Formale Methodik des Entwurfs verteilter objektorientierter Systeme

Bernhard Rumpe

Fakultät für Informatik
der Technischen Universität München

Formale Methodik des Entwurfs verteilter objektorientierter Systeme

Bernhard Rumpe

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. J. Eickel

Prüfer der Dissertation:

1. Univ.-Prof. Dr. M. Broy
2. Univ.-Prof. Dr. M. Paul

Die Dissertation wurde am 26.6.1996 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.12.1996 angenommen.

Kurzfassung

In dieser Arbeit wird eine formale Grundlage für eine objektorientierte Methodik entwickelt. Zur Spezifikation von Struktur und Verhalten verteilter objektorientierter Systeme werden Beschreibungstechniken für Objektmodelle, Klassen und Transitionssysteme definiert.

Für die zustandsbasierte Beschreibung nichtdeterministischen Komponentenverhaltens wird die Theorie buchstabierender Automaten entwickelt. Es werden eine konkrete Darstellungsform, eine abstrakte Syntax, eine denotationelle und eine operationelle Semantik angegeben und gezeigt, daß beide Semantiken übereinstimmen.

Für buchstabierende Automaten wird ein Verfeinerungskalkül definiert, der zur Transformation von abstrakten in detaillierte Verhaltensbeschreibungen verwendet werden kann. Es wird gezeigt, daß dieser Kalkül bezüglich der Semantikdefinition korrekt ist. Der Kalkül wird für die Spezialisierung und die Vererbung von Verhaltensbeschreibungen in verteilten objektorientierten Systemen eingesetzt.

Ein Systemmodell charakterisiert eine Menge von verteilten objektorientierten Systemen, die aus asynchron kommunizierenden Agenten aufgebaut sind. Das Systemmodell dient als Basis für die Definition einer integrierten, formalen Semantik für die oben genannten Beschreibungstechniken.

Methodische Entwicklungsschritte erlauben die Verfeinerung von Dokumenten dieser Beschreibungstechniken. Basierend auf der formalen Semantik der Verfeinerungsschritte werden präzise Aussagen über das Zusammenspiel der verwendeten Beschreibungstechniken definiert.

Die Verbindung graphischer Beschreibungstechniken mit einer integrierten, formalen Semantik nutzt Synergieeffekte formaler und praxisorientierter Ansätze der Softwaretechnik.

Danksagung

Für Kommentare zu Vorversionen dieser Arbeit und interessanten Gesprächen im Zusammenhang mit Automaten bedanke ich mich bei Max Fuchs, Radu Grosu, Cornel Klein, Barbara Paech und Bernhard Schätz. Für das aufmerksame Durchlesen einer Vorversion möchte ich mich bei Wolfgang Frech, Franz Huber, Alexander Schmidt, Oscar Slotosch und Katharina Spies bedanken. Jan Philipps, Birgit Schieder und Franz Regensburger danke ich für die Diskussionen über Beweistechniken.

Weiterhin möchte ich mich bei Prof. Dr. Manfred Broy für die Ermutigung, diese Arbeit zu beginnen und für die Unterstützung bei ihrer Erstellung bedanken. Prof. Dr. Manfred Paul gebührt mein Dank für die Übernahme des zweiten Gutachtens. Ganz besonders danken möchte ich beiden für die maßgebliche Beeinflussung meines wissenschaftlichen Lebens seit Beginn meines Studiums.

Nicht zuletzt danke ich allen meinen Bekannten, meiner Familie und vor allem Sonja für die Geduld, die sie während der Erstellung dieser Arbeit mit mir hatten.

Inhaltsverzeichnis

I Prolog **1**

1 Einführung **3**

1.1 Einleitung . . . 3
1.2 Motivation der Arbeit . . . 5
1.3 Wichtigste Ziele und Ergebnisse . . . 8
1.4 Verwandte Arbeiten . . . 9
1.5 Aufbau dieser Arbeit . . . 10

2 Systementwicklung **13**

2.1 Dokumente . . . 14
2.2 Entwicklungsschritte . . . 19
2.3 Dokumentarten – Kurzdarstellung . . . 21
2.4 Methodische Überlegungen . . . 24
2.5 Zusammenfassung . . . 29

3 Systemmodell **31**

3.1 Kernschicht des Systemmodells . . . 31
3.1.1 Anforderungen an das Systemmodell . . . 32
3.1.2 Black-Box-Sicht von Agenten . . . 33
3.1.3 Das Kommunikationsmedium . . . 34
3.1.4 Systemablauf . . . 38
3.1.5 State-Box-Sicht von Agenten . . . 38
3.2 Klassen im Systemmodell . . . 39
3.2.1 Anforderungen an die Klassifizierung . . . 40
3.2.2 Klassen und Vererbung . . . 42
3.2.3 Werte, Typen und Variablen . . . 43
3.2.4 Attribute und Methoden . . . 44
3.2.5 Dynamische Erzeugung von Agenten . . . 45
3.2.6 Zustand und Verhalten von Agenten . . . 47
3.3 Systemmodell — Zusammenfassung . . . 49
3.4 Bemerkungen zum Systemmodell . . . 49
3.4.1 Ein- und Ausgabe des Systems . . . 49
3.4.2 Alternative Modellierungen . . . 51

II Buchstabierende Automaten 53

4 Darstellung buchstabierender Automaten 55
4.1 Abstrakte Syntax für Automaten 56
4.2 Darstellungstechniken für Automaten 58
4.3 Hüllenbildung 62

5 Semantik für Automaten 65
5.1 Denotationelle Semantik 65
5.1.1 Deterministische, totale Automaten 67
5.1.2 Totale Automaten 68
5.1.3 Allgemeine Automaten 73
5.2 Operationelle Semantik 76
5.2.1 Abläufe 76
5.2.2 Operationelle Semantik 77
5.2.3 Relationsbasierte Semantik 79
5.3 Erweiterungen 81
5.3.1 Automaten gebildet aus stromverarbeitenden Funktionen 81
5.3.2 Gezeitete Semantik buchstabierender Automaten 82
5.3.3 Bündel von Kanälen 83
5.3.4 Taktautomaten 84
5.3.5 Fairneß bei der Auswahl von Transitionen 88
5.4 Verwandte Ansätze 90
5.5 Zusammenfassung 92

6 Verfeinerungstechniken für Automaten 93
6.1 Verfeinerungsrelation für Automaten 93
6.2 Modifikation der Initialmenge 100
6.2.1 Verkleinern der Initialmenge 100
6.2.2 Modifikation finaler Initialelemente 100
6.3 Modifikation von Transitionen 101
6.3.1 Entfernung von Transitionen 101
6.3.2 Hinzufügen von Transitionen 101
6.3.3 Behandlung der Zielzustände finaler Transitionen 102
6.4 Modifikation von Zuständen 103
6.4.1 Entfernung unerreichbarer Zustände 103
6.4.2 Erweiterung der Zustandsmenge 104
6.4.3 Verfeinerung der Zustandsmenge 104
6.5 Schnittstellenverfeinerung 105
6.6 Der Verfeinerungskalkül 107
6.7 Verwandte Arbeiten 112
6.8 Zusammenfassung 113

III Beschreibungstechniken 115

7 Objektmodell 117
7.1 Datentypen und Prädikate . . . 118
7.2 Klassenbeschreibungen . . . 121
7.3 Objektmodelle . . . 127
7.4 Zusammenfassung . . . 134

8 Automatendokumente 135
8.1 Konzepte für Automatendokumente . . . 136
8.1.1 Beispielautomat Stack . . . 136
8.1.2 Abstraktion und Darstellung . . . 140
8.1.3 Spezielle Datentypen . . . 142
8.2 Syntax und Semantik von Automatendokumenten . . . 143
8.2.1 Klassen- und Typautomaten . . . 143
8.2.2 Konkrete Syntax eines Automatendokuments . . . 144
8.2.3 Patternfreie Syntax für Automaten . . . 145
8.2.4 Kontextbedingungen an ein Automatendokument . . . 146
8.2.5 Transformation in einen buchstabierenden Automaten . . . 147
8.2.6 Semantik eines Automatendokuments . . . 149
8.3 Automatenerstellung und -transformation . . . 149
8.3.1 Neuerstellung eines Automatendokuments . . . 150
8.3.2 Verfeinerung eines Automatendokuments . . . 151
8.3.3 Vererbung eines Automatendokuments . . . 158
8.4 Automatendokumente im Dokumentgraph . . . 161
8.4.1 Protokolle und die erweiterte Automatensemantik . . . 161
8.4.2 Integration von Automatendokumenten . . . 162
8.4.3 Entwicklungsschritte und Redundanztest . . . 163
8.4.4 Vollständigkeit eines Dokumentgraphen . . . 164
8.5 Bemerkungen . . . 167
8.6 Zusammenfassung . . . 170

IV Epilog 173

9 Zusammenfassung und Ausblick 175

Literaturverzeichnis 179

V Anhang 189

A Mathematische Grundlagen 191
A.1 Mathematische Grundlagen, Notation . . . 191
A.2 Klassisches, ungezeitetes Focus . . . 193
A.3 Gezeitete Modellierungen . . . 201

B Beweisschemata 205
B.1 Beweisschema I . . . 205
B.2 Beweisschema A . . . 206
B.3 Beweisschema BP . . . 207
B.4 Beweisschema B . . . 208
B.5 Beweisschema C . . . 209

C Beweise 211
C.1 Systementwicklung . . . 211
C.2 Darstellung buchstabierender Automaten . . . 211
C.3 Semantik für Automaten . . . 211
C.4 Verfeinerungstechniken für Automaten . . . 223
C.5 Objektmodell . . . 234
C.6 Automatendokumente . . . 236

D Glossar 241

E Index 245
E.1 Grundlagen-, FOCUS-Index . . . 245
E.2 Allgemeiner Index . . . 246

Teil I

Prolog

Dieser Teil motiviert die Arbeit und stellt ihre Ergebnisse vor. Es wird eine methodische Grundlage für Systementwicklungen definiert, die Entwicklungsschritte als Transformation oder Neuerstellung von Dokumenten formalisiert. Ein Systemmodell dient zur Charakterisierung der Menge von Systemen, an deren Implementierung wir interessiert sind.
Dieser Teil ist wie folgt strukturiert:

1 Einführung

2 Systementwicklung

3 Systemmodell

Kapitel 1

Einführung

1.1 Einleitung

Industrielle Entwicklung von Softwareprodukten besitzt in der heutigen, stark auf Informationstechnik ausgerichteten Wirtschaft eine außerordentliche Bedeutung. Die Erstellung industrieller Softwareprodukte ist daher ähnlich systematischen und strukturierten Richtlinien zu unterwerfen, wie es bei anderen industriellen Produkten üblich ist. Denn nur durch systematische, strukturierte Techniken lassen sich Entwurfs- und Produktionsprozeß sowie die Produktqualität verbessern.

In den klassischen Ingenieurdisziplinen werden deshalb ausgereifte, genormte Techniken eingesetzt, die die verschiedenen Phasen der Erstellung eines Produkts, von der Entwicklung bis zur Wartung, unterstützen. Solche Techniken bieten Produktsichten, in denen einzelne Aspekte herausgearbeitet werden. Beispiele hierzu sind etwa Schaltpläne. Diese Techniken werden zumeist durch Werkzeuge unterstützt, die bei der Anwendung der Techniken Routinearbeiten übernehmen, sowie (überprüfbare) Entwicklungsfehler verhindern. Darüberhinaus gibt es in klassischen Ingenieurdisziplinen genaue Richtlinien für den Produktionsprozeß, der im allgemeinen in mehrere, klar abgegrenzte Phasen untergliedert ist.

Softwaretechnik unterscheidet sich als sehr junge, noch nicht etablierte Ingenieurdisziplin – der Begriff „*Software Engineering*“ ([NR69]) ist noch keine 30 Jahre alt – von anderen Ingenieurdisziplinen wie Maschinenbau oder Elektrotechnik. Software hat gegenüber den materiellen Produkten anderer Disziplinen den Vorteil äußerst leichter Reproduzierbarkeit, weshalb der Produktionsprozeß für Software praktisch vernachlässigt werden kann. Softwaresysteme arbeiten mit vielen und komplex strukturierten Daten, so daß sie eine inhärente Komplexität besitzen, die von anderen Produkten nicht erreicht wird. Deshalb ist es um so wichtiger, daß der Konstruktionsprozeß (in der Informatik auch Entwicklungsprozeß genannt) systematisch und strukturiert ist. Erst dadurch wird Kosten- und Qualitätsmanagement für Softwareentwicklungen möglich.

In 30 Jahren ist es einer so schnell wachsenden und sich verändernden Disziplin wie der Informatik nicht möglich gewesen, die mathematisch fundierten Grundlagen bereitzustellen und zu einem Gesamtkonzept zu integrieren, die zur systematischen, industriellen Softwareentwicklung notwendig sind. Auch aus diesem Grund sind in der Informatik die Hauptrichtungen theoretische Informatik und Softwaretechnik entstanden, die sich beide mit unterschiedlichen Aspekten der Entwicklung von Software beschäftigen.

In der theoretischen Informatik wird unter anderem eine Fundierung von Softwaresystemen durch formale Modelle entwickelt. Ergebnisse solcher Arbeiten sind in Lehrbüchern, wie [BW82] von Bauer/Wössner oder [DIJ76] von Dijkstra, dokumentiert. An der Theorie orientierte Ansätze versuchen jedoch meistens, ein Softwaresystem monolithisch in einem Modell zu beschreiben. Unter Methodik wird in diesen Ansätzen eine Sammlung von Regeln verstanden, die es erlauben, iterativ ein Modell zu transformieren (verfeinern).

Die in der theoretischen Informatik entwickelten Techniken sind zwar mathematisch gut fundiert, aber nur sehr schwer und aufwendig anzuwenden. Sie besitzen nur wenig verständliche Darstellungsformen und werden daher in der Softwaretechnik kaum verwendet. Aufgrund mangelnder Skalierbarkeit eignen sie sich auch nicht zur Entwicklung größerer Systeme.

In der an der Praxis orientierten Softwaretechnik sind zunächst strukturierte und dann objektorientierte Methoden entwickelt worden, die in der systematischen Softwareerstellung Verwendung gefunden haben. Diese Methoden benutzen meist mehrere unterschiedliche Beschreibungstechniken, um verschiedene Sichten des zu erstellenden Softwareprodukts zu definieren.

Im Unterschied zur Theorie bieten solche Softwareentwicklungsmethoden aufgrund der Richtlinien, die bis zu einem Softwareentwicklungsprozeß ausgebaut sein können, die (eingeschränkte) Möglichkeit, den Projektfortschritt zu quantifizieren und damit zu messen. Dadurch werden Maßnahmen zur Steuerung von Projekten anwendbar. Nachteil der meisten heutigen Softwareentwicklungsmethoden ist jedoch das Fehlen einer exakten Semantik der benutzten Beschreibungstechniken. Insbesondere sind die Integration der durch verschiedene Beschreibungstechniken definierten Sichten und die damit verbundenen Konsistenzbedingungen nicht ausreichend geklärt.

Zwischen den theoretischen Ansätzen und den in der Softwaretechnik verwendeten Methoden ist eine Lücke entstanden, die es zu schließen gilt, um die Vorteile beider Welten in eine Softwareentwicklungsmethode zu integrieren.

Die so entstehenden Synergieeffekte könnten dazu führen, daß einerseits den Beschreibungstechniken der pragmatischen Methoden eine formale Semantik unterlegt wird und so mehr Funktionalität in Form von Transformationen oder Konsistenzprüfungen durch unterstützende Werkzeuge angeboten werden kann. Andererseits könnte die bei formalen Techniken fehlende Skalierung auf große Systeme mittels pragmatischer und gleichzeitig formal fundierter Beschreibungstechniken erreicht werden.

1.2 Motivation der Arbeit

In diesem Abschnitt charakterisieren wir die Motivation dieser Arbeit und erläutern die wichtigsten Entwurfsentscheidungen.

Bereits erwähnt wurde die Hauptmotivation dieser Arbeit, eine formale Grundlage für eine praxisnahe, objektorientierte Softwareentwicklungsmethode zu definieren, um so die Synergie theoretischer und praxisorientierter Ansätze zu nutzen.

Methoden der Softwareentwicklung: In den letzten Jahrzehnten sind zunächst strukturierte Softwareentwicklungsmethoden, wie JSD ([JAC83]), SSADM ([AG90]) oder die Methode nach Denert ([DEN91b]), entstanden. Ihnen folgten objektorientierte Softwareentwicklungsmethoden, wie OMT ([RUM91]), Objectory ([JAC93]), Fusion ([Cetal94]), Responsibility-Driven Design ([WWW90]) oder die Methoden nach Booch ([BOO94]), Coad/Yourdon ([CY91, CY91b]) und Shlaer/Mellor ([SM92]). Es gibt damit eine Fülle von praktischen Richtlinien zum Entwurf von Software- und oft auch Hardwaresystemen. Diese wurden für sehr unterschiedliche Bereiche, wie betriebliche Informationssysteme, Multimediasysteme, Kommunikationssysteme oder auch echtzeitkritische, technische Steuersysteme entwickelt und werden daher den sich unterscheidenden Anforderungen dieser verschiedenen Systemklassen unterschiedlich gerecht.

Softwareentwicklungsmethoden unterscheiden sich in den von ihnen genutzten Beschreibungstechniken. Einig sind sich jedoch alle in der Praxis eingesetzten Methoden, daß *eine* homogene Beschreibungstechnik nicht ausreicht, um alle Aspekte eines Softwaresystems zu charakterisieren. Wir werden deshalb mehrere Beschreibungstechniken betrachten, die besonders häufig im Softwareentwurf eingesetzt werden.

Objektmodell: Allen Methoden ist eine Beschreibungstechnik gemeinsam, die auf dem in [CHE76] vorgestellten Entity/Relationship-Modell basiert und dieses in der einen oder anderen Form erweitert. E/R-basierte Beschreibungstechniken beschreiben das statische Datenmodell. Sie beschreiben, mit welchen Daten ein Softwaresystem arbeitet und welche Beziehungen und Restriktionen zwischen diesen Daten gelten. Wie die Praxis der Softwareentwicklung heute zeigt, ist das statische Datenmodell unverzichtbare Grundlage jedes Softwareprojekts. Deshalb werden auch wir uns mit der Entwicklung einer E/R-basierten Beschreibungstechnik, dem *Objektmodell* beschäftigen. Dieses teilen wir in die *Klassenbeschreibung* für die Struktur einzelner Klassen und das eigentliche Objektmodell, das Daten- und Vererbungsbeziehungen zwischen Klassen definiert.

Transitionssysteme: Eine Beschreibungstechnik, die sich in sehr vielen Methoden in der einen oder anderen Form wiederfinden läßt, ist das *Transitionssystem* (*Automat*, siehe [RUM91]). Diese gibt es in vielen Variationen zur Beschreibung des operationellen Verhaltens einzelner Komponenten, z.B. in Form von I/O-Automaten ([LS89]), aber verstärkt auch zur Beschreibung des Lebenszyklus eines Objekts ([SM92, Jetal91]). Automaten erfreuen sich, nicht zuletzt in der hierarchischen Variante *Statecharts* ([HAR87]), steigender Beliebtheit, weil diese neben einem Objektmodell zur Kommunikation zwischen Softwareentwicklern geeignet sind. Wir werden in dieser Arbeit insbesondere Automatenkonzepte genauer untersuchen und eine für die Softwareentwicklung adäquate

Automatenklasse entwickeln. Die Definition dieser Automatenklasse und deren Semantik wird einen guten Teil dieser Arbeit einnehmen, da hier wesentliche wissenschaftliche Neuerungen aufgezeigt werden.

Systemmodell: Jede einzelne Beschreibungstechnik, die wir für unsere Untersuchung ausgewählt haben, besitzt eine formale Semantik. Diese Semantiken sind jedoch nicht integriert. Eine einheitliche semantische Basis für alle benutzten Beschreibungstechniken ist aber für ein Verständnis der Zusammenhänge einzelner Beschreibungsdokumente und für rigorose Konsistenzprüfungen zwischen diesen dringend erforderlich. Wir werden deshalb eine semantische Basis definieren, die es erlaubt, alle von uns definierten Beschreibungstechniken innerhalb eines *Systemmodells* zu formalisieren. Das hier entwickelte Systemmodell beschreibt die Struktur und die Abläufe eines Systems unter Benutzung mathematisch fundierter Modellierungstechniken.

Verteilte asynchrone Systeme: Wir wählen den Anwendungsbereich verteilter, asynchron kommunizierender Systeme, der einerseits industrielle Relevanz besitzt und andererseits mit den heute existierenden praktischen Ansätzen nicht zufriedenstellend behandelt werden kann. Wir konzentrieren uns weiterhin auf verteilte objektorientierte Systeme. Beispiele sind Kommunikationssysteme auf ATM-Basis ([MS95]) und verteilte Bürosysteme, die mit Kommunikation über Formulare (Anträge, etc.) arbeiten und heute verstärkt als lose Kopplung früher unabhängiger Anwendungen integriert werden. Wir werden dabei eine Realisierung mit zentraler Datenbank vernachlässigen, da diese als Client-Server-Systeme ([INM91]) heute bereits relativ gut beherrscht werden.

Grundlage Focus: Die zur Definition des Systemmodells benutzte mathematische Modellierungstechnik muß asynchrone Kommunikation zwischen Komponenten als grundlegendes Kommunikationsmittel zur Verfügung stellen. Sie muß ein Zustandskonzept für Komponenten anbieten, das es erlaubt, Zustände einerseits für Automaten, andererseits für das Objektmodell zu nutzen. Sie muß eine exakte Schnittstellenbeschreibung ermöglichen, um Eingabe- und Ausgabesignaturen zu definieren. Sie muß Kompositionskonzepte anbieten, um die einzelnen beschriebenen Komponenten zu einem Gesamtsystem zusammensetzen zu können. Diese Kompositionskonzepte müssen modular sein, damit eine vom Kontext einer Komponente unabhängige Weiterentwicklung der Komponente erfolgen kann. Als letztes müssen adäquate Verfeinerungskonzepte zur Verfügung stehen, die es erlauben, das Verhalten einer Komponente genauer festzulegen, deren Schnittstelle zu verfeinern und Komponenten hierarchisch zu dekomponieren. Dies alles bietet die am Lehrstuhl Broy entwickelte und von uns als semantische Fundierung benutzte Modellierungstechnikteil von Focus ([Betal93]). Wir werden deshalb die Modellierungstechnik Focus einführen und für unsere Anforderungen erweitern.

Buchstabierende Automaten: Eine mächtige Erweiterung der Modellierungstechnik Focus wird in Form von buchstabierenden Automaten angegeben. Diese buchstabierenden Automaten erhalten eine abstrakte Syntax und eine konkrete graphische Darstellung. Die Definition der denotationellen Semantik eines Automaten wird durch Transformation in ein Prädikat über stromverarbeitenden Funktionen definiert, die das Ein-/Ausgabeverhalten der durch den Automaten charakterisierten Komponente beschreiben.

Verfeinerungsregeln für Automaten: Automaten werden einerseits zur Charakterisierung der Komponenten unseres Systemmodells in Form von Prädikaten eingesetzt. Andererseits werden Automaten als syntaktische Beschreibungstechnik von Zustand und Verhalten einer Komponente verstanden. Sie besitzen insbesondere Transformationsregeln für deren Verfeinerung, die einer Verfeinerung auf semantischer Ebene entsprechen, aber ohne Zuhilfenahme der Semantik verwendet werden können. Insbesondere können damit unsere Automaten gemeinsam mit den darauf definierten Transformationen in einem Softwareentwicklungswerkzeug implementiert werden, ohne daß der Anwender dieses Werkzeugs auf die Semantikdefinition Bezug nehmen muß.

Vererbung von Automaten: Das von uns definierte Objektmodell charakterisiert die statische Sicht des Systems. Es wird deshalb zur Beschreibung der Netzsichten des Systems, der syntaktischen Schnittstellen der Komponenten und deren Datenzustände verwendet und entsprechend formalisiert. Da wir uns in der propagierten Methodik relativ stark an OMT ([RUM91]) und Fusion ([Cetal94]), zwei heute verbreitete Methoden, anlehnen, werden wir für jede mit dem Objektmodell eingeführte Klasse einen Automaten zur Beschreibung des Verhaltens der Instanzen dieser Klasse fordern. Wir werden besprechen, wie Automaten zwischen Klassen vererbt werden, und damit einen Mangel heutiger Softwareentwicklungsmethoden (nicht nur OMT) beheben.

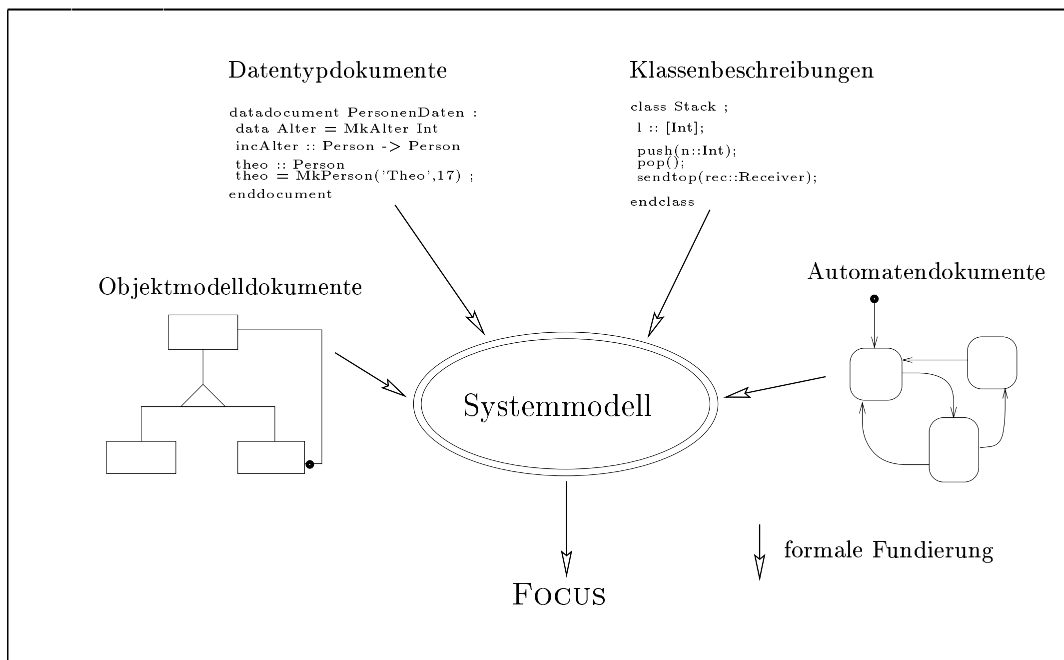


Abbildung 1.1: Formalisierte Dokumentarten dieser Arbeit

Als ein Ergebnis dieser Arbeit entsteht die in Abbildung 1.1 veranschaulichte Formalisierung der oben genannten Beschreibungstechniken.

Methodik: Die Definition und Formalisierung von Beschreibungstechniken ist ohne methodische Überlegungen sicher nicht vollständig. Wir geben daher einige Richtlinien zum Einsatz der hier vorgeschlagenen Beschreibungstechniken an. Die Erfahrung praktisch arbeitender Softwareentwickler zeigt, daß methodische Richtlinien für den Einsatz von Beschreibungstechniken einer Methode erst dann konsolidiert werden können, wenn diese in mehreren Projekten erprobt wurden. Das zeigt sich daran, daß nahezu alle heute in der Praxis verwendeten Methoden, die auch methodische Richtlinien zur Vorgehensweise besitzen, von Praktikern der Softwareentwicklung mit sehr viel Projekterfahrung entwickelt worden sind. Wir werden methodische Überlegungen für die hier definierten Beschreibungstechniken auf die Anwendung in Beispielen beschränken und nutzen dazu eigene und fremde praktische Erfahrungen der Anwendung artverwandter Beschreibungstechniken.

1.3 Wichtigste Ziele und Ergebnisse

In diesem Abschnitt fassen wir noch einmal die Ziele und Ergebnisse dieser Arbeit zusammen:

1. Für den Entwurf verteilter objektorientierter Software wird eine formale Methodik entwickelt, die sich an praktischen Techniken orientiert.
2. Es wird eine formale Grundlage für verteilte objektorientierte Softwareentwicklung geschaffen.
3. Ein Systemmodell definiert präzise die Menge der betrachteten Systeme.
4. Die definierten Beschreibungstechniken Automatendokument, Objektmodell, Klassenbeschreibung und Datendokument erhalten eine integrierte formale Semantik anhand des Systemmodells.
5. Entwicklungsschritte werden als syntaktische Transformationen von Beschreibungsdokumenten definiert. Die definierten Entwicklungsschritte korrespondieren zu der von der Semantik vorgegebenen Verfeinerungsrelation. Dies wird bewiesen.
6. Die definierte Methodik demonstriert, daß in der Praxis eingesetzte graphische Beschreibungstechniken mit formalen Ansätzen kombiniert und so die Vorteile beider Techniken vereint werden können.
7. Weit über bisherige Ansätze zur Formalisierung graphischer Beschreibungstechniken hinaus werden syntaxbasierte Entwicklungsschritte formalisiert und als relativ zur Semantik korrekt bewiesen.

Wir stellen daher die These auf, daß ein Softwareentwicklungswerkzeug, basierend auf diesen Beschreibungstechniken und Entwicklungsschritten, weitaus rigorosere Kontextprüfungen durchführen und besser Beschreibungen einer Technik in eine andere umsetzen kann, als dies bei heute üblichen Werkzeugen der Fall ist.

Als ein wesentlicher Baustein obiger Softwareentwicklungsmethodik wird die Theorie buchstabierender Automaten entwickelt. Sie ist eine eigenständige Einheit dieser Arbeit:

1. Ein für die Modellierungstechnik FOCUS verwendbarer Beitrag entsteht durch die Theorie buchstabierender Automaten, der die Spur- und funktionale Sicht von FOCUS ergänzt.
2. Buchstabierende Automaten erlauben eine kompakte, zustandsbasierte Definition von Komponentenverhalten und besitzen einfache aber mächtige Regeln für die Verfeinerung.
3. Es wird bewiesen, daß der Verfeinerungskalkül für buchstabierende Automaten korrekt ist.
4. Buchstabierende Automaten sind für die Beschreibung objektorientierter Systeme angepaßt. Sie besitzen ein Zustandskonzept, das abstrakt genug ist, um im Gegensatz zu I/O-Automaten auf Kontrollzustände zu verzichten und nur Datenzustände zu benutzen.

Wir formulieren als zweite These, daß eine große Klasse von verteilten Applikationen sich unter Benutzung buchstabierender Automaten mit FOCUS sehr kompakt entwickeln läßt.

1.4 Verwandte Arbeiten

Einfluß auf den theoretischen Teil dieser Arbeit hatten die Definition der I/O-Automaten ([LS89]) und deren Anwendungen. Es gibt Arbeiten im Umfeld von FOCUS, die Automaten zur Modellierung von Komponentenverhalten nutzen, von denen wir hier nur [DEN91], [FUC94] und [SPI94] nennen wollen.

Viel Einfluß auf den an der Praxis orientierten Teil dieser Arbeit hatten praktische Softwareentwicklungsmethoden, speziell die bereits erwähnten Methoden OMT ([RUM91]), Fusion ([Cetal94]) und die Methode nach Booch ([BOO94]).

Die Idee, ein Systemmodell zu formalisieren und darauf aufbauend den benutzten Beschreibungstechniken eine integrierte formale Semantik zu geben, ist in das Projekt SYSLAB ([RKB95]) übernommen worden.

Erste Ansätze zur Erzeugung eines Synergieeffektes zwischen pragmatischen und formalen Methoden werden in den Arbeiten von Hußmann ([HUS93, HUS93b]) deutlich, der diese Idee in [HUS94] mit den in SSADM verwendeten Beschreibungstechniken ausgeführt hat. Eine formale Behandlung von Transformationsschritten auf diesen Beschreibungstechniken wurde aber nicht durchgeführt.

Weitere Arbeiten beschäftigen sich mit der Formalisierung einzelner Beschreibungstechniken, wie E/R-Modellen ([HET93, HET95]), oder Datenflußtechniken ([NIC93]) mit Hilfe der algebraischen Spezifikationssprache SPECTRUM oder Time Sequence Charts ([FAC95]) durch FOCUS.

Neuere Entwicklungsmethoden werden von den Autoren gelegentlich selbst mit einer formalen Semantik unterlegt, wie die Methode nach Embley/Kurz/Woodfield ([EKW92]) in [CEW92a] oder Document-Driven Analysis ([DXT92]). Dabei werden meist eigenschafts- oder modellorientierte formale Sprachen, wie in diesen Beispielen VDM ([JON90]) oder Z ([SPI88]) verwendet. Dies birgt den bereits oben besprochenen Nachteil in sich, daß zwar die Methode selbst eine formale Semantik besitzt, aber alle Schlußfolgerungen (Transformationen und Konsistenzbedingungen) durch ein Verifikationswerkzeug erledigt werden müssen und deren Ergebnisse als große Formeln vorliegen.

Darüberhinaus gibt es objektorientierte Erweiterungen formaler Sprachen, wie etwa OOZE ([AG92]). Siehe dazu auch [STE92a]. Diese sind zur Definition einer Semantik für Beschreibungstechniken gegenüber der Mathematik sehr starr und erfordern oft umständliche Formalisierungen. Gegenüber den vom Softwareentwickler genutzten graphischen Techniken sind sie so nachteilig, daß sie in der Praxis nur dort eine Rolle spielen werden, wo ein logikbasierter Formalismus und ein zugehöriges Verifikationswerkzeug benötigt werden.

Theoretische Kalküle, die speziell auf Objektorientierung zugeschnitten werden, sind von (theoretischem) Interesse, da sie ein Verständnis für Objektorientierung vermitteln. Beispiele hierfür sind etwa der π-Kalkül ([MPW92]), der sich mit der Mobilität von Objekten beschäftigt, die Arbeiten von Grosu ([GRO95]), Abadi ([ABA93]) und Nierstrasz ([NIE91]) oder mehr auf Typkorrektheit abzielende Arbeiten, wie die von Cardelli ([CAR84, CAR93]), Liskov/Wing ([LW93]) und Palsberg/Schwartzbach ([PS91]). Diese haben jedoch häufig den Nachteil, daß sie nur einzelne Aspekte der Objektorientierung betrachten und demgegenüber andere Aspekte vernachlässigen. Objektorientierung macht ihre Vorteile jedoch erst durch das Zusammenspiel all ihrer Einzelaspekte richtig nutzbar.

Eine in ihrer Art neue Zusammenstellung von Techniken zur mathematischen Modellierung von Beschreibungstechniken, die in einer Softwareentwicklungsmethode verwendet werden können, ist die Arbeit [BRO95].

1.5 Aufbau dieser Arbeit

Das zweite Kapitel beschäftigt sich mit der Fundierung des Softwareentwicklungsprozesses. Es wird der Dokumentbegriff formalisiert und ein Konzept für Entwicklungsschritte festgelegt. Darauf aufbauend werden methodische Überlegungen angestellt.

Im dritten Kapitel wird ein Systemmodell definiert. Wir charakterisieren damit die Klasse aller uns interessierenden Systeme. Im ersten Teil des Anhangs werden dazu die Grundlagen von Focus beschrieben, die in dieser Arbeit genutzt werden.

Dann folgen mehrere Kapitel, die die Theorie buchstabierender Automaten entwickeln. Das vierte Kapitel beschäftigt sich mit der Definition eines Automatenbegriffs und der graphischen Darstellung dieser buchstabierenden Automaten.

Im fünften Kapitel wird für die abstrakte Syntax, die zur Charakterisierung von buchstabierenden Automaten verwendet wird, eine denotationelle Semantik in Form einer Menge stromverarbeitender Funktionen definiert. In diesem Kapitel werden auch wesentliche Eigenschaften dieser denotationellen Semantik gezeigt.

Das sechste Kapitel behandelt Verfeinerungstechniken für buchstabierende Automaten. Essenz dieses Kapitels ist ein Kalkül von Verfeinerungstransformationen für buchstabierende Automaten.

Die Kapitel sieben und acht dieser Arbeit formalisieren Beschreibungstechniken auf der Basis des Systemmodells. Im siebten Kapitel wird eine Syntax für die Beschreibung von vereinfachten Objektmodellen und von Klassenbeschreibungen definiert und ebenfalls in das Systemmodell integriert. Dabei wird auch auf Datentypdefinitionen eingegangen.

Das achte Kapitel überträgt die Formalisierung von buchstabierenden Automaten und deren Verfeinerungskalkül aus den Kapiteln vier bis sechs auf Automatendokumente. Automatendokumente bieten eine konkrete syntaktische Darstellung zustandsbasierter Verhaltensbeschreibungen.

Im Anhang werden genutzte mathematische Grundlagen und Beweisschemata dargestellt. Der Anhang enthält darüberhinaus alle Beweise, die nicht in der Arbeit selbst zu finden sind.

Zuletzt sind im Anhang ein Glossar der verwendeten Begriffe und ein Index angegeben.

Kapitel 2

Systementwicklung

Die Entwicklung eines Systems ist bestimmt durch dessen Komplexität und deshalb selbst ein komplexer Vorgang. Deshalb ist es notwendig, den Vorgang der Systementwicklung zu strukturieren und damit zu systematisieren. In diesem Kapitel definieren wir grundlegende Begriffe der Systementwicklung und entwerfen eine formale Grundlage für die Behandlung methodischer Überlegungen. Diesen allgemeinen Rahmen instantiieren wir im Rest der Arbeit mit einer konkreten Menge von Beschreibungstechniken, um so eine Methodik zu definieren.

Einer der Basisbegriffe der Systementwicklung ist das *Dokument*. Es dient zur Beschreibung gewisser Aspekte oder Ausschnitte eines zu erstellenden Software-/Hardwaresystems. Typische Tätigkeiten bei der Systementwicklung sind daher die Erstellung von Dokumenten und die Transformation von Dokumenten in neue Dokumente.

Je nach Art der benutzten Beschreibungstechniken entsteht während der Systementwicklung eine größere Menge auf komplexe Weise miteinander in Verbindung stehender Dokumente unterschiedlicher Arten. Um die Beziehungen der einzelnen Dokumente untereinander besser zu verstehen, ist eine integrierte Semantik dieser Dokumente unerläßlich. Für die präzise Formulierung von Kontextbedingungen zwischen Dokumenten und die semantikerhaltende Übersetzung eines Dokuments (Generierung) muß die Semantik formal sein. Deshalb definieren wir in Kapitel 3 eine formale Semantik, die wir mit *Systemmodell* bezeichnen.

Im ersten Abschnitt dieses Kapitels wird der Dokumentbegriff formalisiert und allgemein charakterisiert, wie eine Semantikdefinition aussieht. Darüberhinaus wird der Dokumentgraph als Basisstruktur der Systementwicklung eingeführt.

Während einer Systementwicklung müssen Dokumente nach bestimmten Regeln erstellt und verarbeitet werden. Deshalb werden im zweiten Abschnitt Entwicklungsschritte und deren Wirkung auf den Dokumentgraphen diskutiert.

Im dritten Abschnitt beschreiben wir im Vorgriff auf die folgenden Kapitel eine Kurzdarstellung der in dieser Arbeit behandelten Dokumentarten.

Der vierte Abschnitt enthält schließlich allgemeine Überlegungen zur zielgerichteten Anwendung von Entwicklungsschritten in der Systementwicklung. Dabei wird der Begriff der Methodik als Sammlung korrekter Entwicklungsschritte ebenso formalisiert, wie der Begriff der Methode als geordnete Kombination dieser Entwicklungsschritte.

Im letzten Abschnitt formulieren wir eine Zusammenfassung dieses Kapitels.

2.1 Dokumente

Ein *Dokument* ist eine bei der Systementwicklung verwendete, abgegrenzte Einheit zur Beschreibung eines Aspektes oder eines Ausschnitts eines zu erstellenden Systems.

Unter *Ausschnitt* eines Systems verstehen wir einen Systemteil, z.B. eine Komponente oder ein Subsystem. Ein *Aspekt* eines Systems charakterisiert eine bestimmte Sicht, wie z.B. die Kommunikationsstruktur innerhalb seiner Komponenten, unter Abstraktion von Details eines Systems, die zu anderen Aspekten des Systems gehören.

Während verschiedener Phasen der Systementwicklung werden unterschiedliche Arten von Dokumenten erstellt und auf unterschiedliche Weise genutzt. Ein Dokument besitzt eine Syntax, die eine eingeschränkte Form der Konsistenz innerhalb des Dokuments sichert, und eine Semantik, die beschreibt, welche Bedeutung dem Dokument zugemessen wird. Dokumente können textbasiert oder, wie heute für viele Dokumentarten üblich, graphenbasiert sein. Jede Systementwicklungsmethode benutzt eine mehr oder weniger explizite Form dieses Dokumentbegriffs. In pragmatischen Softwareentwicklungsmethoden ist darüber hinaus die angegebene Semantik für Dokumente sehr informell.

OMT ([RUM91]) zum Beispiel nutzt ein reichhaltiges Objektmodell zur Darstellung der statischen Struktur, Szenarien, die damit eng verwandten „Event Traces" für exemplarische Abläufe sowie hierarchische Automaten mit Statechart-ähnlicher Notation zur Beschreibung des dynamischen Verhaltens einzelner Komponenten. Als weitere Technik stehen in OMT (Version '91) Datenflußdiagramme zur Beschreibung von funktionalem Verhalten zur Verfügung, deren Verbleib in zukünftigen Versionen von OMT aufgrund der unklaren Semantik allerdings fraglich ist. In [BOO94] verwendet Booch ein Instanzdiagramm, das er mit „Object Diagram" bezeichnet, sowie ein Modul- und ein Prozeßdiagramm zur Darstellung weiterer Aspekte des Systems. In SSADM ([AG90]) werden als wesentlichste Dokumentarten die „Logical Data Structure" eine dem E/R-Modell verwandte Technik, das Datenflußdiagramm und die „Entity Life History" verwendet.

Jede Systementwicklung hat ein lauffähiges System als Ziel. Deshalb ist der Quelltext in einer Programmiersprache eine wichtige Dokumentart. Weitere Dokumentarten sind eher informeller Natur, besitzen aber dennoch eine nicht zu unterschätzende Bedeutung bei der Systementwicklung. Beispiele hierzu sind etwa das Pflichtenheft von SSADM ([AG90]) oder das „Data Dictionary" von Fusion ([Cetal94]). Auch bei formalen Systementwicklungen mit algebraischen oder modellbasierten Spezifikationssprachen wie Spectrum ([Betal93b]) oder Z ([SPI88]) existiert ein Dokumentbegriff, auch wenn es

nur eine Art von Dokumenten gibt. Letztere haben eben wegen der Homogenität der Dokumente eher die Möglichkeit eine formale Semantik anzugeben, als heute typischerweise verwendete pragmatische Softwareentwicklungsmethoden mit ihren vielen unterschiedlichen Arten von Dokumenten.

Zur Definition der Semantik von Dokumenten benutzen wir das Systemmodell $\mathcal{SM}$. Das Systemmodell, das in Kapitel 3 definiert wird, charakterisiert die Menge von Systemen, an deren Entwicklung wir interessiert sind.

Definition 2.1 (Dokument)
Ein Dokument *ist eine bei der Systementwicklung verwendete, abgegrenzte Einheit zur Beschreibung eines Aspektes oder eines Ausschnitts eines zu erstellenden Systems. Ein Dokument besitzt eine exakt definierte* abstrakte Syntax *und eine konkrete Darstellungsform. Die Menge aller Dokumente wird mit* $\mathcal{DOC}$ *bezeichnet. Ein Dokument erhält eine Semantik mittels folgender Funktion:*

$$[\![.]\!] : \mathcal{DOC} \to \mathbb{P}(\mathcal{SM}) \qquad (\Box)$$

Für jede in dieser Arbeit definierte Dokumentart wird eine eigene Semantikdefinition angegeben. Diese wird mit einem hochgestellten Index, z.B. $[\![.]\!]^{om}$ für Objektmodell-Dokumente, markiert. Die konkrete Darstellungsform eines Dokuments spielt eine wichtige Rolle für die Akzeptanz dieser Dokumente als Beschreibungstechnik. Für die Formalisierung der Semantik ist jedoch die abstrakte Syntax ausreichend. Sie bietet eine Reduktion des Dokuments auf die für eine Formalisierung wesentlichen Aspekte an. Im allgemeinen wird so vom graphischen Layout und von Kommentaren, die für das Verständnis der Intention eines Dokuments oft wichtig sind, abstrahiert.

Die Semantik wird so gewählt, daß jedes Dokument eine gewisse Menge von Systemen charakterisiert. Einer Menge von Dokumenten wird damit folgende Semantik gegeben:

Definition 2.2 (Semantik einer Dokumentmenge)
Einer Menge von Dokumenten $D \subseteq \mathcal{DOC}$ *wird eine Semantik mittels der Erweiterung der Semantikabbildung auf Mengen gegeben:*

$$[\![.]\!] : \mathbb{P}(\mathcal{DOC}) \to \mathbb{P}(\mathcal{SM})$$
$$[\![D]\!] \stackrel{def}{=} \bigcap_{d \in D} [\![d]\!] \qquad (\Box)$$

Es ist zu beachten, daß die Semantik der leeren Menge von Dokumenten alle Systeme des Systemmodells beinhaltet.

Nun können die Konsistenz einer Dokumentmenge und die Folgerungsbeziehung formuliert werden:

Definition 2.3 (Konsistenz einer Dokumentmenge)
Eine Menge von Dokumenten $D \subseteq \mathcal{DOC}$ *heißt* konsistent*, wenn ihre Semantik nicht leer ist. Wir schreiben:*

$$consistent(D) \stackrel{def}{\Leftrightarrow} [\![D]\!] \neq \emptyset \qquad (\Box)$$

Definition 2.4 (Folgerungsbeziehung für Dokumente)
Ein Dokument $d \in \mathcal{DOC}$ *ist eine* Folgerung *aus einer Menge von Dokumenten* $D \subseteq \mathcal{DOC}$, *wenn die Hinzunahme von* d *zu* D *keine Änderung der Semantik beinhaltet:*

$$D \models d \overset{def}{\Leftrightarrow} [\![D]\!] = [\![D \cup \{d\}]\!]$$

Ein Dokument $d \in D$ *heißt* redundant in D, *wenn gilt* $(D \setminus d) \models d$.

Wir erweitern $\models$ *auf Mengen:*

$$D_1 \models D_2 \overset{def}{\Leftrightarrow} \forall d \in D_2 .\ D_1 \models d$$

und führen die Modellierungsäquivalenz $\models\!\!=\!\!\mid$ *ein:*

$$D_1 \models\!\!=\!\!\mid D_2 \overset{def}{\Leftrightarrow} D_1 \models D_2 \wedge D_2 \models D_1 \qquad (\square)$$

Nach Definition ist die Folgerungsbeziehung der Syntax mit der Inklusionsbeziehung der Semantik identisch. Es gilt:

$$D_1 \models D_2 \Leftrightarrow [\![D_1]\!] \subseteq [\![D_2]\!]$$

Die hier definierte Semantik für Dokumente steht in einem engen Zusammenhang mit mathematischer Logik ([EFT86]) und algebraischen Spezifikationssprachen mit loser Semantik (vgl. [Betal93b]), die ähnliche Strukturen zur Modellbildung benutzen. Werden nur Axiome als Dokumente benutzt und ein Axiom zur Beschreibung des Systemmodells verwendet, so kann die in der mathematischen Logik verwendete Modellbildung als Semantik verwendet werden. Neben der an der Semantik orientierten Folgerungsbeziehung nutzen wir für verschiedene Arten von Dokumenten eine an der Syntax orientierte Ableitungsbeziehung $\vdash$. Bei den meisten Dokumentarten werden wir jedoch unterschiedlich benannte Transformationsarten definieren. Diese bilden in ihrer Gesamtheit einen Entwicklungskalkül für Dokumente.

Die Entscheidung über die Konsistenz von Dokumenten und die Redundanz einzelner Dokumente in Dokumentmengen sollte bei der Systementwicklung, genau wie bei der mathematischen Logik, nicht durch Betrachten der Semantik, sondern durch syntaktische Kriterien getroffen werden können. In der mathematischen Logik werden Formeln als syntaktisches Darstellungsmittel verwendet, die ab einer gewissen Ausdrucksmächtigkeit (z.B. der Logik erster Stufe) nicht mehr automatisch auf Konsistenz (Widerspruchsfreiheit) und Redundanz (Implikation) überprüft werden können. Hingegen sollte bei der Systementwicklung eine möglichst weitgehende automatisierte Überprüfung von Konsistenz gewährleistet werden. Es ist für die pragmatische Anwendbarkeit einer Systementwicklungsmethode von Vorteil, eine automatisierte Überprüfbarkeit zu besitzen, selbst wenn dadurch Einschränkungen der Beschreibungsmächtigkeit der Dokumentarten in Kauf genommen werden müssen. So verwenden heute übliche Systementwicklungsmethoden weitgehend Beschreibungstechniken, die nur eingeschränkte Ausdrucksmöglichkeiten bieten, und sind dennoch – oder vielleicht gerade deswegen – erfolgreich.

Ein Dokument kann im allgemeinen nur im Kontext anderer Dokumente definiert werden. Beispielsweise können Datentypen in einem Dokument nur verwendet werden, wenn diese vorher definiert wurden. Bei den in Kapitel 8 definierten Automatendokumenten ist eine Transformation häufig nur unter einer Zusicherung, z.B. daß eine Klasse Subklasse

einer anderen ist, möglich. Solche Eigenschaften werden nicht als Teil des Dokuments selbst gefordert, sondern als Kontextbedingungen des Dokuments verstanden. Eine Kontextbedingung ist eine Forderung, die an den Kontext des Dokuments gestellt wird, damit das Dokument selbst sinnvoll und korrekt ist. Zu beachten ist, daß die umgekehrte Sichtweise, daß der Kontext Bedingungen an das Dokument stellt, äquivalent ist. Viele Kontextbedingungen lassen sich aus dem Dokument selbst extrahieren. Bei der in Kapitel 8 definieren Dokumentart für Automaten benötigen wir jedoch dafür wesentliche Informationen, in welcher Weise das Dokument erstellt wurde. Wir strukturieren daher Dokumente in einen

- *konstruktiven* Teil, der das „Ergebnis“ eines Dokuments enthält, und ein
- *Protokoll*, das beschreibt, wie das Dokument entwickelt worden ist.

Ein Dokument $d=(d_{cons},d_{prot})$ besteht also aus zwei Teilen, dem konstruktiven Teil d_{cons} und dem Protokoll d_{prot}. Aus dem Protokoll gehen Kontextbedingungen hervor, deren Einhaltung sichert, daß der konstruktive Anteil sinnvoll ist, und es erlaubt ist, die Semantik für den konstruktiven Teil zu bilden. Zum Beispiel entstehen aus dem Protokoll eines Automatendokuments Verifikationsbedingungen, deren Gültigkeit aus dem Kontext folgen muß. Wir bezeichnen mit $[\![d_{prot}]\!]$ die Menge aller Systeme, die die aus dem Protokoll entstehenden Bedingungen erfüllen, und mit $[\![d_{cons}]\!]$ die Menge aller Systeme, die den konstruktiven Teil erfüllen. Es gilt also

$$[\![(d_{cons},d_{prot})]\!] = \{\ sys \in [\![d_{prot}]\!] \mid sys \in [\![d_{cons}]\!]\ \},$$

und das ist gleichwertig zu:

$$[\![(d_{cons},d_{prot})]\!] = [\![d_{prot}]\!] \cap [\![d_{cons}]\!].$$

Das Dokument d darf nur in Kontexten verwendet werden, in denen die aus dem Protokoll d_{prot} entstehende Zusicherung gilt. Formal bedeutet dies, daß ein korrekter Kontext, bestehend aus einer Menge von Dokumenten D, den Protokollteil impliziert: $D \models d_{prot}$. Nach Definition gilt daher für die gemeinsame Semantik von $D \cup \{d\}$:

$$[\![D \cup \{d\}]\!] = [\![D]\!] \cap [\![d_{prot}]\!] \cap [\![d_{cons}]\!] = [\![D \cup \{d_{cons}\}]\!]$$

Im korrekten Kontext trägt also nur der konstruktive Anteil eines Dokuments zur Semantikbildung bei. Das Protokoll ist notwendig, um

- die Korrektheit des Dokuments und seiner Entwicklung zu sichern,
- die Korrektheit der Verfeinerungsbeziehung zu anderen Dokumenten zu sichern und
- die Entwicklungsgeschichte für eventuelle Replays zu speichern.

Die während einer Systementwicklung auftretenden Dokumente sind oft wechselseitig redundant. Systementwicklung ist aber eine zielgerichtete Aktivität, bei der am Ende eine Menge ausführbarer Dokumente (z.B. Quelltext einer Programmiersprache) steht, während alle anderen Dokumente, die zwischendurch entstanden sind, redundant gegenüber dieser Dokumentmenge sind. Für Wartung und Dokumentation erfüllen diese Dokumente aber weiterhin einen nicht zu unterschätzenden Beitrag, weshalb Dokumente einer Systementwicklung nicht entfernt, sondern als redundant markiert werden.

Die Zielgerichtetheit der Systementwicklung erlaubt es daher nicht, die dabei entstehenden Dokumente als Menge zu verwalten, sondern benötigt eine gerichtete Struktur, die zu jedem Dokument die Menge der Vorgängerdokumente beschreibt und eine Markierung für Redundanz verwaltet. Wir definieren daher den *Dokumentgraphen*, verzichten aber auf die Verwaltung von Versionen desselben Dokuments und Revisionsschritten innerhalb dieses Graphen. Beide sind für eine Systementwicklungsmethode unerläßliche Techniken, führen aber über die Ziele dieser Arbeit hinaus.

Der Dokumentgraph enthält wesentliche Komponenten der Projektbibliothek, die Denert in seinem Plädoyer [DEN93] für eine dokumentenorientierte Vorgehensweise bei der Systementwicklung vorschlägt. Mit dem hier definierten Dokumentbegriff bilden wir eine Grundlage für den formalen Anteil dieser Dokumente und ergänzen so [DEN93].

Für Dokumentgraphen setzen wir eine ausreichend große Grundmenge von Knoten $\mathcal{N}$ voraus.

Definition 2.5 (Dokumentgraph)
Ein Dokumentgraph *ist ein gerichteter Graph (N,E,D,R), bestehend aus*

- *einer endlichen Menge von Knoten* $N \subseteq \mathcal{N}$,
- *einer Abbildung* $D: N \to \mathcal{DOC}$, *die jedem Knoten ein Dokument zuordnet,*
- *einer Teilmenge* $R \subseteq N$, *der* Redundanzmarkierung *und*
- *einer gerichteten, azyklischen Relation* $E \subseteq N \times N$ *von Kanten, die beschreiben, welche Dokumente (Vorgänger) bei der Entwicklung eines Dokuments benutzt wurden.*

Ein Dokument ist höchstens dann als redundant markiert, wenn sein konstruktiver Teil von den konstruktiven Teilen der direkten Nachfolger impliziert wird:

$$\forall n \in N.\ n \in R \Rightarrow D(succ_E.n)_{cons} \models (D.n)_{cons}$$

Die durch das Protokoll geforderten Eigenschaften des Kontexts eines Dokuments muß aus den konstruktiven Teilen der Vorgänger folgen:

$$\forall n \in N.\ D(pred_E.n)_{cons} \models (D.n)_{prot}$$

Für jeden Knoten n wird die Menge der direkten Nachfolger mit $(succ_E.n)$ *und die Menge der direkten Vorgänger mit* $(pred_E.n)$ *bezeichnet. Die Menge der Dokumentgraphen wird mit* $\mathcal{DG}$ *bezeichnet.* (□)

Der Dokumentgraph ist nach Definition azyklisch, weshalb eine wechselseitige Redundanz nicht möglich ist. Die Intention der Redundanzmarkierung ist es, anzuzeigen, von welchen Dokumenten nachgewiesen ist, daß sie in der Dokumentstruktur durch ihre Nachfolger redundant wurden. Die Redundanzmarkierung ist damit eine auf syntaktischen Kriterien beruhende Markierung, die hinreichend ist für die auf semantischen Kriterien basierende Eigenschaft der Redundanz. Dadurch kann bei der Systementwicklung mit Dokumenten gearbeitet werden, ohne auf die Semantik explizit zurückgreifen zu müssen.

Die einem Hypertext ähnliche Struktur des Dokumentgraphen erlaubt seine nichtsequentielle Weiterentwicklung. Sie ermöglicht so, daß mehrere Entwickler parallel und unabhängig voneinander Dokumente entwerfen und verfeinern können. Dies ist für eine effektive teamorientierte Systementwicklung unerläßlich.

Definition 2.6 (Semantik eines Dokumentgraphen)
Die Semantik eines Dokumentgraphen (N,E,D,R) ist durch die Semantik seiner Dokumente festgelegt:

$$[\![(N,E,D,R)]\!] \stackrel{def}{=} [\![D(N)]\!] \qquad (\square)$$

Proposition 2.7 (Redundante Dokumente im Dokumentgraphen)
Die Semantik eines Dokumentgraphen (N,E,D,R) kann bereits aus den konstruktiven Anteilen der nicht redundanten Dokumente geschlossen werden:

$$[\![(N,E,D,R)]\!] = [\![D(N \backslash R)_{cons}]\!]$$ (□, **Beweis siehe** C.1)

Ein Dokumentgraph ist *konsistent*, wenn seine Dokumentmenge konsistent ist. Dazu ist nach Definition nur die Menge von Dokumenten zu betrachten, die nicht redundant sind.

2.2 Entwicklungsschritte

Im Verlauf einer Systementwicklung werden Entwicklungsschritte vorgenommen, die das zu entwickelnde System immer weiter präzisieren und immer implementierungsnähere Beschreibungen einführen. Formal ist ein solcher Entwicklungsschritt eine Modifikation des Dokumentgraphen. Dabei gibt es mehrere Varianten:

1. Das Erstellen und Hinzufügen eines neuen Dokuments,
2. die Weiterentwicklung eines vorhandenen Dokuments zu einem neuen Dokument,
3. die Fusion einer Menge von Dokumenten zu einem neuen Dokument,
4. die Modifikation eines existierenden Dokuments und
5. die Entfernung eines Dokuments.

Die beiden letzten Schritte sind im allgemeinen Revisionsschritte, die bereits getroffene Entscheidungen zurücknehmen und daher Verhalten und Struktur der charakterisierten Systeme nicht erhalten oder verfeinern. Wie bereits dargelegt, sind in einer Systementwicklungsmethode Mechanismen anzubieten, die derartige Schritte unterstützen. Wird ein Dokument modifiziert (und damit ein Entwicklungsschritt zurückgenommen), das bereits weiter verwendet wurde, so ist ein Replay-Mechanismus, wie er etwa in KIV ([RS93]) angeboten wird, erforderlich. Dennoch betrachten wir als Entwicklungsschritte in dieser Arbeit nur Verfeinerungsschritte.

Verfeinernde Entwicklungsschritte bilden die *konstruktive* Umsetzung der Verfeinerungsrelation $\models$. Das bedeutet, Entwicklungsschritte geben aufgrund ihrer konstruktiven Anwendbarkeit eine Anleitung über die Vorgehensweise bei der Systementwicklung, während eine Charakterisierung als Relation $\models$ weniger Anhaltspunkte liefert. Die Definition einer geeigneten Sammlung spezialisierter Entwicklungsschritte ist für den konstruktiven Systementwurf daher von großer Wichtigkeit.

Entwicklungsschritte der ersten Form sind immer dann nötig, wenn ein neues Dokument, zum Beispiel die Beschreibung einer neuen Klasse, entsteht. Die Weiterentwicklung von Dokumenten (2.), etwa die Anreicherung von Klassenbeschreibungen um neue Attribute oder Methoden, geht von einem gegebenen Dokument aus und fügt Informationen hinzu. Dadurch wird also das weiterentwickelte Dokument redundant. Wurden parallele Weiterentwicklungen, zum Beispiel eines Objektmodells, vorgenommen, so ist eine Fusion (3.) bzw. Integration mehrerer Dokumente notwendig. Gerade weil wir eine teamorientierte Entwicklungsmethodik zulassen wollen, ist die Fusion von Dokumenten ein wichtiger Entwicklungsschritt, der für jede einzelne Dokumentart existieren muß.

Der nachfolgend definierte Entwicklungsschritt $\mathcal{DS}$ subsumiert alle oben beschriebenen Varianten. Für konkrete Dokumentarten ist dieser jeweils entsprechend zu instantiieren. Insbesondere sind die hier angegebenen, teilweise auf der semantischen Folgerungsbeziehung $\models$ beruhenden Bedingungen durch syntaktische Kontextbedingungen zu ersetzen. Dann können Kontextbedingungen überprüft oder verifiziert werden, ohne daß dazu auf das Systemmodell Bezug genommen werden muß.

Definition 2.8 (Entwicklungsschritt)
Ein Entwicklungsschritt *ist eine Transformation des Dokumentgraphen (N,E,D,R), bei dem ein Dokument d hinzugefügt wird. Das Dokument d hat die Menge P von Dokumenten als Vorgänger von denen eine Teilmenge $Q \subseteq P$ als redundant markiert wird. Ein Entwicklungsschritt $\mathcal{DS}$ hat folgende Form:*

$$\mathcal{DS} : \mathcal{DG} \times \mathcal{DOC} \times \mathcal{N} \times \mathbb{P}(\mathcal{N}) \times \mathbb{P}(\mathcal{N}) \rightharpoonup \mathcal{DG}$$
$$\mathcal{DS}((N,E,D,R),d,n,P,Q) = (N \cup \{n\},\ E \cup P \times \{n\},\ D + [n \mapsto d],\ R \cup Q)$$

Die Anwendung eines Entwicklungsschritts muß folgende Eigenschaften erfüllen, um korrekt zu sein und damit aus einem Dokumentgraphen wieder einen Dokumentgraphen zu erzeugen:

$$n \notin N$$
$$Q \subseteq P \subseteq N$$
$$D(P)_{cons} \models d_{prot}$$
$$d \models D(Q)_{cons}$$

(□)

Ein Entwicklungsschritt fügt ein neues Dokument d in den Dokumentgraphen *(N,E,D,R)* mit den Vorgängern P ein und markiert einen Teil dieser Vorgänger als redundant. Die Korrektheit des Entwicklungsschritts besteht im wesentlichen in der Zusicherung, daß die Markierung korrekt ist. Die Einführung eines neuen Dokuments ohne Vorgänger wird mit einer leeren Menge P ebenfalls erlaubt.

Die Korrektheit der Anwendung eines Entwicklungsschritts sichert, daß aus einem Dokumentgraphen wieder ein Dokumentgraph erzeugt wird. Dieser muß jedoch nicht konsistent sein.

Manche Entwicklungsschritte sind automatisiert anwendbar. Beispiel hierzu sind etwa die Erzeugung von Objektcode aus Quelltext oder die Übersetzung eines Relationenmodells in ein Datenbankschema. Hier ist das neue, generierte Dokument redundant relativ zu

dem Dokument, aus dem es generiert wurde. Diese Eigenschaft wird jedoch in dieser einfachen Form des Dokumentgraphen nicht festgehalten.

Genau wie die Menge der Dokumente wird die Menge der Entwicklungsschritte von der Systementwicklungsmethode so vorgegeben, daß nur korrekte Entwicklungsschritte möglich sind. Die Dokumentstruktur, insbesondere die Mengen von Knoten und Kanten, werden bei der Systementwicklung nicht explizit zur Verfügung gestellt, sondern von einem (CASE-)Werkzeug verwaltet.

Während der Systementwicklung gibt es Dokumente, die nicht durch einen einzelnen Nachfolger, sondern durch mehrere Nachfolger redundant werden. Dies ist typischerweise dann der Fall, wenn ein Dokument eine globale Sicht auf eine Komponente beschreibt, die durch mehrere einzelne Komponenten verfeinert werden soll. Deshalb gibt es neben den eigentlichen Entwicklungsschritten noch den Redundanztest:

Definition 2.9 (Redundanztest)
Ein Redundanztest *ist eine Transformation des Dokumentgraphen, bei dem ein Dokument auf Redundanz bezüglich seiner Nachfolger getestet wird. Ein Redundanztest $\mathcal{RT}$ hat demnach folgende Form, wobei $n \in N$ gelten muß:*

$$\mathcal{RT} : \mathcal{DG} \times \mathcal{N} \rightarrow \mathcal{DG}$$
$$\mathcal{RT}((N,E,D,R),n) = (N,E,D,R')$$
$$\texttt{where}$$
$$R' \stackrel{def}{=} \begin{cases} R \cup \{n\} & \texttt{if} \quad D(succ_E.n)_{cons} \models (D.n)_{cons} \\ R & \texttt{otherwise} \end{cases}$$

(□)

Der Redundanztest kann, soweit er automatisierbar ist, immer dann ablaufen, wenn ein Dokument einen neuen Nachfolger erhält und noch nicht als redundant markiert wurde.

2.3 Dokumentarten – Kurzdarstellung

Wir haben nun die Dokumentstruktur definiert und festgelegt, wie Entwicklungsschritte in dieser Dokumentstruktur aussehen. Wir geben nun eine kurze informelle Übersicht über die in dieser Arbeit definierten Dokumentarten. Die formale Definition der Dokumentarten sowie die Beweise zur Korrektheit der zugehörigen Entwicklungsschritte sind in den Kapiteln 7 und 8 zu finden. Um einen ersten Eindruck der in dieser Arbeit entwickelten Beschreibungstechniken zu vermitteln, ist in Abbildung 2.1 jeweils ein Dokument jeder Art angegeben.

Die in dieser Arbeit vorgestellten Dokumentarten sind vor allem für die Entwurfsphase einer Systementwicklung gedacht. Für die frühe Phase der Analyse fehlen in dieser Arbeit entsprechende Dokumente, die sich etwa zur Beschreibung von exemplarischen Abläufen oder Geschäftsprozessen eignen.

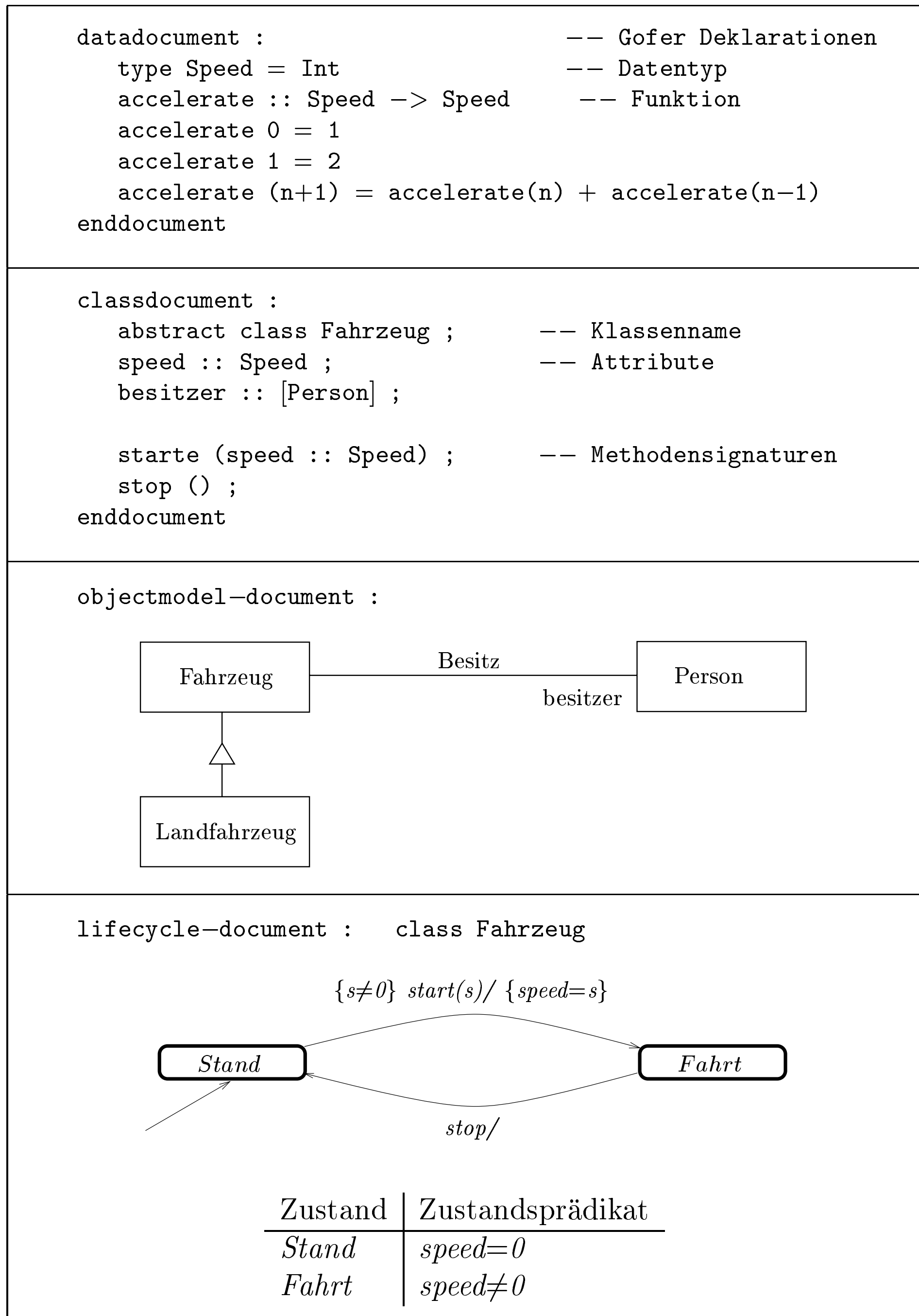


Abbildung 2.1: Die Dokumentarten

Datentypdefinitionen

Datentypdefinitionen sind ein wichtiges Beschreibungsmittel für den Systementwurf. Daher ist es notwendig, komfortabel neue Datentypen definieren zu können. Darüberhinaus ist die Definition von Funktionen auf diesen Datentypen zu ermöglichen.

Es hat sich in der Praxis gezeigt, daß die grundlegenden Datentypen, die in einem System genutzt werden, wie etwa Zahlen, Namen oder Verbunde über diesen, bereits zu einem sehr frühen Zeitpunkt in der Systementwicklung so weit bekannt sind, daß diese nicht axiomatisch charakterisiert werden müssen, sondern als Datentyp formuliert werden können. Deshalb wird eine Dokumentart Datentypdefinition benutzt, die stark an die funktionale Programmiersprache *Gofer* angelehnt ist.

Klassenbeschreibung

Jede einzelne Klasse ist durch ein Dokument, der *Klassenbeschreibung*, beschrieben. Sie definiert die in den Komponenten dieser Klasse gekapselte Datenstruktur in Form einer Menge von Attributen und deren Sorten sowie die Methodensignatur einer Klasse.

Zusätzlich wird in der Klassenbeschreibung festgelegt, wieviele Instantiierungen dieser Klasse in einem System stattfinden können. Ist eine Klasse abstrakt, so wird keine Instantiierung erlaubt. Von manchen Klassen wird genau ein Exemplar benötigt, das während der gesamten Laufzeit zur Verfügung stehen soll.

Objektmodell

Grundlage fast jeder pragmatischen Entwicklungsmethode ist eine Beschreibung der Daten des Systems und deren Beziehungen untereinander. Bekannte Vertreter dieser Gattung sind das E/R-Modell, vorgestellt in [CHE76], und deren Erweiterungen bis hin zum Objektmodell.

Ein *Objektmodell* stellt für uns eine graphische Repräsentation der vorhandenen Klassen von Komponenten eines Systems dar. Beziehungen werden zwischen je zwei Klassen angegeben. Sie beschreiben eine durch entsprechende Attribute realisierte Datenabhängigkeit.

Neben diesen Beziehungen zwischen Klassen gibt es die Vererbung zwischen Klassen, die sich bei der Definition der Klassenbeschreibung und des Lebenszyklus einer Klasse ausdrückt. Wir lassen auch Mehrfachvererbung zu.

Lebenszyklen

Zu jeder Klasse wird neben der Klassensignatur ein *Lebenszyklus* in Form eines Automaten angegeben. Dieser beschreibt die möglichen Zustandsübergänge eines Elements dieser Klasse bei der Verarbeitung von Nachrichten und die mögliche Reaktion darauf.

Ein Lebenszyklus kann sehr fein definiert werden, so daß er sich direkt zur Ausführung in einem Prototyp eignen kann. Er kann aber auch sehr abstrakt definiert werden, indem nur Äquivalenzklassen der Eingabe-, Ausgabenachrichten und der Zustandsmenge betrachtet werden.

Der zu einer Klasse gehörende Lebenszyklus wird von deren Superklassen abgeleitet. Er kann durch einige Verfeinerungstransformationen weiter spezialisiert werden.

Wir definieren zwei unterschiedliche Formen von Lebenszyklen, den Typ- und den Klassenautomaten. Der Typautomat beschreibt Lebenszyklen für alle Elemente einer Klasse und ihrer Subklassen, der Klassenautomat nur für die Instanzen, die direkt zu einer Klasse gehören. Während Klassenautomaten zur Implementierungsbeschreibung eingesetzt werden, dienen Typautomaten als Zusicherung von Verhaltenseigenschaften für die Klienten einer Klasse. Typautomaten werden im Gegensatz zu Klassenautomaten vererbt.

Weitere Dokumentarten

Es gibt eine Vielzahl weiterer Dokumentarten, die in dieser Arbeit nicht formalisiert werden, von denen sich aber für andere Anwendungsgebiete eine Formalisierung lohnen könnte. Einige davon sollen hier erwähnt werden:

- Instanzdiagramme [BOO94],
- Szenarien [RUM91], Objektinteraktionsgraphen [Cetal94],
- Entwurfsmuster [Getal94],
- Datenflußdiagramme [DEM79] und
- Programmiersprachen.

2.4 Methodische Überlegungen

Die Formalisierung des Dokumentbegriffs, einer Semantik und passender Transformationsregeln führen uns zu einer algebraischen Struktur für Dokumente. Durch die Einbettung der Dokumente in Dokumentgraphen und darauf definierten Operationen für das Einbringen neuer Dokumente und den Redundanztest entsteht eine zweite algebraische Struktur für Dokumentgraphen. Schritte, die während einer Systementwicklung vorgenommen werden, sind damit formalisiert und können als korrekt oder nicht korrekt erkannt werden.

Diese Schritte sind jedoch nicht von alleine zielgerichtet. Vielmehr muß bei der Systementwicklung der Entwickler durch sein Wissen und Können und einem gewissen Maß an Intuition Entwicklungsschritte so einsetzen, daß er seinem Ziel, einer lauffähigen Implementierung durch ein System, effektiv näher kommt. Dabei müssen nicht nur funktionale Aspekte, die wir in dieser Arbeit behandeln, sondern oft auch nichtfunktionale Aspekte beachtet werden. Portabilität, Benutzung bestimmter vorgegebener Architekturen oder Bibliotheken, Robustheit und Ausfallsicherheit von Hardwarekomponenten sind dabei zu betrachten.

Methodik

Durch geeignete Richtlinien für die Anwendung von Entwicklungsschritten kann dem Entwickler bei seiner Arbeit Unterstützung gegeben werden. Eine Richtlinie kann sich auf die Anwendung einer einzelnen Regel beziehen, kann aber auch mit dem gesamten Entwicklungsverlauf befaßt sein.

Eine *Methodik* besteht, im Unterschied zum nachfolgend behandelten Begriff der Methode, aus einer Charakterisierung der Dokumentarten, die zur Systementwicklung verwendet werden, und deren Semantik. Darüberhinaus ist eine Menge von Transformationen angegeben, die Dokumente in definierter Weise manipulieren. Für diese Transformationen werden Richtlinien beschrieben, die Ziel und Zweck dieser Transformationen festlegen. Unter Transformation kann neben Modifikation einzelner Dokumente auch Kombination oder Neuerstellung von Dokumenten verstanden werden.

Eine formale Methodik kann Dokumente, deren Semantik und deren Transformationen formalisieren. Richtlinien bleiben jedoch notwendigerweise informell und werden wie Heuristiken eingesetzt. Wir legen fest:

Definition 2.10 (Formale Methodik)
Eine formale Methodik *besteht aus:*

- *einer formal definierten Menge von Dokumentarten,*
- *einer formalen Semantik für Dokumentarten,*
- *einer Sammlung von Entwicklungsschritten, die Dokumente transformieren; auch diese sind formal fundiert, und*
- *einer Sammlung von Richtlinien, die Anwendungsbereiche und Ziele jeder Transformation beschreiben.*

(□)

Ein Beispiel für eine formale Methodik ist Focus ([Betal93]).

Richtlinien, die den Anwendungsbereich und Ziele einer Transformation beschreiben, sind meist stark abhängig vom Umfeld des Problems. Häufig sind mehrere unterschiedliche Techniken anwendbar, die zu unterschiedlichen Lösungen führen. Sehr explizit verwendet wird dieses Prinzip bei den Entwurfsmustern (Design Pattern, siehe [Getal94]), die typischerweise in drei inhaltliche Teile strukturiert sind. Neben der Problembeschreibung wird die Beschreibung der Lösung und Konsequenzen in der Anwendung des Entwurfsmusters beschrieben. In den Entwurfsmustern und den mitgegebenen Anwendungsrichtlinien ist damit sehr viel Wissen von Entwicklern enthalten, die diese Entwurfsmuster in vielen Projekten entworfen und weiterentwickelt haben. Ganz analog ist zur Charakterisierung von Richtlinien für die Transformationen dieser Arbeit die Erfahrung echter Projekte notwendig. Daher wird in dieser Arbeit darauf verzichtet, komplett vorgegebene Richtlinien zu entwerfen und stattdessen durch kleine Anwendungsbeispiele für interessantere Transformationen gezeigt, wie diese angewendet werden können.

Taktiken und Strategien

Wird die Sammlung von Entwicklungsschritten einer Methodik zu neuen zielgerichteten Schritten kombiniert, so sprechen wir von *Taktiken*. Eine Taktik ist dabei typischerweise die Kombination einiger weniger Entwicklungsschritte und einer zugehörigen Beschreibung, welches Ziel mit dieser Kombination erreicht werden soll.

Können Entwicklungsschritte hierarchisch kombiniert werden, so eignen sich diese auch zur Erreichung größerer Ziele. Damit kann der Entwicklungsvorgang hierarchisch untergliedert werden. Auch Taktiken können kombiniert werden, wobei große Taktiken zweckmäßigerweise als *Strategien* bezeichnet werden. Der Unterschied zwischen Strategie und Taktik ist fließend. Während Taktiken mehr für lokale Probleme, zum Beispiel zur Implementierung einer Klasse, eingesetzt werden, sind Strategien mehr für globale Probleme, zum Beispiel Vervollständigung aller Entwurfsdokumente, zuständig.

Definition 2.11 (Taktik und Strategie)
Eine Taktik *ist die*

- *hierarchische Kombination einiger Entwicklungsschritte und/oder Taktiken und*
- *eine zugehörige Beschreibung, welches Ziel mit dieser Kombination erreicht wird.*

Eine Strategie *ist eine Taktik, die zur Erreichung globaler Ziele eingesetzt wird.* (□)

Durch die Kombination korrekter Entwicklungsschritte bleibt auch die Korrektheit von Taktiken und Strategien erhalten. Eine Kombination kann natürlich nicht nur sequentiell, sondern parallel erfolgen. Das heißt, eine Taktik oder Strategie können mehrere Entwickler zumindest teilweise parallel durchführen. Dazu ist es notwendig, daß die Transformationen des Dokumentgraphen als Transaktionen mit großer Unabhängigkeit durchgeführt werden können und parallele Entwicklungen in verschiedenen Teilen des zu implementierenden Systems möglich sind. Dazu ist es zum Beispiel sinnvoll, daß unabhängige Weiterentwicklung von Dokumenten und deren spätere Integration möglich ist. Dies wird bei der Definition der Dokumentarten dieser Arbeit stark berücksichtigt.

Während eine Methodik noch Taktiken niederer Komplexität beschreiben kann, betrachten wir Strategien nicht als Teil der Methodik. Stattdessen sind Strategien Teil einer Methode.

Methode und Entwicklungsprozeß

Eine *Methode* ist eine Vorschrift zur Entwicklung eines Systems, die über die gesamte Entwicklungszeit angewendet wird. Eine Methode kann damit selbst als Strategie, die auf oberster Stufe angewendet wird, aufgefaßt werden. Eine Methode beschreibt damit eine Menge von möglichen Abläufen von Systementwicklungen. Diese nennen wir *Entwicklungsprozesse*.

Definition 2.12 (Methode und Entwicklungsprozeß)
Eine Methode *ist eine auf oberster Stufe definierte Strategie, die Richtlinien für den gesamten Entwicklungsprozeß vorgibt. Ein* Entwicklungsprozeß *ist eine Ausführung (Instanz) einer Methode. Er besteht aus einer sequentiellen, parallelen, iterierten und alternativen Kombination von Entwicklungsschritten.* (□)

Eine Strategie oberster Stufe wäre etwa sequentielle Bearbeitung der Phasen Analyse, Entwurf und Implementierung durch vier entsprechende Teilstrategien gemäß dem Wasserfallmodell. Völlig andere Kombinationen, die Rückkopplung zwischen verschiedenen Phasen, iterative Systementwicklung oder ähnliches erlauben, sind ebenfalls möglich. Eine Methodik legt daher in keiner Weise fest, wie ihre Komponenten zu einer Methode zusammengesetzt werden sollten. Während die Sammlung von Transformationstechniken in einer Methodik sehr groß sein kann, sollte sich eine Methode auf die Verwendung einer kompakten, in sich geschlossenen Teilmenge von Techniken beschränken, um dem Entwickler eine klare Führung durch den Entwicklungsprozeß zu geben.

Durch die hierarchische Definition von Taktiken und Strategien bis hin zur Methode entsteht ein Projektmodell im Sinne von Denert (siehe [DEN91b]).

Projektmanagement und Werkzeugunterstützung

Neben einer möglichst komfortablen Werkzeugunterstützung für eine Methodik, die typischerweise aus Editoren, Generatoren, Transformatoren und ähnlichen Werkzeugen zur Bearbeitung einzelner oder einer kleinen Menge von Dokumenten besteht, ist ein Werkzeug für Projektmanagement interessant, das Unterstützung bei der Durchführung von Strategien und Taktiken bietet. Dies kann in Form animierter Rezepte (siehe [PRE95]) geschehen.

Ebenso könnte eine auf regulären Ausdrücken basierende Taktiksprache benutzt werden, um parallele und sequentielle Kombinationen sowie optionale Ausführung von Schritten zu beschreiben, wie das im CIP-System ([Betal85]) möglich ist.

Wenn die Kombination von Entwicklungsschritten formalisiert beschrieben wird, dann kann der gesamte Entwicklungsvorgang als eine sequentielle und parallele Kombination von Entwicklungsschritten beschrieben werden. Der *Entwicklungsprozeß* wird damit genau so formal faßbar, wie die Syntax und Semantik der darin benutzten Dokumente. Dies kann verstärkt benutzt werden, um den Entwicklungsprozeß durch adäquate Metriken zu messen und so das Projektmanagement zu unterstützen.

Phaseneinteilung einer Methode

Eine Systementwicklung besteht typischerweise aus den Phasen *Analyse*, *Entwurf* und *Implementierung*, die ihrerseits weiter untergliedert sein können. Werden die verschiedenen Dokumentarten unterschiedlichen Phasen der Systementwicklung zugeordnet, so läßt sich am Dokumentgraphen ablesen, wann eine Phase begonnen hat bzw. beendet ist.

Beispielsweise ist die Implementierungsphase beendet, wenn alle zur Implementierung als Softwarekomponenten gedachten Dokumente, die nicht ausführbar sind, als redundant markiert sind. Damit reicht es für die Implementierung des Systems, die Menge dieser ausführbaren Dokumente einem Compiler zu übergeben.

Die Entwurfsphase kann als beendet betrachtet werden, wenn alle Dokumente der Analysephase als redundant markiert sind. Typischerweise werden aber hier zusätzliche Kriterien etabliert, um eine gewisse Feinheit des Entwurfs zu garantieren, so daß in der Implementierungsphase keine wesentlichen Entwurfsentscheidungen mehr zu treffen sind.

Die Analysephase beschäftigt sich vor allem mit der Erstellung von Dokumenten, die beschreiben, *was* das zu entwickelnde System leisten soll. Demgegenüber wird in der Entwurfsphase vor allem die Frage geklärt, *wie* diese Leistungen erbracht werden sollen. Größere Systeme besitzen jedoch den massiven Nachteil, daß es fast unmöglich ist, eine detaillierte Beschreibung des „Was?“ zu erstellen, ohne bestimmte Details des „Wie?“ vorwegzunehmen, indem etwa Teile der Architektur des Systems bereits beschrieben werden.

Dieser Mangel kann auf zwei Arten behoben werden. Die heute übliche Art ist es, die Beschreibung von Entwurfsdetails während der Analyse zuzulassen, um damit das Verhalten des Systems einfacher zu beschreiben, allerdings ohne diese Details für den Entwurf verbindlich anzusehen. Dies führt entweder dazu, daß während der Entwurfsphase eine Überarbeitung dieser Details stattfindet („Redesign“) oder die vorgegebenen Details doch im Entwurf verwendet werden.

Alternativ dazu muß in pragmatischen Systementwicklungsmethoden das Ergebnis der Analysephase keineswegs eine komplette Spezifikation des Verhaltens eines Systems sein, sondern dieses Verhalten kann auf einer abstrakten Ebene beschrieben werden. Dann werden während der Entwurfsphase weitere Details des Verhaltens festgelegt, also das „Was?“ gemeinsam mit dem „Wie?“ weiter verfeinert.

Dieser zweite Ansatz verwischt die klare Trennung zwischen Analysephase und Entwurfsphase etwas, besitzt aber den Vorteil, daß Eigenschaften von Systemen immer dann am besten beschrieben werden können, wenn die besten zugehörigen Beschreibungstechniken dafür zur Verfügung stehen. Außerdem trägt dieser Ansatz der Tatsache Rechnung, daß während der Analyse eines Systems viele Details des Verhaltens noch nicht bekannt sind, weil sich die Anwender darüber noch gar nicht im klaren sind oder aufgrund der Komplexität des Systems einfach übersehen wurden. Solche Details können in späteren Phasen der Systementwicklung entschieden werden.

Die konzeptuelle Trennung zwischen verschiedenen Phasen der Systementwicklung kann aufrecht erhalten werden, wenn eine zeitliche Verschränkung erlaubt wird. Diese zeitliche Verschränkung der Phasen ist sinnvoll, ja sogar notwendig, um etwa besonders kritische oder wichtige Systemteile in sehr frühen Stadien der Systementwicklung durch Prototyping ([PB93]) gegenüber der informellen Vorstellung der Anforderungen validieren zu können. Mit einer zeitlichen Verschränkung wird auch eine parallele Entwicklung verschiedener Systemteile eines Systems durch verschiedene Arbeitsgruppen erleichtert.

2.5 Zusammenfassung

Dieses Kapitel enthält die Formalisierung der methodischen Grundlage, die für formale Systementwicklungen notwendig ist.

Ein Dokument ist eine Beschreibungseinheit, die bei der Softwareentwicklung entweder direkt erstellt oder durch Transformation aus anderen Dokumenten gewonnen wird. Ein Dokument erhält seine Semantik in Form einer Teilmenge des Systemmodells. Basierend auf dieser Semantikdefinition wurde die Folgerungsbeziehung für Dokumente festgelegt. Ein Dokument ist in einen konstruktiven Anteil und ein Protokoll strukturiert. Der konstruktive Anteil trägt zur Weiterentwicklung eines Systems bei und stellt das Ergebnis eines Entwicklungsschritts dar. Das Protokoll hält den Entwicklungsvorgang fest und beschreibt, in welchem Kontext das Dokument sinnvoll eingesetzt werden kann.

Die Struktur der Systementwicklung wird in einem Dokumentgraphen festgehalten, der neben der Menge der entwickelten Dokumente auch festhält, welches Dokument aus welchen Vorgängern entwickelt wurde und welche Dokumente redundant sind. Ein Entwicklungsschritt ist die Erweiterung des Dokumentgraphen um ein neues Dokument, die Etablierung von Vorgängerbeziehungen und die Markierung von Dokumenten als redundant, wenn sie von anderen Dokumenten gefolgert werden können.

Dann sind die in den Kapiteln 7 und 8 formalisierten Dokumentarten Objektmodell, Klassensignatur und Lebenszyklen vorgestellt worden. Das Objektmodell dient zur Beschreibung der Struktur und Beziehungen zwischen Klassen. Die Klassensignatur dient zur Charakterisierung von Attributen und Methodensignaturen einer Klasse. Sie kann auch eine Beschränkung der maximal ausgebildeten Instanzen festlegen. Lebenszyklen beschreiben das Verhalten von Agenten (Objekten) einer Klasse. Dabei wird zwischen Typautomaten und Klassenautomaten unterschieden. Typautomaten dienen als Schnittstellenbeschreibung einer Klasse und werden vererbt. Klassenautomaten werden zur Implementierungsbeschreibung genutzt und nicht vererbt. Für jede Dokumentart müssen

- eine *Syntax* und eine *Semantik* angegeben werden,
- *hinreichende syntaxbasierte Kriterien* für die Verfeinerungsbeziehung $\models$ zwischen Dokumenten gefunden werden,
- Entwicklungsschritte für die *Neuerstellung*, *Weiterentwicklung* und *Integration* von Dokumenten angegeben werden und
- ein *Redundanztest* für ein Dokument formuliert werden.

Im letzten Abschnitt wurde der Begriff Methodik als Sammlung von Entwicklungsschritten zu einer gegebenen Formalisierung von Dokumenten definiert. Taktiken und Strategien sind Kombinationen von Entwicklungsschritten gemeinsam mit Anwendungsrichtlinien, um vorgegebene Ziele zu erreichen. Diese Kombinationstechnik führt zur Methode als Strategie auf oberster Ebene, die zum Ziel hat, ein vollständiges, lauffähiges System zu entwickeln.

Kapitel 3

Systemmodell

Im vorhergehenden Kapitel wurde gefordert, daß die Semantik jedes Dokuments aus einer Teilmenge der Menge aller Systeme $\mathcal{SM}$ besteht.

In diesem Kapitel wird das *Systemmodell* $\mathcal{SM}$ formal definiert. Es charakterisiert, welchen Systembegriff wir in dieser Arbeit benutzen. Das erlaubt die präzise Definition adäquater Beschreibungstechniken und zugehöriger formaler Semantiken. Das Systemmodell charakterisiert die Menge der möglichen Abläufe eines Systems ebenso wie dessen Struktur und Zustandsraum und die Übergänge in diesem Zustandsraum.

Der erste Abschnitt charakterisiert den Kern des Systemmodells mit Hilfe des zentralen Begriffs *Agent*, der eine autonome, abgegrenzte Einheit des Systems darstellt. Darauf aufbauend wird im zweiten Abschnitt die Verbindung zu den syntaktisch orientierten Beschreibungstechniken gebildet, indem eine Klassifizierung der Agenten des Systemmodells vorgenommen wird. Dabei werden syntaktische Begriffe eingeführt, die Agenten in Klassen gruppieren, sowie ein Typsystem für Werte und Nachrichten bilden. Der dritte Abschnitt enthält eine Zusammenfassung aller Komponenten des Systemmodells. Im vierten Abschnitt wird kurz auf Erweiterungen und Alternativen zum Systemmodell eingegangen.

Die Formalisierung des Systemmodells basiert auf dem im Anhang A.2 dargestellten Modellierungstechnikteil von Focus. Seine Kenntnis ist vor der Lektüre dieses Kapitels empfohlen.

3.1 Kernschicht des Systemmodells

Ein *System* ist eine konkrete Ausprägung einer Kombination aus Hardware- und Softwarekomponenten sowie gegebenenfalls menschlichen Akteuren, die eine bestimmte (in gewissen Grenzen veränderbare) Struktur besitzen und ein bestimmtes Verhalten aufweisen. Ein *Systemablauf* ist ein Ablauf eines Systems in einem bestimmten (auch unendlichen) Zeitintervall, der Eingaben, Ausgaben und Zustände für jeden (relevanten)

Zeitpunkt des Ablaufs beinhaltet. Ein System kann zu gegebenen Eingaben und Anfangszuständen verschiedene Abläufe besitzen. Ein solches System heißt *nichtdeterministisch*.

Das *Systemmodell* ist eine Charakterisierung aller für uns interessanten Systeme. Es charakterisiert sowohl das Verhalten als auch die Struktur eines Systems, läßt aber Freiheitsgrade, die durch die konkrete Systementwicklung spezialisiert werden. Das Systemmodell wird in zwei Schichten definiert. Ein Kern definiert die Abläufe von Systemen und führt den Agentenbegriff ein. In einer darauf aufgesetzten zweiten Schicht wird durch die Einführung von Klassen eine Verbindung zwischen der an Semantik orientierten Welt des Systemmodells und der durch syntaktische Beschreibungen gekennzeichneten Modellierung von Systemen geschaffen. Die Definition des Systemmodells und die Verbindung zu den syntaktischen Konstrukten der Modellierungstechniken kann als die Definition einer formalen Semantik von Begriffen eines „Meta-Modells“, oder wie es auch genannt wird „Begriffsmodells“ ([HL93, CDI92]) verstanden werden. Solche Meta-Modelle beinhalten typischerweise Begriffe, wie Agenten, Klassen und ähnliche mehr. Durch die hier angegebene mathematische Modellierung all dieser Begriffe innerhalb des Systemmodells entsteht eine wesentlich präzisere Definition dieser Begriffe, als es in einem E/R-basierten Meta-Modell möglich ist.

3.1.1 Anforderungen an das Systemmodell

Das Systemmodell dient zur semantischen Fundierung von Techniken zur Beschreibung verteilter objektorientierter Software. Es muß deshalb die in der Objektorientierung benutzten Konzepte mathematisch modellieren, ist aber so allgemein zu halten, daß verschiedene konkrete Ausprägungen für unterschiedliche Systeme möglich sind. Dennoch müssen eine Reihe von Modellierungsentscheidungen getroffen werden, da nur so eine Definition adäquater Beschreibungstechniken möglich ist. Diese Modellierungsentscheidungen werden nun formuliert.

Ein objektorientiertes System besteht aus *Agenten*, die *parallel* interagieren können. Ein Agent ist eine konzeptuelle Einheit, die über einen gekapselten Speicher und einer festgelegten Schnittstelle verfügt. Weil Agenten konzeptuell oder physisch verteilt sein können, ist ein gemeinsamer Speicher für Agenten nicht vorhanden. Stattdessen findet jede Form der Interaktion zwischen Agenten mit Hilfe von *Nachrichtenaustausch* statt. *Nachrichten* sind informationstragende Einheiten, die den Agenten zur Kommunikation dienen. Die Kommunikation ist *asynchron*, das heißt, der Sender darf eine Nachricht versenden, unabhängig von der Empfangsbereitschaft des Empfängers. Daraus ergibt sich die Forderung nach einem *Puffersystem* für Nachrichten.

Konzeptuell sind Agenten stark verwandt mit Objekten objektorientierter Systeme. Der gewählte Name *Agent* drückt gegenüber dem eher passiven Objekt eigenständige Aktivität der Komponenten im Systemmodell aus.

Objektorientierte Systeme besitzen stark dynamische Aspekte. So werden zum einen Kommunikationswege, sogenannte *Kanäle* zwischen Agenten dynamisch geändert und zum anderen dynamisch Agenten erzeugt und zerstört. Kommunikationswege zwischen

Agenten werden mittels *Identifikatoren* ermöglicht, die den Empfänger von Nachrichten identifizieren. Ein Kanal besteht also genau dann, wenn der Sender den Identifikator des Empfängers besitzt und damit den Empfänger kennt. Die Menge der möglichen Kommunikationswege ist damit durch eine Kenntnisbeziehung zwischen den Agenten bestimmt.

Das Systemmodell muß auch die dynamische Erzeugung von Agenten zulassen. Die Zerstörung von Agenten kann, wie in objektorientierten Systemen oft verwendet, einem Garbage Collector überlassen werden. Es ist daher nicht notwendig, die Zerstörung von Agenten im Systemmodell zu modellieren.

Um diese dynamischen Aspekte zu modellieren, besitzt jeder Agent einen bereits erwähnten, eindeutigen Identifikator. Der Identifikator eines Agenten ist unveränderlich und ist diesem Agenten bekannt. Das heißt, ein Agent kann seinen eigenen Identifikator versenden und sich damit anderen Agenten bekannt machen.

Für die konkrete Beschreibung von objektorientierten Systemen ist es günstig, die unbeschränkte Menge der Agenten in einem endlichen System zu klassifizieren. Die so eingeführten *Klassen* können in einer *Klassenhierarchie* angeordnet werden.

In verteilten Systemen spielen oft Echtzeitaspekte eine größere Rolle. Deswegen werden das Systemmodell und seine Komponenten in gezeiteter Form beschrieben.

3.1.2 Black-Box-Sicht von Agenten

Jeder im System vorkommende Agent besitzt einen ihn eindeutig identifizierenden *Identifikator* aus der abzählbaren Menge *ID*. Die Interaktion eines Agenten mit seiner Umgebung findet nur mit Hilfe von asynchronem Nachrichtenaustausch statt. Weil das Systemmodell gezeitet modelliert wird, eignen sich gezeitete stromverarbeitende Funktionen zur Modellierung von Agentenverhalten. Weil die Nachrichten mit Identifikatoren adressiert werden, ist ein Eingabe- und ein Ausgabekanal für jeden Agenten ausreichend. Zu jedem Identifikator a werden dafür zwei Kanalnamen a_i und a_o verwendet. Diese Kanalnamen sind in den Mengen ID_i der Eingabekanalnamen und ID_o der Ausgabekanalnamen zusammengefaßt. Beide Mengen sind disjunkt. Für $a \neq b$ ist $a_i \neq b_i$ und $a_o \neq b_o$. Zu jedem Kanalnamen b gibt es eine Menge von Nachrichten $Msg.b$, die auf diesem Kanal fließen können[1]. Ein Agent erhält damit die in 3.1 graphisch veranschaulichte Black-Box-Sicht.

Definition 3.1 (Agent)
Ein Agent besitzt einen eindeutigen Identifikator $a \in ID$. Sein Verhalten wird in Form einer nichtleeren Menge stromverarbeitender Funktionen mit den Eingabenachrichten $Msg(a_i)$ und den Ausgabenachrichten $Msg(a_o)$ beschrieben:

$$behavior_a \subseteq \{a_i\}^{\overline{\Sigma}} \overset{p}{\to} \{a_o\}^{\overline{\Sigma}} \qquad (\square)$$

[1] Die Historie eines Kanals b wird nach Definition A.9 als Abbildung $\{b\} \to (Msg.b)^{\overline{\infty}}$ oder in der Kurzform $\{b\}^{\overline{\Sigma}}$ dargestellt. Sie beschreibt eine unendliche Beobachtung in Form eines gezeiteten Stroms von Nachrichten auf Kanal b.

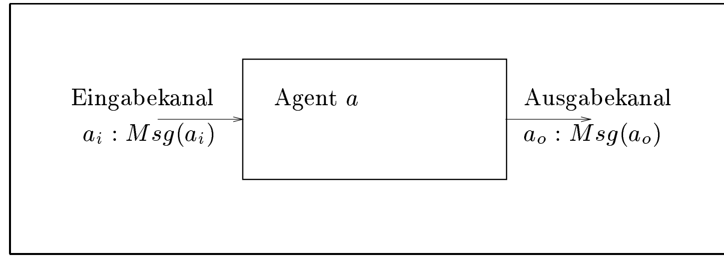


Abbildung 3.1: Ein Agent

Die Mengen der Eingabenachrichten $Msg(a_i)$ und der Ausgabenachrichten $Msg(a_o)$ sind für jeden Agenten adäquat festzulegen. Die Gesamtmenge der Nachrichten *MSG* setzt sich damit wie folgt zusammen:

$$MSG = \bigcup_{a \in ID} Msg(a_i) = \bigcup_{a \in ID} Msg(a_o)$$

Die Funktion *behavior* wird auf Mengen von Agenten erweitert: Für jede Teilmenge $T \subseteq ID$ bezeichnet *behavior(T)* die Parallelkomposition der in T enthaltenen Agenten. Diese bilden wieder eine Menge gezeiteter stromverarbeitender Funktionen:

$$behavior(T) \subseteq T_i^{\overline{\Sigma}} \xrightarrow{p} T_o^{\overline{\Sigma}}$$
$$behavior(T) = ||_{a \in T}\ behavior_a$$

Die Anzahl der Agenten, die während eines Systemablaufs aktiv sein können, ist in objektorientierten Systemen unbeschränkt, da jederzeit neue Agenten aktiviert werden können. Zu jedem Zeitpunkt sind jedoch nur endlich viele Agenten *aktiv*. Zu Beginn eines Systemablaufs ist ein initiales Netz aus endlich vielen Agenten bereits aktiv und alle anderen Agenten inaktiv. Diese Inaktivität nennen wir *initiale Ruhe*. In Abschnitt 3.2.5 werden wir die Aktivierung von Agenten noch genauer behandeln.

Definition 3.2 (Initiale Ruhe für Agenten)
Die endliche Menge StartID $\subseteq$ ID beschreibt die zu Beginn des Systemablaufs aktiven Agenten. Alle anderen Agenten sind initial ruhig. *Ein initial ruhiger Agent wird erst aktiv, wenn er eine erste Nachricht empfangen hat:*

$$\forall a \in ID \setminus StartID.\ \forall f \in behavior_a.\ f(\surd^\infty) = \surd^\infty \qquad (\Box)$$

In vielen objektorientierten Systemen existiert zu Beginn eines Ablaufs genau ein Agent, von dem aus alle weiteren Agenten aktiviert werden. Weil Agenten durch Nachrichten aktiviert werden, aber endlich viele aktive Agenten in jeder Zeiteinheit nur endlich viele Nachrichten versenden können, bleibt die Anzahl der aktiven Agenten zu jedem Zeitpunkt endlich.

3.1.3 Das Kommunikationsmedium

Die parallele Komposition aller Agenten führt zu einer Verhaltensbeschreibung der Form *behavior(ID)*, bei der jedoch keine Verbindung der Kommunikationswege dieser Agenten

existiert. Die Verbindung dieser Kommunikationswege wird mit Hilfe des *Kommunikationsmediums* beschrieben. Das Kommunikationsmedium wirkt als eine Art „Äther“, in dem die Agenten feste Plätze einnehmen und die Nachrichten zwischen den Agenten transportiert werden. Dabei wirken die Kanäle der Agenten gleichsam als „Antennen“. Diese Sichtweise ist in Abbildung 3.2 festgehalten.

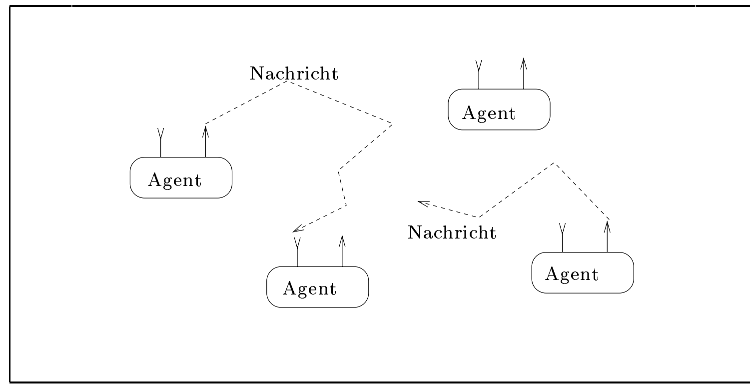


Abbildung 3.2: Das System als in „Äther“ eingebettete Agenten

Das Kommunikationsmedium erfüllt folgende Eigenschaften:

- Nachrichten werden nicht dupliziert oder verloren.
- Nachrichten werden nicht modifiziert.
- Es werden keine Nachrichten erzeugt.
- Die Reihenfolge der Nachrichten zwischen demselben Sender und Empfänger bleibt erhalten.
- Nachrichten werden ohne Verzögerung vom Sender zum Empfänger transportiert.
- Der Empfänger einer Nachricht hängt nur von der Nachricht selbst ab.

Obige Eigenschaften erlauben eine einfache Modellierung des Kommunikationsmediums. Es benötigt keinen eigenen Zustand und muß keine Mechanismen für die Aktivierung von Agenten zur Verfügung stellen. Das Verhalten komplexerer Kommunikationsmedien, die etwa Nachrichten verlieren, verstümmeln oder verzögern, kann aber leicht durch Agenten nachgebildet werden.

Der letzte oben genannte Punkt garantiert die Existenz einer Funktion *destination*, die zu jeder Nachricht den Empfänger ermittelt.

Definition 3.3 (Funktion *destination*)
Zu jeder Nachricht $m \in MSG$ bestimmt die Funktion destination den Empfänger:

$$destination : MSG \rightarrow ID$$

(□)

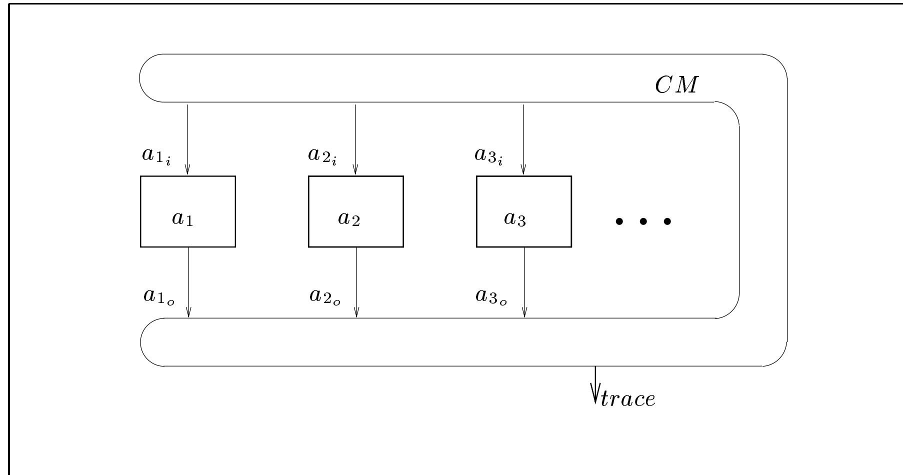


Abbildung 3.3: komponentenorientierte Sicht des geschlossenen Systems

Das Kommunikationsmedium kann ebenso als Komponente des Systems spezifiziert werden wie alle Agenten. Eine komponentenorientierte Sichtweise des Kommunikationsmediums zusammen mit den Agenten ist in Abbildung 3.3 dargestellt.

Das Kommunikationsmedium besitzt dieselben Kanäle wie die Agenten, jedoch in umgekehrter Orientierung. Ein zusätzlicher Ausgabekanal *trace* des Kommunikationsmediums dient einem (fiktiven) Beobachter zur Betrachtung der Interaktionen, die in einem System stattfinden, hat aber keine Auswirkung auf das Systemverhalten selbst. Auf diesem Ausgabekanal werden alle Nachrichten gebündelt, die im System fließen.

Zur Spezifikation des Kommunikationsmediums *CM* wird dieses gemäß Abbildung 3.4 selbst in drei Komponenten zerlegt. Die Komponente *Merge* sammelt alle von Agenten versendeten Nachrichten in einem großen Strom. Die Komponente *Copy* kopiert ihren Eingabestrom in zwei Ausgabeströme. Die Kopie auf Kanal *trace* wird vom Kommunikationsmedium als virtuelle Beobachtung nach außen weitergegeben. Der nachgeschaltete Verteiler *Distribute* verteilt alle Nachrichten an die richtigen Empfänger. Wir erhalten damit den in Abbildung 3.4 dargestellten Aufbau des Kommunikationsmediums.

Die Komponente *Merge* besitzt folgende Funktionalität

$$Merge \subseteq ID_o^{\overline{\Sigma}} \xrightarrow{wp} \{p\}^{\overline{\Sigma}}.$$

Sie sammelt unter Einhaltung obiger Anforderungen, alle ankommenden Nachrichten und gibt diese ohne Verzögerung (Delay) am Ausgabekanal p wieder aus. Werden in einer Zeiteinheit auf unendlich vielen Eingabekanälen Nachrichten empfangen, so ist eine Erfüllung dieser Aufgabe nicht mehr verzögerungsfrei möglich. In unserer Modellierung kann jedoch dieser Fall nicht auftreten, da die initiale Ruhe fast aller Agenten des Systems sichert, daß zu jedem Zeitpunkt nur endlich viele Agenten aktiv sind und daher nur endlich viele Nachrichten gesendet werden. In Abbildung 5.8 ist eine Merge-Komponente beschrieben, die unsere Anforderungen bis auf die Verzögerungsfreiheit erfüllt. Wir verzichten daher hier auf eine Charakterisierung der Merge-Komponente.

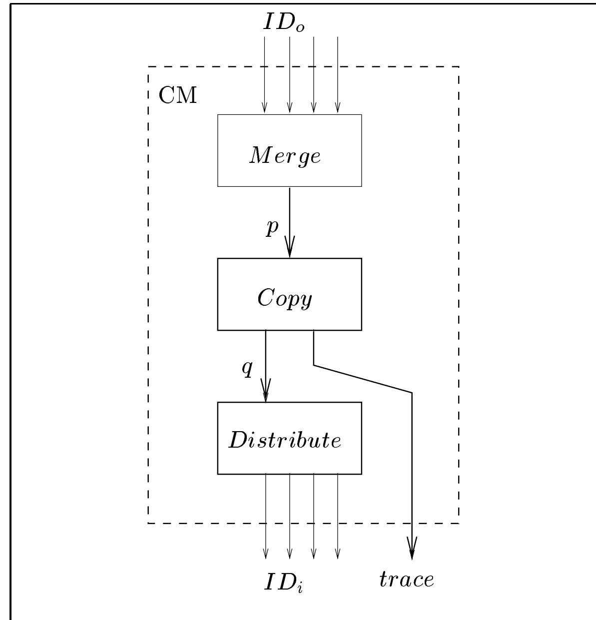


Abbildung 3.4: Aufbau der Spezifikation von *CM*

Die Komponente *Copy* erzeugt zwei Kopien ihres Eingabestroms:

$$Copy \in \{p\}^{\overline{\Sigma}} \xrightarrow{wp} \{q,trace\}^{\overline{\Sigma}}$$
$$Copy(p \mapsto T) \stackrel{def}{=} (q,trace \mapsto T)$$

Die Komponente *Distribute* wird definiert durch[2]:

$$Distribute \subseteq \{q\}^{\overline{\Sigma}} \xrightarrow{wp} {ID_i}^{\overline{\Sigma}}$$
$$Distribute(q \mapsto T).a_i \stackrel{def}{=} \{m,\surd | destination(m)=a\} \copyright\ T$$

Die Verteilung der Nachrichten auf die Kanäle der Empfänger erfolgt unter Benutzung der Funktion *destination*, die den Empfänger jeder Nachricht ermittelt.

Die Komponenten *Merge*, *Copy* und *Distribute* werden gemäß Abbildung 3.4 zusammengesetzt:

Definition 3.4 (Kommunikationsmedium)
Das Kommunikationsmedium wird spezifiziert durch die Komposition der Komponenten Merge, Copy und Distribute:

$$CM \subseteq {ID_o}^{\overline{\Sigma}} \xrightarrow{wp} (ID_i \cup \{trace\})^{\overline{\Sigma}}$$
$$CM \stackrel{def}{=} \mu_{\{p,q\}}(Merge \parallel Copy \parallel Distribute)$$

($\Box$)

Die Konstruktion von *CM* ist wohldefiniert, da bei der Konstruktion nur sequentielle Komposition verwendet wird und keine Rückkopplungsschleife auftritt.

[2] Der Operator $M \copyright s$ filtert den Strom s nach Elementen der Menge M. Siehe auch Anhang A.2.

Das Kommunikationsmedium ist leicht zu implementieren. Es entspricht einem Dienst des Betriebssystems, der die Kommunikation der von ihm verwalteten Prozesse übernimmt. Bei einer echten Verteilung der Agenten ist im allgemeinen eine Verzögerung von Nachrichten beim Transport nicht zu vermeiden. Insofern abstrahiert unser Kommunikationsmedium von dieser Verzögerung, die durch interne Puffer für Ein- oder Ausgabenachrichten in Agenten nachgebildet werden kann.

3.1.4 Systemablauf

Das Kommunikationsmedium kann nun mit den Agenten komponiert werden. Dabei entsteht eine Sicht des Gesamtsystems, die die Menge der Systemabläufe charakterisiert.

Definition 3.5 (Systembeobachtung)
Das Verhalten des Gesamtsystems entsteht gemäß Abbildung 3.3 als:

$$EventTrace \stackrel{def}{=} \mu_{ID_i \cup ID_o}(behavior(ID) \parallel CM)$$

Die Elemente dieser Menge sind von der Sorte $\{trace\}^{\overline{\Sigma}}$ *und können daher als* Systembeobachtungen *verstanden werden.* (□)

Die Komposition von *EventTrace* ist wohldefiniert und eindeutig, da in jeder Rückkopplungsschleife eine (stark) gezeitete Funktion auftritt. Das komponierte Gesamtsystem besitzt keinen Eingabekanal mehr und nur einen Ausgabekanal. Es ist also ein geschlossenes System. Sein Verhalten ist eine reine Spurbeschreibung und kann daher als Menge von Systembeobachtungen verstanden werden. Eine Systembeobachtung enthält die Interaktionen (Nachrichten) aller Agenten des Systems in einer linearen Reihenfolge, wobei nach Definition des Operators *Merge* die zeitliche Reihenfolge aufeinanderfolgender Nachrichten vom selben Sender eingehalten wird und keine Verzögerung dieser Spurbeobachtung auftritt.

3.1.5 State-Box-Sicht von Agenten

Bisher wurde das Verhalten eines Agenten charakterisiert. Jetzt wird eine zustandsbasierte Sicht von Agenten definiert, die über ein Transitionsdiagramm Zustand und Verhalten eines Agenten in Beziehung setzt. Wie definieren daher eine Zustandsübergangsmaschine für Agenten, um deren mögliches Verhalten in Abhängigkeit des Agentenzustands zu beschreiben.

Die Menge der Zustände, die ein Agent $a \in ID$ während eines Systemablaufs einnehmen kann, wird mit $states_a$ bezeichnet. Die Menge $states_a$ ist Teilmenge der zunächst nicht genauer beschriebenen Menge von Zuständen $STATE$. Initialzustände des Agenten werden jeweils mit einer initialen Ausgabe versehen, die von keiner Transition abhängt. Sie sind von der Form $\emptyset \neq inits_a \subseteq states_a \times Msg(a_o)^*$. Zu Beginn jedes Systemablaufs wird ein Initialzustand und eine initiale Ausgabe ausgewählt. Zunächst wird die initiale Ausgabe emittiert und dann der Initialzustand eingenommen.

Das Zustandsübergangsverhalten eines Agenten $a \in ID$ wird durch eine nichtdeterministische Transitionsrelation std_a beschrieben. $(s,in,t,out) \in std_a$ charakterisiert, welche Ausgabe *out* und welcher Zielzustand t abhängig vom aktuellen Zustand s und der Eingabe *in* in der nächsten Zeiteinheit möglich ist.

Definition 3.6 (State-Box-Sicht eines Agenten)
Sei STATE die Menge von Zuständen, die Agenten einnehmen können. Der durch den Identifikator $a \in ID$ identifizierte Agent hat folgende State-Box-Sicht:

- *eine Menge möglicher Zustände $states_a \subseteq STATE$,*
- *eine nichtleere Menge von Initialelementen $inits_a \subseteq states_a \times Msg(a_o)^*$ und*
- *eine Zustandsübergangsrelation $std_a \subseteq states_a \times Msg(a_i)^* \times states_a \times Msg(a_o)^*$.*

(□)

Die zustandsbasierte Sicht eines Agenten kann unter Benutzung der Taktautomaten aus Definition 5.32 mit der in Definition 3.1 angegebenen Verhaltenssicht eines Agenten verbunden werden. Dabei werden die Funktionen *behavior* festgelegt, indem jedem Agenten ein Taktautomat zugeordnet wird:

Definition 3.7 (Charakterisierung von *behavior*)
Jedem Agenten $a \in ID$ ist eine State-Box-Sicht in Form eines Taktautomaten TA_a zugeordnet:

$$TA_a \stackrel{def}{=} (states_a, \{a_i\}, \{a_o\}, std_a, inits_a)$$

Die das Verhalten eines Agenten charakterisierende Funktionsmenge $behavior_a$ wird mittels der Semantik der Taktautomaten 5.33 aus der State-Box-Sicht eines Agenten gewonnen:

$$behavior_a = [\![TA_a]\!]^{ta}$$

(□)

Die Zustandsübergangsfunktion std_a wird in Definition 3.15 gemeinsam mit den Zustandsmengen präzisiert.

3.2 Klassen im Systemmodell

In einer zweiten Schicht wird nun eine Verbindung zwischen dem Kern des Systemmodells und den Beschreibungstechniken definiert. Dazu werden syntaktische Begriffe, wie Typ und Klasse, eingeführt. Dies ist legitim, da im allgemeinen eine direkte Verbindung zwischen den in einem System vorkommenden Werten und deren Sortendefinitionen besteht. Bevor diese Begriffe festgelegt werden, wird auch hier zunächst analysiert, welche Anforderungen diese zweite Schicht erfüllen muß.

3.2.1 Anforderungen an die Klassifizierung

Nachdem eine Beschreibung für Struktur und Verhalten jedes Agenten individuell definiert wurde, wenden wir uns nun der Klassifizierung der Agenten zu. Eine Einteilung in ein endliches System von Klassen bietet die Möglichkeit, die potentiell unbeschränkte Menge von Agenten mit endlich vielen Dokumenten zu beschreiben. Klassen bilden damit eine Brücke zwischen den Beschreibungstechniken, die klassenbasiert sind, und dem Systemmodell, das agentenbasiert ist.

Klassen besitzen in objektorientierten Systemen darüberhinaus weitere Eigenschaften, von denen wir hier einige diskutieren wollen:

1. Klassen werden benutzt, um gleichartige Agenten zu klassifizieren. Das bedeutet jeder Agent eines Systems wird einer Klasse zugeordnet.
2. Klassen bilden ein Typsystem für Agenten. Weil Klassen in einer Klassenhierarchie angeordnet werden, entsteht so eine Subtyphierarchie für die Agenten der Klassen. Agenten können damit zum Typ mehrerer Klassen gehören.
3. Eine Klasse bestimmt die Funktionalität ihrer Agenten. Entlang der Subklassenhierarchie wird eine Erweiterung der Signatur gefordert.
4. Eine Klasse charakterisiert das Verhalten ihrer Agenten. Das Verhalten von Agenten, die zum Typ einer Klasse gehören, kann von dieser Klasse festgelegt werden. Es wird ebenfalls entlang der Subklassenhierarchie vererbt.
5. Eine Klasse definiert die Implementierung ihrer Agenten, da Klassen als Quelltexteinheiten programmiert werden.
6. Eine Klasse definiert einen Typ für die Identifikatoren ihrer Agenten. Die Menge der Identifikatoren wird dadurch ebenfalls typisiert und in einer zur Subklassenhierarchie analogen Subtyphierarchie strukturiert.
7. Eine Klasse speichert gemeinsame Daten aller zu ihr gehörenden Objekte.
8. Eine Klasse wird als die Menge der aktuell zu ihr gehörenden Agenten verwendet. Solche Klassenobjekte erlauben (vor allem im Datenbankbereich) eine schematische Bearbeitung von Mengen von Agenten.

Nicht alle oben genannten Eigenschaften von Klassen werden in dieser Arbeit formalisiert. So werden die letzten beiden Punkte nicht berücksichtigt. Die Speicherung gemeinsamer Daten in Form von *Klassenvariablen* ist speziell in der Programmiersprache *C++* ([STR91a]) vorhanden und widerspricht dem Kapselungsprinzip von Objekten, während *Klassenobjekte* vor allem in Sprachen für Datenbanken wie *Troll* ([Jetal91]) und *Troll–Light* ([CGH92]) Anwendung finden.

In nahezu jeder objektorientierten Sprache und Methodik werden Klassen zur Gruppierung von Agenten benutzt. Einzige Ausnahmen bilden nur die klassenlosen Ansätze, wie *Self* ([Uetal91]), die stattdessen basierend auf *Prototypen* von Agenten arbeiten. Wesentlich deutlicher unterscheiden sich Ansätze bereits bei der Frage, ob ein Klassensystem auch typisiert. Der Sinn einer Typisierung besteht vor allem in zwei Punkten:

- Typsysteme erlauben Definition und Strukturierung von Wertemengen und
- Typsysteme erlauben die statische Überprüfung von Programmeigenschaften.

Vor allem die statische, bereits während der Übersetzungszeit stattfindende Überprüfung von Typfehlern von stark typisierten Programmiersprachen erlaubt die frühzeitige Erkennung von Programmierfehlern, die sonst erst zur Laufzeit auftreten würden. Vertreter stark typisierter Programmiersprachen, wie etwa *C++* oder *Eiffel* ([MEY92]), verhindern damit das Auftreten von Laufzeitfehlern der Form *message not understood*, wie sie in *Smalltalk* ([BUD87]) üblich sind. In objektorientierten Software-Engineering-Methoden wird neben der Signaturkorrektheit oft auch eine Verhaltensverfeinerung der Agenten entlang der Subklassenhierarchie gefordert. Verhaltensverfeinerung ist jedoch in den heute bekannteren Methoden ([RUM91, BOO94, Cetal94, JAC93]) nur informell formuliert. Syntaktische Kriterien zu ihrer Überprüfung gibt es nicht. Genau hier werden wir den in Kapitel 6 entworfenen Verfeinerungskalkül einsetzen, um eine weit über die Signatur hinausgehende Vorstellung von Vererbung von Verhalten zu definieren. Für uns hat also Vererbung folgende Aufgaben (vgl. z.B. [Getal94]):

1. Die Vererbungshierarchie strukturiert die Menge der Klassen.
2. Die Vererbung wird eingesetzt zur *Spezialisierung*: Eine Subklasse ist eine Spezialisierung all ihrer Superklassen. Das hat folgende zwei konkrete Ausprägungen:
3. Die Schnittstelle einer Subklasse (Menge der verstandenen Nachrichten) umfaßt die Schnittstelle der Superklasse und
4. das den Klienten bekannte Verhalten, beschrieben in Form einer Menge stromverarbeitender Funktionen, ist enthalten im Verhalten der Superklasse. Dadurch wird das *Substitutionsprinzip* anwendbar.
5. Vererbung wird außerdem benutzt zur Wiederverwendung von Implementierungsteilen aus der Superklasse. Das hat folgende zwei konkrete Auswirkungen:
6. Die Attributsignatur einer Subklasse umfaßt die Attributsignatur der Superklasse und
7. die Implementierung einer Methode wird, sofern sie nicht redefiniert wird, in der Subklasse wiederverwendet.
8. Die Vererbungshierarchie spiegelt sich außerdem in einer Subtyphierarchie der Identifikatoren wider.

In Programmiersprachen werden Klassen auch als syntaktische Einheiten verwendet, um so die Möglichkeit zu besitzen, Algorithmen und Datenstrukturen ihrer Instanzen, den Agenten zu implementieren. Ein bei Programmiersprachen, wie *C++*, *Modula−3* ([HAR92]) oder *Oberon−2* ([MO93]) wichtiges Konzept zur starken Typisierung ist die Strukturierung der Identifikatoren von Agenten in einer zur Klassenhierarchie analogen Subtyphierarchie. Identifikatoren erhalten, gerade in *C++* sehr anschaulich, eine Existenz als eigenständige Werte mit einer eingeschränkten Signatur und der zusätzlichen Aufgabe beim Austausch von Nachrichten den Empfänger eindeutig zu identifizieren. Auch dies soll im Systemmodell reflektiert werden. In objektorientierten Ansätzen fehlt häufig der vierte Punkt, der fordert, daß Vererbung auch zur Verhaltensspezialisierung eingesetzt wird.

3.2.2 Klassen und Vererbung

Wir modellieren nun die diskutierten Eigenschaften von Klassen und Vererbung in unserem Systemmodell.

Definition 3.8 (Klassifizierung von Agenten)
Sei CN eine endliche Menge von Klassennamen, dann wird jedem Agenten seine Klasse *zugeordnet:*

$$class : ID \rightarrow CN$$

Die Menge der Klassen ist in einer Klassenhierarchie *strukturiert. Diese wird durch die zweistellige, partielle Ordnung*

$$.\sqsubseteq. \subseteq CN \times CN$$

dargestellt. Gilt $c \sqsubseteq d$, *dann heißt c auch* Subklasse *und d* Superklasse. *Die Agenten, die einer Klasse zugeordnet sind, heißen* Instanzen *dieser Klasse.* (□)

Die Ordnung $.\sqsubseteq.$ ist reflexiv, transitiv und antisymmetrisch, weshalb jede Klasse ihre eigene Sub- und Superklasse ist. Alle weiteren Sub- und Superklassen werden auch als *echt* bezeichnet.

Agenten werden gemäß der Klassenhierarchie typisiert.

Definition 3.9 (Typisierung von Agenten)
Zur gegebenen Klassifizierung von Agenten wird folgende Typisierung *mittels dem Prädikat isof definiert:*

$$.isof. \subseteq ID \times CN$$
$$a \; isof \; c \Leftrightarrow class(a) \sqsubseteq c$$

Die Agenten, deren Typ eine Klasse c ist, heißen Elemente *des Typs der Klasse, oder kurz* Klassenelemente. (□)

Damit ist ein Agent Element des Typs seiner Klasse und aller seiner Superklassen. Jede Instanz einer Klasse ist auch ein Element derselben, umgekehrt jedoch ist nicht jedes Element eines Klassentyps auch eine Instanz der Klasse.

Die Wirkung von Verhaltens- und Strukturbeschreibungen bezieht sich, mit Ausnahme der Implementierungsbeschreibung, immer auf die Klasse als Typ und damit auf die Elemente der Klasse. Das bedeutet umgekehrt, daß ein Agent die Beschreibungen aller seiner Klassentypen erfüllt. Dies bedeutet einerseits, daß Modellierer von Superklassen, die in verschiedenen Kontexten wiederverwendet werden sollen, eine gute Modellierung vornehmen müssen, und andererseits, daß sich der Programmierer einer Subklasse an die Vorgaben aller Superklassen halten muß. Diese Problematik drückt sich vor allem in der Modellierung und Programmierung objektorientierter Application Frameworks ([WGM89]) aus, die eine sehr sorgfältige Arbeit und dennoch häufige Redefinition und Restrukturierung erfordern. Aufgrund der formalen Semantik, die wir unseren Beschreibungstechniken auf Basis des Systemmodells geben, müssen wir dieser Problematik besondere Aufmerksamkeit schenken. Beschreibungstechniken müssen in einer losen, erweiterbaren Weise zur Spezifikation von Struktur und Verhalten verwendet werden können.

3.2.3 Werte, Typen und Variablen

Zur Beschreibung der Struktur und des Verhaltens einer Klasse ist es notwendig, ein Universum von Werten, einen Typ- und einen Variablenbegriff einzuführen. Für weitere Aspekte der Typisierung des Werteuniversums verweisen wir auf Arbeiten im Bereich funktionaler Programmiersprachen, wie *Gofer* ([JON93a]) oder *Haskell* ([HJW92]), und Spezifikationssprachen, wie *Spectrum* ([Betal93b]) oder *OBJ* ([Getal92]). Letztere besitzt sogar ein Typsystem mit implizitem Subtyping, wie es in moderneren objektorientierten Programmiersprachen ebenfalls üblich ist. Für unsere Vorstellung reicht es, ein Typsystem, wie das in *Gofer* zur Verfügung stehende, zu verwenden.

Nachfolgend werden einige semantische und syntaktische Begriffe sowie deren Beziehungen als Teil des Systemmodells definiert. Diese Kombination semantischer und syntaktischer Mittel im Systemmodell ist legitim, da sich syntaktische Mittel wie Typdefinitionen in jeder Systemimplementierung in kodierter Form wiederfinden. Deshalb werden beide Ebenen auch bei anderen Semantikdefinitionen, zum Beispiel bei Termalgebren algebraischer Sprachen ([EM85]) verwendet.

Definition 3.10 (Werte, Typen und Variablen)
Das Werteuniversum VAL ist eine Menge von Werten, *die auch alle Identifikatoren beinhaltet:*

$$ID \subseteq VAL$$

Es sei $\langle sort \rangle$ eine Menge von Typen. *Jedem Typ wird seine Wertemenge zugeordnet:*

$$values : \langle sort \rangle \to \mathbb{P}(VAL)$$

Jeder Klassenname wird als Typ der ihm zugeordneten Identifikatoren aufgefaßt:

$$CN \subseteq \langle sort \rangle$$
$$\forall c \in CN.\ values(c) = \{\ id \in ID \mid id\ isof\ c\ \}$$

Weiterhin sei $\langle var \rangle$ eine Menge von Variablennamen. *Jedem Variablennamen wird sein Typ zugeordnet:*

$$type : \langle var \rangle \to \langle sort \rangle$$

Für jede Menge von Variablen $v \subseteq \langle var \rangle$ ist die Menge der Variablenbelegungen *Bel_v gegeben durch alle Abbildungen, die Variablennamen aus v einen Wert zuweisen und die Typisierung der Variablen beachten:*

$$Bel_v \subseteq v \to VAL$$
$$\forall \beta \in Bel_v, a \in v.\ \beta(a) \in values(type(a))$$

Die Menge aller Variablenbelegungen *BEL ist:*

$$BEL \stackrel{def}{=} \bigcup_{v \subseteq \langle var \rangle} Bel_v \ \subseteq \langle var \rangle \rightharpoonup VAL \qquad (\Box)$$

Die Menge der Belegungen $\beta \in BEL$ charakterisiert auch die Belegung von Attributen und Aufrufparametern, die spezielle Arten von Variablen darstellen. Zur Vereinfachung der Modellierung nehmen wir an, daß jeder Variablenname die Menge der möglichen Werte dieser Variablen charakterisiert.

Weil Agenten zum Versenden von Nachrichten den Identifikator des Empfängers nutzen, ist es wesentlich, daß die Menge der Identifikatoren *ID* Teil des Werteuniversums *VAL* ist. Im Gegensatz dazu sind Agenten, bzw. Agentenzustände selbst keine Elemente des Werteuniversums. Auf diese Weise wird eine Rekursion bei der Definition des Werteuniversums vermieden, die das resultierende Typsystem stark komplizieren würde ([LW93, CAR93, GRO95, PS92]).

3.2.4 Attribute und Methoden

Die Signatur einer Methode besteht neben dem Methodennamen auch aus deren Argumenten, also einer Belegung der Parameter. Vereinfachend nehmen wir auch hier an, daß mit dem Methodennamen bereits eindeutig der Typ einer Methode festliegt.

Definition 3.11 (Methode, Nachricht)
Es sei $\langle meth \rangle$ *eine Menge von* Methodennamen. *Jedem Methodennamen wird sein* Methodentyp *in Form einer endlichen Menge von Variablennamen, die seine Argumente bestimmen, zugeordnet:*

$$args : \langle meth \rangle \to \mathbb{P}^{fin}(\langle var \rangle)$$

Die Menge der Nachrichten *eines Systems setzt sich aus dem Empfänger a, dem Methodennamen mn und der Belegung der formalen Parameter* β *zusammen:*

$$MSG \stackrel{def}{=} \{ \ (a,mn,\beta) \mid a{\in}ID \wedge mn{\in}\langle meth \rangle \wedge \beta{\in}BEL \wedge \bullet\beta{=}args(mn) \ \}$$

Die in 3.3 definierte Funktion destination kann nun festgelegt werden:

$$destination(a,mn,\beta) \stackrel{def}{=} a \qquad (\Box)$$

Jede Nachricht des Systems enthält eine Empfängeradresse. Wie in objektorientierten Systemen üblich, wird eine Senderadresse nicht mit der Nachricht übergeben. Die Menge *MSG* ist eine Teilmenge des Produkts $ID{\times}\langle meth \rangle{\times}BEL$. Wir werden diese Nachrichten auch in objektorientierter Schreibweise $a.mn(\beta)$ notieren. Die Menge an Nachrichten, die die Objekte einer Klasse verstehen, und die Attribute, die eine Klasse besitzt, können nun in Form der Klassensignatur angegeben werden:

Definition 3.12 (Klassensignatur)
Die Funktion Σ_{attr} *ordnet jeder Klasse eine endliche Menge von* Attributen *zu. Ein Attribut wird bezeichnet durch seinen Namen:*

$$\Sigma_{attr} : CN \to \mathbb{P}^{fin}(\langle var \rangle)$$

Die Funktion Σ_{meth} *ordnet jeder Klasse eine endliche Menge von* Methoden *zu:*

$$\Sigma_{meth} : CN \to \mathbb{P}^{fin}(\langle meth \rangle)$$

Die Klasse $c{\in}CN$ *hat damit die Signatur* $(\Sigma_{attr}(c),\Sigma_{meth}(c))$. *Klassensignaturen sind in bezug auf die Klassenhierarchie geordnet. Es gilt:*

$$\forall c,d{\in}CN.\ c{\sqsubseteq}d \Rightarrow \Sigma_{attr}(d){\subseteq}\Sigma_{attr}(c) \wedge \Sigma_{meth}(d){\subseteq}\Sigma_{meth}(c) \qquad (\Box)$$

Durch die Festlegung, daß jede Variable und jede Methode einen zugeordneten Typ besitzt, ist mit obiger Definition der Klassensignatur die Datenstruktur, die in einer Klasse definiert wird, sowie die Schnittstelle einer Klasse festgelegt. Die Attribute einer Klasse gehören nicht zur Schnittstelle, sie sind nur innerhalb eines Agenten zugänglich. Weitere Zugriffsniveaus, wie sie etwa *C++* in Form von *public*, *protected* oder *private* anbietet, werden hier nicht formalisiert, obwohl sie interessante Möglichkeiten der Kapselung von Objektdaten bzw. der Vergabe von Zugriffsrechten bieten. Diese lassen sich jedoch auf die Strukturierung von Namensräumen im Quelltext einer Programmiersprache zurückführen (siehe [RUM95]), so daß eine Modellierung im semantischen Systemmodell nicht notwendig erscheint.

Um die Typkorrektheit von Objekten zu sichern, darf ein Methodenaufruf, der von einem Objekt der Superklasse verstanden wird, von einem Objekt der Subklasse nicht zurückgewiesen werden. Dieses Prinzip der *Substituierbarkeit* ([WEG90]) erlaubt den Einsatz von Objekten aus Subklassen an Stellen, wo Superklassenobjekte erwartet werden. Die dynamische Bindung von Methoden erlaubt die Wiederverwendung von Methodendefinitionen aus Superklassen für Subklassenobjekte. Dies erfordert aber, daß die in solchen Methoden verwendeten Attribute in Subklassenobjekten ebenfalls vorhanden sind. Daher ist die als „Record-Extension“ bekannte Technik, daß die Menge der Attribute nur erweitert werden darf, auch hier verwendet worden. Dadurch wird allerdings verhindert, daß eine Subklasse eine Neuimplementierung der Funktionalität der Superklasse mit Hilfe eines Datenstrukturwechsels durchführt. Ist jedoch eine solche Neuimplementierung notwendig, so kann durch Restrukturierung der Klassenhierarchie ein entsprechender Effekt erzielt werden ([OJ93]).

Die in manchen Sprachen erlaubte Erweiterung bzw. Einschränkung des Typs von Attributen und Methoden (siehe Eiffel) wird hier nicht modelliert. Auch ist eine Reihenfolge der Attribute wie auch der Aufrufparameter in Methoden für uns nicht relevant. Die Nachrichten, die ein Agent einer Klasse versteht, liegen damit wie folgt fest:

$$Msg(a_i) \stackrel{def}{=} \{ \ (a,mn,\beta) \in MSG \mid mn \in \Sigma_{meth}(class\ a) \ \}$$

Für die Menge der Ausgabenachrichten gibt es keine Einschränkung.

3.2.5 Dynamische Erzeugung von Agenten

Objektorientierte Systeme zeichnen sich durch eine hohe Dynamik aus. Diese Dynamik besteht einerseits aus der dynamischen Veränderung von Kenntnisbeziehungen und der damit einhergehenden Kommunikationswege, und andererseits aus der Möglichkeit, dynamisch neue Agenten zu erzeugen. Während die Dynamik der Kommunikationswege mit Hilfe des Kommunikationsmediums modelliert wird, ist für die Modellierung der dynamischen Erzeugung von Agenten ein anderer Mechanismus notwendig. Wie bereits erwähnt, werden wir uns mit der Vernichtung von Agenten nicht befassen, da wir annehmen können, daß ein Garbage Collector diese Arbeit übernimmt.

Zur Modellierung der dynamischen Erzeugung von Agenten sind zwei Probleme zu bewältigen. Ein Problem besteht in der Erzeugung und Einbindung des Agenten in das

System, insbesondere in seine Kommunikationsstrukturen. Dieses Problem haben wir gelöst, indem wir annehmen, daß alle Agenten bereits zu Beginn eines Systemablaufs vorhanden sind. Die in Definition 3.2 geforderte initiale Ruhe für fast alle Agenten erlaubt diese Modellierung, da dadurch trotzdem gesichert bleibt, daß zu jedem Zeitpunkt nur endlich viele Agenten aktiv sein können. Mit Ausnahme der endlich vielen bereits zu Beginn des Systemablaufs aktiven Agenten werden Agenten erst durch eine erste, initialisierende Nachricht *aktiviert*.

Agenten werden jedoch nicht nur neu erzeugt bzw. aktiviert, dem erzeugenden Agenten muß der Identifikator des erzeugten Agenten auch bekannt gemacht werden. Dies geschieht typischerweise unter Aufruf einer Betriebssystemroutine, die gleichzeitig einen neuen Agenten und mit ihm seinen Identifikator erzeugt. Im Systemmodell steht kein Betriebssystem zur Verfügung, das diese Aufgabe übernehmen könnte. Stattdessen statten wir jeden Agenten bereits von Anfang an mit einer ausreichend großen Menge an Identifikatoren von Agenten aller Klassen aus. Immer wenn ein neuer Agent einer Klasse aktiviert werden soll, so wird ein passendes Element aus dieser Menge genommen und als neuer Identifikator verwendet. Um zu verhindern, daß ein Agent mehrfach aktiviert wird, darf jeder Identifikator in nur einer Menge erscheinen. Agenten der Menge *StartID*, die bereits zu Beginn aktiviert sind, dürfen in keiner Menge erscheinen. Umgekehrt ist es kein Problem, wenn viele Identifikatoren während eines Systemlaufs niemals verwendet werden, da im Gegensatz zu Programmiersprachen bei uns keine Bijektion zwischen Identifikatoren und Speicheradressen besteht.

Eine alternative Lösung wäre die Benutzung von Agenten zur Verwaltung von allen Instanzen einer Klasse, wie es in der objektorientierten Datenbank-Spezifikationssprache *Troll* ([Jetal91]) üblich ist. Dieser Ansatz würde jedoch bei uns ein komplexeres Kommunikationsprotokoll zwischen dem erzeugenden Agenten und dem Verwaltungsagenten erfordern.

Für einen adäquaten Zugriff auf die Menge von Identifikatoren strukturieren wir diese in eine Abbildung, die jeder Klasse eine unendliche Liste von Identifikatoren zuordnet.

Definition 3.13 (Aktivierungsstruktur)
Die Funktion activate modelliert die Aktivierungsstruktur *innerhalb eines Systems. Dabei wird jedem Agenten eine Abbildung zugeordnet, die zu jeder Klasse, von der er Agenten aktivieren möchte, eine unendliche Liste von Identifikatoren enthält.*

$$activate : ID \to (CN \to ID^{\infty})$$

Jede Liste enthält nur Identifikatoren, die zu der zugeordneten Klasse gehören:

$$\forall a \in ID, c \in CN, n \in \mathbb{N}.\ class((activate.a.c)_n) = c$$

Jeder Identifikator kommt nur einmal, nämlich in der Liste des ihn aktivierenden Agenten, vor:

$$\forall a,b \in ID, c \in CN, n,m \in \mathbb{N}.\ (activate.a.c)_n = (activate.b.c)_m \Rightarrow a=b \wedge n=m$$

Und schließlich werden die Agenten, die bereits zu Beginn aktiv sind, von keinem anderen Agenten aktiviert:

$$\forall a \in ID, c \in CN, n \in \mathbb{N}.\ (activate.a.c)_n \notin StartID \qquad (\Box)$$

Die mit Hilfe der Funktion *activate* definierte Aktivierungsstruktur definiert auf der aktivierbaren Teilmenge von Agenten einen Wald von Bäumen, deren Wurzeln gerade die Agenten der Menge *StartID* sind. In dieser Struktur ist jeder Agent Vater aller von ihm aktivierten Agenten. (Es ist für unsere Modellierung nicht notwendig, daß alle Agenten in diesem Wald vorkommen. Außerdem sind zyklische Aktivierungsabhängigkeiten nicht ausgeschlossen, führen aber dazu, daß keiner der daran beteiligten Agenten aktiviert werden kann.)

3.2.6 Zustand und Verhalten von Agenten

Unter Instantiierung von Klassen verstehen wir die Erzeugung eines *Exemplars* einer Klasse, eines Agenten. Eine Klasse beschreibt das Verhalten und die Struktur seiner Instanzen in uniformer Weise. Neben dieser uniformen Übereinstimmung aller Instanzen einer Klasse in Struktur und Verhalten gibt es auch individuelle Unterschiede. Ein Unterschied beruht darauf, daß jede Instanz ihre Identität kennt und bekanntgeben kann. Dies wird in objektorientierten Sprachen durch ein spezielles, nicht modifizierbares Attribut `self` oder `this` modelliert, das diesen Identifikator beinhaltet. Die zweite Unterscheidung zwischen Agenten derselben Klasse besteht bei uns darin, daß gemäß der in Definition 3.13 festgelegten Aktivierungsstruktur jeder Agent unterschiedliche Agenten erzeugt.

Im Systemmodell werden diese Unterschiede in zwei speziellen Attributen mit den Namen `self` und `actees` festgehalten. Beide sind in jedem Objekt vorhanden, aber nur eingeschränkt verwendbar. Während Attribut `self` zum Lesen verwendet werden kann, aber nicht modifiziert werden darf, steht Attribut `actees` zum Zugriff überhaupt nicht zur Verfügung. Es wird stattdessen bei Erzeugungsaktionen implizit modifiziert. Entsprechende Zugriffssicherungen können in einer Programmiersprache genauso wie in einer Modellierungssprache durch syntaktische Kriterien erreicht werden. Wir legen folgende Struktur fest:

Definition 3.14 (Spezielle Attribute)
Die beiden speziellen Attribute $\texttt{self},\texttt{actees} \in \langle var \rangle$ *sind Teil jedes Agenten:*

$$\forall c.\ \{\texttt{self},\texttt{actees}\} \subseteq \Sigma_{attr}(c)$$

Diese beiden Attribute haben folgende Wertemengen:

$$values(type(\texttt{self})) = ID$$
$$values(type(\texttt{actees})) = (CN \to ID^{\infty}) \qquad (\square)$$

Wir verbinden nun die in der ersten Schicht des Systemmodells definierte Agentensicht mit der klassenbasierten Sicht. Hat eine Methode, die zur Bearbeitung einer Eingabenachricht aufgerufen wird, eine gegenüber der durch $\surd$ gegebenen Zeiteinheit eine lange Berechnungsdauer, so ist die Bearbeitung einer Methode in mehrere Zeiteinheiten unterteilt und besitzt daher Zwischenzustände. Der Zustand eines Agenten setzt sich daher aus dem Datenzustand in Form einer Attributbelegung und einem Kontrollzustand zusammen. Weil dieser Kontrollzustand je nach Sichtweise sehr unterschiedlich definiert sein kann, definieren wir den Zustandsraum eines Agenten abstrakt und geben anschließend eine mögliche Variante für dessen Realisierung an:

Definition 3.15 (Zustandsmenge aller Agenten)
Eine Funktion attrbel selektiert die Attributbelegung aus jedem Zustand eines Agenten:

$$attrbel : STATE \rightarrow BEL$$
$$\forall a{\in}ID, s{\in}states_a.\ \bullet attrbel(s){=}\Sigma_{attr}(class.a) \wedge$$
$$attrbel(s).\mathtt{self}{=}a \wedge attrbel(s).\mathtt{actees}{\in}rests(activates(a)),$$

wobei die Hilfsfunktion rests die Belegungen von `actees` *bestimmt, die durch Wegnahme nur endlich vieler Identifikatoren entsteht:*

$$rests(s) \stackrel{def}{=} \{\ t{\in}(CN{\rightarrow}ID^{\infty}) \mid \exists q{\in}(CN{\rightarrow}ID^{*}).\ s{=}q\hat{\ }t\ \}$$

(□)

Die Funktion *attrbel* kann als Projektion des Agentenzustands auf dessen Datenzustand gesehen werden. Eine mögliche Realisierung weiterer Komponenten des Agentenzustands kann aus einem Eingabepuffer noch nicht verarbeiteter Eingaben und einem Ausgabepuffer zukünftiger Ausgaben bestehen. Dieser Ausgabepuffer wirkt als Vorhersage („Prophecy“) der zukünftigen Ausgaben. Um zu sichern, daß immer wieder Elemente des Ausgabepuffers emittiert werden und damit Fortschritt der Ausgabe stattfindet, ist der Ausgabepuffer eine mit Zeitticks versehene Sequenz von Ausgabenachrichten. Damit ist es möglich, die Menge der Agentenzustände aus drei Komponenten, dem Eingabepuffer, der Attributbelegung und dem Ausgabepuffer, zu kombinieren:

$$STATE \subseteq Msg(a_i)^{*} \times BEL \times Msg(a_o)^{\overline{\omega}}$$

Diese Modellierung des Zustandsraums für Agenten erlaubt es, über noch unbearbeitete Eingaben und über Zustände zu reden, die nach der Verarbeitung eines Eingabezeichens auftreten. Zwischenzustände, die während der Verarbeitung von Eingabezeichen in der Attributbelegung auftreten könnten, fallen so weg. Stattdessen ist die Attributbelegung, ähnlich wie der Ausgabepuffer eine Darstellung der Zukunft, die eintreten wird, sobald der aktuelle Inhalt des Ausgabepuffers emittiert ist.

Eine alternative Sichtweise endlicher Vorhersagen ist die, daß der Agent intern schnell genug ist, die Verarbeitung des Eingabezeichens sofort vorzunehmen, jedoch die Ausgabe so langsam vorangeht, daß die Ausgabe zunächst gepuffert werden muß.

Eine der Implementierung wesentlich nähere und damit weniger abstrakte Realisierung des Zustandsraums eines Agenten kann in Form von Befehlszähler und Belegung lokaler Variablen angegeben werden. Für den Fall, daß beide Varianten benutzt werden sollten, ist eine passende Kongruenz zwischen beiden zu etablieren.

Sind die Zeiteinheiten des Systemmodells groß gegenüber der Verarbeitungszeit von Nachrichten, so kann wie etwa in *Esterel* ([BER93]) davon ausgegangen werden, daß alle eintreffenden Nachrichten sofort verarbeitet werden. In diesem Fall ist der Agentenzustand gleich seinem Datenzustand und *attrbel* die Identität. Ist umgekehrt die Verarbeitungszeit groß, so kann dies modelliert werden, indem immer nur eine Nachricht aus dem Eingabepuffer entnommen wird. Entsprechend sollten auch die ankommenden Nachrichten dünn im Eingabestrom verteilt sein.

3.3 Systemmodell — Zusammenfassung

Nun ist die Charakterisierung der Menge der von uns betrachteten Systeme abgeschlossen. Sie besteht aus zwei Schichten, die in den vorangegangenen Abschnitten definiert wurden. Darin wurden mehrere *Begriffe*, wie Agent, Klasse, Vererbung, Wert oder Typ formalisiert, so daß ein *Begriffsmodell* entstanden ist, das zur Charakterisierung der Menge der möglichen Systeme dient. Um ein konkretes System zu beschreiben, sind die verbleibenden Freiheitsgrade entsprechend zu reduzieren. Neben der Einführung dieser Begriffe wurden in den vorangegangenen Abschnitten auch Beziehungen dieser Begriffe untereinander definiert, die teilweise dazu führen, daß manche Begriffe aus der Definition anderer abgeleitet werden können. In der Tabelle 3.5 sind alle Begriffe zusammengestellt. Auf die Wiederholung ihrer Zusammenhänge verzichten wir hier.

Ein System ist also eine konkrete Ausprägung all der angegebenen Begriffe. Damit ergibt sich die präzise Fassung des Systemmodells:

Definition 3.16 (Systemmodell)
Ein System *ist eine konkrete Ausprägung der in Tabelle 3.5 aufgelisteten Begriffe, die die in diesem Kapitel geforderten Beziehungen zwischen diesen Begriffen erfüllt. Das* Systemmodell *ist die Menge all dieser Systeme und wird mit* $\mathcal{SM}$ *bezeichnet.* (□)

Formal ist damit ein System $sys \in \mathcal{SM}$ ein Tupel, bestehend aus den in Tabelle 3.5 angegebenen Komponenten. Zur Selektion einer dieser Komponenten verwenden wir die Subskriptionsschreibweise. So bezeichnet etwa CN_{sys} die in dem System *sys* vorkommenden Klassen. Geht aus dem Kontext eindeutig hervor, welches System gemeint ist, so lassen wir den Subskript *sys* gelegentlich weg.

3.4 Bemerkungen zum Systemmodell

3.4.1 Ein- und Ausgabe des Systems

In einem geschlossenen System, wie es in dieser Arbeit modelliert wird, sind alle Interaktionen mit der Umgebung eines Systems durch Komponenten zu modellieren. Wir wollen hier anhand eines Beispiels zeigen, wie eine solche Modellierung möglich ist.

Das Beispiel charakterisiert eine Kommunikationsverbindung mit der Außenwelt. Sei $k \in ID$ der Identifikator dieser Kommunikationskomponente. Nach dem Öffnen eines Kanals $c \in CH$ besitzt k die Fähigkeit Pakete $p \in P$ an andere Knoten zu übertragen sowie solche zu empfangen und solange zu puffern, bis diese durch eine Nachricht der Form *request(a)* angefordert werden. Wird ein Paket angefordert obwohl kein Paket eingetroffen ist, so wird ein Timeout zurückgesendet. Die Menge der Ein- und Ausgabenachrichten ist definiert als:

$$
\begin{aligned}
Msg(k_i) &= \{\ k.open(c),\ k.send(p),\ k.request(a) \mid c \in CH,\ p \in P,\ a \in ID\ \} \\
Msg(k_o) &= \{\ a.received(p),\ a.timeout() \mid a \in ID,\ p \in P\ \}
\end{aligned}
$$

ID	Menge der Identifikatoren
VAL	Werteuniversum
$\langle sort \rangle$	Menge der Typen
$\langle var \rangle$	Menge der Variablennamen
$\langle meth \rangle$	Menge der Methodennamen
$values : \langle sort \rangle \rightarrow \mathbb{P}(VAL)$	Wertemenge eines Typs
$type : \langle var \rangle \rightarrow \langle sort \rangle$	Typisierung von Variablen
$Bel_v \subseteq v \rightarrow VAL$	Menge von Belegungen der Variablenmenge v
$BEL \subseteq \langle var \rangle \rightharpoonup VAL$	Menge von partiellen Variablenbelegungen
MSG	Menge aller Nachrichten eines Systems
$Msg(a_i)$	Menge der Eingabenachrichten des Agenten a
$Msg(a_o)$	Menge der Ausgabenachrichten des Agenten a
$destination : MSG \rightarrow ID$	bestimmt Empfänger einer Nachricht
$STATE$	Menge der möglichen Agentenzustände
$states_a \subseteq STATE$	Menge der möglichen Zustände von Agent a
$inits_a \subseteq states_a$	Menge der möglichen Initialzustände von Agent a
$std_a \subseteq states_a \times Msg(a_i)^* \times states_a \times Msg(a_o)^*$	Zustandsübergangsrelation von Agent a
TA_a	Taktautomat des Agenten a
$behavior_a \subseteq \{a_i\}^{\overline{\Sigma}} \xrightarrow{p} \{a_o\}^{\overline{\Sigma}}$	Verhalten des Agenten a
CN	Endliche Menge von Klassennamen
$.\sqsubseteq. \subseteq CN \times CN$	Klassenhierarchie
$class : ID \rightarrow CN$	ordnet jedem Agenten seine Klasse zu
$.isof. \subseteq ID \times CN$	Elementrelation
$\Sigma_{attr}: CN \rightarrow \mathbb{P}^{fin}(\langle var \rangle)$	Attribute einer Klasse
$\Sigma_{meth}: CN \rightarrow \mathbb{P}^{fin}(\langle meth \rangle)$	Methoden einer Klasse
$args: \langle meth \rangle \rightarrow \mathbb{P}^{fin}(\langle var \rangle)$	Argumente einer Methode
$StartID \subseteq ID$	Initial aktive Agenten für Systemstart
$activates : ID \rightarrow (CN \rightarrow ID^{\infty})$	Aktivierungsstruktur
$EventTrace \subseteq \{trace\}^{\overline{\Sigma}}$	Menge der Systemabläufe

Abbildung 3.5: Formalisierte Begriffe des Systemmodells

Die Kommunikationskomponente hat im System das in Tabelle 3.6 in Form eines Transitionssystems definierte Verhalten. Von seinem Verhalten zur externen Umgebung wird damit abstrahiert. Eine formale Semantik für das Transitionssystem wird in Kapitel 5 definiert.

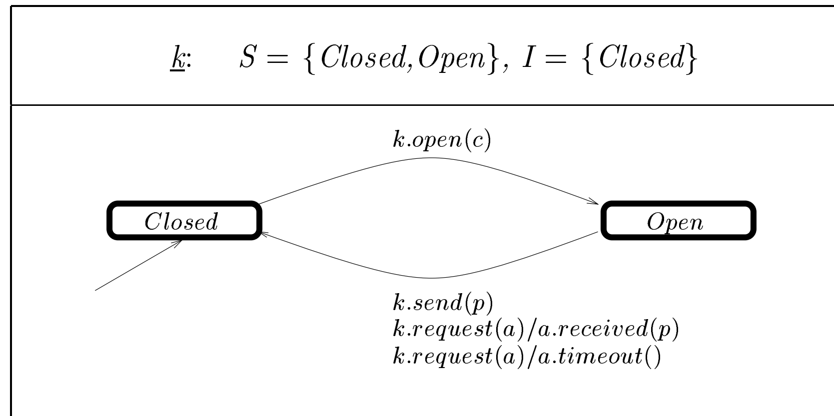


Abbildung 3.6: Automat für Kommunikationsagent k

Ein weiteres Beispiel wäre die Modellierung der Interaktion von menschlichen Benutzern mit dem System. Auf einer niedrigen Ebene könnte jeder Tastendruck und jede Mausbewegung als Ausgabe einer entsprechenden Komponente angesehen werden und Bildschirmausgaben an diese Komponente gesendet werden. Auf einer abstrakteren Ebene kann ein sogenanntes „virtuelles Terminal“ ([DEN91b]) als Komponente definiert werden, das „virtuelle Tastendrücke“ in Abhängigkeit des gerade aktiven Dialogzustands erlaubt. Solche virtuellen Tastendrücke können komplexe Aktionen, wie das Ausfüllen von Dialogfeldern sein. Diese virtuellen Tastendrücke werden als Ausgabe dieser Komponente an einen verarbeitenden Agenten weitergesendet, der als Reaktion darauf den Zustand des virtuellen Terminals verändert. Er kann z.B. einen Dialog schließen oder einen anderen öffnen.

3.4.2 Alternative Modellierungen

Mit der hier vorgestellten Variante eines Systemmodells sind natürlich einige in den vorangegangenen Abschnitten begründete Entscheidungen getroffen worden. Vor allem die im Kernbereich des System getroffenen Entscheidungen präzisieren die uns interessierenden Systeme. Die in der zweiten Schicht getroffenen Entscheidungen zur Formalisierung des Klassenbegriffs eignen sich eher dazu, gewisse Arten von Systemen einfacher, bzw. komplizierter beschreiben zu können.

Eine unserer Annahmen war der flache, nichthierarchische Aufbau unseres Systems, der eine Dekomposition von Agenten nicht zuläßt. Die Einführung einer Hierarchie ist durch eine Beschreibungstechnik möglich, indem etwa Mengen von Agenten, die stark zusammenarbeiten und bei denen nur einer von außen angesprochen wird, als Einheiten beschrieben werden können. Pragmatisch gesehen jedoch, läßt es die stark auf Dynamik

ausgerichtete Fassung des Systemmodells als eher unwahrscheinlich erscheinen, daß eine echte hierarchisch kompositionale Design- und Programmiertechnik in der Objektorientierung verwendet werden wird. Stattdessen finden in der Objektorientierung mehrere Hierarchien für verschiedene Zwecke Anwendung, aber eine starre Dekompositionshierarchie dürfte nur wenig hilfreich sein.

Ein weitere Einschränkung war der Verzicht auf mehr als einen Ein- oder Ausgabekanal pro Agent. Eine entsprechende Erweiterung könnte ohne weiteres vorgenommen werden, erschien aber für die vorliegende Arbeit als nicht notwendig. Erst in Kombination mit Hardwarebeschreibungen, die inhärent mehrere Kanäle (Leitungen) besitzen, gewinnt dies an Relevanz.

Die Restriktion des Systemmodells auf einen Eingabekanal und Verarbeitung einzelner Nachrichten durch Methodenaufrufe legt auch ein sequentielles Ausführungsmodell innerhalb eines Agenten nahe. Eine Erweiterung des Systemmodells um hierarchische Dekomposition von Agenten würde ein Ausführungsmodell mit parallelen Tasks in einem Agenten, wie sie in *ParMod-C* ([SWF91]) zusätzlich zu der asynchronen Kommunikation zur Verfügung stehen, zugrunde legen.

Das in [RKB95] definierte Systemmodell besitzt obige Erweiterungen, da es neben Software- auch auf Hardwarebeschreibungen zugeschnitten ist. Es besteht jedoch nur aus der Kernschicht und keiner darauf aufgebauten Begriffsdefinition für Klassen u.ä.m, da je nach Anwendungsbereich eine sehr unterschiedliche Begriffsbildung notwendig erscheint.

Auch in anderen Arbeiten gewinnt eine formale Beschreibung eines Systemmodells an Bedeutung. Beispielsweise wird in [DEN91b] mit der „virtuellen Maschine“ ein informelles Systemmodell definiert. In [MEY92] wird ein „Ausführungsmodell“ beschrieben, das neben *Eiffel* auch vielen anderen imperativen, objektorientierten Programmiersprachen zugrunde liegt. Nicht zuletzt wegen dieses Gewinns an Bedeutung fand in Paris der Workshop FMOODS'96 mit dem Thema Objektorientierung und Systemmodelle statt (siehe Tagungsband [KRB96]).

Demgegenüber führt das Fehlen einer expliziten Systemcharakterisierung z.B. in der Booch-Methode ([BOO94]) oder OMT ([RUM91]) zu den bereits erwähnten Integrationsschwierigkeiten verschiedener Beschreibungstechniken. In der Methode nach Wirfs-Brock, Wilkerson und Wiener ([WWW90]) und Fusion ([Cetal94]) ist eine Systemcharakterisierung bereits deutlicher zu erkennen. Nicht zuletzt deshalb dürfte der Anspruch gerechtfertigt sein, präziser als die erstgenannten Methoden zu sein. In klassischen strukturierten Methoden findet sich ebenfalls ein deutlich präziseres, aufgrund ihrer funktionalen Natur auch einfacheres Modell der zu entwickelnden Systeme wieder. Beispiele sind Jackson System Development ([JAC83]) oder SSADM ([AG90]).

Teil II

Buchstabierende Automaten

Dieser theoretische Teil der Arbeit formalisiert die Theorie der buchstabierenden Automaten. Diese werden zur Beschreibung des Ein-/Ausgabeverhaltens von asynchron kommunizierenden Agenten eingesetzt.
Nach einer Formalisierung der abstrakten Syntax buchstabierender Automaten und einem Vorschlag zu deren konkreter Darstellung, wird die Semantik buchstabierender Automaten als Abbildung in eine Menge stromverarbeitender Funktionen definiert. Ein Verfeinerungskalkül für buchstabierende Automaten wird entwickelt, und als relativ zur gegebenen Semantik korrekt bewiesen.
Dieser Teil ist wie folgt strukturiert:

4 Darstellung buchstabierender Automaten

5 Semantik für Automaten

6 Verfeinerungstechniken für Automaten

Kapitel 4

Darstellung buchstabierender Automaten

In diesem Kapitel definieren wir die abstrakte Syntax von Automaten sowie Darstellungsmöglichkeiten für diese Automaten.

Wir nutzen Automaten, um das Ein-/Ausgabeverhalten von Komponenten zu modellieren. Wir benutzen dabei keine Black-Box-Sicht, die das Verhalten einer Komponente in Form von Eingabe und Ausgabe miteinander in Beziehung setzt, sondern benutzen zu deren Spezifikation eine Abstraktion des internen Zustands der Komponente. In der Cleanroom Software Engineering Methodik wird dieser Ansatz als „State-Box-View" bezeichnet (vgl. [MDL87, HM93]) und dient als Zwischenschritt zwischen der abstrakten „Black-Box-View" und der „Glass-Box-View", die eine Strukturverfeinerung darstellt.

Ein Automat beschreibt das Verhalten einer Komponente, indem zu jedem Zustand der Komponente die Reaktion auf jeden Eingabestimulus angegeben wird. Diese Reaktion besteht aus der Verarbeitung des Eingabezeichens, der Produktion einer Sequenz von Ausgaben und dem Übergang in einen neuen Zustand, von dem aus das nächste Eingabezeichen verarbeitet wird. Die bei einer Transition stattfindende Ausgabe wird daher wie bei einem Mealy-Automaten ([HU90]) der Transition zugeordnet. Den Sonderfall, daß die Verarbeitung eines Eingabezeichens in eine unendliche Sequenz von Ausgaben mündet, modellieren wir durch sogenannte finale Transitionen, die mit einer unendlichen Ausgabesequenz versehen ist. In Anlehnung an [BRA84] nennen wir diese Automaten *buchstabierend*, weil sie im Gegensatz zu wortverarbeitenden Automaten ([Getal96]) einzelne Nachrichten verarbeiten.

Unser Ansatz Automaten zur Beschreibung des Ein-/Ausgabeverhaltens von Komponenten zu verwenden ist verwandt mit dem Ansatz der I/O-Automaten ([JON87, LS89]). Eine Transition eines I/O-Automaten verarbeitet entweder ein Zeichen oder gibt eines aus, soweit es sich nicht um eine interne Transition handelt. Die Verarbeitung eines Eingabezeichens durch eine Transition triggert dort eine Sequenz von weiteren Transitionen, die jeweils Zeichen ausgeben. Um die Verarbeitung aller anliegenden Zeichen zu garantieren ist für den I/O-Automaten „input enabledness" notwendig. Unser Konzept

ist abstrakter. Es modelliert mit einer Transition sowohl die Eingabe eines Zeichens, als auch die Reaktion darauf. Es werden daher keine Kontrollzustände benötigt, die die Ausgabe der Zeichen kontrollieren, sondern nur Datenzustände, die Äquivalenzklassen der Datenzustände der beschriebenen Komponenten darstellen.

Dieses Kapitel ist wie folgt aufgebaut: Im ersten Abschnitt definieren wir das Konzept der Automaten. Im zweiten Abschnitt werden Darstellungstechniken für Automaten vorgestellt. Im dritten Abschnitt werden Operatoren zur Hüllenbildung für das Transitionssystem eingeführt.

In Kapitel 8 werden buchstabierende Automaten in Dokumentform umgesetzt und deren Semantik mit dem Systemmodell integriert.

4.1 Abstrakte Syntax für Automaten

Die abstrakte Syntax beschreibt die zur Funktionalität beitragenden Anteile der Syntax. Von einer konkreten Darstellung wird dabei abstrahiert. Wir definieren die abstrakte Syntax von Automaten als mathematisches Konzept:

Definition 4.1 (Automat)
Ein Automat ist ein Fünftupel $(S, M_{in}, M_{out}, \delta, I)$, bestehend aus

- *einer nichtleeren Menge von Zuständen S,*
- *einer nichtleeren Menge von Eingabezeichen M_{in},*
- *einer nichtleeren Menge von Ausgabezeichen M_{out},*
- *einer Zustandsübergangsrelation $\delta \subseteq S \times M_{in} \times S \times {M_{out}}^{\omega}$ und*
- *einer Menge $I \subseteq S \times {M_{out}}^{\omega}$ von Paaren von Anfangszuständen und initialen Ausgaben.* (□)

Zur Vereinfachung unterscheiden wir zunächst nicht weiter zwischen Eingabemenge M_{in} und Ausgabemenge M_{out}. Wir setzen daher beide Zeichenmengen gleich M und bezeichnen Automaten als Quadrupel (S, M, δ, I). Die Elemente von I werden als *Initialelemente* bezeichnet. Wir schreiben meist $\delta(s, m, t, out)$ statt $(s, m, t, out) \in \delta$. Statt der relationalen Charakterisierung von δ hätte auch eine äquivalente Charakterisierung als Funktion der Form

$$\delta \in S \times M_{in} \rightarrow \mathbb{P}(S \times {M_{out}}^{\omega})$$

verwendet werden können, die die Abhängigkeit von Zielzustand und Ausgabe von Quellzustand und Eingabe deutlicher beschreibt. Wir bevorzugen jedoch obige, für unsere Zwecke des öfteren technisch einfachere Variante.

Ein Automat modelliert das Ein-/Ausgabeverhalten und die Zustandsänderungen einer Komponente. Eine Zeichensequenz aus ${M_{in}}^{*}$ wird *verarbeitet*, indem nichtdeterministisch

eine zu einem aktuellen Zustand passende Transition aus δ ausgewählt wird, die das erste Zeichen der Eingabesequenz akzeptiert. Bei der Verarbeitung dieses Zeichens werden die bei der Transition angegebenen Ausgabezeichen ausgegeben und am Ende der Verarbeitung in den Zielzustand übergegangen. Von dort aus wird die restliche Eingabe weiterverarbeitet.

Mit dieser operationellen Vorstellung, wie ein Automat Eingabeströme verarbeitet, charakterisieren wir, ähnlich wie andere Automatenansätze ([AL88]), Sicherheitseigenschaften. Lebendigkeitseigenschaften werden beispielsweise in [AL88] durch eine separate Komponente charakterisiert. Eine solche Komponente benötigen wir nicht, da unsere Automaten auch Lebendigkeitseigenschaften ausdrücken können. Unsere Transitionen sind nicht nur mit einem Eingabezeichen, sondern auch mit der Ausgabe behaftet, und da die Durchführung einer Transition als unteilbare „Aktion“ gesehen wird, tritt das beschriebene Verhalten auch ein.

Eine Transition modelliert eine kausale Abhängigkeit der Ausgabe dieser Transition von deren Eingabe. Hat eine Komponente eine Eingabe verarbeitet, so wird notwendigerweise die Ausgabe der dabei benutzten Transition emittiert. Dies bedeutet nicht immer eine zeitlich enge Aufeinanderfolge der Ausgabe nach Einlesen der Eingabe, aber ein im Sinne der Lebendigkeit gesichertes Stattfinden der Ausgabe. Wir werden in Abschnitt 5.3.1 zeigen, daß jedes Prädikat und damit auch jede Sicherheits- und Lebendigkeitseigenschaft mit Hilfe unserer Automaten beschrieben werden kann.

Im Unterschied zur Definition endlicher Automaten wie in [HU90], [PER90] oder [BRA84] erlauben wir auch unendliche Zustandsmengen und erweitern Transitionen um eine Ausgabe. Außerdem fallen Endzustände weg, da wir keine terminierenden Komponenten modellieren, sondern Komponenten mit unendlicher Lebenszeit.

Die Verarbeitung von Eingaben und die Produktion von Ausgaben findet primär in Transitionen statt. Um initiale Ausgaben, das heißt Ausgaben, die nicht als Reaktion auf ein erstes Zeichen produziert werden, zu ermöglichen, wurde die Menge der Initialzustände mit initialen Ausgaben erweitert. Ein ähnliches Phänomen ist bei Mealy- und Moore-Automaten zu beobachten: Mealy-Automaten sind nicht in der Lage auf eine leere Eingabesequenz mit einer nichtleeren Ausgabe zu antworten. Das können nur Moore-Automaten, obwohl beide im Prinzip gleichmächtige Beschreibungsmittel sind.

Definition 4.2 (total)
Ein Automat (S,M,δ,I) heißt total, *wenn für jede Eingabe in jedem Zustand mindestens eine Transition existiert*[1]*:*

$$\forall s{\in}S, m{\in}M.\ \delta(s,m,*,*).$$

Ist ein Automat nicht total, so heißt er partiell. (□)

Neben der Totalität gibt es in der Automatentheorie auch den Begriff des Determinismus, den wir hierher übertragen:

[1] Wie in Anhang A beschrieben, ist $*$ eine auf innerster Ebene existenzquantifizierte, anonyme Variable. Obige Aussage ist also äquivalent zu $\forall s{\in}S, m{\in}M. \exists t, out. \delta(s,m,t,out)$.

Definition 4.3 (deterministisch)
Ein Automat (S,M,δ,I) heißt deterministisch*, wenn für jede Eingabe in jedem Zustand höchstens eine Transition existiert:*

$$\forall s{\in}S, m{\in}M.\ \delta(s,m,*,*) \Rightarrow \exists^1\ t,out.\ \delta(s,m,t,out)$$

und nur ein Anfangszustand existiert, das heißt: $|I|=1$. Andernfalls heißt der Automat nichtdeterministisch. (□)

Für die Semantikdefinition unterscheiden wir zwei Arten von Transitionen. Transitionen heißen final, wenn aufgrund einer unendlichen Ausgabe keine weitere Transition stattfindet. Operationell gesehen terminiert die Durchführung einer finalen Transition nicht, der Zielzustand wird nicht erreicht.

Definition 4.4 (finale Transition)
Eine Transition δ(s,m,t,out) eines Automaten (S,M,δ,I) heißt final*, wenn sie eine unendliche Ausgabe besitzt. Wir teilen die Transitionsrelation δ in finale Transitionen δ^f und nicht finale Transitionen δ^c:*

$$\delta^c \stackrel{def}{=} \delta \cap S{\times}M{\times}S{\times}M^*$$
$$\delta^f \stackrel{def}{=} \delta \setminus \delta^c.$$

(□)

4.2 Darstellungstechniken für Automaten

In diesem Abschnitt definieren wir zwei Arten von Darstellungstechniken für Automaten. Eine basiert auf Tabellen, die andere auf einer graphischen Darstellung der Transitionsrelation. Beide Darstellungstechniken sind in ihrer Beschreibungsmächtigkeit ähnlich, besitzen jedoch jeweils bestimmte Vorteile. So ist eine tabellarische Darstellungsform für Automaten, wie sie in [SPI94], [BRO93] oder [DEN91] verwendet wird, geeignet die Vollständigkeit der Einträge schnell zu erkennen. Demgegenüber bietet die graphische Darstellung in Form eines Automaten dem Betrachter den Vorteil schneller Pfade zwischen Zuständen erkennen zu können. Jedoch tendieren Transitionsgraphen leichter dazu unübersichtlich zu werden.

Ein Problem entsteht dadurch, daß Automaten weder in der Anzahl der Zustände, noch in der Anzahl der Transitionen endlich sein müssen, während ihre Darstellung endlich sein muß. Wir werden deshalb unsere Darstellungsarten durch Variablen so erweitern, daß mit einem Eintrag unendlich viele Transitionen beschrieben werden können. Die entwickelten Darstellungsarten übertragen wir in Kapitel 8 in eine konkrete Syntax für Automatendokumente, wo wir dann auch die Einbindung von Automaten in den objektorientierten Systementwurf vornehmen.

Ein Automat besteht nach Definition 4.1 aus vier Komponenten *(S,M,δ,I)*. Die beiden nicht weiter strukturierten Mengen S für Zustände und M für Ein-/Ausgabezeichen können wir im folgenden als gegeben betrachten. Solche Mengen können durch abstrakte Datentypen definiert werden. Weiterhin existiere eine formale Sprache, die es erlaubt Teilmengen von S und M durch Prädikate zu charakterisieren.

Tabellarische Darstellung

Wenden wir uns zunächst der tabellarischen Darstellungsweise zu. Ist ein Automat endlich, das heißt sind die Anzahl seiner Transitionen, Zustände und die Initialzustandsmenge endlich, so können die Transitionsmenge und die Initialzustandsmenge tabellarisch aufgelistet werden. In Tabelle 4.1 wird beispielsweise der Automat mit Namen *Parity* definiert. Es wird eine Tabelle angegeben, die zu jedem Zustand und jedem Eingabezeichen einen neuen Zustand und eine Ausgabesequenz beschreibt. Ganz analog wird auch die Menge I angegeben. Tabelle 4.1 beschreibt eine Komponente, die die Parität ihres Eingabestroms berechnet und auf Anfrage (*?*) ausgibt. Eine derartige Tabelle entspricht in etwa den „normal function tables“ aus [PAR92].

Parity: $M = \{0, L, ?\}$, $S = \{A, B\}$, $I = \{(A, \varepsilon)\}$

S_{old}	M	S_{new}	Out
A	0	A	ε
	L	B	ε
	$?$	A	$\langle 0 \rangle$
B	0	B	ε
	L	A	ε
	$?$	B	$\langle L \rangle$

Abbildung 4.1: Tabellendarstellung für den *Parity*-Automaten

Ist ein Automat nicht endlich, so läßt sich eine endliche Darstellung erreichen, indem Variablen eingesetzt werden. Jede auftretende Variable wird innerhalb einer Zeile gebunden. Variablen werden im linken Teil durch Unifikation, bzw. deren effiziente Pattern-Matching-Variante, die wir aus funktionalen Sprachen kennen ([THI94, PAU91, HUD90]), gebunden und können im rechten Teil verwendet werden. Durch die Angabe zusätzlicher Prädikate (siehe etwa Spalte 3 in Tabelle 4.2) können weitere Einschränkungen angegeben werden. Tabelle 4.2 beschreibt einen unbeschränkten Puffer, der die Eingabe solange behält bis entsprechende Anfragen (*?*) kommen. Als Datenmenge steht eine nicht näher angegebene Menge D zur Verfügung.

Auch hier kann Unvollständigkeit der verwendeten Patterns auftreten. Das führt zu einer Partialität des beschriebenen Automaten. Umgekehrt können sich die angegebenen Patterns auch überlappen. Solcher Nichtdeterminismus kann gewollt sein oder durch zusätzliche Prädikate, wie im Beispiel 4.2 verhindert werden. Die Transitionsrelation δ enthält eine Transition (s,m,t,out) genau dann, wenn es eine Zeile (a,b,P,c,d) in der Tabelle und eine Variablenbelegung β gibt, so daß P unter β erfüllt ist und $\beta(a,b,c,d)$ gleich (s,m,t,out) ist.

Puffer: $M = D \cup \{?\}$, $S = D^*$, $I = \{(\varepsilon,\varepsilon)\}$

S_{old}	M	Bedingung	S_{new}	*Out*
s	d	$d \neq ?$	$s\hat{\ }d$	ε
ε	$?$		ε	$\langle ? \rangle$
$d\hat{\ }s$	$?$		s	$\langle d \rangle$

Abbildung 4.2: Tabellendarstellung für den *Puffer*-Automaten

Graphische Darstellung

Weitgehend analoge Techniken lassen sich bei der graphischen Darstellung von Transitionsrelationen verwenden. In Abbildung 4.3 ist der Automat *Parity* (vergleiche mit Tabelle 4.1) graphisch abgebildet. Die Knoten, die hier nicht explizit benannt werden müssen, definieren S. Die Transitionsrelation δ wird durch die Kanten festgelegt, die mit Eingabezeichen m und Ausgabestring *out* in der Form *m/out* beschriftet sind. Der leere Ausgabestrom ε und die Klammern $\langle . \rangle$ zur Bildung von Strömen werden dabei weggelassen.

Die Initialzustandsmenge I wird durch Kanten ohne Quellknoten dargestellt und in der Form */out* beschriftet. Die Menge der verarbeiteten Zeichen wird explizit angegeben.

Parity: $M = \{0,L,?\}$

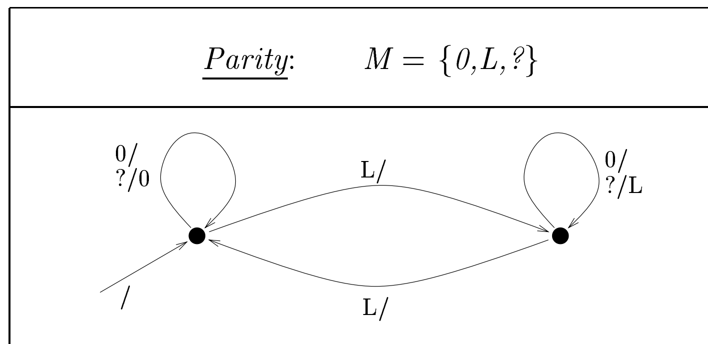


Abbildung 4.3: Graphische Darstellung für den *Parity*-Automaten

In der Graphik von Abbildung 4.3 wird jeder Zustand durch einen Knoten repräsentiert. Auf diese Weise lassen sich nur Automaten mit endlich vielen Zuständen darstellen. Im Gegensatz zur Tabellendarstellung können Unifikations-Techniken auf Transitionen nur für Eingabezeichen verwendet werden. Auch hier lassen sich Einschränkungen in einer formalen Sprache angeben. Im Beispiel 4.4 sind die Einschränkungen global angegeben,

obwohl sich auch hier der Bindebereich jeder Variable auf jeweils eine Transition beschränkt.

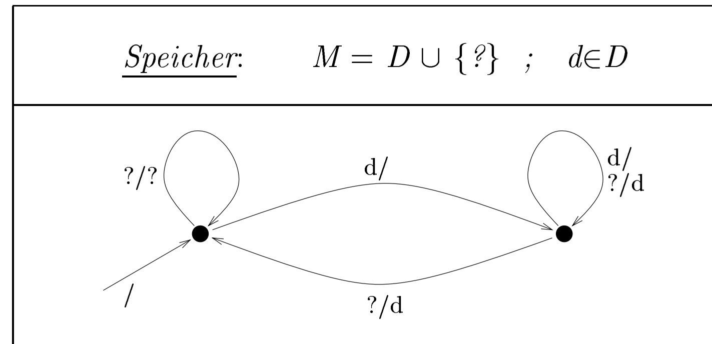
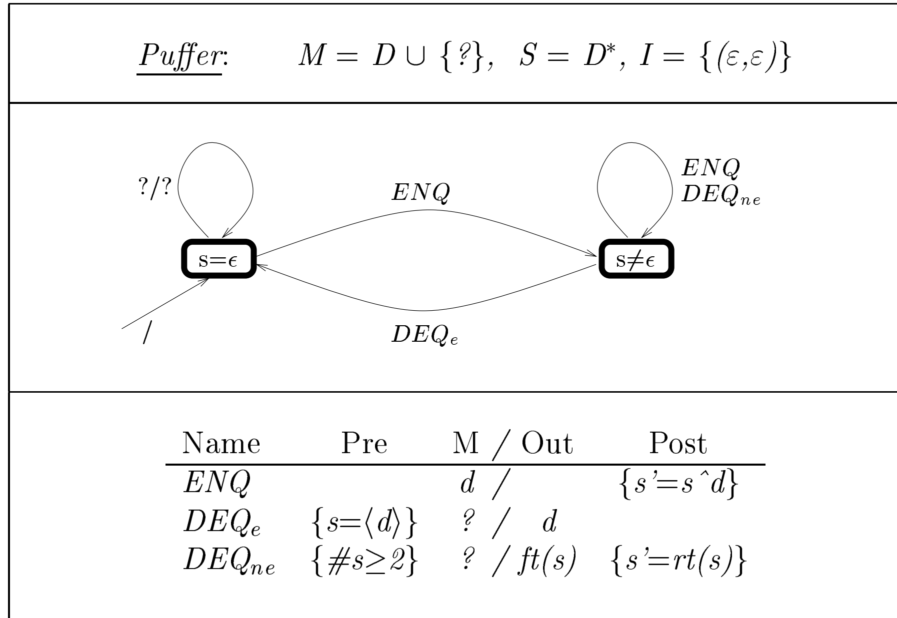


Abbildung 4.4: Graphische Darstellung für den *Speicher*-Automaten

Der in Abbildung 4.4 angegebene Automat besitzt einen Zustand, der einen leeren Speicher symbolisiert, und einen Zustand, der einen nichtleeren Speicher darstellt. Es wird jedoch nichts darüber ausgesagt, wie sich der Speicher im Detail verhält. Er kann als unbegrenzter Puffer genau so arbeiten wie auch als einelementiger Puffer oder als LIFO-Stack. Es ist nicht einmal gefordert, daß die angegebenen Zeichen vorher in die Komponente eingegeben werden müssen. Dennoch charakterisiert Abbildung 4.4 grob das Verhalten eines Speichers. Mit dem in Kapitel 6 definierten Verfeinerungskalkül kann gezeigt werden, daß der in Tabelle 4.2 angegebene Puffer eine Spezialisierung dieser Beschreibung darstellt.

In den Methoden der Programmiertechnik (zum Beispiel [SM92, BOO94, Cetal94]) ist es üblich, endliche Transitionsgraphen, oder wie im Fall [RUM91] eine hierarchische Variante, zur Darstellung des Verhaltens von Komponenten zu verwenden. Da der Zustandsraum einer solchen Komponente im allgemeinen nicht endlich ist, repräsentiert dort ein Knoten des Graphen eine Äquivalenzklasse von Zuständen. Um detaillierter über den Zustand sprechen zu können, werden Vor- und Nachbedingungen genutzt (vgl. [PR94]). Wir benutzen diesen Mechanismus, um in Abbildung 4.5 den Puffer aus Abbildung 4.2 in graphischer Form darzustellen. Wir versehen Transitionen, wo notwendig, mit Vor- und Nachbedingungen einer adäquaten formalen Sprache, die, ähnlich des Hoare'schen Spezifikationsstils, vor beziehungsweise nach den Ein-/Ausgaben stehen. Die Variable s bezeichnet den Quellzustand und s' den Zielzustand. Weil ein Knoten des Graphen eine nichtleere Menge von Zuständen repräsentiert, ist auch eine Zuordnung der Zustände zu diesen Knoten notwendig. Dies geschieht durch Bedingungen, die den Knoten zugeordnet sind, und eine Partition des Zustandsraums bilden. Um die graphische Darstellung des Automaten nicht zu überladen, bietet es sich an, komplexere Bedingungen oder Transitionen durch Namen abzukürzen und diese in einer eigenen Tabelle zu definieren. Dadurch wird auch Mehrfachverwendung möglich.

Die jetzt angegebene graphische Darstellungsform ist etwas allgemeiner als die tabellarische Version, da hier der Zielzustand prädikativ beschrieben wird, also Unterspezifikation zuläßt, während in der Tabellenform genau ein Zielzustand pro Eintrag angegeben wurde.



Name	Pre	M / Out	Post
ENQ		d /	$\{s'=s\hat{\ }d\}$
DEQ_e	$\{s=\langle d\rangle\}$	$?$ / d	
DEQ_{ne}	$\{\#s\geq 2\}$	$?$ / $ft(s)$	$\{s'=rt(s)\}$

Abbildung 4.5: Darstellung für den *Puffer*-Automaten

4.3 Hüllenbildung

Analog zur Theorie endlicher Automaten können Hüllen einer Transitionsrelation δ gebildet werden. Wir betrachten dazu die Transitionsrelation δ als zweistellige Relation zwischen Zuständen, die mit Eingaben bzw. Ausgaben attributiert ist. Wir sind neben endlichen Abläufen auch an unendlichen Abläufen interessiert und bilden daher nicht nur die transitive (oder Kleene-) Hülle, sondern auch die ∞-Hülle über unendlich viele Transitionen.

Wir benötigen diese Hüllen bei der Definition der operationellen Semantik für Automaten und bei der Behandlung von Verfeinerungsschritten auf Automaten. Zum Beispiel wird die Erreichbarkeit von Zuständen über Hüllen definiert.

Zunächst definieren wir ein Prädikat, das charakterisiert, welche endlichen oder unendlichen Sequenzen von Transitionen durch eine Transitionsrelation beschrieben werden. Diese Transitionssequenzen werden zur Hüllenbildung benutzt[2].

Definition 4.5 (Transitionssequenzen)
Sei (S,M,δ,I) ein Automat. Dann charakterisiert die Abbildung

$$ts \in \mathbb{N} \rightharpoonup \delta \ , \ \bullet ts{=}\mathbb{N} \vee \bullet ts{=}[1..n]$$

eine Sequenz von Transitionen des Automaten mit der Länge

$$\#ts \stackrel{def}{=} \begin{cases} n & \texttt{if } \bullet ts{=}[1..n] \\ \infty & \texttt{if } \bullet ts{=}\mathbb{N}, \end{cases}$$

[2] Die Struktur der Ströme und darauf existierende Operatoren sind im Anhang A.2 beschrieben.

wenn diese Transitionen aufeinanderfolgen:

$$\forall 1 \leq n < \#ts.\ ts_n = (*,*,s,*) \Rightarrow ts_{n+1} = (s,*,*,*).$$

Die Menge dieser Transitionssequenzen bezeichnen wir mit $\Theta(\delta)$. *Die Transitionssequenz* ts, *deren Glieder mit* $ts_k = (s_k, m_k, t_k, out_k)$ *bezeichnet werden, besitzt für* $1 \leq \#ts$ *einen Anfangszustand:*

$$source(ts) \stackrel{def}{=} s_1,$$

eine Eingabe- und eine Ausgabesequenz von Zeichen:

$$in(ts) \stackrel{def}{=} \texttt{conc}_{k=1}^{\#ts} m_k$$
$$out(ts) \stackrel{def}{=} \texttt{conc}_{k=1}^{\#ts} out_k,$$

und für endliche Transitionssequenzen ($\#ts < \infty$) *einen Zielzustand:*

$$dest(ts) \stackrel{def}{=} t_{\#ts}. \qquad (\Box)$$

Wir verstehen Transitionssequenzen auch als Ströme mit den üblichen Operationen. Insbesondere gilt:

$$\Theta(\delta) \subseteq (\delta)^\omega.$$

Eine Transitionssequenz beschreibt einen graphtheoretischen Pfad im Transitionssystem. Enthält die Transitionssequenz Glieder mit unendlichen Ausgaben, so entspricht die Transitionssequenz nicht einem operationell möglichen Ablauf. Wir werden operationelle Abläufe in Abschnitt 5.2 noch genauer untersuchen.

Wir definieren nun verschiedene Hüllenoperatoren, die sich vor allem in der Benutzung endlicher und unendlicher Transitionssequenzen unterscheiden.

Definition 4.6 (Hüllenbildung)

Sei δ *eine Transitionsrelation. Dann definieren wir*

- *die transitive Hülle* δ^+,
- *die reflexiv transitive Hülle* δ^*,
- *unendliche Hülle* δ^∞ *und die*
- *totale Hülle* δ^ω

durch folgende Definitionen:

$$\begin{aligned}
\delta^+ &\stackrel{def}{=} \{\ (source(ts), in(ts), dest(ts), out(ts)) \mid ts \in \Theta(\delta) \wedge 1 \leq \#ts < \infty\ \} \\
\delta^\infty &\stackrel{def}{=} \{\ (s, in(ts), s, out(ts)) \mid ts \in \Theta(\delta) \wedge \#ts = \infty \wedge s = source(ts) \} \\
\delta^* &\stackrel{def}{=} \delta^+ \cup \{\ (s, \varepsilon, s, \varepsilon) \mid s \in S\ \} \\
\delta^\omega &\stackrel{def}{=} \delta^* \cup \delta^\infty
\end{aligned} \qquad (\Box)$$

Die transitive und die ∞-Hülle unterscheiden sich nur in der Länge der zugrundegelegten Transitionssequenzen. Wird die entstehende Transition mit Hilfe einer unendlichen Transitionssequenz gebildet (δ^∞-Hülle), so ist der Zielzustand für jegliche Semantikbildung irrelevant. Wir setzen ihn daher gleich dem Quellzustand und erhalten so eine einfach handhabbare Hüllenbildung.

Als erreichbar charakterisieren wir die Menge aller Zustände aus S, die operationell erreicht werden können. Das schließt auch Zustände aus, die nur über finale Transitionen erreicht werden.

Definition 4.7 (Erreichbare Zustände)
Sei (S,M,δ,I) ein Automat. Dann definieren wir die Menge der erreichbaren Zustände *des Automaten als*

$$reach(S,M,\delta,I) \stackrel{def}{=} \{\ t \mid \exists s,in,out,o.\ \delta^{\omega}(s,in,t,out) \wedge (s,o)\in I \wedge \#out<\infty \wedge \#o<\infty\ \}. \qquad (\Box)$$

Über die Menge der tatsächlich erreichbaren Zustände hinaus definieren wir die etwas weiter gefaßte Menge der ω-erreichbaren Zustände, die auch die Zustände enthält, die nur durch eine finale Transition erreichbar sind:

Definition 4.8 (ω-Erreichbare Zustände)
Sei (S,M,δ,I) ein Automat. Dann definieren wir die Menge der ω-erreichbaren Zustände *des Automaten als*

$$\omega{-}reach(S,M,\delta,I) \stackrel{def}{=} \{\ t \mid \exists s,in,out,o.\ \delta^{\omega}(s,in,t,out) \wedge (s,o)\in I\ \}. \qquad (\Box)$$

Proposition 4.9 (Eigenschaften von $\omega{-}reach$)
Es gelten folgende Beziehungen:

$$reach(S,M,\delta,I) \subseteq \omega{-}reach(S,M,\delta,I).$$

und $\omega{-}reach(S,M,\delta,I)$ ist abgeschlossen:

$$\forall s,t.\ s\in\omega{-}reach(S,M,\delta,I) \wedge \delta(s,*,t,*) \Rightarrow t\in\omega{-}reach(S,M,\delta,I).$$

(□, Beweis siehe C.2**)**

Daß die Gleichheit beider Zustandsmengen im allgemeinen nicht gilt, läßt sich sofort an einem Beispiel mit unendlichen Ausgaben erkennen.

Kapitel 5

Semantik für Automaten

In diesem Kapitel definieren wir eine denotationelle und eine operationelle Semantik für buchstabierende Automaten. Die denotationelle Semantik ordnet einem Automaten eine Menge stromverarbeitender Funktionen zu, während die operationelle Semantik eine Menge von Abläufen, in Form von Transitionssequenzen zuordnet. Wir vergleichen beide Semantiken und zeigen, daß die in der operationellen Semantik festgelegte Vorstellung des Ablaufs von Automaten mit der denotationellen Semantik übereinstimmt.

Durch weitere Propositionen bereiten wir einerseits den im Kapitel 6 definierten Verfeinerungskalkül vor und zeigen andererseits, daß die denotationelle Semantik für jeden Automaten (mit nichtleerer Initialmenge) eine konsistente Verhaltensbeschreibung in Form einer nichtleeren Menge stromverarbeitender Funktionen liefert.

In weiteren Abschnitten dieses Kapitels werden Erweiterungen auf Bündel von Kanälen diskutiert und gezeigt, daß unsere Form der Automaten genügend allgemein ist, um jede Menge stromverarbeitender Funktionen darzustellen.

5.1 Denotationelle Semantik

In diesem Abschnitt definieren wir eine denotationelle Semantik für Automaten. Wir gehen dabei vom einfachsten Fall totaler und deterministischer Automaten aus und verallgemeinern, wie in Abbildung 5.1 dargestellt, sukzessive die gegebene Semantik auf weitere Klassen von Automaten.

Automaten werden benutzt, um das Ein-/Ausgabeverhalten abhängig vom jeweiligen Zustand der modellierten Komponente festzulegen. Jede Transition verarbeitet ein Zeichen der Eingabe, gibt eine Sequenz von Zeichen aus und bringt die beschriebene Komponente in den Folgezustand. Transitionen werden nichtdeterministisch ausgewählt.

Das Ein-/Ausgabeverhalten einer Komponente kann mit stromverarbeitenden Funktionen modelliert werden. Diese charakterisieren das Verhalten einer Komponente ohne Zuhilfenahme des internen Zustands. Während ein Automat Nichtdeterminismus einer

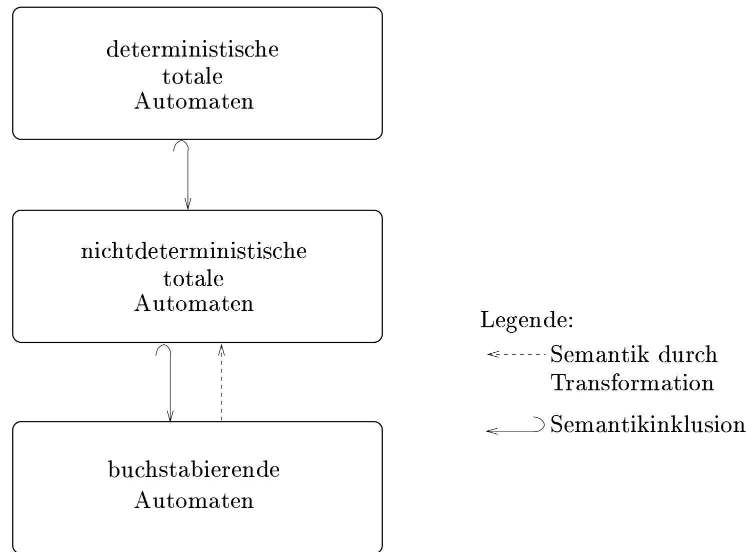


Abbildung 5.1: Semantikdefinitionen für Automaten

Komponente durch die Auswahlmöglichkeit mehrerer Transitionen mit dem gleichen Quellzustand und dem gleichen Eingabezeichen modelliert, ist eine stromverarbeitende Funktion deterministisch. Nichtdeterminismus einer Komponente wird daher durch Charakterisierung einer Menge stromverarbeitender Funktionen modelliert. Ist ein Automat gegeben, der das (nichtdeterministische) Verhalten einer Komponente modelliert, so ist dieser in eine Menge stromverarbeitender Funktionen zu übersetzen. Jede Funktion aus dieser Menge muß sich für jede Eingabe so verhalten, daß sie eine Sequenz von Transitionen des zugrundeliegenden Automaten durchläuft, dabei die Eingabe verarbeitet und die Ausgaben der Transitionssequenz ihrerseits ausgibt.

Wir definieren damit die Semantik eines Automaten als Abbildung in eine Menge stromverarbeitender Funktionen.

Definition 5.1 (Semantikabbildung)
Die Abbildung $[\![(S,M,\delta,I)]\!]^x$ *der Funktionalität*

$$[\![.]\!]^x : (S,M,\delta,I) \to \mathbb{P}(M^\omega \stackrel{s}{\to} M^\omega)$$

definiert zu einem gegebenen Automaten (S,M,δ,I) eine Menge stromverarbeitender Funktionen, die wir als Semantik von (S,M,δ,I) bezeichnen. Wir unterscheiden dabei zwischen Semantikfunktionen für die in Abbildung 5.1 angegebenen drei Klassen von Automaten: der Semantik für deterministische und totale Automaten (x=d), der Semantik für totale Automaten (x=c) und der Semantik für allgemeine Automaten (x=a). (□)

Die Semantikvariante $[\![.]\!]^a$ wird im allgemeinen mit $[\![.]\!]$ abgekürzt. Die Semantikabbildung ordnet also dem Automaten eine auf Focus ([Betal93], siehe auch Anhang A.2) basierende Semantik in Form einer Menge stromverarbeitender Funktionen zu.

5.1.1 Deterministische, totale Automaten

Ein Automat, der deterministisch und total ist, beschreibt genau eine stromverarbeitende Funktion. Er besitzt pro Zustand und Zeichen genau eine Transition, die von diesem Zustand ausgehend das ankommende Zeichen verarbeitet. Aufgrund des Determinismus kann δ als Funktion des Typs $S\times M^\omega \rightarrow S\times M^\omega$ angesehen werden.

Definition 5.2 (Semantik für deterministische und totale Automaten)
Für einen deterministischen und totalen Automaten (S,M,δ,I) definieren wir folgende Semantik:

$$\begin{aligned}
&[\![(S,M,\delta,I)]\!]^d \overset{def}{=} \{\ g\in M^\omega \overset{s}{\rightarrow} M^\omega \mid \\
&\quad \exists\ h\in S\times M^\omega \overset{s}{\rightarrow} M^\omega,\ (s_0,out_0)\in I. \\
&\quad\quad \forall s\in S, m\in M, in\in M^\omega.\ g(in)=out_0\,\hat{}\,h(s_0,in) \wedge \\
&\quad\quad\quad h(s,\varepsilon) = \varepsilon \wedge \\
&\quad\quad\quad h(s,m\,\hat{}\,in) = \texttt{let}\ (t,out) = \delta(s,m)\ \texttt{in}\ \ out\,\hat{}\,h(t,in)\ \} \qquad (\Box)
\end{aligned}$$

Die mit dem aktuellen Zustand s parametrisierte Hilfsfunktion h beschreibt das Verhalten des Automaten in diesem Zustand s. Diese Technik wird in mehreren Spezifikationen verwendet, die stromverarbeitende Funktionen nutzen (z.B. [FUC94] und [SPI94]). Aufgrund der Totalität und des Determinismus von δ sind Zielzustand t und Ausgabe out immer eindeutig gegeben. Das benutzte `let`-Konstrukt zeigt dies an. Die Hilfsfunktion h spiegelt die Reaktivität des Transitionsdiagramms wider. Für eine leere Eingabe wird keine Ausgabe produziert $(h(s,\varepsilon)=\varepsilon \forall s)$. Sonst wird jedes Eingabezeichen durch eine adäquate Transition verarbeitet.

Wie wir im Beweis der nachfolgenden Proposition sehen werden, besitzt die angegebene rekursive Charakterisierung von h einen eindeutigen Fixpunkt[1]. Dies ist nicht selbstverständlich, da die oben definierte Funktion h unter Benutzung der unstetigen Konkatenation $(.\,\hat{}\,.)$ und der Abfrage auf Leerheit des Eingabestroms $(in=\varepsilon)$ rekursiv definiert wurde. Diese Eindeutigkeit obiger rekursiver Gleichung entsteht durch die Forderung $h(s,\varepsilon)=\varepsilon\ \forall s$, die den Rekursionsanfang darstellt, der Forderung $h(s,m\,\hat{}\,in)=out\,\hat{}\,h(t,in)$, die bei jeder Anwendung das Verhalten von h für ein weiteres Eingabezeichen festlegt, sowie der geforderten Stetigkeit von h (siehe Beweis der nachfolgenden Proposition).

Proposition 5.3 (Eindeutigkeit von $[\![.]\!]^d$)
Ist (S,M,δ,I) ein deterministischer und totaler Automat, so ist $|[\![(S,M,\delta,I)]\!]^d| = 1$.

(□, Beweis siehe C.3)

Die Definition 5.2 der Semantik arbeitet auch mit finalen Transitionen. Nach Definition der Stromkonkatenation haben alle Transitionen, die der Automat nach einer Transition mit unendlicher Ausgabe durchführt, keine Auswirkungen auf die Ausgabe mehr. Dies modelliert genau denselben Sachverhalt, als ob keine weitere Transition mehr durchgeführt wird.

Aus dem letzten Teilbeweis, der zeigt, daß das rekursive Funktional τ einen eindeutigen Fixpunkt hat, ist auch ersichtlich, daß τ stetig ist und das Verhalten jeder Funktion $\tau^{n+1}(f)$ (mit beliebigem f) auf Eingaben der Länge bis zu n eindeutig festliegt.

[1] Grundbegriffe der Fixpunkttheorie sind im Anhang A.1 beschrieben.

5.1.2 Totale Automaten

Wir erweitern nun obige Definition auf nichtdeterministische, totale Automaten. Nichtdeterministische Automaten erhalten als Semantik im allgemeinen mehr als eine stromverarbeitende Funktion. Dies kann als Unterspezifikation der Beschreibung ebenso wie als Nichtdeterminismus in der zu beschreibenden Komponente betrachtet werden. Ein Unterschied zwischen beiden Konzepten läßt sich von einem externen Beobachter des Verhaltens einer Komponente nicht erkennen.

Nichtdeterminismus des Automaten kommt vor allem bei der bisher deterministischen Auswahl der Transition $\delta(s,m,t,out)$ zum Tragen. Hier darf eine beliebige Transition ausgewählt werden. Damit sich die zu beschreibende Komponente (im Sinne von Nichtdeterminismus) jedesmal neu entscheiden kann, welche Transition sie bei gegebenen Zustand und Eingabezeichen wählt, wird für jedes Zeichen der Eingabe eine neue stromverarbeitende Funktion h' gewählt, die das weitere Verhalten beschreibt. Die hier getroffene Entscheidung, Nichtdeterminismus so zu interpretieren, daß sich eine Komponente jedesmal neu entscheiden kann, führt zu einer etwas komplexeren Semantik, ist aber fundamental, um Verfeinerungen in der von uns gewünschten Weise zu erlauben. Wir werden dies im Kapitel 6 noch hinreichend sehen.

Weiterer Nichtdeterminismus entsteht bei der Auswahl des Initialpaares aus I.

Definition 5.4 (Semantik für totale Automaten)
Für einen totalen Automaten (S,M,δ,I) definieren wir folgende Semantik:

$$[\![(S,M,\delta,I)]\!]^c \stackrel{def}{=} \{\ g\in M^\omega \stackrel{s}{\rightarrow} M^\omega \mid \\ \exists\ h\in [\![(S,M,\delta,I)]\!]^p,\ (s_i,out_i)\in I.\ \forall in.\ g(in)=out_i \hat{\ } h(s_i,in)\ \}$$

wobei $[\![.]\!]^p$ die größte Menge von mit Zuständen parametrisierten Funktionen charakterisiert, die folgender rekursiver Definition genügt:

$$[\![(S,M,\delta,I)]\!]^p \stackrel{def}{=} \{\ h\in S\times M^\omega \stackrel{s}{\rightarrow} M^\omega \mid (\forall s.\ h(s,\varepsilon) = \varepsilon) \wedge \\ \forall m\in M,s.\exists t,out.\delta(s,m,t,out) \wedge \exists h'\in[\![(S,M,\delta,I)]\!]^p. \\ \forall in\in M^\omega.\ h(s,m\hat{\ }in) = out\hat{\ }h'(t,in)\}$$

($\Box$)

Im nichtdeterministischen Fall benutzen wir das gleiche Rekursionsprinzip für Mengen, wie in Definition 5.2 für die einzelne Funktion h. Die Funktionen der Menge $[\![(S,M,\delta,I)]\!]^p$ reagieren wegen der Forderung $h(s,\varepsilon)=\varepsilon$ auf die leere Eingabe durch eine leere Ausgabe und verarbeiten nichtleere Eingabesequenzen schrittweise durch adäquate Transitionen. Diese Forderungen liefern uns den Induktionsbeginn und den Induktionsschritt für Beweise über Eigenschaften der Funktionen aus $[\![(S,M,\delta,I)]\!]^p$. Der Induktionsschluß kann deshalb getroffen werden, weil alle Funktionen nach Definition stetig sind.

Die Reihenfolge der benutzten Quantoren beeinflußt natürlich die gebildete denotationelle Semantik $[\![(S,M,\delta,I)]\!]^p$. Die Quantoren wurden deshalb in genau dieser Reihenfolge gesetzt, weil die Semantik dadurch der operationellen Verarbeitung von Eingaben entspricht. Die Auswahl einer Transition hängt von dem aktuellen Zustand und dem

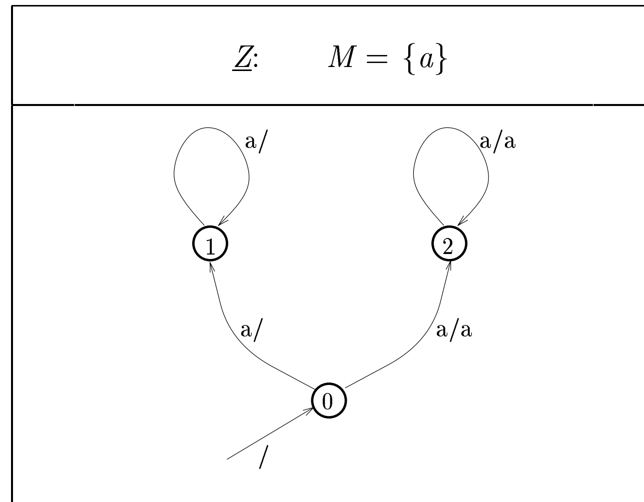


Abbildung 5.2: Graphische Darstellung für den Automaten Z

nächsten Eingabezeichen ab, weshalb $\forall m,s$ vor $\exists t,out$ stehen muß. Nach Ausführung der Transition ist ein neues Verhalten *h'* zu wählen, das alle weiteren Eingaben verarbeitet.

Würde die Auswahl der Transition *t,out* oder des neuen Verhaltens *h'* von der weiteren Eingabe *in* abhängen, das heißt, $\forall in$ vor $\exists t,out,h'$ stehen, so könnte die modellierte Einheit in Vorausschau auf zukünftige Eingaben verschiedene Transitionen wählen. Dies ist nicht in unserem Sinn. Es würde auch eine andere Semantik ergeben, wie der in Abbildung 5.2 dargestellte Automat zeigt. Es ist

$$[\![Z]\!]^c = \{f_1,\, f_2\} \ \texttt{where} \ \forall in{\in}M^\omega \\ f_1(in) = \varepsilon \\ f_2(in) = a^{\#in},$$

weil sich eine Komponente bei der ersten Transition bereits entscheiden muß, in welchen Zustand sie übergeht und diesen Zustand nicht mehr verlassen kann. In der alternativen Semantik mit einer von der Eingabe *in* abhängigen Wahl der ersten Transition würden außerdem folgende Funktionen $g_n, n{\in}\mathbb{N}$ in der Semantik enthalten sein:

$$g_n(in) = \varepsilon \quad \textit{für } \#in{<}n \\ g_n(in) = a^{\#in} \ \textit{für } \#in{\geq}n$$

Auch darf die Auswahl des Nachfolgeverhaltens *h'* nicht vor der Quantifizierung der Eingabe und des Zustands $\forall m,s$ stattfinden. Sonst wäre die Auswahl eines Folgeverhaltens zu sehr eingeschränkt. So wäre in der denotationellen Semantik des Automaten K aus Abbildung 5.3 eine Funktion f mit $f\langle aa\rangle{=}\langle aa\rangle$ und $f\langle ba\rangle{=}\langle bb\rangle$ nicht enthalten.

Wir zeigen jetzt, daß die rekursive Semantikdefinition 5.4 korrekt und eindeutig ist.

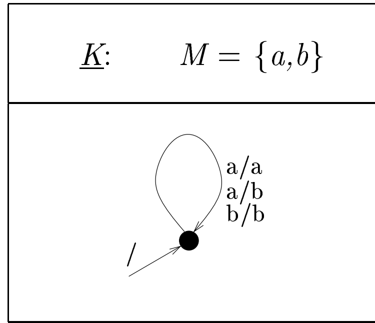


Abbildung 5.3: Graphische Darstellung für den Automaten K

Proposition 5.5 (Semantik totaler Automaten ist wohldefiniert)
Die oben gegebene Semantik totaler Automaten ist wohldefiniert. $[\![(S,M,\delta,I)]\!]^p$ *ist der größte Fixpunkt des folgenden monotonen Funktionals*

(5.1) $\sigma : \mathbb{P}(S\times M^\omega \overset{s}{\to} M^\omega) \to \mathbb{P}(S\times M^\omega \overset{s}{\to} M^\omega)$

(5.2) $\sigma(X) \overset{def}{=} \{\, h\in S\times M^\omega \overset{s}{\to} M^\omega \mid (\forall s.\ h(s,\varepsilon) = \varepsilon) \wedge$

(5.3) $\forall m,s.\exists t,out;h'\in X.\ \delta(s,m,t,out) \wedge \forall in.\ h(s,m\hat{\ }in) = out\hat{\ }h'(t,in)\}$

Das Prädikat Q: $(S\times M^\omega \overset{s}{\to} M^\omega) \to \mathbb{P}(S\times M^\omega \overset{s}{\to} M^\omega) \to \mathbb{B}$ *wird im folgenden verwendet, um die Bedingung der Mengenkomprehension darzustellen. Es gilt:* $\sigma(X)=\{h|Q.h.X\}$.

($\Box$)

Beweis 5.6
Die rekursiv definierte Menge $[\![(S,M,\delta,I)]\!]^p$ erhält ihre Semantik nach [KAH74] durch die Bildung des im vollständigen Mengenverband größten Fixpunktes des oben definierten Funktionals σ, das durch Umformulierung der Definition von $[\![(S,M,\delta,I)]\!]^p$ entsteht.

Weil die Menge X in der Mengenkomprehension von σ nur positiv genutzt wird, ist σ monoton und besitzt daher nach dem Fixpunktsatz von Knaster/Tarski ([TAR55]) einen bezüglich Mengeninklusion größten Fixpunkt $(fix.\sigma)$. Dieser ist die Menge $[\![(S,M,\delta,I)]\!]^p$.

($\Box$)

Aufgrund des Dualismus zwischen Mengen und Prädikaten ist der mengentheoretisch größte Fixpunkt, den wir für rekursive Mengenfunktionale nutzen, gleichzeitig der kleinste Fixpunkt bezüglich der Prädikatenimplikation, wie er etwa in [Betal93] zur Semantikbildung verwendet wird. Neben dem von uns gewünschten größten Fixpunkt gibt es weitere Fixpunkte von σ. Kleinster Fixpunkt ist immer die leere Menge. Charakterisiert der Automat Nichtdeterminismus beziehungsweise Unterspezifikation durch echte Auswahlmöglichkeiten von Transitionen, so gibt es neben dem von uns gewählten Fixpunkt auch Fixpunkte, die eine faire Auswahl aller bzw. bestimmter Transitionen gewährleisten. Nimmt man also (im Gegensatz zu unserem Ansatz) Fairneß-Bedingungen in den Automaten auf, so ist ein anderer Fixpunkt von σ als Semantik zu wählen. Wie wir in 5.3 noch diskutieren werden, impliziert die Einführung einer Fairneß-Komponente Probleme bei der Verfeinerung von Automaten. Neben diesen Gründen verwenden wir vor allem

deshalb den größten Fixpunkt bei der Semantikbildung, weil wir einen Automaten als Spezifikationsmittel mit im Sinne algebraischer Spezifikation loser Semantik verstehen wollen. Das heißt, wir schließen aus der denotationellen Semantik eines Automaten nur die Funktionen aus, die nicht unseren Anforderungen genügen, und lassen alle anderen zu.

Im Rest dieses und im nächsten Kapitel wird das Funktional σ noch eine große Rolle spielen, so daß wir seine Eigenschaften etwas genauer untersuchen wollen. Aufgrund der Definition von σ ist dieses Funktional sowohl bzgl. Mengeninklusion $\subseteq$ als auch der von uns verwendeten Ordnung $\supseteq$ monoton.

Wie das folgende Beispiel zeigt, ist σ im allgemeinen nicht stetig bezüglich der von uns verwendeten Inklusionsordnung. Sei $N=(S,M,\delta,I)$ ein Automat mit nur einem Zustand, und alle Transitionen haben unendlich lange Ausgaben:

$$S \overset{def}{=} \{s\}$$
$$\delta(s,m,s,out) \overset{def}{\Leftrightarrow} \#out=\infty$$

Wir definieren die folgende Menge

$$F \overset{def}{=} \{\ h \in S \times M^\omega \overset{s}{\rightarrow} M^\omega \mid \forall s,in.\ h(s,\varepsilon)=\varepsilon \wedge (in \neq \varepsilon \Rightarrow h(s,in) \in M^\infty)\ \}$$

aller Funktionen, die auf nichtleere Eingabe mit unendlicher Ausgabe reagieren. Dann gilt für jede nichtleere Menge X, daß $\sigma(X)=F$, aber $\sigma(\emptyset)=\emptyset$. Nun gibt es Ketten $K=(k_i)_i$ mit nicht leeren Gliedern $k_i \subseteq (M^\omega \overset{s}{\rightarrow} M^\omega)$, aber einem leeren Durchschnitt $\sqcup K=\emptyset$. Nach Konstruktion ist $\sqcup(\sigma K) = F$, während $\sigma(\sqcup K) = \emptyset$. Also ist σ nicht stetig.

Die Unstetigkeit von σ liegt aber nicht an der Verwendung unendlicher Ausgaben, sondern daran, daß es Ketten stromverarbeitender Funktionen mit lauter nichtleeren Gliedern gibt, deren Durchschnitt dennoch leer ist, und daran, daß bei der Semantikdefinition der Existenzquantor über h' verwendet wurde. σ ist selbst dann unstetig, wenn die Zustandsmenge S und die Zeichenmenge M endlich sind, da im Urbild der Funktionen die unendliche Menge M^ω verwendet wird.

Die Unstetigkeit von σ verhindert leider, daß wir Kleene's Fixpunkttheorem für den Beweis von Eigenschaften über $(fix.\sigma)$ verwenden können. Deshalb sind im Anhang B einige für unsere Problemstellung angepaßte Beweisschemata definiert und als korrekt bewiesen, die bei den Beweisen über Eigenschaften der denotationellen Semantik Unterstützung bieten.

Wir vergleichen nun die beiden Semantiken $[\![.]\!]^c$ und $[\![.]\!]^d$ der vorherigen Definitionen 5.2 und 5.4 auf allen deterministischen Automaten:

Proposition 5.7 ($[\![.]\!]^c$ ist kompatibel mit $[\![.]\!]^d$)
Die Semantik für nichtdeterministische Automaten $[\![.]\!]^c$ ist für deterministische, totale Automaten A kompatibel mit der deterministischen Variante $[\![.]\!]^d$:

$$[\![A]\!]^d = [\![A]\!]^c$$

(□, Beweis siehe C.4**)**

Eine weitere, sehr wichtige Eigenschaft, die für die Semantik totaler Automaten gilt, ist die Tatsache, daß deren Semantik nicht leer ist. Dies ist wichtig, wenn Elemente aus

dieser Menge ausgewählt werden, um komplexere Netze stromverarbeitender Komponenten aufzubauen. Insbesondere wird damit gesichert, daß ein Automat nicht *inkonsistent* sein kann. Inkonsistent wäre ein Automat, wenn es keine Implementierung (stromverarbeitende Funktion) gibt, die sich so verhält, wie der Automat beschreibt. In [SDW96] wird diese Eigenschaft mit „feasibility“, in VDM ([JON90]) „satisfiability“ und in [AL90] „realizability“ benannt.

Dazu beweisen wir folgende Monotonieeigenschaft der Semantikabbildung, auf die wir später noch öfter zurückgreifen werden.

Proposition 5.8 (Monotonie in δ)
Die Semantikabbildung $[\![.]\!]^c$ ist monoton bezüglich δ. Sind (S,M,δ,I) und (S,M,δ',I) zwei totale Automaten, so gilt:

$$\delta' \subseteq \delta \;\Rightarrow\; [\![(S,M,\delta',I)]\!]^c \subseteq [\![(S,M,\delta,I)]\!]^c.$$ **(□, Beweis siehe** C.5**)**

Proposition 5.9 ($[\![.]\!]^c$ besitzt immer Elemente)
Für jeden totalen Automaten (S,M,δ,I) mit $|I|\neq\emptyset$ ist die Semantik nicht leer:

$$[\![(S,M,\delta,I)]\!]^c \neq \emptyset.$$ **(□, Beweis siehe** C.6**)**

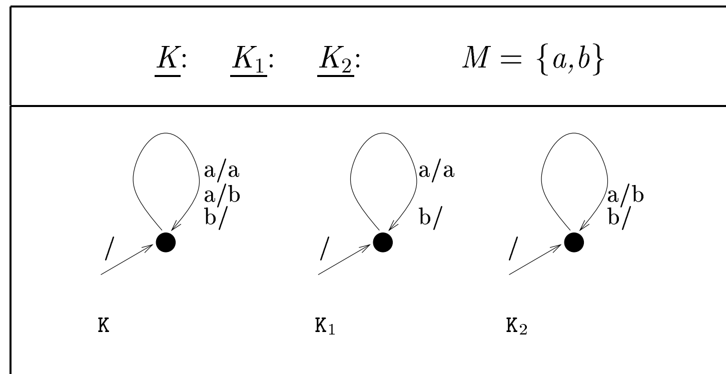


Abbildung 5.4: Automat K mit Teilautomaten

Die Semantik eines nichtdeterministischen, totalen Automaten besteht im allgemeinen aus mehr Funktionen als die Vereinigung der Semantiken aller deterministischen Teildiagramme. Man betrachte dazu das Beispiel aus Abbildung 5.4. Der nichtdeterministische Automat K hat als Semantik alle stetigen Funktionen

$$f \in M^\omega \xrightarrow{s} M^\omega,\ mit\ f(s) \in \{a,b\}^{\#_a(s)},$$

während die beiden (einzigen) deterministischen, totalen Teilautomaten K_1 und K_2 nur die beiden Funktionen f_1 beziehungsweise f_2 als Semantik haben, für die gilt:

$$f_1(s) = a^{\#_a(s)}$$
$$f_2(s) = b^{\#_a(s)}.$$

Nichtdeterminismus in einem Zustand erlaubt es der modellierten Komponente, sich jedesmal in diesem Zustand neu zu entscheiden.

5.1.3 Allgemeine Automaten

In diesem Abschnitt erweitern wir die Semantikdefinition auf partielle Automaten. Dazu ist festzulegen, was es bedeutet, wenn ein Automat in einem Zustand keine Transition bereit hält, um ein Zeichen zu verarbeiten.

Eine stromverarbeitende Funktion modelliert das Verhalten einer Komponente, die asynchron, mit ihrer Umgebung kommuniziert. Diese Komponente hat zwar Einfluß darauf, welche Ausgaben sie tätigt, nicht jedoch auf die Eingaben, die sie von anderen Komponenten erhält. Dementsprechend sind stromverarbeitende Funktionen auf allen Eingabeströmen definiert. Partialitäten in Automaten müssen also geeignet interpretiert werden.

Eine Möglichkeit ist es, Partialität als Fehlerfall zu interpretieren. Das heißt, die modellierte Komponente ist nicht darauf vorbereitet, ein Zeichen in einem bestimmten Zustand zu verarbeiten. Dazu gibt es mehrere Varianten:

Systemfehler: Eine Komponente kann einen Systemfehler auslösen und dabei explizit in einen Fehlerzustand übergehen. Sie produziert dabei keine Ausgabe mehr.

Wiederaufsetzen: Sie kann unter Verarbeitung des unwillkommenen Eingabezeichens mit unkontrollierter, beliebiger Ausgabe in einem zufällig gewählten Zustand wiederaufsetzen und dann entsprechend diesem Zustand fortfahren.

Ignorieren: Eine für Implementierungen partieller Verhaltensbeschreibungen häufig angewendete Variante ist es, die Eingabe zu ignorieren und mit leerer Ausgabe im selben Zustand wieder aufzusetzen.

Chaos: Die Komponente verhält sich von nun an völlig unkontrolliert, das heißt, sie produziert beliebige Ausgaben.

Neben der Interpretation von Partialität als Fehlerfall gibt es die Möglichkeit Partialität als Unterspezifikation anzusehen. Das heißt, es ist ab Auftreten der Partialität jede beliebige Transition erlaubt. Die Interpretation als Unterspezifikation ist sicherlich die interessanteste, da es hier möglich ist, die Verhaltensbeschreibung einer Komponente zu verfeinern, indem dort Transitionen hinzugefügt werden, wo der Automat bisher partiell war.

Während die Interpretation als Systemfehler ohne weitere Ausgabe im allgemeinen nicht der Realität entspricht, ist auch die Idee des Wiederaufsetzens relativ restriktiv: Chaos herrscht hier nur während der Bearbeitung eines Zeichens.

Die Chaosvariante führt auf dieselbe Semantik wie Unterspezifikation: beliebiges Verhalten ab dem Punkt des ersten Auftretens von Partialität. Wir werden deshalb die Partialität eines Automaten gleichzeitig als Unterspezifikation und als Chaos interpretieren. Zur Semantikdefinition benutzen wir daher folgende Funktion *complete*:

Definition 5.10 (Totalisierung eines Automaten)
Für einen gegebenen Automaten (S,M,δ,I) definieren wir Transformation complete, die diesen in einen totalen Automaten mit einem zusätzlichen Zustand $\pm \notin S$ übersetzt. Der neue Zustand $\pm$ übernimmt die Rolle eines Fehlerzustands.

$$complete(S,M,\delta,I) \stackrel{def}{=} (S',M,\delta',I)$$
`where`
$$S' \stackrel{def}{=} S \cup \{\pm\}$$
$$\delta' \subseteq S' \times M \times S' \times M^{\omega}$$
$$\delta' \stackrel{def}{=} \delta \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg\delta(s,m,*,*)\}$$
($\square$)

Proposition 5.11 (*complete* totalisiert)
Ist A ein Automat, so ist complete(A) ein totaler Automat. ($\square$, **Beweis siehe** C.7)

Die Semantik partieller Automaten wird durch Transformation gebildet:

Definition 5.12 (Denotationelle Semantik für Automaten)
Für einen gegebenen Automaten A definieren wir eine Semantik unter Rückführung auf die Semantik für totale Automaten:

$$[\![A]\!] \stackrel{def}{=} [\![complete(A)]\!]^c$$
($\square$)

In Vorbereitung auf den wichtigsten Satz dieses Kapitels zeigen wir nun, daß nicht ω-erreichbare Zustände bei der Bildung der Semantik totaler Automaten keine Rolle spielen.

Proposition 5.13 (Einschränkung auf ω-erreichbare Zustandsmenge)
Ist (S,M,δ,I) ein totaler Automat und gilt

$$R = \omega{-}reach(S,M,\delta,I)$$
$$\delta' = \delta \cap R{\times}M{\times}R{\times}M^{\omega},$$

dann ist (R,M,δ',I) ein totaler Automat und es gilt:

$$[\![(S,M,\delta,I)]\!]^c = [\![(R,M,\delta',I)]\!]^c.$$
($\square$, **Beweis siehe** C.8)

Aussage 5.13 läßt sich sehr leicht erweitern zu:

Proposition 5.14 (Entfernen nicht ω-erreichbarer Zustände)
Sind (S,M,δ,I) und (S',M,δ',I) totale Automaten und gilt

$$\omega{-}reach(S,M,\delta,I) \subseteq S' \subseteq S$$
$$\delta' = \delta \cap S'{\times}M{\times}S'{\times}M^{\omega},$$

dann ist

$$[\![(S,M,\delta,I)]\!]^c = [\![(S',M,\delta',I)]\!]^c.$$
($\square$, **Beweis siehe** C.9)

So gerüstet zeigen wir nun:

Proposition 5.15 **($[\![.]\!]$ ist kompatibel mit $[\![.]\!]^c$)**
Für einen totalen Automaten A sind die Semantikvariante $[\![.]\!]^c$ und $[\![.]\!]$ kompatibel:

$$[\![A]\!] = [\![A]\!]^c$$

(□, **Beweis siehe** C.10)

Wenn ein Automat totalisiert wird, ohne daß ein neuer, expliziter Fehlerzustand eingeführt wird, dann erhält man eine Semantik, die im allgemeinen weniger Funktionen besitzt. Wird nämlich kein expliziter Fehlerzustand eingeführt, so bedeutet dies Wiederaufsetzen in einem definierten Zustand, nachdem ein Zeichen bei einer Transition verarbeitet wurde. Der in Abbildung 5.5 gegebene Automat *Chaos* ist in seinem einzigen Zustand s partiell in Zeichen b. Die dadurch modellierte Komponente *Chaos* kann bei Einlesen eines Zeichen zunächst beliebiges tun, muß aber, soweit es sich nicht entscheidet eine unendliche Ausgabe zu starten, wieder im Zustand s aufsetzen und sich gemäß diesem Zustand verhalten. Insbesondere ist beim Einlesen des Zeichens a weiterhin nur dessen Wiederausgabe möglich.

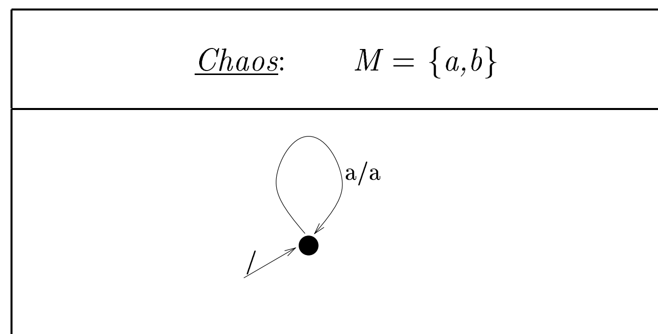


Abbildung 5.5: Graphische Darstellung für den *Chaos*-Automaten

Wir übertragen nun einige Ergebnisse totaler Automaten auf allgemeine Automaten und erhalten so eine Kernaussage dieses Kapitels:

Satz 5.16 **(Eigenschaften von $[\![.]\!]$)**

1. *Ist (S,M,δ,I) ein totaler, deterministischer Automat, so ist $|[\![(S,M,\delta,I)]\!]| = 1$.*
2. *Für jeden Automaten (S,M,δ,I) mit $I\neq\emptyset$ ist $[\![(S,M,\delta,I)]\!]\neq\emptyset$.* (□)

Beweis 5.17
Ist (S,M,δ,I) total und deterministisch, so folgt mit den Propositionen 5.7 und 5.15:

(5.4) $$[\![(S,M,\delta,I)]\!] = [\![(S,M,\delta,I)]\!]^c = [\![(S,M,\delta,I)]\!]^d$$

und mit Proposition 5.3 der erste Teil der Behauptung.

Der zweite Teil folgt unmittelbar aus Definition 5.12 und Proposition 5.9. (□)

5.2 Operationelle Semantik

In diesem Abschnitt behandeln wir die operationelle Semantik für totale Automaten und zeigen den engen Zusammenhang zur denotationellen Semantik. Wir schränken uns hier auf totale Automaten ein, da die operationelle Semantik für partielle Automaten genau wie die denotationelle durch eine Totalisierung mittels *complete* erfolgen kann. In einem weiteren Teilabschnitt wird der Zusammenhang zwischen der von uns benutzten denotationellen Semantik und einer auf dem relationalen Spezifikationsstil ([BS94]) beruhenden, denotationellen Semantik untersucht und gezeigt, warum dieser Stil für unsere Zwecke nicht geeignet ist.

5.2.1 Abläufe

Die operationelle Semantik eines Automaten wird definiert als eine Menge von Abläufen. Ein Ablauf beschreibt die Sequenz der Transitionen, die eine Komponente bei der Verarbeitung der Eingabe ausführt. Ein Ablauf enthält die verarbeitete Eingabe, die produzierte Ausgabe und die Sequenz der dabei eingenommenen Zustände. Ein fiktiver Beobachter, der sowohl Eingabe, Ausgabe als auch den internen Zustand eines Automaten sieht, beobachtet genau diese Menge von Abläufen.

Abläufe stehen in engem Zusammenhang mit den in 4.5 definierten Transitionssequenzen. Sie behandeln jedoch finale Transitionen und Initialelemente mit unendlicher Ausgabe unterschiedlich.

Ein Ablauf wird aus folgenden vier Komponenten zusammengesetzt, die jeweils einzelne *Ablaufelemente* beschreiben:

$s \xrightarrow{in/out} t$ ist eine Transition mit endlicher Ausgabe. Für sie gilt:

$$\delta(s,in,t,out) \wedge \#out{<}\infty$$

$\xrightarrow{out_0} t_0$ beschreibt ein durchlaufenes Initialelement mit endlicher Ausgabe. Es gilt:

$$(t_0,out_0){\in}I \wedge \#out_0{<}\infty$$

$s \xrightarrow{in/out}$ ist eine finale Transition, das heißt also, sie besitzt eine unendliche Ausgabe:

$$\delta(s,in,*,out) \wedge \#out{=}\infty$$

$\xrightarrow{out_0}$ beschreibt ein durchlaufenes Initialelement mit unendlicher Ausgabe:

$$(*,out_0){\in}I \wedge \#out_0{=}\infty$$

Diese Ablaufelemente sind in den Mengen $(S{\times}M{\times}S{\times}M^*)$, $(S{\times}M^*)$, $(S{\times}M{\times}M^\infty)$ und (M^∞) enthalten. Die Menge der Ablaufelemente wird als Vereinigung dieser Mengen betrachtet. Bereits deren unterschiedliche Signatur erlaubt eine Unterscheidung der verschiedenen Elementarten.

Definition 5.18 (Ablaufelemente)
Die Menge der Ablaufelemente *ist wie folgt definiert:*

$$TE_{M,S} \overset{def}{=} (S{\times}M{\times}S{\times}M^{*}) \cup (S{\times}M^{*}) \cup (S{\times}M{\times}M^{\infty}) \cup (M^{\infty}).$$

Wir definieren die Funktionen source und dest zur Selektion des Quell- *bzw.* Zielzustands *für die einzelnen Elemente wie folgt:*

$$source(s \xrightarrow{in/out} t) = source(s \xrightarrow{in/out}) = s$$
$$dest(s \xrightarrow{in/out} t) = dest(\xrightarrow{out} t) = t$$

Beide Funktionen sind für alle anderen Elemente undefiniert.
Die Funktionen in und out liefern jeweils die Eingabe und die Ausgabe des Elements. Dabei wird $in(\xrightarrow{out}) \overset{def}{=} \varepsilon$ *gesetzt.* (□)

Wir bilden die Menge der Abläufe als Teilmenge der Ströme über Ablaufelementen, obwohl diese Elemente bereits Ströme als Komponenten beinhalten bzw. selbst welche sind. Wir benutzen hier den Datentyp Strom, um damit endliche und unendliche Listen zu formalisieren, im Unterschied zur bisherigen Verwendung von Strömen zur Modellierung des Kommunikationsverhaltens auf Kanälen.

Definition 5.19 (Abläufe)
Ein Ablauf *ist eine endliche oder unendliche Sequenz von Ablaufelementen. Dabei dürfen Ablaufelemente ohne Quellzustand nur als erstes und Ablaufelemente ohne Zielzustand nur als letztes auftreten. Die Quell- und Zielzustände aufeinanderfolgender Ablaufelemente müssen identisch sein. Die Menge der Abläufe ist damit eine Teilmenge von* $(TE_{M,S})^{\omega}$*, oder präziser:*

$$Trace_{M,S} \overset{def}{=} \{ \; tr \in (TE_{M,S})^{\omega} \mid tr{\neq}\varepsilon \wedge tr_0{\in}(S{\times}M^{*}){\cup}(M^{\infty}) \wedge$$
$$\forall n.\; 0{\leq}n{<}\#tr.\; \exists s{\in}S.\; s = dest(tr_n) = source(tr_{n+1}) \; \}$$

Die Funktionen in und out werden punktweise auf Abläufe erweitert. Sie beschreiben daher die gesamte Eingabe bzw. Ausgabe des Ablaufs. (□)

Die operationelle Semantik eines Automaten (S,M,δ,I) entspricht einer Teilmenge von $Trace_{M,S}$. Wir lassen unvollständige Abläufe zu. Das sind Abläufe, deren letztes Element einen Zielzustand besitzt, von dem aus weitere Transitionen stattfinden können. Ein unvollständiger Ablauf ist damit ein echter Präfix eines vollständigen Ablaufs. Vollständige Abläufe lassen sich wie folgt darstellen ($\#out_f{=}\infty$):

$$\xrightarrow{out_0} s_1 \xrightarrow{in_1/out_1} s_2 \xrightarrow{in_2/out_2} s_3 \xrightarrow{in_3/out_3} s_4 \;\ldots$$
$$\xrightarrow{out_0} s_1 \xrightarrow{in_1/out_1} s_2 \xrightarrow{in_2/out_2} s_3 \xrightarrow{in_3/out_f}$$
$$\xrightarrow{out_f}$$

5.2.2 Operationelle Semantik

Zu einem gegebenen totalen Automaten (S,M,δ,I) konstruieren wir nun die operationelle Semantik unter Benutzung der Transitionssequenzen. In Definition 4.5 wurde bereits die

Menge der Transitionssequenzen $\Theta(\delta)$ definiert, die jedoch den bereits dort beschriebenen Nachteil besitzt, daß diese über finale Transitionen hinaus weitere Transitionen in einer Transitionssequenz zuläßt. Sie eignet sich deshalb nicht direkt zur Beschreibung der Abläufe einer Komponente, kann aber dazu genutzt werden, diese zu definieren.

Dazu benutzen wir eine Abstraktionsfunktion Υ, die aus jedem Strom, bestehend aus einem Initialelement und weiteren Transitionen $I\hat{\ }\Theta(\delta)$, den tatsächlich durchlaufenen Anteil (den Ablauf also) herausfiltert.

Definition 5.20 (Abstraktionsfunktion Υ)
Zu einem gegebenen Automaten (S,M,δ,I) wird die Abstraktionsfunktion Υ wie folgt definiert:

$$\begin{array}{lcl}
\Upsilon(\varepsilon) & \stackrel{def}{=} & \varepsilon \\
\Upsilon((t_0,out_0)\hat{\ }tr) & \stackrel{def}{=} & \begin{cases} \xrightarrow{out_0} t_0 \ \hat{\ }\Upsilon(tr) & \texttt{if} \quad \#out_0<\infty \\ \xrightarrow{out_0} & \texttt{if} \quad \#out_0=\infty \end{cases} \\
\Upsilon((s,in,t,out)\hat{\ }tr) & \stackrel{def}{=} & \begin{cases} s \xrightarrow{in/out} t \ \hat{\ }\Upsilon(tr) & \texttt{if} \quad \#out<\infty \\ s \xrightarrow{in/out} & \texttt{if} \quad \#out=\infty \end{cases}
\end{array} \qquad (\Box)$$

Damit können wir die operationelle Semantik angeben:

Definition 5.21 (Operationelle Semantik)
Für einen totalen Automaten (S,M,δ,I) definieren wir die folgende operationelle Semantik:

$$[\![(S,M,\delta,I)]\!]^{op} \stackrel{def}{=} \Upsilon(I\hat{\ }\Theta(\delta)) \qquad (\Box)$$

Die operationelle Semantik erlaubt es einer Komponente, ein beliebiges Element aus der Initialmenge zu wählen. Besitzt das Initialelement eine endliche Ausgabe, so wird eine Sequenz von Transitionen durchlaufen. Diese ist nach Definition 4.5 eine Transitionssequenz aus $\Theta(\delta)$ die mit dem Initialzustand beginnt und bei der höchstens die letzte Transition eine finale Transition ist. Ein Ablauf mit bereits unendlicher initialer Ausgabe oder einer finalen Transition am Ende signalisiert dabei, daß operationell kein weiterer Zustand erreicht wird. Dennoch wird weiterhin Eingabe verarbeitet, diese hat aber keine kausale Auswirkung auf die Ausgabe mehr.

Wir stellen nun im folgenden Satz den Zusammenhang zwischen operationeller und denotationeller Semantik her:

Satz 5.22 (Operationelle und denotationelle Semantik sind äquivalent)
Für alle totalen Automaten (S,M,δ,I) gilt:

1. *Jeder Ablauf in $[\![(S,M,\delta,I)]\!]^{op}$ wird durch eine Funktion aus $[\![(S,M,\delta,I)]\!]^{c}$ beschrieben:*

$$\forall tr \in [\![(S,M,\delta,I)]\!]^{op}\ \exists f \in [\![(S,M,\delta,I)]\!]^{c}.\ f(in(tr)) = out(tr)$$

2. *Jeder nach* $[\![(S,M,\delta,I)]\!]^c$ *mögliche Ablauf ist in* $[\![(S,M,\delta,I)]\!]^{op}$ *enthalten:*

$$\forall f\in[\![(S,M,\delta,I)]\!]^c, i\in M^\omega.\ \exists tr\in[\![(S,M,\delta,I)]\!]^{op}.\\ out(tr)=f(i) \wedge (in(tr) = i \vee (in(tr)\sqsubseteq i \wedge \#out(tr)=\infty))$$

3. *Hat ein Automat* (S,M,δ,I) *keine unendlichen Ausgaben, so vereinfacht sich Aussage 2 zu*

$$\forall f\in[\![(S,M,\delta,I)]\!]^c, i\in M^\omega.\ \exists tr\in[\![(S,M,\delta,I)]\!]^{op}.\ out(tr)=f(i) \wedge in(tr) = i$$

(□, **Beweis siehe** C.11)

Dieser Satz zeigt, daß die unseren Automaten zugrundeliegende, intuitive Vorstellung der Abarbeitung, die wir in der operationellen Semantik gefaßt haben, mit unserer denotationellen Semantik übereinstimmt.

5.2.3 Relationsbasierte Semantik

In diesem Abschnitt behandeln wir den Zusammenhang zwischen der von uns benutzten rekursiven Semantikdefinition 5.4 für totale Automaten und einer Semantikdefinition, die auf der Benutzung der ω-Hülle beruht. Wie wir in diesem Abschnitt sehen werden, liefert diese auf dem relationalen Spezifikationsstil ([BS94]) beruhende Technik eine andere Semantik als die von uns angegebene.

Definition 5.23 (Relationsbasierte Semantikdefinition $[\![.]\!]^r$**)**
Für einen totalen Automaten (S,M,δ,I) *definieren wir die folgende auf Relationen basierende Semantik:*

$$[\![(S,M,\delta,I)]\!]^r \stackrel{def}{=} \{\ g\in M^\omega \stackrel{s}{\rightarrow} M^\omega \mid \forall in\in M^\omega.\\ \exists\ (t_0,out_0)\in I, out.\ \delta^\omega(t_0,in,*,out) \wedge g(in) = out_0\ \hat{}\ out\ \}$$

(□)

Wir geben eine dazu äquivalente Charakterisierung mittels der operationellen Semantik $[\![.]\!]^{op}$ an:

Proposition 5.24 (Charakterisierung von $[\![.]\!]^r$ **durch** $[\![.]\!]^{op}$**)**
Für alle totalen Automaten (S,M,δ,I) *gilt:*

$$[\![(S,M,\delta,I)]\!]^r = \{\ g\in M^\omega \stackrel{s}{\rightarrow} M^\omega \mid \forall i\in M^\omega.\ \exists tr\in[\![(S,M,\delta,I)]\!]^{op}.\\ out(tr)=g(i) \wedge (in(tr)=i \vee (in(tr)\sqsubseteq i \wedge \#g(i)=\infty))\ \}$$

(□, **Beweis siehe** C.13)

Natürlich sind die denotationelle Semantik $[\![.]\!]^c$ und die relationsbasierte Semantik $[\![.]\!]^r$ nicht voneinander unabhängig.

Proposition 5.25 ($[\![.]\!]^c \subseteq [\![.]\!]^r$**)**
Für alle totalen Automaten (S,M,δ,I) *gilt:*

$$[\![(S,M,\delta,I)]\!]^c \subseteq [\![(S,M,\delta,I)]\!]^r$$

(□, **Beweis siehe** C.14)

Andererseits gilt im allgemeinen keine Gleichheit zwischen den Semantiken, wie das Gegenbeispiel in Abbildung 5.2 zeigt. Es ist

$$[\![Z]\!]^c = \{f_1,\, f_2\} \; \texttt{where} \; \forall in{\in}M^\omega$$
$$f_1(in) = \varepsilon$$
$$f_2(in) = a^{\#in} \; ,$$

weil sich eine Komponente bei der ersten Transition bereits entscheiden muß, in welchen Zustand sie übergeht und diesen Zustand nicht mehr verlassen kann. In der auf Relationen basierenden Semantik gibt es jedoch weitere Funktionen:

$$[\![Z]\!]^r = \{f_1,\, f_2,\, g_n \mid n{\in}\mathbb{N}\;\} \; \texttt{where}$$
$$g_n(in) = \varepsilon \;\; \textit{für} \; \#in{<}n$$
$$g_n(in) = a^{\#in} \; \textit{für} \; \#in{\geq}n$$

die daraus bestehen, sich bis zu einer bestimmten Eingabelänge zu verhalten als wären sie in Zustand *1*, dann aber plötzlich als wären sie in Zustand *2*. Dies geht in diesem Beispiel deshalb, weil die Ausgabe des einen Zweiges des Transitionsgraphen einen Präfix des anderen Zweiges erzeugt. Beim Wechsel des Zweiges ist dann der gesamte noch fehlende Rest auszugeben. Um einen solchen Wechsel zu verhindern, wäre es für die relationsbasierte Semantik notwendig, eine Fano-Bedingung für Ausgaben von Transitionssequenzen mit gleicher Eingabe und gleichem Startzustand zu fordern. Dadurch wird jedoch interner Nichtdeterminismus, das ist Nichtdeterminismus, der nicht sofort durch Ausgaben beobachtet werden kann, verboten.

Wird ein Automat neben der Verhaltensbeschreibung auch genutzt, um Zustandsfolgen bzw. Lebenszyklen zu beschreiben, wie das in objektorientierten Methoden der Fall ist, dann dürfen Funktionen der Form g_n nicht in der Semantik des Automaten Z aus Abbildung 5.2 enthalten sein. Daher ist die relationsbasierte Semantikdefinition für unsere Zwecke ungeeignet.

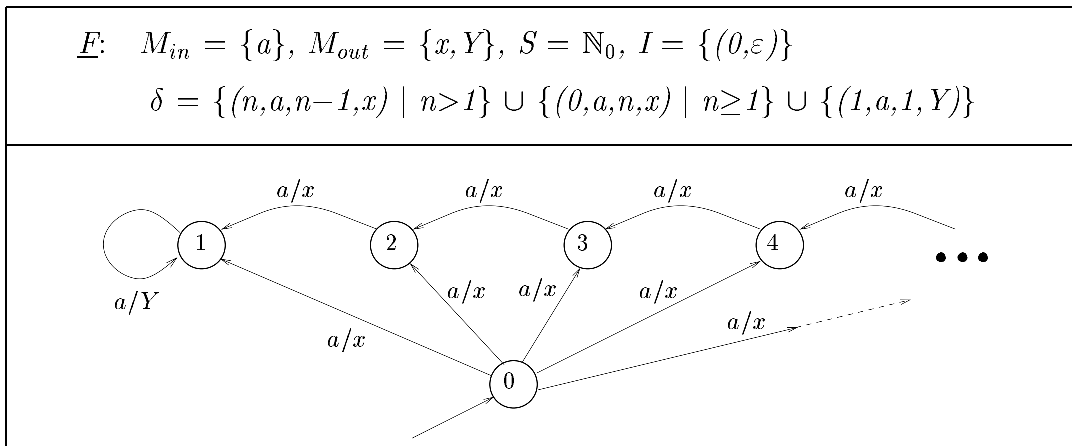


Abbildung 5.6: Automat F

Zur Beschreibung von Fairneß- und Lebendigkeitseigenschaften werden oft Strukturen wie im Automaten F aus Abbildung 5.6 benutzt, die einen Zähler („Prophecy“-Variable)

([AL88], in [SDW96] auch „Hypothesis"-Variable genannt) im Zustand kodieren. Die Zustände $[1,\infty[$ bilden den Zähler, der bei jeder Transition verringert wird und schließlich in der Ausgabe eines Y resultiert. Die Auswahl einer Transition im Zustand 0 setzt den Zähler nichtdeterministisch.

In einer Semantik, die iteriert über endliche Eingabesequenzen gebildet wird, kann für jede endliche Eingabe a^n ein Weg gefunden werden, der die Ausgabe x^n produziert. Eine stetige Erweiterung auf a^∞ erlaubt dann im Widerspruch zur operationellen Semantik die Ausgabe x^∞. Aufgrund der Verwendung der ω-Hülle δ^ω enthält $[\![F]\!]^r$ keine Funktion mit derartigem Ausgabeverhalten. Wegen Proposition 5.25 enthält auch $[\![F]\!]^c$ keine solche Funktion.

Die Semantik aus 5.23 korrespondiert sehr stark mit der relationalen Charakterisierung, die eine Beziehung zwischen Eingabe und Ausgabe festlegt. Sei R wie folgt definiert (vergleiche dazu [BS94]):

$$R \stackrel{def}{=} \{\ (in,out_0\,\hat{}\,out) \in M^\omega \times M^\omega \mid \exists t_0.\ (t_0,out_0) \in I \wedge \delta^\omega(t_0,in,*,out)\ \}$$

Dann wird dadurch genau die Menge der stromverarbeitenden Funktionen $[\![(S,M,\delta,I)]\!]^r$ charakterisiert. Wie wir aber bereits mit dem Automaten Z aus Abbildung 5.2 gezeigt haben, sind damit spontane Sprünge zwischen Zuständen möglich, die wir in unserer Semantikdefinition nicht wollen. Wie wir in Abschnitt 5.1.2 bereits diskutiert haben, entsteht das Problem im wesentlichen bei der Vertauschung von Quantoren, die es erlauben zu jeder Eingabesequenz eine neue Transitionssequenz zu wählen die diese verarbeitet, während bei der Semantik aus Definition 5.4 eine einmal ausgeführte Transition auch ausgeführt bleibt.

5.3 Erweiterungen

5.3.1 Automaten gebildet aus stromverarbeitenden Funktionen

Um zu zeigen, daß mit unserer Form buchstabierender Automaten jede stromverarbeitende Funktion charakterisiert werden kann, geben wir hier eine Konstruktion an, die zu einer stromverarbeitenden Funktion einen totalen Automaten bildet:

Definition 5.26 (Konstruktion *automaton***)**
Sei $f \in M^\omega \stackrel{s}{\rightarrow} M^\omega$ eine stromverarbeitende Funktion, dann ist

$$automaton(f,M) \stackrel{def}{=} (S,M,\delta,I)$$
`where`
$$S \stackrel{def}{=} M^*$$
$$I \stackrel{def}{=} \{(\varepsilon,\ f(\varepsilon))\}$$
$$\delta \stackrel{def}{=} \{(s,m,s\,\hat{}\,m,out) \mid f(s\,\hat{}\,m) = f(s)\,\hat{}\,out,\ m \in M\ \}$$

die Konstruktion eines Automaten aus f. (□)

Wie in [Betal93] vorgeschlagen, merkt sich die Zustandsmenge S die komplette bisherige Eingabegeschichte. Feiner braucht ein Zustandsbegriff nicht gebildet zu werden, aber im allgemeinen können Äquivalenzklassen auf der Zustandsmenge gefunden werden.

Proposition 5.27 (*automaton* **Eigenschaften)**
Die Konstruktion automaton konstruiert zu jeder stromverarbeitenden Funktion f einen auf M totalen Automaten, der als Semantik genau f hat:

$$\forall f.\ [\![automaton(f,M)]\!] = \{f\}.$$

(□, **Beweis siehe** C.15)

Diese Form der Umsetzung kann auf Mengen von stromverarbeitenden Funktionen erweitert werden. Man erweitert dazu den Zustand, der sich jetzt auch vermerkt, welche Funktion modelliert wird. Die Menge der Initialzustände I erlaubt die initiale Auswahl der stromverarbeitenden Funktion, die modelliert wird.

Definition 5.28 (Konstruktion *automaton* **für Mengen)**
Sei $K \subseteq M^\omega \xrightarrow{s} M^\omega$ *eine nichtleere Menge stromverarbeitender Funktionen, dann ist*

$$automaton(K,M) \stackrel{def}{=} (S,M,\delta,I)$$
where
$$S \stackrel{def}{=} K \times M^*$$
$$I \stackrel{def}{=} \{((f,\varepsilon),\ f(\varepsilon)) \mid f \in K\}$$
$$\delta \stackrel{def}{=} \{((f,s),m,(f,s\hat{\ }m),out) \mid f(s\hat{\ }m) = f(s)\hat{\ }out,\ m \in M,\ f \in K\}$$

die Konstruktion eines Automaten aus K. (□)

Proposition 5.29 (*automaton* **Eigenschaften für Mengen)**
Die Konstruktion automaton konstruiert zu einer Menge stromverarbeitender Funktionen K einen auf M totalen Automaten, der als Semantik genau die Menge K hat:

$$\forall K.\ [\![automaton(K,M)]\!] = K.$$

(□, **Beweis siehe** C.16)

Damit ist bewiesen, daß buchstabierende Automaten als Beschreibungsmittel allgemein genug sind, um jedes mögliche Verhalten zu beschreiben.

5.3.2 Gezeitete Semantik buchstabierender Automaten

Zur Verbindung von Automaten mit dem Systemmodell benötigen wir eine gezeitete Semantik buchstabierender Automaten.

Eine ungezeitete stromverarbeitende Funktion $f \in {M_{in}}^\omega \xrightarrow{s} {M_{out}}^\omega$ kann als Spezifikation einer Menge gezeiteter stromverarbeitender Funktionen verstanden werden. Dazu kann die folgende Funktion ρ verwendet werden, die einer Funktion f alle gezeiteten Funktionen g zuordnet, deren Zeitabstraktion dasselbe Verhalten wie f festlegen[2].

[2] Der Operator $\diamond$ abstrahiert von der Zeitinformation ($\surd$) eines Stroms; siehe Abschnitt A.3

Definition 5.30 (Zeitoperator ρ)
Sei $\surd\notin M_{in}\cup M_{out}$. Wir definieren folgenden Operator ρ:

$$\rho : ({M_{in}}^{\omega} \xrightarrow{s} {M_{out}}^{\omega}) \rightarrow ({M_{in}}^{\overline{\infty}} \xrightarrow{p} {M_{out}}^{\overline{\infty}})$$
$$\rho(f) \stackrel{def}{=} \{\ g \mid \forall in{\in}{M_{in}}^{\overline{\infty}}.\ \diamond(g.in) = f(\diamond in)\ \} \qquad (\Box)$$

Jeder Funktion f wird so eine nichtleere Menge gezeiteter stromverarbeitender Funktionen zugeordnet. Diese wird in [BS94] als „weakly time dependent“ bezeichnet. Der Zeitoperator ρ läßt sich durch

$$\rho(F) \stackrel{def}{=} \{\ g \mid \exists f{\in}F.\ \forall in{\in}{M_{in}}^{\overline{\infty}}.\ \diamond(g.in) = f(\diamond in)\ \}$$

punktweise auf Mengen erweitern. Damit können wir einem buchstabierenden Automaten eine gezeitete denotationelle Semantik in Form von gezeiteten stromverarbeitenden Funktionen definieren.

Definition 5.31 (Gezeitete denotationelle Semantik)
Für einen buchstabierenden Automaten A definieren wir folgende gezeitete Semantik:

$$[\![A]\!]^t \stackrel{def}{=} \rho([\![A]\!]) \qquad (\Box)$$

Ein Automat erhält als implizit gezeitete Semantik die Menge aller gezeiteten stromverarbeitenden Funktionen, deren Zeitabstraktion in der ungezeiteten Semantik enthalten ist. Die Ausgabe einer Transition ist gegenüber der Eingabe um eine nicht festgelegte Zeitdauer, aber um mindestens eine Zeiteinheit verzögert, da alle gezeiteten Funktionen Elemente der Menge ${M_{in}}^{\overline{\infty}} \xrightarrow{p} {M_{out}}^{\overline{\infty}}$ sind. Besteht die Ausgabe einer Transition aus mehr als einer Nachricht, so kann sie sich über mehrere Zeiteinheiten erstrecken. Insbesondere erstrecken sich unendliche Ausgaben einer Transition über alle restlichen Zeitintervalle und damit unendlich lange.

5.3.3 Bündel von Kanälen

Bisher haben wir uns darauf beschränkt, Einheiten zu modellieren, die nur einen Ein- und einen Ausgangskanal besitzen. In diesem Abschnitt diskutieren wir, wie sich buchstabierende Automaten auf mehrere Kanäle verallgemeinern lassen.

Die denotationelle Semantik läßt sich auf einfache Weise auf Bündel von Ausgabekanälen erweitern, indem bei der Transitionsrelation δ eines Automaten (S,M,δ,I) statt der Ausgabe $out{\in}M^{\omega}$ Bündel von Ausgaben der Form B^{Ω} zugelassen werden.

Die Anzahl der Eingabekanäle kann leicht erweitert werden, wenn eine synchrone (zeitgleiche) Verarbeitung der Eingabezeichen gefordert wird, wie das etwa bei Hardwarespezifikationen ([FUC94]) der Fall ist. Dann wird ein Bündel von Strömen B^{Ω} isomorph zu einem Strom von gebündelten Elementen $(B{\rightarrow}M)^{\omega}$ behandelt, wodurch die Erweiterung auf die gegebene Theorie zurückgeführt werden kann.

Findet keine synchrone Verarbeitung statt, so können die ankommenden Zeichen der verschiedenen Eingabekanäle einzeln verarbeitet werden. Liegen mehrere Zeichen auf verschiedenen Kanälen an, so hat der Automat die Auswahl. Es entsteht eine weitere Art von

Nichtdeterminismus, die bisher nicht vorhanden war. Ein damit entstehendes Problem ist die Frage nach der Fairneß dieser Auswahl, die zu verschiedenen Semantikvarianten führt. Eine Möglichkeit wäre hier, eine weitere Komponente ähnlich der Fairneß-Mengen der I/O-Automaten [LS89, JON87] einzuführen, in der explizit Fairneß-Forderungen für die Auswahl des Eingabekanals gestellt werden können.

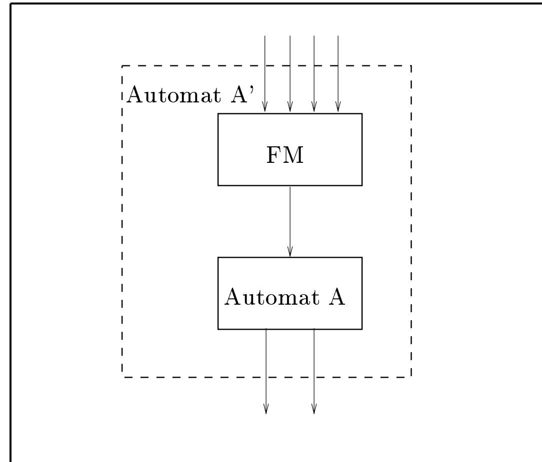


Abbildung 5.7: Fair-Merge von Bündeln von Eingabekanälen

Bündel von Eingabekanälen können auch verarbeitet werden, indem ihre Sequentialisierung einer vorgeschalteten Fair-Merge-Komponente überlassen wird. Wir realisieren damit einen Automaten mit Bündeln von Eingabekanälen wie in Abbildung 5.7 angegeben und trennen so die Fair-Merge-Problematik von der Automatensemantik. Wir spezifizieren die Fair-Merge-Komponente zum Beispiel durch den in Tabelle 5.8 gegebenen Taktautomaten. Es ist sinnvoll, die eigentliche Nachricht mit dem Kanalnamen des Ursprungskanals zu qualifizieren, damit auch diese Information im Automaten verwendet werden kann.

Obige Techniken der Erweiterung auf Bündel besitzen einige Nachteile, die sich in einem gezeiteten Umfeld umgehen lassen. Nicht zuletzt deshalb werden nachfolgend die Ergebnisse dieses Kapitels auf gezeitete stromverarbeitende Funktionen übertragen, weil sich damit zum Beispiel faires Mischen von Strömen direkt formulieren läßt.

5.3.4 Taktautomaten

Nicht zuletzt weil das Systemmodell mit gezeiteten Strömen formalisiert ist, ist es sinnvoll, eine gezeitete Variante buchstabierender Automaten zu definieren. Wir definieren deshalb die bereits im Systemmodell verwendeten Taktautomaten unter Rückführung auf die denotationelle Semantik buchstabierender Automaten.

Taktautomaten beruhen auf der Idee, eine Transition nicht als Eingabe eines Zeichens und Ausgabe der darauf folgenden Reaktion zu sehen, sondern als Fortschritt einer Zeit-

einheit zu betrachten, die die gesamte Eingabesequenz dieser Zeiteinheit verarbeitet und dabei eine Ausgabesequenz von Zeichen ausgibt.

Diese Form der Automaten nennen wir aufgrund ihrer taktsynchronen Arbeitsweise *Taktautomaten*. Taktautomaten werden im Systemmodell aus Kapitel 3 als State-Box-Sichten verwendet, um die Verbindung zwischen dem Verhalten einer Komponente und deren Zustand herzustellen.

Syntax von Taktautomaten

Wir definieren eine abstrakte Syntax für *Taktautomaten*, die auch gleich eine Verallgemeinerung auf mehrere Eingabe- und Ausgabekanäle vornimmt. Wir verwenden dabei die in Abschnitt A.3 definierten Strombündel $I^{\overline{\Sigma}}$ für $I \subseteq B$ und Mengen von Nachrichten $M_b, b \in B$ die keinen Tick enthalten. Die Menge I^{Φ} besteht aus allen Strombündeln mit nur endlichen Strömen auf jedem Kanal.

Definition 5.32 (Taktautomat)
Ein Taktautomat ist ein Fünftupel $(S,I,O,\delta,Init)$, *bestehend aus*

- *einer Menge von Zuständen* S,
- *einer Menge von Eingabekanalnamen* I,
- *einer Menge von Ausgabekanalnamen* O,
- *einer Zustandsübergangsrelation* $\delta \subseteq S \times I^{\Phi} \times S \times O^{\Phi}$ *und*
- *einer Menge* $Init \subseteq S \times O^{\Phi}$ *von Paaren von Anfangszuständen und initialen Ausgaben.* (□)

Zur Kennzeichnung von Taktautomaten schreiben wir auch $(S,I,O,\delta,Init)^{ta}$.

Totalität und Determinismus für Taktautomaten werden analog zu buchstabierenden Automaten definiert (4.2,4.3). Wir gehen nachfolgend davon aus, daß alle Taktautomaten total sind: $\forall s \in S, m \in I^{\Phi}.\ \delta(s,m,*,*)$. Damit entfällt eine Totalisierung ähnlich dem *complete*-Operator. Ein entsprechender Operator kann leicht definiert werden.

Denotationelle Semantik von Taktautomaten

Der Unterschied zwischen einem Taktautomaten und einem buchstabierenden Automaten aus Definition 5.32 besteht vor allem in der Verarbeitung und Ausgabe von Strombündeln und in der Transitionsrelation, die jetzt endliche Worte aus I^{Φ} pro Transition verarbeitet, statt bisher einzelne Zeichen. Außerdem gibt es bei den Taktautomaten keine unendlichen Ausgaben in einer Zeiteinheit.

Wir interpretieren ein Initialelement als Paar, das aus einem Startzustand und einer für die erste Zeiteinheit bestimmte initiale Ausgabe besteht. Pro Zeiteinheit wird genau eine Transition durchgeführt. Dabei wird die Eingabe dieser Transition, bestehend aus einer

endlichen Sequenz von Nachrichten, verarbeitet und die Ausgabe dieser Transition in der nachfolgenden Zeiteinheit ausgegeben. Dadurch entstehen als Semantik gezeitete stromverarbeitende Funktionen. Wir führen die denotationelle Semantik von Taktautomaten auf die denotationelle Semantik von buchstabierenden Automaten zurück, indem wir ein Isomorphie γ zwischen $B^{\overline{\Sigma}}$ und $(B^{\Phi})^{\omega}$ ausnutzen und ein Bündel endlicher Ströme aus B^{Φ} (vorübergehend) als einzelnes Zeichen interpretieren. Diese Isomorphie wird wie folgt definiert:

$$\gamma : B^{\overline{\Sigma}} \to (B^{\Phi})^{\infty}$$
$$\forall a{\in}B^{\Phi}, l{\in}B^{\overline{\Sigma}}.\ \gamma(a \,\hat{}\, \surd \,\hat{}\, l) = \langle a\rangle \,\hat{}\, \gamma(l)$$

Wir definieren die Menge längenerhaltender stromverarbeitender Funktionen als

$$(I^{\Phi})^{\omega} \xrightarrow{l} (O^{\Phi})^{\omega} \stackrel{def}{=} \{\ f{\in}(I^{\Phi})^{\omega} \xrightarrow{s} (O^{\Phi})^{\omega} \mid \forall s.\ \#s = \#f.s\ \}.$$

Dann kann γ durch folgende Definition auf Funktionen erweitert werden[3]:

$$\gamma : (I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}}) \to ((I^{\Phi})^{\omega} \xrightarrow{l} (O^{\Phi})^{\omega})$$
$$\forall l{\in}I^{\overline{\Sigma}}, n{\in}\mathbb{N}\cup\{\infty\}.\ (\gamma.f)((\gamma.l){\downarrow}n) = (\gamma(f.l)){\downarrow}n$$

Aus der Definition A.11 schwach gezeiteter Funktionen folgt die Wohldefiniertheit und die Eindeutigkeit dieser Funktion γ. γ ist daher injektiv. Ihr Bild $\gamma(I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}})$ ist gerade die Menge längenerhaltender stromverarbeitender Funktionen und daher ist γ auch hier eine Isomorphie. Wir erweitern γ auf mit Zuständen parametrisierte Funktionen und punktweise auf Mengen von Funktionen. γ bleibt auch hier jeweils eine Isomorphie. Eine analoge Isomorphie besteht zwischen der Menge (stark) gezeiteter Funktionen und um eins verlängernden stromverarbeitenden Funktionen. Wir bezeichnen auch diese mit γ.

Wird ein Taktautomat als buchstabierender Automat interpretiert, so besteht seine Eingabemenge aus allen endlichen Eingabesequenzen I^{Φ} und die Ausgabemenge analog aus allen endlichen Ausgabesequenzen O^{Φ}. Dadurch wird δ zu einer Transitionsrelation, bei der genau ein Zeichen verarbeitet wird und genau ein Zeichen in der Ausgabe jeder Transition erscheint.

Definition 5.33 (Denotationelle Semantik für Taktautomaten)
Für einen Taktautomaten (S,I,O,δ,Init) definieren wir folgende Taktsemantik:

$$[\![(S,I,O,\delta,Init)]\!]^{ta} \stackrel{def}{=} \gamma^{-1}.[\![(S,I^{\Phi},O^{\Phi},\delta,Init)]\!]^{c}$$

(□)

Die Abstützung der denotationellen Semantik von Taktautomaten auf die Semantik buchstabierender Automaten erlaubt uns die Übertragung der dortigen Ergebnisse. Auch kann die rekursive Semantikdefinition für buchstabierende Automaten auf Taktautomaten übertragen werden.

[3]Mit $s{\downarrow}n$ wird der Präfix von s mit der Beobachtungsdauer $n{\in}\mathbb{N}\cup\{\infty\}$ definiert; siehe Abschnitt A.3

Proposition 5.34 (Eigenschaften der Semantik $[\![.]\!]^{ta}$)
Sei (S,I,O,δ,Init) ein vollständiger Taktautomat. Dann gelten folgende Eigenschaften:

1. *Die Taktsemantik ist wohldefiniert.*
2. *Die Taktsemantik ist für jeden Taktautomat mit nichtleerer Initialmenge konsistent, also nicht leer.*
3. *Ist ein Taktautomat total und deterministisch, so besitzt seine Semantik genau ein Element.*
4. *Die Taktsemantik für Taktautomaten läßt sich wie in Definition 5.12 direkt formulieren.*

$$[\![(S,M,\delta,I)]\!]^{ta} = \{\ g \in I^{\overline{\Sigma}} \xrightarrow{p} O^{\overline{\Sigma}} \mid \\ \exists h \in [\![(S,M,\delta,I)]\!]^{tap},\ (s_i,out_i) \in I.\ \forall in.\ g(in) = out_i \,\hat{}\, \surd \,\hat{}\, h(s_i,in)\ \}$$

wobei $[\![.]\!]^{tap}$ die größte Menge schwach gezeiteter stromverarbeitender Funktionen charakterisiert, die folgender Gleichung genügt:

$$[\![(S,M,\delta,I)]\!]^{tap} \stackrel{def}{=} \{\ h \in S \times I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}} \mid \\ \forall m \in I^{\Phi}, s. \exists t, out \in O^{\Phi}. \delta(s,m,t,out) \wedge \exists h' \in [\![(S,M,\delta,I)]\!]^{tap}. \\ \forall in \in I^{\overline{\Sigma}}.\ h(s,m \,\hat{}\, \surd \,\hat{}\, in) = out \,\hat{}\, \surd \,\hat{}\, h'(t,in)\ \}$$

(□, Beweis siehe C.17**)**

Ein Taktautomat ist damit ein einfacher Mechanismus für die Beschreibung von gezeitetem Verhalten. Ein Beispiel ist die Spezifikation des fairen Mischens. Wir zeigen dies in Abbildung 5.8, die eine Spezialisierung des fairen Mischens ohne Verzögerung (von mehr als einer Zeiteinheit) darstellt. Dieser Misch-Operator arbeitet korrekt, solange er nur endlich viele Eingabenachrichten pro Zeiteinheit bekommt. Andernfalls geht er in einen Fehlerzustand e über und hat dort beliebiges Verhalten.

FairMerge: $S=\{s,e\}$, I, $O=\{o\}$, $Init=\{(s,\varepsilon)\}$

S_{old}	I^{Φ}	Bedingung	S_{new}	O^{Φ}
s	In	$Out \in merge(In) \cap M^*$	s	$o \mapsto Out$
s	In	$merge(In) \in M^{\infty}$	e	$*$
e	$*$		e	$*$

Abbildung 5.8: Fair-Merge als Taktautomat

Die Hilfsfunktion *merge*, die endliche Ströme zu einem Strom mischt, kann dabei wie folgt definiert werden:

$$merge(In) = \{\ T \in M^{\omega} \mid \exists org \in I^{\omega}.\ \forall i \in I.\ In.i = Select(T,org,i)\ \}$$

$$Select(m \,\hat{}\, ms, or \,\hat{}\, ors, o) \stackrel{def}{=} \left\{ \begin{array}{ll} m & \texttt{if } or=o \\ \varepsilon & \texttt{if } or \neq o \end{array} \right\} \hat{}\, Select(ms,ors,o)$$

Als ein Beispiel für Hardwarestrukturen ist in 5.9 ein Taktautomat angegeben, der einen Zähler mit Timeout realisiert, der nach Initialisierung mit einer Wartezeit bis zu zwei Zeiteinheiten einen Timeout abgibt.

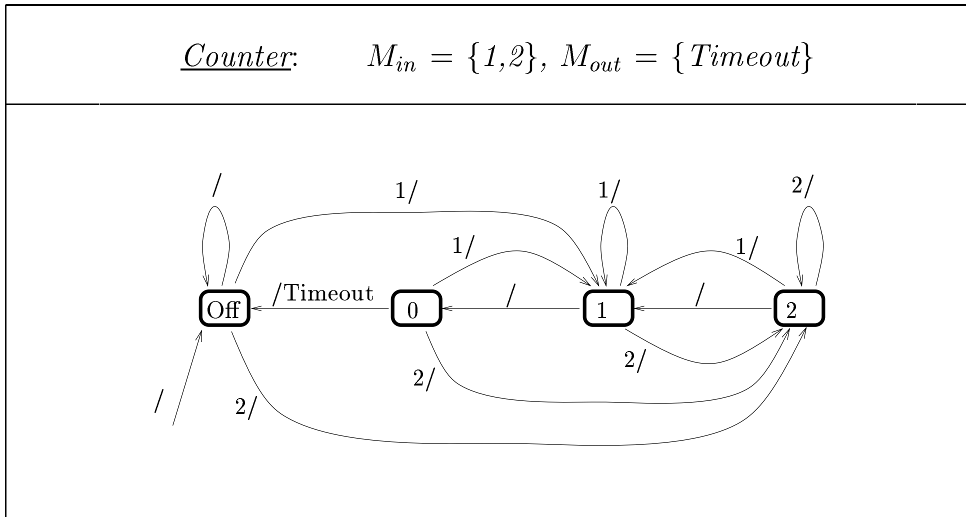


Abbildung 5.9: Zeitsynchroner Taktautomat *Counter*

Dieser Automat ist als Taktautomat hochgradig partiell, da er keine Aussagen über die Verarbeitung von mehr als einem Zeichen pro Zeiteinheit trifft. Unter der in der Hardwaremodellierung üblichen Annahme, daß pro Zeiteinheit (Hardwaretakt) und Kanal höchstens eine Nachricht (Signal) ankommt, ist dieser Automat ausreichend definiert.

5.3.5 Fairneß bei der Auswahl von Transitionen

Der Automat in Abbildung 5.10 erlaubt die nichtdeterministische Auswahl einer passenden Transition, wenn mehr als eine Transition zur Verfügung steht.

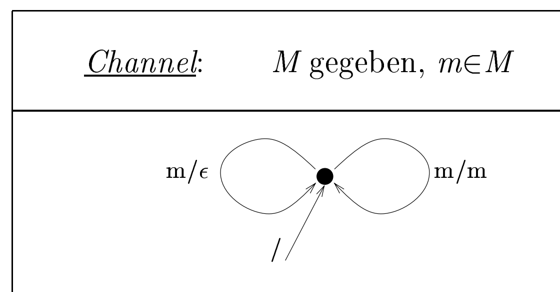


Abbildung 5.10: Fehlerhafter Übertragungskanal

Gemäß der von uns definierten Semantik kann ein mit *Channel* spezifizierter Übertragungskanal immer die Transition m/ε auswählen und so niemals eine Nachricht übertragen. Deshalb ist *Channel* nicht geeignet, um einen Übertragungskanal zu beschreiben,

der sowohl Nachrichten verliert als auch welche überträgt. Um dies zu erreichen, können auch hierfür Fairneß-Mengen als zusätzliche Komponente in Automaten aufgenommen werden. Nachteil dieser Fairneß-Mengen ist jedoch, daß der Automat damit nur noch zur Beschreibung von Nichtdeterminismus, nicht jedoch für Unterspezifikation verwendet werden kann. Denn im Sinne der Präzisierung einer Spezifikation ist es nicht mehr möglich, eine der beiden Transitionen zu entfernen und damit eine Spezialisierung zum Beispiel auf einen sicheren Übertragungskanal durchzuführen. Genau dies ist aber eines der Hauptanliegen des nächsten Kapitels 6.

Um dennoch eine faire Auswahl von Transitionen zu erzwingen, kann mit Hilfe einer „Prophecy"-Variable eine natürliche Zahl in den Zustand aufgenommen werden. Diese Zahl beschreibt, im Sinne einer Vorhersage, wann die nächste Übertragung stattfindet. Sie wird bei jeder nicht stattgefundenen Übertragung verkleinert und nach einer solchen neu gewählt (siehe dazu auch Abbildung 5.6). Eine Verfeinerung von *Channel* in einen fairen Übertragungskanal *ChannelF* ist in Tabelle 5.11 zu sehen.

ChannelF: M, $S = \mathbb{N}$, $I = \{(n,\varepsilon) \mid n \in \mathbb{N}\}$

S_{old}	M	S_{new}	Out
$n+1$	m	n	ε
0	m	n	$<m>$

Abbildung 5.11: Fairer, fehlerhafter Übertragungskanal

Mit den in Kapitel 6 definierten Verfeinerungsschritten kann gezeigt werden, daß der Automat *ChannelF* eine Verfeinerung von *Channel* ist. Dazu wird die Zustandsmenge von einem Zustand in *Channel* zu $\mathbb{N}$ in *ChannelF* verfeinert und alle Transitionen, die nicht in obiger Tabelle 5.11 angegeben sind, entfernt. Dies ist aufgrund der in den Propositionen 6.13 und 6.5 angegebenen Regeln möglich. Wir haben mit einer solchen Verfeinerung genau den Übertragungskanal ausgeschlossen, der jede Nachricht verliert.

Natürlich läßt sich *ChannelF* aus Abbildung 5.12 weiter verfeinern zu dem sicheren Übertragungskanal, der alle Nachrichten überträgt.

Channel kann durch einfache Entfernung der Transition m/ε zu *ChannelS* verfeinert werden. Die Verfeinerung von *ChannelF* zu *ChannelS* kann durch Reduktion der Initialmenge auf $\{(0,\varepsilon)\}$, Entfernung aller Transitionen $\delta(0,m,n,m)$, $n \geq 1$ und der anschließenden Entfernung aller unerreichbaren Zustände n, $n \geq 1$ erreicht werden.

Mit der expliziten Aufnahme einer Fairneß beschreibenden Komponente hätten wir diese Verfeinerungen und die effektive Überprüfung ihrer korrekten Anwendung wesentlich erschwert. Weiterhin zwingt uns Fairneß, alternative Transitionen im Sinne von Nichtdeterminismus und nicht als Unterspezifikation zu interpretieren. Das Resultat 5.29 zeigt uns darüberhinaus, daß Fairneß auch durch unsere Automaten ausgedrückt werden kann.

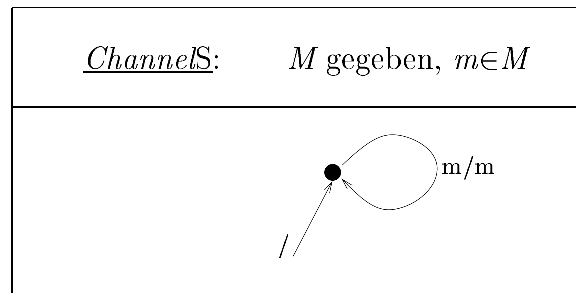


Abbildung 5.12: Sicherer Übertragungskanal

Daher ist es nicht sinnvoll, eine Fairneß-Komponente in die Automatenbeschreibungen aufzunehmen.

5.4 Verwandte Ansätze

In diesem Kapitel wurde das erste Mal systematisch beschrieben, wie Transitionssysteme als Definition stromverarbeitender Funktionen verstanden werden können. Dennoch stammen einige Ideen aus FOCUS-Arbeiten: [Betal93, BRO93, FUC94, SPI94]. Daß die mathematische Fundierung von Automaten weiterhin ein interessantes Thema ist, zeigt die Anzahl neuerer Veröffentlichungen im Umfeld dieses Themas. Ein Beispiel ist [BRO95], in der Zustandsmaschinen (Automaten) als Modell zur Beschreibung von Komponentenverhalten vorgeschlagen werden.

Weitere Transitionssystemtechniken lassen sich etwa bei SDL ([HOG89]) (siehe auch deren FOCUS-basierte Formalisierung in [BRO91]) oder bei Prozeßalgebren wie CSP ([HOA85]) und CCS ([MIL80]) finden.

Jonsson hat in [JON89] Transitionssysteme definiert, die ebenfalls mit den Taktautomaten verwandt sind. Sie verarbeiten und produzieren pro Transition eine endliche Menge von Eingaben und Ausgaben. Diese Transitionen sind aber nicht zeitsynchron. Das heißt es gibt kein unterliegendes Zeitkonzept für die Transitionen. Um Lebendigkeit des Transitionssystems zu sichern, werden daher genau wie bei den I/O-Automaten zusätzlich Fairneß-Mengen angegeben.

Den Taktautomaten sehr verwandt ist die daraus hervorgegangene Notation der „Timed Port Automata“ aus [GR95]. Dort ist allerdings keine konstruktive Semantikdefinition angegeben, die die Existenz von Funktionen in der Semantik und damit deren Konsistenz sofort erkennen läßt.

I/O-Automaten

Verwandt zu den hier definierten Automaten ist das Konzept der I/O-Automaten (siehe [JON87, KAH74, LS89]). Es beschreibt das Verhalten einer Einheit durch die von ihr

durchgeführten *Aktionen*. Bei I/O-Automaten steht mehr der Aktionsbegriff im Vordergrund, während bei unseren Automaten der Nachrichtenbegriff (wir benennen Nachrichten auch Zeichen) eine Rolle spielt. Ein Vergleich beider Techniken wird erst möglich, wenn das Versenden und Empfangen von Nachrichten als einzige Aktionsarten betrachtet werden. Ein I/O-Automat

$$(M_{in}, M_{out}, S, s_0, R, F)$$

besteht aus einer Menge von Eingabeaktionen M_{in}, einer Menge von Ausgabeaktionen M_{out}, einer Menge von Zuständen S, einem Anfangszustand s_0, einer Transitionsrelation R, die darauf eingeschränkt ist ein Zeichen zu lesen, auszugeben oder nichts extern Erkennbares zu tun:

$$R \subseteq S \times (M_{in} \cup M_{out} \cup \{\alpha\}) \times S$$

und einem endlichen System F von Fairneß-Mengen, die Teilmengen von R sind. Andere Varianten erlauben zusätzlich die Angabe einer Menge interner Aktionen. Eine wesentliche Nebenbedingung, die von einem I/O-Automaten gefordert wird, ist die „input enabledness“. Sie besagt, daß der Automat eine ankommende Eingabe jederzeit verarbeiten kann. Dadurch eignen sich I/O-Automaten ebenfalls zur Charakterisierung stromverarbeitender Funktionen, wie dies in [BDDW91] mit verschiedenen Semantiken diskutiert wurde. Das hier vorgelegte Konzept ist gegenüber den I/O-Automaten abstrakter, da es mit einem gröberen Zustandskonzept auskommt. Bei I/O-Automaten muß nach der Eingabe oder Ausgabe jedes Zeichens ein neuer, expliziter Zustand eingenommen werden. Dadurch spiegeln die Zustände des I/O-Automaten den Verarbeitungsgrad einer Nachricht wider. Unser Zustandskonzept abstrahiert jedoch davon: Unsere Zustände stellen definierte Ruhepunkte dar, die zwischen Verarbeitungschritten auftreten. Die Zustände unserer Automaten spiegeln daher Äquivalenzklassen der Daten unserer modellierten Einheiten wider.

Dennoch kann ein I/O-Automat in einen buchstabierenden Automaten

$$(S, M_{in}, M_{out}, \delta, (s, \varepsilon))$$

übersetzt werden, wobei wir Fairneß-Mengen außer acht lassen. Die Mengen der Eingabe- und Ausgabeaktionen sowie der Zustände bleiben dabei erhalten. Transitionen werden aus Sequenzen q von Transitionen von R gebildet, die mit einer Eingabeaktion beginnen und dann nur noch Ausgabeaktionen oder interne Aktionen besitzen, wobei unendlich viele interne Aktionen nur auftreten dürfen, wenn auch unendlich viele Ausgabeaktionen dabei sind:

$$\begin{aligned} steps(R) \stackrel{def}{=} \{\ & q=((u_i,c_i,v_i)_i) \in R^\omega.\ c_1 \in M_{in} \wedge \#(M_{in} ©(c_i)_i)=1 \wedge \\ & (\#(\alpha ©(c_i)_i)=\infty \Rightarrow \#(M_{out} ©(c_i)_i)=\infty)\ \} \end{aligned}$$

Eine Transition der Transitionsrelation δ beginnt beim Startzustand der Sequenz, verarbeitet das Eingabezeichen, gibt die Sequenz der Ausgabezeichen aus und endet beim Endzustand der Sequenz beziehungsweise bei einem beliebigen Zustand, wenn dieser nicht existiert.

$$\begin{aligned} \delta \stackrel{def}{=} \{ & (s,m,t,out) \mid \exists\ q=((u_i,c_i,v_i)_i) \in steps(R). \\ & s = u_1 \wedge (\#q<\infty \Rightarrow t=v_{\#q}) \wedge \\ & m = c_1 \wedge out = (M_{out} ©(c_i)_i)\ \} \end{aligned}$$

Ein Nachteil dieser Umsetzung ist es, daß endliche I/O-Automaten im allgemeinen dennoch in Automaten mit unendlich vielen Transitionen übersetzt werden. Außerdem wird die Auswahl der als nächstes auszuführenden Aktion bei I/O-Automaten später getroffen, als beim buchstabierenden Automaten. Dies spielt aber bei der Beobachtung des Verhaltens einer Einheit keine Rolle, da der Beobachter nicht sehen kann, wann eine Entscheidung getroffen wird.

Auch die umgekehrte Transformation eines buchstabierenden Automaten in einen I/O-Automaten ist möglich. Die Grundidee beruht dabei darauf, unter Einführung neuer Zustände die Transitionen mit längeren Ausgabesequenzen aufzubrechen und einzeln abzuarbeiten, wobei die noch nicht ausgegebenen Zeichen als Vorausschau oder Vorhersage im Zustand aufgehoben werden. Eine solche Vorhersage wird gelegentlich in Form der bereits mehrfach erwähnten „Prophecy"-Variablen verwendet.

Buchstabierende Automaten besitzen ein abstrakteres Zustandskonzept als I/O-Automaten, da sie keine Zwischenzustände für die Ausgabe von Zeichensequenzen benötigen.

5.5 Zusammenfassung

In diesem Kapitel haben wir für drei Klassen von buchstabierenden Automaten eine Semantik in Form eines Prädikates über stromverarbeitenden Funktionen definiert. Zunächst wurde für die deterministischen, totalen Automaten die Semantik $[\![.]\!]^d$ definiert und gezeigt, daß diese konsistent und eindeutig ist. Das heißt, daß einem deterministischen, totalen Automaten genau eine stromverarbeitende Funktion zugeordnet wird. Dann wurde die Semantik $[\![.]\!]^c$ für totale Automaten definiert. Mit Hilfe der vorherigen Ergebnisse wurde gezeigt, daß ein totaler Automat konsistent ist. Einzige Ausnahme dazu ist der Spezialfall eines totalen Automaten mit einer leeren Menge von Initialelementen.

Für allgemeine Automaten war zu überlegen, was es bedeutet, wenn in einem Zustand kein Zeichen akzeptiert werden kann, da in unserem asynchronen Modell eine stromverarbeitende Einheit an sie gesandte Nachrichten immer verarbeiten muß. Wir modellieren diesen Sachverhalt durch Chaos. Die explizite Einführung eines Fehlerzustands und die Totalisierung des Automaten mit beliebigen Transitionen zur Modellierung von Chaos wird durch die Operation *complete* ermöglicht. Damit wird die allgemeine Semantik $[\![.]\!]$ auf den totalen Fall zurückgeführt und Eigenschaften totaler Automaten auf allgemeine Automaten übertragen. Im Satz 5.16 sind diese zusammengefaßt.

Der Vergleich mit der operationellen Semantik in Satz 5.22 zeigt, daß die Definition der denotationellen Semantik mit der in der operationellen Semantik gefaßten Vorstellung der Verarbeitung von Transitionen übereinstimmt.

Weitere Ergebnisse zeigen, daß das von uns verwendete Automatenkonzept mächtig genug ist, um für jede Menge stromverarbeitender Funktionen einen Automaten mit genau dieser Menge als Semantik anzugeben.

Gerüstet mit den in diesen Kapiteln dargestellten Techniken entwickeln wir nun einen Verfeinerungskalkül für Automaten.

Kapitel 6

Verfeinerungstechniken für Automaten

In diesem Kapitel beschäftigen wir uns mit Verfeinerungstechniken, die auf einen buchstabierenden Automaten angewendet werden können, um eine Detaillierung des beschriebenen Verhaltens zu erreichen. Dabei entsteht ein Verfeinerungskalkül für Automaten, dessen Korrektheit bewiesen wird.

Im ersten Abschnitt dieses Kapitels wird eine Verfeinerungsrelation definiert und anhand eines Beispiels gezeigt, welche Verfeinerungsschritte in den Abschnitten zwei bis fünf eingeführt werden. Im sechsten Abschnitt werden diese Verfeinerungsschritte zu einem Verfeinerungskalkül kombiniert. Der siebte Abschnitt behandelt verwandte Ansätze, der achte gibt eine Zusammenfassung des Kapitels.

6.1 Verfeinerungsrelation für Automaten

Verfeinerungstechniken werden verwendet, um Spezifikationen von Komponenten präziser an spezielle Gegebenheiten anzupassen oder durch zusätzliche Eigenschaften zu erweitern. So kann unterspezifiziertes Verhalten verfeinert werden, aber auch gegebene Komponenten durch Anpassung wiederverwendet werden. Es ist daher notwendig, für den Softwareentwicklungsprozeß mehrere Verfeinerungstechniken zur Verfügung zu stellen, die als Transformationen auf der gegebenen Beschreibungstechnik dargestellt werden können und eine genau festgelegte Bedeutung besitzen.

Wir nutzen als Verfeinerungsrelation auf der semantischen Ebene die Inklusionsordnung auf Mengen stromverarbeitender Funktionen. Diese spiegelt die Präzisierung von unterspezifiziertem Verhalten wider und ist darüberhinaus transitiv. Auf syntaktischer Ebene definieren wir eine Reihe von Transformationsregeln, deren Korrektheit jeweils gezeigt wird. Die Transitivität der gewählten Verfeinerungsrelation sichert die Kompositionalität der Transformationsregeln. So entsteht ein Verfeinerungskalkül für buchstabierende Automaten.

Dieser Verfeinerungskalkül wird in zwei Arten von Verfeinerungen geteilt. Ein Satz von Regeln beschäftigt sich mit der Verfeinerung des Verhaltens einer Komponente, eine weitere Regel mit der Verfeinerung der Schnittstelle. Entsprechend definieren wir zunächst die Relation der Verhaltensverfeinerung und in Abschnitt 6.5 eine Verallgemeinerung auf Schnittstellen in der von uns benötigten Form.

Definition 6.1 (Verhaltensverfeinerung)
Ein Automat (S',M,δ',I') heißt Verhaltensverfeinerung *(oder kurz* Verfeinerung*) des Automaten (S,M,δ,I), wenn gilt:*

$$[\![(S',M,\delta',I')]\!] \subseteq [\![(S,M,\delta,I)]\!].$$

Eine Transformation T der Form

$$T : (S,M,\delta,I) \rightarrow (S',M,\delta',I')$$

heißt Verfeinerungsschritt*, wenn T(S,M,δ,I) eine Verfeinerung von (S,M,δ,I) ist. Gilt sogar Gleichheit der Semantiken, so heißt die Transformation* semantikerhaltend.

(□)

Proposition 6.2 (Verfeinerung ist transitiv)
Die Verfeinerungsrelation für Automaten ist transitiv. Die funktionale Komposition zweier Verfeinerungsschritte ist wieder ein Verfeinerungsschritt.

(□, **Beweis siehe** C.18)

Eine Verfeinerungsrelation ist eine wichtige Beziehung zwischen Beschreibungseinheiten der Softwareentwicklung. Aber erst deren operative Umsetzung in Form von Transformationsregeln erlaubt es dem Softwareentwickler diese konstruktiv einzusetzen. Besonders wichtig ist der konstruktive Ansatz der Verfeinerungstransformationen, weil er im Gegensatz zu Verfeinerungsrelationen zielgerichtetes Vorgehen nahelegt. Eine Verfeinerungsrelation ist zwar gerichtet, kann aber in beide Richtungen (top-down und bottom-up) angewendet werden.

Transformationsregeln sind oft mit einschränkenden Bedingungen versehen, die gesichert sein müssen, damit es sich tatsächlich um eine Verfeinerung handelt. Diese Bedingungen lassen sich in drei Klassen einteilen:

1. Bedingungen, die auf der Syntax definiert sind und automatisiert entschieden werden können,
2. Bedingungen, die auf der Syntax definiert sind aber nicht automatisiert entschieden werden können, und
3. Bedingungen, die unter Benutzung der Semantik formuliert sind.

Die erste Form von Bedingungen ist für Softwareentwicklungswerkzeuge sehr wichtig, denn genau solche Überprüfungen müssen in Softwareentwicklungswerkzeugen vorgenommen werden. Ist eine Logiksprache involviert, so ist das häufig nicht möglich. Dann müssen Bedingungen angegeben werden, die unter Unterstützung von Logikkalkülen bzw. Theorembeweisern verifiziert werden können. Keinesfalls jedoch dürfen Bedingungen benutzt werden, die unter Benutzung semantischer Eigenschaften definiert sind. In diesem

Fall müssen entsprechende, syntaxbasierte Kontextbedingungen gefunden werden, die hinreichend für die semantischen Bedingungen sind.

Eine Transformation ist erst dann als konstruktiv anzusehen, wenn ihre Kontextbedingungen syntaktisch behandelt werden können. Es gibt daher oftmals einen gewissen Konflikt, zwischen dem Interesse Transformationen möglichst allgemein und mächtig zu definieren, dafür aber schwer überprüfbare Bedingungen zu haben, und der möglichst automatisierbaren Überprüfung der Korrektheit von Verfeinerungstransformationen, die die Mächtigkeit der Transformationen oftmals einschränken. Für die Praxis ist es im allgemeinen wichtig, einfache und effektive Verfeinerungstransformationen zu verwenden und dafür auf deren Vollständigkeit zu verzichten.

Die Menge der Transformationen dieses Kapitels definiert einen Verfeinerungskalkül der besonders für eine graphische Anwendung geeignet ist. Die abstrakte Syntax und die elementaren Verfeinerungsschritte sind so gewählt, daß die in Kapitel 8 definierten Automatendokumente und die zugehörigen elementaren Verfeinerungsschritte zu möglichst einfachen, syntaxbasierten Kontextprüfungen führen. Demgegenüber sind die Beweise zur Korrektheit der Transformationsregeln oft komplex. Zugunsten der Einfachheit und Eleganz der Transformationsregeln nehmen wir auf die Komplexität der Beweise keine Rücksicht.

Die nachfolgenden Verfeinerungsschritte entfalten ihre volle Wirkung oft erst durch geeignete Komposition zu mächtigeren Verfeinerungsschritten. Nach Proposition 6.2 führt die Komposition von Verfeinerungsschritten immer wieder zu einem Verfeinerungsschritt.

Beispiel *Figure*

Bevor wir jedoch die Verfeinerungsschritte definieren, demonstrieren wir an der Klasse *Figure* und ihrer Subklasse *2D−Figure*, wie diese Schritte angewendet werden können und damit das Verhalten von Komponenten während einer Softwareentwicklung verfeinert und vererbt wird.

Wir nehmen an, daß eine Strukturbeschreibung (zum Beispiel Objektmodell) gegeben ist, die den in Abbildung 6.1 angegebenen Ausschnitt enthält. Vereinfacht gesehen enthält ein Agent der Klasse *Figure* Nachrichten von Agenten der Klasse *Terminal* und sendet Nachrichten zu einem Agenten der Klasse *Window*. Der *Terminal*-Agent stellt die für einen *Figure*-Agenten bestimmten Eingaben des Benutzers an diesen Agenten weiter. Der *Figure*-Agent kontrolliert seinerseits die Bildschirmanzeige über den Agent *Window*.

Die nachfolgend dargestellte Entwicklung spiegelt einen oft typischen Entwurfsprozeß wider, der zunächst bestimmte Festlegungen trifft (hier etwa die Einführung eines Fehlerzustands) und diese durch spätere Verfeinerungen wieder überflüssig macht. Diese etwas erratischen Verfeinerungen geben uns die Möglichkeit alle wesentlichen Verfeinerungsschritte an diesem Beispiel zu demonstrieren, denn die nachfolgende Entwicklung hätte auch in drei Schritten durchgeführt werden können.

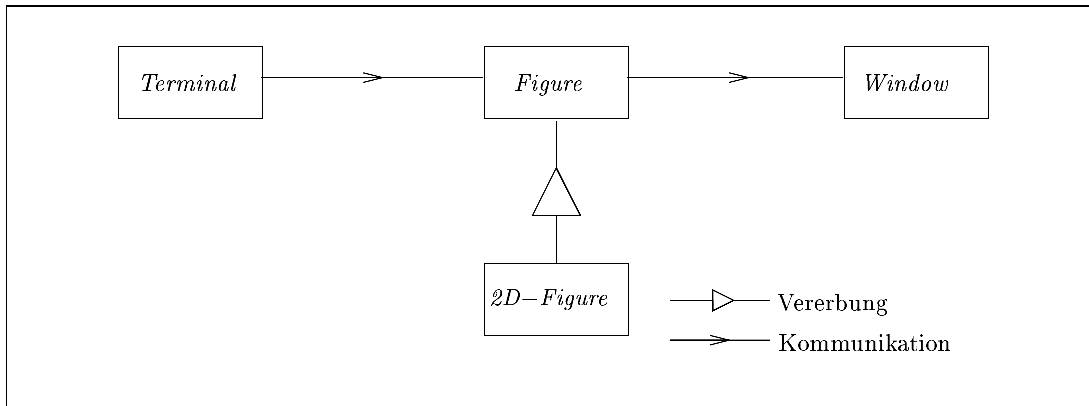
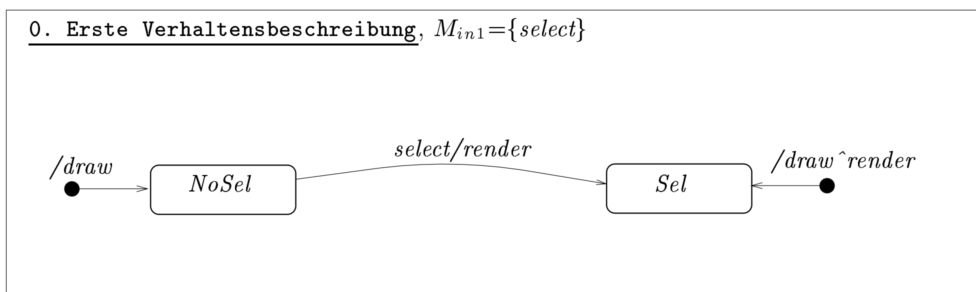
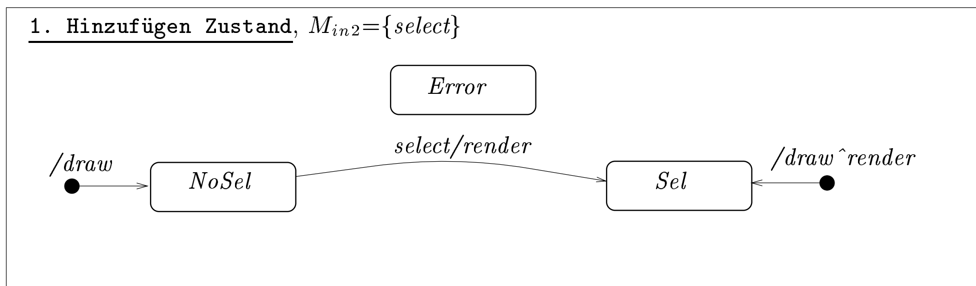


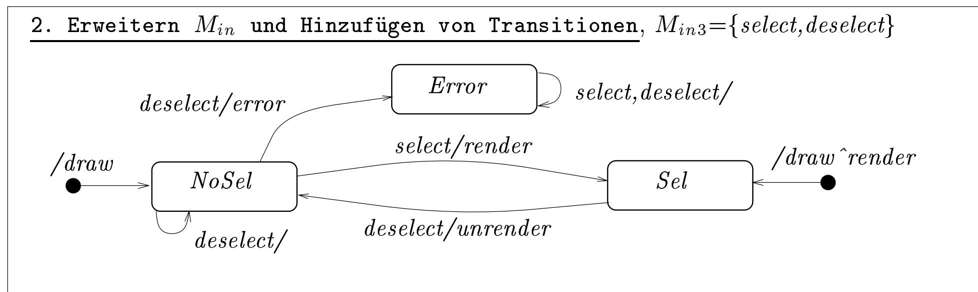
Abbildung 6.1: Strukturausschnitt des *Figure*-Beispiels



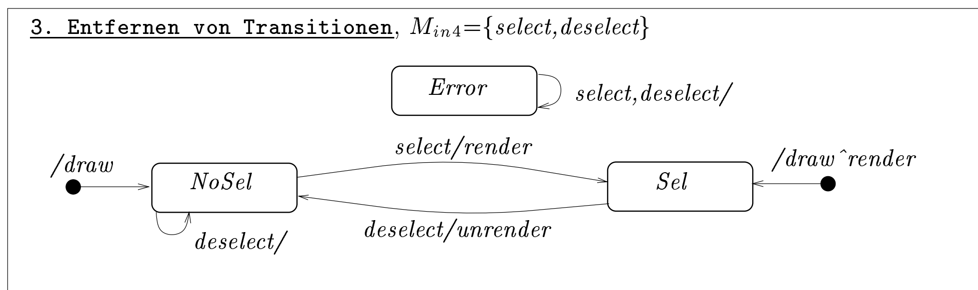
0. Erste Verhaltensbeschreibung: Wir starten unsere Entwicklung mit dem buchstabierenden Automaten $(S_1,M_{in1},M_{out},\delta_1,I_1)$. Die beiden Zustände *Sel* und *NoSel* reflektieren den Selektionszustand einer Figur. Beide sind Initialzustände. Bei Erzeugung der Figur ist diese zu zeichnen (*draw*). Im selektierten Zustand ist der entsprechende Fensterausschnitt hervorgehoben darzustellen (*render*). Die einzige Transition dieses Automaten beschreibt, daß bei einer Selektion im unselektierten Zustand der Fensterausschnitt hervorgehoben wird, und ein Zustandsübergang stattfindet. Das Fehlen einer Transition zur Verarbeitung von *select* im Zustand *Sel* erlaubt Chaos. Damit können wir jederzeit eine robuste Implementierung angeben.



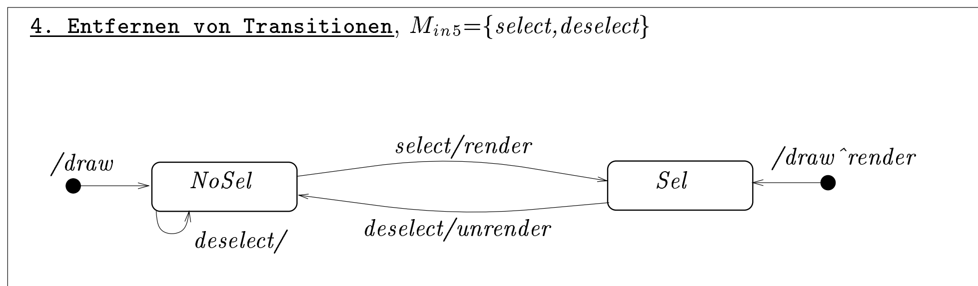
1. Verfeinerungsschritt: Neben der Selektion von Figuren ist auch deren Deselektion wichtig. Wir wissen nicht, ob im *NoSel*-Zustand eine Deselektion sinnvoll ist, und führen daher einen neuen Zustand *Error* ein, der eingenommen werden kann, wenn ein solcher Bedienungsfehler auftritt. Der neue Zustand ist zunächst nicht erreichbar und trägt damit nicht zur Semantikbildung bei.



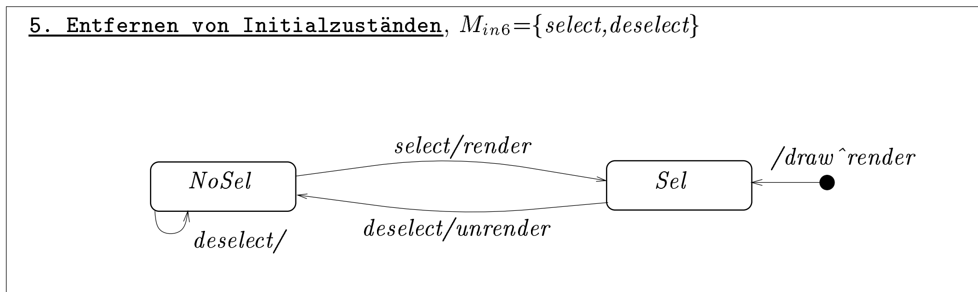
2. Verfeinerungsschritt: Wir führen nun die Nachricht *deselect* für die Deselektion einer Figur ein, und erweitern damit die Eingabemenge. Anschließend fügen wir einige Transitionen hinzu, die das Verhalten einer Figur bei Erhalt einer *deselect*-Nachricht beschreiben. Bei dieser Gelegenheit wird auch eine Transition hinzugefügt, die die Reaktion auf *select* im Fehlerzustand beschreibt. Wir wissen nicht genau, welches Verhalten für *deselect* im *Sel*-Zustand gewünscht ist und geben daher zwei Alternativen an. Eine Transition ignoriert die Nachricht, eine andere Transition führt in den Fehlerzustand. Dies läßt einer Implementierung und jeder Subklasse beide Varianten offen.



3. Verfeinerungsschritt: Der Anwender wünscht generell eine robuste Implementierung. Wir streichen die Möglichkeit, daß *deselect* in den Fehlerzustand führt. Dies ist möglich, weil mit der *deselect*-Schlinge im Zustand *NoSel* eine Alternative zur Verfügung steht.



4. Verfeinerungsschritt: Der Fehlerzustand *Error* ist im Diagramm wieder unerreichbar. Wir beschließen ihn zu streichen. Er hätte also gar nicht eingeführt werden müssen.



5. Verfeinerungsschritt: Als letzten Verfeinerungsschritt für das Verhalten der Agenten der Klasse *Figure* beschließen wir, daß eine Figur, wenn sie erzeugt wird, immer selektiert sein soll. Wir entfernen daher *NoSel* aus der Menge der Initialzustände.

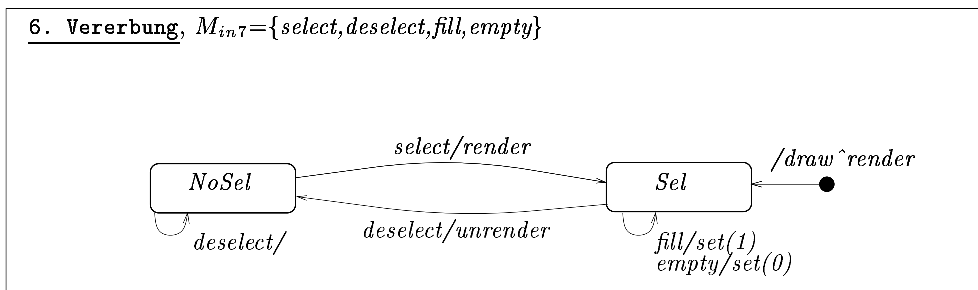
Wir schließen die Entwicklung für Klasse *Figure* hier ab.

Alle sukzessive entwickelten Automaten beschreiben das den Klienten der Klasse *Figure* zugesicherte Verhalten. Jeder Automat verfeinert seinen Vorgänger und enthält so eine detailliertere Beschreibung des Verhaltens von Agenten der Klasse *Figure*.

Klienten der Klasse *Figure* können jede dieser unterschiedlichen Abstraktionsstufen nutzen.

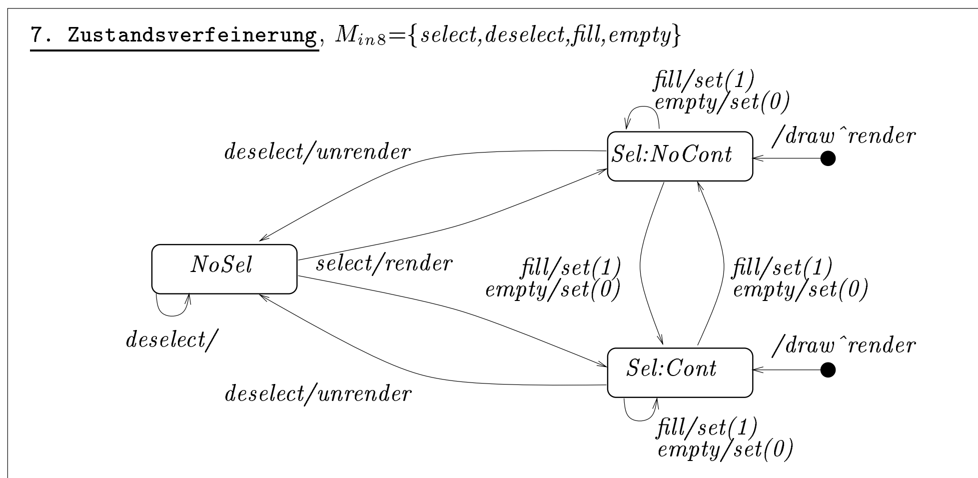
Das Substitutionsprinzip ([WEG90]) für Objekte fordert nun, daß jeder Agent aus der Subklasse *2D−Figure* ebenfalls dieses Verhalten besitzt.

Wir vererben daher die Verhaltensbeschreibung, die durch den letzten Automaten angegeben wird, auf die Klasse *2D−Figure* und spezialisieren hier das Verhalten weiter.

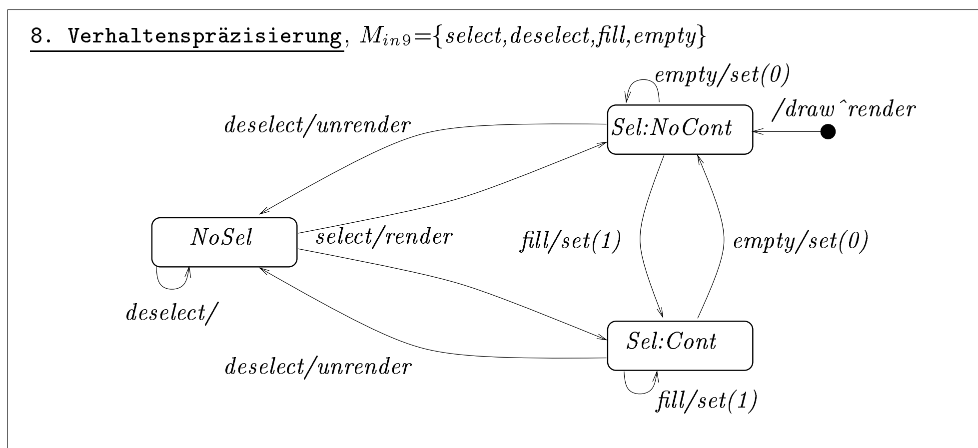


Vererbung und 6. Verfeinerungsschritt: Gleichzeitig mit der Vererbung des Automaten wird dessen Eingabemenge um die Nachrichten *fill* und *empty* erweitert. *2D*-Figuren besitzen eine Fläche, deren Inhalt gefüllt (*fill*) und geleert (*empty*) werden kann. Beide Nachrichten werden im selektierten Zustand akzeptiert und deren verarbeitende Transition verläßt diesen Zustand nicht.

Die Methode *set* wird an den *Window*-Agenten weitergeleitet.



7. Verfeinerungsschritt: Die Reaktion auf *fill* und *empty* kann mit dem momentanen Automatenzustandsraum nicht adäquat wiedergegeben werden. Deshalb führen wir eine Zustandsverfeinerung des Zustands *Sel* in zwei neue Zustände *Sel:NoCont* und *Sel:Cont* durch, die neben der Selektion vermerken, ob der Inhalt der zweidimensionalen Figur gefüllt (*Sel:Cont*) oder leer (*Sel:NoCont*) ist. Dabei werden ankommende und ausgehende Transitionen entsprechend vervielfacht. Zum Beispiel entstehen aus der einen *fill*-Transition vier neue Transitionen. Weil der alte Zustand Initialzustand war, werden beide neuen Zustände ebenfalls Initialzustände.



8. Verfeinerungsschritt: Das Verhalten der Agenten der Subklasse *2D−Figure* wird nun weiter präzisiert, indem ein Initialelement und einige Transitionen entfernt werden. Dadurch wird festgelegt, daß neu erzeugte zweidimensionale Figuren zunächst nicht gefüllt aber selektiert sind, und daß *fill* und *empty* tatsächlich adäquate Zustandsänderungen verursachen.

Die hier beschriebenen Entwicklungsschritte sollten ausreichen, um die Flexibilität und die Mächtigkeit des in diesem Kapitel definierten Verfeinerungskalküls zu demonstrieren. Das Beispiel zeigt, daß damit echte Softwareentwicklungen betrieben werden können, es zeigt aber auch, daß für größere Entwicklungen Werkzeugunterstützung erforderlich ist. Um einen weiteren Schritt in Richtung auf diese Werkzeugunterstützung vorzunehmen,

wird im Kapitel 8 eine konkrete Syntax für Automatendokumente vorgeschlagen, die die Darstellung für solche Verhaltensbeschreibungen auch mit unendlichen Nachrichten-, Zustands- und Transitionsmengen erlaubt. Außerdem wird im Kapitel 8 eine Anbindung der Automatendokumente und des Verfeinerungskalküls an das Systemmodell aus Kapitel 3 durchgeführt. Die hier angegebene Beispielentwicklung läßt sich entsprechend auf Automatendokumente mit konkreter syntaktischer Darstellung übertragen.

6.2 Modifikation der Initialmenge

Es gibt zwei Verfeinerungsregeln, die die Initialmenge modifizieren. Eine erlaubt es, Elemente aus der Initialmenge zu entfernen. Die andere erlaubt die Modifikation des Initialzustands, wenn die zugehörige initiale Ausgabe unendlich lang ist, da in diesem Fall der Initialzustand nicht mehr eingenommen wird.

6.2.1 Verkleinern der Initialmenge

Ein Automat ist unterspezifiziert, wenn er mehrere Initialelemente als Einstiegspunkte besitzt. Diese Menge kann verkleinert werden, um die Spezifikation entsprechend zu verfeinern.

Proposition 6.3 (Verkleinern der Initialmenge)
Durch Verkleinern der Initialmenge *mittels $I' \subseteq I$ wird ein Automat (S,M,δ,I) verfeinert. Es gilt:*

$$[\![(S,M,\delta,I)]\!] \supseteq [\![(S,M,\delta,I')]\!].$$ (□, **Beweis siehe** C.19)

Zu beachten ist, daß die Semantik eines Automaten nach Proposition 5.16 eine leere Menge stromverarbeitender Funktionen wird, wenn die Initialmenge zur leeren Menge verkleinert wird.

6.2.2 Modifikation finaler Initialelemente

Bei Initialelementen mit unendlicher Ausgabe spielt der Initialzustand im Verhalten keine Rolle mehr, weil er aus operationeller Sicht nicht mehr erreicht wird. Der Initialzustand kann in diesem Fall frei geändert werden.

Proposition 6.4 (Modifikation finaler Initialelemente)
Ist ein Automat (S,M,δ,I) und eine Initialmenge $I' \subseteq S \times M^\omega$ gegeben und gilt:

$$I \cap (S \times M^*) = I' \cap (S \times M^*)$$
$$\forall out \in M^\infty.\ (*,out) \in I \Leftrightarrow (*,out) \in I',$$

dann gilt

$$[\![(S,M,\delta,I)]\!] = [\![(S,M,\delta,I')]\!]$$ (□, **Beweis siehe** C.20)

6.3 Modifikation von Transitionen

Transitionen können unter bestimmten Bedingungen zu einem Automaten hinzugefügt oder entfernt werden. Eine weitere Verfeinerungsregel behandelt die Modifikation des Zielzustands einer finalen Transition, da dieser bei der Definition der Semantik keine Rolle spielt und daher frei modifiziert werden kann. Operationell gesehen wird der Zielzustand einer finalen Transition nicht erreicht.

6.3.1 Entfernung von Transitionen

Besitzt ein Automat in einem Zustand für eine Eingabe mehrere Transitionen, so modellieren wir damit Unterspezifikation bei der spezifizierten Komponente. Ein solcher Automat kann durch Entfernung einer Teilmenge dieser Transitionen verfeinert werden.

Dadurch wird unterspezifiziertes Verhalten weiter festgelegt. Dies drückt sich hier aus, indem eine Teilmenge der Transitionen gewählt wird und Alternativen aus dem Automaten entfernt werden. Dies wird durch folgende Proposition festgehalten. Es ist zulässig, in einem Verfeinerungsschritt beliebig viele Transitionen zu entfernen, soweit der modifizierte Transitionsgraph gegenüber dem ursprünglichen keine weiteren Partialitäten besitzt.

Proposition 6.5 (Entfernung von Transitionen)
Ist (S,M,δ,I) ein Automat und δ' eine Transitionsrelation, die in δ enthalten und nicht partieller ist als δ:

$$\delta' \subseteq \delta$$
$$\forall s\in S, m\in M.\ \delta(s,m,*,*) \Rightarrow \delta'(s,m,*,*),$$

so ist die Entfernung der Transitionen *$(\delta\backslash\delta')$ ein Verfeinerungsschritt. Es gilt:*

$$[\![(S,M,\delta,I)]\!] \supseteq [\![(S,M,\delta',I)]\!].$$

(□, **Beweis siehe** C.21)

6.3.2 Hinzufügen von Transitionen

Ist ein Automat in einem Zustand unterspezifiziert, weil keine Transition zur Verarbeitung eines Eingabezeichens angegeben ist, so lassen sich durch Hinzufügen von Transitionen auftretende Partialitäten vermindern. Auch hier wird das Verhalten der beschriebenen Komponente verfeinert.

In der Softwareentwicklung kann diese Verfeinerungstechnik zum Beispiel benutzt werden, um die Partialitäten eines Automaten als Fehlerfälle robust zu implementieren.

Proposition 6.6 (Hinzufügen von Transitionen)
Durch Hinzufügen einer Menge von Transitionen

$$\delta' \in S \times M \times S \times M^{\omega}$$

entsteht aus einem Automaten (S,M,δ,I) ein verfeinerter Automat (S,M,δ∪δ',I), wenn jede Transition aus δ' nur Partialität von δ entfernt:

$$\forall s \in S, m \in M.\ \delta'(s,m,*,*) \Rightarrow \neg\delta(s,m,*,*).$$

Es gilt dann:

$$[\![(S,M,\delta,I)]\!] \supseteq [\![(S,M,\delta\cup\delta',I)]\!].$$

(□, **Beweis siehe** C.22)

6.3.3 Behandlung der Zielzustände finaler Transitionen

Eine finale Transition besitzt nach Definition eine unendliche Ausgabe. Operationell bedeutet dies, daß die Transition nicht vollständig ausgeführt wird und der Zielzustand nicht erreicht wird. Dies spiegelt sich dadurch wider, daß der Zielzustand einer finalen Transition bei der Definition der Semantik keine Rolle spielt. Wir nutzen dies im folgenden Verfeinerungsschritt:

Proposition 6.7 (Zielzustände finaler Transitionen)
Sei (S,M,δ,I) ein Automat, der für s∈S,m∈M und out∈M^{ω} mindestens eine finale Transition besitzt:

$$\delta(s,m,*,out).$$

Sei ferner δ_1 eine beliebige nichtleere Teilmenge von {(s,m,∗,out)}, dann ist mit

$$\delta' = \delta \backslash \{(s,m,*,out)\} \cup \delta_1$$

die Änderung des Zielzustands finaler Transitionen eine semantikerhaltende Verfeinerung:

$$[\![(S,M,\delta,I)]\!] = [\![(S,M,\delta',I)]\!].$$

(□, **Beweis siehe** C.23)

Die mit Proposition 6.7 erlaubte Hinzunahme weiterer finaler Transitionen erhöht die nichtdeterministische Auswahl. Diese ergibt aber keine neuen Verhaltensweisen der modellierten Funktion, da Eingabe und Ausgabe mit einer bereits existierenden Transition übereinstimmen und der Zielzustand bei finalen Transitionen keine Rolle spielt.

Basierend auf dieser Proposition, definieren wir die Transformation *compactify*, die finale Transitionen so modifiziert, daß Quell- und Zielzustand identisch sind. δ^f und δ^c bezeichnen dabei den Anteil finaler und nichtfinaler Transitionen, wie in Definition 4.4 vereinbart.

Dadurch wird die Menge der ω-erreichbaren Zustände eines Automaten so verkleinert, daß sie mit der Menge der erreichbaren Zustände übereinstimmt. Damit wird der Unterschied zwischen beiden Erreichbarkeitsdefinitionen aufgehoben und wir können uns z.B. bei der Entfernung von Zuständen auf nicht ω-erreichbare Zustände einschränken.

Definition 6.8 (Kompaktifizierung von Automaten)
Die Transformation compactify sei wie folgt auf Automaten definiert:

$$compactify(S,M,\delta,I) \stackrel{def}{=} (S,M,\delta',I)$$
`where`
$$\delta' = \delta^c \cup \{(s,m,s,out) \mid \delta^f(s,m,*,out)\}$$

die finale Transitionen in Schlingen wandelt. (□)

Proposition 6.9 (Eigenschaften von *compactify*)

1. *Die Transformation compactify ist semantikerhaltend.*
2. *Für jeden Automaten (S,M,δ,I) gilt:*
$$reach(compactify(S,M,\delta,I)) = reach(S,M,\delta,I).$$
3. *Für jeden Automaten (S,M,δ,I), der nur endliche Initialausgaben hat, gilt:*
$$\omega{-}reach(compactify(S,M,\delta,I)) = reach(S,M,\delta,I).$$

(□, **Beweis siehe** C.24)

6.4 Modifikation von Zuständen

Ähnlich wie Transitionen können auch Zustände hinzugenommen und, wenn sie nicht erreichbar sind, auch entfernt werden. Daneben erlauben wir die Verfeinerung von Zuständen, die in Kombination mit den Verfeinerungsregeln für Automaten eine sukzessive Verhaltensverfeinerung erlaubt.

6.4.1 Entfernung unerreichbarer Zustände

Nicht erreichbare Zustände eines Automaten tragen nur bei partiellen Automaten zur Semantikbildung bei, da die Totalisierung bei der Semantikbildung beliebige Transitionen, also auch in nicht erreichbare Zustände hinzufügt. Die Totalisierung erfolgt jedoch immer so, daß beliebiges Chaos ab Auftreten einer Partialität möglich ist. Werden unerreichbare Zustände entfernt, so ändert sich dadurch die Semantik nicht. Zusammen mit der *compactify*-Transformation wird es möglich auch solche Zustände zu entfernen, die zwar ω-erreichbar, aber nicht erreichbar sind.

Wir erweitern zunächst Proposition 5.13 auf allgemeine Automaten:

Proposition 6.10 (Einschränkung auf ω-erreichbare Zustandsmenge)
Ist (S,M,δ,I) ein Automat und gilt

$$R = \omega{-}reach(S,M,\delta,I)$$
$$\delta' = \delta \cap R \times M \times R \times M^\omega,$$

dann gilt:

$$[\![(S,M,\delta,I)]\!] = [\![(R,M,\delta',I)]\!].$$

(□, **Beweis siehe** C.25)

Die vorangegangene Proposition benutzen wir zur Definition folgenden Verfeinerungsschritts:

Proposition 6.11 (Entfernen nicht ω-erreichbarer Zustände)
Ist (S,M,δ,I) ein Automat und

$$\omega\text{-}reach(S,M,\delta,I) \subseteq S' \subseteq S \ ,$$

dann ist die Entfernung der nicht ω-erreichbaren Zustände *$(S \setminus S')$ ein Verfeinerungsschritt. Die neue Transitionsrelation δ' entsteht aus δ durch Einschränkung:*

$$\delta' = \delta \cap S' \times M \times S' \times M^\omega .$$

Der neue Automat (S',M,δ',I) ist semantikerhaltend:

$$[\![(S,M,\delta,I)]\!] = [\![(S',M,\delta',I)]\!].$$

(□, **Beweis siehe** C.26)

6.4.2 Erweiterung der Zustandsmenge

Weil die Entfernung von Zuständen semantikerhaltend ist, lassen sich auch neue Zustände einführen. Ein Anwendungsgebiet für diese Verfeinerungsform ist die Vererbung zwischen Klassen, bei der oft neue Attribute hinzugefügt werden. Dadurch erweitert sich im allgemeinen der Lebenszyklus eines Objekts.

Proposition 6.12 (Erweiterung der Zustandsmenge)
Die Erweiterung der Zustandsmenge *eines Automaten (S,M,δ,I) auf die neue Zustandsmenge S', wobei $S \subseteq S'$ gilt, ist ein semantikerhaltender Verfeinerungsschritt:*

$$[\![(S,M,\delta,I)]\!] = [\![(S',M,\delta,I)]\!].$$

(□, **Beweis siehe** C.27)

Durch die Erweiterung des Automaten auf neue Zustände wird der neue Automat partiell. Wie Partialität behoben werden kann, haben wir bereits in Proposition 6.6 gesehen. Diese Proposition erlaubt es neue Transitionen einzuführen.

Die neu eingeführten Zustände sind zunächst nicht erreichbar. Durch Hinzufügen von Transitionen, ausgehend von bisher erreichbaren Zuständen werden die neu eingeführten Zustände ebenfalls erreichbar und tragen zur Semantikbildung bei.

Die Totalisierungsoperation *complete* ist nach den Propositionen 6.12 und 6.6, die es erlauben einen Fehlerzustand einzuführen und sämtliche Partialität zu entfernen, ebenfalls ein Verfeinerungsschritt. Nach Proposition 5.15 ist *complete* sogar semantikerhaltend.

6.4.3 Verfeinerung der Zustandsmenge

Wir haben bereits einen Mechanismus kennengelernt, um das Hinzufügen neuer Attribute in eine vererbte Klasse zu beschreiben. Häufiger jedoch wird eine Verfeinerung der Zustandsmenge hilfreich sein. Derselbe Mechanismus kann auch dazu verwendet werden, eine nur sehr grob spezifizierte Einheit, wie etwa den *Speicher* aus Abbildung 4.4 (Seite

61), durch eine speziellere Einheit, hier etwa den *Puffer* aus Abbildung 4.5 (Seite 62) zu verfeinern. In diesem Beispiel muß nach einer Zustandsverfeinerung noch eine Entfernung von Transitionen mittels Proposition 6.5 vorgenommen werden.

Die Verfeinerung des Zustandsraums eines Transitionssystems wurde bereits in [JON87] und [AL88] durch die Definition von Verfeinerungsfunktionen α („refinement mappings") beschrieben. Auch hier wird eine neue, konkrete Zustandsmenge S' eingeführt, und eine Abbildung α angegeben, die jedem konkreten Zustand $s' \in S'$ einen abstrakten Zustand $\alpha(s')$ des ursprünglichen Automaten zuordnet. Es entstehen Äquivalenzklassen der Form $\{s' \in S | \alpha(s')=s\}$, die jeden abstrakten Zustand $s \in S$ verfeinern. Damit diese Äquivalenzklassenbildung eine Kongruenz bezüglich der Transitionen wird, fordern wir, daß eine Transition des abstrakten Automaten in das Kreuzprodukt über alle Verfeinerungszustände abgebildet wird. Das heißt:

$$\forall s',t' \in S', in, out \in M^\omega.\ \delta'(s',in,t',out) \Leftrightarrow \delta(\alpha(s'),in,\alpha(t'),out).$$

Die Surjektivität von α sichert, daß tatsächlich alle abstrakten Zustände aus S verfeinert werden. Wir definieren daher:

Proposition 6.13 (Verfeinerung der Zustandsmenge)
Ist (S,M,δ,I) ein Automat, S' eine weitere Menge von Zuständen und $\alpha{:}S' \to S$ eine totale, surjektive Abbildung. Seien δ' und I' wie folgt definiert:

$$\begin{aligned} \delta' &= \{(s',m,t',out) \mid \delta(\alpha(s'),m,\alpha(t'),out)\}, \\ I' &= \{(s',out) \mid (\alpha(s'),out) \in I\}. \end{aligned}$$

Dann ist (S',M,δ',I') eine semantikerhaltende Verfeinerung:

$$[\![(S,M,\delta,I)]\!] = [\![(S',M,\delta',I')]\!].$$

(□, **Beweis siehe** C.28)

Bei dieser Verfeinerungstechnik entsteht Nichtdeterminismus, der durch das Entfernen von Transitionen nach Proposition 6.5 wieder eingeschränkt werden kann. Nichtdeterminismus, der in der Verfeinerung der Initialzustände entsteht, kann durch Verkleinern dieser Menge nach Proposition 6.3 verringert werden.

6.5 Schnittstellenverfeinerung

Die syntaktische Schnittstelle einer Komponente wird durch ihre Menge von Ein- und Ausgabezeichen beschrieben. Neben der Verfeinerung von Verhalten ist es oft notwendig, diese Schnittstelle einer Komponente zu modifizieren.

Im Zusammenhang mit der Formalisierung objektorientierter Softwareentwicklung stellen wir fest, daß die Eingabemenge einer Komponente nur erweitert wird. Während des Entwurfs werden häufig für bereits gegebene Klassen zusätzliche Methoden eingeführt. Bei der Vererbung werden Methodensignaturen in Subklassen ebenfalls erweitert. So unterschiedlich beide Einsatzgebiete sind, so lassen sie sich doch durch dasselbe Konzept formalisieren. Wir definieren diese Verfeinerung mit Hilfe von Signaturerweiterung.

Eine Einschränkung der Eingabemenge oder deren Verfeinerung tritt in der Objektorientierung nicht auf. Methodisch ist es interessant, zunächst mit einer abstrakten Eingabemenge, z.B. nur Methodennamen, zu spezifizieren und später von diesen Äquivalenzklassen auf die mit Argumenten versehenen Methodenaufrufe zu spezialisieren. Wir werden in der Formalisierung der Automatendokumente in Kapitel 8 dazu ein Patternkonzept nutzen und benötigen deshalb keine Verfeinerung der Eingabemenge.

Die Ausgabemenge einer Komponente unseres Systemmodells wird über die Signaturen aller anderen Komponenten definiert. Sie braucht daher ebenfalls nicht explizit modifiziert zu werden. Wir konzentrieren uns damit auf die Erweiterung der Eingabemenge als Verfeinerungsschritt.

Für diesen Abschnitt unterscheiden wir zwischen der Eingabemenge M_{in} und der Ausgabemenge M_{out} und verwenden für Automaten der Form $(S,M_{in},M_{out},\delta,I)$ eine Teilmenge der stromverarbeitenden Funktionen $M_{in}{}^{\omega} \xrightarrow{\mathcal{S}} M_{out}{}^{\omega}$ als Semantik.

Wird die Schnittstelle eines Automaten modifiziert, so sind die Semantiken für den ursprünglichen und modifizierten Automaten als Mengen aufgrund der unterschiedlichen Signaturen nicht direkt vergleichbar. Stattdessen können Techniken der Schnittstellenverfeinerung angewandt werden, wie sie in [BRO94] definiert wurden. Wir nutzen hier die in Abbildung 6.2 dargestellte Abwärtssimulation. Die Identität wird zur Repräsentation R_o der Ausgabe und eine Einbettung zur Repräsentation R_i der Eingabe verwendet.

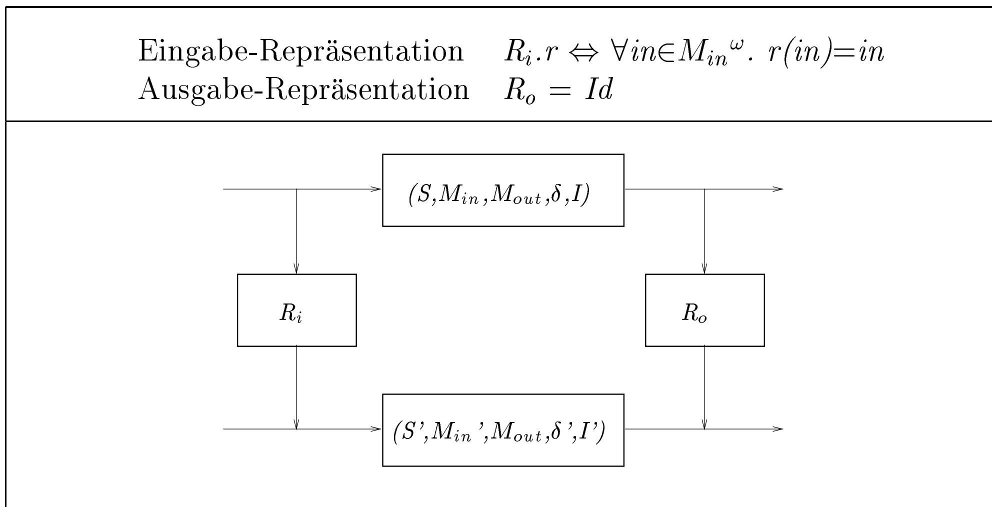


Abbildung 6.2: Erweiterung der Eingabemenge als Abwärtssimulation

Dies kann formuliert werden als:

$$R_i;[\![(S',M_{in}',M_{out},\delta',I')]\!] \subseteq [\![(S,M_{in},M_{out},\delta,I)]\!];R_o$$

Aufgrund der einfachen Struktur beider Repräsentationen kann diese Beziehung einfacher ausgedrückt werden. Wir verallgemeinern den Verfeinerungsbegriff aus Definition 6.1, um auch die Erweiterung von Schnittstellen zuzulassen:

Definition 6.14 (Schnittstellenverfeinerung)
Gilt $M_{in} \subseteq M_{in}'$*, so heißt ein Automat* $(S', M_{in}', M_{out}, \delta', I')$ Schnittstellenverfeinerung *des Automaten* $(S, M_{in}, M_{out}, \delta, I)$*, wenn*

$$[\![(S', M_{in}', M_{out}, \delta', I')]\!]|_{{M_{in}}^{\omega}} \subseteq [\![(S, M_{in}, M_{out}, \delta, I)]\!].$$

(□)

Proposition 6.15 (Schnittstellenverfeinerung)
Die Schnittstellenverfeinerungsrelation für Automaten ist transitiv. Die Schnittstellenverfeinerung beinhaltet die Verhaltensverfeinerung.

(□, **Beweis siehe** C.29)

Wir definieren nun folgenden Verfeinerungsschritt für Automaten:

Proposition 6.16 (Erweiterung der Eingabemenge)
Ist $(S, M_{in}, M_{out}, \delta, I)$ *ein Automat und* $M_{in} \subseteq M_{in}'$*, so ist die* Erweiterung der Eingabemenge *in den Automaten* $(S, M_{in}', M_{out}, \delta, I)$ *wegen*

$$\delta \subseteq S \times {M_{in}}^{\omega} \times S \times {M_{out}}^{\omega} \subseteq S \times (M_{in}')^{\omega} \times S \times {M_{out}}^{\omega}$$

eine Schnittstellenverfeinerung. Es gilt:

$$[\![(S, M_{in}', M_{out}, \delta, I)]\!]|_{{M_{in}}^{\omega}} \subseteq [\![(S, M_{in}, M_{out}, \delta, I)]\!]$$

(□, **Beweis siehe** C.30)

Das Verhalten einer Komponente ist für neue Zeichen aus $(M_{in}')\backslash M_{in}$ völlig unterspezifiziert. Eine Festlegung des Verhaltens kann durch Einführung von Transitionen nach Proposition 6.6 vorgenommen werden.

6.6 Der Verfeinerungskalkül

Im vorangegangen Abschnitt dieses Kapitels haben wir elf Transformationsregeln definiert und als korrekt bewiesen. Davon werden zehn Transformationen von Automaten zur Verfeinerung des beschriebenen Verhaltens und eine Transformation zur Erweiterung der Eingabe-Schnittstelle eingesetzt. Basierend auf diesen Verfeinerungstransformationen läßt sich gemeinsam mit der Kompositionalität der Verfeinerung (siehe Propositionen 6.2 und 6.15) ein Kalkül auf der abstrakten Syntax der Automaten angeben.

Wir strukturieren diesen Kalkül in einen Satz von Regeln zur Verhaltensverfeinerung und eine Erweiterung zur Schnittstellenverfeinerung.

Dieser Kalkül besteht in einfacher Form aus den in Tabelle 6.3 angegebenen acht Regeln, die Verfeinerungsschritten entsprechen. Zu beachten ist, daß die Regeln (RemS), (AddS), (RefS), (ChgFI) und (ChgFT), die die Zustandsmenge eines Automaten verändern oder finale Transitionen behandeln, in beide Richtungen angewendet werden können, während

die Regeln (RemI), (RemT) und (AddT), die die Transitionsmenge beziehungsweise die Initialmenge modifizieren, im allgemeinen echte Verfeinerungen sind.

Regel	Transformation	Bedingung
(RemI)	$\frac{(S,M,\delta,I)}{(S,M,\delta,I')}$	$I' \subseteq I$
(RemT)	$\frac{(S,M,\delta,I)}{(S,M,\delta',I)}$	$\delta' \subseteq \delta$ $\forall s{\in}S, m{\in}M.\ \delta(s,m,*,*) \Rightarrow \delta'(s,m,*,*)$
(AddT)	$\frac{(S,M,\delta,I)}{(S,M,\delta',I)}$	$\delta \subseteq \delta'$ $\forall s{\in}S, m{\in}M.\ (\delta' \setminus \delta)(s,m,*,*) \Rightarrow \neg\delta(s,m,*,*)$
(RemS)	$\frac{(S,M,\delta,I)}{(S',M,\delta',I)}$	$\omega{-}reach(S,M,\delta,I) \subseteq S' \subseteq S$ $\delta' = \delta \cap S' \times M \times S' \times M^\omega$
(AddS)	$\frac{(S,M,\delta,I)}{(S',M,\delta,I)}$	$S \subseteq S'$
(RefS)	$\frac{(S,M,\delta,I)}{(S',M,\delta',I')}$	$\alpha\colon S' \rightarrow S$ total, surjektiv $\delta' = \{(s',m,t',out) \mid \delta(\alpha(s'),m,\alpha(t'),out)\}$ $I' = \{(s',out) \mid (\alpha(s'),out){\in}I\}$
(ChgFI)	$\frac{(S,M,\delta,I)}{(S,M,\delta,I')}$	$I \cap (S{\times}M^*) = I' \cap (S{\times}M^*)$ $\forall out{\in}M^\infty.\ (*,out){\in}I \Leftrightarrow (*,out){\in}I'$
(ChgFT)	$\frac{(S,M,\delta,I)}{(S,M,\delta',I)}$	$\#out{=}\infty$ $\delta(s,m,*,out) \wedge \delta'(s,m,*,out)$ $\delta' \setminus \{(s,m,*,out)\} = \delta \setminus \{(s,m,*,out)\}$

Abbildung 6.3: Regeldarstellung der Verfeinerungsschritte

Die beiden in Tabelle 6.4 angegebenen Regeln (Cplt) und (Cpfy) sind keine elementaren Regeln, sondern komplexere Transformationen, die aus den Regeln der Tabelle 6.3 hergeleitet wurden.

In Tabelle 6.5 ist eine Regel definiert, die die Erweiterung der Eingabemenge erlaubt. Für die Formulierung dieser Regel wurde zwischen der Eingabemenge M_{in} und der Ausgabemenge M_{out} unterschieden. Alle anderen Regeln können sofort auf diese Erweiterung übertragen werden.

(Cplt)	$\dfrac{(S,M,\delta,I)}{(S',M,\delta',I)}$	$(S',M,\delta',I) = complete(S,M,\delta,I)$
(Cpfy)	$\dfrac{(S,M,\delta,I)}{(S,M,\delta',I)}$	$(S,M,\delta',I) = compactify(S,M,\delta,I)$

Abbildung 6.4: Regeldarstellung für *complete* und *compactify*

(ExIn)	$\dfrac{(S,M_{in},M_{out},\delta,I)}{(S,M_{in}',M_{out},\delta,I)}$	$M_{in} \subseteq M_{in}'$

Abbildung 6.5: Regel zur Schnittstellenverfeinerung

Definition 6.17 (Verfeinerungskalkül)
Die elf Regeln der Tabellen 6.3, 6.4 und 6.5 definieren den Verfeinerungskalkül. Wir schreiben für die Transformation des Automaten A in den Automaten A' durch korrekte Regelanwendung:

$A \rightsquigarrow A'$

Die transitive Hülle dieser Relation wird mit $\overset{*}{\rightsquigarrow}$, *die symmetrische Hülle mit* $\leftrightsquigarrow$ *und die symmetrisch transitive Hülle mit* $\overset{*}{\leftrightsquigarrow}$ *bezeichnet.* (□)

Die einzelnen Regeln seien im folgenden noch einmal kurz erläutert:

(RemI): Erlaubt die Entfernung von Initialzuständen (6.3).

(RemT): Erlaubt die Entfernung von Transitionen, wenn dazu alternative Transitionen existieren (6.5).

(AddT): Erlaubt das Hinzufügen von Transitionen, wenn bisher noch keine entsprechenden existiert haben (6.6).

(RemS): Erlaubt die Entfernung unerreichbarer Zustände (6.11).

(AddS): Erlaubt das Hinzufügen neuer Zustände (6.12).

(RefS): Erlaubt die Verfeinerung der Zustandsmenge bzw. einzelner Zustände (6.13).

(ChgFI): Erlaubt die Modifikation der Initialelemente mit unendlicher Ausgabe (6.4).

(ChgFT): Erlaubt die Modifikation der Zielzustände finaler Transformationen (6.7).

(Cplt): Nimmt die Totalisierung *complete* als Verfeinerungstransformation auf (5.15).

(Cpfy): Modifiziert den Zielzustand finaler Transformationen (6.9).

(ExIn): Erlaubt die Erweiterung der Eingabemenge (6.16).

Die Operation *complete* wird zur Definition der Semantik $[\![.]\!]$ für allgemeine Automaten benutzt, und die Operation *compactify* behandelt das Problem, daß mit Regel (RemS) nur Zustände entfernt werden können, die nicht ω-erreichbar sind. Nach Proposition 4.9 besteht ein Unterschied zwischen der effektiv erreichbaren Menge von Zuständen (*reach*) und der durch Transitionen erreichbaren Menge ($\omega-reach$), im allgemeinen ist die zweite echt größer. Durch Umsetzung der Zielzustände finaler Transformationen mit *compactify* wird die Menge der ω-erreichbaren Zustände auf die Menge der erreichbaren Zustände reduziert und somit sind alle unerreichbaren Zustände entfernbar.

Die zentrale Aussage dieses Kapitels ist in folgendem Satz zusammengefaßt, die die Korrektheit aller Regeln des Verfeinerungskalküls sichert:

Satz 6.18 (Korrektheit des Verfeinerungskalküls)
Der Verfeinerungskalkül $\rightsquigarrow$ ist korrekt. Hat der Automat A die Eingabemenge M_{in}, so gilt für die Automaten A und A':

$$A \overset{*}{\rightsquigarrow} A' \;\Rightarrow\; [\![A']\!]|_{{M_{in}}^{\omega}} \subseteq [\![A]\!] \qquad (\Box)$$

Beweis 6.19
Die Korrektheit der einzelnen Verfeinerungstransformationen folgt direkt aus den Propositionen 6.3, 6.5, 6.6, 6.11, 6.12, 6.13, 6.7, 6.4, 5.15, 6.9 und 6.16.

Die Korrektheit der Komposition von Verfeinerungstransformationen folgt aus der Transitivität der Verfeinerungsrelation 6.15 und 6.2. $(\Box)$

Aus dem Satz 6.18 folgt, daß die symmetrisch transitive Hülle $\overset{*}{\leftrightsquigarrow}$ nur semantikerhaltende Transformationen benutzt und keine Schnittstellen-Erweiterung zuläßt. Für $M_{in}'=M_{in}$ vereinfacht sich Satz 6.18 zu folgender Aussage:

$$A \overset{*}{\rightsquigarrow} A' \;\Rightarrow\; [\![A']\!] \subseteq [\![A]\!]$$

Wie bereits erwähnt, spielt neben der Korrektheit dieses Kalküls die praktische Anwendbarkeit und die automatische Überprüfbarkeit der Kontextbedingungen durch entsprechende Werkzeuge eine wesentliche Rolle. Werden Zustände und Transitionen auf endliche Mengen beschränkt, wie dies in Beschreibungstechniken für Software-Engineering-Methoden oft der Fall ist, so ist bis auf die Regel (Cplt) jede Regel tatsächlich effektiv anwendbar. Komplexer wird die Situation, wenn eine endliche Darstellung unendlicher Zustands- oder Transitionsmengen benutzt wird. Werden Patterns zur Darstellung einer Transitionsrelation benutzt, so existiert mit der Unifikation ein effektiver Mechanismus, um Überlappung zu testen und Schnittmenge und Vereinigung von Zeichen- oder Zustandsmengen zu bilden. Wird (zusätzlich) eine Sprache zur prädikativen Beschreibung von Automaten benutzt, so sind auch hier entsprechende Mechanismen notwendig, die aber bei hinreichender Mächtigkeit der Sprache nicht mehr existieren. Während bei regulären Ausdrücken Algorithmen für diese Konstruktionen existieren, gibt es bereits bei kontextfreien Grammatiken keinen Algorithmus mehr, die Schnittmenge zu bilden. Genau wie bei einer Sprache, die auf Prädikatenlogik basiert, entstehen hier Kontextbedingungen, die nicht automatisch verifiziert werden können. Vergleiche dazu auch [PR94] und [PR94b].

Die Vollständigkeit unseres Kalküls ist weniger von praktischem Interesse. Wie sich leicht feststellen läßt, haben die in Abbildung 6.6 gegebenen Automaten dieselbe Semantik, können aber nicht ineinander überführt werden. (Ein formaler Beweis wäre allerdings aufwendig.)

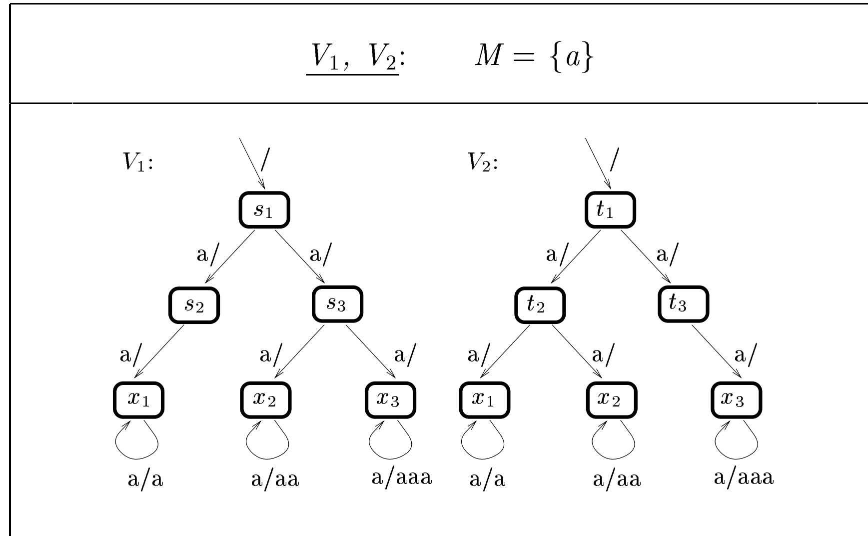


Abbildung 6.6: Zwei Automaten mit gleicher Semantik

Wichtiger ist sicher eine relative Vollständigkeit des Kalküls, die für wesentliche Entwicklungen ausreicht. Da mit der Verfeinerungsregel (RefS) sogar eine Umbenennung von Zuständen möglich ist, und für jede Komponente (Zustandsmenge, Transitionsrelation, Initialmenge) buchstabierender Automaten Transformationen existieren, ist eine für den praktischen Einsatz ausreichende Vollständigkeit des Kalküls gegeben. Dies demonstriert das Beispiel aus Abschnitt 6.1.

Verfeinerungskalkül in der gezeiteten Semantik

In Abschnitt 5.3.2 haben wir buchstabierenden Automaten eine gezeitete denotationelle Semantik gegeben. Wir übertragen nun den Kalkül für Verhaltens- und der Schnittstellenverfeinerung auf diese gezeitete Semantik und zeigen, daß der Kalkül korrekt bleibt.

Proposition 6.20 (Eigenschaften von $[\![.]\!]^t$)

1. *Ist die Initialmenge des buchstabierenden Automaten A nicht leer, so ist der Automat konsistent. Das heißt, die gezeitete Semantik ist nicht leer:*

$$[\![A]\!]^t \neq \emptyset.$$

2. *Stehen zwei Automaten in Verfeinerungsbeziehung, so gilt eine entsprechende Inklusionsbeziehung für die gezeitete Semantik:*

$$[\![A]\!] \subseteq [\![A']\!] \Rightarrow [\![A]\!]^t \subseteq [\![A']\!]^t$$

3. *Ist ein Automat A mit Eingabemenge M_{in} in einer Schnittstellenverfeinerungsrelation mit dem Automaten A', so gilt eine entsprechende Schnittstellenverfeinerungsrelation für die gezeitete Semantik:*

$$[\![A']\!]|_{M_{in}{}^{\omega}} \subseteq [\![A]\!] \Rightarrow [\![A']\!]^{t}|_{M_{in}{}^{\infty}} \subseteq [\![A]\!]^{t}.$$

(□, **Beweis siehe** C.31)

6.7 Verwandte Arbeiten

Im Kontext objektorientierter Systeme gibt es einige Arbeiten, die sich mit der Spezialisierung von Automaten befassen. In [RUM91, Cetal94, SM92] werden Automaten als Verhaltensbeschreibung einzelner Objekte eingesetzt, jedoch werden wegen der informellen Semantik der in diesen Softwareentwicklungsmethoden benutzten Beschreibungstechniken keine Anhaltspunkte über die Beziehung zwischen Automaten von in Vererbungsbeziehung stehenden Klassen gemacht. Diesen Nachteil versuchen einige Ansätze zum Teil mit sehr unterschiedlichen Techniken zu beheben. Ein verwandter Ansatz, der jedoch nicht auf dem Verfeinerungskonzept von Focus basiert, wurde in [PR94] vorgestellt. Er berücksichtigt keine Ausgabeaktionen und besitzt auch nur eine, wenn auch mächtige Verfeinerungstransformation. Er arbeitet mit Formeln einer Logik erster Ordnung um damit auch Vor- und Nachbedingungen von Transitionen auszudrücken.

Ein konzeptionell ähnlicher Ansatz wird in Troll ([Jetal91]) verfolgt, der in [Setal94] beschrieben ist. Dort wird der Lebenszyklus einer Klasse nicht als Zusicherung eines bestimmten, garantierten Verhaltens, sondern hauptsächlich als Restriktion des möglichen Verhaltens, verstanden. Deshalb unterscheiden sich auch die Bedingungen bei der Vererbung eines Lebenszyklus-Diagramms deutlich.

Weitere Arbeiten ([LW93, AC93, CAR93]) basieren weitgehend auf den grundlegenden Ideen von [WZ88] und [WEG90]. Das dort formulierte Prinzip der Substituierbarkeit von Elementen (Objekten) eines Supertyps durch Elemente eines Subtyps erhält die Typsicherheit. Diese Konzepte sind teilweise bereits in heute verwendeten (imperativen) objektorientierten Programmiersprachen integriert, wie zum Beispiel in Modula-3 ([Cetal92, ABA93]), behandeln jedoch nicht die Spezialisierung von Verhalten.

Ein interessanter Ansatz wird von Nierstrasz in [NIE93] verfolgt. Dort werden endliche Automaten als Typisierung verwendet, um zu beschreiben, in welchem Zustand ein Objekt welche Nachrichten akzeptiert, um damit dessen Klienten zuzusichern, welche Nachrichten in welcher Reihenfolge an dieses Objekt gesandt werden dürfen. Darüberhinaus wird eine Subtyprelation definiert, die einer Verhaltensspezialisierung unter Einhaltung der gegebenen Zusicherungen entspricht. Die Subtyprelation arbeitet sowohl im Kontext sequentieller, als auch verteilter, paralleler Kommunikation. Leider wird die Konstruktion der Subtyprelation, nicht zuletzt durch die Verwendung eines synchronen Konzepts zur Nachrichtenübermittlung im Stil von Milner's CCS ([MIL89]), relativ komplex.

Ein auf asynchroner Kommunikation basierender Verfeinerungskalkül für Transitionssysteme, die auf I/O-Automaten beruhen, ist in [SHB96] angegeben. Dieser Ansatz unterscheidet nicht zwischen Ein- und Ausgabezeichen und erlaubt nur ein Zeichen pro Transition. Seine Semantikbildung sowie die zugehörigen Verfeinerungsregeln beruhen auf einer Spursemantik.

In [PR94b] wurde untersucht, wie sich Automaten als Beschreibungstechnik in die algebraische Spezifikationssprache SPECTRUM ([Betal93b, Betal93c]) integrieren lassen. Ein Automat wird dort als spezielle Notation für ein algebraisches Axiom gesehen. Einerseits wird so eine anschauliche Beschreibungstechnik in die Umgebung axiomatischer Spezifikationen integriert, andererseits besitzt man mittels axiomatischer Spezifikation die Möglichkeit, Eigenschaften zu formulieren, die durch einen Automaten nicht ohne weiteres ausdrückbar sind. So lassen sich beispielsweise die Vor- und Nachbedingungen, wie wir sie in unseren Darstellungstechniken benutzen (siehe Abbildung 4.5 auf Seite 62) gut algebraisch formulieren.

Ein anderer Weg wurde in [RUM94] beschritten, wo endliche Automaten benutzt wurden, deren Transitionen mit ausführbaren Programmstücken und ausführbaren Vor- und Nachbedingungen attributiert sind. Dadurch wird ein Automat Teil der Programmiersprache und Laufzeittests ermöglichen den exemplarischen Test der Korrektheit des Zustandskonzepts für die damit charakterisierten Objekte. Hintergrund dieser Überlegungen ist die Problematik, daß bei einer Umsetzung eines Automaten in den Quellcode heute gängiger Programmiersprachen wie *C++* ([STR91a]) von Hand sehr leicht Fehler auftreten und kaum Unterstützung durch Werkzeuge angeboten wird. Für diese erweiterte Programmiersprache mit dem Namen *C++STD*, die Konzepte des Designs und der Programmierung vereint und damit als eine sehr hochstehende Programmiersprache bezeichnet werden kann, existiert eine prototypische Implementierung eines Compilers (auf Basis eines *C++*-Compilers).

6.8 Zusammenfassung

Dieses Kapitel definiert einen Verfeinerungskalkül für Automaten, dessen Regeln (mit einer Ausnahme) nur syntaktische und damit überprüfbare Kontextbedingungen besitzen. Dieser Kalkül ist in Satz 6.18 zusammengefaßt. Damit ist es möglich, das State-Box-Verhalten einer Komponente zunächst sehr abstrakt zu formulieren und durch Entwicklungsschritte zu konkretisieren. Dabei kann mit einer abstrakten Zustandsmenge begonnen werden, die im Verlauf der Entwicklung wiederholt verfeinert wird. Darüberhinaus kann die Menge der Eingabenachrichten sukzessive erweitert werden.

Teil III

Beschreibungstechniken

Dieser Teil der Arbeit formalisiert die Dokumentarten

- Datentypdokument,
- Klassenbeschreibung,
- Objektmodell,
- Automatendokument

und die dazu gehörenden Entwicklungsschritte. Zunächst wird der die Struktur beschreibende Anteil definiert. Dann werden Automatendokumente als Verhaltensbeschreibungen formalisiert.
Dieser Teil ist in folgende Kapitel eingeteilt:

7 Objektmodell

8 Automatendokumente

Kapitel 7

Objektmodell

In Kapitel 2 haben wir das Grundgerüst für formale Systementwicklungen skizziert. Wir haben mehrere Dokumentarten als Beschreibungstechniken motiviert und jeweils ein Beispiel dafür angegeben. Im Kapitel 3 haben wir ein Systemmodell angegeben, das als semantische Grundlage für diese Dokumentarten dient. In diesem und dem folgenden Kapitel definieren wir nun diese Dokumentarten und deren Semantik und geben Entwicklungsschritte dafür an. Dieses Kapitel beschäftigt sich mit der Beschreibung der Struktur objektorientierter Systeme, das nächste Kapitel setzt die Theorie buchstabierender Automaten in Form von Automatendokumenten um.

Viele der aktuellen Softwareentwicklungsmethoden benutzen ein Datenmodell, das auf dem Entity/Relationship-Modell ([CHE76]) beruht. Dieses Modell dient in seiner ursprünglichen Fassung zur Beschreibung der sogenannten „statischen“ Struktur eines Systems, die die im System enthaltenen Daten und ihre Beziehungen zueinander festhält. Im Laufe der Jahre hat dieses Modell viele Erweiterungen und Adaptionen erfahren. Eine davon ist die Anwendung auf objektorientierte Systeme, in denen Daten auf Objekte verteilt und darin gekapselt werden. Solche Objektmodelle beschreiben daher die Struktur von Klassen und die Datenbeziehungen zwischen deren Objekten. Diese Datenbeziehungen bilden darüberhinaus die Grundlage für die Bestimmung von Kommunikationsbeziehungen zwischen Objekten.

In diesem Kapitel werden die Beschreibungstechniken für Objektmodelle in Form von abstrakter Syntax, konkreter Darstellung und einer auf dem Systemmodell basierenden Semantik definiert. Im einzelnen sind dies die für ein Objektmodell notwendigen Beschreibungstechniken:

- Datentypen,
- Klassenbeschreibungen und
- Objektmodelle.

Jeder Beschreibungstechnik wird ein Abschnitt dieses Kapitels gewidmet. Ein abschließender vierter Abschnitt faßt die Ergebnisse dieses Kapitels zusammen.

7.1 Datentypen und Prädikate

Datentypdokumente

Für eine formale Softwareentwicklungsmethode ist es unerläßlich, eine allgemeine Sprache zur Verfügung zu stellen, in der Datentypen definiert, Konstanten und Funktionen darauf angegeben und Aussagen in Form von Prädikaten formuliert werden können. Statt selbst eine entsprechende Syntax und eine formale Semantikdefinition anzugeben, erlauben wir uns, auf existierende Sprachen zu verweisen.

In unseren Beispielen verwenden wir die funktionale Sprache *Gofer* ([JON93a]) als Basissprache zur Definition von Datentypen. Sie besitzt ein mächtiges Typkonzept mit Polymorphie, Typklassen, Funktionen höherer Ordnung und vordefinierte Datentypen für Listen [.] und Tupel *(..)*. Weil die Auswertung in *Gofer* „lazy" erfolgt, lassen sich mit unendlichen Listen die aus FOCUS bekannten Ströme gut simulieren.

Ähnlich gute Kandidaten wären die funktionalen Sprachen HASKELL ([HJW92]) oder *ML* ([PAU91]), die ebenfalls ein umfangreiches System zur Definition von Datentypen und Funktionen zur Verfügung stellen. Auch algebraische Spezifikationstechniken wie SPECTRUM ([Betal93b]) bieten eine komfortable Definition abstrakter Datentypen und darauf operierender Funktionen.

Eine funktionale Sprache hat den Vorteil der sofortigen Ausführbarkeit und ist dennoch so abstrakt, daß sie als Spezifikationssprache verwendet werden kann. Für Datentypen niederer Stufen, die typischerweise als Attribute und Argumente von Methodenaufrufen auftreten, gibt es im allgemeinen eine klare Modellvorstellung, so daß eine im Sinne algebraischer Spezifikationstechniken monomorphe Spezifikation gewünscht ist. Gerade dafür ist eine funktionale Beschreibung besser geeignet als eine rein algebraische.

Außerdem lassen sich die Unifikationstechniken, wie sie in funktionalen Sprachen zur Verfügung stehen, verwenden, um Abstraktionsmechanismen für große Datenmengen zu definieren.

Ein Datentyp definiert neben der Sorte immer auch eine Menge von Ausdrücken, deren Werte in dieser Sorte liegen. Wir nutzen die Sprachen

- $\langle sort \rangle$ zur Definition von Sortenausdrücken,
- $\langle expr \rangle$ für Wertausdrücke und
- $\langle patt \rangle$ für Patterns, die eine Teilmenge von $\langle expr \rangle$ sind,

zur Definition von Datentypen. Die Sprache $\langle sort \rangle$ entspricht dabei den in [JON93a] beschriebenen monomorphen, typvariablenfreien Sortenausdrücken. In Abbildung 7.1 ist ein Datendokument angegeben. Es definiert drei Datentypen *Alter*, *Adresse* und *PersonenDaten*, eine Funktion *incAlter* und eine Konstante *theo*. Das Dokument selbst besitzt den Dokumentnamen *PersonenDaten*.

```
datadocument PersonenDaten :
   data Alter         = Alter Int
   data Adresse       = Inland (Stadt,Strasse,TelNr)
                      | Ausland (Land,Stadt,Strasse)
   data PersonenDaten = Pers(Name,Alter,Adresse)

   incAlter :: PersonenDaten -> PersonenDaten
   incAlter (Pers(n,Alter i,ad)) = Pers(n,Alter(i+1),ad)

   theo :: PersonenDaten
   theo = Pers("Theo",Alter 17,theosAdr)
enddocument
```

Abbildung 7.1: Datendokument PersonenDaten

Definition 7.1 (Datentypdokumente)
Die Semantik von Sorten, Ausdrücken und Patterns wird mittels Interpretation im Systemmodell definiert. Die (partielle) Interpretation von Wertausdrücken $\langle expr \rangle$ hängt von einer Belegung der Variablen des Ausdrucks ab:

$$[\![.,.]\!]^{expr} : \langle expr \rangle \times BEL \rightharpoonup VAL$$

Ein Datentypdokument $d \in \mathcal{DOC}_{dt}$ besteht aus einer Menge von Definitionen von Datentypen, Konstanten und Funktionen. Ihre Semantik besteht aus der Menge von Systemen, bei denen das Werteuniversum VAL, die Menge der Sorten $\langle sort \rangle$, die Belegungen BEL und die Wertemengen einer Sorte values eingeschränkt werden. (□)

Jedem Wertausdruck wird die Menge der freien Variablen zugeordnet:

$$fvar : \langle expr \rangle \to \mathbb{P}(\langle var \rangle)$$

Der Operator *fvar* wird kanonisch auf Mengen von Ausdrücken erweitert. Wir gehen davon aus, daß alle Datentypdokumente konsistent verwendet werden und gehen daher nicht auf Konsistenzbedingungen ein.

Für jede Klasse $c \in CN$ gibt es eine funktionale Sorte, deren Elemente Objektidentifikatoren von Agenten dieser Klasse sind. Wegen des engen Zusammenhangs zwischen einer Objektidentifikatorsorte und der zugehörigen Klasse erhalten beide denselben Namen. Wir legen daher $CN \subseteq \langle sort \rangle$ fest. Damit keine Objektidentifikatoren berechnet, sondern diese nur vom System gemeinsam mit den zugehörigen Agenten generiert werden können, sind keine Konstruktoren der Sorten *CN* bekannt. Dies bedeutet, daß es keine geschlossenen (variablenfreien) Ausdrücke geben kann, die zu einer Objektidentifikatorsorte gehören und damit ein Objekt identifizieren. Auf diese Weise lassen sich keine Identifikatoren generieren, die nicht vom System zur Verfügung gestellt sind.

Objektidentifikatorsorten und weitere Grundsorten wie *Int* oder *Bool* sowie ein Listenkonstruktor [.] sind standardmäßig vorhanden. Alle anderen benutzten Datentypen werden durch Dokumente festgelegt.

Ein Datentypdokument fordert die Existenz von Sortenausdrücken, verbietet aber nicht die Existenz weiterer Sortenausdrücke. So bleibt jede Menge von Dokumenten um neue Sorten erweiterbar und die Semantik eines Datentypdokuments ist damit „lose“ definiert. Ein Datentypdokument kann verfeinert werden, indem es um weitere Funktions- und Sortendefinitionen erweitert wird. Wir schlagen folgende Entwicklungsschritte für Datentypdokumente vor:

1. Ein neues Datentypdokument in den Dokumentgraphen einfügen,
2. ein Datentypdokument um neue Funktions- und Sortendefinitionen erweitern und
3. aus mehreren Datentypdokumenten des Dokumentgraphen alle Definitionen in einem Dokument sammeln.

Die drei Entwicklungsschritte sind (in dieser informellen Fassung) korrekt nach Definition 2.8. Während im ersten Fall kein anderes Dokument redundant wird, ist im zweiten Fall das vorhergehende Dokument redundant und kann als solches markiert werden. Im dritten Fall werden alle zur Sammlung der Definitionen benutzten Datentypdokumente redundant.

Prädikatenlogik

Ähnlich wie Datentypen spielt die nachfolgend beschriebene Prädikatenlogik in dieser Arbeit nur eine untergeordnete Rolle. Wir erlauben uns daher, auf algebraische Spezifikationssprachen ([WIR90]) wie SPECTRUM ([Betal93b]) zu verweisen.

Basierend auf Datentypdefinitionen wird eine Prädikatenlogik erster Stufe als gegeben angenommen, die es erlaubt, Aussagen über Eigenschaften von Datentypen und darauf operierenden Funktionen zu formulieren und Teilmengen von Werten eines Datentyps zu charakterisieren. Die hier benutzte Prädikatenlogik kann als Teilmenge von *HOLCF* ([REG94]) verstanden werden. *HOLCF* ist eine Erweiterung der Logik höherer Ordnung *HOL* ([GM93]) um die Logik berechenbarer Funktionen *LCF* ([GMW79]) und im generischen Theorembeweiser *Isabelle* ([PAU94]) instantiiert. Es ist nicht daran gedacht, innerhalb dieser Arbeit auf Verifikationsfragen einzugehen. Wir nehmen stattdessen an, daß alle zu beweisenden Aussagen sowie Datentyp- und Funktionsdefinitionen in *HOLCF* übersetzt und notwendige Beweise in *Isabelle* geführt werden.

Die Sprache $\langle pred \rangle$ basiert auf der Sprache $\langle expr \rangle$ der Wertausdrücke. Prädikatenlogische Aussagen werden aus den üblichen Junktoren, der Negation und den Quantoren aufgebaut. Fest eingebaut ist die zweiwertige Gleichheit $.=.$

Wir verwenden prädikatenlogische Aussagen im Kapitel 8 zur Präzisierung von Vor- und Nachbedingungen von Transitionen und zur Charakterisierung von Äquivalenzklassen von Datenzuständen. Wir nutzen die übliche Interpretation prädikatenlogischer Aussagen:

Definition 7.2 (Interpretation prädikatenlogischer Aussagen)
Die Interpretation von Prädikataussagen $\langle pred \rangle$ *hängt von einer Belegung der freien Variablen ab:*

$$.,.\models . : \mathcal{SM} \times BEL \times \langle pred \rangle \rightarrow \mathbb{B}$$

Geschlossene Aussagen (also ohne freie Variablen) werden interpretiert durch:

$$.\models . : \mathcal{SM} \times \langle pred \rangle \rightarrow \mathbb{B}$$

Beide Formen werden kanonisch auf Mengen von Aussagen erweitert. Wir belegen nicht wohlgeformte Formeln ebenfalls mit ff. Eine Folgerungsbeziehung zwischen prädikatenlogischen Aussagen wird definiert als

$$.\models . : \langle pred \rangle \times \langle pred \rangle \rightarrow \mathbb{B}$$
$$P \models Q \quad \stackrel{def}{=} \quad \forall sys {\in} \mathcal{SM}.\ sys \models P \Rightarrow sys \models Q$$

Sie wird auf Mengen von Aussagen ausgeweitet. (□)

Wir gehen davon aus, daß prädikatenlogische Aussagen korrekt verwendet werden. Wir gehen daher genau wie bei den Datentypdefinitionen nicht auf sprachspezifische Kontextbedingungen ein.

7.2 Klassenbeschreibungen

Eine Klassenbeschreibung dient dazu, die Struktur und die Schnittstelle einer Klasse zu definieren. Darüberhinaus können weitere Eigenschaften festgelegt werden, die während der Systementwicklung einzuhalten sind. So kann zum Beispiel die Anzahl der möglichen Instanzen einer Klasse eingeschränkt werden. Wir formalisieren nachfolgend Klassenbeschreibungen, wie sie in objektorientierten Methoden verwendet werden.

Zu einer Klasse können während einer Systementwicklung mehrere Klassenbeschreibungen entstehen, die zum Teil durch Verfeinerung auseinander hervorgegangen sind, zum Teil aber auch unabhängig voneinander entwickelt wurden. Wir werden daher nicht nur angeben, wie Klassenbeschreibungen verfeinert werden, sondern auch eine Integration mehrerer Klassenbeschreibungen in ein Dokument definieren.

Dokumentart Klassenbeschreibung

Für eine Klasse ist jeweils anzugeben, von welchen Klassen sie abgeleitet wird. Dadurch sind bereits gewisse Anforderungen an Struktur und Schnittstelle definiert.

Neu hinzu kommende Merkmale der Struktur und Schnittstelle werden in Form neuer Attribute und Methodensignaturen festgelegt. Vererbte Attribute und Methodensignaturen können, aber müssen nicht wiederholt werden.

In der Objektorientierung werden Klassen als Strukturierungsmittel verwendet, um gemeinsame Eigenschaften mehrerer Klassen in einer Superklasse zu faktorisieren. Eine solche Superklasse ist im allgemeinen *abstrakt*, das heißt, von ihr wird während des Ablaufs eines Systems keine Instanz gebildet.

Abstrakte Superklassen können auch verwendet werden, um verschiedenen Klienten einer Klasse unterschiedliche Schnittstellen zur Verfügung zu stellen, indem für jeden Klienten eine abstrakte Superklasse gebildet wird, die genau die für den Klienten bestimmte Schnittstelle besitzt. Dieser Mechanismus erlaubt filigrane Schnittstellendefinitionen, kann aber nur unter Benutzung von Mehrfachvererbung eingesetzt werden, da die Implementierung Subklasse aller Schnittstellendefinitionen ist.

Häufig wird eine Klasse auch als Modul genutzt. Charakterisierend für eine solche Klasse ist, daß während eines Systemablaufs genau eine Instanz dieser Klasse existiert. Dieser Modulcharakter von Klassen wird in [Getal94] durch das Design Pattern „Singleton" beschrieben.

Wir erlauben die Angabe einer *Kardinalität*, die die Anzahl der maximal in einem Ablauf erzeugbaren Instanzen einer Klasse beschreibt. Die Kardinalität ist eine natürliche Zahl bzw. ∞, wenn diese unbeschränkt ist.

In der konkreten Darstellung sind für die oben charakterisierten häufigsten Anwendungsfälle, die Schlüsselwörter *abstract* und *module* sinnvoll.

```
classdocument LandfahrzeugDok :
   abstract class Landfahrzeug ;  -- Klassenname
   subclassof Fahrzeug ;          -- Liste der Superklassen

   fahrer :: Person ;             -- Attribute
   zulassung :: Zulassung ;

   setFahrer (p::Person) ;        -- Methodensignaturen
   starte(speed::Int) ;
enddocument
```

Abbildung 7.2: Klassenbeschreibung `LandfahrzeugDok`

Die in Abbildung 7.2 angegebene Klassenbeschreibung definiert die Klasse *Landfahrzeug*. Sie erweitert die von Superklasse *Fahrzeug* geerbten Attribute und Methoden um *fahrer*, *zulassung*, *setFahrer* und *starte* mit jeweils festgelegten Signaturen. Die Klasse *Landfahrzeug* ist abstrakt. Sie besitzt keine eigenen Instanzen.

Definition 7.3 (Dokumentart Klassenbeschreibung)
Die abstrakte Syntax einer Klassenbeschreibung ist ein Fünftupel (c,sup,meth,attr,card), bestehend aus

- *einem Klassennamen* $c \in CN$,
- *einer Menge von Namen von Superklassen* $sup \subseteq CN$,
- *einer Menge von Methodensignaturen* $meth \subseteq \langle meth \rangle \times \mathbb{P}(\langle var \rangle \times \langle sort \rangle)$,
- *einer Menge von Attributen* $attr \subseteq \langle var \rangle \times \langle sort \rangle$ *und*
- *einer Kardinalität* $card \in \mathbb{N} \cup \{\infty\}$.

Die Menge dieser Dokumente wird mit $\mathcal{DOC}_{cl} \subseteq \mathcal{DOC}$ *bezeichnet.* (□)

Definition 7.4 (Semantik einer Klassenbeschreibung)
Die Semantik einer Klassenbeschreibung (c,sup,meth,attr,card) wird definiert als die Menge von Systemen, die folgende Eigenschaft erfüllt:

$$[\![(c,sup,meth,attr,card)]\!]^{cl} = \{\ sys \in \mathcal{SM}\ |$$
$$c \in CN_{sys} \wedge sup \subseteq CN_{sys} \wedge (\forall d \in sup.\ c \sqsubseteq_{sys} d) \wedge \tag{7.1}$$
$$(\forall (m,varset) \in meth.\ m \subseteq \Sigma_{meth,sys}(c) \wedge args_{sys}(m) = \pi_1(varset) \wedge \tag{7.2}$$
$$\forall (v,tp) \in varset.\ type_{sys}(v) = tp) \wedge \tag{7.3}$$
$$\forall (a,tp) \in attr.\ a \in \Sigma_{attr,sys}(c) \wedge type_{sys}(a) = tp \wedge \tag{7.4}$$
$$\forall run \in EventTrace_{sys}.\ |\{a \in ID\ | class(a) = c \wedge (a,*,*) \in run\}| \leq card\ \} \tag{7.5}$$
(□)

In (7.1) der Semantikdefinition wird die Einordnung der neu definierten Klasse in die Subklassenhierarchie vorgenommen. Es wird gefordert, daß die angegebenen Vererbungsbeziehungen eingehalten werden, weitere Vererbungsbeziehungen sind jedoch nicht ausgeschlossen. Mit (7.2-7.3) werden die in der Klasse c mindestens existierenden Methoden eingeordnet. Weitere Methoden können dieser Klasse hinzugefügt werden, die Parameter und damit die Signatur einer bereits definierten Methode sind nicht mehr veränderbar. In (7.4) wird die Attributsignatur festgelegt, wobei auch hier weitere Attribute hinzugenommen werden können. Diese Semantik ist also lose in der Menge der geforderten Attribute und Methoden, aber monomorph in deren Sortenspezifikation. In (7.5) wird die Kardinalität einer Klasse eingeschränkt. Dies wird erreicht, indem in der Menge aller Abläufe eines Systems gefordert wird, daß höchstens an so viele Agenten eine Nachricht gesendet wird, wie die Kardinalität erlaubt. Das heißt, die angegebene Kardinalität bezieht sich auf die maximale Anzahl von aktiven Instanzen einer Klasse während eines Ablaufs.

Verfeinerung einer Klassenbeschreibung

Gemäß den obigen Ausführungen können wir nun die Folgerungsbeziehung für Klassenbeschreibungen formulieren, die einer Verfeinerung von Klassenbeschreibungen entspricht. Weil eine lose Semantik für Klassenbeschreibungen definiert wurde, lassen sich diese durch Erweiterungen verfeinern. Eine Klassenbeschreibung folgt aus einer anderen, wenn diese

1. die Menge der Superklassen erweitert,
2. die Menge der Methoden erweitert,
3. die Menge der Attribute erweitert und
4. die Kardinalität einschränkt.

Diese Verfeinerungsschritte sind in der objektorientierten Softwareentwicklung sehr häufig zu finden. Die Erweiterung einer Klasse um Funktionalität in Form von Methoden und Zustandskomponenten in Form von Attributen während der Entwicklung und Implementierung dient dazu, neu hinzugekommene externe Anforderungen zu erfüllen und die Implementierung extern genutzter Funktionalität auf mehrere Methoden zu verteilen.

Weitere Superklassen werden im allgemeinen in eine Klassenbeschreibung eingetragen, um zusätzliche Funktionalität oder Attribute zu erhalten. So wird in der Objektorientierung oft eine abstrakte Superklasse *Saveable* oder *Object* ([WGM89]) verwendet, die Methoden zum persistenten Speichern eines Objekts zur Verfügung stellt.

Proposition 7.5 (Verfeinerungsbeziehung für Klassenbeschreibungen)
Seien zwei Klassenbeschreibungen (c,sup,meth,attr,card) und (c',sup',meth',attr',card') gegeben. Diese stehen in Folgerungsbeziehung

$$(c,sup,meth,attr,card) \models (c',sup',meth',attr',card'),$$

wenn folgende Bedingungen erfüllt sind:

$$c'{=}c \wedge sup' \subseteq sup \wedge meth' \subseteq meth \wedge attr' \subseteq attr \wedge card' \geq card$$

(□, **Beweis siehe** C.32)

Integration von Klassenbeschreibungen

Die in Proposition 7.5 definierte relationale Charakterisierung der Verfeinerung von Klassenbeschreibungen kann zur konstruktiven Erweiterung von Klassenbeschreibungen oder zu deren Fusion verwendet werden. Eine Integration ist zum Beispiel notwendig, wenn in verschiedenen Subsystemen eine gemeinsame Klasse auf unterschiedliche Weise weiterentwickelt wurde und diese Entwicklungslinien wieder zusammengefaßt werden sollen. Der Operator *fuseClasses* integriert mehrere Klassenbeschreibungen derselben Klasse c in eine Klassenbeschreibung:

$$\begin{aligned} &fuseClasses : \mathbb{P}(\mathcal{DOC}_{cl}) \rightharpoonup \mathcal{DOC}_{cl} \\ &fuseClasses \; \{ \; (c,sup_i,meth_i,attr_i,card_i)_{i \in I} \; \} = \\ &\qquad (c, \textstyle\bigcup_i sup_i, \bigcup_i meth_i, \bigcup_i attr_i, \texttt{min}_i card_i) \end{aligned}$$

Aus der Konstruktion folgt sofort, daß die Fusion von Klassenbeschreibungen alle Informationen der fusionierten Klassenbeschreibungen erhält, die Dokumentmenge cs und die daraus integrierte Klassenbeschreibung also äquivalent sind:

$$\forall cs \subseteq \mathcal{DOC}_{cl}.\; cs \in \bullet fuseClasses \Rightarrow \; cs \models\!\!\!= fuseClasses(cs)$$

Entwicklungsschritte für Klassenbeschreibungen

Die obigen Ausführungen legen bereits die möglichen und sinnvollen Entwicklungsschritte für Klassenbeschreibungen nahe. Für eine gegebene Klassenbeschreibung ist es möglich:

1. die Menge der Superklassen zu erweitern,
2. die Menge der Methoden zu erweitern,
3. die Menge von Attributen zu erweitern und
4. die Kardinalität zu spezialisieren.

Immer möglich ist die:

5. Einführung einer neuen Klasse.

Aus obigen Entwicklungsschritten lassen sich weitere Entwicklungsschritte in Form von Taktiken konstruieren. Zum Beispiel kann die Faktorisierung gemeinsamer Funktionalität verschiedener Klassen in eine abstrakte Superklasse, wie in [OJ93] beschrieben, durch Einführung dieser neuen abstrakten Klasse und durch nachfolgenden Eintrag als Superklasse vorgenommen werden. Wir definieren folgende Entwicklungsschritte, dem Schema in Definition 2.8 folgend, die die oben angegebenen Schritte subsumieren:

Definition 7.6 (Entwicklungsschritte für Klassenbeschreibungen)
Wir definieren folgende zwei Entwicklungsschritte für Klassenbeschreibungen:

AddClass fügt für $n \notin \mathcal{N}$ *das Dokument d in den Dokumentgraphen ein:*

$$AddClass : \mathcal{DG} \times \mathcal{DOC}_{cl} \times \mathcal{N} \rightharpoonup \mathcal{DG}$$
$$AddClass((N,E,D,R),d,n) \stackrel{def}{=} \mathcal{DS}((N,E,D,R),d,n,\emptyset,\emptyset)$$

ExtClass erweitert die Klassenbeschreibung einer bereits definierten Klasse:

$$ExtClass : \mathcal{DG} \times \mathcal{DOC}_{cl} \times \mathcal{N} \times \mathcal{N} \rightharpoonup \mathcal{DG}$$
$$ExtClass((N,E,D,R),d,n_{neu},n) \stackrel{def}{=}$$
$$\mathcal{DS}((N,E,D,R),fuseClasses\{D.n,d\},n_{neu},\{n\},\{n\})$$

Dabei gelten folgende Bedingungen:

$$D.n \in \mathcal{DOC}_{cl},\ n_{neu} \notin \mathcal{N},\ \pi_1 d = \pi_1(D.n) \qquad (\square)$$

Die Einführung einer neuen Klassenbeschreibung für eine Klasse mittels *AddClass* ist auch möglich, wenn diese bereits eine Klassenbeschreibung im Dokumentgraphen besitzt. Wird Software durch ein Team entwickelt, so können auf diese Weise verschiedene Sichten einer Klasse entwickelt und später integriert werden. Es ist leicht zu sehen, daß die in Definition 7.6 angegebenen Entwicklungsschritte für Klassen korrekt sind.

Es ist häufig der Fall, daß eine bereits gegebene Klasse, die z.B. in einer Bibliothek zur Verfügung steht, nicht verändert werden darf und damit nicht angepaßt werden kann. In diesem Fall ist eine neue Subklasse zu bilden, um hier die notwendigen Anpassungen

durchzuführen. Es empfiehlt sich, diese Vererbungstechnik auch anzuwenden, wenn eine gegebene Klasse bereits selbst Subklassen besitzt, da sonst auch andere Subklassen modifiziert werden. In dieser Hinsicht kann die Modifikation von bereits existierenden Klassen nicht als modular angesehen werden. Klassen, die in Vererbungsbeziehung stehen, sind als stark gekoppelt zu betrachten, weshalb sich in der Objektorientierung die Faustregel immer mehr durchsetzt, eine Vererbungshierarchie sollte nicht tiefer als vier Stufen sein. Dies ist auch deshalb wünschenswert, weil eine über die Klassenhierarchie verstreute Funktionalität schwerer zu verstehen ist als eine gebündelte Beschreibung. Andererseits erleichtert die Benutzung von Vererbung die Entwicklung von Klassen, da bei der Weiterentwicklung nur die neuen Methoden und neuen Attribute anzugeben sind. Die Vererbungshierarchie strukturiert damit die Entwicklung und dokumentiert diese Entwicklungshierarchie.

Redundanztest

Neben den Entwicklungsschritten, die neue Dokumente hinzufügen, gibt es den Redundanztest, der zu einer Klassenbeschreibung untersucht, ob sie von ihren Nachfolgerdokumenten impliziert und damit redundant wird. Ist ein Dokument im Dokumentgraphen redundant, so besitzt es Nachfolgerdokumente, die es verfeinern, und muß bei der weiteren Systementwicklung nicht mehr beachtet werden. Eine Klassenbeschreibung ist redundant, wenn ihre Nachfolger im Dokumentgraphen

1. alle Subklassenbeziehungen beinhalten,
2. alle Methodenbeschreibungen beinhalten,
3. alle Attributbeschreibungen beinhalten und
4. die Kardinalität mindestens ebensoweit einschränken.

Dies kann durch neue Klassenbeschreibungen zur selben Klasse, aber auch durch Klassenbeschreibungen von Superklassen geschehen. Letzteres ist z.B. der Fall, wenn eine Teilfunktionalität in eine neue Superklasse faktorisiert wird oder Funktionalität von der Subklasse in die Superklasse übernommen wird. Auch in den in Abschnitt 7.3 definierten Objektmodell-Dokumenten werden Subklassenbeziehungen festgehalten, die hier zu beachten sind. Für ein Klassendokument kann eine vollständige Klassenbeschreibung einer Klasse bestimmt werden, die die transitive Hülle alle Subklassenbeziehungen und alle expandierten Methoden- und Attributdefinitionen enthält. Wird aus der Menge der Nachfolgerdokumente *D(succ.n)* eine solche Klassenbeschreibung expandiert, so kann diese mit der Klassenbeschreibung $D.n$ durch einfachen Inklusionstest verglichen und so Redundanz von n festgestellt werden. Solche expandierten Klassenbeschreibungen können auch genutzt werden, um die Widerspruchsfreiheit von Klassenbeschreibungen zu testen. Zum Beispiel dürfen keine Zyklen in der Subklassenrelation auftreten.

7.3 Objektmodelle

Die beiden bisherigen Dokumentarten werden im allgemeinen als Teile eines Objektmodells angesehen. Wir haben diese ausgegliedert und unabhängig davon behandelt. So verbleiben im Objektmodell nur Klassen und ihre Beziehungen. Neben der bereits mehrfach erwähnten Vererbungsbeziehung, die im Objektmodell dargestellt werden kann, gibt es allgemeine *Beziehungen* zur Darstellung bestimmter Sachverhalte. Während die Vererbung eine Beziehung zwischen den beteiligten Klassen ist, treffen alle anderen Beziehungen Aussagen über die Agenten der beteiligten Klassen. Solche Beziehungen unterscheiden sich in:

- ihrer Stelligkeit,
- ihrer Orientierung (gerichtet oder ungerichtet),
- ihrer Kardinalität (*1–zu–1*, *1–zu–n* etc.) und
- ihrem Zweck (Daten-, Teile/Ganzes- oder Kommunikationsverbindungen).

Beziehungen im Objektmodell dienen zunächst der Aufgabe, Datenabhängigkeiten zwischen beteiligten Klassen zu modellieren. Die prozeduralen Techniken, die zur Implementierung bei E/R-Modellen benutzt werden, haben Zugriff auf eine globale Datenmenge und können so entlang von Beziehungen frei navigieren. In verteilten objektorientierten Systemen existiert keine global zugreifbare Datenmenge. Ebenso existiert keine zentrale Einheit zur Verarbeitung von Daten. Stattdessen sind die Verarbeitungseinheiten und damit auch das Verarbeitungswissen verteilt und die Daten in Agenten gekapselt. Jeder Agent einer Klasse „kennt“ nur seine Attribute und die Agenten seiner Umgebung. Er verarbeitet die ihm zugegangenen Informationen (Nachrichten), um die Ergebnisse anschließend an die Umgebung zurückzusenden. Wegen dieser Verteilung der Verarbeitung ist ein Konzept zur Kommunikation zwischen Agenten notwendig.

Gute Kandidaten für diese *Kommunikationsbeziehungen* sind die im Objektmodell dargestellten Datenabhängigkeiten. Nicht jede Datenbeziehung muß jedoch zur Kommunikation genutzt werden. Umgekehrt ist es auch nicht zwingend, daß Kommunikation nur entlang der Datenbeziehungen möglich ist. So können global bekannte Agenten (ein solcher könnte etwa das Betriebssystem oder die graphische Oberfläche repräsentieren) von jedem Agenten aus angesprochen werden. Andere Agenten könnten Nachrichten erhalten, weil ihr Identifikator durch Nachrichtenvermittlung weitergereicht wurde. Solche Identifikatoren sind dann nicht notwendigerweise im Datenzustand (den Attributen) eines Agenten, sondern in den Parametern einer aktiven Methode wiederzufinden. Weil dadurch die Kommunikationsbeziehungen zwischen den verschiedenen Klassen sehr unübersichtlich werden können, wird z.B. in [BUD91] das ursprünglich auf [LH89] zurückgehende „Law of Demeter“ angegeben, das Kommunikationsbeziehungen, die sich nicht im Objektmodell wiederfinden lassen, einschränkt.

Weitere Unterschiede treten in der Betrachtungsweise der Beziehungen auf. Manche Beziehungen sind als *dynamisch* zu betrachten, andere als *statisch*. Ein Auto besitzt immer

vier Reifen, diese wechseln aber während der Lebenszeit eines Autos. Eine Verwandtschaftsbeziehung z.B. zwischen Mutter und Sohn ist im Gegensatz dazu als statisch zu betrachten, da sie sich nicht ändert.

Manche Beziehungen spiegeln eine *Teile-Ganzes-Beziehung* wider. So gehört ein Reifen wie auch die Karosserie zu einem Auto. Während jedoch Reifen gewechselt werden können, ist die Karosserie essentieller Bestandteil des Autos und damit der Lebenslauf der Karosserie an den des Autos gekoppelt. Darüberhinaus werden *Teile-Ganzes-Beziehungen* oft dazu verwendet, die Kommunikationswege einzuschränken: Ein Teil kann nur Nachrichten von dem ihm übergeordneten Agenten und dessen anderen Teilen empfangen und oft auch nur an diesen senden.

Eine Klassifizierung aller unterschiedlichen, interessanten Varianten dieser Beziehungen ist nicht Teil dieser Arbeit, kann aber auf der hier geschaffenen Basis verwirklicht werden. Wir schränken uns in der Formalisierung des Objektmodells ein, Datenbeziehungen zu modellieren, ohne über Kommunikationsbeziehungen Aussagen zu treffen. Wir konzentrieren uns auf binäre Beziehungen zwischen Klassen. An einer solchen Beziehung sind zwei verschiedene Klassen oder zweimal dieselbe Klasse beteiligt. Alle mehrstelligen Beziehungen können durch zweistellige dargestellt werden. Einstellige Beziehungen können durch Attribute realisiert werden.

In vielen Objektmodellen, z.B. in [RUM91] oder [Cetal94], ist es wie im E/R-Modell üblich, einer Beziehung keine Orientierung zu geben. Wir werden dies ebenfalls nicht tun. Dadurch kann die Entscheidung, wie die Beziehungen in den beteiligten Agenten implementiert werden, auf spätere Phasen der Softwareentwicklung verschoben werden. Ist eine Klasse an einer Beziehung beteiligt, so wird diese an all ihre Subklassen vererbt.

Wir benutzen ähnlich zu Fusion und OMT *Rollen*, um zu beschreiben, in welcher Form ein Agent an der Beziehung teilnimmt. Dies ist zum Beispiel notwendig, um reflexive Beziehungen wie die *Vater-Sohn-Beziehung* zu charakterisieren. Jede Beziehung erhält einen Namen. Dieser suggeriert häufig eine Orientierung, die jedoch nicht festgelegt ist. Erst die Rollenverteilung gibt die Möglichkeit eine Beziehung zu orientieren.

Rollen werden in der Semantikdefinition auch zur Darstellung der Beziehungen in Form von Attributen genutzt. Die so definierten Attribute besitzen den Datentyp $[c]$, wobei $c \in CN$ die Klasse darstellt, mit der eine Beziehung eingegangen wird.

Dokumentart Objektmodell

Ein Objektmodell stellt die oben erwähnten Klassen, deren binäre, mit Rollennamen attributierte Datenbeziehungen und deren Vererbungsbeziehungen dar. Rollennamen sind optional. Wir markieren das Fehlen eines expliziten Rollennamens mit ϕ.

Rollen werden bei der Semantikdefinition durch Attribute realisiert. Deshalb ist die Menge der Rollennamen die um ϕ erweiterte Menge der Variablennamen $\langle var \rangle$, mit der wir Attribute darstellen.

Definition 7.7 (Dokumentart Objektmodell)
Die Menge der Rollennamen $\langle var\rangle^{\phi}$ *besteht aus dem speziellen Rollennamen* ϕ *und der Menge der Variablen:* $\langle var\rangle^{\phi} \stackrel{def}{=} \langle var\rangle \cup \{\phi\}$.
Die abstrakte Syntax eines Objektmodells *ist ein Tripel (C,S,R), bestehend aus*

- *einer Menge von Klassennamen* $C \subseteq CN$,
- *einer Menge von Vererbungsbeziehungen* $S \subseteq CN \times CN$ *und*
- *einer Menge von Datenbeziehungen*

$$R \subseteq CN \times \langle var\rangle^{\phi} \times Relname \times CN \times \langle var\rangle^{\phi},$$

wobei mindestens der erste Rollenname besetzt sein muß ($\neq\phi$).
Die Menge der Objektmodell-Dokumente wird mit $\mathcal{DOC}_{om} \subseteq \mathcal{DOC}$ *bezeichnet.* ($\Box$)

Eine Rolle besteht aus einem Rollennamen und den beiden daran beteiligten Klassen. Die Elemente einer Relation R werden mit $r=(c,rc,rel,d,rd)$ bezeichnet. Der Relationsname wird dabei mit $r_{rel}=rel$, die Rolle der Klasse c mit $r_{\overleftarrow{ro}}=(c,rc,d)$ und die Rolle der Klasse d mit $r_{\overrightarrow{ro}}=(d,rd,c)$ selektiert.

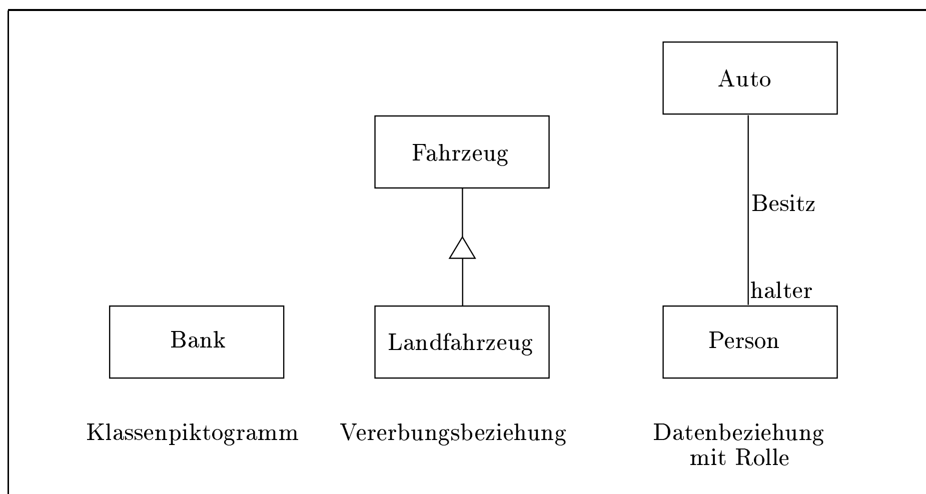


Abbildung 7.3: Darstellungselemente eines Objektmodells

In einer konkreten graphischen Darstellung werden die Elemente der Menge C, wie in Abbildung 7.3 gezeigt, durch Kästen gezeichnet. Die Vererbungsbeziehungen S werden durch mit Pfeilen versehene Linien dargestellt, wobei die Spitze eines Pfeils auf die Superklasse zeigt. Die Menge R definiert Datenbeziehungen zwischen zwei beteiligten Klassen mit dem jeweils gegebenen Relationsnamen. In der konkreten Darstellung wird der Rollenname ϕ nicht explizit angegeben. Kontextbedingungen, wie die Forderung, daß jeder Relationsname nur einmal vergeben werden kann oder jeder Rollenname eindeutig in der Klasse ist, werden überprüft.

Da sämtliche Informationen des beschriebenen Systems in Agenten abzulegen sind, sind auch diese Datenbeziehungen innerhalb von Agenten zu formalisieren. Wir benutzen

dazu Rollen, die jeder an einer Datenbeziehung beteiligten Klasse zugeordnet werden können. In der Implementierung wird ein Rollenname, als Attribut der in der Beziehung gegenüberliegenden Klasse interpretiert. Dieses gleichnamige Attribut besitzt einen Datentyp, der es erlaubt Identifikatoren zu speichern. So wird im Beispiel in Abbildung 7.3 die Rolle *halter*, die die Klasse *Person* einnimmt, als Attribut *halter*::[*Person*] der Klasse *Auto* interpretiert.

Semantik eines Objektmodells

Die abstrakte Syntax eines Objektmodells charakterisiert eine bestimmte Menge von Systemmodellen. Jede einzelne Komponente des Objektmodells stellt dabei eine bestimmte Restriktion an das Systemmodell. Wir definieren zunächst die Semantik einer Rolle $(c,r,d) \in CN \times \langle var \rangle^{\phi} \times CN$:

$$[\![(c,r,d)]\!]^{ro} \stackrel{def}{=} \{\ sys \in \mathcal{SM} \mid c,d \in CN_{sys} \wedge (r=\phi \vee (r \in \Sigma_{attr}.sys(d) \wedge type_{sys}(r)=[c]))\ \}$$

Die Semantik $[\![(c,r,d)]\!]^{ro}$ beschreibt die Tatsache, daß für $r \neq \phi$ in Klasse d das Attribut r dazu benutzt wird, die Datenbeziehung zur Klasse c darzustellen. Für $r=\phi$ wird nur gefordert, daß beide Klassen existieren.

Die Semantik einer Datenbeziehung aus der Menge R besteht darin, daß beide Rollen erfüllt werden müssen. Der Relationsname wird bei der Semantikbildung nicht verwendet:

$$[\![r]\!]^{rel} = \{\ sys \in \mathcal{SM} \mid sys \in [\![r_{\overleftarrow{ro}}]\!]^{ro} \wedge sys \in [\![r_{\overrightarrow{ro}}]\!]^{ro}\ \}.$$

Definition 7.8 (Semantik eines Objektmodells)
Die Semantik eines Objektmodells (C,S,R) wird definiert als die Menge von Systemen, die folgende Eigenschaft erfüllt:

$$[\![(C,S,R)]\!]^{om} = \{\ sys \in \mathcal{SM} \mid C \subseteq CN_{sys} \wedge (\forall (c,d) \in S.\ c \sqsubseteq_{sys} d) \wedge \forall r \in R.\ sys \in [\![r]\!]^{rel}\ \} \qquad (\Box)$$

Mit dem ersten Teil der Konjunktion wird gefordert, daß die Menge der angegebenen Klassen tatsächlich existiert, im zweiten und dritten Teil, daß die angegebenen Subklassenbeziehungen und Datenbeziehungen gelten. Zu beachten ist, daß eine Datenbeziehung zu einer abstrakten Klasse nicht bedeutet, daß beide Klassen abstrakt sind. Vielmehr können Beziehungen zu Instanzen aus Subklassen eingegangen werden.

Werden in einer Datenbeziehung beide Rollennamen vergeben, so wird diese Datenbeziehung in den Agenten der beiden beteiligten Klassen implementiert. Dies kann zu Inkompatibilitäten führen. Sei im Beispiel in Abbildung 7.3 die Rolle *auto* an die Klasse *Auto* vergeben, dann können mehrere Personen dasselbe Auto besitzen (Attribut *auto*), während dieses nur einem Halter zugeordnet ist (Attribut *halter*). Solche Inkompatibilitäten sind im Systemmodell zugelassen. Es liegt im Verantwortungsbereich des Entwicklers, diese zu verhindern, indem er z.B. nur *eine* Rolle pro Datenbeziehung vergibt. Umgekehrt ist es auch wichtig, diese Inkompatibilität zuzulassen, wenn beide Rollennamen (Attribute) definiert sind, denn durch die Verteilung der Agenten und der asynchronen Kommunikation kann eine simultane Modifikation beider Attribute nicht sichergestellt werden.

Verfeinerung eines Objektmodells

Die Semantikdefinition für Objektmodell-Dokumente ist lose gehalten. Ein Objektmodell fordert in diesem Sinn nur positive Eigenschaften, also die Existenz von Klassen, die Existenz von Vererbungsbeziehungen und die Existenz von Datenbeziehungen. Entsprechend sind Verfeinerungen eines Objektmodells durch Hinzufügen von Definitionen möglich. Diese entsprechen genau den Transformationsschritten für Objektmodelle, die mit Ausnahme von Revisionen bei der Softwareentwicklung am häufigsten verwendet werden. Dies sind:

1. Hinzunahme neuer Klassen,
2. Hinzunahme neuer Vererbungsbeziehungen,
3. Hinzunahme neuer Datenbeziehungen und
4. Vergabe von Rollen in einer Datenbeziehung.

Die Hinzunahme neuer Klassen (1.) wird in der Analyse verwendet, wenn neue Konzepte identifiziert sind und notiert werden. Im Entwurf werden sogenannte Entwurfsklassen hinzugenommen, die implementierungstechnische Details enthalten, aber im Gegensatz zu Analyseklassen keinen Ausschnitt der realen Welt modellieren. Während der Analyse wird häufig auf die Festlegung von Vererbungsbeziehungen verzichtet, die nachträglich eingeführt werden (2.), weil sie teilweise erst später identifiziert werden können. Die Hinzunahme neuer Datenbeziehungen (3.) entspricht analog zur Hinzunahme neuer Attribute in Klassenbeschreibungen dem Wissenszuwachs in der Analyse, bzw. Entwurfsentscheidungen in der Entwicklung. Ähnlich verhält es sich auch mit der Präzisierung von Datenbeziehungen durch die Vergabe von Rollennamen (4.).

Zunächst definieren wir hinreichende Bedingungen für die Folgerungsbeziehung von Datenbeziehungen und Rollen. Dazu führen wir eine binäre Ableitungsrelation $\vdash$ ein. Für Rollen wird die Ableitungsbeziehung $\vdash$ definiert als:

$$(c,r,d) \vdash (c',r',d') \overset{def}{\Leftrightarrow} c=c' \wedge d=d' \wedge (r'=\phi \vee r=r')$$

Für Datenbeziehungen wird die Ableitungsbeziehung $\vdash$ definiert als:

$$r \vdash r' \overset{def}{\Leftrightarrow} r_{rel}=r'_{rel} \wedge ((r_{\overleftarrow{ro}} \vdash r'_{\overleftarrow{ro}} \wedge r_{\overrightarrow{ro}} \vdash r'_{\overrightarrow{ro}}) \vee ((r_{\overleftarrow{ro}} \vdash r'_{\overrightarrow{ro}} \wedge r_{\overrightarrow{ro}} \vdash r'_{\overleftarrow{ro}}))$$

Die Ableitungsrelation für Datenbeziehungen charakterisiert, in welcher Form Datenbeziehungen modifiziert werden dürfen, damit eine Verfeinerungsbeziehung besteht. Wir geben nun ein hinreichendes Kriterium für die Verfeinerungsbeziehung von Objektmodellen an:

Proposition 7.9 (Verfeinerungsbeziehung für Objektmodelle)
Seien zwei Objektmodelle (C,S,R) und (C',S',R') gegeben. Wenn gilt:

$$C' \subseteq C \wedge S' \subseteq S \wedge \forall r{\in}R'.\exists r{\in}R.\ r \vdash r',$$

dann stehen diese in Folgerungsbeziehung:

$$(C,S,R) \models (C',S',R')$$

(□, **Beweis siehe** C.33)

Integration von Objektmodellen

Die relationale Charakterisierung der Verfeinerung von Objektmodellen kann in konstruktive Entwicklungsschritte umgesetzt werden. In der Praxis wird die Verfeinerung von Objektmodellen typischerweise dadurch ausgeführt, daß weitere Klassen, Subklassen- und Datenbeziehungen angegeben werden, die teilweise mit bereits vorhandenen Elementen eines Objektmodells überlappen. Dies führt zur Notwendigkeit, mehrere Objektmodelle in eines zu integrieren. Dazu entwickeln wir einen Operator *fuseOMs*, der mehrere Objektmodelle in eines fusioniert. Die komplexe Fusion von Relationen wird durch mehrere Hilfsfunktionen realisiert. Diese basieren direkt auf den vorangehend definierten Ableitungsrelationen.

Aus der Definition der Rollen folgt die Existenz des Operators $.\bowtie.$ zur Fusion zweier Rollen und ein Prädikat $.\overset{?}{\bowtie}.$, das die Fusionierbarkeit beschreibt:

$$(c,r,d)\bowtie(c',r',d') \overset{def}{=} \begin{cases} (c,r,d) & \texttt{if} \;\; r'=\phi \lor r'=r \\ (c',r',d') & \texttt{if} \;\; r=\phi \end{cases}$$

$$(c,r,d)\overset{?}{\bowtie}(c',r',d') \overset{def}{\Leftrightarrow} c=c' \land d=d' \land (r=r' \lor r=\phi \lor r'=\phi).$$

Das Prädikat $.\overset{?}{\bowtie}.$ charakterisiert die Existenz einer gemeinsamen Ableitung zweier Rollen:

$$ro\overset{?}{\bowtie}ro' \Leftrightarrow \exists ro''.\; ro\vdash ro'' \land ro'\vdash ro''$$

Für die integrierte Rolle gilt die Informationserhaltung:

$$\forall ro,ro'.\; ro\overset{?}{\bowtie}ro' \Rightarrow \{ro,ro'\} \models\!\!\dashv ro\bowtie ro'$$

Ein entsprechender Fusionsoperator $.\overset{?}{\bowtie}.$ kann auch für Relationen $r,r'\in R$ angegeben werden, der bei Übereinstimmung der Relationsnamen die beiden Rollen $r_{\overleftarrow{ro}}$ und $r'_{\overleftarrow{ro}}$ sowie $r_{\overrightarrow{ro}}$ und $r'_{\overrightarrow{ro}}$, bzw. deren symmetrische Vertauschung fusioniert. Ausgehend von einer binären Fusion kann durch Iteration die Fusion endlicher Mengen von Relationen vorgenommen werden. So entsteht der in Definition C.34 angegebene Operator:

$$fuseRELs : \mathbb{P}(R) \rightarrow \mathbb{P}(R),$$

der Relationen soweit wie möglich fusioniert. Für die Fusion von Mengen von Relationen gilt nach Proposition C.35:

$$\forall rs\subseteq R.\; rs \models\!\!\dashv fuseRELs(rs)$$

Damit können wir nun die Fusion von Objektmodellen definieren:

Definition 7.10 (Integration von Objektmodellen)
Wir definieren folgende Integration von Objektmodell-Dokumenten:

$$fuseOMs : \mathbb{P}(\mathcal{DOC}_{om}) \rightharpoonup \mathcal{DOC}_{om}$$
$$fuseOMs\; \{(C_i,S_i,R_i)_i\} = (\bigcup_i C_i,\; \bigcup_i S_i,\; fuseRELs(\bigcup_i R_i)) \qquad (\square)$$

Proposition 7.11 (Integration von Objektmodellen)
Eine Objektmodell-Integration erhält die Information:

$$\forall oms\subseteq\mathcal{DOC}_{om}.\;\; (fuseOMs\; oms) \models\!\!\dashv oms$$ (□, **Beweis siehe** C.37)

Entwicklungsschritte für Objektmodelle

Der Fusionsoperator *fuseOMs* von Objektmodellen bildet die Grundlage der folgenden Entwicklungsschritte, die im Softwareentwurf interessant sind. Neben der

1. Einführung eines neuen Objektmodells und der
2. Integration mehrerer Objektmodelle,

ist die Weiterentwicklung eines Objektmodells durch

3. Einführung neuer Klassen,
4. Einführung neuer Subklassenbeziehungen,
5. Einführung neuer Datenbeziehungen und durch
6. Vergabe von Rollennamen in gegebenen Datenbeziehungen

möglich. Das Einfügen eines neuen Objektmodells in den Dokumentgraphen (1.) zur Darstellung neuer, eventuell unabhängig entwickelter Systemteile ist immer möglich. Dies macht die Integration von Objektmodellen (2.) notwendig, da wir es erlauben, mehrere unabhängige Objektmodelle parallel zu entwickeln, um zunächst Systemausschnitte darzustellen. Häufig besitzen diese Objektmodelle Überschneidungen in Form gemeinsamer Klassen und Beziehungen.

Die Weiterentwicklung eines Objektmodells (3.-6.) unterliegt den Restriktionen aus Proposition 7.9 zur Erhaltung der Verfeinerungsbeziehung. Wir bieten die Möglichkeit alle Schritte (3.-6.) durch die Extension eines Datenmodells anzugeben, die wir nachfolgend als eine Operation definiert haben.

Definition 7.12 (Entwicklungsschritte für Objektmodelle)
Wir definieren folgende Entwicklungsschritte für Objektmodelle:
AddOM fügt für $n \notin \mathcal{N}$ ein neues Objektmodell d in den Dokumentgraphen:

$$AddOM : \mathcal{DG} \times \mathcal{DOC}_{om} \times \mathcal{N} \rightharpoonup \mathcal{DG}$$
$$AddOM((N,E,D,R),d,n) = \mathcal{DS}((N,E,D,R),d,n,\emptyset,\emptyset)$$

ExtOM erweitert ein bereits definiertes Objektmodell:

$$ExtOM : \mathcal{DG} \times \mathcal{DOC}_{om} \times \mathcal{N} \times \mathcal{N} \rightharpoonup \mathcal{DG}$$
$$ExtOM((N,E,D,R),d,n_{neu},n) = \mathcal{DS}((N,E,D,R),fuseOMs\{D.n,d\},n_{neu},\{n\},\{n\})$$

Dabei gelten $D.n \in \mathcal{DOC}_{om}$ und $n \notin \mathcal{N}$.

IntOM integriert eine Menge gegebener Objektmodelle:

$$IntOM : \mathcal{DG} \times \mathbb{P}(\mathcal{N}) \rightharpoonup \mathcal{DG}$$
$$IntOM((N,E,D,R),M) = \mathcal{DS}((N,E,D,R),fuseOMs(D.M),M,M)$$

Dabei ist $M \neq \emptyset$ und $D.M \subseteq \mathcal{DOC}_{om}$. (□)

Aus Proposition 7.11 folgt sofort:

Proposition 7.13 (Entwicklungsschritte für Objektmodelle)
Die in Definition 7.12 angegebenen Entwicklungsschritte für Objektmodelle sind korrekt. (□)

Redundanztest eines Objektmodells

Der Redundanztest überprüft, ob ein Dokument im Dokumentgraphen redundant geworden ist, weil alle in ihm enthaltenen Informationen in den Nachfolgerdokumenten enthalten sind. Ein Objektmodell-Dokument ist redundant, wenn die Nachfolgerdokumente

1. alle Klassennamen,
2. alle Subklassenbeziehungen und
3. alle Datenbeziehungen einschließlich der Rollen definieren.

Dies kann durch andere Objektmodelle, aber auch durch Klassenbeschreibungen geschehen. Weil jede Datenbeziehung eines Objektmodells einen Rollennamen besitzt, kann das Objektmodell durch Klassenbeschreibungen redundant werden. Rollen werden dann als Attribute der entsprechenden Klassen interpretiert, die eine Datenbeziehung realisieren.

7.4 Zusammenfassung

In diesem Kapitel haben wir Dokumentarten definiert, die benutzt werden, um den statischen Anteil von Systemen zu charakterisieren. Die in vielen objektorientierten Entwicklungsmethoden unter dem Begriff Objektmodell zusammengefaßten Beschreibungstechniken dienen zur Darstellung von

- Datentypen,
- Klassenbeschreibungen mit Methoden und Attributen sowie
- dem eigentlichen Objektmodell mit dessen Vererbungs- und Datenbeziehungen.

Wir haben dafür drei unterschiedliche Dokumentarten $\mathcal{DOC}_{dt}$, $\mathcal{DOC}_{cl}$ und $\mathcal{DOC}_{om}$ entwickelt. Wir haben jeweils:

- die abstrakte Syntax definiert,
- eine konkrete (graphische) Darstellung anhand von Beispielen vorgeschlagen,
- eine auf dem Systemmodell basierende Semantik angegeben,
- die Folgerungsbeziehung für Dokumente untersucht,
- Entwicklungsschritte für den Dokumentgraphen angegeben und deren Korrektheit gezeigt.

Kapitel 8

Automatendokumente

In den Kapiteln 4 und 5 wurde die theoretische Basis für buchstabierende Automaten entwickelt. Im Kapitel 6 sind Verfeinerungsschritte für buchstabierende Automaten in Form eines Kalküls charakterisiert. Diese Theorie buchstabierender Automaten basiert auf einer mathematischen Modellierung des Automatenbegriffs, der unendliche Mengen von Zuständen und Transitionen zuläßt. In Abschnitt 4.2 wurden bereits einige Darstellungstechniken besprochen, die eine graphische Darstellung von Automaten ermöglichen.

Darauf aufbauend wird in diesem Kapitel eine konkrete Syntax für Automatendokumente entwickelt. Wir setzen Automaten ein, um das Verhalten von Agenten des Systemmodells zu beschreiben. Diese Beschreibung beschränkt sich nicht auf die Ein- und Ausgabe eines Agenten, sondern legt auch dessen Zustand fest. Damit werden Automaten eingesetzt, um den *Lebenszyklus* ([RUM91]) von Agenten zu definieren.

Im ersten Abschnitt wird eine Darstellungsform für Automatenzustände sowie Ein- und Ausgabezeichen eingeführt, die es erlaubt, den unendlichen Zustandsraum sowie die unendlichen Mengen der Ein- und Ausgabezeichen eines Agenten in endlich viele Äquivalenzklassen zu partitionieren. Ein Beispiel demonstriert die konkrete Darstellung eines Automaten.

Der zweite Abschnitt behandelt die Definition der Syntax und der Semantik von Automatendokumenten. Dabei werden Kontextbedingungen für Automatendokumente definiert, die nicht automatisch überprüfbar sind, weil Prädikate als Vor- und Nachbedingungen für Transitionen verwendet werden können. Die Semantik eines Automatendokuments wird durch Abbildung auf das Systemmodell definiert. Der Lebenszyklus eines Automatendokuments wird dabei zunächst zu einem buchstabierenden Automaten transformiert. Dieser wird dann benutzt, um das Verhalten von Agenten zu bestimmen.

Der dritte Abschnitt beschreibt, wie die Neuerstellung und die Verfeinerung von Automatendokumenten durchgeführt wird. Dabei werden Generatoren für entstehende Beweisverpflichtungen eingeführt. Die angegebenen Transformationsschritte werden auf den Verfeinerungskalkül 6.17 zurückgeführt. Neben der Verhaltensverfeinerung für Agenten sind in diesem Abschnitt auch zwei Spezialisierungsoperationen angegeben, die sich mit

den beiden unterschiedlichen Arten von Automaten, dem Typ- und dem Klassenautomaten beschäftigen. Insbesondere wird dabei gezeigt, daß Vererbung von Verhalten zwischen Klassen einen Spezialfall der Verfeinerungsoperationen auf Automaten darstellt.

Im vierten Abschnitt dieses Kapitels wird die Einbettung der Automatendokumente in den Dokumentgraphen behandelt. Dabei wird die Integration mehrerer Lebenszyklen verschiedener Automatendokumente besprochen und Entwicklungsschritte und ein Redundanztest für Automatendokumente behandelt.

Der fünfte Abschnitt behandelt einige mögliche Erweiterungen und Alternativen für Automatendokumente. So werden einige Taktiken definiert, die beschreiben, wie Kombinationen von Verfeinerungsschritten des ersten Abschnitts verwendet werden können, um komplexere Verfeinerungen durchzuführen.

8.1 Konzepte für Automatendokumente

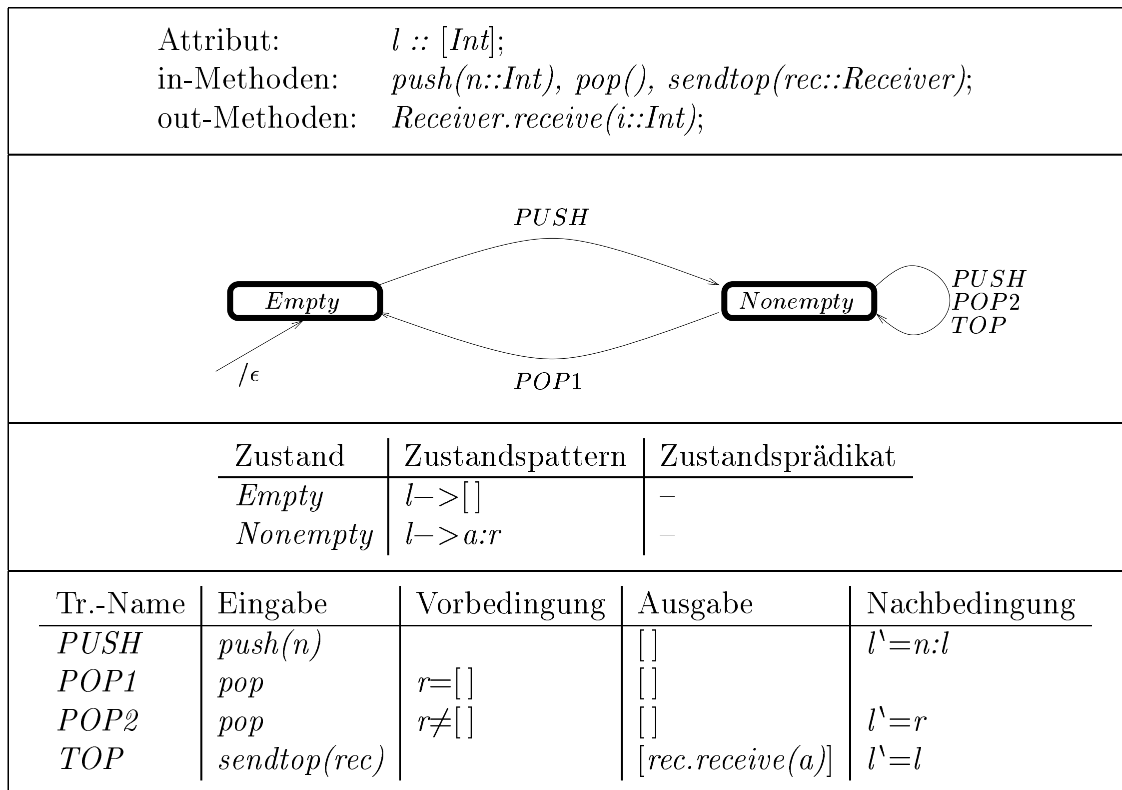
In diesem Abschnitt entwickeln wir Konzepte für die konkrete syntaktische Darstellung von Automatendokumenten. Wir erläutern diese zunächst an einem Beispiel und diskutieren dann, welche Abstraktionsmechanismen wir für die endliche Darstellung unendlicher Mengen benötigen. In einem weiteren Abschnitt charakterisieren wir erweiterbare Datentypen für spezielle Variablen wie `self`.

8.1.1 Beispielautomat Stack

In Abbildung 8.1 ist ein Automat angegeben, der einen Stack von ganzen Zahlen verwaltet. Ein Stack eignet sich deshalb sehr gut als Beispiel, weil dieses Konzept allgemein bekannt ist und wesentliche Eigenschaften wie unendliche Zustands- und Nachrichtenmengen besitzt.

Die Methoden *push* und *pop* modifizieren einen Stack. Die Methode *sendtop* führt zu einer Weiterleitung des obersten Elements an den Empfänger der Klasse *Receiver*. In der ersten Tabelle der Abbildung 8.1 sind Attribute und die ein- und ausgehenden Nachrichten eines *Stack*-Agenten aufgeführt. Die Liste der ein- und ausgehenden Nachrichten dient nur zur Illustration. Sie ist nicht Teil des Automatendokuments. Die zwei Automatenzustände *Empty* und *Nonempty* des Transitionsdiagramms charakterisieren jeweils eine Äquivalenzklasse von Datenzuständen. Diese Äquivalenzklassen werden in der zweiten Tabelle durch Patterns auf dem Attribut l erklärt. Die Möglichkeit Zustandsprädikate anzugeben ist hier nicht genutzt. Die Komponenten dieser Patterns werden durch Variablen bezeichnet (hier a und r) und können im Zustandsprädikat sowie bei der Vor- und Nachbedingung von Transitionen verwendet werden.

Alle Transitionen werden in dieser graphischen Variante mit Namen attributiert. In einer dritten Tabelle wird der Inhalt jeder Transition dargestellt. Der Graph legt Quell- und



Attribut: $l :: [Int]$;
in-Methoden: *push(n::Int), pop(), sendtop(rec::Receiver)*;
out-Methoden: *Receiver.receive(i::Int)*;

Zustand	Zustandspattern	Zustandsprädikat
Empty	$l\text{->}[\,]$	–
Nonempty	$l\text{->}a\text{:}r$	–

Tr.-Name	Eingabe	Vorbedingung	Ausgabe	Nachbedingung
PUSH	*push(n)*		[ ]	$l\text{`}=n\text{:}l$
POP1	*pop*	$r=[\,]$	[ ]	
POP2	*pop*	$r\neq[\,]$	[ ]	$l\text{`}=r$
TOP	*sendtop(rec)*		[*rec.receive(a)*]	$l\text{`}=l$

Abbildung 8.1: *Stack*-Automat

Zielzustand einer Transition fest, so daß in der Tabelle nur das Eingabepattern, die Vorbedingung, der Ausgabeausdruck und die Nachbedingung festzulegen sind. Das Eingabepattern und die Vorbedingung charakterisieren Nachrichten, die von der Transition verarbeitet werden können. In der Vorbedingung können die Variablen des Eingabepattern, der Zustandspatterns, Attributnamen und die spezielle Variable `in` verwendet werden. Letztere steht für den Wert der Eingabenachricht. Analog können in der Nachbedingung Variablen des Eingabepattern und des Ausgabeausdrucks sowie Zustandsvariablen des Quellzustands (hier l) und des Zielzustands (hier $l\text{`}$) verwendet werden.

Spezielle Datentypen werden benutzt, um Ein- und Ausgabenachrichten zu definieren. Der Datentyp `IN` definiert die Menge aller Eingabenachrichten, die ein Automat verarbeiten kann. Methodennamen werden dabei als Konstruktoren verstanden und deren Argumente als Konstruktorargumente. Ähnlich wird die Sorte `OUT` für Ausgabenachrichten definiert:

`data IN` = *push Int* | *pop* | *sendtop Receiver* | ..
`data OUT` = *Receiver.receive Int* | ..

Die Punkte .. bedeuten, daß der Datentyp um neue Konstruktoren erweitert werden kann.

Aufgrund der als Bedingungen verwendeten prädikatenlogischen Ausdrücke muß der Automat einige Eigenschaften erfüllen, die wir in Form von Kontextbedingungen noch

genauer formalisieren werden. Am Beispiel des Automaten *Stack* lassen sich diese gut motivieren.

Mit der Definition eines Automaten ist eine intuitive Vorstellung verbunden, die der angegebene Automat vermittelt. Es ist wichtig, daß diese mit der Semantik übereinstimmt. Sonst wäre eine graphische Darstellung weniger eine Unterstützung bei der Softwareentwicklung als vielmehr ein Hindernis.

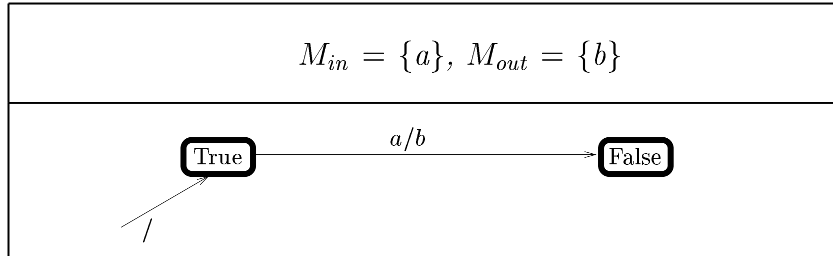


Abbildung 8.2: Automat mit unerfüllbarem Zustand

Betrachten wir zum Beispiel den Automaten aus Abbildung 8.2. Dem linken Automatenzustand ist der komplette Datenzustandsraum zugeordnet. Der rechte Automatenzustand ist nicht erfüllbar. Es gibt mehrere Alternativen, Abbildung 8.2 zu interpretieren bzw. zu formalisieren. Keine ist wirklich befriedigend. Zum Beispiel kann der linke Zustand im Widerspruch zur Intuition als partiell in bezug auf Eingabe a gelten, weil die Transition a/b nicht schalten kann. Alternativ dazu kann der linke Zustand als total angenommen werden, wobei jedoch kein Zielzustand existiert. Ähnliche Probleme ergeben sich bei einer nicht erfüllbaren Nachbedingung einer Transition.

Wir fordern daher folgende Bedingungen für Automatendokumente, um eine Übereinstimmung von Intuition und Semantik zu sichern:

- Automatenzustände bilden nichtleere Äquivalenzklassen von Datenzuständen (Erfüllbarkeit und Disjunktheit der Zustandsprädikate).
- Die Schaltbereitschaft einer Transition hängt nur vom Quellzustand und der Vorbedingung ab (Enabledness von Transitionen).

Die erste Bedingung kann im Beispiel automatisch überprüft werden, da keine prädikatenlogischen Ausdrücke, sondern nur Patterns zur Partitionierung des Datenzustandsraums verwendet wurden. Würden aus den Zustandsbeschreibungen Beweisverpflichtungen erzeugt, so würden diese wie folgt aussehen:

$$\exists l.\ l=[\,]$$
$$\exists l,a,r.\ l=a{:}r$$
$$\neg(l=[\,] \land (\exists a,r.\ l=a{:}r))$$

Folgende beiden Beweisverpflichtungen entstehen aus den beiden mit *PUSH* markierten Transitionen, um die Enabledness-Bedingung zu sichern:

$$l=[\,] \land \texttt{in}=push(n) \Rightarrow \exists l',a',r',\texttt{out}.\ l'=n{:}l \land l'=a'{:}r' \land \texttt{out}=[\,]$$
$$l=a{:}r \land \texttt{in}=push(n) \Rightarrow \exists l',a',r',\texttt{out}.\ l'=n{:}l \land l'=a'{:}r' \land \texttt{out}=[\,]$$

Die drei restlichen Transitionen *POP1*, *POP2* und *TOP* erzeugen folgende Axiome:

$$l{=}a{:}r \wedge \texttt{in}{=}pop \wedge r{=}[\,] \Rightarrow \exists l`,\texttt{out}.\ l`{=}[\,] \wedge l`{=}[\,] \wedge \texttt{out}{=}[\,]$$
$$l{=}a{:}r \wedge \texttt{in}{=}pop \wedge r{\neq}[\,] \Rightarrow \exists l`,a`,r`,\texttt{out}.\ l`{=}r \wedge l`{=}a`{:}r` \wedge \texttt{out}{=}[\,]$$
$$l{=}a{:}r \wedge \texttt{in}{=}sendtop(rec) \Rightarrow \exists l`,a`,r`,\texttt{out}.\ l`{=}l \wedge l`{=}a`{:}r` \wedge \texttt{out}{=}[rec.receive(a)]$$

Zu beachten ist, daß die angegebenen Axiome jeweils fordern, daß Ausgaben und Zielzustände einen definierten Wert darstellen, da wir fordern, daß $\exists l`,\texttt{out}$ nur über definierte Werte quantifiziert. Die freien Variablen der angegebenen Axiome sind außen $\forall$-quantifiziert.

Unter den angegebenen Beweisverpflichtungen ist keine, die nicht von einem Theorembeweiser automatisiert oder unter wenig manueller Unterstützung bewiesen werden könnte. Dies ist einer der großen Vorteile unserer Vorgehensweise bei formaler Softwareentwicklung. Wir nutzen strukturierte Dokumentarten wie Automaten teilweise unter Zuhilfenahme von prädikatenlogischen Aussagen. Anstatt alles in Axiome umzusetzen, und dabei die durch den Automaten vorgegebene Strukturierung zu verlieren, erhalten wir diese Strukturierung und nutzen sie, um Eigenschaften soweit wie möglich automatisiert zu überprüfen. Nur an Stellen, die eine automatisierte Überprüfung nicht zulassen, wird ein Theorembeweiser verwendet. Dabei sind Beweisverpflichtungen zu zeigen, die gegenüber einer Umsetzung des gesamten Dokuments in eine große Formel wesentlich kleiner und damit leichter behandelbar sind.

Werden darüberhinaus Patterns statt nur prädikatenlogische Formeln verwendet, so

- können manche Prüfungen automatisiert werden,
- kann eine elegantere Darstellung verwendet werden (vgl. Pattern-Matching funktionaler Sprachen) und
- ist Rapid-Prototyping durch Umsetzung in eine funktionale Sprache möglich.

Wir werden deshalb in Automatendokumenten neben prädikatenlogischen Ausdrücken auch Patterns nutzen, obwohl die Beschreibungsmächtigkeit dadurch nicht erhöht wird. Ein Pattern der Form $l{-}{>}a{:}r$ kann als Gleichung $l{=}a{:}r$ aufgefaßt werden. Dies erlaubt eine technisch einfach behandelbare *patternfreie* Darstellung von Transitionen und Initialelementen, indem alle Patterns und Wertausdrücke als Gleichungen in die Bedingungen aufgenommen werden. Die Transitionsrelation und die Initialmenge lassen sich dann in eine in Tabelle 8.3 dargestellte Form bringen. Wir erläutern die dazu notwendigen Umformungen an der Transition *POP2*. Die Vorbedingung wird um das Quellzustandsprädikat erweitert und die Patternvariablen $\exists$-quantifiziert:

$$\exists a,r.\ l{=}a{:}r \wedge \texttt{in}{=}pop \wedge r{\neq}[\,]$$

und (falls möglich) vereinfacht. Die Erweiterung der Vorbedingung um das Quellzustandsprädikat ist notwendig, um die Patternvariablen (hier r) korrekt zu binden.

Die Nachbedingung wird um das Quell- und Zielzustandsprädikat sowie die alte Vorbedingung erweitert:

$$\exists a,r,a`,r`.\ l{=}a{:}r \wedge \texttt{in}{=}pop \wedge r{\neq}[\,] \wedge l`{=}a`{:}r` \wedge \texttt{out}{=}[\,] \wedge l`{=}r$$

und unter Annahme der Gültigkeit der neuen Vorbedingung sowie beider Zustandsprädikate ebenfalls vereinfacht:

$$\exists a,r.\ l{=}a{:}r \wedge \texttt{out}{=}[\,] \wedge l`{=}r.$$

Dies sind für Automaten semantikerhaltende Umformungen. Dabei werden alle freien Variablen bis auf die Attribute sowie die Variablen `in` und `out` gebunden. Die beiden Darstellungen des *Stack*-Automaten zeigen, daß die Benutzung von Patterns bei der Darstellung von Automaten zu wesentlich lesbareren Dokumenten führt.

Zustand	Zustandsprädikat
Empty	$l{=}[\,]$
Nonempty	$\exists a,r.\ l{=}a{:}r$

Tr.-Name	Vorbedingung	Nachbedingung
PUSH	$\exists n.\ \texttt{in}{=}push(n)$	$\exists n.\ \texttt{in}{=}push(n) \wedge \texttt{out}{=}[\,] \wedge l`{=}n{:}l$
POP1	$\exists a.\ l{=}[a] \wedge \texttt{in}{=}pop$	$\texttt{out}{=}[\,]$
POP2	$\exists a,r.\ l{=}a{:}r \wedge \texttt{in}{=}pop \wedge r{\neq}[\,]$	$\exists a,r.\ l{=}a{:}r \wedge \texttt{out}{=}[\,] \wedge l`{=}r$
TOP	$\exists rec.\ \texttt{in}{=}sendtop(rec)$	$\exists a,r,rec.\ l{=}a{:}r \wedge \texttt{in}{=}sendtop(rec) \wedge \texttt{out}{=}[rec.receive(a)] \wedge l`{=}l$

Abbildung 8.3: Patternfreie Darstellung des *Stack*-Automaten (ohne Bild)

8.1.2 Abstraktion und Darstellung

Die konkrete Darstellung von Automaten und deren Komponenten ist notwendigerweise endlich. Der Datenzustand eines Agenten und dessen Menge von Ein- und Ausgabenachrichten sind aufgrund der verwendeten Datentypen meist unendlich. Dadurch sind wir gezwungen, unendliche Mengen in endlicher Form darzustellen. Dazu bietet sich eine Äquivalenzklassenbildung unter Benutzung einer Logiksprache an.

Zustandsräume

Der Zustandsraum eines Agenten ist durch die Menge seiner Attribute Σ_{attr} festgelegt. Attribute besitzen im allgemeinen einen unendlichen Wertebereich. Dies impliziert, daß der Zustandsraum eines Agenten unendlich groß ist. Selbst wenn ein Agent nur endlich viele *Datenzustände* hätte, ist oft aus praktischen Gründen eine Abstraktion vom konkreten Zustandsraum des Agenten in eine handhabbare, endliche Menge von *Automatenzuständen* sinnvoll. Damit wird es möglich, die Menge der Automatenzustände graphisch darzustellen. Ein Automatenzustand korrespondiert also zu einer Menge von Datenzuständen. Diese Korrespondenz gibt es in zwei Varianten:

1. Die Automatenzustände partitionieren den Datenzustandsraum oder
2. einem Automatenzustand wird eine beliebige Teilmenge des Datenzustandsraums zugeordnet.

Im ersten Fall existiert eine Abbildung, die jedem Datenzustand genau einen Automatenzustand zuordnet. Also kann das Verhalten eines Agenten alleine aus dem Datenzustand des Agenten abgeleitet werden. Im zweiten Fall können sich die den Automatenzuständen zugeordneten Mengen von Datenzuständen beliebig überschneiden. Das Verhalten, das sich als Reaktion auf eine Eingabe ergibt, ist daher nicht direkt von den Datenzuständen ablesbar. Stattdessen ist eine weitere Komponente notwendig, die den *Kontrollzustand* des Agenten festlegt. Diese zusätzliche Komponente im Zustand eines Agenten kann durch eine geeignete Realisierung als Attribut in den Datenzustandsraum kodiert werden. Daher ist es nicht notwendig, einen weiteren Mechanismus zur Behandlung von Kontrollzuständen einzuführen. Wir benutzen daher die erste Variante, in der jeder Automatenzustand eine Äquivalenzklasse des Datenzustandsraums darstellt (siehe Beispiel *Stack*).

Die von uns gewählte Variante besitzt auch den Vorteil, daß die Kopplung zwischen Datenzuständen und Automatenzuständen wesentlich enger ist. Die intuitive Aussagekraft eines Transitionsdiagramms wird dadurch im allgemeinen deutlich erhöht. Nicht zuletzt deshalb haben Softwareentwicklungsmethoden, wie OMT ([RUM91]) diesen Standpunkt bei der Definition von Lebenszyklen ihrer Objekte eingenommen. Die Äquivalenzklassenbildung auf dem Datenzustandsraum kann nun auf mehrere Arten erfolgen. Für die Überprüfung, ob

1. einem Automatenzustand eine leere Datenzustandsmenge zugeordnet ist,
2. sich Automatenzustände überschneiden,
3. die Gesamtheit der Automatenzustände den Datenzustandsraum überdeckt, und
4. eine Zustandsverfeinerung korrekt ist,

muß die verwendete Sprache

1. abgeschlossen sein unter endlicher Schnitt- und Komplementbildung sowie
2. einen Test für den Vergleich mit dem Prädikat *False* (der leeren Menge) besitzen.

Aus der Vollständigkeit unter Schnitt und Komplement folgt natürlich die Vollständigkeit unter Vereinigung. Der Test auf Leerheit liefert dann Tests für Disjunktheit und Enthaltensein.

Die Prädikatenlogik erster Stufe erfüllt mit Hilfe von $\wedge$ und $\neg$ beide Anforderungen. Die Tests sind jedoch nicht effektiv ausführbar. Wenn wir nur Prädikatenlogik verwenden, müssen wir immer auf einen Theorembeweiser zur Verifikation zurückgreifen. Wir wollen stattdessen eine Technik verwenden, die es uns zumindest teilweise erlaubt, auf Verifikation zu verzichten. Dazu nutzen wir Patterns, die uns aus Definition der Datentypen in Abschnitt 7.1 zur Verfügung stehen. Patterns eignen sich als Strukturierungsmittel für Wertemengen eines Datentyps, weil ein Pattern die Teilmenge von Werten beschreibt,

die zu diesem Pattern passen. Wir wählen deshalb Patterns als Grundlage für unsere Äquivalenzklassenbildung. Aufgrund der für Patterns existierenden Unifikationsalgorithmen ([AVE95]) erhalten wir mit endlichen Mengen von Patterns eine Sprache, die beide oben genannten Kriterien erfüllt und dennoch mächtig ist.

Nachrichten

Analog zu dem Zustandsraum eines Agenten ist auch die Menge der von einem Agenten empfang- und versendbaren Nachrichten unendlich. Für eine endliche Darstellung ist damit vom *Nachrichtenraum* eine Abstraktion zu bilden. Der Nachrichtenraum *MSG* ist gemäß seiner Definition 3.11 bereits dadurch strukturiert, daß in jeder Nachricht ein Methodenname verwendet wird. Neben dem Methodennamen wird eine Belegung der Parameter der Methode übergeben. Die Menge der Nachrichten kann daher als funktionaler Datentyp verstanden werden, dessen Konstruktoren die Methodennamen sind (siehe Beispiel *Stack*).

Diese Sichtweise erlaubt es uns, bei der Abstraktion von Nachrichten ebenfalls mit Patterns zu arbeiten. Auch für Eingabenachrichten benötigen wir die Möglichkeit, Schnitte und Komplemente zu bilden, sowie den Test auf Enthaltensein. Einzelne Nachrichten können daher mittels eines Pattern charakterisiert werden. Ströme von Nachrichten, wie sie bei einer Transition als Ausgabe verwendet werden, können mit Hilfe von listenwertigen Ausdrücken behandelt werden. Diese Sprache erfüllt nicht alle oben angegebenen Kriterien, ist aber für die Definition der Ausgabe ausreichend.

8.1.3 Spezielle Datentypen

Das in Definition 3.14 eingeführte Attribut `actees` enthält die Menge aller Identifikatoren, die von einem Agenten aus aktiviert werden können. Um dieses Attribut in den Bedingungen verwenden zu können, führen wir den Datentyp `ACTEES` ein, der den Typ von `actees` darstellt und für keine andere Variable verwendet werden kann. In Transitionen wird die Eingabe und die Ausgabe ebenfalls durch Variablen benannt und entsprechend typisiert:

```
in : IN,
out : [OUT],
actees : ACTEES
```

Durch die Verwendung von Prädikaten zur Charakterisierung von Zuständen, Vor- und Nachbedingungen ist es möglich, Eigenschaften von Systemkomponenten implizit zu beschreiben. So ist es zum Beispiel möglich, die Menge der Eingabenachrichten festzulegen:

$$\forall i \in \mathtt{IN}.\ i=a \vee i=b. \quad (8.1)$$

Diese Bedingung wird bei jeder Erweiterung der Eingabemenge unerfüllbar. Um die Erweiterbarkeit der Nachrichtenmengen sicherzustellen, werden sogenannte erweiterbare Datentypdefinitionen für `in` und `out` eingeführt. Eine erweiterbare Datentypdefinition

kann als unvollständig betrachtet werden, in dem Sinn, daß nicht alle Konstruktorterme angegeben sind. Syntaktisch wird die erweiterbare Datentypdefinition unter Benutzung von Punkten .. dargestellt:

(8.2) $\texttt{data IN} = a \mid b \mid ..$

Mit dieser Datentypdefinition kann es Systeme geben, in denen Aussage (8.1) gilt. Hergeleitet werden kann diese Aussage aus (8.2) jedoch nicht[1]. Ein erweiterbarer Datentyp kann in Axiome einer Logik höherer Ordnung, z.B. *HOLCF*, umgesetzt werden. Diese Axiome sichern, daß je zwei verschiedene Konstruktorterme verschiedene Elemente denotieren. Es wird jedoch kein Induktionsaxiom, das die Termerzeugtheit des Datentyps beschreibt, angegeben. Für die Definition der beiden Sorten `IN` und `OUT` werden Methodensignaturen benutzt, die aus anderen Dokumenten, z.B. Klassenbeschreibungen, extrahiert werden. Die Erweiterbarkeit der Sorten `IN` und `OUT` reflektiert die Erweiterbarkeit der Methodensignaturen in diesen Dokumenten.

8.2 Syntax und Semantik von Automatendokumenten

Ein Automatendokument dient zur Charakterisierung des Verhaltens und der Zustandsfolgen eines Agenten und damit von dessen Lebenszyklus. Das Verhalten wird durch die Transitionsrelation beschrieben, die die Eingabe und den aktuellen Zustand mit der Ausgabe und dem nächsten Zustand in Beziehung setzt. Dabei wird mittels Patterns von den konkreten Werten abstrahiert und, wenn gewünscht, mittels Prädikaten eine präzise Verhaltensbeschreibung angegeben. Außerdem wird durch einen Klassennamen und eine Automatenart angegeben, für welche Agenten diese Automatenbeschreibung gültig ist.

Die Semantik eines Automatendokuments wird durch Rückführung auf die in Kapitel 5 definierte denotationelle Semantik von Automaten definiert. Auf diese Weise wird eine zweistufige Semantikdefinition für Automatendokumente vorgenommen. Zunächst werden diese in einen buchstabierenden Automaten umgesetzt, dann wird dieser Automat unter Benutzung der Eigenschaften aus Proposition 6.20 mit dem Systemmodell in Beziehung gesetzt.

8.2.1 Klassen- und Typautomaten

Jeder Automat ist einer Klasse zugeordnet. Er beschreibt das Verhalten der zu dieser Klasse gehörenden Agenten. Dabei gibt es zwei Varianten. Ein *Klassenautomat* beschreibt das Verhalten aller Instanzen seiner Klasse, ein *Typautomat* das aller Elemente seiner Klasse. Ein Typautomat ist allgemeiner, er beschreibt auch das Verhalten der Instanzen

[1] Die Erweiterbarkeit von Datentypen ist eine Erweiterung heutiger Konzepte funktionaler Datentypen, die zum Beispiel bei der Modularisierung von funktionalen Programmen verwendet werden könnte. In einem Grundmodul könnte der Datentyp $\texttt{data}\ Error = Ok \mid ..$ eingeführt werden, der in jedem weiteren Modul um spezifische Fehlermeldungen erweitert wird.

aus Subklassen. Ein Klassenautomat wird dagegen nicht vererbt und kann spezifischer das Verhalten der Instanzen einer Klasse beschreiben. Ein Klassenautomat wird daher als abstrakte Beschreibung der Implementierung einer Klasse betrachtet, während ein Typautomat die Schnittstelle einer Klasse beschreibt. Gemäß dem Substitutionsprinzip können sich Klienten einer Klasse darauf verlassen, daß ein Agent, der Element dieser Klasse ist, sich gemäß der Spezifikation des Typautomaten verhält. Für Klassenautomaten gilt dieses Substitutionsprinzip nicht. Wir unterscheiden beide Automatenarten durch die Schlüsselwörter `class` und `type`. Objektorientierte Methoden wie OMT ([RUM91]) benutzen nur Klassenautomaten, da keine Vererbung von Automaten betrachtet wird.

8.2.2 Konkrete Syntax eines Automatendokuments

Zur Darstellung von Automatendokumenten wird eine nicht weiter strukturierte Menge von Namen für Automatenzustände $\mathcal{AUT}$ eingeführt. Die endliche Menge der Automatenzustände wird mit Hilfe einer Tabelle mit den Datenzuständen des Agenten verbunden. Diese Tabelle ordnet jedem Automatenzustand ein Prädikat und ein Pattern für jedes Attribut zu.

Für die Notation der Eingabenachricht verwenden wir ein Pattern, für die Charakterisierung der Ausgabesequenz einen Wertausdruck. Zur genaueren Definition dienen Prädikate, die jedem Automatenzustand eine Äquivalenzklasse von Datenzuständen zuordnen und zu jeder Transition eine Vor- und Nachbedingung angeben. Neben dem konstruktiven Anteil des Automatendokuments gibt es die später näher beschriebenen Protokolle, die zur Semantik eines Automatendokuments nicht beitragen, aber die Korrektheit eines Automatendokuments sichern. Ein Protokoll beschreibt, wie ein Automatendokument entstanden ist, z.B. durch Transformation aus einem anderen Dokument, oder durch Neuerstellung. Protokolle sind notwendig, weil aufgrund der Benutzung von prädikatenlogischen Ausdrücken eine automatisierte Überprüfung der Kontextbedingungen nicht immer möglich ist.

Definition 8.1 (Dokumentart Automat)
Ein Tupel der Form $(c,ak,S,\Lambda,\delta,I)$ definiert die abstrakte Syntax eines Automatendokuments. Es besteht aus:

- *einem Klassennamen $c \in CN$,*
- *einer Automatenart $ak \in \{\texttt{class},\texttt{type}\}$,*
- *einer nichtleeren, endlichen Menge von Automatenzuständen $S \subseteq \mathcal{AUT}$,*
- *einer Abbildung $\Lambda \in S \to ((\langle var\rangle \rightharpoonup \langle patt\rangle) \times \langle pred\rangle)$, die jedem Automatenzustand ein Zustandsprädikat zuordnet,*
- *einer Transitionsrelation $\delta \subseteq S\times\langle patt\rangle\times\langle pred\rangle\times S\times\langle expr\rangle\times\langle pred\rangle$, die ebenfalls endlich ist und die Menge aller Transitionen, bestehend aus Quellzustand, Eingabepattern, Vorbedingung, Zielzustand, Ausgabeausdruck und Nachbedingung, charakterisiert, und*
- *einer nichtleeren, endlichen Initialmenge $I \subseteq S\times\langle expr\rangle\times\langle pred\rangle$.* (□)

Automaten, die mit diesen Dokumenten beschrieben werden, verarbeiten pro Transition eine Eingabenachricht und produzieren eine beliebige endliche oder unendliche Sequenz von Ausgabenachrichten. Der Datenzustandsraum eines Agenten wird bestimmt durch die Menge seiner Attribute. Sie kann aus dem Automatendokument bestimmt werden:

$$attr(\Lambda) \stackrel{def}{=} \{\texttt{self},\texttt{actees}\} \cup \bigcup_{s \in S} \bullet\pi_1(\Lambda s).$$

In der Attributmenge $attr(\Lambda)$ sind die beiden speziellen Attribute `self` und `actees` immer enthalten, müssen also in der konkreten Darstellung nicht explizit angegeben werden.

Die Charakterisierung einer Eingabenachricht durch ein Pattern und ein Prädikat (Vorbedingung) erlaubt hohe Flexibilität. So kann durch das angegebene Pattern zunächst eine Auswahl der Nachrichtenart (z.B. $push(*)$) getroffen werden. Freie Variablen des Patterns werden dabei gebunden und können in der Vorbedingung verwendet werden, um eine weitere Auswahl zu treffen. Die Vorbedingung darf dabei auch freie Variablen der Zustandspatterns verwenden. In vielen Fällen ist keine explizite Vorbedingung mehr nötig. Umgekehrt jedoch kann auch auf die Verwendung eines Patterns verzichtet werden und nur mit Hilfe einer Vorbedingung die Auswahl einer Teilmenge der Eingabenachrichten erfolgen.

Ähnlich können Nachbedingungen in Form eines Prädikats gestellt werden, wobei hier neben den durch den Ausgabeausdruck gebundenen Variablen auch Variablen des Eingabepatterns und des Datenzustandsraums verwendet werden können. Durch Verwendung einer Kopie der Variablen für den Nachfolgezustand, ähnlich wie in der Hoare-Logik ([HOA69]), ist es möglich, durch Vor- und Nachbedingungen die exakte Transformation des Datenzustandsraums zu beschreiben. Dazu benutzen wir den Apostroph .` als Operator, um von Variablen eine mit dem Apostroph versehene Kopie zu erzeugen: $\{a,b,\ldots\}^{`} = \{a^{`},b^{`},\ldots\}$. Die Typisierung dieser Variablen aus $VAR^{`}$ bleibt erhalten. Der Operator .` wird kanonisch auf Wertausdrücke und Prädikate erweitert.

8.2.3 Patternfreie Syntax für Automaten

Wie wir bereits im Abschnitt 8.1.1 motiviert haben, dienen Patterns zur übersichtlicheren Darstellung von Prädikaten für Zustände, Vor- und Nachbedingungen. Die Übersetzung der im Automatendokument genutzten Patterns in Formeln und deren Integration mit bereits existenten Formeln führt zu einer äquivalenten patternfreien Darstellung von Automatendokumenten, die wir im Rest dieses Kapitels verwenden werden.

Definition 8.2 (Patternfreie Dokumentart Automat)
Ein Tupel der Form $(c,ak,S,\Lambda,\delta,I)$ definiert die patternfreie abstrakte Syntax eines Automatendokuments. Es besteht aus:

- *einem Klassennamen $c \in CN$,*
- *einer Automatenart $ak \in \{\texttt{class},\texttt{type}\}$,*
- *einer nichtleeren, endlichen Menge von Automatenzuständen $S \subseteq \mathcal{AUT}$,*

- *einer Abbildung* $\Lambda \in S \rightarrow \langle pred \rangle$, *die jedem Automatenzustand ein Zustandsprädikat zuordnet,*
- *einer endlichen Transitionsrelation* $\delta \subseteq S \times \langle pred \rangle \times S \times \langle pred \rangle$ *und*
- *einer nichtleeren, endlichen Initialmenge* $I \subseteq S \times \langle pred \rangle$.

Die Menge dieser Automatendokumente wird mit $\mathcal{DOC}_a \subseteq \mathcal{DOC}$ *bezeichnet.* $(\Box)$

Gegenüber den Automatendokumenten aus Definition 8.1 sind alle Patterns und Wertausdrücke als Gleichungen in die prädikatenlogischen Ausdrücke aufgenommen worden. Wir haben dies in Abbildung 8.3 am *Stack*-Automaten demonstriert. Zur Vereinfachung nachfolgender Definitionen führen wir die Abkürzung $\Delta \overset{def}{=} \mathcal{AUT} \times \langle pred \rangle \times \mathcal{AUT} \times \langle pred \rangle$ für die Menge aller Transitionen ein. Darüberhinaus steht A im Rest dieses Kapitels für $(c, ak, S, \Lambda, \delta, I)$.

Für eine Transition $T=(s,ci,t,co) \in \delta$ heißt die Menge aller Paare aus Datenzustand und Eingabezeichen, die T schalten lassen, deren *Schaltbereich*. Für ein gegebenes Paar des Schaltbereichs wird der *Zielbereich* von T als Menge der Paare aus Zielzustand und Ausgabe definiert, die T schalten lassen.

Die *Vorbedingung* der Transition T wird mit $T_{pre}=(\Lambda s \wedge ci)$ und die *Nachbedingung* mit $T_{post}=((\Lambda t)^{\backprime} \wedge co)$ bezeichnet. Die Vorbedingung bestimmt den Schaltbereich und die Nachbedingung den Zielbereich in Abhängigkeit des Schaltbereichs. Wir verwenden T auch als *Transitionsprädikat* in der Form $T=(T_{pre} \wedge T_{post})$.

Entsprechend wird jedes Initialelement $(t,c) \in I$ auch als *Initialprädikat* $(\Lambda t \wedge c)$ verstanden.

8.2.4 Kontextbedingungen an ein Automatendokument

Die Umsetzung der Theorie buchstabierender Automaten in eine konkrete Syntax führt uns zu Kontextbedingungen für Automatendokumente. Wir wollen in dieser Arbeit nicht zu tief auf Kontextbedingungen eingehen und fordern daher sehr allgemein, daß nachfolgende Eigenschaften gelten. Im weiteren werden wir die Gültigkeit dieser Kontextbedingungen voraussetzen.

KB1: Die in Automatendokumenten vorkommenden Patterns, Wert- und Prädikatenausdrücke müssen wohlgeformt sein. Insbesondere haben die Variablen `in` und `out` feste Sorten, die sich aus den Eingabe- und Ausgabemethoden ableiten.

KB2: Die speziellen Attribute `self` und `actees` sind immer Teil der Attributmenge *attr*(Λ) und müssen nicht explizit angegeben werden. `self`$^{\backprime}$ darf nicht verwendet werden. Es gilt immer: `self`$^{\backprime}$=`self`.

KB3: Die Variablen `actees` und `actees`$^{\backprime}$ dürfen nur in der Nachbedingung in der Form $\exists v.$`actees`$=v\,\hat{}\,$`actees`$^{\backprime}$ verwendet werden, wobei die Sorte von v sichert, daß dabei nur endlich viele Objektidentifikatoren aus `actees` entnommen werden.

Weil der Datentyp der Variable `self` eine Identifikatorsorte ist von der keine Konstruktoren bekannt sind und nur die Gleichheit darauf existiert, kann die Variable `self` in Prädikaten nur in eingeschränkter Form verwendet werden. Dies sichert, daß alle formulierbaren Prädikate uniform über `self` sind. Das heißt, die Erfüllbarkeit eines Prädikats ist unabhängig von einer etwaigen Vorgabe der Belegung der Variable `self`. Die Einschränkung von `actees` modelliert die Tatsache, daß bei jeder Transition nur endlich viele Objekte aktiviert werden können. Obige syntaktische Bedingung ist damit hinreichend für die in Definition 3.15 geforderten Eigenschaften von Agenten.

In den Abschnitten 8.1.1 und 8.1.2 haben wir motiviert, welche weiteren Kontextbedingungen an ein Automatendokument gestellt werden müssen, damit die Semantikbildung mit der Intuition übereinstimmt. Dabei haben wir folgende zu verifizierenden Bedingungen gefordert, die für Systeme $sys{\in}\mathcal{SM}$, gelten sollen:

VB1: **(Erfüllbarkeit der Zustandsprädikate)** Die Abbildung Λ darf einem Automatenzustand keine leere Menge von Datenzuständen zuordnen.

$$\forall s{\in}S.\exists\beta{\in}BEL.(sys,\beta)\models\Lambda s \tag{8.3}$$

VB2: **(Disjunktheit der Zustandsprädikate)** Die Abbildung Λ muß allen Automatenzuständen disjunkte Datenzustände zuordnen:

$$\forall s,t{\in}S,\beta_1,\beta_2{\in}Bel_{attr(\Lambda)}.\ (sys,\beta_1)\models\Lambda s \wedge (sys,\beta_2)\models\Lambda t \Rightarrow s{=}t \tag{8.4}$$

VB3: **(Enabledness von Transitionen)** Transitionen dürfen keine falsifizierenden Nachbedingungen enthalten. Das heißt, für jedes Paar aus Quellzustand und Eingabe, das die Vorbedingung erfüllt, muß ein Zielzustand und eine Ausgabe existieren, die die Nachbedingung erfüllen.

$$\forall T{\in}\delta,\beta{\in}Bel_{attr(\Lambda)\cup\{in\}}.\ (sys,\beta)\models T_{pre} \Rightarrow \exists\beta_1{\in}BEL.\ (sys,\beta{+}\beta_1)\models T_{post} \tag{8.5}$$

Wenn die Bedingungen VB1 und VB2 erfüllt sind, dann nimmt die Abstraktion Λ eine Äquivalenzklassenbildung auf dem Datenzustandsraum vor. Neben den benannten Äquivalenzklassen der Menge S gibt es eine weitere, unbenannte Äquivalenzklasse, die allen Datenzuständen entspricht, die keinem Automatenzustand zugeordnet sind. Diese korrespondiert zu dem impliziten Fehlerzustand $\pm$ der bei der Totalisierung von Automaten in Definition 5.10 verwendet wird. Bedingung VB3 besagt, daß für das Schalten einer Transition T nur die Vorbedingung T_{pre} erfüllt sein muß. In diesem Fall findet sich immer ein Zielzustand und ein Ausgabestrom, die die Nachbedingung T_{post} erfüllen.

8.2.5 Transformation in einen buchstabierenden Automaten

Die komplexe Struktur des Automatendokuments wird nun durch eine Transformation in die einfachere Struktur eines buchstabierenden Automaten umgesetzt. Dabei geht die durch das Automatendokument vorgegebene Strukturierung verloren.

Nachfolgend werden Elemente des Systemmodells benutzt, um abstrakte Automatenzustände in Datenzustände zu transformieren. Weil Datenzustände Teil des Systemmodells sind, wird diese Transformation relativ zu einem System des Systemmodells durchgeführt.

Definition 8.3 (Umsetzung eines Automatendokuments)
Die Transformation automaton übersetzt Automatendokumente in buchstabierende Automaten. Diese Transformation wird relativ zu einem gegebenen System $sys \in \mathcal{SM}$ *und für einen Agenten* $id \in ID_{sys}$ *durchgeführt, der von der angegebenen Klasse ist:* $id\ isof_{sys}\ c$.

$$
\begin{array}{l}
automaton_{sys,id}(c,ak,S,\Lambda,\delta,I) \stackrel{def}{=} (S',M_{in}',M_{out}',\delta',I') \\
\texttt{where} \\
\quad att = \Sigma_{attr,sys}.c \\
\quad S' = \{\ \beta \in BEL_{sys} \mid \bullet\beta = att \wedge \beta.\texttt{self} = id\ \} \\
\quad M_{in}' = \{\ (a,mn,\beta) \in MSG_{sys} \mid a = id\ \} \\
\quad M_{out}' = MSG_{sys} \\
\quad I' = \{\ (\beta|_{att},\beta.\texttt{out}) \in S' \times (M_{out}')^{\omega} \mid \beta \in BEL_{sys} \wedge \exists E \in I.\ (sys,\beta) \models E\ \} \\
\quad \delta' = \{\ (\beta_s,\beta.\texttt{in},\beta_t,\beta.\texttt{out}) \in S' \times M_{in}' \times S' \times (M_{out}')^{\omega} \mid \\
\qquad\qquad \exists T \in \delta,\ \beta \in BEL_{sys}.\ (sys,\beta) \models T \wedge \beta|_{att} = \beta_s \wedge \beta|_{att`} = \beta_t`\ \}
\end{array}
$$

($\square$)

Die Zustandsmenge des Automaten setzt sich aus allen Belegungen der Attribute von Agent *id* zusammen. Die Transformation $automaton_{sys}$ erzeugt einen im allgemeinen nichtdeterministischen und partiellen Automaten. Bei der Transformation *automaton* wurde die Automatenart *ak* nicht benutzt.

Proposition 8.4 (Eigenschaften der Transformation *automaton***)**
Sei $(c,ak,S,\Lambda,\delta,I) \in \mathcal{DOC}_a$ *ein Automatendokument,* $sys \in \mathcal{SM}$ *ein System des Systemmodells und* $id \in ID_{sys}$ *mit* $id\ isof_{sys}\ c$*, dann gilt für den erzeugten buchstabierenden Automaten*

$$(S',M_{in}',M_{out}',\delta',I') = automaton_{sys,id}(c,ak,S,\Lambda,\delta,I):$$

1. *Die Menge der Zustände ist nicht leer:* $S' \neq \emptyset$.
2. *Die Menge der Automatenzustände partitioniert den Datenzustandsraum. Die einzig leere Äquivalenzklasse des Datenzustandsraums kann die Klasse der Datenzustände, die keinem Automatenzustand zugeordnet sind, sein:*

$$
\begin{array}{l}
\forall s \in S.\ \exists s' \in S'.\ (sys,s') \models \Lambda s \\
\forall s,t \in S, s' \in S'.\ (sys,s') \models \Lambda s \wedge (sys,s') \models \Lambda t \Rightarrow s = t
\end{array}
$$

3. *Die Schaltbereitschaft einer Transition aus* δ *hängt nur von der Vorbedingung ab:*

$$
\begin{array}{l}
\forall T \in \delta.\ s \in S',\ m \in M_{in}'.\ (sys,s+(\texttt{in} \mapsto m)) \models T_{pre} \Rightarrow \\
\quad \exists t \in S', out \in M_{out}'.\ (sys,s+t`+(\texttt{in} \mapsto m,\texttt{out} \mapsto out)) \models T
\end{array}
$$

($\square$, **Beweis siehe** C.38)

8.2.6 Semantik eines Automatendokuments

Mit der obigen Definition kann nun die Semantik eines Automatendokuments definiert werden.

Definition 8.5 (Semantik eines Automatendokuments)
Die Semantik eines Automatendokuments A=(c,ak,S,Λ,δ,I) wird definiert durch:

$$(8.6)\quad [\![A]\!]^a \stackrel{def}{=} \{\ sys \in \mathcal{SM} \mid c \in CN_{sys} \wedge$$
$$(8.7)\quad \forall a \in ID_{sys}.\ ((ak = \texttt{class} \wedge class_{sys}(a) = c) \vee (ak = \texttt{type} \wedge a\ isof_{sys}\ c)) \Rightarrow$$
$$(8.8)\quad \exists ba.\ automaton_{sys,a}(A) \stackrel{*}{\rightsquigarrow} ba \wedge [\![ba]\!]^t = behavior_{sys}.a\ \ \}$$

(□)

Der Kern dieser Semantikdefinition besteht darin, die gegebene abstrakte Syntax des Automatendokuments mit der Transformation *automaton* aus Definition 8.3 zunächst in einen buchstabierenden Automaten zu übersetzen. Das Verhalten aller betroffenen Agenten ist nun eine Verfeinerung *ba* dieses Automaten und damit eine Verfeinerung von dessen gezeiteter Semantik. Nach Proposition 6.20 und Satz 6.18 ist ein Automatendokument *A'* mit gleicher Klasse *c* und gleicher Automatenart *ak* eine Verfeinerung von *A*, wenn gilt:

$$\forall a \in ID.\ automaton_{sys,a}(A) \stackrel{*}{\rightsquigarrow}\ automaton_{sys,a}(A').$$

Aufgrund dieser Eigenschaft läßt sich der Verfeinerungskalkül aus Definition 6.17 auf Automatendokumente übertragen.

8.3 Automatenerstellung und -transformation

In diesem Abschnitt behandeln wir die Neuerstellung eines Automatendokuments und die Verfeinerung gegebener Dokumente. Dabei entwickeln wir syntaxbasierte Beweisverpflichtungen, deren Gültigkeit die Korrektheit des neu erstellten bzw. durch Transformation entstandenen Dokuments sichert.

Bei der Neuerstellung eines Dokuments sind die Kontextbedingungen (KB1–KB3, VB1–VB3) einzuhalten, um die Wohlgeformtheit des Dokuments zu sichern.

Die Verfeinerung eines Automatendokuments wird als Spezialisierung der Information über das zu implementierende System verstanden. Dies erfolgt durch Hinzunahme von Information über das Verhalten von Agenten des Systems. Zu diesem Zweck übertragen wir den in Kapitel 6 gegebenen Verfeinerungskalkül auf Automatendokumente. Bei der Verfeinerung entstehen Beweisverpflichtungen, die einerseits die Wohlgeformtheit des Dokuments und andererseits die Verfeinerungseigenschaft gegenüber dem Vorgängerdo-

kument sichern. Auf den Umgang mit Beweisverpflichtungen wird im Abschnitt 8.4.1 eingegangen.

8.3.1 Neuerstellung eines Automatendokuments

Bei der Neuerstellung eines Automatendokuments ist ein korrekter Lebenszyklus anzugeben, der alle Kontextbedingungen erfüllt. Weil das Systemmodell in den Bedingungen VB1–VB3 benutzt wird, sind diese Bedingungen auf Basis semantischer Eigenschaften formuliert und daher nicht syntaktisch überprüfbar. Daher müssen andere, auf syntaktischer Ebene formulierte Bedingungen angegeben werden, die für das Einhalten dieser semantischen Bedingungen hinreichend sind. Diese formulieren wir in Form von *Beweisverpflichtungen*. Dies haben wir bereits am Beispiel des *Stack*-Automaten in Abschnitt 8.1.1 gezeigt. Eine Beweisverpflichtung ist eine prädikatenlogische Formel, deren Gültigkeit für die Korrektheit eines Automatendokuments notwendig ist. Eine hinreichende endliche Menge von Beweisverpflichtungen sichert uns die Gültigkeit der semantischen Bedingungen VB1–VB3. Bei der Generierung der Beweisverpflichtungen nutzen wir die patternfreie Fassung von Automatendokumenten. Werden Patterns benutzt, so kann oft bereits durch Unifikation entschieden werden, daß die gewünschte Eigenschaft gilt, und es entsteht keine Beweisverpflichtung.

Die Bedingung VB1 (8.3) fordert, daß jedes Zustandsprädikat erfüllbar sein muß. Wird für jeden Zustand $s{\in}S$ eine Beweisverpflichtung

$$\exists\overline{\alpha}.\ \Lambda s$$

mit den Attributen $\overline{\alpha}{=}attr(\Lambda)$ generiert, so ist dies hinreichend für Bedingung VB1, denn für jedes System $sys{\in}\mathcal{SM}$ gilt (vgl. (8.3)):

$$sys{\models}(\exists\overline{\alpha}.\ \Lambda s) \ \Rightarrow\ \exists\beta{\in}BEL.\ (sys,\beta){\models}\Lambda s$$

Im wesentlichen werden bei der Definition der Beweisverpflichtungen folgende Umsetzungen der Bedingungen VB1–VB3 durchgeführt:

- endliche Quantoren über Komponenten des Automatendokuments, wie Automatenzustände, Initialelemente und Transitionen werden expandiert, so daß aus einer Bedingung mehrere Axiome generiert werden, und
- die semantische Quantifizierung verwendeter Belegungen der Menge *BEL* wird durch die syntaktische Quantifizierung vorkommender Variablen ersetzt.

Bedingung VB2 (8.4) fordert, daß Automatenzustände eine Partitionierung des Datenzustandsraums bilden. Für jedes Paar $s,t{\in}S$ von Automatenzuständen mit $s{\neq}t$ ist daher die Disjunktheit der zugeordneten Prädikate zu sichern:

$$\forall\overline{\alpha}.\ \neg(\Lambda s \wedge \Lambda t)$$

Es ist leicht zu verifizieren, daß die endliche Menge dieser Beweisverpflichtungen hinreichend für Bedingung VB2 ist.

Bedingung VB3 (8.5) fordert, daß die Schaltbereitschaft einer Transition nur von ihrer Vorbedingung abhängt. Folgende Menge von Beweisverpflichtungen ist für VB3 hinreichend ($T \in \delta$):

$$\forall \overline{\alpha}, \texttt{in}.\ T_{pre} \Rightarrow \exists \overline{\alpha}', \texttt{out}.\ T_{post}$$

Wird ein Automat neu erstellt, so wird die oben genannte Menge von Beweisverpflichtungen generiert, die für die Korrektheit des Automatendokuments, also für die Einhaltung der Bedingungen VB1–VB3, hinreichend ist. Zusammengefaßt ist dies die endliche Menge (äußere $\forall$-Quantoren jeweils weggelassen):

$$\begin{aligned}\Gamma_{newAt}(A) = &\ \{\ \exists \overline{\alpha}.\Lambda s \mid s \in S,\ \overline{\alpha} = attr(\Lambda)\ \} \\ &\cup \{\ \neg(\Lambda s \wedge \Lambda t) \mid s,t \in S,\ s \neq t\ \} \\ &\cup \{\ T_{pre} \Rightarrow \exists \overline{\alpha}', \texttt{out}.\ T_{post} \mid T \in \delta,\ \overline{\alpha} = attr(\Lambda)\ \}\end{aligned}$$

Für den *Stack*-Automaten aus Abschnitt 8.1.1 sind das die ab Seite 138 angegebenen acht Axiome.

8.3.2 Verfeinerung eines Automatendokuments

In diesem Abschnitt behandeln wir die Verfeinerung von Automatendokumenten innerhalb einer Klasse c. Wir fixieren den Klassennamen und die Automatenart und übertragen die aus dem Verfeinerungskalkül aus Definition 6.17 bekannten Transformationen.

Übertragung des Verfeinerungskalküls

Wir übertragen nun wesentliche Teile des Verfeinerungskalküls aus Definition 6.17 auf Automatendokumente. Im einzelnen definieren wir folgende Verfeinerungsschritte:

addS: Erweiterung der Menge der Automatenzustände

remS: Einschränkung der Menge der Automatenzustände, wenn die entfernten Zustände im Diagramm nicht erreichbar sind

refS: Verfeinerung von Automatenzuständen

addT: Hinzunahme von Transitionen des Diagramms, für Kombinationen aus Eingabe und Quellzustand, für die bisher keine Transition existiert hat (Totalisierung)

remT: Entfernung von Transitionen des Diagramms, wenn Alternativen existieren (Entfernung von Unterspezifikation)

refT: Verfeinerung einer Transition

remI: Entfernung von Initialelementen (Entfernung von Unterspezifikation)

refI: Verfeinerung von Initialelementen

Bei der Umsetzung des Verfeinerungskalküls für Automaten auf die konkrete Syntax der Automatendokumente ergeben sich einige Unterschiede. Die oben erwähnten Transformationen *remS*, *remT*, *addT*, *remS*, *addS* und *refS* entsprechen jeweils den gleichnamigen Kalkülregeln der Tabelle 6.3.

Die Regeln (ChgFI), (ChgFT) und (Cpfy) werden nicht umgesetzt, obwohl dies möglich ist. Sie behandeln unendliche Ausgaben und damit nichtterminierendes Verhalten und sollten bei der Spezifikation von Agenten nicht verwendet werden.

Die Regel (Cplt) aus Tabelle 6.4 wird als abgeleitete Taktik in Abschnitt 8.5 formuliert.

Bei den Lebenszyklen werden die beiden Transformationen *refT* und *refI* benutzt, die zusätzliche Elemente einführen, die Spezialisierungen bereits vorhandener Elemente sind. Entsprechend sind beide Transformationen semantikerhaltend, da die mit *automaton* erzeugten buchstabierenden Automaten jeweils gleich sind. Es gibt daher keine Entsprechung beider Transformationen für buchstabierende Automaten. Beide Transformationen werden im wesentlichen vorbereitend auf weitere Verfeinerungen eingesetzt. Sie verändern die Strukturierung von Lebenszyklen, die durch Gruppierung von Automatenzuständen und Transitionen entsteht.

Die Regel (ExIn) zur Schnittstellenverfeinerung aus Tabelle 6.5 wird bei der Semantikdefinition 8.5 verwendet, aber als Transformation nicht umgesetzt. Eine entsprechende Regel ist nicht notwendig, da im Gegensatz zu buchstabierenden Automaten die Mengen der Eingabe- und Ausgabenachrichten nicht in Automatendokumenten charakterisiert werden. Stattdessen werden diese bei der Umsetzung mit *automaton* durch das Argument $sys \in \mathcal{SM}$ bestimmt.

Weil die Semantikdefinition relativ zu einem gegebenen Modell (System) stattfindet, aus dem Zustands- und Nachrichtenmengen extrahiert werden, müssen Ein- und Ausgabenachrichten nicht modifiziert werden.

Wir nutzen hier das Konzept der losen Semantikdefinition, wie bei anderen Dokumentarten auch. Im Gegensatz dazu erhalten buchstabierende Automaten eine konstruktive, der Termerzeugung von Datentypen verwandte Semantik, da ihre Elemente direkt zur Konstruktion der Semantik verwendet werden.

Die Verfeinerungsschritte auf Automaten bieten eine hohe Flexibilität, ein gegebenes Automatendokument zu modifizieren. Anders als bei den eher einfach strukturierten Objektmodellen und Klassenbeschreibungen ist es nicht möglich, eine syntaktisch überprüfbare Verfeinerungsbeziehung anzugeben. Stattdessen werden bei Durchführung obiger Verfeinerungsschritte weitere Beweisverpflichtungen entstehen. Neben diesen Beweisverpflichtungen sind auch die Kontextbedingungen KB1–KB3 von Seite 146 zu erfüllen.

Transformationen mit zusätzlichen Argumenten führen zu weiteren Kontextbedingungen. Wir werden dies anhand der Transformation *addS* demonstrieren. Weil für jede Transformation dasselbe Prinzip angewendet wird, setzen wir diese Kontextbedingungen in allen nachfolgenden Transformationen als erfüllt voraus und geben nur die Beweisverpflichtungen an.

Erweiterung der Menge der Automatenzustände

Ist ein Teil des Datenzustandsraums nicht in einer durch einen Automatenzustand gebildeten Äquivalenzklasse enthalten, so können neue Automatenzustände hinzugefügt werden, die einen Teil dieser bisher nicht abgedeckten Datenzustände übernehmen. Zu jedem neuen Automatenzustand ist ein passendes Zustandsprädikat anzugeben, das die zugehörige Datenzustandsmenge beschreibt. Wir definieren die Transformation zur Erweiterung der Menge der Automatenzustände:

$$addS : (\mathcal{AUT} \rightharpoonup \langle pred \rangle) \times \mathcal{DOC}_a \rightarrow \mathcal{DOC}_a$$
$$addS\ \Lambda_2\ A \stackrel{def}{=} (c, ak, S \cup \bullet\Lambda_2,\ \Lambda + \Lambda_2,\ \delta, I),$$

wobei die Kontextbedingungen KB1–KB3 (Seite 146) im neuen Automaten erfüllt bleiben und die Abbildung Λ_2 mit Λ kompatibel ist. Darüberhinaus entstehen Beweisverpflichtungen, die zur Korrektheit des entstehenden Automatendokuments notwendig sind. Durch die Einführung neuer Automatenzustände können die beiden Bedingungen VB1 und VB2 verletzt werden. Gehen wir von der Korrektheit des zu verfeinernden Dokuments aus, so ist zu fordern, daß:

1. die Abbildung Λ_2 keinem Automatenzustand eine leere Menge von Datenzuständen zuordnet (vergleiche VB1) und
2. die Abbildung $\Lambda + \Lambda_2$ allen Automatenzuständen disjunkte Datenzustände zuordnet (vergleiche VB2).

Zur Einhaltung dieser Bedingungen ist folgende Menge von Beweisverpflichtungen hinreichend:

$$\Gamma_{addS}(\Lambda_2, A) \stackrel{def}{=} \{\ \exists \overline{\alpha}.\ \Lambda_2 s \mid s \in \bullet\Lambda_2,\ \overline{\alpha} = attr(\Lambda + \Lambda_2)\ \}$$
$$\cup\ \{\ \neg(\Lambda_2 s \wedge (\Lambda + \Lambda_2) t) \mid s \in \bullet\Lambda_2,\ t \in S \cup \bullet\Lambda_2,\ s \neq t\ \}$$

Die Bedingung VB3 gilt unverändert, da die Transitionsmenge nicht verändert wurde. Die Transformation *addS* verfeinert die im Dokument enthaltene Verhaltensbeschreibung:

Proposition 8.6 (Erweiterung der Menge der Automatenzustände)
Unter der Zusicherung, daß die mit Γ_{addS} erzeugten Beweisverpflichtungen gelten, gilt die Folgerungsbeziehung für Dokumente:

$$\forall A \in \mathcal{DOC}_a, \Lambda_2.\ addS(\Lambda_2, A),\ \Gamma_{addS}(\Lambda_2, A) \models A$$

Insbesondere sichert die Gültigkeit von $\Gamma_{addS}(\Lambda_2, A)$ die Gültigkeit der Bedingungen VB1–VB3 im Automatendokument $addS(\Lambda_2, A)$. (□, **Beweis siehe** C.39)

Der Beweis C.39 zeigt, daß die Erweiterung der Menge der Automatenzustände keine wirkliche Verfeinerung der Verhaltensbeschreibung darstellt. Die Umsetzung mit *automaton* erzeugt für beide Dokumente denselben buchstabierenden Automaten. Nur bestimmte Datenzustände, die vorher anonym waren, gehören jetzt zu einem explizit benannten, neu eingeführten Automatenzustand.

Die Umsetzung aller weiteren Transformationsregeln erfolgt ähnlich und wird daher nur knapp behandelt. Dabei wird für jede Transformation t ein Generator Γ_t für Beweisverpflichtungen definiert.

Einschränkung der Menge der Automatenzustände

So, wie es möglich ist, neue Automatenzustände hinzuzufügen, können Automatenzustände auch entfernt werden. Dies ist möglich, wenn die zu entfernenden Automatenzustände im Transitionsdiagramm nicht erreichbar sind. Dadurch werden bisher benannte Datenzustände wieder zu anonymen Datenzuständen:

$$remS : \mathbb{P}^{fin}(\mathcal{AUT}) \times \mathcal{DOC}_a \to \mathcal{DOC}_a$$
$$remS\ S_2\ A \stackrel{def}{=} (c,ak,S\backslash S_2,\Lambda|_{S\backslash S_2},\delta,I)$$

Kommen in der Initialmenge die entfernten Automatenzustände nicht vor, und führen Transitionen nicht aus der Menge $(S\backslash S_2)$ hinaus:

$$I \subseteq (S\backslash S_2) \times \langle pred\rangle$$
$$\forall s{\in}S\backslash S_2,\ t{\in}S.\ \delta(s,*,t,*) \Rightarrow t{\in}S\backslash S_2,$$

dann ist obige Transformation ein Verfeinerungsschritt. Es gilt: $remS(S_2,A) \models A$. Analog zur Erweiterung der Menge der Automatenzustände ist auch deren Einschränkung keine echte Verfeinerung. Beweisverpflichtungen entstehen hier keine: $\Gamma_{remS} = \emptyset$.

Hinzunahme von Transitionen des Diagramms

Ist die Transitionsrelation eines Automatendokuments partiell, weil es für eine Kombination aus Quellzustand und Eingabenachricht keine Transition gibt, die schalten kann, so wird damit Chaos modelliert. Für eine robuste Verfeinerung von Chaos können Transitionen in das Diagramm aufgenommen werden, die ein Verhalten für unterspezifizierte Paare aus Quellzustand und Eingabenachricht definieren:

$$addT : \mathbb{P}^{fin}(\Delta) \times \mathcal{DOC}_a \to \mathcal{DOC}_a$$
$$addT\ \delta_2\ A \stackrel{def}{=} (c,ak,S,\Lambda,\delta\cup\delta_2,I)$$

Diese Transformation beruht auf der Regel (AddT) des Verfeinerungskalküls. Deshalb müssen die Schaltbereiche der neuen Transitionen disjunkt zu den Schaltbereichen bereits existierender Transitionen sein. Es entstehen folgende Beweisverpflichtungen (siehe auch VB3):

$$\Gamma_{addT}(\delta_2,A) = \{\ T_{pre} \Rightarrow \exists\overline{\alpha}`,\mathtt{out}.\ T_{post} \mid T{\in}\delta_2\ \}$$
$$\cup \{\ \neg(T_{1pre} \wedge T_{2pre}) \mid T_1{\in}\delta,\ T_2{\in}\delta_2\ \}$$

Wieder kann leicht festgestellt werden, daß unter Einhaltung der Beweisverpflichtungen $addT(\delta_2,A) \models A$ gilt. Die Hinzunahme von Transitionen ist im allgemeinen eine echte Verfeinerung.

Die neuen Transitionen müssen einen disjunkten Schaltbereich gegenüber alten Transitionen haben. Werden Patterns in der Eingabe angegeben, wie dies nach Definition 8.1 erlaubt ist, so kann diese Disjunktheit teilweise automatisiert überprüft werden. Es empfiehlt sich gerade hier die Verwendung von Patterns, um die mit der Anzahl der Transitionen quadratisch wachsende Menge von Beweisverpflichtungen zu reduzieren.

Entfernung von Transitionen des Diagramms

Stehen für die Behandlung einer Eingabenachricht in einem gegebenen Datenzustand mehrere Transitionen zur Verfügung, so kann eine derartige Unterspezifikation durch Entfernen einer Transition vermindert werden. Eine Transition des Automatendokuments kann also entfernt werden, wenn ihr Schaltbereich durch den Schaltbereich anderer Transitionen überdeckt ist. Wir definieren einen Schritt, der eine Transition entfernt. Die Entfernung mehrerer Transitionen kann dann iterativ erfolgen:

$$remT : \Delta \times \mathcal{DOC}_a \rightarrow \mathcal{DOC}_a$$
$$remT\ T\ A \stackrel{def}{=} (c,ak,S,\Lambda,\delta\backslash\{T\},I)$$

Der Schaltbereich der entfernten Transition ist von den Schaltbereichen anderer Transitionen zu überdecken. Es entsteht daher folgende Beweisverpflichtung:

$$\Gamma_{remT}(T,A) = \{\ T_{pre} \Rightarrow \bigvee_{D\in\delta, D\neq T} D_{pre}\ \}$$

Aus der Gültigkeit der Beweisverpflichtungen folgt unter Benutzung der Kalkülregel (RemT), daß $remT(T,A) \models A$ gilt.

Verfeinerung von Transitionen

Ein weiterer elementarer Schritt zur Modifikation eines Automatendokuments ist die Verfeinerung von Transitionen. Dabei wird eine neue Transition hinzugefügt, die eine Spezialisierung einer bereits vorhandenen Transition ist:

$$refT : \Delta \times \mathcal{DOC}_a \rightarrow \mathcal{DOC}_a$$
$$refT\ T\ A \stackrel{def}{=} (c,ak,S,\Lambda,\delta\cup\{T\},I)$$

T muß die Enabledness (Bedingung VB3) einhalten und darüberhinaus eine Verfeinerung einer bereits vorhandenen Transition sein. Das heißt, für jede Belegung von Quellzustand und Eingabe, für die T schalten kann, gibt es eine Transition T_1, die ebenfalls schalten kann, und der zugehörige Zielbereich von T ist eine Spezialisierung des Zielbereichs von T_1. Es entstehen daher folgende Beweisverpflichtungen:

$$\Gamma_{refT}(T,A) = \{\ T_{pre} \Rightarrow \exists \overline{\alpha}\text{`},\texttt{in}.\ T_{post}\ \ ;\ \ T_{pre} \Rightarrow \bigvee_{D\in\delta} D_{pre}\ \ ;\ \ T \Rightarrow \bigvee_{D\in\delta} D\ \}$$

Aus der Gültigkeit der Beweisverpflichtungen folgt $refT(T,A) \models A$. Die beiden zunächst sehr großen Formeln, die als Disjunktion über alle Transitionen des Automatendokuments entstehen, können rasch reduziert werden, da nur Transitionen, die denselben Quell- und Zielzustand wie T haben, in Frage kommen können.

Die Hinzunahme einer verfeinerten Transition ist eine semantikerhaltende Transformation, denn die Hinzunahme einer neuen Transition im Automatendokument, die bereits durch andere Transitionen abgedeckt wird, verändert kein Verhalten. Interessant wird dieser Schritt vor allem in Kombination mit der Entfernung der verfeinerten Transitionen. Das kann z.B. genutzt werden, um Bedingungen einer Transition neu zu formulieren, um sie in eine ausführbare Fassung zu bringen. Genauso können Nachbedingungen oder Ausgabeausdrücke spezialisiert werden. Durch wiederholte Einführung solcher Transitionen, die jeweils einen Teil des Schaltbereichs einer Transition abdecken, kann eine Transition durch mehrere Transitionen verfeinert werden. Aber auch die Zusammenlegung von Transitionen mit demselben Quell- und Zielzustand ist möglich.

Einschränkung der Menge der Initialelemente

Besitzt ein Automatendokument mehrere Initialelemente, so können entsprechend der Regel (RemI) Initialelemente entfernt werden, um damit die initiale Ausgabe und den Initialzustand präziser festzulegen.

$$remI : \mathbb{P}^{fin}(\langle pred \rangle) \times \mathcal{DOC}_a \to \mathcal{DOC}_a$$
$$remI\ I_2\ A \stackrel{def}{=} (c, ak, S, \Lambda, \delta, I \backslash I_2)$$

Es folgt sofort $remI(I_2,A) \models A$. Es sollte jedoch darauf geachtet werden, daß mindestens ein erfüllbares Initialelement im Automatendokument erhalten bleibt, da das entstehende Dokument sonst eine leere Menge von Verhalten beschreibt. Beweisverpflichtungen entstehen hier keine: $\Gamma_{remI} = \emptyset$.

Verfeinerung von Initialelementen

Analog zur Verfeinerung von Transitionen können auch Initialelemente verfeinert werden. Dabei wird ein neues Initialelement hinzugefügt, das eine Spezialisierung bereits vorhandener Initialelemente ist.

$$refI : \langle pred \rangle \times \mathcal{DOC}_a \to \mathcal{DOC}_a$$
$$refI\ E\ A \stackrel{def}{=} (c, ak, S, \Lambda, \delta, I \cup \{E\})$$

Das neue Initialelement ist eine Spezialisierung:

$$\Gamma_{refI}(E,A) = \{\ E \Rightarrow \bigvee_{E_1 \in I} E_1\ \}$$

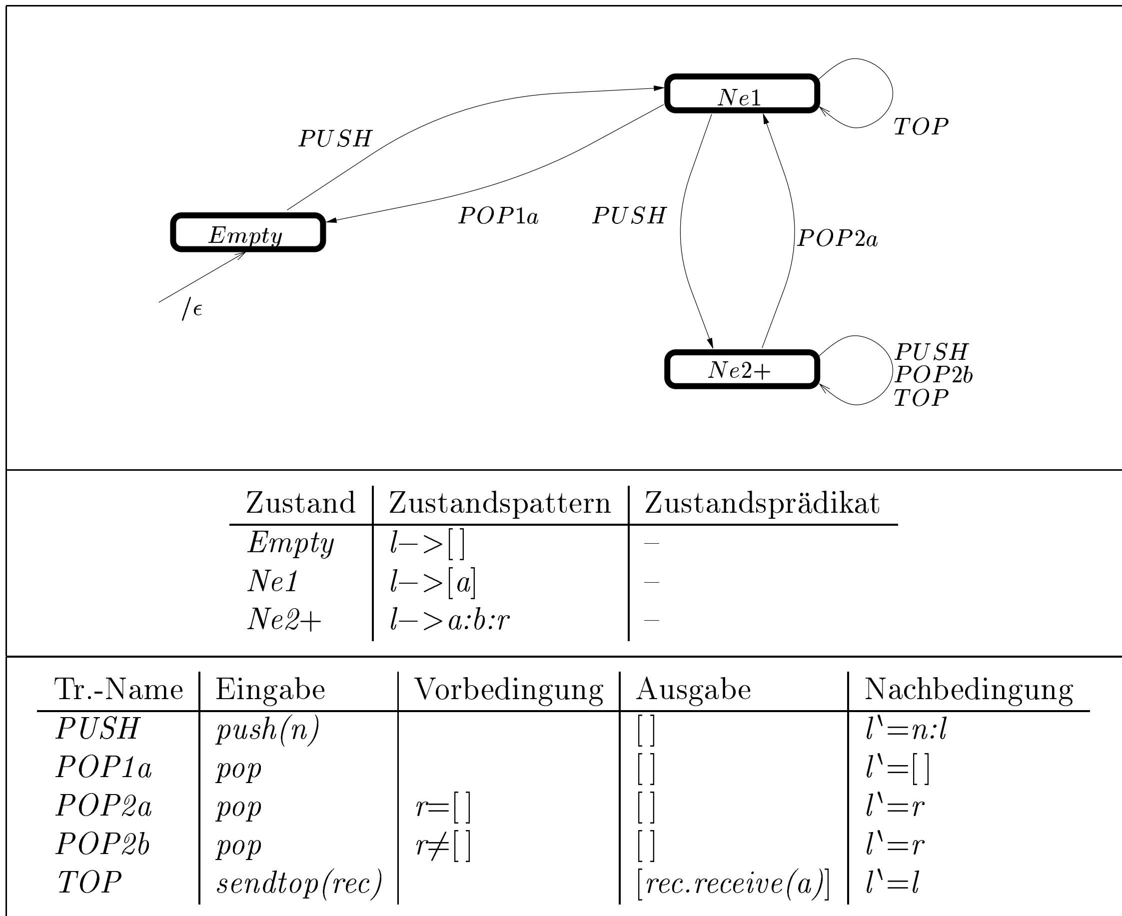
Verfeinerung von Automatenzuständen

Um eine Verhaltensbeschreibung zu präzisieren, ist es oft sinnvoll, einen Automatenzustand zu teilen und das Verhalten für jeden neuen Automatenzustand individuell zu spezialisieren. Die jeweils zugeordneten Datenzustandsräume partitionieren den Datenzustandsraum des alten Automatenzustands. Bei der Teilung von Automatenzuständen ist es notwendig, beteiligte Transitionen und Initialelemente zu modifizieren. Dadurch wird diese Verfeinerung komplex.

In Abbildung 8.4 ist das Ergebnis einer Zustandsverfeinerung des Automatenzustands *Nonempty* in zwei Automatenzustände *Ne1* und *Ne2+*, die *Stack*-Zustände mit einem bzw. mehr als einem Element repräsentieren, dargestellt. Bei der Darstellung wurden die modifizierten Vorbedingungen bereits wieder vereinfacht. Die betroffenen Transitionen werden, ähnlich wie bei der Regel (RefS) im Verfeinerungskalkül buchstabierender Automaten, aufgespalten und dabei vervielfältigt.

Neben der Erfüllbarkeit (VB1) der neuen Zustände entstehen folgende Beweisverpflichtungen, die sichern, daß die beiden neuen Zustände den alten Zustand *Nonempty* tatsächlich partitionieren und damit auch VB2 sichern:

$$\neg((\exists a.\ l{=}[a]) \wedge (\exists a,b,r.\ l{=}a{:}b{:}r))$$
$$(\exists a,r.\ l{=}a{:}r) \Leftrightarrow (\exists a.\ l{=}[a]) \vee (\exists a,b,r.\ l{=}a{:}b{:}r)$$



Zustand	Zustandspattern	Zustandsprädikat
Empty	*l−>[]*	–
Ne1	*l−>[a]*	–
Ne2+	*l−>a:b:r*	–

Tr.-Name	Eingabe	Vorbedingung	Ausgabe	Nachbedingung
PUSH	*push(n)*		[]	*l'=n:l*
POP1a	*pop*		[]	*l'=[]*
POP2a	*pop*	*r=[]*	[]	*l'=r*
POP2b	*pop*	*r≠[]*	[]	*l'=r*
TOP	*sendtop(rec)*		*[rec.receive(a)]*	*l'=l*

Abbildung 8.4: Verfeinerung des *Stack*-Automaten mit Patterns

Wir definieren folgende Verfeinerungstransformation, basierend auf der Abstraktionsfunktion γ, die jedem neuen Zustand einen alten Zustand zuordnet.

Definition 8.7 (Verfeinerung der Menge der Automatenzustände)
Die Transformation[2]

$$refS : (\mathcal{AUT} \rightharpoonup \mathcal{AUT}) \times (\mathcal{AUT} \rightharpoonup \langle pred \rangle) \times \mathcal{DOC}_a \to \mathcal{DOC}_a$$

$$refS\ \gamma\ \Lambda_2\ A \overset{def}{=} (c,ak,\bullet\Lambda_2,\Lambda_2,\delta_2,I_2)$$

$$\texttt{where } I_2 \overset{def}{=} \{(t,c) \mid (\gamma.t,c) \in I\}$$

$$\delta_2 \overset{def}{=} \{\ (s,ci_2,t,co) \mid \exists ci.\ \delta(\gamma.s,ci,\gamma.t,co) \wedge ci_2 \widehat{=} (ci \wedge \exists \overline{\alpha}', \texttt{out}.co \wedge (\Lambda_2 t)')\ \}$$

verfeinert die Menge der Automatenzustände eines Automaten, wenn die Abstraktionsfunktion γ surjektiv auf der Menge der neuen Zustände ist ($\gamma(\bullet\Lambda_2)=\bullet\Lambda$) und folgende Beweisverpflichtungen gelten:

$$\Gamma_{refS}(\gamma,\Lambda_2,A) = \{\ \exists\overline{\alpha}.\Lambda_2 s \mid s \in \bullet\Lambda_2\ \}$$
$$\cup\ \{\ \neg(\Lambda_2 s \wedge \Lambda_2 t) \mid s,t \in \bullet\Lambda_2,\ s \neq t\ \}$$
$$\cup\ \{\ \Lambda s \Leftrightarrow \bigvee_{t \in \gamma^{-1}(s)} \Lambda_2 t \mid s \in S\ \} \qquad (\square)$$

[2] $\widehat{=}$ beschreibt die syntaktische Gleichheit zweier Formeln.

Die Beweisverpflichtungen sichern zum einen VB1 und VB2, aber auch, daß die Transformation tatsächlich eine Verfeinerung des Zustandsraums darstellt. Die Enabledness-Bedingung von Transitionen (VB3) wird dadurch gesichert, daß die Vorbedingungsprädikate *ci* der betroffenen Transitionen mit $\exists \overline{\alpha}^{\backprime}, \mathtt{out}.co \wedge (\Lambda_2 t)^{\backprime}$ eingeschränkt werden. Dadurch können manche Vorbedingungen zu *False* reduziert und die zugehörigen Transitionen entfernt werden.

Proposition 8.8 (Verfeinerung der Menge der Automatenzustände)
Die Transformation refS ist semantikerhaltend. Unter der Zusicherung, daß die angegebenen Eigenschaften gelten, gilt die Folgerungsbeziehung für Dokumente:

$$\forall A \in \mathcal{DOC}_a, \gamma, \Lambda_2.\ \Gamma_{refS}(\gamma, \Lambda_2, A) \Rightarrow (refS(\gamma, \Lambda_2, A) \models\!\!\dashv A)$$

(□, Beweis siehe C.40**)**

Wird diese Transformation angewandt, um einen einzelnen Zustand durch neue Zustände zu verfeinern, dann sind nur die mit dem neuen Zustand verbundenen Beweisverpflichtungen relevant; alle anderen sind bereits gezeigt worden. Auch sind nur die Transitionen zu modifizieren, deren Zielzustand verfeinert wird.

8.3.3 Vererbung eines Automatendokuments

Neben der Verfeinerung innerhalb einer Klasse ist es möglich, einen Lebenszyklus auf eine kleinere Menge von Agenten zu spezialisieren. Dazu gibt es zwei Transformationen:

type2class: spezialisiert einen Typautomaten zu einem Klassenautomaten und

inherit: vererbt einen Typautomaten an eine Subklasse.

Beide Operationen spezialisieren einen Lebenszyklus auf eine Teilmenge von Agenten. Dort kann der Lebenszyklus dann entsprechend verfeinert werden.

Typautomaten und Klassenautomaten

Die ersten zwei Komponenten eines Automatendokuments, der Klassenname *c* und die Automatenart *ak*, charakterisieren, welche Agenten einer Verhaltensspezifikation durch das Automatendokument unterliegen. Ist *ak*=`type`, so haben alle Agenten, die Elemente der Klasse *c* sind, die Verhaltensbeschreibung zu erfüllen. Dazu zählen auch die Agenten der Subklassen von *c*. Ist *ak*=`class`, so gilt die Verhaltensbeschreibung nur für die Instanzen der Klasse *c* selbst. Beide Automatenarten haben in der methodischen Verwendung sehr unterschiedliche Aufgaben.

Der *Typautomat* wird im Entwurf eingesetzt um bestimmte Verhaltensmerkmale zu charakterisieren, die alle Agenten eines Typs haben. Er bildet damit eine Schnittstellenbeschreibung, auf die sich Klienten verlassen können. Typautomaten sind im allgemeinen starke Abstraktionen.

Der *Klassenautomat* wird in der Implementierung eingesetzt, um sehr spezifisch das Verhalten von Agenten einer Klasse zu entwickeln. Das Ziel des Klassenautomaten ist eine ausführbare Beschreibung, die zum Prototyping benutzt werden kann oder eine fast ausführbare, detaillierte Verhaltensbeschreibung, die von Hand einfach in Code einer adäquaten Programmiersprache umgesetzt werden kann.

In der Softwareentwicklung sind wir daran interessiert, einen Typautomaten der Klasse c zu einem Klassenautomaten der Klasse c zu spezialisieren, um ihn dann für die Implementierung zu präzisieren. Darüberhinaus wollen wir einen Typautomaten zu Typautomaten von Subklassen vererben, um in der Subklasse das Verhalten zu spezialisieren[3].

Diese Transformationen sind jedoch keine Dokumentverfeinerungen, weil sie das zur Transformation benutzte Dokument nicht implizieren. Die Transformation

$$\begin{aligned} &\mathit{type2class} : \mathcal{DOC}_a \rightarrow \mathcal{DOC}_a \\ &\mathit{type2class}\ A \stackrel{def}{=} (c, \texttt{class}, S, \Lambda, \delta, I) \end{aligned}$$

modifiziert einen Typautomaten in einen Klassenautomaten. Beweisverpflichtungen entstehen dabei nicht. Wie bereits oben erwähnt, ist diese Transformation keine Verfeinerung des Automatendokuments. Aber es gilt folgende Eigenschaft, die besagt, daß sich die Verhalten der Instanzen der beschriebenen Klasse nicht ändern:

Proposition 8.9 (Transformation vom Typautomat zum Klassenautomat)
Sei $A \in \mathcal{DOC}_a$ ein Typautomat. Dann erhält die Transformation type2class die Verhaltensbeschreibungen für alle Agenten, die Instanzen der beschriebenen Klasse c sind.

$$\begin{aligned} &\mathit{beh}_c(A) = \mathit{beh}_c(\mathit{type2class}(A)) \\ &\texttt{where}\ \mathit{beh}_c(A) = \{\ \mathit{behavior}_{sys}.a \mid sys \in [\![A]\!]^a \wedge \mathit{class}_{sys}(a) = c\ \} \end{aligned}$$

(□, **Beweis siehe** C.41)

Die Spezialisierung eines Typautomaten zum Klassenautomaten erhält die Verhaltensbeschreibung der Instanzen der beschriebenen Klasse c. Dadurch wird der Typautomat einer Klasse jedoch nicht redundant. Das kann erst geschehen, wenn der Typautomat auf alle Subklassen vererbt wurde.

Vererbung

Typautomaten beschreiben das Verhalten aller Elemente einer Klasse und ihrer Subklassen. Sie werden entlang der Subklassenhierarchie vererbt, wodurch eine Verhaltensspezialisierung entlang der Subklassenhierarchie entsteht. Diese kann konstruktiv umgesetzt werden, indem aus einem gegebenen Typautomaten zur Klasse c eine Verfeinerung gebildet und diese auf die Subklasse d eingeschränkt wird. Klienten der Klasse c können

[3] Interessanterweise sind genau die beiden gegenteiligen Transformationen Verfeinerungen von Automatendokumenten, da durch die Ausweitung der Gültigkeit einer Verhaltensbeschreibung auf weitere Agenten das neue Dokument das alte impliziert. Diese korrespondieren zu Techniken der Refaktorisierung von Klassen ([OJ93]).

sich darauf verlassen, daß jedes Element der Klasse c das im Typautomaten spezifizierte Verhalten erfüllt. Für verschiedene Subklassen kann diese Verhaltensspezifikation unterschiedlich verfeinert sein, aber die Substituierbarkeit, die Wegner ([WEG90]) für Subtyping fordert, bleibt für Verhalten gültig.

Ganz analog zur Einschränkung der Verhaltensbeschreibung vom Typautomaten zum Klassenautomaten definieren wir die Einschränkung des Verhaltens vom Typautomaten zum Typautomaten einer Subklasse. Wir behandeln zunächst den Fall, daß nur von einer Klasse geerbt wird und geben folgende Transformation an:

$$\begin{array}{l} \mathit{inherit} : CN \times CN \times \mathcal{DOC}_a \rightarrow \mathcal{DOC}_a \\ \mathit{inherit}\ c\ d\ (c,\mathtt{type},S,\Lambda,\delta,I) \stackrel{def}{=} (d,\mathtt{type},S,\Lambda,\delta,I) \end{array}$$

Beweisverpflichtungen entstehen dabei nicht, aber diese Transformation darf nur auf Typautomaten unter der Zusicherung, daß d eine Subklasse von c ist, ausgeführt werden. Dies wird mit $d \sqsubseteq c$ notiert und deren Semantik festgelegt mit:

$$[\![d \sqsubseteq c]\!]^{inh} \stackrel{def}{=} \{ sys \in \mathcal{SM} \mid c,d \in CN_{sys} \wedge d \sqsubseteq_{sys} c \}$$

Wie auch die Einschränkung zu Klassenautomaten, ist diese Transformation für $c \neq d$ keine Verfeinerung des Automatendokuments. Aber es gilt folgende Eigenschaft, die besagt, daß sich die Verhalten der Elemente der Subklasse d nicht ändern:

Proposition 8.10 (Vererbung von Typautomaten)
Sei $A=(c,\mathtt{type},S,\Lambda,\delta,I) \in \mathcal{DOC}_a$ ein Typautomat. Dann erhält die Transformation inherit die Verhaltensbeschreibungen für alle Elemente der Subklasse, auf die spezialisiert wird:

$$\begin{array}{l} \{\ behavior_{sys}.a \mid sys \in [\![A]\!]^a \wedge a\ isof_{sys}\ c\ \} = \\ \quad \{\ behavior_{sys}.a \mid sys \in [\![inherit(c,d,A)]\!]^a \wedge sys \in [\![d \sqsubseteq c]\!] \wedge a\ isof_{sys}\ d\ \} \end{array}$$

(□, **Beweis siehe** C.42)

Durch obige Transformation ist die Vererbung von Lebenszyklen von einer Klasse möglich. Die im Systemmodell erlaubte Mehrfachvererbung läßt sich durch die in Abschnitt 8.4.2 beschriebene Integration von Automatendokumenten behandeln.

In verteilten objektorientierten Programmiersprachen ist oft von der „Inheritance Anomaly" die Rede. Sie basiert auf der Beobachtung, daß Methodenimplementierungen bei der Vererbung nebenläufiger Klassen einen sehr geringen Wiederverwendungsfaktor besitzen. Dies liegt daran, daß je nach Zustand des Agenten (Objekts) bei der Kommunikation mit der Umgebung Synchronisation notwendig ist, der bei Vererbung typischerweise neu definiert werden muß (siehe [MTY93]). Diese Probleme treten bei uns aus mehreren Gründen nicht auf. Zum einen findet bei uns keine synchrone, sondern eine asynchrone Nachrichtenübermittlung statt, Synchronisationsprobleme entfallen also. Zum zweiten nutzen wir Vererbung nicht zur Wiederverwendung von Code, sondern zur Spezialisierung von Struktur und Verhalten. Diese Aspekte der Vererbung sind besonders beim Entwurf von Software wesentlich wichtiger. Ihre Beachtung würde die „Inheritance Anomaly" auf ihre tatsächliche, geringere Bedeutung reduzieren.

8.4 Automatendokumente im Dokumentgraph

Wir legen nun die Syntax und Semantik für Automatendokumente und ihre Protokolle fest. Dann integrieren wir diese Dokumentart in den Dokumentgraphen. Wir besprechen, wie mehrere Lebenszyklen derselben Klasse in einen verfeinerten Lebenszyklus integriert werden können und setzen die Entwicklungsschritte einzelner Automatendokumente in Entwicklungsschritte für Automatendokumente im Dokumentgraphen um.

8.4.1 Protokolle und die erweiterte Automatensemantik

Jede einzelne angegebene Transformation ist für sich korrekt. Aufgrund der Transitivität der Verfeinerungsrelation, die auf Mengeninklusion beruht, können Transformationen zu komplexen Folgen von Transformationen komponiert werden. Ein *Protokoll* beschreibt die Sequenz der Transformationen der Entwicklung des Automatendokuments. Das Protokoll enthält zum einen alle Beweisverpflichtungen der einzelnen Transformationen. Zum anderen kann ein Protokoll für Replays der Entwicklung von leicht geänderten Vorgängerdokumenten aus benutzt werden. Wir legen ein Protokoll als Anhang zum verfeinerten Automatendokument im Dokumentgraphen ab. Da ein Automatendokument mehrere Vorgänger haben kann, kann es mehrere Protokolle besitzen. Damit wird auch die später behandelte Mehrfachvererbung und Integration von Verhaltensbeschreibungen möglich. Wir werden im folgenden für jeden einzelnen Verfeinerungsschritt einen Protokolleintrag definieren. Eine spezielle Protokollform *newAt* ist für die Neuerstellung von Dokumenten gedacht.

Definition 8.11 (Protokolle für Automatendokumente)
Die Menge der Protokolleinträge $\mathcal{PE}$ wird funktional definiert als:

$$\begin{array}{rllll} \mathcal{PE} \stackrel{def}{=} & addS(L) & \mid remS(S) & \mid refS(\gamma,L) & \mid addT(TM) \\ & \mid remT(T) & \mid refT(T) & \mid remI(I) & \mid refI(E) \\ & \mid type2class() & \mid inherit(c,d) & & \end{array}$$

wobei $L \in \mathcal{AUT} \rightharpoonup \langle pred\rangle$; $\gamma \in \mathcal{AUT} \rightharpoonup \mathcal{AUT}$; $S \subseteq \mathcal{AUT}$; $c,d \in CN$; $TM, \{T\} \subseteq \Delta$; $I, \{E\} \subseteq \langle pred\rangle$.
Protokolle werden definiert als:

$$\mathcal{PROT} \stackrel{def}{=} newAt(F) \mid refAt(\mathcal{N}, \mathcal{PE}^*, F)$$

mit Mengen $F \subseteq \langle pred\rangle$ von Beweisverpflichtungen, die bei der Anwendung der Transformationen bzw. der Neuerstellung entstehen.
Die Menge $\mathcal{DOC}_{aut} \stackrel{def}{=} \mathcal{DOC}_a \times \mathbb{P}^{fin}(\mathcal{PROT})$ definiert die um Protokolle erweiterten Automatendokumente. (□)

Ein Automatendokument wird entweder neu erstellt und der Anhang besteht aus einem *newAt* Protokoll und den dabei entstehenden Beweisverpflichtungen. Oder es wird aus einem oder mehreren anderen Dokumenten durch Verfeinerung entwickelt und der Anhang beinhaltet entsprechende Transformationen. Ein Protokolleintrag beschreibt eine gleichnamige Transformation der Form $\mathcal{DOC}_a \to \mathcal{DOC}_a$ (siehe Signatur der Transformationen).

Entsprechend definiert eine Sequenz von Protokolleinträgen eine funktional komponierte Sequenz von Transformationen. Jeder Protokolleintrag $pe \in \mathcal{PE}$ erzeugt eine (evtl. leere) Menge von Beweisverpflichtungen Γ_{pe}. Diese werden im Argument F des Protokolls gesammelt.

Wir erweitern nun die Semantik von Automatendokumenten unter Einschluß der Protokolle.

Definition 8.12 (Semantik eines Automatendokuments)
Sei $(A,P) \in \mathcal{DOC}_{aut}$ ein Automatendokument mit einer Menge von Protokollen P. Ist subs(P) die Menge aller auftretenden Subklassenbeziehungen der Form (inherit c d) und proofobl(P) die Menge aller Beweisverpflichtungen, dann wird die Menge aller Systeme, die die Zusicherungen des Protokolls erfüllen, definiert als:

$$[\![P]\!]^{prot} \stackrel{def}{=} \{\ sys \in \mathcal{SM} \mid sys \models proofobl(P) \wedge sys \in [\![subs(P)]\!]^{inh}\ \}$$

Die Semantik des gesamten Dokuments wird definiert als:

$$[\![(A,P)]\!]^{aut} \stackrel{def}{=} [\![A]\!]^{a} \cap [\![P]\!]^{prot} \qquad (\Box)$$

Der konstruktive Anteil eines Automatendokuments $(A,P) \in \mathcal{DOC}_{aut}$ wird durch den Automaten A beschrieben. Das Protokoll P fordert Kontextbedingungen in Form von Beweisverpflichtungen und Subklassenbeziehungen, deren Gültigkeit aus anderen Dokumenten folgen muß. Dies erfordert die Verifikation der Beweisverpflichtungen. Wird ein Automatendokument aus einem anderen, bereits als korrekt erkannten Automatendokument entwickelt, so können Beweise des Ursprungsdokuments in modifizierter Form wiederverwendet werden.

Bei der konkreten Erstellung bzw. Verfeinerung eines Automatendokuments empfiehlt sich folgende Vorgehensweise: Zunächst werden alle konstruktiven Entwicklungsschritte durchgeführt. Die dabei entstehenden Beweisverpflichtungen werden gesammelt und an einen Theorembeweiser übergeben. Dann wird mit möglichst stark automatisierter Unterstützung versucht, die Korrektheit aller Beweisverpflichtungen aus anderen Dokumenten herzuleiten. Gelingt dies, so ist das entstandene Dokument korrekt und kann in den Dokumentgraphen eingefügt werden.

8.4.2 Integration von Automatendokumenten

Durch die parallele Entwicklung von mehreren Lebenszyklen für dieselbe Klasse, aber auch durch Weiterentwicklung von Lebenszyklen einer Superklasse oder durch Mehrfachvererbung kann es passieren, daß mehrere Automatendokumente unterschiedliche Lebenszyklen derselben Klasse beschreiben. Diese müssen im Verlauf einer Entwicklung zu einem Dokument integriert werden. Im Gegensatz zu den in Kapitel 7 behandelten Dokumentarten ist jedoch die Integration von Lebenszyklen nicht automatisiert möglich.

Ein Produktautomat von Lebenszyklen ist nicht hilfreich, da dadurch ein Kreuzprodukt der Zustandsmengen entsteht, das nicht mehr mit der durch die Attribute festgelegten Zustandsmenge konform wäre. Eine nachträgliche Vergröberung des Produkt-Zustandsraums zu dem gewünschten Zustandsraum ist kaum möglich. Bisimulationstechniken ([MIL90]) können ebenfalls nicht angewendet werden, da diese ausgelegt sind, um

die Äquivalenz zweier Transitionssysteme zu überprüfen und nicht, um eine gemeinsame Verfeinerung derselben zu finden.

Wir wählen daher einen konstruktiven Weg zur Integration mehrerer Lebenszyklen. Ist die Integration mehrerer Automatendokumente in ein neues notwendig, so ist für jedes Dokument eine Reihe von Verfeinerungstransformationen anzugeben, die jeweils das gleiche Resultat erzeugen. Dieser resultierende Lebenszyklus ist eine Verfeinerung der ursprünglichen Lebenszyklen und stellt somit deren Integration dar. Typischerweise werden bei der Integration vom Entwickler neue Entwurfsideen gefunden und können hier auch sofort eingearbeitet werden. Daher ist es erlaubt, daß diese Integration nicht semantikerhaltend, sondern semantikverfeinernd ist.

Enthalten Lebenszyklen widersprüchliche Angaben über das Verhalten eines Automaten, so sind diese nicht konsistent integrierbar. Es ist im allgemeinen nicht möglich, solche Widersprüche automatisiert zu erkennen. Das bedeutet, es liegt im Verantwortungsbereich des Entwicklers, zu erkennen, ob eine Integration möglich ist und wie diese aussehen sollte. Natürlich können Werkzeuge Unterstützung bieten, insbesondere wenn Patterns eingesetzt werden.

Für Standardfälle lassen sich einfache und hilfreiche Taktiken angeben. Werden etwa mehrere Typautomaten von unterschiedlichen Superklassen geerbt und haben diese disjunkte Attribute, so kann ein der Konstruktion des Produktautomaten ähnlicher Mechanismus verwendet werden. Wird zunächst ein Typautomat auf eine Subklasse oder zu einem Klassenautomaten spezialisiert, dann aber verfeinert, so kann ein Replay-Mechanismus entsprechende Verfeinerungen auf den spezialisierten Lebenszyklen nachbilden.

8.4.3 Entwicklungsschritte und Redundanztest

Die als Anhang der Automatendokumente definierten Protokolle legen bereits sämtliche möglichen Entwicklungsschritte nahe. Wir definieren daher nur eine Funktion, die ein neues Automatendokument in den Dokumentgraphen einfügt und dabei aus den angegebenen Protokollen Information über Vorgängerdokumente und Kontextbedingungen extrahiert.

Definition 8.13 (Entwicklungsschritt für Automatendokumente)
Wir definieren folgenden Entwicklungsschritt AddAutomaton für Automatendokumente:

$$AddAutomaton : \mathcal{DG} \times \mathcal{DOC}_{aut} \times \mathcal{N} \times \mathbb{P}^{fin}(\mathcal{N}) \rightharpoonup \mathcal{DG}$$
$$AddAutomaton((N,E,D,R),(A,PS),n,P_1) \stackrel{def}{=} \mathcal{DS}((N,E,D,R),(A,PS),n,P_1 \cup P_2,Q)$$
$$\texttt{where } P_2 \stackrel{def}{=} \{dn \in \mathcal{N} \mid refAt(dn,*) \in PS\}$$
$$Q \stackrel{def}{=} \{dn \in \mathcal{N} \mid refAt(dn,(pe_i)_i) \in PS \wedge \forall i. pe_i \notin \{type2class, inherit *\}\}$$

Wir erhalten folgende Kontextbedingungen:

1. *Wie bereits in Definition 2.8 angegeben, muß* $P_1 \cup P_2 \subseteq N$ *und* $n \notin N$ *gelten.*
2. *Alle angegebenen Protokolle stellen Transformationen des jeweiligen Vorgängerdokuments in den Automaten A dar:*

$$\forall p \in PS.\ p = refAt(dn,tl) \Rightarrow apply(tl,(D.dn)_{cons}) = A$$

3. *Der Protokollteil PS muß aus* $D(P_1)$ *folgen:*

$$D(P_1)_{cons} \models PS \qquad (\square)$$

Weil ein Protokoll zum Teil aus nicht automatisiert überprüfbaren Beweisverpflichtungen besteht, ist die Korrektheit der Aussage $D(P_1)_{cons} \models PS$ im allgemeinen durch einen Theorembeweiser zu zeigen.

Satz 8.14 (Entwicklungsschritt für Automatendokumente)
Der Entwicklungsschritt AddAutomaton 8.13 ist korrekt. Wenn die Kontextbedingungen der Argumente des Entwicklungsschritts erfüllt sind, dann entsteht eine korrekte Erweiterung des Dokumentgraphen.

(□, **Beweis siehe** C.43)

Ein Redundanztest ist für Automatendokumente ebenfalls nicht automatisiert durchführbar. Deshalb wird auch dafür ein konstruktiver Ansatz gewählt. Ein Automatendokument kann aus zwei Gründen redundant werden:

1. Es wird eine verfeinerte Version für dieselbe Menge von Agenten angegeben, oder
2. es handelt sich um einen Typlebenszyklus, der zu einem Klassenlebenszyklus derselben Klasse und zu Typlebenszyklen für alle direkten Subklassen spezialisiert wurde.

Im ersten Fall wird der entsprechende Automat bereits durch *AddAutomaton* redundant markiert. Im zweiten Fall müßte die vollständige Menge aller Subklassen einer Klasse bekannt sein. Da aufgrund der losen Semantikdefinitionen keine Dokumentart in der Lage ist, die Menge der Subklassen einer Klasse einzuschränken, kann damit ein Typlebenszyklus nur durch einen anderen Typlebenszyklus derselben Klasse redundant gemacht werden. Es ist deshalb kein eigener Redundanztest für Automatendokumente notwendig.

8.4.4 Vollständigkeit eines Dokumentgraphen

Während einer Softwareentwicklung werden iterativ neue Dokumente in den Dokumentgraphen eingefügt. Dabei werden neue Informationen hinzugefügt, bis eine gewisse Sättigung stattgefunden hat. Je nach Phase in der sich die Entwicklung befindet, ist eine Sättigung nach unterschiedlichen Kriterien erreicht.

Mit den in dieser Arbeit definierten Beschreibungstechniken können wir die Entwurfsphase als beendet betrachten, wenn für jede in irgendeinem Dokument auftretende Klasse

- eine vollständige Klassenbeschreibung existiert,
- ein Typlebenszyklus und
- ein Klassenlebenszyklus angegeben ist.

Darüberhinaus fordern wir die

- Existenz eines vollständigen Objektmodells.

Klarerweise müssen alle verwendeten Datentypen, Funktionen, Konstanten etc. definiert worden sein. Dies wird bereits durch die Überprüfung der syntaktischen Korrektheit (Kontextbedingungen) jedes Dokuments gesichert.

Die Vollständigkeit eines Objektmodells sichert, daß jede auftretende Klasse und Vererbungsbeziehung im Objektmodell dargestellt ist. Auch alle Datenbeziehungen, wie sie in Kapitel 7 definiert sind, werden im vollständigen Objektmodell dargestellt.

Die Vollständigkeit von Klassenbeschreibungen wird analog festgelegt. Sie sichert, daß jede Methode und jedes Attribut, das von einer Superklasse geerbt wird, in der Klassenbeschreibung genannt ist. Weil jede Information eines Objektmodells mit Ausnahme der Relationsnamen als Information entsprechender Klassenbeschreibungen dargestellt werden kann, Relationsnamen selbst aber keine Restriktionen an die Systeme darstellen (vgl. 7.8), fordern wir für die Vollständigkeit des Dokumentgraphen, daß

- alle Objektmodelle redundant sind.

Dadurch ist sämtliche zur Implementierung notwendige Information bereits in Klassenbeschreibungen vorhanden, und das vollständige Objektmodell dient nur zu Dokumentationszwecken. Sind alle Klassenbeschreibungen vollständig, so ist für jede Klasse ebenfalls nicht mehr als eine solche notwendig. Wir fordern daher, daß

- pro Klasse genau eine Klassenbeschreibung nicht redundant ist.

Diese ist natürlich vollständig. Analog fordern wir, daß

- pro Klasse genau ein Typlebenszyklus und
- genau ein Klassenlebenszyklus nicht redundant ist.

Der Klassenlebenszyklus dient als Vorschrift zu Implementierung, der Typlebenszyklus als Verhaltensbeschreibung. Wie in Abschnitt 8.4.3 erklärt, können nicht alle Typlebenszyklen redundant werden. Ist jedoch eine Entwicklung abgeschlossen, und die Anzahl der Klassen und deren Vererbungsstruktur festgelegt, so kann festgestellt werden, ob ein Typlebenszyklus zur Implementierung herangezogen werden muß. Wir fordern, daß

- jeder Typlebenszyklus eine Spezialisierung der Typlebenszyklen seiner direkten Superklassen ist und
- jeder Klassenlebenszyklus eine Spezialisierung des Typlebenszyklus derselben Klasse darstellt.

Diese Spezialisierungen sind durch entsprechende Kanten im Dokumentgraph festgehalten. Auf diese Weise entsteht zu einer gegebenen Klassenhierarchie eine gleichartige Hierarchie der Lebenszyklen, wie in Abbildung 8.5 gezeigt. Für die Implementierung reicht dann die Betrachtung von Klassenlebenszyklen aus.

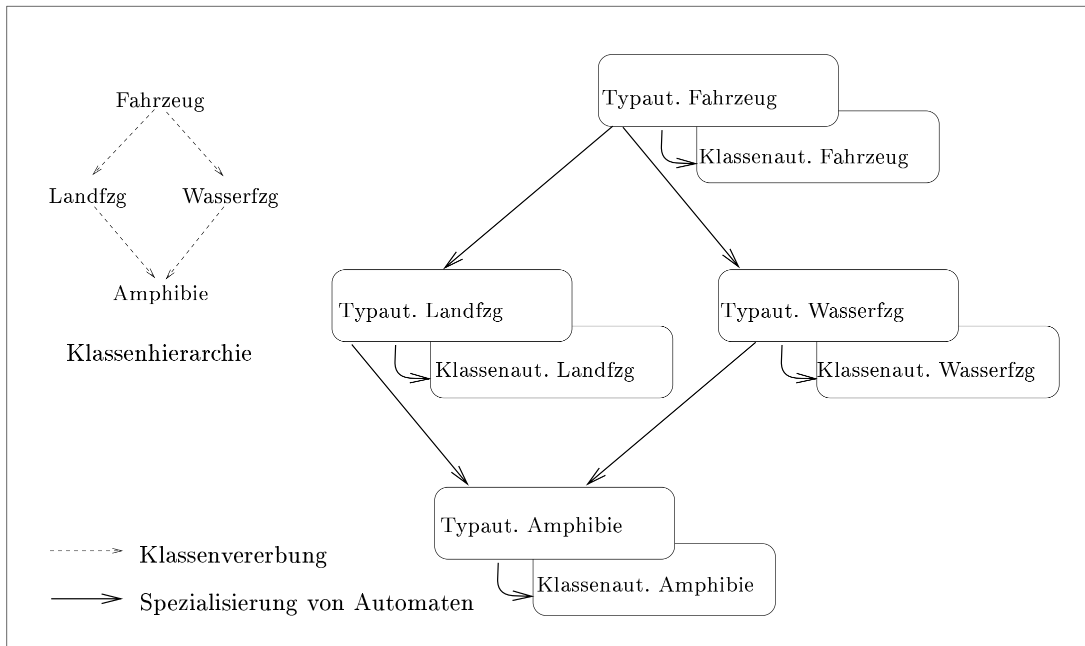


Abbildung 8.5: Spezialisierungsstruktur der Lebenszyklen für Fahrzeugarten

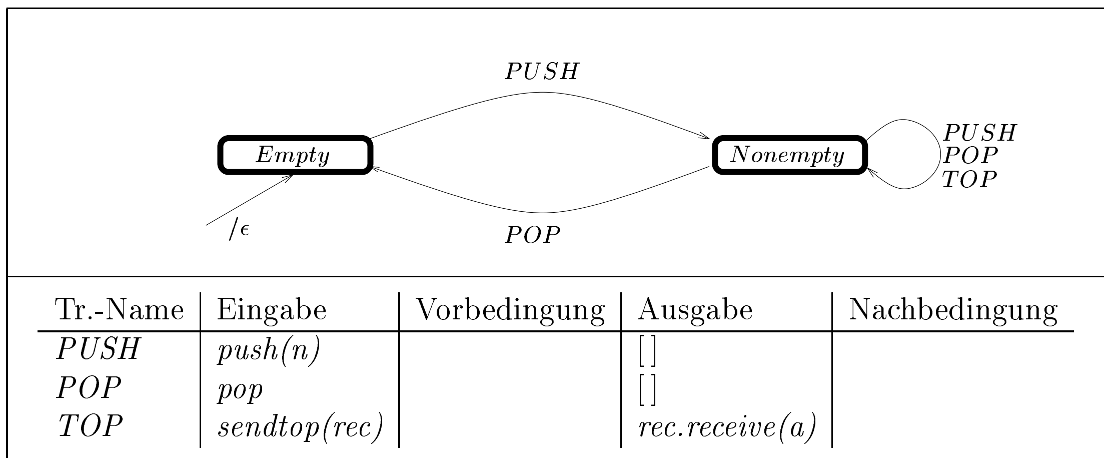
Ein bereits vollständiger Dokumentgraph kann um neue Dokumente erweitert werden, die ihn wieder unvollständig werden lassen. Im allgemeinen ist jedoch vorgesehen, daß nach Abschluß der Entwurfsphase, die in einem vollständigen Dokumentgraphen resultiert, keine weiteren Klassen oder Vererbungsbeziehungen mehr aufgenommen werden. Wir bilden damit ein *minimales Modell* in bezug auf Klassen und Vererbungsbeziehungen, das den gestellten Anforderungen genügt.

Auch bei einer nachfolgenden Implementierung, die etwa Klassendokumente in eine Programmiersprache übersetzt, werden noch Entwurfsentscheidungen getroffen. So kann es dem Entwickler freigestellt werden, bestimmte Verhaltensdetails frei festzulegen, um etwa Kriterien der Effizienz oder der Wiederverwendung bereits gegebener Komponenten zu erfüllen.

8.5 Bemerkungen

Zustandsabstraktion

Ein interessanter Aspekt der Typautomaten ist, daß diese die Verhaltensbeschreibung einer Schnittstelle für Klienten einer Klasse darstellen. Während Klienten auf die interne Realisierung einer Klasse, also auf deren Datenzustand keinen Zugriff haben, wird dieser Datenzustandsraum im Lebenszyklus durch die angegebenen Zustandsprädikate transparent. Dadurch geht die Kapselung und die damit verbundene Abstraktion verloren. Dies kann teilweise durch geschickte Definition von Lebenszyklen verhindert werden, indem etwa keine durch den Datenzustand belegte Variablen in der Vorbedingung, der Nachbedingung oder dem Ausgabeausdruck verwendet wird. Die Zustandsprädikate für jeden Automatenzustand müssen jedoch in der hier formalisierten Syntax angegeben werden.



Tr.-Name	Eingabe	Vorbedingung	Ausgabe	Nachbedingung
PUSH	*push(n)*		[ ]	
POP	*pop*		[ ]	
TOP	*sendtop(rec)*		*rec.receive(a)*	

Abbildung 8.6: Abstrakter *Stack*-Lebenszyklus

Aus einem gegebenen Lebenszyklus kann durch Existenzquantifizierung aller Zustandsvariablen ein abstrakter Lebenszyklus gebildet werden. Diese Abstraktionsleistung ist jedoch im allgemeinen mit der Einführung zusätzlicher Unterspezifikation verbunden, die häufig nicht gewollt ist. So kann der Lebenszyklus *Stack* aus Abbildung 8.1 zu *AbstractStack* aus Abbildung 8.6 abstrahiert werden. Dabei werden die Vorbedingungen der *pop*-Transitionen durch $\exists r.r{=}[\,]$ bzw. $\exists r.r{\neq}[\,]$, also *True* ersetzt. Ebenso fallen alle Nachbedingungen weg und die Ausgabevariable a der Transition *TOP* ist völlig unterspezifiziert. Das in Abbildung 8.6 definierte Verhalten wird auch von anderen Container-Klassen wie Mengen oder Sequenzen erfüllt. Auch beschränkte Container wie etwa ein einelementiger Puffer oder eine Speicherzelle erfüllen das angegebene Verhalten.

Wie das Beispiel zeigt, ist es in manchen Fällen nicht vorteilhaft, einen abstrakten, zustandsfreien Lebenszyklus als Verhaltensbeschreibung zu definieren. Stattdessen ist es interessant, eine Teilabstraktion durchzuführen, indem etwa eine abstrakte Zustandsmenge verwendet wird, die sich aus wenigen Variablen zusammensetzt, so daß die Kapselung der Schnittstelle für Klienten nur partiell aufgegeben werden muß. Die in dieser

Arbeit definierte objektorientierte Modellierungstechnik bietet genau diesen Mechanismus an, indem sie abstrakte Klassen zuläßt, die eine gewisse Menge an Attributen und Methoden besitzen und deren Verhaltensbeschreibung auf diese Attribute und Methoden eingeschränkt ist. Jedoch sind diese Zustandsdefinitionen nicht völlig abstrakt. In jeder Implementierung von Subklassen werden alle Attribute entsprechend dieser Verhaltensbeschreibung verändert. Dies erlaubt keinen Wechsel der Datenstruktur, z.B. von Kartesischen Koordinaten in Polarkoordinaten. Das kann durch eine Umstrukturierung der Klassenhierarchie (Refaktorisierung, siehe [OJ93]) vorgenommen werden.

Animierbare Lebenszyklen

Im Hinblick auf eine Implementierung, insbesondere auf einen animierten Prototyp ist es interessant, ausführbare Lebenszyklen zu verwenden. Jedoch nur ein Teil der Automatendokumente erfüllt diese Forderung. Folgende Kriterien reichen aus, um ein Automatendokument in eine *Gofer*-Funktion umzusetzen, die eines der beschriebenen Verhalten realisiert:

- Es gibt nur Eingabepatterns, oder alle Vorbedingungen sind ausführbar.
- Nachbedingungen sind Konjunktionen aus *var*=*expr* mit Variablen *var* und Wertausdrücken *expr*.

Werden nicht alle Variablen des Zielzustands oder des Ausgabeausdrucks belegt, so müssen passende Ersatzwerte eingesetzt werden. Eine Umsetzung in eine *Gofer*-Funktion, die ein spezielles Verhalten des Automaten simuliert, ist dann einfach.

Komplexere Transformationen als Taktiken

Aufgrund der vielen Komponenten, die in einem Automatendokument verwendet werden, gibt es eine ganze Reihe weiterer Transformationen, die auf Automatendokumenten durchgeführt werden können. Nachfolgend werden beispielhaft einige davon angegeben. Diese sind nicht elementar, sondern lassen sich durch bereits behandelte Transformationen auf Automatendokumenten ausdrücken. Es sind also Taktiken im Sinne der Definition 2.11, die der Erreichung bestimmter Ziele dienen. Durch die Korrektheit der einzelnen angewandten Transformationsschritte wird auch die Korrektheit der Taktiken gesichert.

Taktik 1: Einengung eines Zustandsprädikats,
um bisher im Automatenzustand enthaltene Datenzustände in die Menge der anonymen, unerreichbaren Datenzustände zu transferieren.

1. Dazu wird zunächst eine Verfeinerung des Automatenzustands A in zwei neue Automatenzustände A_1, A_2 vorgenommen (*refS*), so daß unerwünschte Datenzustände in A_2 abgespalten werden.

2. Alle nach A_2 führenden Transitionen werden entfernt (*remT*). Im allgemeinen existieren ja mit den zu A_1 führenden Transitionen Alternativen.
3. Der Automatenzustand A_2 kann entfernt werden (*remS*).

Taktik 2: Verschärfung des Zielbereichs einer Transition,
um eine Verhaltensbeschreibung deterministischer zu machen.

1. Dazu wird eine Verfeinerung der Transition hinzugefügt, die den Zielbereich einengt (*refT*).
2. Dann wird die ursprüngliche Transition entfernt (*remT*).

Taktik 3: Teilung von Transitionen,
um anschließend die einzelnen Transitionen weiter zu bearbeiten.

1. Zunächst werden alle neuen Transitionen hinzugefügt (*refT*).
2. Dann wird die ursprüngliche Transition entfernt (*remT*).

Taktik 4: Vervollständigung des Zustandsraums,
um alle Datenzustände in Automatenzuständen darzustellen.

1. Es ist zu bestimmen, welche Datenzustände in keiner durch die Automatenzustände festgelegten Äquivalenzklasse liegen. Dazu wird ein Prädikat gebildet, das die Negation aller anderen Zustandsprädikate darstellt.
2. Es ist zu überprüfen, ob dieses Prädikat identisch *False* ist. Falls ja, ist nichts weiter zu tun.
3. Sonst ist ein neuer Automatenzustand mit genau diesem Prädikat einzuführen (*addS*).

Eine Werkzeugunterstützung ist nicht nur bei der letzten Taktik sinnvoll. Eine Fülle weiterer Taktiken kann angegeben werden, um die Erreichung weiterer Ziele zu unterstützen.

Alternative Formalisierungen für Lebenszyklen

In der hier angegebenen Formalisierung von Lebenszyklen in Automatendokumenten wurden einige Entwurfsentscheidungen getroffen, deren Alternativen wir hier diskutieren wollen.

Attribute und Methodennamen im Lebenszyklus

In der hier vorgestellten Formalisierung werden im Automatendokument weder Attribute und Methodennamen noch Parameter für ankommende oder ausgehende Nachrichten angegeben, da dafür Klassenbeschreibungen vorgesehen sind. Dennoch werden diese in Prädikaten benutzt. Diese Abhängigkeit der syntaktischen Wohlgeformtheit eines Automatendokuments von anderen Dokumenten haben wir im Dokumentgraphen formalisiert. Es wäre eine Alternative, bei der Umsetzung *automaton* aus Definition 8.3 den Datenzustandsraum sowie die Ein- und die Ausgabemenge nicht aus der Semantik in Form eines Systems, sondern aus einer beigeordneten Klassenbeschreibung zu extrahieren. Die

Umsetzung *automaton* wäre dann konstruktiver, da Zustands-, Eingabe- und Ausgabemengen dabei selbst definiert würden. Es wären aber Verfeinerungstechniken notwendig, um z.B. die Ausgabemenge buchstabierender Automaten zu modifizieren.

Bedingungen VB1–VB3

Wir haben in den Bedingungen VB1–VB3 einige Eigenschaften von Automaten gefordert, die deren intuitives Verständnis erhöhen, aber zur Semantikbildung nicht notwendig sind. Bei der Anwendung von Verfeinerungstransformationen, in die Prädikate involviert sind, sind immer Beweisverpflichtungen zu zeigen, so daß die Forderung der Bedingungen VB1–VB3 nur eine graduelle Mehrbelastung an Beweisaufwand darstellt.

In [Getal96] ist eine Formalisierung angegeben, die keine Einschränkungen der Form VB1–VB3 stellt. Dort können Automatenzustände mit nicht erfüllbaren oder überlappenden Zustandsprädikaten versehen sein. Wir haben diese unintuitive Darstellung eines Verhaltens zur Motivation der Bedingungen in 8.2.4 bereits behandelt.

Die mit Bedingung VB2 geforderte Disjunktheit der Automatenzustände verhindert, daß für Agenten neben dem Datenzustandsraum eine Kontrollzustandskomponente notwendig ist. Der Kontrollzustand eines Agenten kann durch ein Attribut geeigneten Datentyps (meist eine Aufzählung) in den Datenzustand kodiert werden. Deshalb ist die Datenzustandssicht zum einen ausreichend und zum anderen auch oft intuitiver als die Kontrollzustandssicht. Darüberhinaus entspricht dies typischen objektorientierten Sprachen, bei denen die Bearbeitung einer Nachricht durch den Aufruf einer Methode realisiert wird. Diese Methode hat nur Zugriff auf den Datenzustand; es existiert kein weiterer Kontrollzustand. Gleichzeitig führt dies zu einer mehr zustandsorientierten Spezifikationstechnik.

Auch die Bedingung VB3 könnte weggelassen werden, wie das in [Getal96] der Fall ist. Dort hängt der Schaltbereich einer Transition nicht nur von der Erfüllbarkeit der Vorbedingung, des Eingabepatterns und des Quellzustands, sondern auch von der Erfüllbarkeit der Nachbedingung, des Ausgabeausdrucks und des Zielzustands ab. Dadurch erhält die Vorbedingung denselben Status wie die Nachbedingung und beide könnten zusammengelegt werden.

Eine interessante Variante wäre es, die Unerfüllbarkeit der Nachbedingung einer Transition damit gleichzusetzen, daß diese Transition nicht terminiert. Weil aber die Ausgabenachrichten dennoch produziert werden, wäre eine Teilung der Nachbedingung in bis zu drei Prädikate interessant. Ein Prädikat beschreibt das Aussehen der Ausgabe, ein weiteres Prädikat beschreibt die Terminierung einer Transition, und ein drittes Prädikat definiert den Zielzustand unter der Annahme, daß das Terminierungsprädikat erfüllt ist.

8.6 Zusammenfassung

In diesem Kapitel haben wir Lebenszyklen von Agenten in Form von Automatendokumenten formalisiert. Lebenszyklen beschreiben das Ein-/Ausgabeverhalten von Agenten und deren interne Zustandsübergänge.

Zwei Arten von Lebenszyklen, Klassenautomaten und Typautomaten, wurden eingeführt. Typautomaten charakterisieren die Lebenszyklen aller Elemente einer Klasse. Sie werden vererbt und stellen damit eine Schnittstellenbeschreibung für Agenten dar. Demgegenüber werden Klassenautomaten benutzt, um die Lebenszyklen der Instanzen einer Klasse zu charakterisieren. Klassenautomaten sind im allgemeinen wesentlich detaillierter und dienen als Beschreibung der Implementierung der Klasse.

Die Formalisierung der konkreten Syntax und der Semantik von Automatendokumenten wurde, genau wie bei allen anderen Dokumentarten, relativ zum Systemmodell durchgeführt, wobei eine Zwischenübersetzung in buchstabierende Automaten vorgenommen wurde.

Zur endlichen und übersichtlichen Darstellung unendlicher Mengen von Zuständen und Transitionen wurden prädikatenlogische Ausdrücke eingesetzt. Aufgrund der Benutzung prädikatenlogischer Ausdrücke, um Zustandsmengen, Vor- und Nachbedingungen von Transitionen präzise zu charakterisieren, sind gewisse Kontextbedingungen nicht mehr automatisiert testbar. Es entstehen Beweisverpflichtungen. Deshalb haben wir Protokolle, die auch die Entwicklungsgeschichte eines Automatendokuments enthalten, neben dem konstruktiven Anteil in die Automatendokumente aufgenommen.

Dann wurden Bearbeitungsschritte für Automatendokumente definiert, indem der Verfeinerungskalkül für buchstabierende Automaten umgesetzt wird. Es ist möglich:

- neue Automatendokumente zu erstellen,
- Automatenzustände zu erweitern, zu entfernen oder zu verfeinern,
- Transitionen hinzuzufügen, wo bisher Partialität herrschte, zu entfernen, um Unterspezifikation zu reduzieren und zu verfeinern sowie
- Initialelemente zu entfernen und zu verfeinern.

Neben diesen Schritten, die die Art des Automaten und die davon betroffenen Agenten unverändert lassen, gibt es Spezialisierungen, um

- Typautomaten zu Klassenautomaten zu modifizieren und
- Typautomaten zu vererben.

Da die Vererbung von Verhalten ebenfalls über die oben gegebenen Verfeinerungsschritte beschrieben wird, ist Vererbung von Automaten als Verhaltensbeschreibung gut in den Entwicklungsvorgang integriert. Insbesondere wird damit die Substituierbarkeit von Agenten aus Subklassen nicht nur unter Erhaltung der Signatur eines Agenten, sondern auch dessen Verhalten möglich.

In diesem Kapitel wurde weiterhin untersucht, wie Automatendokumente im Dokumentgraphen verwendet werden. Es wurden die

- Integration mehrerer Automatendokumente,
- Entwicklungsschritte und
- Redundanztest für Automatendokumente

beschrieben. Darüberhinaus wurde ein Kriterium angegeben, das die Vollständigkeit des gesamten Dokumentgraphen behandelt. Diese Vollständigkeit spiegelt das Ende der Entwurfsphase wider.

Teil IV

Epilog

Der letzte Teil der Arbeit enthält eine Zusammenfassung der Ergebnisse und einen Ausblick über mögliche zukünftige Arbeiten. Ein Literaturverzeichnis gibt Hinweise auf weiterführende Texte.
Dieser Teil ist wie folgt strukturiert:

9 Zusammenfassung und Ausblick

Literaturverzeichnis

Kapitel 9

Zusammenfassung und Ausblick

In diesem Kapitel fassen wir noch einmal kurz die Ergebnisse dieser Arbeit zusammen und geben einen Ausblick auf zukünftige, darauf aufbauende Arbeiten.

Ergebnisse

Für den Entwurf verteilter objektorientierter Systeme haben wir eine formale Methodik entwickelt, die sich an in der Praxis eingesetzten Techniken orientiert. Dazu haben wir die Beschreibungstechniken

- Automatendokument,
- Objektmodell,
- Klassenbeschreibung und
- Datentypdokument

definiert. Für jede dieser Beschreibungstechniken haben wir eine abstrakte Syntax angegeben und anhand von Beispielen eine Darstellungsform, teilweise mit graphischen Elementen, motiviert.

Ein Datentypdokument definiert die in einem System benutzten Basisdatentypen und deren Funktionen. In einer Klassenbeschreibung wird die Signatur in Form einer Menge von Methoden und Attributen einer Klasse festgelegt. Ein Objektmodell beschreibt die Struktur des Systems durch Daten- und Vererbungsbeziehungen zwischen Klassen. Ein Automatendokument beschreibt schließlich das Verhalten von Agenten einer Klasse auf unterschiedlichen Abstraktionsstufen.

Als formale Grundlage für Softwareentwicklung nutzen wir, wie in Abbildung 9.1 dargestellt, das **Systemmodell**, das die Menge der betrachteten Systeme präzise definiert.

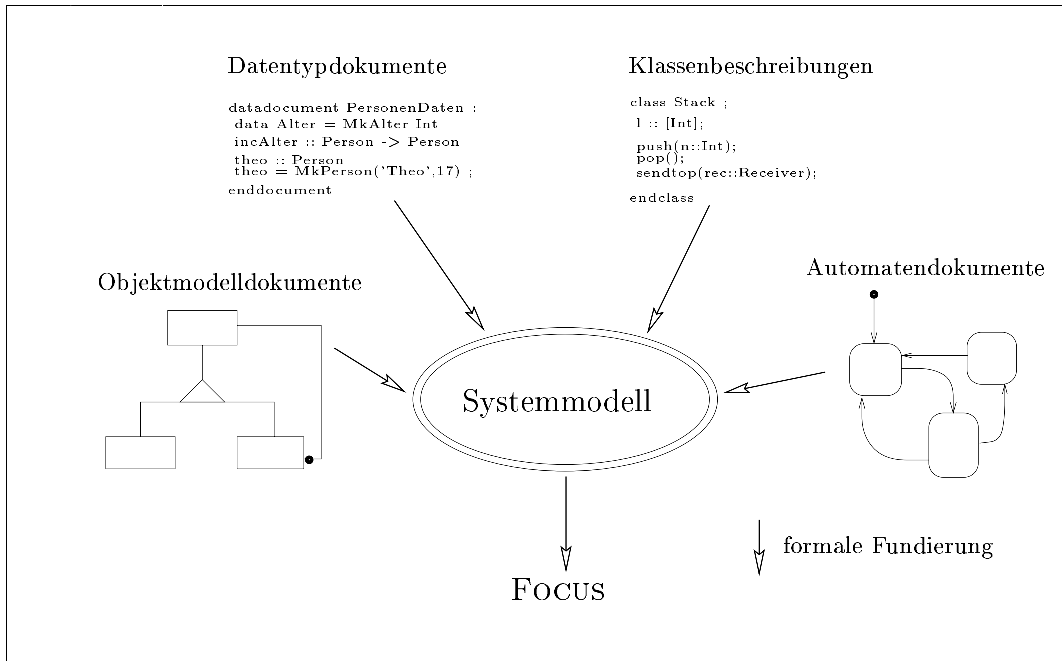


Abbildung 9.1: Formalisierte Dokumentarten dieser Arbeit

Ein System besteht aus einer Menge von Agenten unterschiedlicher Klassen, die asynchron miteinander kommunizieren. Die Kernschicht des Systemmodells formalisiert das Verhalten und den Zustandsraum von Agenten und setzt diese miteinander in Verbindung. Eine zweite Schicht formalisiert syntaxorientierte Begriffe wie Klassen, Attribute und Methoden.

Den Beschreibungstechniken haben wir auf der Basis des Systemmodells eine formale Semantik gegeben. Für jede Beschreibungstechnik haben wir Entwicklungsschritte als syntaktische Transformationen definiert, die eine Präzisierung ihrer Dokumente erlaubt. Wir haben durch formale Beweise gezeigt, daß diese Entwicklungsschritte Verfeinerungen der semantischen Modelle entsprechen. Diese Technik geht weit über bisherige Ansätze zur Formalisierung graphischer Beschreibungstechniken hinaus.

Die definierte Methodik demonstriert, daß in der Praxis eingesetzte graphische Beschreibungstechniken mit formalen Ansätzen kombiniert und so die Vorteile beider Techniken vereint werden können.

Wir untermauern damit die in Kapitel eins aufgestellte These, daß ein Softwareentwicklungswerkzeug, basierend auf diesen Beschreibungstechniken und Entwicklungsschritten, weitaus rigorosere Kontextprüfungen durchführen und besser Beschreibungen einer Technik in eine andere umsetzen kann, als dies bei heute üblichen Werkzeugen der Fall ist.

Als selbständigen Teil obiger Methodik haben wir die **Theorie buchstabierender Automaten** entwickelt. Sie verbindet die Verhaltensbeschreibung einer Komponente mit deren Zustandsbeschreibung und wird in Form von Automatendokumenten in obiger

Methodik eingesetzt. Die Theorie buchstabierender Automaten stellt den Kern der Formalisierung der genannten Beschreibungstechniken dar.

Die Theorie buchstabierender Automaten ist ein Beitrag für das Grundlagenprojekt Focus und ergänzt die Spur- und funktionale Sicht von Focus. Buchstabierende Automaten erlauben eine kompakte, zustandsbasierte Definition von Komponentenverhalten. Sie zeichnen sich dadurch aus, daß eine Transition ein Eingabezeichen verarbeitet und eine Sequenz von Ausgaben produziert. Buchstabierende Automaten können nichtdeterministisch und partiell sein. Sie erlauben sowohl unendliche Zustands- als auch Transitionsmengen.

Für buchstabierende Automaten wurden eine abstrakte Syntax, eine denotationelle und eine operationelle Semantik definiert, und es wurde gezeigt, daß beide Semantiken übereinstimmen.

Ein Kalkül mit einfachen, aber mächtigen Verfeinerungsregeln wurde für buchstabierende Automaten definiert. Es wurde bewiesen, daß dieser Kalkül korrekt bezüglich der denotationellen Semantikdefinition ist. Die Regeln des Verfeinerungskalküls erlauben es bei Einhaltung jeweils bestimmter, syntaxbasierter Kontextbedingungen unter anderem,

- Transitionen hinzuzufügen, wenn das Transitionsdiagramm partiell ist,
- Transitionen zu entfernen, wenn Alternativen existieren,
- Zustände hinzuzufügen,
- unerreichbare Zustände zu entfernen,
- Zustände zu verfeinern,
- Initialelemente zu entfernen und
- eine robuste Totalisierung durchzuführen.

Buchstabierende Automaten sind für die Beschreibung objektorientierter Systeme wesentlich adäquater als I/O-Automaten ([LS89]). Sie besitzen ein Zustandskonzept, das abstrakt genug ist, um in der Beschreibung von Objektverhalten auf Kontrollzustände zu verzichten und nur Datenzustände zu benutzen.

Wir untermauern damit auch die zweite in Kapitel eins formulierte These, daß eine große Klasse von verteilten, reaktiven Applikationen sich damit sehr kompakt entwickeln läßt.

Ausblick

Nun wird kurz auf zukünftige Arbeiten eingegangen, die sich als Weiterentwicklung der Ergebnisse dieser Arbeit anbieten.

Sinnvoll ist zum Beispiel die Erweiterung des methodischen Rahmens und insbesondere des darin benutzten Dokumentgraphen. Zum einen könnten verschiedenartige Beziehungen zwischen Dokumenten des Dokumentgraphen verwendet werden, zum anderen ist ein Zustandskonzept für die Dokumente selbst interessant. So könnte ein Dokument im Dokumentgraph ein Zustandskonzept besitzen, das neben der Redundanz auch Konsistenz oder eine durchgeführte Validierung repräsentiert. Die erlaubten Zustandswechsel für Dokumente können selbst durch ein Transitionssystem beschrieben werden, deren Einhaltung von einem Werkzeug überprüft wird. Darüberhinaus bietet sich die Implementierung dieses Meta-Softwareentwicklungswerkzeugs als allgemeine Plattform für verschiedene Methoden an.

Eine weitere Aufgabe könnte darin bestehen, die hier formalisierten Beschreibungstechniken und Entwicklungsschritte zu implementieren. Eine solche Implementierung besteht unter anderem aus graphischen Dokumenteditoren und Verfahren zur Integration, Transformation und dem Test der Kontextkorrektheit von Dokumenten. Darüberhinaus haben solche Werkzeuge Unterstützung für die graphische Verfeinerung von Dokumenten anzubieten.

Die in dieser Arbeit definierten Beschreibungstechniken konzentrieren sich im wesentlichen auf die zustandsbasierte Beschreibung von Agentenverhalten. Für das Objektmodell bietet sich deshalb eine Erweiterung um allgemeinere Datenbeziehungen an.

Darüberhinaus ist eine Definition weiterer Dokumentarten wie etwa Geschäftsprozesse für die Analysephase oder einer adäquaten Programmiersprache für die Implementierungsphase sinnvoll, um eine durchgängige Methodik für alle Phasen der Systementwicklung zu erhalten. Diese Beschreibungstechniken sind nach dem gleichen Verfahren in das Systemmodell abzubilden, wie dies in dieser Arbeit erfolgt ist. Dadurch entsteht eine größere Sammlung sich ergänzender Beschreibungstechniken.

Eine Methodik besteht aus einer Menge einzelner Entwicklungsschritte, die erst durch eine Methode zu einer zielgerichteten Vorgehensweise kombiniert werden. Daher ist es sinnvoll, Methoden auszuprägen die, angepaßt an das jeweilige Problemumfeld, eine Teilmenge der angebotenen Beschreibungstechniken und Entwicklungsschritte in einer zielgerichteten Vorgehensweise kombinieren. Dies kann jedoch nur anhand von praxisorientierten Beispielentwicklungen geschehen und evaluiert werden.

Literaturverzeichnis

[ABA93] M. Abadi. Baby Modula-3 and a Theory of Objects. SRC Research Report 95, Digital Equipment Corporation, February 1993.

[AC93] R. M. Amadio and L. Cardelli. Subtyping Recursive Types. SRC Research Report 62, Digital Equipment Corporation, January 1993.

[AG90] C. Asworth and M. Goodland. *SSADM – A practical approach*. McGraw-Hill, 1990.

[AG92] A. J. Alencar and J. A. Goguen. OOZE with Examples. Technical report, Programming Research Group, Oxford University Computing Laboratory, 1992.

[AL88] M. Abadi and L. Lamport. The Existence of Refinement Mappings. SRC Research Report 29, Digital Equipment Corporation, 1988.

[AL90] M. Abadi and L. Lamport. Composing Specifications. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems*, volume LNCS 430, pages 1–41. Springer-Verlag, Berlin, 1990.

[AVE95] J. Avenhaus. *Reduktionssysteme*. Springer-Verlag Berlin, 1995.

[BDDW91] M. Broy, F. Dederichs, C. Dendorfer, and R. Weber. Characterizing the Behaviour of Reactive Systems by Trace Sets. SFB-Bericht 342/2/91 A, Technische Universität München, February 1991.

[BER93] G. Berry. The Semantics of Pure Esterel. In M. Broy, editor, *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.

[Betal85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L*. LNCS 183. Springer-Verlag, 1985.

[Betal93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems — An Introduction to FOCUS – revised version –. SFB-Bericht 342/2-2/92 A, Technische Universität München, January 1993.

[Betal93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 1. Technical Report TUM-I9312, Technische Universität München, 1993.

[Betal93c] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 2. Technical Report TUM-I9312, Technische Universität München, 1993.

[BOO94] G. Booch. *Object Oriented Analysis and Design with Applications.* Benjamin/Cummings Publishing Company, Inc., 1994.

[BRA84] W. Brauer. *Automatentheorie: eine Einführung in die Technik endlicher Automaten.* Teubner, 1984.

[BRO91] M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

[BRO92] M. Broy. Compositional Refinement of Interactive Systems. SRC Research Report 89, Digital Equipment Corporation, July 1992.

[BRO93] M. Broy. Specification of an Unreliable Buffer of Length One. Report 2, Working Material of Marktoberdorf Summer School, 1993.

[BRO94] M. Broy. (Inter-) Action Refinement, The Easy Way. In Jerzy Rozenblit and Klaus Buchenrieder, editors, *Codesign – Computer Aided Software/Hardware Engineering*, chapter 3, pages 67–97. IEEE Press New York, 1994.

[BRO95] M. Broy. Mathematical Models in Software Engineering. In J. van Leeuwen, editor, *Computer Science Today*, pages 292–306. Springer-Verlag, LNCS 1000, 1995.

[BS94] M. Broy and K. Stølen. Specification and Refinement of Finite Dataflow Networks — a Relational Approach. In *Proc. FTRTFT'94*, LNCS 863, pages 247–267. Springer-Verlag, Berlin, 1994.

[BUD87] T. Budd. *A Little Smalltalk.* Addison-Wesley Publishing Company, 1987.

[BUD91] T. Budd. *An introduction to object oriented programming.* Addison-Wessley Publishing Company, 1991.

[BW82] F. L. Bauer and H. Wössner. *Algorithmic language and program development.* Springer-Verlag Berlin, 1982.

[CAR84] L. Cardelli. A Semantics of Multiple Inheritance. In *Proc. Semantics of Data Types*, pages 51–68. Springer-Verlag, 1984.

[CAR93] L. Cardelli. Extensible Records in a Pure Calculus of Subtyping. SRC Research Report 81, Digital Equipement Corporation, January 1993.

[CC92] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *19th POPL*, pages 83–94. ACM, 1992.

[CDI92] CDIF Technical Commitee. Introduction to CDIF: The CASE data interchange format standards, April 1992. CDIF Technical Commitee, LBMS, Evelyn House, 62 Oxford Street, London W1N 9LF, United Kingdom.

[Cetal92] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Language Definition. *ACM Sigplan Notices*, 27(8):15–42, August 1992.

[Cetal94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[CEW92a] S. Clyde, D. Embley, and S. Woodfield. The Complete Formal Definition for the Syntax and Semantics of OSA. Technical Report BYU-CS-92-2, Brigham Young University, Computer Science Department, February 1992.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL light: A Core Language for Specifying Objects. Technical report, Technische Universität Braunschweig, Informatik, June 1992.

[CHE76] P. Chen. The entity-relationship model – Towards a unified view of data. In *ACM Transactions on Database Systems*. ACM Press, 1976.

[CY91] P. Coad and E. Yourdon. *Object-oriented design*. Prentice Hall, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-oriented analysis*. Prentice Hall, 1991.

[DED92] F. Dederichs. *Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen*. PhD thesis, Technische Universität München, Juni 1992.

[DEM79] T. DeMarco. *Structrured analysis and system specification*. Prentice Hall, 1979.

[DEN91] C. Dendorfer. Funktionale Modellierung eines Postsystems. SFB-Bericht 342/28/91 A, Technische Universität München, October 1991.

[DEN91b] E. Denert. *Software-Engineering*. Springer-Verlag, 1991.

[DEN93] E. Denert. Dokumentenorientierte Software-Entwicklung. *Informatik-Spectrum*, 16(3):159–164, June 1993.

[DEN95] C. Dendorfer. *Methodik funktionaler Systementwicklung.* PhD thesis, Technische Universität München, 1995.

[DIJ76] E. W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[DXT92] J. Drake, W. Xie, and W. Tsai. Document-Driven Analysis: Description and Formalization. Technical report, Department of Computer Science University of Minnesota, 1992.

[EFT86] H-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik.* Wissenschaftliche Buchgesellschaft Darmstadt, 1986.

[EKW92] D. Embley, B. Kurtz, and S. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach.* Yourdon Press, Englewood Cliffs, 1992.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[FAC95] C. Facchi. *Methodik zur formalen Spezifikation des ISO/OSI Schichtenmodells.* PhD thesis, Technische Universität München, 1995.

[FUC94] M. Fuchs. *Technologieabhängigkeit von Spezifikationen digitaler Hardware.* PhD thesis, Technische Universität München, Juli 1994.

[Getal92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report CSL-92-03, Computer Science Laboratory, SRI, March 1992.

[Getal94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[Getal96] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State transition diagrams. Technical Report to appear, Technische Universität München, 1996.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* LNCS 78, Springer-Verlag, 1979.

[GR95] R. Grosu and B. Rumpe. Concurrent timed port automata. Technical Report TUM-I9533, Technische Universität München, 1995.

[GRO95] R. Grosu. *A Formal Foundation for Concurent Object Oriented Programming.* PhD thesis, Technische Universität München, Januar 1995.

[GS95] R. Grosu and K. Stølen. A Denotational Model for Mobile Point-to-Point Dataflow Networks. Technical Report TUM-I9527, Technische Universität München, 1995.

[HAR87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[HAR92] S. Harbison. *Modula-3.* Prentice-Hall, 1992.

[HET93] R. Hettler. Zur Übersetzung von E/R-Schemata nach SPECTRUM. Technical Report TUM-I9333, Technische Universität München, 1993.

[HET95] R. Hettler. *Entity/Relationship-Datenmodellierung in axiomatischen Spezifikationssprachen.* PhD thesis, Technische Universität München, 1995.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the Programming Language Haskell, A Non-strict Purely Functional Language. In *Sigplan Notices*, volume 27 of *ACM*. ACM Press, May 1992.

[HL93] H. J. Habermann and F. Leymann. *Repository - eine Einführung.* Handbuch der Informatik. Oldenbourg, 1993.

[HM93] A. Hevner and H. Mills. Box-Structured Methods fos Systems Development with Objects. *IBM Systems Journal*, 2:232–251, 1993.

[HOA69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[HOA85] C.A.R. Hoare. *Communicationg Sequential Processes.* Prentice-Hall, 1985.

[HOG89] D. Hogrefe. *Estelle, LOTOS und SDL: Standard–Spezifikationssprachen für verteilte Systeme.* Springer-Verlag, 1989.

[HU90] J. Hopcroft and J. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 1990.

[HUD90] P. Hudak et al. Report on the programming language HASKELL. Technical Report YALEU/DCS/RR-777, Yale University, CS Dept., April 1990.

[HUS93] H. Hußmann. Synergy Between Formal and Pragmatic Software Engineering Methods. Technical Report TUM-I9323, Technische Universität München, 1993.

[HUS93b] H. Hußmann. Zur formalen Beschreibung der funktionalen Anforderungen an ein Informationssystem. Technical Report TUM-I9332, Technische Universität München, 1993.

[HUS94] H. Hußmann. Formal Foundations for SSADM. Technische Universität München, Habilitation Thesis, 1994.

[INM91] W. Inmon. *Client/Server–Anwendungen, Planung und Entwicklung.* Springer-Verlag, Berlin, 1991.

[JAC83] M. Jackson. *System Development.* Prentice Hall, 1983.

[JAC93] I. Jacobson. *Object-Oriented Software Engineering - a Use Case Driven Approach*. Addison Wesley, 1993.

[Jetal91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The Troll Language (Version 0.01). Informatik-Berichte 91-04, Technische Universität Braunschweig, December 1991.

[JON87] B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Uppsala University, Uppsala Sweden, 1987.

[JON89] B. Jonsson. A Fully Abstract Trace Model for Dataflow Networks. In *ACM SIGPLAN Symposium on Principles of Programing Languages*. ACM, 1989.

[JON90] C. B. Jones. *Systematic Software Development Using VDM, Second Edition*. Prentice-Hall, 1990.

[JON93a] M. P. Jones. *An Introduction to Gofer*, 1993.

[KAH74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Journal of Information Processing*, 74:471–475, 1974.

[KLE52] S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[KM77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977. Proc. IFIP Congress.

[KRB96] C. Klein, B. Rumpe, and M. Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model - . In Jean-Bernard Stefani Elie Naijm, editor, *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, pages 323–338. ENST France Telecom, 1996.

[LAM77] L. Lamport. Proving the Correctness of Multiprocessor Systems. *IEEE Journal on Software Engineering*, 3(2), 1977.

[LH89] K. Lieberherr and I. Holland. Assuring Good Style for Object-Oriented Programms. *IEEE Software*, 6(5):38–48, 1989.

[LS89] N. Lynch and E. Stark. A Proof of the Kahn Principle for Input/Output Automata. *Information and Computation*, 82:81–92, 1989.

[LW93] B. Liskov and J. M. Wing. A New Definition of the Subtype Relation. In *ECOOP 93*, pages 118–141, 1993.

[MDL87] H. Mills, M. Dyer, and R. Linger. Cleanroom Software Engineering. *IEEE Software*, 4:19–24, 1987.

[MEY92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[MIL80] R. Milner. *A Calculus of Communicating Systems.* LNCS 92, Springer-Verlag, 1980.

[MIL89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[MIL90] R. Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science*, volume B, chapter 19, pages 1202–1242. Elsevier Science Publisher, 1990.

[MO93] H. Mössenböck. *Objektorientierte Programmierung in Oberon-2.* Springer-Verlag, 1993.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus for Mobile Processes. *Information and Computation*, 100:1–77, 1992.

[MS95] D. McDyson and D. Spohn. *ATM: Theory and application.* McGraw-Hill Series on Computation, McGraw-Hill, 1995.

[MTY93] S. Matsuoka, K. Taura, and A. Yonezawa. Highly Effizient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages. In A. Paepke, editor, *OOPSLA '93.* ACM Press, October 1993.

[NIC93] F. Nickl. Ablaufspezifikation durch Datenflußmodellierung und stromverarbeitende Funktionen. Technical Report TUM-I9334, Technische Universität München, 1993.

[NIE91] O. Nierstrasz. Towards an Object Calculus. In P. Wegner M. Tokoro, O. Nierstrasz, editor, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS, 1991.

[NIE93] O. Nierstrasz. Regular Types for Active Objects. In A. Paepke, editor, *OOPSLA '93.* ACM Press, October 1993.

[NR69] P. Naur and B. Randell. *Software Engineering.* Nato Science Commitee, 1969.

[OJ93] W. F. Opdyke and R. E. Johnson. Creating Abstract Superclasses by Refactoring. Technical report, Department of Computer Science, University of Illinois and AT&T Bell Laboratories, 1993.

[PAR92] C. Parnas. Tabular Representation of Relations. Technical Report 260, CRL, October 1992.

[PAU91] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[PAU94] L. Paulson. *Isabelle: A Generic Theorem Prover.* LNCS 929, Springer-Verlag, 1994.

[PB93] G. Pomberger and G. Blaschek. *Software Engineering.* Carl Hanser Verlag, 1993.

[PER90] D. Perrin. Finite Automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Band B, chapter 1, pages 1–58. Elsevier Publisher Amsterdam, 1990.

[PR94] B. Paech and B. Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *FME'94, Formal Methods Europe, Symposium '94*, LNCS 873. Springer-Verlag, Berlin, October 1994.

[PR94b] B. Paech and B. Rumpe. Spezifikationsautomaten: Eine Erweiterung der Spezifikationssprache SPECTRUM um eine graphische Notation. In *Formale Grundlagen für den Entwurf von Informationssystemen*, GI-Workshop, Tutzing 24.-26. Mai 1994 (GI FG 2.5.2 EMISA). Institut für Informatik, Universität Hannover, May 1994.

[PRE95] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[PS91] J. Palsberg and M. Schwartzbach. What is Type-Safe Code Reuse? In *ECOOP '91*, LNCS 512, pages 325–341. Springer-Verlag, 1991.

[PS92] J. Palsberg and M. Schwartzbach. Three Discussions on Object-Oriented Typing. *OOPS Messenger*, 3(2):31–39, 1992.

[REG94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, November 1994.

[RKB95] B. Rumpe, C. Klein, and M. Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab Systemmodell -. Technical Report TUM-I9510, Technische Universität München, March 1995.

[RS93] W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 284–293. LNCS 761, 1993.

[RUM91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[RUM94] B. Rumpe. Verwendung endlicher Automaten zur Implementierung des dynamischen Verhaltens von C++ Objekten. In G. Snelting U. Meyer, editor, *Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, 9402. Justus-Liebig-Universität Giessen, March 1994.

[RUM95] Bernhard Rumpe. Gofer Objekt-System – Imperativ Objektorientierte und Funktionale Programmierung in einer Sprache vereint . In Tiziana Margaria,

editor, *Kolloquium Programmiersprachen und Grundlagen der Programmierung, Adalbert Stifter Haus, Alt Reichenau, 11.10.1995*, pages 35–47. Universität Passau, MIP-9519, 1995.

[SDW96] K. Stølen, F. Dederichs, and R. Weber. Specification and Refinement of Networks of Asynchronously Communicating Agents Using the Assumption/Commitment Paradigm. *Formal Aspects of Computing*, 8:127–161, 1996.

[Setal94] G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, and R. Feenstra. Inheritance Conditions for Object Life Cycle Diagramms. In *Formale Grundlagen für den Entwurf von Informationssystemen*, GI-Workshop, Tutzing 24.-26. Mai 1994 (GI FG 2.5.2 EMISA). Institut für Informatik, Universität Hannover, May 1994.

[SHB96] B. Schätz, H. Hußmann, and M. Broy. Graphical Development of Consistent System Specifications. In J. Woodcock M. Gaudel, editor, *FME'96: Industrial Benefit and Advances In Formal Methods*, pages 248–267. Springer-Verlag, 1996. LNCS 1051.

[SM92] S. Shlaer and S. Mellor. *Object Lifecycles, Modeling the World in States.* Yourdon Press, 1992.

[SPI88] J. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[SPI94] K. Spies. Funktionale Spezifikation eines Kommunikationsprotokolls. SFB-Bericht 342/08/94 A, Technische Universität München, Mai 1994.

[STE92a] S. Stepney, R. Barden, and D. Cooper. A Survey of Object Orientation in Z. *Software Engineering Journal*, March 1992.

[STO95] K. Stølen. A Framework for the Specification and Development of Reactive Systems. In *Proc. 5. GI/ITG-Fachgespräch, Formale Beschreibungstechniken für verteilte Systeme*, pages 41–50, 1995.

[STR91a] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[SWF91] T. Schnekenburger, A. Weininger, and M. Friedrich. Introduction to the Parallel and Distributed Programming Language ParMod-C. SFB-Bericht 342/27/91 A, Technische Universität München, Oktober 1991.

[TAR55] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[THI94] P. Thiemann. *Grundlagen der funktionalen Programmierung.* Teubner Stuttgart, 1994.

[Uetal91] D. Ungar, C. Chambers, B. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3), 1991.

[WEG90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, 1990.

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and Implementation of ET++ a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2), 1998.

[WIR90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier Science Publishers B. V., 1990.

[WWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

[WZ88] P. Wegner and S. Zdonik. Inheritance as an Incremental Modification Mechanism. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88, LNCS 322*, pages 55–77. Springer-Verlag, Oslo, August 1988.

Teil V

Anhang

Der Anhang enthält die grundlegenden mathematischen Hilfsmittel, die in dieser Arbeit benutzt werden. Insbesondere werden alle verwendeten Focus-Konzepte in kompakter Weise zusammengetragen und dargestellt.
Längere Beweise zu Propositionen und Sätzen dieser Arbeit und dafür entwickelte Beweisschemata sind ebenfalls im Anhang gesammelt. Ein Glossar erklärt die verwendeten Begriffe. Ein Index beschreibt die benutzten Symbole.
Dieser Teil ist wie folgt strukturiert:

A Mathematische Grundlagen

B Beweisschemata

C Beweise

D Glossar

E Index

Anhang A

Mathematische Grundlagen

Dieses Kapitel enthält grundlegende Definitionen der von uns genutzten mathematischen Modellierungstechnik FOCUS. FOCUS wurde am Lehrstuhl Broy entwickelt und dient zur mathematischen Fundierung von Beschreibungen verteilter Systeme. In der Einführung [Betal93] zu FOCUS und einigen erweiternden Arbeiten ([DED92, DEN95]) werden neben grundlegenden Definitionen auch eine Methodik zur Anwendung von FOCUS im Softwareentwurf definiert.

Ein *verteiltes System* wird als eine konzeptuell oder räumlich verteilte Menge von Komponenten verstanden, die mittels *Nachrichten asynchron kommunizieren*. Ein verteiltes System kann als Kahn-Netzwerk ([KAH74, KM77]) verstanden werden, in dem als einziges Kommunikationsmittel Datenfluß über Kanäle erlaubt wird. FOCUS besitzt Kompositions- und Verfeinerungstechniken, wie sie auch im Cleanroom Software Engineering Ansatz (siehe [HM93, MDL87]) propagiert werden.

Von FOCUS werden im wesentlichen die mathematischen Modellierungskonzepte für verteilte Systeme genutzt. Diese Grundlagen gehen neben der bereits oben genannten Einführung im wesentlichen auf [BRO92], [GS95] und [STO95] zurück. Im Index E.1 findet sich eine Kurzbeschreibung der verwendeten Notation.

A.1 Mathematische Grundlagen, Notation

Mengen

Zu einer Menge M sei $\mathbb{P}(M)$ die Potenzmenge und $\mathbb{P}^{fin}(M)$ die Menge aller endlichen Teilmengen von M.

Funktionen

Eine *Funktion* f zwischen den Mengen X und Y ist ein Tripel mit den Mengen X und Y und einer ebenfalls mit f bezeichneten Teilmenge von $X \times Y$ als Komponenten, wobei

jedem $x \in X$ höchstens ein Bild $y \in Y$ zugeordnet wird. Wir schreiben $\bullet f$ für die Urbildmenge X und $f\bullet$ für die Bildmenge Y. Die Funktionsapplikation schreiben wir geklammert $f(x)$, ungeklammert $f\ x$, in linksassoziativer Punktschreibweise $f.x$ oder in Subskriptionsschreibweise f_x.

Die Menge der Funktionen mit Urbild $\bullet f = X$ und Bild $f\bullet \subseteq Y$ bezeichnen wir mit $X \to Y$. Die Menge der *partiellen Funktionen* $X \rightharpoonup Y$ bezeichnet die Menge aller Funktionen f mit Urbild $\bullet f \subseteq X$ und Bild $f\bullet \subseteq Y$.

Ein *Prädikat* über einer Menge X betrachten wir wahlweise als Teilmenge von X oder als Funktion der Form $X \to \mathbb{B}$ in die Menge der Booleschen Werte $\mathbb{B} = \{tt, ff\}$. $P.x$ ist dementsprechend gleichbedeutend mit $x \in P$.

Wir verwenden die üblichen Operationen zur Manipulation von Funktionen, Mengen und Prädikaten. Insbesondere erweitern wir die Funktionsapplikation punktweise auf Teilmengen des Urbilds

$$Y \subseteq X \Rightarrow f(Y) \stackrel{def}{=} \bigcup_{y \in Y} f(y)$$

und schreiben die Vereinigung von kompatiblen Funktionen, das heißt $f|_{\bullet g} = g|_{\bullet f}$, in der Form

$$(f+g).x \stackrel{def}{=} \begin{cases} f.x & \texttt{if} \quad x \in \bullet f \\ g.x & \texttt{if} \quad x \in \bullet g. \end{cases}$$

Die mathematische *Funktionsabstraktion* schreiben wir in λ-Notation $(\lambda x{:}M.F)$ mit Funktionsrumpf F. $[x \mapsto y]$ bezeichnet die einelementige Funktion mit Urbild $\{x\}$ und Bild $\{y\}$.

Vollständig partiell geordnete Mengen

Eine *Kette* C ist eine nichtleere, bezüglich einer partiellen Ordnung $\sqsubseteq$ vollständig geordnete Teilmenge einer Menge X. Wenn die Menge X ein *kleinstes Element* (meistens als $\perp$ bezeichnet) enthält und für deren Ketten $C \subseteq X$ kleinste obere Schranken in X existieren, dann nennen wir X eine *vollständig partiell geordnete Menge* (kurz: *CPO*). Für Ketten C schreiben wir $chain(C)$ für ihre kleinste obere Schranke $\sqcup C$. Wir notieren Ketten auch als (monotone) Abbildungen der Form $I \to C$ über einer vollständig geordneten Indexmenge I und schreiben $C = (C_i)_{i \in I}$ oder kurz $C = (C_i)_i$. Für abzählbare Ketten sei $I = \mathbb{N}$.

Eine Funktion $\tau \in X \to Y$ zwischen CPO's $(X, \sqsubseteq)$ und $(Y, \sqsubseteq)$ heißt *monoton*, wenn

$$\forall x, y \in X.\ x \sqsubseteq y \Rightarrow \tau.x \sqsubseteq \tau.y$$

und *stetig*, wenn für alle abzählbaren Ketten $C \subseteq X$ gilt:

$$chain(C) \Rightarrow \tau(\sqcup C) = \sqcup(\tau.C).$$

Wir bezeichnen die Menge der stetigen Funktionen von X nach Y mit $X \stackrel{s}{\to} Y$.

Nach [TAR55] besitzt eine bezüglich $\sqsubseteq$ monotone Funktion $\tau \in X \to X$ einen eindeutigen *kleinsten Fixpunkt*, den wir mit $fix.\tau$ bezeichnen. Eine Eigenschaft P auf X heißt *zulässig* (engl. „admissible“) für beliebige Ketten, wenn sie folgende Definition erfüllt:

$$\forall (C_i)_i.\ chain((C_i)_i) \Rightarrow ((\forall i.\ P(C_i)) \Rightarrow P(\sqcup_i C_i))$$

Wir schreiben dann $adm(P)$. Ist τ bezüglich $\sqsubseteq$ stetig, so kann $\mathit{fix}.\tau$ durch Grenzwertbildung $\mathtt{lim}_{n\to\infty}\tau^n(\bot)$ berechnet werden.

Für jede Menge X ist der Potenzmengenverband $\mathbb{P}(X)$ gemeinsam mit der inversen Mengeninklusion $\supseteq$ ein CPO, dessen kleinstes Element der vollen Menge X entspricht und das wir mit $\top \stackrel{def}{=} X$ bezeichnen. Der kleinste Fixpunkt bezüglich $\supseteq$ ist gerade der mengentheoretisch größte Fixpunkt.

Wir nennen ein Element $Y \subseteq \top$ *Präfixpunkt*, wenn gilt $Y \subseteq \tau(Y)$. Der mengentheoretisch größte Fixpunkt läßt sich als Vereinigung aller Präfixpunkte charakterisieren. Es gilt: $\mathit{fix}.\tau = \bigcup\{Y \mid Y \subseteq \tau(Y)\}$.

Für jedes Paar von CPO's $(X,\sqsubseteq)$ und $(Y,\sqsubseteq)$ ist die Menge der stetigen Funktionen $X \stackrel{s}{\to} Y$ vermöge der punktweisen Approximation

$$f \sqsubseteq g \Leftrightarrow \forall x.\ f.x \sqsubseteq g.x$$

ebenfalls ein CPO. Diese Definition werden wir insbesondere bei stromverarbeitenden Funktionen verwenden.

Zu jedem Paar von CPO's $(X,\sqsubseteq)$ und $(Y,\sqsubseteq)$ ist das Kreuzprodukt $X \times Y$ vermöge

$$(x_1,y_1) \sqsubseteq (x_2,y_2) \Leftrightarrow x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$$

ebenfalls ein CPO. Für einfache Mengen S, z.B. von Zuständen, läßt sich der flache CPO $(S^\bot,\sqsubseteq)$ wie folgt definieren:

$$S^\bot \stackrel{def}{=} S \cup \{\bot\}$$

$$s \sqsubseteq t \Leftrightarrow s=\bot \vee s=t.$$

Joker

Zur kompakten Darstellung von logischen Formeln und von Mengenkomprehension verwenden wir, ähnlich dem Joker in der funktionalen Programmierung, das Zeichen $*$ als anonyme Variable. In einer Formel stellt jedes Auftreten von $*$ eine auf innerster Ebene existenzquantifizierte Variable dar. Mehrere $*$ stehen für unabhängig quantifizierte Variablen. Beispielsweise ist $P \wedge \delta(s,m,*,*)$ äquivalent mit $P \wedge \exists a,b.\delta(s,m,a,b)$.

In einer Mengenkomprehension wird $*$ für eine beliebige, nicht weiter eingeschränkte Variable verwendet. Beispiel $\{(s,m,*,*) \mid P(s,m)\}$ ist die Kurzform für $\{(s,m,a,b) \mid P(s,m)\}$.

A.2 Klassisches, ungezeitetes FOCUS

In diesem Abschnitt werden die in FOCUS definierten Modellierungstechniken, die auf ungezeiteten Strömen beruhen, eingeführt.

Ströme

Definition A.1 (Monoid der Ströme)
Sei M eine Menge von Zeichen, *die wir im folgenden auch als* Nachrichten *bezeichnen, so bildet die Menge*

M^* *die Menge aller* endlichen Ströme *über M, die als die Menge aller endlichen Tupel über M definiert ist,*

M^∞ *die Menge aller* unendlichen Ströme, *die als die Menge aller (abzählbar) unendlichen Tupel über M definiert ist und*

$M^\omega \stackrel{def}{=} M^* \cup M^\infty$ *die Menge aller* Ströme *über M.*

Wir führen folgende Bezeichnungen ein:

ε *für den* leeren Strom,

$\langle . \rangle$ *für einelementige Ströme,*

$.\hat{}.$ *für die Konkatenation zweier Ströme, die wir durch Überladung auch dazu verwenden einzelne Elemente mit Strömen zu konkatenieren, und*

$\#.$ *für die Länge eines Stroms, die eine natürliche Zahl für endliche Ströme und* ∞ *für unendliche Ströme bezeichnet.*

Für $s,t \in M^\omega$ *und* $m \in M$ *gelten folgende Gesetze:*

$$\varepsilon\,\hat{}\,s = s\,\hat{}\,\varepsilon = s$$
$$(s\,\hat{}\,t)\,\hat{}\,u = s\,\hat{}\,(t\,\hat{}\,u)$$
$$\#\varepsilon = 0$$
$$\#\langle m\rangle = 1$$
$$\#(s\,\hat{}\,t) = \#s + \#t$$
$$\#s = \infty \Rightarrow s\,\hat{}\,t = s.$$

Wegen der Assoziativität der Konkatenation lassen wir die Klammern im allgemeinen weg. Wir haben so einen Monoid mit Längenfunktion definiert. ($\Box$)

Ein Strom modelliert das Geschehen auf einem *Kanal*, das im Laufe der Zeit, beginnend ab einem bestimmten Zeitpunkt bis in die unendliche Zukunft, beobachtet werden kann. Die Konkatenation einer geordneten Menge $(s_i)_i$ von Strömen wird mit $\texttt{conc}_{i=1}^{\infty} s_i$ bezeichnet.

Definition A.2 (Approximation von Strömen)
Wir definieren die Präfixrelation $.\sqsubseteq.$ *auf Strömen mittels*

$$\forall s,t \in M^\omega.\ \ s \sqsubseteq t \Leftrightarrow \exists u \in M^\omega.\ s\,\hat{}\,u = t. \qquad (\Box)$$

Die Menge $(M^\omega, \sqsubseteq)$ stellt damit einen CPO dar. Da es für unsere Zwecke ausreicht, nutzen wir eine flache Approximationsordnung auf der zugrundeliegenden Menge M von Zeichen. Mit der obigen Approximationstechnik lassen sich rekursiv definierte Ströme durch kleinste Fixpunkte im Sinne von Knaster/Tarski ([TAR55]) beschreiben, die aufgrund von Stetigkeitsüberlegungen des in so einem Falle zugrundegelegten Funktionals durch eine Kleene-Kette endlicher Ströme approximiert werden können ([KLE52]).

Definition A.3 (Hilfsfunktionen auf Strömen)
Wir definieren folgende Hilfsfunktionen

$A©s$ *als Filterfunktion, die aus einem Strom s nur die Menge der Zeichen A filtert.*

$\#_A s$ *als Zählfunktion, die angibt, wieviele Zeichen der Menge A im Strom s vorkommen.*

Es gelten folgende Gesetze:

$$m \in A \Rightarrow A©(m\hat{}s) = m\hat{}(A©s)$$
$$m \notin A \Rightarrow A©(m\hat{}s) = A©s$$
$$\#_A s = \#(A©s). \qquad (\Box)$$

Stromtupel und Strombündel

Zur Modellierung des Geschehens auf mehreren Kanälen definieren wir:

Definition A.4 (Stromtupel und Strombündel)
Sei B eine beliebige Menge von Kanalnamen *und* $(M_b)_{b \in B}$ *ein indiziertes System von Nachrichtenmengen.* M_b *heißt dann* Kanaltyp *von Kanal b. Sei* $M = \bigcup_{b \in B} M_b$*, dann wird der* Bündeltyp B^Ω *definiert als*

$$B^\Omega \stackrel{def}{=} \{\ s \in B \rightarrow M^\omega \mid \forall b \in B.\ s.b \in {M_b}^\omega\ \}.$$

Die Elemente aus B^Ω *heißen* Strombündel *vom Typ B.*
Mittels der punktweisen Erweiterung der Präfixordnung auf Bündel

$$\forall s,t \in B^\Omega.\ s \sqsubseteq t \stackrel{def}{\Leftrightarrow} \forall b \in B.\ s.b \sqsubseteq t.b$$

wird auch B^Ω *zu einem CPO.*

Wir definieren die Menge der endlichen Strombündel *als*

$$B^\Phi \stackrel{def}{=} \{\ s \in B \rightarrow M^* \mid \forall b \in B.\ s.b \in {M_b}^*\ \}. \qquad (\Box)$$

Für endliche Abschnitte $B=[1..n] \subseteq \mathbb{N}$ sprechen wir statt von Strombündeln auch von *Stromtupeln*, und bezeichnen diese mit $(M^\omega)^n$. Für einelementige Mengen $\{b\}$ identifizieren wir die isomorphen Mengen $\{b\}^\Omega$ und ${M_b}^\omega$, wo der Kontext dies eindeutig erlaubt.

Im folgenden benutzen wir meistens eine feste Menge M für den Typ aller Kanäle und setzen $M_b \stackrel{def}{=} M$ für alle $b \in B$. Wir erweitern die auf Ströme definierten Funktionen punktweise auf Strombündel, wobei die Längenfunktion $\#.$ die Länge des kleinsten im Bündel vorhandenen Stroms definiert.

Stromverarbeitende Funktionen

Die Schnittstelle einer Komponente modellieren wir durch ein Paar (I,O) von Eingabekanälen und Ausgabekanälen mit $I,O \subseteq B$, wobei I und O nicht notwendigerweise disjunkt sein müssen. Das Verhalten einer Komponente läßt sich dann als mathematische Abbildung $f \in I^\Omega \rightarrow O^\Omega$ modellieren, die das Ausgabestrombündel mit dem Eingabestrombündel in Beziehung setzt. Graphisch läßt sich eine so modellierte Komponente wie in Abbildung A.1 veranschaulichen.

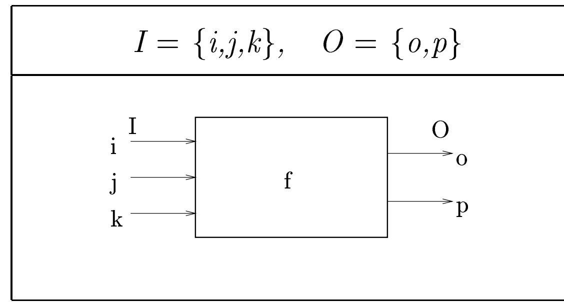


Abbildung A.1: Graphische Darstellung einer Komponente

Die Abbildung f unterliegt einigen Einschränkungen, damit sie beobachtbares, operationelles Verhalten charakterisiert. Die Monotonie von f bzgl. $\sqsubseteq$ sichert, daß die modellierte Komponente einmal stattgefundene Ausgaben nicht zurücknehmen kann. Die Stetigkeit garantiert, daß jede Ausgabenachricht einer Komponente als Reaktion auf einen endlichen Anteil der Eingabe entsteht. Die Monotonie folgt aus der Stetigkeit.

Definition A.5 (Stromverarbeitende Funktionen)
Eine Funktion $f \in I^{\Omega} \rightarrow O^{\Omega}$ heißt stromverarbeitende Funktion*, wenn sie stetig ist. Die Menge der stromverarbeitenden Funktionen mit Eingabestrombündel I und Ausgabestrombündel O bezeichnen wir mit $I^{\Omega} \stackrel{s}{\rightarrow} O^{\Omega}$.*

($\square$)

Prädikate über stromverarbeitenden Funktionen

Eine Komponente verhält sich *nichtdeterministisch*, wenn sie bei einer Eingabegeschichte verschiedene Ausgaben erzeugen kann. Eine nichtdeterministische Komponente läßt sich nicht durch eine einzelne stromverarbeitende Funktion charakterisieren. Stattdessen verwenden wir eine Menge stromverarbeitender Funktionen, die wir durch ein Prädikat definieren.

Vom Standpunkt des Beobachters aus gibt es keinen Unterschied, ob eine Komponente *nichtdeterministisch implementiert* ist, oder der Beobachter nur unzureichende Information über die Komponente besitzt, also diese für ihn *unterspezifiziert* ist. Nichtdeterminismus und Unterspezifikation werden beide durch Prädikate $P \subseteq I^{\Omega} \stackrel{s}{\rightarrow} O^{\Omega}$ über stromverarbeitenden Funktionen modelliert.

Die Potenzmenge über den stetigen stromverarbeitenden Funktionen stellt gemeinsam mit der umgekehrten Mengeninklusion $\supseteq$ einen CPO dar. Die Ordnung $\sqsubseteq$, die auf den stetigen Funktionen definiert ist, spielt dabei keine Rolle.

Kompositionstechniken

Das klassische Focus bietet drei Kompositionsoperationen an,

- die *sequentielle Komposition*,
- die *parallele Komposition* und
- die *Rückkopplung*,

deren Wirkung in Abbildung A.2 graphisch dargestellt ist.

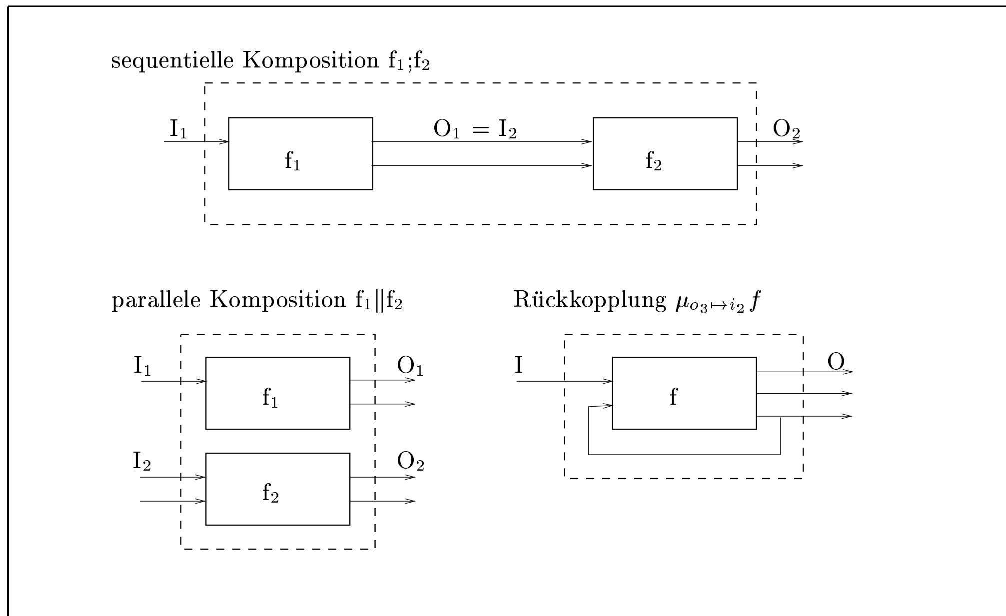


Abbildung A.2: Klassische Focus Kompositionsoperatoren

Weil eine Komponente durch Funktionen modelliert wird, ist sequentielle Komposition durch Funktionskomposition darstellbar. Bei der parallelen Komposition wird eine neue Funktion auf dem Produktbereich beider Domänen konstruiert, deren Projektionen jeweils die ursprünglichen Funktionen ergeben. Bei der Rückkopplung von Ausgabeströmen zu Eingabeströmen besitzt die charakterisierende rekursive Gleichung mehrere Lösungen. Wir verwenden daher den kleinsten Fixpunkt des aus der Gleichung entstehenden Funktionals.

Wir benutzen in unserer Arbeit die an [DEN95] und [GS95] angelehnte parallele Komposition $\|$ aus Abbildung A.3 und den Rückkopplungsoperator μ aus Abbildung A.4.

Definition A.6 (Parallele Komposition $\|$)
Sei N eine Menge von Namen und für jedes $n{\in}N$ eine stromverarbeitende Funktion $f_n{\in}{I_n}^{\Omega}\overset{s}{\rightarrow}{O_n}^{\Omega}$ gegeben, so daß alle f_n kombinierbar *sind, also paarweise disjunkte Ausgabekanäle besitzen:*

$$\forall n,m{\in}N.\ n{\neq}m \Rightarrow O_n{\cap}O_m = \emptyset$$

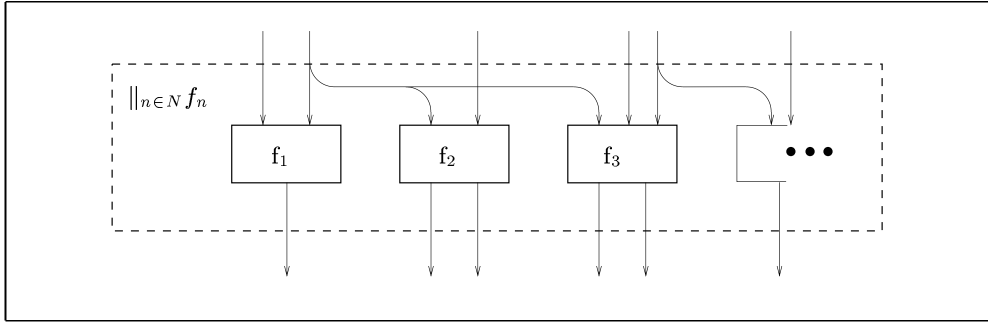


Abbildung A.3: Parallele Komposition $\|$

Seien $I \stackrel{def}{=} \bigcup_{n\in N} I_n$ *und* $O \stackrel{def}{=} \bigcup_{n\in N} O_n$. *Die parallele Komposition* $\|_{n\in N} f_n$ *wird wie folgt definiert:*

$$\|_{n\in N} f_n \in I^\Omega \stackrel{s}{\rightarrow} O^\Omega$$
$$\forall m\in N, si\in I^\Omega, o\in O_m.\ (\|_{n\in N} f_n)(si).o \stackrel{def}{=} f_m(si|_{I_m}).o \qquad (\Box)$$

Es wird hier weder gefordert, daß Eingabekanäle disjunkt sind ($j\neq k \Rightarrow I_j\cap I_k=\emptyset$), noch, daß Eingabe- und Ausgabekanäle disjunkt sind ($I_j\cap O_j=\emptyset$). Ersteres erlaubt die implizite Verdopplung von Strömen und damit sogar Broadcasting. Letzteres nutzen wir, um einen Rückkopplungsoperator μ zu definieren, der gleichbenannte Ein- und Ausgabekanäle verbindet. Bei der Rückkopplung reicht es aus, wie in Abbildung A.4 beschrieben, eine einzige Komponente zu betrachten.

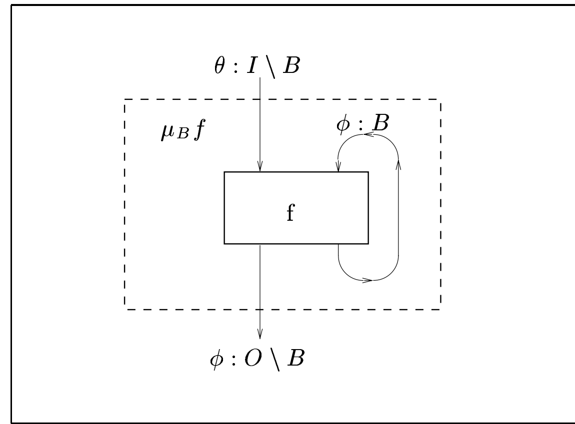


Abbildung A.4: Rückkopplungsoperator

Definition A.7 (Rückkopplung μ_B)
Seien eine stromverarbeitende Funktion $f\in I^\Omega \stackrel{s}{\rightarrow} O^\Omega$ *und die Menge* $B\subseteq I\cap O$ *gegeben. Dann definieren wir die Rückkopplung* $\mu_B f$ *der gleichbenannten Ausgabe- und Eingabekanäle* B *als kleinste Funktion, die folgender Gleichung genügt:*

$$\mu_B f \in (I\setminus B)^\Omega \stackrel{s}{\rightarrow} (O\setminus B)^\Omega$$
$$\forall \theta\in (I\setminus B)^\Omega.\ \exists \phi\in B^\Omega.\ (\mu_B f).\theta + \phi = f.(\theta+\phi) \qquad (\Box)$$

Diese in ϕ rekursive Gleichung besitzt im allgemeinen mehrere Lösungen für das Rückkopplungsbündel ϕ und charakterisiert daher $\mu_B f$ nicht eindeutig, weshalb die kleinste Lösung für $\mu_B f$ zu wählen ist.

Wir definieren die Rückkopplung aller Kanäle $\mu f \stackrel{def}{=} \mu_{I \cap O} f$. Mit der Komposition $(\mu\|)$ besitzen wir einen Kompositionsoperator für beliebige Systeme von Funktionen, unter dem sich die klassischen Operatoren aus [BRO93] genau so subsumieren, wie die in [GS95] und [DEN95] vorgeschlagenen.

Für jede Funktion $f \in I^\Omega \stackrel{s}{\to} O^\Omega$ gilt $\|\{f\}=f$ und $\mu_\emptyset f = f$. Beide Operatoren sind idempotent, das heißt für kombinierbare Mengen stromverarbeitender Funktionen F gilt die Identität $\|(\|F) = \|(F)$ und $\mu(\mu F) = \mu F$. Wir schreiben auch $f\|g$ statt $\|\{f,g\}$ und erhalten so einen kommutativen und assoziativen zweistelligen Operator für die Parallelkomposition. Diese Assoziativität läßt sich zum Glätten von Kompositionshierarchien verallgemeinern: Sei $(F_n)_{n \in N}$ ein System von Funktionsmengen und deren Vereinigung kompatibel, dann gilt

$$(\|\bigcup_{n \in N} F_n) = \|_{n \in N}(\|F_n),$$

und wenn die Eingabekanäle der einzelnen Funktionsmengen F_n paarweise disjunkt sind, gilt sogar

$$(\mu\|\bigcup_{n \in N} F_n) = \mu\|_{n \in N}(\mu\|F_n).$$

In Anlehnung an die sequentielle Komposition schreiben wir auch $f;g$ statt $\mu_H\|\{f,g\}$, wenn für $f \in I^\Omega \stackrel{s}{\to} H^\Omega$ und $g \in H^\Omega \stackrel{s}{\to} O^\Omega$ gilt $I \cap H = H \cap O = \emptyset$. $f;g$ entspricht damit der aus der Mathematik bekannten Funktionskomposition.

Wie üblich werden diese Formen der Komposition punktweise auf Prädikate über stromverarbeitenden Funktionen erweitert.

Verfeinerungstechniken

Neben der Komposition von Beschreibungen für Komponenten ist deren *Verfeinerung* in FOCUS wichtig. Es gibt folgende Arten von Verfeinerungen für Komponenten:

- *Verhaltensverfeinerung*,
- *strukturelle Dekomposition* und
- *Schnittstellenverfeinerung.*

Die *Verhaltensverfeinerung* einer Komponente wird als Einschränkung der möglichen Verhaltensweisen einer Komponente definiert, sei es durch Hinzunahme von Wissen während der Analysephase einer Softwareentwicklung oder durch Entwurfentscheidungen, die während der Entwicklungsphase getroffen werden. Seien P und P' Prädikate zur Beschreibung einer Komponente mit der gleichen Schnittstelle (I, O). P' ist eine *Verhaltensverfeinerung* von P genau dann, wenn $P' \subseteq P$.

Die *strukturelle Dekomposition* basiert auf den bereits eingeführten Kompositionsoperatoren $\|$ und μ. Bei der strukturellen Dekomposition wird eine bis dahin durch ein

Prädikat P charakterisierte Komponente konzeptuell oder physisch verteilt und die neu eingeführten Teilkomponenten entsprechend komponiert. Durch diese Komposition entsteht ein Prädikat P', das einer Verhaltensverfeinerung entspricht $P' \subseteq P$.

Die *Schnittstellenverfeinerung* einer Komponente beinhaltet einen Signaturwechsel. Es werden Hilfsprädikate in Form von *Repräsentationen* und *Abstraktionen* eingeführt, die es ermöglichen, die Schnittstellenkanäle in Beziehung zu setzen. Für eine systematische Behandlung siehe [BRO94]. Wir nutzen nur die Technik der *Abwärtssimulation*. Seien $P_a \subseteq I^\Omega \stackrel{s}{\rightarrow} O^\Omega$ und $P_c \subseteq {I_c}^\Omega \stackrel{s}{\rightarrow} {O_c}^\Omega$. Dann ist P_c eine Schnittstellenverfeinerung von P_a bezüglich Abwärtssimulation, wenn es zwei nichtleere *Repräsentationen* der Eingabe $R_i \subseteq I^\Omega \stackrel{s}{\rightarrow} {I_c}^\Omega$ und der Ausgabe $R_o \subseteq O^\Omega \stackrel{s}{\rightarrow} {O_c}^\Omega$ gibt, so daß gemäß der Abbildung A.5 gilt: $R_i;P_c \subseteq P_a;R_o$.

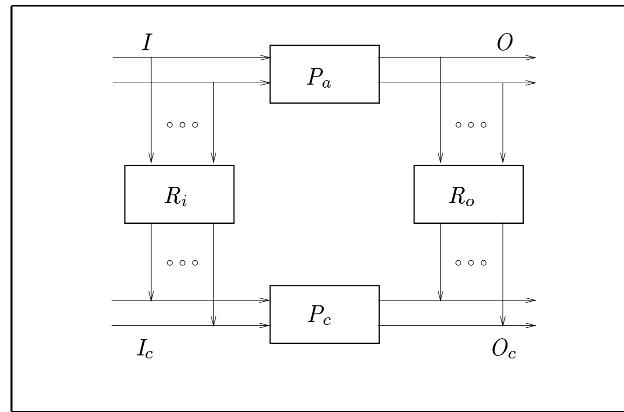


Abbildung A.5: Abwärtssimulation

Sicherheits- und Lebendigkeitseigenschaften

Eine *Sicherheitseigenschaft* fordert informell, daß unerwünschtes Verhalten während der Ausführung eines Programmes nicht auftreten darf, während eine *Lebendigkeitseigenschaft* fordert, daß gewünschtes Verhalten irgendwann auftreten muß [LAM77]. Dementsprechend können Sicherheitseigenschaften durch endliche Beobachtungen des Systems widerlegt werden, während Lebendigkeitseigenschaften durch unendliche Beobachtungen widerlegbar sind.

Wir definieren in Anlehnung an [DEN95] zunächst die Menge der Funktionen, die stets nur endliche Ausgaben liefern $\{f \in I^\Omega \stackrel{s}{\rightarrow} O^\Omega \mid \forall x \in M^\omega, b.\#f.x.b < \infty\}$ und bezeichnen diese als $I^\Omega \stackrel{s}{\rightarrow} {O^\Omega}_{fin}$. Ein Prädikat $P \subseteq I^\Omega \stackrel{s}{\rightarrow} O^\Omega$ ist genau dann eine *Sicherheitseigenschaft*, wenn gilt:

$$\forall f \in I^\Omega \stackrel{s}{\rightarrow} O^\Omega.\ P.f \Leftrightarrow (\forall g \in I^\Omega \stackrel{s}{\rightarrow} {O^\Omega}_{fin}.\ g \sqsubseteq f \Rightarrow P.g)$$

Ein Prädikat $P \subseteq I^\Omega \stackrel{s}{\rightarrow} O^\Omega$ ist genau dann eine *Lebendigkeitseigenschaft*, wenn gilt:

$$\forall f \in I^\Omega \stackrel{s}{\rightarrow} {O^\Omega}_{fin}.\ \exists g \in I^\Omega \stackrel{s}{\rightarrow} O^\Omega.\ f \sqsubseteq g \wedge P.g$$

A.3 Gezeitete Modellierungen

Die im ersten Teil eingeführten FOCUS Grundlagen werden jetzt auf den gezeiteten Modellierungstil spezialisiert. Eine gezeitete Modellierung nutzt neben den Zeichen der Menge M ein spezielles Zeichen $\surd$, genannt *Tick*, um damit Voranschreiten von Zeit zu modellieren. Damit eine stromverarbeitende Funktion ein korrektes Fortschreiten der Zeit modelliert, werden einige Einschränkungen gestellt.

Diese Modellierungstechnik geht davon aus, daß in jeder Komponente und auch bei Beobachtern von Kanälen genügend genaue Uhren zur Verfügung stehen, die Zeiteinheiten messen. Mit Hilfe solcher Uhren steht ein Mechanismus zur Verfügung, die ursprünglich asynchrone Kommunikation nun *zeitsynchron*, also bis auf die kleinste beobachtete Zeiteinheit exakt zu beschreiben. Das Auftreten eines Ticks in einem Strom modelliert das Voranschreiten der Zeit um eine Zeiteinheit. Der n-te Tick eines jeden Stroms im System markiert den Zeitpunkt n (nach Beginn des Systemlaufs). Von den Nachrichten zwischen dem n-ten und $(n{+}1)$-ten Tick ist nur deren Reihenfolge bekannt, und daß sie im Zeitraum $[n,n{+}1]$ aufgetreten sind.

Die klassische FOCUS-Theorie kann durch die Aufnahme von Zeitticks spezialisiert werden. Technisch können damit die Probleme der Mehrdeutigkeit der Rückkopplung von Kanälen mit dem μ-Operator gelöst und die Definition des fairen Mischens von Kanälen durch gezeitete Funktionen beschrieben werden.

Gezeitete Ströme

Ein gezeiteter Strom beschreibt die zeitlich unbegrenzte Beobachtung eines Kanals, beginnend mit einem Startzeitpunkt. Wir modellieren das Voranschreiten einer Zeiteinheit im Kanal durch das Auftreten eines Ticks $\surd$. Deshalb sind in jeder vollständigen Beobachtung notwendigerweise unendlich viele Ticks enthalten. Für technische Zwecke definieren wir weiterhin die Menge der zeitlich endlichen Beobachtungen, die bis zum Ende einer Zeiteinheit gemacht werden. Wir lassen im folgenden offen, ob $\surd \in M$ gilt. Dadurch können nachstehend definierte Operatoren sowohl auf die Menge der eigentlichen Zeichen $M \setminus \{\surd\}$ als auch auf die Erweiterung $M \cup \{\surd\}$ angewendet werden.

Definition A.8 (Gezeitete Ströme)
Sei M eine Menge von Zeichen *und* $\surd$ *ebenfalls ein Zeichen. Wir definieren die Mengen*

$M^{\overline{*}}$ *die Menge aller* endlichen, gezeiteten Ströme *über M:*

$$M^{\overline{*}} \stackrel{def}{=} \{\ s \in (M \cup \{\surd\})^{*} \mid s = \varepsilon \vee \exists t.\ s = t\,\hat{}\,\surd \}$$

$M^{\overline{\infty}}$ *die Menge aller* unendlichen, gezeiteten Ströme *über M:*

$$M^{\overline{\infty}} \stackrel{def}{=} \{\ s \in (M \cup \{\surd\})^{\infty} \mid \#_{\surd} s = \infty\ \}$$

$M^{\overline{\omega}} \stackrel{def}{=} M^{\overline{*}} \cup M^{\overline{\infty}}$ *die Menge aller* gezeiteten Ströme *über M.*

Neben der Länge eines Stroms nutzen wir dessen Beobachtungsdauer $\#_{\surd}$*, die zeitliche Dauer der Beobachtung des Stroms:*

$s{\downarrow}n$ *ist der Präfix von s mit Beobachtungsdauer* $n{\in}\mathbb{N}{\cup}\{\infty\}$

$\diamond s$ *ist der ungezeitete Anteil eines Stroms.*

Es gilt:

$$s{\downarrow}n = \begin{cases} s & \texttt{if} \quad \#_{\surd}s{\leq}n \\ t & \texttt{if} \quad \#_{\surd}t{=}n \wedge t{\sqsubseteq}s \end{cases}$$
$$\diamond s = M\backslash\{\surd\} \copyright\, s \qquad (\square)$$

Wir benutzen dieselbe Approximationsordnung auf gezeiteten Strömen, wie sie uns auf ungezeiteten Strömen zur Verfügung steht, und erhalten damit einen CPO $(M^{\overline{\omega}},\sqsubseteq)$, der in dem CPO $((M{\cup}\{\surd\})^{\omega},\sqsubseteq)$ enthalten ist. Weil endliche Ströme immer mit einem $\surd$ abschließen, wird verhindert, daß echt aufsteigende Ketten existieren, deren Supremum nur endlich viele Ticks enthalten würde, und so nicht in $M^{\overline{\omega}}$ enthalten wäre. Die Konkatenation ist in $M^{\overline{\omega}}$ abgeschlossen, und für zwei Ströme $s{\in}M^{\overline{\omega}}$, $t{\in}M^{\overline{*}}$ mit $t{\sqsubseteq}s$ existiert ein eindeutiges $u{\in}M^{\overline{\omega}}$ mit $t\,\hat{}\,u = s$. Auch sind kleinste obere Schranken und damit auch kleinste Fixpunkte rekursiver Definitionen in beiden CPO's identisch.

Gezeitete Strombündel und Stromtupel

Analog zur ungezeiteten Version lassen sich jetzt gezeitete Varianten von Strombündeln und Tupeln definieren. Die einzelnen Ströme eines Bündels können zwar verschiedene Längen besitzen, haben jedoch eine einheitliche Beobachtungsdauer.

Definition A.9 (Gezeitete Strombündel)
Sei B eine beliebige Menge von Kanalnamen *und* $(M_b)_{b\in B}$ *ein indiziertes System von Nachrichtenmengen. Sei* $M = \{\surd\}{\cup}\bigcup_{b\in B}M_b$*, dann wird die Menge der* gezeiteten Strombündel $B^{\overline{\Omega}}$ *wie folgt definiert:*

$$B^{\overline{\Omega}} \stackrel{def}{=} \{\ s{\in}\ B{\rightarrow}M^{\overline{\omega}} \mid (\forall b{\in}B.\ s.b \in M_b{}^{\overline{\omega}}) \wedge \exists n{\in}\mathbb{N}{\cup}\{\infty\}.\ \forall b{\in}B.\ \#_{\surd}s.b{=}n\ \}$$

Wir definieren die Menge der unendlichen und der endlichen gezeiteten Strombündel:

$$B^{\overline{\Sigma}} \stackrel{def}{=} \{\ s{\in}\ B^{\overline{\Omega}} \mid \#_{\surd}s{=}\infty\ \}$$
$$B^{\overline{\Phi}} \stackrel{def}{=} \{\ s{\in}\ B^{\overline{\Omega}} \mid \#_{\surd}s{<}\infty\ \} \qquad (\square)$$

$(B^{\overline{\Omega}},\sqsubseteq)$ bildet einen CPO, der, wenn $\surd{\in}M_b$ für alle $b{\in}B$, homomorph in den CPO $(B^{\Omega},\sqsubseteq)$ eingebettet ist. Wieder stimmen also kleinste obere Schranken und kleinste Fixpunkte überein. Wir erweitern $s{\downarrow}n$ und $\diamond s$ punktweise auf Bündel.

Gezeitete stromverarbeitende Funktionen

Sei für die nächsten Abschnitte $\surd \in M_b$ für alle Kanaltypen M_b. Wir konstruieren die Menge der *gezeiteten Funktionen* als Restriktion einer Teilmenge der stromverarbeitenden Funktionen auf unendlichen Strömen, und können so die bisherige Theorie, inklusive Kompositionstechniken und Verfeinerungstechniken weiter nutzen. Die Charakterisierung gezeiteter Funktionen entspricht genau der in [STO95] definierten „pulse-drivenness“ und ersetzt die bisher geforderten Eigenschaften Monotonie und Stetigkeit von stromverarbeitenden Funktionen. Sie sind nur auf unendlichen und damit kompletten Eingaben definiert und ergeben unendliche Ausgaben.

Definition A.10 (Gezeitete stromverarbeitende Funktionen)
Eine Funktion $f \in I^{\overline{\Sigma}} \to O^{\overline{\Sigma}}$ heißt gezeitete stromverarbeitende Funktion*, wenn die Ausgabe der ersten (n+1) Takte nur von der Eingabe der ersten n Takte abhängt.*

$$\forall r,s \in I^{\overline{\Sigma}}, n \in \mathbb{N}.\ r \downarrow n = s \downarrow n \Rightarrow f(r) \downarrow (n+1) = f(s) \downarrow (n+1)$$

Die Menge der gezeiteten Funktionen bezeichnen wir mit $I^{\overline{\Sigma}} \xrightarrow{p} O^{\overline{\Sigma}}$. (□)

Die Verwendung gezeiteter Funktionen hat den Vorteil, daß bei der Rückkopplung immer genau ein Fixpunkt existiert, jedoch den Nachteil, daß jede Modellierung mit diesen Funktionen eine Verzögerung impliziert. Eine Funktion kann aufgrund obiger Einschränkung immer erst mit einer Verzögerung von mindestens einer Zeiteinheit auf eine Eingabe reagieren.

Schwache gezeitete Funktionen

Zur Definition gezeiteter Funktionen verwenden wir oft Hilfsfunktionen, die selbst jedoch nicht gezeitet sind, für die aber eine schwächere Bedingung existiert.

Definition A.11 (Schwach gezeitete Funktionen)
Eine Funktion $f \in I^{\overline{\Sigma}} \to O^{\overline{\Sigma}}$ heißt schwach gezeitete Funktion*, wenn die Ausgabe der ersten n Takte nur von der Eingabe der ersten n Takte abhängt.*

$$\forall r,s \in I^{\overline{\Sigma}}, n \in \mathbb{N}.\ r \downarrow n = s \downarrow n \Rightarrow f(r) \downarrow n = f(s) \downarrow n$$

Die Menge der schwach gezeiteten Funktionen bezeichnen wir mit $I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}}$. (□)

Zur Abgrenzung nennen wir die eigentlichen gezeiteten Funktionen manchmal *stark*. Eine schwache Funktion g kann genutzt werden, um eine starke Funktion f zu definieren, z.B. durch $f(x) = initialstrom \hat{}\surd\hat{}g(x)$.

Gezeitete Komposition

Mit dem Übergang von ungezeiteten zu gezeiteten Funktionen haben wir die beiden Kompositionsoperatoren $\|$ und μ_B ebenfalls auf gezeitete Funktionen und Prädikate übertragen. Wie bereits erwähnt, sind Fixpunkte über starke Funktionen, die bei der Rückkopplung benötigt werden, nun eindeutig definiert.

Anhang B

Beweisschemata

In diesem Kapitel sind Beweisschemata definiert und als korrekt bewiesen, die spezifisch auf den Beweis von Eigenschaften der denotationellen Semantik zugeschnitten sind. Diese sind aufgrund der Definition mittels einer rekursiven Menge stromverarbeitender Funktionen relativ komplex.

Die denotationelle Semantik wird in 5.4 definiert. Die dort verwendete rekursive Mengendefinition erhält ihre Semantik als (mengentheoretisch) größter Fixpunkt des in Proposition 5.5 definierten Funktionals σ der Form:

(B.1) $\sigma : \mathbb{P}(S \times M^{\omega} \stackrel{s}{\rightarrow} M^{\omega}) \rightarrow \mathbb{P}(S \times M^{\omega} \stackrel{s}{\rightarrow} M^{\omega})$

(B.2) $\sigma(X) \stackrel{def}{=} \{\, h \in S \times M^{\omega} \stackrel{s}{\rightarrow} M^{\omega} \mid Q.h.X \,\}$

(B.3) `where`

(B.4) $\quad Q.h.X \stackrel{def}{\Leftrightarrow} (\forall s.\ h(s,\varepsilon) = \varepsilon) \wedge$

(B.5) $\quad\quad \forall m \in M, s. \exists t, out; h' \in X.\ \delta(s,m,t,out) \wedge \forall in \in M^{\omega}.\ h(s, m\,\hat{}\,in) = out\,\hat{}\,h'(t,in)$

Das Funktional σ ist monoton im CPO $(\mathbb{P}(S \times M^{\omega} \stackrel{s}{\rightarrow} M^{\omega}), \supseteq)$, aber nicht stetig. Deshalb kann der Kleene'sche Fixpunktsatz ([KLE52]) nicht verwendet werden. Im folgenden sind daher einige Beweisschemata definiert, die bei Beweisen über Eigenschaften der denotationellen Semantik Unterstützung bieten. Dabei wird eine Definition der Zulässigkeit unter Benutzung beliebiger Ketten verwendet.

B.1 Beweisschema *I*

Ist eine Eigenschaft P *zulässig*, so läßt sich die Ordinalzahl-Induktion nutzen, um P für den größten Fixpunkt des Funktionals zu zeigen. Das Beweisschema *(I)* ist in Tabelle B.1 formuliert und folgt direkt aus Proposition 35 von ([CC92]). Die Ordinalzahlinduktion tritt in Schema *(I)* nicht explizit auf, stattdessen ist sie als Scott's Regel zur schrittweisen Berechnungsinduktion in *(I.d)* kodiert.

(a)	$\sigma\colon \mathbb{P}(\top) \to \mathbb{P}(\top)$, monoton
(b)	$P \subseteq \mathbb{P}(\top)$, $adm(P)$
(c)	$P(\top)$
(d)	$\forall X.\ P(X) \Rightarrow P(\sigma.X)$
	$P(fix.\sigma)$

Abbildung B.1: Beweisschema *(I)* für zulässige Eigenschaften P

B.2 Beweisschema A

Beim Beweis der Korrektheit einer Verfeinerungsregel sind im allgemeinen die Fixpunkte zweier Funktionale σ und σ' zu vergleichen. Wir definieren dafür weitere Beweisschemata *(A)*, *(BP)*, *(B)* und *(C)*. Einige davon bestehen aus Teilschemata für je eine Inklusionsrichtung. Schema *(A)* ist in Tabelle B.2 dargestellt. Es basiert auf einer Funktion α, die mit σ bzw. σ' kommutiert.

(a)	$\sigma\colon \mathbb{P}(\top) \to \mathbb{P}(\top)$, monoton
(b)	$\sigma'\colon \mathbb{P}(\top') \to \mathbb{P}(\top')$, monoton
(c)	$\alpha\colon \mathbb{P}(\top) \to \mathbb{P}(\top')$
(d)	$\forall X.\ \alpha(\sigma.X) = \sigma'(\alpha.X)$
(e)	$\alpha(\top) = \top'$
(f)	$P(X) = (\alpha.X \supseteq K),\ \ adm(P)$
	$\alpha(fix.\sigma) = (fix.\sigma')$

Abbildung B.2: Beweisschema *(A)* für Fixpunktvergleiche

Proposition B.1 (Korrektheit von Beweisschema *(A)*)
Das in Tabelle B.2 dargestellte Beweisschema (A) ist korrekt. $(\Box)$

Beweis B.2
Inklusion $\subseteq$: Es gilt:

(B.6) $\alpha(fix.\sigma) = \alpha(\sigma(fix.\sigma)) \overset{(d)}{=} \sigma'(\alpha(fix.\sigma))$.

Damit ist $\alpha(fix.\sigma)$ ein Fixpunkt von σ' und es folgt die Inklusion:

(B.7) $\alpha(fix.\sigma) \subseteq (fix.\sigma')$.

Inklusion $\supseteq$: Dazu verwenden wir Beweisschema *(I)* mit folgender Eigenschaft:

(B.8) $P(X) \overset{def}{\Leftrightarrow} \alpha.X \supseteq (fix.\sigma')$.

Wegen *(f)* ist P zulässig. Wegen *(e)* gilt

(B.9) $P(\top)$.

Gilt $P(X)$, so folgt

(B.10) $\alpha(\sigma.X) \overset{(d)}{=} \sigma'(\alpha.X)$

und wegen *(b)*, der Monotonie von σ':

(B.11) $\supseteq \sigma'(fix.\sigma') = fix.\sigma'$

Also $P(\sigma.X)$. Mit *(I)* folgt nun die Behauptung $P(fix.\sigma)$. (□)

B.3 Beweisschema *BP*

Das Schema *(BP)* besteht aus zwei Teilschemata, die beide in Tabelle B.3 dargestellt sind. Punkt *(BP.d)* fordert, daß Funktion α in gewisser Weise mit σ bzw. σ' kommutiert. Für Teilschema $(BP_{\supseteq})$ muß ein passendes β gefunden werden, das unter der Einschränkung P zu α invers ist. Die Eigenschaft P muß ihrerseits invariant unter σ' sein.

(a) $\sigma: \mathbb{P}(\top) \to \mathbb{P}(\top)$, monoton

(b) $\sigma': \mathbb{P}(\top') \to \mathbb{P}(\top')$, monoton

(c) $\alpha: \mathbb{P}(\top) \to \mathbb{P}(\top')$

Teilschema $(BP_{\subseteq})$:

$$(d)\quad \frac{\forall X.\ \alpha(\sigma.X) \subseteq \sigma'(\alpha.X)}{\alpha(fix.\sigma) \subseteq (fix.\sigma')}$$

Teilschema $(BP_{\supseteq})$:

(e) $\beta : \mathbb{P}(\top') \to \mathbb{P}(\top)$

(f) α monoton

(g) $P \subseteq \mathbb{P}(\top')$, $adm(P)$

(h) $P(\top')$

(i) $\forall Y.\ P(Y) \Rightarrow P(\sigma'.Y)$

(j) $\forall Y.\ P(Y) \Rightarrow Y \subseteq \alpha(\beta.Y)$

(k) $\forall Y.\ P(Y) \Rightarrow \beta(\sigma'.Y) \subseteq \sigma(\beta.Y)$

$$\overline{\alpha(fix.\sigma) \supseteq (fix.\sigma')}$$

$$\overline{\alpha(fix.\sigma) = (fix.\sigma')}$$

Abbildung B.3: Beweisschema *(BP)* für Fixpunktvergleiche

Proposition B.3 (Korrektheit von Beweisschema *(BP)*)
Das in Tabelle B.3 dargestellte Beweisschema (BP) ist korrekt. (□)

Beweis B.4
Teilschema $(BP_{\subseteq})$: Es gilt:

(B.12) $\alpha(fix.\sigma) = \alpha(\sigma(fix.\sigma)) \overset{(d)}{\subseteq} \sigma'(\alpha(fix.\sigma))$.

Damit ist $\alpha(fix.\sigma)$ ein Präfixpunkt von σ' und es folgt die Behauptung.

Teilschema $(BP_{\supseteq})$: Wegen (g), (h) und (i) gilt nach Beweisschema (I):

(B.13) $P(fix.\sigma')$.

Damit folgt analog zu (B.12):

(B.14) $\beta(fix.\sigma') = \beta(\sigma'(fix.\sigma')) \overset{(k)}{\subseteq} \sigma(\beta(fix.\sigma'))$.

Damit ist $\beta(fix.\sigma')$ Präfixpunkt von σ und es gilt:

(B.15) $\beta(fix.\sigma') \subseteq (fix.\sigma)$.

Damit folgt:

(B.16) $(fix.\sigma') \overset{(j)}{\subseteq} \alpha(\beta(fix.\sigma')) \overset{(f)}{\subseteq} \alpha(fix.\sigma)$. $(\Box)$

B.4 Beweisschema B

Beweisschema B ist eine Spezialisierung von BP.

(a) $\sigma: \mathbb{P}(\top) \rightarrow \mathbb{P}(\top)$, monoton

(b) $\sigma': \mathbb{P}(\top') \rightarrow \mathbb{P}(\top')$, monoton

(c) $\alpha: \mathbb{P}(\top) \rightarrow \mathbb{P}(\top')$

Teilschema $(B_{\subseteq})$:

(d) $\dfrac{\forall X.\ \alpha(\sigma.X) \subseteq \sigma'(\alpha.X)}{\alpha(fix.\sigma) \subseteq (fix.\sigma')}$

Teilschema $(B_{\supseteq})$:

(e) $\beta : \mathbb{P}(\top') \rightarrow \mathbb{P}(\top)$

(f) α monoton

(g) $\forall Y.\ Y \subseteq \alpha(\beta.Y)$

(h) $\forall Y.\ \beta(\sigma'.Y) \subseteq \sigma(\beta.Y)$

$\overline{\alpha(fix.\sigma) \supseteq (fix.\sigma')}$

$\overline{\alpha(fix.\sigma) = (fix.\sigma')}$

Abbildung B.4: Beweisschema (B) für Fixpunktvergleiche

Proposition B.5 (Korrektheit von Beweisschema (B))
Das in Tabelle B.4 dargestellte Beweisschema (B) ist korrekt. $(\Box)$

Beweis B.6
Folgt sofort aus Beweisschema (BP) mit $P{=}True$. $(\Box)$

B.5 Beweisschema C

Aufgrund der speziellen Struktur der monotonen Funktionen σ aus der Definition (B.1-B.5) und der Tatsache, daß die Verbindung zwischen beiden Funktionsräumen oft mittels einer punktweise auf Mengen erweiterten Funktion α:$\top\to\top'$ erfolgt, können wir Beweisschema *(B)* spezialisieren zu Beweisschema *(C)* das seinerseits aus zwei Beweisschemata $(C_\subseteq)$ und $(C_\supseteq)$ aufgebaut ist, die die beiden Inklusionen beweisen.

(a) $\sigma(X) \stackrel{def}{=} \{f \mid Q.f.X\}$

(b) $\sigma'(Y) \stackrel{def}{=} \{g \mid Q'.g.Y\}$

(c) $\alpha\colon \top \to \top'$

(d) $\dfrac{\forall f\in\top, X\subseteq\top.\ Q.f.X \Rightarrow Q'.(\alpha.f).(\alpha.X)}{\alpha(fix.\sigma) \subseteq (fix.\sigma')}$

Teilschema $(C_\subseteq)$

(e) $\alpha(\top) = \top'$

(f) $\dfrac{\forall f\in\top, X\subseteq\top.\ Q'.(\alpha.f).(\alpha.X) \Rightarrow Q.f.X}{\alpha(fix.\sigma) \supseteq (fix.\sigma')}$

Teilschema $(C_\supseteq)$

$$\overline{\alpha(fix.\sigma) = (fix.\sigma')}$$

Abbildung B.5: Beweisschema *(C)* für Fixpunktvergleiche

Proposition B.7 (Korrektheit von Beweisschema *(C)*)
Das in Tabelle B.5 dargestellte Beweisschema (C) ist korrekt für die Definition von Q und Q' in der Form (B.1-B.5). (□)

Beweis B.8
Wir zeigen die Korrektheit von Beweisschema *(C)* unter Rückführung auf *(B)*. Es mögen die Teilaussagen *(C.a)* bis *(C.c)* aus Tabelle B.5 gelten. Aufgrund der Definition von σ und σ' folgen sofort *(B.a)* und *(B.b)*. Die Funktion α wird punktweise auf Mengen erweitert und erhält so Signatur *(B.c)*.

Teilschema $(C_\subseteq)$: Wir nehmen nun zusätzlich die Teilaussage *(C.d)* an. Es bleibt *(B.d)* zu zeigen. Dazu sei $X\subseteq\top$ und

(B.17) $\quad f \in \alpha(\sigma.X)$.

Wegen *(C.a)* gilt dann:
(B.18) $f \in \{ \alpha.g \mid Q.g.X \}$.
Aus *(C.d)* folgt:
(B.19) $f \in \{ \alpha.g \mid Q'.(\alpha.g).(\alpha.X) \}$.
Umgeformt mit $f{=}\alpha.g$ folgt:
(B.20) $Q'.f.(\alpha.X)$.
Mit *(C.b)* folgt nun:
(B.21) $f \in \sigma'(\alpha.X)$.
Die Implikationskette (B.17) bis (B.21) liefert Aussage *(B.d)*, und mit Proposition B.5 folgt die Korrektheit von Teilschema $(C_{\subseteq})$.

Teilschema $(C_{\supseteq})$: Statt *(C.d)* mögen nun die Teilaussagen *(C.e)* und *(C.f)* gelten. Wir definieren eine Funktion β wie folgt:
(B.22) $\beta\colon \mathbb{P}(\top') \rightarrow \mathbb{P}(\top)$
(B.23) $\beta(Y) \overset{def}{=} \{ f \mid \alpha.f \in Y \}$
und erfüllen damit Anforderung *(B.e)*. Auch *(B.f)* gilt, da α als punktweise definierte Funktion auf $\mathbb{P}(\top)$ monoton ist.

Nach Definition von β und wegen der Surjektivität von α *(C.e)* ist für alle $Y \subseteq \top'$
(B.24) $\alpha(\beta(Y)) = \{ \alpha.f \mid \alpha.f \in Y \} = Y$,
weshalb *(B.g)* erfüllt ist.

Bleibt *(B.h)* zu zeigen. Dazu sei $Y \subseteq \top'$ und
(B.25) $f \in \beta(\sigma'.Y)$.
Nach Definition von β und wegen *(C.b)* folgt daraus:
(B.26) $\alpha.f \in \{ g \mid Q'.g.Y \}$
oder umgeformt:
(B.27) $Q'.(\alpha.f).Y$.
Wegen (B.24) ist dies identisch zu
(B.28) $Q'.(\alpha.f).(\alpha(\beta.Y))$.
Mit *(C.f)* folgt nun:
(B.29) $Q.f.(\beta.Y)$
und mit *(C.a)*:
(B.30) $f \in \sigma(\beta.Y)$.
Die Implikationskette (B.25) bis (B.30) liefert Aussage *(B.h)*, und mit Proposition B.5 folgt die Korrektheit von Teilschema $(C_{\supseteq})$. $(\Box)$

Anhang C

Beweise

Dieses Kapitel des Anhangs enthält Beweise gegliedert nach den Kapiteln der zugehörigen Propositionen. Einzelne, technische Definitionen und Propositionen sind ebenfalls hier aufgenommen worden.

Für komplexere Beweise wird zunächst die Beweisidee skizziert. Einige Beweise sind in Teilbeweise untergliedert, deren Teilziel jeweils zu Beginn angegeben ist. Teilziele sind unterstrichen.

C.1 Systementwicklung

Beweis C.1 zu 2.7, Seite 19: (Redundante Dokumente im Dokumentgraphen)
Folgt durch Induktion über die Graphstruktur aus den Definitionen 2.5, 2.4, 2.6 und der Charakterisierung der Dokumentstruktur (d_{cons}, d_{prot}). (□)

C.2 Darstellung buchstabierender Automaten

Beweis C.2 zu 4.9, Seite 64: (Eigenschaften von $\omega{-}reach$)
Die erste Aussage folgt sofort aus den Definitionen 4.7 und 4.8 von *reach* und $\omega{-}reach$. Die zweite Aussage läßt sich aus der Definition 4.6 von δ^ω und der Definition 4.5 der Transitionssequenzen folgern. (□)

C.3 Semantik für Automaten

Beweis C.3 zu 5.3, Seite 67: (Eindeutigkeit von $[\![.]\!]^d$)
Beweisidee: Wir zeigen, daß es genau ein Element $h \in S \times M^\omega \overset{s}{\rightarrow} M^\omega$ gibt, das der rekursiven Gleichung aus Definition 5.2 genügt. Wir definieren dazu das Funktional τ, das

umformuliert aus der rekursiven Gleichung für h entsteht[1]:

(C.1) $\tau : (S\times M^\omega \stackrel{s}{\rightarrow} M^\omega) \rightarrow (S\times M^\omega \stackrel{s}{\rightarrow} M^\omega)$

(C.2) $\tau(f)(s,in') \stackrel{def}{=} \begin{cases} \varepsilon & \texttt{if } in'=\varepsilon \\ out\hat{\ }f(t,rt(in')) & \texttt{if } in'\neq\varepsilon \wedge (t,out) = \delta(s,ft(in')) \end{cases}$

Wir zeigen im folgenden, daß τ wohldefiniert, monoton und stetig ist. Damit läßt sich nach Kleene ([KLE52]) h als kleinster Fixpunkt von τ festlegen. Dieser ist sogar der einzige Fixpunkt von τ. Weil mit h auch g in Definition 5.2 eindeutig ist, gilt damit die Behauptung. Wir nutzen dabei statt dem CPO $(S^\perp\times M^\omega \stackrel{s}{\rightarrow} M^\omega)$ die Teilmenge $(S\times M^\omega \stackrel{s}{\rightarrow} M^\omega)$, da dies für unsere Zwecke ausreicht, und eine strikte Einbettung mit $f(\perp,s)=\perp$ vorgenommen werden kann.

<u>τ wohldefiniert</u>: Sei f stetig, $s\in S$ und $k=(k_i)_i\in chain(M^\omega)$ eine Kette. Ist $\sqcup k=\varepsilon$, so gilt trivialerweise

(C.3) $\sqcup(\tau(f)(s,k)) = \varepsilon = \tau(f)(s, \sqcup k)$,

Andernfalls kann o.E. angenommen werden, daß $k_i\neq\varepsilon\forall i$. Aufgrund der Monotonieeigenschaft einer Kette und der flachen Ordnung auf M gibt es $m\in M$ und $q_i\in M^\omega$, so daß

(C.4) $\forall i.\ k_i = m\hat{\ }q_i$.

Dann ist $q = (q_i)$ wieder eine Kette und es gilt

(C.5) $\sqcup k = m\hat{\ }(\sqcup q)$.

Seien t und out wie in (C.2) definiert, dann gilt nach (C.2) und (C.4):

(C.6) $\tau(f)(s,k_i) = out\hat{\ }f(t,q_i)$.

Es gilt damit:

(C.7) $\sqcup(\tau(f)(s,k)) = \sqcup(out\hat{\ }f(t,q))$

und aufgrund (C.2), (C.5) und der Eigenschaften von $\sqcup$:

(C.8) $\quad = out\hat{\ }f(t,\sqcup q) = \tau(f)(s, m\hat{\ }\sqcup q) = \tau(f)(s, \sqcup k)$.

Nach (C.3) und (C.8) ist $\tau(f)$ stetig und τ wohldefiniert.

<u>τ monoton</u>: Seien $f\sqsubseteq g$ stetige stromverarbeitende Funktionen und $s\in S$, dann gilt für ε:

(C.9) $\tau(f)(s,\varepsilon) = \varepsilon \sqsubseteq \tau(g)(s,\varepsilon)$

und für $in\in M^\omega$:

(C.10) $\tau(f)(s,m\hat{\ }in) = out\hat{\ }f(t,in)$,

wobei wieder t und out wie in (C.2) definiert sind. Nach Voraussetzung ist $f\sqsubseteq g$ und damit

(C.11) $\quad \sqsubseteq out\hat{\ }g(t,in) = \tau(g)(s,m\hat{\ }in)$.

Also ist τ monoton. Gemeinsam mit der Wohldefiniertheit von τ folgt die Existenz von $(fix.\tau)$ als stetige Funktion.

<u>τ stetig</u>: Sei $K\in chain(S\times M^\omega \stackrel{s}{\rightarrow} M^\omega)$ eine Kette stromverarbeitender Funktionen. Da τ monoton ist, ist $\tau(K)$ ebenfalls eine Kette und es gibt den Fixpunkt $\sqcup(\tau\ K)$. Es ist zu zeigen

(C.12) $\tau(\sqcup K) = \sqcup(\tau\ K)$,

Sei wieder $s\in S$ sowie t und out wie in (C.2) definiert. Es gilt:

(C.13) $\tau(\sqcup K)(s,\varepsilon) = \varepsilon = (\sqcup(\tau\ K))(s,\varepsilon)$.

Für $m\in M$, $in\in M^\omega$ gilt nach (C.2):

(C.14) $\tau(\sqcup K)(s,m\hat{\ }in) = out\hat{\ }(\sqcup K)(t,in)$.

[1] ft und rt sind Funktionen, die das erste Element bzw. den Rest eines Stroms selektieren

Aufgrund der Eigenschaften von $\sqcup$ auf stetigen Funktionsräumen läßt sich dies umformen zu:

(C.15) $\quad = \; out\hat{}(\sqcup(K(t,in)) = \sqcup(out\hat{}K(t,in))$.

(C.2) auf die Elemente der Kette angewandt und umgeformt:

(C.16) $\quad = \sqcup((\tau\ K)(s,m\hat{}in)) = \sqcup(\tau\ K)(s,m\hat{}in)$.

Mit (C.13) und (C.14-C.16) folgt die Stetigkeit von τ.

τ hat genau einen Fixpunkt: Sei $h \in S \times M^\omega \xrightarrow{s} M^\omega$ ein Fixpunkt von τ. Wir zeigen $h = (fix.\tau)$ mittels folgender Behauptung, die mit Induktion über n bewiesen wird:

(C.17) $\forall n \in \mathbb{N}.\ \forall s,in.\ \#in \leq n \Rightarrow h(s,in) = \tau^{n+1}(\bot)(s,in)$.

Für beliebiges $n \in \mathbb{N}$ (insbesondere Induktionsanfang $n=0$) und $in=\varepsilon$ ist

(C.18) $\tau^{n+1}(\bot)(s,\varepsilon) = \varepsilon = \tau(h)(s,\varepsilon) = h(s,\varepsilon)$.

Der Induktionsschritt nach $n+1$ nutzt $in = m\hat{}in'$, da $in=\varepsilon$ bereits in (C.18) behandelt wurde. Es gilt nach (C.2):

(C.19) $\tau^{n+2}(\bot)(s,in) \; = out\hat{}(\tau^{n+1}(\bot))(t,in')$

mit $(t,out)=\delta(s,in)$. Die Induktionsvoraussetzung wird für n und (C.2) rückwärts eingesetzt:

(C.20) $\quad \stackrel{IV}{=} out\hat{}h(t,in') = h(s,in)$.

Damit gilt (C.17), und es läßt sich folgern, daß für endliche $in \in M^*$:

(C.21) $(fix.\tau)(s,in) = \sqcup_{m \in \mathbb{N}}(\tau^m(\bot))(s,in) = h(s,in)$.

Weil h und $(fix.\tau)$ stetig sind, gilt (C.21) auch für $in \in M^\infty$, und daher insgesamt: $h = (fix.\tau)$, was zu beweisen war. $(\Box)$

Beweis C.4 zu 5.7, Seite 71: ($[\![.]\!]^c$ ist kompatibel mit $[\![.]\!]^d$)

$[\![.]\!]^c \subseteq [\![.]\!]^d$: Sei (S,M,δ,I) ein deterministischer und totaler Automat. Weil (S,M,δ,I) deterministisch ist, kann das Mengenfunktional σ aus Proposition 5.5,(5.1) unter Benutzung des Funktionals τ aus Beweis C.3(C.1) umgeformt werden zu

(C.22) $\sigma(X) = \{\tau(h) \mid h \in X\} = \tau(X)$.

Sei $h \stackrel{def}{=} (fix.\tau)$, dann ist $\{h\}=\tau\{h\}$ Fixpunkt von σ und da $(fix.\sigma)$ größtes Element unter den Fixpunkten ist, gilt also, $\{h\} \subseteq (fix.\sigma)$. Daraus folgt:

(C.23) $[\![A]\!]^c \subseteq [\![A]\!]^d$.

$[\![.]\!]^d \subseteq [\![.]\!]^c$: Wir zeigen nun induktiv über die Länge n der Eingabe folgenden Satz:

(C.24) $\forall n \in \mathbb{N}\ \forall g \in (fix.\sigma)\ \forall s \in S, in \in M^n.\ g(s,in) = h(s,in)$.

Der Beweis verläuft analog zu dem bereits in C.3 gegebenen induktiven Beweis: Aus $g \in (fix.\sigma)$ folgt zunächst die Existenz eines $g' \in (fix.\sigma)$ mit $g = \tau(g')$. Der Induktionsanfang mit $n=0$ lautet:

(C.25) $g(s,\varepsilon) = \tau(g')(s,\varepsilon) = \varepsilon = h(s,\varepsilon)$.

Der Induktionsschritt nach $n+1$ impliziert, daß $in \in M^{n+1}$ von der Form $in = m\hat{}in'$ ist. Es gilt:

(C.26) $g(s,in) = \tau(g')(s,m\hat{}in')$

Ist $(t,out) = \delta(s,m)$ wie in (C.1) definiert, so ist dies identisch mit:

(C.27) $\quad = out\hat{}g'(t,in')$

Die Induktionsvoraussetzung sowie (C.1) rückwärts angewendet liefern:

(C.28) $\quad \stackrel{IV}{=} out\hat{}h(t,in') = h(s,in)$

Damit folgt (C.24). Weil $g \in (fix.\sigma)$ und h stetige Funktionen sind, folgt für $in \in M^\infty$, daß $g(s,in)=h(s,in)$ und daher insgesamt: $g = h$. Also folgt $(fix.\sigma) \subseteq \{h\}$, oder gleichbedeutend:

(C.29) $[\![A]\!]^d \subseteq [\![A]\!]^c$.

Gemeinsam mit (C.23) folgt die Behauptung. (□)

Beweis C.5 zu 5.8, Seite 72: (Monotonie in δ)

Es gelte $\delta' \subseteq \delta$ und beide Transitionsrelationen seien total. Die Proposition ist nach Definition der Semantikabbildung $[\![.]\!]^c$ offensichtlich äquivalent zu der Aussage:

(C.30) $[\![(S,M,\delta',I)]\!]^p \subseteq [\![(S,M,\delta,I)]\!]^p$.

Seien σ und σ' die zu δ und δ' gehörenden Funktionale, wie sie in Proposition 5.5,(5.1) eingeführt wurden. Mit der Wahl der Identität für α gelten die Voraussetzungen von Beweisschema $(B_\subseteq)$ und damit Aussage (C.30). (□)

Beweis C.6 zu 5.9, Seite 72: ($[\![.]\!]^c$ besitzt immer Elemente)

Weil (S,M,δ,I) total ist, gibt es zu jedem Paar bestehend aus Quellzustand und Eingabezeichen $(s,m) \in S \times M$ eine Transition in δ, die das Eingabezeichen in diesem Quellzustand verarbeitet. Mit Hilfe des Auswahlaxioms wählen wir eine deterministische, totale Transitionsrelation $\delta' \subseteq \delta$, die genau eine solche Transition für jedes Paar auswählt. Ebenso wählen wir ein Element $(s_i, out_i) \in I$ und setzen

(C.31) $I' \stackrel{def}{=} \{(s_i, out_i)\}$.

Dann ist (S,M,δ',I') total und deterministisch und hat nach Propositionen 5.3 und 5.7 genau ein Element f als Semantik:

(C.32) $\{f\} = [\![(S,M,\delta',I')]\!]^c$.

Dieses ist nach Proposition 5.8 auch in $[\![(S,M,\delta,I')]\!]^c$ enthalten. Nach Definition 5.4 sieht man sofort, daß mit $I' \subseteq I$ folgt, daß

(C.33) $f \in [\![(S,M,\delta,I)]\!]^c$.

Insbesondere ist $[\![(S,M,\delta,I)]\!]^c$ nicht leer. (□)

Beweis C.7 zu 5.11, Seite 74: (*complete* totalisiert)

Die Konstruktion von δ' ist gerade so gewählt, daß δ in jedem Paar bestehend aus Zustand $s \neq \pm$ und Eingabezeichen m erweitert wird, an dem δ partiell ist. Ebenso ist zu jedem Eingabezeichen m im Zustand $\pm$ eine Transition vorhanden, z.B. $\delta'(\pm,m,\pm,\varepsilon)$. (□)

Beweis C.8 zu 5.13, Seite 74: (Einschränkung auf ω-erreichbare Zustandsmenge)

Beweisidee: Zunächst zeigen wir, daß das entstehende Transitionsdiagramm δ' wieder total ist. Dann führen wir die Behauptung auf die beiden Fixpunkte der zur Semantikdefinition verwendeten Funktionale zurück. Mit zwei Funktionen α aus (C.37) und β aus (C.47) sowie dem Prädikat P aus (C.51) zeigen wir die Voraussetzungen von Beweisschema *(BP)*. Damit folgt die geforderte Gleichheit.

Totalität: Sei $s \in R, m \in M$. Da δ total ist, gibt es $t \in S, out \in M^\omega$, so daß $\delta(s,m,t,out)$. Nach Proposition 4.9 ist R abgeschlossen, also $t \in R$. Damit ist auch δ' total.

$[\![.]\!]^c$ zurückgeführt auf $[\![.]\!]^p$: In Definition 5.4 wird zur Definition von $[\![(S,M,\delta,I)]\!]^c$ die Funktion $h\in[\![(S,M,\delta,I)]\!]^p$ nur auf Zustände $s_i\in\pi_1(I)\subseteq R$ angewendet. Deswegen gilt die obige Behauptung bereits, wenn
(C.34) $\quad [\![(S,M,\delta,I)]\!]^p|_{R\times M^\omega} = [\![(R,M,\delta',I)]\!]^p$.

Funktional α: Das Funktional α definiert die Restriktion auf erreichbare Zustände:
(C.35) $\quad \alpha : (S\times M^\omega \overset{s}{\to} M^\omega) \;\to\; (R\times M^\omega \overset{s}{\to} M^\omega)$
(C.36) $\quad \alpha(f)(t,in) \overset{def}{=} f(t,in)$.
Das Funktional α ist surjektiv und besitzt folgende punktweise Erweiterung auf Mengen von Funktionen X:
(C.37) $\quad \alpha(X) = X|_{R\times M^\omega}$,
die monoton bezüglich Mengeninklusion ist. Damit ist (BP.f) erfüllt. Seien σ und σ' die zu $[\![(S,M,\delta,I)]\!]^p$ und $[\![(R,M,\delta',I)]\!]^p$ gehörenden Funktionale, wie in Proposition 5.5,(5.1) eingeführt. Q und Q' seien die zugehörigen Prädikate. Damit sind (BP.a) bis (BP.c) erfüllt.

Aussage (BP.d): Seien $f\in S\times M^\omega \overset{s}{\to} M^\omega$ und $X \subseteq S\times M^\omega \overset{s}{\to} M^\omega$. Es gelte
(C.38) $\quad Q.f.X$,
dann gilt nach Definition von Q:
(C.39) $\quad \forall s\in S.\ f(s,\varepsilon) = \varepsilon \wedge$
(C.40) $\quad \forall m;s\in S.\exists t\in S,out;g\in X.\ \delta(s,m,t,out) \wedge \forall in.\ f(s,m\hat{\ }in) = out\hat{\ }g(t,in)$.
Aus (C.39) und (C.40) folgt
(C.41) $\quad \forall s\in R.\ f(s,\varepsilon) = \varepsilon \wedge$
(C.42) $\quad \forall m;s\in R.\exists t\in S,out;g\in X.\ \delta(s,m,t,out) \wedge \forall in.\ f(s,m\hat{\ }in) = out\hat{\ }g(t,in)$.
Wegen der Abgeschlossenheit von R folgt aus (C.42) auch $t\in R$ und damit
(C.43) $\quad Q'.(\alpha.f).(\alpha.X)$.
Die Implikationskette (C.38) bis (C.43) liefert die Aussage *(C.d)*. Aus Beweisschema $(C_\subseteq)$ folgt damit die Gültigkeit der Aussage *(BP.d)*:
(C.44) $\quad \forall X.\ \alpha(\sigma.X) \subseteq \sigma'(\alpha.X)$.
Beweisschema $(BP_\subseteq)$ liefert:
(C.45) $\quad \alpha(fix.\sigma) \subseteq (fix.\sigma')$.

Funktional β: Wir definieren die Menge K als Restriktion von Funktionen aus $(fix.\sigma)$ auf unerreichbare Zustände:
(C.46) $\quad K \overset{def}{=} (fix.\sigma)|_{(S\setminus R)\times M^\omega} = \{\ k: (S\setminus R)\times M^\omega \overset{s}{\to} M^\omega \mid \exists f\in(fix.\sigma).\ f|_{(S\setminus R)\times M^\omega}=k\ \}$.
Aufgrund Proposition 5.9 ist K nicht die leere Menge: $K\neq\emptyset$.
Das Funktional β definiert eine Umkehrung des Funktionals α, das eine Funktion auf unerreichbaren Zuständen passend durch Elemente aus K erweitert:
(C.47) $\quad \beta : \mathbb{P}(R\times M^\omega \overset{s}{\to} M^\omega) \;\to\; \mathbb{P}(S\times M^\omega \overset{s}{\to} M^\omega)$
(C.48) $\quad \beta(Y) \overset{def}{=} \{\ g{+}k \mid g\in Y \wedge k\in K\ \}$.
$g{+}k$ ist die Summe der beiden Funktionen g und k mit disjunktem Urbild. β ist monoton. Nach Konstruktion von β gilt wegen $K\neq\emptyset$:
(C.49) $\quad \alpha(\beta.Y) = \{\ (g{+}k)|_{R\times M^\omega} \mid g\in Y \wedge k\in K\ \} = Y$
und damit (BP.j). Nach Definition gilt:
(C.50) $\quad Y\neq\emptyset \Rightarrow \beta(Y)|_{(S\setminus R)\times M^\omega}=K$.

Prädikat P: Das Prädikat P charakterisiert alle Mengen, die den Fixpunkt von σ' enthalten:

(C.51) $P(Y) \stackrel{def}{=} (fix.\sigma') \subseteq Y$

Damit ist P zulässig, und es gilt (BP.g). Offensichtlich ist auch (BP.h) richtig. Aus der Implikationskette

(C.52) $P(Y) \Rightarrow (fix.\sigma') \subseteq Y \Rightarrow (fix.\sigma') = \sigma'(fix.\sigma') \subseteq \sigma'(Y) \Rightarrow P(\sigma'.Y)$

folgt (BP.i).

Aussage (BP.k): $P(Y) \Rightarrow \beta(\sigma'.Y) \subseteq \sigma(\beta.Y)$: Seien $f \in S \times M^\omega \stackrel{s}{\rightarrow} M^\omega$ und $Y \subseteq R \times M^\omega \stackrel{s}{\rightarrow} M^\omega$. Es gelte

(C.53) $P(Y) \wedge f \in \beta(\sigma'.Y)$.

$P(Y)$ impliziert, daß $Y \neq \emptyset$. Nach Definition von β folgt

(C.54) $\exists g \in \sigma'(Y),\ k \in K.\ g+k = f$.

Aus (C.54) folgt für g durch Expansion von σ':

(C.55) $\forall s \in R.\ g(s,\varepsilon) = \varepsilon\ \wedge$

(C.56) $\forall m;s \in R. \exists t \in R, out; g' \in Y.\ \delta'(s,m,t,out) \wedge \forall in.\ g(s,m\hat{\ }in) = out\hat{\ }g'(t,in)$.

Wegen $s \in R$ gilt obiges auch für $f = g+k$ statt g. Wegen $\delta' \subseteq \delta$ kann δ' ersetzt werden. Es gilt:

(C.57) $\forall s \in R.\ f(s,\varepsilon) = \varepsilon\ \wedge$

(C.58) $\forall m;s \in R. \exists t \in R, out; g' \in Y.\ \delta(s,m,t,out) \wedge \forall in.\ f(s,m\hat{\ }in) = out\hat{\ }g'(t,in)$.

Aus der Existenz eines $g' \in Y$ folgt die Existenz eines $f' \in \beta(Y)$ mit $f' = g'+k'$ mit $k' \in K$. Damit folgt aus (C.58):

(C.59) $\forall m;s \in R. \exists t \in R, out; f' \in \beta(Y).\ \delta(s,m,t,out) \wedge \forall in.\ f(s,m\hat{\ }in) = out\hat{\ }f'(t,in)$.

Aus (C.54) folgt für $k \in K$ mittels Definition von σ:

(C.60) $\forall s \in (S \backslash R).\ k(s,\varepsilon) = \varepsilon\ \wedge$

(C.61) $\forall m;s \in (S \backslash R). \exists t \in S, out; k' \in (fix.\sigma).\ \delta(s,m,t,out) \wedge \forall in.\ k(s,m\hat{\ }in) = out\hat{\ }k'(t,in)$.

Wegen $s \in S \backslash R$ gilt obiges auch für $f = k+g$ statt k:

(C.62) $\forall s \in (S \backslash R).\ f(s,\varepsilon) = \varepsilon\ \wedge$

(C.63) $\forall m;s \in (S \backslash R). \exists t \in S, out; k' \in (fix.\sigma).\ \delta(s,m,t,out) \wedge \forall in.\ f(s,m\hat{\ }in) = out\hat{\ }k'(t,in)$.

Aus der Eigenschaft $P(Y)$ und (C.45) folgern wir die Inklusionskette:

(C.64) $Y \supseteq (fix.\sigma') \supseteq \alpha(fix.\sigma)$.

Wir wenden β darauf an und erhalten wegen der Monotonie von β und (C.49):

(C.65) $\beta(Y) \supseteq \beta(\alpha(fix.\sigma)) = (fix.\sigma)$.

Aus (C.63) folgt damit

(C.66) $\forall m;s \in (S \backslash R). \exists t \in S, out; k' \in \beta(Y).\ \delta(s,m,t,out) \wedge \forall in.\ f(s,m\hat{\ }in) = out\hat{\ }k'(t,in)$.

Die Zusammensetzung aus (C.57), (C.59), (C.62) und (C.66) liefert:

(C.67) $f \in \sigma(\beta(Y))$.

Die Beweisführung (C.53-C.67) zeigt die Aussage (BP.k):

(C.68) $P(Y) \Rightarrow \beta(\sigma'.Y) \subseteq \sigma(\beta.Y)$.

Ergebnis: Mit Beweisschema *(BP)* folgt die Aussage (C.34) und damit die Behauptung.

($\Box$)

Beweis C.9 zu 5.14, Seite 74: (Entfernen nicht ω-erreichbarer Zustände)
Sei $R \stackrel{def}{=} \omega{-}reach(S,M,\delta,I)$. Mit $R \subseteq S'$ und

(C.69) $\delta_1 \stackrel{def}{=} \delta \cap R \times M \times R \times M^\omega \;=\delta' \cap R \times M \times R \times M^\omega$

folgt $R = \omega{-}reach(S',M,\delta',I)$. Zweimaliges Anwenden von Proposition 5.13 liefert die gewünschte Aussage:

(C.70) $[\![(S,M,\delta,I)]\!]^c = [\![(R,M,\delta_1,I)]\!]^c = [\![(S',M,\delta',I)]\!]^c$. $(\Box)$

Beweis C.10 zu 5.15, Seite 75: ($[\![.]\!]$ ist kompatibel mit $[\![.]\!]^c$)
Ist (S,M,δ,I) ein totaler Automat, so entsteht bei der Totalisierung ein Automat

(C.71) $(S',M,\delta',I) \stackrel{def}{=} complete(S,M,\delta,I)$,

bei dem sich δ und δ' nur in den Transitionen unterscheiden, die als Quelle den Zustand $\pm$ haben.

Offensichtlich ist $\pm$ kein ω-erreichbarer Zustand. Daher spielt das Verhalten des totalisierten Automaten im Zustand $\pm$ keine Rolle. Es gilt:

(C.72) $\omega{-}reach(S',M,\delta',I) \subseteq S$

(C.73) $\delta = \delta' \cap S \times M \times S \times M^\omega$.

Mit Proposition 5.14 folgt daher:

(C.74) $[\![(S',M,\delta',I)]\!]^c = [\![(S,M,\delta,I)]\!]^c$

und mit Definition 5.12 wegen (C.71):

(C.75) $[\![(S,M,\delta,I)]\!] = [\![(S,M,\delta,I)]\!]^c$. $(\Box)$

Beweis C.11 zu 5.22, Seite 78: (Operationelle und denotationelle Semantik sind äquivalent (Teil A))
Die Aussage 3 ist ein Spezialfall von Aussage 2 für den Fall, daß keine unendliche initiale Ausgabe und keine finalen Transitionen auftreten. In diesem Fall ist $\#out(tr){=}\infty$ genau dann, wenn $\#tr{=}\infty$. Das impliziert aber $\#in(tr){=}\infty$, also notwendigerweise $in(tr){=}in$.

Beweisidee: Um Aussage 2 zu zeigen, benutzen wir die erweiterte Semantikdefinition $[\![.]\!]^e$, die strukturell identisch ist mit der Semantikdefinition $[\![.]\!]^d$ in 5.4. Wir zeigen dann eine Verbindung zwischen $[\![.]\!]^e$ und $[\![.]\!]^d$ und eine Verbindung zwischen $[\![.]\!]^e$ und $[\![.]\!]^{op}$ mittels Vergißoperatoren.

$[\![.]\!]^e$ und $[\![.]\!]^E$: Die Funktionen in $[\![.]\!]^E$ verhalten sich wie die in $[\![.]\!]^p$, geben jedoch immer die gerade ausgeführte Transition bekannt. Die Funktionen aus $[\![.]\!]^e$ geben zusätzlich anfangs den gewählten Initialwert aus.

(C.76) $[\![(S,M,\delta,I)]\!]^e \stackrel{def}{=} \{\, g \in M^\omega \stackrel{s}{\rightarrow} (M \cup I \cup \delta)^\omega \mid$

(C.77) $\quad \exists\, h \in [\![(S,M,\delta,I)]\!]^E,\ (t_0,out_0) \in I.\ \forall in.\ g(in) = \langle (t_0,out_0)\rangle\,\hat{}\,out_0\,\hat{}\,h(t_0,in)\,\}$

wobei $[\![.]\!]^E$ die größte Menge von mit Zuständen parametrisierten Funktionen charakterisiert, die folgender rekursiver Definition genügt:

(C.78) $[\![(S,M,\delta,I)]\!]^E \stackrel{def}{=} \{\, h \in S \times M^\omega \stackrel{s}{\rightarrow} (M \cup \delta)^\omega \mid (\forall s.\ h(s,\varepsilon) = \varepsilon) \wedge$

(C.79) $\quad \forall m,s. \exists t,out. \delta(s,m,t,out) \wedge \exists h' \in [\![(S,M,\delta,I)]\!]^E.$

(C.80) $\quad\quad \forall in.\ h(s,m\,\hat{}\,in) = \langle (s,m,t,out)\rangle\,\hat{}\,out\,\hat{}\,h'(t,in)\}$

Verbindung $[\![.]\!]^e$ und $[\![.]\!]^c$: Mit Hilfe der beiden Vergiß-Funktoren γ, die wie folgt definiert werden:

(C.81) $\gamma : (S{\times}M^\omega \overset{s}{\to} (M{\cup}\delta)^\omega) \to (S{\times}M^\omega \overset{s}{\to} M^\omega)$

(C.82) $\gamma(g)(s,i) \overset{def}{=} M©g(s,i)\ \forall i{\in}M^\omega, s{\in}S$

(C.83) $\gamma : (M^\omega \overset{s}{\to} (M{\cup}I{\cup}\delta)^\omega) \to (M^\omega \overset{s}{\to} M^\omega)$

(C.84) $\gamma(g)(i) \overset{def}{=} M©g(i)\ \forall i{\in}M^\omega$

läßt sich aufgrund der strukturellen Übereinstimmung der Definitionen 5.12 und (C.78-C.80) zeigen, daß

(C.85) $\gamma[\![(S,M,\delta,I)]\!]^E = [\![(S,M,\delta,I)]\!]^p.$

Daraus folgt dann sofort:

(C.86) $\gamma[\![(S,M,\delta,I)]\!]^e = [\![(S,M,\delta,I)]\!]^c.$

Verbindung $[\![.]\!]^e$ und $[\![.]\!]^{op}$: Wir definieren uns dazu eine weitere Vergiß-Funktion β, die aus einem Strom die Transitionssequenz und den Initialwert filtert:

(C.87) $\beta : (M{\cup}I{\cup}\delta)^\omega \to (I{\cup}\delta)^\omega$

(C.88) $\beta(i) \overset{def}{=} (I{\cup}\delta)©i\ \forall i{\in}M^\omega$

Wir betrachten die Funktion $g{\in}[\![(S,M,\delta,I)]\!]^e$ und $i{\in}M^\omega$. Dann folgt nach Definitionen (C.76-C.77) und (C.78-C.80), daß $g(i)$ von der Form

(C.89) $g(i) = \langle(t_0,out_0)\rangle\ \hat{}\ out_0\ \hat{}\ \mathtt{conc}_{k=1}^{n}(\langle(s_k,in_k,t_k,out_k)\rangle\ \hat{}\ out_k)$

ist, für passende

(C.90) $n{\in}\mathbb{N}{\cup}\{\infty\},\ (t_0,out_0){\in}I,\ (s_k,in_k,t_k,out_k){\in}\delta$

(C.91) $i = \mathtt{conc}_{k=1}^{n} in_k.$

Nach Konstruktion gilt außerdem

(C.92) $\forall 0{\le}k{<}n.\ t_k = s_{k+1}.$

Aufgrund finaler Transitionen oder einer unendlichen initialen Ausgabe können einige out_k unendlich lang sein. Sei $r \overset{def}{=} min\{k|\#out_k{=}\infty\}$ der kleinste solche Index ($r{\ge}0$). Falls kein solcher existiert, setzten wir $r{=}n$. Dann gilt nach (C.89):

(C.93) $g(i) = \langle(t_0,out_0)\rangle\ \hat{}\ out_0\ \hat{}\ \mathtt{conc}_{k=1}^{r}(\langle(s_k,in_k,t_k,out_k)\rangle\ \hat{}\ out_k)$

und nur das letzte Glied der Konkatenation kann unendlich lang sein. Damit ist

(C.94) $\beta(g.i) = \langle(t_0,out_0)\rangle\ \hat{}\ \mathtt{conc}_{k=1}^{r}\langle(s_k,in_k,t_k,out_k)\rangle$

Mit (C.90) und (C.92) folgt $\beta(g.i){\in}I\ \hat{}\ \Theta(\delta)$. Also ist Υ anwendbar und wir definieren

(C.95) $tr \overset{def}{=} \Upsilon(\beta(g.i)).$

Weil nur out_r unendlich lang sein kann, folgt:

$$\text{(C.96)}\quad tr = \begin{cases} \xrightarrow{out_0} & \texttt{if}\ \ r{=}0 \wedge \#out_r{=}\infty \\ \xrightarrow{out_0} t_0\ \hat{}(\mathtt{conc}_{k=1}^{r-1} s_k \xrightarrow{in_k/out_k} t_k)\ \hat{}\ s_r \xrightarrow{in_r/out_r} & \texttt{if}\ \ r{>}0 \wedge \#out_r{=}\infty \\ \xrightarrow{out_0} t_0\ \hat{}(\mathtt{conc}_{k=1}^{r} s_k \xrightarrow{in_k/out_k} t_k) & \texttt{if}\ \ r{=}\infty \vee \#out_r{<}\infty \end{cases}$$

In jedem dieser drei Fälle gilt wegen (C.89) und (C.84):

(C.97) $out(tr) = \mathtt{conc}_{k=0}^{r} out_k = \mathtt{conc}_{k=0}^{n} out_k = (\gamma\ g)(i).$

In den ersten beiden Fällen gilt für die Eingabe:

(C.98) $in(tr) = \mathtt{conc}_{k=1}^{r} in_k \sqsubseteq \mathtt{conc}_{k=1}^{n} in_k = i$

und gleichzeitig

(C.99) $\#out(tr) \ge \#out_r = \infty$

Im letzteren Fall gilt mit $n{=}r$:
(C.100) $in(tr) = \texttt{conc}_{k=1}^{r} in_k = \texttt{conc}_{k=1}^{n} in_k = i.$
Damit erfüllt der Ablauf $tr{\in}[\![(S,M,\delta,I)]\!]^{op}$ wegen (C.97-C.100) die in Aussage 2 gestellten Bedingungen für $f{=}(\gamma\ g)$.

Verbindung $[\![.]\!]^c$ und $[\![.]\!]^{op}$: Seien nun $f{\in}[\![(S,M,\delta,I)]\!]^c$ und $i{\in}M^\omega$ wie in Aussage 2 gefordert, dann gibt es nach (C.86) ein $g{\in}[\![(S,M,\delta,I)]\!]^e$ mit $f{=}\gamma g$. Wir stellen fest, daß $tr \stackrel{def}{=} \Upsilon(\beta(g.i))$ die in Aussage 2 geforderten Bedingungen erfüllt. (□)

Beweis C.12 zu 5.22, Seite 78: (Operationelle und denotationelle Semantik sind äquivalent (Teil B))
Beweisidee zu Aussage 1: Zu einem gegebenem Ablauf tr des Automaten (S,M,δ,I) wird mittels verfeinernden Umformungen ein deterministischer Automat konstruiert, der diesen Ablauf enthält. Die einzige Funktion f der Semantik dieses Automaten erfüllt die geforderte Eigenschaft.

Sei ein Ablauf $tr{\in}[\![(S,M,\delta,I)]\!]^{op}$ gegeben. Dann gibt es ein initiales Element $(t_0,out_0){\in}I$ und eine Transitionssequenz $ts{=}((s_k,in_k,t_k,out_k)_k){\in}\Theta(\delta)$, so daß
(C.101) $\Upsilon((t_0,out_0)\hat{\ }ts) = tr$
Ohne Einschränkung sei ts von minimaler Länge, das heißt $in(tr){=}in(ts)$ und damit auch $out(tr){=}out(ts)$. Wir konstruieren nun eine Funktion f, die auf die Eingabe $in(tr)$ mit der Ausgabe
(C.102) $f(in(tr)) = out(tr)$
reagiert, indem wir den Automaten zunächst zu einem azyklischen Automaten (Baum) auffalten und dann durch Entfernen von Transitionen deterministisch machen. Die Semantik des entstehenden Automaten ist in $[\![(S,M,\delta,I)]\!]^c$ enthalten und besteht nur aus der gesuchten Funktion f.

Der azyklische Automat: Wir benutzen im Vorgriff auf Kapitel 6 den Verfeinerungskalkül aus Definition 6.17.
Zunächst definieren wir den Automaten (S',M,δ',I') mit
(C.103) $S' \stackrel{def}{=} S \times M^*$
(C.104) $I' \stackrel{def}{=} \{\ ((t_0,*),\ out_0) \mid (t_0,out_0){\in}I\ \}$
(C.105) $\delta' \stackrel{def}{=} \{\ ((s,*),\ m,\ (t,*),\ out) \mid \delta(s,m,t,out)\ \}.$
Mit der Projektion $\alpha{=}\pi_1$ ist nach Regel (RefS) aus 6.13 der neue Automat eine Verfeinerung des alten:
(C.106) $[\![(S',M,\delta',I')]\!] \subseteq [\![(S,M,\delta,I)]\!]$
und er ist ebenfalls total. Die neu eingeführte Zustandskomponente M^* wird nun genutzt, um die bisherige Historie der Eingabe zu speichern. Dazu wird die Initialmenge verkleinert:
(C.107) $I_2 \stackrel{def}{=} \{\ ((t_0,\varepsilon),\ out_0) \mid (t_0,out_0){\in}I\ \} \subseteq I'$
und das Transitionssystem verkleinert:
(C.108) $\delta_2 \stackrel{def}{=} \{\ ((s,oldin),\ m,\ (t,oldin\hat{\ }m),\ out) \mid \delta(s,m,t,out)\ \} \subseteq \delta'.$
Weil δ total ist, ist auch δ_2 total. Damit ist Regel (RemT) aus 6.5 anwendbar und liefert gemeinsam mit Regel (RemI) aus 6.3
(C.109) $[\![(S',M,\delta_2,I_2)]\!] \subseteq [\![(S',M,\delta_2,I')]\!] \subseteq [\![(S,M,\delta',I')]\!]$

Der Automat *(S',M,δ_2,I_2)* ist nach Konstruktion zyklenfrei. Jeder seiner Pfade beschreibt einen Ablauf.

<u>Deterministischer Teilautomat:</u> Unter Benutzung der Komponenten von *tr* aus (C.101) definieren wir
(C.110) $I_3 \overset{def}{=} \{((t_0,\varepsilon),out_0)\} \subseteq I_2$
Das Transitionssystem δ_3 wird wie folgt konstruiert: Für jeden Zustand *(s,oldin)*$\in$*S'* und jede Eingabe $m \in M$ wird festgestellt, ob ein $n \in \mathbb{N}$ existiert, so daß
(C.111) $s{=}s_n,\ oldin{=}\mathtt{conc}_{k=1}^{n-1} in_k,\ m{=}in_k$
Ist das der Fall, so wird
(C.112) $((s,oldin),m,\ (t_n,oldin\,\hat{}\,m),\ out_n) \in \delta_2$
gewählt. Sonst wird eine beliebige Transition aus δ_2 gewählt.
Nach Konstruktion ist δ_3 total und deterministisch und es ist $\delta_3 \subseteq \delta_2$. Die Regeln (RemT) und (RemI) liefern:
(C.113) $[\![(S',M,\delta_3,I_3)]\!] \subseteq [\![(S',M,\delta_3,I_2)]\!] \subseteq [\![(S,M,\delta_2,I_2)]\!]$

<u>Funktion *f*:</u> Sei
(C.114) $\{f\} = [\![(S',M,\delta_3,I_3)]\!]^d$.
Nach (C.106), (C.109), (C.113) und den Propositionen 5.7 und 5.15 ist
(C.115) $f \in [\![(S,M,\delta,I)]\!]^c$.
Sei τ das zur Semantikdefinition von *f* benutzte Funktional (siehe (C.1)) und $h{=}(fix.\tau)$, dann ist
(C.116) $f(in(tr)) = out_0\ \hat{}\ h((t_0,\varepsilon),\ in(tr)).$
Wir zeigen durch Induktion über n, daß:
(C.117) $\forall n,m \in \mathbb{N}.\ n{+}m{<}1{+}\#in(tr) \Rightarrow$
(C.118) $\quad h((s_{m+1},\mathtt{conc}_{k=1}^{m} in_k),\mathtt{conc}_{k=m+1}^{m+n} in_k) = \mathtt{conc}_{k=m+1}^{m+n} out_k$
Für $n{=}0$ ist die Aussage wegen $h(*,\varepsilon){=}\varepsilon$ erfüllt.
Für $n{+}1{>}0$ ist
(C.119) $h((s_{m+1},\mathtt{conc}_{k=1}^{m} in_k),\mathtt{conc}_{k=m+1}^{m+n+1} in_k) =$
(C.120) $= h((s_{m+1},\mathtt{conc}_{k=1}^{m} in_k),in_{m+1}\,\hat{}\,\mathtt{conc}_{k=m+2}^{m+n+1} in_k) =$
wegen $h{=}\tau(h)$ und (C.111-C.112) sowie $t_{m+1}{=}s_{m+2}$
(C.121) $= out_{m+1}\ \hat{}\ h((s_{m+2},\mathtt{conc}_{k=1}^{m+1} in_k),\mathtt{conc}_{k=m+2}^{m+n+1} in_k) =$
das ist nach Induktionsvoraussetzung identisch mit
(C.122) $= out_{m+1}\ \hat{}\ \mathtt{conc}_{k=m+2}^{m+1+n} out_k.$
Damit folgt die Behauptung für alle $n \in \mathbb{N}$. Weil h stetig ist folgt Behauptung (C.117) damit auch für $n{=}\infty$.
Mit $m{=}0$ folgt wegen $t_0{=}s_1$ aus (C.117) in (C.116) eingesetzt:
(C.123) $f(in(tr)) = out_0\ \hat{}\ \mathtt{conc}_{k=m+2}^{\#in(tr)} out_k = out(tr)$
und damit die Behauptung. ($\Box$)

Beweis C.13 zu 5.24, Seite 79: (Charakterisierung von $[\![.]\!]^r$ durch $[\![.]\!]^{op}$)
<u>Beweisidee</u>: Die Behauptung folgt für einen totalen Automaten *(S,M,δ,I)*, eine stromverarbeitende Funktion $g \in M^\omega \overset{s}{\rightarrow} M^\omega$ und $i \in M^\omega$ unmittelbar aus der Äquivalenz der beiden Aussagen, die wir durch wechselseitige Implikation zeigen:
(C.124) $\exists tr \in [\![(S,M,\delta,I)]\!]^{op}.\ out(tr){=}g(i) \wedge (in(tr){=}i \vee (in(tr){\sqsubseteq}i \wedge \#g(i){=}\infty))$
(C.125) $\exists (t_0,out_0) \in I,out_t.\ g(i) = out_0\,\hat{}\,out_t \wedge \delta^\omega(t_0,i,*,out_t)$

(C.124) $\Rightarrow$ (C.125): Wir nehmen an, daß Aussage (C.124) gilt: Sei $tr \in [\![(S,M,\delta,I)]\!]^{op}$ mit den Eigenschaften

(C.126) $out(tr)=g(i) \wedge (in(tr)=i \vee (in(tr) \sqsubseteq i \wedge \#g(i)=\infty)$

gegeben. Nach Definition 5.21 gibt es ein x von der Form

(C.127) $x = \langle (t_0,out_0) \rangle \hat{~} ts \in I \hat{~} (\Theta(\delta))$

mit $\Upsilon.x{=}tr$. Wir wählen ein x mit minimaler Länge, so daß gilt $in(tr)=in(ts)$. Nach Definition 4.6 von δ^ω gilt

(C.128) $\delta^\omega(t_0,in(ts),*,out(ts))$.

Setzen wir nun $out_t \stackrel{def}{=} out(ts)$, so folgt wegen (C.126), daß gilt

(C.129) $g(i) = out(tr) = out_0 \hat{~} out_t$.

Ist nun $in(tr)=i$, so gilt wegen $in(tr)=in(ts)$ die Aussage (C.125). Im anderen Fall gilt nach (C.126) $in(ts)=in(tr) \sqsubseteq i$ und $\#out(tr)=\infty$. Aufgrund der Vollständigkeit des betrachteten Automaten (S,M,δ,I) kann die Transitionssequenz ts so zu $ts' \in \Theta(\delta)$ verlängert werden, daß $in(ts')=i$. Aus (C.128) mit ts' statt ts folgt nun wieder Aussage (C.125).

(C.125) $\Rightarrow$ (C.124): Wir nehmen an, daß Aussage (C.125) gilt: Sei $(t_0,out_0) \in I$ und $out_t \in M^\omega$ gegeben, so daß gilt

(C.130) $g(i) = out_0 \hat{~} out_t \wedge \delta^\omega(t_0,i,*,out_t)$.

Ist $\#out_0{=}\infty$, so setzen wir $tr \stackrel{def}{=} \xrightarrow{out_0}$ und erhalten wegen $out(tr)=out_0=out_0 \hat{~} out_t=g(i)$ und $in(tr)=\varepsilon \sqsubseteq i$ die gewünschte Aussage (C.124). Ist andernfalls $i{=}\varepsilon$, so setzen wir $tr \stackrel{def}{=} \xrightarrow{out_0} t_0$ und erhalten wieder Aussage (C.124).

Sei ab jetzt $i \neq \varepsilon$ und $\#out_0 < \infty$. Nach Definition 4.6 gibt es eine Transitionssequenz $ts \in \Theta(\delta)$ mit der Eigenschaft

(C.131) $in(ts) = i,\ out(ts) = out_t,\ source(ts) = t_0$

Wir setzen $tr \stackrel{def}{=} \Upsilon(\langle (t_0,out_0) \rangle \hat{~} ts)$. Nach Konstruktion ist $tr \in [\![(S,M,\delta,I)]\!]^{op}$. Weiterhin gilt

(C.132) $in(tr) \sqsubseteq in(ts) = i$

Ist $\#g(i) < \infty$, so enthält ts keine Transition mit unendlicher Ausgabe, und es gilt die Eigenschaft $in(tr)=in(ts)$. Damit gilt auch $in(tr)=i$, woraus unmittelbar Aussage (C.124) folgt. $(\square)$

Beweis C.14 zu 5.25, Seite 79: ($[\![.]\!]^c \subseteq [\![.]\!]^r$)
Folgt aus der Proposition 5.24 und Aussage 2 von Satz 5.22. $(\square)$

Beweis C.15 zu 5.27, Seite 82: (*automaton* Eigenschaften)
Es gilt $|I|{=}1$. Zu jedem (bereits verarbeiteten) Eingabewort $s \in M^*$ und jedem Zeichen $m \in M$ gibt es eine Ausgabe out, die f als Reaktion auf m zusätzlich produziert. Deshalb ist δ total. Die Ausgabe out ist genau dann sogar eindeutig bestimmt, wenn bisher noch keine unendliche Ausgabe erfolgt ist, d.h. $\#f(s) < \infty$. Im anderen Fall ist out nach Definition von $\hat{~}$ keine echte Ausgabe mehr: $f(s) \hat{~} out = f(s)$. Tritt dieser Fall jedoch nicht auf, so ist *automaton* sogar deterministisch.

Nach Proposition 5.16 ist $[\![automaton(f,M)]\!]$ nicht leer. Sei also $g \in [\![automaton(f,M)]\!]$. Nach Proposition 5.25 und Definition 5.23 gilt für alle $in \in M^\omega$:

(C.133) $g(in) = f(\varepsilon) \hat{~} out \wedge \delta^\omega(\varepsilon,in,*,out)$.

Wie sich induktiv über *in* leicht zeigen läßt, gilt daher:
(C.134) $g(in) = f(in)\ \forall in \in M^*$.
Weil beide Funktionen stetig sind gilt damit: $f{=}g$ und es folgt die Behauptung. (□)

Beweis C.16 zu 5.29, Seite 82: (*automaton* Eigenschaften für Mengen)
Nachdem mittels I eine initiale Auswahl getroffen wurde, welche stromverarbeitende Funktion modelliert wird, verläuft alles analog zum Beweis der Proposition 5.27. (□)

Beweis C.17 zu 5.34, Seite 87: (Eigenschaften der Semantik $[\![.]\!]^{ta}$)
Aussage 1: Sei σ das Funktional aus Proposition 5.5, das zur Semantikdefinition $[\![.]\!]^c$ des Taktautomaten $(S, I^\Phi, O^\Phi, \delta, Init)$ verwendet wird.
Es ist induktiv über n zu beweisen, daß die Ergebnisfunktionen aus $\sigma^n(X)$ nach n-maliger Anwendung des Funktionals auf Eingaben der Länge kleiner als n längenerhaltend sind:
(C.135) $\forall X, l \in (I^\Phi)^*, f \in \sigma^n(X).\ \#l < n \Rightarrow \#l = \#f.l$
Die Aussage (C.135) folgt durch Anwendung der Definition von σ und der speziellen Form von δ, in der jede Ausgabe genau ein Element hat.
Nach Definition von $[\![.]\!]^c$ ist mit (C.135) jede Funktion $f \in (fix.\sigma)$ aufgrund der einelementigen initialen Ausgabe um eins verlängernd. Dies impliziert die Wohldefiniertheit der Taktsemantik.

Aussagen 2 und 3 sind eine unmittelbare Konsequenz von Satz 5.16 und der Isomorphie-Eigenschaft von γ^{-1}.

Aussage 4: Wir zeigen die Übereinstimmung beider Semantikdefinitionen durch Fixpunktinduktion mit Beweisschema (A). Dazu definieren wir das für die rekursive Definition von $[\![.]\!]^{tap}$ notwendige Funktional σ^{ta}:
(C.136) $\sigma^{ta} : \mathbb{P}(S \times I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}}) \to \mathbb{P}(S \times I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}})$
(C.137) $\sigma^{ta}(X) = \{\ h \in S \times I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}}) \mid$
(C.138) $\quad \forall m \in I^\Phi, s. \exists t; out \in O^\Phi.\ \delta(s,m,t,out) \wedge \exists h' \in X.$
(C.139) $\quad\quad \forall in \in I^{\overline{\Sigma}}.\ h(s, m\hat{\ }\surd\hat{\ }in) = out\hat{\ }\surd\hat{\ }h'(t,in)\ \}$
Weil die Menge X in der Mengenkomprehension nur positiv (unter dem Existenzquantor) benutzt wird, ist σ^{ta} monoton, und es existiert nach Knaster/Tarski ([TAR55]) ein eindeutiger, mengentheoretisch größter Fixpunkt $(fix.\sigma^{ta})$, der die Menge $[\![(S,M,\delta,I)]\!]^{tap}$ darstellt.
Die Menge der längenerhaltenden stromverarbeitenden Funktionen $\top' = (I^\Phi)^\omega \xrightarrow{l} (O^\Phi)^\omega$ ist selbst ein CPO und das Funktional σ darauf monoton. Sei F der größte Fixpunkt von σ in diesem CPO. Die Aussage
(C.140) $\forall X.\ \gamma(\sigma^{ta}(X)) = \sigma(\gamma(X))$
folgt durch Expandieren der Definitionen auf beiden Seiten, wobei durch die strukturelle Ähnlichkeit beider Funktionale die gewünschte Gleichheit gezeigt werden kann.
Aus der Isomorphie von γ auf der Menge der längenerhaltenden stromverarbeitenden Funktionen $\top'$ folgt die Gültigkeit von *(A.e)*. Aus der Isomorphie-Eigenschaft von γ folgt die Gültigkeit von *(A.f)*. Die Anwendung von Beweisschema (A) liefert
(C.141) $\gamma(fix.\sigma^{ta}) = F$.
Weil F auch ein Fixpunkt im CPO $(I^\Phi)^\omega \xrightarrow{s} (O^\Phi)^\omega$ ist, folgt
(C.142) $F \subseteq (fix.\sigma)$.

Die Aussage (C.135) zeigt, daß alle Funktionen in $(fix.\sigma)$ längenerhaltend sind und damit die umgekehrte Inklusion. Es folgt:

(C.143) $\gamma(fix.\sigma^{ta}) = (fix.\sigma)$.

Durch Einsetzen von (C.143) in $[\![.]\!]^c$ und $[\![.]\!]^{ta}$ folgt die Aussage 4.

Bemerkenswert ist, daß der wesentliche Unterschied zwischen σ und σ^{ta} darin besteht, daß bei σ der Rekursionsanfang $(\forall s.h(s.\varepsilon)=\varepsilon)$ gefordert wird. Dies ist bei gezeiteten stromverarbeitenden Funktionen durch γ gewährleistet, da gilt

(C.144) $\forall in.\ (\gamma.f)(in\downarrow 0) = *\downarrow 0 = \varepsilon$ $(\Box)$

C.4 Verfeinerungstechniken für Automaten

Beweis C.18 zu 6.2, Seite 94: (Verfeinerung ist transitiv)

Folgt aus der Verwendung der Mengeninklusion als Verfeinerungsrelation in Definition 6.1. $(\Box)$

Beweis C.19 zu 6.3, Seite 100: (Verkleinern der Initialmenge)

Der Beweis folgt sofort aus den Definitionen 5.4 und 5.12 der Semantikfunktionen $[\![.]\!]^c$ und $[\![.]\!]$, sowie der Tatsache, daß die Funktion *complete* die Initialmenge nicht verändert. $(\Box)$

Beweis C.20 zu 6.4, Seite 100: (Modifikation finaler Initialelemente)

Die Funktion *complete* modifiziert die Initialmenge nicht. Daher folgt diese Äquivalenz sofort aus den Definitionen 5.12 und 5.4. $(\Box)$

Beweis C.21 zu 6.5, Seite 101: (Entfernung von Transitionen)

Mit Proposition 5.8 haben wir bereits ein mächtiges Hilfsmittel zur Verfügung, das uns die obige Aussage auf totalen Automaten liefert.

Wir betrachten die Totalisierung beider Automaten nach Definition 5.10:

(C.145) $(S_c,M,\delta_c,I) \stackrel{def}{=} complete(S,M,\delta,I)$

(C.146) $(S_c,M,\delta_c',I) \stackrel{def}{=} complete(S,M,\delta',I)$.

Es gilt nach Definition:

(C.147) $S_c = S \cup \{\pm\}$

(C.148) $\delta_c = \delta \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg\delta(s,m,*,*)\ \}$

(C.149) $\delta_c' = \delta' \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg\delta'(s,m,*,*)\ \}$.

Wegen der geforderten Beziehung zwischen δ und δ' gilt folgende Äquivalenz:

(C.150) $\neg\delta(s,m,*,*) \Leftrightarrow \neg\delta'(s,m,*,*)$

und wegen $\delta'\subseteq\delta$ auch:

(C.151) $\delta_c' \subseteq \delta_c$.

Mit Proposition 5.8 folgt nun aus (C.151):

(C.152) $[\![(S_c,M,\delta_c',I)]\!] \subseteq [\![(S_c,M,\delta_c,I)]\!]$,

was sofort die Behauptung zeigt. $(\Box)$

Beweis C.22 zu 6.6, Seite 102: (Hinzufügen von Transitionen)
Totalisierungen: Analog zum Beweis C.21 betrachten wir wieder die Totalisierungen beider Automaten:
(C.153) $(S_c,M,\delta_c,I) \stackrel{def}{=} complete(S,M,\delta,I)$
(C.154) $(S_c,M,\delta_c',I) \stackrel{def}{=} complete(S,M,\delta\cup\delta',I)$.
mit
(C.155) $S_c = S \cup \{\pm\}$
(C.156) $\delta_c = \delta \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg\delta(s,m,*,*)\}$
(C.157) $\delta_c' = \delta \cup \delta' \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg(\delta\cup\delta')(s,m,*,*)\}$.

$\delta_c' \subseteq \delta_c$: Sei nun $(s,m,t,out)\in\delta_c'$. Dann gibt es vier Fälle:

1) $\delta(s,m,t,out)$, dann gilt auch $\delta_c(s,m,t,out)$,

2) $s{=}\pm$, dann gilt ebenfalls $\delta_c(s,m,t,out)$,

3) $\neg(\delta\cup\delta')(s,m,*,*)$: das impliziert $\neg\delta(s,m,*,*)$

und damit wieder $\delta_c(s,m,t,out)$,

4) $\delta'(s,m,t,out)$: das impliziert nach Wahl von δ' ebenfalls

$\neg\delta(s,m,*,*)$, also wieder $\delta_c(s,m,t,out)$.

Daher gilt:
(C.158) $\delta_c' \subseteq \delta_c$
und nach Proposition 5.8 die Behauptung. ($\Box$)

Beweis C.23 zu 6.7, Seite 102: (Zielzustände finaler Transitionen)
Totalisierungen: Wieder betrachten wir die Totalisierungen beider Automaten:
(C.159) $(S_c,M,\delta_c,I) \stackrel{def}{=} complete(S,M,\delta,I)$
(C.160) $(S_c,M,\delta_c',I) \stackrel{def}{=} complete(S,M,\delta',I)$.
mit
(C.161) $S_c = S \cup \{\pm\}$
(C.162) $\delta_c = \delta \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg\delta(s,m,*,*)\}$
(C.163) $\delta_c' = \delta' \cup \{(s,m,*,*) \mid s{=}\pm \vee \neg\delta'(s,m,*,*)\}$.
Weil wir keine Partialität hinzugefügt oder entfernt haben, unterscheiden sich beide Totalisierungen höchstens in Transitionen der Form $(s,m,*,out)$, das heißt:
(C.164) $\delta_c\backslash\{(s,m,*,out)\} = \delta_c'\backslash\{(s,m,*,out)\}$.

$\sigma = \sigma'$: Seien σ und σ' die zur Semantik von (S_c,M,δ_c,I) beziehungsweise (S_c,M,δ_c',I) gehörenden Funktionale aus Proposition 5.5,(5.1) und Q, Q' die zugehörigen Bedingungen. Dann gilt:
(C.165) $Q.f.X \Leftrightarrow$
(C.166) $\forall s_1.\ h(s_1,\varepsilon) = \varepsilon \wedge$
(C.167) $\forall s_1,m_1.\ \exists t,out_1;f'\in X.\ \delta_c(s_1,m_1,t,out_1) \wedge \forall in.\ f(s_1,m_1\,\hat{}\,in) = out_1\,\hat{}\,f'(t,in)$
Wir spalten die Zeile (C.167) durch Konjunktion mit
(C.168) $tt = ((s,m,out) = (s_1,m_1,out_1) \vee (s,m,out) \neq (s_1,m_1,out_1))$

und Distribution mit DeMorgan in zwei Fälle und behandeln diese.
Im Fall $(s,m,out) \neq (s_1,m_1,out_1)$ gilt wegen (C.164):
(C.169) $\delta_c(s_1,m_1,t,out_1) \Leftrightarrow \delta_c'(s_1,m_1,t,out_1)$
und Zeile (C.167) ist äquivalent zu:
(C.170) $\forall s_1,m_1.\ \exists t,out_1;f'\in X.\ \delta_c'(s_1,m_1,t,out_1) \wedge \forall in.\ f(s_1,m_1\hat{\ }in) = out_1\hat{\ }f'(t,in)$
Im anderen Fall $(s,m,out) = (s_1,m_1,out_1)$ gilt nach Definition (C.163) von δ_c':
(C.171) $\exists t.\ \delta_c(s,m,t,out) \Leftrightarrow \exists t_1.\ \delta_c'(s,m,t_1,out)$.
Bei Existenz eines solchen t und damit eines t_1 gilt wegen $\#out_1{=}\infty$:
(C.172) $f(s_1,m_1\hat{\ }in) = out_1\hat{\ }f'(t,in) = out_1 = out_1\hat{\ }f'(t_1,in)$.
Also ist Zeile (C.167) auch in diesem Fall äquivalent zu (C.170): Das ist aber in beiden Fällen die Definition von Q', also gilt:
(C.173) $\sigma = \sigma'$
und damit die geforderte Aussage:
(C.174) $[\![(S,M,\delta,I)]\!] = [\![(S_c,M,\delta_c,I)]\!]^c = [\![(S_c,M,\delta_c',I)]\!]^c = [\![(S,M,\delta',I)]\!]$. $(\Box)$

Beweis C.24 zu 6.9, Seite 103: (Eigenschaften von *compactify*)
Der erste Teil folgt durch Anwendung der Proposition 6.7 in einer Kette von Transformationen für jedes (s,m,out)-Tripel. Dies führt zu einer Kette von äquivalenten Umformungen, deren Ergebnis genau der Automat $compactify(S,M,\delta,I)$ ist. Wesentlich ist dabei, daß unabhängig von der Mächtigkeit der beteiligten Mengen S und δ^f immer ein Grenzwert existiert, der durch diese Umformungen entsteht.

Die zweite Aussage gilt nach Definition 4.7 der Funktion *reach* offensichtlich.

Für die dritte Aussage reicht es wegen Aussage 2 und Proposition 4.9 zu zeigen, daß
(C.175) $\omega{-}reach(compactify(S,M,\delta,I)) \subseteq reach(compactify(S,M,\delta,I))$.
Sei also $t\in\omega{-}reach(compactify(S,M,\delta,I))$. Dann gibt es $(s,o)\in I$, $in,out\in M^\omega$ mit
(C.176) $\delta^\omega(s,in,t,out)$.
Nach Voraussetzung ist $\#o<\infty$. Gilt nun $s{=}t$, so ist nach Definition 4.7
(C.177) $t{=}s\in reach(compactify(S,M,\delta,I))$.
Ist neben o auch out endlich, also $\#out<\infty$, so folgt nach Definition 4.7 von *reach*:
(C.178) $t\in reach(compactify(S,M,\delta,I))$.
Ist $\#in{=}\infty$ oder $\#in{=}0$, so ist nach Definition 4.6 von δ^ω $s{=}t$, also wieder
(C.179) $t\in reach(compactify(S,M,\delta,I))$.
Sonst ist $s\neq t$, $1\leq\#in<\infty$ und $\#out{=}\infty$. Es gilt also $\delta^+(s,in,t,out)$. Dann gibt es eine Transitionssequenz $ts\in\Theta(\delta)$ mit
(C.180) $(source(ts),in(ts),dest(ts),out(ts)) = (s,in,t,out)$.
Durch Konstruktion von
(C.181) $ts' = \{(s',*,t',out) \mid \#out\neq\infty,\ s',t'\in S\}$ © ts
entsteht eine neue Transitionssequenz $ts'\in\Theta(\delta)$, da nach Definition von *compactify* für Transitionen mit unendlicher Ausgabe gilt $s'{=}t'$. Jede Transition in ts' hat nur endliche Ausgaben. Da auch ts' endlich ist, gilt:
(C.182) $\#out(ts') < \infty$.

Wegen $s \neq t$ ist *ts'* nicht die leere Sequenz, und es folgt mit $source(ts') = source(ts) = s$ und $dest(ts') = dest(ts) = t$ damit auch hier:
(C.183) $t \in reach(compactify(S,M,\delta,I))$.
Aus den Zeilen (C.177), (C.178), (C.179) und (C.183) folgt die Behauptung (C.175).
($\square$)

Beweis C.25 zu 6.10, Seite 103: (Einschränkung auf ω-erreichbare Zustandsmenge)
Totalisierungen: Wieder betrachten wir die Totalisierungen beider Automaten:
(C.184) $(S_c,M,\delta_c,I) \stackrel{def}{=} complete(S,M,\delta,I)$
(C.185) $(R_c,M,\delta_c',I) \stackrel{def}{=} complete(R,M,\delta',I)$.
mit
(C.186) $S_c = S \cup \{\pm\}$, $R_c = R \cup \{\pm\}$
(C.187) $\delta_c = \delta \cup \{(s,m,*,*) \in S_c \times M \times S_c \times M^\omega \mid \neg\delta(s,m,*,*)\}$
(C.188) $\delta_c' = \delta' \cup \{(s,m,*,*) \in R_c \times M \times R_c \times M^\omega \mid \neg\delta'(s,m,*,*)\}$.

R ist nach Proposition 4.9 abgeschlossen, weshalb für $s \in R$ und $m \in M$ gilt:
(C.189) $\exists t \in S, out.\ \delta(s,m,t,out)$
ist äquivalent mit
(C.190) $\exists t \in R, out.\ \delta(s,m,t,out)$
das ist nun äquivalent mit
(C.191) $\exists t \in R, out.\ \delta'(s,m,t,out)$.
Und weil weder $\delta(\pm,*,*,*)$ noch $\delta'(\pm,*,*,*)$ gilt, folgt insgesamt:
(C.192) $\forall s \in R_c, m.\ \delta(s,m,*,*) \Leftrightarrow \delta'(s,m,*,*)$.

δ_c und δ_c': Wir betrachten
(C.193) $\delta_c \cap R_c \times M \times R_c \times M^\omega$
Die Definition (C.187) von δ_c eingesetzt liefert:
(C.194) $= (\delta \cap R_c \times M \times R_c \times M^\omega) \cup \{(s,m,*,*) \in R_c \times M \times R_c \times M^\omega \mid \neg\delta(s,m,*,*)\}$
das wegen der Beziehung zwischen δ und δ' und wegen (C.192) gleich ist zu:
(C.195) $= \delta' \cup \{(s,m,*,*) \in R_c \times M \times R_c \times M^\omega \mid \neg\delta'(s,m,*,*)\}$
und das ist nach Definition (C.188) von δ_c' gleich mit:
(C.196) $= \delta_c'$.

Anwendung Proposition 5.13: Nach (C.193-C.196) gilt:
(C.197) $\delta_c' = \delta_c \cap R_c \times M \times R_c \times M^\omega$.
Mit (C.197) wird Proposition 5.13 anwendbar und es gilt:
(C.198) $[\![(S_c,M,\delta_c,I)]\!]^c = [\![(R_c,M,\delta_c',I)]\!]^c$.
Wegen (C.184) und (C.185) folgt insgesamt die Behauptung. ($\square$)

Beweis C.26 zu 6.11, Seite 104: (Entfernen nicht ω-erreichbarer Zustände)
Zweimalige Anwendung von Proposition 6.10 mit
(C.199) $R \stackrel{def}{=} \omega\!-\!reach(S,M,\delta,I) = \omega\!-\!reach(S',M,\delta',I)$
und
(C.200) $\delta_1 \stackrel{def}{=} \delta \cap R \times M \times R \times M^\omega = \delta' \cap R \times M \times R \times M^\omega$

liefert:
(C.201) $[\![(S,M,\delta,I)]\!] = [\![(R,M,\delta_1,I)]\!] = [\![(S',M,\delta',I)]\!]$
und damit die Behauptung. (□)

Beweis C.27 zu 6.12, Seite 104: (Erweiterung der Zustandsmenge)
Nach Wahl von (S',M,δ,I) kann der ursprüngliche Automat (S,M,δ,I) durch Entfernung der Zustände $(S' \setminus S)$ wieder erhalten werden. Deshalb folgt nach Proposition 6.11 sofort die Behauptung. (□)

Beweis C.28 zu 6.13, Seite 105: (Verfeinerung der Zustandsmenge)
Beweisidee: Der nachfolgende Beweis ist sehr komplex, wir erläutern deshalb hier die wesentlichen Schritte. Dazu beachte man auch die Bemerkungen am Ende des Beweises. Wir nutzen Beweisschema *(BP)* aus Abbildung B.3, um die Aussage für die Totalisierungen beider Automaten zu zeigen. Wir führen dazu α ein, das abstrakte auf konkrete stromverarbeitende Funktionen abbildet. Dann wird das Prädikat P, das eine gewisse Abstraktheitseigenschaft von Mengen konkreter stromverarbeitender Funktionen fordert, und β einführt, das auf der durch P gekennzeichneten Menge als Umkehrfunktion zu α wirkt. Nachdem alle Voraussetzungen von Beweisschema *(BP)* gezeigt sind, folgt die Aussage.

Totalisierung: Wir betrachten wieder die Totalisierungen beider Automaten
(C.202) $(S_c,M,\delta_c,I) \stackrel{def}{=} complete(S,M,\delta,I)$
(C.203) $(S_c',M,\delta_c',I') \stackrel{def}{=} complete(S',M,\delta',I')$.
und die zur Semantikdefinition gehörenden Funktionale σ beziehungsweise σ' definiert wie in Proposition 5.5. Zunächst erweitern wir α surjektiv auf $S_c' \rightarrow S_c$ durch die Definition
(C.204) $\alpha(\pm) = \pm$.

δ_c' und δ_c sind kongruent: Wir zeigen nun, daß zwischen δ_c' und δ_c dieselbe Kongruenz gilt wie zwischen δ und δ'. Es gilt nach Definition von δ_c' (siehe 5.10):
(C.205) $\delta_c' = \delta' \cup \{(s',m,*,*) \in S' \times M \times S' \times M^\omega \mid s' = \pm \vee \neg\delta'(s',m,*,*)\}$
oder umgeformt:
(C.206) $= \{(s',m,t',out) \mid \delta(\alpha\ s',m,\alpha\ t',out) \vee \alpha\ s' = \pm \vee \neg\delta(\alpha\ s',m,*,*)\}$
und durch Einsetzen von δ_c (siehe 5.10):
(C.207) $= \{(s',m,t',out) \mid \delta_c(\alpha\ s',m,\alpha\ t',out)\}$.
Gleichung (C.205-C.207) bedeutet also:
(C.208) $\forall s',t' \in S_c', m \in M, out \in M^\omega.\ \delta_c(\alpha\ s',m,\alpha\ t',out) \Leftrightarrow \delta_c'(s',m,t',out)$.
Mit (C.204) und (C.208) können wir ohne Einschränkung annehmen, daß die Automaten (S,M,δ,I) und (S',M,δ',I) bereits total sind.

Einführung von α: Wir führen folgendes Mengenfunktional α ein, das Mengen stromverarbeitender Funktionen, die mit abstrakten Zuständen parametrisiert sind, in solche übersetzt, die mit konkreten Zuständen parametrisiert sind:
(C.209) $\alpha \in \mathbb{P}(S \times M^\omega \stackrel{s}{\rightarrow} M^\omega) \rightarrow \mathbb{P}(S' \times M^\omega \stackrel{s}{\rightarrow} M^\omega)$
(C.210) $\alpha(X) = \{g \mid \forall s' \in S'.\ \exists f \in X.\ f(\alpha\ s') = g(s')\}$
Dabei steht die letzte Gleichung $f(\alpha\ s') = g(s')$ als currisierte Abkürzung für
(C.211) $\forall in.\ f(\alpha\ s',in) = g(s',in)$

Das Funktional α ist monoton und total. Es gilt nach Konstruktion:
(C.212) $\alpha(S \times M^\omega \xrightarrow{s} M^\omega) = (S' \times M^\omega \xrightarrow{s} M^\omega)$

$\alpha(\sigma.X) \subseteq \sigma'(\alpha.X)$: Sei dazu $g \in \alpha(\sigma.X)$. Dann gilt:
(C.213) $\forall s' \in S'.\ \exists f' \in (\sigma.X).\ f'(\alpha\ s') = g(s')$
Mit der Definition von σ folgt sofort:
(C.214) $\forall s' \in S'.\ g(s',\varepsilon) = \varepsilon$
Ebenfalls durch Expansion von σ folgt aus (C.213):
(C.215) $\forall s' \in S'\ \exists f'.$
(C.216) $(\forall m, s \in S\ \exists t, out; f \in X.\ \delta(s,m,t,out) \wedge \forall in.\ f'(s, m\hat{\ }in) = out\hat{\ }f(t,in)) \wedge$
(C.217) $f'(\alpha\ s') = g(s').$
Durch Instantiierung mit $s{=}\alpha.s'$ folgt daraus:
(C.218) $\forall s' \in S'\ \exists f'\ \forall m\ \exists t, out; f \in X.\ \delta(\alpha\ s', m, t, out) \wedge$
(C.219) $\forall in.\ g(s', m\hat{\ }in) = f'(\alpha\ s', m\hat{\ }in) = out\hat{\ }f(t,in)$
Die Quantifizierung $\exists f'$ wird entfernt. Wegen der Surjektivität von α auf S kann $t \in S$ ersetzt werden durch das gleichwertige $t' \in S'$ mit $t{=}\alpha.t'$. Es folgt aus (C.218-C.219):
(C.220) $\forall s' \in S', m\ \exists t' \in S', out; f \in X.\ \delta(\alpha\ s', m, \alpha\ t', out) \wedge$
(C.221) $\forall in.\ g(s', m\hat{\ }in) = out\hat{\ }f(\alpha\ t', in)$
Mit (C.208) folgt:
(C.222) $\forall s' \in S', m\ \exists t' \in S', out; f \in X.\ \delta'(s', m, t', out) \wedge \forall in.\ g(s', m\hat{\ }in) = out\hat{\ }f(\alpha\ t', in)$
Zu jedem $t' \in S'$ und $f \in X$ konstruieren wir $g' \in S' \times M^\omega \xrightarrow{s} M^\omega$ mit:
(C.223) $g' \stackrel{def}{=} \lambda s_1', in.\ f(\alpha\ s_1', in)$
Nach Konstruktion ist $g' \in \alpha.X$, und es gilt:
(C.224) $g'(t') = f(\alpha\ t')$
Mit (C.222) folgt daraus
(C.225) $\forall s' \in S', m\ \exists t' \in S', out; g' \in \alpha(X).\ \delta'(s', m, t', out) \wedge$
(C.226) $\forall in.\ g(s', m\hat{\ }in) = out\hat{\ }g'(t', in)$
und damit $g \in \sigma'(\alpha.X)$. Es gilt also:
(C.227) $\forall X.\ \alpha(\sigma.X) \subseteq \sigma'(\alpha.X)$

Abstraktionsprädikat P: Wir führen nun einen Abstraktionsoperator Δ wie folgt ein:
(C.228) $\Delta : \mathbb{P}(S' \times M^\omega \xrightarrow{s} M^\omega) \to \mathbb{P}(S' \times M^\omega \xrightarrow{s} M^\omega)$
(C.229) $\Delta(Y) \stackrel{def}{=} \{\ g \mid \forall s' \in S'\ \exists g' \in Y, s_2 \in S'.\ \alpha.s' = \alpha.s_2 \wedge g(s') = g'(s_2)\ \}$
Der Operator Δ ist ein Hüllenoperator. Es gilt also für jedes Y: $Y \subseteq \Delta(Y)$ und Δ ist idempotent.

Wir definieren nun das Prädikat P mit
(C.230) $P : \mathbb{P}(S' \times M^\omega \xrightarrow{s} M^\omega) \to \mathbb{B}$
(C.231) $P(Y) \stackrel{def}{\Leftrightarrow} (Y = \Delta(Y))$
Mengen stromverarbeitender Funktion, die die Eigenschaft P erfüllen, bieten eine gewisse Uniformität ihrer Funktionen in bezug auf die Äquivalenzklassenbildung unter α.

Offensichtlich gilt:
(C.232) $P(S' \times M^\omega \xrightarrow{s} M^\omega)$

Wir zeigen nun, daß σ' das Prädikat P respektiert.

<u>$P(Y) \Rightarrow P(\sigma'.Y)$</u>: Es gelte $P(Y)$, d.h. $Y{=}\Delta(Y)$. Sei $g{\in}\Delta(\sigma'.Y)$, dann ist zu zeigen, daß $g{\in}\sigma'(Y)$. Es gilt nach Definition (C.229)

(C.233) $\forall s'{\in}S'\ \exists g'{\in}(\sigma'.Y), s_2{\in}S'.\ \alpha.s'{=}\alpha.s_2 \wedge g(s'){=}g'(s_2).$

Die Definition (5.1) von σ' liefert:

(C.234) $\forall s'{\in}S'\ \exists g', s_2{\in}S'.\ \alpha.s'{=}\alpha.s_2 \wedge g(s'){=}g'(s_2) \wedge$

(C.235) $(\forall s_3.\ g'(s_3,\varepsilon) = \varepsilon) \wedge$

(C.236) $(\forall m, s_3.\exists t', out; g_2{\in}Y.\ \delta'(s_3,m,t',out) \wedge \forall in.\ g'(s_3,m\hat{\ }in) = out\hat{\ }g_2(t',in))$

Mit (C.234) und (C.235) folgt sofort

(C.237) $\forall s'.\ g(s',\varepsilon) = \varepsilon$

Mit der Instantiierung $s_3{=}s_2$ erhalten wir aus (C.234) und (C.236):

(C.238) $\forall s'{\in}S'\ \exists g', s_2{\in}S'.\ \alpha.s'{=}\alpha.s_2 \wedge \forall m\ \exists t', out; g_2{\in}Y.$

(C.239) $\delta'(s_2,m,t',out) \wedge \forall in.\ g(s',m\hat{\ }in) = g'(s_2,m\hat{\ }in) = out\hat{\ }g_2(t',in)$

Wegen (C.208) kann $\delta'(s_2,m,t',out)$ durch das äquivalente $\delta'(s',m,t',out)$ ersetzt werden. Es folgt:

(C.240) $\forall s'{\in}S', m.\ \exists t', out; g_2{\in}Y.\ \delta'(s',m,t',out) \wedge \forall in.\ g(s',m\hat{\ }in) = out\hat{\ }g_2(t',in)$

Gemeinsam mit (C.237) folgt $g{\in}\sigma'(Y)$, und es gilt

(C.241) $P(Y) \Rightarrow P(\sigma'.Y)$

<u>P ist zulässig</u>: Dazu sei $C{=}(C_i)$ eine Kette und $k{=}\sqcup C{=}\bigcap C$. Weiter gelte für jedes Kettenglied $P(C_i)$, das heißt $C_i{=}\Delta.C_i$. Sei

(C.242) $g{\in}\Delta.k = \Delta(\bigcap\Delta.C_i).$

Dann gilt nach Definition von Δ:

(C.243) $\forall s'{\in}S'\ \exists g'{\in}(\bigcap\Delta.C_i), s_2{\in}S'.\ \alpha.s'{=}\alpha.s_2 \wedge g(s'){=}g'(s_2)$

Wir expandieren die Definition von $\bigcap\Delta.C_i$ und erhalten:

(C.244) $\forall s'{\in}S'\ \exists g'; s_2{\in}S'.\ \alpha.s'{=}\alpha.s_2 \wedge g(s'){=}g'(s_2) \wedge$

(C.245) $\forall i.\ \forall s_3{\in}S'.\ \exists g_2{\in}C_i, s_4{\in}S'.\ \alpha.s_3{=}\alpha.s_4 \wedge g'(s_3){=}g_2(s_4)$

Wir ziehen $\forall i$ nach vorne und instantiieren $s_3{=}s_2$:

(C.246) $\forall i.\ \forall s'{\in}S'.\ \exists g'; s_2{\in}S'.\ \exists g_2{\in}C_i, s_4{\in}S'.\ \alpha.s'{=}\alpha.s_2{=}\alpha.s_4 \wedge$

(C.247) $g(s'){=}g'(s_2){=}g_2(s_4)$

verkürzt bedeutet dies:

(C.248) $\forall i.\ \forall s'{\in}S'.\ \exists g_2{\in}C_i, s_4{\in}S'.\ \alpha.s'{=}\alpha.s_4 \wedge g(s'){=}g_2(s_4)$

Damit folgt:

(C.249) $\forall i.\ g \in \Delta.C_i$

wegen $C_i{=}\Delta.C_i$ also: $g{\in}k$ und es gilt $P(k)$. Damit ist P zulässig.

<u>Einführung von β</u>: Wir definieren nun eine zu α umkehrende Funktion β:

(C.250) $\beta \in \mathbb{P}(S' \times M^\omega \xrightarrow{s} M^\omega) \to \mathbb{P}(S \times M^\omega \xrightarrow{s} M^\omega)$

(C.251) $\beta(Y) = \{f \mid \forall s'{\in}S'.\ \exists g{\in}Y.\ f(\alpha\ s') = g(s')\}$

Gilt $P(Y)$, so kann die Definition von β umformuliert werden. Wir betrachten die stromverarbeitenden Funktionen f, für die gilt:

(C.252) $\forall s{\in}S.\ \exists g{\in}Y, s'{\in}S'.\ \alpha.s'{=}s \wedge\ f(\alpha\ s') = g(s')$

und wählen $s_1' \in S'$ fest. Nach (C.252) gibt es zu $s{=}\alpha.s_1'$ ein $g \in Y$ und $s' \in S$ mit

(C.253) $s = \alpha.s_1' = \alpha.s'$; $g(s'){=}f(s)$

Wegen $P(Y)$ gibt es für $g \in Y$ ein $g' \in Y$, so daß

(C.254) $g(s'){=}g'(s_1')$

Mit (C.253) folgt damit

(C.255) $\forall s_1' \in S'.\ \exists g' \in Y.\ g'(s_1') = f(\alpha.s_1')$

und damit $f \in \beta(Y)$. Nach Definition von β gilt sogar die Gleichheit:

(C.256) $P(Y) \Rightarrow \beta(Y) = \{f \mid \forall s \in S.\ \exists g \in Y, s' \in S'.\ \alpha.s'{=}s \wedge\ f(\alpha\ s') = g(s')\ \}$

$\underline{P(Y) \Rightarrow\ Y \subseteq \alpha(\beta(Y))}$: Dazu gelte $P(Y)$ und es sei $g \in Y$. Wir definieren

(C.257) $M(g) \stackrel{def}{=} \{f \mid \forall s \in S.\ \exists s_2 \in S'.\ \alpha.s_2{=}s \wedge\ f(\alpha\ s_2) = g(s_2)\ \}$

Nach (C.256) ist

(C.258) $M(g) \subseteq \beta(Y)$

Nach Konstruktion von $M(g)$ gibt es zu jedem $s' \in S'$ ein $f_{s'} \in M(g)$ mit

(C.259) $f_{s'}(\alpha\ s') = g(s')$

Nach Definition von α ist

(C.260) $g \in \alpha(\beta(Y))$

genau dann, wenn

(C.261) $\forall s' \in S'.\ \exists f \in \beta(Y).\ f(\alpha\ s') = g(s')$

Zu gegebenem $s' \in S'$ erfüllt $f_{s'}$ jeweils Bedingung (C.261), weswegen mit (C.258) folgt, daß (C.260) erfüllt ist. Also folgt:

(C.262) $P(Y) \Rightarrow\ Y \subseteq \alpha(\beta(Y))$

$\underline{P(Y) \Rightarrow \beta(\sigma'.Y) \subseteq \sigma(\beta.Y)}$: Dazu gelte $P(Y)$ und es sei

(C.263) $f \in \beta(\sigma'.Y)$

Nach Definition von β ist:

(C.264) $\forall s' \in S'.\ \exists g \in (\sigma'.Y).\ f(\alpha\ s') = g(s')$

Die Expansion der Definition von σ' liefert:

(C.265) $\forall s' \in S'.\ \exists g.\ f(\alpha\ s') = g(s') \wedge$

(C.266) $\quad (\forall s_2.\ g(s_2,\varepsilon) = \varepsilon) \wedge$

(C.267) $\quad \forall m, s_2. \exists t', out; g' \in Y.\ \delta'(s_2,m,t',out) \wedge \forall in.\ g(s_2, m\hat{\ }in) = out\hat{\ }g'(t',in)$

Mit (C.265) und (C.266) folgt sofort:

(C.268) $\forall s \in S.\ f(s,\varepsilon) = \varepsilon$

Seien nun $m \in M$ und $s \in S$ fest. Dann gibt es dazu ein $s' \in S'$ mit $\alpha.s'{=}s$. Nach (C.267) gibt es $t' \in S', out; g' \in Y$, so daß

(C.269) $\delta'(s',m,t',out) \wedge \forall in.\ f(\alpha.s', m\hat{\ }in) = out\hat{\ }g'(t',in)$

Wir wählen nun ein $f' \in (S \times M^\omega \stackrel{s}{\rightarrow} M^\omega)$ mit

(C.270) $\forall s_1 \in S.\ \exists s_1' \in S'.\ \alpha.s_1'{=}s_1 \wedge f'(s_1) = g'(s_1')$

indem wir zu jedem $s_1 \in S$ ein s_1' frei auswählen und für $s_1{=}\alpha.t'$ genau t' wählen. Nach Konstruktion ist wegen (C.256) $f' \in \beta(Y)$. Es gilt weiterhin $f'(\alpha\ t'){=}g'(t')$. In (C.269) eingesetzt liefert dies:

(C.271) $\delta'(s',m,t',out) \wedge \forall in.\ f(\alpha.s', m\hat{\ }in) = out\hat{\ }f'(\alpha\ t',in)$

Mit (C.268) und (C.271) folgt nun $f \in \sigma(\beta.Y)$ und damit diese Teilbehauptung:

(C.272) $P(Y) \Rightarrow \beta(\sigma'.Y) \subseteq \sigma(\beta.Y)$

$\alpha(fix.\sigma) = (fix.\sigma')$: Wir haben nun alle Voraussetzungen für das Beweisschema *(BP)* definiert. *(BP.d)* ist mit (C.227) gezeigt, *(BP.h)* ist identisch mit (C.232) *(BP.i)* folgt mit (C.241), *(BP.j)* steht in (C.262) und *(BP.k)* in (C.272). Damit gilt:
(C.273) $\alpha(fix.\sigma) = (fix.\sigma')$

$[\![(S,M,\delta,I)]\!]^c \subseteq [\![(S',M,\delta',I')]\!]^c$: Wir betrachten dazu zunächst α. Die Expansion der Definition von α liefert:
(C.274) $\alpha(fix.\sigma) = \{g \mid \forall s'\in S'.\ \exists f\in(fix.\sigma).\ f(\alpha\ s') = g(s')\}$
Für $f\in(fix.\sigma)$ können wir folgende Funktion
(C.275) $g' \stackrel{def}{=} \lambda s',in.f(\alpha\ s',in)$
definieren, für die gilt: $g\in\alpha(fix.\sigma)$. Gemeinsam mit (C.273) gilt:
(C.276) $\forall f\in(fix.\sigma),s'\in S'.\ \exists g\in(fix.\sigma').\ f(\alpha\ s') = g(s')$

Ist nun $g\in[\![(S,M,\delta,I)]\!]^c$, dann gilt wegen der Definition von $[\![(S,M,\delta,I)]\!]^c$ in 5.4:
(C.277) $\exists h\in(fix.\sigma),(s_i,out_i)\in I.\ \forall in.\ g(in) = out_i\,\hat{}\,h(s_i,in)$.
Zu s_i gibt es wegen wegen der Surjektivität von α ein $s_i'\in S'$ mit $\alpha.s_i'=s_i$. Zu $h\in(fix.\sigma)$ und $s_i'\in S'$ gibt es nach (C.276) ein $h'\in(fix.\sigma')$ mit:
(C.278) $\forall in.\ h(\alpha\ s_i',in) = h'(s_i',in)$
Damit folgt aus (C.277) und der Definition von I':
(C.279) $\exists h'\in(fix.\sigma'),(s_i',out_i)\in I'.\ \forall in.\ g(in) = out_i\,\hat{}\ h'(s_i',in)$
also $g\in[\![(S',M,\delta',I')]\!]^c$.

$[\![(S,M,\delta,I)]\!]^c \supseteq [\![(S',M,\delta',I')]\!]^c$: Die Expansion von α in Aussage (C.273) liefert
(C.280) $\forall g\in(fix.\sigma'),s'\in S'.\ \exists f\in(fix.\sigma).\ f(\alpha\ s') = g(s')$

Sei nun $g\in[\![(S',M,\delta',I')]\!]^c$, dann gilt nach Definition:
(C.281) $\exists h'\in(fix.\sigma'),(s_i',out_i)\in I'.\ \forall in.\ g(in) = out_i\,\hat{}\,h'(s_i',in)$.
Nach (C.280) folgt die Existenz eines $h\in(fix.\sigma)$ das gemeinsam mit der Einführung von $s_i=\alpha.s_i'$ und nach Definition von I' gilt:
(C.282) $\exists h\in(fix.\sigma),(s_i,out_i)\in I.\ \forall in.\ g(in) = out_i\,\hat{}\,h(s_i,in)$.
Also $g\in[\![(S,M,\delta,I)]\!]^c$ und es folgt:
(C.283) $[\![(S,M,\delta,I)]\!]^c \supseteq [\![(S',M,\delta',I')]\!]^c$
also insgesamt die Behauptung dieser Proposition. (□)

Einige Anmerkungen zu diesem Beweis: Der Beweis zur Gleichheit der Semantik bei der Verfeinerung von Zuständen ist sehr komplex. Um dennoch die Übersicht über diesen Beweis zu ermöglichen, wurde er in eine Reihe von Einzelbeweisen zerlegt und diese mit einem allgemeinen Beweisschema *(BP)* zum Gesamtbeweis zusammengesetzt. Eines der wesentlichen Probleme bei diesem Beweis besteht darin, daß zwar leicht eine Funktion α gefunden werden konnte, die eine Inklusion $\alpha(fix.\sigma) \subseteq (fix.\sigma')$ bewerkstelligt. Diese Inklusion ist leider zuwenig, da sie nur besagt, daß bei der Verfeinerung keine Funktionen aus der Semantik entfernt werden. Wir benötigen jedoch vor allem die umgekehrte Inklusion.

α ist für unendliche Zustandsmengen nicht stetig und $\alpha.X\supseteq K$ nicht zulässig in X. Daher kann Beweisschema *(A)* nicht angewendet werden. Eine modifizierte Version von Beweisschema *(A)*, bei der auf die Bedingung *(A.f)*, der Forderung nach Zulässigkeit des Prädikats $(\alpha.X\supseteq K)$, verzichtet wird, konnte nicht gefunden werden.

Es lies sich auch keine Umkehrfunktion β finden, die alle notwendigen Eigenschaften für Beweisschema *(B)* erfüllt. Mehrere Kandidaten, die sich im wesentlichen in der Reihenfolge der Quantoren und der Quantifizierung über abstrakte und konkrete Zustände unterscheiden, haben jeweils eine der Bedingungen von *(B)* nicht erfüllen können.

Bei diesen Untersuchungen hat sich jedoch gezeigt, daß unter einer zusätzlichen Annahme $P(Y)$ für die betrachteten Mengen Y mehrere der oben diskutierten Varianten von β dieselben Bildmengen haben. Annahme P definiert dabei eine bestimmte Abstraktheitseigenschaft. Sie fordert, daß für Mengen stromverarbeitender Funktionen, die mit konkreten Zuständen parametrisiert sind, jeder konkrete Zustand einer Äquivalenzklasse dasselbe Verhalten besitzt. Dies ist in Prädikat P aus (C.230) charakterisiert. Aufgrund dieser Abstraktheitscharakterisierung von P ist P sogar eine zulässige Eigenschaft.

Gilt nun P für eine Menge Y, so kann für β jede der beiden Charakterisierungen (C.251) oder (C.256) verwendet werden. Unter der Einschränkung $P(Y)$ gelten jetzt alle für Beweisschema *(B)* notwendigen Aussagen. Dies hat letztendlich zu dem erweiterten Beweisschema *(BP)* geführt, das bei diesem Beweis verwendet wurde.

Als Nebeneffekt dieses Beweises zeigt sich, daß die folgende Abstraktheitseigenschaft (C.288) für mit Zuständen parametrisierte Funktionen gilt. Der Beweis hierfür folgt aus der Verwendung von $\sigma' = \sigma$ und der Identität für α. Sei der Hüllenoperator Δ wie folgt definiert:

(C.284) $\Delta : \mathbb{P}(S \times M^\omega \overset{s}{\to} M^\omega) \to \mathbb{P}(S \times M^\omega \overset{s}{\to} M^\omega)$

(C.285) $\Delta(X) \overset{def}{=} \{ f \mid \forall s{\in}S\ \exists g{\in}X.\ \forall in.\ f(s,in){=}g(s,in) \}$

dann gelten folgende Aussagen:

(C.286) $\forall X.\ X \subseteq \Delta.X$

(C.287) $\forall X.\ \Delta.X = \Delta(\Delta.X)$

(C.288) $\forall X.\ \Delta(fix.\sigma) = (fix.\sigma)$

(C.289) $adm(\lambda X.\ X = \Delta.X)$

Beweis C.29 zu 6.15, Seite 107: (Schnittstellenverfeinerung)
Die erste Aussage folgt direkt aus der Definition der Schnittstellenverfeinerung. Die zweite Aussage folgt mit der Einschränkung $M_{in}' {=} M_{in}$ aus Definition 6.1. (□)

Beweis C.30 zu 6.16, Seite 107: (Erweiterung der Eingabemenge)
Totalisierung: Wie üblich betrachten wir die Totalisierungen beider Automaten:

(C.290) $(S_c,M_{in},M_{out},\delta_c,I) \overset{def}{=} complete(S,M_{in},M_{out},\delta,I)$

(C.291) $(S_c,M_{in}',M_{out},\delta_c,I) \overset{def}{=} complete(S,M_{in}',M_{out},\delta,I)$.

mit

(C.292) $S_c = S \cup \{\pm\}$

(C.293) $\delta_c = \delta \cup \{(s,m,*,*) \mid \neg\delta(s,m,*,*),\ m{\in}M_{in}\}$

(C.294) $\delta_c' = \delta \cup \{(s,m,*,*) \mid \neg\delta(s,m,*,*),\ m{\in}M_{in}'\}$

Die Zeilen (C.293) und (C.294) zeigen, daß folgende Beziehung gilt:

(C.295) $\delta_c = \delta_c' \cap S{\times}M_{in}{}^\omega{\times}S{\times}M_{out}{}^\omega$.

Seien σ und σ' die zur Semantik von $(S_c,M_{in},M_{out},\delta_c,I)$ bzw. $(S_c,M_{in}',M_{out},\delta_c',I)$ gehörenden Funktionale aus Proposition 5.5,(5.1). Wir definieren die Verbindung zwischen beiden Semantiken mit einer Funktion α, wie folgt:

(C.296) $\alpha \in (S_c \times (M_{in}')^\omega \stackrel{s}{\to} M_{out}{}^\omega) \to (S_c \times M_{in}{}^\omega \stackrel{s}{\to} M_{out}{}^\omega)$

(C.297) $\alpha.f = f|_{S\times M_{in}{}^\omega}$

α wird punktweise zu einem Mengenfunktional erweitert. Die Mengenfunktion α ist auf der Potenzmenge $\mathbb{P}(S \times (M_{in}')^\omega \stackrel{s}{\to} M_{out}{}^\omega)$ total und surjektiv.

$\underline{\alpha(\sigma'.X) \subseteq \sigma(\alpha.X)}$: Sei $g{\in}\alpha(\sigma'.X)$, das heißt,

(C.298) $\exists f.\ g{=}\alpha.f \wedge (\forall s.f(s,\varepsilon) = \varepsilon) \wedge$

(C.299) $\forall m{\in}M_{in}',s.\exists t,out;f'{\in}X.\ \delta_c'(s,m,t,out) \wedge \forall in{\in}(M_{in}')^\omega.\ f(s,m\hat{\ }in) = out\hat{\ }f'(t,in)$

Aus (C.298) folgt sofort:

(C.300) $\forall s.\ g(s,\varepsilon) = \varepsilon.$

Mit (C.299) folgt außerdem

(C.301) $\forall m{\in}M_{in},s.\ \exists t,out;f'{\in}X.\ \delta_c'(s,m,t,out) \wedge \forall in{\in}M_{in}{}^\omega.$

(C.302) $g(s,m\hat{\ }in) = f(s,m\hat{\ }in) = out\hat{\ }f'(t,in)$

Mit $m{\in}M_{in}$ und (C.295) kann δ_c' durch δ_c ersetzt werden. Weil auch $in{\in}M_{in}{}^\omega$ ist, kann f' durch $(\alpha\ f')$ ersetzt werden:

(C.303) $\forall m{\in}M_{in},s\ \exists t,out;f'{\in}X.\ \delta_c(s,m,t,out) \wedge \forall in{\in}M_{in}{}^\omega.$

(C.304) $g(s,m\hat{\ }in) = out\hat{\ }(\alpha\ f')(t,in)$

Gemeinsam mit (C.300) folgt also:

(C.305) $g{\in}\sigma(\alpha.X).$

Damit gilt:

(C.306) $\alpha(\sigma'.X) \subseteq \sigma(\alpha.X)$

Beweisschema $(B_\subseteq)$ liefert die Behauptung:

(C.307) $\alpha(fix.\sigma') \subseteq (fix.\sigma).$ $(\Box)$

Beweis C.31 zu 6.20, Seite 111: (Eigenschaften von $[\![.]\!]^t$)

<u>Aussage 1</u> folgt aufgrund Teil 2 von Satz 5.16, der leicht zu erkennenden Tatsache, daß für jede Funktion f gilt $\rho(f){\neq}\emptyset$.

<u>Aussage 2</u> ist ein Spezialfall von Aussage 3.

<u>Aussage 3</u> Es gelte:

(C.308) $[\![A']\!]|_{M_{in}{}^\omega} \subseteq [\![A]\!]$

Nach den Definitionen 5.30 und 5.31 ist

(C.309) $[\![A]\!]^t = \{\ g{\in}M_{in}{}^{\overline{\infty}} \stackrel{p}{\to} M_{out}{}^{\overline{\infty}} \mid \exists f{\in}[\![A]\!].\ \forall in{\in}M_{in}{}^{\overline{\infty}}.\ \diamond(g.in) = f(\diamond in)\ \}$

(C.310) $[\![A']\!]^t = \{\ g{\in}(M_{in}')^{\overline{\infty}} \stackrel{p}{\to} M_{out}{}^{\overline{\infty}} \mid \exists f{\in}[\![A']\!].\ \forall in{\in}(M_{in}')^{\overline{\infty}}.\ \diamond(g.in) = f(\diamond in)\ \}$

Für $g{\in}[\![A']\!]^t$ gilt also

(C.311) $\exists f{\in}[\![A']\!].\ \forall in{\in}(M_{in}')^{\overline{\infty}}.\ \diamond(g.in) = f(\diamond in)$

Nach (C.308) gibt es ein $h{\in}[\![A]\!]$ mit $h{=}f|_{M_{in}{}^\omega}$. Aus (C.311) folgt:

(C.312) $\forall in{\in}M_{in}{}^{\overline{\infty}}.\ \diamond(g.in) = h(\diamond in)$

Damit ist $g|_{M_{in}{}^{\overline{\infty}}}{\in}[\![A']\!]^t$ und es folgt die Behauptung. $(\Box)$

C.5 Objektmodell

Beweis C.32 zu 7.5, Seite 124: (Verfeinerungsbeziehung für Klassenbeschreibungen)
Wenn die angegebene Bedingung

(C.313) $c'=c \wedge sup'\subseteq sup \wedge meth'\subseteq meth \wedge attr'\subseteq attr \wedge card'\geq card$

gilt, so folgt für $sys\in[\![(c,sup,meth,attr,card)]\!]^{cl}$ nach Definition 7.4 sofort

(C.314) $sys \in [\![(c',sup',meth',attr',card')]\!]^{cl}$

Also gilt

(C.315) $(c,sup,meth,attr,card) \models (c',sup',meth',attr',card')$. (□)

Beweis C.33 zu 7.9, Seite 131: (Verfeinerungsbeziehung für Objektmodelle)
Korrektheit der Ableitungsbeziehung für Rollen: Folgt sofort aus der Definition. Für Rollen $ro,ro'\in CN\times\langle var\rangle^{\phi}\times CN$ gilt:

(C.316) $ro\vdash ro' \;\Rightarrow\; ro\models ro'$

Korrektheit der Ableitungsbeziehung für Relationen: Folgt ebenfalls sofort aus der Definition. Für Datenbeziehungen $r,r'\in R$ gilt:

(C.317) $r\vdash r' \;\Rightarrow\; r\models r'$

Korrektheit der Aussage: Sei $sys\in\mathcal{SM}$ ein System, für das gilt:

(C.318) $sys\in[\![(C,S,R)]\!]^{om}$,

und (C',S',R') erfülle die angegebenen Bedingungen:

(C.319) $C'\subseteq C \wedge S'\subseteq S \wedge \forall r\in R'.\exists r\in R.\; r\vdash r'$

Für die Menge der Klassennamen gilt mit Definition 7.8:

(C.320) $C'\subseteq C\subseteq CN_{sys}$.

Für die Menge der Subklassenbeziehungen S' folgt mit Definition 7.8:

(C.321) $\forall(c,d)\in S'.\; c \sqsubseteq_{sys} d$.

Sei nun $r'\in R'$ eine Relation. Dann gibt es nach (C.319) eine Relation $r\in R$ mit:

(C.322) $r \vdash r'$

Nach (C.317) gilt:

(C.323) $sys \in [\![r']\!]^{rel}$.

Aus (C.320), (C.321) und (C.323) folgt nun mit Definition 7.8:

(C.324) $sys\in[\![(C',S',R')]\!]^{om}$.

und mit Definition 2.4 die Behauptung. (□)

Definition C.34 (Integration von Datenbeziehungen)
Wir definieren folgenden Operator $.\bowtie.$ *zur Fusion zweier Relationen und ein Prädikat* $.\stackrel{?}{\bowtie}.$*, das die Fusionierbarkeit beschreibt:*

$$
\begin{array}{l}
r\bowtie r' \stackrel{def}{=} \texttt{if } r_{\overleftarrow{ro}}\stackrel{?}{\bowtie}r'_{\overleftarrow{ro}} \wedge r_{\overrightarrow{ro}}\stackrel{?}{\bowtie}r'_{\overrightarrow{ro}} \\
\quad\quad \texttt{then } (c,rc'',r_{rel},d,rd'') \\
\quad\quad\quad\quad \texttt{where } (c,rc'',d) = r_{\overleftarrow{ro}}\bowtie r'_{\overleftarrow{ro}} \;;\; (*,rd'',*) = r_{\overrightarrow{ro}}\bowtie r'_{\overrightarrow{ro}} \\
\quad\quad \texttt{else } (c,rc'',r_{rel},d,rd'') \\
\quad\quad\quad\quad \texttt{where } (c,rc'',d) = r_{\overleftarrow{ro}}\bowtie r'_{\overrightarrow{ro}} \;;\; (*,rd'',*) = r_{\overrightarrow{ro}}\bowtie r'_{\overleftarrow{ro}} \\
r\stackrel{?}{\bowtie}r' \stackrel{def}{\Leftrightarrow} r_{rel}=r'_{rel} \wedge ((r_{\overleftarrow{ro}}\stackrel{?}{\bowtie}r'_{\overleftarrow{ro}} \wedge r_{\overrightarrow{ro}}\stackrel{?}{\bowtie}r'_{\overrightarrow{ro}}) \vee (r_{\overleftarrow{ro}}\stackrel{?}{\bowtie}r'_{\overrightarrow{ro}} \wedge r_{\overrightarrow{ro}}\stackrel{?}{\bowtie}r'_{\overleftarrow{ro}}))
\end{array}
$$

Eine Menge von Relationen wird fusioniert, indem paarweise solange Fusionen stattfinden, bis keine mehr möglich ist:

$$fuseRELs : \mathbb{P}(R) \rightarrow \mathbb{P}(R)$$

$$\begin{aligned} fuseRELs\ rs = \texttt{if}\ & \exists r,r' \in rs.\ r \neq r' \wedge r \overset{?}{\bowtie} r' \\ & \texttt{then}\ fuseRELs\ (\{r \bowtie r'\} \cup rs \setminus \{r,r'\}) \\ & \texttt{else}\ rs. \end{aligned} \qquad (\square)$$

Die Fusion einer Menge von Datenbeziehungen ist durch einen Pseudoalgorithmus beschrieben, dessen Ergebnis nicht determiniert ist, da es von der Auswahl der jeweils fusionierten Relationen abhängt. Die Komplexität der Fusion von Relationen besteht darin, daß die Rollen einer Relation miteinander fusioniert werden und die Relation keine Orientierung besitzt, also (c,rc,rel,d,rd) als äquivalent zu (d,rd,rel,c,rc) betrachtet wird.

Proposition C.35 (Integration von Datenbeziehungen)

1. *Für die Integrierbarkeit zweier Relationen gilt:*

$$\forall r,r' \in R.\ r \overset{?}{\bowtie} r' \Leftrightarrow \exists r''.\ r \vdash r'' \wedge r' \vdash r''$$

2. *Für die Integration zweier Relationen gilt die Informationserhaltung:*

$$\forall r,r' \in R.\ r \overset{?}{\bowtie} r' \Rightarrow \{r,r'\} \models\!\!\dashv r \bowtie r'$$

3. *Für die Integration von Mengen von Relationen gilt:*

$$\forall rs \subseteq R.\ rs \models\!\!\dashv fuseRELs(rs)$$ ($\square$, **Beweis siehe** C.36)

Beweis C.36 zu C.35, Seite 235: (Integration von Datenbeziehungen)
Aussage 1, $\Rightarrow$-Richtung: Seien zwei Relationen $r,r' \in R$ gegeben. Aus der Integrierbarkeit $r \overset{?}{\bowtie} r'$ folgt die Gleichheit der beiden Relationsnamen

(C.325) $r_{rel} = r'_{rel}$

und die paarweise Fusionierbarkeit der Rollen. Die Fusionierbarkeit der Rollen charakterisiert aber die Existenz jeweils einer Rolle, die Ableitung der zu fusionierenden Rollen ist. Mit Definition der Ableitungsbeziehung für Relationen folgt damit die eine Implikation.

Aussage 1, Rückrichtung: Die Umkehrung folgt analog.

Aussage 2: Die Fusion zweier Relationen erfolgt unter Fusion derer Rollen. Da diese semantikerhaltend ist, folgt die Semantikerhaltung für fusionierbare Relationen.

Aussage 3: Die Funktion *fuseRELs* ist als iterierte Anwendung von $.\bowtie.$ definiert. Weil für jede korrekte Anwendung gilt:

(C.326) $(\{r \bowtie r'\} \cup rs \setminus \{r,r'\}) \models\!\!\dashv rs$

und $.\models\!\!\dashv.$ transitiv ist, ist auch das Ergebnis von *fuseRELs* semantikerhaltend. ($\square$)

Beweis C.37 zu 7.11, Seite 132: (Integration von Objektmodellen)
Sei
(C.327) $oms = \{om_i = (C_i,S_i,R_i) \mid i{\in}I\}$
(C.328) $om = (C,S,R) = fuseOMs\ oms$
Sei $sys{\in}\mathcal{SM}$ ein System $sys{\in}[\![oms]\!]$. Dies ist nach Definition 2.2 äquivalent mit:
(C.329) $\forall i{\in}I.\ sys \in [\![om_i]\!]^{om}$.
Expansion der Definition 7.8 liefert die Äquivalenz mit:
(C.330) $\forall i{\in}I.\ C_i \subseteq CN_{sys} \wedge (\forall (c,d){\in}S_i.\ c \sqsubseteq_{sys} d) \wedge \forall r{\in}R_i.\ sys{\in}[\![r]\!]^{rel}$
Dies ist äquivalent zu:
(C.331) $C \subseteq CN_{sys} \wedge (\forall (c,d){\in}S.\ c \sqsubseteq_{sys} d) \wedge \forall r{\in}R.\ sys{\in}[\![r]\!]^{rel}$
und wegen Proposition C.35, Aussage 3. äquivalent zu:
(C.332) $sys \in [\![fuseOMs\ oms]\!]$.
Die Kette der Äquivalenzumformungen zeigt die Behauptung. ($\Box$)

C.6 Automatendokumente

Beweis C.38 zu 8.4, Seite 148: (Eigenschaften der Transformation *automaton*)
Sei $(c,ak,S,\Lambda,\delta,I){\in}\mathcal{DOC}_a$ ein Automatendokument, $sys{\in}\mathcal{SM}$ ein System des Systemmodells und $id{\in}ID_{sys}$ mit $id\ isof_{sys}\ c$ gegeben und
(C.333) $(S',M_{in}',M_{out}',\delta',I') = automaton_{sys,id}(c,ak,S,\Lambda,\delta,I)$
der erzeugte buchstabierende Automat.

Aussage 1: Folgt aus Aussage 2, weil $S{\neq}\emptyset$.

Aussage 2: Sei $s{\in}S$ ein Automatenzustand. Bedingung VB1 liefert zunächst die Existenz einer Belegung β_1, für die gilt:
(C.334) $\exists\beta_1{\in}BEL.(sys,\beta_1){\models}\Lambda s$
Aufgrund der Uniformität der Variable `self` existiert eine Belegung $\beta_2{\in}BEL$ mit
(C.335) $\beta_2.\mathtt{self} = id \wedge (sys,\beta_2){\models}\Lambda s$.
Ohne Einschränkung sei $att \subseteq \bullet\beta_2$. Damit definieren wir
(C.336) $s' \stackrel{def}{=} \beta_2|_{att} \in S'$.
und es ist der erste Teil der Aussage
(C.337) $\forall s{\in}S.\ \exists s'{\in}S'.\ (sys,s'){\models}\Lambda s$
gezeigt. Der zweite Teil der Aussage folgt unmittelbar aus der Bedingung VB2.

Aussage 3: Sei $T{\in}\delta$ eine Transition, $s{\in}S'$ eine Belegung der Attribute und $m{\in}M_{in}'$ eine Eingabenachricht mit
(C.338) $(sys,s{+}(\mathtt{in}{\mapsto}m)){\models}T_{pre}$.
Wir definieren $\beta_1 \stackrel{def}{=} (s{+}(\mathtt{in}{\mapsto}m))$, so daß $\beta_1{\in}Bel_{attr(\Lambda)\cup\{in\}}$.
Wir konstruieren nun einen Zielzustand und eine Ausgabe, die nach Schalten der Transition T eingenommen werden kann. Nach Bedingung VB3 existiert $\beta_2{\in}BEL$ mit
(C.339) $(sys,\beta_1{+}\beta_2){\models}T_{post}$

Aufgrund der in Bedingung KB2 postulierten Einschränkung der Benutzung der Variable `self`' gilt außerdem:
(C.340) $\beta_2.\texttt{self}' = id.$
Ohne Einschränkung gelte
(C.341) $attr(\Lambda)' \cup \{\texttt{out}\} \subseteq \bullet\beta_2.$
Gemeinsam mit (C.340) folgt damit
(C.342) $\beta_2|_{attr(\Lambda)'} \in (S')' \wedge \beta_2.\texttt{out} \in (M_{out}')^\omega,$
so daß Zielzustand und Ausgabe der Transition gefunden sind. (□)

Beweis C.39 zu 8.6, Seite 153: (Erweiterung der Menge der Automatenzustände)
Seien gegeben
(C.343) $A \stackrel{def}{=} (c,ak,S,\Lambda,\delta,I) \in \mathcal{DOC}_a$
(C.344) $A_2 \stackrel{def}{=} (c,ak,S\cup\bullet\Lambda_2,\ \Lambda+\Lambda_2,\ \delta,I)$
und ein System
(C.345) $sys \in [\![A_2]\!]^a$

Gültigkeit der Bedingungen VB1–VB3: Bedingung VB3 gilt unverändert, da die Transitionsrelation δ nicht verändert wurde.

Für jeden Zustand $s\in\bullet\Lambda_2$ sichert die Menge $\Gamma_{addS}(\Lambda,A)$ nach Definition, daß gilt
(C.346) $sys \models \exists\overline{\alpha}.\ \Lambda_2 s$
und damit
(C.347) $\exists\beta\in BEL.\ (sys,\beta) \models \Lambda_2 s.$
Bedingung VB1 ist für Zustände $s\in S$ weiterhin gesichert, da sich deren Zustandsprädikate nicht verändert haben. Insgesamt folgt damit, daß Bedingung VB1 weiter gilt.

Für die Disjunktheit der Automatenzustände ist ein Test mit allen bisherigen Zuständen notwendig. Ist $s\in\bullet\Lambda_2$ ein neuer Zustand und $t\in S\cup\bullet\Lambda_2$ ein beliebiger Zustand mit $s\neq t$, so wird durch $\Gamma_{addS}(\Lambda,A)$ gesichert, daß
(C.348) $sys \models \neg(\Lambda_2 s \wedge (\Lambda+\Lambda_2)t).$
Damit folgt zunächst die Disjunktheit der Datenzustände für diese Paare von Automatenzuständen (s,t). Weil für $s,t\in S$ diese Disjunktheit bereits im alten Automaten gegolten hat, folgt damit die Gültigkeit von Bedingung VB2 und damit der zweite Teil dieser Aussage.

Verfeinerungseigenschaft der Transformation: Wir betrachten die jeweils mit *automaton* erzeugten buchstabierenden Automaten. Weil die neu hinzugekommenen Automatenzustände $\bullet\Lambda_2$ weder in I noch δ vorkommen und S in der Transformation *automaton* nicht verwendet wird, folgt sofort:
(C.349) $automaton_{sys,id}(A) = automaton_{sys,id}(A_2)$
Damit folgt insgesamt:
(C.350) $[\![A]\!]^a = [\![A_2]\!]^a$
woraus unsere Behauptung folgt. (□)

Beweis C.40 zu 8.8, Seite 158: (Verfeinerung der Menge der Automatenzustände)
Seien
(C.351) $A_1 \stackrel{def}{=} (c,ak,S,\Lambda,\delta,I) \in \mathcal{DOC}_a$
(C.352) $A_2 \stackrel{def}{=} (c,ak,\bullet\Lambda_2,\Lambda_2,\delta_2,I_2)$
gegeben, wobei δ_2 und I_2 wie in 8.7 definiert sind. Die Bedingungen VB1 und VB2 werden durch die Beweisverpflichtungen gesichert.

Verfeinerung einer Transition: Wir betrachten die Transition $T{=}(s,ci,t,co){\in}\delta$, bei der der Quellzustand s durch $(s_i)_{i\leq m}$ und der Zielzustand t durch $(t_j)_{j\leq n}$ verfeinert wird. Die Randfälle $n{=}1$ und $m{=}1$ sind erlaubt. Die Beweisverpflichtungen sichern:
(C.353) $\Lambda s \Leftrightarrow \bigvee_i \Lambda_2 s_i$
(C.354) $\Lambda t \Leftrightarrow \bigvee_j \Lambda_2 t_j$.
Es entstehen die neuen Transitionen T^{ij} der Form:
(C.355) $T^{ij}_{pre} = \Lambda_2 s_i \wedge ci \wedge \exists \overline{\alpha}`,\mathtt{out}.\ (\Lambda_2 t_j)` \wedge co$
(C.356) $T^{ij}_{post} = (\Lambda_2 t_j)` \wedge co.$
Nach Konstruktion ist offensichtlich die Enabledness (VB3) jeder Transition T^{ij} gesichert.

Überdeckung des Zielbereichs: Nach Definition von T_{post} gilt:
(C.357) $T_{post} \Leftrightarrow co \wedge (\Lambda t)`$
Das ist wegen (C.354) äquivalent mit:
(C.358) $\Leftrightarrow \bigvee_j (co \wedge (\Lambda_2 t_j)`)$
und das ist wegen (C.356) äquivalent mit
(C.359) $\Leftrightarrow \bigvee_{ij} T^{ij}_{post}$.

Überdeckung des Schaltbereichs: Nach Definition von T_{pre} gilt:
(C.360) $T_{pre} \Leftrightarrow ci \wedge \Lambda s$
Das ist wegen (C.353) und der Enabledness (VB3) äquivalent mit:
(C.361) $\Leftrightarrow ci \wedge \bigvee_i (\Lambda_2 s_i) \wedge \exists \overline{\alpha},\mathtt{out}.\ T_{post}$
und das ist wegen (C.357-C.359) äquivalent mit
(C.362) $\Leftrightarrow \bigvee_{ij} (ci \wedge \Lambda_2 s_i \ \wedge \exists \overline{\alpha},\mathtt{out}.\ T^{ij}_{post})$
und wegen (C.355) gleichwertig mit:
(C.363) $\Leftrightarrow \bigvee_{ij} T^{ij}_{pre}$.

Äquivalenz der Transitionen: Aus (C.357-C.359) folgt die Äquivalenz der Nachbedingungen, aus (C.361-C.363) der Vorbedingungen. Wegen $T{=}T_{pre}{\wedge}T_{post}$ folgt damit:
(C.364) $T \Leftrightarrow \bigvee_{ij} T^{ij}$
Analoges gilt für verfeinerte Initialelemente.
(C.365) $E \Leftrightarrow \bigvee_i E_i$

Daraus folgt wegen der Definition 8.3 von *automaton*, daß für alle Systeme $sys{\in}[\![A_2]\!]^a$ und $id{\in}ID_{sys}$ gilt:
(C.366) $automaton_{sys,id}(A_1) = automaton_{sys,id}(A_2)$
Daraus folgt die Behauptung. (□)

Beweis C.41 zu 8.9, Seite 159: (Transformation vom Typautomaten zum Klassenautomaten)
Nach Definition 3.9 von *.isof.* gilt:
(C.367) $type_{sys}(a){=}c \Rightarrow (a\ isof_{sys}\ c)$
Daraus folgt mit Definition 8.5, Zeile (8.7) sofort die Behauptung. (□)

Beweis C.42 zu 8.10, Seite 160: (Vererbung von Typautomaten)
Weil nach Definition 3.9 und 3.8 gilt
(C.368) $d{\sqsubseteq}c \wedge (a\ isof_{sys}\ d) \Rightarrow (a\ isof_{sys}\ c)$,
folgt mit Definition 8.5, Zeile (8.7) die Behauptung. (□)

Beweis C.43 zu 8.14, Seite 164: (Entwicklungsschritt für Automatendokumente)
Um die Korrektheit des Entwicklungsschritts zu zeigen, sind die Kriterien aus Definition 2.8 zu erfüllen. Die ersten beiden Kriterien sind trivial. Das dritte Kriterium folgt aus der Kontextbedingung 3 der Definition 8.13.

Für das vierte Kriterium betrachten wir die Elemente *dn* der redundant gewordenen Dokumente
(C.369) $dn \in Q$.
Nach Definition von Q existiert ein Protokoll $p{\in}PS$ der Form
(C.370) $p = refAt(dn,tl)$
Nach Kontextbedingung 2 folgt:
(C.371) $apply(tl,(D.dn)_{cons}) = A$.
Weil *inherit* und *type2class* nicht verwendet werden, jeder einzelne Verfeinerungsschritt korrekt ist und die Verfeinerungsrelation $\models$ transitiv ist, folgt, daß:
(C.372) $A,\ \Gamma(tl) \models (D.dn)_{cons}$.
Weil aber
(C.373) $PS \models \Gamma(tl)$
folgt damit:
(C.374) $(A,PS) \models D(Q)_{cons}$.
Damit ist der Entwicklungsschritt *AddAutomaton* korrekt. (□)

Anhang D

Glossar

Ablauf ist eine von einem Automaten ausführbare Transitionssequenz. 77

Abstrakte Syntax enthält die zur Funktionalität beitragenden Anteile der Syntax. Von einer konkreten Darstellung wird abstrahiert. 56

Agent ist eine konzeptuelle Einheit, die über eine gekapselte Menge von Attributen und eine festgelegte Schnittstelle verfügt. 32

Attribut ist eine Programmvariable, die eine Komponente des Datenzustands eines Agenten beschreibt. 44

Automat ist eine zustandsbasierte Modellierungstechnik für das Verhalten eines Agenten. 56

Automatenzustand ist ein abstrakter Zustand eines Agenten. Er definiert eine Äquivalenzklasse von Datenzuständen. 140

Axiom beschreibt eine geforderte prädikatenlogische Aussage über Sorten und auf ihnen definierte Funktionen. 120

Beweisverpflichtung ist eine prädikatenlogische Formel, deren Gültigkeit für die Korrektheit eines Dokuments oder einer Verfeinerung notwendig ist. Sie wird durch einen Theorembeweiser gezeigt. 150

Buchstabierender Automat ist ein Automat, der pro Transition genau ein Zeichen der Eingabe verarbeitet. 55

Chaos tritt ein, wenn eine Komponente beliebiges Verhalten besitzt. 73

Datenbeziehung definiert eine Beziehung zwischen den Elementen einer Klasse, die auf den Daten beruht. 129

Datentyp beschreibt Sorten und auf ihnen definierte Funktionen. 119

Datenzustand ist ein konkreter Zustand eines Agenten, charakterisiert durch seine Attribute. 140

Denotationelle Semantik für Automaten ist eine auf stromverarbeitenden Funktionen basierende Semantikdefinition. 74

Deterministisch ist ein Automat, wenn für jede Eingabe in jedem Zustand höchstens eine Transition existiert. 58

Dokument ist eine bei der Systementwicklung verwendete, abgegrenzte syntaktische Einheit zur Beschreibung eines Aspektes oder eines Ausschnitts eines zu erstellenden Systems. 14

Dokumentgraph ist ein Entwicklungsgraph, bestehend aus Dokumenten, der eine Folgerungsbeziehung festhält, die bei der Entwicklung nachgewiesen wurde. 18

Element eines Klassentyps ist ein Agent, der Instanz der Klasse oder einer Subklasse ist. 42

Entwicklungsschritt ist eine Transformation des Dokumentgraphen, bei der ein neues Dokument hinzugefügt wird. 20

Erweiterbarer Datentyp ist ein Datentyp, bei dem nicht alle Konstruktoren bekannt sind. 142

Finale Transition ist eine Transition mit unendlicher Ausgabe. 58

Folgerungsbeziehung für Dokumente gilt, wenn die Semantik des gefolgerten Dokuments in der Semantik der Ursprungsdokumente enthalten ist. 16

Formale Methodik besteht aus der Formalisierung von Dokumentarten, einer formalen Semantik, einer Sammlung von Entwicklungsschritten, die Dokumente transformieren, und einer Sammlung von Taktiken. 25

Gezeitete Semantik ist eine Menge gezeiteter stromverarbeitender Funktionen als Semantik für einen buchstabierenden Automaten. 82

Gezeitete stromverarbeitende Funktion ist die Verhaltensbeschreibung einer Komponente unter Einbeziehung von Zeit. 203

Identifikator identifiziert einen Agenten eindeutig im System. 32

Initiale Ruhe hält ein Agent ein, wenn er erst nach Erhalt einer Aktivierungsnachricht Nachrichten versendet. 34

Initialelement ist ein Paar, bestehend aus Initialzustand und initialer Ausgabe. 56

Instanz einer Klasse ist ein Agent, der der Klasse zugeordnet ist. Jeder Agent ist Instanz genau einer Klasse. 42

Kanal ist ein gerichteter Kommunikationsweg zwischen zwei Komponenten. 32

Klasse dient unter anderem zur Klassifizierung von Agenten. 39

Klassenautomat beschreibt das Verhalten aller Instanzen einer Klasse. 143

Klassenbeschreibung definiert die Schnittstelle und Signatur einer Klasse. 123

Klassenhierarchie ist eine partielle Ordnung auf Klassen. 42

Kommunikationsmedium ist die Komponente im System, die den Transport der Nachrichten vom Sender zum Empfänger durchführt. 34

Methode ist eine auf oberster Stufe definierte Strategie, die Richtlinien für den gesamten Entwicklungsprozeß vorgibt. 27

Nachricht ist eine informationstragende Einheit, die den Agenten zur Kommunikation dient. 32

Nachrichtenraum ist die Menge aller Nachrichten. 142

Objektmodell definiert Daten- und Vererbungsbeziehungen zwischen Klassen. 129

Operationelle Semantik eines Automaten ist die Menge seiner Abläufe. 78

Partiell ist ein Automat der nicht total ist. 57

Protokoll beschreibt, mit welchen Verfeinerungsschritten ein Automatendokument aus einem anderen entwickelt wurde. Es wird zur Sicherstellung der Korrektheit einer Verfeinerungsbeziehung eingesetzt. 161

Protokolleintrag ist Element eines Protokolls. Er beschreibt einen Verfeinerungsschritt des Dokuments. 161

Redundanztest ist eine Transformation des Dokumentgraphen, bei dem ein Dokument auf Redundanz bezüglich seiner Nachfolger getestet wird. 21

Relationsbasierte Semantik eines Automaten ist eine auf Relationen zwischen Eingabe und Ausgabe eines Automaten basierende Semantikdefinition. 79

Rolle beschreibt die Form, an der die Elemente einer Klasse an einer Beziehung teilnehmen. 129

Schnittstellenverfeinerung ist eine gerichtete Beziehung zwischen zwei syntaktischen Einheiten. 107

Schnittstellenverfeinerungskalkül ist ein Verfeinerungskalkül für die Transformation von Schnittstelle und Verhalten eines Automaten. 109

Strategie ist eine Taktik zur Erreichung globaler Ziele. 26

Strom ist eine endliche oder unendliche Sequenz von Zeichen, die die Historie eines Kanals darstellen. 194

Strombündel ist eine mit Kanalnamen indizierte Menge von Strömen. 195

Stromverarbeitende Funktion ist eine Verhaltensbeschreibung einer Komponente, die die Eingabe mit der Ausgabe in Beziehung setzt. 196

System ist eine konkrete Ausprägung einer Kombination aus Hardware- und Softwarekomponenten sowie gegebenenfalls menschlichen Akteuren, die eine bestimmte (in gewissen Grenzen veränderbare) Struktur besitzen und ein bestimmtes Verhalten aufweisen. 31

Systemablauf ist ein Ablauf eines Systems in einem bestimmten (auch unendlichen) Zeitintervall, der Eingaben, Ausgaben und Zustände der Systemkomponenten für jeden (relevanten) Zeitpunkt des Ablaufs beinhaltet. 31

Systemmodell ist eine Charakterisierung aller interessanten Systeme. Es charakterisiert deren Verhalten und Struktur. 31

Taktautomat ist ein gezeiteter Automat, bei dem eine Transition genau einer Zeiteinheit entspricht. Er verarbeitet in einer Transition beliebige endliche Sequenzen von Eingaben. 85

Taktik ist die hierarchische Kombination von Entwicklungsschritten und Taktiken, und eine zugehörige Heuristik. 26

Total ist ein Automat, wenn für jede Eingabe in jedem Zustand mindestens eine Transition existiert die diese verarbeitet. 57

Totalisierung konstruiert einen totalen Automaten aus einem partiellen Automaten. 74

Transition ist ein Schritt (Zustandsübergang) eines Automaten. 56

Typautomat beschreibt das Verhalten aller Elemente einer Klasse. 143

Vererbung strukturiert unter anderem Klassen in einer Hierarchie. Schnittstelle und Verhalten werden entlang der Vererbungshierarchie verfeinert. 41

Verfeinerung ist eine gerichtete Beziehung zwischen zwei syntaktischen Einheiten, deren Semantiken in Inklusionsbeziehung stehen. 94

Verfeinerungskalkül für Automaten ist ein aus verfeinernden Transformationsregeln aufgebauter Kalkül. 108

Zustandsübergangsrelation ist eine Komponente eines Automaten. Sie beschreibt die Reaktion eines Agenten in Abhängigkeit von Zustand und Eingabe. 56

Anhang E

Index

E.1 Grundlagen-, Focus-Index

$\surd$	Zeittick, Tick	201
$\#.$	Länge eines Stroms	194
$\#_{\surd}$	Zeitliche Dauer der Beobachtung des Stroms	201
$\#_A s$	Zählt die Vorkommen von Zeichen aus A in s	195
$(\lambda x{:}M.F)$	Mathematische Funktionsabstraktion	191
$*$	Anonyme Variable; sie ist auf innerster Ebene existenzquantifiziert	193
$.\sqsubseteq.$	Präfixrelation auf Strömen	194
$.\,\hat{}\,.$	Konkatenation zweier Ströme	194
$A \copyright s$	Filtert aus einem Strom s nur die Menge der Zeichen A	195
$B^{\overline{\Omega}}$	Menge der endlichen und unendlichen gezeiteten Strombündel	202
$B^{\overline{\Phi}}$	Menge der endlichen gezeiteten Strombündel	202
$B^{\overline{\Sigma}}$	Menge der unendlichen gezeiteten Strombündel	202
B^{Ω}	Ungezeitetes Strombündel	195
$I^{\overline{\Sigma}} \xrightarrow{p} O^{\overline{\Sigma}}$	Menge der gezeiteten Funktionen mit Signatur (I,O)	203
$I^{\overline{\Sigma}} \xrightarrow{wp} O^{\overline{\Sigma}}$	Menge der schwach gezeiteten Funktionen mit Signatur (I,O).	203
$I^{\Omega} \xrightarrow{s} O^{\Omega}$	Menge der stromverarbeitenden Funktionen mit Signatur (I,O)	196
M^{∞}	Menge der unendlichen Ströme	194
M^{ω}	Menge aller Ströme	194
$M^{\overline{\infty}}$	Menge der unendlichen gezeiteten Ströme	201
$M^{\overline{\omega}}$	Menge der gezeiteten Ströme	201
$M^{\overline{*}}$	Menge der endlichen gezeiteten Ströme	201

M^*	Menge der endlichen Ströme	194
M_b	Menge von Nachrichten, die am Kanal b fließen können	195
$X \to Y$	Menge der totalen Funktionen	191
$X \rightharpoonup Y$	Menge der partiellen Funktionen	191
$X \stackrel{s}{\to} Y$	Menge der stetigen Funktionen	192
$\langle . \rangle$	Einelementiger Strom	194
$\mathbb{P}^{fin}(M)$	Menge der endlichen Teilmengen von M	191
$\bullet f$	Urbildmenge	191
ε	Leerer Strom	194
$\sqcup C$	Kleinste obere Schranke	192
$\mu_B f$	Rückkopplung der Kanäle B	198
$\|_{n \in N} f_n$	Parallele Komposition	197
$\diamond s$	Filtert die Nachrichten aus einem gezeiteten Strom	201
$adm(P)$	Eigenschaft P ist zulässig	192
$chain(C)$	C ist eine nichtleere, vollständig geordnete Teilmenge (Kette) eines CPO	192
$f+g$	Vereinigung von kompatiblen Funktionen	191
$f\bullet$	Bildmenge	191
$fix.\tau$	Eindeutiger kleinster Fixpunkt der monotonen Funktion τ	192
$s\downarrow n$	Präfix mit Beobachtungsdauer n	201

E.2 Allgemeiner Index

δ	Transitionsrelation eines Automaten	56
δ^∞	Unendliche Hülle der Transitionsrelation δ	63
δ^ω	Totale Hülle der Transitionsrelation δ	63
δ^*	Reflexiv transitive Hülle der Transitionsrelation δ	63
δ^+	Transitive Hülle der Transitionsrelation δ	63
$\Theta(\delta)$	Menge der Transitionssequenzen	62
(C,S,R)	Abstrakte Syntax eines Objektmodells	129
(N,E,D,R)	Dokumentgraph	18
(S,M,δ,I)	Automat mit identischer Eingabe und Ausgabemenge	56
$(S,M_{in},M_{out},\delta,I)$	Buchstabierender Automat	56
$(c,ak,S,\Lambda,\delta,I)$	Abstrakte Syntax eines Automatendokuments ohne Anhang	144

Symbol	Bedeutung	Seite
$.^{\backprime}$	Apostroph-Operator für Nachbedingungen im Hoare-Stil	145
$.\vdash.$	Ableitungsrelation für Rollen und Datenbeziehungen	131
$.\models.$	Folgerungsbeziehung für Dokumente	16
$.\models.$	Folgerungsbeziehung für prädikatenlogische Aussagen	120
$.\models.$	Interpretationsrelation für prädikatenlogische Aussagen	120
$.\models\!\dashv.$	Äquivalenzbeziehung für Dokumente	16
$.\sqsubseteq.$	Partielle Ordnung zur Darstellung der Klassenhierarchie	42
$.\rightsquigarrow.$	Verfeinerungsschritt für Automaten	109
$.\leftrightsquigarrow.$	Symmetrische Hülle der Verfeinerungsschritte für Automaten	109
$.\overset{*}{\leftrightsquigarrow}.$	Symmetrisch transitive Hülle der Verfeinerungsschritte für Automaten	109
$.\overset{*}{\rightsquigarrow}.$	Transitive Hülle der Verfeinerungsschritte für Automaten	109
$.isof.$	Typisierung von Agenten	42
Δ	Menge aller Transitionen von Automatendokumenten	146
ϕ	Modelliert fehlenden Rollennamen im Objektmodell	129
Υ	Abstraktionsfunktion, die aus einer Transitionssequenz einen maximalen Ablauf bestimmt	78
$\pitchfork$	Chaos-Zustand eines totalisierten Automaten	74
$\omega{-}reach(A)$	Erreichbare Zustände im Automaten	64
T_{post}	Nachbedingung einer Transition eines Automatendokuments	146
T_{pre}	Vorbedingung einer Transition eines Automatendokuments	146
d_{cons}	Konstruktiver Anteil des Dokuments d	17
d_{prot}	Protokollteil des Dokuments d	17
$s \xrightarrow{in/out} t$	Transition mit Eingabe in und Ausgabe out	76
$AddAutomaton$	Entwicklungsschritt für Automatendokumente	163
$AddClass$	Entwicklungsschritt für Klassenbeschreibungen	125
$AddOM$	Entwicklungsschritt für Objektmodelle	133
BEL	Menge von Variablenbelegungen	43
CM	Kommunikationsmedium	37
CN	Endliche Menge von Klassennamen	42
$EventTrace$	Menge der Systembeobachtungen	38
$ExtClass$	Entwicklungsschritt für Klassenbeschreibungen	125
$ExtOM$	Entwicklungsschritt für Objektmodelle	133
ID	Menge der Identifikatoren	33
$IntOM$	Entwicklungsschritt für Objektmodelle	133

MSG	Gesamtmenge der Nachrichten eines Systems	33
$Msg(a_i)$	Menge der Eingabenachrichten des Agenten a	33
$Msg(a_o)$	Menge der Ausgabenachrichten des Agenten a	33
$STATE$	Menge der möglichen Agentenzustände	39
$StartID$	Menge der nicht initial ruhigen Agenten	34
TA_a	Taktautomat des Agenten $a \in ID$	39
$TE_{M,S}$	Menge der Ablaufelemente mit Zustandsmenge S und Zeichenmenge M	77
$Trace_{M,S}$	Menge der Abläufe mit Zustandsmenge S und Zeichenmenge M	77
VAL	Werteuniversum, enthält keine Agenten, jedoch Identifikatoren	43
$\mathcal{DG}$	Menge der Dokumentgraphen	18
$\mathcal{DOC}_{aut}$	Menge der Automatendokumente, erweitert um Protokolle	161
$\mathcal{DOC}$	Menge aller Dokumente	15
$\mathcal{DOC}_{cl}$	Menge der Klassenbeschreibungen	123
$\mathcal{DOC}_{dt}$	Menge der Datentypdokumente	119
$\mathcal{DOC}_{om}$	Menge der Objektmodelle	129
$\mathcal{DS}$	Entwicklungsschritt für Dokumentgraphen	20
$\mathcal{N}$	Grundmenge von Knoten für Dokumentgraphen	18
$\mathcal{PROT}$	Menge der Protokolle	161
$\mathcal{RT}$	Redundanztest für Dokument im Dokumentgraphen	21
$\mathcal{SM}$	Menge der Systemmodelle	49
Σ_{attr}	Attributmenge einer Klasse	44
Σ_{meth}	Methoden einer Klasse	44
$\langle expr \rangle$	Menge der Wertausdrücke	119
$\langle konstruktor \rangle$	Menge von Konstruktoren	43
$\langle meth \rangle$	Menge von Methodennamen	44
$\langle patt \rangle$	Menge der Patternausdrücke	119
$\langle pred \rangle$	Menge der prädikatenlogischen Ausdrücke	120
$\langle sort \rangle$	Menge der Sortenausdrücke	119
$\langle var \rangle$	Menge von Variablennamen	43
$[\![.,.]\!]^{expr}$	Semantik für Wertausdrücke	119
$[\![.]\!]$	Denotationelle Semantik für buchstabierende Automaten	74
$[\![.]\!]$	Semantik für ein Dokument bzw. eine Dokumentmenge	15
$[\![.]\!]^a$	Semantik für den konstruktiven Anteil der Automatendokumente	149
$[\![.]\!]^c$	Denotationelle Semantik für totale Automaten	68

$[\![.]\!]^{cl}$	Semantik einer Klassenbeschreibung	123
$[\![.]\!]^{d}$	Denotationelle Semantik für deterministische, totale Automaten	67
$[\![.]\!]^{om}$	Semantik eines Objektmodells	130
$[\![.]\!]^{op}$	Operationelle Semantik eines Automaten	78
$[\![.]\!]^{p}$	Denotationelle Hilfssemantik für totale Automaten	68
$[\![.]\!]^{r}$	Relationsbasierte Semantik für Automaten	79
$[\![.]\!]^{ro}$	Semantik einer Rolle	130
$[\![.]\!]^{t}$	Gezeitete denotationelle Semantik für Automaten	83
$[\![.]\!]^{ta}$	Denotationelle Semantik für Taktautomaten	86
$[\![.]\!]^{aut}$	Semantik für Automatendokumente	162
$[\![.]\!]^{prot}$	Semantik für Protokolle	162
$[\![.]\!]^{rel}$	Semantik einer Datenbeziehung	130
`ACTEES`	Datentyp des speziellen Attributs `actees`	142
`IN`	Datentyp der Eingabenachricht `in`	142
`OUT`	Datentyp der Ausgabenachricht `out`	142
`in`	Variable charakterisiert die Eingabenachricht	142
`out`	Variable charakterisiert die Ausgabenachricht	142
activate	Aktivierungsstruktur	46
addS	Automatentransformation: Erweiterung der Menge der Automatenzustände	153
addT	Automatentransformation: Hinzunahme von Transitionen	154
args	Zuordnung Argumente für Methode	44
automaton	Transformation eines Automatendokuments in einen buchstabierenden Automaten	148
$behavior_a$	Verhalten des Agenten a	33
class	Zuordnung der Klasse für jeden Agenten	42
compactify	Kompaktifizierung für Automaten	103
complete	Totalisierung eines Automaten	74
consistent	Konsistenzprädikat für Dokumentmenge	15
dest(ts)	Finalzustand der Transitionssequenz *ts*	62
destination	Empfänger einer Nachricht	35
fuseClasses	Integriert Klassenbeschreibungen	124
fuseOMs	Integriert Objektmodelle	132
in(ts)	Verarbeitete Eingabe der Transitionssequenz *ts*	62
inherit	Automatentransformation: Vererbung eines Typautomaten	159

$inits_a$	Menge der möglichen Initialzustände von Agent a	39
newAt	Protokolleintrag in Automatendokumenten	161
out(ts)	Ausgabe der Transitionssequenz *ts*	62
$pred_E.n$	Vorgängermenge des Knotens n im Dokumentgraphen	18
reach(A)	Operationell erreichte Zustandsmenge im Automaten	64
refAt	Protokolleintrag in Automatendokumenten	161
refI	Automatentransformation: Entfernung von Initialelementen	156
refI	Automatentransformation: Verfeinerung eines Initialelements	156
refS	Automatentransformation: Verfeinerung von Automatenzuständen	157
refT	Automatentransformation: Verfeinerung von Transitionen	155
remS	Automatentransformation: Entfernung von Automatenzuständen	154
remT	Automatentransformation: Entfernung einer Transition	155
sortdef	Sortendefinition durch Konstruktoren	43
source(ts)	Startzustand der Transitionssequenz *ts*	62
$states_a$	Menge der möglichen Zustände von Agent a	39
std_a	Zustandsübergangsrelation von Agent a	39
$succ_E.n$	Nachfolgemenge des Knotens n im Dokumentgraphen	18
type2class	Automatentransformation: Typautomat zu Klassenautomat	159
type	Typisierung von Variablen	43
values	Wertemenge eines Typs	43