeleitet werden können, wohingegen das für Modelle der Lösung eher der Fall ist. Modelle des Problems sind ausschließlich dem Bereich des Requirements Engineering zuzuordnen, Modelle der Lösung auch dem Design.

Maschine und Umwelt; Testmodelle

Zur Validierung (Test, Simulation) von ausführbaren Verhaltensmodellen müssen diese im Normalfall in Abstraktionen ihres Ausführungskontextes, ihrer Umwelt, eingebettet werden. Aus verschiedenen Gründen kann es sinnvoll sein, diese Umwelt ebenfalls zu modellieren. Modelle der bereits existenten Umwelt – Betriebssysteme, Legacy-Code, Aktuatorik und Sensorik – sind dann deskriptive Modelle. Wenn die Umwelt noch nicht existiert, sind Modelle davon ggf. konstruktiver Art.

Objekt der Modellierung sind also Maschinen oder ihre Umwelt. Wenn solche Modelle für die Testfallgenerierung eingesetzt werden, hat das Konsequenzen für die Modellierung, u.U. die Partitionierung des Systems, das Abstraktionsniveau der Modelle und allgemein die Entscheidung, ob Maschine oder Umwelt modelliert werden sollen. Modelle werden in verschiedenen Hinsichten verwendet:

- Modelle der Maschine, die für die Codeerstellung verwendet werden. Die Codeerstellung kann manuell oder automatisiert erfolgen. Im ersten Fall übernehmen Modelle die Rolle der Spezifikation; im zweiten die von Code auf einem hohen Abstraktionslevel.

- Modelle der Maschine, die für die Testfallgenerierung verwendet werden. Da die Testfallgenerierung im wesentlichen als ein Suchproblem aufgefaßt wird (Kap. 3), ist der aufgespannte Suchraum möglichst klein zu halten. Im Beispiel der Chipkarte ist die Abstraktion von Zufallszahlen in einen symbolischen Wert *someRND* durch den Einsatzzweck des Modells für die Testfallgenerierung motiviert. Aus diesem Modell kann offenbar kein Code für den Zufallszahlengenerator generiert werden.

- Modelle der Maschinenumwelt, um Ausführbarkeit des Maschinenmodells zu gewährleisten, die Testfallgenerierung zu ermöglichen oder für die Testfallgenerierung den Suchraum einzuschränken.

    - Ein Ansatz des Simultaneous Engineering im Maschinenbau (Bender und Kaiser, 1995) zielt aus Gründen früher Validierbarkeit darauf ab, gleichzeitig eine Maschinensteuerung (SPS) und ein Modell einer zu steuernden Anlage zu entwickeln. Aus dem Modell der Anlage, beispielsweise einer Drahtziehanlage (Bender u. a., 2002), können dann Testfälle sowohl für die SPS, als auch für die Anlage selbst generiert werden. Die Erzeugung von Code aus dem Modell der Anlage ist für rein mechanische Teile, die nur simuliert werden, nicht sinnvoll. Auf dieses Beispiel wird auf S. 131 im Kontext der kompositionalen Testfallerzeugung erneut eingegangen.

  - Die Integration neuer Steuergeräte in das Steuergerätenetzwerk eines Automobils erfordert bei Interaktion des neuen mit den vorhanden Steuergeräten entweder (a) einen Abgleich der Abstraktionsniveaus zwischen dem Modell des neuen und dem Code der alten Geräte oder (b) eine Modellierung der Legacy-Komponenten, um ein Modell des neuen Steuergeräts zur Testfallgenerierung zu verwenden. Das betrifft erneut die Erzeugung von Testfällen für das neue Steuergerät und die Erzeugung von Integrationstestfällen.

- Modelle, die eine Menge von Szenarien codieren und insofern eine Einschränkung der Umwelt darstellen. Im Fall der Chipkarte ist ein Beispiel durch das Modell verschiedener Permutationen des Protokolls zur Berechnung digitaler Signaturen (Abschnitt 3.2.1) gegeben. Während ein solches Umweltmodell durchaus dem Zweck der Protokollverifikation dienen kann – dann wird die Umwelt zum System –, wird es für die Codeerzeugung nicht benötigt. Fehlermodelle fallen ebenfalls in diese Kategorie.

Im folgenden wird unter Testmodell sowohl ein für den Test erstelltes Maschinenmodell, als auch ein eine Menge von Szenarien codierendes Modell verstanden. Aus dem Zusammenhang wird die Lesart klar.

Zusammenfassung Wenn Modelle erstellt werden, muß klargestellt werden, ob ein Problem oder seine Lösung modelliert wird. Im Requirements Engineering spielen Modelle des Problems eine Rolle. Im Design, der Implementierung und der Durchführung von aus Modellen erzeugten Testfällen spielen eher Modelle der Lösung eine Rolle. Eine klare Unterscheidung von Umwelt und Maschine ist wünschenswert. Modelle finden Verwendung als

- Modelle der Maschine
  - zur Bestimmung von Eigenschaften dieser Maschine,
  - zur Codeerzeugung oder manuellen Codeerstellung, d.h. als Spezifikation,
  - zum Zweck der Testfallgenerierung,
- Modelle der Maschinenumwelt oder ausgewählter möglicher Verhalten dieser Umwelt (Szenarien), um
  - Ausführbarkeit einer zugehörigen Maschine oder eines Maschinenmodells zu gewährleisten oder
  - Testfälle zu erzeugen.

Modellierungssprache: Anforderungen

Nach den obigen Erläuterungen stellt sich die Frage nach einer geeigneten Sprache, mit der Modelle aufgeschrieben werden können. Wenn Modelle, die dem Test einer Implementierung dienen, nicht adäquat notiert werden können, dann werden sich Technologien zur automatischen Testfallgenerierung auch nur

schwierig durchsetzen lassen. Im Bewußtsein, die Frage nach der „Qualität“ einer Modellierungssprache hier nicht umfassend diskutieren oder beantworten zu können, wird eine Sprache dann als adäquat angesehen (vgl. Ward und Mellor (1985)), wenn

- sie die konzise Formulierung eines Sachverhalts gestattet und somit über Konstrukte verfügt, die die fundamentalen Konzepte einer Domäne beschreiben,
- sie über eine möglichst einfache und eindeutige Semantik verfügen,
- in ihr geschriebene Programme (oder Modelle) intellektuell beherrschbar und u.U. maschinell analysierbar sind und
- effizienter und u.U. zu Zertifizierungszwecken verständlicher Code aus ihr generiert werden kann.

Diese Zielsetzungen sind nicht unabhängig voneinander, wie die folgenden Beispiele zeigen:

Analysierbarkeit Je mächtiger ein Beschreibungsmittel ist, desto schwieriger ist es zu analysieren. Beispiele für ausdrucksmächtige Beschreibungsmittel sind Statecharts oder Zeiger in C.

- Statecharts mit dem Konzept hierarchischer Zustände und der damit verbundenen Notwendigkeit von History Transitions – bei Verlassen einer Zustandsebene muß der zuletzt besuchte Zustand für den Wiedereintritt in diese Ebene gespeichert werden – erlauben zwar eine knappe Formulierung (Harel, 1988), sind aber gerade in Zusammenhang mit Produktzuständen häufig schwierig zu verstehen und zu analysieren.
- Die Verwendung von Zeigern in C erlaubt ebenfalls die konzise Formulierung auch komplexer Zusammenhänge, die aber zu schwierig zu verstehenden und zu analysierenden dynamischen Objektgeflechten führt.
- Code mit GOTO-Statements kann u.U. zu effizienterem Code führen als solcher mit ausschließlich der strukturierten Programmierung zugeordneten Konstrukten, führt zugleich aber zu schwer verständlichen und analysierbaren Programmen (Dijkstra, 1968).

Es gibt einen Trend, Beschreibungstechniken für komplexe Konzepte zu verbieten:

- Das SPARK-Profil von Ada (Barnes, 1997) verzichtet u.a. explizit auf die Prozeßkommunikationsmechanismen von Ada.
- In Safer-C (Hatton, 1995) dürfen Zeiger bei maximal einer Indirektion nur zur by-reference-Übergabe von Parametern verwendet werden. Dynamische Speicherallokation, Rekursion, Union-Typen, implizite Typkonversionen und goto-Anweisungen sind verboten.

- In Java gibt es keine Zeiger.
- JavaCard verbietet dynamische Speicherallokation.

Alle diese Einschränkungen führen zu leichter verifizier- und wartbaren Programmen.

Komplexität Komplexe Semantiken wie die von Statecharts (von der Beeck, 1994) oder Message Sequence Charts (Krüger, 2000) führen zu Interpretationsschwierigkeiten. Das Problem wird noch verschärft, wenn innerhalb eines Diagramms syntaktisch zwischen verschiedenen Semantiken gewechselt wird, wie das in Live Sequence Charts (Damm und Harel, 1999) oder für verschiedene Formen des Parallelismus in ESTELLE (Budkowski u. a., 1988) geschieht. Formalisierung allein schafft hier i.a. keine Abhilfe; stattdessen scheint Einfachheit der Semantik (und damit eine vergleichsweise triviale Formalisierung) die Probleme der Verständlichkeit auszuräumen. Das wird vom Verfasser als einer der großen Vorteile des Werkzeugs AutoFocus angesehen. Darüberhinaus besitzt die Beobachtung von Brooks (1986) sicherlich nach wie vor Gültigkeit, Komplexität sei eine essentielle und keine akzidentelle Eigenschaft eines Systems. Komplexität kann also nicht immer allein aufgrund der Verwendung einer bestimmten Sprache reduziert werden.

Effizienz Codegeneratoren (Beacon, TargetLink) für kontinuierliche Matlab-Modelle erfüllen oft nur bei restriktiver Verwendung der Modellierungssprache die Effizienzanforderungen der Entwickler. Hochoptimierter Code ist seinerseits aufgrund der häufig unübersichtlichen Struktur im Normalfall nicht zertifizierbar.

Ausdrucksmächtigkeit Als erste Anforderung an Beschreibungstechniken wurde oben die knappe Formulierbarkeit auch komplexer Sachverhalte genannt. Komplexe Sachverhalte werden mit der Hilfe von Abstraktionen dargestellt. Wie im ergänzenden Abschnitt A.1 exemplifiziert, betreffen klassische Abstraktionen

- die Loslösung von Hardware („Programme“) oder registerorientierten Formalismen wie Assembler,
- den Kontrollfluß durch konditionale Strukturen, Exceptions und Konstrukte für Nebenläufigkeit und Garbage Collection, sowie
- damit eng verbunden die Struktur eines Programms in Form von Unterprogrammen, Modulen, (Unter-)Klassen, Bibliotheken und Komponenten.

Modellbasierte Formalismen stellen eine Weiterentwicklung dieser Abstraktionen dar. Das wird im übernächsten Abschnitt über Domänenabhängigkeit erläutert.

Zusammenfassung Bei der Auswahl oder Definition von Beschreibungstechniken ist der Tradeoff zwischen Ausdrucksmächtigkeit, Eindeutigkeit, Verständlichkeit und Analysierbarkeit zu berücksichtigen. Ausdrucksmächtigkeit wird durch Abstraktion gewährleistet, was die Entwicklung der Programmiersprachen zeigt. Nach Ansicht des Autors stellt AutoFocus im Vorgriff auf Abschnitt 4.2 einen gelungenen Kompromiß dar: Datenabhängigkeiten zwischen Komponenten werden durch Kanäle explizit notiert, was die Analyse erleichtert. Die zeitsynchrone Semantik ist sehr einfach, die Mischung aus erweiterten Zustandsmaschinen und funktionaler Sprache für die Spezifikation von Verhalten sehr ausdrucksstark. Wenn Modelle nicht mit angemessenen Ressourcen erstellt werden können, weil die Modellierungssprache inadäquat ist, dann werden sie nicht erstellt werden, und dann scheitert die automatisierte Ableitung von Testfällen unabhängig von der Leistungsfähigkeit der entsprechenden Testfallgeneratoren.

Domänenabhängigkeit Wenn sie zur Testfallgenerierung eingesetzt werden sollen, müssen Modelle also mit vergleichsweise geringem Aufwand erstellt werden können. Das setzt Strukturierungsmittel, Abstraktionen, voraus. Abstraktionen können offenbar die Struktur von Programmen unabhängig von einer konkreten Anwendung betreffen oder auf eine bestimmte Problemklasse zugeschnitten sein. Im allgemeinen erscheint eine (nicht vollständig orthogonale) Separierung von Modellen für Daten, Funktionalität, Struktur – z.B. logische und technische Architektur –, Kommunikation, Scheduling, Servicequalität, Informationssicherheit und Fehlertoleranz wünschenswert (Schätz u. a., 2002a; Bender u. a., 2002; Sgroi u. a., 2000; Keutzer u. a., 2000).

Diese Abstraktionen erscheinen zunächst unabhängig von einer speziellen Domäne. Die OMG-Initiative der Model-Driven-Architecture (Soley, 2000) propagiert beispielsweise plattformunabhängige Modelle (PIMs), die von der Kommunikationsinfrastruktur vollständig abstrahieren. Das geschieht im Rahmen von Businessinformationssystemen, die je nach Blickwinkel als domänenspezifisch oder nicht angesehen werden können.

Historisch finden sich andererseits viele Beispiele für domänenspezifische Ausrichtungen von Beschreibungsmitteln:

- Differentialgetriebe und Integrator in Bushs Differentialanalysator haben sich für die Problemklasse der Lösung totaler Differentialgleichungen als geeigneter erwiesen als die Verwendung von Bauteilen für die arithmetischen Grundoperationen (von Neumann, 1960, S. 17).
- Die Sprache C wurde ursprünglich für die Entwicklung von Unix auf der DEC PDP-11 konzipiert.
- Bibliotheken wie Swing stellen dedizierte Konstrukte für die Erstellung graphischer Oberflächen zur Verfügung.
- Frameworks und Architekturen für bestimmte Problemklassen haben sich bewährt, z.B. für client-server-Architekturen oder Businessinformationssysteme (Siedersleben und Denert, 2000).

- Die Sprache TTCN-3 dient der Beschreibung von Testfällen und wird in Abschnitt 3.5 beschrieben.

Der Vorteil einer domänenspezifischen Ausrichtung von Beschreibungsmitteln wird in einer erhöhten Effizienz bei der Systementwicklung sowie in der Möglichkeit der Entwicklung spezialisierter Validierungs- und Verifikationstechniken gesehen. Je allgemeiner ein Problem ist, desto schwieriger ist normalerweise seine Lösung. Ein Grund für die Erfolge beispielsweise formaler Verifikationstechniken im Bereich des Chipentwurfs mag in der Existenz domänenspezifischer Abstraktionen auf verschiedenen Ebenen, z.B. Transistor- und Gatterebene, sowie für Standardkomponenten wie Pipelines liegen (vgl. Qadeer und Tasiran (2002)).

Im ergänzenden Abschnitt A.2 werden weitere Beispiele für Domänenspezifika der Programmiersprachen APT, Fortran, COBOL, Lustre, Esterel, LISP, Prolog sowie von Hardwarebeschreibungssprachen, Entwurfsmustern und Prozeßmustern angeführt.

Beispiel 7 (Adäquatheit der Modellierungsprache). *Bei der Entwicklung des Modells der Chipkarte hat sich herausgestellt, daß erweiterte Zustandsmaschinen mit einer ausgezeichneten Projektion des Zustandsraums in Form von Kontrollzuständen für die Domäne der Chipkarten nicht uneingeschränkt geeignet sind. Der Grund ist, daß es eine solche ausgezeichnete Projektion hier nicht gibt. Explizite Modellierung des Produkts der Datenzustände durch Kontrollzustände oder Auslagerung jeder Komponente des Datenzustands in eine eigene* AutoFocus-*Komponente führt zu exponentieller Vergrößerung und Unübersichtlichkeit des Modells, weshalb die Funktionalität in die textuellen Funktionsbeschreibungen ausgelagert wurde. Die Automaten bestehen im Normalfall aus einem Kontrollzustand, höchstens aus drei Kontrollzuständen. Diese Funktionalität wurde vorher in Form von Tabellen erfaßt und dann von Hand in funktionale Programme konvertiert. Konsequenz ist, daß für die Modellierung der Chipkarte Tabellen geeigneter erscheinen als endliche erweiterte Zustandsmaschinen.*

Zusammenfassung Domänenspezifische Beschreibungstechniken für die Modellerstellung erlauben wegen der bewußten Einschränkung ihrer Ausdrucksmächtigkeit die Erstellung knapper und analysierbarer Artefakte. Für den effizienten Einsatz von Testfallgeneratoren ist zum einen die einfache Erstellbarkeit von Modellen und zum anderen ihre Analysierbarkeit Voraussetzung. Für den domänenspezifischen Einsatz in verteilten Systemen mit zeitsynchroner Kommunikation etwa erscheint AutoFocus wegen der zugrundeliegenden Ausführungssemantik geeignet. Für nicht-verteilte Systeme wie die Chipkarte bietet die zeitsynchrone Kommunikation die Möglichkeit der sauberen Spezifikation der Datenabhängigkeiten zwischen Komponenten, ausgedrückt durch Kanäle.

Modellierungssprache: Separation of Concerns

Die oben aufgezählten Abstraktionen werden im Rahmen der Aspektorientierung ebenfalls betrachtet. Als Ergänzung traditioneller Programmiersprachen definiert AOP dedizierte syntaktische Konstrukte für bestimmte Teile eines Programms, beispielsweise Fehlerbehandlung oder Kommunikation. Kiczales u. a. (1997) definieren einen Aspekt als „eine Eigenschaft, die nicht sauber in einer generalisierten Prozedur gekapselt werden kann“. Damit wird die inhärente Schwierigkeit der Orthogonalisierung gespiegelt, die durch das Konzept der „Separation of Concerns“ ebenfalls angestrebt wird.

Der Vorteil einer solchen konzeptionellen Trennung liegt, soweit machbar, in der Möglichkeit

- der zeitlich oder logisch unabhängigen Entwicklung von Modulen
- und, im Rahmen dieser Arbeit relevanter, des getrennten Tests von Modulen

für die einzelnen Aspekte. Kommunikationsdetails eines bestimmten Bussystems mögen erst dann relevant erscheinen, wenn die Kernfunktionalität eines Systems bereits entworfen ist; Realzeitanforderungen werden möglicherweise in frühen Entwicklungsphasen als noch nicht relevant angesehen. Gerade das zweite Beispiel zeigt aber, daß zumindest nach heutigem Stand der Kunst Performanceerwägungen bereits beim ersten Grobdesign der Funktionalität berücksichtigt werden müssen. So, wie in Businessinformationssystemen heute von Kommunikationsdetails aufgrund gekapselter Infrastrukturen wie J2EE oder .NET abstrahiert werden kann, ist es ebenfalls vorstellbar, eine ähnliche Kapselung für die Trennung von Berechnung und Kommunikation mit Konsequenzen für das zeitliche Verhalten zu erzielen. Vorstellbar ist beispielsweise die Beschreibung von Systemen mit AutoFocus, die aufgrund ihrer konzeptionellen Nähe zu zeitgesteuerten Architekturen (TTA) zunächst unabhängig vom verwendeten Protokoll (TTP, ByteFlight, FlexRay) modelliert werden.

Der Vorteil der systematischen Trennung liegt auch für die Qualitätssicherung auf der Hand. Testfälle können für die einzelnen Aspekte separat berechnet und später integriert werden. Soweit die Trennung machbar ist, ermöglicht sie aufgrund der Konzentration auf jeweils einen wesentlichen Teil nicht nur die intellektuelle Durchdringung des Problems, sondern wegen der resultierenden Vereinfachung auch bessere Chancen, für diesen Teil systematisch automatisiert Testfälle zu errechnen.

Zum Zweck der Konsistenzsicherung sollten diese Aspekte dedizierte Teile eines expliziten integrierten Produktmodells sein – was bei AOP (noch) nicht der Fall ist –, das sowohl abstrakte Syntax (Konzeptmodell) als auch Semantik (Systemmodell) sowie Konsistenzbedingungen festlegt. Die Aspekte können dann als Sichten (Huber u. a., 1997) auf das umfassende Produktmodell angesehen werden. Flexibilität ist gewährleistet, wenn für jeden Aspekt verschiedene Beschreibungstechniken zur Verfügung stehen, wenn Verhaltensbeschreibungen in Form von Zustandsübergängen also durch Tabellen oder verschiedene Automatendialekte dargestellt werden können.

Die genannte Separierung von Aspekten ist bis heute nicht zufriedenstellend gelöst. Das liegt zum einen daran, daß Hersteller von CASE-Tools aus ökonomischen Gründen eine Konzentration auf spezialisierte Anwendungsdomänen oftmals als nicht sinnvoll erachten. Gravierender ist, daß – ob Ursache oder Konsequenz, sei dahingestellt – für viele der o.g. Aspekte gekapselte Abstraktionen im Sinne allgemein akzeptierter ontologischer Entitäten z.B. für das Scheduling noch nicht gefunden sind. Richtig eingesetzt, bietet beispielsweise AutoFocus die Möglichkeit, zwischen Berechnung und Kommunikation zu trennen. Diese Teillösung erfordert allerdings im Normalfall eine Auslagerung wesentlicher Teile der Funktionalität in die textuelle Funktionsbeschreibungssprache.

Zusammenfassung Um der wachsenden Systemkomplexität Herr zu werden, müssen die Systeme in möglichst orthogonale Aspekte der Entwicklung zerteilt werden. Die Verquickung aller Aspekte macht eine intellektuelle Beherrschung unmöglich, weshalb es keine Alternative zu diesem Vorgehen zu geben scheint, auch wenn Orthogonalität nicht immer gegeben ist. Diese Beobachtung ist erneut unabhängig davon, ob Modelle als Grundlage für eine Implementierung oder als Grundlage für Testfallgenerierung dienen.

Einfluß der Modellierungssprache auf das Testen

Als Ausprägung der Eigenschaft der Analysierbarkeit haben die Konzepte der Modellierungssprache Einfluß auf Effektivität und Effizienz der Testfallgenerierung. Dynamische Speicher- oder Objektallokation, dynamisches Binden bei Polymorphie, die freie Verwendung von Zeigern oder Statecharts mit tiefen Hierarchien und entsprechenden History-Transitionen sind Beispiele für Modellierungskonstrukte, die eine Analyse des Programms erschweren (s. S. 45). Systeme mit komplexem zeit-asynchronen Nachrichtenfluß vergrößern offenbar den Zustandsraum. Wenn Systeme adäquat mit zeitsynchronem Nachrichtenfluß modelliert werden können, ist diese Form der Kommunikation zum Testen also vorzuziehen.

Das ist nicht immer möglich, wenn beispielsweise gerade die Kommunikationsinfrastruktur getestet werden soll oder ein System in einer zeitasynchronen Umgebung wie einem CAN-Bus zum Einsatz kommt und von dieser Tatsache nicht abstrahiert werden kann. Daß Beschreibungstechniken also nicht beliebig einfache Semantiken zugewiesen werden können, weil das zu umständlichen Modellen führen würde, wurde weiter oben bereits diskutiert.

AutoFocus gestattet die einfache Spezifikation von Produkt- und Summentypen. Die Erfahrung zeigt, daß diese Typdefinitionen ein sehr mächtiges Mittel zur eleganten Spezifikation reaktiver Systeme darstellen und demzufolge häufig Verwendung finden. Ein Testfallgenerator muß dann in der Lage sein, mit solchen Typen umzugehen. Wenn die Spezifikationssprache des Testfallgenerators Produkttypen nicht erlaubt, dann ist i.a. eine Aufzählung aller möglichen Produkte erforderlich, was den Zustandsraum sehr schnell sehr groß werden läßt. Der in Kapitel 4 vorgestellte Testfallgenerator kann ohne explizite Produktbildung mit solchen Typen umgehen.

In Abschnitt 4.6 wird ein Verfahren zur automatisierten Berechnung von Integrationstests präsentiert. Das geschieht auf der Grundlage von vorher erzeugten Unit-Tests. Grob gesagt, ist ein Problem bei Unit-Tests die Tatsache, daß Units implizite Vorbedingungen beinhalten. Es wird darauf vertraut, daß diese Vorbedingungen durch Integration mit dem Rest des Systems eingehalten werden. Wenn nun Units allein getestet werden, dann können Testfälle entstehen, die niemals im System ablaufen können. Wenn ein Modellierungsformalismus die explizite Formulierung solcher Vorbedingungen gestattet (und diese Möglichkeiten im Modell auch genutzt werden), dann können Unit-Tests, die niemals Teil tatsächlicher Systemabläufe darstellen, zu einem großen Teil ausgeschlossen werden.

Im letzten Abschnitt wurde auf die eventuelle Notwendigkeit von Nichtdeterminismus in Testmodellen hingewiesen. Offenbar muß der Modellierungsformalismus für diese Art von Modellen also Nichtdeterminismus zulassen. In Abschnitt 2.3.2 wurde erläutert, daß bestimmte Teile der Maschine nicht mitmodelliert werden müssen (kryptographische Operationen, Zufallszahlen). Das Zusammenspiel zwischen Testtreiber und Modell – die Implementierung der Abstraktions- bzw. Konkretisierungsabbildung – wurde in der Fallstudie adhoc ermöglicht. Zum Zweck der Verfolgbarkeit wäre eine Einbettung von Informationen über den Treiber in das Modell wünschenswert, d.h. die Modellierungssprache sollte über verschiedene Abstraktionsstufen gleichzeitig Aussagen treffen können.

2.3.2. Verschiedene Abstraktionsstufen

Der Begriff der Abstraktion kann nicht nur horizontal – im Sinne ontologischer Komplexe –, sondern auch vertikal – im Sinne verschiedener Präzisierungen z.B. durch Verfeinerung eines einzelnen Aspekts – verstanden werden. Die durch die Terminologie suggerierte Orthogonalität ist dabei, je nach Sichtweise, nicht unbedingt gegeben: Die Abstraktion von Kommunikation (z.B. durch gekapselte Sende- und Empfangsprimitive) oder zeitlichem Verhalten (durch Kausalität) kann sowohl horizontal als auch vertikal aufgefaßt werden.

Wie bereits erwähnt, erlaubt die Konzentration auf einzelne Aspekte die Beherrschung komplexer Zusammenhänge. Zum Zweck eines eingehenden Verständnisses und der Beseitigung von Mehrdeutigkeiten in Pflichtenheft und Spezifikationsdokumenten dienen die entstehenden Modelle im Rahmen der Systementwicklung als Prototypen. Ausführbarkeit scheint mindestens in späten Phasen des den Prozeß begleitenden Requirements Engineerings eine wesentliche Hilfe beim Verständnis der Systeme zu sein.

In den i.a. nicht ausführbaren Beschreibungstechniken von Erweiterungen der Strukturierten Analyse um Zustandsübergangsdiagramme oder -maschinen werden ebenfalls verschiedene Abstraktionsniveaus propagiert. McMenamin und Palmer (1984) unterscheiden zwischen essentiellen und Implementierungsmodellen,[11] Hatley und Pirbhai (1987) zwischen Systemanforderungs- und Architekturmodellen. Essentielle System- oder Systemanforderungsmodelle dienen im

[11] In der zitierten Arbeit werden sie „Inkarnationsmodelle“ genannt; der Terminus „Implementierungsmodelle“ stammt von Ward und Mellor (1985).

wesentlichen der intellektuellen Durchdringung eines Problems bzw. eigentlich seiner Lösung. Allein der Vorgang der präzisen Modellierung setzt eine gezwungenermaßen intensive Auseinandersetzung mit dem Problem voraus, und die Erfahrung zeigt, daß allein dieser sonst nicht unbedingt erfolgende Vorgang viele Unstimmigkeiten des Pflichtenhefts oder der Spezifikation aufzeigt. Implementierungsmodelle sind mehr oder weniger abstrakte Beschreibungen des zu entwickelnden Systems und dienen als Blaupause für dessen Codierung. Explizit für den Test erstellte Modelle dienen der systematischen Testfallerzeugung.

Codegenerierung Wenn dem Wunsch nach automatisierter Codeerzeugung aus Modellen Rechnung getragen werden soll, dann ergibt sich der fundamentale Widerspruch, daß Modelle gleichzeitig abstrakt und konkret sein sollen. Hier wird die Doppelnatur von Abstraktionen im Sinne ontologischer Komplexe und im Sinne verschiedenen Präzisierungsstufen offenbar. Im ersten Fall kann beispielsweise die Kommunikation unter Vernachlässigung der Kommunikationsinfrastruktur mit geschichteten Protokollen formuliert und automatisch durch voll funktionalen Code ersetzt werden. Im zweiten Fall ist das normalerweise nicht möglich: Information, die nicht explizit im Modell oder in der Semantik seiner Beschreibungstechniken gegeben ist, muß vom Systementwickler geliefert werden. Andernfalls, so Lem (1964, S. 305), wäre das „ein wahres Perpetuum mobile der Information“ – eine Variation der berühmten Beobachtung zur Komplexität von Brooks (1986).

Die Einbettung generierten Codes in eine teils tatsächlich existente, teils simulierte Umwelt, erfordert also entweder die Aufnahme beliebig feingranularer Details in das Modell selbst oder Adapter, die die verschiedenen Abstraktionsniveaus miteinander in Bezug setzen. Wenn im Fall eingebetteter Systeme das Modell mit Signalen operiert, die abstrakter sind als die von der Hardwareabstraktionsschicht oder dem Betriebssystem zur Verfügung gestellten, dann müssen die Signale des Modells geeignet konkretisiert werden, um das tatsächliche System oder Hardware- bzw. Software-in-the-loop-Simulationen ablaufen zu lassen. Für die Durchführung von Tests gilt dasselbe: Abstrakte Kommandos eines Chipkartenmodells müssen in APDUs übersetzt werden, und ebenso müssen abstrakt modellierte Kommandos zum Test von Prozessoren in tatsächliche Instruktionsfolgen des entsprechenden Prozessors konvertiert werden, vgl. z.B. Geist u. a. (1996); Dushina u. a. (2001).

Beispiel 8 (Abstraktionsniveaus). *Im Modell der Chipkartenanwendung tauchen tatsächliche Ziffernfolgen als PINs nicht auf. Stattdessen wird mit abstrakten, symbolischen Werten wie „PinGRef“ oder „PinNRRef“ gerechnet, die von der Testdurchführungsumgebung, dem Testtreiber, durch tatsächliche PINs ersetzt werden. Das geschieht unter Rückgriff auf eine Konfigurationsdatei. In dieser Umgebung werden außerdem im Modell nicht oder nicht adäquat beschreibbare Zusammenhänge konkretisiert.*

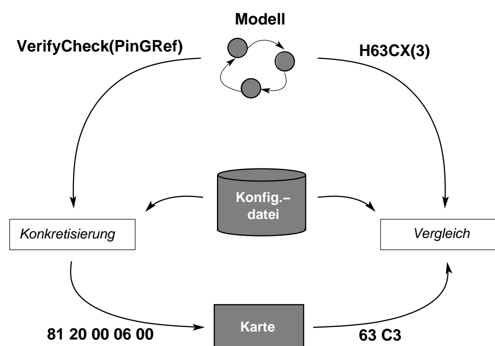
- *Ein Beispiel für nicht im Modell beschreibbare Zusammenhänge ist der Zufallszahlengenerator. Zufallszahlen, die im Modell erzeugt werden, sind immer unterschiedlich von den in der tatsächlichen Karte erzeugten. Im*

Modell verwendete abstrakte Werte dienen als Platzhalter für tatsächlich von der Karte erzeugte.

- *Ein Beispiel für einen insbesondere zum Zweck des Testens im Modell nicht adäquat beschreibbaren Vorgang ist der der Berechnung einer digitalen Signatur. Im Modell wird deshalb mit abstrakten Werten gerechnet, die alle zur Erzeugung der Signatur notwendigen Daten (Schlüssel, Zertifikate) enthalten. In der Testdurchführungskomponente werden aus diesen abstrakten Werten dann tatsächliche Signaturen berechnet, die mit den von der Karte berechneten Signaturen verglichen werden.*

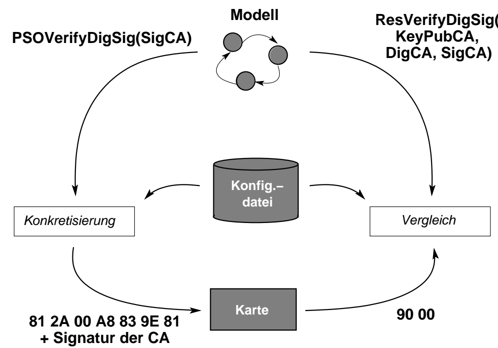
- *Einige Befehle (vor allem die Cardholder Verification-Befehle, aber auch die meisten Befehle zur Modifikation der Sicherheitsumgebungen) sind im Modell so präzise dargestellt, daß die Antwort aus dem Modell direkt zum Vergleich mit der Kartenantwort herangezogen werden kann.*

  Ein typischer Vertreter dieser Klasse ist der Befehl „Verify" mit leerer PIN-Sequenz. Er dient zur Abfrage der noch verbliebenden Verifikationsbefehle für eine PIN. Da die Cardholder-Verification-Funktionalität im Modell präzise nachgestellt wurde, sieht der Ablauf im Kommando-Interpreter wie folgt aus: Auf die abstrakte Eingabe VerifyCheck(PinGRef) folgt die abstrakte Ausgabe H63CX(3). Nach Konkretisierung ergibt sich für die Eingabe die APDU 81 20 00 06 00 und für die Ausgabe 63 C3.



  Die Kartenantwort kann direkt mit der Modellantwort verglichen werden (letztere wird dazu in die von der Karte zu erwartende Byte-Sequenz übersetzt, der Vergleich ist damit trivial).

- *Komplizierter ist die Behandlung der kryptographischen Befehle. Prinzipiell ließen sie sich, da sie deterministisch sind und auf bekannten Algorithmen beruhen, im Modell nachbilden. Praktisch sind aber die Modellierungssprachen von* AutoFocus *(und anderer CASE-Tools) dafür ungeeignet. Für die Testfallgenerierung würde darüberhinaus der Zustandsraum zu groß. Die folgende Abbildung zeigt am Beispiel der Überprüfung einer digitalen Signatur das Vorgehen bei dieser Befehlsklasse:*



> *Statt einer konkreten zu erwartenden Antwort liefert das Kartenmodell eine symbolische Antwort mit allen zur Überprüfung der Signatur nötigen Parametern. Auf die abstrakte Eingabe PSOVerifyDigSig(SigCA) wird die abstrakte Antwort ResVerifyDigSig(KeyPubCA, DigCA, SigCA) geliefert: der öffentliche Schlüssel der Zertifizierungsbehörde, der Digest und die Signatur. Die ersten beiden Parameter wurden mit vorangehenden Kommandos gesetzt.*

Für die Codegenerierung eines Zufallszahlengenerators beispielsweise ist das Modell offenbar ungeeignet. Für die Testfallgenerierung hingegen ist es geeignet und kann auch als ausführbare Schnittstellenspezifikation aufgefaßt werden.

2.3.3. Zusammenfassung Modellbasierung

Modellbasierte Entwicklung und modellbasiertes Testen beruhen auf der Verwendung abstrakter Sprachen zur Formalisierung von Sachverhalten. Diese Sprachen sollen

- domänenspezifisch sein, um
  - einerseits die relevanten Zusammenhänge einer Domäne konzise ausdrücken zu können und
  - andererseits die Vorteile eingeschränkter Beschreibungsmittel in bezug auf ihre Analysierbarkeit ausnutzen zu können,
- soweit das möglich ist, die separate und integrierbare Beschreibung unterschiedlicher Aspekte eines Systems ermöglichen, und die Sprachen sollen
- die Formalisierung und Integration verschiedener Abstraktionsstufen der einzelnen Aspekte erlauben, um im Sinne einer – mathematisch nicht unbedingt zu präzisierenden – schrittweisen Verfeinerung der Komplexität Herr werden zu können.

Das zentrale Problem ist der Wunsch, gleichzeitig abstrakt und zu verschiedenen Zwecken – Testfallgenerierung, Codeerzeugung – hinreichend präzise zu sein.

In diesem Abschnitt wurden bewußt eher Anforderungen als Lösungen formuliert. Die Suche nach geeigneten Modellierungssprachen mit Werkzeugunterstützung bedarf weitergehender Anstrengungen. Der über den Einsatz von

AutoFocus und die in dieser Arbeit beschriebene Technologie hinausgehende Erfolg des modellbasierten Testens ist davon abhängig, ob sich weitere adäquate und in der Praxis einsetzbare Beschreibungstechniken für Modelle finden lassen.

2.3.4. Inkrementalität, Refactorings und Regressionstests

Eins der Hauptprobleme des Software- und Systems Engineering ist, daß sich die Anforderungen während der Systementwicklung ändern. Im Automobilbereich beispielsweise passiert es, daß Zulieferer ihre Versprechungen nicht halten können, oder daß entschieden wird, ein Subsystem mit weiterer Funktionalität zu versehen. Evolutionäre oder spiralartige Prozesse begegnen diesem Umstand, indem das System schrittweise aufgebaut wird und in kurzen Abständen Rücksprache mit dem Auftraggeber gehalten wird. Das führt zu einer Klärung von Anforderungen und hilft, Mißverständnisse auszuschließen.

Interaktion mit Kunden gestaltet sich einfacher, wenn ausführbare Artefakte, Prototypen, präsentiert werden können. Ausführbare Verhaltensmodelle stellen solche Prototypen dar. Sie können später als Grundlage automatisierter Codeerzeugung dienen. Attraktiv erscheint auch der Einsatz als ausführbare Spezifikation, anhand derer die Implementierung manuell erstellt wird – wenn beispielsweise effiziente Codegeneratoren nicht existieren – und anhand derer Testfälle generiert werden, die dem Test der Implementierung dienen.

Die Erweiterung einer Entwicklungsstufe eines Systems wird als Inkrement bezeichnet. Einschränkungen dieses vagen Begriffs wie im Cleanroom Software Engineering, die ein Inkrement als vollständig vollendetes Subsystem ansehen, werden nicht getroffen. Eine Präzisierung erfolgt bei Philipps (2003).

Beispiel 9 (Inkremente). *Beispiele für Inkremente im Fall der Entwicklung des Systems zum Test der Chipkarte sind*

- *das Hinzufügen einer Komponente für das Dateisystem, nachdem eine Komponente für die Verwaltung von PINs und PUKs erstellt wurde (funktionale Dekomposition).*
- *Änderungen in der Komponente für die Verwaltung von PINs und PUKs, nachdem bei Entwicklung der Komponente für Sicherheitsoperationen (z.B. Berechnung einer digitalen Signatur) Unklarheiten in der Spezifikation geklärt wurden.*
- *die Erstellung von Software zur Berechnung von Testfällen, des Testtreibers zur Kommunikation mit dem Terminal und zur Übersetzung symbolischer Datentypen in für die Karte verständliche Bytefolgen.*
- *Änderungen in Modell und Testtreiber nach Untersuchung der Verdikte, die zur Klärung von Unstimmigkeiten in der Spezifikation führten. Der Einwand, hier würde das Modell solange geändert, bis es zum tatsächlichen System passe – was nicht Motivation des Testens sein sollte – ist berechtigt. Allerdings sind diese Änderungen auf Interpretationen der Spezifikation zurückzuführen, die durchaus auch anders hätten ausfallen können: Die Karte verhält sich gemäß der Spezifikation nicht „falsch“.*

Im folgenden werden Inkremente ausschließlich auf das Modell bezogen, also beispielsweise nicht auf die Entwicklung des Testtreibers. Die Einbettung der Modellierungsaktivitäten in den Entwicklungsprozeß erfolgt in Abschnitt 2.4.

Testen und Inkrementalität

Testdaten können für jedes Modellinkrement generiert werden. Falls eine weitere formalisierte Spezifikation des Systems vorhanden ist, was im Fall der Chipkarte nicht der Fall ist, können die Testdaten sogar zu Testfällen für das Modell vervollständigt werden: Das ausführbare Modell wird dann gegen diese formalisierte Spezifikation getestet. Die formalisierten Spezifikationen müssen ihrerseits validiert werden. Wenn solche formalen Spezifikationen nicht vorliegen, müssen die Ausgaben des Modells manuell validiert werden. Der Vorteil einer automatisierten Testdatengenerierung z.B. auf der Basis von Überdeckungsmetriken als Testfallspezifikation liegt in der im Vergleich zur rein manuellen Validierung erhöhten Systematik.

Wenn ein Inkrement zusätzliche Funktionalität einführt, die vermeintlich orthogonal zu Teilen der Funktionalität früherer Inkremente ist, dann können Testfälle – die Ausgaben wurden ja bereits validiert – für diese als invariant angenommene Funktionalität verwendet werden. Es findet also ein Regressionstest auf Modellinkrementebene statt (Abb. 2.5). Darüberhinaus können Testfälle für ein Inkrement als Grundlage für neue Testfälle für ein späteres dienen. Eine Technik zur kompositionalen Erzeugung von Testfällen, die das Fortschreiten der inkrementellen Entwicklung als Grundlage für die Entwicklung von Testfällen hat, wird in Abschnitt 4.6 vorgestellt.

Im iterativen Prozeß wird also zunächst ein erstes Modell erstellt. Zusammen mit einer vom Testingenieur oder Entwickler gelieferten Testfallspezifikation werden daraus automatisch Testdaten generiert, die der Validierung des Modells dienen. Zusätzliche Maßnahmen wie Reviews werden helfen, das Vertrauen in die Validität zu erhöhen. Wenn das Modell als valide angesehen wird, wird es erweitert. Bei Entwicklung des erweiterten Modells können potentielle Mißverständnisse im Vorgängermodell detektiert und ausgeräumt werden.[12] Die zwei Modelle beeinflussen sich also wechselseitig. Dann werden neue Testdaten für das erweiterte Modell generiert. Das geschieht anhand neuer Testfallspezifikationen sowie der Testfälle aus dem Vorgängermodell – jetzt handelt es sich um Testfälle und nicht mehr um Testdaten, weil die Ausgaben des Vorgängermodells als valide angesehen werden. Das Vorgehen wird iteriert, bis am Ende ein valides Systemmodell zur Verfügung steht. Aus diesem können dann auf Basis der im folgenden Abschnitt 2.4 diskutierten Entwicklungsszenarien (Codeentwicklung vor, nach, oder gleichzeitig mit Modellentwicklung) Testfälle für die Maschine generiert werden (vgl. auch Sax u. a. (2002) zum entwicklungsbegleitenden Test eingebetteter Systeme).

Dasselbe gilt für Transformationen, die das Verhalten nicht ändern sollen. Ein geeigneter Beobachtungsbegriff wird dabei vorausgesetzt (Philipps und Rumpe, 2001); im Fall von AutoFocus-Modellen ist er wegen der Abstraktion

[12] Vgl. Rushby (2002), der die Verwendung von Model Checking zur Detektion von Differenzen zwischen verschiedenen Systemen bzw. Inkrementen vorschlägt.

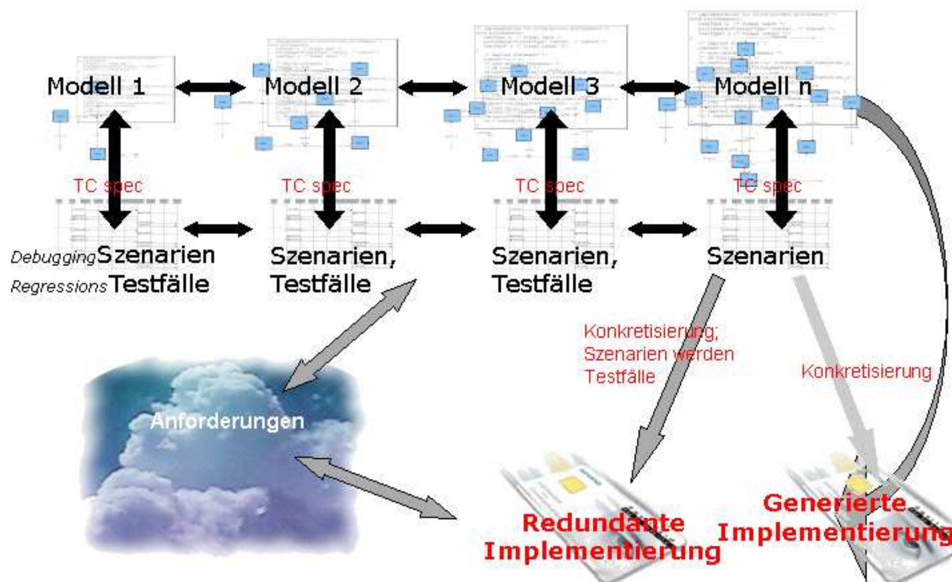


Abbildung 2.5.: Modellierung und Test im inkrementellen Entwurf

von zeitlichen Aspekten im Normalfall trivial. Solche strukturellen Änderungen werden als Refactorings (Fowler, 1999) bezeichnet. Wenn die Korrektheit solcher Transformationen nicht a priori nachgewiesen wurde, dann können Testfälle generiert werden, die bei erfolgreichem Ablauf das Vertrauen in die Korrektheit der Transformation erhöhen. Der Vorteil liegt hier insbesondere bei vollautomatisierten Verfahren darin, daß der Preis für (a) die beispielsweise zufällige Erzeugung großer Mengen von Testfällen und für (b) deren Durchführung praktisch gleich Null ist.

2.4. Modelle für Codeerzeugung, Verifikation und Validierung

Modelle als ausführbare Repräsentationen einer Spezifikation können zur Qualitätssicherung zweierlei Funktionen erfüllen. Zum einen können sie der manuellen oder automatisierten Test*daten*generierung dienen. Zum anderen können sie als Orakel Verwendung verwenden, wenn für eine Menge von Stimuli die entsprechende Menge von Ausgaben (kombiniert: Testfälle) bestimmt werden muß. Schließlich können Modelle die Grundlage der Systemimplementierung darstellen. Modelle dienen also

- der Validierung des Modells (Testdaten werden erzeugt, die entsprechenden Ausgaben manuell validiert),

- der Verifikation einer Maschine (das bereits validierte Modell dient als Referenz) und
- als Grundlage einer manuellen oder automatisierten Implementierung.

In diesem Abschnitt wird auf der Grundlage von Pretschner und Philipps (2002) diskutiert, wie die Entwicklung von Modellen und Code miteinander verwoben werden kann. Dabei wird besonderes Augenmerk auf Modelle als Grundlage für eine automatisierte Testfallgenerierung gelegt. Die drei unterschiedlichen Szenarien können selbstverständlich miteinander kombiniert werden. Ein viertes Szenario wurde bereits auf S. 42 im Kontext des Simultaneous Engineering (Bender und Kaiser, 1995) im Maschinenbau diskutiert. Wenn Modelle einer zu steuernden Anlage zum Test der Steuerung erstellt werden, dann kann es wie im Fall großer Teile einer Drahtziehanlage sein, daß gar kein Code erzeugt werden soll, weil der entsprechende Teil ausschließlich mechanischer Natur ist. Auf diesen Fall soll hier nicht weiter eingegangen werden.

2.4.1. Codegenerierung aus einem Modell

Wenn Modelle einer Maschine so präzise sind, daß sie sich als Grundlage der Überprüfung von Verhaltensübereinstimmungen einsetzen lassen, dann stellt sich aus ökonomischer Sicht die Frage, ob der Code nicht automatisch aus dem Modell generiert werden sollte.

- Das ist nicht zwingend für alle Aspekte eines Systems der Fall; denkbar ist beispielsweise ein Modell eines verteilten Systems, das von der Kommunikationsinfrastruktur abstrahiert und somit zur Testfallgenerierung, nicht aber direkt zur Generierung des Systemcodes geeignet ist. Für isoliert betrachtete, gekapselte Steuergeräte hingegen ist die automatisierte Codegenerierung vorstellbar.
- Das Beispiel der Behandlung von Zufallszahlen im Modell der Chipkarte zeigt, daß ein Modell zwar zur Testfallgenerierung geeignet sein kann. Für die Codegenerierung ist es aber zu abstrakt.

Codegeneratoren für Matlab-Modelle wie Beacon (ADI) oder TargetLink (dSpace) oder im Werkzeug ASCET-SD (ETAS) sind Beispiele für diesen bereits heute partiell erfolgreichen Ansatz. Schwierigkeiten ergeben sich im Bereich eingebetteter Systeme insbesondere aus der Integration verschiedener Abstraktionsebenen (Zusammenführen von Funktionalität auf Modellebene mit Hardware, Legacy-Systemen und Echtzeitbetriebssystemen; Deployment), die bis jetzt noch nicht befriedigend gelöst sind.

Die automatisierte Ableitung von Produktionscode aus Modellen setzt also voraus, daß geeignete Codegeneratoren existieren, die Effizienz- und Lesbarkeitsanforderungen des resultierenden Codes gewährleisten. Für die Domäne der Chipkarten existieren solche Codegeneratoren noch nicht. Modelle dienen dann als Spezifikationen oder Prototypen. Dieses Szenario wird in Abschnitt 2.4.3 untersucht.

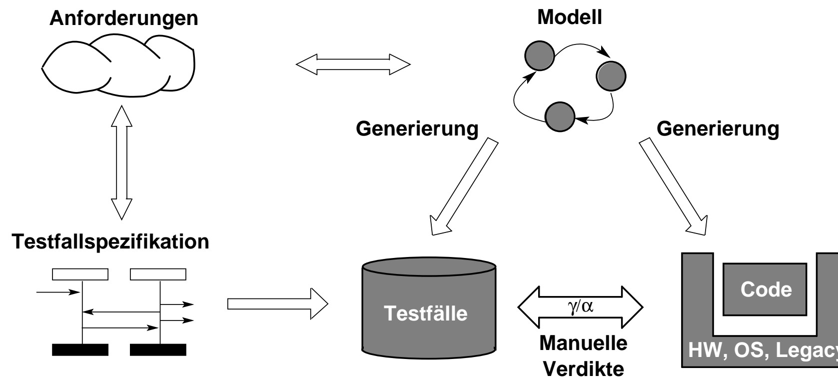


Abbildung 2.6.: Code und Testfälle aus Modell generiert

Abb 2.6 verdeutlicht den Zusammenhang mit der Ableitung von Testfällen. Ausgehend von den Anforderungen, werden Modell und Testfallspezifikationen erstellt. Aus dem Modell werden dann Testfälle und Produktionscode generiert. Wenn die Modelle zur Testfallgenerierung herangezogen werden, dann können sie nicht gleichzeitig als Orakel für eine automatisch erzeugte Implementierung fungieren: Der generierte Code würde „gegen sich selbst" getestet werden. Deshalb müssen Verdikte nach der Transformation von Testfällen (Konkretisierung γ der Eingaben, Abstraktion α der Ausgaben; vgl. S. 35) manuell gebildet werden.

Die Verwendung von Modellen für die Testfallgenerierung in diesem Szenario ist trotzdem nicht ganz unsinnig. Zum einen ist es u.U. einfacher, für bereits existierende Stimuli die erwarteten Ausgaben manuell zu validieren, als diese Stimuli selbst zu finden. Vielversprechend ist in diesem Szenario also die Verwendung von Modellen zur Test*daten*generierung für die manuelle Validierung von Maschinen. Durch den höheren Abstraktionsgrad von Modellen und die eingängigen Beschreibungstechniken erfordern Reviews eines Modells weniger Einarbeitung als in Code; Fragen zur Systemfunktionalität lassen sich (auch zusammen mit dem Auftraggeber) effizienter klären.

Zum anderen können die Testfälle zur Überprüfung von Umweltannahmen und der Korrektheit von Codegeneratoren oder Übersetzern und insb. Optimierern (Simon, 1999) verwendet werden. Insbesondere im Kontext kontinuierlicher Systeme, in denen ein formaler Nachweis der Übereinstimmung von Simulations- und Produktionscodegeneratoren wegen auftretender numerischer Probleme i.a. sehr schwierig ist, wird der Sinn eines solchen Ansatzes deutlich. Hier ist das Problem, daß die Semantik der Codegeneratoren nicht exakt festgelegt oder festlegbar ist und zwei Generate (Produktions- und Simulationscode) u.U. sogar mit Werkzeugen zweier verschiedener Hersteller erstellt werden, wie das Beispiel der Erzeugung von Produktionscode aus Matlab-Modellen mit TargetLink oder Beacon illustriert. Ein weiteres Problem sind Umweltannahmen, die beispielsweise in Form einer zeitkritischen Einbettung in den Hardware- und Betriebssystemkontext Einfluß auf die korrekte Funktion des generierten Codes ausüben.

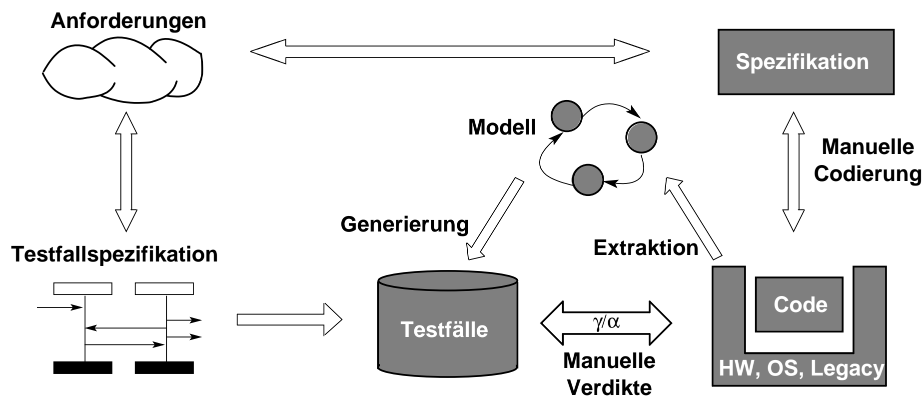


Abbildung 2.7.: Modell entsteht nach Code

2.4.2. Modellextraktion aus Code

Wenn die Wahl auf eine codezentrierte Entwicklung fällt, dann ist der Code oft zu kompliziert, als daß er direkt zur Verifikation herangezogen werden könnte. Im Bereich des Model Checking und auch der Testfallerzeugung gibt es Ansätze, halbautomatisch Modelle aus Code zu erzeugen,[13] für die dann Eigenschaften überprüft werden. Abb. 2.7 verdeutlicht das Szenario. Ausgehend von den Anforderungen, werden zunächst Spezifikation und Testfallspezifikationen entwickelt. Aus dem Code wird dann das Modell extrahiert, das als Grundlage für die Berechnung von Testfällen dient.

Rückschlüsse vom Modell auf den Code sollten möglich sein, die Abstraktion konservativ (z.B. Dams u. a. (1997); vgl. die Diskussion der Problematik der Abstraktion bei existentiellen Eigenschaften auf S. 80). Testfälle sind mit der Inversen der Abbildung der Codeabstraktion u.U. zu konkretisieren. Wenn solche Modelle zur Testfallgenerierung herangezogen werden, können sie ebenfalls nicht direkt als Orakel Verwendung finden: der Code würde wiederum gegen sich selbst getestet. Eine manuelle Überprüfung der Ausgaben der Testfälle ist natürlich denkbar.

Abstraktionen existieren immer nur in bezug auf einen bestimmten Zweck. Da die Extraktion von Modellen aus Code deshalb eine nur teilweise zu automatisierende Tätigkeit darstellt, ist es wegen häufiger Änderungen während der Entwicklung nicht immer einfach, die Modelle zu extrahieren (Holzmann, 2001). Wenn der Code erst nach Fertigstellung abstrahiert wird, hat das den Nachteil, daß Testfälle ebenfalls erst nach der Codierung definiert werden. Wenn die aus dem abstrahierten Code erstellten Testfälle allerdings komplementär zu bereits erstellten Testfällen verwendet werden, ist dieser Nachteil trivialerweise behoben.

Dieses Szenario ist relevant auch im Zusammenhang mit der Validierung neuartiger Testfallgenerierungsverfahren. Dabei werden Testfälle für ein exi-

[13]Beispiele sind Holzmann (2001), Graf und Saïdi (1997); Lie u. a. (2001) für FLASH-Protokollcode mit Slicing; Shen und Abraham (1999) für Verilog-Code; Chow (1978) oder Peled u. a. (1999) für den Black-Box-Test von FSMs.

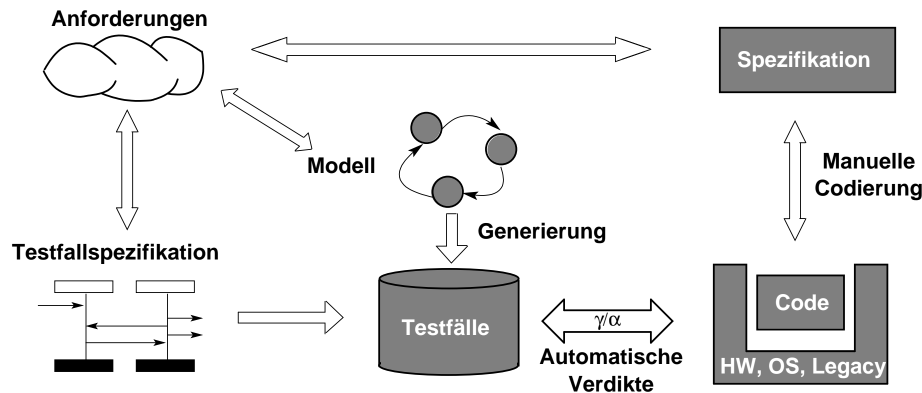


Abbildung 2.8.: Modell und Code entstehen unabhängig

stierendes System erzeugt und mit einer existierenden Testsuite verglichen.

Schließlich kann es notwendig sein, Modelle für Legacy-Komponenten, z.B. Steuergeräte in einem automobilen Bordnetz zu erstellen. Das ist dann der Fall, wenn (a) das Modell einer neuen Komponente ausführbar sein und die Umweltannahmen nicht explizit in das Modell dieser neuen Komponente aufgenommen werden sollen oder (b) Integrationstestfälle für das komponierte System, bestehend aus neuer Komponente und Legacy-Komponenten, erzeugt werden sollen. Diese Problematik wurde bereits auf S. 42 diskutiert.

2.4.3. Unabhängige oder simultane Entwicklung

Wenn Codegeneratoren für eine bestimmte Zielsprache nicht existieren, wenn Entwicklungs- und Qualitätssicherungsabteilung aus organisatorischen Gründen getrennt werden, oder wenn Codegeneratoren nicht ausreichend effizienten Code erzeugen, dann können System und Modell unabhängig voneinander und redundant erstellt werden. Abbildung 2.8 verdeutlicht das: Ausgehend von den Anforderungen, werden eine, oder im Idealfall zwei Spezifikationen entwickelt, von denen eine als Grundlage für die Erstellung des Modells dient und die andere als Grundlage für die Entwicklung von Code. Aus dem Modell werden dann Testfälle berechnet: Modelle der QS-Abteilung werden zum Test des Codes der Entwicklungsabteilung verwendet.

Da Testen von Beginn an erfolgen sollte, ergibt sich der Koordinationsaufwand zwischen verschiedenen Modell- und Codeversionen als Schwierigkeit. Eine solche redundante Entwicklung ist außerdem teuer, für den Test aber unabdingbar. Der große Vorteil dieses Szenarios liegt – bei Übereinstimmung der Schnittstellen und Abgleich der Abstraktionsniveaus – in der Möglichkeit, Modelle automatisch als Orakel zu verwenden. Wenn es Diskrepanzen zwischen Modell und Code gibt, muß geprüft werden, ob das Modell oder der Code fehlerhaft ist.

Obige Ausführungen betrachten den Entwicklungsprozeß innerhalb einer Firma. Im Automobilbereich etwa wird Software allerdings häufig von externen Zulieferern zur Verfügung gestellt. Wenn der Automobilhersteller die Kompe-

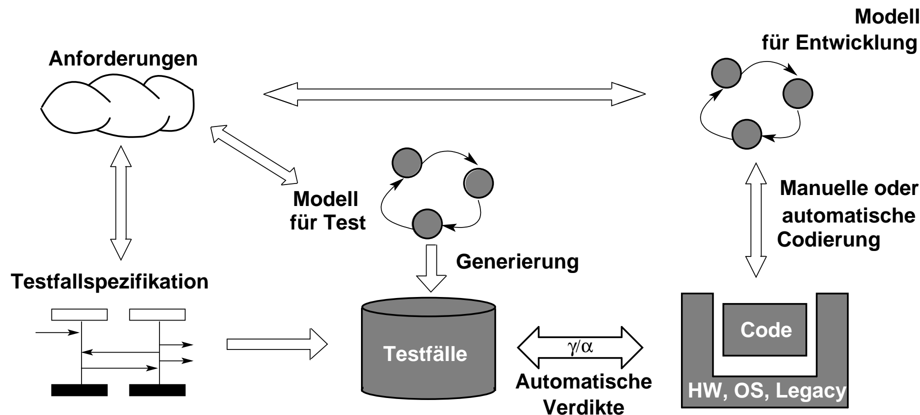


Abbildung 2.9.: Modelle für Entwicklung und Testfallgenerierung

tenz für Softwareentwicklung nicht bei sich selbst wünscht, besitzt das Argument, Code erzeugen zu wollen, keine Grundlage. Stattdessen können die Modelle als Spezifikationen betrachtet werden, die als Grundlage einer unabhängigen Codeentwicklung dienen. Bei adäquatem Abgleich der Abstraktionsniveaus können vom Automobilbauer aus dem Modell erzeugte Testfälle zum Test der vom Zulieferer erstellten Software dienen. Explizite und ausführbare Verhaltensmodelle dienen dann einerseits als Spezifikation für die Zulieferer und andererseits als Referenz, anhand derer der Automobilbauer die Übereinstimmung des vom Zulieferer gelieferten Steuergerätes mit der Spezifikation z.B. durch automatisierte Testfallgenerierung überprüfen kann. Dieses Szenario ist insbesondere auch in politischer Hinsicht von Interesse, weil der hohe Modellierungsaufwand nicht nur

- durch die Existenz einer präzisen Spezifikation, sondern auch
- durch die Möglichkeit der automatisierten Überprüfung der vom Zulieferer erstellten Subsysteme

gerechtfertigt werden kann.

Dieses Szenario macht zunächst keine Aussage darüber, ob sich die Modellentwicklung auf Modul-, d.h. Subsystem-, oder auf Systemebene bewegt. Wenn die Spezifikation eines Moduls auszuprogrammieren ist und die Spezifikation nicht ausführbar ist, kann ein Modell erstellt werden, das der Erzeugung von Testfällen dient. Im Extremfall ist es denkbar, daß der Programmierer selbst ein zur Testfallgenerierung oder -durchführung geeignetes Modell erstellt, wie das in agilen Ansätzen propagiert wird. Problematisch dabei ist, daß in diesem Fall Mißverständnisse höchstwahrscheinlich nicht nur im Modell, sondern auch im Code anzutreffen sind, was bei getrennter Entwicklung weniger wahrscheinlich ist.

Als weiteres Szenario ist denkbar, daß zwei Modelle unabhängig erstellt werden, eins für die Entwicklung und eins für die Testfallgenerierung (Abbildung 2.9). Dieses Szenario entspricht im wesentlichen dem vorherigen, mit dem

Unterschied, daß hier auch die Erstellung des Systems von den Vorteilen einer modellbasierten Entwicklung profitieren kann.

Die Chipkartenstudie stellt einen Fall dar, in dem Modell und Code unabhängig voneinander entwickelt wurden.

2.5. Andere QS-Maßnahmen

Teile der folgenden Ausführungen sind Huber u. a. (2001, Kap. 5.3) entnommen.

Unter passiver Qualitätssicherung werden diejenigen QS-Maßnahmen verstanden, die sich aus spezifischen Anforderungen an den Entwicklungsprozeß ergeben: Vorgaben, wie die Dokumentation zu erfolgen hat, die Verwendung spezifischer Werkzeuge, oder Coding Standards etwa. Unter aktiver oder konstruktiver QS werden solche Maßnahmen verstanden, die eigene Aktivitäten innerhalb des SW-Entwicklungsprozesses darstellen (beispielsweise der Vorgang der Dokumentation, Tests, oder formale Beweise der Korrektheit kritischer Systemteile).

Beispiele für passive Qualitätssicherungsmaßnahmen sind Zero-Defect-Strategien wie Poka-Yoke oder, mit stärkerem Bezug zur Informatik, top-down-Entwicklungsmethoden (Broy, 1993; Back und Wright, 1998). Die Idee bei Poka-Yoke ist es, für bekannte Probleme eine Lösung zu finden und diese in Form von Modifikationen der produzierenden Anlagen in den Entwicklungsprozeß einzubetten. Es wird also probiert, von vornherein „korrekte“ Artefakte zu produzieren. Eine ähnliche Idee des correct-by-design liegt dem top-down-Programmentwurf zugrunde. Dort wird probiert, ausgehend von einer als korrekt und valide angenommenen und auf fixen Anforderungen beruhenden Spezifikation ein System schrittweise zu verfeinern, so daß sich zwischen jeweils zwei Entwicklungsschritten eine mathematische Beziehung (Implikation, Verhaltensinklusion) feststellen läßt. Das Endprodukt ist dann eine Verfeinerung der Spezifikation, die nach Konstruktion korrekt ist.

Im folgenden wird zum Zweck der Abgrenzung kurz auf einige zum Testen komplementäre aktive und passive QS-Maßnahmen eingegangen. Aktive Maßnahmen umfassen Walkthroughs, Inspektionen, Reviews, formale Verifikation (Model Checking, Beweisen) sowie dynamische Analyseverfahren, die hier unter „Testen“ subsumiert werden. Stellvertretend für viele passive Qualitätssicherungsmaßnahmen wird auf das strukturierte Testen mit TMaP/TPI eingegangen.

Prozeßbezug Audits und Assessments sind prozeßbezogene Prüfmethoden. ISO 8402 definiert das Qualitätsaudit als eine „systematische, unabhängige Untersuchung, um festzustellen, ob die qualitätsbezogenen Tätigkeiten und die damit zusammenhängenden Ergebnisse den geplanten Anordnungen entsprechen und ob diese Anordnungen wirkungsvoll verwirklicht und geeignet sind, die Ziele zu erreichen.“ Audits werden beispielsweise routinemäßig bei der ISO 9000-Zertifizierung durch unabhängige Zertifizierungsstellen durchgeführt.

Unter „Assessment“ im QS-Kontext wird üblicherweise ein Bewertungsverfahren mit Hilfe von Fragebögen verstanden, die den SW-Entwicklungsprozeß

im Rahmen des Capability Maturity Models (CMM) betreffen.

Schließlich gibt es Bemühungen, den gesamten Testprozeß zu strukturieren. Das wird beispielsweise durch „Test Management Process“, TMaP (Pol und van Veenendaal, 1998), angestrebt. Ein Verfahren zur Optimierung des Testprozesses ist das „Test Process Improvement“, TPI (Koomen und Pol, 1999).

TMaP strukturiert den Testprozeß zunächst in vier Basiskonzepte: Phasenmodell, Techniken, Infrastruktur/Werkzeuge und Organisation. Diese werden dann in verschiedene Kernbereiche untergliedert. Im Rahmen von TPI werden u.a.

- dem *Phasenmodell* Teststrategien und Lebenszyklusmodell,
- den *Techniken* Testplanung, Techniken zur Testspezifikation, statische Testverfahren (Inspektionen) und Metriken,
- der *Infrastruktur* Testwerkzeuge, die Testumgebung und Fragen wie die der Büroausstattung und
- der *Organisation* Fragen der Motivation und der Mitarbeiterschulung

zugeordnet und definiert. Die Kernbereiche werden dann zunächst qualitativ in verschiedene Ebenen der Verwirklichung unterteilt, die den aktuellen und den erwünschten Zustand des Testprozesses beschreiben. Die qualitativen Ausprägungen werden dann weiter quantitativ zergliedert. Eine resultierende sog. Entwicklungsmatrix dient dann der Analyse des aktuellen Prozesses sowie der Erkennung und Nutzung von Optimierungspotentialen.

Produktbezug Die Abgrenzung von Walkthroughs, Inspektionen und Reviews in der Literatur ist nicht einheitlich. Ihr Ziel ist die „semantische“ Überprüfung eines (Teil-) Systems oder Dokuments. Gemeinsamkeiten sind (Balzert, 1998):

- Produkte und Teilprodukte werden manuell analysiert, geprüft und begutachtet.
- Ziel ist es, Fehler, Defekte, Inkonsistenzen und Unvollständigkeiten zu finden.
- Die Überprüfung erfolgt in einer Gruppensitzung durch ein kleines Team mit definierten Rollen.

Voraussetzungen für den Einsatz dieser Methoden sind u.a. genau dokumentierte Planung der benötigten Ressourcen, nachgewiesene Qualifikation der Teilnehmer, die Abwesenheit von Vorgesetzten sowie ein Einverständnis darüber, daß die Prüfungen hohe Priorität besitzen.

Als Vorteile gelten u.a. die Eigenschaft dieser Verfahren,

- oft die einzig mögliche Vorgehensweise zur semantischen Überprüfung darzustellen,
- die Verantwortung auf die gesamte Gruppe zu übertragen,

- sowie von hoher Effektivität zu sein. Reviews stellen beispielsweise die zentrale Technik der Qualitätssicherung im Cleanroom Software Engineering (Prowell u. a., 1999) dar.

Reviews, Inspektionen und Walkthroughs werden manchmal bzgl. ihrer Formalisierung unterschieden. Inspektionen gelten dann als am stärksten und Walkthroughs am wenigsten formalisiert; Reviews befinden sich in dieser Bewertung in der Mitte.

Gemäß ANSI/IEEE 729-1983 ist die Inspektion „a formal evaluation technique in which SW requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems“. Es handelt sich also um eine „manuelle, formalisierte Prüfmethode, um schwere Defekte in schriftlichen Dokumenten anhand von Referenzunterlagen zu identifizieren und durch den Autor beheben zu lassen“ (Balzert, 1998). Empirische Untersuchungen weisen darauf hin, daß mit Inspektionen bis zu 75% aller Entwurfsfehler gefunden werden können. Explizite Verhaltensmodelle bieten sich an, die Rolle der Referenzunterlagen zu übernehmen; wenn die Modelle selbst einer Inspektion unterworfen werden, müssen andere Referenzen verwendet werden. Es ist beispielsweise denkbar, die Inspektionen zusammen mit dem Auftraggeber durchzuführen.

Ein Review ist eine „manuelle, semiformale Prüfmethode, um Stärken und Schwächen eines schriftlichen Dokuments anhand von Referenzunterlagen zu identifizieren und durch den Autor beheben zu lassen“ (Balzert, 1998). Wie bereits erwähnt, wird die nicht eindeutige Unterscheidung zur Inspektion üblicherweise im Grad der Formalisierung gesehen.

Ein (structured) walkthrough ist gemäß ANSI/IEEE 729-1983 „a review process in which a designer or programmer leads one or more other members of the development through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, possible errors, violation of development standards, and other problems“. Ein Kernpunkt ist die Anwesenheit des Programmierers oder Designers. Eine ausführliche Diskussion der Unterschiede zwischen diesen drei manuellen Prüfmethoden findet sich bei Balzert (1998).

Teilautomatisierte Techniken der QS Diese klassischen, erfolgreichen manuellen Prüfmethoden werden üblicherweise durch andere aktive Qualitätssicherungsmaßnahmen, insb. Testen, komplementiert. Mögliche Techniken sind die Fault Injection (s. S. 3.4), bei der absichtlich Fehler in die SW (oder HW) eingebracht werden, um die Reaktion des Systems unter diesen Umständen zu testen. Sinn dabei ist üblicherweise, defekte HW zu simulieren und somit Aussagen über die Robustheit des Systems treffen zu können.

Eine weitere Technik ist das statistische Testen, bei dem Aussagen über die Wahrscheinlichkeit des Auftretens bestimmter Eingaben zu bestimmten Zeitpunkten getroffen werden. Grundlage des statistischen Testens sind somit sog. statistische Nutzungs- oder Umgebungsmodelle, die im Zusammenhang mit dem System entwickelt werden müssen. Wesentlich ist die Annahme einer bestimmten Verteilung der Eingaben (Duran und Ntafos, 1984; Hamlet und Taylor,

1990; Thévenod-Fosse u. a., 1995). Ein allgemein anerkannter Kritikpunkt ist die Schwierigkeit, plausible und realitätsnahe Aussagen über die Wahrscheinlichkeit des Auftretens bestimmter Eingaben (oder Eingabefolgen) zu treffen. Im Cleanroom Software Engineering (Prowell u. a., 1999) dient das Testen der Messung der Prozeßqualität und nicht der Detektion von Fehlern. Letztere sollen in diesem Vorgehensmodell durch Reviews entdeckt werden.

Formale Analysetechniken Formale Techniken können für kleinere Systeme teil- oder vollautomatisch dabei helfen, Aussagen mit mathematischer Beweiskraft über ein Modell zu treffen. Der Unterschied zwischen Modell und Implementierung in diesem Zusammenhang wurde in der Einleitung diskutiert. Unter formale Techniken fallen Beweisen und Model Checking:

- Beweisen als Validierungstechnik umfaßt üblicherweise maschinengestützte Aussagen über oder Generierung von Invarianten, Aussagen über Zusammenhänge von Vor- und Nachbedingungen oder, als synthetische Technik, in top-down Entwicklungsprozessen den Nachweis einer Verfeinerungs- bzw. Abstraktionsbeziehung (Broy, 1993; Dijkstra, 1976; Morgan, 1990; Back und Wright, 1998; Roever und Engelhardt, 1998; Cousot und Cousot, 1977) zwischen zwei Systemen.

- Model Checking (Clarke u. a., 1999) bezeichnet die exhaustive Zustandsraumexploration zum Zweck der Überprüfung einer Eigenschaft. Der Zustandsraum kann symbolisch – durch Constraints oder BDDs – oder explizit – in Form von beispielsweise Hashtabellen – abgespeichert werden. Im ersten Fall spricht man von symbolischem oder globalem, im zweiten von explicit-state oder lokalem Model Checking.

Beide Ansätze sind wegen der zu hohen Komplexität in der Praxis für größere Systeme noch nicht anwendbar.

2.6. Zusammenfassung

Zweck dieses Kapitels ist zunächst eine Klärung der für diese Arbeit wesentlichen Termini:

- Testen bezeichnet die Gesamtheit der Aktivitäten Testplanung, Testorganisation, Testdokumentation, Testfallgenerierung, Testdurchführung, Monitoring und Testauswertung.

- Ein Testfall ist die Repräsentation einer endlichen Menge von Eingabe- und erwarteten Ausgabedaten. Bei zu testenden deterministischen oder deterministisch kontrollierbaren Systemen ist ein Testfall ein Modelltrace. Dieser kann einer Menge von Maschinentraces entsprechen. Bei nichtdeterministischen Systemen ist ein Testfall ein Transitionssystem, das aus mehr als einem Pfad besteht. Testfälle werden in Testsuiten zusammengefaßt. Ein Testfallgenerator erzeugt Testfälle aus einem Modell des zu testenden Systems und einer Formalisierung des Testziels, d.h. der zu testenden Eigenschaft.

- Ein Testtreiber ist ein ausführbares Programm, das Testfälle (a) so transformiert, daß sie auf die zu testende Maschine angewendet werden können, (b) den Eingabeteil dieser transformierten Testfälle auch tatsächlich auf das System anwendet und (c) Ausgaben der Maschine u.U. auf das Niveau des Modells zum Zweck eines Vergleichs abstrahiert. Testtreiber gleichen also insbesondere die Abstraktionsniveaus zwischen Modell und Maschine ab. Ein Monitor vergleicht dann Soll- und Istverhalten und liefert eine Beurteilung, ein Verdikt.

Im Rahmen dieser Arbeit erfolgt Testen anhand von expliziten ausführbaren Verhaltensmodellen des zu testenden Systems, gegeben in Form von erweiterten Zustandsmaschinen in AutoFocus. Modellierung ist eine anspruchsvolle Tätigkeit, und ohne adäquate Modellierungssprachen ist eine Durchsetzung der präsentierten Ideen unwahrscheinlich. Als Anforderungen an eine Modellierungssprache wurden skizziert:

- Unterstützung der integrierbaren Modellierung verschiedener Aspekte (u.a. Funktionalität, Struktur, Kommunikation, Zeitverhalten, Deployment), die ihrerseits in verschiedenen Detaillierungsgraden vorliegen können. Das ist für die Beherrschung des Entwicklungsprozesses bei stetig wachsender Systemkomplexität notwendig. Das höhere Abstraktionsniveau von Modellen im Vergleich zu Code erleichtert die intellektuelle Beherrschung. Dazu wird Ausführbarkeit als wünschenswert angesehen. Wenn sie machbar ist, ist die Separation of concerns nicht nur für die Spezifikation, sondern auch für eine u.U. kompositionale Testfallgenerierung von Vorteil.

- Domänenabhängigkeit, d.h. die Existenz sprachlicher Konstrukte für die eine Domäne auszeichnenden fundamentalen Entitäten und ihre Zusammenhänge. Domänenabhängigkeit bedeutet Einschränkung und als Resultat leichtere Analysierbarkeit.

- Einfachheit der zugrundeliegenden Semantik zum Zweck der intellektuellen Beherrschbarkeit.

AutoFocus ist domänenspezifisch in bezug auf zeitsynchrone Kommunikation und vergleichsweise einfache Datentypen. Explizite Konstrukte, die eine Domäne auszeichnen, z.B. die von Chipkarten, existieren nicht. Beispiele für solche Konstrukte wären die Behandlung von Zufallszahlen oder Stereotypen für Permutationen auf Byteebene.

Modelle finden Verwendung als

- Modelle des Problems,

- Modelle der Maschinenumgebung,
  - die eine Charakterisierung einer Menge von Szenarien darstellen, aus denen einzelne Testfälle abgeleitet werden oder
  - um Ausführbarkeit zu gewährleisten oder im Fall der Testfallgenerierung den Zustandsraum einzuschränken,

- Modelle der Maschine
  - zum Zweck der Bestimmung von Maschineneigenschaften,
  - für die Testfallgenerierung,
  - zur automatisierten Codeerzeugung
  - oder als Spezifikation.

Wenn Modelle zum Testen eingesetzt werden, sind sie redundante Beschreibungen der Maschine. Da aus ihnen berechnete Testfälle nicht nur zur Validierung des Modells sondern auch zur Verifikation des Systems verwendet werden sollen, stellt sich

- die prinzipielle Frage, wie Modelle gleichzeitig abstrakt – zur Testfallgenerierung geeignet – und konkret – zum Test der Maschine geeignet – sein können und wie die verschiedenen Abstraktionsebenen miteinander in Bezug gesetzt werden können (etwa durch Architekturen und klar definierte Schnittstellen) und

- aus ökonomischer Sicht die Frage, ob diese Modelle dann nicht auch zur Generierung von Produktionscode verwendet werden können. Im allgemeinen wird es schwierig sein, Codegeneratoren für eine bestimmte Modellierungssprache zu erstellen, wenn der Code starken Ressourcenbeschränkungen unterliegt. Das ist bei Chipkarten der Fall. Adapter, die die aus dem Modell erzeugten Testfälle auf die Chipkarte anwendbar machen, sind hingegen vergleichsweise einfach zu erstellen.

Wenn Modelle als Grundlage für Testfälle und Produktionscode dienen sollen, dann muß sichergestellt werden, daß die für den automatischen Test erforderliche Redundanz gewährleistet ist. Verschiedene Szenarien der Erstellung von Code und Modell wurden diskutiert:

- Modelle werden nach der Implementierung auf Grundlage letzterer erstellt,

- Modelle werden vor der Implementierung zum Zweck der Codeerzeugung erstellt, oder

- Modelle werden unabhängig vom Code für die Testfallerzeugung oder als virtuelle Umgebungskomponente erstellt.

Die Suche nach geeigneten Modellierungsformalismen und Beschreibungstechniken stellt eines der Kernprobleme des modernen Software-Engineering dar. Das betrifft alle Phasen des Entwicklungsprozesses, also nicht nur das Testen. Alle oben vorgestellte Konzepte in modellbasierten Entwicklungsprozessen (Modellierungsformalismen, Semantiken, Beschreibungstechniken) bedürfen starker fortgesetzter Anstrengungen, um die prinzipielle Idee – die aufgrund des gegenwärtigen Mangels an evaluierbarer Werkzeugunterstützung angreifbar ist – praxistauglich umzusetzen. Eine Alternative zur Beherrschung stetig wachsender, komplexerer Systeme ist nicht offensichtlich.

3. Testfallspezifikation

Das zentrale Problem des Testens ist die Festlegung relevanter zu testender Eigenschaften. Das gilt auch für andere Verifikationstechniken wie Model Checking oder deduktivem Beweisen. „Interessante“ Eigenschaften müssen während der üblicherweise phasenübergreifenden Aktivität des Requirements Engineering festgelegt werden und sind im Normalfall applikationsspezifisch. Thema dieses Kapitels ist die Problematik, welche Klassen von Eigenschaften es gibt, wie sie notiert und zur Testfallgenerierung verwendet werden können.

Der Stand der Kunst erlaubt zur Zeit nicht die Angabe allgemeingültiger oder domänenspezifischer Heuristiken zur Ableitung der relevanten Eigenschaften.[1] Fehlerklassifikationen, wie sie beispielsweise von Lutz (1993) propagiert werden, scheinen ein sinnvoller Schritt in die Richtung eines systematischen Qualitätssicherungsprozesses zu sein. Nicht-domänenspezifische Fehler, etwa Laufzeitfehler, wurden bereits in Kap. 2 diskutiert.

Eng verbunden mit der Charakterisierung „interessanter“ bzw. „relevanter“ Eigenschaften ist die Bemessung der Qualität einer Testsuite (der Überblicksartikel von Zhu u. a. (1997) diskutiert die dreifache Natur als Qualitätsmaß, Testendekriterium und Testfallspezifikation). Da auch eine solche Bewertung der Qualität gegenwärtig allgemein nicht möglich ist, wird im Rahmen dieser Arbeit davon ausgegangen, daß Formalisierungen der Eigenschaften, d.h. Testfallspezifikationen, vom Testingenieur bzw. Spezifikateur geliefert werden: Testfälle – von einem Treiber auf eine Maschine anwendbare – Traces sind so gut, wie die Testfallspezifikationen – die Spezifikation der zu testenden Eigenschaft, aus der ein Testfallgenerator Testfälle berechnet – sind, und letztere werden als gegeben vorausgesetzt. Auf Arbeiten, die die Güte von Testfällen anhand beispielsweise axiomatischer Charakterisierungen beschreiben, wie das Weyuker (1986) und formalisiert Parrish und Zweben (1991) propagieren, wird wegen ihrer mangelnden Operationalisierbarkeit nicht weiter eingegangen.

Gliederung Dieses Kapitel ist wie folgt gegliedert. Abschnitt 3.1 geht auf die Unterscheidung universeller und existentieller Eigenschaften ein. Abschnitt 3.2 diskutiert funktionale Testfallspezifikationen auf der Basis gegebener Szenarien. Es wird exemplarisch demonstriert, wie Invarianzeigenschaften schematisch durch existentielle Eigenschaften approximiert werden können. Das kann durch Übersetzung einer CTL-Formel in einen Automaten geschehen, für den dann Abdeckungskriterien definiert werden, oder durch Transformation der entsprechenden Formeln. Abschnitt 3.3 präsentiert Überdeckungskriterien als Grundla-

[1] Der Versuch einer Klassifizierung von Eigenschaften auf der Basis syntaktischer Formelelemente für die Bildung von Spezifikationsmustern findet sich bei Dwyer u. a. (1998).

ge der Testfallgenerierung. Es wird gezeigt, unter welchen Umständen mit oder ohne explizite Annotationen des Modells eine Abdeckung von *Anforderungen* erzielt werden kann. Abschnitt 3.4 diskutiert stochastische und randomisierte Testfallspezifikationen. In Abschnitt 3.5 schließlich wird das Problem einer adäquaten Sprache für Testfallspezifikationen untersucht. Verwandte Arbeiten werden in den entsprechenden Abschnitten diskutiert.

Fokus dieser Arbeit ist nicht die Definition sinnvoller Testfallspezifikationen. Das ist der Grund für den überblicksartigen Charakter dieses Kapitels. Der im folgenden beschriebene Testfallgenerator ist in der Lage, für alle als relevant identifizierten Klassen von Testfallspezifikationen Testfälle zu erzeugen. Auf die Angabe einer konkreten Syntax für Testfallspezifikationen wird verzichtet.

3.1. Eigenschaften und Testfallspezifikationen

Der Test von Maschinen impliziert einen wesentlichen Unterschied zur Überprüfung von Modellen. Die betrachteten Systeme sind reaktive Systeme, die sich u.a. durch potentiell unendliche Abläufe auszeichnen. Auf endlicher Modellebene können nun in endlicher Zeit Aussagen über unendliche Abläufe abgeleitet werden, denn unendliche Abläufe lassen sich stets durch Schleifen im Zustandsraum charakterisieren, die dann mit endlichen Repräsentationen behandelt werden können. So können Sicherheitseigenschaften der Art „niemals kann eine digitale Signatur berechnet werden, ohne daß zuvor eine korrekte PIN eingegeben wurde“ auf Modellebene nachgewiesen werden. Wenn mit Model Checking eine Invarianzeigenschaft verifiziert wird, ist das für den Test der Maschine nicht direkt hilfreich, weil kein einzelner Zeuge existiert, sondern eine Menge von Zeugen, nämlich alle Traces des Modells. Das gilt verallgemeinert für alle pfaduniversellen Eigenschaften.

Universelle Eigenschaften Reaktive Systeme können vollständig anhand zweier Klassen von Eigenschaften spezifiziert werden, den Sicherheits- und den Lebendigkeitseigenschaften (Safety und Liveness (Alpern und Schneider, 1985), vgl. Chang u. a. (1991) und Broy und Stølen (2001)).

- Informell sind Sicherheitseigenschaften Eigenschaften, deren Verletzung immer in endlicher Zeit beobachtet werden kann. Ein typisches Beispiel sind Invarianzeigenschaften, hier verstanden als Zustandsformeln, die in jedem erreichbaren Zustand des Systems gelten ($AG\varphi$ für eine Zustandsformel φ).

- Die Verletzung von Lebendigkeitseigenschaften hingegen kann immer nur bei Beobachtung unendlicher Abläufe beobachtet werden. Ein klassisches Beispiel sind Responseeigenschaften der Art $AG(\varphi \Rightarrow AF\psi)$ für Zustandsformeln φ, ψ.

Die Verifikation solcher Eigenschaften erfordert immer die Beobachtung unendlicher Abläufe. Zeugen für diese Eigenschaften oder Kombinationen von ihnen sind möglicherweise unendliche Mengen unendlich langer Traces.

Die Menge der Zeugen muß aber nicht von unendlicher Kardinalität sein, wie das Beispiel der Eigenschaft $EG\varphi$ demonstriert: Sie besitzt als Zeugen möglicherweise eine endliche – aus einem Trace bestehende – Menge mit einem unendlichen Trace. Der Nachweis der Eigenschaft ist durch einen solchen Trace erbracht.

Eigenschaften, die das Kriterium erfüllen, daß für ihren Nachweis das Betrachten möglicherweise unendlich vieler Traces unendlicher Länge notwendig ist, werden im folgenden als *universelle Eigenschaften* bezeichnet.

Das Problem beim Test von Maschinen ist nun, daß Testsuiten endliche Mengen endlicher Traces sind. Andernfalls wäre der Testprozeß von unendlicher Dauer. Lebendigkeitseigenschaften der Maschine sind demnach weder verifizier-, noch falsifizierbar. Sicherheitseigenschaften sind in der Maschine endlich falsifizierbar. Die Schwierigkeit des modellbasierten Ansatzes ist, daß das Modell als valide angesehen wird, d.h. jede zu testende Sicherheitseigenschaft des Modells gilt im Modell. Für Invarianzeigenschaften beispielsweise ist jeder der unendlich vielen Modelltraces ein möglicher Kandidat für einen Testfall. Ein Testfallgenerator steht vor dem Problem, aus dieser unendlichen Menge „interessante" Repräsentanten auszuwählen. Dieser Umstand war auch Hauptgrund für die Unterscheidung von Testzielen und Testfallspezifikationen in den Abschnitten 2.2.3 und 2.2.5. Die Selektion kann u.a. zufällig geschehen („zufällig 1% aller Traces der Länge 5 auswählen"; Abschnitt 3.4).

Existentielle Eigenschaften Wegen der prinzipiellen Probleme beim Test universeller Eigenschaften findet im Rahmen dieser Arbeit eine Fokussierung auf existentielle Eigenschaften statt. Im Unterschied zu universellen Eigenschaften seien existentielle Eigenschaften als Eigenschaften definiert, deren Nachweis mit einer endlichen Menge endlicher Traces erfolgen kann. Das bedeutet nicht, daß die Zeugenmenge dieser Eigenschaften endlich sein muß.

Beispiel 10 (Existentielle Eigenschaft). *Im Fall der Chipkarte gibt es wegen der Existenz eines Reset-Kommandos unendliche viele Traces, die die Formel $EF init$ erfüllen, wenn init den Initialzustand beschreibt. Ein einzelner Trace $< CardReset, ATR >$ ist aber ausreichend, die Gültigkeit der Eigenschaft nachzuweisen.*

Auch aus dieser Menge müssen dann „interessante" Repräsentanten ausgewählt werden. Im Unterschied zu universellen Eigenschaften kann aber damit die partielle Übereinstimmung des Maschinen- mit dem Modellverhalten nachgewiesen werden. Zum Testen werden deshalb zunächst solche Formeln betrachtet, deren Nachweis mit einem Trace endlicher Länge erbracht werden kann, $EF\varphi$-Eigenschaften beispielsweise.

Szenarien (Abschnitt 3.2.1) und Überdeckungskriterien (Abschnitt 3.3) können als Kombinationen solcher existentieller Eigenschaften aufgefaßt werden. Wenn Szenarien in Form von beispielsweise Sequenzdiagrammen insofern unvollständig sind, als (a) Komponentenachsen weggelassen wurden oder (b) eine lose Semantik zugrundeliegt, die das Auftreten von nicht-spezifizierten Signalen zwischen zwei Pfeilen gestatten, dann besteht die Aufgabe der Testfallgenerierung darin, einen vollständigen Trace jeweils zwischen zwei Signalen zu finden.

Ein weiteres Anwendungsfeld existentieller Eigenschaften besteht in der Approximation universeller Eigenschaften durch existentielle Eigenschaften, die auf den Seiten 76 ff. diskutiert wird.

Klassifikation Testfallspezifikationen – Formalisierungen und Operationalisierungen von Testzielen – lassen sich neben ad-hoc angegebenen grob in

- funktionale,
- strukturelle und
- stochastische

einteilen, aus denen ohne weitere Modifikationen mit dem in den folgenden Kapiteln beschriebenen Testfallgenerator Testfälle berechnet werden können. Funktionale Testfallspezifikationen basieren auf relevanten Anwendungsszenarien, die im Rahmen des Requirements Engineering oder der Qualitätssicherung identifiziert wurden. Strukturelle Spezifikationen basieren auf der syntaktischen Struktur eines Modells oder Programms. Beispiele sind die Abdeckung von Anweisungen oder von Transitionen. Stochastische Spezifikationen schließlich nutzen ein Modell, um zufällig (randomisiert gleichverteilt) oder anhand von Nutzungsprofilen Testfälle zu erzeugen. Auf strukturellen und randomisierten Kriterien basierende Testfallspezifikationen können im Unterschied zu funktionalen automatisch erzeugt werden.

Suchraumeinschränkung Testfallspezifikationen dienen auch der Einschränkung des Suchraums. Mengen von Szenarien in Form einer nichtdeterministischen endlichen erweiterten Zustandsmaschine (EFSM; eine FSM mit lokalen Variablen) können getestet werden, indem beispielsweise Kontrollzustandsüberdeckung auf dieser EFSM gefordert wird. Die Transitionen zwischen den Zuständen schränken dann die Menge aller Systemabläufe ein.

3.2. Funktionale Testfallspezifikationen

In diesem Abschnitt werden funktionale Testfallspezifikationen, d.h. Mengen explizit angegebener Szenarien, diskutiert. Dabei findet eine Konzentration auf modellbezogene Testfallspezifikationen statt, d.h. Testziele wie beispielsweise Bytepermutationen auf APDU-Ebene im Fall der Chipkarte werden nicht betrachtet. Für solche Tests könnten auch Modelle verwendet werden. Ihre Codierung in der Testtreiberumgebung erscheint allerdings vielversprechender: Auf Modellebene kommen APDUs gar nicht vor.

3.2.1. Test von Szenarien

Unter funktionalen Testzielen werden explizit in Anforderungsdokumenten spezifizierte Szenarien verstanden.[2] Diese werden dann in Testfallspezifikationen

[2] Im Extreme Programming und dem Test-Driven Development (Beck, 2003) übernehmen Testfälle die Rolle der (unvollständigen) Spezifikation (Pretschner u. a., 2003a).

konvertiert. Als Beispiel wird die auf asymmetrischer Kryptographie beruhende Berechnung einer digitalen Signatur betrachtet. Digitale Signaturen dienen der Sicherstellung der Authentizität einer Nachricht, dem Nachweis also, daß die Nachricht nicht verändert worden ist. Dazu stellt man der Nachricht einen sog. message authentication code (MAC) nach und sendet sowohl die eigentliche Nachricht als auch den MAC über einen potentiell unsicheren Kanal zum Empfänger. Der berechnet ebenfalls einen MAC und überprüft beide – sind sie identisch, so wurde die Nachricht nicht verändert.

Asymmetrische Kryptographie beruht auf der Existenz von Paaren von öffentlichen (K_{pub}) und privaten (K_{pr}) Schlüsseln für jeden Kommunikationspartner. Diese werden in zwei Funktionen zur Ver- und Entschlüsselung verwendet, *enc* und *dec*. Für eine Nachricht m ist die wesentliche Eigenschaft

$$m = dec(K_{pr}, enc(K_{pub}, m)) = enc(K_{pub}, dec(K_{pr}, m)).$$

Grob wird bei auf RSA basierender Signaturberechnung aus einem Dokument m zuerst ein Hash-Wert berechnet und, um Platz zu sparen, nur dieser Hash signiert. Dazu wird der Hash $H(m)$ zunächst mit einem privaten Schlüssel der Karte entschlüsselt; die Signatur ist dann $\Sigma = dec(K_{pr}, H(m))$. Die Konkatenation $m \circ \Sigma$ wird dann an den Empfänger gesendet, der $m' \circ \Sigma'$ erhält. Er trennt beide Teile und berechnet ebenfalls den Hash $H(m')$ von m'. Dann wird $H' = enc(K_{pub}, \Sigma')$ mit dem öffentlichen Schlüssel berechnet. Ist $H' = H(m')$, so ist m' authentisch, d.h. $m = m'$.

Demzufolge enthält die Spezifikation der WIM ein Szenario der erfolgreichen Berechnung einer digitalen Signatur:

Beispiel 11 (Funktionales Testziel). *Zur Berechnung einer digitalen Signatur muß (1) die richtige Sicherheitsumgebung gesetzt, (2) die PIN verifiziert, (3) der private Schlüssel gesetzt und (4) die digitale Signatur berechnet werden. Die Aneinanderreihung dieser vier Kommandos zusammen mit erwarteten Ausgaben stellt direkt ein Testziel dar.*

Vorher muß noch die Quelle eines zu verschlüsselnden Bytestrings angegeben werden (im Modell werden symbolische Namen verwendet, die vom Testtreiber beliebig instantiiert werden können). Die Eingabe der PIN dient der Identifizierung des Benutzers. Das Setzen des privaten Schlüssels entspricht im wesentlichen der Selektion derjenigen – natürlich nicht öffentlich lesbaren – Datei auf der Chipkarte, die den Schlüssel enthält.

Wenn die verschiedenen Operationen des Testziels in für das Kartenmodell verständliche Kommandos übersetzt werden und somit eine Testfallspezifikation erzeugt wird, können diese in das Modell gespeist und die entsprechenden Ausgaben protokolliert werden. Auf Testtreiberebene werden dann die fehlenden Werte – tatsächliche Schlüssel und zu signierendes Dokument – eingesetzt und an die tatsächliche Karte gesendet. Nach Abstraktion der Kartenantworten erfolgt die Verdiktbildung.

Es kann davon ausgegangen werden, daß beim Test dieser Sequenz keine Fehler auftreten. Der Grund ist der, daß die Information explizit in den Spezifikationsdokumenten vorhanden ist und somit insbesondere auch den Entwicklern

zur Verfügung steht. Fehler scheinen aber verstärkt dort aufzutreten, wo Anforderungen nicht explizit aufgeführt oder wo Anforderungen für Teilsysteme klar sind, aber nicht für integrierte Systeme (vgl. die klassischen Beispiele der Feature Interaction). Allgemein stellt sich die Frage, welche Modifikationen eines explizit spezifizierten Szenarios legal und welche illegal sind. Szenarien werden häufig als Sequenzdiagramme angegeben, deren Vorteile – ihr exemplarischer Charakter – zugleich ihr Nachteil – Unvollständigkeit – ist.

Beispiel 12 (Modifikation von Szenarien). *Für den Test der digitalen Signaturberechnung stellt sich die Frage,*

- *wie sich das Einstreuen beliebiger Kommandos zwischen den vier relevanten Phasen auswirkt und*
- *welche Permutationen der Phasen ebenfalls zu einer erfolgreichen Berechnung führen.*

Es ist z.B. irrelevant, ob zuerst die Sicherheitsumgebung ausgewählt oder zuerst die PIN verifiziert wird. Der Schlüssel kann hingegen erst nach Auswahl der Sicherheitsumgebung gesetzt werden, weil er das entsprechende Template der Umgebung manipuliert. Die legalen und illegalen Permutationen müssen explizit vom Testingenieur angegeben werden, um getestet werden zu können. Erneut gilt, daß Testfälle nur so gut wie die entsprechenden Testfallspezifikationen sein können.

„Vervollständigung“ von Szenarien Neben der Permutation der relevanten Phasen eines Szenarios (eines Sequenzdiagramms) gibt es im wesentlichen drei Möglichkeiten, zwischen zwei Signalen bzw. Pfeilen weitere Signale einzufügen.

- Das eingefügte Signal ändert den Zustand des Szenarios nicht, kann also ignoriert werden. Da zwischen Modellen und Maschinen unterschieden wird und das Modell das Soll-Verhalten codiert, ist es präziser, zu fordern, das eingestreute Signal *solle* den Zustand nicht ändern – beim Test der Maschine muß das überprüft werden. Ein Beispiel für die Signaturberechnung ist die Erzeugung einer Zufallszahl, die den Status nicht ändern sollte.
- Ein zwischen zwei Phasen eingestreutes Signal ändert den Status des Protokolls mit Sicherheit. Das ist fast immer für Reset-artige Signale der Fall. Im Fall der Chipkarte ist ein weiteres Beispiel das Setzen einer anderen Sicherheitsumgebung als die für die Signaturberechnung benötigte, weil bei Änderung der Sicherheitsumgebung u.U. bereits gesetzte Schlüssel gelöscht werden.
- Das zwischen zwei Phasen eingestreute Kommando ändert den Zustand potentiell. Das heißt, daß ein Zustandswechsel im Szenario vom Gesamtzustand des Systems und der Systemreaktion abhängt. Der Grund für das Auftreten dieses Falles liegt darin, daß Sequenzdiagramme unvollständig sind und daß – zumeist bewußt – nicht der gesamte Systemzustand

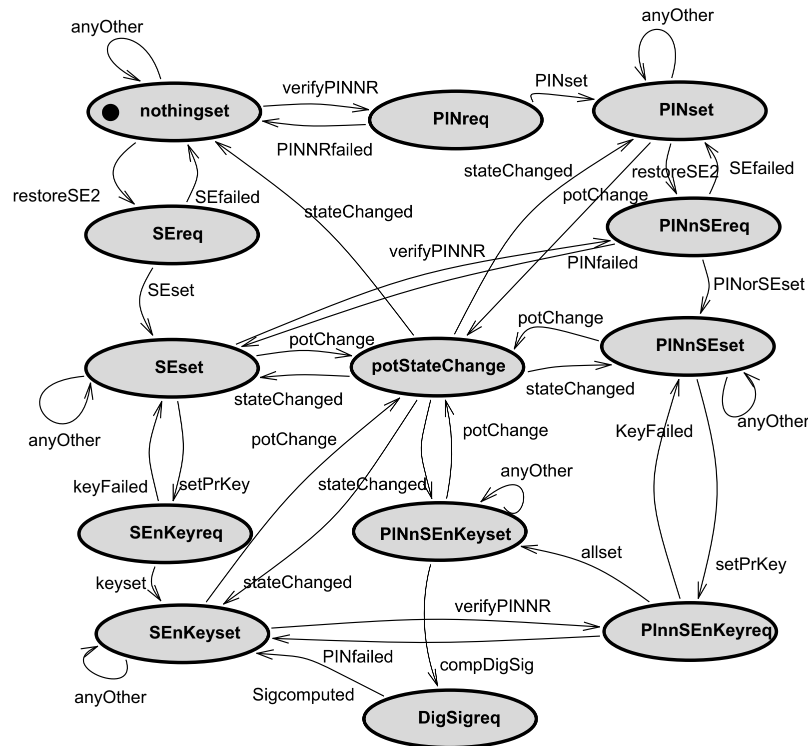


Abbildung 3.1.: Digitale Signaturberechnung

berücksichtigt wird. Im Fall der Chipkarte sind Beispiele das Wechseln eines Kanals (das bestätigt werden kann oder nicht) oder das Setzen einer Sicherheitsumgebung, das von der Karte zurückgewiesen wird.

Mengen von Szenarien können als Zustandsmaschinen codiert werden. Ein Beispiel ist in Abb. 3.1 angegeben, das die Kontrollzustände einer nichtdeterministischen EFSM zum Test des Protokolls der digitalen Signatur zeigt. Das Modell ist ein Testmodell (Abschnitt 2.3.1), das gemäß Abb. 3.2 mit der Karte zusammenschaltet wird. Ausgaben des Testmodells sind also Eingaben an die Karte. Alle Phasen sind in zwei Unterphasen untergliedert, eine Anforderung an und die Bestätigung durch das Kartenmodell. Die `anyOther`-Transitionen sind diejenigen Transitionen, die den Zustand nicht verändern (sollten), also z.B. die Abfrage einer Zufallszahl oder die erneute Selektion der richtigen Sicherheitsumgebung direkt nachdem sie gesetzt wurde. Für die Implementierung wurden Listen solcher Kommandos definiert, aus denen während der Testfallgenerierung ein oder alle Elemente ausgewählt werden. Der Zustand in der Mitte, `potStateChange`, wird betreten, wenn ein Kommando ausgeführt wird, das den Zustand potentiell verändert. Der Datenzustand der Komponente merkt sich, in welchem Zustand das System zuletzt war, und springt dorthin zurück, wenn das Kommando den Zustand letztlich nicht geändert hat bzw. in einen anderen Zustand, wenn der Zustand geändert wurde. Diese Informationen sind in Tabellen abgelegt, die in funktionalen Programmen codiert wurden.

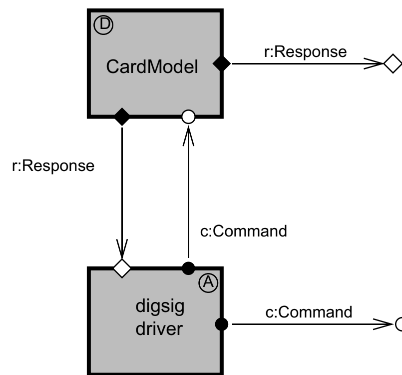


Abbildung 3.2.: Karte und Testmodell

Das Testmodell für Modifikationen des Szenarios der Signaturberechnung ist noch keine Testfallspezifikation. Vielmehr codiert es all die Szenarien, die für die Berechnung der digitalen Signatur relevant sind. In Abhängigkeit davon, welche Kommandos in den `anyOther`-Transitionen oder in denjenigen Transitionen codiert werden, die potentielle oder sicher vorhersagbare Zustandswechsel hervorrufen, ergibt sich eine Einschränkung des Zustandsraums. So ist es unter Berücksichtigung der Anforderung, möglichst kleine Testsuiten zu generieren, u.U. nicht sinnvoll, in jedem Zustand den Einfluß der Erzeugung einer Zufallszahl zu überprüfen.

Testfallspezifikationen ergeben sich in Zusammenhang mit dem Testmodell, wenn zusätzlich Aussagen über die zu generierenden Traces getroffen werden. Alle Traces zu erzeugen ist wegen der Schleifen unmöglich. Mögliche Einschränkungen sind

- alle Traces einer bestimmten endlichen Länge,
- eine zufällig ausgewählte Untermenge davon, was für große Längen sinnvoll ist (Abschnitt 3.4), oder
- Überdeckungsmaße (Abschnitt 3.3) auf dem Testmodell, z.B. die Forderung, alle Zustände oder alle Transitionen mindestens einmal zu besuchen.

Der letzte Fall ist insofern von Interesse, als er in existentiellen Spezifikationen wie den oben diskutierten resultiert. Für jeden einzelnen (Kontroll- oder Datenzustand) kann eine explizite existentielle Eigenschaft generiert werden; dasselbe gilt für jede einzelne Transition, wenn bei der Testfallgenerierung die Information über die gefeuerten Transitionen zur Verfügung steht, was für den im nächsten Kapitel diskutierten Testfallgenerator der Fall ist.

3.2.2. Test universeller Eigenschaften

In diesem Abschnitt wird betrachtet, wie universelle Eigenschaften getestet werden können. Existentielle Eigenschaften können als Szenarien aufgefaßt werden, und ihr Test wurde im vergangenen Abschnitt untersucht.

Ein erster Ansatz für die Approximation universeller Eigenschaften besteht darin, die entsprechende Formel in einen äquivalenten Automaten zu übersetzen und eine zu definierende Überdeckung auf diesem zu fordern. Wenn die Eigenschaft sehr einfach ist, sind es die Automaten ebenfalls, und eine wie auch immer geartete strukturelle Abdeckung auf ihnen ist vergleichsweise trivial. Untersucht werden im folgenden deshalb Approximationen für Invarianzeigenschaften der Art $AG\varphi$ für Zustandsformeln φ oder solche φ, die sich höchstens auf den nächsten Schritt beziehen, d.h. EX- oder AX-Quantoren besitzen dürfen. Das Verfahren ist schematisch und wird anhand eines Beispiels demonstriert.

Die einfache Idee besteht darin, die Quantoren unter Semantikverlust zu vereinfachen, also AG durch EF zu ersetzen. Für Zustandsformeln findet ein Überdeckungsmaß Anwendung, das alle „wesentlichen" Teile berührt. Dabei finden bei geordneten Typen klassische Heuristiken wie das Grenzwerttesten Anwendung. Es wird also eine Abdeckung der Eigenschaft definiert. Vernachlässigbar sind dabei Ausgaben, wenn sie nicht im Antecedens einer Implikation auftauchen. Der Grund ist, daß die Ausgaben vom Modell geliefert werden. Vernachlässigbar sind häufig auch Auswertungen eines Antecedens zu „falsch", weil sie die Implikation in einem Sinn wahrmachen, der häufig als trivial empfunden wird. Schließlich muß überprüft werden, ob ein spezifizierter Systemzustand auch tatsächlich erreicht wurde, was durch Postambeln geschieht. Die vollständige Automatisierung erscheint nicht realistisch, weil beispielsweise die o.g. trivialen Belegungen für die Auswertung einer Implikation immer problemspezifisch sind.

Beispiel 13 (Schematische Approximation). *Die erfolgreiche Berechnung der digitalen Signatur erfolgt genau dann, wenn eine PIN verifiziert, die entsprechende Sicherheitsumgebung gesetzt und der Schlüssel ebenfalls korrekt gesetzt ist. Im Fall der PIN-NR (Sicherheitsumgebung 2) muß also* PSK_{NR} *gelten mit*

$$PSK_{NR} \equiv PINNR = verified \wedge SE = 2 \wedge KEY = Correct,$$

wobei die Zustandsvariablen abstrakt dargestellt sind (im Modell sind diese Werte in komplexeren Datenstrukturen verborgen). Für die PIN-G (Sicherheitsumgebung 1) muß alternativ gelten

$$PSK_G \equiv PING = verified \wedge SE = 1 \wedge KEY = Correct.$$

Wenn die Signatur korrekt berechnet wurde, wird der Wert `H61XX` *zurückgegeben; die Signatur muß direkt danach mit einem* `GetResponse`*-Befehl ausgelesen werden. Wenn die Signatur nicht berechnet werden kann, wird* `H6982` *ausgegeben; wenn der Puffer bei* `GetResponse` *leer ist, wird* `H6F00` *ausgegeben. Es gilt also*[3]

$$\begin{aligned} AG(&input = computeDigSig(_) \Rightarrow \\ &\quad (output = H61XX \wedge \\ &\qquad AX(input = getResponse \Rightarrow output = DigSig)) \\ &\vee \quad (output = H6982 \wedge \\ &\qquad AX(input = getResponse \Rightarrow output = H6F00))) \end{aligned} \tag{3.1}$$

[3] Auf die Angabe einer formalen Kripke-Semantik von AutoFocus für die Überprüfung von CTL-Formeln wird hier verzichtet. Wesentlich ist, daß Ein- und Ausgaben des Kartenmodells im selben Schritt erfolgen (Sequenzdiagramm auf S. 141).

für einen beliebigen zu signierenden String _, dessen abstrakte Signatur mit DigSig bezeichnet ist. Im folgenden werden die Abkürzungen $i?X$ und $o!X$ für input = X und output = X, csd für computeDigSig(_), gR für getResponse und ds für DigSig verwendet.

Da es darum geht, die Berechnung der digitalen Signatur zu testen, wird das Antecedens als wahr angenommen; Approximationen von 3.1 sind dann

$$EF(i?csd \wedge o!H61XX \wedge EX(i?gR \wedge o!ds)) \text{ bzw.}$$
$$EF(i?csd \wedge o!H6982 \wedge EX(i?gR \wedge H6F00)).$$

Die Implikation $i?csd \Rightarrow o!H61XX$ wird also zu $i?csd \wedge (i?csd \Rightarrow o!H61XX)$ und damit zu $i?csd \wedge o!H61XX$ verstärkt (für die zweite Implikation gilt Analoges). Zusammen mit einem Selektionskriterium („Betrachte nur Traces bis zur Länge n“) werden daraus Testfälle abgeleitet. Diese sind dadurch gekennzeichnet, daß die Eingabe irgendwann ein csd enthält, und direkt danach als Eingabe gR vorkommt. Weil zwei Eigenschaften getestet werden, enthält die eine auf das Kommando csd die Antwort H61XX und die andere die Antwort H6982.

Verstärkung und interner Zustand *Das Verhalten bei Berechnung der digitalen Signatur läßt sich präziser ausdrücken. Insbesondere kann der interne Zustand des Modells berücksichtigt werden. Die digitale Signaturberechnung läßt sich mit*

$$\begin{aligned} AG\Big(i?csd \Rightarrow \big(o!H61XX \Leftrightarrow ((PSK_G \vee PSK_{NR}) \wedge AX(i?gR \Rightarrow o!ds))\big)\Big) \\ \wedge AG\Big(i?csd \Rightarrow \big(o!H6982 \Leftrightarrow (\neg(PSK_G \vee PSK_{NR}) \wedge AX(i?gR \Rightarrow o!H6F00))\big)\Big) \\ \wedge AG\Big(\neg(i?csd) \Rightarrow AX\big(i?gR \Rightarrow \neg(o!ds)\big)\Big) \end{aligned} \tag{3.2}$$

als Verstärkung der Eigenschaft 3.1 charakterisieren.

Für das dritte Konjunkt ergibt sich als Testfallspezifikation analog zu den obigen Fällen

$$EF\big(\neg(i?csd) \wedge EX(i?gR)\big),$$

was zu Testfällen führt, in denen an beliebiger Stelle gR als Eingabe auftaucht, ohne daß vorher ein csd gesendet wurde.

Zum Test der ersten zwei Konjunkte der Eigenschaft 3.2 wird erneut das Antecedens der Implikation als wahr angenommen. Getestet wird zuerst die Äquivalenz „von links nach rechts“. Dazu wird zunächst das erste Konjunkt der rechten Seite der Äquivalenz berücksichtigt. Vier Approximationen ergeben sich sofort, nämlich

$$EF\big(i?csd \wedge (o!H61XX \wedge \overline{P}SKPostambule_{NR})\big),$$
$$EF\big(i?csd \wedge (o!H61XX \wedge \overline{P}SKPostambule_{G})\big),$$
$$EF\big(i?csd \wedge (o!H6982 \wedge \overline{P}(\overline{SK})Postambule_{NR})\big) \text{ und}$$
$$EF\big(i?csd \wedge (o!H6982 \wedge \overline{P}(\overline{SK})Postambule_{G})\big),$$

wobei die $\overline{P}SKPostambule$ und $\overline{P}(\overline{SK})Postambule$ Befehlsketten sind, die den Status der Zustandsvariablen PIN, SE und KEY abfragen: Im Fall von $\overline{P}SK$-Postambule wird überprüft, ob Sicherheitsumgebung und Schlüssel gesetzt sind,

im Fall von $\overline{P}(\overline{SK})$*Postambule wird überprüft, ob eins der beiden Elemente nicht gesetzt ist. Die PIN wird mit Wirkung für den nächsten Systemschritt nach Berechnung der digitalen Signatur in einen nicht-verifizierten Zustand versetzt. Deshalb wird in den Postambeln überprüft, ob die PIN nicht verifiziert ist, vgl. die Fußnote auf S. 29.*

Für das zweite Konjunkt der rechten Seite der Äquivalenz entstehen die Spezifikationen

$$EF\big(i?csd \wedge o!H61XX \wedge EX(i?gR)\big),$$
$$EF\big(i?csd \wedge o!H6982 \wedge EX(i?gR)\big),$$

was zu Testfällen führt, in denen das Kommando csd mit H61XX oder H6982 quittiert wird, woraufhin ein gR gesendet wird.

Für den Test der Äquivalenz von „rechts nach links“ ergeben sich für die erste Gleichung analog

$$EF(PSK_{NR} \wedge i?csd \wedge EX(i?gR)) \text{ bzw.}$$
$$EF(PSK_G \wedge i?csd \wedge EX(i?gR)).$$

Auch hier können Kommandos eingefügt werden, die vor Absenden des Kommandos csd die Gültigkeit von $PSK_{\{G,NR\}}$ *überprüfen. Auf die Spezifikation der Ausgabe kann verzichtet werden, weil auf Modellebene* $PSK_{\{G,NR\}}$ *erzwingt, daß auf ein csd die Ausgabe H61XX folgt und analog* $\neg PSK_{\{G,NR\}}$ *die Ausgabe H6982 auf ein csd erzwingt.*

Für die zweite Gleichung lauten die Approximationen ganz ähnlich:

$$EF(\neg PSK_{NR} \wedge i?csd \wedge EX(i?gR)) \text{ bzw.}$$
$$EF(\neg PSK_G \wedge i?csd \wedge EX(i?gR)),$$

die zuletzt noch mit einem strukturellen Abdeckungskriterium für Bedingungen verfeinert werden können. Im folgenden bezeichnen P, S und K die entsprechenden Konjunkte von PSK_G*; analog erfolgt die Verfeinerung für* PSK_{NR}*.* $\cdot$ *bezeichne die Konjunktion,* $\overline{X}$ *sei als* $\neg X$ *definiert. Die Idee ist einfach, alle Konjunkte dazu zu bringen, die Formel zu falsifizieren:*

$$EF(\overline{P} \cdot S \cdot K \wedge i?csd \wedge EX(i?gR))$$
$$EF(P \cdot \overline{S} \cdot K \wedge i?csd \wedge EX(i?gR))$$
$$EF(P \cdot S \cdot \overline{K} \wedge i?csd \wedge EX(i?gR))$$
$$EF(\overline{P} \cdot \overline{S} \cdot K \wedge i?csd \wedge EX(i?gR))$$
$$EF(\overline{P} \cdot S \cdot \overline{K} \wedge i?csd \wedge EX(i?gR))$$
$$EF(P \cdot \overline{S} \cdot \overline{K} \wedge i?csd \wedge EX(i?gR))$$
$$EF(\overline{P} \cdot \overline{S} \cdot \overline{K} \wedge i?csd \wedge EX(i?gR)),$$

wobei ein beliebiges anderes Coveragekriterium Verwendung finden kann. In allen diesen Formeln kann auf die Spezifikation der Ausgabe verzichtet werden, weil sie vom Modell geliefert wird.

Mit dem im folgenden Kapitel beschriebenen Testfallgenerator sind zusammen mit einer zusätzlichen Einschränkung (z.B. der maximalen Tracelänge) daraus Testfälle generierbar. Diese beinhalten alle als interessant erachteten Situationen für die Ausführung von csd.

Diese Betrachtungen sind bewußt exemplarisch gehalten und spiegeln einen Ansatz wider, den normalerweise impliziten Vorgang der Ermittlung von Testfallspezifikationen explizit zu machen.

Abstraktionen Im Bereich der formalen Verifikation finden zumeist Abstraktionen Verwendung, die konservativ bzgl. universeller Eigenschaften sind (Dams u. a., 1997; Loiseaux u. a., 1995; Graf und Saïdi, 1997). Diese Abstraktionen überapproximieren die Menge der erreichbaren Zustände, und die Verifikation einer universellen Eigenschaft auf abstrakter Ebene läßt sich auf die konkrete Ebene übertragen. Für falsifizierte universelle Eigenschaften gilt das nicht. Dual lassen sich auf konkreter Ebene verifizierte existentielle Eigenschaften auf die abstrakte Ebene übertragen; für falsifizierte hingegen gilt das nicht. Konsequenz ist, daß die klassischen Verfahren der u.U. automatisierten Berechnung von Abstraktionen für das Testen zwar verwendet werden können, die errechneten Testsequenzen aber ggf. nicht Sequenzen des abstrahierten Modells sind – so, wie Gegenbeispiele für falsifizierte universelle Eigenschaften nicht gezwungenermaßen Traces des abstrahierten Systems sind. Auf abstrakter Ebene generierte Testfälle müssen dann auf Anwendbarkeit im konkreten Modell überprüft werden.

3.2.3. Methodik

Zum Abschluß dieses Abschnitts werden methodische Aspekte der Verwendung von Szenarien, universellen Eigenschaften und Automaten kurz skizziert.

Szenarien Szenariobasierte Beschreibungen sind für ein erstes Verständnis des Verhaltens eines Systems durchaus geeignet. Nach Ansicht des Autors sollte die Spezifikation eines Systems allerdings möglichst vollständig sein, was sich auch in der Verwendung von expliziten, ausführbaren Verhaltensmodellen für die Testfallgenerierung niederschlägt: Modelle einer Maschine für die Testfallgenerierung beschreiben das Verhalten eines Systems auf abstrakte Weise vollständig und können somit auch als Spezifikation der Maschine angesehen werden. Sind diese Modelle – auf einem zu definierenden Abstraktionsgrad – nicht vollständig, können sie zur Testfallgenerierung auch nur eingeschränkt verwendet werden. Alle nicht spezifizierten Teile können nicht systematisch getestet werden, weil das Sollverhalten nicht explizit vorliegt. Analog können die nicht spezifizierten Teile auch nicht systematisch erstellt werden – wie das Krüger u. a. (1999) für die Übersetzung von MSCs in Statecharts und Lieberman (2001) für das Programming-By-Example propagieren, sondern müssen von den Entwicklern „erraten“ werden. Diese Betrachtungen lassen einen weiteren problematischen Aspekt der Sequenzdiagramme außer Acht, ihre komplizierte Semantik (Krüger, 2000) und damit verbunden die schwierige Interpretation durch einen Menschen nämlich.

Universelle Eigenschaften Allgemeine universelle Eigenschaften (allgemeine CTL-Formeln) scheinen aus einem methodischen Grund für den Zweck des Testens nicht uneingeschränkt geeignet zu sein. Ihre Formulierung ist nämlich

sehr schwierig – das ist auch ein Grund, warum sich szenario- oder Use-Case-basierte Spezifikationstechniken vergleichsweise hoher Beliebtheit erfreuen. Die Erfahrung der schwierigen Formulierung macht jeder, der solche Eigenschaften nachzuweisen versucht – die Erfahrung zeigt, daß sich in etwa der Hälfte der Fälle herausstellt, daß die Eigenschaft nicht korrekt spezifiziert wurde. Die intellektuelle Beherrschbarkeit von existentiellen Eigenschaften ist natürlicherweise wegen ihrer beschränkten Aussagekraft höher. Das gilt auch für Pfade, die sich aus Zustandsmaschinen als Testfallspezifikation ableiten lassen (obwohl es, je nach Automatendialekt, natürlich Äquivalenzen bzgl. der Ausdrucksmächtigkeit mit CTL-Fragmenten gibt).

Automaten Der Vorteil einer zustandsmaschinenbasierten Spezifikation, wie sie auch in AutoFocus Anwendung findet, besteht darin, einer operationellen Denkweise eher zugänglich zu sein, als das für CTL-Formeln der Fall ist. In Abhängigkeit von einem spezifischen Automaten- und Logikdialekt lassen sich zwar Äquivalenzen zwischen Formeln und Automaten aufstellen, Ausführbarkeit kann aber zu einem schnelleren Verständnis des Systems führen.

3.3. Überdeckungsmaße als Testfallspezifikationen

Funktionale Testfallspezifikationen sind inhärent domänenabhängig. Die Bemessung ihrer Qualität ist abgesehen von Aussagen der Art „die im Szenario codierte Anforderung ist getestet“ schwierig. Darüberhinaus wurde in Abschnitt 3.2.1 diskutiert, daß im Normalfall eine Vervollständigung von Szenarien notwendig ist.

Strukturelle oder Überdeckungskriterien sind im Gegensatz dazu nicht domänenabhängig. Ausgehend vom Code des zu testenden Artefakts, werden als Testfallspezifikationen klassischerweise Abdeckungen von Kontroll- und Datenfluß definiert. Das Artefakt ist üblicherweise Code; im Rahmen dieser Arbeit wird die Übertragung auf Modelle untersucht. Die betrachteten Techniken sind White-Box-Testverfahren (s. die Diskussion von White-Box und Black-Box im Zusammenhang mit modellbasiertem Testen auf S. 38) auf Modellebene und in Abhängigkeit vom gewählten Modell-/Code-Erstellungsszenario Black-Box-Tests auf Codeebene. Die Domänenunabhängigkeit bzw. auf syntaktischen Kriterien basierende Definition bedeutet insbesondere, daß entsprechende Testfallspezifikationen und damit auch Testfälle automatisch erzeugt werden können (vgl. Abschnitt 4.7).

Kontrollflußbasierte Kriterien definieren, in welcher Form die Elemente des Kontrollflußgraphen abgedeckt werden müssen. Das betrifft Anweisungen, Zweige, Paare von Zweigen, Pfade oder Schleifen. Analog definieren datenflußbezogene Kriterien, wie Elemente des Datenflußgraphen abgedeckt werden müssen, daß also beispielsweise mindestens ein Pfad von der Definition einer Variablen bis zu jeder Nutzung der Variablen abgedeckt werden muß.

Abdeckungen auf der Basis von Anforderungen werden im Rahmen dieser Arbeit in verschiedener Hinsicht untersucht.

- Zum ersten können die Beschreibungselemente von Modellen (Funktionsdefinitionen, Zustände, Transitionen und damit Kanalbelegungen) explizit mit Abschnitten eines Spezifikationsdokuments annotiert werden.

- Zum zweiten wird diskutiert, inwiefern Modelle von Maschinen Modelle der Anforderungen sind und insofern Abdeckungen des Modells direkt, d.h. ohne explizite Annotationen, eine Anforderungsabdeckung implizieren. Der Test eines Szenarios – eine funktionale Testfallspezifikation – stellt ebenfalls die Abdeckung einer oder mehrerer Anforderungen dar. Im Fall der Menge von Szenarien, die durch die erweiterte Zustandsmaschine für Modifikationen des Protokolls zur Berechnung der digitalen Signatur beschrieben werden, zeigt sich allerdings, daß hier Anforderungen getestet werden, die in den Spezifikationsdokumenten nicht explizit auftauchen.

- Zum dritten wird gezeigt, wie aus einem Modell Ein-/Ausgabepaare generiert werden können, die ihrerseits Anforderungen darstellen. Diese Form der Anforderungsabdeckung funktioniert nur, wenn Anforderungen aus Spezifikationsdokumenten auf Ein- und Ausgabepaare von Modellen abgebildet werden können.

Vorteile struktureller Kriterien Abdeckungsbezogene Kriterien werden bzgl. ihrer Fähigkeit, Fehler zu entdecken, kontrovers diskutiert (Howden, 1978b; Ntafos, 1984; Dupuy und Leveson, 2000; Thévenod-Fosse und Waeselynck, 1992; Thévenod-Fosse u. a., 1995). Riedemann (2002) zitiert Fehleraufdeckungsquoten von 12–92%. Die Notwendigkeit funktionaler Tests als Komplement wird deshalb auch nirgends ernsthaft bestritten. Der Vorteil struktureller Kriterien liegt

- in der automatisierbaren Erzeugung entsprechender Testfallspezifikationen und damit Testfällen,

- daraus resultierend in ihrer kostengünstigen Erzeugung und – im Idealfall – automatisierten Durchführung und

- in der Tatsache, daß sie quantifizierbar sind. So kann beispielsweise 100%ige Anweisungsüberdeckung oder MC/DC-Überdeckung (Abschnitt 4.7) gefordert werden, wie das beispielsweise von einem Standard der zivilen Avionik, dem DO-178B (RTCA und EUROCAE, 1992), vorgeschlagen wird.

Die ersten zwei Vorteile gelten ebenfalls für stochastische Testfallspezifikationen – wenn die Kosten für einen Test gegen Null gehen, dann ist ein Test mehr besser als einer weniger.

Kontrollflußbezogene Überdeckungskriterien sind üblicherweise auf der Basis von Units (z.B. Module in Modula oder Klassen in Java) definiert, die zunächst nicht direkt zum Integrationstest verwendet werden können. Ein Ansatz zur automatischen Generierung von Integrationstests mit MC/DC-Abdeckung wird in Abschnitt 4.7 vorgestellt.

Modell und Maschine Der Präzisionsunterschied zwischen Modell und Maschine führt sofort zu der Frage, welche Aussagen aus einer Testsuite, die auf modellbezogenen strukturellen Kriterien basiert, sich in bezug auf die Maschine treffen lassen.

Die Verwendung von Verhaltensmodellen wurde in zweifacher Hinsicht diskutiert. Einerseits dienen Modelle der Spezifikation einer Maschine, und andererseits werden sie als Testmodelle verwendet (z.B. die erweiterte Zustandsmaschine zum Test der digitalen Signaturberechnung).

- Modelle von Maschinen sind eine mehr oder weniger abstrakte Repräsentation nicht nur der Maschine selbst, sondern auch der zugrundeliegenden Anforderungen. Die Übertragbarkeit von abstrakten Modell- auf konkrete Maschinentraces erneut vorausgesetzt, liefern Testfälle auf der Basis der Modellstruktur somit eine Abdeckung der entsprechenden Anforderungen. In Abschnitt 3.3.2 wird gezeigt, wie einzelne Modellelemente direkt mit Anforderungen annotiert werden können. Es lassen sich nicht alle Anforderungen auf eine Transition oder eine spezifische Funktionsdefinition zurückführen – Sequenzen von Ausführungsschritten sind so nicht faßbar. Bei geeigneter Testfallspezifikation können die Modelle zur Generierung von Tests, die solche sich über die Zeit erstreckende Anforderungen abdecken, verwendet werden. Die Annahme ist, daß die wesentlichen Kontrollstrukturen auf funktionaler Ebene in Modell und Maschine übereinstimmen.

- Testmodelle codieren eine Menge von Szenarien. Eine in ihrer Stärke zu definierende strukturelle Abdeckung auf solchen Testmodellen gewährleistet dann den sorgfältigen Test der Menge der codierten Szenarien.

Im ersten Fall wird durch die Abdeckung des Modells eine Abdeckung der implizit im Modell codierten Anforderungen gewährleistet. Im zweiten Fall werden diese Anforderungen explizit gemacht. Im ersten Fall wird die Lösung eines Problems, die Struktur des Modells, zur Grundlage der Testfallgenerierung gemacht. Die implizite Annahme ist, daß sich Modell und Code nicht zu stark unterscheiden. Im zweiten Fall wird Abdeckung auf einer Menge von Szenarien gewährleistet.

3.3.1. Kontroll- und Datenfluß

Ntafos (1988) gibt einen Überblick über diverse strukturbezogene Teststrategien. Frankl und Weyuker (1988) konzentrieren sich auf datenflußbezogene Kriterien. In beiden Arbeiten werden die jeweiligen Abdeckungskriterien hinsichtlich ihrer „Stärke“ miteinander verglichen. Ein Vorschlag, wann welches Kriterium einzusetzen ist, ergibt sich daraus nicht. Im folgenden wird ein kurzer Überblick über die wichtigsten Kriterien gegeben.

Kontrollfluß Die einfachsten kontrollflußbezogenen Kriterien sind Überdeckungen von Anweisungen, Zweigen und Pfaden. Anweisungsüberdeckung fordert, daß jede Anweisung mindestens einmal getestet wird und ist eins der schwächsten

Kriterien, das deshalb als Minimalanforderung gilt. Zweigabdeckung fordert, daß jeder Transfer des Kontrollflusses einmal ausgeführt wird. Pfadabdeckung fordert die Ausführung jedes Ausführungspfads (Traces) und ist im Normalfall nicht erzielbar. Strukturiertes Pfadtesten und Boundary-Interior-Pfadtesten sind Einschränkungen des Pfadtestens. Verschiedene Kriterien werden definiert, die sich in der Anzahl der Iterationen von Schleifen unterscheiden. Im Boundary-Interior-Test werden für jede Schleife zwei Tests gefordert: einer, der die Schleife niemals durchläuft (boundary) und einer, der sie mindestens einmal (interior) durchläuft. Im strukturierten Pfadtesten wird für ein üblicherweise kleines k gefordert, daß jede Schleife höchstens k-mal durchlaufen wird. LCSAJs (linear code sequence and jump) werden anhand des Programmtextes definiert. Ein LCSAJ ist eine Sequenz aufeinanderfolgender Anweisungen im Programmtext, die an einem Eingangspunkt oder nach einem Sprunglabel beginnt und mit einem Ausgangspunkt oder einem Sprung endet. Schließlich gibt es diverse Kriterien zum Test von Bedingungen, z.B. die Forderung, daß jedes Literal einmal zu wahr und einmal zu falsch ausgewertet werden muß. Abschnitt 4.7 geht ausführlich auf ein vergleichsweise starkes Kriterium ein, MC/DC nämlich.

Datenfluß Datenflußbezogene Kriterien basieren auf Interaktionen von Variablendefinitionen und -referenzen. Eine Variable X wird referenziert, wenn eine Anweisung den Wert von X anfordert. X wird definiert, wenn eine Zuweisung an X stattfindet. Das einfachste datenflußbezogene Kriterium (2-dr bzw. 2-du für usage) besteht dann darin, einen Programmpfad mit einer Definition und einer Referenz zu testen. Man kann dann fordern, daß alle 2-dr-Interaktionen getestet werden, was Zweigabdeckung nicht gewährleistet. Die required-pairs-Strategie fordert auf der Grundlage von Datenflußanalysen die Abdeckung erforderlicher Paare: Für jede 2-dr-Interaktion wird mindestens ein Paar generiert. Für jede 2-dr-Interaktion in einem Zweigprädikat wird ein Paar für jede Auswertung des Konditionals erzeugt. Wenn Definition oder Referenz in einer Schleife auftauchen, wird ein Paar erzeugt, das sicherstellt, daß die Schleife bei der erstbesten Möglichkeit verlassen wird und eins, das eine festgelegte Anzahl von Schleifendurchläufen sicherstellt. Die required-k-tuples-Strategie erweitert das 2-dr-Kriterium auf Sequenzen der Länge k von Paaren. Weitere Datenflußkriterien beruhen auf der Unterscheidung zwischen Referenzen in Konditionalen (predicates, p-use) und in Berechnungen (computations, c-use). Die p-uses werden mit den Zweigen in Verbindung gebracht, die der Auswertung des Prädikats entsprechen. Man kann dann fordern, daß alle Interaktionen zwischen einem c-use und einem p-use getestet werden. Diese Strategie heißt all-uses. Verschiedene Abschwächungen dieses Kriteriums werden dann definiert (all-defs, all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses). Die all-du-paths-Strategie fordert, daß jede Interaktion zwischen einem p- oder c-use und einer Variablendefinition, die die Referenz erreichen, für jeden zyklenfreien Pfad zwischen beiden Anweisungen getestet wird.

Zustandsmaschinen (FSMs) Auf FSMs basierende strukturelle Kriterien werden von Ural (1992) und Fujiwara u. a. (1991) zusammengefaßt. Im wesentlichen

werden bei verschiedenen Anforderungen an die Struktur der FSM verschiedene Variationen von Transitions- und Zustandsabdeckung gefordert. Dabei können verschiedene (Beobachtungs-)Äquivalenzen (Milner, 1989; Tretmans, 1996; Brinksma, 1988) zur Reduktion der Transitionssysteme verwendet werden (Koppol u. a., 2002). Da AutoFocus auf erweiterten FSMs und nicht auf FSMs basiert und insbesondere beliebige funktionale Programme auf Transitionsebene ausgeführt werden können, erzielen diese Kriterien normalerweise nicht einmal Anweisungsüberdeckung und werden deshalb im folgenden nicht weiter betrachtet. Stattdessen wird in Abschnitt 4.7 ein Überdeckungskriterium für EFSMs definiert. Im Kontext von Testmodellen oder universellen Eigenschaften, die in Zustandsmaschinen übersetzt werden, besitzen Überdeckungskriterien auf FSMs aber durchaus Relevanz, weil diese häufig (Testmodelle, z.B. die Maschine zum Test digitaler Signaturen) oder immer (übersetzte Funktionen) keinen Datenzustand besitzen. Möglich ist eine explizite Übersetzung von EFSMS in FSMs.

3.3.2. Abdeckung von Anforderungen

Testen dient der Überprüfung der Übereinstimmung einer Maschine mit seinem intendierten Verhalten. Im Rahmen dieser Arbeit werden dazu Modelle verwendet, die die in den Spezifikationsdokumenten codierten Anforderungen zumeist implizit enthalten. Szenarien, d.h. exemplarische Abläufe, finden sich in den entsprechenden vollständigen Automaten wieder. Der Test des entsprechenden ggf. vervollständigten Szenarios liefert dann eine Abdeckung dieser speziellen Anforderung.

Wenn es gelingt,

- Elemente des Modells explizit mit Anforderungen zu annotieren oder
- aus dem Modell direkt Anforderungen zu extrahieren,

dann können für diese explizit Testfälle erzeugt werden.

Der erste Fall, der offenbar eng mit dem Problem der Anforderungsverfolgung zusammenhängt, wird dadurch realisiert, daß jede atomare Komponente mit einer weiteren Variablen v versehen wird, deren Datentyp A durch Seitenangaben oder Abschnitte des Spezifikationsdokuments definiert wird. In AutoFocus wird Verhalten mit erweiterten Zustandsmaschinen spezifiziert, auf deren Transitionen beliebige funktionale Programme ausgeführt werden können (Abschnitt 4.2). Wenn Transitionen schalten, dann wird diese Variable v auf den Wert, d.h. die Anforderung, gesetzt, der das Feuern der Transition entspricht. Feingranularer können Funktionsdefinitionen geliftet werden, indem der Resultattyp R zu einem Paar (R, A) erweitert wird, und der Wert der zweiten Komponente wird erneut durch die entsprechenden Passage in den Anforderungsdokumenten festgelegt. Das ergibt Schwierigkeiten, wenn Funktionsdefinitionen ihrerseits Funktionen aufrufen, oder wenn unterschiedliche Komponenten dieselbe Funktion aufrufen. In jedem Fall wird der Wert einer neuen Komponentenvariable dann bei Ausführung auf den Wert der zweiten Projektion gesetzt. Die Testfallspezifikation fordert dann Abdeckung einer spezifischen oder

aller Anforderungen, indem Abdeckung der jeweiligen Werte der hinzugefügten Komponentenvariablen gefordert wird.

Beispiel 14 (Modellannotation). *Im Chipkartenmodell können alle Transitionen der Komponenten für kryptographische Operationen mit den entsprechenden Abschnitten des Spezifikationsdokuments annotiert werden. Da die Funktionalität im wesentlichen in Funktionsdefinitionen codiert ist, werden genaugenommen nicht die Transitionen, sondern Funktionsdefinitionen annotiert. Funktionsdefinitionen für kryptographische Operationen werden ausschließlich von dieser Komponente verwendet.*

Die zweite Möglichkeit basiert auf der expliziten Ableitung von Anforderungen aus dem bestehenden Modell, z.B. interessante Kombinationen von Eingabe und Ausgabe. Betrachtet werden dann entsprechende Szenarien beliebiger Länge oder Szenarien der Länge n für ein festzulegendes n. Im Fall der Chipkarte liefert das bei unterspezifiziertem Initialzustand für $n = 1$ die Menge aller möglichen Ein-/Ausgabepaare auf Modellebene, die in den Spezifikationsdokumenten nicht explizit aufgeführt wird. Diese Information kann dann nicht nur zum Testen verwendet werden, sondern auch zur Erweiterung der Spezifikationen für Weiterentwicklungen des betrachteten Systems.

Beispiel 15 (Generierung von Anforderungen). *Die funktionale Dekomposition der Chipkarte erlaubt eine separate Erzeugung aller möglichen Ein- und Ausgabepaare. Dazu werden die relevanten Komponenten auf Toplevel (Abb. 2.1 ohne Prä- und Postprozessor) getrennt betrachtet und symbolisch (Kapitel 4) ausgeführt. Wenn alle möglichen Sequenzen der Länge* 1 *für jede Komponente berechnet werden, liefert das die Gesamtheit aller möglichen abstrakten, nicht instantiierten (Abschnitt 4.5) Ein-/Ausgabepaare. Im Fall des Chipkartenmodells sind das 81, von denen 4 aufgrund von Mehrdeutigkeiten in der Spezifikation in der tatsächlichen Karte nicht erreichbar sind. Für jedes einzelne Kommando wurden dann Testsequenzen generiert, indem nach einem Modelltrace gesucht wird, an dessen Ende das entsprechende Kommandopaar auftritt.*

3.4. Stochastische Testfallspezifikationen

Die dritte wesentliche Klasse von Testfallspezifikationen sind stochastische Testfallspezifikationen.

- Unter randomisierten Testfallspezifikationen werden solche verstanden, die gleichverteilt zufällig ausgewählte Eingaben über die Zeit liefern.
- Stochastische Testfallspezifikationen sind eine Obermenge der randomisierten Testfallspezifikation. Hier werden nicht-gleichverteilte Selektionen betrachtet, die beispielsweise auf der Grundlage von Nutzungsprofilen definiert werden.
- Randomisierte und stochastische Testfallspezifikationen basieren auf einer zufälligen Auswahl von Eingabedaten. Die Technik des Mutationstestens basiert auf einer zufälligen Modifikation von zu testenden Artefakten.

Randomisierte Spezifikationen Der Vorteil randomisierter Testfallspezifikationen liegt darin, daß sie ohne Aufwand erstellt werden können. In ihrer Wirksamkeit bzgl. Fehleraufdeckung sind sie allerdings umstritten (Ntafos, 1998, 1988; Hamlet und Taylor, 1990). Ihre Motivation liegt in einfachen – empirisch evaluierten – stochastischen Modellen, anhand derer nachgewiesen wird, daß randomisiertes Testen der auf Partitionierungen des Eingabedatenraums basierenden Alternative dann nicht unterlegen ist, wenn keine Aussage über erhöhte Fehlerwahrscheinlichkeit einer Partition getroffen werden kann. Das ist

- intuitiv einsichtig, denn wenn keine erhöhten Fehlerwahrscheinlichkeiten angegeben werden können, ist es offenbar egal, aus welcher Partition Daten ausgewählt werden und
- steht nicht im Widerspruch zur vergleichsweise hohen Verbreitung von Heuristiken wie dem Grenzwerttesten, das ja auf einer Partitionierung des Datenraums basiert. Der Grund ist, daß für solche Grenzwertfälle angeblich empirische Evidenz existiert, daß in Grenzbereichen verstärkt Fehler auftreten. Insofern ergibt sich für diese Partitionen eine erhöhte Fehlerwahrscheinlichkeit.

Der in den folgenden Kapiteln präsentierte Testfallgenerator basiert auf Systemspezifikationen, die mit EFSMs erstellt und die beliebige funktionale Ausdrücke auf Transitionen zulassen. Die Problematik der Ordnung von Funktionsdefinitionen in funktionalen Programmen stellt sich bei der erfolgenden Übersetzung nicht, weshalb die Implementierung einer zufälligen Auswahl von (a) Transitionen und (b) Funktionsdefinitionen trivial ist. Eine zufällige Auswahl von Transitionen und Funktionsdefinitionen impliziert die zufällige Auswahl von Eingabedaten.

Nutzungsprofile Anstelle von Gleichverteilungen für die Auswahl können auch andere Verteilungen angenommen werden. Das kann anhand von Nutzungsprofilen geschehen, die sich aus empirischen Auswertungen vergleichbarer, bereits verwendeter Systeme ergeben. Die berechtigte Frage, ob solche Nutzungsprofile adäquate Testfallspezifikationen darstellen, soll hier nicht diskutiert werden – diese Art von Spezifikationen liefert mit erhöhter Wahrscheinlichkeit Traces, die ihrerseits mit erhöhter Wahrscheinlichkeit auftreten. Wenn aufgrund von zu geringen Wahrscheinlichkeiten bestimmte Programmabläufe nicht ausgeführt werden, stellt sich die Frage, ob und inwiefern diese trotzdem getestet werden sollten. Nutzungsprofile werden im Rahmen der Testaktivitäten von Cleanroom (Prowell u. a., 1999) sowie zur Bestimmung garantierter Mean-time-to-failure verwendet. Im folgenden finden sie keine weitere Beachtung.

Mutationstesten und Fault Injection Die Doppelnatur von Testfallspezifikation und Qualitätsmaß für eine Testsuite (Zhu u. a., 1997) findet Ausdruck in der Idee des Mutationstestens (Ammann und Black, 1999) bzw. der Mutationsanalyse. Die Idee besteht darin, zunächst ein Artefakt geringfügig zu verändern, eine sog. Mutante zu bilden. Das ist auch die Grundidee der Fault Injection (Voas u. a., 1997). Mutationen werden beispielsweise durch das Abändern von

relationalen oder arithmetischen Operatoren (Ersetzung von Addition durch Subtraktion, von Ungleichheit durch Gleichheit) gebildet. Eine Testsuite wird dann gemäß ihrer Fähigkeit bewertet, solche Mutanten aufzudecken, also zu unterschiedlichen Ausgaben bei identischen Eingaben im originalen und im mutierten Artefakt zu führen. Ein Problem dabei ist, daß es Mutanten gibt, die die Semantik eines Programms nicht verändern. Solche sind üblicherweise schwer auszumachen.

Umgekehrt können mutierte Modelle zur Testfallgenerierung dienen. Im Unterschied zum bisher propagierten Verfahren kann dann die Tatsache, daß das Verhalten von mutiertem Modell und Code differiert, als Indiz dafür gewertet werden, daß das ursprüngliche Modell dem tatsächlichen Code entspricht. Offenbar ist es sinnvoll, sowohl nicht-mutierte als auch mutierte Modelle zur Grundlage der Testfallgenerierung heranzuziehen.

Einsatz Stochastische Testfallspezifikationen fanden starke Verwendung beim Test der Chipkarte: Mit Eigenschaften wurde zunächst eine Klasse „interessanter“ Traces definiert. Wenn deren Anzahl zu groß war, wurden zufällig einige wenige ausgewählt.

3.5. Formalismen für die Testfallspezifikation

Die Generierung von Testfällen aus Testfallspezifikationen ist ein neuer Zweig der Forschung, und demzufolge gibt es nach Kenntnis des Verfassers in der Literatur für Beschreibungsmittel für Testfallspezifikationen im hier verwendeten Sinn keine Vorschläge. Das liegt nicht zuletzt daran, daß das Verständnis von Testfallgenerierung als Suchproblem in der Literatur nicht verbreitet ist.

Testfallspezifikationen in der Literatur

Für die Notation von Testfällen, die keine Suche mehr erforderlich machen, finden programmiersprachliche, tabellen-, automaten- und sequenzdiagrammbasierte Formalismen Anwendung. Für rein transformative Systeme haben Grochtmann und Grimm (1993) den Classification Tree Editor eingeführt, mit dem Datentypen für einen speziellen Testfall partitioniert werden können. Die Übertragung auf reaktive Systeme ist nicht offenbar, weil Partitionierungen dort offenbar in jedem Schritt des Systems berücksichtigt werden müssen.

Automaten Da Testfälle Menge von I/O-Traces beschreiben, bieten sich Automaten unabhängig von der erforderlichen Überbrückung der Abstraktionsniveaus von Modell und Maschine direkt als Testfallspezifikation an. Auf Seite 75 findet sich ein Beispiel für den Test der Berechnung der digitalen Signatur. Automaten als Testfallspezifikation werden u.a. in den Werkzeugen Lurette (Raymond u. a., 1998) und Lutess (du Bousquet u. a., 2000) verwendet. Ein Selektionskriterium – wie Transitionsüberdeckung im Fall der Berechnung der digitalen Signatur – wird in den zitierten Arbeiten nicht gefordert, muß aber angegeben werden, um die Anzahl der zu testenden Traces und ihre Länge zu

beschränken. Testfallgenerierung auf der Basis von Model Checkern kann mit Automaten erfolgen, die (nach Negation der entsprechenden Eigenschaft) mit dem System parallel komponiert werden.

Programmiersprachen und Logiken Solche Automaten können auch

- in Form logischer Formeln notiert (s. die obige Diskussion) und
- programmiersprachlich formuliert werden, wie das in der Spezifikationssprache Promela für den Model Checker Spin oder in der Spezifikationssprache für den Model Checker Murφ geschieht. Diese Programmiersprachen dienen im wesentlichen der Spezifikation paralleler Prozesse und besitzen wenig mächtige Datentypen.

Eine standardisierte Testfallbeschreibungssprache ist die Tree and Tabular Combined Notation (TTCN). TTCN ist in Teil 3 des internationalen Standards 9646 OSI Conformance Testing Methodology and Framework definiert und liegt mittlerweile als TTCN-3 (Grabowski, 2000) unter dem Namen Test and Test Control Notation vor. TTCN-3 setzt Schnittstellen der zu testenden Implementierung voraus, mit dem in TTCN-3 spezifizierte Testfälle kommunizieren können. Die Sprache TTCN-3 ist modular organisiert. Neben den üblichen prozeduralen Sprachelementen (Definition lokaler Variablen, Konditionale, Zuweisung, Iteration, Funktionen) gibt es Konstrukte zur Implementierung von Timern, synchrone und asynchrone Befehle zum Senden und Empfangen von Nachrichten sowie für die Spezifikation von Verdikten (S. 37). Insbesondere für den Test nichtdeterministischer Systeme (Abschnitte 2.2.4 und 6.2) können Default Behaviors definiert werden, mit denen u.a.

- im Testfall nicht explizit spezifizierte Signale zugelassen oder verboten werden können und
- Standardtimer definiert werden können, die für jedes antizipierte empfangene Signal eine maximale Zeitspanne vorgeben.

TTCN-3 erlaubt auch die Spezifikation von Permutationen („Interleavings“) von Signalen, wenn die genaue Reihenfolge eintreffender Nachrichten irrelevant ist.

Sequenzdiagramme Neben einer tabellarischen Variante gibt es auch eine graphische Repräsentation (Baker u. a., 2001) von TTCN-3 in Form von Sequenzdiagrammen, die gegenwärtig als Profil der UML für die Testfallspezifikation diskutiert wird. Mit einer Erweiterung von Sequenzdiagrammen um gesonderte textuelle Beschreibungen von Datentypen und Variablendeklarationen entsprechen diese hierarchisch gegliederten Sequenzdiagramme der textuellen Programmiersprache: Alternativen beispielsweise werden durch Inlining spezifiziert, Signalpermutationen durch Coregionen. Sequenzdiagrammen wird ein Default-Behavior zugeordnet, das die Behandlung nicht explizit spezifizierter eingehender Signale (Ignorieren, Fehler) und Timing-Constraints vorschreibt.

Sequenzdiagramme scheinen zur Spezifikation von Testfällen eher als zur Systemspezifikation (Abschnitt 3.2.3) geeignet zu sein, weil einerseits die unvollständige Beschreibung in der Natur des Testens liegt und andererseits die Komposition von Sequenzdiagrammen mit ggf. unterschiedlichen Interpretationen (universell, existentiell, lose, exakt) nicht notwendig ist. Das Beispiel der graphischen Repräsentation von TTCN-3 zeigt, daß eine klar definierte Semantik mit Default Behaviors einfach zu erstellen ist. Sequenzdiagramme eignen sich insbesondere auch für die Testfallspezifikation im Sinn dieser Arbeit (d.h. als Grundlage für die Testfallgenerierung), wenn Komponenten weggelassen werden, deren Verhalten vom Testfallgenerator erzeugt wird, und wenn zwischen zwei Signalen andere zugelassen werden sollen, die nicht explizit im Sequenzdiagramm auftreten.

Constraints Die Spezifikation solcher „implizit“ zugelassener Signale kann mit Constraints erfolgen. Einfache Beispiele sind das Verbot oder Zulassen bestimmter Signale und das Verbot oder das Zulassen von Signalen in einer bestimmten Reihenfolge. Solche Constraints werden im Rahmen dieser Arbeit mit Constraint Handling Rules implementiert und müssen sich nicht auf die Signale zwischen zwei Signalen, sondern können sich auf den gesamten Testfall beziehen. Die Constraints sind im wesentlichen einfache funktionale Programme, die die Erzeugung der Modelltraces zur Laufzeit einschränken.

Abstraktionsniveaus Schließlich stellt sich die Frage nach der Überbrückung der Abstraktionsniveaus von Modell und Maschine. Wenn Testfälle für Modelle spezifiziert werden, muß zusätzlich festgelegt werden, wie die abstrakten Modelltraces in konkrete Maschinentraces übersetzt werden. Wenn eineindeutige Zuordnungen möglich sind, kann das tabellarisch geschehen. Wenn nicht – wie im Fall der Behandlung kryptographischer Befehle oder der Frage nach Zufallszahlen im Beispiel der Chipkarte –, muß eine wohldefinierte Übersetzung angegeben werden. Im Fall der Chipkarte erfolgt diese Spezifikation ad-hoc. Ein systematischerer Zugang, der auch die Problematik der abstrakten Spezifikation von Maschinen und das Verhältnis dieser Spezifikation zur Implementierung beinhaltet, ist Teil zukünftiger Arbeiten.

Im Werkzeug Testframe, das auf den Action Words von Buwalda und Kasdorp (1999) beruht, findet eine Beschreibung von Testfällen auf verschiedenen Abstraktionsniveaus statt. Testfälle werden zunächst (manuell) in Cluster zergliedert, die in etwa dasselbe übergeordnete Testziel auf in etwa demselben Detaillierungsgrad beschreiben. Die Testfälle eine Clusters werden dann in sog. Test Conditions gruppiert und mit einer konzisen umgangssprachlichen Beschreibung versehen, die der Referenzierung dient. Jede Test Condition besteht dann aus mehreren Testfällen, die auf verschiedenen Abstraktionsniveaus (low-level, intermediate-level und high-level) beschrieben werden. Die Konkretisierung erfolgt mit den im Testwerkzeug vorhandenen Skriptsprachen.

Testfallspezifikationen in dieser Arbeit

In dieser Arbeit werden Testfallspezifikationen als Grundlage für die Suche nach Testfällen verwendet. Auf die Angabe einer konkreten Syntax wird verzichtet; im verbleibenden Teil dieses Abschnitts werden deshalb nur einige Beispiele für die im Rahmen der Fallstudie aufgetretenen Spezifikationen genannt.

Funktionale Spezifikationen Funktionale Spezifikationen werden in Form von Zustandsmaschinen mit einem Auswahlkriterium der Traces, in Form einfacher temporallogischer Formeln oder als Kombination beider formuliert. Als Beispiel für temporallogische Spezifikation dienen die Spezifikationen aus Beispiel 13. Diese können direkt in den Testfallgenerator (Kap. 4) eingegeben werden; der erste die Spezifikation erfüllende Trace wird dann ausgegeben. Möglich ist es auch, alle die Eigenschaft erfüllende Traces bis zu einer bestimmten Tiefe ausgeben zu lassen; die Spezifikation lautet dann `all(E,N)` für eine Eigenschaft E mit ausschließlich existentiellen Quantoren und eine maximale Suchtiefe N. Ein Beispiel für die Spezifikation mit Automaten ist das der digitalen Signaturberechnung (Abb. 3.1). Auf diesem Automaten muß ein zusätzliches stochastisches oder überdeckungsbezogenes Kriterium (s.u.) angegeben werden.

Stochastische Spezifikationen Stochastische Spezifikationen werden in Kombination mit der o.a. Spezifikation `all` angewendet, indem sie zufällig einen bestimmten Prozentsatz aller Traces auswählen: `all(E,N) and choose(P)`, wobei P den Prozentsatz spezifiziert. Stochastische Spezifikationen wurden auf den Automaten für die digitale Signaturberechnung und auf das Modell der WIM selbst angewendet.

Abdeckungskriterien Die Forderung, Testfälle zu erzeugen, die alle Transitionen der Maschine zur digitalen Signaturberechnung bei vorgegebener maximaler Suchtiefe mindestens einmal ausprobiert, wird mit dem Constraint `coverTransitions(20)` für eine Maximaltiefe 20 angegeben. Coveragekriterien finden aber auch auf dem Modell der WIM selbst Anwendung, wie das etwa in Abschnitt 4.7 erfolgt. Die Testfallspezifikation lautet dann `mcdc(20)`, wobei der Parameter erneut die maximale Suchtiefe angibt.

Constraints zur Suchraumeinschränkung Im Fall der Chipkarte ist es für viele Tests sinnvoll, das `CardReset`-Kommando auszuschließen. Das geschieht mit einem Constraint `never(CardReset)`. Für den Test der Cardholder Verification (Verifikation von PINs) wurde für einige der Tests gefordert, daß die entsprechenden Wiederholzähler nur abnehmen oder gleichbleiben, aber niemals zunehmen dürfen. Das geschieht mit einem Constraint `decreasing`, der den entsprechenden Zählerwert im Modell beschränkt. Alternativ hätten auch alle die Kommandos explizit ausgeschlossen werden können, die die Zähler erhöhen, also die korrekte Eingabe einer PIN oder der zugehörigen PUK. Alle in dieser Arbeit verwendeten Arten von Constraints werden in Abschnitt 5.3 zusammengefaßt.

3.6. Zusammenfassung

Testfälle sind immer nur so gut wie die zugehörigen Testfallspezifikationen. Die in der industriellen Praxis verbreitete Hoffnung, allein auf Grundlage eines Modells eine „ausreichende“ Menge „guter“ Testfälle ableiten zu können, kann zum gegenwärtigen Zeitpunkt nicht befriedigt werden. Das liegt insbesondere daran, daß keine allgemeingültigen Kriterien für die Güte einer Testsuite existieren, und es ist auch unwahrscheinlich, daß solche gefunden werden.

Funktionale Spezifikationen Domänen- und applikationsspezifische Testfälle werden auf der Grundlage funktionaler Testfallspezifikationen erstellt. In Spezifikationsdokumenten explizit codierte Anforderungen beispielsweise in Form von Sequenzdiagrammen stellen solche Anforderungen dar. Es wurde gezeigt, wie Sequenzdiagramme für Modifikationen der relevanten Szenarien in EFSMs vervollständigt werden können. Diese Testmodelle – Testtreiber – dienen dann als Grundlage weiterer struktureller oder stochastischer Testfallspezifikationen.

Strukturelle Spezifikationen Strukturelle Testfallspezifikationen basieren auf den syntaktischen Elementen eines Modells. Verschiedene kontroll- und datenflußbezogene Kriterien wurden präsentiert und ihre Transformation von Code auf Modelle skizziert. Eine Detaillierung erfolgt in Abschnitt 4.7. Solche Kriterien können sowohl auf das Modell der Maschine, als auch auf ein Testmodell angewendet werden.

Der fundamentale Vorteil solcher struktureller Kriterien – wie auch der dritten diskutierten Klasse, den stochastischen oder auf Mutationen basierenden Testfallspezifikationen – liegt in der Tatsache, daß sie vollautomatisch erzeugt werden können. In Anwesenheit automatischer Testfallgeneratoren können so große Mengen von Tests zu vergleichsweise geringen Kosten erzeugt und in Abhängigkeit von der Natur der Maschine auch ausgeführt werden.

Anforderungsabdeckung Strukturelle Kriterien können auch auf Anforderungen bezogen werden. Zunächst kann ein Modell als Modell der Anforderungen begriffen werden, so daß jede syntaktische Abdeckung eine Abdeckung der Anforderungen darstellt. Angesichts der Tatsache, daß zur Testfallgenerierung geeignete Modelle allerdings in bezug auf den untersuchten Aspekt (Abschnitt 2.3.1) durchaus sehr präzise sind – denn andernfalls ist die Überbrückung der Abstraktionsniveaus von Modell und Maschine sehr schwierig –, ist diese Aussage für die Testfallgenerierung insofern wenig hilfreich, als sie für jede Form von Code ebenfalls gilt. Das motiviert u.a. die Notwendigkeit funktionaler Tests beispielsweise in Form von Testmodellen. Es wurde gezeigt, wie bestimmte Anforderungen direkt zur Annotierung von Modellelementen verwendet werden können und wie für die Art von Anforderungen, für die das möglich ist, Anforderungsabdeckung erzielt werden kann.

Stochastische Spezifikationen Die dritte zentrale Klasse ist die der stochastischen Testfallspezifikationen. Unter Annahme einer Verteilung der Eingabe-

daten können bzgl. dieser Verteilung Traces berechnet werden. Der Vorteil des in den folgenden Kapiteln präsentierten Testfallgenerators liegt u.a. darin, daß für alle drei Klassen direkt Testfälle erzeugt werden können.

Interner Zustand Da der Test von Maschinen anhand von I/O-Traces erfolgt, muß unter Umständen überprüft werden, ob der vom Modell antizipierte Zustand auch tatsächlich dem Maschinenzustand entspricht. Das geschieht mit Postambeln, an die eigentlichen Testsequenz angehängte I/O-Traces, die der Überprüfung dienen, ob der Zustand auch tatsächlich erreicht wurde. Postambeln müssen ebenfalls explizit spezifiziert werden. Ihre automatische Erzeugung erscheint schwierig, da in Abhängigkeit vom System festgelegt werden muß, wie vom Ein-/Ausgabeverhalten auf den internen Zustand geschlossen werden kann.

Suchraumeinschränkung In technischer Hinsicht dienen Testfallspezifikationen nicht nur der Spezifikation von Testfällen, sondern auch der Suchraumeinschränkung. Testfallspezifikationen definieren, inwiefern die Testfallgenerierung sich auf bestimmte Abschnitte dieses Suchraums konzentriert. Die Verwendung zusätzlicher Einschränkungen – Constraints – wird als Schlüssel zu einer graceful degradation des in den folgenden Kapiteln präsentierten Testfallgenerators verstanden (Abschnitt 5.3).

Universelle Eigenschaften Universelle Eigenschaften können nicht direkt getestet werden. Wenn ein Modell eine Invarianzeigenschaft erfüllt, liefert diese z.B. von Model Checkern gelieferte Einsicht keinerlei Erkenntnis, wie die entsprechende Maschine getestet werden soll. Ein Ansatz besteht in der Übersetzung in EFSMs, aufgrund derer dann anhand struktureller oder stochastischer Testfallspezifikationen Testfälle zum Test dieser Eigenschaft abgeleitet werden. Wenn die resultierenden Automaten zu einfach sind, sind diese Kriterien oftmals trivial. Ein Ausweg besteht darin, Überdeckungen des Zustandsteils einer Eigenschaft bei gleichzeitiger Ersetzung von Allquantoren durch Existenzquantoren zu definieren. Inwieweit diese Testfallspezifikationen beispielsweise stochastischen überlegen sind, muß sich in der Praxis zeigen.

Sprachen für die Testfallspezifikation Schließlich wurde die Problematik der Beschreibung von Testfallspezifikationen diskutiert. Sprachen zur Testfallspezifikation, die als Eingabe für Testfallgeneratoren verwendet werden, sind z.Z. wegen eines Mangels an Testfallgeneratoren nicht vorhanden. In dieser Arbeit werden Testfallspezifikationen als Constraints beschrieben: ein bestimmter Zustand soll erreicht werden, bestimmte Signale oder Signalfolgen sollen verboten werden. Sequenzdiagramme mit solchen Constraints stellen einen attraktiven Kandidaten dar.

Zukünftige Arbeiten Zukünftige Arbeiten ergeben sich in den folgenden Bereichen.

- Das Problem der Beschreibung von Testfallspezifikationen für Testfallgeneratoren ist weitgehend ungelöst.

- Auch wenn der Vorteil struktureller Kriterien darin liegt, daß sie zur automatischen Erzeugung von Testfallspezifikationen und dann Testfällen verwendet werden können, müssen sie doch durch funktionale Tests komplementiert werden. Das ist eine intellektuell anspruchsvolle Aufgabe, für die nach Wissen des Autors keine methodischen Hilfestellungen gegeben werden. Eine domänen- und ggf. sogar applikationsklassenabhängige Klassifizierung potentieller Fehler anhand von Erfahrungen mit Vorläufer- oder ähnlichen Produkten liefert einen Ansatz zur Lösung des Problems. Das kann domänenspezifische Ausprägungen von Fehlerbäumen einschließen. Solche Klassifikationen anhand empirischer Studien sind nicht zuletzt aufgrund psychologischer Aspekte sehr schwierig: Die explizite Dokumentation von Fehlern läßt immer auch Rückschlüsse auf die Verursacher zu, die dementsprechend in ihrer Bereitschaft zur Dokumentation zurückhaltend sein werden. Auch wenn die in den folgenden Kapiteln erörterten Technologien für die Testfallgenerierung notwendig sind, basieren sie für funktionale Spezifikationen auf der Abhängigkeit explizit angegebener Szenarien, ohne deren Herkunft genau zu durchleuchten.

- Die Diskussion um existentielle und universelle Eigenschaften liegt im Herz der Fragestellung nach adäquaten Beschreibungsformalismen und Beschreibungstechniken. Existentielle, also szenariobasierte, weisen ebenso Vor- und Nachteile auf wie universelle (Automaten, Logiken wie CTL). Orthogonal dazu stellt sich die Frage nach der Verwendung deklarativer (algebraische Spezifikationen, temporallogische Formalismen) versus operationeller (Automaten, abstrakter Code) Spezifikationstechniken. Diese Fragen betreffen nicht nur den Test, sondern selbstverständlich auch den gesamten Spezifikations- und Designprozeß. Die Suche nach Kombinationen der Techniken, die die Vorteile kombinieren und Nachteile vermeiden, ist eines der Kernprobleme des modernen Software Engineering.

- Die existentielle Approximation ist für andere Eigenschaften als Invarianten zu untersuchen. Responseeigenschaften $AG(\varphi \Rightarrow AF\psi)$ können beispielsweise approximiert werden, indem zunächst Traces ausgewählt werden, die zu einem φ erfüllenden Zustand führen, von dem aus dann jeweils ein ψ erfüllender Zustand gesucht wird, d.h. $EF(\varphi \Rightarrow EF\psi)$. Das scheint ähnlich wie in Beispiel 13 für Invarianten machbar zu sein.[4] Die Approximation universeller durch existentielle Eigenschaften sollte in bezug auf ihre Fähigkeit, Fehler zu entdecken, empirisch evaluiert werden. Das würde bedeuten, daß zwei unabhängige Teams Testfälle für ein System entwickeln, von denen eins explizit nach dem vorgestellten Verfahren Testfälle aus universellen Eigenschaften entwirft und das andere nicht. Gemäß einer festzulegenden Metrik wären die beiden Testsuiten zu bewerten, etwa in der Anzahl gefundener Fehler.

- Modellmutationen wurden nicht untersucht, sondern nur angedacht. Eine Implementierung erscheint vergleichsweise einfach, bedarf aber einer

[4] In der Chipkarte scheint es keine „interessanten“ Lebendigkeitseigenschaften zu geben.

Evaluierung. Das würde wiederum Tests von zwei unabhängigen Teams erforderlich machen, die geeignet zu vergleichen wären.

- Im folgenden Kapitel wird erörtert, wie für ein spezifisches Überdeckungskriterium, MC/DC, Testfallspezifikationen und Testfälle generiert werden können. Eine Übertragung auf beliebige, insbesondere auch datenflußbezogene Kriterien erscheint aufgrund des Automatisierungspotentials zumindest für digitale Hardware oder Software wünschenswert: Die Kosten für die automatische Testerstellung sind gering, und wenn das ebenfalls für die Kosten der Testdurchführung gilt und Fehler gefunden werden, sind sie sofort amortisiert.

- Transitionen und Funktionen können mit Worst-Case-Ausführungszeiten annotiert werden. Die Testfallgenerierung kann dann dazu verwendet werden, maximale (minimale, durchschnittliche) Ausführungszeiten zu ermitteln, wenn als „Testfallspezifikation" die Forderung nach möglichst hohen (niedrigen) Ausführungszeiten als Summe aller Ausführungszeiten aller Komponenten für Traces einer vorgegebenen Länge angegeben wird.

- Ernst u. a. (2001) beschreiben ein Verfahren, mit dem ausgehend von Testfällen (Szenarien) Invarianten eines Programms „geraten" werden, die für Modifikationen von Programmen verwendet werden, die die Invariante nicht verändern sollen (methodisch vergleichbar also der Erzeugung von Testfällen für Refactoring-Schritte, s.S. 55). Der Schritt hier ist allerdings vom Szenario zur Eigenschaft. Zu untersuchen wäre etwa, wie sich automatisch generierte strukturelle Testfälle zur Generierung von Invarianten eignen.

- Schließlich scheint es keine generell anwendbaren Kriterien für die Qualität einer Testsuite zu geben. Die fortgesetzte diesbezügliche Untersuchung von Überdeckungsmetriken erscheint nach wie vor notwendig.

4. Testfallgenerierung

Dieses Kapitel beschreibt die Erzeugung von Testfällen für AutoFocus-Modelle. Testfälle werden aus der Konjunktion eines Modells und einer Testfallspezifikation abgeleitet. Das geschieht durch die symbolische Ausführung des Modells. Resultat sind insofern Äquivalenzklassen von Traces des Modells, als sie nicht durch die Abfolge von Signalen, sondern vielmehr durch Mengen von Signalen definiert sind. Gerechnet wird nicht mit einzelnen Werten, sondern mit Repräsentationen von Mengen von Werten. Betrachtet wird ein Spezialfall der Generierung von Testfällen, nämlich die Erzeugung solcher, die keine interne Ablauflogik in Form von Alternativen besitzen. Wenn Modell der Maschine und der Umgebung deterministisch sind, zum Zweck der Testdurchführung adäquat deterministisch abstrahiert oder deterministisch kontrolliert werden können, stellt das keine Einschränkung dar (Abschnitt 2.2.4). Die Generierung von Testfällen mit interner Ablauflogik wird im Ausblick im abschließenden Kapitel skizziert.

Der Ansatz eignet sich zur kompositionalen Generierung von Integrationstestfällen. Ausgehend von Testfällen für einzelne Komponenten, werden Testfälle für komponierte Systeme generiert. Das Verfahren wird iteriert, bis Testfälle für das Gesamtsystem generiert wurden. Der Vorteil dieses Vorgehens liegt darin, daß wegen des kleineren Zustandsraums Testfälle für einzelne Komponenten leichter zu generieren sind als solche für integrierte Systeme. Testfälle für einzelne Komponenten beschränken dann den Suchraum für zusammengesetzte Systeme.

Gliederung Abschnitt 4.1 skizziert zunächst das Vorgehen. Die Beschreibungstechniken und zeitsynchrone Ausführungssemantik des CASE-Werkzeugs AutoFocus werden in Abschnitt 4.2 vorgestellt und anhand des Chipkartenbeispiels illustriert.

Grundlage der symbolischen Ausführung ist eine Übersetzung in eine Constraint-Logik-Programmiersprache (CLP-Sprache), deren Grundlagen in Abschnitt 4.3 vorgestellt werden. Es folgt die Übersetzung der Modellierungselemente von AutoFocus nach CLP. Dabei handelt es sich um funktionale Programme, erweiterte Zustandsmaschinen und komponierte Zustandsmaschinen. Die Übersetzung ist intuitiv eingängig. Schwierigkeiten ergeben sich für verschiedene Formen der bei der Übersetzung auftauchenden Negation (Pattern Matching, Idle-Transitionen), die deshalb sorgfältig untersucht wird. Dabei wird außer in Ausnahmefällen auf die Verwendung einer konkreten Programmiersprache verzichtet. Stattdessen wird der abstrakte Formalismus der Hornklauseln gewählt, die sich direkt in tatsächliche CLP-Programme übertragen

lassen, was im Rahmen des Implementierungsteils dieser Arbeit auch geschehen ist. Der Abschnitt schließt mit einer kurzen Übersicht der verwendeten Optimierungen. Die Übersetzung von EFSMs (erweiterten endlichen Zustandsmaschinen) ist von Lötzbeyer und Pretschner (2000a) und Lötzbeyer und Pretschner (2000b) publiziert, die Übersetzung funktionaler Programme von Pretschner und Philipps (2003).

Abschnitt 4.4 erläutert, wie anhand der generierten CLP-Programme Äquivalenzklassen von Testfällen generiert werden. Diese Äquivalenzklassen werden dann randomisiert oder mit Grenzwertanalysen in Testfälle instantiiert, was in Abschnitt 4.5 beschrieben wird.

In Abschnitt 4.6 wird gezeigt, wie die einfache zeitsynchrone Ausführungssemantik von AutoFocus zur effizienten kompositionalen Generierung von Testfällen ausgenutzt werden kann. Die Technik ist insofern kompositional, als Testfälle für eine Komponente zur Erzeugung von Testfällen für ein diese Komponente beinhaltendes Subsystem verwendet werden. Insbesondere können so automatisch Integrationstestfälle erzeugt werden. Generiert werden nicht nur Testfälle, sondern auch Testfallspezifikationen.

Das wird anhand des Beispiels eines vergleichsweise starken kontrollflußbezogenen Coveragekriteriums (MC/DC) in Abschnitt 4.7 illustriert. Resultat der diskutierten Technik ist eine Testsuite, die das Coveragekriterium nicht nur auf Unitebene (d.h. Funktionsdefinitions-, Transitions- oder EFSM-Ebene) erfüllt, sondern sogar für das integrierte Gesamtsystem. Um die Komplexität der Testfallerstellung zu reduzieren, sind kontrollflußbezogene Coveragekriterien aus pragmatischen Gründen im Normalfall auf Units beschränkt. Der Nachteil, daß entstehende Testsuiten Testfälle enthalten können, die im integrierten System nicht durchführbar sind, wird mit dem vorgestellten Verfahren behoben. Im wesentlichen besteht der Ansatz darin, aus einem Modell automatisiert *Testfallspezifikationen* abzuleiten, aus denen dann später Testfälle generiert werden können. Experimentelle Resultate finden sich im folgenden Kapitel im Abschnitt 5.4. Zentrale Inhalte dieses Abschnitts sind von Pretschner (2003) und Bender u. a. (2002) publiziert.

Abschnitt 4.8 präsentiert verwandte Arbeiten und Techniken. Abschnitt 4.9 faßt die Resultate dieses Kapitels zusammen.

Publizierte Anwendungen des Testfallgenerators sind Firewalls (Jürjens und Wimmel, 2001), das Modell einer Ampelsteuerung (Pretschner u. a., 2003a), das Modell einer Vorflügelsteuerung (Blotz u. a., 2002), Chipkarten zur Zugangskontrolle (Pretschner u. a., 2001b) und die WIM (Philipps u. a., 2003).

4.1. Vorgehen

Unabhängig von der in Abschnitt 2.4 diskutierten Reihenfolge von Modell- und Codeentwicklung wird im folgenden das Vorgehen zur Berechnung von Testfällen, d.h. der im Rahmen dieser Arbeit entwickelte Testfallgenerator, geschildert. Die Idee ist, aus einem AutoFocus-Modell zunächst ein CLP-Programm zu generieren. Wie im vorigen Kapitel diskutiert, können funktionale, strukturelle und stochastische Testfallspezifikationen als existentielle Eigen-

schaften aufgefaßt werden:

- Wenn ein Sequenzdiagramm eine Testfallspezifikation darstellt und nicht einen Testfall, weil im spezifizierten Szenario nur ausgewählte Systemkomponenten auftreten oder weil die Semantik insofern lose ist, als zwischen zwei Signalen durchaus andere Signale auftreten können, dann besteht die Aufgabe des Testfallgenerators darin, die „fehlenden“ Signale zu generieren.
- Strukturelle Kriterien wie Transitions-, Zustands- oder Bedingungsüberdeckung lassen sich als eine Menge existentieller Testfallspezifikationen begreifen, die jeweils einen Pfad zu jedem Zustand, jeder Transition oder jeder Bedingung spezifizieren.
- Stochastische Testfallspezifikationen fordern, daß eine Menge von Traces beliebig ausgewählt wird, oder daß die Auswahl anhand einer Verteilung typischer Eingabe- oder Ausgabedaten erfolgt. Es soll also eine Menge von Pfaden gefunden werden, die den stochastischen Kriterien genügt.

Das Problem ist offenbar ein Suchproblem: Zunächst muß ein Trace gefunden werden, der zum ersten spezifizierten Signal führt, dann muß ein Trace gefunden werden, der zum zweiten spezifizierten Signal führt usw. Unter Umständen ist dabei Backtracking erforderlich. Die entsprechenden Testfallspezifikationen werden dann ebenfalls in ein CLP-Programm übersetzt und zusammen mit dem übersetzten Modell simuliert. Dabei werden die fehlenden Signale gesucht. Das Resultat ist ein Testfall, der die Testfallspezifikation insofern erfüllt, als alle Signale der Testfallspezifikation tatsächlich in der spezifizierten Reihenfolge auftauchen. Unter Umständen ist der Suchraum zu groß. Dann müssen vom Benutzer Constraints angegeben werden, die den Suchraum einengen.

Die verwendete Technik ist die der symbolischen Ausführung. Anstatt alle Programm- bzw. Modellpfade auszuprobieren, werden Mengen (bzgl. der Auswahl von Transitionen) äquivalenter Testfälle erzeugt. Die resultierenden Testfälle sind dann symbolischer Natur und müssen instantiiert werden, d.h. aus den verschiedenen Mengen muß jeweils ein Repräsentant ausgewählt werden. Das kann etwa zufällig oder durch Grenzwertanalysen geschehen.

Der Unterschied zwischen dem Abstraktionsniveau der erzeugten Testfälle und dem der Maschine wird dann von einem Testtreiber überbrückt.

4.2. Systemmodellierung mit AutoFocus

Dieser Abschnitt beschreibt die Modellierungssprache des Werkzeugs AutoFocus (Huber u. a., 1997; Schätz u. a., 2002a; Schätz und Huber, 1999; Philipps und Slotosch, 1999) sowie die in dieser Arbeit betrachtete Fallstudie und ihr Modell.

4.2.1. AutoFocus

Struktur In AutoFocus werden Systeme als Netzwerke hierarchischer Komponenten, sog. Systemstrukturdiagramme (SSD), spezifiziert, die über getypte

und gerichtete Kanäle miteinander kommunizieren. Die Schnittstelle von Kanal und Komponente wird als Port bezeichnet; Ports müssen denselben Typen besitzen wie die angeschlossenen Kanäle. Ein Ausgabeport kann mit mehreren Kanälen verbunden sein, ein Eingabeport mit höchstens einem. Atomare Komponenten, d.h. Komponenten an Blattpositionen der Hierarchie, sind mit einer erweiterten Mealy-Zustandsmaschine (EFSM) ausgestattet, die auf für die Komponente definierte Variablen zugreifen kann. Komponenten kapseln also einen Teil des Verhaltens zusammen mit einem Teil des Datenraums. Abb. 4.3 zeigt eine Ebene des Systemstrukturdiagramms der WIM.[1]

Kommunikation Alle Komponenten führen ihre Berechnungen zeitgleich durch. Wenn ein globaler Tick vorliegt, lesen alle Komponenten zeitgleich ihre Eingabeports, führen ihre Berechnungen durch und schreiben Ausgaben auf ihre Ausgabeports. Von dort werden sie vor dem folgenden Tick auf die entsprechenden Eingabeports kopiert, d.h. über den Kanal übertragen. Dort können die Werte dann beim folgenden Tick gelesen werden. Neben dieser einfachen zeitsynchronen Kommunikation mit Puffergröße 1 gibt es auch eine immediate Kommunikation (Ports gekennzeichnet durch Rauten). Immediate Kommunikation ist dadurch gekennzeichnet, daß das Ergebnis der Berechnung einer Komponente nicht erst beim nächsten Tick, sondern bereits davor vorliegt. Das bedeutet, daß die Komponenten nicht mehr exakt gleichzeitig ihre Werte lesen, sondern daß durch die immediate Kommunikation eine Ordnung induziert wird. Eine Komponente k_1 ist bzgl. dieser Ordnung kleiner als eine Komponente k_2, falls ein Ausgabeport von k_1 mit einem Eingabeport von k_2 verbunden ist. Die Ausführungsreihenfolge ist dann durch diese topologische Sortierung festgelegt. Offenbar dürfen keine Zyklen in der immediaten Kommunikation auftreten, was in der Realität zu einer Verletzung von Kausalitätsprinzipien führen würde. Das wird vom Werkzeug überprüft. Intuitiv kann wegen des einfachen zeitsynchronen Kommunikationsschemas ein Kanal als lokale Variable derjenigen Komponente aufgefaßt werden, in der die beiden kommunizierenden Komponenten eingebettet sind.

Verhalten: Zustandsmaschinen (EFSMs) Die AutoFocus-Automaten sind erweiterte Mealy-Zustandsmaschinen, Mealymaschinen mit lokalen Variablen also. Kontrollzustände, dargestellt durch Ovale, werden durch Transitionen (Pfeile) miteinander verbunden. Abb. 4.1 zeigt als Beispiel die Zustandsmaschine der Komponente für die Verwaltung der Sicherheitsumgebungen der WIM (Abschnitt 2.1). Initialzustände sind mit einem schwarzen Punkt gekennzeichnet; alle Zustände sind (im unendlichen Fall Büchi-) Endzustände. Eine Transition besteht aus einem Wächter und einer Zuweisung. Der Wächter liest Werte von denjenigen Eingabeports, die Eingabeports der der Maschine zugeordneten Komponente sind. Für diese Werte können Bedingungen formuliert werden, die außerdem die komponentenlokalen Variablen betreffen können. Wenn die Bedingung zu *true* evaluiert, kann die Transition feuern; es werden dann die in der Zuweisung formulierten Werte auf Ausgabeports geschrieben, lokale Variablen

[1] Das Modell der WIM wurde von Jan Philipps erstellt.

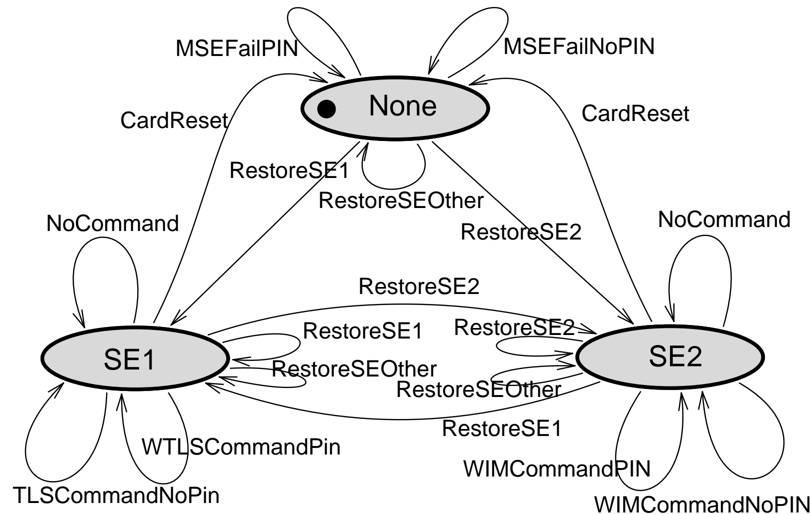


Abbildung 4.1.: Erweiterte Zustandsmaschine für Security Environment

aktualisiert und der Kontrollzustand gemäß dem Transitionspfeil gewechselt. Ein Beispiel ist die Transition *WIMCommandNoPIN* vom Zustand *SE2* zu sich selbst. Sie ist definiert als

Eingaben `c?Cmd; pg?`

Auf Kanal *c* wird ein Kommando erwartet, das an Variable *Cmd* gebunden wird. Auf Kanal *pg* wird kein Signal erwartet, was durch Abwesenheit eines Arguments nach dem Fragezeichen gekennzeichnet wird. Kanalbelegungen können mit Pattern Matching abgefragt werden. Ein Beispiel ist `c?MSERestore(S)` für einen Variablenidentifikator *S*, der dann in der Bedingung abgefragt werden kann. In diesem Fall wird ein Kommando erwartet, dessen Kopfterm *MSERestore* ist.

Bedingung `(isSecurityEnvCommand(Cmd) && not(is_MSERestore(Cmd)))`

Das anliegende Kommando *Cmd* wird durch Aufrufe zweier Funktionen daraufhin überprüft, ob es (1) ein Kommando ist, das sich auf die Sicherheitsumgebungen bezieht und (2) kein Kommando ist, das eine Sicherheitsumgebung setzt – im wesentlichen also Kommandos, die Schlüssel in den Sicherheitstemplates setzen.

Ausgaben `se!SE2(seWimTransition(Cmd,env2,Unverified));`
`r!seWimOutput(Cmd,env2,Unverified)`

Auf den Kanal *se* wird die aktualisierte Sicherheitsumgebung geschrieben. Der Kopfterm *SE2* ist ein Typkonstruktor; *seWimTransition* ist eine Funktion, die hier in Abhängigkeit von *Cmd* bei nicht verifizierter PIN die neue Sicherheitsumgebung berechnet (s.u.). Mithilfe der definierten Funktionen *seWimOutput* wird die Antwort der Karte auf das Kommando *Cmd* berechnet und auf den Kanal *r* geschrieben. Auf nicht angegebene Kanäle wird kein Wert geschrieben.

```
seWimOutput: Command -> WIM_SE -> PINState -> Response;

fun seWimOutput(MSECTSetPrivateKey(fileId, keyRef), s, pg) = H9000
  |  seWimOutput(MSEDSTSetPrivateKey(fileId, keyRef), s, pg) = H9000
  |  seWimOutput(MSEDSTSetPublicKey(newb1,newb2), s, pg) = H9000
  |  seWimOutput(MSECTSetPublicKey(b), s, pg) = H6A80
  |  seWimOutput(MSECCTSetLength(n), s, pg) = H6A80
  |  seWimOutput(MSECCTDeriveMaster(b), s, Unverified) = H6982
  |  seWimOutput(MSECCTDeriveMaster(b), s, Verified) = H6600
  |  seWimOutput(c, s, pg) = NoResponse;
```

Abbildung 4.2.: Funktionsdefinition

Variablenupdate `env2 = seWimTransition(Cmd,env2,Unverified)`

Die auf den Kanal *se* geschriebene Information wird in die Variable *env2* geschrieben. Der Kopfterm *SE2* ist nicht nötig, weil zwei Variablen vorhanden sind, eine für jede Sicherheitsumgebung. Lokale Variablen, deren Wert nicht explizit geändert wird, behalten der Wert vor dem Tick.

Wenn im aktuellen Kontrollzustand mehrere Transitionen feuern können, wird nichtdeterministisch eine ausgewählt. Das Modell der Chipkarte ist deterministisch. Wenn keine Transition feuern kann, verharrt das System im aktuellen Kontroll- und Datenzustand und gibt auf den entsprechenden Ausgabekanälen nichts aus. Das bedeutet, daß das modellierte System auf jeden (korrekt getypten) Stimulus reagieren kann; Systeme sind input-enabled (z.B. Lynch und Tuttle (1989)).

Verhalten: Funktionsdefinitionen Teile des Verhaltens werden üblicherweise in einer getypten funktionalen Sprache erster Stufe mit eager Auswertungssemantik definiert. Das ist dann der Fall, wenn reine Datentransformationen beschrieben werden sollen oder dann, wenn es für das modellierte Teilsystem keine ausgezeichnete Projektion des Zustandsraums gibt, die adäquat mit Kontrollzuständen (oder Modi) modelliert werden kann. Ein Beispiel ist die oben verwendete Funktion *seWimOutput*, die als Parameter ein Kommando, eine Sicherheitsumgebung und einen PIN-Zustand erwartet und eine Antwort der Karte zurückgibt. Ihre Definition ist in Abb. 4.2 angegeben. Die Interpretation des Pattern Matching ist wie üblich; die erste „passende“ Definition wird während des Programmablaufs ausgewählt (Abschnitt4.3.3). Die angegebene Funktion benötigt keine Funktionsaufrufe auf rechten Seiten; ein Beispiel für solche Funktionen wird in Beispiel 24 auf den Seiten 139 ff. angegeben.

Datentypen Die funktionale Sprache stellt einfache Summen- und Produkttypen zur Verfügung. Kommandos sind beispielsweise durch `data Command = PSOEncipher | MSERestore(SERef) | ...` unter Rückgriff auf die Definition `data SERef = SE1Ref | SE2Ref | SEOtherRef` definiert. Polymorphie wird ebenfalls unterstützt. Es ist also beispielsweise möglich, polymorphe Listen durch `data list(T) = nil | c(T,list(T))` zu definieren. Die Verwendung

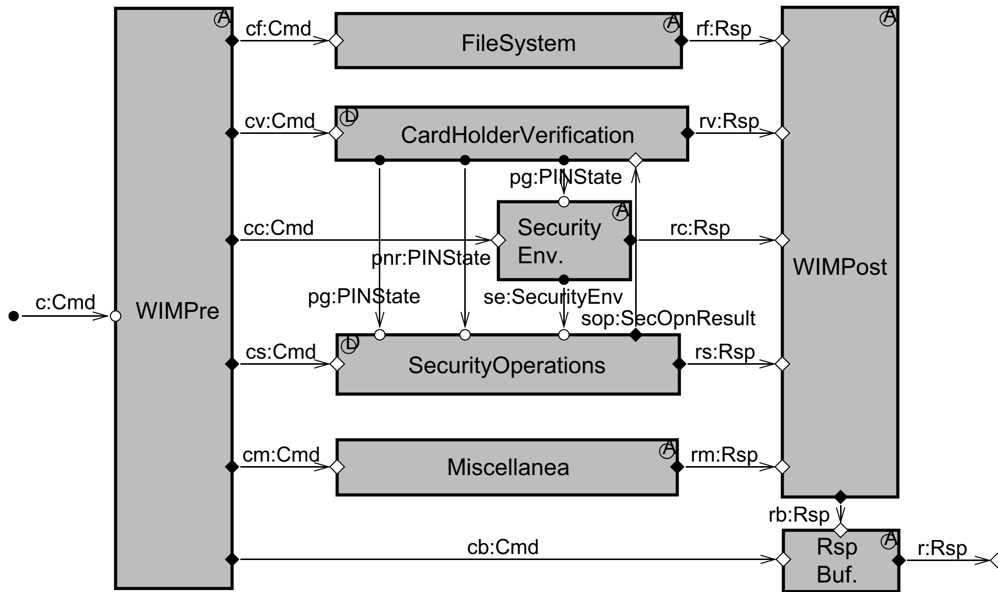


Abbildung 4.3.: Systemstrukturdiagramm der WIM

polymorpher Typen wird im Rahmen der Diskussion der Übersetzung in Abschnitt 4.3.3 behandelt.

4.2.2. Fallstudie

Die Definition der AutoFocus-Beschreibungstechniken ermöglicht nun ein genaueres Verständnis der AutoFocus-Modelle. Um ein Rückblättern zu vermeiden, wird das Strukturdiagramm der obersten Ebene hier nocheinmal aufgeführt (Abb. 4.3). Die Funktionalität der einzelnen Komponenten wurde bereits in Abschnitt 2.1 erläutert. Auf die Angabe der Zustandsmaschinen wird verzichtet, weil sie im wesentlichen aus nur einem Kontrollzustand bestehen; die Funktionalität ist in funktionale Programme ausgelagert.

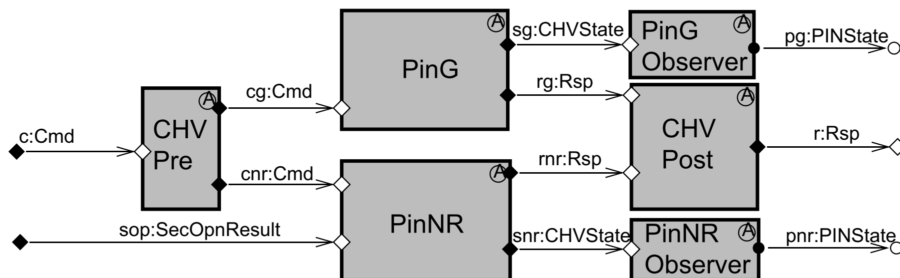


Abbildung 4.4.: Systemstrukturdiagramm für Cardholder Verification

Der Komponente `CardholderVerification`, deren Verfeinerung in Abb. 4.4 wiedergegeben ist, obliegt die Verwaltung der PINs. Für jede der beiden PINs

gibt es eine verantwortliche Komponente mit einer lokalen Variable vom Typen `CHVState = CS(CHVVerif, Pin, Int, Pin, Int)`, wobei `Pin` eine Aufzählung verschiedener symbolischer Namen für PINs und PUKs ist. Die erste Komponente speichert den Zustand der PIN (verifiziert, nicht verifiziert, abgeschaltet), die zweite einen symbolischen Namen der PIN selbst, die dritte den zugehörigen Wiederholzähler, die vierte die zugehörige PUK und die fünfte den zur PUK gehörigen Wiederholzähler. Die Antworten der jeweiligen Komponente werden in der Komponente `CHVPost` zusammengefaßt; die beiden Observerkomponenten dienen ausschließlich dem Zweck der Visualisierung. Darin werden die verschiedenen Kombinationen (z.B. „verifiziert und blockiert" oder „abgeschaltet und blockiert") als Kontrollzustände repräsentiert; die volle Funktionalität ist hingegen in funktionalen Programmen codiert (vgl. S. 47).

Für den Rest des Systems ist allein entscheidend, ob eine PIN verifiziert ist oder nicht. Die Komponente `CardholderVerification` schickt deshalb entsprechende Projektionen der PIN-Zustände an die Komponenten `SecurityEnvironment` und `SecurityOperations`. Die Kanäle sind verzögernd, weil sich andernfalls mit dem Kanal zwischen den Komponenten `SecurityOperations` und `CardholderVerification` ein Zyklus ergäbe. Das `SecurityEnvironment` benötigt den Status der PIN-G zum Beispiel für die Berechnung des Master Secrets im Rahmen des RSA-Handshakes. Die aktuelle Sicherheitsumgebung (im wesentlichen die Belegungen der diversen Templates) werden an die Komponente `SecurityOperations` weitergereicht. Letztere benötigt neben den aktuellen PIN-Zuständen beispielsweise Informationen über gesetzte Schlüssel. Der Kanal zwischen `SecurityOperations` und `CardholderVerification` teilt letzterer mit, ob die letzte Operation eine erfolgreiche NR-Signaturberechnung war. In diesem Fall wird gemäß der Spezifikation der Zustand der PIN-NR von `Verified` auf `Unverified` gesetzt, und offenbar benötigt die `CardholderVerification` diese Information. Die vertikalen Kanäle stellen also den gemeinsamen Speicher der durch sie verbundenen Komponenten da.

Die Funktionalität der anderen Komponenten wurde in Abschnitt 2.1 diskutiert. `FileSystem` übernimmt die rudimentär modellierte Verwaltung des Dateisystems, `SecurityEnvironements` die Verwaltung der Sicherheitsumgebungen, `SecurityOperations` die Berechnung kryptographischer Funktionen, und `Miscellanea` berechnet u.a. Zufallszahlen. Im `ResponseBuffer` werden ggf. Resultate kryptographischer Berechnungen hinterlegt.

4.3. Übersetzung in Constraint-Logik-Programmierung

Für die Testfallgenerierung werden AutoFocus-Modelle in eine Constraint-Logiksprache übersetzt. Da die Logikprogrammierung die Verwendung ungebundener Variablen zuläßt, kann dasselbe übersetzte System ohne Modifikation

- nicht nur zu Simulationszwecken verwendet werden – alle Eingaben an das System werden vollständig instantiiert übergeben und die Ausgaben protokolliert,

- sondern auch zur Generierung solcher Eingaben, indem Eingaben unter-

spezifiziert übergeben werden, die im Verlauf der (dann symbolischen) Ausführung des CLP-Programms gebunden werden. Die Ausgaben werden wiederum protokolliert.

Diese Möglichkeit der doppelten Verwendung des übersetzten Systems ist in der deklarativen Natur von CLP-Sprachen begründet, in der es keine Unterscheidung zwischen Ein- und Ausgaben gibt.[2] Wenn nur Ausgaben spezifiziert werden, findet das CLP-System die dazu passenden Eingaben. Diese Eingaben sind Testdaten für Testfälle, und im letzten Kapitel wurde geschildert, daß Testfallgenerierung im wesentlichen ein Suchproblem ist.

Constraints, d.h. Bedingungen an Variablenbelegungen, dienen dabei (a) der Spezifikation von Testfällen (z.B. „wenn Signal s_1 anliegt, muß im unmittelbar folgenden Schritt Signal s_2 anliegen“ oder „auf einem Kanal werden die Signale s_1 und s_2 ausgeschlossen“) und (b) der Möglichkeit, mit spezifizierten Mengen von Werten anstelle einzelner Werte zu rechnen. Im Rahmen der Logikprogrammierung ist es möglich, durch Ausführung des Programms Eingaben der Art $i(t) = d(X)$ zu einem bestimmten Zeitpunkt t für einen Kanal i, einen Datentypkonstruktor d und eine Variable X unterspezifiziert zu fordern. Wenn X zusätzlich Typinformation oder weitere Einschränkungen beinhalten soll, kann das mit Constraints geschehen: $i(t) = \{d(X) : X \in \mathit{Int} \wedge 4 \leq X \leq 8\}$. Gegenüber einer vollständigen Aufzählung des Suchraums, wie das in explicit-state-Model Checkern wie Spin geschieht, bietet das offenbar Raum für Effizienzgewinne: Anstatt alle Werte für $X \in \{4, 5, 6, 7, 8\}$ auszuprobieren – fünf Kinder des aktuellen Knotens im Suchraum müssen erzeugt werden –, wird symbolisch mit nur einem unterspezifizierten Wert gerechnet. Schließlich wird mit Mengen von Werten nicht nur gerechnet, sondern es werden auch Zustandsmengen gespeichert, um den Suchraum einzuschränken. Auch das geschieht mit Constraints. Der Nachteil gegenüber explicit-state Model Checkern besteht darin, daß Repräsentationen von Zustandsmengen nicht ohne weiteres in einer Hashtabelle gespeichert werden können. Die Überprüfung darauf, ob ein (einzelner) Zustand bereits besucht wurde, kann demnach nicht in konstanter Zeit erfolgen. Die Verwendung von Constraints zur Testfallspezifikation wurde in Kapitel 3 erörtert; Constraints zur Zustandsspeicherung werden in Abschnitt 5.2 diskutiert. Abschnitt 5.3 enthält eine Aufzählung aller für die Testfallgenerierung relevanten Constraints.

4.3.1. Constraint-Logik-Programmierung

Bevor die Übersetzung von AutoFocus-Modellen in eine Constraint-Logiksprache angegeben wird, wird dieses Programmierparadigma kurz vorgestellt. Betrachtet werden zunächst nur sequentielle Hornklausel-basierte Sprachen.

[2] Zur Effizienzsteigerung bieten viele Prologcompiler die Annotation von Formalparametern mit expliziten *Modes* an, die eine Aussage darüber treffen, ob ein Argument stets instantiiert, stets nicht instantiiert oder einmal instantiiert und einmal nicht instantiiert ist.

Logikprogrammierung

Im Rahmen der Logikprogrammierung werden Programme P als Hornklauseln formuliert. Literale sind positive oder negierte Prädikate $p(c_1, \ldots, c_n)$ für $n \geq 0$, deren Argumente aus Variablen $\mathcal{V}_P$ und u.U. Funktionssymbolen $\mathcal{F}_P$ zusammengesetzt sind, d.h. jedes c_i ist ein Term über $\mathcal{F}_P \cup \mathcal{V}_P$. Im folgenden bezeichnet $\mathcal{V}$ die Menge aller Variablen und $\mathcal{V}(t)$ die Variablen des Terms t. Die Mengen der Funktionssymbole $\mathcal{F}_P$ und Prädikatnamen $\mathcal{P}_P$ sind disjunkt. Für jedes Funktions- und Prädikatsymbol existiert eine Stelligkeit, die durch die Anzahl der Argumente charakterisiert wird. Das wird durch p/n oder f/m gekennzeichnet; hier hat p die Stelligkeit n und f die Stelligkeit m. Hornklauseln sind dann universell quantifizierte geschlossene Disjunktionen von Literalen mit höchstens einem positiven Literal:

$$\forall \vec{x} \bullet h \Leftarrow b_1 \wedge \ldots \wedge b_n,$$

wobei $\vec{x}$ alle in $h, b_1, \ldots, b_n$ auftretenden Variablen sind. Der Allquantor wird üblicherweise als implizit vorausgesetzt. Hornklauseln ohne positives Literal heißen Anfragen, solche ohne negative Literale heißen Fakten. Eine Hornklausel ohne positive und negative Klauseln heißt die leere Klausel, die als Widerspruch interpretiert wird. Alle anderen Hornklauseln heißen Regeln. Ein Programm ist eine endliche Menge (Konjunktion) von Fakten und Regeln. Programme P werden nun ausgeführt, indem Anfragen an sie gestellt werden. Diese Anfragen Q werden positiv formuliert, d.h. $Q = \exists \vec{x} \bullet \bigwedge_{i=1}^{n} q_i$ und nicht $Q' = \forall \vec{x} \bullet \bigvee_{i=1}^{n} \neg q_i$. Überprüft wird dann, ob Q logische Konsequenz von P ist, ob also $P \models Q$ und damit $P \models \exists \vec{x} \bullet \bigwedge_{i=1}^{n} q_i$ gilt. Man zeigt leicht, daß $P \models Q$ genau dann gilt, wenn $P \wedge \neg Q$ unerfüllbar ist. Einsetzen liefert dann die Überprüfung von $P \wedge \forall \vec{x} \bullet \bigvee_{i=1}^{n} \neg q_i$ auf Unerfüllbarkeit. Ein Beispiel ist gegeben durch das Programm

$$P = \big(e(0) \Leftarrow true\big) \wedge \big(\forall x \bullet e(s(s(x))) \Leftarrow e(x)\big),$$

das intuitiv die Menge gerader Peano-Zahlen charakterisiert, und die Anfrage $Q = e(0)$.

$$P \wedge \neg Q = \big(e(0) \Leftarrow true\big) \wedge \big(\forall x \bullet e(s(s(x))) \Leftarrow e(x)\big) \wedge \big(\neg e(0)\big)$$

ist offenbar unerfüllbar, d.h. $P \models e(0)$ gilt.

Semantik Interpretiert werden Logikprogramme P über dem Herbranduniversum U_P, das die Menge aller Terme enthält, die nur Funktionssymbole und keine Variablen enthalten. Die Herbrandbasis B_P entsteht aus dem Herbranduniversum, indem alle Argumente aller Prädikate mit allen möglichen Werten aus U_P belegt werden, d.h.

$$B_P = \bigcup_{p/n \in \mathcal{P}_P} \{p(c_1, \ldots, c_n) : c_i \in U_P\}.$$

Funktionssymbole werden termalgebraisch durch sich selbst interpretiert. Dann kann die Interpretation von Prädikaten mit Teilmengen von B_P identifiziert

werden. Solche Interpretationen heißen Herbrandinterpretationen; Modelle, die Herbrandinterpretationen sind, heißen Herbrandmodelle.

Jedes Logikprogramm P hat offenbar B_P als Modell. Da der Durchschnitt zweier Herbrandmodelle wiederum ein Herbrandmodell ist, ist auch der Durchschnitt aller Herbrandmodelle $M_P = \bigcap\{M : M \textit{ ist Herbrandmodell von } P\}$ ein Herbrandmodell. Dieses minimale Herbrandmodell kennzeichnet genau die Menge aller logischer Konsequenzen von P, d.h. $M_P = \{A \in B_P : P \models A\}$. Eine Fixpunktcharakterisierung ergibt sich mithilfe des Operators $T_P : 2^{B_P} \to 2^{B_P}$:

$$T_P(I) = \Big\{A \in B_P : g_P\big(A \Leftarrow \bigwedge_{i=1}^{n} A_i\big) \textit{ und } \bigcup_{i=1}^{n}\{A_i\} \subseteq I\Big\}, \tag{4.1}$$

wobei I eine Herbrandinterpretation ist und $g_P(x)$ dann wahr ist, wenn x Grundinstanz einer Klausel in P ist, d.h. wenn alle Variablen in x durch Terme aus U_P belegt sind. T_P ist monoton und stetig, und der kleinste Fixpunkt von T_P ist genau M_P. Auf diese fixpunkttheoretische Charakterisierung wird im Abschnitt 5.2 zurückgegriffen, wenn Zustandsspeicherstrategien diskutiert und die Testfallgenerierung mit CLP und Model Checking verglichen werden. Die Berechnung der Semantik eines Programms mit einem T_P-artigen Operator wird in der Datenbankliteratur als bottom-up bezeichnet.

Resolution Neben modell- und fixpunkttheoretischer Semantik gibt es noch einen operationellen Ansatz zur Charakterisierung der Bedeutung eines Programms. Im Gegensatz zum obigen bottom-up-Ansatz wird dieser Ansatz als top-down bezeichnet. Die Erläuterung der operationellen Semantik – der Resolution – ist hier notwendig, weil das Beschneiden des Resolutionsbaums durch den sog. Cut später notwendig sein wird. Von Interesse sind dabei Belegungen von Variablen in der Anfrage, unter denen die Anfrage Konsequenz des Programms ist. Für eine (positive) Anfrage $Q = \exists\vec{x}\bullet\bigwedge_{i=1}^{n} q_i$ heißt eine Substitution[3] σ der in Q vorkommenden Variablen Antwort. Die Menge der Substitutionen wird im folgenden mit *Subst* bezeichnet. Eine Antwort σ wird korrekt genannt, wenn $P \models \exists\vec{x}\bullet\sigma(\bigwedge_{i=1}^{n} q_i)$ gilt. Mit dem Verfahren der Resolution können nun alle allgemeinsten korrekten Antworten berechnet werden. Man kann zeigen, daß die (SLD-)Resolution korrekt und vollständig bzgl. der berechneten Antworten ist.

Für die Resolution wird ein Literal ℓ_q der Anfrage $Q = \bigwedge_{i=1}^{n} \ell_i$ ausgewählt und mit dem einzigen positiven Literal L_j aller Programmklauseln $p_j : L_j \Leftarrow \bigwedge_{k=1}^{m} R_{jk}$ unifiziert. Wenn der allgemeinste Unifikator[4]μ existiert, wird Q durch $\mu(\bigwedge_{i\neq q} \ell_i \wedge \bigwedge_{i=1}^{m} R_{ji})$ ersetzt und das Verfahren iteriert. Dabei werden in jedem Schritt frische Variablen in den Programmklauseln angenommen. Wenn μ für die Regel p_j nicht existiert, wird eine weitere Regel ausprobiert. Wenn

[3]Eine *Substitution* ist eine idempotente Abbildung von Termen auf Termen, die zumeist als kanonische Erweiterung einer Abbildung von Variablen auf Terme verstanden wird.

[4]Ein *Unifikator* ist eine Substitution σ, die zwei Terme t_1, t_2 identifiziert, d.h. $\sigma(t_1) = \sigma(t_2)$. Eine Substitution σ heißt allgemeiner als eine Substitution ϑ, falls es eine Substitution η gibt, so daß für alle Terme t gilt, daß $\eta(\sigma(t)) = \vartheta(t)$. Der *allgemeinste Unifikator* oder mgu (most general unifier) zweier Terme ist ein Unifikator dieser Terme, so daß keine allgemeinerer Unifikator existiert. Wenn ein mgu existiert, ist er eindeutig bis auf α-Konversion.

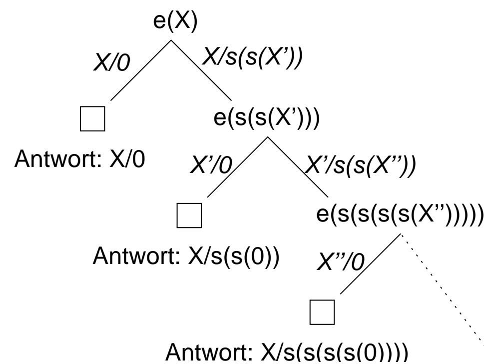


Abbildung 4.5.: Resolution

μ für kein p_j existiert, erfolgt Backtracking zu früheren Berechnungsschritten. Wenn schließlich Q soweit ersetzt wurde, daß es aus der leeren Klausel und einer Substitution σ – die Komposition aller allgemeinster Unifikatoren der vorherigen Schritte – besteht, wird σ als Antwortsubstitution ausgegeben, und Backtracking zu vorherigen Schritten erfolgt. Das Verfahren endet, wenn kein Backtracking mehr möglich ist. Abbildung 4.5 zeigt ein Beispiel für das o.a. Programm zur Charakterisierung gerader Peano-Zahlen und der Anfrage $e(X)$. Substitutionen sind kursiv angegeben. Jeweils nach links wird die Regel $e(0) \Leftarrow true$ ausgewählt; jeweils nach rechts die andere.

Die Resolution ist nicht nur ein Verfahren zur Berechnung von Antworten, sondern führt auch zu einer operationellen Interpretation von Logikprogrammen. Diese interpretiert Prädikate als Funktionen oder Prozeduren in imperativen Programmiersprachen; wenn der Kopf (das positive Literal) eines Prädikats „aufgerufen“ wird, heißt das, daß alle (negativen) Literale des Rumpfes hintereinander „ausgeführt“ werden müssen.

Die Behandlung unendlicher Zweige, wie sie im Beispiel auftritt, liegt in der Verantwortung des Programmierers. Logikprogrammiersprachen wie Prolog stellen dazu ein nicht-logisches Konstrukt zur Verfügung, den Cut, mit dem der Resolutionsbaum explizit beschnitten werden kann. Die Verwendung des Cuts muß in Abschnitt 4.3.3 diskutiert werden, weshalb er hier vorgestellt wird. Dazu setzt der Programmierer in seinen Code eine entsprechende Direktive, die dem System kennzeichnet, daß der Suchbaum an dieser Stelle insofern beschnitten wird, als keine alternativen Prädikataufrufe für die sich vor dem Cut befindlichen Prädikate ausprobiert werden.

Constraint-Logikprogrammierung

Eine fundamentale Einschränkung der Logikprogrammierung ist die Einschränkung auf Herbrandinterpretationen. Ganze oder rationale Zahlen anstelle von Codierungen durch beispielsweise Peanoterme können nicht einfach in die Theorie integriert werden. Eine Erweiterung des Interpretationsbegriffs auf beliebige Domains erfolgt im Rahmen der Constraint-Logikprogrammierung (Jaffar und Lassez, 1987; Höhfeld und Smolka, 1988; Jaffar und Maher, 1994). CLP-

Programme sind Erweiterungen von LP-Programmen. Die zentrale Idee besteht darin, Substitutionen während der Resolution durch allgemeine Constraints – Gleichungen, Ungleichungen oder beliebige andere Relationen – zu ersetzen. Die Menge der Prädikatsymbole $\mathcal{P}_P$ wird partitioniert in Symbole, die durch einen Constraintlöser behandelt werden (z.B. arithmetische Gleichheit und arithmetische Operationen) und solche, die durch das Constraint-Logikprogramm definiert werden. Constraints können im Rumpf von Klauseln auftreten. Constraintsymbole werden über festzulegenden Bereichen $\mathcal{X}$ definiert.[5] Viele Resultate bzgl. modelltheoretischer, fixpunkttheoretischer und operationeller Semantik lassen sich von CLP-Kalkülen auf LP übertragen. Gabbrielli u. a. (1995) untersuchen CLP-Semantiken, die u.a. im Gegensatz zum oben erwähnten T_P-Operator partielle Antworten berücksichtigen.

Die operationelle Semantik von CLP beruht wie die der Logikprogrammierung auf der Resolution. Zusätzlich gibt es einen Speicher für Constraints, den sog. Constraintstore. In jedem Resolutionsschritt wird mit einem Constraintlöser zusätzlich zur tatsächlichen Resolution überprüft, ob die im Verlauf der Programmauswertung erzeugten Constraints erfüllbar sind. Eine typisches Muster bei der Implementierung von Logikprogrammen ist das Prinzip des *generate-and-test*. Dazu wird zunächst die Grundmenge aller ein Programm wahrmachenden Belegungen erzeugt. Sobald eine neue Belegung gefunden wurde, wird getestet, ob sie ein bestimmtes Kriterium erfüllt. Es folgt die Berechnung einer weiteren Belegung und der erneute Test usw. CLP modifiziert diesen Ansatz insofern, als das Testkriterium als Constraint formuliert und in jedem Resolutionsschritt überprüft wird, ob eine wahrmachende Belegung überhaupt noch gefunden werden kann (*constrain-and-generate*). Der Tradeoff besteht offenbar zwischen den durch das frühzeitige Beschneiden des Suchbaums erlangten Vorteilen und dem Nachteil, daß die Constraints in jedem Resolutionsschritt überprüft werden müssen. Eine leicht verständliche Einführung in die Theorie der Constraint-Programmierung findet sich im Buch von Frühwirth und Abdennadher (1997).

Constraintlöser können vom CLP-System zur Verfügung gestellt werden oder durch eigene Sprachen definiert werden. Ein Beispiel für eine solche Sprache sind die Constraint Handling Rules (Frühwirth, 1995, 1998). Die Idee ist, die Gleichungen, Ungleichungen oder allgemeinen Relationen durch eine eigene Programmiersprache zu codieren. Im folgenden werden diverse Ungleichheiten in einer Form notiert, die direkt in Constraint Handling Rules übersetzt werden kann.

4.3.2. Übersetzung von EFSMs

Die Übersetzung von AutoFocus-EFSMs in CLP-Sprachen gestaltet sich wegen der zeitsynchronen Kommunikation vergleichsweise einfach. Die Idee ist, zunächst atomare Komponenten, d.h. mit EFSMs versehene Komponenten der Blätter der Systemhierarchie zu übersetzen und dann rekursiv Prädikate zu erzeugen, die Berechnungen der tieferliegenden Komponenten anstoßen und die

[5] Höhfeld und Smolka (1988) gestatten die Verknüpfung verschiedener Domains; Jaffar und Lassez (1987) sind auf eine Domäne beschränkt, z.B. die reellen oder die rationalen Zahlen.

Kommunikation realisieren. Die zeitsynchrone Kommunikation erlaubt die Interpretation von Kanälen als lokale Variablen derjenigen Komponente, in die die kommunizierenden Komponenten eingebettet sind. Eine ausführliche Diskussion der Übersetzung, die die Problematik der Quantifizierung ungebundener Variablen allerdings vernachlässigt, findet sich bei Lötzbeyer und Pretschner (2000a). Auf eine Fallunterscheidung nach An- oder Abwesenheit von lokalen Variablen oder Eingabe-/Ausgabekanälen wird hier aus Gründen der Einfachheit verzichtet.

Eine atomare Komponente c wird für jeden einzelnen Transitionspfeil durch jeweils ein Prädikat $step^c$ codiert:

$$step^c(\vec{\sigma}_{src}, \vec{\iota}, \vec{o}, \vec{\sigma}_{dst}) \Leftarrow g \wedge assgmt(\vec{o}, \vec{\sigma}_{dst}).$$

Bei gegebenem Eingabevektor $\vec{\iota}$, der Belegung aller Eingabekanäle zu einem bestimmten Zeitpunkt, kann eine Transition vom Kontroll- und Datenzustandstupel σ_{src} in den Zustand σ_{dst} durchgeführt werden, wenn der Wächter g zu *true* ausgewertet wird. Dabei wird der Vektor $\vec{o}$ auf die Ausgabekanäle geschrieben und lokale Variablen sowie der Kontrollzustand aktualisiert. Das geschieht vermöge der Zuweisung *assgmt*. Die Übersetzung von Wächter und Zuweisung wird im folgenden Abschnitt diskutiert.

Die Übersetzung einer nicht-atomaren Komponente K, die sich aus den Komponenten $k_1, \ldots, k_n$ zusammensetzt, erfolgt dann rekursiv in Form eines Prädikats $step^K$. Pro zusammengesetzter Komponente ist genau ein Prädikat nötig und nicht eins pro Transition wie im Fall atomarer Komponenten:

$$step^K(\vec{\sigma}^K_{src}, \vec{\iota}^K, \vec{o}^K, \vec{\sigma}^K_{dst}) \Leftarrow \bigwedge_{j=1}^{n} step^{k_j}(\vec{\sigma}^{k_j}_{src}, \vec{\iota}^{k_j}, \vec{o}^{k_j}, \vec{\sigma}^{k_j}_{dst}).$$

Die Übersetzung einer solchen zusammengesetzten Komponente ist also im wesentlichen durch die Konjunktion der Übersetzung ihrer Subkomponenten definiert, was wegen der einfachen zeitsynchronen Semantik von AutoFocus möglich ist. $\vec{\iota}^K$ und $\vec{o}^K$ bezeichnen die Ein- und Ausgabekanäle von K, $\vec{\sigma}^K_{src}$ und $\vec{\sigma}^K_{dst}$ den Start- und Endzustand der Transition, der jeweils als Produkt der darunterliegenden Komponenten definiert ist. Sie enthalten ebenfalls die Belegungen interner Kanäle, die als lokale Variablen von K interpretiert werden können. Die Reihenfolge der $step^{k_j}$ ist bei Abwesenheit unendlicher Resolutionsbäume bzgl. der Semantik irrelevant; für die Effizienzsteigerung der Suche ist sie durchaus von Belang, wie in Abschnitt 5.1 ausgeführt wird.

Nicht-atomare Komponenten besitzen keine lokalen Variablen und sind nicht mit EFSMs ausgestattet. Ihr Zustand ist das Produkt der Zustände aller ihrer Subkomponenten und wegen des rekursiven Charakters der Übersetzung damit das Produkt der Zustände aller in der Komponentenhierarchie unter ihr liegenden atomaren Komponenten incl. der Belegungen der Kanäle. Diese müssen allerdings nicht explizit berechnet werden: Prädikate für nicht-atomare Komponenten enthalten als Formalparameter ausschließlich Variablen, die durch Aufrufe an die die Unterkomponenten codierenden Prädikate gebunden werden.

Da Variablen unterschiedliche Werte vor und nach einem Tick besitzen können, müssen sie durch jeweils zwei Werte codiert werden. Bei der Überset-

```
stepSecurityOperations(
   (SSOPre, SSOPost, SEncipher, SSEDist, SPINDist),   % Startzustand
   (Se1, C1, R1, Pg1, Sep),                           % Kommunikation
   (C, Se, Pnr, Pg), (R, Sop),                        % Ein-/Ausgaben
   (DSOPre, DSOPost, DEncipher, DSEDist, DPINDist)):- % Endzustand
      stepPINDist(SPINDist, (Pg, Pnr), (Pg1), DPINDist),
      stepSEDist(SSEDist, Se, (Se1,Sep), DSEDist),
      stepSOPre(SSOPre, C, C1, DSOPre),
      stepEncipher(SEncipher, (Se1, C1, Pg1), (R1), DEncipher),
      stepSOPost(SSOPost, (R1, Sep), (R, Sop), DSOPost).

stepPINDist(sMain, ((msg, Pg), no_msg), (msg, Pg), sMain).
stepSEDist(sMain, (msg, Se), ((msg, Se), (msg, Se)), sMain).
stepSOPre(sMain, (msg, cPSOEncipher), (msg, cPSOEncipher), sMain).
stepEncipher(sEncipher, ((msg, Env), (msg, cPSOEncipher), (msg, _)),
             (msg, Rsp), sEncipher):-
   computeGuard(encipherOutput(Env,cVerified), Rsp).
stepSOPost(sMain, ((msg, Rsp), _), ((msg, Rsp), no_msg), sMain).
```

Abbildung 4.6.: Partielle Übersetzung von `SecurityOperations`

zung der internen Kommunikation muß zwischen immediaten und verzögernden Ports unterschieden werden. Verzögernde Ports werden direkt in lokale Variablen der übergeordneten Komponente übersetzt und sind wiederum durch zwei Werte, vor und nach dem Tick, zu codieren. Für Kanäle, die an immediate Ausgabeports angeschlossen sind, ist das nicht notwendig. Kanäle zwischen Ports werden dadurch implementiert, daß die entsprechenden Ein- und Ausgabevariablen in der Liste der Formalparameter denselben Bezeichner tragen.

Als Beispiel sei eine vereinfachte Variante der Komponente `SecurityOperations` angegeben, wobei die Komponenten `Decipher`, `ComputeCryptoCS`, `Decipher`, `VerifyDigSig` und `ComputeDigSig` ausgelassen werden. Keine Unterkomponente besitzt lokale Variablen, und alle Ports sind immediat. Für alle atomaren Unterkomponenten wird nur jeweils eine Transition angegeben. Damit ergibt sich als Übersetzung die in Abb. 4.6 angegebene. Variablen beginnen mit Großbuchstaben, `no_msg` signalisiert die Abwesenheit eines Signals und `(msg,M)` die Anwesenheit eines Signals `M`. Anonyme Variablen, d.h. Variablen, die nur einmal referenziert werden, werden als _ notiert.

Die Übersetzung des Wächters für die angegebene Transition der Komponente `Encipher` wird weiter unten erläutert.

Wächter g von Transitionen sind von der Form

$$g \equiv match(\vec{\iota}_{form}, \vec{\iota}_{act}) \wedge \mu_{\vec{\iota}_{form},\vec{\iota}_{act}}(cond)$$

für ein Prädikat *match*, das die in der Transition angegebenen Formalparameter für die Eingabe $\vec{\iota}_{form}$ mit den Aktualparametern $\vec{\iota}_{act}$ während der Ausführung vergleicht. Aus Gründen der Einfachheit wird angenommen, daß alle diejenigen Typen, die Kanäle typisieren, um ein ausgezeichnetes Element *no_msg* geliftet sind, das die Abwesenheit eines Signals kennzeichnet. In den

Transitionen wird diese Situation durch $c?$ für einen Kanal c ausgedrückt und als $c?no_msg$ gelesen. Es kann zunächst angenommen werden, daß Aktualparameter Grundinstanzen der gelifteten Typdefinitionen sind, d.h. $\mathcal{V}(\vec{\iota}_{act}) = \emptyset$. Unter diesen Voraussetzungen entscheidet $match(t_1, t_2)$ die Unifizierbarkeit zweier Terme t_1 und t_2 mit $\mathcal{V}(t_2) = \emptyset$.

$\mu_{\vec{\iota}_{form}, \vec{\iota}_{act}}$ bezeichnet die Matchingsubstitution von $\vec{\iota}_{form}$ und $\vec{\iota}_{act}$, die wiederum wegen der Variablenfreiheit des zweiten Arguments der allgemeinste Unifikator (mgu) beider Terme ist:

$$g \equiv match(\vec{\iota}_{form}, \vec{\iota}_{act}) \wedge mgu_{\vec{\iota}_{form}, \vec{\iota}_{act}}(cond)$$

Dieser mgu wird auf die Bedingung *cond* angewendet und das Resultat ausgewertet. *cond* enthält neben einer Teilmenge von $\mathcal{V}(\vec{\iota}_{form})$ nur komponentenlokale Variablen, die während der Ausführung zunächst erneut als Grundinstanzen angenommen werden können. In Abb. 4.6 tauchen die Matching-Substitutionen nicht explizit auf, weil sie zur Laufzeit vom CLP-System berechnet werden.

Vervollständigung mit idle-Transitionen

Eine Komponente verharrt in ihrem Zustand, wenn kein Wächter der $n \geq 1$ aus diesem Zustand herausführenden Transitionen zu *true* ausgewertet werden kann. Subskripte i beziehen sich im folgenden auf die i-te Transition. $\vec{\iota}_{form(i)}$ bezeichnet die Formalparameter der i-ten Transition und $\vec{\iota}_{act}$ die Aktualparameter zu einem gegebenen Zeitpunkt. Idling erfolgt demnach genau dann, wenn

$$\bigwedge_{i=1}^{n} \neg g_i \quad bzw. \quad \bigwedge_{i=1}^{n} \neg match(\vec{\iota}_{form(i)}, \vec{\iota}_{act}) \vee \neg \mu_{\vec{\iota}_{form(i)}, \vec{\iota}_{act}}(cond_i)$$

gilt. Diese Form der Übersetzung ist adäquat, wenn Modelle simuliert werden, die Eingaben an das System also vollständig instantiiert zur Verfügung stehen.

Die Situation wird interessanter, wenn zum Zweck der symbolischen Ausführung partielle Datenstrukturen zugelassen werden. In diesem Fall ist der Matchingoperator ein Unifikationsoperator, weil er Variablen auf beiden Seiten bindet und nicht deshalb, weil für Grundterme auf der rechten Seite Matching und Unifikation äquivalent sind. Variablen werden nicht mehr nur im Zuweisungsteil einer Transition gebunden, sondern auch während der Abfrage von Nachrichten auf Kanälen. Die Auswertung eines Wächters liefert dann nicht nur das Resultat *true* oder *false*, sondern zusätzlich Belegungen freier Variablen, die per Backtracking aufgezählt werden.

Da partielle Datenstrukturen später beliebig instantiiert werden können, ist es sinnvoll, die allgemeinsten Belegungen für diese Variablen zu finden. Eine idle-Transition schaltet genau dann, wenn

$$\bigwedge_{i=1}^{n} \forall \sigma \in Subst \bullet \sigma(\vec{\iota}_{form(i)}) = \sigma(\vec{\iota}_{act}) \Rightarrow \neg \sigma(cond_i).$$

PSfrag replacements

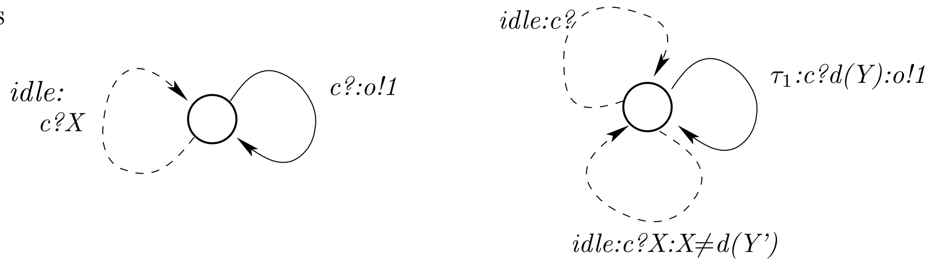


Abbildung 4.7.: Idle-Transitionen

Da allgemeinste Unifikatoren existieren, wenn ein Unifikator existiert, ist das äquivalent zu der Bedingung

$$\bigwedge_{i=1}^{n} unif(\vec{\iota}_{form(i)}, \vec{\iota}_{act}) \Rightarrow \neg mgu_{\vec{\iota}_{form(i)},\vec{\iota}_{act}}(cond_i),$$

was den Allquantor eliminiert. $unif(t_1, t_2)$ entscheidet, ob t_1 und t_2 unifizierbar sind. Da o.B.d.A. $\forall i \bullet \mathcal{V}(\vec{\iota}_{form(i)}) \cap \mathcal{V}(\vec{\iota}_{act}) = \emptyset$ gilt, gibt es keine zirkulären Gleichungen.

Für die Implementierung der Negation in CLP kann prinzipiell auf die übliche Form der negation-as-finite-failure (Lloyd, 1993) zurückgegriffen werden. Das erfordert eine Annotation von Variablen um Typen, wie das Beispiel der Ungleichheit demonstriert: $X \neq 5$ soll etwa nur Instantiierungen von X aus der Domäne der ganzen Zahlen zulassen. Aufzählen aller typkorrekten Instanzen auf linker und rechter Seite der Ungleichung mit Backtracking führt dann zum erwünschten Verhalten. Für Typen mit großer Kardinalität, beispielsweise die ganzen Zahlen, ist das nicht effizient realisierbar. Die Idee ist deshalb, negierte Prädikate zu verzögern, d.h. solange zu warten, bis Variablen gebunden sind und die Ungleichheit ohne Aufzählen aller möglicher Werte entschieden werden kann.

Als Beispiel sei eine einzelne atomare Komponente betrachtet. Sie besitze einen Inputkanal c und einen Outputkanal o. Die Zustandsmaschine besteht aus einem Zustand mit einer einzigen Transition $\tau_1 : c?no_msg \rightarrow o!1$ und der entsprechenden idle-Transition

$$\tau_{idle} : c?X \wedge X \neq no_msg \rightarrow o!no_msg$$

(Abb. 4.7, links). Gemäß der obigen Definition (Gleichheit als Unifikationsgleichheit) kann bei negation-as-failure τ_{idle} niemals schalten. Für den zum Kanal c gehörigen Formalparameter C ergibt sich bei der Auswertung, wenn eine ungebundene Variable X anliegt,

$$\neg(\{C \mapsto X\}C = no_msg) \text{ oder } X \neq no_msg,$$

was bei der Interpretation von $\neq$ als Nicht-Unifizierbarkeit wegen des impliziten Allquantors offenbar unerfüllbar ist. Konsequenz ist, daß in einer solchen Situation die Auswertung der Ungleichheit verzögert werden muß. Die Definition

eines Constraint-Handlers für eine implizit existentiell quantifizierte symmetrische Ungleichheit $\neq^{sym}$ löst das Problem:

$$\ell \neq^{sym} r = \begin{cases} \mathit{false}: & \ell, r \text{ identische Konstanten} \\ \mathit{true}: & \ell, r \notin \mathcal{V}, hd(\ell) \neq^{sym} hd(r) \\ \bigvee_{i=1}^{n} a_i \neq^{sym} b_i: & \ell = h(a_1, \ldots, a_n), r = h(b_1, \ldots, b_n), \end{cases} \tag{4.2}$$

wobei $hd(t)$ das Kopfsymbol eines nichtvariablen Terms extrahiert. Wenn eine der beiden Seiten eine Variable ist, wird die Auswertung der Ungleichheit verzögert, bis die Variable gebunden wird. Die Definition stellt also ein Termersetzungssystem dar. Im obigen Beispiel kann τ_{idle} feuern; der Constraint $X \neq^{sym} no_msg$ bleibt solange bestehen, bis X z.B. durch eine zugeschaltete Komponente oder während der Instantiierung von Testfällen gebunden wird oder Backtracking erfolgt.

Diese Definition der Ungleichheit zeitigt allerdings eine unerwünschte Konsequenz. Wiederum sei eine einzelne Komponente mit einer einzelnen Transition $\tau_1 : c?d(Y) \to o!1$ betrachtet. Die idle-Transition ist dann trivialerweise beschrieben durch

$$\tau_{idle} : c?no_msg \vee c?X \wedge X \neq d(Y') \to o!no_msg$$

(Abb. 4.7 rechts; die Disjunktion ist auf zwei Transitionen verteilt). Wenn die Umgebung nun den Wert $d(1)$ auf den Kanal i schreibt, dann kann τ_1 schalten. Bei der Evaluierung von τ_{idle} ergibt sich als mgu die Substitution $\{X \mapsto d(1), Y' \mapsto 1\}$, und es entsteht der Constraint $d(1) \neq^{sym} d(Y')$ bzw. $1 \neq^{sym} Y'$. Die idle-Transition kann also ebenfalls bei Anlegen von $d(1)$ feuern, was inkorrekt ist.

Y' ist eine transitionslokale Pattern-Matching-Variable, die im weiteren Verlauf der Berechnung niemals gebunden werden wird. Das Problem mit $\neq^{sym}$ ist, daß diese transitionslokale Variable nur ein einziges Mal auftaucht und deshalb irrelevant ist – die wesentliche Information liegt in dem Term direkt über ihr. Deshalb wird für den Pattern-Matching-Teil von idle-Transitionen eine weitere Ungleichheit $\neq^{idl}$ definiert, die genau diesen Fall berücksichtigt. Es bezeichne $\mathcal{W}(t)$ die Menge aller transitionslokaler Variablen, die genau einmal in der Transition auftauchen und durch Zugriff auf einen Kanal, d.h. den ?-Operator, gebunden werden. $\neq^{idl}$ ist dann definiert als

$$\ell \neq^{idl} r = \begin{cases} \mathit{false}: & \ell, r \text{ identische Konstanten} \\ \mathit{true}: & \ell, r \notin \mathcal{V}, hd(\ell) \not\equiv hd(r) \\ \bigvee_{a_i, b_i \notin \mathcal{W}(t)} a_i \neq^{idl} b_i: & \ell = h(a_1, \ldots, a_n), r = h(b_1, \ldots, b_n), \\ & \bigcup_{i=1}^{n} \{a_i, b_i\} \not\subseteq \mathcal{W}(t) \\ \mathit{true}: & \ell \in \mathcal{W}(t) \text{ oder } r \in \mathcal{W}(t) \text{ oder} \\ & \ell = h(a_1, \ldots, a_n), r = h(b_1, \ldots, b_n) \\ & \text{und } \bigcup_{i=1}^{n} \{a_i, b_i\} \subseteq \mathcal{W}(t). \end{cases} \tag{4.3}$$

Die Parametrierung mit t erfolgt während der Übersetzung. Eine ähnliche weitere Ungleichheit für das Pattern Matching in Funktionsdefinitionen wird

```
step(sMain,no_msg,(msg,1),sMain).
step(sMain,(msg,_),no_msg,sMain).

step(sMain,(msg,d(_)),(msg,1),sMain).
step(sMain,X,no_msg,sMain):- idle_ineq([],X,no_msg)
                             ; idle_ineq([Y],X,d(Y)).
```

Abbildung 4.8.: Übersetzung: Idle-Transitionen

weiter unten angegeben. Erneut stellt die Definition ein Ersetzungssystem dar: Wenn ℓ eine Variable mit $\ell \notin \mathcal{W}(t)$ (analog für r) ist, wird die Ungleichheit verzögert.

Die Übersetzung der Beispiele aus Abb. 4.7 ist in Abb. 4.8 angegeben. Die ersten zwei Prädikate codieren das linke System, die anderen zwei das rechte. In beiden Fällen wird davon ausgegangen, daß der Name des Kontrollzustands `sMain` ist. Der erste Parameter ist der Quell-Kontrollzustand, der zweite die Eingabe, der dritte die Ausgabe, der vierte der Ziel-Kontrollzustand. Die zweite und die vierte Zeile stellen die idle-Transitionen dar; `idle_ineq` ist der Name des Prädikats, das $\neq^{idl}$ implementiert. Der erste Parameter ist eine Liste der irrelevanten transitionslokalen Variablen, der zweite und dritte sind das linke bzw. rechte Argument von $\neq^{idl}$.

Fokus dieser Beobachtungen war die Ungleichheit. Die Übertragung auf einen allgemeinen Negationsoperator für beliebige Funktionen eröffnet wiederum zwei prinzipielle Möglichkeiten: Negationen können verzögert werden oder, da es sich um Boolesche Funktionen handelt, einfach durch Inversion des Resultatwerts implementiert werden. Beide Vorgehensweisen haben Vor- und Nachteile; der für diese Arbeit erstellte Testfallgenerator verzögert nur Ungleichheiten und wertet andere Negationen direkt aus.

- Alle Negationen (oder allgemein Funktionsaufrufe) zu verzögern ist dann ineffizient, wenn das Resultat einer Berechnung mit wenigen Backtrackschritten zu erzielen ist. Die Anzahl verzögerter Ziele hat Konsequenzen für die Effizienz, und die Tatsache, daß nicht-ausführbare Pfade beschritten wurden, wird erst sehr spät erkannt. Dieser Ansatz wird von Marre und Arnould (2000) verfolgt, die von Zeit zu Zeit heuristisch das Binden von Variablen erzwingen, um unerfüllbare Pfade zu identifizieren und die Anzahl verzögerter Aufrufe nicht zu groß werden zu lassen.

  Der Vorteil dieses Vorgehens liegt darin, daß mit u.U. sehr großen Mengen von Werten gleichzeitig gerechnet werden kann, wie das Beispiel der Ungleichheit zeigt.

- Negation in Funktionsaufrufen durch Inversion des Resultats ist dann ineffizient, wenn die Anzahl der Lösungen für einen negierten Funktionsaufruf sehr groß wird, wie das z.B. für die Gleichheit über entsprechend mächtigen Typen wie den ganzen Zahlen der Fall ist.

  Dieses Vorgehen ist insofern vorteilhaft, als unerfüllbare Pfade schneller erkannt werden können.

$$
\begin{array}{rrcl}
 & f(\vec{p}_1) & = & rhs_1 \\
| & f(\vec{p}_2) & = & rhs_2 \\
 & & \ldots & \\
| & f(\vec{p}_n) & = & rhs_n
\end{array}
$$

Abbildung 4.9.: Funktionsdefinitionen

4.3.3. Übersetzung von Funktionen

Wächter und Zuweisungen werden mit einer funktionalen Sprache spezifiziert, die eine eager Auswertungssemantik besitzt. Die Übersetzung nach CLP basiert im wesentlichen auf der Auflösung verschachtelter Funktionsaufrufe in flache Prädikate.

Funktionsdefinitionen sind von der in Abb. 4.9 angegebenen Form, wobei die $\vec{p}_i$ Tupel von Parametern darstellen, und die rechten Seiten rhs_i beliebige, auch rekursive, Funktionsaufrufe beinhalten können. Dabei ist Linkslinearität der Regeln gefordert, d.h.

$$\forall i \leq n \bullet \mathcal{V}(rhs_i) \subseteq \mathcal{V}(\vec{p}_i),$$

und jede Variable einer linken Seite taucht links nur einmal auf. Im folgenden wird o.B.d.A. angenommen, daß die Variablenmengen der einzelnen Definitionen untereinander disjunkt sind, d.h. zusätzlich zur Linkslinearität gilt

$$\forall i, j \leq n \bullet i \neq j \Rightarrow \mathcal{V}(\vec{p}_i) \cap \mathcal{V}(\vec{p}_j) = \emptyset. \tag{4.4}$$

Für jede einzelne Definition wird ein Prädikat mit Kopf $isFunc(f, V, R)$ erzeugt, wobei V das entsprechende Pattern – die Formalparameter für f – ist, und R einen Platzhalter für das Resultat der Berechnung beinhaltet. Der Körper des *isFunc*-Prädikats besteht aus zwei Teilen, zum einen der korrekten Implementierung des Pattern Matchings und zum anderen der Berechnung der rechten Seiten.

Pattern Matching Die intuitive Bedeutung des Pattern Matching ist, daß zunächst das Matching eines Aktualparameters mit dem Formalparameter untersucht wird (Peyton Jones und Hughes, 1999). Die erste passende Definition wird ausgewählt. Für die Testfallgenerierung mit CLP ist es nicht ausreichend, eine Übersetzung anzugeben, die einen Cut (s. S. 108) nach jeder Definition setzt. Das liegt daran, daß alle Möglichkeiten ausprobiert werden müssen. Die Funktionsdefinitionen `f(1)=1` und `f(X)=0` zeigen das Problem: Wenn die Funktion mit dem Wert `1` aufgerufen wird, ist nur die erste Definition anwendbar. Wird das dadurch erzielt, daß hinter der ersten Definition ein Cut gesetzt wird, dann wird bei Aufruf der Funktion mit einer Variablen `Y` als Argument ebenfalls nur die erste Definition ausprobiert. Das resultiert in der Bindung von `Y` an `1`. Die zweite Definition wird nicht berücksichtigt, was Unvollständigkeit als Konsequenz hat.

Daraus ergibt sich die Notwendigkeit, die Unifikation des Aktualparameters mit dem Formalparameter um eine zusätzliche Bedingung zu erweitern, nämlich

der, daß eine Funktionsdefinition mit Muster $\vec{p}_i$ nur dann ausgewählt werden darf, wenn die Unifikation mit allen $\vec{p}_j$ für $j < i$ gescheitert ist. Der Grund für den Übergang von Matching zu Unifikation ist der, daß im Rahmen der Testfallgenerierung durch die Auswahl einer Funktion Aktualparameter partiell gebunden werden sollen. Für $\vec{p}_k = t_{k1} \ldots t_{km}$ für ein m-stelliges f wird dies durch die Bedingung

$$\bigwedge_{\ell=1}^{k-1} \bigvee_{j=1}^{n} \exists \vec{x} \forall \vec{y} \bullet t_{kj}.\vec{x} \not\equiv t_{\ell j}.\vec{y}$$

präzisiert. Dabei kennzeichnet $t.\vec{x}$ die Ersetzung freier Variablen in t durch $\vec{x}$, und $\not\equiv$ ist die übliche symmetrische Ungleichheit, definiert durch

$$\ell \not\equiv r \iff \exists \vec{x}, \vec{y} \bullet \neg(\ell.\vec{x} = r.\vec{y})$$

bzw. wegen $\mathcal{V}(\ell) \cap \mathcal{V}(r) = \emptyset$ als Konsequenz von Gl. 4.4

$$\ell \not\equiv r \iff \exists \vec{x} \bullet \neg(\ell.\vec{x} = r.\vec{x}).$$

Eine quantorfreie Definition für das Pattern Matching der k-ten Definition ergibt sich als

$$\bigwedge_{\ell=1}^{k-1} \bigvee_{j=1}^{m} t_{kj} \neq^{pm} t_{\ell j}$$

unter Rückgriff auf die folgende asymmetrische Definition von $\neq^{pm}$:

$$\ell \neq^{pm} r = \begin{cases} \mathit{false}: & \ell, r \text{ identische Konstanten} \quad (1) \\ \mathit{true}: & \ell, r \notin \mathcal{V}, hd(\ell) \not\equiv hd(r) \quad (2) \\ \mathit{false}: & r \in \mathcal{V} \quad (3) \\ \bigvee a_i \neq^{pm} b_i: & \ell = h(a_1, \ldots, a_n), r = h(b_1, \ldots, b_n) \quad (4) \end{cases} \tag{4.5}$$

Erneut wird der Aufruf im Fall $\ell \in \mathcal{V}$ verzögert. Die Motivation für die dritte Gleichung ist die Beobachtung, daß

$$\exists \vec{x} \forall \vec{y} \bullet \ell.\vec{x} \not\equiv r.\vec{y}$$

nicht erfüllbar ist, wenn $\vec{x} = \vec{y}$ und ℓ und r unifizierbar sind. Dieser Fall tritt dann auf, wenn ein Argument in der i-ten Definition allgemeiner ist als in der j-ten für $j > i$. Ein Beispiel ist durch zwei Definitionen $f(X, 1) = r_1$ und $f(1, Y) = r_2$ für $X, Y \in \mathcal{V}$ gegeben. $1 \not\equiv X$ gilt nicht für die Substitution $\{X \mapsto 1\}$. Die Implementierung des Pattern Matching für die zweite Definition reduziert hingegen zu $Y \neq^{pm} 1$, und diese Negation wird wie im Fall der idle-Transitionen verzögert.

Abb. 4.10 zeigt die Anwendung von $\neq^{pm}$ beispielhaft. Das erste Argument ist der Funktionsname, das zweite das Argument, das dritte das Resultat. `pm_ineq` implementiert $\neq^{pm}$. Die für die zweite Definition erforderliche Ungleichheit $1 \neq^{pm} 2$ wird zur Übersetzungszeit automatisch herausoptimiert.

```
% Funktionsdefinitionen: f(1)=1, f(2)=2, f(X)=0

isFunc(f,1,1).
isFunc(f,2,2).
isFunc(f,X,0):- pm_ineq(X,1),pm_ineq(X,2).
```

Abbildung 4.10.: Übersetzung von Pattern Matching

Rechte Seiten Die Übersetzung der rechten Seiten in flache Prädikate ist Standard. Für jede Funktionsdefinition $f(t_1,\ldots,t_m) = rhs$ mit[6] $m \geq 0$ wird eine rechte Seite $\beta(rhs, R)$ für eine frische – bei Aufruf von β ungebundene – Variable R erzeugt. Definierte Funktionssymbole sind solche, die Kopfterm der linken Seite einer Funktionsdefinition sind. Frische Variablen sind eindeutig identifizierbare solche, die während der Berechnung von β vorher nicht aufgetaucht sind und die in keiner der Funktionsdefinitionen vorkommen. Da die Auswertungssemantik eager ist, ist $\beta(I, R)$ für nicht-konditionale Ausdrücke, d.h. $I \neq \mathit{ite}(C, T, E)$, dann rekursiv definiert durch

- $\bigwedge_{i=1}^{k} \beta(X_i, R_i) \wedge \mathit{isFunc}(f, (R_1,\ldots,R_k), R)$, falls f in $I = f(X_1,\ldots,X_k)$ für $k \geq 0$ ein definiertes Funktionssymbol ist und die R_i frische Variablen sind,
- $\bigwedge_{i=1}^{k} \beta(X_i, R_i) \wedge R = c(R_1,\ldots,R_k)$, falls c in $I = c(X_1,\ldots,X_k)$ für $k \geq 0$ kein definiertes Funktionssymbol und keine Variable ist und die R_i frische Variablen sind und
- $R = I$, falls I eine Variable ist.

Da partielle Funktionsdefinitionen zugelassen sind, z.B. Selektorfunktionen für Typkonstruktoren, dürfen Konditionale nicht mit dieser leftmost-innermost-Strategie übersetzt werden. Die einfache Definition per $\mathit{ite}(\mathit{true}, T, E) = T$ und $\mathit{ite}(\mathit{false}, T, E) = E$ scheidet also aus. Stattdessen ergibt sich unter Annahme einer Links-Rechts-Strategie der SLD-Resolution die Übersetzung als

$$\beta(\mathit{ite}(C, T, E), R) = \Big(\big(\beta(C, \mathit{true}) \wedge \beta(T, R)\big) \vee \big(\beta(C, \mathit{false}) \wedge \beta(E, R)\big)\Big)$$

Andere Standardfunktionen wie Boolesche Junktoren können einfach durch funktionale Programme oder durch Rückgriff auf bestehende Constraint-Löser definiert werden. Arithmetische Operationen und Komparatoren auf den natürlichen oder reellen Zahlen werden aufgrund der termalgebraischen Semantik der Logikprogrammierung durch externe, von CLP($\mathcal{X}$) zur Verfügung gestellte Operationen implementiert.

Insgesamt ergibt sich damit die Übersetzung der k-ten Definition einer m-stelligen Funktion f für eine Variable R als

$$\mathit{isFunc}(f, (t_{k1},\ldots,t_{km}), R) \Leftarrow \Big(\bigwedge_{\ell=1}^{k-1} \bigvee_{j=1}^{m} t_{kj} \neq^{pm} t_{\ell j}\Big) \wedge \beta(rhs_k, R).$$

[6] $m = 0$ wird zur Definition von Konstanten benötigt.

```
% Funktionsdefinitionen: f(1)=1, f(X)=2*X, g(X)=f(f(X)).

isFunc(f,1,1).
isFunc(f,X,R):- pm_ineq(X,1),isFunc('*',(2,X),R).
isFunc(g,X,R):- isFunc(f,X,R1),isFunc(f,R1,R).
isFunc('*',(X,Y),R):- R#=X*Y.
```

Abbildung 4.11.: Übersetzung von Funktionen

In Wächtern und Zuweisungen auftretende Funktionsaufrufe werden auf dieselbe Art und Weise wie rechte Seiten von Funktionsdefinitionen in Prädikate compiliert. Um das Anlegen unerwünschter Choicepoints zu verhindern, bietet es sich an, die Aufrufe eindeutig zu kennzeichnen, beispielsweise durch eine ganze Zahl, die beim Übersetzungsvorgang für jede Transition inkrementiert wird.

Abb. 4.11 zeigt ein einfaches Beispiel der Übersetzung. Das erste Argument ist der Funktionsname, das zweite die Parameter, das dritte das Resultat. Die Definition von `*` ist handerstellt, weil es sich um eine vordefinierte Funktion handelt. `#=` und `*` sind Operationen, die vom CLP-System zur Verfügung gestellt werden.

Datentypen Monomorphe Datentypen werden ebenfalls mit der Übersetzungsfunktion β übersetzt. Dazu werden Datentypdefinitionen der Form

$$data\ d = c_1 | \cdots | c_n$$

für ein u.U. geschachteltes und mit Typen parametriertes d und evtl. geschachtelte, durch Datentypen und Konstruktoren definierte c_i, deren Kopfsymbol kein definierter Datentyp sein darf, wie folgt transformiert. Für jedes c_i wird eine Funktionsdefinition der Form $fun\ d = c_i$ erzeugt, die dann mit β übersetzt wird. Datentypen wie die ganzen oder rationalen Zahlen werden von CLP($\mathcal{X}$) zur Verfügung gestellt.

Die Übertragung auf polymorphe Typen wird von Lötzbeyer und Pretschner (2000a) erörtert. Aus Effizienzgründen, insb. um dynamisches Binden im generierten Simulations- oder Produktionscode zu vermeiden, empfiehlt sich die Verwendung polymorpher Typen aber nur für die Definition von Hilfsfunktionen z.B. zur Listenverwaltung. Für die betrachteten Systeme kann von einer monomorphen Typisierung der Kommunikationskanäle mit der Umwelt ausgegangen werden. Diese wird durch Modelle von Sensoren und Aktuatoren festgelegt, die zur Compilezeit bekannt sind. Als Konsequenz können solche polymorphen Hilfsfunktionen bereits zur Compilezeit monomorph instantiiert werden. Aus Gründen der Wiederverwendbarkeit ist Polymorphie wünschenswert; vor der Testfall- und Codegenerierung muß dann eine Instanz festgelegt werden.

4.3.4. Optimierungen

Bei der Implementierung der obigen Übersetzung offenbaren sich mehrere mögliche Optimierungen: Common subexpression elimination bzw. term sharing und

eine Minimierung der Funktionsaufrufe in β sowie eine Einschränkung der (generierten) Idle-Transitionen. Aufgrund der häufig komplexen generierten Prädikate für idle-Transitionen ergab sich in Experimenten aus der letzten Optimierung ein signifikanter Laufzeitgewinn von bis zu 14%. Eine weitere Optimierung ergibt sich aus der Verwendung von Membership-Constraints, die die funktionale AutoFocus-Sprache nicht, übliche Constraintlöser aber sehr wohl zur Verfügung stellen.

- Identische Unterterme eines Terms müssen nur einmal ausgewertet werden. Das ist sehr einfach zu implementieren, indem bei der Berechnung von β protokolliert wird, welche Unterausdrücke bereits ausgewertet wurden und doppelte Berechnungen verhindert werden. Als Beispiel für Term sharing sei die Umsetzung von $\beta(ite(C,T,E),R)$ genannt, die effizienter durch

  $$\beta(C,R') \wedge \Big(\big((R' = true) \wedge \beta(T,R)\big) \vee \big((R' = false) \wedge \beta(E,R)\big)\Big)$$

  realisiert wird.

- Falls in der Berechnung von $\beta(I,R)$ I ein Term ohne definierte Funktionssymbole und Konditionale ist, wird die Übersetzung zu $\beta(I,R) = (I = R)$ und entsprechender Modifikation der zweiten Regel für β vereinfacht.

- Aufrufe der Pattern-Matching-Bedingungen und Wächter[7] von idle-Transitionen erfolgen erst dann und genau einmal, wenn alle anderen Komponenten ausgeführt worden sind, die keine idle-Transition feuern. Das ist dann sinnvoll, wenn die automatisch erzeugte idle-Transition umständlichen und nicht-optimierten Code enthält. Diese Optimierung ist zwar korrekt, schränkt aber die Vollständigkeit ein, d.h. der kleinste Fixpunkt des Programms wird unterapproximiert. Die Verzögerung ist korrekt und vollständig; problematisch ist die Einschränkung auf genau einen erfolgreichen Aufruf der idle-Transition. Ein Beispiel ist durch eine einzelne Komponente mit Eingabekanal i und Ausgabekanal o gegeben. Die zugehörige Zustandsmaschine besitze genau einen Zustand und eine Transition $\tau_1 : i?X \wedge X \neq 1 \wedge X \neq 2 \rightarrow o!X$. Als idle-Transition ergibt sich

  $$\tau_{idle} = i?no_msg \vee i?X \wedge (X = 1 \vee X = 2) \rightarrow o!no_msg,$$

  wie in Abb. 4.12 skizziert (idle-Transitionen gestrichelt; eine idle-Transition für den Fall, daß ein Signal anliegt, eine für den Fall, daß kein Signal anliegt).

  Eine mögliche Übersetzung der idle-Transition ist gegeben durch

  $$step(s, I, no_msg, s) \Leftarrow I = no_msg \vee I = (msg, 1) \vee I = (msg, 2).$$

  Wenn der Aufruf dieses Prädikats so erfolgt, daß er genau einmal erfolgreich sein muß (durch Einfügen eines Cuts), dann wird durch die Links-Rechts-Abarbeitung von Prolog für die idle-Transition die Variable für

[7] Ausgabekanäle erhalten den Wert *no_msg* unverzögert, um Nichtdeterminismus zu vermeiden.

PSfrag replacements

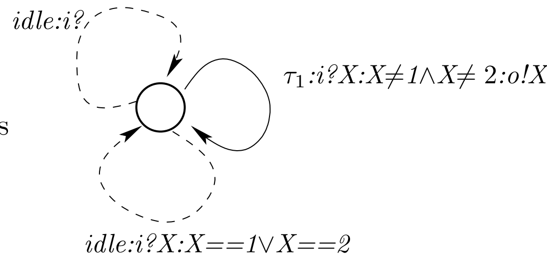


Abbildung 4.12.: Beispielsystem für Optimierung von Idle-Transitionen

den Eingabekanal I nur einmal an *no_msg* gebunden und die Werte 1 und 2 ignoriert. Der Tradeoff zwischen Unvollständigkeit und erhöhter Effizienz ist zu bewerten. Die Unvollständigkeit tritt nur in Situationen wie der obigen auf, wenn eine Idle-Transition auf einen „offenen“, mit der Umwelt verbundenen Kanal zugreift. Wenn alle in einer Idle-Transition abgefragten Kanäle mit anderen Komponenten verbunden sind, stellt das Backtracking in den anderen Komponenten sicher, daß ein vollständiges Aufzählen aller Traces erfolgt.

- Die Anzahl der zu erzeugenden Choicepoints kann durch die explizite Berechnung von Membershipconstraints reduziert werden. Die Funktion f sei durch

  $$f(a) = f(b) = \mathit{true} \;\; \mathit{und} \;\; f(c_1(X)) = f(c_2(X)) = \mathit{false}$$

  definiert. Die Übersetzung mit β liefert nach Eliminierung redundanter Patternmatching-Bedingungen

  $$\begin{array}{c} \mathit{isFunc}(f, a, \mathit{true}) \Leftarrow \mathit{true}, \\ \mathit{isFunc}(f, b, \mathit{true}) \Leftarrow \mathit{true}, \\ \mathit{isFunc}(f, c_1(X), \mathit{false}) \Leftarrow \mathit{true} \;\; \mathit{und} \\ \mathit{isFunc}(f, c_2(X), \mathit{false}) \Leftarrow \mathit{true}. \end{array}$$

  Der Aufruf von $\mathit{isFunc}(f, Y, R)$ für eine ungebundene Variable R führt dazu, daß vier Definitionen ausprobiert werden müssen.

  Die Optimierung besteht nun in der Berechnung von Membershipconstraints:

  $$\begin{array}{rcll} \mathit{isFunc}(f, X, \mathit{true}) & \Leftarrow & X \in \{a, b\} & \mathit{und} \\ \mathit{isFunc}(f, X, \mathit{false}) & \Leftarrow & (\mathit{nonvar}(X) \rightarrow & \mathit{hd}(X) \in \{c_1, c_2\}) \\ & \wedge & \big(\mathit{var}(X) \rightarrow & \mathit{susp}^{\mathit{inst}(X)}(\mathit{hd}(X) \in \{c_1, c_2\})\big). \end{array}$$

  Dabei extrahiert $\mathit{hd}(X)$ das Kopfsymbol eines nichtvariablen Terms und $\mathit{susp}^{\mathit{inst}(V)}(G)$ suspendiert das Prädikat G, bis Variable V instantiiert wird. *var* und *nonvar* entscheiden, ob das Argument eine Variable ist. Mit dieser Optimierung müssen nunmehr nur zwei Definitionen ausprobiert werden.

PSfrag replacements

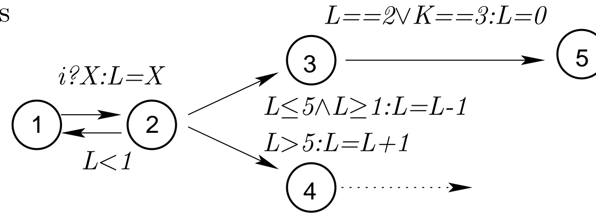


Abbildung 4.13.: Beispielsystem für symbolische Ausführung

Beispiel 16 (Membershipconstraints). *Im Chipkartenbeispiel tritt diese Situation z.B. in der Klassifikation von Kommandos auf, anhand derer entschieden wird, zu welchem Funktionsblock ein Kommando geschickt wird.*

4.4. Testfallgenerierung mit symbolischer Ausführung

Die obige Übersetzung gestattet die Simulation von AutoFocus-Modellen. Wenn das Prädikat $step^{top}$ für die oberste Komponente der SSD-Hierarchie mit vollständig instantiierten Werten für den Anfangszustand (Kontroll- und Datenzustände) sowie Belegungen der Eingabekanäle ausgewertet wird, werden ein neuer Kontrollzustand und Belegungen für Ausgabekanäle sowie aktualisierte lokale Variablen berechnet. Der neue Kontrollzustand und aktualisierte Variablenbelegungen sowie die Belegungen von Kanälen, die nicht mit der Umwelt verbunden sind, können dann als Ausgangspunkt für einen weiteren durch $step^{top}$ definierten Schritt dienen.

Offenbar beschreibt $step^{top}$ zusammen mit den anderen Prädikaten die Transitionsrelation T des Systems.

$$post[T]X \equiv \exists x \bullet Xx \wedge Txx'$$

bezeichne die Menge aller Nachfolgezustände einer Zustandsmenge X in bezug auf die Relation T. Es sei (unpräzise[8])

$$T = \{\big((\vec{\sigma}_{src}, \vec{\iota}), (\vec{\sigma}_{dst}, \vec{o})\big) : step^{top}(\vec{\sigma}_{src}, \vec{\iota}, \vec{o}, \vec{\sigma}_{dst})\}$$

unter Berücksichtigung der in der Definition von *step* verwendeten Verbindung von Komponenten durch identische Variablen.

Als Beispiel sei die in Abb. 4.13 teilweise angegebene EFSM betrachtet, die sich in einer Komponente mit einem Eingabekanal i vom Typen *Int* und einer lokalen Variable L vom selben Typ befinde. Die Transitionsrelation T ist dann

[8] Eine Präzisierung würde die genaue Definition der Verschaltung von Komponenten notwendig machen; hier wird darauf verzichtet. Siehe Lötzbeyer und Pretschner (2000a).

unter Vernachlässigung der idle-Transitionen definiert als

$$\begin{array}{ll} T(1,L)(2,X) & \textit{falls } i?X \\ T(2,L)(1,L) & \textit{falls } L<1 \\ T(2,L)(3,L-1) & \textit{falls } L\leq 5 \wedge L\geq 1 \\ T(2,L)(4,L+1) & \textit{falls } L>5 \\ T(3,L)(5,0) & \textit{falls } L=2 \vee L=3, \end{array}$$

wobei die erste Komponente der Paare den Kontrollzustand kennzeichnet und die zweite den Wert der lokalen Variable. Die Transitionsrelation spiegelt genau den erzeugten CLP-Code (ohne idle-Transitionen; _ kennzeichnet eine anonyme Variable; der zweite Parameter ist die Eingabe und der dritte der Wert der lokalen Variable vor und nach dem Tick):

```
step(s1,(msg,X),(_,X),s2).
step(s2,_,(L,L),s1):- L#<1.
step(s2,_,(L1,L2),s3):- L1#<=5, L1#>=1, L2#=L1-1.
step(s2,_,(L1,L2),s4):- L1#>5, L2#=L1+1.
step(s3,_,(L,0),s5):- L#=2; L#=3.
```

Für Repräsentationen von Zustandsmengen σ_1 und σ_2 gelte also $T\sigma_1\sigma_2$, wenn

$$\forall s_1 \in [\![\sigma_1]\!] \exists s_2 \in [\![\sigma_2]\!] \bullet Ts_1s_2 \wedge \forall s_2 \in [\![\sigma_2]\!] \exists s_1 \in [\![\sigma_1]\!] \bullet Ts_1s_2, \tag{4.6}$$

wobei $[\![\cdot]\!]$ die Menge aller typkorrekten Instanzen des Arguments – einer durch eine Menge von Constraints beschriebene Zustandsmenge – bezeichnet. Im Zug der symbolischen Ausführung definieren die erreichbaren Zustände σ Äquivalenzklassen $[\![\sigma]\!]$. Intuitiv sind zwei Zustände dann äquivalent, wenn sie dieselben Folgeausführungen des Modells zulassen.

Sofort stellt sich die Frage nach der „Granularität“ von T: Wenn Σ die Menge aller erreichbaren Zustände bezeichnet, erfüllt T definiert durch $T\Sigma\Sigma$ wegen der Input-Abgeschlossenheit offenbar Eigenschaft 4.6. Umgekehrt kann die Transitionsrelation für alle vollständig instantiierten Werte angegeben werden, etwa durch eine endliche Zustandsmaschine, die als Zustände alle möglichen Produkte der Zustände einer entsprechenden EFSM besitzt. Für die symbolische Ausführung ist die Granularität von T durch die Übersetzung definiert: Das Feuern der Transition von Kontrollzustand 2 zu Kontrollzustand 3 resultiert in der symbolischen Transition $T(2,L)(3,L-1)$ mit dem Constraint $1 \leq L \leq 5$. Für die Transition von 3 nach 5 ergeben sich für die angegebene Übersetzung zwei symbolische – hier vollständig instantiierte – Transitionen, nämlich $T(3,2)(5,0)$ und $T(3,3)(5,0)$. Die Übersetzung kann stattdessen mit Membership-Constraints (s. S. 121) durchgeführt werden:

$$\textit{step}(s3, _, (L, 0), s5) \Leftarrow L \in \{2, 3\}.$$

In diesem Fall ist die symbolische Transition durch $T(3,L)(5,0)$ mit dem Constraint $L \in \{2,3\}$ definiert. Im ersten Fall werden mit Backtracking die Fälle $L=2$ und $L=3$ hintereinander ausprobiert, im zweiten Fall wird mit dem Constraint $L \in \{2,3\}$ gerechnet.

Nun ergibt sich die Menge aller erreichbaren Zustände als kleinster Fixpunkt $\mu Y \bullet post[T]Y \vee init$ für den Initialzustand *init*, der wie oben angegeben mit der Kleene-Sequenz $S_0 \equiv init$ und $S_{i+1} \equiv S_i \vee post[T]S_i$ berechnet werden kann. Dabei wird davon ausgegangen, daß die Überprüfung des Abbruchkriteriums $S_i \equiv S_i + 1$ auf allen typkorrekten Grundinstanzen von S_i und S_{i+1} erfolgt. Das impliziert dann die Gleichheit der im Verlauf der symbolischen Ausführung erzeugten Constraintmengen. Wie in Abschnitt 4.3.1 beschrieben, ist das genau die Semantik des CLP-Programms, und in Prolog-Implementierungen wird sie durch eine Tiefensuche realisiert. Der Unterschied besteht darin, daß dort nicht mit Grundinstanzen gerechnet wird, sondern mit Verallgemeinerungen. Es ist dabei zu unterscheiden zwischen der Exploration des Zustandsraums des Modells und der Auswertung eines Prologprogramms: Letzteres wird top-down ausgewertet (s.S. 107). Die Berechnung der Programmsemantik erfolgt nicht – wie das mit Transformationen wie den Magischen Templates (Ramakrishnan, 1988) effizient bisweilen im Bereich der deduktiven Datenbanken geschieht – durch die explizite Berechnung der Kleene-Sequenz für den Operator T_P.[9] Stattdessen geschieht die Berechnung top-down mit Resolution. Der Zustandsraum des Modells wird abhängig von der Suchstrategie wie die Kleene-Sequenz (Breitensuche) oder bis zum Erreichen eines gesuchten Zustands mit Tiefensuche aufgebaut (vgl. Abschnitt 5.1).

Mit der zentralen Idee, Regeln eines Prolog-Programms als Transitionsrelation zu begreifen, kann sie nicht nur wie in dieser Arbeit zur Testfallgenerierung verwendet werden, sondern auch direkt zum Model Checking auf Prolog-Ebene (Delzanno und Podelski, 1999; Cui u. a., 1998; Fribourg, 1999).

Testfallgenerierung In der obigen Definition der Transitionsrelation wurde nicht gefordert, daß alle Paare von Zustandsrepräsentationen aus vollständig instantiierten Werten bestehen.[10] Das ist für die Ausführung des übersetzten Modells auch nicht nötig. Vielmehr können beliebige Werte unterspezifiziert bleiben. CLP-Sprachen können mit partiellen Datenstrukturen rechnen, d.h. mit Datenstrukturen, die „Löcher“ in Form nicht gebundener oder sog. logischer Variablen enthalten. Das bedeutet, daß das CLP-Programm nicht nur zur Simulation verwendet werden kann. Es ist beispielsweise möglich, in einem Aufruf von $step^{top}$ nur die Ausgaben zu spezifizieren. Die CLP-Implementierung findet dann Belegungen für alle anderen Parameter und insbesondere für den Anfangszustand, von dem ausgehend genau die spezifizierten Ausgaben erzeugt werden können. Das ist das Prinzip des Testfallgenerators dieser Arbeit: Sofern sie nicht stochastischer Natur sind, werden Testfallspezifikationen als existentielle Eigenschaften aufgefaßt, die Pfade beschreiben, die zu einem spezifizierten Zustand führen, einer Ausgabe etwa. Die CLP-Implementierung findet dann die entsprechenden fehlenden Werte. Da die betrachteten Systeme reaktiver Natur sind, müssen bisher erfolgte Schritte protokolliert werden, um Testsequenzen zu erzeugen. Aus Gründen der Einfachheit wird diese triviale Ergänzung hier

[9] Gleichung 4.1; genaugenommen, wird ein Operator verwendet, der auch partielle Antworten berechnet; die Menge der Grundinstanzen ist aber identisch (Gabbrielli u. a., 1995).

[10] Typkorrektheit wird durch entsprechende Constraints sichergestellt.

nicht weiter betrachtet (Lötzbeyer und Pretschner, 2000a).

Partielle Datenstrukturen Bei der Berechnung eines Testfalls *für ein deterministisches Modell*, also eines I/O-Traces des Modells, dessen Eingabeteil von einem Treiber auf die Maschine angewendet werden kann, werden nicht gezwungenermaßen alle Werte vollständig instantiiert. Das passiert dann, wenn die Information für das Fortschreiten der Berechnung nicht notwendig ist. Insbesondere im Zusammenhang mit den oben diskutierten verschiedenen Negationskonzepten ist das von Belang. Für große Wertebereiche ist es nicht effizient, alle möglichen Werte für eine Variable „auszuprobieren". Wenn ein Wächter beispielsweise spezifiziert, daß der Wert auf einem Kanal ungleich 7 sein soll, also $i?X : X \neq 7$, dann werden nicht alle möglichen Belegungen für X berechnet, sondern es wird mit dem symbolischen Constraint $X \neq 7$ gerechnet, der unter Umständen im Verlauf der Berechnung weiter eingeschränkt wird.[11] Es wird also mit Mengen von Werten gerechnet und nicht gezwungenermaßen mit einzelnen Werten. Offensichtlich hat dieses Vorgehen Vorteile im Vergleich mit explicit-state Model Checking. Symbolisches Model Checking ggf. unendlicher Systeme beruht auf exakt derselben Idee, nämlich große oder unendliche Mengen von Werten durch symbolische Repräsentationen zu behandeln (Alur u. a., 1995; Fribourg, 1999). Die erzeugten symbolischen Testfälle sind im Fall des Testfallgenerators Äquivalenzklassen, die alle dieselben Transitionsschaltungen und Funktionsaufrufe repräsentieren.

Die berechneten Traces sind Repräsentationen von Mengen von Testfällen, die zum Zweck des Tests einer Maschine typkorrekt instantiiert werden müssen. Da die repräsentierten Sequenzen in bezug auf die Auswahl der Transitionen äquivalent sind, kann eine Instantiierung mit Grenzwertanalysen im Fall geordneter Typen sinnvoll sein. Der große Vorteil ist, daß diese Äquivalenzklassen automatisch erzeugt werden, und zwar für jeden Schritt des Systems. Partitionierungen von Eingabedaten müssen bei transformativen Programmen nur einmal erfolgen; bei reaktiven ist eine solche „globale" Partitionierung im Normalfall nicht adäquat. Das Problem der Instantiierung wird im folgenden Abschnitt 4.5 behandelt.

Testfallgenerierung geschieht also wie folgt: Aus einem System wird ein CLP-Programm generiert, und ebenso wird aus einer Testfallspezifikation (Kapitel 3) ein CLP-Programm generiert. Für den Moment können diese Testfallspezifikationen als Spezifikation eines zu erreichenden Zustands (oder einer zu erreichenden Zustandsmenge) angesehen werden.

Constraints Die Verwendung von Constraints ist nicht nur in bezug auf die symbolische Ausführung von Vorteil. Mit Constraints kann der Resolutionsbaum auch explizit beschnitten werden, was die Effizienz der Suche steigert. Wenn im Fall der Chipkarte beispielsweise allein die Komponente `CardHolderVerification` getestet werden soll, kann es sinnvoll sein, zu verbieten, daß Kommandos berücksichtigt werden, die von der Komponente `Miscellenea` be-

[11] Im Zug der Berechnung der Idle-Transition wird auch der zum Zweck des Testens u.U. interessante Fall $X = 7$ berücksichtigt.

arbeitet werden, etwa die Generierung von Zufallszahlen. Es kann auch sinnvoll sein, eine Testsuite zu erzeugen, in der eine PIN höchstens dreimal abgeschaltet wird (die durchaus kritische Annahme wäre dann, daß viermaliges Abschalten keine Änderung des Verhaltens bewirkt). Eine ausführliche Diskussion erfolgt in Kapitel 3. Weitere Beispiele für derartige Constraints sind Einschränkungen des Wertebereichs der ersten Ableitung des Verlaufs eines Eingabesignals oder andere Umweltannahmen. Solche Zusammenhänge können sehr einfach mit Constraints beschrieben werden – die Modifikation von Modellen erscheint in diesem Fall durchaus aufwendiger. Der Zweck von Constraints ist technisch begründet: Der Suchraum soll eingeschränkt werden (genau so, wie kompliziertere als die bisher betrachteten Testfallspezifikationen der Einschränkung des Suchraums dienen). Je mehr manuelle Constraints eingefügt werden, desto kleiner wird der Suchraum. Dieses Vorgehen ist der Schlüssel zu einer graceful degradation des präsentierten Ansatzes.

4.5. Instantiierung und Konkretisierung

Die symbolische Ausführung liefert Äquivalenzklassen von Traces, indem für Mengen möglicher eingehender Signale jeweils eine Repräsentation generiert wird. Diese Repräsentation ergibt sich im wesentlichen aus den in einem System vorkommenden Konditionalen. Solche Konditionale treten in Funktionsdefinitionen sowie in Wächtern und Zuweisungen von Transitionen auf.

Wegen der Repräsentantenunabhängigkeit von Äquivalenzklassen kann dann jede Äquivalenzklasse in beliebige Repräsentanten instantiiert werden. Das kann zufällig geschehen, stochastisch auf der Basis von Nutzungsprofilen für Eingabesignale, oder bei geordneten Typen unter Rückgriff auf Grenzwertanalysen.

Beispiel 17 (Instantiierung). *Die Testfallspezifikation*

$$EF(output = ResComputeDigSig(_, _, _))$$

liefert u.a. einen symbolischen Trace der Länge 5:

$< Verify(pinGRef, PinG), H9000 >,$
$< MSERestore(SE1Ref), H9000 >,$
$< MSEDSTSetPrivateKey(EF_4451, Key1Ref), H9000 >,$
$< PSOComputeDigSig(X : X \in \{BytesA, BytesB, BytesC\}), H61XX >,$
$< GetResponse, ResComputeDigSig(EF_4451, Key1Ref, X) > .$

X ist dabei eine Variable, die für verschiedene zu signierende Bytefolgen stehen kann. Eine mögliche zufällige Instantiierung ist die mit BytesA, was einen Trace des Modells liefert, der keine ungebundenen Variablen mehr enthält.

Beispiel 18 (Grenzwertige Instantiierungen). *Auf Modellebene der Chipkarte finden sich (geordnete) Integer-Typen bei der Länge von zu berechnenden Zufallszahlen sowie bei Längen- und Offsetbytes für Dateizugriffe. Für diese wurden dann die typischen Grenzwert-Testfälle generiert. Auf Testtreiberebene wurden weiterhin Grenzwertanalysen für Nullbytefolgen als Schlüssel sowie für die beschränkte Länge von zu signierenden Daten ausgeführt und entsprechende Testfälle auf das System angewendet.*

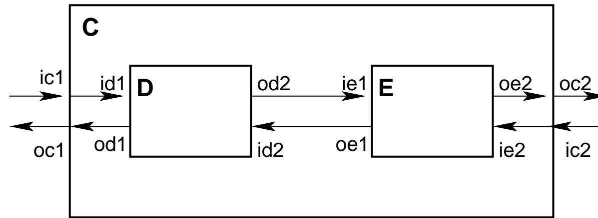


Abbildung 4.14.: Beispielsystem für kompositionale Testfallerzeugung

Die Testfallspezifikation $EF(input = AskRandom(N))$ *liefert u.a. einen symbolischen Trace der Länge* 1*, nämlich* $< AskRandom(N), ResRandom >$*. Daraus werden die folgenden grenzwertigen Testfälle der Länge 1 instantiiert:*

- $< AskRandom(0), ResRandom >$,
- $< AskRandom(1), ResRandom >$,
- $< AskRandom(254), ResRandom >$ *und*
- $< AskRandom(255), ResRandom >$.

Die klassische Technik der Grenzwertanalyse wird also auf reaktive Systeme erweitert. Für transformative Systeme besteht die Idee darin, Klassen für das Eingabedatum zu bilden, die insofern „Äquivalenzklassen" darstellen, als angenommen wird, daß jeder Repräsentant zu einem identischen Fehler führt. Es handelt sich also nicht um mathematisch präzise faßbare Äquivalenzklassen. Darüberhinaus ist der Ansatz auf *eine* Partitionierung des Datenraums beschränkt, was für reaktive Systeme erweitert werden muß: Die Partitionierung zu einem Zeitpunkt kann durchaus unterschiedlich zu der eines zweiten Zeitpunktes sein. Der vorgestellte Ansatz rechnet insofern mit echten Äquivalenzklassen, als jeder Repräsentant einer Eingabeklasse zu einem bestimmten Zeitpunkt zu identischen möglichen Folgetraces führt. Für jedes einzelne Signal kann zu *jedem* Zeitpunkt eine Grenzwertanalyse durchgeführt werden.

4.6. Kompositionalität und Integrationstests

Testfallgenerierung mit symbolischer Ausführung gestattet das Suchen nach (partiellen) Systemzuständen, die z.B. durch spezifische Ausgaben angegeben werden können. Dieses Charakteristikum macht eine kompositionale Testfallgenerierung möglich, die u.a. Grundlage für die in Abschnitt 2.3.4 beschriebene Technik zur verzahnten Entwicklung von Testfällen und Modellinkrementen ist.

Kompositionalität Abb. 4.14 zeigt ein aus drei Komponenten C, D und E bestehendes einfaches Beispielsystem. $oK\ell$ bezeichnet den ℓ-ten Ausgabeport der Komponente K; $iK\ell$ den ℓ-ten Eingabeport der Komponente K. Mit dem vorgestellten Verfahren können für alle drei Komponenten Testfälle erzeugt werden, $\mathcal{T}_C$, $\mathcal{T}_D$ und $\mathcal{T}_E$. Offenbar stellen nun die auf *od2* geschriebenen Ausgaben von

D zugleich Testdaten für E auf *ie1* dar und die Ausgaben von E auf *oe1* Testdaten für D auf *id2*. Nach Berechnung der entsprechenden Eingaben ergeben sich erweiterte Testsuiten $\mathcal{T}'_D \cup \mathcal{T}_D$ und $\mathcal{T}'_E \cup \mathcal{T}_E$.

Umgekehrt können die Eingaben aus $\mathcal{T}_D$ auf *id2* als putative Ausgaben von E auf *oe1* und die Eingaben aus $\mathcal{T}_E$ auf *ie1* als putative Ausgaben von D auf *od2* aufgefaßt werden. Mit der symbolischen Ausführung können so die fehlenden Ein- und Ausgaben berechnet werden, um erweiterte Testsuiten $\mathcal{T}''_D \cup \mathcal{T}'_D$ und $\mathcal{T}''_E \cup \mathcal{T}'_E$ zu erhalten. Wenn beispielsweise Eingaben auf *ie1* zur Testfallgenerierung für D verwendet werden soll, dann müssen entsprechende Eingabewerte für *id1* und *id2* sowie Ausgaben für *od1* berechnet werden. Offenbar liefern die erweiterten Testsuiten $\mathcal{T}''_D$ und $\mathcal{T}''_E$ auch neue Testfälle für die übergeordnete Komponente C.

Das ist die Grundidee der kompositionalen Testfallerzeugung. Dieses einfache Verfahren funktioniert in der Allgemeinheit nicht immer. Wenn beispielsweise $\mathcal{T}''_D$ auf der Grundlage von *ie1* aus $\mathcal{T}_E$ berechnet wird, dann muß gleichzeitig auf eine Übereinstimmung von *oe1* und *id2* geachtet werden. Unter Umständen ist das gar nicht möglich, weil die Verschaltung von C und D bestimmte Abläufe von C oder D allein gar nicht zuläßt.

Umweltannahmen Ein einfaches Beispiel illustriert den Zusammenhang. Wenn D eine Timer-Komponente ist, die in periodischen Abständen auf *od2* einen Timeout sendet, dann kann es bei der Generierung von Testfällen für E in Isolation durchaus passieren, daß diese Testfälle Timeouts in nicht periodischen Abständen verlangen. Das kann dann eintreten, wenn die Information über die Periodizität der Timeouts nicht explizit in E codiert ist. Das Verhalten einer isolierten Komponenten ist üblicherweise eine Erweiterung der entsprechenden Projektion eines integrierten Systems (die tatsächlich ausführbare Tracemenge wird erweitert). Komposition mit anderen Komponenten führt dann zu einer Konkretisierung. Dieses Phänomen wird bei der Verwendung von Testmodellen (Abschnitt 2.3.1) zur Reduktion des Suchraums ausgenutzt.

Der Sachverhalt ist Instanz eines übergeordneten Problems. Komponenten sind wie Prozeduren insofern unterspezifiziert, als sie normalerweise stets implizite Annahmen an die Eingabeparameter (und in reaktiven Systemen an deren Reihenfolge) beinhalten. Es ist u.a. genau diese Beobachtung, die

- zu Assumption-Commitment-artigen Spezifikationstechniken,
- zur Einführung von Kontrakten (Meyer, 1992) (insb. des – beliebig schwachen – *requires*-Teils) in Sprachen wie Eiffel oder von Annotationen in Spark-Ada oder statischen Analysetools wie ESC und
- zur Erweiterung von Schnittstellenspezifikationen mit Verhalten (Nierstrasz und Papathomas, 1990; Nierstrasz, 1995) – Typen als partielle Verhaltensspezifikation –, die insbesondere in dynamischen Dienstarchitekturen eine Rolle spielt,

geführt hat (vgl. auch de Alfaro und Henzinger (2001) zur legalen Komposition bzgl. *einer* vs. *aller* möglichen Umgebungen). AutoFocus gestattet bei der

Spezifikation von Wächtern das Formulieren beliebig starker Vorbedingungen. Diese sind aber auf den jeweils aktuellen Zustand beschränkt, wenn die Systemhistorie nicht in expliziten Variablen abgespeichert wird oder jeder Komponente eine weitere vorgeschaltet wird, die die Vorbedingung ggf. abstrakt codiert. Solche Komponenten dienen, wie viele Testfallspezifikationen auch, der Einschränkung des Suchraums. Der offenbar bestehende Einfluß eines Modellierungsformalismus mit oder ohne Möglichkeiten, Kontrakte zu spezifizieren, wurde bereits in Abschnitt 2.3.1 diskutiert.

Das Problem findet seinen Niederschlag auch in der Unzulänglichkeit von Unit-Tests. Testdaten für Units (z.B. Subkomponenten eines Modells) können durchaus derart gestaltet sein, daß sie niemals von den umgebenden Komponenten erzeugt werden. Die Notwendigkeit von Integrationstests ist genau darin begründet, daß Unit-Tests normalerweise nicht ausreichen, das korrekte Funktionieren des Gesamtsystems zu garantieren. Wenn es erfolgreich angewendet werden kann, liefert das oben skizzierte Verfahren offenbar Integrationstestfälle.

Daß das Verfahren nicht in aller Allgemeinheit immer erfolgreich Integrationstestfälle berechnen kann, bedeutet aber nicht, daß es deshalb niemals angewendet werden kann. Ganz im Gegenteil zeigt sich beispielsweise bei der kompositionalen Berechnung von Integrationstestfällen für die Chipkarte (Abschnitt 5.4), daß es durchaus erfolgreich angewendet werden kann. Ob das Verfahren anwendbar ist, hängt immer vom betrachteten Modell und insbesondere von der Kopplung der Komponenten ab.

Beispiel 19 (Kopplung). *Im Fall der Chipkarte ist diese Kopplung eher lose – in horizontaler Richtung können aus den funktionalen Blöcken in der Mitte direkt Testfälle für die Komponente* `WIMPre` *abgeleitet werden. Die Kopplung in der vertikalen Richtung (Kanäle, die shared memory zwischen Komponenten implementieren) ist dergestalt, daß sie bei Anwendung einer weiteren Technik, die das Aufspalten von Testfällen für eine Komponente in die Zustände in jedem Tick beinhaltet, ohne weiteres zum Erfolg führt.*

Integrationstests Die Beschreibung dieser Technik bildet den folgenden abschließenden Teil dieses Abschnitts.

Reaktive Systeme bergen die Schwierigkeit, daß Unit-Testfälle nicht nur für einen Schritt „zusammenpassen“ müssen, sondern über die gesamte Länge des durch die Testfälle beschriebenen Traces. Anstatt nun für einen Gesamttrace T_E von beispielsweise E einen direkt passenden Trace für D zu suchen – was sich durch die zeitsynchrone Ausführungssemantik von AutoFocus anbietet –, wird T_E aufgespalten. Da Testfälle aus dem Modell erzeugt werden, kann davon ausgegangen werden, daß T_E eine Sequenz von Projektionen von Zuständen des Gesamtsystems ist, die nicht nur die lokalen Variablen von E beinhalten, sondern auch die Werte aller an E angeschlossenen Kanäle.

Im folgenden wird exemplarisch gezeigt, wie aus einem Testfall für E einer für C abgeleitet werden kann. Es sei also

$$T_E = < \varepsilon_1, \ldots, \varepsilon_n >$$

PSfrag replacements

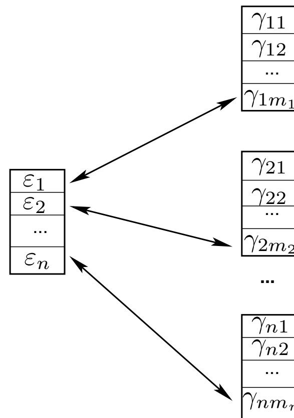


Abbildung 4.15.: Generierung fehlender Signale

ein Testfall für E. Abb. 4.15 skizziert den Zusammenhang. Mit gerichteter Suche (Abschnitt 5.1) wird nun zunächst probiert, ausgehend vom Initialzustand γ_{11} von C einen Trace

$$T_{C1} =< \gamma_{11}, \ldots, \gamma_{1m_1} >$$

des aus D und E zusammengesetzten Systems zu finden. T_{C1} soll die Eigenschaft besitzen, daß die Projektion von γ_{1m_1} auf die Komponente E eine Variante, Spezialisierung oder Verallgemeinerung (Abschnitt 5.2) von ε_1 ist. Danach wird probiert, einen Trace

$$T_{C2} =< \gamma_{1m_1}, \gamma_{21}, \ldots, \gamma_{2m_2} >$$

zu finden, wiederum mit der Eigenschaft, daß die Projektion von γ_{2m_2} auf E eine Variante, Verallgemeinerung oder Spezialisierung von ε_2 ist. Das Verfahren wird iteriert mit einem Trace

$$T_{C3} =< \gamma_{2m_2}, \gamma_{31}, \ldots, \gamma_{3m_3} >$$

usw., bis die Projektion auf E des letzte Zustand eines Traces T_{Cn} semantisch mit ε_n vergleichbar ist. Offenbar erfordert das Backtracking nicht nur innerhalb der Berechnung eines T_{Ci}, sondern u.U. muß auch ein früherer Teiltrace neu berechnet werden. Das ist der Fall, wenn sich herausgestellt hat, daß die entsprechenden Zustände dieses Teiltrace dazu führen, daß ein späterer Zustand nicht erreicht werden kann. Wenn ein resultierender Trace

$$T_{C1} \circ \ldots \circ T_{Cn}$$

mit $\circ$ als Konkatenationsoperator generiert werden kann, handelt es sich um einen Integrationstest für C.

Der Vorteil dieses Ansatzes gegenüber einer Big-Bang-Erzeugung von Testfällen für das Gesamtsystem liegt darin, daß durch die Testfälle der einzelnen Komponenten der Suchraum eingeschränkt wird.

Beispiel 20 (Kompositionale Testfallerzeugung). *Im Fall der Chipkarte kann eine Testsuite, die ein Coveragekriterium auf Integrationslevel erfüllen soll, aus Gründen mangelnder Effizienz des Testfallgenerators nicht erzeugt werden, wenn das Gesamtsystem zugrundegelegt wird. Mit der kompositionalen Technik gelingt das allerdings.*

Als Beispiel sei eine Transition einer Unterkomponente der Komponente `CardHolderVerification` *genannt. Die Berechnung einer Testsuite (Testfälle der Länge 1), die das Coveragekriterium für jede Transition dieser Komponente erfüllt, enthält unter anderem den Fall, daß der Wiederholzähler einer der PINs des Systems gleich Null ist. Es bezeichne σ den Zustandsvektor von* `CardHolderVerification`*, in dem das Feld für den Wiederholzähler der PIN gleich Null ist. Es wird dann zunächst ein Testfall für* `CardHolderVerification` *gesucht, der für diese Komponente ausgehend von ihrem Initialzustand zu σ führt. Das ist hier ein Testfall der Länge drei. Daraus wird dann ein Testfall für das Gesamtsystem berechnet. Dieser Testfall (ebenfalls Länge 3) hat die Eigenschaft, daß die Projektion des Zustandsvektors seines letzten Eintrags auf die Unterkomponente von* `CardHolderVerification` *genau σ ist.*

Ein ausführliches, komplexeres Beispiel wird im Zusammenhang mit der Berechnung von Integrationstestfällen für ein strukturelles Überdeckungsmaß auf den Seiten 139ff. (Beispiel 24) beschrieben.

Das präsentierte Verfahren beschränkt sich nicht auf ein Verständnis von „Units“ als Komponenten. Im folgenden Abschnitt 4.7 wird erläutert, wie das Verfahren verwendet werden kann, um Testfälle für Funktionsdefinitionen zur Definition von Testfällen für Transitionen, Komponenten und Komponentenaggregationen verwendet werden kann.

Bevor darauf genauer eingegangen wird, soll erneut das Beispiel des Simultaneous Engineering im Maschinenbau (S. 2.3.1) bemüht werden, um einen weiteren Aspekt der kompositionalen Testfallgenerierung zu verdeutlichen: Die Definition von Komponentenaggregationen kann durchaus die Systemgrenze überschreiten. Wenn gleichzeitig ein Modell der Anlage und ein Modell der SPS entwickelt werden dann sind die Testfälle für jedes einzelne offenbar komplementär zueinander: Ausgaben der Anlage (des Modells der Anlage) sind zugleich Eingaben für die SPS und umgekehrt.

4.7. Generierung integrierter MC/DC-Testsuiten

Funktionale Testfallspezifikationen sind domänen- und anwendungsabhängig, und ihre Ermittlung stellt den wesentlichen intellektuellen Anspruch an den Tester dar (Kap. 3). Für strukturelle Testziele können automatisch Testfallspezifikationen berechnet werden. Das wird in diesem Abschnitt beschrieben. In Abschnitt 3.3.2 wurde bereits erläutert, inwiefern die syntaktische Überdeckung von Modellen als Abdeckung von Anforderungen verstanden werden kann.

Wenn Modelle mit EFSMs definiert werden, die auf Transitionen beliebig komplexe Funktionen aufweisen, dann sind Überdeckungsmaße wie Transitions- oder Zustandsabdeckung sehr schwach und beinhalten noch nichteinmal Anweisungsüberdeckung. Als Beispiel wird deshalb das modified condition/decision

coverage-Abdeckungskriterium betrachtet (MC/DC, Definition auf S. 133). Abdeckungskriterien sind üblicherweise auf Unit-Ebene (s.o.) definiert. Der Grund dafür ist, daß so die Komplexität der Testerzeugung reduziert werden kann. Auf der anderen Seite bedeutet das, daß Testfälle für Units Situationen darstellen können, die im integrierten System niemals auftreten – genau diese Tatsache motivierte im letzten Abschnitt die kompositionale Testfallgenerierung. Es wird ein Verfahren vorgestellt, das Testfälle auf Systemebene erzeugt, so daß das Coveragekriterium auf Unitebene erfüllt ist (wenn solche Testfälle existieren und der Testfallgenerator mächtig genug ist, was er im Fall der Chipkarte ist).

Vorgehen Die Idee ist die folgende. Zunächst wird ein Verfahren präsentiert, mit dem für beliebige funktionale Ausdrücke MC/DC erfüllende Testsuiten mit dem auf symbolischer Ausführung basierenden Testfallgenerator erzeugt werden können. Im wesentlichen basiert das auf einem einfachen Rewriting der Funktionsdefinitionen.

Mit derart modifizierten funktionalen Ausdrücken können dann Testsuiten für funktionale Programme erzeugt werden – wenn die Definition von Units auf Basis solcher Programme erfolgt, ist das offenbar ein erster sinnvoller Ansatz. Derselbe Ansatz kann aber auch verwendet werden, um für jede Transition – Wächter und Zuweisung – einer EFSM eine MC/DC erfüllende Testsuite zu erzeugen. Das Resultat dieser Berechnung sind Repräsentationen der Vor- und Nachbedingungen der Transitionen. Wenn Units als einzelne Transitionen definiert sind, ist das wiederum offenbar ein sinnvoller Ansatz.

Wenn die Definition von Units hingegen auf der Basis von EFSMs erfolgt, dann müssen Testsequenzen erzeugt werden – Präambeln –, die zu Instanzen der o.g. Vorbedingungen von Transitionen führen. In Abschnitt 5.1 werden Verfahren der gerichteten Suche präsentiert, mit denen solche Traces effizient erzeugt werden können. Ergebnis des Vorgehens sind dann Testsequenzen auf EFSM-Ebene, die MC/DC auf Transitionsebene erfüllen.

Schließlich bleibt die Problematik bestehen, daß Testfälle für Komponenten wiederum Sequenzen repräsentieren können, die sich nach Integration mit anderen Komponenten als undurchführbar herausstellen. Die Idee besteht dann darin, das im letzten Abschnitt 4.6 präsentierte kompositionale Verfahren zur Testfallgenerierung anzuwenden. Für jeden Zustand eines Traces einer Komponente, der MC/DC auf Transitionsebene erfüllt, wird probiert, einen entsprechenden Trace des Gesamtsystems (oder eines entsprechend definierten Teilsystems) zu finden. Resultat dieses Schritts ist eine Testsuite für das betrachtete (Teil-)modell, das MC/DC auf Transitionsebene für jede involvierte EFSM gewährleistet.

Wenn in einem der aufeinander aufbauenden o.g. Schritte eine entsprechende Testsuite nicht gefunden werden kann, was aus Gründen der Effizienz geschehen kann oder weil beispielsweise ein Zustand einer MC/DC erfüllenden Testsuite auf EFSM-Ebene tatsächlich nicht erreichbar ist, dann muß das Verfahren mit einer anderen MC/DC-Suite für die „darunter“ liegende Schicht iteriert werden. Solche anderen MC/DC-Suiten existieren im Normalfall, weil zum einen in vielen Fällen die Definition einer MC/DC-Suite ohnehin nicht eindeutig ist, und

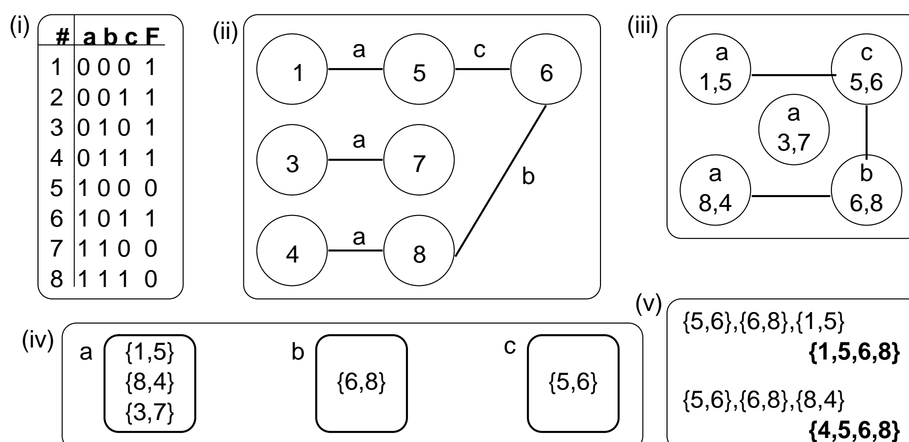


Abbildung 4.16.: MC/DC-Testfälle

weil zum anderen im Zug der symbolischen Ausführung Wertinstantiierungen vorgenommen werden, die genauso gut auch anders hätten durchgeführt werden können, z.B. durch die Auswahl einer anderen Funktionsdefinition. Das in CLP-Systemen eingebaute Backtracking erlaubt einen komfortablen Umgang mit Situationen dieser Art.

Schließlich kann es passieren, daß MC/DC-Testsuiten auf EFSM- oder auf Subsystemebene nicht auch gleichzeitig MC/DC auf Funktionsdefinitionsebene erfüllen. In diesem Fall wird protokolliert, welche Funktionsdefinitionen nicht ausreichend abgedeckt wurden, und es wird versucht, Wächter und Zuweisungen derjenigen Transitionen, die die entsprechende Funktionsdefinition u.U. indirekt aufrufen, derart zu belegen, daß MC/DC-Abdeckung auf Funktionsdefinitionsebene gewährleistet wird. Das o.g. Verfahren wird dann entsprechend iteriert.

Es kann passieren, daß das Verfahren nicht zu einer minimalen Testsuite führt. Wenn beispielsweise Testsequenzen auf EFSM-Ebene berechnet werden, dann werden auf dem Weg zu einer bestimmten Transition bereits bestimmte Teile einer MC/DC-Suite der vorherigen Transitionen abgedeckt. Es bietet sich dann an, die Testsuiten im Nachhinein zu minimieren, was einen Teil zukünftiger Arbeiten darstellt.

MC/DC Die Verwendung von MC/DC wird vom DO-178B, einem Standard der zivilen Luftfahrtindustrie (RTCA und EUROCAE, 1992), als Komplement zu funktionalen Tests empfohlen. MC/DC fordert, daß jede Bedingung so getestet wird, daß für jedes Literal ℓ der Bedingung zwei Testfälle – Belegungen der Literale – existieren, so daß die Bedingung einmal zu *true* und einmal zu *false* evaluiert und in einem Fall ℓ zu *true* und im anderen Fall zu *false* evaluiert, während alle anderen Literale ihren Wert nicht ändern. Eine Formalisierung findet sich bei Vilkomir und Bowen (2001).

Wenn eine MC/DC-Testsuite existiert (Gegenbeispiel 21), dann besteht sie bei n Literalen aus $n+1$ Testfällen, d.h. verschiedenen Variablenbelegungen.

Als Beispiel sei die Formel $F = \neg a \vee (\neg b \wedge c)$ betrachtet. Die Wahrheitstabelle

findet sich in Abb. 4.16 (i). Für diese Formel existieren zwei MC/DC-Testsuiten, nämlich $S_1 = \{1, 5, 6, 8\}$ und $S_2 = \{4, 5, 6, 8\}$. In S_1 ändern die Fälle 1 und 5 den Wert von a und den von F, $(5, 6)$ ändern c, und $(6, 8)$ ändern b. In S_2 ändern $(4, 8)$ den Wert von a anstelle von $(1, 5)$ in S_1 (Abb. 4.16 (v)).

MC/DC-Testsuiten existieren nicht immer (Tautologien oder direkte Abhängigkeit eines Literals von einem anderen):

Beispiel 21 (Nichtexistenz einer MC/DC-Belegung). *Die EFSM der Komponente* `WIMPre` *verteilt das Kommando zur Berechnung der digitalen Signatur an die Komponenten* `CardholderVerification` *und* `SecurityOperations`. *Alle anderen Kommandos, die von* `CardholderVerification` *bearbeitet werden, werden ausschließlich von dieser Komponente bearbeitet. Demzufolge gibt es in* `WIMPre` *eine Transition*

```
c?Cmd: isVerificationCommand(Cmd)&&not(is_PSOComputeDigSig(Cmd)):
  cv!Cmd.
```

Für das Kommando $PSOComputeDigSig(X)$ gilt nun `isVerificationCommand(PSOComputeDigSig(X))` *und zugleich (ausschließlich)* `is_PSOComputeDigSig(PSOComputeDigSig(X))`. *Offenbar kann es für den Wächter der Transition keine MC/DC-Suite geben. Es gilt nämlich*

`is_PSOComputeDigSig(PSOComputeDigSig(X))` $\Rightarrow$
`isVerificationCommand(PSOComputeDigSig(X))`.

Bedingungen Um automatisiert MC/DC-Testsuites für Modelle reaktiver Systeme berechnen zu können, besteht ein erster Schritt darin, eine MC/DC-Testsuite für Bedingungen F zu berechnen. Der folgende Algorithmus erledigt das.

Für n Variablen seien alle Belegungen durch mit 1 beginnende aufeinanderfolgende Zahlen benannt. In einem ersten Schritt wird jeder Testfall i für $1 \leq i < 2^n$ mit allen folgenden Testfällen j, $i < j \leq 2^n$ verglichen, und wenn ein Paar $(\vec{v}, \vec{w}) = ((v_1, \ldots, v_n), (w_1, \ldots, w_n))$ von Variablenbelegungen die MC/DC-Bedingung erfüllt, d.h.

$$\exists k \bullet \neg(F(\vec{v}) \leftrightarrow F(\vec{w})) \wedge \bigwedge_{\substack{i=1 \\ i \neq k}}^{n} v_i \leftrightarrow w_i \wedge \neg(v_k \leftrightarrow w_k)$$

gilt, wird diese Information abgespeichert (Abb. 4.16 (ii)). Kanten repräsentieren Variablen, Knoten repräsentieren Testfälle. Wenn eine MC/DC-Testsuite existiert, muß also eine Kantenabdeckung des Graphen gefunden werden, so daß jede Kantenbeschriftung genau einmal vorkommt. Dualisiert man den Graphen (Abb. 4.16 (iii)), so muß offenbar eine minimale Knotenabdeckung gefunden werden, so daß der Teil der Beschriftung, der eine Variable repräsentiert, genau einmal vorkommt. Da nun bekannt ist, daß im Fall der Existenz einer MC/DC Testsuite diese aus $n+1$ Testfällen besteht, so reduziert sich die Suche auf zusammenhängende Pfade. Tatsächlich ist die explizite Graphenbildung gar nicht nötig, sondern die Information kann direkt aus der Gruppenbildung des ersten Schritts entnommen werden (Abb. 4.16 (iv)). Die Testfälle müssen dann

so ausgewählt werden, daß der Durchschnitt zwischen zwei Klassen bzgl. eines involvierten Testfalls jeweils leer ist.

Die Komplexität des Algorithmus liegt in $\mathcal{O}(2^{2n})$. Für die in der Praxis auftretenden Fälle $n < 8$ stellt das kein Problem dar.

Funktionale Ausdrücke Um eine Testsuite für einen funktionalen Ausdruck – die rechte Seite einer Funktionsdefinition, einen Wächter oder eine Zuweisung – zu berechnen, wird jeweils ein Konditional $ite(C, T, E)$ fixiert. Die grundlegende Idee besteht darin, dieses Konditional in ein anderes zu „verpacken", so daß die Auswertung der Atome in C so erzwungen wird, daß die berechnete Testsuite MC/DC erfüllt.

In einem ersten Schritt werden die Atome $A_1, \ldots, A_n$ einer Bedingung C in symbolische Namen $S_1, \ldots, S_n$ abstrahiert, was zu einer abstrakten Bedingung C' führt. Beispielsweise wird

$$e \equiv f(X, ite(\neg g(X) \vee Y < 7, 123, 456))$$

zu $f(X, ite(C', 123, 456))$ mit $C' = \neg S_1 \vee S_2$ und der Substitution $\sigma = \{S_1 \mapsto g(X), S_2 \mapsto Y < 7\}$. Dann wird eine MC/DC-Suite $\pi^c_{MC/DC}(C')$ für C' wie oben beschrieben berechnet. In diesem Fall gibt es nur eine, nämlich $\pi^c_{MC/DC}(C') = \{\{S_1 = \bot \wedge S_2 = \bot, S_1 = \top \wedge S_2 = \bot, S_1 = \top \wedge S_2 = \top\}\}$. Für die Testfallgenerierung wird dann die Menge

$$M = \big\{\{f(X, ite\big(\sigma(m), ite(C, 123, 456), fail\big) : \{m\} \in \pi^c_{MC/DC}(e)\}\big\}$$

berechnet, wobei (1) $\sigma(x)$ die Anwendung der Substitution σ auf x bezeichnet und (2) $fail$ einen speziellen Wert denotiert, der das Scheitern der Berechnung kennzeichnet. Die Formeln in $\pi^c_{MC/DC}(e)$ sind Konjunktionen von Belegungen einzelner Variablen, womit das gewünschte Ziel einer Auswertung des Ausdrucks, die MC/DC erfüllt, erreicht wird.

In diesem Fall besteht $M = \{e_1, e_2, e_3\}$ aus den drei funktionalen Ausdrücken

$$\begin{aligned} e_1 &= f(X, ite\big(\neg g(X) \wedge Y \geq 7, ite(\neg g(X) \vee Y < 7, 123, 456), fail)\big), \\ e_2 &= f(X, ite\big(g(X) \wedge Y \geq 7, ite(\neg g(X) \vee Y < 7, 123, 456), fail)\big) \text{ und} \\ e_3 &= f(X, ite\big(g(X) \wedge Y < 7, ite(\neg g(X) \vee Y < 7, 123, 456), fail)\big). \end{aligned}$$

Wird das obige Vorgehen auf alle Bedingungen eines funktionalen Ausdruckes angewandt, so erhält man mit symbolischer Ausführung eine MC/DC-Suite für jede Bedingung, sofern eine solche existiert.

Um die Ersetzung allgemein formulieren zu können, bezeichne $\mathcal{O}(t)$ die Menge der Positionen in einem Term t. $t|_p$ ist der Subterm von t an Stelle $p \in \mathcal{O}(t)$, und für $p \in \mathcal{O}(t)$ bezeichne $t[s]_p$ die Ersetzung des Subterms von t an Position $p \in \mathcal{O}(t)$ durch s. $hd(t)$ sei das Kopfsymbol eines Terms t. Dann ist

- $\pi^c_{MC/DC}(C)$ für eine Bedingung $ite(C, T, E)$ eine Menge von Testsuiten (eine Menge von Mengen von Variablenbelegungen), die MC/DC für C erfüllen und die mit dem o.g. Algorithmus berechnet werden,

- für $t|_p = ite(C, T, E)$

$$\pi^w_{MC/DC}(t,p) = \bigcup_{M \in \pi^c_{MC/DC}(C)} \{\{t\big[ite(\sigma(m), ite(C,T,E), fail)\big]_p : m \in M\}\}$$

eine Menge von Mengen von Termen, in denen jeweils das zu testende Konditional eingebettet wurde, und σ ist die o.g. Substitution, die die Inverse der Abstraktionsabbildung für Bedingungen ist, und

- die Menge aller ersetzten Terme $\{T'_1, \ldots, T'_n\}$, wobei T'_i eine Menge ist, die für jede alternative MC/DC-Belegung der entsprechenden Bedingung eine Menge von Ausdrücken enthält, deren Auswertung MC/DC für das entsprechende Konditional erfüllt, ist

$$\pi^e_{MC/DC}(t) = \bigcup_{p \in \mathcal{O}(t) \wedge hd(t|_p) = ite} \{\pi^w_{MC/DC}(t,p)\}.$$

Symbolische Ausführung der so modifizierten funktionalen Ausdrücke liefert dann Belegungen der Variablen, so daß MC/DC für das entsprechende Konditional erfüllt wird.

Funktionsdefinitionen Die funktionale Sprache von AutoFocus sieht m Definitionen $d_1, \ldots, d_m$ der Form

$$f(\vec{a}_1) = rhs_1, \ldots, f(\vec{a}_m) = rhs_m$$

für n-stellige Funktionen mit $n \geq 1$ für Argumenttupel $\vec{a}_i = a_{i1}, \ldots, a_{in}$ vor. Die intuitive Bedeutung des Pattern Matching ist, daß zur Laufzeit die erste „passende“ Definition ausgewählt wird (Abschnitt 4.3.3), die erste Definition also, deren Formalparameter mit den Aktualparametern unifizierbar sind. Das motiviert das Rewriting von Funktionsdefinitionen in eine einzelne Funktion $f(\vec{A}) = \varphi(d_1)$ vermöge einer Abbildung φ für $1 \leq i \leq m$

$$\varphi(d_i) = ite\big(unif(\vec{a_i}, \vec{A}), mgu_{(\vec{a_i},\vec{A})}(rhs_i), \varphi(d_{i+1})\big) \tag{4.7}$$

mit einem Tupel $\vec{A} = A_1, \ldots, A_n$ frischer Variablen und $\varphi(d_{m+1}) = fail$. Zur Laufzeit werden die A_i an die Aktualparameter von von f gebunden. $unif(t_1, t_2)$ entscheidet die Unifizierbarkeit von t_1 und t_2; $\lambda s.mgu_{(t_1,t_2)}(s)$ berechnet den allgemeinsten Unifikator von t_1 und t_2 und wendet ihn auf s an.[12] Natürlich ist diese Entscheidung einigermaßen beliebig. Die folgende Definition, die die Unifikation für jedes einzelne Argument explizit macht, ist ebenfalls berechtigt.

$$\varphi(d_i) = ite\Big(\bigwedge_{j=1}^{n} unif(a_{ij}, A_j), mgu_{(\vec{a_i},\vec{A})}(rhs_i), \varphi(d_{i+1})\Big) \tag{4.8}$$

ist äquivalent zu 4.7, beeinflußt aber offensichtlich die Anzahl notwendiger MC/DC-Testfälle. Das liegt an der zugrundegelegten Definition von $unif$. Die

[12] Die Ersetzung des Matching durch Unifikation ist notwendiger Teil der Übersetzung zum Zweck der symbolischen Ausführung.

Erweiterung auf positionsweise Unifikation ist nicht möglich, weil die Anzahl der Positionen zur Übersetzungszeit nicht bekannt ist.

Als Beispiel für die erste Interpretation sei die Boolesche Äquivalenz, definiert durch *eq* mit $d_1 : eq(\top,\top) = \top$, $d_2 : eq(\bot,\bot) = \top$ und $d_3 : eq(X,Y) = \bot$ betrachtet. Mit der Notation T für $(\top,\top)$ und $F = (\bot,\bot)$ werden diese Definitionen zu

$$\varphi(d_1) = ite\Big(unif(T,\vec{A}), \top, ite\big(unif(F,\vec{A}), \top, ite\big(unif((X,Y),\vec{A}), \bot, fail)\big)\Big),$$

was sofort zu

$$\varphi(d_1) = ite\Big(unif(T,\vec{A}), \top, ite\big(unif(F,\vec{A}), \top, \bot)\Big)$$

vereinfacht werden kann.

Für die so modifizierten Funktionen kann dann eine Menge von MC/DC-Testsuiten berechnet werden, wie das im vorherigen Abschnitt beschrieben ist. Dabei muß bei der Berechnung von $\pi^c_{MC/DC}$ für die Negation von *unif* die Ungleichheit $\neq^{pm}$ verwendet werden. Die Gründe dafür werden im Kontext der Zustandsspeicherung in Abschnitt 5.2 (Gleichung 5.6) ausführlich diskutiert.

Transitionen Im folgenden werden atomare Komponenten mit $p \geq 1$ Eingabekanälen, $q \geq 1$ lokalen Variablen und $r \geq 1$ Ausgabekanälen betrachtet (der Fall $\{p,q,r\} \cap \{0\} \neq \emptyset$ wird ähnlich behandelt). Jede Transition der angeschlossenen EFSM ist von der Form

$$\vec{I} = \vec{\iota} : g : \vec{V}' = \vec{v} : \vec{O} = \vec{o}$$

mit der folgenden intuitiven Bedeutung. Unter der Voraussetzung, daß ein Matching zwischen den aktuellen Eingaben $\vec{I} = I_1,\ldots,I_p$ mit gegebenen Werten $\vec{\iota} = \iota_1,\ldots,\iota_p$ erfolgen kann, wird geprüft, ob der $\vec{I}$ und die lokalen Variablen $\vec{V} = V_1,\ldots,V_q$ involvierende Wächter g zu $\top$ evaluiert. Wenn das der Fall ist, werden den lokalen Variablen $\vec{V}'$ und den Ausgabekanälen $\vec{O} = O_1,\ldots,O_r$ neue Werte zugewiesen, nämlich $\vec{v}$ und $\vec{o}$. Im folgenden wird davon ausgegangen, daß idle-Transitionen explizit gegeben sind. In Anlehnung an die obige Ersetzung im Fall von Funktionsdefinitionen wird jede Transition zu

$$t(\vec{I},\vec{V},\vec{V}',\vec{O}) = ite\Big(unif(\vec{\iota},\vec{I}), mgu_{(\vec{\iota},\vec{I})}\big(ite(g,a,st(\vec{I},\vec{V}))\big), st(\vec{I})\Big), \qquad (4.9)$$

wobei die Zuweisung $a = \big(\vec{V}' = \vec{v} : \vec{O} = \vec{o}\big)$ Ausgabekanälen und lokalen Variablen neue Werte zuweist. Die Funktion *st* speichert die Werte ihrer Argumente und schlägt dann fehl, ist also eine Erweiterung der oben verwendeten Funktion *fail*. Die Motivation ist, daß bestimmte Eingaben von einer Transition nicht verarbeitet werden, sondern u.U. von einer anderen: Input-Enabledness garantiert, daß für jedes Eingabemuster und jeden Wächter eine solche Transition existiert. Deshalb müssen die korrespondierenden Werte für $\vec{I}$ für alle anderen vom selben Kontrollzustand ausgehenden Transitionen ausprobiert werden. Dasselbe Problem ergibt sich für u.U. nicht abgedeckte Teile von Funktionsdefinitionen,

die mit einer identischen Funktion *st* detektiert werden können. Um eine Testsuite zu berechnen, die MC/DC nicht nur auf Transitions-, sondern auch auf Funktionsdefinitionsebene erfüllt, wird überprüft, welche Transitionen möglicherweise indirekt Funktionen aufrufen, die nicht abgedeckt wurden. Für diese Funktionen wird der fehlende MC/DC-Testfall erzwungen (indem beispielsweise die anderen Definitionen ignoriert werden, was möglich ist, wenn Funktionen nur einmal aufgerufen werden), was zu Testfällen mindestens einer Transition führt, die die fehlenden Definitionen abdecken.

Für den funktionalen Ausdruck in Gleichung 4.9 können dann MC/DC-Testsuiten berechnet werden. Da Quell- und Zielkontrollzustand durch die EFSM eindeutig festgelegt sind, ergeben sich durch die Berechnung Einschränkungen des Quell- und Zieldatenzustands sowie für Ein- und Ausgabekanäle. Bei der Negation von *unif* muß die Ungleichheit $\neq^{pm}$ verwendet werden (Gleichung 4.5, zur Begründung siehe Anhang B.3, Gleichung B.13).

Beispiel 22 (MC/DC für Transition). *Für die Transition* `c?Cmd:is_PSOComputeDigSig(Cmd)::cv!Cmd` *der Komponente* `WIMPre` *ergeben sich drei MC/DC-Belegungen für* `Cmd` *und damit Testfälle. Ein- und Ausgaben sind identisch, weil die Komponente als Demultiplexer fungiert:*

- <`no_msg, no_msg`>*,*
- <`VerifyCheck(PinG), VerifyCheck(PinG)`>*mit der Konstanten* `PinG` *und*
- <`PSOComputeDigSig(B), PSOComputeDigSig(B)`> *für eine nicht eingeschränkte Variable* `B`*.*

Die ersten zwei Fälle werden von anderen Transitionen behandelt.

EFSMs Die Testfälle, die aus der Transformation 4.9 für Transitionen abgeleitet wurden, dienen dann als Grundlage für die Berechnung von Testsuiten für diejenige EFSM, die die Transition beinhaltet. Vom Initialzustand ausgehend, werden Traces zu einem Zustand gesucht, der der Vorbedingung der zu feuernden Transition entspricht, sog. Präambeln. Wenn die durch einen Testfall induzierte Vorbedingung einer Transition nicht erreicht werden kann, liegt das entweder daran, daß der entsprechende Zustand nicht erreichbar oder die Suche nicht ausreichend effizient ist. Es wird dann probiert,

- durch Backtracking in der symbolischen Ausführung die Anwendung anderer Funktionsdefinitionen zu erzwingen und die entsprechend anderen Vorbedingungen zum Zielzustand der Suche zu machen oder
- das Verfahren für eine andere Testsuite der Transition zu iterieren.

Beispiel 23 (MC/DC-Suite für eine EFSM). *Abb. 4.17 zeigt die EFSM der Komponente* `WIMPre`*, die auch die Transition der letzten beiden Beispiele enthält. Für diese Komponente werden mit dem vorgestellten Verfahren automatisch 11 Testfälle erzeugt, die MC/DC-Abdeckung der Zustandsmaschine erzwingen, insofern das möglich ist. Die minimale Suite besteht aus 9 Testfällen.*

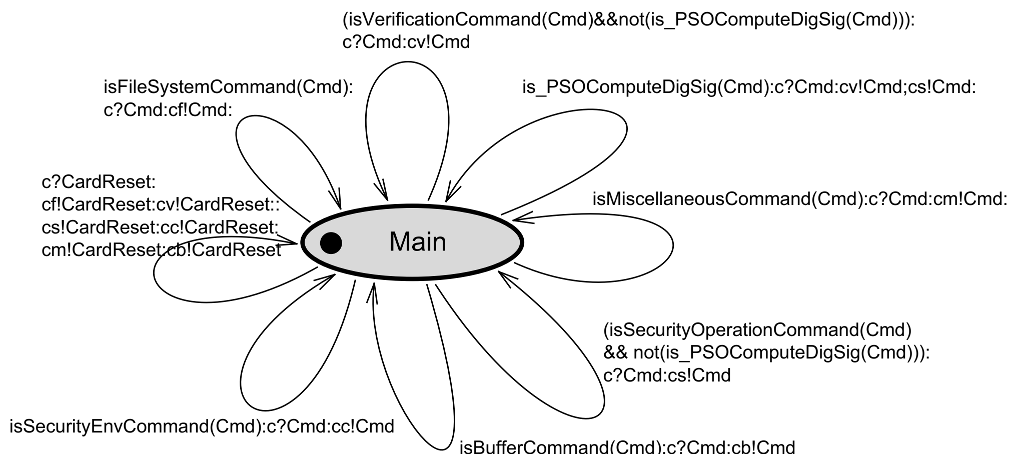


Abbildung 4.17.: Komponente `WIMPre`

Für aggregierte Subsysteme werden dann Testfälle mit dem in Abschnitt 4.6 vorgestellten Verfahren generiert. Eine experimentelle Evaluierung des Vorgehens findet sich in Abschnitt 5.4; ein Beispiel wurde bereits auf S. 131 angegeben. Ein komplexeres Beispiel ist das folgende:

Beispiel 24 (Kompositionalität und MC/DC). *Für die Berechnung der digitalen Signatur muß in der Komponente* `ComputeDigSig`*, die eine Unterkomponente von* `SecurityOperations` *ist, die folgende Funktion so aufgerufen werden, daß alle Konditionale zu true evaluieren (das wird durch die Transformation zur MC/DC-Berechnung erzwungen).*

```
Res = computeDSOutput(B, sE1(seWtls(dst(fS(eF_4451),kr(R),B1,B2),Ct,Cct)),
                Pg, Pnr) =
        (if isWtlsKeyReference(R)
            then (if Pg == verified
                     then resComputeDigSig(eF_4451,R,B)
                     else h6982
                  fi)
             else h6A88
         fi)
```

Variablen beginnen mit Großbuchstaben. Der erste Parameter, B, ist der zu signierende Bytestring. Der zweite Parameter ist die Datenstruktur einer Sicherheitsumgebung, und zwar der Sicherheitsumgebung SE1 mit PIN-Status (Pg,Pnr) und gesetztem digitalen Signaturtemplate, das verschiedene Parameter wie selektiertes File (eF_4451), Schlüssel (R) und Digest enthält. Symbolisches Ausführen dieser Funktion liefert drei symbolische Testfälle:

- $R = key1Ref,\ Pg = verified,$
  $Res = resComputeDigSig(eF_4451, key1Ref, B),$
- $R = key1Ref,\ Pg \neq verified,\ Res = h6982$ *und*

- $Res = h6A88,\ R \neq key1Ref$.

Nur Fall (1) liefert das gewünschte Resultat, das eine Evaluierung aller Bedingungen zu true erzwingt.

In einem zweiten Schritt wird nun versucht, zu erzwingen, daß eine entsprechende Transition die Funktion mit den entsprechenden Werten aufruft. In der Zustandsmaschine gibt es ohne Idle-Transition 8 Transitionspfeile, von denen 2 die Funktion mit entsprechenden Aktualparametern aufrufen. Die Zustandsmaschine besteht nur aus einem Kontrollzustand ohne Datenzuständen. Die relevanten Transitionen sind
`c?pSOComputeDigSig(B); se?E; pg?P; pnr? :`
`r!computeDSOutput(B,E,verified,unverified)` *und*
`c?pSOComputeDigSig(B); se?E; pg?P; pnr?Q :`
`r!computeDSOutput(B,E,verified,verified)`*, die sich in bezug einen der Eingabekanäle unterscheiden, nämlich den, der den Status der PIN-NR liefert (Anwesenheit eines Signals bedeutet, daß die PIN verifiziert ist; Abwesenheit, daß sie es nicht ist). Daraus wird ein Testfall für die übergeordnete Komponente,* `SecurityOperations` *berechnet. Dazu kann das inkrementelle Verfahren angewendet werden, worauf hier aus Gründen der Übersichtlichkeit verzichtet wird. Alternativ können auch direkt Testfälle für diese Komponente berechnet werden, indem die aus den Transitionen resultierenden Werte für Kanäle und lokale Variablen direkt in dasjenige Prädikat eingesetzt werden, das* `SecurityOperations` *entspricht. Zwei Testfälle resultieren, für jede der oben angegebenen Transitionen einer.*

Für beide Transitionen sind (der durch die internen Kanäle der Komponente definierte Datenzustand wird hier ignoriert) die Ein- und Ausgaben an die Komponente

```
cs?pSOComputeDigSig(_); pg?_;
se?sE1(seWtls(dst(fS(eF_4451), kr(key1Ref), _, _), _, _));
rs!resComputeDigSig(eF_4451,key1Ref,_),
```

wobei _ *eine beliebige Variable darstellt, und der Kanal* `pnr` *wird im ersten Fall mit einem Wert belegt (*`pnr?_`*), und im zweiten mit keinem (*`pnr?`*).*

Daraus können nun wiederum Testfälle für die angeschlossenen Komponenten berechnet werden. Alternativ kann auch für das Gesamtsystem durch symbolische Ausführung direkt ein entsprechender Testfall gefunden werden. Für die Situation `pnr?` *ergibt sich der in Abb. 4.18 angegebene Trace; ein Testfall für das Modell sind dann nur die Eingaben an das System und seine Ausgaben (in der Abb. die von rechts in das Bild kommenden Pfeile und die nach rechts das Bild verlassenden Pfeile). Das letzte Kommando wurde manuell eingefügt; es fordert den sich im* `ResponseBuffer` *befindlichen Wert an. Das erste Kommando ist für den Testfall nicht relevant. Sein Auftreten ist durch die zufällige Suche im Testfallgenerator begründet. Im Beispiel wird der symbolische Wert* `BytesB` *signiert; das ist ein Resultat der zufälligen Instantiierung des symbolischen Testfalls.*

Zusammenfassend ist festzuhalten, daß die geschichtete Generierung von MC/DC erfüllenden Testsuiten auf einer Generierung von *Testfallspezifikationen* beruht. Das automatische Rewriting von Funktionsdefinitionen und Tran-

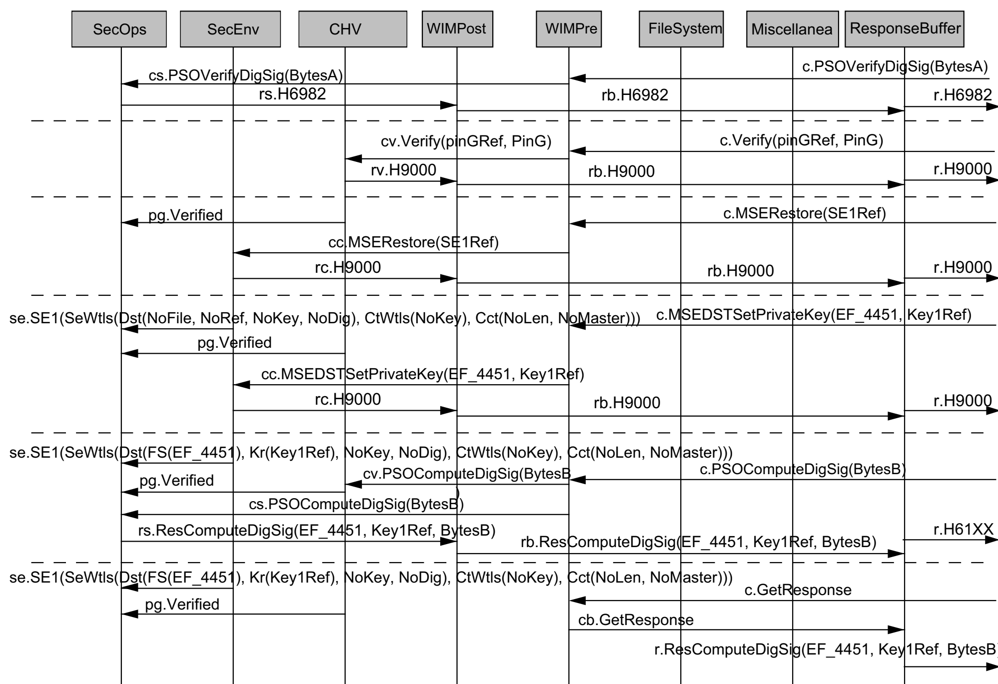


Abbildung 4.18.: Trace für digitale Signaturberechnung

sitionen führt genau zu solchen Testfallspezifikationen, für die dann – ebenfalls automatisch – Testfälle generiert werden können.

Diskussion

Die Mächtigkeit der kompositionalen Testfallgenerierung ist natürlich durch die Komplexität des Systems beschränkt.

- Wenn in Funktionsdefinitionen oder Wächtern von Transitionen Konditionale der Art $f(X) = g(Y)$ und die Gleichung nichtlinear oder auf anderen Datentypen als Zahlen operiert, dann wird die Lösung der Gleichung durch Aufzählen aller möglichen Paare X, Y gelöst. Wenn die zugrundeliegenden Datentypen sehr groß sind oder f und g komplexe rekursive Abhängigkeiten aufweisen, dann kann das Verfahren u.U. die Lösung nicht mehr effizient berechnen.

- Bei Berechnung eines Testfalls für eine Transition wird ein Quellzustand berechnet. Es hängt dann von der Komplexität der EFSM und der Zuweisungen in den einzelnen Transitionen ab, ob dieser Zustand z.B. mit den Verfahren der gerichteten Suche (s. Kap. 5) mit symbolischer Ausführung gefunden wird.

- Dasselbe Argument gilt auch für komponierte Systeme. Wenn die einzelnen Zustände der Testsequenz für eine Komponente „schwierig“ zu finden

sind, dann ist die Berechnung einer integrierten Testsuite u.U. nicht effizient machbar.

In solchen Fällen kann der Benutzer dem System Hinweise geben, wie der Suchraum verkleinert oder vereinfacht werden kann. Das geschieht mit Constraints (Abschnitt 5.3). Das Beispiel der Chipkarte stimmt optimistisch, daß die Verfahren zumindest für vergleichbar komplexe Systeme effizient arbeiten.

4.8. Verwandte Arbeiten

Synchrone Programmiersprachen Die Verwendung synchroner Programmiersprachen bietet sich wegen der einfachen Ausführungssemantik zur Generierung von Testfällen an. Das Werkzeug Lurette (Raymond u. a., 1998) generiert Testfälle für Lustre-Spezifikationen (Caspi u. a., 1987), die als black boxes betrachtet werden. Die Idee besteht darin, „relevante“ Verhalten in einem Beobachterknoten zu codieren (die Testfallspezifikation; vgl. die EFSM zur Berechnung von digitalen Signaturen). Beobachterknoten bestehen im wesentlichen aus Constraints über Booleschen Werten und reellen Zahlen; zufällige Instantiierungen dienen dann der Testfallerzeugung.

Lutess (du Bousquet u. a., 2000) verfolgt eine ähnliche Idee zum Testen von Invarianzeigenschaften. Ein- und Ausgabetypen sind auf Boolesche Werte beschränkt; der Beobachterknoten (die Testfallspezifikation) wird als BDD codiert. In jedem Schritt wird ein die durch dieses BDD gegebenen Constraints erfüllender Wert ausgewählt. Das geschieht mit diversen Strategien, die eigenschaftsbasiert, randomisiert oder stochastisch definiert sind.

Gatel (Marre und Arnould, 2000) berechnet Testfälle für Lustre-Programme, die als Constraints codiert werden und somit direkt die Datenflußgleichungen des Programms widerspiegeln. Verschiedene Heuristiken (basierend u.a. auf der Kardinalität der involvierten Typen und der Anzahl erzeugter Choicepoints) werden für die Entscheidung verwendet, welche Constraints wann zu instantiieren sind. Strukturierte Typen werden nicht unterstützt.

SDL, LOTOS, Z, FSMs Das TGV-Werkzeug (Fernandez u. a., 1996) berechnet Testsequenzen für LOTOS- und SDL-Spezifikationen, die in beschriftete I/O-Transitionssysteme (IOLTS) übersetzt werden. Im wesentlichen wird das synchrone Produkt von Testfallspezifikation und System traversiert, um dieses Produkt mit Verdikten und Zählern zu annotieren. Resultat ist ein Testgraph, der Testsequenzen und die entsprechenden Verdikte repräsentiert. Datenzustände müssen explizit in Zustände des IOLTS übersetzt werden. Rusu u. a. (2000) erweitern IOLTS deshalb um Variablen, d.h. Datenzustände (IOSTS). Testfälle werden dann wie in TGV erzeugt.

Wie TGV basieren einige der an die Testgenerator-Architektur TorX (Vries u. a., 2000) angeschlossenen Testfallgeneratoren auf der ioco-Testtheorie von Brinksma (1988). TorX gestattet Black-Box-Testing von SDL-, LOTOS- und Promela-Spezifikationen. Der Algorithmus für die on-the-fly-Generierung von Testfällen ist insofern ähnlich zu Lutess oder Lurette, als das Produkt von Testfallspezifikation und System traversiert werden.

Die Verwendung von IOLTS als Zwischensprache basiert in den zitierten Arbeiten auf der expliziten Berechnung des Zustandsraums als Produkt diverser Komponenten des Systems und der Testfallspezifikation. Zumindest im Fall von IOSTS führt das zu einer großer Zahl statisch bestimmbarer unerfüllbarer Transitionen, die während der Zustandsraumexploration alle überprüft werden müssen. Rusu u. a. (2000) propagieren deshalb die Verwendung von PVS und HyTech zur automatischen Berechnung von Invarianten, die den statischen Ausschluß solcher unerfüllbarer Transitionen gewährleisten können.

Koch u. a. (1998) beschreibt ein Werkzeug zur Erzeugung von Testfällen aus SDL-Spezifikationen, das auf der Suche nach Zuständen beruht. Für den praktischen Einsatz unabdingbare Techniken wie Zustandsspeicherung und gerichtete Suche (Kap. 5) werden nicht diskutiert.

Bourhfir u. a. (1996) präsentieren diverse Verfahren zur Berechnung von Testfällen aus Zustandsmaschinen. Chow (1978) beispielsweise zeigt, wie unter der Voraussetzung, daß eine obere Schranke für die Zustandszahl bekannt ist, vollständige Testsuiten für black-box-Testen erzeugt werden können (die sog. charakterisierende Mengen- oder W-Methode; vgl. dazu auch den Ansatz des Black-Box-Checking (Peled u. a., 1999) sowie die verschiedenen klassischen formalen Testmethoden zum Conformancetest deterministischer Zustandsmaschinen (Ural, 1992)). Im Bereich der Prozessorvalidierung verwenden u.a. Shen und Abraham (1999), Dushina u. a. (2001) und Fournier u. a. (1999) endliche Zustandsmaschinen – z.B. automatisch von existierendem VHDL-Code abstrahiert –, um Testfälle zu erzeugen. Dushina u. a. (2001) beschreiben ebenfalls, wie abstrakte Testfälle für die Maschinenebene konkretisiert werden. Dabei werden genau wie im Fall der Chipkarte im wesentlichen abstrakte Kommandos durch (Sequenzen von) Kommandos ersetzt, d.h. es findet eine Makroexpansion statt. Die Zwischenschicht des Testtreibers, der nicht im Modell vorhandene Informationen einfügt (Schlüssel, Zufallszahlen) ist dort nicht notwendig.

Die Verwendung des Theorembeweisers Isabelle zur Generierung von Testfäl--len aus Z-Spezifikationen (Spivey, 1992) wird u.a. von Helke u. a. (1997); Sadeghipour und Singh (1998); Burton u. a. (2001) diskutiert. Die Interaktivität des Beweisers läßt die Frage nach dem praktischen Einsatz solcher Methoden aufkommen.

Symbolische Ausführung Logische Variablen sind eine natürliche Grundlage für die symbolische Ausführung zum Zweck der Testfallgenerierung. Denney (1991) beispielsweise abstrahiert von Rekursion in Prolog-Programmen und definiert heuristisch Äquivalenzklassen auf dem resultierenden abstrakten Flußgraphen, der seinerseits zur Grundlage der Testfallgenerierung mit symbolischer Ausführung gemacht wird. Da Prolog ungetypt ist, werden Typen nicht berücksichtigt. Das ist allerdings im Werkzeug QuickCheck (Claessen und Hughes, 2000) der Fall, in dem im wesentlichen für Haskell-Programme auf der Basis einer externen Spezifikation des Programms randomisiert Testfälle berechnet werden.

Symbolische Ausführung für die Testfallgenerierung war in den Siebzigern populär (King, 1976; Clarke, 1976; Howden, 1977, 1978a; Ramamoorthy u. a.,

1976). Wegen der damals nicht ausreichend mächtigen Constraintlösertechnologie benötigen diese Systeme häufig Hilfe in Form beispielsweise vorspezifizierter Programmpfade, für die dann entsprechende Daten generiert werden.

Meudec (2000) benutzt CLP zur symbolischen Ausführung von Spark-Ada-Code (Barnes, 1997), um Testfälle zu erzeugen. Der Fokus ist weder auf reaktive noch auf nebenläufige Systeme – in der Tat ist eine der Einschränkungen von Spark die, daß Tasks nicht unterstützt werden.

Legeard und Peureux (2001) verwenden einen Constraintlöser zur Berechnung von Testfällen aus nach CLP übersetzten B-Spezifikationen (Abrial, 1996). Zustandsspeicherung und gerichtete Suche scheinen keine Verwendung zu finden. Das gilt auch für zum Zweck der Testfallgenerierung direkt in CLP codierte hybride Systeme bei Ciarlini und Frühwirth (1999b,a) sowie Ciarlini und Frühwirth (2000), die dann symbolisch ausgeführt werden.

Andere Suchverfahren Tracey (2000) und Wegener (2001) verwenden genetische Algorithmen zur Generierung von Testfällen. Eine – praktisch notwendige – Zustandsspeicherung zur Einschränkung des Suchraums bei Programmen mit Schleifen (Kap. 5) findet nicht statt. Da genetische Algorithmen auf Chromosomen einer fixen Länge basieren, ist die Übertragung auf reaktive Systeme – Sequenzen von Testvektoren anstelle einzelner Testvektoren – nicht sofort offensichtlich. Da zum Zweck der Reduktion des Suchraums i.a. auch für das in dieser Arbeit präsentierte Verfahren eine maximale Suchtiefe vorgegeben wird, besteht ein erster Ansatz darin, diese maximale Länge als Chromosomenlänge zu definieren und u.U. nach der Testfallgenerierung überflüssige Teilsequenzen am Ende abzuschneiden. Inwieweit das praktikabel wäre, muß sich zeigen.

Kompositionale Techniken zur Testfallgenerierung sind nach Wissen des Verfassers nicht publiziert. Harrold und Soffa (1991) erweitern datenflußbezogene Abdeckung auf interprozedurale Abhängigkeiten zum Zweck des Integrationstests.

Coverage Die automatisierte Generierung von Testsuiten, die ein Coveragekriterium erfüllen, wird von Wegener u. a. (2002) auf der Basis des evolutionären Testens präsentiert. Der Ansatz ist nicht kompositional und zunächst auf transformative Systeme beschränkt. Hong u. a. (2001) übersetzen Coveragekriterien auf Transitions- und Zustandsebene in Formeln, für die dann Model Checking stattfindet. Es ist nicht sofort klar, wie eine Übertragung auf kompliziertere Coveragekriterien wie MC/DC stattfinden kann; der Ansatz ist darüberhinaus ebenfalls nicht kompositional. Auch Rayadurgan und Heimdahl (2001) präsentieren ein Verfahren zur Berechnung coveragebasierter Testsuiten auf Unit-Ebene, das auf Model Checking basiert und gerichtete Suchverfahren nicht berücksichtigt. Die Berechnung von Integrationstestfällen wird nicht betrachtet.

Dupuy und Leveson (2000) geben eine empirische Evaluierung von MC/DC in bezug auf das Fehlererkennungspotential und zeigen für ihre Anwendung, daß MC/DC-Testfälle Fehler gefunden haben, die mit funktionalen Tests nicht entdeckt wurden.

4.9. Zusammenfassung

Inhalt dieses Kapitels ist die symbolische Ausführung von AutoFocus-Modellen zum Zweck der Testfallgenerierung. Es wird gezeigt, wie

- die verschiedenen AutoFocus-Beschreibungstechniken aussehen und kombiniert werden,
- AutoFocus-Modelle automatisch in CLP-Programme übersetzt werden,
- durch Ausführung dieser Programme aus Modellen und Testfallspezifikationen Äquivalenzklassen von Testfällen (Traces des Modells) erzeugt werden,
- diese Äquivalenzklassen zu jedem Zeitpunkt unterschiedlich instantiiert werden können und insofern eine Erweiterung klassischer Grenzwertanalysen von transformativen auf reaktive Systeme stattfindet,
- der kompositionale Charakter von AutoFocus-Modellen zur kompositionalen Erzeugung von Testfällen auf Ebenen bis hin zum Integrationstest verwendet wird, was u.a. die verzahnte Entwicklung von Testfällen und Modellinkrementen (Abschnitt 2.3.4) ermöglicht, und
- am Beispiel des MC/DC-Coveragekriteriums zunächst automatisch Testfallspezifikationen, daraus dann Testfälle für die atomaren Elemente (Transitionen, Funktionsdefinitionen) und daraus endlich mithilfe der kompositionalen Technik inkrementell Testfälle für Subsysteme und schließlich das Gesamtsystem berechnet werden.

Die Übersetzung von AutoFocus-EFSMs nach CLP gestaltet sich aufgrund der simplen zeitsynchronen Ausführungs- und Kommunikationssemantik vergleichsweise trivial. Für die symbolische Ausführung funktionaler Programme muß etwas Aufwand betrieben werden; das liegt im wesentlichen daran, daß das Konzept der Negation in verfügbaren CLP-Systemen aus Effizienz- und theoretischen Gründen zumeist insofern eingeschränkt ist, als die – zumeist unendliche – Menge der Lösungen, die eine Anfrage nicht erfüllt, i.a. nicht berechnet werden kann. Für die Zwecke der Testfallgenerierung ist es möglich, mithilfe spezieller Constraint-Handler für Ungleichheiten auf SLDNF-Resolution (Lloyd, 1993) oder konstruktive Negation (Stuckey, 1991) zu verzichten. Dazu wurden Constrainthandler definiert, die sich direkt in Constraint Handling Rules übersetzen lassen.

Die kompositionale Generierung von Testfällen bis hin zur automatisierten Generierung von Integrationstests und die automatische Generierung von Testfallspezifikationen, die zum Zweck der Generierung von MC/DC erfüllenden Testfällen verwendet werden können, stellen die wesentlichen Neuerungen dar, die im Rahmen dieses Kapitels präsentiert werden. Ohne die im folgenden Kapitel geschilderten effizienzsteigernden Maßnahmen sind sie wegen der Explosion des Zustandsraums in der Praxis nicht anwendbar. Die einfache zeitsynchrone Semantik und die klare Kapselung von Kommunikation und lokalen

Variablen scheinen wesentlich für den Erfolg der Techniken zu sein. Wenn zeitasynchrone Kommunikation mit expliziten Pufferkomponenten eingeführt wird, wird der Suchraum u.U. zu groß und muß durch geeignete domänenabhängige Abstraktionen reduziert werden.

Weitere Arbeiten bieten sich in den folgenden Bereichen an:

- Im Unterschied zu Testfallgeneratoren auf der Basis von Model Checkern wird das Produkt der EFSMs eines Modells nicht explizit gebildet. Inwieweit das nach entsprechenden Datenflußanalysen reduzierte explizite Produkt effizienter zur symbolischen Ausführung verwendet werden kann, bleibt zu untersuchen.

- Die generierten MC/DC-Suiten weisen als Resultat ihrer Erzeugung Redundanz auf. Ideen für effiziente Algorithmen zur Detektion und Eliminierung dieser Redundanz – z.B. durch ex-post-Filtern oder Rückgriff auf Wissen aus der Generierung früherer Testfälle – ergeben sich sofort, sind aber in der Praxis zu evaluieren (s. z.B. Jones und Harrold (2001)).

- Im Rahmen der Erzeugung von Integrationstests ist der Einfluß der Reihenfolge, in der Subsysteme integriert werden, zu untersuchen.

- Betrachtet wurden hier nur kontrollflußbezogene Überdeckungsmaße. AutoFocus mit Beschreibungstechniken, die keine Seiteneffekte zulassen, bietet sich direkt zur automatischen Generierung datenflußbezogener Testfallspezifikationen an (Abschnitt 3.3) an. Erneut liegt der Vorteil solcher struktureller Kriterien in der automatisierbaren Erzeugung von Testfallspezifikationen und damit u.U. der kostengünstigen Erzeugung großer Mengen von Testfällen.

- Mit der Verfügbarkeit automatischer Testfallgeneratoren bietet sich eine Evaluierung von SW-Komplexitätsmaßen wie der zyklomatischen Komplexität von McCabe (1976) an:[13] Inwiefern korreliert die – z.B. durch erforderliche Rechenzeit quantifizierbar zu definierende – Schwierigkeit, Testfälle abzuleiten mit der Komplexität des zu testenden Artefakts? Im Bereich des evolutionären Tests transformativer Systeme gibt es erste Indizien, daß erhöhte Systemkomplexität nicht mit erhöhter Schwierigkeit der Testfallgenerierung einhergeht (Wegener, 2002).

- Die Übersetzung funktionaler Programme in flache CLP-Programme kann umgangen werden, wenn stattdessen funktional-logische Sprachen wie Curry – ergänzt um Constraints – mit einer lazy operationalen Auswertungssemantik (Antoy u. a., 1994; Hanus, 2000) verwendet werden. Die noch mangelnde Verfügbarkeit effizienter Compiler motivierte die Verwendung von CLP.

[13] Watson und McCabe (1996) definieren Testfallspezifikationen anhand des für die zyklomatische Komplexität wesentlichen Konzepts einer *Basis* aller Programmpfade: eine minimale Menge linear unabhängiger Pfade, die alle Anweisungen abdeckt.

- Schließlich stellt sich die Frage nach einer Übertragung der Technik auf andere Beschreibungsmittel. Der Testfallgenerator ist zusammen mit den im folgenden Kapitel vorgestellten Effizienzsteigerungen u.a. deshalb erfolgreich einsetzbar, weil die Semantik des zugrundeliegenden Formalismus sehr einfach ist (zeitsynchron, nachrichtenasynchron, keine globalen Variablen). Symbolische Ausführung ist unabhängig von der Semantik der zugrundeliegenden Beschreibungstechnik. Demzufolge kann eine Variante des Testfallgenerators natürlich für andere Sprachen wie Java oder Statecharts eingesetzt werden. Inwiefern das effektiv und effizient durchführbar ist, ist zu untersuchen.

5. Effizienzsteigerung und Anwendung

Dieses Kapitel beschreibt Strategien, die zum praktischen Einsatz des in Kapitel 4 beschriebenen Testfallgenerators auf der Basis der symbolischen Ausführung notwendig sind.

Es wurde bereits erläutert, daß Testfallgenerierung als Suchproblem aufgefaßt werden kann. Abschnitt 5.1 diskutiert deswegen die Implementierung verschiedener Suchstrategien. Klassische Suchverfahren wie Tiefen- und Breitensuche werden ebenso erläutert wie Wettbewerbsparallelität und die Verwendung heuristischer Suchverfahren, die auf Distanzmaßen im Zustandsraum basieren. Zentrale Inhalte des Abschnitts über Suchverfahren sind von Pretschner (2001) und Pretschner u. a. (2003b) publiziert.

In Abschnitt 5.2 wird erläutert, wie der Suchraum durch Verbieten des mehrfachen Besuchs von Zuständen eingeschränkt werden kann. Im Zug der symbolischen Ausführung wird mit Mengen von Werten gerechnet, woraus sich ergibt, daß anstelle einzelner Zustände immer Mengen von Zuständen besucht werden. Diese Zustandsmengen werden nun symbolisch abgespeichert, d.h. anstelle aller Instanzen der Zustände wird ein diese Menge beschreibendes Prädikat abgespeichert. Im wesentlichen ist dieser Abschnitt in *technischer Hinsicht*

- einerseits eine auf den Zweck der Testfallgenerierung (Überprüfung von EF-Eigenschaften) zugeschnittene Verallgemeinerung des Memoing (Warren, 1992) bzw. der tabulierten Resolution (Cui u. a., 1998) auf die Constraint-Logikprogrammierung und

- andererseits eine Spezialisierung (auf EF-Eigenschaften) von auf tabulierter Resolution basierenden Model Checkern, die auf den Overhead der Implementierung der tabulierten Resolution verzichten kann. Das geschieht durch die Implementierung syntaktischer Charakterisierungen der Inklusionsbeziehung sowie der Differenzbildung von Zustandsmengen. Das präsentierte Verfahren kombiniert die Vorteile von explicit-state und symbolischem Model Checking insofern, als (a) der für die Eigenschaft relevante Teil des Zustandsraums nicht vor Überprüfung einer Eigenschaft aufgebaut werden muß und (b) mit Mengen äquivalenter Zustände gerechnet wird.

Konsequenzen der Vermeidung doppelter Zustandsbesuche für die Testfallgenerierung auf Maschinenebene werden diskutiert. In *methodischer Hinsicht* gibt es neben der Berücksichtigung nicht nur von Modellen, sondern auch Maschinen, einen weiteren wesentlichen Unterschied zwischen Model Checking und Testfallgenerierung. Ziel der Testfallgenerierung ist die Ableitung einer Menge von Traces. Ziel des Model Checking ist der Nachweis oder die Falsifikation einer

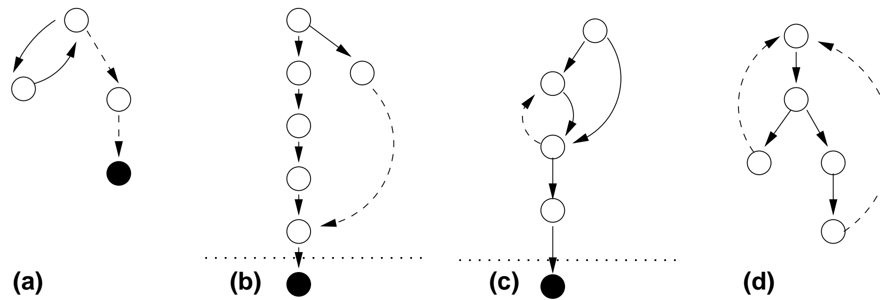


Abbildung 5.1.: Suchstrategien

Eigenschaft. Traces werden dann höchstens zur Fehlerlokalisation im Modell verwendet. Daraus folgt für die Testfallgenerierung die Konzentration auf existentielle Eigenschaften im Gegensatz zu universellen Eigenschaften im Model Checking (Kap. 3).

Zentrale Inhalte des Abschnitts über Zustandsspeicherung sind bei Pretschner und Philipps (2003) publiziert.

Abschnitt 5.3 faßt die Verwendung von Constraints für die Testfallgenerierung zusammen. In Abschnitt 5.4 erfolgt eine experimentelle Bewertung der verschiedenen effizienzsteigernden Verfahren. In Abschnitt 5.5 werden verwandte Arbeiten diskutiert. Das schließt einen kursorischen Vergleich mit den verwandten Techniken des explicit-state Model Checking und des Memoing ein. Abschnitt 5.6 faßt die Ergebnisse dieses Kapitels zusammen.

5.1. Suchstrategien

Im Rahmen der Testfallgenerierung mit symbolischer Ausführung können die zugrundeliegenden Modelle als nichtdeterministisch angesehen werden. Das liegt daran, daß nicht alle Eingaben spezifiziert werden müssen, und das „Raten“ dieser Eingaben ist ein nichtdeterministischer Prozeß. Die Freiheit besteht in bezug auf die nächste auszuwählende Transition oder vielmehr ein Produkt von Transitionen, für jede Komponente eine. Das betrifft nicht nur die Transitionspfeile der assoziierten EFSMs, sondern auch die Auswahl der auszuwählenden Definition der in Wächtern und Zuweisungen vorkommenden Funktionen. Keine der hier zitierten Suchstrategien ist neu. Neu ist allerdings der Einsatz in der Testfallgenerierung für reaktive Systeme bzw. für Verfahren, die den Zustandsraum eines Systems explorieren. Experimentelle Resultate finden sich in Abschnitt 5.4.

Naive Links-Rechts-Auswahl Die einfachste Strategie besteht darin, die Transitionen gemäß der Reihenfolge ihres Auftretens im CLP-Programm auszuwählen. Backtracking erfolgt automatisch durch die im CLP-System zur Verfügung gestellten Mechanismen. Dieses Verfahren ist ohne Erweiterungen insofern ineffizient, als es bereits besuchte Zustände erneut besucht und in Zyklen laufen kann. Abb. 5.1 (a) demonstriert diese Situation, wenn die vom Initialzustand

ausgehende linke Transition als erste im CLP-Programm auftaucht. Offenbar hat die Reihenfolge der CLP-Prädikate Einfluß auf die Effizienz dieser Strategie.

Interleaving von Transitionen Das Zyklusproblem mit der naiven Strategie liegt an einer unfairen Selektion von Transitionen. Wenn $Tr_{s,i} =< t_1, \ldots, t_n >$ die Reihenfolge der Transitionen beschreibt, die beim i-ten Besuch von Kontrollzustand s Anwendung fand, dann kann durch zyklisches Verschieben dieser Reihenfolge das Problem abgemildert werden: $Tr_{s,i+1} =< t_2, \ldots, t_n, t_1 >$ führt beispielsweise dazu daß beim zweiten Besuch des Initialzustands in Abb. 5.1 die gestrichelte Transition ausgewählt wird. Interleaving von Transitionen geschieht auf Transitionspfeilebene, weil sie auf Transitionsebene das Interleaving von Funktionsaufrufen erforderlich machen würde.

Globale Zustandsspeicherung Beide bisher präsentierten Suchstrategien besuchen Zustände u.U. mehr als einmal. Das Verbot solcher doppelten Besuche wird im folgenden Abschnitt 5.2 diskutiert. Hier soll der Einfluß der Suchstrategie diskutiert werden.

Die symbolische Ausführung erfolgt im Normalfall mit einer beschränkten maximalen Tracelänge. Motivation dafür ist die Einschränkung des Suchraums. Das kann dazu führen, daß Zustände, die in weniger Schritten als durch die maximale Tracelänge vorgegeben erreichbar sind, nicht besucht werden. Abb. 5.1 (b) zeigt ein Beispiel. Wenn zuerst der linke Pfad ausgewählt wird, dann wird der schwarze Zustand nicht erreicht. Die gestrichelte Linie repräsentiert die maximale Tracelänge, hier 5. Der Zustand direkt über der gestrichelten Linie wird bei Beschreiten des linken Pfades als besucht markiert und kann deshalb nicht über die gestrichelte Transition des rechten Pfades erreicht werden. Das passiert, obwohl per Backtracking über den rechten Pfad der schwarze Zustand in $3 < 5$ Schritten erreicht werden könnte.

Lokale Zustandsspeicherung Das motiviert eine weitere Strategie zum Speichern von Zuständen. Die Tiefe eines Zustands sei definiert als die Anzahl der Transitionen, die notwendig war, ihn zu erreichen. Zustände dürfen erneut besucht werden, wenn ihre Tiefe zum Zeitpunkt der Speicherung echt unterhalb ihrer Tiefe zum Zeitpunkt des erneuten Besuchs lag. Abb. 5.1 (c) demonstriert das. Die gestrichelte Transition wird nicht ausgeführt, weil die Tiefe ihres Endzustands zum Zeitpunkt des Speicherns 1 war, sie zum Zeitpunkt des erneuten Besuchs allerdings 3 beträgt. Hingegen wird die rechte Transition vom Initialzustand ausgeführt, weil die Tiefe ihres Endzustands zum Zeitpunkt ihres Abspeicherns bei Durchlaufen des linken Pfades 2 betrug, beim erneuten Besuch hingegen 1 ist. Das Problem mit dieser Strategie ist, daß beim erneuten Besuch eines Zustands alle darunterliegenden ebenfalls erneut besucht werden. Es zeigt sich, daß das hohe zeitliche Kosten verursacht. Der benötigte Speicherplatz bleibt gleich; es wird nur die mit einem Zustand gespeicherte Tiefe modifiziert. Da das Problem durch Erhöhung der maximalen Suchtiefe gelöst werden kann und bei der Testfallgenerierung ohnehin mit Unvollständigkeiten gerechnet werden muß, wird diese Strategie im folgenden nicht weiter betrach-

tet. Stattdessen wird globale Zustandsspeicherung betrachtet. Zu beachten ist, daß die erforderliche maximale Suchtiefe nicht berechnet werden kann, ohne den vollständigen Suchraum bis zu einer bestimmten Tiefe abzudecken.

Durch Zustandsspeicherung kann der Erfolg der Berechnung von Testsuiten mit Coveragekriterien beeinflußt werden. Wenn beispielsweise Transitionstouren berechnet werden sollen, dann werden bestimmte Transitionen ausgeschlossen. In Abb. 5.1 (d) etwa wird der Initialzustand genau einmal besucht, und die gestrichelten Transitionen können nicht feuern. Das Problem wird gelöst, indem bei der symbolischen Ausführung Transitionen feuern dürfen, auch wenn sie in einen bereits besuchten Zustand münden. Dieser bereits besuchte Zustand wird dann allerdings als Quellzustand des folgenden Schritts ausgeschlossen.

Gerichtete Suche Wie in Abschnitt 5.4 quantitativ untermauert, spielt die Reihenfolge der ausgewählten Transitionen und der aufgerufenen Funktionsdefinitionen wenig überraschend eine große Rolle für die Effizienz der Suche. Würde in jedem Schritt die „richtige“ Transition ausgewählt, könnte die Testfallgenerierung deterministisch erfolgen. Das Problem ist, daß i.A. nicht zu bestimmen ist, was die „richtige“ Transition ist.[1]

Man kann versuchen, eine „möglichst“ richtige Transition auszuwählen, indem für alle möglichen Transitionen vom aktuellen Zustand aus die Distanz zum Zielzustand berechnet wird. Die Transition mit der geringsten Distanz wird zuerst ausgewählt, die mit der zweitkleinsten als zweites usw.

Zwei Probleme ergeben sich sofort:

- Die Definition eines Distanzmaßes ist gerade im Zusammenhang mit nicht geordneten Typen schwierig.

- Offenbar ist die Hoffnung bei der Anwendung der obigen Strategie, daß sich der kürzeste Pfad zum Zielzustand aus Transitionen zusammensetzt, die jeweils lokal optimal bzgl. der Distanzfunktion sind (Bellmannsches Optimalitätsprinzip). Das ist natürlich normalerweise nicht der Fall, weil (a) die Distanzmaße nur Schätzungen darstellen und (b) greedy Algorithmen nur in Spezialfällen optimal sind. Vielmehr kann es für die Gesamtdistanz zum Zielzustand bisweilen sinnvoll sein, nicht die aktuell „billigste“ Transition zu wählen, sondern eine teurere – lokale Optimierung impliziert nicht globale Optimierung.

Das zweite Problem ist heute nicht allgemein lösbar. Als Heuristik bietet sich das Verfahren allerdings trotzdem an, insbesondere wenn stochastische Abberationen wie im Simulated Annealing zugelassen werden, daß also bisweilen zufällig nicht die billigste Transition ausgewählt wird.

[1] Es ist denkbar, die Suche nicht vom Initialzustand ausgehend vorwärts durchzuführen, sondern vom gesuchten Zustand aus rückwärts. Im zweiten Fall wird nach dem Initialzustand gesucht. Da Initialzustand und gesuchter Zustand sich nicht prinzipiell unterscheiden, ist es unwahrscheinlich, daß sich die beiden Suchverfahren in ihrer Effizienz unterscheiden. Auf einen experimentellen Beleg dieser These wird hier verzichtet.

Ausdruck	Distanz
$x \leq c$	$x - c$
$x < c$	$x - c + \varepsilon$
$x = c$	$abs(x - c)$
$x \neq c$	$\varepsilon - abs(x - c)$
$\neg p$	$-c_p + \varepsilon$
$p \vee q$	$min(c_p, c_q)$
$p \wedge q$	$max(c_p, c_q)$
$p \wedge q$	$c_p + c_q$
$p \vee q$	$\frac{c_p \cdot c_q}{c_p + c_q}$

Tabelle 5.1.: Beispiele für Distanzmaße

Dynamische gerichtete Suche Das erste Problem ist ebenfalls ein schwieriges. In der Praxis zeigt sich allerdings, daß die folgenden Metriken oder Kombinationen davon (Pretschner und Philipps, 2001) häufig adäquat sind. Wenn geordnete Typen mit natürlichem Distanzmaß wie beispielsweise Zahlen verwendet werden, kann die Differenz zwischen aktuellem und gewünschtem Wert als Basis für die Distanzberechnung herangezogen werden. Bei mehreren relevanten Werten bietet sich die Verwendung gewichteter Summen an. Kontrollzustände sind (komponentenlokal) über die sie verbindenden Transitionen ebenfalls relativ einfach zu ordnen, nämlich durch die minimale Pfadlänge, gemessen in Anzahl der Transitionspfeile, zwischen zwei Kontrollzuständen (Pretschner, 2001). Wenn der Zielzustand als ein Prädikat mit mehreren Literalen definiert ist, kann die Differenz aus der Anzahl der bereits erfüllten Literale und der notwendigen Anzahl der zu erfüllenden Literalen bestimmt werden. Konjunktionen und Disjunktionen können mit Summen oder Produkten kombiniert werden (Edelkamp u. a., 2001; Tracey, 2000). Beispiele für Fitneßfunktionen – Inverse einer Distanzfunktion – werden in Abschnitt 5.4 gegeben. Tab. 5.1 zeigt beispielhaft mögliche Distanzfunktionen. ε ist dabei eine kleine Konstante und c_P der Wert der Distanzfunktion des Prädikats P. Auf Probleme der Art, daß nicht für alle Distanzfunktionen logisch äquivalente Ausdrücke dieselbe Distanz ergeben, soll hier nicht eingegangen werden.

Statische gerichtete Suche Die Bestimmung der Distanz erfolgt normalerweise zur Laufzeit. Wenn allein Kontrollzustände und das Distanzmaß der kürzesten Pfade gewählt werden, kann die Distanzfunktion allerdings zur Compilezeit berechnet werden und die Transitionsprädikate entsprechend ordnen. Das funktioniert natürlich i.A. nur für eine einzelne Komponente.

Die Idee besteht darin, zunächst alle Vorgänger bzgl. der Transitionspfeilrelation des Zielkontrollzustands zu betrachten. Von allen ausgehenden Transitionen werden als erstes diejenigen ausprobiert, die zum Zielzustand führen. Danach werden die Vorgänger der Vorgänger des Zielzustands betrachtet. Für deren Transitionsordnung gilt, daß zuerst diejenigen Transitionen ausprobiert werden sollen, die direkt zu einem Vorgänger des Zielzustands führen. Die wei-

$$
\begin{array}{l}
\textit{Eingabe}: \ \textit{Zielkontrollzustand } s, \ \textit{Transitionsrelation } \xrightarrow{t} \\
\textit{Ausgabe}: \ \textit{Transitionsordnungen } T_\sigma \\
M = \{s\}; \ E = \emptyset; \ \textit{forall } \sigma \in S: \ T_\sigma = <>; \\
\textit{repeat} \\
\quad M' = \bigcup_{m \in M} \{(\sigma, t) : \sigma \xrightarrow{t} m\}; \\
\quad \textit{forall } (\sigma, t) \in M': \ T_\sigma = T_\sigma \circ < t >; \\
\quad E = E \cup M; \\
\quad M = \{\sigma : (\sigma, t) \in M'\} - E; \\
\textit{until } M = \emptyset;
\end{array}
$$

Abbildung 5.2.: Algorithmus für statische Transitionsordnungen

tere Iteration des Verfahrens liefert Transitionsordnungen für alle diejenigen Zustände, von denen aus der Zielzustand erreicht werden kann. Der Algorithmus in Abb. 5.2 formalisiert das.

Dabei ist s der Zielkontrollzustand. M bezeichnet die Menge der noch zu bearbeitenden Kontrollzustände; S die Menge aller Kontrollzustände; M' die Menge der Vorgängerzustände von M incl. Transition, und E bezeichnet die Menge der bereits abgearbeiteten Kontrollzustände. T_σ ist die Transitionsordnung für Zustand σ. Die leere Sequenz wird durch $<>$ dargestellt, Konkatenation erfolgt mit $\circ$. $\sigma \xrightarrow{t} m$ gilt, falls es einen Transitionspfeil t von Zustand σ nach m gibt. Bei maximalem Fanout f eines Kontrollzustands ist die Komplexität bei n Knoten in $\mathcal{O}(f \cdot n)$; im schlimmsten Fall einer einzigen Clique also in $\mathcal{O}(n^2)$. Der Beweis (Abschnitt B.1) von Vollständigkeit und Terminierung folgt sofort aus der Konstruktion; die Korrektheit wird durch vollständige Induktion über die Anzahl der Schleifendurchläufe nachgewiesen.

Für die symbolische Ausführung kann man zwar jede Komponente als erste ausführen lassen, insbesondere also auch die, für die eine best-first-Transitionsordnung statisch berechnet wird. Wenn man die Komponenten in einer Art anordnet, die dem Datenfluß möglichst genau entspricht (was offenbar im Normalfall ein schwieriges Unterfangen ist), wird für die Testfallgenerierung häufig Nichtdeterminismus verhindert. Es kann passieren, daß bei der Transitionsauswahl für eine Komponente ein nicht erreichbarer Zustand gewählt wird, und für alle anderen Komponenten müssen dann wegen der Tiefensuche alle (in der Gesamtheit von vornherein nicht ausführbaren) Transitionen ausprobiert werden.

Darüberhinaus weist dieses Vorgehen den Nachteil auf, daß allein die Transitionspfeile als Grundlage der gerichteten Suche dienen. Funktionsaufrufe auf Transitionen werden also nicht berücksichtigt. Wenn Modelle – wie im Fall der Chipkarte – keine ausgezeichnete Zustandsraumprojektion besitzen und demzufolge jede Komponente nur aus sehr wenigen Kontrollzuständen besteht, dann sind die Erfolgsaussichten der präsentierten Strategie offenbar gering. Im Rahmen der Experimente (Abschnitt 5.4) wird sich darüberhinaus herausstellen, daß der Gewinn durch eine statische anstelle einer dynamischen Ordnung ohnehin gering ist, weshalb der dynamischen Variante im Normalfall der Vorzug zu

gewähren ist. Alle genannten Probleme einer statischen Festlegung der Transitionsreihenfolge werden damit behoben:

- Da zur Berechnung der Distanz das Produkt der Transitionen aller Komponenten gewählt wird, wird die Wahl einer in einen unerreichbaren Zustand mündenden Transition von vornherein ausgeschlossen.
- Es wird nicht nur der Transitionspfeil, sondern auch alle möglichen Funktionsaufrufe auf den Transitionspfeilen für die Entscheidung berücksichtigt.
- Wenn es gleichgültig ist, ob als Zielzustand eine Komponente des Datenzustands oder ein Kontrollzustand gewählt wird, dann spielt es offenbar auch keine Rolle, wieviele Kontrollzustände vorhanden sind.

In der Praxis zeigt sich, daß das Verfahren gut für geordnete Typen funktioniert (s. Abschnitt 5.4), sich aber bzgl. der Effizienz für Fitneßfunktionen, die ausschließlich auf ungeordneten Typen basieren, Schwierigkeiten ergeben. Ein weiteres Problem ergibt sich i.a. für parallel komponierte Systeme. Fitneßfunktionen werden häufig lokal, d.h. für eine Komponente definiert. Die Möglichkeiten der Transitionsauswahl der anderen Komponenten wird dadurch im wesentlichen nicht beschränkt, was bzgl. dieser Komponenten nach wie vor zu hohem Nichtdeterminismus führt.

Breitensuche und Wettbewerbsparallelität Die von CLP-Systemen zur Verfügung gestellte Suchstrategie ist eine Tiefensuche. Im vergangenen Abschnitt wurde die Wichtigkeit der Ordnung der Ausführung nicht nur von Transitionen, sondern auch von Komponenten angesprochen.

Die Probleme mit der Tiefensuche sind klassischer Natur. Stellt sich die Entscheidung, eine bestimmte Transition in einem frühen Schritt zu wählen, bzgl. des zu findenden Zustands als ungünstig heraus, kann es lange dauern, bis diese Entscheidung durch Auswahl einer alternativen Transition revidiert wird: Die Suche „konzentriert“ sich zunächst auf die tiefen Ebenen des Suchbaums.

Mit Breitensuche tritt dieses Problem nicht auf. Allerdings erfordert sie das explizite Anlegen aller Zustände der Knoten einer Ebene des Suchraums, was insbesondere bei großem Fanout der Knoten zu prohibitivem Speicheraufwand führt.

Eine Mischung aus Breiten- und Tiefensuche kann durch Wettbewerbsparallelität erreicht werden. Die Idee ist, mehrere Instanzen der z.B. bzgl. der Transitionsauswahl randomisierten Tiefensuche parallel ablaufen zu lassen. Motivation ist die Zufälligkeit der Suche und die daraus resultierende hohe Varianz der Ergebnisse (notwendige Zeit bzw. Speicher, vgl. Abschnitt 5.4) bei unabhängigen Experimenten. Wenn die „falsche“ Transition weit oben im Suchraum ausgewählt wird, so wird diese Entscheidung bei Tiefensuche erst spät wieder „rückgängig“ gemacht. Motivation für die parallele Suche ist, die Wahrscheinlichkeit zu erhöhen, die „richtige“ Transition oben im Suchbaum auszuwählen.

Dabei gibt es zwei Möglichkeiten:

- Die Suche findet in unabhängigen *Prozessen* statt, die nicht über gemeinsamen Speicher verfügen. Bei Einprozessormaschinen und n Prozessen

ergibt sich unter Vernachlässigung von Kontextwechseln als Aufwand für Speicher und Zeit in etwa das n-fache des Aufwands des „besten“ Prozesses. Bei Mehrprozessormaschinen mit m Prozessoren und $n \leq m$ ist der Zeitaufwand der des besten Prozesses. Für $n > m$ ist der Zeitaufwand ungefähr das n/m-fache des besten Prozesses. Der Speicheraufwand ergibt sich als Aufwand des besten Prozesses multipliziert mit n.

- Die Suche findet in *Threads* statt, die über einen gemeinsamen Speicher verfügen und auf einem Prozessor laufen. Weil EF-Eigenschaften überprüft werden, gelten die bereits besuchten Zustände für alle Instanzen als bereits besucht, was zu – in der Praxis durch Variation der maximalen Suchtiefe beherrschbaren – Problemen führt, die ähnlich zu dem in Abb. 5.1 (b) aufgeführten sind. Wenn ein Thread auch nach Backtracking keinen Folgezustand mehr findet, der nicht von einem anderen Thread vorher besucht wurde, dann terminiert er. Der Vorteil dieses Verfahrens im Vergleich zur Breitensuche liegt darin, daß die Zustände einer Ebene des Suchraums nicht alle explizit angelegt werden müssen. Stattdessen selektiert im Rahmen der Tiefensuche jeder Thread nur einen Folgeknoten zufällig. Das CLP-System kümmert sich um das Anlegen von Choicepoints (Verzeigerung von noch zu erfolgenden Prädikataufrufen in der Warren Abstract Machine). Diese Strategie läßt sich einfach durch sequentielles Interleaving des Verhaltens der unterschiedlichen Threads implementieren, so daß keine „echte“ Parallelität stattfindet.

5.2. Zustandsspeicherung und EF-Model Checking

Reaktive Systeme sind u.a. durch potentiell unendliche Abläufe gekennzeichnet. Testfallgeneratoren wie der in dieser Arbeit präsentierte, die auf einer Exploration des Zustandsraums basieren, müssen diesem Umstand offenbar Rechnung tragen.

Ein Ansatz dazu besteht darin, einmal besuchte Zustände nicht wieder zu besuchen. Für den Test eines *Modells* ist das unproblematisch, weshalb eine ähnliche Technik auch in explicit-state Model Checkern wie Spin (Holzmann, 1997) zum Tragen kommt (in der Tat ist eines der intensivst bearbeiteten Themenkomplexe im Umfeld von Spin die effiziente Zustandsspeicherung). Da Modelle Abstraktionen von Maschinen sind und jeder Modellzustand normalerweise mehreren Maschinenzuständen entspricht, ergeben sich für den Test der Maschine allerdings Gefahren. In der Realität treten häufig Fehlerfälle der Art auf, daß eine bestimmte Eingabensequenz bei n-facher Ausführung korrekte Ausgaben liefert. Bei der $n+1$-ten Ausführung der Maschine tritt dann aus üblicherweise nicht direkt nachvollziehbaren Gründen ein Fehler auf. Für die Modellvalidierung ist das irrelevant. Testsuiten zum Zweck der Maschinenverifikation sind aber bisweilen gerade dann interessant, wenn sie Zustandsdopplungen enthalten.

Beispiel 25 (Zustandsdopplung). *Wenn bei der Testfallgenerierung für die Chipkarte doppelt besuchte Zustände verboten werden, dann gibt es – bei festen*

Sicherheitsumgebungen – in den Testsuiten niemals aufeinanderfolgende `Verify`*-Kommandos, die im ersten Durchgang eine PIN verifizieren und sie im zweiten annullieren. Das zweite Kommando würde das Modell in einen vorher besuchten Zustand versetzen.*

Nichtsdestotrotz ist diese Form der Einschränkung häufig sinnvoll. Klassische Testverfahren müssen ebenfalls die Anzahl der Iterationen von Schleifen beschränken. Wenn anstelle der erneuten Berücksichtigung von bereits besuchten Zuständen Paare oder kurze Sequenzen von Zuständen bei der Testfallgenerierung ausgeschlossen werden, kann dieses Problem abgemildert werden – der Tester muß sich dieser Tatsache wohl bewußt sein. Für strukturelle Überdeckungskriterien ist das Verbieten wiederholter Teilsequenzen offenbar (abhängig vom Überdeckungskriterium) unproblematisch. Eine Kombination von Testfällen ohne wiederholte Zustände mit solchen, die beliebige Schleifen zulassen, erscheint sinnvoll.

In der semantischen Charakterisierung des einem AutoFocus-Modell entsprechenden CLP-Programms (Abschnitt 4.4) tritt dieser Unterschied nicht auf. Bei der Berechnung der Kleene-Sequenz $S_0 \equiv \mathit{init}$ und $S_{i+1} \equiv S_i \vee \mathit{post}[T]S_i$ kann offenbar die zweite Definition durch

$$S_{i+1} \equiv S_i \vee \mathit{post}[T](S_i \wedge \neg S_{i-1}) \tag{5.1}$$

ersetzt werden (s. auch den Beweis von Gleichung 5.10). Da die Berechnung von $\mathit{post}[T]$ genau der Ausführung eines Zyklus des Systems entspricht, ergibt sich in operationeller Hinsicht hingegen durchaus ein Unterschied. Erneut sei darauf hingewiesen, daß die S_i Repräsentationen von Zustandsmengen sind, die während der symbolischen Ausführung erzeugt werden.

Speichern von Zustandsmengen Im Kontext der symbolischen Ausführung kann die Überprüfung auf bereits besuchte Zustände nicht einfach durch den Test auf syntaktische Gleichheit erfolgen, wie das in explicit-state Model Checkern wie beispielsweise Spin erfolgt. Unter – der nicht immer realistischen – Annahme kanonischer Repräsentation von Constraints ist ein Vergleich auf syntaktische Gleichheit modulo Umbenennung von aktuellem Zustand und bereits besuchten Zuständen incl. Constraints ein erster Ansatz. Das ist aber nicht immer ausreichend.

Angenommen, eine Komponente d des Datenzustands ist durch einen Ungleichheitsconstraint $d \notin \{c_1, c_2\}$ für Datentypkonstruktoren c_1, c_2 und der Extension $\{c_1, c_2, c_3, c_4\}$ des Typen von d definiert. Wenn nun ein neuer Zustand mit $d = c_3$ besucht wird, dann ist dieser Zustand syntaktisch modulo Variablenumbenennung nicht mit dem bereits besuchten identifizierbar. Er ist vielmehr eine Spezialisierung des vorher besuchten Zustands und soll deshalb nicht erneut besucht werden (hier werden die anderen Komponenten des Systemzustands ignoriert). Ein Zustand soll hingegen dann besucht werden, wenn er in bezug auf alle vorher besuchten Zustände unvergleichbar ist oder eine Verallgemeinerung darstellt.

Die symbolische Ausführung erlaubt darüberhinaus, mit unterspezifizierten Werten zu rechnen. Wenn die Modellierungssprache Produkttypen zuläßt, dann

können Repräsentationen von Zustandsmengen als Terme aufgefaßt werden, die Variablen beinhalten, etwa $c(X)$ oder $c(X)$ mit $X \in \mathbb{N}, X \geq 2$ für einen Konstruktor c und eine Variable X. Wird danach ein Zustand $c(X)$ mit $X \geq 3$ besucht, so stellt er eine Spezialisierung des ersten Falls dar und kann ignoriert werden.

Intuitiv „verliert“ man dadurch keine erreichbaren Zustände, die zu testen wären, weil allgemeinere Zustände immer in beliebige speziellere instantiiert werden können. Ein auf den erreichbaren Zuständen basierendes Vollständigkeitskonzept wird am Ende dieses Abschnitts definiert und in Abschnitt B.4 nachgewiesen. Eine Erweiterung der Ideen zur Zustandsspeicherung besteht darin, nicht nur solche Zustandsmengen zu verbieten, die Spezialisierung einer bereits besuchten sind, sondern zusätzlich vom aktuell besuchten Zustand alle bereits besuchten zu subtrahieren. Es wird sich herausstellen, daß für beide Fälle – Zustandsinklusion und -differenz – elegante syntaktische Charakterisierungen existieren, die die explizite Komplementbildung von Termen nicht erfordern, sondern nur die Negation von Constraints auf solchen Termen.

Syntaktische Charakterisierung der Inklusion Eine Repräsentation einer Zustandsmenge σ ist ein Paar $< \tilde{\sigma}, \mathcal{C}_\sigma >$. Die erste Komponente $\tilde{\sigma}$ repräsentiert die syntaktische Komponente von σ. Die zweite Komponente ist eine Menge von Constraints über den Variablen von $\tilde{\sigma}$, eine Menge von Gleichungen, Ungleichungen und anderen Relationen. Die Repräsentation einer Zustandsmenge ist also ein Term mit Constraints. Im folgenden wird die Typinformation über Variablen, die ebenfalls als Constraint gespeichert wird, ignoriert. Insbesondere ist sie nicht Teil von $\mathcal{C}_\sigma$. Ein Beispiel mit einer Variablen X ist $< \mathit{verify}(\mathit{pinGRef}, X), X \neq \mathit{pinG} >$, das eine Komponente des Zustandsvektors der WIM darstellt, die eingehenden Kommandos nämlich. Es bezeichnet alle diejenigen Kommandos zur Verifikation der PIN-G, die mit falscher PIN aufgerufen werden.

Für den Moment bezeichne $[\![\sigma]\!]$ alle typkorrekten Grundinstanzen von $\tilde{\sigma}$, die $\mathcal{C}_\sigma$ erfüllen (d.h. typkorrekte, $\mathcal{C}_\sigma$ erfüllende Instantiierungen *aller* Variablen in $\mathcal{C}_\sigma$), also

$$s \in [\![< \tilde{\sigma}, \mathcal{C}_\sigma >]\!] \Leftrightarrow \mathcal{V}(s) = \emptyset \wedge \mathit{unif}(s, \tilde{\sigma}) \wedge \mathit{mgu}_{(s,\tilde{\sigma})}(\mathcal{C}_\sigma), \tag{5.2}$$

wobei erneut *unif* die Unifizierbarkeit der zwei Argumente entscheidet und $\mathit{mgu}_{(s,t)}(x)$ den allgemeinsten Unifikator zweier Terme s und t auf ein x anwendet. $[\![\mathcal{C}]\!]$ bezeichne analog die Menge aller typkorrekten Grundinstanzen der Variablen des Constraints $\mathcal{C}$. Im obigen Beispiel ist

$$[\![< \mathit{verify}(\mathit{pinGRef}, X), X \neq \mathit{pinG} >]\!]$$

die Menge aller der Terme, in denen X durch die im Modell auftretenden PIN-Codes ersetzt wird, d.h. etwa `pinA, pinB, pukG, ...`. Es bezeichne darüberhinaus Φ die Menge der Repräsentationen aller bereits besuchten Zustandsmengen zu einem gegebenen Zeitpunkt. Die genaue Form der Repräsentation wird auf den Seiten 164 ff. angegeben.

Eine Zustandsspeicherung, die die Techniken des explicit-state Model Checking auf Mengen von Zuständen erweitert, beruht darauf, daß die einer Repräsentation ϑ entsprechende Zustandsmenge $[\![\vartheta]\!]$ dann nicht besucht wird, falls

$$\big(\exists\varphi \in \Phi \bullet [\![\vartheta]\!] \subseteq [\![\varphi]\!]\big) \ \textit{bzw.}\ \big(\exists\varphi \in \Phi \bullet [\![\vartheta]\!] \cap \overline{[\![\varphi]\!]} = \emptyset\big) \tag{5.3}$$

gilt, ϑ also Spezialisierung einer bereits besuchten Zustandsrepräsentation ist. Dabei bezeichnet $\overline{[\![\varphi]\!]}$ das Komplement von $[\![\varphi]\!]$, dessen explizite Berechnung nicht notwendig ist (Gl. 5.4).

Offenbar besteht der Ansatz darin, Zustandsmengen abzuspeichern und in jedem Schritt zu überprüfen, ob der aktuelle Zustand Instanz eines bereits besuchten ist. Wenn das der Fall ist, wird der aktuelle Zustand nicht weiter betrachtet.

Um das Problem der Negation von Zustandsmengen anzugehen, wird etwas Notation benötigt. Erneut bezeichne *Subst* die Menge aller Substitutionen. Ein Term $\tilde{\sigma}$ ist syntaktisch allgemeiner als ein Term $\tilde{\vartheta}$, $\tilde{\sigma} \leq \tilde{\vartheta}$, gdw. gilt

$$\exists\eta \in \mathit{Subst} \bullet \eta(\tilde{\sigma}) = \tilde{\vartheta}.$$

$\tilde{\vartheta}$ ist syntaktisch spezifischer als $\tilde{\sigma}$ gdw. $\tilde{\sigma}$ syntaktisch allgemeiner als $\tilde{\vartheta}$ ist. $\tilde{\sigma}$ und $\tilde{\nu}$ heißen Varianten, falls $\tilde{\sigma} \leq \tilde{\nu}$ und $\tilde{\nu} \leq \tilde{\sigma}$ gilt, was durch Variablenumbenennung erreicht wird. Falls $\tilde{\sigma}$ und $\tilde{\vartheta}$ nicht bzgl. $\leq$ vergleichbar sind, heißen sie syntaktisch unvergleichbar.

Eine syntaktische Charakterisierung – ein CLP-Programm – der *negierten* Inklusionsbeziehung ergibt sich dann zu

$$[\![\nu]\!] \cap \overline{[\![\sigma]\!]} \neq \emptyset \Longleftrightarrow \mathcal{C}_\nu \wedge \big(\tilde{\sigma} \leq \tilde{\nu} \Rightarrow mgu_{(\tilde{\nu},\tilde{\sigma})}(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu)\big), \tag{5.4}$$

falls die Kardinalität von $[\![\cdot]\!]$ größer als 1 angenommen wird. Die konstruktive Negation von Constraints $\neg\mathcal{C}$ bzw. $\overline{\mathcal{C}}$ wird auf S. 165 diskutiert. Intuitiv werden nur auf Gleichheit bzw. Ungleichheit zurückführbare entscheidbare Constraints betrachtet. Die Negation des leeren Constraintstores ist unerfüllbar, d.h.

$$\mathcal{C} = \top \Rightarrow \overline{\mathcal{C}} = \bot.$$

Beispiele Beispiele für Terme – Repräsentationen von Zustandsmengen –, in denen Variablen durch Großbuchstaben gekennzeichnet sind, sind in der Rückrichtung der Äquivalenz 5.4

- $\sigma =< c(a), \top >$ und $\nu =< c(X), \top >$: Es ist $\tilde{\sigma} \geq \tilde{\nu}$, was $[\![\nu]\!] \not\subseteq [\![\sigma]\!]$ impliziert.
- $\sigma =< c(X), \top >$ und $\nu =< c(a), \top >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \bot$, was $[\![\nu]\!] \subseteq [\![\sigma]\!]$ impliziert.
- $\sigma =< c(X), \top >$ und $\nu =< c(X), X \geq 3 >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \bot$, was $[\![\nu]\!] \subseteq [\![\sigma]\!]$ impliziert.
- $\sigma =< c(X, a), \top >$ und $\nu =< c(b, Y), \top >$: $\tilde{\sigma}$ und $\tilde{\nu}$ sind bzgl. $\leq$ unvergleichbar, was $[\![\nu]\!] \not\subseteq [\![\sigma]\!]$ impliziert, wenn die Kardinalität der Typen, zu deren Extension a bzw. b gehören, größer als eins ist. Diese Forderung wird im folgenden nicht mehr explizit vermerkt.

- $\sigma =< c(X), X \neq a >$ und $\nu =< c(b), \top >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \{X = a\}$. Die Unifikation von $c(X)$ mit $c(b)$ führt modulo $\overline{\mathcal{C}}_\sigma$ zu einem Widerspruch. Es ist $[\![\nu]\!] \subseteq [\![\sigma]\!]$.

- $\sigma =< c(X), X \neq a >$ und $\nu =< c(a), \top >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \{X = a\}$. Die Unifikation von $c(X)$ mit $c(a)$ führt zu keinem Widerspruch. Es ist $[\![\nu]\!] \not\subseteq [\![\sigma]\!]$.

- $\sigma =< c(X), X \notin \{a, b\} >$ und $\nu =< c(X'), X' \in \{a, b\} >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \{X = a \vee X = b\}$. Die Unifikation ergibt die Substitution $\{X \mapsto X'\}$ und damit die erfüllbaren Constraints $\{X' = a \vee X' = b\}$. Also ist $[\![\nu]\!] \not\subseteq [\![\sigma]\!]$.

- $\sigma =< c(X), X \in \{1, 2\} >$ und $\nu =< c(X'), X' \in \{1, 2\} >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \{X \neq 1 \wedge X \neq 2\}$. Unifikation liefert die Substitution $\{X \mapsto X'\}$, und die Konjunktion der Constraints $X' \neq 1 \wedge X' \neq 2 \wedge X' \in \{1, 2\}$ ist unerfüllbar. Es ist $[\![\nu]\!] \subseteq [\![\sigma]\!]$.

- $\sigma =< c(X), X \in \{1, 2\} >$ und $\nu =< c(X'), X' \in \{2, 3\} >$: Es ist $\tilde{\sigma} \leq \tilde{\nu}$ und $\overline{\mathcal{C}}_\sigma = \{X \neq 1 \wedge X \neq 2\}$. Unifikation liefert die Substitution $\{X \mapsto X'\}$, und die Konjunktion der Constraints $X' \neq 1 \wedge X' \neq 2 \wedge X' \in \{2, 3\}$ ist für die Substitution $\{X' \mapsto 3\}$ erfüllbar. Es ist $[\![\nu]\!] \not\subseteq [\![\sigma]\!]$.

Der Beweis der Aussage 5.4 erfolgt in Abschnitt B.2. Die Idee ist, zunächst das Komplement für Terme ohne Constraints zu definieren und die Aussage über eine Fallunterscheidung des Verhältnisses von $\tilde{\sigma}$ und $\tilde{\nu}$ bzgl. $\leq$ zu beweisen. Der Beweis für Terme mit Constraints charakterisiert zunächst die $\mathcal{C}_\sigma$ erfüllenden Instanzen von $\tilde{\sigma}$ durch eine Schnittmengenbildung, und unter Rückgriff auf den Fall ohne Constraints erfolgt der Beweis dann wiederum über eine Fallunterscheidung der Zustände bzgl. $\leq$.

Verallgemeinerung Bis jetzt wurde eine symbolische Repräsentation der Überprüfung auf eine mögliche Inklusionsbeziehung von Zustandsmengen diskutiert. Motivation dafür war der Wunsch, Spezialisierungen einmal besuchter Zustände nicht wieder zu besuchen (Gleichung 5.3). Es stellt sich dann sofort die Frage, ob dieses Vorgehen nicht dahingehend verallgemeinert werden kann, daß nur Teile der aktuell besuchten Zustandsmenge weiterhin betrachtet werden, nämlich genau die Teile, die noch nicht vorher besucht wurden. Anstatt aktuelle Zustandsmengen dann zu ignorieren, wenn sie Spezialisierungen bereits besuchter Zustandsmengen darstellen, soll der aktuelle Zustand um die bereits besuchten Teile vermindert werden. Das wird auch in Gleichung 5.1 suggeriert. Offenbar können noch nicht besuchte Teile der aktuellen Zustandsmenge $[\![\vartheta]\!]$ dadurch berechnet werden, daß von der aktuellen Zustandsmenge alle bereits besuchten subtrahiert werden:

$$[\![\vartheta]\!] - \bigcup_{\varphi \in \Phi} [\![\varphi]\!] \; bzw. \; [\![\vartheta]\!] \cap \bigcap_{\varphi \in \Phi} \overline{[\![\varphi]\!]} \tag{5.5}$$

PSfrag replacements

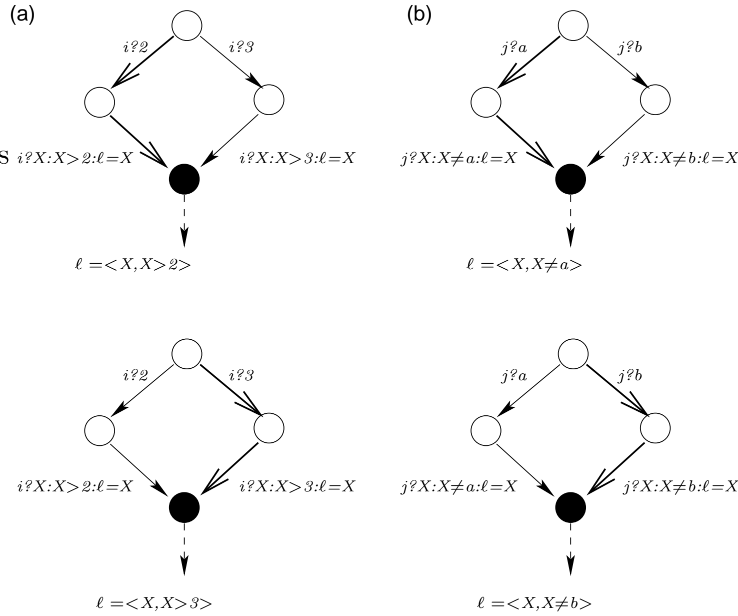


Abbildung 5.3.: Zustandsinklusion und nichtleerer Durchschnitt

Abb. 5.3 demonstriert den Unterschied. Angegeben sind zwei erweiterte Zustandsmaschinen. Für Teil (a) besitze die Komponente einen Eingabekanal i vom Typ *Int* und eine lokale Variable ℓ vom selben Typ. Wenn in (a, oben) zunächst der linke Pfad gewählt wird, dann geht die gestrichelte Transition von der Zustandsmenge $X > 2$ (Kontrollzustand wird ignoriert) aus. Wenn danach der rechte Pfad gewählt wird, wird bei Erreichen des schwarzen Kontrollzustands festgestellt, daß $[\![< X, X > 3 >]\!]$ eine Spezialisierung von $[\![< X, X > 2 >]\!]$ ist, und die Suche wird abgebrochen. Wenn umgekehrt zuerst der rechte Pfad gewählt wird (a, unten), ist die Zustandsmenge, von der die gestrichelte Transition ausgeht, $[\![< X, X > 3 >]\!]$. Wird danach der linke Pfad beschritten, so wird festgestellt, daß

$$[\![< X, X > 2 >]\!] \not\subseteq [\![< X, X > 3 >]\!]$$

ist, und die gestrichelte Transition geht von der Zustandsmenge $[\![< X, X > 2 >]\!]$ aus. Das bedeutet, daß im zweiten Fall die Zustandsmenge

$$[\![< X, X > 2 >]\!] \cap [\![< X, X > 3 >]\!] = [\![< X, X > 3 >]\!]$$

doppelt beschritten wird. Wenn anstelle der Überprüfung auf Inklusion alle bereits besuchten Zustände vom aktuellen Zustand subtrahiert werden, so ergibt sich im Fall des linken Pfades als erstem Pfad, daß die gestrichelte Transition für den rechten Pfad nicht erneut berücksichtigt werden muß wegen

$$[\![< X, X > 3 >]\!] - [\![< X, X > 2 >]\!] = \emptyset.$$

Wird der rechte Pfad zuerst gewählt, dann ist die zu berücksichtigende Zustandsmenge

$$[\![< X, X > 2 >]\!] - [\![< X, X > 3 >]\!] = \{3\},$$

und dieser Zustand wurde für den rechten Pfad noch nicht berücksichtigt.

Abb. 5.3 (b) stellt eine analoge Situation dar, in der die durch die Transitionen induzierten Zustandsmengen allerdings nicht bzgl. $\subseteq$ geordnet sind. Die entsprechende Komponente besitze einen Eingabekanal j vom Typ $\{a, b, c\}$ und eine lokale Variable ℓ vom selben Typ. Wird Zustandsspeicherung auf der Basis von Mengeninklusion eingesetzt, sind der linke und rechte Pfad bzgl. dieser Tatsache völlig unterschiedlich. Wird hingegen Zustandssubtraktion angewendet, so ergibt sich für die Reihenfolge links-rechts (b, oben), daß beim zweiten Besuch die Zustandsmenge $\{a, c\} - \{b, c\} = \{a\}$ (und nicht $\{a, c\}$) berücksichtigt werden muß und für die umgekehrte Reihenfolge (b, unten) die Zustandsmenge $\{b, c\} - \{a, c\} = \{b\}$ (und nicht $\{b, c\}$).

Für diesen Ansatz kann nicht dieselbe Repräsentation wie im vorherigen Abschnitt verwendet werden, um den aktuell besuchten Zustände um alle vorher besuchten zu vermindern, wie das Gleichung 5.5 vermuten ließe. Das Problem besteht darin, daß eine Charakterisierung der Art

$$\overline{[\![\tilde{\sigma}]\!]} = \{\tilde{\nu} : \tilde{\sigma} \not\leq \tilde{\nu}\},$$

wie sie im Beweis verwendet wird, nicht herangezogen werden kann. Das zeigt das Beispiel $\tilde{\sigma} = c(a)$: Für $x \in \mathcal{V}$, falls x also eine Variable ist, erfüllt $\tilde{\nu} = c(x)$ zwar die genannte Bedingung, aber die Information, daß $c(a)$ bereits besucht wurde, fehlt.

Das Komplement der Repräsentation einer Zustandsmenge σ besteht offenbar aus all denjenigen Zustandsrepräsentationen, die

- syntaktisch nicht mit $\tilde{\sigma}$ unifizierbar oder
- syntaktische Varianten von $\tilde{\sigma}$ sind, die die Constraints $\mathcal{C}_\sigma$ nicht erfüllen.

Unter der Voraussetzung, daß im syntaktischen Teil eines Zustands jede Variable nur einmal auftaucht (das kann immer durch explizite Gleichheitsconstraints erzielt werden), ist dann

$$\overline{[\![\sigma]\!]} = \begin{cases} [\![< R, R \neq^{pm} \tilde{\sigma} >]\!] \cup [\![< \tilde{\sigma}, \overline{\mathcal{C}}_\sigma >]\!], & \textit{falls } \mathcal{C}_\sigma \neq \top \\ [\![< R, R \neq^{pm} \tilde{\sigma} >]\!], & \textit{falls } \mathcal{C}_\sigma = \top. \end{cases} \quad (5.6)$$

Dabei wird auf die Ungleichheit $\neq^{pm}$ zurückgegriffen, die in Gleichung 4.5 definiert wurde. Intuitiv ist der Grund dafür der folgende: Die Komplementbildung von Zuständen ist hier insofern von Interesse, als ihr Schnitt mit anderen Zuständen zum Zweck der Differenzbildung vorgenommen wird. Wenn zunächst ein Zustand $< c(a), \top >$ abgespeichert wird und Gleichung 5.6 mit *unif* anstelle von $\neq^{pm}$ implementiert wird – was *nach* Einbettung in $[\![\cdot]\!]$ die Semantik nicht verändert –, dann wird das Komplement von $c(a)$ als $\neg \textit{unif}(R, c(a))$ abgespeichert. Der Schnitt mit einem neuen Zustand $c(X)$ führt dann bei Unifikation mit R zu $\neg \textit{unif}(c(X), c(a))$, was zu *fail* reduziert. Die Definition mit $\neq^{pm}$ hingegen führt zu $< c(Y), Y \neq^{pm} a >$. Der Unterschied ist, daß das implizit allquantifizierte $\neg\textit{unif}$ direkt evaluiert wird, und $\neq^{pm}$ stattdessen Bedingungen liefert, unter denen die beiden Argumente nicht unifizierbar sind.

Der Beweis von Gl. 5.6 ist in Abschnitt B.3 angegeben. Im wesentlichen folgt die Aussage sofort, wenn der Zusammenhang zwischen $\neq^{pm}$ und *unif* präzisiert wird.

Zum Beispiel ist $\overline{[\![< c(a,a), \top >]\!]} = [\![< R, R \neq^{pm} c(a,a)]\!]$ und

$$\begin{aligned}&\overline{[\![< c(X,X), \top >]\!]} = \overline{[\![< c(X,Y), X = Y >]\!]} = \\ &\quad [\![< R, R \neq^{pm} c(X',Y') \vee R = c(X',Y') \wedge Y' \neq^{sym} X' >\end{aligned}$$

für eine frische Variable R.

Implementierung der Differenz Der Durchschnitt des Komplement eines alten Zustands σ mit einem neuen Zustand ν erfolgt dann über den Aufruf des CLP–Programms κ_σ mit[2]

$$\begin{aligned}\kappa_\sigma(\nu, K) \Leftarrow \quad & \mathcal{C}_\nu \wedge K = \nu \\ \wedge \quad & \mathcal{C}_\sigma = \top \Rightarrow K \neq^{pm} \tilde{\sigma}' \\ \wedge \quad & \mathcal{C}_\sigma \neq \top \Rightarrow (K \neq^{pm} \tilde{\sigma}' \vee mgu_{(K,\tilde{\sigma}')}(\overline{\mathcal{C}}_{\sigma'}))\end{aligned} \tag{5.7}$$

für eine echte Variante σ' von σ. Nach Aufruf enthält K eine Repräsentation der Differenz $[\![\nu]\!] - [\![\sigma]\!]$. Für $\sigma =< c(a,a), \top >$ und $\nu =< c(X,X), \top >$ gilt beispielsweise $\kappa_\sigma(\nu, c(X,X)) \Leftarrow X \neq^{pm} a$; vertauscht man die Rollen von ν und σ, ist κ_σ ungültig.

Im Beispiel $\sigma =< c(b, d(X)), X \neq a >$ ergibt sich für einen neuen Zustand $\nu =< c(Y,Z), Y \neq b >$ für $Y, Z \in \mathcal{V}$ die Differenz $[\![\nu]\!] - [\![\sigma]\!]$

$$< c(Y,Z), Y \neq^{pm} b \vee (Z \neq^{pm} d(X) \wedge Y \neq^{pm} b).$$

Je nach Mächtigkeit des Constraintlösers wird das zum schwächsten Constraint $Y \neq^{pm} b$ reduziert oder nicht. Wenn das nicht der Fall ist, dann entstehen offenbar neue überflüssige Constraints für ν.

Model Checking Die zwei in diesem Abschnitt genannten Verfahren ergeben sofort ein Verfahren zum Model-Checking von CTL-Formeln $EF\varphi$ für Zustandsformeln φ. Ob nun Spezialisierungen einmal besuchter Zustände ignoriert werden oder die aktuell besuchte Zustandsmenge um alle bereits besuchten vermindert wird, ist für die Korrektheit natürlich irrelevant. Bezüglich der Performance ist bei geeigneter Repräsentation von Zustandskomplementen der Trade-Off zwischen dem Gewinn, der aus der Subtraktion entsteht und dem Verlust, der sich aus der Verwaltung von entsprechenden Constraintmengen ergibt, abzuwägen. Eine Evaluation ist Teil zukünftiger Arbeiten, ebenso wie die Erweiterung auf Model-Checking für CTL- oder μ-Kalkül-Fragmente, indem größte Fixpunkte durch kleinste berechnet werden (Cui u. a., 1998; Delzanno und Podelski, 1999; Nilsson und Lübcke, 2000).

[2] Die Erfüllbarkeit der Constraints $\mathcal{C}_\nu$ ist gesichert, weil sie zusammen mit ν an κ übergeben werden. Der Aufruf erfolgt hier nur aus Gründen der expliziten Darstellung.

Zusammenfassung Zusammenfassend ist festzuhalten, daß der Unterschied zwischen den beiden diskutierten Verfahren darin besteht, daß

- im durch Gleichung 5.3 implizierten und durch Gleichung 5.4 charakterisierten Verfahren eine neu erreichte Zustandsmenge dann ignoriert wird, wenn sie Spezialisierung einer vorher bereits besuchten ist und
- im durch Gleichung 5.5 implizierten und durch Gleichung 5.7 charakterisierten Verfahren vom aktuell besuchten Zustand alle vorher besuchten subtrahiert werden. Das zweite Verfahren stellt sicher, daß einmal besuchte Zustände niemals erneut besucht werden, wohingegen das im ersten Verfahren nicht sichergestellt wird.

In bezug auf die Berechnungskomplexität ist das zweite natürlich sehr viel aufwendiger, weil explizite Ungleichheits-Constraints ($\neq^{pm}$) nicht nur für negierte Constraints, sondern auch für die Negation des syntaktischen Teils eines Terms gebildet und verwaltet werden müssen. In Experimenten mit der WIM ist das Verfahren in Zeit- und Platzkomplexität etwa zehnmal schlechter als die einfachere Variante.

Im folgenden wird das erste Verfahren zugrundegelegt, das Zustandsmengen nur dann ignoriert, wenn sie Spezialisierung einer einmal besuchten Zustandsmenge sind (Gleichung 5.3).

Implementierung der Inklusionsbeziehung Die für die Überprüfung auf Inklusion erforderliche Komplementierung der im i-ten Schritt der Ausführung besuchten Repräsentation einer Zustandsmenge $\sigma_i =< \tilde{\sigma}_i, \mathcal{C}_{\sigma_i} >$ wird, falls $[\![\sigma_i]\!]$ nicht selbst Spezialisierung einer bereits besuchten Zustandsmenge ist, zur Laufzeit als ein Prädikat

$$\begin{array}{rll} state_i(X) \Leftarrow & & \mathcal{C}_{\sigma_i} = \top \Rightarrow \tilde{\sigma}_i \not\leq X \\ & \wedge & \mathcal{C}_{\sigma_i} \neq \top \Rightarrow \tilde{\sigma}_i \not\leq X \vee (\tilde{\sigma}_i \leq X \wedge \overline{\mathcal{C}}_{\sigma_i}) \wedge \sigma_i = X \end{array}$$

abgespeichert, wobei = die Unifikationsgleichheit ist. Auf die explizite Angabe von $\mathcal{C}_\nu$ (d.h. die Constraints über X) kann verzichtet werden, weil sie während der Laufzeit beim Aufruf von $state_i$ ohnehin mit X assoziiert sind. Die Überprüfung der Gleichheit von $\mathcal{C}_{\sigma_i}$ und $\top$ muß nur einmal berechnet werden.

In der Implementierung werden alle Constraints $\bigwedge_i c_i$, die mit den Variablen einer gegebenen Zustandsmenge assoziiert sind, aufgesammelt. Jedes c_i wird dann negiert (S. 165), und das resultierende negative Zustandsprädikat wird als Disjunktion negierter c_i gespeichert: $\bigvee_i \neg c_i$.

Um zu überprüfen, ob eine neu besuchte Zustandsmenge ϑ ignoriert werden kann, wird $state_i(\vartheta)$ für alle vorher gespeicherten Zustände σ_i, $1 \leq i \leq n$, ausgewertet. Falls $state_i(\vartheta)$ erfolgreich für alle i berechnet werden kann, gilt

$$\bigwedge_{i=1}^{n} \exists t \in [\![\vartheta]\!] \bullet \neg(unif(t, \tilde{\sigma}_i) \wedge mgu_{(t,\tilde{\sigma}_i)}(\mathcal{C}_{\sigma_i})).$$

Das bedeutet, daß ϑ entweder semantisch allgemeiner als oder semantisch unvergleichbar mit allen bereits abgespeicherten Zustandsmengen ist. Also wird ϑ komplementiert und als $state_{n+1}(\vartheta)$ abgespeichert. Falls hingegen die Berechnung von $state_i(\vartheta)$ für ein i fehlschlägt, ist ϑ spezieller als eine der vorher besuchten Zustandsmengen und kann ignoriert werden.

Eine Konsequenz dieser Strategie ist, daß einmal abgespeicherte Zustandsmengen nicht aus der Datenbank entfernt werden müssen. Um Speicher zu deallozieren, ist das natürlich eigentlich wünschenswert. Auf der anderen Seite bedeutet das Entfernen bereits gespeicherter Zustandsmengen, daß in jedem Schritt die neu besuchte Zustandsmenge mit allen bereits gespeicherten verglichen werden muß, um diejenigen Prädikate zu identifizieren, die gelöscht werden können. Verzichtet man darauf, so kann die neubesuchte Zustandsmenge bereits beim ersten Fehlschlag eines Aufrufs von $state_i$ ignoriert werden.

Zur Illustration der Aussage, daß auf das Entfernen von Elementen aus der Datenbank verzichtet werden kann, seien drei Zustandsmengen betrachtet, σ_1, σ_2 und σ_3 mit $[\![\sigma_1]\!] \subseteq [\![\sigma_2]\!] \subseteq [\![\sigma_3]\!]$. Wenn zuerst σ_1 gespeichert wird und dann σ_3, dann ist das Entfernen von σ_1 nicht notwendig. Beim Speichern von σ_2 wird festgestellt, daß σ_3 allgemeiner ist, und σ_2 wird nicht gespeichert.

Es gibt keinen Grund, die diskutierte Strategie, Zustände dann nicht zu besuchen, wenn allgemeinere Zustände vorher bereits besucht worden sind, nicht beliebig zu modifizieren. Das kann beispielsweise dadurch geschehen, daß erneut auftretende Zustandspaare oder -tripel ignoriert werden. Dadurch werden offenbar doppelte Besuche von Zuständen zugelassen, was das anfänglich geschilderte Problem abmildert, daß eine Funktionalität bei n Durchläufen korrekt ist, aber nicht beim $n + 1$-ten Durchlauf.

Negation von Constraints Offenbar beinhaltet die Berechnung von $\overline{\mathcal{C}}_\sigma$ die Negation von Constraints. In der Praxis treten im wesentlichen zwei Formen von Constraints auf: Membershipconstraints über endlichen, konstanten Bereichen sowie Gleichheits- und Ungleichheitsconstraints.

- Membershipconstraints sind von der Form $X \in \{c_1, \ldots, c_n\}$ für Atome c_i. Ihre Negation ergibt sich trivial als $X \neq c_1 \wedge \cdots \wedge X \neq c_n$. Dabei ist die Wahl des Ungleichheitsconstraints irrelevant, da alle c_i Atome sind. Membershipconstraints treten z.B. bei der auf S. 121 angegebenen Optimierung auf. Diese Constraints können suspendiert sein (wenn auf die Instantiierung einer Variable gewartet wird). Die Negation bezieht sich dann nicht nur auf die Membershipconstraints, sondern auch auf die Suspendierung, d.h. bei der Negation wird wiederum die Fallunterscheidung in variable oder nichtvariable Terme vorgenommen.

- Gleichheits- und Ungleichheitsconstraints werden trivial negiert, indem die Komparatoren jeweils durch ihr Komplement ersetzt werden. Das trifft sowohl auf alle im letzten Kapitel beschriebenen Ungleichheiten auf Termebene, als auch auf beliebige arithmetische Constraints zu. Die Negation von Operatoren wie $\geq, \leq$ erfolgt ebenfalls durch triviale Komplementierung.

Kompaktifizierung Die Speicherung von Zustandsmengen durch Prädikate erlaubt eine natürliche Form der Reduzierung der Anzahl der gespeicherten Prädikate. Aus Gründen der Einfachheit der Präsentation wird im folgenden davon ausgegangen, daß Zustandsmengen nicht wie im vorigen Abschnitt beschrieben komplementiert abgespeichert werden, sondern unverändert mit einem Prädikat $state'$. Die Übertragung auf die komplementierte Darstellung stellt keine Probleme dar.

Im folgenden wird der Einfachheit halber das Wort „Zustand" für die Termini „Zustandsmenge" und „Zustandsmengenrepräsentation" verwendet. Zur Illustration sei der Zustandsraum durch zwei Elemente definiert, von denen das erste vom Typ *Int* ist und das zweite von einem Typen, dessen Extension $\{a, b\}$ ist. Zwei Zustände $\sigma_1 = (1, a)$ und $\sigma_2 = (2, a)$ werden durch $state'(1, a) \Leftarrow true$ und $state'(2, a) \Leftarrow true$ gespeichert. Offenbar können diese zu einem Zustandsprädikat zusammengefaßt werden, nämlich $state'(X, a) \Leftarrow X \in \{1, 2\}$. Ein weiterer Zustand $\sigma_3 = (3, a)$ kann abgespeichert werden, indem dieses Prädikat durch $state'(X, a) \Leftarrow X \in \{1, 2, 3\}$ ersetzt wird.

Der Vorteil des Verfahrens liegt in reduziertem Speicheraufwand. Anstelle zweier Prädikate muß nur eines abgespeichert werden. Je länger die Zustandsvektoren sind, desto mehr Redundanz kann offenbar vermieden werden.

Die Ersetzung des zuletzt angegebenen $state'$-Prädikats durch $state'(X, Y) \Leftarrow X \in \{1, 2, 3\} \wedge Y \in \{a, b\}$ bei Anwesenheit eines neuen Zustands $\sigma_4 = (1, b)$ ist insofern illegal, als das Prädikat die nicht besuchten Zustände $(2, b)$ und $(3, b)$ ebenfalls repräsentiert. Das neue Prädikat abstrahiert die Menge der besuchten Zustände, indem es eine Art konvexer Hülle bildet (S. 168).

Kompaktifizierungen ohne die Bildung einer konvexen Hülle sind anwendbar in dem Sinn, daß sie die Menge gespeicherter Zustände nicht verändern, wenn für zwei Zustandsvektoren die Grundinstanzen der Projektion aller bis auf eine Komponente identisch sind. Für einen Vektor mit n Elementen bezeichne π_i für $1 \le i \le n$ die Projektion auf die i-te Komponente. Die Kompaktifizierung zweier Zustandsmengen σ und ϑ ist dann anwendbar, wenn

$$\exists i \bullet \bigwedge_{j=1}^{n} j \neq i \Leftrightarrow [\![\pi_j(\sigma)]\!] = [\![\pi_j(\vartheta)]\!] \tag{5.8}$$

gilt. Die den Zustandsmengen entsprechenden Prädikate $state'(X_1, \ldots, X_n) \Leftarrow \bigwedge_{j=1}^{n} \mathcal{C}^{\sigma}_{X_j}$ und $state'(X_1, \ldots, X_n) \Leftarrow \bigwedge_{j=1}^{n} \mathcal{C}^{\vartheta}_{X_j}$ können dann zu

$$state'(X_1, \ldots, X_n) \Leftarrow \Big((\mathcal{C}^{\vartheta}_{X_i} \vee \mathcal{C}^{\sigma}_{X_i}) \wedge \bigwedge_{j=1}^{n} (j \neq i \Rightarrow \mathcal{C}^{\vartheta}_{X_j}) \Big)$$

zusammengefaßt werden. Dabei ist es irrelevant, ob wegen der Identität im zweiten Konjunkt $\mathcal{C}^{\vartheta}_{X_j}$ oder $\mathcal{C}^{\sigma}_{X_j}$ gewählt wird.

Die Entscheidung, welche Zustandsprädikate zusammengefaßt werden, ist nicht immer eindeutig. Wenn $\sigma_1 = (a, b)$, $\sigma_2 = (c, d)$ und $\sigma_3 = (c, b)$ in dieser Reihenfolge besucht werden, dann kann σ_3 mit beiden zusammengefaßt werden. Die resultierenden Zustandsprädikate wären dann $state'(X, b) \Leftarrow X \in \{a, c\}$ und $state'(c, d) \Leftarrow true$ bzw. $state'(c, X) \Leftarrow X \in \{d, b\}$ und $state'(a, b) \Leftarrow true$.

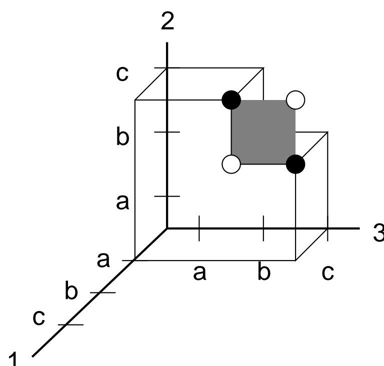


Abbildung 5.4.: Konvexe Hülle von (a,b,c) und (a,c,b)

Der Unterschied wird dann offenbar, wenn ein neuer Zustand $\sigma_4 = (a, c)$ besucht wird. Im ersten Fall muß ein neues Prädikat $state'(a, c) \Leftarrow true$ angelegt werden. Im zweiten kann eine Kompaktifizierung mit dem zweiten Prädikat zu $state'(a, X) \Leftarrow X \in \{b, c\}$ erfolgen. Da eine duale Situation bei Besuch des Zustands $\sigma_5 = (b, d)$ vorliegt und die Wahrscheinlichkeit, daß σ_4 und σ_5 erreichbar sind, ohne weiteres Wissen als gleich angenommen werden kann, spielt es allerdings keine Rolle, welche Variante gewählt wird. Probleme der Variablenordnung wie bei der Konstruktion von BDDs treten nicht auf, weil immer nur zwei Prädikate miteinander verglichen werden. Alternative Speicherstrategien, die eine Fallunterscheidung über die Komponenten des Zustandsvektors vornähmen, würden genau diese Probleme aufweisen (und auch zu noch kompakteren Speicherungen führen). Solche Strategien würden im wesentlichen auf einer Verallgemeinerung von BDDs auf andere Bereiche als die Booleans basieren.

Abstraktion Es ist natürlich möglich, Abstraktionen von Zustandsvektoren abzuspeichern, indem bestimmte Komponenten des Vektors weggelassen werden.

Beispiel 26 (Zustandsabstraktion). *Dieses Vorgehen wurde für die Berechnung bestimmter Testsuiten bei der Chipkarte verwendet, indem die Eingabe an das Modell ignoriert wird. Dadurch kann eine Testsuite erzeugt werden, deren Instanzen die Abdeckung des gesamten Datenzustandsraums des Modells erlauben. Bei zusätzlicher Berücksichtigung der Kommandos ist der Zustandsraum zu groß.*

Weitere denkbare Abstraktionen bestehen darin, mehrere Zustandsprädikate unter Informationsverlust zu einem zu kombinieren, (Pretschner, 2001). Als Beispiel für eine solche Zustandsabstraktion diene die folgende Kompaktifizierung, die für Zustandsmengen σ und ϑ dann anwendbar sei, wenn

$$\exists i \bullet [\![\pi_i(\sigma)]\!] = [\![\pi_i(\vartheta)]\!]$$

gilt. Wenn mehrere identische Projektionen existieren, i also nicht eindeutig ist, bezeichne $\mathcal{I} = \{i : [\![\pi_i(\sigma)]\!] = [\![\pi_i(\vartheta)]\!]\}$ die entsprechende Indexmenge. Die

PSfrag replacements

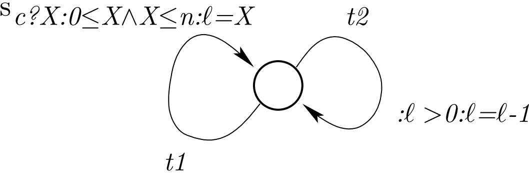


Abbildung 5.5.: Zustandsmaschine; Semantik abhängig von $[\![\cdot]\!]$

den Zustandsmengen entsprechenden Prädikate $\mathit{state}'(X_1,\ldots,X_n) \Leftarrow \bigwedge_{j=1}^{n} \mathcal{C}_{X_j}^{\sigma}$ und $\mathit{state}'(X_1,\ldots,X_n) \Leftarrow \bigwedge_{j=1}^{n} \mathcal{C}_{X_j}^{\vartheta}$ können dann zu

$$\mathit{state}'(X_1,\ldots,X_n) \Leftarrow \Big(\bigwedge_{j=1}^{n}\big(j \in \mathcal{I} \Rightarrow \mathcal{C}_{X_j}^{\vartheta}\big) \wedge \bigwedge_{j=1}^{n}\big(j \notin \mathcal{I} \Rightarrow (\mathcal{C}_{X_j}^{\sigma} \vee \mathcal{C}_{X_j}^{\vartheta})\big)\Big) \quad (5.9)$$

zusammengefaßt werden. Dabei ist es irrelevant, ob wegen der Identität als erstes Konjunkt $\mathcal{C}_{X_j}^{\vartheta}$ oder $\mathcal{C}_{X_j}^{\sigma}$ gewählt wird. Daß diese Zustandsspeicherung einen Informationsverlust beinhaltet, wird am Beispiel zweier zu speichernder Zustände (a,b,c) und (a,c,b) deutlich. Nach Gleichung 5.8 kann nichts zusammengefaßt werden. Als kompaktifiziertes Prädikat ergibt sich hingegen gemäß Gleichung 5.9 $\mathit{state}'(a,X,Y) \Leftarrow (X \in \{b,c\} \wedge Y \in \{c,b\})$. Dieses Prädikat speichert zwei Zustände zuviel, nämlich (a,b,b) und (a,c,c). In diesem Sinne wird die konvexe Hülle berechnet (Abb. 5.4; 1 (2,3) ist die Achse der ersten (zweiten, dritten) Komponente; die schwarzen Kreisflächen sind die Zustände (a,b,c) und (a,c,b) und die weißen die fehlenden Punkte ihrer konvexen Hülle).

Operationell bedeutet die Abstraktion von Zuständen, daß bei Berechnung der Kleene-Sequenz anstelle von $S_{i+1} \equiv S_i \vee \mathit{post}[T](S_i \wedge \neg S_{i-1})$ eine kleinere Menge berechnet wird, nämlich $S_{i+1} \equiv S_i \vee \mathit{post}[T](S_i \wedge \neg\alpha S_{i-1})$ für einen Prädikatentransformer α mit $S_{i-1} \Rightarrow \alpha S_{i-1}$. Der Fixpunkt wird unterapproximiert, weil für die Entscheidung, ob ein neuer Zustand besucht werden soll, die Menge der bereits besuchten Zustände vergrößert wird. Ob das für die Testfallgenerierung sinnvoll ist – vgl. dazu auch die verwandte Idee, ein großes BDD in mehrere kleinere Projektionen zu abstrahieren (Govindaraju, 2000) und somit die Menge erreichbarer Zustände zu überapproximieren –, oder ob sich identische Resultate durch zufälliges Ignorieren von Zustandsmengen ergeben, ist Teil zukünftiger Arbeiten. Zufälliges Ignorieren von Zustandsmengen entspräche der Technik des Beam-Search.

Definition von $[\![\cdot]\!]$ Die Definition von $[\![\cdot]\!]$ auf S. 158 beruhte auf allen typkorrekten Grundinstanzen aller Variablen des Arguments, die die assoziierten Constraints erfüllen. Das folgende Beispiel zeigt, daß es notwendig ist, alle Variablen zu betrachten.

Betrachtet wird eine Komponente mit einem Eingabekanal c und einer lokalen Variable ℓ, beide vom Typ *Int*. Die zugehörige Zustandsmaschine (Abb. 5.5) besteht aus einem Kontrollzustand und zwei (drei incl. Idle-Transition) Transitionen. Eine liest einen Wert von c und verlangt, daß er im Intervall $0\ldots n$ für

eine beliebige Konstante $n > 0$ liegt, um ihn dann ℓ zuzuweisen, d.h.

$$t_1 : c?X : 0 \leq X \leq n : \ell = X.$$

Unter der Voraussetzung $\ell > 0$ dekrementiert die zweite Transition ℓ,

$$t_2 : \ell > 0 : \ell = \ell - 1.$$

Es sei nun angenommen, daß nicht *alle* Variablen bei der Instanzenbildung berücksichtigt werden. Wenn der relevante Systemzustand als aus ℓ bestehend angenommen wird, dann feuert t_2 niemals, wenn t_1 einmal gefeuert hat. ℓ nach t_1 ist ein symbolischer Wert $\ell \in \{0, \ldots, n\}$. Nach t_2 wäre $\ell \in \{0, \ldots, n-1\}$ wegen Wächter und Zuweisung von t_2. Das ist eine Spezialisierung von $\ell \in \{0, \ldots, n\}$, was das Feuern von t_2 verhindert.

Der Grund für obiges Verhalten – der die später diskutierte Vollständigkeit nicht beeinflußt – liegt in der Tatsache, daß das Feuern von t_2 die durch t_1 gebundenen Werte im Nachhinein modifiziert: Für die auf den Wertebereich von ℓ projizierten Traces ergibt sich unter der Annahme $n \geq 2$ nach dem ersten Schritt (t_1): $< \{0, \ldots, n\} >$, nach dem zweiten (t_2): $< \{1, \ldots, n\}, \{0, \ldots, n-1\} >$, nach einem erneuten Feuern von t_2: $< \{2, \ldots, n\}, \{1, \ldots, n-1\}, \{0, \ldots, n-2\} >$ usw. Diese ex-post-Änderung wird nicht zur Zustandsspeicherung propagiert. Das ist durchaus Absicht, denn gespeicherte Zustände sind Backtracking-persistent. Die zu speichernden Zustandsmengen wären $\ell \in \{0, \ldots, n\}$, $\ell \in \{0, \ldots, n-1\}$, $\ell \in \{0, \ldots, n-2\}$ usw.

In der Prolog-Implementierung tritt dieses Problem nicht auf. Im Verlauf der symbolischen Ausführung wird in jedem Schritt eine neue Variable für jede Komponente des Zustandsvektors erzeugt. $X\{R\}$ bezeichne eine Variable X, deren Wertebereich auf R festgelegt ist. Nach dem Feuern von t_1 wird der Constraint $0 \leq X \leq n$ als Zustand abgespeichert. Nach dem ersten Feuern von t_2 wird eine neue Variable X' angelegt und *zwei* Constraints $1 \leq X \leq n$ und $X'\{0 \ldots n-1\} = X\{1 \ldots n\} - 1$ abgespeichert (genaugenommen, werden die negierten Constraints abgespeichert, s.o.). Nach dem zweiten Feuern von t_2 wird eine neue Variable X'' angelegt und *drei* Constraints abgespeichert, nämlich $2 \leq X \leq n$, $X'\{1 \ldots n-1\} = X\{2 \ldots n\} - 1$ und $X''\{0 \ldots n-2\} = X'\{1 \ldots n-1\} - 1$. Die Grundinstanzen dieser Constraintmengen sind unvergleichbar, weil sie über unterschiedlichen Variablenmengen rangieren.

Mächtigkeit von Constraintlösern Manchmal sind Constraintlöser ohne explizites Aufzählen der Wertebereiche der involvierten Variablen nicht in der Lage, die Unerfüllbarkeit eines Constraints zu entscheiden. Zwei Situationen sind zu unterscheiden: Constraints für die aktuell besuchte Zustandsmenge sind unerfüllbar, und die zu speichernde Negation von Zustandsconstraints ist unerfüllbar.

- Wenn aus Effizienzgründen darauf verzichtet wird, die Typannotationen einer Variable zu speichern, kann für eine Variable d, deren Typextension $\{c_1, c_2\}$ ist, bei Existenz eines Constraints $d \neq c_1 \wedge d \neq c_2$ dessen Unerfüllbarkeit nicht entschieden werden. Die zu speichernde Negation des

Constraints in einem Prädikat $state_j$ ist $d = c_1 \vee d = c_2$, was ohne weitere Constraints auf d natürlich stets erfüllbar ist. Das bedeutet, daß der Aufruf von $state_j$ in späteren Berechnungsschritten stets erfolgreich sein wird. Da eine Zustandsmenge nur dann als unbesucht angesehen und abgespeichert wird, wenn alle $state_i$ erfolgreich ausgeführt werden können, stellt diese Situation offenbar kein Problem dar: $state_j$, das einen nichterreichten Zustand repräsentiert, ist für die Entscheidung, ob ein anderer Zustand unbesucht ist, irrelevant.

- Die umgekehrte Situation hingegen kann durchaus Probleme aufwerfen. Wenn der mit einem Zustand assoziierte Constraint $d = c_1 \vee d = c_2$ ist, wird seine Negation in $state_j$ als $d \neq c_1 \wedge d \neq c_2$ abgespeichert, die unerfüllbar ist. Wenn nun ein neuer Zustand mit $d = c_1$ besucht wird, wird er wegen des Scheiterns des Aufrufs von $state_j$ als bereits besucht zurückgewiesen. Um solche Situationen zu vermeiden, kann auf den expliziten Erfüllbarkeitstest nicht verzichtet werden. In der Praxis tritt das Problem auf, ist aber von untergeordneter Relevanz, da für Datentypdefinitionen effiziente Repräsentationen existieren. Darüberhinaus existieren für große Datentypen wie den ganzen oder den reellen Zahlen i.a. potente Löser als Teil der Implementierung des CLP-Systems.

Die als problematisch identifizierte zweite Situation hat Einfluß auf das Zeitverhalten der Testfallgenerierung: Wenn ein neuer Zustand erreicht wurde, müssen seine assoziierten Constraints und insb. die übersetzte Datentypdefinition auf Erfüllbarkeit getestet werden, und seine zu speichernde Negation ebenfalls. Auf der anderen Seite führt ein gescheiterter Erfüllbarkeitstest für den aktuellen Zustand dazu, daß seine – bzgl. dieses Traces nicht erreichbaren – Folgezustände nicht untersucht werden.

Für den benötigten Speicher hat das ausschließlich positive Konsequenzen: Im Zweifelsfall werden noch weniger Zustandsprädikate abgespeichert. Insgesamt ergibt sich, daß das Problem in der Praxis kein große Bürde darstellt.

Komplexitätsbetrachtungen der Überprüfung von Zustandsinklusion bzw. der Folgerungsbeziehung zwischen Constraintmengen in der Literatur werden in Abschnitt 5.5 skizziert.

Vollständigkeit Im Abschnitt über verschiedene Suchstrategien wurde bereits der Einfluß der Zustandsspeicherung diskutiert. Wenn man davon absieht, daß nur Traces endlicher Länge betrachtet werden, dann läßt sich ein Vollständigkeitsbegriff definieren und nachweisen. Im folgenden sind *Traces* Sequenzen von Zustandsmengen eines Modells. Resolution im Zusammenhang mit (vollständigen) Constraintlösern ist nicht nur korrekt, sondern auch vollständig in dem Sinn, daß für jeden Zustand des minimalen Herbrandmodells, hier also jeden Zustand jedes berechneten Traces, ein allgemeinerer berechnet wird.

$\Sigma = \mu Y.init \vee post[T]Y$ sei die Menge aller ohne Zustandsspeicherung erreichbaren Zustände (bzw. symbolische Repräsentationen davon), Θ die Menge der mit Zustandsspeicherung erreichbaren Zustände (bzw. Repräsentationen davon), unabhängig davon, ob einmal besuchte Zustände exakt einmal oder

u.U. mehrfach besucht werden. $post'[T]$ bezeichne die Nachfolgerelation mit Zustandsspeicherung oder Zustandssubtraktion, d.h. $\Theta = \mu Y.init \vee post'[T]Y$. Es ist

$$\bigcup_{\sigma \in \Sigma} [\![\sigma]\!] = \bigcup_{\vartheta \in \Theta} [\![\vartheta]\!] \tag{5.10}$$

zu zeigen, d.h. Zustandsraumexploration mit Zustandsspeicherung ist

- korrekt im Sinn $\forall \vartheta \exists \sigma \bullet (\vartheta \in \Theta \wedge \sigma \in \Sigma) \Rightarrow (x \in [\![\vartheta]\!] \Rightarrow x \in [\![\sigma]\!])$ und
- vollständig im Sinn $\forall \sigma \exists \vartheta \bullet (\sigma \in \Sigma \wedge \vartheta \in \Theta) \Rightarrow (x \in [\![\sigma]\!] \Rightarrow x \in [\![\vartheta]\!])$.

Der Beweis erfolgt in Abschnitt B.4. Die Korrektheit ist trivial; für die Vollständigkeit wird ein Widerspruchsbeweis für einen minimalen Pfad bzgl. *post* angegeben, der zu einem Zustand führt, der nicht in Θ enthalten ist. Die Vollständigkeit bezieht sich also auf die besuchten Zustände und nicht die Menge aller möglichen Pfade.

5.3. Verwendung von Constraints

Dieser Abschnitt faßt noch einmal zusammen, an welcher Stelle Constraints für die Testfallgenerierung verwendet werden. Im wesentlichen sind die folgenden Einsatzgebiete identifiziert worden.

- *Constraints zur Testfallspezifikation.* Die Formulierung von $EF\varphi$ - Eigenschaften für Zustandsformeln φ wird durch einfache Membershipconstraints auf Listen definiert. In der Praxis werden sie in Konjunktion mit einfachen (bei großen Systemen endlich beschränkten) Eigenschaften kombiniert, die die Präsenz oder Absenz von Werten für alle Positionen eines Traces erzwingen. Spezifikationen auf der Basis von Constraint Handling Rules wurden in Kap. 3 diskutiert. Der Vorteil der Verwendung von Constraints liegt darin, daß Erfüllung oder Nichterfüllung der Spezifikationen nicht erst bei Erreichen der maximalen Suchtiefe überprüft wird, sondern in jedem Schritt. Das ist relevant bei großen Suchtiefen.
- *Constraints zur expliziten Beschneidung des Suchraums, d.h. Effizienz- und Umweltconstraints.* Um den Suchraum einzuschränken, können bei entsprechendem Wissen Teile des Modells zur Testfallgenerierung ignoriert werden (eine Art manuelles Slicing also (Tip, 1995)). Wenn beispielsweise Testfälle für die Cardholder Verification generiert werden, kann auf eine Berücksichtigung des Dateisystems in bezug auf alle Schreibkommandos verzichtet werden. Der Vorteil der Verwendung von Constraints liegt darin, daß nicht das Modell als solches modifiziert werden muß, sondern daß der erwünschte Effekt allein durch das Hinzufügen von Einschränkungen erzielt werden kann.
- *Constraints für die symbolische Ausführung zur Typisierung und zum Rechnen mit Zustandsmengen.* Wenn Wächter von Transitionen oder Konditionale in Zuweisungen und Funktionsdefinitionen Ungleichheiten enthalten, werden nicht alle bzgl. eines Typs korrekten Werte aufgezählt,

sondern es wird mit einer symbolischen Repräsentation, d.h. Mengen von Werten, gerechnet.[3] Wenn Wächter Membershipconstraints enthalten, z.B. $x = c_1 \vee x = c_2 \equiv x \in \{c_1, c_2\}$, können diese nach entsprechenden Transformationen der AutoFocus-Spezifikation direkt verwendet werden. Prinzipiell können Constraints beliebige Funktionsaufrufe involvieren. Die Idee ist, die Funktionen erst dann aufzurufen, wenn Ein- oder Ausgaben weitgehend instantiiert sind. Das kann Nichtdeterminismus reduzieren. Allerdings können so große Mengen von Constraints entstehen, und es müssen geeignete Heuristiken definiert werden, wann Funktionsaufrufe getätigt werden, wann also Ein- oder Ausgaben „genügend" instantiiert sind. Eine effiziente Überprüfung auf Zustandsinklusion oder die effiziente Berechnung von Zustandsdifferenzen ist mit Funktionsaufrufen als Teil der Zustandsrepräsentation nicht sofort offenbar. Dieser Ansatz wird im Werkzeug Gatel (Marre und Arnould, 2000) verfolgt.

- *Constraints zum Speichern von Zustandsmengen.* Die im Rahmen der symbolischen Ausführung errechneten Mengen von Werten werden zum Zweck der Zustandsspeicherung komplementiert. Die explizite Komplementierung von Termen im Herbranduniversum ohne Constraints ist wegen dessen normalerweise gegebener Unendlichkeit ohne Constraints unmöglich.

- *Constraints für diverse Definitionen der Negation.* Aus einer technischen Sicht ergibt sich nicht nur für die Zustandsspeicherung für die symbolische Ausführung häufig das Problem der Komplementierung von Mengen oder der Negation von (Un-)Gleichheiten (z.B. im Pattern Matching oder zur Definition von Idle-Transitionen). Mit Constraints können solche Negationsoperatoren elegant formuliert werden (s. die diversen Ungleichheiten aus Kapitel 4.

5.4. Experimente

Das Modell der WIM ist so komplex, daß eine Modellierung der Funktionalität allein mit Zustandsmaschinen auf der Basis von Kontrollzuständen nicht adäquat ist. Stattdessen wurden wesentliche Teile der Funktionalität als Transitionsrelation auf dem Datenraum einer EFSM definiert (s.S. 47), die mit funktionalen Programmen implementiert wurde. Das erhöht die Lesbarkeit des Modells. Auf der anderen Seite ist die Konsequenz, daß die Zustandsmaschinen im Normallfall aus einem und höchstens aus drei Kontrollzuständen bestehen. Um verständlich zu sein, würden Experimente zur Demonstration der vorgestellten Verfahren dann eine präzise Schilderung der Funktionalität und der zugrundeliegenden Logik in Form funktionaler Programmen erforderlich machen. Aus diesem Grund – und um die Wirkung der gerichteten Suche mit statischer Transitionsordnung zu demonstrieren – wird zur Demonstration der verschiedenen

[3] Wenn alle Wächter nur Gleichheiten enthalten und implizite Idle-Transitionen existieren, dann ergeben sich Ungleichheiten für die Idle-Transitionen.

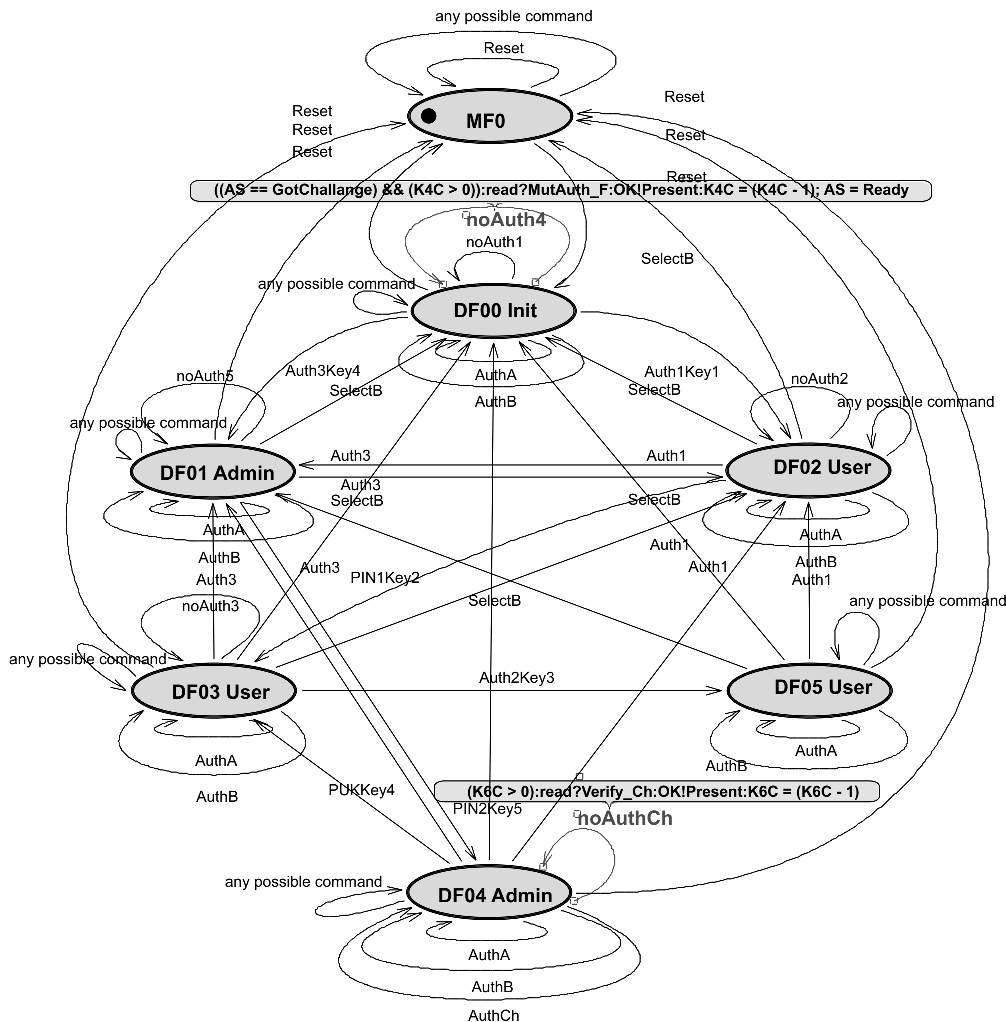


Abbildung 5.6.: Inhouse-Karte

Verfahren das Modell eines einfacheren Systems herangezogen, das nur aus *einer* Zustandsmaschine besteht.

Resultate zur Testfallgenerierung für die WIM – die auf alle präsentierten Techniken zurückgreifen – finden sich dann in komprimierter Form am Ende dieses Abschnitts. Die Güte der Testfälle wird im wesentlichen nicht diskutiert, weil sie eine Definition von „Güte“ voraussetzen würde. Im Fall der Testfälle für die WIM werden Überdeckungskriterien auf Modell- und Codeebene kurz diskutiert. Alle angegebenen Zahlen wurden auf einem Pentium III mit 750MHz gemessen.

5.4.1. Fallstudie 2: Inhouse-Karte

Das betrachtete Modell ist wiederum das einer Chipkarte (Abb. 5.6), das wiederum zusammen mit Validas AG und Giesecke&Devrient bearbeitet wurde. Die

Inhouse-Karte (Pretschner u. a., 2001b, 2003b) dient als Sicherheitstoken und wird beispielsweise zur Zugangskontrolle in Gebäuden verwendet. Eine Inhouse-Karte implementiert verschiedene Zugangsrechte, die ein Benutzer durch Eingabe einer entsprechenden PIN oder PUK erhalten kann. Die unteren sechs Kontrollzustände in Abb. 5.6 entsprechen sechs verschiedenen Zugangsrechten. Zwischen den einzelnen Zuständen wird durch geglückte oder mißglückte Verifikation einer PIN gewechselt. In jedem Zustand wird die Verifikation der PIN über ein Challenge/Response-Protokoll durchgeführt. Die einzelnen Schritte dieses Protokolls sind in einen entsprechenden Datenzustand ausgelagert. Mit jeder PIN sind Wiederholzähler assoziiert, die bei mißglückter Verifikation der PIN dekrementiert werden und bei Wechseln des Kontrollzustands – Authentisierung mit einer anderen PIN – wieder auf den Maximalwert gesetzt werden können. Reset-Transitionen in den Zustand MF0 verändern den Wert der Zähler nicht. Bevor in einem Kontrollzustand ein Zähler dekrementiert werden kann, müssen zwei der drei Phasen des challenge/response-Protokolls durchgeführt werden. Diese sind durch Schleifen in den Kontrollzuständen codiert; die dritte Phase führt sofort zum Dekrementieren des Zählers. Die Funktionalität entspricht in vereinfachter Form dem der Komponente `CardHolderVerification` der WIM.

Ohne Zustandsspeicherung scheitern alle Suchverfahren (multipliziert man die maximalen Zählerwerte $14, 3, 14, 14, 3, 15$, so ergibt sich ein Wert von 370.440 Zuständen; zusammen mit den Kontrollzuständen (7), den Kommandos (49) und den Datenzuständen für das Authentisierungsprotokoll (3) umfaßt der Zustandsraum somit weniger als $3 \cdot 10^8$ Zustände).

5.4.2. Suchverfahren

Für die verschiedenen Suchverfahren werden im folgenden für die Inhouse-Karte und die WIM Experimente zur Messung ihrer Leistungsfähigkeit angegeben. Am Ende dieses Abschnitts 5.4.2 erfolgt eine Diskussion. Für keines der Beispiele war die Suche ohne Zustandsspeicherung sowie die Suche ohne Zustandsspeicherung mit Interleaving von Transitionen erfolgreich.

Suche in der Inhouse-Karte

Zur Demonstration der verschiedenen Suchverfahren werden zwei Fälle betrachtet, die die Zähler der mit den Kontrollzuständen `DF00Init` und `DF04Admin` assoziierten PINs auf Null dekrementieren sollen.[4] Die Anfangswerte sind 14 bzw. 15. Die Tabellen 5.2 und 5.3 zeigen experimentelle Resultate für entsprechende maximale Suchtiefen für die Zähler 4 (DF00Init) und 6 (DF04Admin). Betrachtet werden die Suchverfahren mit Links-Rechts-Auswertung ($\rightarrow$) sowie randomisierte Suche auf Prozeß- und auf Threadebene (rP bzw. rT). Die Werte in Klammern hinter der Strategie geben die Anzahl derjenigen Experimente an, für die eine Lösung gefunden wurde. Wenn die Angabe fehlt, waren alle Experimente erfolgreich. Der Grund dafür, daß für die geringeren Suchtiefen nicht

[4]Da MF0 nur drei ausgehende Transitionen besitzt, wurde der Initialzustand mit DF00Init festgelegt, um parallele Ausführung sinnvoll zu machen.

Tiefe	Strat.	Zeit	Zustandsmengen	Speicher
150	$\rightarrow$	.5-.8-178	46-333-9470	185-512-14546
	$\rightarrow$: σ/μ	53/22	2852/1505	4381/2312
	rP	.8-102-4724	232-7405-52403	356-11374-80491
	rP:σ/μ	1798/1298	20286/18539	31180/28446
	rT	2.6-153-621	653-8773-17693	1003-13476-27177
	rT:σ/μ	264/336	6849/10918	16771/10520
300	$\rightarrow$	.5-.6-377	56-79-14440	86-122-22180
	$\rightarrow$: σ/μ	113/38	4297/1552	6560/2383
	rP	1-2.9-2432	354-956-37282	544-1467-57265
	rP:σ/μ	915/534	14284/10035	21941/15413
	rT	1.5-1.7-93	330-398-6907	507-612-10609
	rT:σ/μ	30/16	2253/1570	3460/2411
400	$\rightarrow$	.6-.7-4706	42-112-46366	65-172-71218
	$\rightarrow$: σ/μ	1111/371	13853/4808	21278/7385
	rP	1-2.9-2432	354-956-37282	544-1469-57265
	rP:σ/μ	915/534	14284/10	21941/15414
	rT	1.3-1.7-3.3	254-386-945	390-593-1451
	rT:σ/μ	.7/265	229/668	352/404
500	$\rightarrow$	.6-.7-2900	70-140-41984	108-215-64488
	$\rightarrow$: σ/μ	870/291	12527/4403	19242/6764
	rP	1-1.2-3083	410-471-43107	630-724-66213
	rP:σ/μ	924/378	13434/6667	20635/10240
	rT	1.4-1.7-285	243-444-12154	373-682-18669
	rT:σ/μ	1.4/30.2	243/1640	373/2519

Tabelle 5.2.: Suchverfahren für $cnt_4 \rightarrow 0$

alle Experimente erfolgreich sind, liegt in der Unvollständigkeit, die sich aus tiefenbeschränkter Suche mit Zustandsspeicherung ergibt (Abschnitt 5.1). Da es keine ausgezeichnete Transitionsordnung gibt, sind für jeweils zehn Experimente Minimum, Median und Maximum angegeben. Die Zeilen darunter enthalten Standardabweichung (σ) und Durchschnitt (μ). Der Unterschied zwischen $\rightarrow$ und den anderen Strategien besteht darin, daß im ersten Fall für jeden Kontrollzustand bei mehrfachem Besuch dieselbe Transitionsreihenfolge ausprobiert wird und im zweiten bei jedem erneuten Besuch eine zufällig ausgewählte Transitionsreihenfolge. Die Anzahl der Threads wurde für die Strategie rT ebenfalls auf zehn gesetzt.

Die zweite Spalte zeigt die benötigte Prozessorzeit in Sekunden für das Erreichen des Nullwerts, die dritte die Anzahl der besuchten symbolischen Zustandsmengenrepräsentationen und die vierte den benötigten Speicherplatz in KB.

Zu beobachten ist, daß Median und Maximum im Normalfall kleiner für Threads als für Prozesse sind. Für die Minimalwerte ergibt sich bei diesen Strategien kein einheitliches Bild; die Werte sind aber in etwa vergleichbar. Die Standardabweichung ist für parallele Threads im Normalfall deutlich kleiner. Der erhöhte Speicheraufwand bei parallelen Threads für die gleichzeitige Verwaltung von Suchstacks fällt nicht ins Gewicht. Um die jeweiligen besten Ergeb-

Tiefe	Strat.	Zeit	Zustandsmengen	Speicher
50	$\rightarrow$ (7)	13.4-72.3-269	2485-6349-12445	3817-9752-19116
	$\rightarrow: \sigma/\mu$	80/85	3600/5686	4731/9323
	rP (5)	58-242-346	5282-11578-13909	5282-11578-13909
	rP:σ/μ	95/219	2962/13909	4551/21365
	rT (4)	52-100-389	5072-7044-13756	7790-10033-21129
	rT:σ/μ	149/224	3824/9815	6024/14880
100	$\rightarrow$ (8)	.6-2.6-1026	23-745-24587	35-1145-37766
	$\rightarrow: \sigma/\mu$	359/202	8861/6133	13611/9420
	rP (9)	19-1057-2124	2955-24851-35255	4539-38171-54151
	rP:σ/μ	827/930	12880/19192	19783/29479
	rT (8)	30-96-1173	4091-7150-23534	6284-10982-36148
	rT:σ/μ	402/319	6662/10445	16044/10233
150	$\rightarrow$	.5-64-1498	47-5803-28363	72-8914-43566
	$\rightarrow: \sigma/\mu$	442/274	8385/8920	12880/13701
	rP	3.9-317-1714	1156-12643-31422	1775-19419-48264
	rP:σ/μ	620/656	11536/15734	17719/24167
	rT	3.2-145-1113	928-8816-22928	1426-13541-35218
	rT:σ/μ	318/267	6345/9656	9747/14832
200	$\rightarrow$	.9-70.5-685	349-6046-20271	536-9287-31137
	$\rightarrow: \sigma/\mu$	271/285	7487/10439	11500/16035
	rP	.9-10.4-196	357-2179-10513	549-3347-16148
	rP:σ/μ	64/48	3268/3888	5020/5973
	rT	3.4-129-1057	1002-8229-22532	1539-12640-3904776
	rT:σ/μ	413/355	7949/1.02E04	1.5E06/665951
300	$\rightarrow$	.7-46.3-9122	244-4622-59001	375-7099-111096
	$\rightarrow: \sigma/\mu$	2765/1353	17677/12686	32968/22107
	rP	96-1225-2314	7074-25550-35256	10866-39245-54153
	rP:σ/μ	701/1287	8894/24983	13660/38374
	rT	22-30-427	3354-3699-15094	5152-5682-23185
	rT:σ/μ	165/149	4724/7352	7257/11294

Tabelle 5.3.: Suchverfahren für $cnt_6 \rightarrow 0$

nisse auf einer Einprozessormaschine zu erzielen, müssen die Minimalwerte für die Zeit mit etwa zehn multipliziert werden; auf einer Zehnprozessormaschine ergeben sich die Minimalwerte als Resultat eines einzigen Prozesses und können damit bei den angegebenen Minimalwerten erzielt werden. Der minimal erforderliche Speicheraufwand muß in beiden Fällen mit zehn multipliziert werden. Die beiden Verfahren sind in etwa vergleichbar.

Am besten schneidet die Strategie $\rightarrow$ ab. Das liegt an der Struktur des Problems: Die Analyse der Suchergebnisse zeigt, daß gute Ergebnisse (wenig Zeit, wenig Speicher) dann erzielt werden, wenn nur wenige Kontrollzustände tatsächlich besucht werden. Die Wahrscheinlichkeit, ein gutes Ergebnis zu erzielen, ist für viele Permutationen der Transitionsordnungen vergleichsweise hoch; wenn sie also einmal zu Beginn der Suche festgelegt werden, ist die Wahrscheinlichkeit hoch, eine „gute“ Transitionsordnung für jeden Zustand auszuwählen. Wenn bei jedem Besuch eine neue permutierte Transitionsordnung ausgewählt wird, ist die Wahrscheinlichkeit höher, daß ein weiterer Kontrollzustand besucht

wird.

Gerichtete Suche Für die gerichtete Suche setzen sich die Fitneßfunktionen aus einer möglichst kleinen euklidischen Distanz zu diesen Kontrollzuständen (denn mit Transitionsschleifen dort werden die Zähler dekrementiert) und einer möglichst kleinen Distanz des entsprechenden PIN-Wiederholzählers zu Null zusammen. Für den Zähler des Zustands `DF04Admin` (cnt_6 mit Maximalwert max_6) ergibt sich dann

$$\varphi(\sigma) = 1/\delta(\pi_{ctl}(\sigma), DF04Admin) + (max_6 - \pi_{cnt_6}(\sigma))$$

Dabei bezeichnet $\pi_{ctl}(\sigma)$ die Projektion eines Zustands σ auf den Kontrollzustand und $\pi_{cnt_6}(\sigma)$ die Projektion auf den Wert des Zählers cnt_6. δ berechnet die minimale Transitionslänge zwischen zwei Kontrollzuständen. In jedem Schritt wird dann im Rahmen der gerichteten Suche φ für die nächsten möglichen Zustände berechnet, und die Transition mit dem maximalen φ wird ausgewählt (wenn es mehrere gibt, wird zufällig eine ausgewählt). Die Codierung der Transitionsordnung kann sowohl statisch als auch dynamisch erfolgen; die Berechnung der Testfälle für das Dekrementieren von Zählern erfolgt dann in Sekundenbruchteilen bei minimalem Speicheraufwand. Dasselbe gilt für eine Fitneßfunktion, die nur die euklidische Distanz auf Kontrollzuständen berücksichtigt und die auf die zusätzliche Information des aktuellen Zählerstands verzichtet.

Die Verallgemeinerbarkeit ist von der Existenz geeigneter einfacher Fitneßfunktionen und der „Monotonie“ des Suchraums abhängig: In den betrachteten Beispielen werden Werte immer nur dekrementiert, und jedes Dekrement erhöht die Güte des besuchten Zustands. Wenn solche Fitneßfunktionen nicht existieren, z.B. weil die zugrundeliegenden Datentypen ungeordnet sind, dann ist die Richtung der Suche schwierig. In solchen Fällen können syntaktische Kriterien (z.B. Anzahl der erfüllten Literale eines zusammengesetzen Prädikats) Verwendung finden. Der Erfolg solcher Heuristiken ist natürlich abhängig von der entsprechenden Anwendung.

Suche in der WIM

Für die WIM wurden analog drei Testfallspezifikationen zugrundegelegt. Die erste besteht darin, den Wiederholzähler einer PIN (Anfangswert 3) auf Null zu dekrementieren, was durch mehrfache falsche Eingabe einer PIN erfolgt. Die zweite soll zu einem Zustand führen, in dem der Wiederholzähler einer PUK (Anfangswert 10) auf Null dekrementiert werden. Die dritte schließlich ist die Suche nach einem Zustand, indem eine digitale Signatur erfolgreich berechnet werden kann; dazu sind das Setzen der korrekten Sicherheitsumgebung, die Verifikation mit einer PIN und das Setzen des Schlüssels erforderlich.

Die Tabellen 5.4 und 5.5 zeigen die Resultate der Experimente mit eingeschalteter Zustandsspeicherung. Betrachtet werden erneut die Strategie $\rightarrow$, in der für jeden Zustand jeder Komponente einmal festgelegt wird, wie seine Transitionsordnung aussieht, die Strategie rP, in der bei jedem Besuch eines

Tiefe	Strat.	Zeit	Zustandsmengen	Speicher
PIN-G-Counter→ 0				
4	→	3-4-15.1	86-108-491	347-485-2535
	→: σ/μ	3.4/5.6	117/163	627/766
	rP	3.9-6.1-7.4	110-179-218	447-839-965
	rP:σ/μ	.97/6.1	29/178	160/796
	rT	3.5-5.5-19.3	83-142-651	339-559-3483
	rT:σ/μ	4.3/7	155/197	869/937
10	→	3.1-5.4-13.7	87-135-384	350-960-22296
	→: σ/μ	3.6/6.9	100/166	8177/4852
	rP	3.5-10.2-15.9	88-237-706	363-2125-4431
	rP:σ/μ	4.1/9.8	204/306	1203/2029
	rT	3.5-6.3-14.5	71-159-415	552-1078-2491
	rT:σ/μ	3.3/7.6	98/196	689/1295
20	→	1.1-13.8-21.2	14-273-312	41-2375-2722
	→: σ/μ	7.1/12.6	106/199	1062/1843
	rP	2.7-7.6-18.9	53-235-423	362-1305-4082
	rP:σ/μ	4.8/7.9	111/206	1120/1549
	rT	2.6-7.9-18.6	40-171-706	275-1483-4011
	rT:σ/μ	5.6/10.9	255/338	1332/2193
PUK-G-Counter→ 0				
40	→	1.2-2125-14022	12-15703-31102	35-163049-412205
	→: σ/μ	6286/5452	14046/13477	174800/154100
	rP	.8-1.1-3.8	29-32-122	38-41-192
	rP:σ/μ	.8/1.38	27/43	192/112
	rT	.8-2.5-4.6	29-86-33	38-199-637
	rT:σ/μ	.7/2.7	18/40	144/245
50	→	.2-12802-17322	11-29916-36554	180-367292-444711
	→: σ/μ	8643/13447	19159/33228	90096/501776
	rP	.8-1.3-1.7	30-39-43	38-43-45
	rP:σ/μ	.3/1.2	4.0/36	2.1/41
	rT	2.1-2.4-4.5	27-32-94	165-191-727
	rT:σ/μ	.6/2.6	19/38	160/236

Tabelle 5.4.: Suche in der WIM (Zähler)

Kontrollzustands erneut eine Transitionsordnung festgelegt wird, und die Strategie rT, bei der zehn Threads parallel die Suche durchführen und über eine gemeinsame Tabelle der bereits besuchten Zustandsmengen verfügen. Alle Experimente wurden zehnfach wiederholt. Angegeben sind Minimum, Median und Maximum sowie in den Zeilen darunter Standardabweichung und Durchschnitt. Zustandsspeicherung ist erneut erforderlich, um die Suche zum Erfolg zu führen.

Zu beobachten ist erneut eine ungefähre Vergleichbarkeit aller Verfahren. Erneut ist die Varianz für parallele Threads und parallele Prozesse vergleichsweise gering. Im Fall des PUK-G-Zählers schneiden randomisierte parallele Prozesse und Threads im Schnitt besser ab als die Strategie →, in der für jeden Kontrollzustand einmal eine Transitionsordnung zufällig festgelegt wird und nicht in jedem Schritt. Im Vergleich zur Inhouse-Karte ist das ein abweichendes Verhalten. Die PIN können vier Kommandos dekrementieren, die PUK nur

Tiefe	Strat.	Zeit	Zustandsmengen	Speicher
4	$\rightarrow$	2.1-11.9-12802	23-292-29916	113-1511-167292
	$\rightarrow$: σ/μ	4231/1607	9836/3894	55029/21709
	rP	3.9-9-188	145-327-296	103-1056-1443
	rP:σ/μ	54/35	89/254	549/592
	rT	1.3-2.7-8.7	52-109-332	133-445-1363
	rT:σ/μ	2.4/3.6	113/143	540/606
10	$\rightarrow$	1.8-48.2-12242	14-1132-28888	76-5843-390123
	$\rightarrow$: σ/μ	5310/4460	15868/14238	133064/64965
	rP	.4-67-1800	14-1550-3132	80-7760-23041
	rP:σ/μ	638/370	3284/2197	3.4E04/1.9E04
	rT	.1-1.4-9.3	13-60-284	80-130-1401
	rT:σ/μ	2.4/2.7	72/107	271/442

Tabelle 5.5.: Suche in der WIM (digitale Signatur)

eins; das erklärt das unterschiedliche Verhalten für das Dekrementieren von PIN und PUK. Der Unterschied zum Verhalten bei der Inhouse-Karte liegt in der Struktur des Problems begründet: Das Kommando zur Dekrementierung des PUK-Zählers kann jederzeit gesendet werden; es gibt also weder besonders „gute“, noch besonders „schlechte“ Transitionsordnungen.

Gerichtete Suche Auch in der WIM kann die gerichtete Suche eingesetzt werden. Als Beispiel soll der Wiederholzähler einer PUK auf Null dekrementiert werden. Es zeigt sich, daß bei der Angabe der entsprechenden Fitneßfunktion auch für ein solches komponiertes System die entsprechenden Testfälle in Sekundenbruchteilen bei minimalem Speicheraufwand gefunden werden:

$$\varphi(\sigma) = max_{PUK} - \pi_{Retry_{PUK}}(\sigma)$$

Ein Beispiel für nicht-geordnete Typen ergibt sich, wenn nach einem Testfall gesucht wird, der zu einem Zustand führt, in dem die digitale Signatur berechnet werden kann. Dazu ist es nötig, daß die korrekte Sicherheitsumgebung und der Schlüssel gesetzt sind und die PIN verifiziert ist (Beispiel für die PIN-G):

$$\varphi(\sigma) = \pi_{PINStatus}(\sigma) + \pi_{SecEnv}(\sigma) + \pi_{Key},$$

wobei π die entsprechende Projektion berechnet und zugleich

- $\pi_{PINStatus}(\sigma) = 0$ setzt, falls $PINStatus = Verified$ und 1 andernfalls,
- $\pi_{SecEnv}(\sigma) = 0$ setzt, falls $SecEnv = SE1$ und 1 andernfalls und schließlich
- $\pi_{Key}(\sigma) = 0$ setzt, falls $Key = set$ und 1 andernfalls.

Wiederum erfolgt die Berechnung eines Testfalls in Sekundenbruchteilen bei minimalem Speicheraufwand.

Diskussion

Die Experimente zeigen, daß gerichtete Suche mit Zustandsspeicherung die beste Strategie darstellt. Wenn Fitneßfunktionen angegeben oder berechnet werden können und der Zustandsraum „gutartig“ ist, ist dieses die vorzuziehende Strategie.

Wenn die Voraussetzungen nicht gegeben sind, kann die Suche mit Wettbewerbsparallelität effizient gestaltet werden. Bei n unabhängigen Experimenten erfordert das den n-fachen Speicheraufwand des „besten“ Experiments; die benötigte Zeit ist abhängig von der Anzahl der zur Verfügung stehenden Prozessoren.

Im allgemeinen ist die Wahl der Ausprägung wettbewerbsparalleler Suche (→, rP, rT) abhängig von der Struktur des Problems. Wenn es gute Transitionsordnungen gibt und ihre Anzahl im Vergleich zur Gesamtzahl möglicher Transitionsordnungen hoch, dann wird die Strategie → bei geeigneter Auswahl der Prozeßanzahl zu guten Ergebnissen führen. Wenn die Zustände, die eine Projektion des Systemzustands auf die gesuchten relevanten Komponenten darstellen, viele Vorgänger besitzen und diese rekursiv wiederum viele Vorgänger, werden alle Suchstrategien in etwa gleich abschneiden. Wenn die Anzahl der (transitiven) Vorgänger gering ist, dann werden die Strategien mit stochastischer Auswahl von Transitionen in jedem Schritt besser abschneiden. Der Erfolg der Suche ist also vom Grad der Vernetzung des Suchraums abhängig. Wenn die Vernetzung gering und die Suche sehr speicherplatzaufwendig ist und demzufolge nicht viele Prozesse gleichzeitig ausgeführt werden können, dann wird eine sehr kleine Anzahl von Prozessen, die ihrerseits mehrere Suchthreads implementieren (Strategie rT), wegen der vergleichsweise geringen Varianz der Ergebnisse den aussichtsreichsten Kandidaten repräsentieren. Der zu erwartende, in den Experimenten aber nicht zu beobachtende Aufwand durch die Verwaltung mehrerer Suchstacks, tritt bei der Strategie rP nicht auf.

Wissen um das Problem kann die Suche immer verbessern. Im Beispiel der Inhouse-Karte etwa werden alle Zähler unabhängig voneinander dekrementiert. Dieses Wissen kann benutzt werden, um die gespeicherten Zustandsmengen zu abstrahieren (S. 167 ff.), indem nur der Stand desjenigen Zählers gespeichert wird, der dekrementiert werden soll. Die Suche ist dann fast deterministisch. Im Fall der WIM kann etwa die Datenstruktur der PIN-NR bei der Zustandsspeicherung ignoriert werden. Das ist einfacher, als die entsprechenden Kommandos mit Constraints zu verbieten, weil deren Anzahl hoch ist.

5.4.3. WIM: Funktionale und stochastische Spezifikationen

Für die WIM wurden für die drei Klassen von Testfallspezifikationen Testfälle errechnet. In diesem Abschnitt werden Testfälle auf funktionaler und stochastischer Basis diskutiert; strukturelle Testfallerzeugung wird im folgenden Abschnitt 5.4.4 behandelt.

Insgesamt wurden etwa 100.000 (unterschiedliche) Testfälle erzeugt, aus denen zufällig 1-2% ausgewählt wurden (jeder Testfall wurde mit der Wahrscheinlichkeit 1% ausgewählt). Diese Testfälle wurden dann auf die tatsächliche Karte

Testfallspezifikation	#Seq.	∅ Länge	Differenzen	Überdeckung
Digitale Signatur	322	10	0	15%
Security environment	32	38	20	40%
Card holder verification	275	10	10	49%
Modellüberdeckung (5)	312	5	8	42%
Requirements-Überdeckung	528	7	32	63%
Szenarien, Differenzen	41	7	14	60%
Summe	1506	8	84	93%

Tabelle 5.6.: Testfälle für die WIM

angewandt.

Tab. 5.6 zeigt entsprechende Daten. Die erste Spalte zeigt die Testfallspezifikation, die zweite die Anzahl der ausgewählten Sequenzen, die dritte die durchschnittliche Länge der Testfälle, die vierte die Anzahl der Differenzen zwischen dem Verhalten von Modell und Code, und die fünfte den Anteil an der Überdeckung aller möglichen Ein-/Ausgabepaare, die aus dem Modell erzeugt wurden, vgl. Beispiel 15 auf S. 86.

Die erste Zeile zeigt Testfälle, die aus der Zustandsmaschine für die Berechnung der digitalen Signatur abgeleitet wurden. Zum einen wurden randomisiert Testfälle ohne und mit Zustandsspeicherung erzeugt. Da es zum Test der digitalen Signatur gewollt ist, daß mehrere Modellzustände mehrfach besucht werden, wurde die Zustandsspeicherung zum anderen insofern modifiziert, als bei der Speicherung eines Zustands Paare und Tripel von aufeinanderfolgenden Zuständen abgelegt wurden. Damit konnte Transitionsabdeckung gewährleistet werden. Die geringe Abdeckung aller auftretenden Kommando-/Antwortpaare überrascht nicht, da zum Test der digitalen Signatur im wesentlichen dedizierte Befehle Anwendung finden.

Die Befehle für Sicherheitsumgebungen und Cardholder Verification wurden als besonders kritisch angesehen, weshalb für die entsprechenden Komponenten explizit eigene Testfälle erzeugt wurden (die dann automatisch in Testfälle für die WIM erweitert wurden). Hier wurden zum einen randomisiert mit Zustandsspeicherung Testfälle erstellt. Zum anderen wurde die Zustandsspeicherung abgeschaltet, und mit Constraints wurden Kommandodopplungen erzwungen. Nachdem eine PIN verifiziert wurde, wurden etwa mehrere Überprüfungen des Status der PIN mit dem Kommando `VerifyCheck` erzwungen. Darüberhinaus wurden nach jedem Kommando, das eine der beiden Komponenten potentiell modifiziert, Befehle erzwungen, die den Status der Karte indirekt überprüfen (Postambeln), etwa die (versuchte) Berechnung einer digitalen Signatur.

Die vierte Zeile schließlich bezieht sich auf die Generierung von Testfällen, die auf der vollständigen Exploration des Modells bis zur Suchtiefe 5 beruhen. Die vergleichsweise geringe Abdeckung von möglichen Kommando-/Antwortpaaren beruht auf der geringen Anzahl von Traces, die ausgewählt wurden (312 aus etwa 50.000).

In der fünften Zeile sind Testfälle repräsentiert, die explizit darauf abzielen, Kommando-/Antwortpaare abzudecken. Die erstellte Suite (Zustandsspeicherung, gerichtete Suche) deckt in der Tat alle Paare ab; die vergleichsweise geringe Überdeckung ergibt sich wiederum daraus, daß ein sehr geringer Prozentsatz der erstellten Traces ausgewählt wurde. Manche Kommando/-Antwortpaare kommen sehr häufig vor (beispielsweise die `CardHolderVerification`- oder `SecurityEnvironment`-bezogenen), andere eher selten, zum Beispiel die erfolgreiche Berechnung einer digitalen Signatur.

Die letzte Zeile schließlich repräsentiert einige wenige handgeschriebene Sequenzen, die die Gründe für die auftretenden Differenzen zwischen Modell- und Implementierungsverhalten erhellen sollen. Diese Differenzen treten auf, weil die entsprechenden Standards keine eindeutigen Spezifikationen enthalten und im Modell eine Interpretation ausgewählt wurde. Diese Differenzen sind keine „Fehler“, weil das abweichende Verhalten der Implementierung einer anderen Interpretation entspricht.

Die erreichte Abdeckung von Kommando- und Antwortpaaren beträgt 93%. Das liegt ebenfalls an Mehrdeutigkeiten in der Spezifikation; die fehlenden 7% repräsentieren Paare, die nicht vorkommen können. Ohne diese Paare beträgt die Abdeckung 100%.

Die Bewertung der Testfälle durch Domänenexperten bei Giesecke&Devrient beinhaltete zum einen die Messung der Codecoverage in der Kartenimplementierung. Die Abdeckung entspricht genau der der bei Giesecke&Devrient manuell erstellten Suiten. Angesichts der Schwierigkeit, die Qualität von Testsuiten zu bemessen, wurden die Testfälle zum anderen manuell inspiziert und mit den manuell erstellten Testfällen verglichen. Basierend auf einem intuitiven Verständnis von „Äquivalenz“ von Testfällen, wurden die automatisch erzeugten den manuell erzeugten als vergleichbar angesehen.

5.4.4. WIM: MC/DC und Kompositionalität

Schließlich wurde eine Testsuite für die WIM errechnet, die MC/DC auf Systemlevel erfüllt (sofern das möglich ist, d.h., sofern für die einzelnen Elemente wirklich MC/DC-Suiten existieren; vgl. S. 134). Diese Suite deckt die möglichen Kommando-/Antwortpaare ebenfalls maximal, d.h. zu 93% ab.

Wie bereits geschildert, ist das Vorgehen gestuft: Zuerst werden (1) Testfälle für einzelne Transitionen ausgerechnet, dann (2) Testfälle für die Zustandsmaschine, in der sich die Transition befindet (Präambeln) und schließlich (3) Testfälle für die aggregierende Komponente, sofern dieses nicht die Toplevelkomponente ist (d.h. für die hierarchischen Komponenten `SecurityOperations` und `CardHolderVerification`. Daraus wurden dann (4) Testfälle für das Gesamtmodell abgeleitet. Für diejenigen Funktionsdefinitionen, die von dieser Testsuite nicht abgedeckt wurden, wurden zuletzt erneut Testfälle für die entsprechende Transition, dann EFSM, dann Subsystem und schließlich das Gesamtmodell berechnet. In den Fällen (2), (3) und (4) ist die Suche von Pfaden erforderlich.

Die Übersetzung des Modells in funktionale Programme, die für die Berechnung der MC/DC-Suite erforderlich ist, benötigt etwa 45 Minuten. Die

Komponente	Subkomp.	Trans.	MC/DC	Ausführungen
WimPre	–	9	11	33
FileSystem	–	12	12	16
CardHolderVerification	6	Σ68	Π28	Σ187, Π2530
SecurityEnvironment	–	19	35	69
SecurityOperations	9	Σ51	Π41	Σ159, Π388
Misc	–	3	3	3
WimPost	–	5	32	6
ResponseBuffer	–	7	13	11
System	21	Σ174	407	Π94, 244

Tabelle 5.7.: MC/DC Testfälle für einzelne Komponenten

Berechnung der Gesamtsuite benötigt dann zusätzlich etwa 90 Minuten. Dabei wird die Technik der Zustandsspeicherung und das Verbieten bereits besuchter Zustände angewendet, wenn aus Testfällen für einzelne Transitionen Testfälle für die entsprechende EFSM generiert werden und wenn aus Testfällen für ein Subsystem Testfälle für das entsprechende integrierende System erzeugt werden. Model Checking mit SMV kann wegen der Größe des Modells für die Testfallgenerierung nicht verwendet werden. Da eine Anbindung expliziter Model Checker an AutoFocus nicht existiert, kann über die Verwendung von Spin oder Murφ keine Aussage getroffen werden.

Tab. 5.7 führt relevante Daten auf. Für jede Komponente des Systems ist die Anzahl der Unterkomponenten sowie die Anzahl der Transitionspfeile – wegen der Codierung der wesentlichen Logik in funktionalen Programme entspricht jeder Transitionspfeil einer Menge von Transitionen – angegeben. Für die hierarchischen Komponenten ist die Summe der Anzahl der Transitionspfeile ihrer Unterkomponenten angegeben. Die vierte Spalte schließlich zeigt die Anzahl der Testfälle, die für die entsprechende Komponente MC/DC erfüllen. Es handelt sich dabei um Pfade der entsprechenden Automaten, die vom Initialzustand der Komponente ausgehen. Für die hierarchischen Komponenten ist hier die Menge an Testfällen angegeben, die MC/DC für das komplette Subsystem erfüllen. Die letzte Spalte schließlich zeigt als Vergleichswert die Anzahl der symbolischen Schritte (Testfälle der Länge 1), die jede Komponente ohne Einschränkung des Ausgangszustands durchführen kann. Für die hierarchischen Komponenten sind wiederum die Summe der Schritte aller Unterkomponenten (Σ) sowie die Anzahl der Schritte der integrierten Komponente (Π) angegeben. Die Schritte sind insofern symbolisch, als sie Äquivalenzklassen von Zuständen und Ein-/Ausgaben darstellen – jede kombinierte Instantiierung der Variablen der symbolischen Ein- und Ausgaben führt zum Schalten derselben Transitionen (nicht nur Transitionspfeile) des Modells.

Die angegebenen Testfälle erfüllen MC/DC auf EFSM-Ebene. Damit sind aber noch nicht alle Funktionsdefinitionen abgedeckt (s.o.). Wenn bei Durchführung der Testfälle protokolliert wird, welche Funktionsdefinitionen bereits Anwendung gefunden haben und insbesondere, wie große Teile der entspre-

chenden MC/DC-Testfallspezifikation bereits durchgeführt worden sind, bleibt als letzter Schritt die Suche nach Testfällen für die WIM, die auch die fehlenden Funktionsdefinitionen bzw. MC/DC-Testfälle für die entsprechenden Definitionen abdecken. Insgesamt ergibt sich dann für die WIM eine aus 407 Testfällen bestehende Suite, deren Maximallänge in etwa der Anzahl des größten Wiederholzählers entspricht. Der Grund liegt darin, daß das Dekrementieren dieses Zählers auf Null den längsten Testfall im Modell darstellt, der erforderlich ist, um die entsprechende Funktionsdefinition anzuwenden.

Die Testfälle für die Komponenten `WimPost` und `ResponseBuffer` repräsentieren den Fall, daß die Anzahl der möglichen Ausführungen kleiner ist als die entsprechende Anzahl möglicher MC/DC-Ausführungen. Das liegt an Redundanzen in den erzeugten MC/DC Testsuiten. Für die anderen Komponenten ist die Anzahl der MC/DC-Testfälle bisweilen signifikant kleiner als die Anzahl der möglichen (symbolischen) Schritte der entsprechenden Komponente. Das ist genau im Sinn der überdeckungsbasierten Testfallgenerierung: Aus der Menge möglicher Transitionen sollen möglichst wenige „interessante" ausgewählt werden.

5.4.5. Diskussion

Das Beispiel der Chipkarte zeigt, daß modellbasierte Testfallgenerierung möglich ist. In diesem Abschnitt werden die einzelnen Techniken zur Testfallgenerierung anhand der Experimente kurz abschließend diskutiert; auf wirtschaftliche („Lohnt sich die modellbasierte Testfallerstellung?"), organisatorische („Wie lassen sich die Techniken in die Unternehmen tragen?") und Abstraktionsniveaus betreffende („Ist die explizite Definition des Zusammenhangs zwischen Modell und Testtreiber bzw. Implementierung nicht nur ad-hoc möglich sondern in ein allgemeines Schema einbettbar?") wird hier nicht eingegangen.

Die Testfallerzeugung erfordert angesichts der üblicherweise großen Zustandsräume die Vermeidung des doppelten Besuchs von Zuständen. Die präsentierten Techniken zur Speicherung von Zustandsmengen sind unabdingbar, auch wenn ihr Einsatz genau geprüft werden muß: Manchmal ist der mehrfache Besuch einer Äquivalenzklasse von Zuständen durchaus erwünscht.

Wenn das Transitionssystem nicht zu stark verzweigt ist, ist wettbewerbsparallele Suche eine effiziente Technik. Die Kopplung randomisierter Techniken mit Zustandsspeicherung ermöglicht die schnelle und speicherplatzsparende Berechnung.

Die gerichtete Suche ist anwendbar, wenn Fitneßfunktionen definiert werden können, die in bezug auf den zu suchenden Zustand tatsächlich den besten Folgezustand ermitteln können. Bei nicht geordneten Datentypen ist das schwierig. Für einzelne Komponenten ergibt sich als natürliche Grundlage solcher Fitneßfunktionen die euklidische Distanz auf Kontrollzuständen. Die Fitneßfunktionen können dann sogar statisch in die Transitionsreihenfolge codiert werden.

Für parallel komponierte Komponenten ist das schwieriger. Die Strategie der statischen Transitionsordnung kann dann nicht durchgeführt werden, wenn das Produkt des Zustandsraums nicht explizit gebildet wird, was im Rahmen des präsentierten Testfallgenerators auch nicht geschieht. Das Problem bei kom-

ponierten Systemen ist, daß die Fitneßfunktion sich im Normalfall auf eine oder sehr wenige Komponenten des Zustandsvektors bezieht und damit auch nur auf eine oder sehr wenige Komponenten des Gesamtsystems. Für diese Komponente kann dann die nächstbeste Transition ausgewählt werden. Damit wird die Anzahl der möglichen Transitionen der anderen Komponenten zwar eingeschränkt, aber zumeist existieren doch mehrere Möglichkeiten, die dann zufällig exploriert werden müssen.

Die kompositionale Testfallgenerierung ist machbar. Ausgehend von Testfällen für eine Komponente, konnte gezeigt werden, wie daraus schrittweise Testfälle für das integrierte System berechnet werden können. Dabei spielen wiederum Such- und Zustandsspeicherstrategien eine Rolle.

Die für die Chipkarte erstellten Testfälle sind schließlich intuitiv in ihrer Qualität manuell erstellten Testsuiten vergleichbar. Die Frage, ob die Erstellung des Modells mit automatischer Ableitung von Testfällen aufwendiger ist als die manuelle Erstellung von Testfällen, muß Gegenstand zukünftiger Forschungen sein. Die Errechnung von Testsequenzen (und nicht nur einzelnen Kommando-/Antwortpaaren) wird von den Domänenexperten als schwieriges Problem aufgefaßt, das mit den präsentierten Techniken gelöst werden kann. Der Vorteil liegt im hohen Automatisierungspotential: Im Zweifelsfall ist ein Testfall mehr besser als einer weniger, und (im wesentlichen identische) Chipkarten lassen sich sehr elegant parallel testen. Dazu ist nur eine genügend große Anzahl an Terminals erforderlich.

5.5. Verwandte Arbeiten

Suche Die Integration gerichteter Suchverfahren in Model Checking-artige Technologien wurde zeitgleich mit Pretschner (2001) von Edelkamp u. a. (2001) für Spin publiziert. Dort wird volles Model Checking unter Berücksichtigung von A*-Strategien untersucht. Anwendungen dieser Technologie finden sich im Java Pathfinder-Projekt (Visser u. a., 2000), für das gerichtete Suchverfahren in die entsprechenden Model Checker integriert wurden (Groce und Visser, 2002).

Eine Alternative zur präsentierten best-first-Suche besteht darin, genetische Algorithmen zur Suche zu verwenden (Tracey, 2000; Wegener, 2001). In beiden Fällen müssen geeignete Fitneßfunktionen definiert werden. Diese Arbeiten wurden bereits in Abschnitt 4.8 diskutiert.

CLP und Model Checking Die Zusammenhänge zwischen Model Checking und Logikprogrammierung werden von Fribourg (1999) analysiert. Die Aussage ist, daß zur Darstellung unendlicher Zustandsmengen endliche Repräsentationen – Constraints – benötigt werden, wie das z.B. Alur u. a. (1995) für hybride Automaten mit linearen Constraints über reellen Zahlen durchführen.

Delzanno und Podelski (1999) zeigen für ein CTL-Fragment (EF, EG, AF, AG), wie CLP zum Model Checking unendlicher Systeme verwendet werden kann und verwenden als Optimierungen Magic-Set-Transformationen und Widening-Operationen (Cousot und Cousot, 1977) zur Abstraktion. Da keine Produkt- und Summentypen verwendet werden, ist eine explizite Charakterisierung

der Inklusionsbeziehung auf Zustandsmengen, die sowohl die Negation von Termen, als auch die Negation beispielsweise arithmetischer Constraints beinhaltet, nicht notwendig.

Urbina (1996) verwendet CLP, um Eigenschaften hybrider Systeme nachzuweisen, die als hybride Automaten spezifiziert sind. Die Berechnung erreichbarer Zustände erfolgt im wesentlichen wie im Ansatz von Delzanno und Podelski (1999). Zustandsspeicherung spielt dabei keine Rolle.

Die Idee der tabulierten Resolution oder des Memoing (Warren, 1992) ist es, die Antworten bestimmter Anfragen abzuspeichern und sie später zur direkten Berechnung varianter Anfragen zu verwenden, ohne die entsprechenden Resolutionsschritte erneut durchführen zu müssen. Du u. a. (2000) benutzen tabulierte Resolution im Zusammenhang mit Constraints, indem Constraints Handles zugewiesen werden, die als Teil des Aufrufs von Prädikaten übergeben werden. Zweck ist auf tabulierter Resolution basiertes Model Checking (Cui u. a., 1998). Wie im Fall von Delzanno und Podelski (1999) werden Eigenschaften als Prolog-Regeln codiert, die dann zusammen mit der Transitionsrelation bei Vermeidung des doppelten Aufrufs von Prädikaten ausgeführt werden. Pemmasani u. a. (2002) benutzen Termdarstellungen von Difference Bound-Matrizen für die Darstellung von Uhren in sicheren gezeiteten Automaten; eine Erweiterung desselben auf tabulierter Resolution basierenden Model Checkers wird dann zum Nachweis von Eigenschaften verwendet. Allgemeine Constraints auf Termen werden nicht berücksichtigt. Nilsson und Lübcke (2000) definieren ein auf CLP basierendes Verfahren zum Model Checking eines bei Existenz konstruktiver Negation – nicht nur auf Termebene, wie im Rahmen dieser Arbeit propagiert, sondern auf Prädikatebene – vollständigen CTL-Fragments (EX, EU, EG), das für endliche Systeme die Berechnung größter Fixpunkte (EG) durch die Berechnung kleinster Fixpunkte emuliert. Die explizite Negation nicht nur beliebiger Constraints und Terme, sondern auch Prädikate, wird als gegeben vorausgesetzt. In den Arbeiten zum Model Checking mit CLP wird die für die Testfallgenerierung essentielle Frage der Erzeugung von Gegenbeispielen oder Zeugen nicht betrachtet.

Negation in CLP Apt und Bol (1994) geben einen Überblick über das Problem der Negation in der Logikprogrammierung. Stuckey (1991) zeigt, wie mit Constraints eine korrekte und vollständige Form der Negation für CLP-Programme definiert werden kann, ohne die Negation auf Grundterme zu beschränken. Eine syntaktische Charakterisierung der Inklusionsbeziehung bzw. Differenzbildung auf Mengen von Termen wird nicht gegeben, weil der Fokus ein anderer ist, nämlich ein Konzept für die Negation von Prädikaten. Die Arbeit ist motiviert durch die Einschränkung der SLDNF-Resolution (Lloyd, 1993) allein auf Grundterme, die negiert werden können. Grundterme tauchen im Rahmen von CLP aber natürlicherweise eher selten auf. Da in dieser Arbeit nur Terme und Constraints negiert werden, nicht aber der Aufruf von Prädikaten – erneut im Unterschied auch zu Nilsson und Lübcke (2000) –, kann auf die einfachen syntaktischen Charakterisierungen zurückgegriffen werden.

Model Checking Die hier präsentierten Techniken laufen im wesentlichen auf explicit-state Model Checking auf der Basis von Zustandsmengen hinaus. Die Ausnutzung der Einschränkung auf EF-Eigenschaften führt zu den sehr einfachen und effizient implementierbaren Charakterisierungen. Model Checking dient nicht primär dem Erzeugen von Traces. Der Grund für die Einschränkung ist, daß es zum Zweck der Testfallgenerierung nicht hilfreich ist, den Beweis beispielsweise einer Invarianz zu ermitteln. Das wurde bereits in Kapitel 3 diskutiert. Auf die Vollständigkeit des Model Checking wird ganz bewußt verzichtet, ist diese doch häufig der Grund dafür, daß Model Checker mit größeren Modellen nicht mehr umgehen können. Die hier diskutierten Techniken werden auch nicht als Konkurrenz zum Model Checking gesehen – zum Zweck der *Modellverifikation* kann und soll Model Checking, wenn es denn erfolgreich ist, durchaus eingesetzt werden. In technischer Hinsicht ergibt sich ein Unterschied zwischen der vorgestellten Technologie und vielen explicit-state Model Checkern daraus, daß das Produkt des Zustandsraums nicht explizit gebildet wird.

(Bounded) Model Checking zur Generierung von Testfällen wird u.a. von Ammann u. a. (1998) und Wimmel u. a. (2000) diskutiert. Farchi u. a. (2002) benutzen Murφ, um zustandsmaschinenbasiert Testfälle für Betriebssystem- und Exception-Handling-Routinen für Java zu generieren; Dushina u. a. (2001) im Werkzeug Genevieve denselben Model Checker zur Testfallgenerierung für digitale Signalprozessoren.

Zustandsspeicherung und -inklusion McMillan (1992) verwendet BDDs zum symbolischen Model Checking von CTL-Formeln. Der Nachteil solcher Verfahren ist, daß der Zustandsraum vor Überprüfung der Eigenschaft aufgebaut werden muß, was mit BDDs für große Systeme praktisch und für unendliche Systeme theoretisch nicht möglich ist. Dafür wird in jedem Schritt simultan mit allen möglichen Mengen von Werten gerechnet. Ein Beispiel für explicit-state Model Checking ist Spin (Holzmann, 1997), das nicht mit Mengen von Werten rechnet. Eine Kombination beider Techniken wird in der bereits zitierten Arbeit von Nilsson und Lübcke (2000) präsentiert, die auf CLP-Systemen mit integrierter tabulierter Resolution basiert. Bultan (1998) benutzt Presburger Constraints zur Repräsentation von Zustandsmengen. In bezug auf Abstraktionen von Zustandsmengen sei die Dissertation von Govindaraju (2000) genannt, in der ein großes BDD durch mehrere kleine approximiert wird.

Überprüfung auf Inklusion von Constraintmengen entspricht der Überprüfung einer Folgerungs- oder *globalen* Subsumptionsbeziehung auf Constraints. Eine Constraintmenge $\mathcal{C}$ wird *lokal* von $\mathcal{C}'$ subsumiert, falls jeder Constraint in $\mathcal{C}$ durch einen in $\mathcal{C}'$ subsumiert wird. Lokale Subsumption erfordert den Vergleich quadratisch vieler Constraints. Im Vergleich zur globalen Subsumption ergibt sich offenbar ein Informationsverlust, ist aber effizienter. Srivastava (1993) zeigt, daß lokale Subsumption für arithmetische Constraints polynomiell ist, während globale Subsumption in co-NP liegt. Delzanno und Podelski (1999) zitieren Resultate, nach denen globale Subsumption allgemein in co-NP liegt, wenn sie nicht mit lokaler Subsumption zusammenfällt.

5.6. Zusammenfassung

Inhalt dieses Kapitels sind für den praktischen Einsatz des Testfallgenerators unabdingbare Techniken zur Effizienzsteigerung im Sinn von Platz- und Zeitverbrauch. Diese beziehen sich auf Suchstrategien und Strategien der Speicherung von Zustandsmengen zur Vermeidung von Schleifen. Der Testfallgenerator stellt dann aus technischer Sicht einen (u.U. Bounded (Biere u. a., 1999)) Model Checker für die im Rahmen der Testfallgenerierung relevanten EF-Eigenschaften dar. Dabei werden die Vorteile des explicit-state und des symbolischen Model Checking kombiniert.

- Best-first-Suchstrategien basieren auf der Definition von Distanzmaßen vom aktuellen Zustand zum gewünschten Zielzustand. Die Transition, d.h. der Transitionspfeil und entsprechende Funktionsdefinitionen, deren Zielzustand dem gewünschten Zielzustand am nächsten kommt, wird zuerst ausprobiert, die mit der zweitnächsten Entfernung als zweites usw. Solche Strategien gehen davon aus, daß der Suchraum insofern monoton ist, als der kürzeste Pfad von einem Zustand zum Zielzustand mit der Transition beginnt, die zu einem Zustand führt, dessen Distanz zum gewünschten Zielzustand minimal ist. Fitneßfunktionen für geordnete Datentypen ergeben sich i.a. direkt. Als Distanzmaße können außerdem die euklidische Distanz auf dem Graphen der EFSM sowie die Anzahl erfüllter Literale für zusammengesetzte Zustandsbeschreibungen verwendet werden. Der Einfluß einer vorgegebenen maximalen Suchtiefe zur Einschränkung des Suchraums wurde diskutiert. Im Zusammenhang mit Zustandsspeicherung wurde gezeigt, daß es zu – in der Praxis wenig relevanten – Unvollständigkeiten kommen kann.

- Um den wiederholten Besuch von Zuständen zu vermeiden, können Spezialisierungen bereits besuchter Zustände ignoriert werden, oder vom aktuellen Zustand können alle bereits besuchten subtrahiert werden. Die aus Modellierungssicht wünschenswerte Verwendung von Produkttypen in AutoFocus erfordert besonderes Augenmerk auf geschachtelte Terme. Da sie in der Literatur in Abstraktionen nach Wissen des Autors nicht verwendet werden, gibt es dementsprechend keine Behandlung der Überprüfung auf Zustandsinklusion auf der Basis von geschachtelten Termen mit Constraints.

  Zwei elegante syntaktische Charakterisierungen der Inklusionsbeziehung, d.h. CLP–Programme, wurden vorgestellt.

  - Zur Überprüfung $[\![\nu]\!] \not\subseteq [\![\sigma]\!]$ eignet sich

    $$\begin{aligned} state_i(X) \Leftarrow \quad & \mathcal{C}_{\sigma_i} = \top \Rightarrow \tilde{\sigma}_i \not\leq X \\ \wedge \quad & \mathcal{C}_{\sigma_i} \neq \top \Rightarrow \tilde{\sigma}_i \not\leq X \vee (\tilde{\sigma}_i \leq X \wedge \overline{\mathcal{C}}_{\sigma_i}) \wedge \sigma_i = X, \end{aligned}$$

  - und für die Berechnung der Differenz $[\![K]\!] = [\![\nu]\!] - [\![\sigma]\!]$

    $$\begin{aligned} \kappa_\sigma(\nu, K) \Leftarrow \quad & K = \nu \\ \wedge \quad & \mathcal{C}_\sigma = \top \Rightarrow K \neq^{pm} \tilde{\sigma}' \\ \wedge \quad & \mathcal{C}_\sigma \neq \top \Rightarrow (K \neq^{pm} \tilde{\sigma}' \vee mgu_{(K,\tilde{\sigma}')}(\overline{\mathcal{C}}_{\sigma'})) \end{aligned}$$

für eine echte Umbenennung σ' von σ.

Die für die Testfallgenerierung ausreichende Einschränkung auf EF-Eigenschaften erlaubt die Verwendung dieser Charakterisierungen, die in beliebigen CLP-Systemen verwendet werden können. Der Einsatz ist dann sinnvoll, wenn mit geschachtelten Typen gerechnet wird. Systeme mit dem Overhead einer integrierten tabulierten Resolution werden also nicht benötigt. Der methodische Unterschied zwischen Model Checking und Testfallgenerierung liegt im verfolgten Ziel: Mit Model Checking sollen letztlich Eigenschaften nachgewiesen werden, während in der Testfallgenerierung Traces erzeugt werden sollen.

Abstraktionen und Kompaktifizierungen von Zustandsmengen ergeben sich sofort aus der Repräsentation als CLP-Prädikate. Die Berechnung von Inklusionsbeziehung und Komplement erfordert die Negation von Constraints. Hier werden nur – in der Praxis ausreichend – Gleichheits- und Ungleichheitsconstraints betrachtet, die vergleichsweise einfach konstruktiv zu negieren sind.

Der Einfluß der Techniken wurde mit verschiedenen Beispielen illustriert. Die Studie der Chipkarte hat ergeben, daß modellbasierte Testfallgenerierung machbar ist, und daß die erzeugten Testfälle mindestens so gut wie manuell erstellte sind.

Das Chipkartenbeispiel ist für die an AutoFocus angeschlossenen Model Checker zu komplex, weshalb hier keine Vergleiche mit anderen Technologien erfolgen. Auf die explizite Codierung des Modells in die SMV-Eingabesprache oder in Promela wurde verzichtet (die SMV-Anbindung von AutoFocus führt zu Übersetzungen, die nicht mehr modelcheckbar sind). Ohne die angegebenen Optimierungen können für viele Testfallspezifikationen keine erfüllenden Testsuiten generiert werden.

Die erzeugten Testfälle wurden abgesehen von der erzielten Coverage auf Code- und Anforderungsebene nicht quantitativ bewertet. Insbesondere wurden die manuell und die automatisch erstellten Testfälle nicht quantitativ miteinander verglichen. Der Grund ist, daß es keine allgemein akzeptierten Metriken gibt, mit denen sie verglichen werden könnten. Das ist eine Variation der Problematik der Bemessung der Qualität von Testfällen. So ist auch die Frage nach den Kosten der Erstellung nicht berücksichtigt worden. Das würde ein teures Experiment mit zwei unabhängigen Testentwicklerteams erfordern.

Weitere Arbeiten bieten sich in den folgenden Bereichen an:

- Um die Leistungsfähigkeit der Technologie mit der anderer zu vergleichen, müßten entweder Modelle in der entsprechenden Eingabesprache codiert werden, oder es müßten Übersetzungen aus der AutoFocus-Sprache implementiert werden. Bei verschiedenen Eingabesprachen ergibt sich die Schwierigkeit eines Vergleichs, weil werkzeug- bzw. sprachabhängig manuell Optimierungen eingebaut werden können. Die Modelle sind nur so gut wie derjenige, der sie modelliert. Bindet man hingegen verfügbare Model Checker an, so können umgekehrt die Vorteile eines spezifischen

Werkzeugs nicht ausgenutzt werden, weil nicht-optimierter Code aus dem Modell erzeugt wird. Als zukünftige Arbeit bietet sich letzteres Verfahren dennoch an. Interessant sind Anbindungen von Spin oder Murφ, wobei im ersten Fall aufgrund des asynchronen Ausführungsmodells explizite Synchronisationsmechanismen eingeführt werden müssen, die die Anwendung von Techniken wie der Partial-Order-Reduction (Godefroid und Wolper, 1994) schwierig machen. Nach Informationsstand des Autors sind für Murφ hingegen keine gerichteten Suchverfahren in das Werkzeug integriert. In beiden Fällen ist nicht unmittelbar einsichtig, wie die Generierung von Gegenbeispielen gesteuert werden kann, was in der vorgestellten Implementierung mit Constraints sehr elegant zu lösen ist. Die Angabe expliziter Transitionswahrscheinlichkeiten wird nicht unterstützt.

- Die Implementierung in LP-Systemen mit tabulierter Resolution zur a-priori-Verhinderung doppelt besuchter Zustände ist mit dem hier direkt in das einem Modell entsprechende CLP-Programm auf Unterschiede in der Effizienz zu untersuchen.
- Die Suche mit Wettbewerbsparallelität wurde auf Single-Prozessor-Maschinen durchgeführt. Eine Parallelisierung ist für Multi-Prozessor-Architekturen mit geteiltem Speicher bei entsprechender Unterstützung durch das CLP-System möglich. Die vermuteten Effizienzgewinne sind zu validieren.
- Schließlich bleibt zu untersuchen, wie der Ansatz auf Modellebene zu einem vollständigen Model Checking-Algorithmus erweitert werden kann.

6. Ergebnisse und Ausblick

Zentrale Ergebnisse dieser Arbeit sind

- die Konzeption und Diskussion verschiedener Ausprägungen des modellbasierten Testens,
- die Bereitstellung einer prototypischen technologischen Infrastruktur und
- der exemplarische Nachweis, daß modellbasierte Testfallgenerierung und -durchführung möglich sind.

Die Neuartigkeit des Ansatzes erzwingt seine Erprobung und Validierung mit bereits existierenden Systemen. Das ist im Fall der Chipkarte geschehen: Chipkarte und Spezifikation existierten zu dem Zeitpunkt, als die Methoden des modellbasierten Testens angewendet wurden. Die Resultate stimmen in bezug auf die Vision, explizite ausführbare Verhaltensmodelle

- als Grundlage für die Implementierung, d.h. als Spezifikation und
- als Grundlage für die Testfallgenerierung

zu verwenden, optimistisch. Modellierung als solche deckt i.a. zahlreiche Unstimmigkeiten und Auslassungen in den entsprechenden Spezifikationsdokumenten auf. Das motiviert den Einsatz ausführbarer Verhaltensmodelle zu Spezifikationszwecken. Wenn Code nicht aus dem Modell erzeugt wird, was gerade im Bereich eingebetteter Systeme mit starken Ressourcenbeschränkungen in aller Allgemeinheit nicht unmittelbar bevorzustehen scheint, dann können automatisch generierte Testfälle zum Test der Implementierung verwendet werden: Der Nachweis der Übereinstimmung von Spezifikations- und Maschinenverhalten kann strukturiert und automatisiert erfolgen.

Allgemein sind Methoden und Techniken zur Qualitätssicherung reaktiver Systeme Inhalt dieser Arbeit. Anhand expliziter und ausführbarer Modelle einer Maschine werden relevante Modelltraces generiert. Motivation ist die Beobachtung, daß aktuelle Testprozesse häufig unsystematisch, unstrukturiert, implizit und nicht reproduzierbar sind. Der Grund ist, daß das Sollverhalten üblicherweise nur implizit in den Köpfen der Entwickler bzw. der Testingenieure vorhanden ist. Diesem Mißstand wird mit expliziten Verhaltensmodellen begegnet. Um Modelle zu validieren, müssen diese bei Abwesenheit einer weiteren formalen Spezifikation manuell überprüft werden. Dazu können mit dem präsentierten Testfallgenerator erzeugte Testdaten und andere Qualitätssicherungsmaßnahmen wie etwa Reviews verwendet werden. Wegen des im Vergleich zu Code höheren Abstraktionsgrads stellt das einen Vorteil gegenüber der Validierung

von Code dar. Wenn das Modell nach einem iterativen Entwicklungs- und Qualitätssicherungsprozeß als valide angesehen wird, kann es zum Test der Maschine verwendet werden, indem die Ausgaben von Modell und Maschine miteinander verglichen werden.

Daraus resultieren vier fundamentale Fragestellungen:

- Was sind adäquate Beschreibungstechniken und Semantiken für die Notation von Modellen?
- Was sind „relevante" Modelltraces bzw. Testfälle, und wie werden sie erzeugt?
- Können die Abstraktionsniveaus von Modell und Maschine zum Zweck der Testdurchführung auf Maschinenebene überbrückt werden?
- Wie kann der Prozeß der Testfallspezifikation und -generierung in den Systementwicklungsprozeß eingebettet werden?

Der modellbasierte Test reaktiver Systeme betrifft also sämtliche Aktivitäten ihrer Entwicklung. Die Erstellung von Modellen als Abstraktionen tatsächlicher Systeme beeinflußt die Requirements Engineering-, die Spezifikations- und u.U. die Design- und Implementierungsaktivitäten (Codegenerierung). Die Definition „relevanter" Modelltraces ist Teil des Requirements Engineering, der Spezifikations- und der Testaktivitäten. Die Überbrückung der Abstraktionsniveaus von Modell und Maschine ist für den Test der Maschine gegen ihre Spezifikation sowie für die Implementierung und Generierung von Code oder Codefragmenten relevant. Schließlich müssen die Erkenntnisse aus dem Test von Modellen oder Maschinen möglicherweise in allen Phasen des Entwicklungsprozesses berücksichtigt werden.

Ob der aus den vorgeschlagenen Verfahren und Methoden resultierende Overhead diese Verfahren insofern rechtfertigt, als letztendlich bessere oder kostengünstigere Produkte erstellt werden können, muß die Praxis zeigen. Die im Rahmen dieser Arbeit betrachtete Fallstudie einer Chipkartenanwendung stimmt optimistisch.

6.1. Ergebnisse

Das zentrale Resultat dieser Arbeit ist ein in der industriellen Fallstudie der WIM erfolgreich eingesetztes System, das ausgehend von dem Modell einer Maschine und einer funktionalen, strukturellen oder stochastischen Testfallspezifikation eine Menge von Testfällen – Modelltraces – berechnet. Diese Testfälle sind immer nur so „gut" wie die zugrundeliegende Testfallspezifikation. Die Güte einer Testsuite ist u.a. wegen der Abhängigkeit von einer spezifischen Anwendung schwierig zu bemessen.

- Für funktionale Testfallspezifikationen wird deren Existenz vorausgesetzt; sie müssen vom Testingenieur geliefert werden.

- Überdeckungskriterien sind in ihrer Aussagekraft bzgl. der Qualität der entsprechenden Testfälle umstritten, besitzen aber den Vorteil, quantifizierbar und zur automatischen Berechnung von Testfallspezifikationen und Testfällen geeignet zu sein.

- Die Güte randomisiert erstellter Testfälle ist ebenfalls seit langem Gegenstand kontroverser Diskussionen. Ihr Vorteil liegt darin, daß sie vollautomatisch zu geringen Kosten erzeugt werden können.

Der Wunsch nach möglichst kleinen Mengen möglichst kurzer Testfälle stellt ein weiteres Kriterium dar.

Wenn ein Modell als valide angesehen wird, die Abstraktionsniveaus von Modell und Maschine überbrückt werden können und die Ausführung von Testfällen auf Maschinenebene nicht kosten- oder zeitintensiv ist, dann ist ein Testfall mehr besser als einer weniger. Der Test von Prozessoren wie dem in der Fallstudie untersuchten ist ein Beispiel dafür, daß die Prämissen erfüllt werden können.

Technologie Der Testfallgenerator basiert auf der symbolischen Ausführung von deterministischen AutoFocus-Modellen. Da Testfälle i.a. Mengen endlicher Traces sind, stellt sich die Frage, wie sie spezifiziert werden können. Pfaduniverselle Eigenschaften eignen sich nicht direkt, weil sie im Normalfall nicht einen Trace spezifizieren, sondern ein ganzes Transitionssystem mit Schleifen. Nicht zuletzt aus Gründen der einfachen Anwendbarkeit konzentriert sich diese Arbeit deshalb auf Testfallspezifikationen, die sich auf das Erreichen eines Zustands reduzieren lassen. Das ist für strukturelle Überdeckungskriterien und Szenarien aus Spezifikationsdokumenten der Fall. Überdeckungskriterien finden Verwendung sowohl auf der Ebene von Modellen von Maschinen, als auch auf der Ebene von Testmodellen, die eine Menge von Szenarien codieren.

Testfallgenerierung läßt sich als Suchproblem im Zustandsraum des Modells begreifen. Wächter und Zuweisungen von Transitionen führen zu Constraints, die Äquivalenzklassen von Werten induzieren. Es wird mit Mengen von Werten und nicht mit einzelnen Werten gerechnet. In der Praxis hat sich herausgestellt, daß das für die effiziente und effektive Ableitung von Testfällen nicht ausreichend ist, weshalb

1. schleifenvermeidende Verfahren zur Zustandsspeicherung sowie

2. die Integration gerichteter Suchverfahren

untersucht wurden. Es stellt sich heraus, daß die Verwendung von Constraints zu sehr eleganten und zumindest im ersten Fall effizient implementierbaren Charakterisierungen der Überprüfung von Zustandsinklusion und der Berechnung der Differenz von Zustandsmengen führt. Gerichtete Suchverfahren berechnen eine Distanz vom aktuell besuchten Zustand zu demjenigen, der durch die Testfallspezifikation spezifiziert ist. Ihr Gewinn ist stark abhängig von dem betrachteten Problem.

Die entstandene Technologie ist kompositional. Testfälle für eine Komponente können dazu verwendet werden, automatisch Testfälle für parallel komponierte Komponenten zu erzeugen.

Das wird am Beispiel der Erzeugung von MC/DC-Testsuiten demonstriert. Es wird gezeigt, wie mit syntaktischen Transformationen eines Modells MC/DC-Testsuiten für die einzelnen Beschreibungselemente von AutoFocus generiert werden. Die Beschreibungselemente umfassen Funktionsdefinitionen, Transitionen, EFSMs und schließlich Netzwerke von Komponenten. Der Testfallgenerator ist demzufolge auch direkt für funktionale Programmiersprachen und FSMs anwendbar. Resultat sind Testfälle, die MC/DC nicht nur auf Unitebene erfüllen, sondern sogar für das integrierte System. Damit wird einem zentralen Problem coveragebezogener Testfälle begegnet, das darin besteht, daß Testfälle auf Unitebene u.U. im integrierten System niemals ausführbar sind.

Prozeßbezug Die kompositionale Natur der Technologie erlaubt ihre Anwendbarkeit in inkrementellen Entwicklungsprozessen. Testfälle für ein Inkrement können verwendet werden, um automatisch Testfälle für ein späteres Inkrement zu berechnen.

Die betrachtete Testfallgenerierung ist also gewinnbringend in den Entwicklungsprozeß des Modells einbettbar. Die Einbettung in den übegreifenden Prozeß der Systementwicklung wurde sorgfältig analysiert. Im wesentlichen gibt es drei Szenarien:

- Modelle werden nach der Implementierung auf Grundlage letzterer erstellt,
- Modelle werden vor der Implementierung zum Zweck der Codeerzeugung erstellt, oder
- Modelle werden unabhängig vom Code für die Testfallerzeugung oder als virtuelle Umgebungskomponente erstellt.

Es stellt sich heraus, daß die zeitliche Abfolge der Entwicklung von Modell und Maschine insbesondere in Zusammenhang mit automatischen Codegeneratoren Konsequenzen für die Anwendbarkeit der Technologie zeitigt. Wenn Code und Testfälle aus demselben Artefakt generiert werden, dann wird dieses Artefakt im wesentlichen gegen sich selbst getestet.

Ein vielversprechendes Szenario in diesem Zusammenhang scheint durch das Zusammenspiel von Automobilherstellern und ihren Zulieferern gegeben zu sein. In diesem Fall sind die Autohersteller zunächst nicht daran interessiert, den Code der Systeme auch selbst zu entwickeln. Wenn sie allerdings ausführbare Verhaltensmodelle als Spezifikation zur Verfügung stellen, dann werden zum einen die Probleme mehrdeutiger und unvollständiger Spezifikationsdokumente abgemildert. Zum anderen kann eine Qualitätssicherung erfolgen: Aus den Modellen werden – auf Autoherstellerseite – Testfälle generiert, die nach Abgleich der Abstraktionsniveaus auf die vom Zulieferer erstellten Komponenten angewendet werden können. Prinzipiell können die Zulieferer Codegeneratoren verwenden, aber dazu müssen effiziente solche existieren.

Modellbasierung So vorteilhaft dieses Szenario erscheint, so ist seine Implementierung doch von vielen Faktoren abhängig. Die Durchsetzung modellbasierter Spezifikationstechniken hat natürlich Konsequenzen für die Firma, die an ihrer Einführung interessiert ist. Modellierung ist wie Programmierung eine anspruchsvolle Tätigkeit, die erlernt werden muß. Geeignete Abstraktionen zu finden, ist ein intellektuell anspruchsvoller und kreativer Prozeß. Organisationsbezogene Konsequenzen, die sich durch die Überlappung von Qualitätssicherungs- und Spezifikationsabteilungen ergeben, werden natürlich in jedem Fall zur Opposition vieler Beteiligter führen.

Ein zentrales Problem ist abgesehen davon die Verwendung geeigneter Sprachen und Werkzeuge für die Modellierung. Es wurde argumentiert, daß eine domänenspezifische Ausprägung dieser Sprachen – mit allen Vor- und Nachteilen – aufgrund einer möglichen semantischen Beschränkung der Beschreibungsmittel wünschenswert ist. Eingeschränkte Beschreibungsmittel mit einfacher Semantik sind leichter zu verstehen und analysieren als ganz allgemein verwendbare mit komplexer Semantik. Der Tradeoff besteht zwischen Ausdrucksmächtigkeit und Analysierbarkeit. Es wurde aufgezeigt, daß modellbasierte Beschreibungstechniken auf einer Separierung verschiedener Aspekte basieren sollten, soweit das möglich ist. Funktionalität und Kommunikation auf verschiedenen Abstraktionsstufen bis hinunter auf den Bus beispielsweise sollten zum Zweck der intellektuellen Beherrschbarkeit getrennt werden.

Solche Sprachen und sie unterstützende Werkzeuge existieren im Bereich der reaktiven Systeme zur Zeit noch nicht. Ein erster Schritt in diese Richtung ist der verstärkte Einsatz von Modellierungswerkzeugen wie ASCET-SD oder Matlab Simulink im Bereich der kontinuierlichen Systeme. Angesichts stetig wachsender Komplexität und Vernetzung heutiger Systeme ist eine Alternative zu diesem Vorgehen, das im Bereich der Businessinformationssysteme mit von der Kommunikation abstrahierenden Architekturen wie MDA bereits Wirklichkeit ist, schwierig vorstellbar.

Modelle finden in verschiedener Ausprägung Verwendung. Sie finden Anwendung als

- Modelle des Problems,
- Modelle der Maschinenumgebung, um Ausführbarkeit zu gewährleisten oder im Fall der Testfallgenerierung den Zustandsraum einzuschränken,
- Modelle der Maschine, d.h. der Lösung, zum Zweck der
  - Testfallgenerierung oder
  - Codeerzeugung

  und als
- Charakterisierung einer Menge von Szenarien, aus denen einzelne Testfälle abgeleitet werden.

Einschränkungen Der erfolgreiche industrielle Einsatz des modellbasierten Testens hängt davon ab, ob Methoden, Beschreibungstechniken und Werkzeuge in einer Form verfügbar sind, die auf Akzeptanz der Beteiligten stoßen. Die Verwendung modellbasierter Techniken für Entwicklung und Test hat Einfluß auf den gesamten Entwicklungsprozeß, wie weiter oben und in Kapitel 2 dargelegt wurde.

In technischer Hinsicht ist der erfolgreiche Einsatz der Verfahren zur Testfallgenerierung von der Komplexität der zu testenden Systeme abhängig. Zeitasynchrone Kommunikation etwa führt üblicherweise zu exponentiellem Blowup des Zustandsraums. Wenn keine adäquaten Abstraktionen gefunden werden können, die den Zustandsraum verkleinern, ist das Verfahren für sehr große Systeme nicht praktikabel. Das ist allerdings ein prinzipielles Problem und nicht ein Problem der vorgestellten Verfahren: Wenn der Zustandsraum zu groß wird, ist das üblicherweise eine essentielle und keine akzidentelle Eigenschaft des Systems (Brooks, 1986).

Die Beschreibungstechniken von AutoFocus bieten nicht für jede Form von Problemen einen adäquaten Rahmen für die Modellierung. Dynamische Aspekte etwa wie das Erzeugen und Zerstören von Komponenten zur Laufzeit werden nicht unterstützt. Wenn Probleme, die solche Beschreibungstechniken erforderlich machen, nicht adäquat mit den AutoFocus-Beschreibungstechniken beschrieben werden können – etwa durch eine fixe Grundmenge möglicher Komponenten, die mit einem Flag versehen werden, das ihre „Existenz“ im System anzeigt –, dann muß geprüft werden, inwiefern die Technologie zur Testfallgenerierung auf andere Beschreibungsmittel praktikabel übertragbar ist.

6.2. Zukünftige Arbeiten

Ausführliche Vorschläge für weitergehende Studien wurden insbesondere in den technisch geprägten Kapiteln 4 und 5 in den einzelnen Abschnitten gemacht und sollen hier nicht wiederholt werden.

Auch wenn erste industrierelevante Erfolge modellbasierter Techniken nicht nur im Test – wie in dieser Arbeit präsentiert –, sondern auch für die Entwicklung – Generatoren für Produktionscode aus Matlab- und ASCET-SD-Modellen – vorliegen, bedarf das Feld einer weiteren ausführlichen Erforschung. Darunter fallen domänenspezifische Formalismen, mit denen aussagekräftige und verständliche Modelle effizient erstellt werden können. Insbesondere im Bereich des Deployment sind hier noch grundlegende Arbeiten vonnöten. Dieses Problem ist Instanz der übergeordneten Fragestellung, wie die verschiedenen Abstraktionsniveaus von Modell und Maschine miteinander in Bezug gesetzt werden können. Wenn die eingangs erwähnte Vision von ausführbaren Spezifikationen auf vergleichsweise hohem Abstraktionsniveau Wirklichkeit werden soll und die Generierung von Produktionscode zunächst zurückgestellt wird, dann müssen Mechanismen gefunden werden, die die abstrakten Abläufe des Modells konkreten Abläufen der Maschine zuordnen.

Für die Testfallgenerierung lassen sich ad-hoc-Lösungen vermutlich immer finden: Im Fall der Chipkarte sind ein Beispiel die im Testtreiber implementier-

ten kryptographischen Operationen. Zugrunde liegen mathematische Formeln, die in diesem Fall explizit angegeben werden könnten. Im Fall des Tests eines Multimediabusses in Fahrzeugen treten häufig mögliche Permutationen von Signalen auf, die zwar prinzipiell mit Nichtdeterminismus modelliert werden können. Darunter leidet aber die Verständlichkeit des auszuführenden Verhaltensmodells. Eine Möglichkeit besteht hier darin, Permutationen von Signalen als Vektoren von Signalen zu notieren, für die dann explizit angegeben wird, daß ihre Reihenfolge irrelevant ist. Es würde also auf der Ebene des Modells eine bewußte Überspezifikation erfolgen. Die Ermittlung und Klassifizierung auftretender notwendiger Konkretisierungsbeziehungen zusammen mit geeigneten Beschreibungstechniken wird vom Verfasser zumindest im Bereich des modellbasierten Testens als ein wesentliches (lösbares) Problem angesehen. Die systematische Notation der Konkretisierungs- und Abstraktionsbeziehungen setzt voraus, daß auch für die konkrete Ebene formale syntaktische Schnittstellen existieren.

Ein weiteres Problem existiert unabhängig von der Modellbasierung und betrifft die entscheidende Frage, wie zu testende Eigenschaften ermittelt werden können. Die Frage nach der Qualität einer Testsuite kann dann definiert werden: Eine Testsuite ist gut, wenn sie alle wesentlichen Eigenschaften umfassend testet. Eine Klassifikation anwendungsspezifischer und allgemeiner Fehlerquellen bzw. Eigenschaften könnte ein Ansatz sein. Allgemeine Fehlerquellen sind zum Beispiel solche, die zu Laufzeitfehlern führen. In Analogie zu Typfehlern sind andere allgemeine Kriterien identifiziert worden: Identische Zuweisungen, toter Code oder unerlaubte Zugriffe auf gesperrte Variablen in kritischen Regionen. Anwendungsspezifische Klassifikationen von Fehlerquellen könnten sich aus der Evolution eines Produkts ergeben, wenn ein umfassendes Fehlermanagement durchgeführt wird und daraus Rückschlüsse auf besonders kritische Teile oder Eigenschaften des Systems gezogen werden können.

Teile der vorliegenden Arbeit – die behandelte Fallstudie – können als positiv ausgefallene Machbarkeitsstudie des modellbasierten Testens angesehen werden. Weitere Studien – wie die genannte des Multimediabussystems im Auto – sind notwendig, um den positiven Eindruck zu überprüfen. Eine ökonomische Bewertung der praktischen Einsetzbarkeit in der industriellen Praxis kann nur auf solchen empirischen Daten fußen. Nicht unerwähnt bleiben soll hier die Beobachtung, daß die Erstellung stetig komplexer werdender Systeme mit in der Praxis verwendeten Techniken der Entwicklung und Qualitätssicherung bereits heute an die Grenzen ihrer Machbarkeit stößt. Vermutlich liegt das weniger an der mangelnden Eignung bekannter Verfahren wie Reviews als daran, daß solche Verfahren nicht oder nicht angemessen eingesetzt werden. Eine Kombination des modellbasierten Testens mit Reviews auf Modell- und auf Codeebene erscheint sinnvoll.

Nichtdeterminismus

Wenn das zu testende System oder seine Umgebung nicht deterministisch sind, nicht durch explizite Verhaltensschnittstellen deterministisch gemacht werden (d.h. abstrahiert werden) und die vollständige Umgebung nicht durch determi-

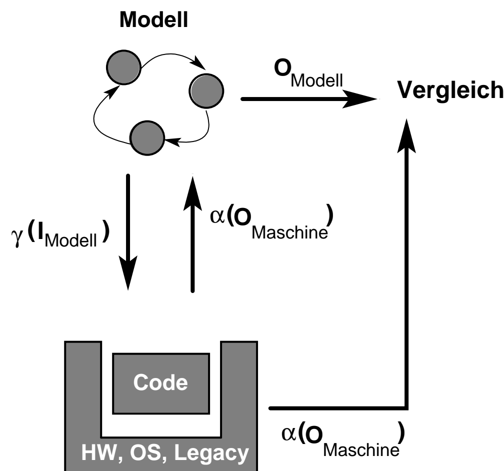


Abbildung 6.1.: Test nichtdeterministischer Systeme

nistische Stubs ersetzt werden kann – z.B. bei zeitasynchroner Kommunikation –, dann reichen Traces als Testfälle nicht aus (Kap. 2). Testfälle benötigen in diesem Fall eine Entscheidungslogik: Sie übernehmen ja die Rolle der Umgebung und sind in bezug auf ein bestimmtes Testziel definiert. Testfallgenerierung als Suchproblem mit Backtracking gestaltet sich in diesem Fall schwieriger, weil alle Eventualitäten im Testfall abgedeckt werden müssen.

Testfallgenerierung bedeutet dann, daß Transitionssysteme anstelle einzelner Traces generiert werden müssen. Solche Transitionssysteme könnten erzeugt werden, indem im Modell des zu testenden Systems für das Testziel irrelevante Zustände, Signale oder Signalfolgen mit Constraints verboten werden. Dabei muß natürlich gewährleistet bleiben, daß der Testfall (die Umgebung) auf jedes Signal von der zu testenden Implementierung reagieren kann, sofern diese Signale nicht sofort die Aussage zulassen, der Testfall sei gescheitert. Dieses eingeschränkte Modell läuft dann parallel mit der zu testenden Maschine. Dabei muß eine zwischengeschaltete Komponente den Abgleich der Abstraktionsniveaus zur Laufzeit übernehmen. Abb. 6.1 verdeutlicht den Zusammenhang: Konkretisierte Eingaben des ggf. eingeschränkten Modells werden in die Maschine gespeist. Deren abstrahierte Ausgaben werden in das Modell zurückgefüttert, und es findet ein Vergleich der abstrahierten Maschinenausgaben mit den Modellausgaben statt. Backtracking ist dann nicht möglich, im Gegensatz zum in dieser Arbeit vorgestellten Testfallgenerator.

Die Einschränkung des Modells ist genau durch das entsprechende Testziel festgelegt. Um einen bestimmten Zustand der Implementierung zu erreichen (die Testfallspezifikation), muß der Testfall (die Umgebung) zu jedem Zeitpunkt dasjenige Signal an die zu testende Implementierung senden, das die besten Chancen hat, die Implementierung in möglichst kurzer Zeit in den spezifizierten Zustand zu versetzen. Das kann z.B. mit Fitneßfunktionen geschehen, die im Rahmen der Testfallgenerierung für deterministische Systeme bereits diskutiert wurden. Für das einfache Beispiel der Kontrollzustandscoverage wäre

eine Möglichkeit, die bereits besuchten Zustände zu protokollieren und in jedem Schritt das Signal zu senden, das möglichst nahe an einen noch nicht erreichten Kontrollzustand heranführt.

Realzeitsysteme Der Test von Realzeit-Systemen wurde in dieser Arbeit nicht untersucht. Durch Einführung von Timer-Komponenten als Teil der Umwelt, die nichtdeterministisch einen Timeout senden oder nicht, kann das Zeitverhalten modelliert werden. Der Autor vermutet, daß solche Abstraktionen für den Test von Realzeit-Systemen ausreichend sind, wenn das konkrete Zeitverhalten auf Treiberebene abgeprüft wird. Testfallgenerierung für Realzeitsysteme wäre demnach mit den vorgestellten Verfahren machbar. Natürlich bedarf diese Vermutung einer empirischen Be- oder Widerlegung.

Kontinuierliche und gemischt diskret-kontinuierliche Systeme

Der präsentierte Testfallgenerator ist für alle Systeme geeignet, die sich mit AutoFocus spezifizieren lassen. Für bestimmte erforderliche – niedrige – Abstraktionsniveaus sind die Modellierungssprachen von AutoFocus nicht uneingeschränkt geeignet. Das ist der Fall für kontinuierliche und gemischt diskret-kontinuierliche Systeme. Solche Systeme werden in der Praxis häufig mit Werkzeugen wie Matlab Simulink/Stateflow spezifiziert. Abschließend soll skizziert werden, wie solche Systeme getestet werden könnten (Hahn u. a., 2003a).

Kontinuierliche Datentypen und komplexe funktionale Zusammenhänge lassen den Zustandsraum schnell zu groß für die systematische Exploration werden (die vollständige Analyse solcher Systeme für einfache Differentialgleichungen mit abstrakter Interpretation wird beispielsweise von Alur u. a. (1995) diskutiert). An diskreten, manuell erstellten Abstraktionen führt dann normalerweise kein Weg vorbei. Solche Abstraktionen bilden Klassen von Trajektorien des Systems. Ein Beispiel ist „das Auto beschleunigt schnell“ für eine Klasse von Geschwindigkeitstrajektorien, deren Ableitung über die Zeit in einem bestimmten Intervall liegt. Für den Test kann man dann zwischen open-loop und closed-loop-Systemen unterscheiden.

Im folgenden wird das Zusammenspiel von Regler und Strecke betrachtet. Getestet werden soll die Strecke. Für den Test des Reglers ergibt sich ein dualer Ansatz. Für open-loop-Systeme können die abstrakten Ausgaben eines abstrakten *Steuerungsmodells*, das ausgewählte Szenarien codiert, in entsprechende Trajektorien konkretisiert werden. Der Eingabeteil solcher Trajektorien wird in die Strecke gefüttert, und die Ausgaben werden protokolliert. Nach Abstraktion der Ausgabetrajektorien der Maschine können diese Abstraktionen mit den Ausgaben des diskreten Modells verglichen werden. Da die Abstraktionsabbildung sehr grob ist – das Steuerungsmodell codiert nur eine kleine Anzahl grob abstrahierter Szenarien – und die Verdiktbildung somit sehr schwierig, ist es erforderlich, die konkretisierten Eingaben des abstrakten diskreten Modells in ein *Modell der Strecke* zu füttern, ein gemischt diskret-kontinuierliches. Die Ausgaben dieses diskret-kontinuierlichen Modells werden dann mit den Maschinenausgaben verglichen. Das wird in Abb. 6.2 illustriert; für open-loop-Systeme ist die Rückkopplung von der Strecke zum Controller nicht nötig. Da die Differenti-

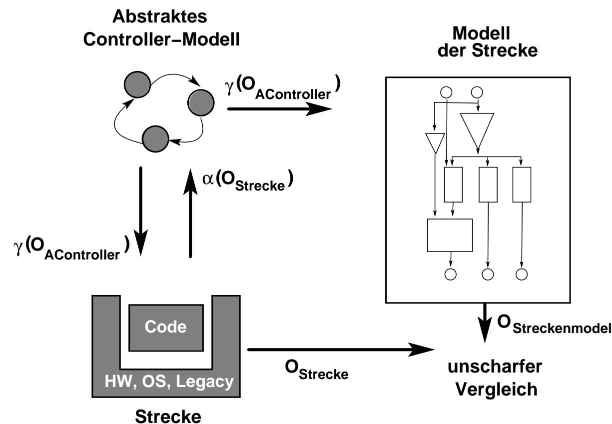


Abbildung 6.2.: Test kontinuierlich-diskreter Systeme

algleichungen im Modell häufig zu präzise und das Verhalten der Maschine nur ungefähr widerspiegeln, müssen Umschaltzeitpunkte und kontinuierliche Werte in bestimmten Toleranzen erlaubt werden.

Für closed-loop-Systeme ist die Situation noch schwieriger, da die Rückkopplung von Strecke zum Controller bzw. zum abstrakten Modell des Controllers berücksichtigt werden muß (Abb. 6.2). Prinzipiell ist das Vorgehen sonst identisch: Die Ausgaben $o_{AController}$ einer groben diskreten Abstraktion des Controllers werden in kontinuierliche Trajektorien konkretisiert, $\gamma(o_{AController})$.

- $\gamma(o_{AController})$ wird nach Überbrückung des Abstraktionsunterschieds zur Maschine auf die Strecke angewendet, und die Strecke liefert eine Trajektorie $o_{Strecke}$ als Funktion der Eingabe $\gamma(o_{AController})$.

- Ein gemischt diskret-kontinuierliches Modell der Strecke erhält als Eingabe ebenfalls $\gamma(o_{AController})$ und liefert die Ausgabetrajektorie $o_{Streckenmodell}$.

Unter Berücksichtigung von Toleranzen (Wiesbrock u. a., 2002) im Zeit- und Wertbereich wird dann ein Verdikt als Resultat des Vergleichs von $o_{Strecke}$ und $o_{Streckenmodell}$ gebildet. Wegen des Feedbacks von Strecke zu Controller wird außerdem $o_{Strecke}$ grob diskret abstrahiert, und diese Abstraktion dient als Eingabe für das grobe diskrete Modell des Controllers.

Für den Test ist dann festzulegen, welche Traces des abstrakten Controllermodells interessante Testfälle darstellen. Wenn einerseits das abstrakte diskrete und das gemischt diskret-kontinuierliche Modell technisch so gekoppelt werden können, daß Backtracking möglich ist – dazu muß das diskret-kontinuierliche Modell „angehalten“ werden können – und andererseits das Modell der Strecke ein sehr präzises ist, dann können Testfälle, d.h. Traces des abstrakten Controllers, wie in dieser Arbeit beschrieben berechnet werden. Dazu werden dann diskretes Modell des Controllers und gemischt diskret-kontinuierliches Modell der Strecke verschaltet. Dabei wird die Ausgabe des kontinuierlichen Modells anstelle der Ausgabe der Strecke nach Abstraktion in den Controller rückgefüttert. Wenn die Voraussetzungen – präzises kontinuierlich-diskretes Modell, Möglich-

keit zum Anhalten des Modells – nicht gegeben sind, kann wie im Fall nichtdeterministischer Systeme versucht werden, Überdeckungsmaße zur Laufzeit mit Heuristiken zu erfüllen.

Inwieweit dieser Vorschlag ein gangbares Verfahren darstellt, muß die Praxis zeigen. Hahn u. a. (2003a) zeigen für das Beispiel einer adaptiven Geschwindigkeitskontrolle mit Tempomat im Fahrzeug, daß der Ansatz prinzipiell anwendbar sein kann.

Zusammenfassung zukünftiger Arbeiten

Zusammenfassend ergeben sich neben den Vorschlägen in den Kapiteln 3, 4 und 5 die folgenden weiteren Arbeitsgebiete:

- Techniken und Methoden zur Validierung und Qualitätssicherung von Modellen,
- Konzepte, Beschreibungsmittel und Techniken zur Überbrückung der Abstraktionsniveaus von Modell und Maschine,
- Klassifizierungen „interessanter“ Eigenschaften oder Fehlerklassifikationen,
- Testfallgenerierung für nichtdeterministische Systeme,
- Testfallgenerierung für gemischt diskret-kontinuierliche und Realzeitsysteme,
- Optimierungen des Testfallgenerators,
- Algorithmen zur Reduktion von Testsuiten,
- eine empirische Überprüfung der Vermutung, daß Testfallgenerierung für Realzeitsysteme mit den vorhandenen Techniken ohne Modifikationen praktikabel ist,
- die Übertragung der Technologie zur Testfallgenerierung auf andere Beschreibungstechniken wie etwa Statecharts oder Java-Programme,
- eine Analyse des Kosten-Nutzen-Verhältnisses des modellbasierten Testens insbesondere im Vergleich mit alternativen Techniken wie Reviews, wobei die Vermutung ist, daß der Hauptaufwand in der Erstellung des Modells liegt, das für Reviews ebenfalls notwendig wäre, sowie
- eine Analyse möglicher Verquickungen von modellbasiertem Testen und anderen Techniken der Qualitätssicherung, um ein akzeptables Kosten-Nutzen-Verhältnis zu erzielen.

A. Programmiersprachen: Abstraktionen und Domänenabhängigkeit

A.1. Abstraktionen und Programmiersprachen (2.3.1)

Die Entwicklung von Programmiersprachen scheint mit der Einführung jeweils höherer Abstraktionsniveaus einherzugehen (von Neumann, 1960; Barnes, 1983, S. 3):

„Programme" In den frühen Analogrechnern wurde ein Steuerungsglied (z.B. Integrator oder Differentialgetriebe) für jedes Element der auszuführenden Berechnung eingesetzt. Die Einführung von Registern insbesondere in den frühen Digitalrechnern ermöglichte es dann, pro grundlegender Berechnung nur ein Steuerelement vorzusehen. Der Schritt von durch Stecktafeln fest verdrahteten Programmen hin zum Einsatz flexibler *Pläne* – Programme, vorweggenommen durch Zuses Plankalkül (Bauer und Wössner, 1972) – auf Basis bedingter Sprungbefehle mit der bahnbrechenden Idee, Daten und Programme durch dieselben Größen darzustellen, stellt einen fundamentalen Abstraktionsschritt dar. Weitere solche Schritte lassen sich in bezug auf Kontroll- und Datenfluß, auf die Programmstruktur und auf die Entwurfsmethodik ausmachen:

Kontrolle, Daten Die Entwicklung von Hochsprachen wie Fortran und Algol[1] erlaubte die abstraktere Beschreibung eines Problems bzw. seiner Lösung, indem die Existenz von Registern transparent für den Programmierer wurde: Komplexe Ausdrücke traten an die Stelle von Folgen von Registerbewegungen. Eine teilweise Maschinenunabhängigkeit war damit erreicht. Mit Lisp und Algol erfolgte die Einführung konditionaler if-then-else-Strukturen durch McCarthy (1978), die von einem Übersetzer in entsprechende Sprunganweisungen übersetzt werden. Die später Verbreitung findenden Exceptions stellen eine weitere Abstraktion im Bereich des Kontrollflusses dar. Diese Sprunganweisungen mußten in den Fortran-Dialekten explizit gemacht werden. Mit Sprachen wie Algol und Pascal (Wirth, 1971b) erhielt das Prinzip der Datenabstraktion Einzug in die Programmiersprachen. Zusammengehörige Daten können strukturiert werden. Der Zugriff erfolgt mit qualifizierten Bezeichnern.

[1] Die formale Definition der Syntax mit BNF stellt natürlich ebenfalls einen wesentlichen Abstraktionsschritt auf der Metaebene dar.

Strukturierung Im Bereich der Programmstruktur bieten Unterprogramme und Modularisierung eine Möglichkeit, Programme zu strukturieren (Wirth, 1971a; Parnas, 1976). Eine Trennung von Programmblöcken für dedizierte Aufgaben wird damit möglich. Das Konzept der Klasse und Unterklasse (Dahl und Nygaard, 1967) erlaubt die Strukturierung und konzeptionelle Integration von Daten und Operationen. Jackson Structured Programming (Jackson, 1976) propagiert die Übereinstimmung einer Programm- mit der zugrundeliegenden Datenstruktur sequentieller Dateien. Die Einführung von Schichtenarchitekturen erlaubt die Konzentration auf die in jeder einzelnen Schicht behandelte Problemstellung. Komponenten, Bibliotheken und Frameworks stellen weitere Strukturierungsmöglichkeiten dar.

Nebenläufigkeit, Kommunikation Zu Semaphoren äquivalente Primitive für die Prozeßsynchronisation und -kommunikation zeigen beispielsweise in Form von Monitoren eindringlich den Vorteil abstrakter Formulierungen fundamentaler Problemstrukturen. Garbage Collectors für moderne objektorientierte, funktionale und logische Programmiersprachen abstrahieren von der expliziten Speicherverwaltung. Middleware wie CORBA, .NET oder MDA (Soley, 2000) erlaubt die Programmierung (mehr oder weniger) unabhängig von der in einem System eingesetzten technischen Kommunikationsinfrastruktur.

A.2. Domänenspezifische Programmiersprachen (2.3.1)

Domänenspezifische Sprachen behaupten sich trotz der unverkennbaren Verbreitung von C oder Java durchaus:

APT Als frühes Beispiel läßt sich APT für die Programmierung numerisch gesteuerter (NC) Maschinen nennen.

Fortran, COBOL Fortran ist für numerische Berechnungen konzipiert. COBOL als Ersatz für das unzureichend erscheinende Flow-Matic (Sammet, 1972, 1978) mit dedizierten Konstrukten für die Manipulation heterogener Datenstrukturen, Dezimalarithmetik, Reporterzeugung und Massenspeicherzugriffen (Glass, 1999) ist ein weiteres Beispiel für eine weitverbreitete Sprache mit expliziter Einschränkung auf eine Domäne, hier die geschäftlicher Anwendungen.[2]

HDL, Blockdiagramme Auch Hardwarebeschreibungssprachen wie VHDL und Verilog für den Schaltungsentwurf oder die Blockdiagrammnotation für kontinuierliche Systeme exemplifizieren den Einsatz domänenabhängiger Sprachen – allerdings kann sie in der Domäne der kontinuierlichen Probleme durchaus als general-purpose angesehen werden.

[2] Interessanterweise fordern COBOL-Compiler die explizite Angabe der Zielplattform im Programm (Eipper, 1986) – kein Beispiel für gelungene Abstraktion.

Lustre, Esterel Formalismen für die Entwicklung zeitsynchroner reaktiver Systeme wie Lustre (Caspi u. a., 1987) oder Esterel (Berry und Gonthier, 1992) sind klar auf die Beschreibung des Daten- bzw. Kontrollflusses ausgerichtet und verfügen, anders als beispielsweise AutoFocus, nur über rudimentäre Beschreibungen von Datentypen.

LISP, Prolog Die Sprache Lisp stellt einen interessanten Hybridfall dar: Eingesetzt wird sie für Probleme, deren Lösung elegant auf der Basis von Listenstrukturen formuliert werden kann. Angewendet wird sie fast ausschließlich im Problembereich der Künstlichen Intelligenz – es ist also jeweils festzulegen, ob mit Domäne ein Anwendungsfeld oder eine bestimmte Problemstruktur unabhängig von der Anwendungsklasse gemeint ist. Prolog ist in dieser Hinsicht ähnlich. Die Stärken dieser Sprache kommen dann zur Geltung, wenn das Problem im wesentlichen ein Suchproblem ist, für das Prolog beispielsweise Backtracking mehr oder weniger transparent zur Verfügung stellt.

Entwurfsmuster Auf Entwurfsebene ist mit Entwurfsmustern (Gamma u. a., 1995) ein ähnliches Phänomen beobachtbar: Entwurfsmuster lösen oftmals Probleme der objektorientierten technologischen Infrastruktur, und sind nicht auf eine bestimmte Anwendung oder Anwendungsklasse zugeschnitten. Die Übertragung auf Prozeßmuster (Ambler, 1998) weist zumeist keine direkte fachliche Domänenabhängigkeit auf.

Im Zug der rasanten Entwicklung der Informationstechnologie mag eine Ursache für die weite Verbreitung von general-purpose Sprachen wie C, Ada oder Java und u.U. in Zukunft C# die ökonomische Problematik sein, im Fall der domänenspezifischen Sprachen Entwickler für viele verschiedene Sprachen ausbilden zu müssen. Aufgrund des natürlicherweise größeren Marktvolumens und der daraus resultierenden Stärke mancher Hersteller ergeben sich kürzere Innovationszyklen als für domänenspezifische Sprachen. Diese Entwicklung ist vergleichbar mit dem gescheiterten Ansatz, im Rahmen des Fifth Generation Project in Japan dedizierte Prozessoren für Logikprogrammierung auf der Basis von Warren's Abstract Machine zu bauen: Es stellte sich heraus, daß general-purpose-Prozessoren aufgrund ihrer schnelleren Entwicklung für die Problemklasse, für die Logikprozessoren gebaut wurden, effizienter eingesetzt werden konnten.

B. Beweise

B.1. Transitionsordnungen (Algorithmus 5.2)

Im folgenden bezeichne $|T|$ die Länge der Sequenz T und $|\pi|$ die Länge $n-1$ (Anzahl der Transitionen) des Pfades $\pi = \sigma_1 \xrightarrow{t_1} \sigma_2 \xrightarrow{t_2} \cdots \sigma_{n-1} \xrightarrow{t_{n-1}} \sigma_n$. $\rightarrow$ ist die Transitionspfeilrelation zwischen Kontrollzuständen, $\rightarrow^*$ der transitive und reflexive Abschluß. $\sigma_1 \rightarrow \sigma_2$ bedeutet, daß es einen Transitionspfeil von σ_1 nach σ_2 gibt; $\sigma_1 \xrightarrow{t} \sigma_2$ ($\sigma_1 \xrightarrow{\pi} \sigma_2$) bedeutet, daß es einen Transitionspfeil t (einen Pfad π) von σ_1 zu σ_2 gibt. $E^{(i)}$ bezeichne die Menge E am Ende des i-ten Durchlaufs der Schleife. Analog bezeichnet $T_e^{(i)}$ die Transitionssequenz T_e am Ende des i-ten Durchlaufs, und $T[j]$ ist das j-te Element der Sequenz T.

Zu zeigen ist:

1. Der Algorithmus terminiert.
2. Für jede Transition t auf einem Pfad $\sigma \xrightarrow{\pi} s$ gibt es ein T_e mit $T_e = X \circ < t > \circ Y$ für Sequenzen X, Y.
3. Die Länge der minimalen Pfade von $e \in E^{(i)} - \{s\}$ zu s ist i (Hilfsaussage), und es gilt für alle $e \in E^{(i)}$

$$\begin{aligned}&\forall j \forall k \forall \pi \forall \pi' \forall \varphi \forall \varphi' \bullet \\ &|T_e^{(i)}| \geq k > j \geq 0 \wedge e \rightarrow e' \wedge e \rightarrow e'' \wedge e' \xrightarrow{\varphi} s \wedge e'' \xrightarrow{\varphi'} s \\ &\wedge e' \xrightarrow{\pi} s \wedge e'' \xrightarrow{\pi'} s \wedge |e' \xrightarrow{\varphi} s| \geq |e' \xrightarrow{\pi} s| \wedge |e'' \xrightarrow{\varphi'} s| \geq |e'' \xrightarrow{\pi'} s| \\ &\Rightarrow |e \xrightarrow{T_e^{(i)}[k]} e' \xrightarrow{\pi} s| \geq |e \xrightarrow{T_e^{(i)}[j]} e'' \xrightarrow{\pi'} s|.\end{aligned} \tag{B.1}$$

Beweis. Die Terminierung folgt aus der Endlichkeit von S und der Menge der Transitionsordnungen sowie der Tatsache, daß die Kardinalität von E in jedem Durchlauf mit $M \neq \emptyset$ monoton wächst. Für $M = \emptyset$ terminiert der Algorithmus.

Für den Beweis der zweiten Aussage wird wird angenommen, daß t die erste Transition eines minimalen Pfades $\sigma_1 \xrightarrow{t} \sigma_2 \rightarrow^* s$ ist, aber in keinem $T_e^{(i)}$ vorkommt. Offenbar gibt es dann keinen Schleifendurchlauf mit $\sigma_2 \in M$, denn andernfalls wäre t in $T_{\sigma_1}^{(i)}$ für ein i. Dann gibt es kein t' als erste Transition auf einem Pfad von σ_2 zu s, im Widerspruch zur Minimalität.

Der Beweis der dritten Aussage erfolgt durch vollständige Induktion über i.

$i = 1$ Die Negation der Aussage wird zum Widerspruch geführt. Nach Konstruktion sind alle $e \in E^{(1)}$ direkte Vorgänger von s, also ist $e' = e'' = s$, und damit sind π, π' leere Pfade. Daraus folgt $1 < 1$, ein Widerspruch.

Darüberhinaus ist die Länge der minimalen Pfade von den Zuständen in $E^{(1)} - \{s\}$ zu s nach Konstruktion gleich 1.

$i > 1$ Es ist $T_e^{(i+1)} = T_e^{(i)} \circ < t_1, \ldots, t_n >$, und die Aussage gilt für $T_e^{(i)}$. Angenommen, es gibt ein t_p für $1 \leq p \leq n$ mit $|e \xrightarrow{t_p} e' \xrightarrow{\pi} s| < |e \xrightarrow{T_e^{(i)}[j]} e'' \xrightarrow{\pi'} s|$. Nach Konstruktion ist $e' \in E^{(i)} - \bigcup_{\ell<i} E^{(\ell)}$ und $e'' \in E^{(\ell)}$ für $\ell < i$.

- Falls $i + 1 = 2$, so ist nach Konstruktion $e'' = s$ und π nichtleer, was zum Widerspruch $2 < 1$ führt. Die Länge der minimalen Pfade ist $1 + 1 = 2$.
- Falls $i + 1 > 2$, so ergibt sich für die Länge der minimalen Pfade $e \to e'' \to^* s$ nach Induktionsvoraussetzung $\ell + 1$, und es ist $\ell < i$. Die Länge der minimalen Pfade $e \to e' \to^* s$ ist $i + 1$, und offenbar ist $i + 1 < \ell + 1$ ein Widerspruch. Die Länge der minimalen Pfade von $e \in E^{(i+1)}$ zu s ist dann $i + 1$.

□

B.2. Charakterisierung der Inklusion (5.4)

Um einen Zustand σ zu komplementieren, wird zunächst der Fall betrachtet, daß $\mathcal{C}_\sigma = \top$ ist. Offenbar besteht in dem Fall σ aus genau den Termen ϑ, für die keine Variablenbelegung existiert, die ϑ und σ identifizieren würden:

$$\overline{[\![\tilde{\sigma}]\!]} = \bigcup_{\neg unif(\tilde{\sigma},\tilde{\vartheta})} [\![\tilde{\vartheta}]\!], \tag{B.2}$$

und für $GSubst = \{\eta \in Subst : \forall(t) \bullet \mathcal{V}(\eta(t)) = \emptyset\}$ gilt

$$e \in \overline{[\![\tilde{\sigma}]\!]} \Longleftrightarrow \exists x \exists \gamma \in GSubst \bullet \neg unif(x, \tilde{\sigma}) \wedge \gamma(x) = e. \tag{B.3}$$

Eine syntaktische Charakterisierung von Gleichung 5.3 für den Fall $\mathcal{C}_\sigma = \top$ lautet wie folgt.

$$[\![\nu]\!] \cap \overline{[\![\tilde{\sigma}]\!]} \neq \emptyset \Longleftrightarrow \tilde{\sigma} \not\lesssim \tilde{\nu} \wedge \mathcal{C}_\nu. \tag{B.4}$$

Dabei wird davon ausgegangen, daß Constraints $\mathcal{C}_\nu$ so weit wie möglich instantiiert sind, d.h. falls eine Variable bzgl. $\mathcal{C}_\nu$ eindeutig belegbar ist, wird sie als instantiiert angenommen. Weiterhin wird angenommen, daß die Variablenmengen eines Zustands disjunkt zu den Variablenmengen vorheriger Zustände sind, was durch die Prolog-Implementierung sichergestellt wird, da in jedem Schritt neue Variablen angelegt werden – Variablen in Prolog können im Verlauf einer Berechnung höchstens spezialisiert werden.

Beweis von B.4. Es bezeichne $GSubst_\mathcal{C}$ die Menge der Substitutionen, die die Variablen eines Terms unter Berücksichtigung der Constraints $\mathcal{C}$ auf Grundterme abbilden:

$$GSubst_{\mathcal{C}_\sigma} = \{\eta \in Subst : \mathcal{V}(\eta(\tilde{\sigma})) = \emptyset \wedge mgu_{(\eta(\tilde{\sigma}),\tilde{\sigma})}(\mathcal{C}_\sigma)\} \tag{B.5}$$

In der Hinrichtung ist dann

$$\begin{aligned}&\exists e \exists x \exists \gamma_1 \in GSubst_\top \exists \gamma_2 \in GSubst_{\mathcal{C}_\nu} \bullet \\ &\quad \neg unif(x, \tilde{\sigma}) \wedge \gamma_1(x) = e \wedge \gamma_2(\tilde{\nu}) = e \Longrightarrow \tilde{\sigma} \not\leq \tilde{\nu} \wedge \mathcal{C}_\nu\end{aligned} \tag{B.6}$$

zu zeigen. Angenommen, $\tilde{\sigma} \leq \tilde{\nu} \vee \neg \mathcal{C}_\nu$ gilt zusammen mit der Prämisse, d.h.

$$\begin{aligned}&\exists e \exists x \exists \gamma_1 \in GSubst_\top \exists \gamma_2 \in GSubst_{\mathcal{C}_{\tilde{\nu}}} \exists \eta \in Subst \bullet \\ &\quad \neg unif(x, \tilde{\sigma}) \wedge \gamma_1(x) = e \wedge \gamma_2(\nu) = e \wedge \big(\eta(\tilde{\sigma}) = \nu \vee \neg \mathcal{C}_\nu\big).\end{aligned} \tag{B.7}$$

Da $\exists \gamma_2 \in GSubst_{\mathcal{C}_\nu} \bullet \neg \mathcal{C}_\nu \wedge \gamma_2(\nu) = e$ unerfüllbar ist, impliziert das (Quantorenpräfix bleibt bestehen) $\neg unif(x, \tilde{\sigma}) \wedge \gamma_2(\eta(\tilde{\sigma})) = \gamma_1(x)$, was wiederum $\eta(\tilde{\sigma}) \leq \gamma_1(x)$ bedeutet, und damit sind offenbar $\tilde{\sigma}$ und x wegen $\mathcal{V}(x) \cap \mathcal{V}(\tilde{\sigma}) = \emptyset$ unifizierbar, was zu einem Widerspruch führt.

Für die Rückrichtung bezeichne $\mathcal{O}(t)$ die Menge der Positionen im Term t, und $t|_p$ bezeichne für $p \in \mathcal{O}(t)$ den Subterm von t an Stelle p. Zu zeigen ist

$$\begin{aligned}&\tilde{\sigma} \not\leq \tilde{\nu} \wedge \mathcal{C}_\nu \Longrightarrow \\ &\quad \exists e \exists x \exists \gamma_1 \in GSubst_\top \exists \gamma_2 \in GSubst_{\mathcal{C}_\nu} \bullet \neg unif(x, \tilde{\sigma}) \wedge \gamma_1(x) = e \wedge \gamma_2(\tilde{\nu}) = e.\end{aligned} \tag{B.8}$$

Der Beweis erfolgt durch Fallunterscheidung über $\tilde{\sigma} \not\leq \tilde{\nu}$. Die Idee ist, eine Variable in $\tilde{\sigma}$ so zu belegen, daß $\tilde{\sigma}$ und $\tilde{\nu}$ nicht mehr unifizierbar sind. Es kann o.B.d.A. angenommen werden, daß die Kardinalität der Extension aller Typen größer ist als eins.

- Angenommen, $\tilde{\nu} < \tilde{\sigma} \wedge \mathcal{C}_\nu$. Dann $\exists p \in \mathcal{O}(\tilde{\nu}) \cap \mathcal{O}(\tilde{\sigma}) \bullet \tilde{\nu}|_p \in \mathcal{V} \wedge \tilde{\sigma}|_p \notin \mathcal{V}$. Man setzt $x = \gamma_2(\tilde{\nu})$ für $\gamma_2 \in GSubst_{\mathcal{C}_\nu}$ mit $\gamma_2 \supseteq \{\tilde{\nu}|_p \mapsto k\}$ für ein $k \neq \tilde{\sigma}|_p$, $\gamma_1 = \emptyset$ und $e = x$. Da $\mathcal{C}_{\tilde{\nu}|_p} \wedge \tilde{\nu}|_p \neq \tilde{\sigma}|_p$ nach Voraussetzung[1] erfüllbar ist, folgt die Behauptung nach Konstruktion.

- Angenommen, $\neg unif(\tilde{\sigma}, \tilde{\nu}) \wedge \mathcal{C}_\nu$. Wählt man irgendein $\gamma_2 \in GSubst_{\mathcal{C}_\nu}$, $\gamma_1 = \emptyset$ und setzt man $x = \gamma_2(\tilde{\nu}) = e$, so folgt die Behauptung nach Konstruktion.

- Angenommen, $\mathcal{C}_\nu$ gilt und $\tilde{\nu}$ und $\tilde{\sigma}$ sind bzgl. $\leq$ unvergleichbar, aber unifizierbar. Dann $\exists p \in \mathcal{O}(\tilde{\nu}) \cap \mathcal{O}(\tilde{\sigma}) \bullet \tilde{\nu}|_p \in \mathcal{V} \wedge \tilde{\sigma}|_p \notin \mathcal{V}$, und der Beweis erfolgt wie im ersten Fall.

□

Im Fall $\mathcal{C}_\sigma \neq \top$ müssen für die Komplementierung von $[\![\sigma]\!]$ zusätzlich die Terme zugelassen werden, die Instanzen von $\tilde{\sigma}$ sind, die aber nicht $\mathcal{C}_\sigma$ erfüllen. Diese Menge ist beschrieben durch

$$\overline{[\![\sigma]\!]} = \overline{[\![\tilde{\sigma}]\!]} \cup \Big([\![\tilde{\sigma}]\!] - \bigcup_{\eta \in Subst_{\mathcal{C}_\sigma}} [\![\eta(\tilde{\sigma})]\!]\Big) = \bigcap_{\eta \in Subst_{\mathcal{C}_\sigma}} \overline{[\![\eta(\tilde{\sigma})]\!]}, \tag{B.9}$$

und gemäß Gleichung B.3 ergibt sich daraus

$$e \in \overline{[\![\sigma]\!]} \Longleftrightarrow \forall \eta \in Subst_{\mathcal{C}_\sigma} \exists x \exists \gamma \in GSubst_\top \bullet \neg unif(x, \eta(\tilde{\sigma})) \wedge \gamma(x) = e \tag{B.10}$$

[1] Forderungen nach minimaler Kardinalität und Instantiierung, wenn möglich.

Eine syntaktische Charakterisierung – ein CLP-Programm – der Teilmengenbeziehung

$$\begin{aligned}&[\![\nu]\!] \cap \overline{[\![\sigma]\!]} \neq \emptyset \Longleftrightarrow \\ &\quad \exists e \forall \eta \in \mathit{Subst}_{\mathcal{C}_\sigma} \exists x \exists \gamma_1 \in \mathit{GSubst}_\top \exists \gamma_2 \in \mathit{GSubst}_{\mathcal{C}_\nu} \bullet \\ &\qquad \neg \mathit{unif}(x, \eta(\tilde{\sigma})) \wedge \gamma_1(x) = e \wedge \gamma_2(\tilde{\nu}) = e\end{aligned} \tag{B.11}$$

ist Gleichung 5.4, die jetzt bewiesen werden kann:

$$[\![\nu]\!] \cap \overline{[\![\sigma]\!]} \neq \emptyset \Longleftrightarrow \mathcal{C}_\nu \wedge \big(\tilde{\sigma} \leq \tilde{\nu} \Rightarrow \mathit{mgu}_{(\tilde{\nu},\tilde{\sigma})}(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu)\big).$$

Beweis von 5.4 In der Hinrichtung wird wiederum die Negation der Aussage zu einem Widerspruch geführt. Da erneut $\exists \gamma_2 \in \mathit{GSubst}_{\mathcal{C}_\nu} \bullet \neg \mathcal{C}_\nu \wedge \gamma_2(\tilde{\nu}) = e$ nicht erfüllbar ist, bleibt (Quantorenpräfix bleibt bestehen)

$$\neg \mathit{unif}(x, \eta(\tilde{\sigma})) \wedge \gamma_1(x) = \gamma_2(\tilde{\nu}) \wedge \tilde{\sigma} \leq \tilde{\nu} \wedge \neg \mathit{mgu}_{(\tilde{\nu},\tilde{\sigma})}(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu) \tag{B.12}$$

zu zeigen. x ist nach Definition ein Term, der nicht mit einer $\mathcal{C}_\sigma$ erfüllenden Instanz von $\tilde{\sigma}$ unifizierbar ist. Der Fall $\mathcal{C}_\sigma = \top$ wurde bereits behandelt, also sei $\mathcal{C}_\sigma \neq \top$. Der Beweis trifft eine Fallunterscheidung über das Verhältnis von x und $\tilde{\sigma}$.

- Wenn $\tilde{\sigma}$ und x nicht unifizierbar sind, dann gibt es wegen $\tilde{\sigma} \leq \tilde{\nu}$ keine gemeinsame Instanz e von x und $\tilde{\nu}$.

- Es seien also $\tilde{\sigma}$ und x unifizierbar. $\neg \mathit{mgu}_{(\tilde{\nu},\tilde{\sigma})}(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu)$ läßt sich als $\forall \zeta \in \mathit{Subst} \bullet \zeta(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu) \Rightarrow \zeta(\tilde{\nu}) \neq \zeta(\tilde{\sigma})$ schreiben. $\exists \xi \in \mathit{Subst} \bullet \xi(\tilde{\sigma}) = \tilde{\nu}$ wegen $\tilde{\sigma} \leq \tilde{\nu}$, also folgt durch Einsetzen $\forall \zeta \in \mathit{Subst} \bullet \zeta(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu) \Rightarrow \zeta(\xi(\tilde{\sigma})) \neq \zeta(\tilde{\sigma})$. Nach Voraussetzung ist $\mathcal{C}_\sigma \neq \top$ und damit wegen $\mathcal{V}(\mathcal{C}_\nu) \cap \mathcal{V}(\overline{\mathcal{C}}_\sigma) = \emptyset$ $\zeta(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu)$ erfüllbar. Die Aussage läßt sich negieren in $\exists \zeta \in \mathit{Subst} \bullet \zeta(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu) \wedge \zeta(\xi(\tilde{\sigma})) = \zeta(\tilde{\sigma})$, was offenbar gültig ist, denn Terme sind immer mit umbenannten Instanzen von sich selbst unifizierbar. Damit ist die ursprüngliche Aussage nicht gültig, und der Widerspruch ist gezeigt.

Für die Rückrichtung wird wiederum über die Beziehung von $\tilde{\nu}$ und $\tilde{\sigma}$ unterschieden.

- Der Beweis für $\mathcal{C}_\nu \wedge \tilde{\sigma} \not\leq \tilde{\nu}$ erfolgt wegen $[\![\tilde{\sigma}]\!] \supseteq [\![\sigma]\!]$, damit $\overline{[\![\tilde{\sigma}]\!]} \subseteq \overline{[\![\sigma]\!]}$ und also $[\![\nu]\!] \cap \overline{[\![\tilde{\sigma}]\!]} \neq \emptyset \Longrightarrow [\![\nu]\!] \cap \overline{[\![\sigma]\!]} \neq \emptyset$ in Analogie zum entsprechenden Fall im Beweis von Gleichung B.4.

- $\tilde{\sigma} \leq \tilde{\nu}$ und $\mathit{mgu}_{(\tilde{\nu},\tilde{\sigma})}(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu)$ gelte. $\zeta \in \mathit{Subst}_{\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu}$ mit $\zeta(\tilde{\nu}) = \zeta(\tilde{\sigma})$ existiert nach Voraussetzung. Für $x = \zeta(\tilde{\nu}) = \zeta(\tilde{\sigma})$, ein beliebiges γ_1, $\gamma_2 = \gamma_1 \circ \zeta$ und schließlich $e = \gamma_1(x) = \gamma_2(\tilde{\nu})$ gilt Gleichung B.11 nach Konstruktion.

□

B.3. Charakterisierung des Komplements (5.6)

Zu zeigen ist

$$\overline{[\![\sigma]\!]} = \begin{cases} [\![< R, R \neq^{pm} \tilde{\sigma} >]\!] \cup [\![< \tilde{\sigma}, \overline{\mathcal{C}}_\sigma >]\!], & \textit{falls } \mathcal{C}_\sigma \neq \top \\ [\![< R, R \neq^{pm} \tilde{\sigma} >]\!], & \textit{falls } \mathcal{C}_\sigma = \top. \end{cases}$$

Für den Beweis muß die Äquivalenz mit Charakterisierung B.10 nachgewiesen werden. Offenbar muß zunächst der Zusammenhang zwischen Nichtunifizierbarkeit und $\neq^{pm}$ hergestellt werden. Es gilt zwar $\neg \mathit{unif}(A, \tilde{\sigma}) \Rightarrow A \neq^{pm} \tilde{\sigma}$, aber nicht die Rückrichtung (für $A = c(X)$ mit $X \in \mathcal{V}$ und $\tilde{\sigma} = c(a)$ ergibt sich die Reduktion zu $X \neq^{pm} a$, was erfüllbar ist, aber $\neg \mathit{unif}(c(X), c(a))$ gilt nicht). Auf Grundinstanzebene läßt sich die Äquivalenz jedoch herstellen:

$$[\![< A, \neg \mathit{unif}(A, \tilde{\sigma}) >]\!] = [\![< A, A \neq^{pm} \tilde{\sigma} >]\!]. \tag{B.13}$$

Beweis. Die Richtung „$\subseteq$" ist trivial, weil $\neq^{pm}$ eine Abschwächung von $\neg \mathit{unif}$ ist. In der Richtung „$\supseteq$" gilt zunächst gemäß der Definition von $[\![\cdot]\!]$ (Gleichung 5.2) $t \in [\![< A, A \neq \tilde{\sigma} >]\!] \Rightarrow t \in [\![< t, t \neq^{pm} \tilde{\sigma} >]\!]$ für Grundterme t. Es sei also $t \in [\![< t, t \neq^{pm} \tilde{\sigma} >]\!]$. Der Beweis erfolgt durch Induktion über den Aufbau von $\tilde{\sigma}$ und t. Angenommen, t und $\tilde{\sigma}$ sind unifizierbar.

Anker Falls $\tilde{\sigma} \in \mathcal{V}$, so gilt $t \neq^{pm} \tilde{\sigma}$ nicht. Falls $\tilde{\sigma}$ ein Atom ist und

- t ein identisches Atom, so gilt $\tilde{\sigma} \neq^{pm} t$ nicht,
- $t \neq \tilde{\sigma}$ ist, so sind t und $\tilde{\sigma}$ nicht unifizierbar, im Widerspruch zur Annahme.

Schritt Es sei $\tilde{\sigma} = c(a_1, \ldots, a_n)$.

- Falls $hd(t) \neq c$, so sind $\tilde{\sigma}$ und t nicht unifizierbar, im Widerspruch zur Annahme.
- Es sei also $t = c(b_1, \ldots, b_n)$, und die Induktionsvoraussetzung gelte für alle Argumente, d.h. $a_i \neq^{pm} b_i \Rightarrow \neg \mathit{unif}(a_i, b_i)$. Dann reduziert $t \neq^{pm} \tilde{\sigma}$ nach Definition zu $\bigvee_{i=1}^{n} a_i \neq^{pm} b_i$. Aus der angenommenen Unifizierbarkeit von t und $\tilde{\sigma}$ folgt aus der Tatsache, daß Variablen nur einmal vorkommen, $\bigwedge_{i=1}^{n} \mathit{unif}(a_i, b_i)$. Daraus ergibt sich sofort der Widerspruch.

□

Damit kann die Charakterisierung 5.6 von $e \in \overline{[\![\sigma]\!]}$ für $\mathcal{C}_\sigma \neq \top$,

$$\begin{aligned} &\forall \eta \in \mathit{Subst}_{\mathcal{C}_\sigma} \exists x \exists \gamma \in \mathit{GSubst}_\top \bullet \neg \mathit{unif}(x, \eta(\tilde{\sigma})) \wedge \gamma(x) = e \\ &\quad \Longleftrightarrow e \in [\![< R, R \neq^{pm} \tilde{\sigma} >]\!] \cup [\![< \tilde{\sigma}, \overline{\mathcal{C}}_\sigma >]\!] \end{aligned}$$

und im Fall $\mathcal{C}_\sigma = \top$,

$$\begin{aligned} &\forall \eta \in \mathit{Subst}_{\mathcal{C}_\sigma} \exists x \exists \gamma \in \mathit{GSubst}_\top \bullet \neg \mathit{unif}(x, \eta(\tilde{\sigma})) \wedge \gamma(x) = e \\ &\quad \Longleftrightarrow e \in [\![< R, R \neq^{pm} \tilde{\sigma} >]\!] \end{aligned}$$

bewiesen werden:

Beweis von 5.6. Zunächst gelte $\mathcal{C}_\sigma \neq \top$. In der Hinrichtung gilt, daß e Grundinstanz eines Terms x ist, der mit keiner $\mathcal{C}_\sigma$ erfüllenden Instanz von $\tilde{\sigma}$ unifizierbar ist. Dann ist x (und damit e) nicht mit $\tilde{\sigma}$ unifizierbar, oder x (und damit e) ist mit $\tilde{\sigma}$ unifizierbar, aber nicht bzgl. $\mathcal{C}_\sigma$. Gilt $\neg unif(e, \tilde{\sigma})$, so gilt auch $e \neq^{pm} \tilde{\sigma}$. Gilt $unif(x, \tilde{\sigma})$, aber $\forall \eta \in Subst_{C_\sigma} \bullet \neg unif(x, \eta(\tilde{\sigma}))$, und ist e Grundinstanz von x, dann erfüllt e $\overline{\mathcal{C}_\sigma}$, d.h. $mgu_{(e,\tilde{\sigma})}(\overline{\mathcal{C}_\sigma})$ gilt, und damit ist $e \in [\![< \tilde{\sigma}, \overline{\mathcal{C}}_\sigma >]\!]$.

In der Rückrichtung sei zunächst $e \in [\![< R, R \neq^{pm} \tilde{\sigma} >]\!]$. Aus Gleichung B.13 folgt $e \in [\![< A, \neg unif(A, \tilde{\sigma}) >]\!]$, und die Definition von $[\![\cdot]\!]$ liefert $\mathcal{V}(e) = \emptyset \wedge unif(e, A) \wedge mgu_{(e,A)}(\neg unif(A, \tilde{\sigma}))$ und damit $\neg unif(e, \tilde{\sigma})$. Setzt man $x = e$ und $\gamma = \emptyset$, so folgt auch $\forall \eta \in Subst_{\mathcal{C}_\sigma} \bullet \neg unif(e, \eta(\tilde{\sigma}))$. Für den zweiten Fall, $e \in [\![< \tilde{\sigma}, \overline{\mathcal{C}}_\sigma >]\!]$, setzt man ebenso $x = e$ und $\gamma = \emptyset$, und damit ist nach Konstruktion e mit keiner $\mathcal{C}_\sigma$ erfüllenden Instanz von $\tilde{\sigma}$ unifizierbar.

Schließlich gelte $\mathcal{C}_\sigma = \top$. Die Hinrichtung ist ein Spezialfall des angegebenen Beweises: Der Fall $unif(x, \tilde{\sigma})$ und $\forall \eta \in Subst_{C_\sigma} \bullet \neg unif(x, \eta(\tilde{\sigma}))$ kann nicht auftreten. Die Rückrichtung ist bereits Teil des obigen Beweises.

□

B.4. Vollständigkeit der Zustandsspeicherung (5.10)

Zu zeigen ist

- $\forall \vartheta \exists \sigma \bullet (\vartheta \in \Theta \wedge \sigma \in \Sigma) \Rightarrow (x \in [\![\vartheta]\!] \Rightarrow x \in [\![\sigma]\!])$ und
- $\forall \sigma \exists \vartheta \bullet (\sigma \in \Sigma \wedge \vartheta \in \Theta) \Rightarrow (x \in [\![\sigma]\!] \Rightarrow x \in [\![\vartheta]\!])$.

Die Korrektheitsaussage gilt trivialerweise, weil Zustandsspeicherung die Menge der besuchten Zustände höchstens einschränkt. Im Fall der Vollständigkeit sei $x' \in [\![\sigma']\!]$ für $\sigma' \in \Sigma$ ein Zustand, für den kein $\vartheta \in \Theta$ mit $x' \in [\![\vartheta]\!]$ existiert. Dann gibt es σ'' mit $x' \in [\![\sigma'']\!]$, so daß der Pfad $\sigma_0, \ldots, \sigma_n$ bzgl. $post[T]$ vom Initialzustand zu $\sigma_n = \sigma''$ minimale Länge besitzt (wenn es mehrere kürzeste Pfade gibt, wird einer ausgewählt). Offenbar gibt es nun ein minimales j, so daß $\exists x'' \in [\![\sigma_j]\!] \forall \vartheta \in \Theta \bullet x'' \notin [\![\vartheta]\!]$ und $\forall i < j \exists \vartheta \in \Theta \bullet s \in [\![\sigma_i]\!] \Rightarrow s \in [\![\vartheta]\!]$ gilt. Weil der Initialzustand mit Sicherheit in Σ und Θ liegt, ist $j > 0$. Das bedeutet, daß es eine Menge von Pfaden $\vartheta_{k1}, \ldots, \vartheta_{km_k}$ bzgl. $post'[T]$ mit $\forall i < j \exists k \exists \ell \bullet s \in [\![\sigma_i]\!] \Rightarrow s \in [\![\vartheta_{k\ell}]\!]$ gibt. $\mathcal{J}$ bezeichne die Indexmenge aller Paare (k, ℓ), die die Gleichung erfüllen. Insbesondere gilt dann $\forall s \exists (k', \ell') \in \mathcal{J} \bullet s \in [\![\sigma_{j-1}]\!] \Rightarrow s \in [\![\vartheta_{k'\ell'}]\!]$. Wenn nun $x'' \in [\![\sigma_j]\!]$ ist, aber $x'' \notin \bigcup_{(k,\ell)\in\mathcal{J}} [\![post'[T]\vartheta_{k\ell}]\!]$, dann muß es nach Konstruktion von $post'[T]$ ein $t \in \Theta$ mit $x'' \in [\![t]\!]$ geben, denn andernfalls wäre $x'' \in \bigcup_{(k,\ell)\in\mathcal{J}} [\![post'[T]\vartheta_{k\ell}]\!]$. Das liefert einen Widerspruch zur Annahme $\neg \exists \vartheta \in \Theta \bullet x'' \in [\![\vartheta]\!]$.

□

Abbildungsverzeichnis

2.1. Systemstrukturdiagramm der WIM 16
2.2. Systemstrukturdiagramm für kryptographische Funktionen . . . 18
2.3. Modellbasiertes Testen 27
2.4. Modelle, Umwelt und Testfälle 30
2.5. Modellierung und Test im inkrementellen Entwurf 56
2.6. Code und Testfälle aus Modell generiert 58
2.7. Modell entsteht nach Code 59
2.8. Modell und Code entstehen unabhängig 60
2.9. Modelle für Entwicklung und Testfallgenerierung 61

3.1. Digitale Signaturberechnung 75
3.2. Karte und Testmodell 76

4.1. Erweiterte Zustandsmaschine für Security Environment 101
4.2. Funktionsdefinition 102
4.3. Systemstrukturdiagramm der WIM 103
4.4. Systemstrukturdiagramm für Cardholder Verification 103
4.5. Resolution 108
4.6. Partielle Übersetzung von `SecurityOperations` 111
4.7. Idle-Transitionen 113
4.8. Übersetzung: Idle-Transitionen 115
4.9. Funktionsdefinitionen 116
4.10. Übersetzung von Pattern Matching 118
4.11. Übersetzung von Funktionen 119
4.12. Beispielsystem für Optimierung von Idle-Transitionen 121
4.13. Beispielsystem für symbolische Ausführung 122
4.14. Beispielsystem für kompositionale Testfallerzeugung 127
4.15. Generierung fehlender Signale 130
4.16. MC/DC-Testfälle 133
4.17. Komponente `WIMPre` 139
4.18. Trace für digitale Signaturberechnung 141

5.1. Suchstrategien 150
5.2. Algorithmus für statische Transitionsordnungen 154
5.3. Zustandsinklusion und nichtleerer Durchschnitt 161
5.4. Konvexe Hülle von (a,b,c) und (a,c,b) 167
5.5. Zustandsmaschine; Semantik abhängig von $[\![\cdot]\!]$ 168
5.6. Inhouse-Karte 173

6.1. Test nichtdeterministischer Systeme 198
6.2. Test kontinuierlich-diskreter Systeme 200

Tabellenverzeichnis

5.1. Beispiele für Distanzmaße 153
5.2. Suchverfahren für $cnt_4 \rightarrow 0$ 175
5.3. Suchverfahren für $cnt_6 \rightarrow 0$ 176
5.4. Suche in der WIM (Zähler) 178
5.5. Suche in der WIM (digitale Signatur) 179
5.6. Testfälle für die WIM 181
5.7. MC/DC Testfälle für einzelne Komponenten 183

Literaturverzeichnis

[3GPP 2001] 3GPP: *3rd Generation Partnership Project: Technical Specification Group Terminals, Specification of the Subscriber Identity Module—Mobile Equipment (SIM-ME) interface.* 2001. – V 8.5.0; Release 1999

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

[Alpern und Schneider 1985] Alpern, B. ; Schneider, F.: Defining Liveness. In: *Information Processing Letters* 21 (1985), S. 181–185

[Alur u. a. 1995] Alur, R. ; Courcoubetis, C. ; Halbwachs, N. ; Henzinger, T.A. ; Ho, P.-H. ; Nicollin, X. ; Olivero, A. ; Sifakis, J. ; Yovine, S.: The algorithmic analysis of hybrid systems. In: *Theoretical Computer Science* 138 (1995), Nr. 1, S. 3–34

[Ambler 1998] Ambler, S.: *Process Patterns: Building Large-Scale Systems Using Object Technology.* Cambridge University Press, 1998

[Ambler 2002] Ambler, S.: *Agile Modeling.* J. Wiley and Sons, 2002

[Ammann und Black 1999] Ammann, P. ; Black, P.: Abstracting Formal Specifications to Generate Software Tests via Model Checking. In: *Proc. 18th Digital Avionics Systems Conference (DASC'99)* Bd. 2. St. Louis, MO : IEEE, October 1999, S. 10.A.6.1–10

[Ammann u. a. 1998] Ammann, P. ; Black, P. ; Majurski, W.: Using model checking to generate tests from specifications. In: *Proc. 2nd IEEE Intl. Conf. on Formal Engineering Methods*, 1998, S. 46–54

[ANSI/IEEE 1983] ANSI/IEEE: *Standard Glossary of Software Engineering Terminology.* 1983

[Antoy u. a. 1994] Antoy, S. ; Echahed, R. ; Hanus, M.: A Needed Narrowing Strategy. In: *Proc. 21st ACM Symp. on Principles of Programming Languages (POPL'94)*, 1994, S. 268–279

[Apt und Bol 1994] Apt, K. ; Bol, R.: Logic Programming and Negation: A Survey. In: *J. Logic Programming* 19/20 (1994), S. 9–71

[Back und Wright 1998] Back, R.-J. ; Wright, J. von: *Refinement Calculus: A Systematic Introduction.* Springer, 1998

[Baker u. a. 2001] BAKER, P. ; RUDOLPH, E. ; SCHIEFERDECKER, I.: Graphical Test Specification—The Graphival Format of TTCN-3. In: *Proc. 10th SDL Forum*, 2001

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

[Barnes 1983] BARNES, J.: *Programmieren in ADA*. Carl Hanser Verlag, 1983

[Barnes 1997] BARNES, J.: *High Integrity Ada: The Spark Approach*. Addison Wesley, 1997

[Bauer und Wössner 1972] BAUER, F. ; WÖSSNER, H.: The “Plankalkül” of Konrad Zuse: A Forerunner of Today’s Programming Languages. In: *Communications of the ACM* 15 (1972), July, Nr. 7, S. 678–685

[Beck 1999] BECK, K.: *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999

[Beck 2003] BECK, K.: *Test-Driven Development By Example*. Addison Wesley, 2003

[Beizer 1983] BEIZER, B.: *Software Testing Techniques*. Van Nostrand Reinhold Company, 1983

[Bender u. a. 2002] BENDER, K. ; BROY, M. ; PÉTER, I. ; PRETSCHNER, A. ; STAUNER, T.: Model based development of hybrid systems: specification, simulation, test case generation. In: *Modelling, Analysis and Design of Hybrid Systems*, 2002 (Springer LNCIS 279), S. 37–52

[Bender und Kaiser 1995] BENDER, K. ; KAISER, O.: Simultaneous Engineering durch Maschinenemulation. In: *CIM Management* 11 (1995), Nr. 4, S. 14–18

[Berry und Gonthier 1992] BERRY, G. ; GONTHIER, G.: The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. In: *Science of Computer Programming* 19 (1992), S. 87–152

[Biere u. a. 1999] BIERE, A. ; CIMATTI, A. ; CLARKE, E. ; ZHU, Y.: Symbolic Model Checking without BDDs. In: *Proc. TACAS/ETAPS’99*, 1999 (Springer LNAI 1249), S. 193–207

[Binder 1999] BINDER, R.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 1999

[Blotz u. a. 2002] BLOTZ, A. ; HUBER, F. ; LÖTZBEYER, H. ; PRETSCHNER, A. ; SLOTOSCH, O. ; ZÄNGERL, H.-P.: Model-Based Software Engineering and Ada: Synergy for the Development of Safety-Critical Systems. In: *Proc. Ada Deutschland Tagung*, March 2002

[Bourhfir u. a. 1996] BOURHFIR, C. ; DSSOULI, R. ; ABOULHAMID, E.: Automatic test generation for EFSM-based systems / University of Montreal. August 1996 (IRO 1043). – Forschungsbericht

[Braun u. a. 2003] Braun, P. ; Broy, M. ; Cengarle, M. ; Philipps, J. ; Prenninger, W. ; Pretschner, A. ; Rappl, M. ; Sandner, R.: The Automotive CASE. In: *Modelle, Werkzeuge, Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, Wiley-VCH, 2003. – To appear

[Breitling 2001] Breitling, M.: *Formale Fehlermodellierung für verteilte reaktive Systeme*, Technische Universität München, Dissertation, 2001

[Brinksma 1988] Brinksma, E.: A theory for the derivation of tests. In: *Proc. 8th Intl. Conf. on Protocol Specification, Testing, and Verification*, 1988, S. 63–74

[Brockhaus 1989] Brockhaus: *Lexikon*. Deutscher Taschenbuch Verlag, 1989

[Brooks 1986] Brooks, F.: No Silver Bullet. In: *Proc. 10th IFIP World Computing Conference*, 1986, S. 1069–1076

[Broy 1993] Broy, M.: Interaction Refinement – The Easy Way. In: Broy, M. (Hrsg.): *Program Design Calculi* Bd. 118, Springer, 1993

[Broy und Stølen 2001] Broy, M. ; Stølen, K.: *Specification and Development of Interactive Systems – Focus on Streams, Interfaces, and Refinement.* Springer, 2001

[Budkowski u. a. 1988] Budkowski, S. ; Dembinski, P. ; Diaz, M.: *ISO Standardized Description technique ESTELLE.* 1988. – URL `www-lor.int-evry.fr/idemcop/uk/est-lang/download`

[Bultan 1998] Bultan, T.: *Automated symbolic analysis of reactive systems*, University of Maryland, Dissertation, 1998

[Burton u. a. 2001] Burton, S. ; Clark, J. ; McDermid, J.: Automatic generation of tests from Statechart specifications. In: *Proc. Formal Approaches to Testing of Software*, 2001, S. 31–46

[Buwalda und Kasdorp 1999] Buwalda, H. ; Kasdorp, M.: Getting Automated Testing Under Control. In: *Software Testing& Quality Engineering* (1999), November/December, S. 39–44

[Caspi u. a. 1987] Caspi, P. ; Pilaud, D. ; Halbwachs, N. ; Plaice, J.: LUSTRE: A Declarative Language for Programming Synchronous Systems. In: *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, 1987, S. 178–188

[Chang u. a. 1991] Chang, E. ; Manna, Z. ; Pnueli, A.: The Safety-Progress Classification. In: *Logic and Algebra of Specifications*, Springer, 1991 (NATO Advanced Science Institutes Series), S. 143–202

[Chow 1978] Chow, T.: Testing Software Design Modeled by Finite-State Machines. In: *IEEE Transactions on Software Engineering* SE-4 (1978), May, Nr. 3, S. 178–187

[Ciarlini und Frühwirth 1999a] CIARLINI, A. ; FRÜHWIRTH, T.: *Symbolic Execution for the Derivation of Meaningful Properties of Hybrid Systems.* December 1999. – Poster

[Ciarlini und Frühwirth 1999b] CIARLINI, A. ; FRÜHWIRTH, T.: Using Constraint Logic Programming for Software Validation. In: *5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation*, March 1999

[Ciarlini und Frühwirth 2000] CIARLINI, A. ; FRÜHWIRTH, T.: Automatic derivation of meaningful experiments for hybrid systems. In: *Proc. ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*, March 2000

[Claessen und Hughes 2000] CLAESSEN, K. ; HUGHES, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: *Proc. Intl. Conf. on Functional Programming*, September 2000

[Clarke u. a. 1999] CLARKE, E. ; GRUMBERG, O. ; PELED, D.: *Model Checking.* MIT Press, 1999

[Clarke 1976] CLARKE, L.: A System to Generate Test Data and Symbolically Execute Programs. In: *IEEE Transactions on Software Engineering* SE-2 (1976), September, Nr. 3, S. 215–222

[Cousot und Cousot 1977] COUSOT, P. ; COUSOT, R.: Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proc. 4th ACM symp. on Principles of Programming Languages (POPL'77)*, 1977, S. 238–252

[Coyne 1995] COYNE, R.: *Designing Information Technology in the Postmodern Age: From Method to Metaphor.* MIT Press, 1995

[Cui u. a. 1998] CUI, B. ; DONG, Y. ; DU, X. ; NARAYAN KUMAR, K. ; RAMAKRISHNAN, C. ; RAMAKRISHNAN, I. ; ROYCHOUDHURY, A. ; SMOLKA, S. ; WARREN, D.: Logic programming and model checking, 1998 (Springer LNCS 1490), S. 1–20

[Dahl und Nygaard 1967] DAHL, O.-J. ; NYGAARD, K.: Class and Subclass Declarations. In: *Simulation Programming Languages* (1967), S. 158–174

[Damm und Harel 1999] DAMM, W. ; HAREL, D.: LSC's: Breathing Life into Message Sequence Charts. In: *Proc. 3rd Intl. Conf. on Formal Methods for open object-based distributed systems (FMOODS'99)*, 1999

[Dams u. a. 1997] DAMS, D. ; GRUMBERG, O. ; GERTH, R.: Abstract Interpretation of Reactive Systems. In: *ACM Transactions on Programming Languages and Systems* 19 (1997), Nr. 2, S. 22–43

[de Alfaro und Henzinger 2001] DE ALFARO, L. ; HENZINGER, T.: Interface Automata. In: *Proc. 9th Annual ACM Symp. on Foundations of Software Engineering*, 2001

[Delzanno und Podelski 1999] Delzanno, G. ; Podelski, A.: Model Checking in CLP. In: *Proc. Tools and Algorithms for Construction and Analysis of Systems*, 1999, S. 223–239

[DeMarco 1978] DeMarco, T.: *Structured Analysis and System Specification.* Yourdon Inc., 1978

[DeMillo u. a. 1977] DeMillo, R. ; Lipton, R. ; Perlis, A.: Social Processes and Proofs of Theorems and Programs. In: *Proc. 4th ACM SIGACT-SIGPLAN Symp. of Programming Languages*, 1977, S. 206–214

[Denney 1991] Denney, R.: Test-Case Generation from Prolog-based Specifications. In: *IEEE Software* 8 (1991), March/April, Nr. 2, S. 49–57

[Dijkstra 1968] Dijkstra, E.: Go To Statement Considered Harmful. In: *Communications of the ACM* 11 (1968), Nr. 2, S. 147–148

[Dijkstra 1970] Dijkstra, E.: Notes on Structured Programming / TU Eindhoven, Department of Mathematics. 1970 (70-WSK-03). – Forschungsbericht. EWD 249

[Dijkstra 1976] Dijkstra, E.: *A Discipline of Programming.* Prentice Hall, 1976

[Du u. a. 2000] Du, X. ; Ramakrishnan, C. ; Smolka, S.: Tabled Resolution + Constraints: A Recipe for Model Checking Real-Time Systems. In: *Proc. IEEE Real-Time Systems Symposium*, 2000

[du Bousquet u. a. 2000] du Bousquet, L. ; Ouabdesselam, F. ; Parissis, I. ; Richier, J.-L. ; Zuanon, N.: Specification-based Testing of Synchronous Software. In: *Proc. 5th Intl. Workshop on Formal Methods for Industrial Critical Systems*, April 2000

[Dupuy und Leveson 2000] Dupuy, A. ; Leveson, N.: An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software. In: *Proc. Digital Aviation Systems Conf.*, 2000

[Duran und Ntafos 1984] Duran, J. ; Ntafos, S.: An Evaluation of Random Testing. In: *IEEE Transactions on Software Engineering* SE-10 (1984), July, Nr. 4, S. 438–444

[Dushina u. a. 2001] Dushina, J. ; Benjamin, M. ; Geist, D.: Semi-Formal Test Generation with Genevieve. In: *Proc. DAC*, 2001

[Dwyer u. a. 1998] Dwyer, M. ; Avrunin, G. ; Corbett, J.: Property Specification Patterns for Finite-state Verification. In: *Proc. 2nd Workshop on Formal Methods in Software Practice*, 1998

[Edelkamp u. a. 2001] Edelkamp, S. ; Lluch-Lafuente, A. ; Leue, S.: Directed Explicit Model Checking with HSF-SPIN. In: *8th Intl. SPIN Workshop on Model Checking Software*, 2001

[Eipper 1986] EIPPER, P.: *COBOL – Handbuch für Microcomputer*. Markt&Technik, 1986. – 3. Auflage

[Engler u. a. 2001] ENGLER, D. ; CHEN, D. ; HALLEM, S. ; CHOU, A. ; CHELF, B.: Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In: *Proc. 18th ACM Symp. on Operating Systems Principles*, 2001

[Ernst u. a. 2001] ERNST, M. ; COCKRELL, J. ; GRISWOLD, W. ; NOTKIN, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. In: *IEEE Transactions on Software Engineering* 27 (2001), February, Nr. 2, S. 99–123

[Fagan 1976] FAGAN, M.: Design and Code Inspections to Reduce Errors in Program Development. In: *IBM Systems Journal* 15 (1976), Nr. 3, S. 182–211

[Fagan 2002] FAGAN, M.: Reviews and Inspections. In: *Software Pioneers–Contributions to Software Engineering*, Springer Verlag, 2002, S. 562–573

[Farchi u. a. 2002] FARCHI, E. ; HARTMAN, A. ; PINTER, S.: Using a model-based test generator to test for standard conformance. In: *IBM Systems Journal* 41 (2002), Nr. 1, S. 89–110

[Fernandez u. a. 1996] FERNANDEZ, J.-C. ; JARD, C. ; JÉRON, T. ; VIHO, C.: Using on-the-fly verification techniques for the generation of test suites. In: *Proc. 8th Intl. Conf. on Computer-Aided Verification*, July 1996

[Fetzer 1988] FETZER, J.: Program Verification: The Very Idea. In: *Communications of the ACM* 37 (1988), September, Nr. 9, S. 1048–1063

[Fournier u. a. 1999] FOURNIER, L. ; KOYFMAN, A. ; LEVINGER, M.: Developing an Architecture Validation Suite—Application to the PowerPC Architecture. In: *Proc. 36th ACM Design Automation Conf.*, 1999, S. 189–194

[Fowler 1999] FOWLER, M.: *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999

[Frankl und Weyuker 1988] FRANKL, P. ; WEYUKER, E.: An Applicable Family of Data Flow Testing Criteria. In: *IEEE Transactions on Software Engineering* 14 (1988), October, Nr. 10, S. 1483–1498

[Fribourg 1999] FRIBOURG, L.: Constraint logic programming applied to model checking. In: *Proc. 9th Intl. Workshop on Logic-based Program Synthesis and Transformation*, 1999 (Springer LNCS 1817)

[Frühwirth 1995] FRÜHWIRTH, T.: Constraint Handling Rules. In: *Constraint Programming: Basics and Trends*, 1995 (Springer LNCS 910), S. 90–107

[Frühwirth 1998] FRÜHWIRTH, T.: Theory and Practice of Constraint Handling Rules. In: *J. Logic Prog.* 37(1-3) (1998), October, S. 95–138

[Frühwirth und Abdennadher 1997] Frühwirth, T. ; Abdennadher, S.: *Constraint–Programmierung.* Springer, 1997

[Fujiwara u. a. 1991] Fujiwara, S. ; v. Bochmann, G. ; Khendek, F. ; Amalou, M. ; Ghedamsi, A.: Test Selection Based on Finite State Models. In: *IEEE Transactions on Software Engineering* 17 (1991), June, Nr. 6, S. 591–603

[Gabbrielli u. a. 1995] Gabbrielli, M. ; Dore, G. ; Levi, G.: Observable Semantics for Constraint Logic Programs. In: *J. of Logic and Computation* 2 (1995), Nr. 5, S. 133–171

[Gamma u. a. 1995] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995

[Geist u. a. 1996] Geist, D. ; Farkas, M. ; Landver, A. ; Lichtenstein, Y. ; Ur, S. ; Wolfstahl, Y.: Coverage-Directed Test Generation Using Symbolic Techniques. In: *1st Intl. Conf. on Formal Methods in Computer-Aided Design* Bd. 1166, 1996, S. 143–158

[Glass 1999] Glass, R.: COBOL: Is it Dying—or Thriving. In: *The DATA BASE for Advances in Information Systems* 30 (1999), Nr. 1, S. 15–18

[Godefroid und Wolper 1994] Godefroid, P. ; Wolper, P.: A Partial Approach to Model Checking. In: *Information and Computation* 110 (1994), S. 305–326

[Govindaraju 2000] Govindaraju, G.: *Approximate Symbolic Model Checking using overlapping Projections*, Stanford University, Dissertation, 2000

[Grabowski 2000] Grabowski, J.: TTCN-3 - A new Test Specification Language for Black-Box Testing of Distributed Systems. In: *Proc. 17th Intl. Conf. and Exposition on Testing Computer Software (TCS'2000)*, 2000

[Graf und Saïdi 1997] Graf, S. ; Saïdi, H.: Construction of abstract state graphs with PVS. In: *Proc. 9th Conf. on Computer-Aided Verification*, 1997, S. 72–83

[Groce und Visser 2002] Groce, A. ; Visser, W.: Model Checking Java Programs using Structural Heuristics. In: *Proc. Intl. Symp. on Software Testing and Analysis*, 2002, S. 12–21

[Grochtmann und Grimm 1993] Grochtmann, M. ; Grimm, K.: Classification trees for partition testing. In: *Software Testing, Verification, and Reliability* 3 (1993), S. 63–82

[Gupta u. a. 1997] Gupta, V. ; Henzinger, T.A. ; Jagadeesan, R.: Robust timed automata. In: *Proc. Hybrid and Real-Time Systems.* 1997 (Springer LNCS 1201), S. 331–345

[Hahn u. a. 2003a] HAHN, G. ; PHILIPPS, J. ; PRETSCHNER, A. ; STAUNER, T.: *Prototype-based Tests for Hybrid Reactive Systems.* 2003. – To appear in Proc. Rapid System Prototyping

[Hahn u. a. 2003b] HAHN, G. ; PHILIPPS, J. ; PRETSCHNER, A. ; STAUNER, T.: Tests for mixed discrete-continuous reactive systems / Technische Universität München. 2003 (TUM-I0301). – Forschungsbericht

[Hamlet und Taylor 1990] HAMLET, D. ; TAYLOR, R.: Partition Test Does Not Inspire Confidence. In: *IEEE Transactions on Software Engineering* 16 (1990), Dezember, Nr. 12, S. 1402–1411

[Hanus 2000] HANUS, M.: *Curry: An Integrated Functional Logic Language.* 2000. – http://www.informatik.uni-kiel.de/~curry/report.html

[Harel 1988] HAREL, D.: On Visual Formalisms. In: *Communications of the ACM* 31 (1988), May, Nr. 5, S. 514–530

[Harrold und Soffa 1991] HARROLD, M. ; SOFFA, M.: Selecting and Using Data for Integration Testing. In: *IEEE Software* 8 (1991), March/April, Nr. 2, S. 58–65

[Hartmann 2001] HARTMANN, N.: *Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik*, Universität Karlsruhe, Dissertation, 2001

[Hatley und Pirbhai 1987] HATLEY, D. ; PIRBHAI, I.: *Strategies for Real-Time System Specification.* Dorset House, 1987

[Hatton 1995] HATTON, L.: *Safer C: Developing Software for High-integrity and Safety-critical Systems.* McGraw-Hill International Ltd., 1995

[Helke u. a. 1997] HELKE, S. ; NEUSTUPNY, T. ; SANTEN, T.: Automating Test Case Generation from Z Specifications with Isabelle. In: *Proc. 10th Intl. Conf. of Z Users*, 1997, S. 52–71. – Springer LNCS 1212

[Henzinger u. a. 2001] HENZINGER, T. ; HOROWITZ, B. ; KIRSCH, C. M.: Giotto: A Time-Triggered Language for Embedded Programming. In: *Proceedings of EMSOFT 2001, LNCS 2211*, 2001

[Höhfeld und Smolka 1988] HÖHFELD, M. ; SMOLKA, G.: Definite Relations over Constraint Languages / IWBS, IBM Stuttgart. October 1988 (LILOG Report 53). – Forschungsbericht

[Holzmann 1997] HOLZMANN, G.: The Spin Model Checker. In: *IEEE Trans. on Software Engineering* 23 (1997), May, Nr. 5, S. 279–295

[Holzmann 2001] HOLZMANN, G.: From Code to Models. In: *Proc. 2nd Intl. Conf. on Applications of Concurrency to System Design*, 2001, S. 3–10

[Hong u. a. 2001] HONG, H. ; LEE, I. ; SOKOLSKY, O. ; CHA, S.: Automatic Test Generation from Statecharts Using Model Checking. In: *Proc. Formal Approaches to Testing of Software*, August 2001, S. 15–30

[Howden 1977] HOWDEN, W.: Symbolic Testing and the DISSECT Symbolic Evaluation System. In: *IEEE Transactions on Software Engineering* SE-3 (1977), July, Nr. 4, S. 266–278

[Howden 1978a] HOWDEN, W.: An Evaluation of the Effectiveness of Symbolic Testing. In: *Software—Practice and Experience* 8 (1978), S. 381–397

[Howden 1978b] HOWDEN, W.: Empirical Studies of Software Validation. In: *Software Testing & Validation Techniques*, 1978, S. 280–285

[Huber u. a. 2001] HUBER, F. ; LÖTZBEYER, H. ; PRETSCHNER, A. ; SLOTOSCH, O. ; STAUNER, T. ; ZÄNGERL, H.-P.: *MOBASIS – Ergebnisse der Analysephase*. 2001

[Huber u. a. 1997] HUBER, F. ; SCHÄTZ, B. ; EINERT, G.: Consistent Graphical Specification of Distributed Systems. In: *Proc. 4th Intl. Symp. of Formal Methods Europe (FME'97)* Bd. 1313, Springer Verlag, 1997, S. 122 – 141

[Jackson 1976] JACKSON, M.: Constructive Methods of Program Design. In: *Proc. 1st Conf. of the European Cooperation in Informatics*, 1976, S. 236–262. – Springer LNCS 44

[Jackson 2001] JACKSON, M.: *Problem Frames: Analyzing and structuring software development problems*. Addison Wesley, 2001

[Jaffar und Lassez 1987] JAFFAR, J. ; LASSEZ, J.-L.: Constraint Logic Programming. In: *Proc. 14th. ACM Symposium on Principles of Programming Languages (POPL'87)*, 1987, S. 111–119

[Jaffar und Maher 1994] JAFFAR, J. ; MAHER, M.J.: Constraint Logic Programming: A Survey. In: *J. Logic Programming* 20 (1994), S. 503–581

[Jones und Harrold 2001] JONES, J. ; HARROLD, M.: Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. In: *Proc. IEEE Intl. Conf. on Software Maintenance*, 2001, S. 92–101

[Jürjens und Wimmel 2001] JÜRJENS, J. ; WIMMEL, G.: Specification-Based Testing of Firewalls. In: *Proc. Andrei Ershov 4th Intl. Conf. on Perspectives of System Informatics*, 2001

[Keutzer u. a. 2000] KEUTZER, K. ; MALIK, S. ; NEWTON, R. ; RABAEY, J. ; SANGIOVANNI-VINCENTELLI, A.: System Level Design: Orthogonalization of Concerns and Platform-Based Design. In: *IEEE Transactions on Computer-Aided Design of Circuits and Systems* 19 (2000), December, Nr. 12

[Kiczales u. a. 1997] KICZALES, G. ; LAMPING, J. ; MENDHEKAR, A. ; MAEDA, C. ; VIDEIRA LOPES, C. ; LOINGTIER, J. ; IRWIN, J.: Aspect-Oriented Programming. In: *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, 1997 (Springer LNCS 1241)

[King 1976] KING, J.: Symbolic Execution and Program Testing. In: *Communications of the ACM* 19 (1976), July, Nr. 7, S. 385–394

[Koch u. a. 1998] Koch, B. ; Grabowski, J. ; Hogrefe, D. ; Schmitt, M.: AutoLink—A Tool for Automatic Test Generation from SDL Specifications. In: *Proc. IEEE Intl. Workshop on Industrial Strength Formal Specification Techniques*, October 1998

[Koomen und Pol 1999] Koomen, T. ; Pol, M.: *Test Process Improvement – A practical step-by-step guide to structured testing.* Addison Wesley, 1999

[Kopetz und Bauer 2001] Kopetz, H. ; Bauer, G.: The time-triggered architecture / Institut für Technische Informatik, Technische Universität Wien. 2001 (11/2001). – Forschungsbericht

[Koppol u. a. 2002] Koppol, P. ; Carver, R. ; Tai, K.-C.: Incremental Integration Testing of Concurrent Porgrams. In: *IEEE Transactions on Software Engineering* 28 (2002), June, Nr. 6, S. 607–623

[Krüger 2000] Krüger, I.: *Distributed Systems Design with Message Sequence Charts*, Technische Universität München, Dissertation, 2000

[Krüger u. a. 1999] Krüger, I. ; Grosu, R. ; Scholz, P. ; Broy, M.: From MSCs to Statecharts. In: *Proc. Distributed and Parallel Embedded Systems*, 1999, S. 61–71

[Legeard und Peureux 2001] Legeard, B. ; Peureux, F.: Génération de séquences de tests à partir d'une spécification B en PLC ensembliste. In: *Proc. Approches Formelles dans l'Assistance au Développement de Logiciels*, June 2001, S. 113–130

[Lem 1964] Lem, S.: *Summa technologiae.* Suhrkamp, 1964. – Deutsche Ausgabe von 1981

[Lie u. a. 2001] Lie, D. ; Chou, A. ; Engler, D. ; Dill, D.: A Simple Method for Extracting Models from Protocol Code. In: *Proc. 28th Annual Intl. Symp. on Computer Architecture*, 2001, S. 192–203

[Lieberman 2001] Lieberman, H.: *Your Wish is my Command—Programming By Example.* Morgan Kaufman Publishers, 2001

[Lloyd 1993] Lloyd, J.: *Foundations of Logic Programming.* Springer Verlag, 1993

[Loiseaux u. a. 1995] Loiseaux, C. ; Graf, S. ; Sifakis, J. ; Bouajjani, A. ; Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. In: *Formal Methods in System Design* 6 (1995), Nr. 1, S. 11–44

[Lötzbeyer und Pretschner 2000a] Lötzbeyer, H. ; Pretschner, A.: AutoFocus on Constraint Logic Programming. In: *Proc. (Constraint) Logic Programming and Software Engineering*, 2000

[Lötzbeyer und Pretschner 2000b] Lötzbeyer, H. ; Pretschner, A.: Testing Concurrent Reactive Systems with Constraint Logic Programming. In: *Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming.* Singapore, 2000

[Lutz 1993] Lutz, R.: Targeting Safety-Related Errors During Software Requirements Analysis. In: *Proc. ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 1993, S. 99–106

[Lynch und Tuttle 1989] Lynch, N. ; Tuttle, M.: An Introduction to Input/Output Automata. In: *CWI Quarterly* 2 (1989), Nr. 3, S. 219–246

[Marre und Arnould 2000] Marre, B. ; Arnould, A.: Test Sequence Generation from Lustre Descriptions: GATEL. In: *Proc. 15th IEEE Intl. Conf. on Automated Software Engineering (ASE'00).* Grenoble, 2000

[McCabe 1976] McCabe, T.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), December, Nr. 4

[McCarthy 1978] McCarthy, J.: History of LISP. In: *Proc 1st ACM SIGPLAN conf. on History of programming languages*, 1978, S. 217–223

[McMenamin und Palmer 1984] McMenamin, S. ; Palmer, J.: *Essential System Analysis.* Yourdon Press, 1984

[McMillan 1992] McMillan, K.L.: *Symbolic Model Checking: An Approach to the State Explosion Problem*, Carnegie Mellon University, Dissertation, 1992

[Meudec 2000] Meudec, C.: ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. In: *Proc. 1st Intl. workshop on Automated Program Analysis, Testing, and Verification.* Limerick, 2000

[Meyer 1992] Meyer, B.: Applying "Design by Contract". In: *IEEE Computer* 25 (1992), October, Nr. 10, S. 40–51

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

[Morgan 1990] Morgan, C.: *Programming from Specifications.* Prentice Hall, 1990

[MOST Cooperation 2001] MOST Cooperation: *MOST – Media Oriented Systems Transport. Specification Rev 2.1, Version 2.1-00.* February 2001

[Myers 1979] Myers, G.: *The Art of Software Testing.* J. Wiley and Sons, 1979

[Nierstrasz 1995] Nierstrasz, O.: Regular Types for Active Objects. In: *Object-Oriented Software Composition*, Prentice Hall, 1995, S. 99–121

[Nierstrasz und Papathomas 1990] Nierstrasz, O. ; Papathomas, M.: Viewing Objects as Patterns of Communicating Agents. In: *ACM SIGPLAN Notices* 25 (1990), Nr. 10, S. 38–43

[Nilsson und Lübcke 2000] Nilsson, U. ; Lübcke, J.: Constraint Logic Programming for Local and Symbolic Model Checking. In: *Proc. Computational Logic*, 2000 (Springer LNAI 1861)

[Ntafos 1984] Ntafos, S.: An Evaluation of Required Element Strategies. In: *Proc 7th. Intl. Conf. on Software Engineering*, 1984, S. 250–256

[Ntafos 1988] Ntafos, S.: A Comparison of Some Structural Testing Strategies. In: *IEEE Transactions on Software Engineering* 14 (1988), June, Nr. 6, S. 868–874

[Ntafos 1998] Ntafos, S.: On Random and Partition Testing. In: *Proc. Intl. Symp. on Software Testing and Analysis*, 1998, S. 42–48

[Parnas 1976] Parnas, D.: On the Design and Development of Program Families. In: *IEEE Transactions on Software Engineering* 2 (1976), March, Nr. 4, S. 1–9

[Parrish und Zweben 1991] Parrish, A. ; Zweben, S.: Analysis and Refinement of Software Test Data Adequacy Properties. In: *IEEE Transactions on Software Engineering* 17 (1991), June, Nr. 6, S. 565–581

[Paulson u. a. 1989] Paulson, L. ; Cohen, A. ; Gordon, M.: Letters to the Editors. In: *Communications of the ACM* 32 (1989), Nr. 3, S. 375

[Peled u. a. 1999] Peled, D. ; Vardi, M. ; Yannakakis, M.: Black Box Checking. In: *Proc. Formal Techniques for Networked and Distributed Systems / Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE)*, 1999, S. 225–240

[Pemmasani u. a. 2002] Pemmasani, G. ; Ramakrishnan, C. ; Ramakrishnan, I.: Efficient model checking of real time systems using tabled logic programming and constraints. In: *Proc. Intl. Conf. in Logic Programming*, 2002

[Peyton Jones und Hughes 1999] Peyton Jones, S. ; Hughes, J.: *Report on the Programming Language Haskell 98.* 1999. – http://www.haskell.org/definition

[Philipps 2003] Philipps, J.: *Incremental Design of Embedded Systems Software*, Technische Universität München, Dissertation, 2003. – To be submitted

[Philipps u. a. 2003] Philipps, J. ; Pretschner, A. ; Slotosch, O. ; Aiglstorfer, E. ; Kriebel, S. ; Scholl, K.: *Model-based test case generation for smart cards.* 2003. – Submitted to FMICS'03

[Philipps und Rumpe 2001] Philipps, J. ; Rumpe, B.: Roots of Refactoring. In: *Proc. 10th OOPSLA Workshop on Behavioral Semantics: Back to Basics*, October 2001, S. 187–199

[Philipps und Slotosch 1999] Philipps, J. ; Slotosch, O.: The quest for correct systems: Model checking of diagrams and datatypes. In: *Proc. IEEE Asian Pacific Software Engineering Conf. (APSEC'99)*, 1999, S. 449–458

[Pol und van Veenendaal 1998] Pol, M. ; van Veenendaal, E.: *Structured Testing of Information Systems – an introduction to TMap*. Kluwer, 1998

[PolyspaceTechnologies 1999] PolyspaceTechnologies: *Automatic Runtime Error Detection at Compilation Time*. February 1999. – Polyspace Verifier Technical Presentation

[Pretschner 2001] Pretschner, A.: Classical search strategies for test case generation with Constraint Logic Programming. In: *Proc. Formal Approaches to Testing of Software*, August 2001, S. 47–60

[Pretschner 2003] Pretschner, A.: *Compositional generation for MC/DC test suites*. 2003. – To appear in Electronic Notes in Theoretical Computer Science 82(6)

[Pretschner und Lötzbeyer 2001] Pretschner, A. ; Lötzbeyer, H.: Model Based Testing with Constraint Logic Programming: First Results and Challenges. In: *2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01)*, May 2001

[Pretschner u. a. 2001a] Pretschner, A. ; Lötzbeyer, H. ; Philipps, J.: Model Based Testing in Evolutionary Software Development. In: *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping*, 2001, S. 155–160

[Pretschner u. a. 2003a] Pretschner, A. ; Lötzbeyer, H. ; Philipps, J.: Model Based Testing in Incremental System Development. In: *The Journal of Systems and Software* (2003). – To appear

[Pretschner und Philipps 2001] Pretschner, A. ; Philipps, J.: Heuristische Suche in der Testfallgenerierung. In: *Softwaretechnik-Trends* 21 (2001), Nr. 3, S. 11–12

[Pretschner und Philipps 2002] Pretschner, A. ; Philipps, J.: Szenarien modellbasierten Testens / Institut für Informatik, Technische Universität München. 2002 (TUM-I0205). – Forschungsbericht

[Pretschner und Philipps 2003] Pretschner, A. ; Philipps, J.: *Constraints for test case generation*. 2003. – Submitted to J. Theory and Practice of Logic Programming

[Pretschner und Schätz 2001] Pretschner, A. ; Schätz, B.: Modellbasiertes Testen mit AutoFocus/Quest. In: *Softwaretechnik-Trends* 21 (2001), Nr. 1, S. 20–23

[Pretschner u. a. 2003b] PRETSCHNER, A. ; SLOTOSCH, O. ; AIGLSTORFER, E. ; KRIEBEL, S.: *Model Based Testing for Real—The Inhouse Card Case Study.* 2003. – Submitted to J. Software Tools for Technology Transfer

[Pretschner u. a. 2001b] PRETSCHNER, A. ; SLOTOSCH, O. ; LÖTZBEYER, H. ; AIGLSTORFER, E. ; KRIEBEL, S.: Model Based Testing for Real: The Inhouse Card Case Study. In: *Proc. 6th Intl. Workshop on Formal Methods for Industrial Critical Systems*, 2001, S. 79–94

[Pretschner u. a. 2000] PRETSCHNER, A. ; SLOTOSCH, O. ; STAUNER, T.: Developing Correct Safety Critical, Hybrid, Embedded Systems. In: *Proc. New Information Processing Techniques for Military Systems.* Istanbul, October 2000

[Prowell u. a. 1999] PROWELL, S. ; TRAMMELL, C. ; LINGER, R. ; POORE, J.: *Cleanroom Software Engineering.* Addison Wesley, 1999

[Qadeer und Tasiran 2002] QADEER, S. ; TASIRAN, S.: Promising Directions in Hardware Design Verification. In: *Proc. 3rd IEEE Intl. Symp. on Quality Electronic Design*, 2002

[Ramakrishnan 1988] RAMAKRISHNAN, R.: Magic Templates: A Spellbinding Approach to Logic Programs. In: *Proc. Intl. Conf. on Logic Programming*, 1988, S. 140–159

[Ramamoorthy u. a. 1976] RAMAMOORTHY, C. ; HO, S. ; CHEN, W.: On the Automated Generation of Program Test Data. In: *IEEE Transactions on Software Engineering* SE-2 (1976), December, Nr. 4, S. 293–300

[Rankl und Effing 1999] RANKL, W. ; EFFING, W.: *Handbuch der Chipkarten.* Hanser Verlag, 1999

[Rayadurgan und Heimdahl 2001] RAYADURGAN, S. ; HEIMDAHL, M.: Coverage Based Test-Case Generation using Model Checkers. In: *Proc. 8th Intl. Conf. and Workshop on the Engineering of Computer Based Systems*, 2001, S. 83–93

[Raymond u. a. 1998] RAYMOND, P. ; WEBER, D. ; NICOLLIN, X. ; HALBWACHS, N.: Automatic Testing of Reactive Systems. In: *Proc. 19th IEEE Real-Time Systems Symposium*, 1998

[Riedemann 2002] RIEDEMANN, E.: Empirische Analyse der Leistungsfähigkeit von strukturorientierten Testmethoden. In: *Softwaretechnik-Trends* 22 (2002), August, Nr. 3

[Roever und Engelhardt 1998] ROEVER, W.-P. de ; ENGELHARDT, K.: *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998

[RTCA und EUROCAE 1992] RTCA ; EUROCAE: *DO-178B/ED-12B: Software Considerations in Airborne Systems and Equipment Certification.* December 1992

[Rushby 1995] Rushby, J.: Formal Methods and their Role in the Certification of Critical Systems / Computer Science Laboratory, SRI. 1995 (CSL-95-1). – Forschungsbericht

[Rushby 2001] Rushby, J.: Bus Architectures For Safety-Critical Embedded Systems. In: *Proc. 1st Workshop on Embedded Software (EMSOFT'01)*, Springer Verlag, 2001, S. 306–323

[Rushby 2002] Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. In: *J. Reliability Engineering and System Safety* 75 (2002), Nr. 2, S. 167–177

[Rusu u. a. 2000] Rusu, V. ; du Bousquet, L. ; Jéron, T.: An Approach to Symbolic Test Generation. In: *Proc. Integrated Formal Methods*, 2000

[Sadeghipour und Singh 1998] Sadeghipour, S. ; Singh, H.: *Test Strategies on the Basis of Extended Finite State Machines.* 1998. – Report FT3/SM-98-04, Daimler-Benz AG

[Sammet 1972] Sammet, J.: Programming Languages: History and Future. In: *Communications of the ACM* 15 (1972), July, Nr. 7, S. 601–610

[Sammet 1978] Sammet, J.: The early History of COBOL. In: *ACM SIGPLAN Notices* 13 (1978), August, Nr. 8, S. 121–161

[Sax u. a. 2002] Sax, E. ; Willibald, J. ; Müller-Glaser, K.: Seamless Testing of Embedded Control Systems. In: *Proc. 3rd IEEE Latin American Test Workshop*, 2002, S. 151–153

[Schätz und Huber 1999] Schätz, B. ; Huber, F.: Integrating Formal Description Techniques. In: *Proc World Congress on Formal Methods in the Development of Computing Systems II*, 1999 (Springer LNCS 1709), S. 1206–1225

[Schätz u. a. 2002a] Schätz, B. ; Pretschner, A. ; Huber, F. ; Philipps, J.: Model-Based Development / Institut für Informatik, Technische Universität München. 2002 (TUM-I0204). – Forschungsbericht

[Schätz u. a. 2002b] Schätz, B. ; Pretschner, A. ; Huber, F. ; Philipps, J.: Model-Based Development of Embedded Systems. In: *Advances in Object-Oriented Information Systems* Bd. 2426, Springer LNCS, 2002, S. 298–311

[Sgroi u. a. 2000] Sgroi, M. ; Lavagno, L. ; Sangiovanni-Vincentelli, A.: Formal Models for Embeded System Design. In: *IEEE Design& Test of Computers, Special Issue on System Design* (2000), June, S. 2–15

[Sharygina und Peled 2001] Sharygina, N. ; Peled, D.: A Combined Testing and Verification Approach for Software Reliability. In: *Proc. Formal Methods Europe*, Springer, 2001 (LNCS), S. 611–628

[Shen und Abraham 1999] Shen, J. ; Abraham, J.: An RTL Abstraction Technique for Processor Micorarchitecture Validation and Test Generation. In: *J. Electronic Testing: Theory&Application* 16 (1999), February, Nr. 1-2, S. 67–81

[Siedersleben und Denert 2000] Siedersleben, J. ; Denert, E.: Wie baut man Informationssysteme? - Überlegungen zur Standardarchitektur. In: *Informatik-Spektrum* 23 (2000), Nr. 4, S. 247–257

[Simon 1999] Simon, D.: *An Embedded Software Primer*. Addison Wesley, 1999

[Soley 2000] Soley, R.: *Model Driven Architecture*. 2000. – OMG white paper

[Spitzer 2001] Spitzer, M.: *Modellbasierter Hardware-in-the-Loop Test von eingebetteten elektronischen Systemen*, Universität Karlsruhe, Dissertation, 2001

[Spivey 1992] Spivey, J.M.: *The Z Notation: A Reference Manual*. Prentice Hall, 1992

[Srivastava 1993] Srivastava, D.: Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. In: *Annals of Mathematics and Artificial Intelligence* 8 (1993), Nr. 3–4, S. 315–343

[Stuckey 1991] Stuckey, P.: Constructive Negation for Constraint Logic Programming. In: *Proc. Logic in Computer Science*, 1991

[Thévenod-Fosse und Waeselynck 1992] Thévenod-Fosse, P. ; Waeselynck, H.: On Functional Statistical Testing Designed From Software Behavior Models / Centre national de la recherche scientifique – laboratoire d'analyse et d'architecture des systèmes. February 1992 (92042). – Forschungsbericht

[Thévenod-Fosse u. a. 1995] Thévenod-Fosse, P. ; Waeselynck, H. ; Crouzet, Y.: Software Statistical Testing / Centre national de la recherche scientifique – laboratoire d'analyse et d'architecture des systèmes. March 1995 (95178). – Forschungsbericht

[Tichy 1998] Tichy, W.: Should Computer Scientists Experiment More? In: *IEEE Computer* (1998), May, S. 32–40

[Tip 1995] Tip, F.: A survey of program slicing techniques. In: *J. Programming Languages* 3 (1995), Nr. 3, S. 121–189

[Tracey 2000] Tracey, N.: *A search-based automated test-data generation framework for safety-critical software*, University of York, Dissertation, 2000

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

[Ural 1992] URAL, H.: Formal methods for test sequence generation. In: *Computer Communications* 15 (1992), June, Nr. 5, S. 311–325

[Urbina 1996] URBINA, L.: Analysis of Hybrid Systems in CLP(R). In: *Proc. 2nd Intl. Conf. on Principles and Practice of Constraint Programming*. Cambridge, Massachusetts, USA : Springer Verlag, 1996 (LNCS 1118), S. 451–467

[Vilkomir und Bowen 2001] VILKOMIR, S. ; BOWEN, J.: Formalization of control-flow criteria of software testing / South Bank University. 2001 (SBU-CISM-01-01). – Forschungsbericht

[Visser u. a. 2000] VISSER, W. ; HAVELUND, K. ; BRAT, G. ; PARK, S.: Java PathFinder - Second Generation of a Java Model Checker. In: *Proc. Workshop on Advances in Verification*, 2000

[Voas u. a. 1997] VOAS, J. ; MCGRAW, G. ; KASSAB, L. ; VOAS, L.: Fault-injection: A Crystal Ball for Software Quality. In: *IEEE Computer* 30 (1997), June, Nr. 6, S. 29–36

[von der Beeck 1994] VON DER BEECK, M.: A Comparison of Statecharts Variants. In: *Proc. Formal Techniques in Real–Time and Fault–Tolerant Systems*, 1994 (Springer LNCS 863), S. 128–148

[von Neumann 1960] VON NEUMANN, J.: *Die Rechenmaschine und das Gehirn*. R. Oldenbourg München, 1960

[Vries u. a. 2000] VRIES, R. d. ; TRETMANS, J. ; BELINFANTE, A. ; FEENSTRA, J. ; FEIJS, L. ; MAUW, S. ; GOGA, N. ; HEERINK, L. ; HEER, A. d.: Côte de Resyste in Progress. In: *Progress 2000 – Workshop on Embedded Systems*, October 2000, S. 141–148

[WAP Forum 2001] WAP FORUM: *Wireless Identity Module. Part: Security*. 2001. – Wireless Application Protocol WAP-260-WIM-20010712-a

[Ward und Mellor 1985] WARD, P. ; MELLOR, S.: *Structured Development for Real-Time Systems*. Prentice Hall, 1985

[Warren 1992] WARREN, D.S.: Memoing for Logic Programs. In: *Communications of the ACM* 35 (1992), March, Nr. 3, S. 93–111

[Watson und McCabe 1996] WATSON, A. ; MCCABE, T.: *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. 1996. – NIST Special Publication 500-235

[Wegener 2001] WEGENER, J.: *Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen*, Humboldt Universität Berlin, Dissertation, 2001

[Wegener 2002] WEGENER, J.: *Personal Communication*. September 2002

[Wegener u. a. 2002] WEGENER, J. ; BUHR, K. ; POHLHEIM, H.: Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. In: *Proc. Genetic and Evolutionary Computation Conference*, July 2002

[Weyuker 1986] WEYUKER, E.: Axiomatizing Software Test Data Adequacy. In: *IEEE Transactions on Software Engineering* SE-12 (1986), December, Nr. 12, S. 1128–1138

[Wiesbrock u. a. 2002] WIESBROCK, H.-W. ; CONRAD, M. ; FEY, I. ; POHLHEIM, H.: Ein neues automatisiertes Auswerteverfahren für Regressions- und Back-to-Back-Tests eingebetteter Regelsysteme. In: *Softwaretechnik-Trends* 22 (2002), August, Nr. 3, S. 22–27

[Wimmel u. a. 2000] WIMMEL, G. ; LÖTZBEYER, H. ; PRETSCHNER, A. ; SLOTOSCH, O.: Specification Based Test Sequence Generation with Propositional Logic. In: *J. Software Testing, Validation, and Reliability* 10 (2000), Nr. 4, S. 229–248

[Wirth 1971a] WIRTH, N.: Program Development by Stepwise Refinement. In: *CACM* 14 (1971), April, Nr. 4, S. 221–271

[Wirth 1971b] WIRTH, N.: The programming language Pascal. In: *Acta Inf.* 1 (1971), S. 35–63

[Xie und Engler 2002] XIE, Y. ; ENGLER, D.: Using Redundancies to Find Errors. In: *Proc. 10th Intl. Symposium on the Foundations of Software Engineering*, 2002

[Zave und Jackson 1997] ZAVE, P. ; JACKSON, M.: Four dark corners of requirements engineering. In: *ACM Transactions on Software Engineering and Methodology* 6 (1997), Nr. 1, S. 1–30

[Zhu u. a. 1997] ZHU, H. ; HALL, P. ; MAY, J.: Software Unit Test Coverage and Adequacy. In: *ACM Computing Surveys* 29 (1997), December, Nr. 4, S. 366–427