Grundlagen für die formale Spezifikation modularer zustandsbasierter Systeme

Dissertation

zur Erlangung des Grades des

D o k t o r s   d e r   N a t u r w i s s e n s c h a f t e n

der Universität Dortmund
am Fachbereich Informatik

Vorgelegt von

Claus Pahl

Dortmund
1996

Tag der mündlichen Prüfung: 7. Juni 1996

Dekan: Prof. Dr. H. Müller

Gutachter: Prof. Dr. E.-E. Doberkat
Prof. Dr. P. Padawitz

Zusammenfassung

Diese Arbeit stellt Konzepte vor, die im Kontext zustands- oder objektbasierter Systeme die gemeinsame Behandlung von Implementierungssprachen und Spezifikationssprachen gestatten. Sie befaßt sich zum einen mit der formalen Definition einer Programmiersprache und zum anderen mit dem Entwurf einer Spezifikationssprache, die auf die Programmiersprache ausgerichtet ist. Abhängigkeiten zwischen diesen beiden Aspekten werden herausgearbeitet. Die Definition beider Sprachen erfolgt auf einem eigenständigen Berechnungsmodell, einer formal definierten abstrakten Maschine, zur Modellierung des Verhaltens von Objekten. Erweiterungen des Berechnungsmodells, die Rekursion, Verschachtelung von Programmeinheiten oder Typfragen betreffen, werden vorgestellt. Zur Spezifikation zustandsbasierter Systeme wird dynamische Logik, eine Erweiterung einer Prädikatenlogik erster Stufe, die Zustände explizit macht, eingesetzt. Mit Hilfe der dynamischen Logik kann das Verhalten von Objekten abstrakt beschrieben werden. Ein Beweissystem für die Logik wird definiert, mit dem auch die Verifikation einer Implementierung bezüglich einer Spezifikation möglich ist. Hierzu wird ein Korrektheitsbegriff definiert, der durch das Beweissystem operationalisiert wird. Zur Beschreibung von modularen Software-Systemen werden formale Parametrisierungs- und Schnittstellenkonzepte erarbeitet. Eine Reihe von Relationen wird definiert, die es ermöglichen, verschiedene Beziehungen zwischen Systemkomponenten zu modellieren. Horizontale und vertikale Entwicklung wird betrachtet.

Danksagung
Diese Arbeit entstand in den Jahren 1991 bis 1993 in der Arbeitsgruppe *Software Engineering* der Universität GH Essen und anschließend am Lehrstuhl *Software Technologie* der Universität Dortmund.
Mein besonderer Dank gilt Herrn Prof. Dr. Ernst-Erich Doberkat für seine Anregungen und sorgfältige Durchsicht meiner Arbeit. Prof. Dr. Michael Goedicke und Prof. Dr. Peter Padawitz danke ich für ihre Anmerkungen und Hinweise zu dieser Arbeit. Mein Dank geht auch an Rix Groenboom für seine Kommentare, sowie an die Essener und Dortmunder Kollegen und die Diplomanden Klaus Alfert und Thomas Tenbeitel für Anregungen und Diskussionen im Umfeld der Arbeit.
Nicht zuletzt schulde ich meinen Eltern Dank für ihre Unterstützung und die mir gegebene Freiheit, die die Voraussetzungen zum Entstehen dieser Arbeit waren.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und kurzer Überblick

Die Strukturierung der realen Welt in Objekte kann auch als Anhaltspunkt genutzt werden, Software-Systeme anhand von modellierten Objekten zu strukturieren. Objekte der Realität sind Entitäten, die bzgl. ihres Aufbaus und ihres Verhaltens charakterisiert werden können. Objekte der Realität können in Computer-Systemen modelliert werden. Die Beschreibung ihrer Charakteristika kann entweder konkret durch Angabe von Realisierungen in einer Programmier- oder Implementierungssprache oder abstrakter durch Formulierung von Anforderungen an Struktur und Verhalten umgesetzt werden.

In dieser Arbeit soll das *Berechnungsmodell* der *Σ-Objekte* vorgestellt werden. Σ-Objekte sind formale, mathematisch beschriebene Entitäten, die die Entsprechungen der Objekte der realen Welt formal repräsentieren. Σ beschreibt dabei das zur Verfügung stehende Vokabular. Ein *Objekt* umfaßt einen verkapselten Zustand, auf den mit Operationen in Form von Attributen und Prozeduren zugegriffen werden kann. Der lokale Zustand ist definiert durch die Werte von objektlokalen Variablen, deren interne Struktur dem Benutzer eines Objektes verborgen bleiben soll. Operationen bilden die Benutzungsschnittstelle eines Objektes. Operationen sind entweder Attribute (diese dürfen den Zustand nicht ändern) oder Prozeduren (sie dürfen den Zustand ändern). Die Kommandos, mit denen das Objekt verändert werden kann, sind imperativ. Die Basiskonstrukte einer imperativen Sprache, wie *Zuweisung*, *Definition* und *Aufruf (rekursiver) Prozeduren*, *bedingte Anweisung* und *Anweisungssequenz*, sollen zur Verfügung gestellt werden.

Eine imperative Programmiersprache läßt sich somit auf dem Berechnungsmodell der Σ-Objekte leicht definieren, indem die Σ-Objekte als eine abstrakte, formal definierte Maschine angesehen werden. Eine Spezifikationslogik kann definiert werden, indem Σ-Objekte als semantische Strukturen angesehen werden, in denen logische Aussagen interpretiert werden können. Ziel ist, durch Σ-Objekte sowohl operationale als auch deskriptive *zustandsbasierte* Sprachen definierbar zu machen.

Zwei wichtige formale Spezifikationsmethoden sind *eigenschaftsorientierte* (d.h. axiomatische) und *modellorientierte* Methoden (etwa *VDM* oder *Z*), siehe [Som92], [Bjø91], [NP91]. Die hier erarbeiteten Ergebnisse sollen Basis eines Ansatzes zum eigenschaftsorientierten Spezifizieren von Modulen bilden. Es werden sequentielle, deterministische imperative Sprachen als Implementierungssprachen betrachtet. Damit soll innerhalb des imperativen Programmierparadigmas das (abstrakte) Spezifizieren des Ein-/Ausgabeverhaltens von Operationen realisiert werden. Um Beziehungen zwischen operationalen (hier imperativ formulierten) und deskriptiven (hier axiomatisch formulierten) Spezifikationen in einer semantischen Struktur untersuchen zu können, soll für operationale, d.h. imperative Beschreibungen ein mathematisches Modell definiert werden. Dieses Modell soll zur Modellklasse einer abstrakten, algebraisch orientierten Spezifikation gehören. Die Formalisierung der Σ-Objekte wird im *denotationalen Stil* vorgenommen.

Eine Zielsetzung, die mit dem vorliegenden Ansatz erreicht werden soll, liegt in der Umsetzbarkeit auf reale Sprachen. So soll folgendes Szenario realisiert werden können: Es sei eine imperative oder objektbasierte[1] Programmiersprache gegeben. Diese Sprache soll um einen Spezifikationsansatz erweitert werden, d.h. Komponenten der Sprache (etwa Prozeduren oder Module) sollen semantisch abstrakt beschreibbar sein. Dabei sind die folgenden Fragen zu klären:

- Wie sieht die formale Semantik der Programmiersprache aus?
- Auf welchen Konzepten und Methoden soll die Spezifikationssprache basieren?
- Wie sieht die formale Semantik der Spezifikationssprache aus?

Um Spezifikation und Programmierung kombinieren zu können, besteht die Notwendigkeit, sie auf einer gemeinsamen formalen Basis integrieren zu können. Während Konzepte einer Programmiersprache entweder durch die Vorgabe einer konkreten Sprache festgelegt oder sonst im üblichen Rahmen als bekannt vorausgesetzt werden können, muß im Bereich der Spezifikationssprachen hier noch eine nähere Untersuchung stattfinden. Es ist aber nicht nur zu betrachten, wie Komponenten beschrieben werden sollen, sondern auch, wie sie verknüpft werden können, also, wie mit der Sprache Systeme, d.h. Komponentenarchitekturen, modelliert werden können. Im folgenden soll mit dem Begriff der *Software-Komponente* ein in sich inhaltlich abgeschlossenes Fragment des zu erstellenden Software-Systems verstanden werden. Ziel dieser Arbeit ist, ein Komponentenkonzept zu realisieren, das über die Mächtigkeit von Modulkonzepten üblicher imperativer Sprachen hinausgeht, indem es Konstrukte zur Komposition und Modifikation von Komponenten und zur Beschreibung von Beziehungen zwischen ihnen bereitstellt. Damit soll insbesondere die Entwicklung von Software-Systemen in Teams unterstützt werden.

Die hier angesprochenen Fragen zur zustandsbasierten Spezifikation stellen sich neben den schon angesprochenen Programmiersprachen mit dem Aufkommen objektorientierter Konzepte auch im Bereich der Datenbanken. Objektorientierte Datenbanken sind zustandsbasierte Systeme, die ebenfalls auf den Begriffen Objekt, Zustand und Veränderung fußen. Die formale Spezifikation und Implementierung dieser Objekte ist auch im Bereich Datenbanken Gegenstand der Forschung.

Es soll hier *nicht* eine Programmiersprache definiert werden. Ebensowenig wird eine Spezifikationssprache definiert. Diese wird nur durch die formale Definition einer Spezifikationslogik, von Strukturierungs- und Modellierungskonzepten auf Spezifikationsebene vorbereitet. Der Ansatz soll aber so gestaltet werden, daß er zur Definition von Programmiersprachen angewendet werden kann. Dabei soll, soweit möglich, auch die Realisierung oder Umsetzung von Konzepten durch Werkzeuge, die die angesprochenen Sprachen in einer Entwicklungsumgebung benutzbar machen, betrachtet werden.

[1] Objektbasierte oder objektorientierte Sprachen werden wir als Spezialfall imperativer Sprachen ansehen.

1.1.1 Gliederung der Arbeit

Die Arbeit ist wie folgt gegliedert: Nach der Einleitung wird im Teil I das Berechnungsmodell der Σ-Objekte vorgestellt. Kapitel 2 führt in die Problematik ein und stellt vergleichbare Ansätze vor. Kapitel 3 umfaßt ein Basismodell der Σ-Objekte. Eine Reihe sinnvoller Erweiterungen wird in Kapitel 4 vorgeschlagen.

Teil II der Arbeit befaßt sich mit der Spezifikationslogik. Kapitel 5 definiert eine modale Spezifikationslogik. Kapitel 6 enthält Erweiterungen der Spezifikationslogik, die bei Integration mit einer Implementierungssprache von Bedeutung sind. Kapitel 7 umfaßt Betrachtungen zu Beweissystemen und Verifikation.

In Teil III schließlich werden verschiedene Relationen zwischen Spezifikationen untersucht, die den Prozeß der Entwicklung von Software unterstützen sollen. Kapitel 8 definiert eine Implementierungsrelation und analysiert die vertikale Entwicklung. Die horizontale Entwicklung und ihre Relationen sind Thema von Kapitel 9. In Kapitel 10 wird objektorientierte Entwicklung im Kontext des vorliegenden Ansatzes betrachtet. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick.

1.2 Entwurf und Implementierung von Software-Systemen

Um den Kontext der vorliegenden Arbeit zu beschreiben, wird der vorliegende Ansatz nun in den Prozeß der Entwicklung von Software eingeordnet.

1.2.1 Stadien der Software-Entwicklung

Es existieren verschiedenste Ansätze, die die Entwicklung von Software unterstützen. Den Ansätzen gemein ist die Unterteilung des Entwicklungsprozesses, auch *Software-Prozeß* genannt, in *Stadien*, deren Identifizierung anhand charakteristischer Aufgabenstellungen für die Entwickler erfolgt. Im einfachsten dieser Ansätze, dem Wasserfall-Modell der Software-Entwicklung (siehe [Som92, GJM91]) werden die Stadien in eine lineare zeitliche Ordnung gebracht. Das Wasserfall-Modell hat sich allerdings als nicht flexibel genug erwiesen, um die Anforderungen der Software-Entwicklung zufriedenstellend bewältigen zu können. Neuere Ansätze, wie das Spiralmodell von Barry Boehm (siehe etwa [GJM91]), integrieren Zyklen in den Entwicklungsprozeß; man bedient sich *inkrementeller* oder *evolutionärer* Techniken. Das Software-Produkt wird schrittweise durch Erweitern von Inkrementen erstellt. Eine klassische Stadienbildung umfaßt die Stadien Anforderungsanalyse, Entwurf, Implementierung, Testen, Installation und Wartung. Kennzeichnend für ein Stadium sind die *Aktivitäten*, die in ihm vorzunehmen sind.

So hat der *Entwurf* die Umsetzung der maschinenunabhängigen Anforderungen in ein Modell des Gesamtsystems zum Inhalt, welches, umgesetzt in ein Programm, die Anforderungen erfüllt. Es wird das Gesamtsystem in überschaubare Einzelbausteine zerlegt, die Funktionalitäten dieser Bausteine und ihre Beziehungen untereinander werden beschrieben. Ergebnis ist eine Spezifikation des Systems. In der *Implementierung* wird ein ablauffähiges Programm erstellt, das in seinem Ein-/Ausgabeverhalten der Spezifikation entspricht. Nichtfunktionale Anforderungen an Software-Komponenten, also auch an das Endprodukt, das Software-System, werden als *Qualitätsfaktoren* bezeichnet. Hierzu gehören nach B. Meyer [Mey88] Wiederverwendbarkeit, Erweiterbarkeit, Kompatibilität, Korrektheit und Robustheit. Diesen

Hauptkriterien lassen sich noch Vollständigkeit, Präzision, Eindeutigkeit, Effizienz, Portabilität und Verifizierbarkeit hinzufügen (siehe auch [GJM91]). Diese Qualitätsfaktoren müssen in der Methodik berücksichtigt werden und von der Sprache durch Techniken realisierbar gemacht werden.

Diese Aufgabenbeschreibungen lassen erkennen, daß zwei wesentliche Elemente zur Lösung der Aufgaben bereitgestellt werden müssen: eine **Methodik** und eine **Sprache**, wobei sich die Sprache in den methodischen Rahmen einordnen soll.

1.2.2 Prinzipien, Methoden und Techniken

Eine *Methodik* (für ein Stadium) besteht aus einer Sammlung von Richtlinien, die die Lösung der gegebenen Aufgaben unterstützen soll. Die einzelnen Richtlinien sind dabei nicht beliebig zusammengestellt, sondern ordnen sich allgemeinen, übergeordneten *Prinzipien* unter. Prinzipien beschreiben wünschenswerte Eigenschaften des Software-Prozesses oder des Software-Produktes. In einer auf die Methodik abgestimmten Spezifikations- oder Programmiersprache werden die sprachrelevanten Prinzipien durch *Methoden* und *Techniken* für den Entwickler realisierbar gemacht. Methoden sind allgemeine Richtlinien, die die Ausführung von Aktivitäten beschreiben. Techniken sind mechanischer oder technischer als Methoden, häufig mit eingeschränkterer Anwendbarkeit. Aus den Prinzipien leiten sich so Anforderungen an die Sprachen ab. Die Entwicklung von Software erfolgt mit dem Ziel der Umsetzung von Qualitätsanforderungen. Die Unterstützung des Software-Prozesses durch eine Methodik muß dem Rechnung tragen. Das hat zur Folge, daß einzelne Prinzipien durchaus in mehreren Stadien zum Einsatz kommen, da sich Qualitätsanfordungen nicht nur auf das Endprodukt, sondern auch durchgängig auf alle Zwischenprodukte beziehen.

Ein Prinzip, das Begriffe wie Hierarchiebildung oder Systemdekomposition vereint, wird häufig als *divide and conquer*-Prinzip oder *Modularität* bezeichnet [GJM91]. *Modularisierung* ist eine wichtige Technik in Spezifikation und Implementierung, die hier auch als *Komponentenbildung* bezeichnet werden soll. Weitere Techniken sind *abstrakte Datentypen*, *Typisierung*, *Schnittstellen* und *Vererbung*. Sie werden in den folgenden Kapiteln noch angesprochen. Ein weiteres Prinzip, das die Beziehungen zwischen Software-Komponenten zum Inhalt hat, schließt sich direkt an die Hierarchiebildung durch *divide and conquer* an. *Vertragsorientiertes Entwickeln* [Mey92a] hat das Ziel, den Umfang von Komponentenbeschreibungen und den Aufwand zu deren Erstellung zu reduzieren. Die Beschreibung einer Software-Komponente wird als Vertrag angesehen. Vertragspartner sind der Entwickler und der Benutzer der Komponente. Der Entwickler garantiert Leistungen seiner Komponente, falls der Benutzer spezifizierte Vorleistungen erbringt.

Ein Prinzip, das sich nicht den Beziehungen zwischen Komponenten oder deren Benutzung zuwendet, sondern die interne Strukturierung einer Komponente zum Thema hat, wird mit *separation of concerns* [GJM91] bezeichnet. Unterschiedliche Aspekte einer Komponente sollen in getrennten Abschnitten behandelt werden. Weitere Prinzipien, die in der Software-Entwicklung von Bedeutung sind, sind *Formalität*, *Abstraktion*, *Antizipation von Veränderung* und *Allgemeinheit*.

Wiederverwendung (Entwurf mit Wiederverwendung) bezeichnet das Verwenden (d.h. Auffinden, Anpassen und Integrieren) von Bausteinen, die zu diesem Zweck entwickelt wurden (Entwurf zur Wiederverwendung) oder die zumindest geeigneten Qualitätsanforderungen genügen. Bei der Entwicklung großer Systeme sollte das Augenmerk auf neuen Aspekten liegen und nicht auf der Entwicklung schon realisierter Algorithmen und Architekturen. Wiederverwendung ist somit mit den Prinzipien Abstraktion, Antizipation von Veränderung und Allgemeinheit verknüpft. Die Technik *Implementieren* senkt das Abstraktionsniveau einer Komponentenbeschreibung. Durch *Verfeinern* wird die Beschreibung einer Komponente präziser. Diese beiden Techniken sind Elemente des *vertikalen Entwickelns*. Eine andere, schon angesprochene Technik ist *Modularisierung*. Sie wird der *horizontalen Entwicklung* zugeordnet. Wiederverwendung wäre ebenfalls hier einzuordnen. Die Qualität des Endprodukts hängt, wie schon beschrieben, von der Güte der Methoden und Techniken ab. So muß sichergestellt sein, daß die Implementierung verifizierbar ist, d.h. daß das Ergebnis korrekt in bezug auf die Ausgangskomponente ist. Die Verfeinerung darf nicht zu Widersprüchen führen. Die Anforderungen der Ausgangskomponente müssen beachtet werden. Bei der Wiederverwendung von Komponenten ist sicherzustellen, daß wiederverwendbare Komponenten den Erwartungen entsprechen. Modularisierung verlangt eine semantische Unterstützung der Schnittstellen der Komponenten. Die *Formalität* ist also eine wünschenswerte Voraussetzung. [GJM91] definieren das Prinzip Formalität als das streng an mathematischen Gesetzmäßigkeiten ausgerichtete Vorgehen bei der Entwicklung von Software. Auf formale Methoden werden wir in Abschnitt 1.3 detailliert eingehen.

1.2.3 Methodik und Sprache

Die Aufgabe einer Sprache besteht in der Beschreibung der Eigenschaften des Software-Systems. Eine allgemeine Klassifizierung von Sprachen läßt sich anhand folgender vier Merkmale durchführen (nach [Lud93]):

1. das Stadium der Software-Entwicklung, in dem sie eingesetzt wird,
2. das Prinzip der Interpretation oder Verarbeitung, also das Konzept des virtuellen Rechners, den der Anwender voraussetzt (das Paradigma),
3. die Form der Darstellung,
4. das Fehlen oder Vorhandensein bestimmter Ausdrucksmittel.

Je nach Stadium werden unterschiedliche Anforderungen an die Sprache gestellt. So hängt der Grad an Formalität vom Umfang des Wissens über das Problem ab. Es macht somit wenig Sinn, in der Wissensermittlung durch die Anforderungsanalyse die formale, präzise Formulierung rudimentärer, unvollständiger Informationen zu fordern.

Sprachen haben einen mittelbaren Effekt. Sie beeinflussen den Programmierstil und haben Einfluß auf die Denkstrukturen der Entwickler. Dieser Effekt muß von der Methodik neutralisiert werden und in einer Sammlung von Richtlinien zur Anwendung der Sprache strukturiert und koordiniert werden, falls diese Auswirkungen den Ideen der Methodik zuwiderlaufen. Die Methodik kann helfen, eine Software-Komponente in Substrukturen zu zerlegen oder Alternativen zu einer gegebenen Lösung zu finden. Über die sprachbezogenen Richtlinien hinaus sollte die Methodik den Entwicklern auch in organisatorischen Problemen unterstützen. So sollten von ihr Probleme der Kommunikation mit dem Auftraggeber oder anderen Projektbeteiligten betrachtet werden. Wichtig beim Einsatz einer bestimmten Sprache ist es also, deren mittelbaren Effekt auf Denkweise und Programmierstil zu erfassen und in einer Methodik optimal einzusetzen.

1.2.4 Zur aktuellen Situation in der Software-Entwicklung

In den vorangegangenen Abschnitten wurde keine scharfe Trennung zwischen Konzepten für den Entwurf und Konzepten für die Implementierung gemacht. Die vorgestellten Prinzipien, Methoden und Techniken können in der Regel in Methodik und Sprachen beider Stadien eingehen. Um den Qualitätsanforderungen gerecht zu werden, scheint es sinnvoll, daß der gesamte Software-Entwicklungsprozeß auf einem einheitlichen konzeptuellen Modell der Software basiert. Eine Formalisierung des Begriffs der Software-Komponente und darauf anwendbarer Aktivitäten könnte dies leisten. Allerdings offenbart der heutige Stand der Technik Diskrepanzen zwischen den bereitgestellten Mitteln der Sprachen, insbesondere der Sprachen der Implementierung, und den Anforderungen von Spezifikationsmethodiken des Entwurfs. Für alle Aktivitäten werden Beschreibungsmittel für die Systemeigenschaften auf unterschiedlichen Ebenen der Abstraktion und der Präzision benötigt.

Ein konzeptuelles Modell, wie es von einem komponentenorientierten Ansatz induziert wird, wird in einigen Programmiersprachen ansatzweise realisiert. Objektorientierte Sprachen strukturieren Software-Systeme durch Objektbildung. Den Vorgang der Komponentenbildung realisieren sie durch Klassenbildung. In vielen imperativen Sprachen sind Modulkonzepte realisiert.

Spezifikationssprachen in ihrer algebraischen Ausprägung ermöglichen die Beschreibung abstrakter Datentypen. Ihnen fehlt aber in der Regel die Mächtigkeit, alle Aspekte von Software-Systemen, wie etwa Zustände, beschreiben zu können. Modellbasierte Ansätze wie VDM oder Z schneiden zwar bzgl. der Zustandsmodellierung besser ab, ihnen fehlt aber die Abstraktion auf ein eigenschaftsorientiertes Niveau. Zudem fehlt eine starke semantische Unterstützung von Modularisierung- und Schnittstellenkonzepten.

1.3 Formale Methoden in der Software-Entwicklung

1.3.1 Einführende Betrachtungen

Das Prinzip der Formalität wurde schon angesprochen. Formale Methoden sind hier grundlegend. Die Betrachtungen zum Einsatz formaler Methoden in der Software-Entwicklung sollen sich im wesentlichen auf die Stadien Entwurf und Implementierung konzentrieren. Als erstes stellt sich die Frage, welche Aufgaben die Entwicklung von Software in diesen Stadien mit sich bringt.

Die IFIP (International Federation for Information Processing) hat 1991 einen *State-of-the-Art* Bericht zu diesem Thema herausgegeben [NP91]. Dieser Bericht stützt sich auf Diskussionen und Ergebnisse der Arbeitsgruppe TC2 *Programming* und die Erfahrungen aus Seminaren zu Themen der Programmierung, wobei Aspekte des Entwurfs in erheblichem Maße eingeflossen sind. Der Bericht enthält verschiedene Aufsätze zum Thema der formalen Methoden zur Definition und Spezifikation sowie der Werkzeuge für die Programmierung, die sich im einzelnen den Themen Semantik von Programmen, Spezifikation und Transformation, algebraische Spezifikation, verteilte, nebenläufige und reaktive Systeme, Programmverifikation und Typisierung widmen.

Drei Bereiche kristallisieren sich heraus (siehe Abbildung 1.1), wenn man diese Themen angesichts der Aufgaben der Software-Entwicklung in den Stadien Entwurf und Implementierung betrachtet.

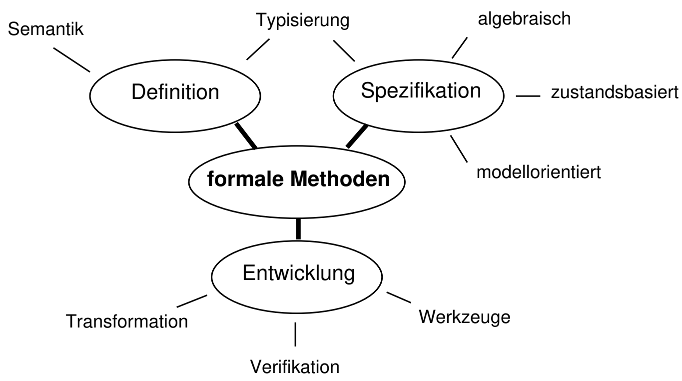


Abbildung 1.1: Formale Methoden

Formale Methoden für die Definition: Ausgehend von einer konzeptionellen Beschreibung einer Sprache wird diese durch geeignete Methoden formalisiert. Zwei Vertreter dieser Methoden zur Definition der *Semantik* einer Programmiersprache sind die denotationale [Mos91] und die operationelle Semantik [Ast91]. Definition und Implementierung einer Sprache profitieren vom Einsatz formaler Methoden [MTH90, Has94]. Formale Semantiken dienen nicht nur der Beschreibung allein, sondern sind auch ein Werkzeug zur Analyse der Konzeption. Sie forcieren die präzise, exakte Beschreibung der Konzepte und der Relationen zwischen Konzepten und helfen damit, Fehler aufzudecken und Unklarheiten oder Mehrdeutigkeiten zu vermeiden. Zudem kann die formale Definition als funktionale Anforderungsdefinition an die Implementierung angesehen werden. Verschiedene Ansätze zu diesem Thema werden im Abschnitt 2.2 diskutiert.

Ein Aspekt, der noch zum Teil in diesen Bereich fällt, ist der der *Typisierung*. Als Bestandteil der Sprache ist das Typsystem ebenso formal zu definieren, wie es von allen anderen Konstrukten gefordert ist. Mit dem Typsystem verbinden sich aber auch in starkem Maße Modellierungsaspekte. Dies wird an geeigneter Stelle diskutiert (vgl. Abschnitt 4.3).

Formale Methoden für die Spezifikation: Bei Einsatz einer formalen Spezifikationssprache ist, im Unterschied zu einer formal definierten Programmiersprache, ein zugrundeliegender mathematischer Ansatz explizit vom Benutzer anzuwenden. Vertrautheit mit dem mathematischen Ansatz ist für den Benutzer auch entscheidende Voraussetzung, um die Sprache überhaupt einsetzen zu können. Die formale Definition der Sprache ist in einem solchen Fall für den Sprachentwerfer natürlich unabdingbar. Es geht um das implementierungsunahbängige, abstrakte Beschreiben von Software-Systemen. Bedeutende Ansätze in diesem Bereich sind die eigenschaftsorientierten und die modellorientierten (oder modelltheoretischen). Unter eigenschaftsorientierten Spezifikationen werden algebraische und andere logisch basierte Ansätze subsummiert. Ihr Ziel ist die Spezifikation funktionaler Eigenschaften von Software-Systemen unabhängig von der konkreten Darstellung von Daten und unabhängig von der Implementierung. Modelltheoretische Ansätze haben eine Semantik, die den einzelnen Sprachkonstrukten direkt ein mathematisches Modell zuordnet, etwa eine Menge oder eine Funktion. Das Systemverhalten wird direkt im mathematischen Modell beschrieben, während die algebraischen Ansätze

Eigenschaften der Modelle definieren. Formale Spezifikationen werden später noch vertieft (vgl. Kapitel 5 bis 7 zur Spezifikationslogik und Kapitel 8 bis 9 zu Relationen zwischen Spezifikationen).

Formale Methoden für die Entwicklung: Wie in vorangegangenen Abschnitten schon erläutert, stehen die Methoden und Techniken, die in den Spezifikations- oder Programmiersprachen realisiert werden, in engem Zusammenhang mit den Prinzipien der Methodik. Die Bedeutung des Entwicklungsprozesses wurde schon festgestellt. Betrachtet man die Qualitätsanforderungen, so erkennt man leicht, daß z.B. das Ziel der Korrektheit nur erreicht werden kann, wenn nicht nur die Sprachen formalisiert sind, sondern auch die Aktivitäten, wie etwa Transformation oder Verifikation, auf den Software-Komponenten formal definiert sind. Daher befaßt man sich auch mit der formalen Definition von Entwicklungsaktivitäten. Dieser Versuch kann natürlich nicht vollständig in Hinsicht auf eine ganze Methodik sein, da diese auch nichttechnische Aspekte, wie die Kommunikation zwischen Projektbeteiligten umfaßt. Der Begriff der Korrektheit spielt, wie auch noch detaillierter erläutert wird, eine zunehmend zentralere Rolle in der Software-Entwicklung. Er kann entscheidend durch die Verifikation, also dem mathematischen Nachweis der Korrektheit, von Aktivitäten als Qualitätsziel unterstützt werden. Die Verifikation wird in Kapitel 7 detailliert diskutiert. Die formale Definition von Entwicklungsaktivitäten erleichtert auch die Umsetzung in Werkzeuge zu deren Unterstützung.

1.3.2 Korrektheit von Software

Programme werden in einem syntaktischen Formalismus formuliert. Ihre Ausführung wird häufig als operationelle Semantik gedeutet. Mit Mitteln der Mathematik kann die Ausführung abstrakt und rechnerunabhängig definiert werden. Damit ist die Grundlage bereitet, die Frage nach der Korrektheit von Programmen zu stellen und auch beantworten zu können. Die Schlüsselfrage bei der Entwicklung und dem Einsatz von Programmen lautet:

> Löst das entwickelte Programm die gestellte Aufgabe?

Dies ist die Frage nach der *Korrektheit* des Programmes. Die Frage ist so fundamental, weil weitere Anforderungen wie Effizienz oder Benutzungsfreundlichkeit ihre Bedeutung verlieren, falls die Korrektheit nicht gegeben ist. Im allgemeinen fallen Aussagen zur Korrektheit allerdings schwer (zu dieser Diskussion siehe auch [Kre91]). In der Regel wird Korrektheit nur durch Plausibilitätsbetrachtungen oder Testen begründet. Nur unter günstigen Umständen kann eine verläßlichere Aussage erwartet werden. Eine Korrektheitsaussage zu beweisen oder zu widerlegen ist nur innerhalb einer mathematischen Theorie möglich. Die Aussage muß dazu vollständig in der Theorie formulierbar sein, ebenso müssen die Anforderungen ausdrückbar sein. Der Aussagekraft von Beweisen sind allerdings Grenzen gesetzt. Kreowski [Kre91] schreibt dazu:

> Ist die Korrektheit eines Programmes, daß es also leistet, was es soll, innerhalb einer logischen Theorie gezeigt, so gilt diese Erkenntnis nur in dieser Theorie.

Weiterhin gibt es Rahmenbedingungen, die die Eingeschränkheit des Ansatzes formaler Korrektheitsbetrachtung im Gesamtkontext der Software-Entwicklung beschreiben. Kreowski [Kre91] schreibt dazu:

> Die eigentliche Aufgabenstellung der Software-Entwicklung stammt aus dem sozialen Gefüge lebendiger Menschen. Wie gut sie sich in die formalisierten Anforderungen einer Theorie übertragen läßt, entzieht sich der Klärung.

und

> Ein Programm ist nicht nur eine substanzlose Größe in einer Theorie, sondern vielmehr materielles Objekt, das auf einer Maschine läuft [...]. Es ist damit offen für äußere, unkalkulierbare Einflüsse von Manipulation und Sabotage bis zum Materialfehler und Stromausfall.

Die Korrektheit von Software hat sich zu einem bedeutenden Qualitätsfaktor entwickelt, wie auch in dem vom Bundesministerium für Forschung und Technologie (BMFT) initiierten Verbundprojekt *Korrekte Software* (KORSO) erkannt wurde [BJ94]. Die Bedeutung der Korrektheit läßt sich aus folgenden Argumenten erkennen:

- Software kann in sicherheitsrelevanten Bereichen überhaupt nur eingesetzt werden, wenn eine weitgehende Funktionsgarantie gegeben werden kann.
- Produkthaftung, auch in nicht sicherheitsrelevanten Bereichen, ist nur möglich, wenn Zuverlässigkeit durch abgesicherte Entwicklungsmethoden erreicht werden kann.
- Fehler im Entwurf und deren nachträgliche Beseitigung erhöhen die Kosten unangemessen.
- Die Wiederverwendbarkeit von Software erfordert generische Bausteine mit genau spezifizierten Leistungsdaten und deren Prüfbarkeit im Rahmen eines Korrektheitsbeweises.

Ziel des Projektes KORSO war die Sicherstellung von Korrektheit durch Erarbeitung und Weiterentwicklung geeigneter Methoden, Sprachen und Werkzeuge für große Programmsysteme und deren Evaluierung in Fallstudien. Zur Realisierung korrekter Software haben sich folgende Aufgaben herauskristallisiert:

- Konzeption modularer und wiederverwendbarer Spezifikationen,
- formale Entwicklungsmethodiken zur Spezifikation und Transformation,
- Verifikation von Entwicklungsschritten.

1.3.3 Formale Spezifikationssprachen

Der Einsatz formaler Spezifikationen hat die abstrakte, also problemnahe und implementierungsunabhängige Beschreibung von Software-Systemen zum Ziel. Sie sollen dabei eine problemnahe Begriffsbildung ermöglichen, präzise und modular sein und durch ihre Formalität einen weitgehenden Rechnereinsatz erlauben. Problemnahe Begriffsbildung erleichtert die Kommunikation mit dem Auftraggeber und auch zwischen anderen Projektbeteiligten. Präzision hilft Widersprüche und Mehrdeutigkeiten zu vermeiden. Die Abstraktion ermöglicht die Verständlichkeit und im Zusammenhang mit Modularisierung auch die Übersichtlichkeit der Spezifikation.

Diese Punkte sind Ansätze zur Lösung von Problemen, die unter dem Schlagwort *Software-Krise* bekannt geworden sind (siehe [Fen93, NS89, Mey85]). Programme, die eine bestimmte Größe überschritten hatten, erwiesen sich als unkalkulierbar in ihren Entwicklungskosten, Entwicklungszeiten und Wartungskosten. Mit dem aufkommenden Zweig der Software-Technologie in der Informatik wurden Lösungsansätze gefunden, die auf strukturierten Entwicklungsmethoden und Spezifikationsansätzen beruhten. Vor allem durch höhere Anforderungen an Robustheit, Zuverlässigkeit und Korrektheit fanden formale Methoden mehr und mehr Eingang in die Software-Technologie. Eigenschaften formaler Spezifikationen sind Vollständigkeit, Widerspruchsfreiheit und Eindeutigkeit. Damit bieten sie die Möglichkeit, die Kommunikation zwischen Projektbeteiligten zu verbessern und außerdem den Konstruktions- und Verifikationsprozeß zu unterstützen.

Der Sinn formaler Spezifikationen liegt nicht nur in der Ermöglichung von Korrektheitsnachweisen. Als Hauptvorteil wird von vielen gesehen, daß formale Spezifikationen während des Spezifikationsprozesses helfen, Anforderungen oder Fragen zum Systementwurf zu klären, Entwurfsfehler und Mehrdeutigkeiten aufzudecken und Entscheidungen über funktionale Eigenschaften zum richtigen Zeitpunkt zu fällen.

Mit Kritik am Einsatz formaler Methoden setzen sich u.a. [Fen93, NS89, Mey85, Krä92, Gau91] auseinander.

1.3.4 Verifikation

Die Qualitätssicherung, besonders in Hinsicht auf den Faktor Korrektheit, wurde schon mehrfach angesprochen. Die Korrektheit eines sich entwickelnden Systems muß durch das Führen von Beweisen sichergestellt werden. Dies wird *Verifikation* genannt [AO91, AO94, Fra92, LS84]. Da formale Methoden bisher wenig verbreitet sind, wird Qualitätssicherung in der Regel durch Testen erreicht. Testen erlaubt aber nur das Zeigen der Anwesenheit von Fehlern. Das eigentliche Ziel, die Abwesenheit von Fehlern systematisch zu beweisen, ist die Aufgabe der Programmverifikation. In Formeln einer Logik (in der Regel Prädikatenlogik erster Stufe) wird das gewünschte Verhalten des Programmes spezifiziert. Mit einem Beweissystem (einer Menge von Axiomen und Regeln) kann das tatsächliche Programmverhalten ermittelt und auf Korrektheit gegenüber der Spezifikation überprüft werden. Dieses Vorgehen wird *axiomatisches Beweisen* genannt. Zum axiomatischen Beweisen ist zusätzlich zur Sprache, in der das zu beweisende Programm vorliegt, ein weiterer (logischer) Formalismus notwendig, der die relevanten Programmeigenschaften ausdrückbar macht. Die meisten der Beweissysteme im Kontext zustandsbasierter Systeme werden als *Hoare-style* Beweissysteme bezeichnet (nach C.A.R. Hoare [Hoa69, Apt81]). Dieser Ansatz nennt sich auch *Zusicherungsmethode*. Formeln, die Zustände eines Programmes beschreiben, heißen auch *Zusicherungen*.

Beweise für größere Software-Systeme sind in der Regel so komplex, daß sie ohne maschinelle Unterstützung für den Menschen nicht beherrschbar sind. Auch für Verifikationsverfahren und Verifikationswerkzeuge stellt sich immer mehr die Forderung nach Modularität und Kompositionalität. D.h. die Verifikation großer Systeme läßt sich auf die Verifikation von Einzelsystemen zurückführen. Schon geleistete Beweisarbeit sollte sinnvoll wiederverwendet werden können. Der Werkzeugeinsatz zur Verifikation ist unumgänglich.

Verifikation ist immer ein *a posteriori*-Verfahren, d.h. erst wenn Spezifikation und Programm vorhanden sind, kann die Korrektheit der Implementierung verifiziert werden. Um systematische Programmentwicklung konstruktiv betreiben zu können, ist Verifikation nicht geeignet.

Die Ansätze nach Dijkstra [Dijk76] und Gries [Gri81] zeigen aber, daß Prinzipien und Konzepte durchaus einsetzbar sind. Vielversprechend sind auch Ansätze wie *temporale* oder *dynamische Logiken*, die die Hoare-Logik in ihrer Mächtigkeit erweitern (siehe [KT90, Sti91, Har84]).

1.4 Zustandsbasierte Beschreibung von Software-Systemen

1.4.1 Beschreibung von Software

Spezifikationen und Implementierungen bilden zusammen die wesentlichen Beschreibungen eines Software-Systems. Während Implementierungen nur Beschreibungen eines Software-Systems sind, die ausführbar sein sollen, müssen mit Spezifikationen mehrere Aufgaben bewältigt werden. Sie sind zum einen Beschreibungen der Kundenwünsche; die Spezifikation dient somit als Vertrag zwischen Auftraggeber und Auftragnehmer. Der Auftragnehmer entwickelt das Software-System auf Basis dieses Vertrages, die Spezifikation ist also eine Anforderungsbeschreibung an die Implementierung. Da Software nach der Erst-Installation noch weiterentwickelt wird, sind Spezifikationen auch Referenz für die Wartung. Zwei Betrachtungsweisen sind also bei der Beschreibung von Software-Systemen zu berücksichtigen und geeignet zu integrieren: *Was* soll ein System leisten und *wie* ist es realisiert?

Beschreibungen werden in unterschiedlichen Stilen abgefaßt. Zum einen variiert der Grad der Formalität, er reicht von *informeller*, d.h. natürlich-sprachlicher Beschreibung, über *semiformale* (strukturierte natürlich-sprachliche Beschreibungen) bis zu vollständig formalen Beschreibungen. Zum anderen werden *operationale* und *deskriptive* Spezifikationen unterschieden. Operationale Beschreibungen geben ein Modell für das gewünschte System an. Vertreter dieser Art sind Datenflußdiagramme, endliche Automaten und Petrinetze. Deskriptive Spezifikationen beschreiben Eigenschaften des gewünschten Systems. Vertreter sind Entity-Relationship-Diagramme sowie logische und algebraische Spezifikationen.

1.4.2 Komponentenbegriff

Traditionelle Bereiche der Software-Beschreibung sind Spezifikation und Implementierung. Die Grenze zwischen diesen Bereichen wird immer stärker verschwimmen. Die zukünftige Entwicklung sollte Software-Beschreibungssprachen bringen, die prinzipiell ausführbar sind. Eine Programmiersprache, die die Grenze schon in Teilen überschreitet, ist die Prototyping-Sprache ProSet [FGH+93, DFG+92a, DFG+92b]. ProSet soll in bezug auf das Komponentenmodell CM (*Component Model*, siehe [Goe93]) evaluiert werden (siehe auch [Pah94]). Das Komponentenmodell stellt ein Evaluierungsschema für existierende Sprachen vor, dessen Kriterien sich aus den Anforderungen von Software-Entwicklungsprozessen unter besonderer Berücksichtigung von Wiederverwendung und verteilten Entwicklungsteams ergeben. Diese Kriterien decken nicht nur die produktorientierte Sicht (d.h. Anforderungen an die Fähigkeit der Sprache, Komponenten zu beschreiben) ab, sondern beziehen auch die Unterstützung des Entwurfsprozesses in die Evaluierung ein.

Im einzelnen soll die Sprache ProSet anhand folgender Kriterien untersucht werden (siehe auch [Goe93, Pah94]):

- *Unterstützung von Einkapselung:* Dies ist das essentielle Kriterium der produktorientierten Sicht. Es sind die interne Integrität der Komponenten, das Vorhandensein einer

Schnittstelle mit geeigneter Sprache, Beschreibbarkeit von Semantik in Schnittstellen sowie aktueller und formaler Import zu untersuchen.

- *Trennung zwischen Komponenten und Konfiguration zwischen Komponenten:* Realisierungsmöglichkeiten in einer Sprache sind explizite bzw. implizite und komponenteninterne bzw. separate Beschreibung von Komponentenbeziehungen.

- *Verfügbare Strukturen für Beziehungen zwischen Komponenten:* Hier erfolgt die Analyse der Beziehungen (etwa Graphenstrukturen, Subkonfigurationen).

- *Behandlung von Systemaspekten:* Umfaßte Aspekte sind Funktionalität, Nebenläufigkeit, Persistenz, Ausnahme- und Fehlerbehandlung, Verteiltheit, quantitative Aspekte.

- *Abdeckung des Software Life Cycles:* Anforderungsdefinition, Architekturentwurf, Detailentwurf und Implementation sind durch die Sprache zu unterstützen.

- *Einfluß von Design-Operationen:* Der Einfluß der Entwurfsprinzipien *divide and conquer* und *separation of concerns* auf die Auswahl von Entwurfsoperatoren ist zu untersuchen.

Die Evaluierung von ProSet liefert folgende Bewertung[2], dargestellt in einem Bewertungsschema (hier nur für Module) anhand einer Klassifikation des Unterstützungsgrades für jedes Kriterium (*keine/schwache/starke Unterstützung*):

ProSet	Komponenten:	Relationen zwischen Komponenten:
	Module	Import/Export, formale/aktuelle Parameter

Einkapselungsunterstützung:	Trennung von Komponenten und deren Konfigurationsbeschreibung:	Struktur der Beziehungen zwischen Komponenten:	Behandlung von Systemaspekten:	Abdeckung des Software Life Cycles:	Einfluß von Design-Operationen:	Zusammenfassung:
schwach	schwach/ stark	schwach	stark	schwach	schwach/ stark	schwach

Aus Sicht des Komponentenmodells realisiert ProSet zum komponentenorientierten Entwickeln einige geeignete Konzepte. Es existiert ein Modulkonzept mit Import- und Exportschnittstellen. Formaler Import ist möglich. ProSet verknüpft Komponenten dynamisch. Die Aktualisierung des formalen Modulimports (Konfigurationen) kann in der Sprache durch ein Instantiierungskonstrukt ausgedrückt werden. Mächtige Ausdrucksmittel, die im operationalen Bereich verschiedene Abstraktionsebenen abdecken, sind vorhanden (ProSet ist eine Breitbandsprache und unterstützt transformationelles Programmieren, siehe [DFG+92a, Eve95]). Prozeduren und Ausnahmebehandlung zur Realisierung des *divide and conquer*-Prinzips sind vorhanden. Nicht in vollem Umfang zum komponentenorientierten Entwickeln geeignete Konzepte von ProSet sind die mangelhafte Trennung von Sichten (das Prinzip *separation of concerns* ist nur ansatzweise unterstützt), die fehlende semantische Beschreibbarkeit von Prozeduren und Modulen und die nicht ausreichende Unterstützung der Beschreibung

[2] Eine Analyse anderer Sprachen, wie etwa Ada, Modula oder Eiffel, hätte ähnliche Bewertungen zur Folge.

auf unterschiedlichen Ebenen der Abstraktion/Detaillierung. Die mangelnde Unterstützung der Beschreibung von Schnittstellen wurde insbesondere im Einsatz der Sprache ProSet deutlich (vgl. Abschlußbericht der Projektgruppe 240 [Pro95]). Diese wesentlichen Kritikpunkte treffen auch auf viele andere imperative oder objektorientierte Sprachen zu.

1.4.3 Zustandsbasierte Systeme

Ein wesentliches Ziel dieser Arbeit ist die Motivation formaler zustandsbasierter Spezifikation im Kontext komponentenorientierter Software-Beschreibungssprachen. *Objekte* sind die Grundbausteine, aus denen sich ein Software-System zusammensetzt. Objekte verkapseln jeweils einen Teil des Systemzustands. Dieser Zustand kann von anderen Objekten nur durch zur Verfügung gestellte Operationen inspiziert oder modifiziert werden. Verschiedene Objekte kommunizieren miteinander nur über Operationsaufrufe. Dieser Ansatz wird in der Literatur auch *objektbasiert* genannt. Bei Benutzung von Vererbungsbeziehungen wäre er *objektorientiert* (vgl. [Weg90]).

Als einführendes, motivierendes Beispiel sollen Fragmente einer imperativen Spezifikation (die Syntax ist an ProSet angelehnt) eines Schalters mit zwei Zuständen (*on, off*) mit darauf arbeitenden Operationen, die den Status des Schalters abfragen ($GetStatus$), und die ihn ein- bzw. ausschalten ($SetOn, SetOff$), vorgestellt werden. Der aktuelle Zustand des Schalters wird in der modulinternen Variable *status* gespeichert. Es soll aus Gründen der Übersichtlichkeit nur eine Operation spezifiziert werden.

```
module Schalter;
        exports SetOn, SetOff, GetStatus;      – Export von Operationen
        visible status;                        – globale, sichtbare Variable im Modul
begin
        procedure SetOn (id);
        begin
                if GetStatus(id) = off  then
                        status := on;
        end SetOn;
end Schalter;
```

Eine algebraisch orientierte Schnittstellenspezifikation der Funktionalität der Operationen, etwa

$$\begin{array}{l} \textit{procedure SetOn: ident} \rightarrow \textit{ident} \\ \qquad\qquad \textit{GetStatus(id)} = \textit{off} \rightarrow \textit{GetStatus(SetOn(id))} = \textit{on} \end{array}$$

würde eine geeignete Formulierung der Prozedur *SetOn* mit einem Rückgabewert des Typs *ident* voraussetzen. Die im imperativen Programmierparadigma übliche Formulierung über Seiteneffekte auf dem Zustand kann damit nicht adäquat spezifiziert werden. Eine zustandsorientierte Formulierung mit Vor- und Nachbedingungen[3]

[3] Es wird hier die Syntax des *box*-Operators $[.]\phi$ modaler Spezifikationen benutzt (vgl. Kapitel 5).

module Schalter;
exports
procedure SetOn: state $\times$ ident $\rightarrow$ state
GetStatus(id) = off $\rightarrow$ [SetOn(id)] GetStatus(id) = on,
attribute GetStatus: state $\times$ ident $\rightarrow$ statustype
...

ermöglicht eine angemessene Spezifikation eines zustandsbasierten Systems. Mit *state* ist hier eine Abstraktion des internen Zustandes beschrieben.

Einerseits erweisen sich algebraische (d.h. funktionale) Ansätze also nicht immer als adäquat, zustandsbasierte Systeme natürlich zu beschreiben. Andererseits reicht ein in die Programmiersprache integrierter, auf booleschen Ausdrücken basierender Zusicherungsansatz (*assertions*) ebenfalls nicht, um ausreichend abstrakt beschreiben zu können ('*Spezifikationen sind Aussagen über Programme*'). Es liegt also nahe, ein Konzept einzusetzen, das Objekte als abstrakte Speicher zu beschreiben gestattet, ohne allerdings explizit ausführungsbezogen spezifizieren zu müssen. Der Begriff des Objekts muß dafür formalisiert werden. Ein *Berechnungsmodell* wird dazu entwickelt. Dieses wird sowohl operationale als auch axiomatische zustandsbasierte Beschreibungen erlauben. Die axiomatische Beschreibung wird auf einer zustandsbasierten Spezifikationslogik basieren, die als Kern die oben schon benutzten Vor- und Nachbedingungen enthält.

Zu einem komponentenorientierten Ansatz gehören auch parametrisierte Spezifikationen und Relationen zwischen Spezifikationen. Untersuchungen [Goe93] haben gezeigt, daß neben der produktorientierten Sicht die prozeßorientierte Sicht gleichrangig ist. Dieses Ergebnis wird auch im Komponentenmodell CM ausgedrückt. Hierbei geht es um die Umsetzung von Entwurfs- und Implementierungstechniken durch Konzepte der Sprachen. Insbesondere unterstützt ein Parametrisierungskonzept für Komponenten die Modularisierung. Verfeinerung, Implementierung oder auch Wiederverwendung können durch geeignete Operatoren und Relationen operationalisiert werden. Im Abschnitt 1.2 wurde bereits angesprochen, daß die Methodik von besonderer Bedeutung für die Sprachgestaltung ist. Horizontale und vertikale Entwicklung werden betrachtet. Unter *vertikaler Entwicklung* versteht man das Implementieren, also das Senken des Abstraktionsniveaus. *Horizontale Entwicklung* besteht aus der Anwendung von Spezifikationsoperatoren und Modularisierung ohne Senken des Abstraktionsniveaus. Wichtig ist eine Gestaltung der Relationen so, daß diese die verteilte Entwicklung in Gruppen ermöglichen und weitgehend durch Werkzeuge unterstützbar sind.

Algebraische Spezifikationen haben sich als geeignet erwiesen, verschiedenste Datentypen abstrakt zu beschreiben oder die Syntax und die Semantik von Programmiersprachen zu definieren. Häufig werden aber Mechanismen benötigt, die es erlauben, zustandsbasierte Systeme problemnah beschreiben zu können. Dazu gehören Programmvariablen oder Operationen mit Seiteneffekten. Basierend auf von-Neumann'schen Rechnerarchitekturen ist die imperative Programmierung immer noch das am weitesten verbreitete Programmierparadigma (und wird es wohl auch noch auf absehbare Zeit bleiben). Daher ist es nur sinnvoll, imperative Programmierung durch spezielle Mechanismen zum Spezifizieren und Beweisen zu ergänzen. Diese Mechanismen könnten die aus modelltheoretischen Ansätzen wie VDM oder Z bekannten Vor- und Nachbedingungen und Invarianten sowie zugehörige Beweissysteme sein. Eine Integration algebraischer und modelltheoretischer Ansätze soll hier versucht werden. Der Wunsch nach Unterstützung zustandsbasierter Spezifikation läßt sich aber nicht nur aus Sicht der Programmierung begründen. Zustandsbasierte Spezifikationsmechanismen ermöglichen eine problemnahe Handhabung bestimmter zu modellierender Systeme, wie etwa Dateisysteme oder Datenbanksysteme. Hierdurch kann auch insbesondere die Kombination algebraischer und zustandsbasierter Techniken motiviert werden. Zustände eines Datenbanksystems sind so komplex strukturiert, daß sich zur Beschreibung der Zustände selbst algebraisch spezifizierte Datentypen anbieten. Zustände werden so als Algebren realisiert[4]. In zustandsbasierten Systemen ist es möglich, auf die explizite Benutzung des Zustandes zu verzichten, so wie man es auch von der Benutzung eines realen, zustandsbasierten Systems (etwa eines Betriebssystems) erwarten würde, da der Zustand dort inhärent ist. Eine weitere Unterstützung dieses Ansatzes stammt aus dem Bereich der objektorientierten (objektbasierten) Entwicklung. Dort sind Objekte die strukturgebenden Blöcke eines Software-Systems, die sich an dem zu modellierenden Weltausschnitt orientieren. Objekte kapseln einen Teil des Zustands des Gesamtsystems und stellen Operationen bereit, über die ausschließlich der gekapselte Teil des Zustands inspiziert oder manipuliert werden kann. Diese objektbasierte Sicht wird auch von Spezifikationssprachen wie Π [CFGGR91] favorisiert.

Auf dem gleichen konzeptionellen Modell der Objekte basieren auch objektorientierte Datenbanken [Heu91]. Die hier aufgeworfenen Probleme der Formalisierung des Objektbegriffs sowie der Spezifikation und Implementierung von Objektsystemen sind auch im Bereich der Datenbanken mit besonderem Interesse betrachtet worden. Hier soll insbesondere das ESPRIT-BRA-Projekt IS-CORE (Information Systems – Correctness and Reusability) genannt werden [LK93].

An dem motivierenden Beispiel läßt sich erkennen, daß drei wesentliche formale Bereiche bearbeitet werden müssen, um einen solchen Sprachansatz realisieren zu können:

- **I** Ein formales **Berechnungsmodell**.
- **II** Eine *imperative Sprache* und eine **Spezifikationslogik**.
  Beide können auf dem Berechnungsmodell definiert werden.
- **III** **Relationen zwischen Spezifikationen**.
  Sie dienen der horizontalen und vertikalen Entwicklung.

Die wesentlichen Elemente des Ansatzes sind in Abbildung 1.2 zusammengefaßt. Die Basis bildet das Berechnungsmodell (die Σ-Objekte). Darauf sind sowohl operationale als auch deskriptive Spezifikationssprachen als zwei Formen zustandsbasierter Spezifikation definierbar. Darüberhinaus werden horizontale und vertikale Entwicklung durch Konzepte für beide Sprachen unterstützt.

Der angesprochene Sprachansatz muß so gestaltet werden, daß ein fließender Übergang von axiomatischen zu operationalen Spezifikationen möglich ist. Es erfolgt eine Beschränkung auf sequentielle, deterministische Programme. Verteiltheit und Parallelität werden nicht betrachtet.

[4] Eine weitergehende Motivation der zustandsbasierten Spezifikation findet sich in [FJ92, Feij94]. Die dort beschriebene Sprache COLD ist aus den Erfahrungen der ESPRIT-Projekte METEOR und FAST entstanden. Dort wurde mit rein algebraischen Sprachen wie ASL oder PLUSS begonnen. Es hat sich gezeigt, daß bedingte Gleichungen und initiale Semantik nicht mehr reichen, sondern daß für komplexe Anwendungen der volle Umfang einer Prädikatenlogik und lose Semantik unentbehrlich sind. Diese Erkenntnisse flossen dann in den Entwurf der Sprache COLD ein.

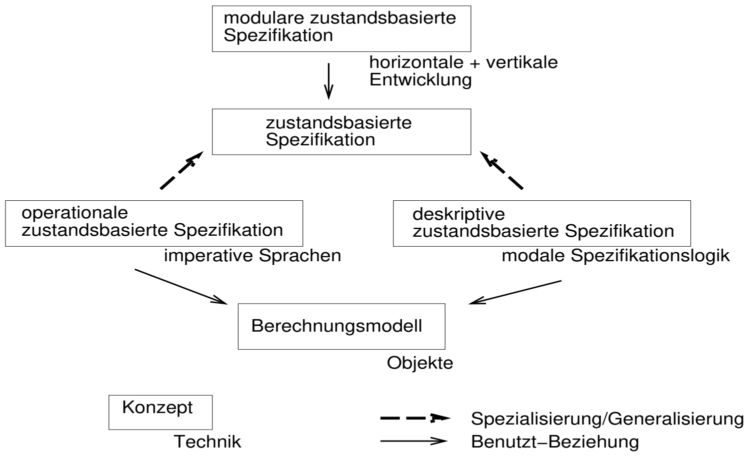


Abbildung 1.2: Zustandsbasierte Spezifikation

1.5 Vergleichbare Ansätze

In diesem Abschnitt sollen einige Ansätze vorgestellt werden, die zu dem hier präsentierten in Beziehung stehen. Eine Reihe von Spezifikationsansätzen ist hier interessant. Diese lassen sich in drei Kategorien einteilen:

1. Spezifikationsansätze mit Ausführungsaspekten sind:

   - die komponentenorientierte Spezifikationssprache Π [GSC91, GS94, Cra94],
   - der zustandsbasierte Spezifikationsansatz COLD [FJ92, Feij94],
   - der algebraische Ansatz CIP [BBB$^+$85],
   - der Ansatz nach R. Breu [Bre91].

   Alle hier angeführten Ansätze bieten formal definierte Spezifikationslogiken, COLD auf Spezifikationsebene sogar zustandsbasiert. Die prozeßorientierte Sicht wird etwa von Π in weitem Umfang unterstützt. CIP unterstützt Implementierungsaspekte durch Transformationskonzepte. Aspekte der Programmierung werden von allen unterstützt.

2. Spezifikationsansätze ohne Ausführungsaspekte sind:

   - algebraische Ansätze wie ACT ONE, ASL, OBSCURE, SPECTRUM (eine Übersicht ist in [Wir95] zu finden),
   - modelltheoretische Ansätze wie VDM und Z [Jon90, Dil91].

   VDM und Z sind zustandsbasiert, formal definiert und werden in der Praxis eingesetzt; sie sind aber nicht eigenschaftsorientiert. Außerdem unterstützen sie Programmiersprachen nicht direkt. In einigen algebraischen Sprachen sind weitreichende Spezifikationsoperatoren und Spezifikationsrelationen integriert. Ebenfalls in diese Klasse gehören die zustandsbasierten, axiomatischen Ansätze von Ryan, Fiadero und Maibaum [FMR91] und Wieringa [Wie91]. Diese beiden Arbeiten sind dem datenbankorientierten Kontext zuzuordnen.

3. Erweiterungen realer Programmiersprachen zu Spezifikationsansätzen sind:
   - die Erweiterung von Standard-ML zu Extended-ML [KST94, ST90a, SST90],
   - Annotierungen von Ada-Programmen durch Anna [LvH85],
   - die Interface-Sprachen der Larch-Familie von Spezifikationssprachen [Win87, CL94, GH93].

   Alle Ansätze sind formal definiert. Anna ist allerdings nicht durch eine Spezifikationslogik definiert. ML ist eine funktionale Sprache, gleiches gilt somit für EML. Auf Spezifikationsebene unterstützt Larch algebraische Spezifikationen. Die prozeßorientierte Sicht wird von allen drei Ansätzen nur ansatzweise unterstützt.

Es existieren noch eine Reihe anderer Ansätze, die mit dem vorliegenden vergleichbar sind. Im Bereich der *Berechnungsmodelle* sind evolvierende Algebren [Gur93, Gur94], Kripke-Modelle [Har84, KT90], Algebren [Wir90, EM85] und Action Semantics [Wat91, Mos89] zu nennen. Sie werden später noch eingehend betrachtet. Einige Konzepte imperativer und objektorientierter Programmiersprachen wie Vererbung, Zusicherungen, Ausnahmen und Typisierung werden aufgegriffen. Im Kontext der Verifikation spielen die Kalküle des Hoare'schen Ansatzes eine bedeutende Rolle [Fra92, AO94, LS84].

Einige der *objektorientierten Sprachen*, wie z.B. Eiffel [Mey92b, Mey88, Mey92a], verbinden Programmierung mit Ansätzen zur Spezifikation abstrakter Datentypen und der Operationsspezifikation über Vor- und Nachbedingungen. R. Fischbach fordert in [Fis92] eine stärkere Unterstützung formaler Spezifikation in Eiffel. Rosenblum [Ros95] beschreibt den Einsatz eines Zusicherungskonzeptes, das über den Umfang von Eiffel hinausgeht. F. van der Linden [vdL94] kombiniert COLD mit objektorientierten Methoden.

Einige der eben erwähnten Ansätze sollen nun detaillierter vorgestellt werden.

1.5.1 Spezifikationsansätze mit Ausführungsaspekten

Die Sprache Π

Π ist eine Spezifikationssprache zur Unterstützung der komponentenorientierten Software-Entwicklung [GSC91, CFGGR91, Goe93, GS94, Cra94]. Π ist also eine Komponentenbeschreibungssprache. *Objekte* sind hier die strukturgebenden Komponenten eines Software-Systems. Basiskonzept zur Beschreibung von Objekten ist ein Konzept abstrakter Datentypen. Konzepte zur Beschreibung modularer und hierarchischer Strukturen für eine Systembeschreibung werden bereitgestellt.

Das *3C-Konzept* (siehe [GSC91]) gibt eine Strukturierung von Komponenten durch die Aspekte *Content*, *Concept* und *Context* vor. *Content*, der Inhalt, ist die eigentliche Realisierung der Komponente, *Concept* eine Abstraktion der Realisierung und *Context* legt die von der Umgebung benötigten Ressourcen fest. Dieses *3C-Konzept* wird in den Π-Komponenten umgesetzt.

Hauptkonstrukt von Π ist das CEM (Concurrently Executable Module)[5], es dient zur Beschreibung einer Klasse von Objekten (ein Objekt ist also die Implementierung eines abstrakten Datentypen). Resultierend aus den Prinzipien *divide and conquer* und *separation of concerns* gibt es orthogonal zueinander

[5] Die Begriffe *Datenabstraktion* und *Nebenläufigkeit* spielen in Π eine gleichwertige Rolle.

1. ein Sichtenkonzept mit den Sichten *type view* (ausführungsinvariante Eigenschaften), *imperative view* (ausführungsbezogene Eigenschaften) und *concurrency view* (Sequentialisierungsoperationen) und

2. eine Zergliederung in die Sektionen *import section* (formale Anforderungen an den Import, beschreibt den *Context*), *export section* (Auflistung der von außen benutzbaren Operationen und ihrer Eigenschaften, beschreibt das *Concept*), *body section* (Definition der Implementierung, beschreibt den *Content*) und *common parameters section* (Systemparameter).

Das konzeptionelle Modell eines Software-Systems in Π wird im folgenden charakterisiert. Objekte sind hierarchisch strukturiert, d.h. Zustände von Objekten werden auf die Subobjekte verteilt. Der Aufbau von Hierarchien erfolgt durch Aktualisierung der Importe der parametrisierten Komponentenbeschreibung in einem Konfigurationskonstrukt. Die Kommunikation von Objekten erfolgt nur über Operationsaufrufe (*shared operations*), sofern die Concurrency-Integritätsbedingungen dies zulassen. Durch gemeinsam genutzte Objekte (*shared objects*) kann aus der Baumstruktur der Subobjektstruktur auch ein Graph entstehen.

Der einzige von vornherein unterstützte Datentyp ist *bool*. Alle weiteren müssen vom Benutzer zur Definition seiner Implementierungplattform festgelegt werden. Um eine Implementierungsplattform auf geeignetem, vom Benutzer bestimmtem Abstraktionsniveau definieren zu können, können Spezifikation als *is_basic* typisiert werden. Die Implementierung solcher Basis-CEMs wird nicht in der *body*-Sektion beschrieben, sondern als an anderer Stelle realisiert angenommen.

Die *Typsicht* (type view) umfaßt Beschreibungen statischer Eigenschaften der Objekttypen. Von Aspekten wie Speicherung oder Ausführung wird abstrahiert. Der benutzte Ansatz ist der der algebraischen Spezifikationen [EM85, Wir90]. Mehrere Sektionen bilden jeweils zusammen eine algebraische Spezifikation. Common Parameters und Export bilden das *Export Interface*. Common Parameters und Import bilden das *Import Interface*. Rumpf, Common Parameters und Import bilden die *Rumpfspezifikation*.

In jeder Sektion sind Folgen von Datentypspezifikationen erlaubt. Export und Rumpf definieren zusammen den Objekttypen des CEM. In einer Konstruktionsklausel kann die Repräsentation des Objekttypen näher spezifiziert werden. Datentypspezifikationen sind in getrennte Operationsspezifikationen aufgeteilt. Operationen werden durch Gleichungen spezifiziert.

Während die Aspekte Speicherung und Ausführung in der Typsicht ausgeschlossen waren, sind sie in der *imperativen Sicht* (imperative view) die zentralen Anliegen. Hier sind Objekte abstrakte Speicher, die von Operationen inspiziert oder modifiziert werden. Es können Algorithmen zur Ausführung dieser Operationen angegeben werden, die aus üblichen imperativen Konstrukten sowie einem Parallelitätskonstrukt bestehen können. Diese Algorithmen dürfen nur im Rumpf spezifiziert werden, in den Schnittstellen sind nur syntaktische Informationen (Signaturen) erlaubt. Parameter können als Wert-, Resultat- bzw. Wert/Resultat-Parameter gekennzeichnet werden.

Die *Nebenläufigkeitssicht* (concurrency view) stellt Mittel bereit, die Auswertungsreihenfolge auf einem Objekt zu spezifizieren. Insbesondere durch die Möglichkeit, Operationen nebenläufig auszuführen, ist die Notwendigkeit zur Festlegung von Ausführungsabhängigkeiten gegeben, etwa um Konsistenz zu sichern. Als Formalismus werden Prädikatpfadausdrücke, eine Erweiterung regulärer Ausdrücke über Operationsnamen, verwendet. Damit kann u.a. Sequenz, Alternative, Optionalität oder Nebenläufigkeit ausgedrückt werden. Außerdem können Vorbedingungen für Operationsausführungen angegeben werden.

In *Konfigurationen* werden einzelne CEMs durch Aktualisierung der Importschnittstelle verknüpft. Π eignet sich besonders zum Einsatz in einem komponentenorientierten Wiederverwendungskontext [CDG94]. Aspekte der Wiederverwendung in Π sind detailliert in [Cra94] nachzulesen.

COLD

COLD [Jon89a, KRdL89, Jon89b, Feij89, FJ92, Feij94, RdL94b, RdL94a, GRdL94] ist eine Entwurfssprache, die im Rahmen des ESPRIT-Projektes METEOR [WB89a, BF91] entwickelt wurde. Entwurfssprache bedeutet in diesem Zusammenhang, daß sie sowohl Spezifikations- als auch Implementierungselemente enthält. Der COLD-Ansatz stellt einen von Implementierungsaspekten unabhängigen Kern COLD-K zur Verfügung.

Klassen sind das Kernkonzept zur Strukturierung von Software-Systemen. Sie sind mit Modulen in Modula 2 oder Packages in Ada vergleichbar. Der Sprachansatz von COLD soll es erlauben, auf verschiedenen Ebenen der Abstraktion, d.h. von einer frühen Entwicklungsphase bis zur Implementierung, adäquat spezifizieren zu können. Eine Klasse kann als Beschreibung einer *abstrakten Maschine* gesehen werden. Sie besteht aus einer Kollektion von Zuständen mit einem Initialzustand. Zustände entsprechen Algebren. Der Zustand wird durch Prozeduren (den Instruktionen der Maschine) verändert. Zustände bestehen aus einzelnen Zustandskomponenten, die in einer Zustandssignatur zusammengefaßt werden. Eine Signatur umfaßt Sorten, Prädikate, Funktionen und Prozeduren. Der Beschreibungsteil für Zustandstransitionen ist vom Rest der Signatur getrennt.

Algebraische Klassenbeschreibungen sind eigentlich ein Spezialfall des allgemeinen, zustandsbasierten Ansatzes von COLD, in dem keine Prozeduren verwendet werden, der Zustand also nicht verändert wird. Somit ist die Semantik einer solchen Spezifikation durch eine Algebra bestimmt. Die Spezifikation der Prädikate und Funktionen erfolgt durch Hornklauseln einer Prädikatenlogik erster Stufe. Es wird ein Definiertheitsprädikat bereitgestellt, mit dem spezifiziert werden kann, daß ein Argument immer definiert sein soll bzw. auch partiell sein kann. Es gibt einen speziellen undefinierten Wert für jede Sorte. Funktionen und Prädikate sind strikt bzgl. dieses Wertes. Variablen existieren nicht explizit, sie müssen über Funktionsanwendung modelliert werden.

In den *axiomatischen zustandsbasierten Klassenbeschreibungen* sind implizite Prozedurdefinitionen möglich, d.h. die Prozeduren werden durch Axiome definiert. Der Formalismus ist dynamische Logik. Box- und Diamond-Operator der modalen Logiken (in COLD *always*- und *sometimes*-Operator genannt) sind definiert. Außerdem werden Sprachmittel bereitgestellt, die methodische Vorgehensweisen unterstützen. Mit einer *modifies*-Klausel kann angegeben werden, welche Zustandskomponenten von einer Prozedur verändert werden (siehe Abschnitt 4.5). Mit einer *depends*-Klausel kann für Funktionen und Prädikate angegeben werden, nach welchen Änderungen welcher Zustandskomponenten sich der Wert der Funktion bzw. des Prädikates ändert. Variablen können hier explizit benutzt werden. Der *PREV*-Operator erlaubt das Zugreifen auf den Wert einer Variablen vor Ausführung einer Prozedur. Mit ihm werden Zustandsänderungen beschrieben.

Algorithmische Klassenbeschreibungen erlauben die Implementierung der spezifizierten Operationen durch Algorithmen. Dazu werden primitive Kontrollstrukturen auf einer Kommandoebene bereitgestellt. Hierzu gehören ein leeres Kommando, *guarded commands*, Kommandokomposition, Auswahl und Wiederholung. Algorithmische Beschreibungen sind ausführbar.

Modularisierungs- und Parametrisierungskonzepte existieren. Signaturbasierte Exportschnittstellen sind formulierbar. Eine Importklausel kann flache Spezifikationen enthalten. Sie erlaubt benannten, d.h. nichtformalen Import. Als Anpassungsoperation steht ein Umbenennungsoperator zur Verfügung. Es gibt auch die Möglichkeit zum formalen Import. Parametrisierte Klassenbeschreibungen können über Lambda-Abstraktion definiert werden. Deren Komposition erfolgt dann über Lambda-Applikation. Einige weitergehende Konzepte sind verfügbar, die den Einsatz von COLD in großen Projekten erleichtern sollen. Hierzu gehören eine graphische Notation für Klassen oder auch *Designs*, eine Strukturierungsmöglichkeit auf Klassenebene.

Der Spezifikationsformalismus ist durch MLCM (Modal Logic of Creation and Modification), einer dynamischen Logik, definiert [GRdL94]. Die Formalisierung des Parametrisierungskonstruktes erfolgte durch das $\lambda\pi$-Kalkül [Feij89]. Description Algebras [Jon89a] bilden die semantischen Strukturen, in denen die Logik interpretiert wird.

Algebraische Spezifikation und imperative Programmierung

Ruth Breu verknüpft in ihrer Dissertation [Bre91] algebraische Spezifikation mit objektorientierter Programmierung. Basierend auf algebraisch spezifizierten Klassen werden drei Konzepte zur Verknüpfung von Klassen betrachtet: Subtypbeziehung, Vererbung und Benutztbeziehung. Subtypen unterstützen die datenorientierte Form der Verfeinerung (es wird Subtyp- oder Inklusionspolymorphismus realisiert [CW85]). Vererbung ist modulorientiert.

Die Semantik der Klassen wird durch Algebren definiert. Es wurde die Theorie partieller abstrakter Datentypen [BW82] mit loser Modellbildung [Wir90] eingesetzt. Vererbung und Subtypen sind Konzepte der vertikalen Entwicklung. Vererbung wird durch Modellverfeinerung realisiert, d.h. ein Modell eines Erben ist eine Erweiterung eines Modells seines Vorfahrens. Subtypen werden über Teilmengenrelationen auf Trägermengen der Klassen realisiert. Die Theorie hierfür liefern *order-sorted algebras* [GM87b, GM87a]. Die Benutztbeziehung ist ein Konzept der horizontalen Entwicklung und wird durch hierarchische abstrakte Datentypen [WPP$^+$83] definiert. Die horizontale Kompositionseigenschaft wird gezeigt.

Semantisch ist dieser Ansatz von der algebraischen Spezifikationssprache ASL [Wir86] beeinflußt. Zudem wurden objektorientierte Entwurfsprinzipien [Mey88] umgesetzt. Objektorientierte und damit zustandsbasierte Klassenbeschreibungen werden durch eine Menge von Algebren und eine Menge von Funktionen darauf modelliert, wobei letztere Interpretationen der Methoden sind, die den Objektzustand ändern. Der Ansatz basiert auf dem Ansatz von P. America [Ame90].

Es wird eine Beispielsprache OP vorgestellt, die in die gleiche Sprachklasse wie Eiffel oder C++ fällt (streng typisiert, sequentiell, objektorientiert). Eine Implementierungsrelation, die algebraische Spezifikation und objektorientierte Klassenbeschreibung verknüpft, wird durch Homomorphismen, d.h. Abstraktionsfunktionen auf Datenbereichen, definiert [BMPW86], die an den Zustandsbegriff von OP angepaßt wurden.

Aus methodischer Sicht sind hier besonders inkrementeller Entwurf und Wiederverwendung untersucht worden. Es wurde eine Integration von Spezifikation und Implementierung erreicht.

Einen ähnlichen Ansatz wie R. Breu verfolgt H. Lin in [Lin93]. Es geht dort um die Implementierung algebraischer Spezifikationen in einer imperativen Programmiersprache. Eine

Implementierung besteht aus einem Repräsentationstypen, Invarianten und einer geeigneten Äquivalenzrelation über ihnen sowie Prozeduren für jeden Operator der Spezifikation. Ein formaler Kalkül wird entwickelt, der es erlaubt, in bezug auf die Spezifikation korrekte Implementierungen zu erzeugen. Ein formaler Verfeinerungsbegriff wird eingeführt, der es erlaubt, eine algebraische Spezifikation einer Operation über eine Vor-/Nachbedingungsspezifikation in eine ausführbare Prozedurimplementierung zu transformieren. Damit wird eine formale Verknüpfung zwischen algebraischer Spezifikation und der Technik der Vor-/Nachbedingungsspezifikation geschaffen. Terme der Spezifikation werden in Programme transformiert. Dabei werden etwa Funktionskompositionen in Kommandosequenzen umgesetzt. Dijkstra's *guarded command language* ist Zielsprache. Korrekte Implementierungen müssen als Modelle der Spezifikation nachgewiesen werden. Weiterhin muß die Implementierung noch weitere, das beobachtbare Verhalten betreffende Eigenschaften erfüllen. Sind die Operationsspezifikationen durch Vor-/Nachbedingungspaare realisiert, können diese durch Techniken, wie sie in [Mor94, Dijk76, Gri81] vorgestellt sind, in ausführbaren Code transformiert werden.

1.5.2 Spezifikationsansätze ohne Ausführungsaspekte

VDM und Z

VDM [Jon90, Bjø91] und Z [SBC92, PST91, Dil91] sind zwei formale, modelltheoretisch orientierte Spezifikationssprachen. Beide Sprachen werden als modelltheoretisch bezeichnet, da sie die Semantik von Spezifikationen direkt in mathematischen Modellen definieren. In eigenschaftsorientierten Sprachen wird im Gegensatz dazu noch eine Interpretation zwischengeschaltet.

Z wurde an der Oxford University entwickelt. Die Sprache basiert auf Prädikatenlogik erster Stufe und endlicher Mengenlehre. Z unterstützt deklarative, separate Spezifikationen über Implementierungen. Dieser prozedurale Abstraktionsprozeß wird in bezug auf Operationen durch Vor- und Nachbedingungen erreicht. Sequenzen von Operationen können gebildet werden. Z beschreibt Daten, ihre Beziehungen und das funktionale Verhalten eines Systems durch Mengen, Relationen und Funktionen der Mengentheorie. Statische und dynamische Aspekte eines Systems werden gemeinsam in Schemata beschrieben. Schemata bestehen aus Definitionen (Signaturen) und Axiomen. Schemata können generisch sein. Zur Spezifikation kann auf eine Bibliothek vordefinierter Typen, Relationen und Funktionen zurückgegriffen werden. Schemata können kombiniert werden. Dazu stellt ein Schema-Kalkül Operationen wie Instantiierung, Konjunktion oder Disjunktion zur Verfügung.

VDM ist eigentlich eine Methode zur formalen Entwicklung von Software-Systemen. VDM besteht aus einer Spezifikationssprache VDM-SL, Regeln zur Daten- und Operationsverfeinerung und einer Beweistheorie, um Eigenschaften von Spezifikationen zeigen und die Korrektheit von Verfeinerungsschritten nachweisen zu können. In VDM-SL werden Systemzustände durch Datenmodelle und das Verhalten durch Operationen beschrieben. Operationen werden durch ihr Ein-/Ausgabeverhalten definiert. Vorbedingungen, Nachbedingungen und Invarianten spezifizieren Daten und Operationen. Alle Teile einer Spezifikation werden in einem Modul zusammengefaßt. Zu VDM gehören Entwicklungstechniken wie z.B. die Verfeinerung.

Für beide Sprachen gibt es Ansätze, die sich mit der Ausführbarkeit beschäftigen, diese sind aber nicht Bestandteil der Sprachen selbst.

MAL

Modal Action Logic (MAL) [FMR91] ist eine modale Logik, die im Rahmen des datenbankorientierten ESPRIT-Projekts IS-CORE entwickelt wurde. Statt von Objekten wird hier von *Agenten* (oder *Aktionen*) gesprochen. Der Verhaltensaspekt soll hier stärker betont werden. Agenten besitzen einen veränderbaren Zustand. In MAL ist es möglich, nicht nur einzelne Zustandsübergänge, sondern ganze Lebenszyklen eines Objektes spezifizieren zu können. Es können sogenannte Sicherheits- und Lebendigkeitseigenschaften gefordert werden (engl. *safety and liveness conditions*). Diese werden mit Hilfe *deontischer Axiome* ausgedrückt, die Agenten Zwänge auferlegen (*obligations*) und Erlaubnisse (Zugriffe) verweigern (*permissions*). Logiken der Zwänge und Erlaubnisse werden deontische Logiken genannt. Der nichtdeontische Teil von MAL entspricht einer eingeschränkten dynamischen Logik. MAL-Spezifikationen werden durch *Kripke-Modelle* [KT90] interpretiert.

Innerhalb von IS-CORE wurden noch weitere Formalisierungen von Objekten vorgenommen (siehe etwa [LK93, SJS91, FSMS91, FM90]). Da MAL dem hier vorliegenden Ansatz am nächsten kommt, werden die anderen hier nicht vorgestellt.

CMSL

R. Wieringa [Wie91, WM91] stellt die Sprache CMSL (Conceptual Model Specification Language) vor, die ordnungssortierte Gleichungslogik [GM87b] um Elemente einer dynamischen Logik erweitert. Wieringa nutzt CMSL um *dynamische Objekte* zu spezifizieren, d.h. Objekte, die einen veränderlichen Zustand haben. Anwendungen seines Ansatzes sieht Wieringa in der Spezifikation des Verhaltens objektorientierter Datenbanken. CMSL ist auf Kripke-Modellen sowie einer davon unabhängigen Prozeßalgebra definiert.

Zur Spezifikation von Zustandstransformationen wird der Box-Operator der modalen Logik benutzt. Er kann nur auf Prozeduraufrufe angewendet werden. Der Zugriff auf den Zustand eines Objektes (ggf. modifizierend) wird Ereignis genannt. Um diese Ereignisse beschreiben zu können, wird eine spezielle Sorte *event* eingeführt. Terme der Sorte *event* werden als Funktionen auf Kripke-Modellen (dort mögliche Welten genannt) interpretiert. Wird eine solche Spezifikation um die *event*-bezogenen Elemente reduziert, erhält man eine algebraische Spezifikation im üblichen Sinne. Durch die Ereignisspezifikationen sind nur neue Funktionselemente, aber keine Datenelemente hinzugenommen worden.

1.5.3 Erweiterungen von Programmiersprachen zu Spezifikationsansätzen

Extended ML

Extendend ML (EML) [KST94, ST90b, SST90] wurde als Rahmen für die formale Entwicklung von Standard ML (SML) Software-Systemen konzipiert. Die Entwicklung beginnt mit einer Spezifikation des Verhaltens und endet mit ausführbaren SML-Programmen. Die Erhaltung der Korrektheit ist dabei von besonderer Bedeutung. Die Spezifikationen erfolgen in der EML-Spezifikationssprache, die eine Erweiterung von SML um Axiome zur Beschreibung der Eigenschaften von Modulkomponenten ist. SML ist eine formal definierte Sprache, die ein ausgereiftes Modulkonzept zur Verfügung stellt. Damit wird formales Beweisen von Verhalenseigenschaften von SML-Programmen im EML-Kontext ermöglicht. SML-Module werden zu einem Konzept der '*Entwicklung im Großen*' erweitert.

Wie die Sprache ANNA [LvH85] Ada-Programme annotiert, ist auch EML ein Spezifikationsansatz, der eine reale Programmiersprache integriert. Ebenfalls ähnlich ist der Larch-Ansatz [Win87]. EML erweitert SML um Axiome, die in den SML-Signaturen und -Modulrümpfen benutzt werden dürfen. Die Logik ist eine Erweiterung einer Prädikatenlogik erster Stufe um Konzepte, die es erlauben, über den speziellen Sprachkonstrukten von SML (etwa polymorphe Typen und Funktionstypen) zu argumentieren. Die Semantik von EML ist durch eine *natürliche Semantik* (oder strukturell operationelle Semantik nach G. Plotkin, siehe [ST90a, MTH90]) definiert. Es werden statische (Typüberprüfung), dynamische (Ausführung) und Verifikationsaspekte unterschieden.

Larch

Larch ist ein Spezifikationsansatz, der Spezifikationskonzepte auf zwei semantisch unterschiedlichen Sprachebenen bereitstellt [Win87, CL94, GH93].

Larch-Spezifikationen haben Bezug sowohl zur modelltheoretischen als auch zur algebraischen Spezifikation. Im zustands- bzw. objektbasierten Kontext werden so Datentypabstraktionen von den Spezifikationen der Zustandstransformationen getrennt. Larch-Spezifikationen werden immer mit den Implementierungen einer bestimmten Programmiersprache verknüpft. Spezifikationen, die das Verhalten der Implementierungen abstrahieren, heißen *Interface-Komponenten*. Sie sind modelltheoretisch definiert. Im Falle einer zustandsbasierten Programmiersprache (Larch-Interfaces existieren etwa für Smalltalk, C oder C++) werden Zustandstransformationen in diesen Interface-Komponenten durch Vor- und Nachbedingungen, ausgedrückt in Prädikatenlogik, spezifiziert. Eigenschaften, die unabhängig vom Implementierungsparadigma sind, werden in den sogenannten *gemeinsamen Komponenten* (shared components) spezifiziert. Hierzu wird im algebraischen Stil eine Gleichungslogik erster Stufe eingesetzt. Damit ist dieser Teil von der Implementierungssprache unabhängig. Die neben den Zustandstransformationen implementierungsrelevanten Spezifikationen, wie Seiteneffekte, Fehler- und Ausnahmebehandlung oder Allokation von Ressourcen, müssen in den Interface-Komponenten erfolgen.

In Larch erfolgt eine strikte Trennung zwischen der Spezifikation von Modellen (abstrakten Werten) auf algebraischer Ebene und der Spezifikation von Interfaces für Programmodule. Interface-Spezifikationen erlauben keine Modellbildung im algebraischen Sinne. Dies impliziert, daß Interface-Spezifikationen auch nicht in den Vor- und Nachbedingungsspezifikationen benutzt werden dürfen. Das Vokabular hierfür wird durch algebraische Spezifikationen vorgegeben. Diese Spezifikationen werden *Traits* genannt. Die Traits spezifizieren ein mathematisches Modell für die Interface-Spezifikationen, das aber kein Konzept eines Zustands oder der Veränderung kennt. Traits sind Gleichungsspezifikationen. Aus ihnen läßt sich eine Theorie (die Menge der geltenden Eigenschaften) ableiten. Diese Theorien können in den Interface-Spezifikationen explizit verfügbar gemacht werden.

Auf Interface-Ebene gibt es ein Verknüpfungskonstrukt. Auf algebraischer Ebene können Spezifikationen lediglich durch einen Makromechanismus verknüpft werden.

1.6 Evaluierung und Zielsetzung

Die in Abschnitt 1.5 betrachteten Ansätze sollen nun im Kontext der Zielsetung dieser Arbeit bewertet werden. Die Ansätze EML, Larch, COLD und Π kommen den beschriebenen Anforderungen am nächsten. EML basiert auf einer realen Programmiersprache. Allerdings ist EML nicht zustandsbasiert und realisiert das Komponentenmodell nur teilweise. Larch bietet keine eigenschaftsorientierte, zustandsbasierte Spezifikationslogik. Π fehlen Möglichkeiten zur axiomatischen, zustandsbasierten Spezifikation. Abbildung 1.3 zeigt, daß im Bereich der zustandsbasierten Systeme COLD den hier geforderten Eigenschaften am nächsten kommt. Ein formales Berechnungsmodell, auf dem auch eine imperative Programmiersprache geeignet definiert werden kann, ist dort aber nicht vorhanden. Imperative Aspekte werden zwar realisiert, eine Anbindung an eine Programmiersprache existiert jedoch nicht. In den auf realen Programmiersprachen basierenden Ansätzen EML und Larch hat sich gezeigt, daß eine *konstruktive Modellbildung* für die Spezifikationsebene von Bedeutung ist. Damit ist eine Modellbildung der Spezifikationssprache basierend auf der Semantik der Programmiersprache gemeint. Abbildung 1.2 zeigt, daß hier diese Konstruktivität durch eine durchgängige Benutzung der Σ-Objekte erreicht wird. Obwohl Konzepte zur komponentenorientierten Spezifikation in COLD realisiert sind, werden Anforderungen des Komponentenmodell, wie z.B. semantische Beschreibungen in Exportschnittstellen, nicht bereitgestellt. In die Abbildung 1.3 wurde noch die Sprache *Eiffel* aufgenommen, die als Programmiersprache in Richtung auf eine Entwurfssprache erweitert wurde. Die dort vorhandenen Spezifikationstechniken *Vor- und Nachbedingungen* und *Invarianten* sind allerdings nicht durch eine Logik formalisiert. VDM und Z sind im Vergleich dazu zwar durch die Hoare-Logik definiert, eine eigenschaftsorientierte Spezifikation ist aber nicht möglich.

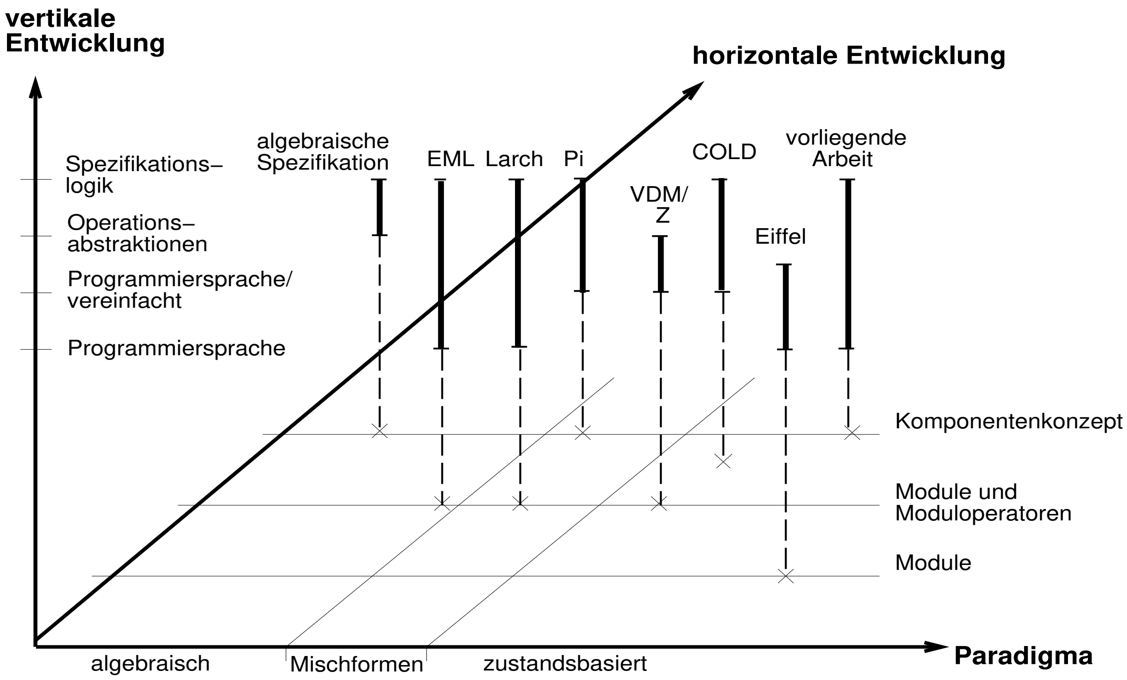


Abbildung 1.3: Vergleich von Spezifikationsansätzen

Die Breite der betrachteten Sprachkonzepte von abstrakten In dieser Arbeit soll ein Ansatz vorgestellt werden, der die Stärken der eben angesprochenen erhält, d.h. formal ist, zustandsbasiert ist (wie COLD) und das Komponentenmodell unterstützt (wie Π), und die angesprochenen Schwächen vermeidet. Diese Arbeit wird also einen zustandsbasierten Beschreibungsansatz vorstellen, der Spezifikation und Implementierung integriert. Dabei wird Wert auf Anwendbarkeit des Ansatzes zur Definition von Sprachen und deren Realisierung in einer konkreten Entwicklungsumgebung gelegt. Zur Unterstützung der Implementierung werden die gängigen Programmiersprachenkonstrukte betrachtet, die Spezifikation wird durch Konstrukte zur horizontalen und vertikalen Entwicklung unterstützt.

Spezifikationskonstrukten bis zu einer Vielzahl von imperativen Konstrukten, die alle in einem Komponentenkonzept eingesetzt werden können, ist im Rahmen umfangreicher, teamorientierter Software-Entwicklung notwendig, aber, wie Abbildung 1.3 zeigt, im Bereich zustandsbasierter Systeme nur ansatzweise von COLD erreicht. Diese Arbeit wird die notwendigen Grundlagen zur Definition eines Spezifikationsansatzes im definierten Kontext motivieren und formal definieren. Ziel ist eine umfassende Zusammenstellung von Grundlagen, die alle Bereiche der vertikalen Entwicklung abdeckt und Komponentenorientierung als Basis der horizontalen Entwicklung hat.

Eine wesentliche Aufgabe der Software-Technologie liegt in der Integration verschiedener Formalismen, die zur Bearbeitung verschiedener Aspekte in der Software-Entwicklung eingesetzt werden. Im Kontext zustandsbasierter Systeme müssen etwa mögliche Zustände und Zustandsübergänge, operationale Beschreibungen von Algorithmen oder abstrakte Eigenschaften von Objekten (etwa Sicherheits- oder Lebendigkeitseigenschaften, die die Vermeidung unerwünschter oder das Erreichen erwünschter Eigenschaften über längere Zustandsübergangsfolgen hinweg) ausdrückbar sein. Um das Gesamtverhalten eines Systems betrachten zu können, ist eine Integration auf zwei Ebenen notwendig (diese Forderung wird etwa auch von Fiadero und Maibaum [FM95] unterstützt). Zum einen ist auf sprachlicher Ebene eine Integration, etwa durch eine erweiterte Prädikatenlogik, zu erreichen. Dynamische Logiken (und ihre Erweiterungen) bieten die Möglichkeit zur deskriptiven Spezifikation, wobei die explizit werdenden Programmfragmente auch zur operationalen Spezifikation dienen können. Die zweite Ebene ist die Semantik. Fiadero und Maibaum schlagen Kategorientheorie vor. Wir werden hier die Integration durch Σ-Objekte erreichen. Der mathematische Rahmen besteht hier aus einer Anpassung üblicher algebraischer Techniken auf den zustandsbasierten Kontext.

Die Arbeit konzentriert sich auf zustandsbasierte Systeme. Diese sollen operational (durch eine Programmiersprache) und deskriptiv (durch eine Spezifikationssprache) beschrieben werden können. Bisher hat sich aber noch kein Semantikstandard herausgebildet, der zur Definition beider Beschreibungsformen geeignet ist. Imperative und objektorientierte Programmiersprachen sind aktuell die am weitesten verbreiteten Programmiersprachen. Formal gestützte Entwicklungsmethoden haben bisher aber kaum ihren Weg in die praktische Unterstützung dieser Sprachen gefunden.

Teil I

Berechnungsmodell

Kapitel 2

Ein Berechnungsmodell

2.1 Motivation

In diesem Kapitel und in den folgenden zwei Kapiteln wird ein *Berechnungsmodell*, d.h. eine semantische Struktur, entwickelt, auf dem im Kontext zustandsbasierter Systeme sowohl operationale (imperative oder objektbasierte) als auch axiomatische (auf modaler Logik basierende) Sprachen definiert werden können.

Die Modellierung zustandsbasierter Systeme erfordert adäquate Berechnungsmodelle. Turing-Maschinen sind nicht praktikabel genug, da sie nicht an verschiedene Abstraktionsebenen anpaßbar sind. Algebren sind z.B. nur bedingt geeignet, da sie keinen Zustandsbegriff kennen und dieser somit modelliert werden muß. Ein inhärent zustandsbasiertes, an verschiedene Abstraktionsebenen anpaßbares Berechnungsmodell soll hier vorgestellt werden. Dieses Modell soll geeignet sein, Aspekte der *Programmsemantiken*, d.h. der Definition von Programmiersprachen, und der *Programmlogiken*, d.h. der Anwendung von Logik zur Spezifikation von Programmen, zu vereinen. Die Unterstützung zustandsbasierter Ansätze ist insbesondere dann von Bedeutung, wenn Algorithmen im von-Neumann'schen oder Turing'schen Sinne als Beschreibungen zustandsmodifizierender Funktionen verstanden werden. Das im folgenden vorgestellte Berechnungsmodell ist die formale Umsetzung der Idee einer abstrakten Maschine.

Warum Berechnungsmodelle?

Der Begriff des Datentypen ist fundamental in der Informatik. Die Definition von Datentypen erfolgt auf zwei Ebenen. Auf der ersten — *syntaktischen* — Ebene werden durch *Signaturen* Namen für Datenmengen sowie Namen, Argument- und Wertebereiche für Operationen angegeben. Auf der zweiten — *semantischen* — Ebene werden Mengen und Funktionen angegeben, die zur Signatur korrespondieren. In der Logik wird für diese Kollektionen aus Mengen und Funktionen der Begriff der *(semantischen) Struktur* verwendet. Da es in diesem Ansatz um zustandsbasierte Systeme geht, also um Systeme mit einem Veränderungsbegriff, der sich aus der Ausführung von Operationen (der *Berechnung* des Effektes von Operationen) ergibt, soll hier der Begriff *Berechnungsmodell* dem der Struktur vorgezogen werden, da er den operationalen Charakter stärker betont, aber im Gegensatz zur *abstrakten Maschine* auch den formalen Aspekt umfaßt. Der Begriff 'Modell' darf hier nicht mit dem Modellbegriff z.B. algebraischer Spezifikationen verwechselt werden, wo unter einem Modell eine Algebra verstanden wird, die spezifizierte Eigenschaften erfüllt.

Das Berechnungsmodell dient hier zweierlei Zwecken:

1. als *semantische Struktur* zur Definition eines Gültigkeitsbegriffs für Formeln einer *Spezifikationslogik*,

2. als *operationales Berechnungsmodell* zur Definition imperativer und damit auch objektbasierter *Programmbeschreibungen*.

Berechnungsmodelle existieren für den Bereich der üblichen Logiken erster Stufe, so z.B. Algebren für die Gleichungslogik algebraischer Spezifikationen [Wir90]. Für den Bereich zustandsbasierter Systeme gibt es die aus der modalen Logik bekannten Kripke-Modelle [KT90], die einen Zustandsbegriff umfassen. Aus dem Bereich der von-Neumann-Rechnerarchitekturen und der imperativen Programmiersprachen stammt das Berechnungsmodell der Turing-Maschinen. Sowohl Kripke-Modelle als auch Turing-Maschinen scheinen jeweils nicht optimal geeignet, die beiden hier anstehenden Benutzungszwecke vollständig zu unterstützen. Es sind zusätzlich Mechanismen erforderlich, die im Fall der Turingmaschinen die Anpaßbarkeit an verschiedene Abstraktionsniveaus ermöglichen, und die im Fall der Kripke-Modelle praxisrelevante Sprachen geeignet operational definierbar machen. Daher wird hier das Berechnungsmodell der **Σ-Objekte** entwickelt, das die Stärken beider Modelle realisiert, d.h. die Operationalität der Turing-Maschine und die Formalität und Abstraktheit der Kripke-Modelle, und deren Schwächen vermeidet. Σ bezeichnet das gegebene Vokabular, also die Signatur. Der Begriff 'Objekt' wurde gewählt, um die Zerlegung eines Systems in mehrere, unabhängige Teilsysteme – die Objekte – anzudeuten.

In der Logik erster Stufe werden nichtleere Mengen mit Operationen und Relationen als *Strukturen* bezeichnet. Diese dienen dazu, den Termen und Formeln der Logik (ggf. über einem gegebenen Vokabular, d.h. einer Signatur) eine Semantik zu geben. In der universellen Algebra werden Strukturen ohne Relationen auch als *Algebren* bezeichnet. Diese haben sich als geeignet erwiesen, funktional orientierten Spezifikationen einer Gleichungslogik als Semantik zu dienen. Der Ansatz der Algebren soll hier übernommen werden. Eine Anpassung zum einen an zustandsorientierte Anforderungen und zum anderen an programmiersprachliche Anforderungen hat zu erfolgen. Das Berechnungsmodell, das in diesem Kapitel vorgestellt wird, läßt sich durch die Idee einer *abstrakten Maschine* veranschaulichen. Der Umfang der Funktionalität der Maschine wird durch die Kombination verschiedener *Semantikpakete* bestimmt. Es werden eine Basismaschine und eine Reihe von Erweiterungspaketen vorgestellt. In den Erweiterungspaketen werden die üblichen Konzepte von Programmiersprachen (siehe etwa [Hor84]) zusammengefaßt. In exogenen modalen Logiken, zu denen auch die Programmlogiken gehören, werden auch Programmkonstrukte explizit. Somit sind auch für die Logikdefinition (eine solche Logik wird in Kapitel 5 vorgestellt) die Sprachkonstrukte der Σ-Objekte von Bedeutung.

Verschiedene Ansätze im Kontext des Entwurfs von Software [Mey88, Goe93] verstehen unter dem Begriff des Objektes das zentrale, strukturgebende Konzept zur Konstruktion von Software. Objekte sind ebenfalls das strukturgebende Konzept in diesem Ansatz. Die in einem zu modellierenden System identifizierten Objekte werden durch den Begriff der *Komponente* auf das zu entwickelnde Software-System abgebildet. Die Abbildung wird durch ein *Komponentenkonzept* [Goe93, GSC91, CFGGR91] unterstützt. Komponenten sind ein Konzept zur Unterstützung von Wiederverwendung und verteilter Entwicklung innerhalb der Entwicklung von Software. Dies wird durch die Bereitstellung semantisch beschreibbarer Schnittstellen und der Verkapselung der Interna erreicht. Diese Aspekte werden in den Teilen II und III dieser Arbeit betrachtet.

Mit diesem Ansatz sollen Sprachen entwickelbar und definierbar sein, die den Übergang vom Entwurf zur Programmierung mit einem durchgängigen Verständnis des Objektbegriffs ermöglichen. Für den Entwurf werden Spezifikationssprachen und für die Programmierung werden Implementierungssprachen benötigt, die zu Objektbeschreibungssprachen zusammengefaßt werden können. Das beiden Sprachklassen zugrundeliegende Berechnungsmodell wird in den folgenden zwei Kapiteln vorgestellt. Zuvor werden noch vergleichbare Ansätze betrachtet.

2.2 Vergleichbare Semantikansätze

Unter der Semantik eines Programmes versteht man die Beschreibung des Verhaltens, das das Programm produziert, wenn es durch einen Computer ausgeführt wird. Es haben sich zur formalen Definition der Semantik einer Programmiersprache drei klassische Richtungen entwickelt:

- *denotationale Semantik* [Mos90, Sto77, LS84, AS88],
- *operationelle Semantik* [Ast91, Ten91, AS88],
- *axiomatische Semantik* [BWP87, Cou90].

Vergleichende Beschreibungen sind in [Sto77], [AS88], [Ten91] und [Wat91] zu finden. Im Anschluß an diese drei Formen werden *Action Semantics, Evolving Algebras, algebraische Spezifikationen* sowie *Programmlogiken* und *Kripke-Modelle* betrachtet.

Die wesentlichen Eigenschaften der Ansätze sollen kurz vorgestellt und die Eignung für die vorliegende Arbeit soll evaluiert werden.

2.2.1 Klassische Programmsemantiken

Die **denotationale Semantik** ordnet den Programmiersprachenkonstrukten mathematische Modelle zu. Modelle lassen sich aus gegebenen mathematischen Entitäten zusammensetzen. Jeder Phrase der Programmiersprache wird ein solches mathematisches Objekt zugewiesen. Die Phrase *denotiert* oder *bezeichnet* ein solches Objekt. Die Bedeutung oder Denotation einer Phrase ist immer auf den Bedeutungen ihrer Subphrasen konstruiert. Dieser Aspekt der Kompositionalität ist zentral im denotationalen Ansatz. Damit wird der Einsatz struktureller Induktion als Beweistechnik ermöglicht. Denotationale Semantiken stellen eine Basis zur Durchführung von Korrektheitsbeweisen dar, z.B. zur Definition von Beweisregeln für Korrektheitszusicherungen. Konsistenz und Vollständigkeit eines Kalküls können untersucht werden, indem die denotationale Semantik der Programmiersprache benutzt wird, um die Formeln der Logik zu interpretieren. In der denotationalen Semantik werden Operationen durch mathematische Funktionen definiert, die Eingabedaten auf Ausgabedaten abbilden, die also abstrakt das Ein-/Ausgabeverhalten beschreiben. Die Abstraktion über jeder Form von Implementierungsdetails ist eine weitere Kerneigenschaft der denotationalen Semantik.

Der *predicate transformers approach* ist ein Versuch, über dem zum Teil als zu operational empfundenen expliziten Zustandsbegriff in denotationaler Semantik zu abstrahieren (siehe

[Dijk76], [Gri81]). Die Denotation eines Kommandos ist hier eine Funktion, die eine geforderte Nachbedingung auf die schwächste Vorbedingung abbildet, so daß die Ausführung des Programmes in einem Zustand terminiert, der die Nachbedingung erfüllt. [Plo79] zeigt, daß dies zur denotationalen Semantik äquivalent ist.

Eine **operationelle Semantik** ist eine Semantik, in der repräsentationsabhängige Informationen über einer Basismaschine mit einem Zustand und einer Reihe von primitiven Instruktionen explizit sind. Für jedes Konstrukt werden Berechnungssequenzen angegeben, die die einzelnen Ausführungsschritte des Konstrukts auf der Basismaschine beschreiben. Operationelle Semantiken sind in der Regel näher an der intuitiven Semantik der Sprache, wie sie z.B. in einer informellen Sprachdefinition zu finden ist, als denotationale Semantiken. Durch das Darlegen der Berechnung, ggf. sogar über Zwischenzustände, ist die operationelle Semantik weniger abstrakt als die denotationale Semantik, die Programme als Funktionen von Eingabezuständen auf Ausgabezustände interpretiert. Die denotationale Semantik benutzt keine Basismaschine, sondern Mengen und Funktionen direkt zur Definition von Sprachelementen.

Die **axiomatische Semantik** ist eine weitere Möglichkeit, die Semantik eines Programmes zu beschreiben. Ein formales System aus Axiomen und Inferenzregeln wird dazu benutzt. Jedes Programmkonstrukt wird durch ein Axiom beschrieben. Haupteinsatz ist die Spezifikation, Transformation und Verifikation von Programmen. Ein bedeutender Vertreter dieser Art ist die Hoare-Logik. VDM ist als Spezifikationssprache unter anderem entstanden, um die Definition von Programmiersprachen mit denotationaler Semantik zu vereinfachen.

Es gibt außerdem noch eine algebraische Ausprägung der axiomatischen Semantik (siehe [BWP87, Wat91, GM87b]). Die Eignung algebraischer Spezifikationen, abstrakte Datentypen zu spezifizieren, wird benutzt, die essentiellen Eigenschaften eines Typs zu beschreiben, ohne daß irgendeine Repräsentation oder Implementierung einfließt.

2.2.2 Weitere Semantiken

Neben den klassischen Formen haben sich in den letzten Jahren weitere Formen der Programmsemantik entwickelt, von denen zwei hier kurz vorgestellt werden sollen:

- *Action Semantics* nach [Mos89] oder [Wat91] und
- *Evolving Algebras* nach [Gur93, Gur94].

Außerdem werden

- *Algebraische Spezifikation* nach [Wir90, EM85, EM90] und
- *Programmlogiken* nach [KT90]

angesprochen.

Action Semantics sind aus dem Versuch heraus entstanden, formale Semantikansätze für Programmiersprachen gegenüber den klassischen Ansätzen '*denotational*' oder '*operationell*' verständlicher zu gestalten und damit stärker in der Informatik zu verbreiten.

Die Semantik einer Sprache wird durch *Aktionen* (*actions*) ausgedrückt, die direkt die operationellen Konzepte der Sprachen reflektieren. Primitive Aktionen erlauben das Speichern von Werten, Binden von Werten an Bezeichner oder Testen von Wahrheitswerten. Primitive Aktionen können durch Sequenzen, Selektionen und Iterationen verknüpft werden. Die Sprachkonzepte zur Beschreibung sind an die natürliche (englische) Sprache angelehnt. Mit Action Semantics soll die Semantik einer Sprache auf zwei Ebenen spezifiziert werden. Auf der oberen Ebene wird die Semantik durch Aktionen leicht verständlich ausgedrückt. Die untere Ebene besteht aus den formalen Definitionen für die Aktionen selbst. Um verschiedenartige Sprachkonstrukte wie Ausdrücke, Kommandos oder Deklarationen beschreiben zu können, kann jede Aktion in drei Facetten strukturiert werden. In der funktionalen Facette werden Daten durch Aktionen berechnet, die direkt an andere Aktionen weitergegeben werden können. In der deklarativen Facette produzieren Aktionen Bindungen zwischen Bezeichnern und Daten. In der imperativen Facette können Aktionen Daten in Speicher schreiben. Primitive Aktionen sind einfach-facettiert. Durch Aktionskombinatoren können multi-facettierte Aktionen konstruiert werden. Mechanismen zur Behandlung prozeduraler Abstraktionen oder der Parameterübergabe stehen zur Verfügung. Zusammengesetzte Typen sind formulierbar. Ein einfaches Konzept zur Ausnahmebehandlung existiert.

Die formale Grundlage der Action Semantics bilden *Unified Algebras* [Mos89]. Diese erlauben gegenüber üblichen algebraischen Ansätzen zusätzlich Operationen auf Sorten. Dadurch werden parametrisierte Sorten, partielle Funktionen und Nichtdeterminismus modelliert. Der Begriff 'Unified Algebras' wurde gewählt, da kein Unterschied zwischen Sorten und Sortenelementen gemacht wird. Anspruch der Action Semantics ist, modularere Sprachsemantiken zu erlauben, als dies bei Benutzung von denotationaler oder operationeller Semantik der Fall ist. Action Semantics sind eine Kombination aus operationeller Schnittstelle und unterliegender denotationaler Definition.

Evolvierende Algebren — im weiteren kurz E-Algebren oder EA genannt — sind ein Berechnungsmodell, das zur Definition von Spezifikationssprachen herangezogen werden kann [Gur93, Gur94]. E-Algebren übernehmen den Grundgedanken der Turingmaschinen, daß jeder Algorithmus durch eine zustandsbasierte, abstrakte Ausführungsmaschine — die Turingmaschine — simuliert werden kann. Somit können Turingmaschinen dazu dienen, Algorithmen eine operationelle Semantik zu geben. Versuche, konkrete Algorithmen auf Turingmaschinen zu definieren, führen zu äußerst komplexen, umfangreichen Beschreibungen. E-Algebren behalten den Grundgedanken bei, daß Algorithmen Zustandsübergänge auf einer zustandsbasierten Maschine ausführen, sie sollen jedoch immer an ein den Algorithmen adäquates Abstraktionsniveau anpaßbar sein. Die Grundidee ist, jeden Zustand der Maschine durch eine Algebra zu beschreiben. Zustandsübergänge liefern neue, veränderte Algebren zurück. Um die Kompositionalität der Algorithmen zu simulieren (Algorithmen arbeiten auf den Algorithmen niedrigerer Abstraktionsebenen), müssen auch E-Algebren hierarchisierbar, d.h. zusammensetzbar sein, um die Forderung nach einer Anpaßbarkeit an verschiedene Abstraktionsebenen zu realisieren. In einfacher Form sind E-Algebren für sequentielle, deterministische Zustandsübergänge definiert; der Ansatz konnte aber auch auf verteilte, parallele Anwendungen ausgedehnt werden. In E-Algebren formulierte Sprachen sind durch einen EA-Interpreter ausführbar. Der EA-Ansatz scheint geeignet zur Behandlung temporaler oder dynamischer Logiken (siehe [Gur93] S.4). Dieser Aspekt ist etwa in [GRdL95] oder [Sch95] untersucht worden.

Weitere bedeutende Semantikansätze im Kontext dieser Arbeit sind *Algebren* und *Kripke-Modelle*. Beide sind Strukturen im Sinne der Logik. Sie dienen dazu, die Semantik von Spezifikationslogiken zu definieren. Sie werden im Rahmen von Vergleichen mit dem hier vorgestellten Berechnungsmodell später noch formal vorgestellt (siehe Abschnitt 5.4). Grundlagen der algebraischen Spezifikation finden sich im Anhang.

Algebraische Spezifikation [Wir90] stellt Mittel zur funktionalen Beschreibung von Datenabstraktionen bereit. Die Grundidee des algebraischen Ansatzes besteht in der Beschreibung von Datenstrukturen durch Benennung der verschiedenen Datenmengen, der Funktionen und ihrer Eigenschaften. Die Eigenschaften werden durch Formeln einer Prädikatenlogik beschrieben. Ein abstrakter Datentyp wird durch eine Klasse isomorpher Datenstrukturen, also mehrsortiger Algebren, definiert. Komplexe Datenstrukturen werden durch hierarchische Spezifikationen unterstützt. Parametrisierung dient ebenfalls der Strukturierung von Software-Systemen. Es existiert eine Vielzahl von Spezifikationssprachen, die z.B. schrittweise Verfeinerung oder horizontale und vertikale Entwicklung unterstützen (vgl. [Wir95]).

Programmlogiken [KT90] sind formale Systeme zur Beschreibung und Verifikation von Programmen. Programmlogiken sind gegenüber klassischen Logiken dynamisch und nicht statisch. In klassischer Prädikatenlogik wird der Wahrheitswert einer Formel durch eine Bewertung der freien Variablen über einer semantischen Struktur bestimmt. Die Bewertung wird als unveränderlich angesehen. In Programmlogiken gibt es explizite syntaktische Konstrukte, die Programme beschreiben, die den Wert von Variablen und damit auch den Wahrheitswert von Formeln ändern. *Dynamische Logik* ist eine solche (erweiterte) Programmlogik. **Kripke-Modelle** sind die semantischen Strukturen, in denen Programmlogiken und auch modale Logiken üblicherweise interpretiert werden. Zustände (insbesondere die von Programmen) werden in Kripke-Modellen durch eine Bewertung von Variablen in einer mehrsortigen semantischen Struktur mit Trägermengen, Funktionen und Relationen realisiert. Programme sind dann Transformationen auf Zuständen.

2.2.3 Bewertung

Für diese Arbeit ist es wichtig, über eine eindeutig beschriebene Bedeutung eines Programmes zu verfügen und nicht etwa nur über eine mögliche Implementierung unter mehreren. Da sich denotationale und operationelle Semantik in ihrer Ausdrucksmächtigkeit nicht unterscheiden, soll hier dem denotationalen Ansatz der Vorzug gegeben werden, obwohl noch zu sehen sein wird, daß der Begriff der Maschine, wenn auch in abstrakter, vollständig mathematisch beschriebener Form wieder auftaucht. Das übliche Problem operationeller Semantiken — die nicht präzise formale Definition der Maschine — wird dann gelöst. Das Wesen des hier vorgestellten Berechnungsmodells bleibt jedoch denotational. Die Bedeutung einer Prozedur, deren intuitive Semantik das Verändern des Zustandes einer abstrakten Basismaschine ist, wird hier kompositional durch eine Funktion definiert. Da hier auch die Definition einer Logik über semantischen Strukturen zu erfolgen hat, ist der axiomatische Ansatz ungeeignet. Hier soll eine konstruktive Modellbildung realisiert werden, d.h. der Modellbegriff einer axiomatischen Spezifikation ist über die Semantik der Programmiersprache definiert. Um die Modelle imperativer Spezifikationen auch als Modelle einer abstrakteren algebraischen Spezifikation betrachten zu können, sind Axiome zur Definition von Programmiersprachen hier ungeeignet. Programmiersprachen sollen hier denotational durch die Σ-Objekte definiert werden. Dieses Vorgehen wird im Kontext von Spezifikationsansätzen wie VDM oder Z auch *modelltheoretisch* genannt. Die zugeordneten mathematischen Objekte sind die Modelle des definierten Sprachkonstrukts.

Algebren fehlt der hier notwendige Zustandsbegriff. Darüberhinaus bietet der Ansatz der algebraischen Spezifikation aber einen formalen Rahmen, der als Grundlage des weiteren Vorgehens, d.h. der Erarbeitung einer semantischen Struktur, einer Spezifikationslogik und von

Konzepten zur Entwicklung komplexer Software-Systeme dienen soll. An Action Semantics ist die Aufteilung in eine handhabbare Schnittstellenschicht und eine formale Schicht interessant. Das Berechnungsmodell wird hier auch so gestaltet werden, daß die wesentlichen programmiersprachlichen Konstrukte in ihm reflektiert werden. Die Grundidee der E-Algebren scheint für den vorliegenden Ansatz geeignet, falls deren Definition formal wäre und sich die Vermutung bestätigt, daß die Behandlung dynamischer Logik möglich ist. Programmlogiken und Kripke-Modelle bieten den Zustandsbegriff, der in der algebraischen Spezifikation fehlt. Sie sind aber nicht zur Definition von Programmiersprachen konzipiert. Hierzu fehlt die Betrachtung verschiedener Programmiersprachenkonzepte.

In dieser Arbeit wird das Vorgehen von der Theorie der algebraischen Spezifikation bestimmt. Es wird in Kapitel 3 und 4 ein Berechnungsmodell entwickelt, das aus einer Benutzungsschicht und einer denotational definierten, darunterliegenden formalen Schicht besteht. Das Berechnungsmodell der **Σ-Objekte** wird in mehreren Schritten, die jeweils einzelne, in sich abgeschlossene Semantikpakete liefern, vorgestellt. Zuerst wird in Kapitel 3 ein Basismodell definiert. Auf diesem basierend werden dann in Kapitel 4 einige Erweiterungsmöglichkeiten vorgestellt. Am Ende von Kapitel 4 wird kurz aufgezeigt, wie sich das Modell zur Definition einer imperativen Sprache einsetzen läßt. Auf dem Berechnungsmodell wird in Kapitel 5 eine erweiterte Programmlogik, eine modale Prädikatenlogik, entwickelt.

Kapitel 3

Das Basismodell

Das Basismodell des Berechnungsmodells ist eine formale Definition einer abstrakten Maschine. Die Benutzungsschnittstelle der Maschine und deren Definition erfolgt in diesem Kapitel. Die bearbeitbaren Daten werden auf Daten primitiver Typen beschränkt. Die Instruktionen der Maschine (Auswertung von Ausdrücken, Ausführung von Anweisungen) entsprechen denen einer primitiven, imperativen Sprache. Da Σ-Objekte auch den Charakter einer semantischen Struktur haben, werden sie analog zu einem Vorgehen bei der Definition eines algebraischen Spezifikationsansatzes vorgestellt.

3.1 Basistypen und Wertebereiche

Für den folgenden Ansatz werden eine Reihe von Basistypen bereitgestellt. Es soll sich hierbei um primitive Datentypen einer üblichen Programmiersprache wie *Integer* oder *Bool* handeln. Daneben gibt es noch einen Typ *Void*, dessen Wertemenge nur aus dem Wert *null* besteht. Syntaktisch werden sie über die Sortenbezeichner *int*, *bool* und *void* angesprochen. Jedem Typ wird eine Halbordnung (siehe Anhang A.4) zugeordnet:

$$\begin{aligned} \mathit{Int}_\omega &= (INT_\omega, \sqsubseteq) \\ \mathit{Bool}_\omega &= (BOOL_\omega, \sqsubseteq) \\ \mathit{Void}_\omega &= (\{null\}_\omega, \sqsubseteq) \end{aligned}$$

INT ist die Menge der ganzen Zahlen $\mathcal{Z}$, $BOOL = \{true, false\}$. Alle Halbordnungen sind ω-Erweiterungen, d.h. der Menge der Elemente (z.B. INT) wird jeweils der undefinierte Wert ω hinzugefügt. Die Elemente der jeweiligen Mengen sind untereinander mit Ausnahme des undefinierten Elementes ω bzgl. $\sqsubseteq$ nicht geordnet. Solche Halbordnungen nennt man *flach*. ω ist das bottom-Element der jeweiligen Halbordnung. Das *Universum* aller Werte VAL soll die Vereinigung der Wertemengen aller Typen sein.

Um im Sinne von Datentypen benutzbar zu sein, müssen Operationen auf den Datenmengen vorhanden sein. Für Funktionen, die die angebotenen Operationen realisieren, wird in Kapitel 4 Stetigkeit gefordert werden (siehe Anhang A.4). Da flache Halbordnungen cpos sind, bilden alle vorgestellten Basistypen cpos. cpos sind in Definition A.4.4 im Anhang definiert. Stetigkeit und cpo-Eigenschaft sind Forderungen, die in Hinsicht auf eine denotationale Fixpunktsemantik von Bedeutung sind. Die Definition rekursiver Funktionen wird damit möglich. Dies wird im Detail in Kapitel 4 diskutiert.

Als Bezeichner für einen globalen *Namensraum* wird *ident* eingeführt. Die Halbordnung

$$Ident_\omega = (IDENT_\omega, \sqsubseteq)$$

auf der Menge der Bezeichner $IDENT$ sei ebenfalls flache cpo. $IDENT$ sei disjunkt zu allen anderen Wertemengen.

3.2 Syntaktische Konstrukte

3.2.1 Objektsignatur

Σ-Objekte haben den Charakter einer abstrakten Maschine. Zu deren Benutzung wird eine Schnittstelle bereitgestellt. Die *Benutzungsschnittstelle* der Σ-Objekte wird in Form von Signaturen und Termen dargestellt. Eine *Signatur* beschreibt die syntaktische Struktur eines Objekts.

Definition 3.2.1 *Eine* **Objektsignatur** *(oder kurz* **Signatur***)* Σ *besteht aus*

- *einer Menge von Datensorten* $S = \{s_1, \ldots, s_l\}$ *und einer Zustandssorte state mit* $state \notin S$,
- *einer Menge von Zustandsbezeichnern* $Z = \{z_i, | i = 1, \ldots, m\}$ *und jeweils zu einem* z_i *zugeordneter Datensorte* $s_i \in S$, $i = 1, .., m$, *wobei die* s_i *nicht verschieden sein müssen (man schreibt* $z_i : s_i$*),*
- *einer Menge von Operationen* OP *der Form*

  $$op \ : \ state \ \times \ s_1 \ \times \ldots \times s_n \rightarrow state \times s$$

  mit $op \in OP$ *und* $s_i, s \in S$ *für* $i = 0, .., n$.

Signaturen sollen syntaktisch anhand folgender Grammatik festgelegt werden:

signature	::=	*'sig' sig-id 'is'*
		*'sorts' [sort-id]**
		'state_def' typed_list
		'opns' op_sig_list
		'end sig'
typed_list	::=	*[state-id ':' sort-id]**
op_sig_list	::=	*[op-id ': state' ['×' sort-id]* '→ state ×' sort-id]**

Der Signaturbezeichner *sig-id* sowie die weiteren Bezeichner *sort-id* $\in S$, *state-id* $\in Z$ und *op-id* $\in OP$ sind Terminalzeichen und sollen durch Bezeichner interpretiert werden. *[..]** beschreibt die Wiederholbarkeit des Klammerinhalts in beliebiger Anzahl. Außerdem ist als Kurzform zur Beschreibung von Signaturen

$$\Sigma = < S, Z, OP >$$

zugelassen, wobei S die Sorten, Z die Zustandsbezeichner und OP die Operationen bezeichnet. Es seien außerdem folgende Projektionen für $\Sigma = < S, Z, OP >$ vereinbart:

$$\begin{array}{lcl} sorts(<S,Z,OP>) & := & S \\ state\_comp(<S,Z,OP>) & := & Z \\ opns(<S,Z,OP>) & := & OP \end{array}$$

Die Menge der Zustandsbezeichner kann leer sein. Der Sortenvorrat kann etwa aus den oben vorgestellten Sorten *int*, *bool*, *void* und *state* bestehen. Operationen, die den Zustand nicht verändern dürfen, heißen **Attribute**; für sie gilt

$$op(st, t_1, \ldots, t_n) = [st, t],$$

wobei $st : state$, $t_i : s_i (i = 1, .., n)$ und $t : s$ Terme sind. st bezeichnet den aktuellen Zustand, der unverändert bleibt. Attribute sind in der Menge $ATTR$ zusammengefaßt. Operationen, die den Zustand verändern dürfen, heißen **Prozeduren**. Sie haben im Gegensatz zu Attributen keinen Rückgabewert. Dies wird durch den Wert $null : void$ als zweite Komponente des Ergebnispaares modelliert. Für sie soll also

$$op(st, t_1, \ldots, t_n) = [st', null]$$

gelten. st' bezeichnet den Folgezustand. Die Menge der Prozeduren ist $PROC$. Es gilt:

$$OP \;=\; ATTR \;\cup\; PROC$$

Das Paar $(st, null)$ kann sowohl als Attribut als auch als Prozedur aufgefaßt werden. Dieses Paar wird mit *skip* bezeichnet und bildet die Schnittmenge von $ATTR$ und $PROC$.

$$ATTR \cap PROC = \{skip\}$$

skip wird im folgenden immer als Prozedur benutzt.

Zunächst sollen beispielhafte Signaturen, also Signaturen, die nicht vollständig in Hinblick auf praktische Benutzbarkeit sind, vorgestellt werden. Um die mit der Namensgebung intendierte Semantik sicherzustellen, sind Axiome erforderlich, die aber erst in einem späteren Kapitel eingeführt werden. Es sei also angenommen, daß für $BOOL$ Axiome $true \neq false$ oder $not(not(x)) = x$ für alle booleschen Werte x gelten. Für INT seien die Peano'schen Axiome und eine geeignete Erweiterung auf ganze Zahlen vorausgesetzt.

$$\begin{array}{l} sig\ BOOL\ is \\ \qquad sort\ \ bool \\ \qquad attr\ \ true : state \rightarrow state \times bool \\ \qquad attr\ \ false : state \rightarrow state \times bool \\ end\ sig \end{array}$$

$$\begin{array}{l} sig\ INT\ is \\ \qquad sort\ \ int \\ \qquad attr\ \ zero : state \rightarrow state \times int \\ \qquad attr\ \ succ : state \times int \rightarrow state \times int \\ \qquad attr\ \ neg : state \times int \rightarrow state \times int \\ end\ sig \end{array}$$

$$\begin{array}{l} sig\ VOID\ is \\ \qquad sort\ \ void \\ \qquad attr\ \ null : state \rightarrow state \times void \\ end\ sig \end{array}$$

3.2.2 Zustand

Unter dem Begriff der *Umgebung* wird generell die Bindung von Werten an Namen verstanden (siehe [LS84, ASU88]). Dieses Konzept soll hier unter dem Begriff *Zustand* realisiert werden. Dazu wird eine Abbildung $STATE$ definiert, die die Bindungen verwaltet. Ein Name wird von $STATE$ auf einen Wert abgebildet. Der Zustand ist der veränderliche Teil eines Objektes. In imperativen Programmiersprachen spricht man vom *Programmzustand*, womit die Belegung aller Programmvariablen mit Werten gemeint ist. Die Realisierung des Zustands hier soll kurz motiviert werden; sie wird in den anschließenden Abschnitten formal definiert.

Eine Signatur legt die Struktur eines Objektes fest. Vorerst sollen Signaturen noch nicht als Datentypspezifikationen angesehen werden. Zu einer Objektsignatur gibt es zwar eine Reihe möglicher Objekte, diese sollen aber noch nicht den Wertevorrat eines Datentypen bilden. Daher sei zunächst ein beliebiges, einzelnes Objekt angenommen. Es gibt eine Sorte *state* in jeder Signatur. Die Trägermenge der Sorte *state* im Objekt wird mit *State* bezeichnet. $STATE$ ist ein beliebiges Element aus *State*. $STATE$ repräsentiert den Programm- bzw. Objektzustand zu einem bestimmten Zeitpunkt. In $STATE$ werden die Bindungen verwaltet, d.h. $STATE$ bildet hier im Ansatz die Zustandsbezeichner auf ihre aktuellen Werte ab. Die Programmvariablen dieses Ansatzes sind die Zustandsbezeichner Z einer Signatur. $STATE$ soll immer als Abbildung interpretiert werden. Die Menge der Zustandsabbildungen *State* soll nur stetige Abbildungen enthalten. Daß eine Definition von *State* durch stetige Abbildungen $STATE$ möglich ist (wie es später für die Fixpunktsemantik notwendig ist), wird noch gezeigt (Lemma 4.1.4). $STATE$ bildet die Menge der Bezeichner für Programmvariablen $IDENT$ in die Menge aller Werte, genannt VAL, ab. $(VAL, \sqsubseteq_{VAL})$ soll immer eine cpo bilden. $IDENT$ ist eine Obermenge von Z, da $IDENT$ den Wertevorrat für die Zustandsbezeichner Z einer beliebigen Signatur bildet. In später erfolgenden Erweiterungen werden auch operationslokale Variablen und Parameter von $STATE$ verwaltet. Die cpo-Eigenschaft von $IDENT$ und VAL wird immer sichergestellt (siehe Lemmata 4.1.3 und 4.1.4 in Abschnitt 4.1.1).

Beispiel 3.2.1

sig STACK_SPEC is
    sorts   *list, elem, bool, void, int*
    state_def   *stack : list*
            nb_elements : int
    opns   *push : state × elem → state × void*
            pop : state → state × void
            top : state → state × elem
            is_empty : state → state × bool
end sig

stack ist in diesem Beispiel ein Zustandsbezeichner, *list* die ihm zugeordnete Datensorte. *stack* ist die Variable, die jeweils die aktuellen Stackelemente in einer Liste enthält. *nb_elements* ist die andere Zustandskomponente, in der die aktuelle Anzahl der Elemente eines Stacks gespeichert werden kann. Die Zustandssorte *state* ist eine Abstraktion über den Datensorten der einzelnen Zustandskomponenten, d.h. *state* ist die Sorte einer Abbildung $STATE$, in der die Bindungen von *stack* und *nb_elements* verwaltet werden. In Abbildung 3.1 ist ein Zustandsübergang dargestellt, der sich durch Ausführung des Kommandos *push(st,6)* ergibt. Durch *push* soll der Stack um das Element 6 erweitert und die Anzahl der Stackelemente

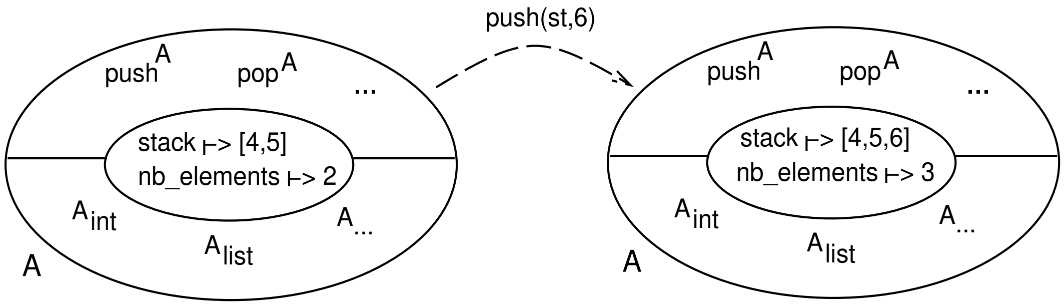


Abbildung 3.1: Zustandstransformation Stack

entsprechend um eins erhöht werden. In der Abbildung ist ein Σ-Objekt A in einem Zustand durch ein Oval beschrieben. In der oberen Hälfte sind die auf dem Objekt ausführbaren Funktionen ($push^A, pop^A$) zu finden, in der unteren Hälfte sind die Trägermengen ($A_{int}, A_{list}, ..$). Den Kern bildet der Zustand mit den Bindungen von Werten an Zustandsbezeichner.

3.2.3 Terme

Sei für diesen Abschnitt $\Sigma = < S \cup \{state\}, Z, OP = (ATTR \cup PROC) >$ eine Signatur. Terme werden benutzt, um semantische Entitäten zu bezeichnen. Die Zuordnung zwischen Termen und den semantischen Entitäten wird durch eine *Interpretation* definiert. Terme werden auf Basis von Namen mit Funktionsanwendung, Klammer usw. konstruiert. Terme werden jeweils einem Typ (oder einer Sorte) zugeordnet. Dieser allgemeine Termbegriff soll nun mit Signaturen in Beziehung gesetzt werden. Σ-Terme sind dann Terme, wobei jeder Sorten- und jeder Operationsname des Termes in der Signatur Σ enthalten ist. Eine Signatur Σ induziert dann eine Menge syntaktisch korrekter Ausdrücke, die aus freien Variablen und den Operationssymbolen der Signatur geformt werden können.

Sei $X = (X_s)_{s \in S}$ eine S-sortierte Menge von *freien Variablen* gebildet über dem Namensraum $IDENT$. Nur die Zustandsbezeichner und formalen Parameter von Operationen können frei in Σ-Termen auftreten. Die freien Variablen werden später durch eine Bewertung mit Werten belegt. Diese Bewertung wird durch die oben schon beschriebene Abbildung $STATE$ realisiert. $STATE$ wird bei Zustandsübergängen verändert. Mit dieser Konstruktion können Zustandsbezeichner die Eigenschaft *imperativer Variablen* haben, daß der an sie gebundene Wert verändert werden kann.

Zur Behandlung von Operationen werden noch folgende Definitionen eingefügt:

$$\overline{ATTR} := \{pr_2(a) | a \in ATTR\}$$

und

$$\overline{PROC} := \{pr_1(p) | p \in PROC\}$$

$a : state \times s_1 \times \ldots \times s_n \to state \times s$ induziert also $pr_2(a) : state \times s_1 \times \ldots \times s_n \to s$. Die Projektionssymbole pr_1 und pr_2 gewährleisten die Zuordnung von Attributen und Prozeduren zur jeweils maßgeblichen Sorte. $pr_2(a)(st, t_1, \ldots, t_n)$ ist also ein Term der Sorte s, falls die t_i Terme jeweils der Sorte s_i sind und $st : state$. Σ-Terme sollen jetzt formal definiert werden.

Definition 3.2.2 *Ein* **Σ-Term** *der Datensorte $s \in S$ über X_s ist*

- *jedes $x \in X_s$,*
- *jedes $\overline{a} \in \overline{ATTR}$, falls $a \in ATTR$ mit $a\ :\to state \times s$ (Konstante) und $\overline{a} = pr_2(a)$,*
- *jedes $pr_2(a)(st, t_1, \ldots, t_n)$, wobei $a \in ATTR\quad a\ :\ state\ \times\ s_1\ \times \ldots \times s_n \to state \times s$ und t_i Σ-Term der Datensorte $s_i \in S$ $(i = 1, \ldots, n)$, $\quad st : state$.*

Σ-Terme *der Zustandssorte state sind*

- *st als Bezeichner für die Zustandsabbildung,*
- *$pr_1(p)(st, t_1, \ldots, t_n)$, wobei $p \in PROC\quad p\ :\ state\ \times\ s_1\ \times \ldots \times s_n \to state \times s$ und t_i Σ-Term der Datensorte $s_i \in S$, $i = 1, \ldots, n$,*
- *$asgn(st, z, t)$, $seq(st, c_1, c_2)$, $if(st, t, c)$ und skip, falls die Sorte von des Σ-Terms t Datensorte ist, die Sorte von c, c_1, c_2 die Zustandssorte state ist, $st : state$ und z ein Zustandsbezeichner.*

$T(\Sigma, X)_s$ bezeichne alle Σ-Terme der Sorte $s \in S \cup \{state\}$. $T(\Sigma, X)_s$ ist die kleinste Menge, die alle wie oben konstruierbaren Σ-Terme der Sorte s enthält. $T(\Sigma)_s := T(\Sigma, \emptyset)_s$. Terme ohne freie Variable aus $T(\Sigma)_s$ heißen **Grundterme**. *Terme der Datensorten $s \in S$ heißen* **Ausdrücke**, *Terme der Zustandssorte state heißen* **Kommandos**.

Die ausgewählten, vordefinierten Kommandos für Σ-Objekte sind die primitiven Anweisungen einer *sequentiellen, deterministischen imperativen Programmiersprache* (vgl. [Hor84]). Schleifen werden hier nicht behandelt. Sie werden später nach Behandlung rekursiver Prozeduren definierbar. Die Kommandos sind im einzelnen:

- *Leeres Kommando*: *skip*. Es wird nichts ausgeführt. Da keine Zustandsänderung bewirkt wird, ist *skip* auch ein Attribut.
- *Zuweisung*: $asgn(z, expr)$. $expr$ ist ein Ausdruck der Sorte t, falls $z : t$; z darf in $expr$ enthalten sein.
- *Bedingtes Kommando*: $if(b, c)$. b ist ein Ausdruck (der Sorte *bool*), c ein Kommando. b ist die Vorbedingung der Ausführung des Kommandos. Das Kommando c legt den Effekt des bedingten Kommandos fest.
- *Prozeduraufruf*: $p(t_1, \ldots, t_n)$. p ist ein Prozedursymbol, die t_i sind Ausdrücke.
- *Sequenz*: $seq(c_1, c_2)$. Falls c_1 und c_2 Kommandos sind, dann ist auch $seq(c_1, c_2)$ ein Kommando. Es ist auch die Infix-Notation $c_1; c_2$ zulässig.

Σ-Terme sind also die Instruktionen der Maschine der Σ-Objekte.

Die obigen Termdefinitionen beschreiben Terme so, wie sie von der Signatur induziert werden. In allen nichtprimitiven Termen tritt der primitive Term *st* auf. *st* bezeichnet die jeweils aktuelle Zustandsabbildung. Die Semantik jedes Termes ist seine Interpretation durch eine abstrakte Maschine. Ein Term wird immer in einem Zustand der abstrakten Maschine interpretiert. Ein Zustand wird durch die freien Variablen bestimmt. Die freien Variablen werden in der Zustandsabbildung $STATE$ verwaltet. Da der aktuelle Zustand st somit immer vorhanden ist, kann der Zustandsterm st auch aus den Termen herausgelöst werden. Statt $a(st, t_1, \ldots, t_n)$ oder $asgn(st, z, t)$ heißt es dann $a(t_1, \ldots, t_n)$ oder $asgn(z, t)$. Diese Terme werden **implizite Σ-Terme** genannt, wohingegen Terme, die st enthalten, **explizite Σ-Terme** heißen. Falls es nicht zu Mehrdeutigkeiten kommt, werden wir im folgenden auch die implizite Form benutzen.

Die Zuordnung von Ausdrücken und Kommandos zu den Sorten $s \in S$ bzw. *state* erfolgt in Hinsicht auf die beabsichtigte Semantik der jeweiligen Terme. Beide Terme liefern semantisch zwar ein Wertepaar aus einem Zustandswert (Sorte *state*) und einem Datenwert (Sorte s), es ist jedoch immer einer der Werte irrelevant. Bei Ausdrücken ist nur der Datenwert, bei Kommandos nur der Zustandswert relevant. Kommandos modellieren Funktionen auf Zuständen. Sie geben den u.U. veränderten Zustand zurück, daher werden sie *state* zugeordnet. Die syntaktische Vereinheitlichung dient der Vorbereitung der noch vorzunehmenden Vereinheitlichung von Ausdrücken und Kommandos (dies ermöglicht dann Seiteneffekte in Ausdrücken, siehe Abschnitt 4.2.3).

Definition 3.2.3 *Eine Signatur heißt* **sinnvoll**, *wenn sie mindestens einen Grundterm zuläßt.*

Im folgenden sollen alle Signaturen sinnvoll sein.

Definition 3.2.4 $\Sigma' = < S', Z', OP' >$ *ist* **Subsignatur** *von* $\Sigma = < S, Z, OP >$, *falls* Σ' *sinnvoll,* $S \subseteq S'$, $Z' \subseteq Z$, $OP' \subseteq OP$ *und falls* $op \in OP' : state \times s_1 \times \ldots \times s_n \rightarrow state \times s_0$, *dann* $s_i \in S', \; i = 0, .., n$ *gilt.*

Definition 3.2.5 *Seien* $\Sigma = < S, Z, OP >$ *und* $\Sigma' = < S', Z', OP' >$ *zwei Signaturen. Dann sei definiert:*

$$\Sigma \cup \Sigma' := < S \cup S', Z \cup Z', OP \cup OP' >$$

sofern Σ *und* Σ' *ungleich einer undefinierten Signatur* $\perp_{Sign}$ *sind.* $'\cup'$ *sei strikt bezüglich* $\perp_{Sign}$. *Also ist*

$$\Sigma \cup \Sigma' := \perp_{Sign}, \; falls \; \Sigma = \perp_{Sign} \; oder \; \Sigma' = \perp_{Sign}.$$

3.2.4 Signaturmorphismen

Es sollen Abbildungen zwischen Signaturen, genannt Signaturmorphismen, definiert werden.

Definition 3.2.6 *Seien* $\Sigma_1 = < Z_1, S_1 \cup \{state\}, OP_1 >$ *und* $\Sigma_2 = < Z_2, S_2 \cup \{state\}, OP_2 >$ *Objektsignaturen. Ein* **Signaturmorphismus** $\sigma : \Sigma_1 \rightarrow \Sigma_2$ *ist ein Tripel von Abbildungen* $< \sigma_Z, \sigma_S, \sigma_{OP} >$ *mit* $\sigma_Z : Z_1 \rightarrow Z_2, \sigma_S : S_1 \rightarrow S_2, \sigma_{OP} : OP_1 \rightarrow OP_2$, *so daß stets gilt: ist* $op \in OP$ *mit* $op : state \times s_1 \ldots \times s_n \rightarrow state \times s$, *so ist* $\sigma_{OP}(op) : state \times \sigma_S(s_1) \times \ldots \times \sigma_S(s_n) \rightarrow state \times \sigma_S(s)$.

Bemerkung 3.2.1 *Es wurde die schwächste mögliche Form gewählt, d.h. ATTR und PROC werden nicht unterschieden, und der Zustandsbezeichner* $z_i \in Z$ *kann ohne Rücksicht auf Sortenkonformität* $(sorts(z_i) = sorts(\sigma_S(z_i)))$ *abgebildet werden.*

Ein Signaturmorphismus soll benutzt werden, um die Bereitstellung der Basistypen *int*, *bool* und *void* zu beschreiben. Diese Typen sind in Abschnitt 3.1 definiert. Da deren Signaturen disjunkt sind, können sie problemlos vereinigt werden.

$$\begin{aligned} BASETYPES = BOOL \cup INT \cup \{null\} \quad &= \\ < \emptyset, \{\ bool,\ int,\ void\ \} \cup \{\ state\ \}, \{\ true,\ false,\ null,\ zero,\ succ,\ neg\ \}\ > \end{aligned}$$

Diese Signatur *BASETYPES* kann nun automatisch in jede weitere benutzerdefinierte Signatur integriert werden:

sig MY_SIG is
    extend BASETYPES by
    ⋮
end sig

Die **Erweiterung** *extend* sei wie folgt definiert:

$$extend\ \Sigma_1\ by\ \Sigma_2 := \Sigma_1 \cup \Sigma_2 = < Z_1 \cup Z_2, S_1 \cup S_2 \cup \{state\}, OP_1 \cup OP_2 >$$

OP_1 und OP_2 müssen verträglich sein, d.h. für ein $op \in OP_1 \cap OP_2$ müssen die Signaturen von op in beiden Signaturen gleich sein. Sei $\Sigma_1 \subseteq \Sigma_2$ (d.h. Inklusion der Einzelkomponenten ist gegeben). Dann ist die **Einbettung** $in : \Sigma_1 \rightarrow \Sigma_2$

$$in(x) = x \text{ für } x \in Z_1 \cup S_1 \cup OP_1$$

ein Signaturmorphismus. Es wird im folgenden angenommen, daß die Basissignatur *BASETYPES* durch *in* in alle Objektsignaturen eingebettet ist.

3.3 Semantische Konstrukte

Die Idee des Berechnungsmodells und die grundlegenden Konstruktionen sollen in diesem Kapitel an einer Teilmenge des gesamten Berechnungsmodells verdeutlicht werden. Dieses Basismodell hat folgende Eigenschaften: Operationen sind nichtrekursiv und ihre Definitionen dürfen nicht ineinander verschachtelt werden. Sie dürfen auch keine lokalen Objekte haben. Die Parameter dürfen nur *Call-by-Value* übergeben werden. Ziel dieser Vereinfachung ist neben der Bereitstellung eines Kerns die Verdeutlichung der Arbeitsweise der Operationen auf dem Objektzustand. Operationen können lesend auf die Parameter und die Zustandskomponenten, Prozeduren können auch schreibend auf die Zustandskomponenten zugreifen.

Diese Arbeit lehnt sich an die in [LS84] verwendeten Notationen zur denotationalen Semantik und die dort vorgestellten Ergebnisse an. Die wichtigsten Definitionen und Sätze sind in Anhang A.4 aufgeführt. Basistypen, Wertebereiche und der Begriff der Objektsignatur wurden schon in den Abschnitten 3.1 und 3.2 vorgestellt.

3.3.1 Σ-Objekte

Jetzt sollen die Σ-Objekte formal definiert werden.

Definition 3.3.1 *Sei Σ eine Objektsignatur. Ein* **Σ-Objekt** $A = (S^A, OP^A, STATE)$ *besteht aus*

- *einer* $S' = S \cup \{state\}$*-sortierten Menge* $S^A = \{A_s | s \in S'\}$ *von Trägermengen (die Trägermenge* A_{state} *wird auch mit State bezeichnet, State sei definiert als Menge von Abbildungen* $STATE : IDENT \rightarrow (\bigcup_{s \in S} A_s \cup \{\omega\})$*),*
- *einer Menge* OP^A *von Abbildungen* $op^A \; : \; State \; \times \; A_{s_1} \; \times \ldots \times A_{s_n} \rightarrow State \times A_s$ *für jedes* $op \in OP$ *mit* $op \; : \; state \; \times \; s_1 \; \times \ldots \times s_n \rightarrow state \times s$*,*
- *einer dynamischen (d.h. veränderbaren) Abbildung* $STATE \in State$*, die den Wert liefert, der an einen Bezeichner gebunden ist.*

Zur Projektion auf eines der beiden Elemente eines Ergebnispaares einer Operation wird die Schreibweise $op(..)[1]$ bzw. $op(..)[2]$ benutzt.

Definition 3.3.2 *Zu jedem Σ-Objekt A ist ein* **Σ-Termobjekt** $T(A)$ *assoziiert:*

- *Für jede Sorte* $s \in S \setminus \{state\}$ *sei* $(T(A))_s := T(\Sigma)_s$ *und für* $s = state$ *sei* $(T(A))_{state} := \{st\}$*.*
- *Für jede Operation* $op \in OP$ *mit* $op : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$ *und* $t_i \in T(A)_{s_i}, i = 1, .., n$ *sei:*

$$op^{T(A)}(st, t_1, \ldots, t_n) := op(st, t_1, \ldots, t_n)$$

- *Es sei* $STATE^{T(A)} := st(x) \quad$ *für* $x \in IDENT$*.*

3.3.2 Zustände und Algebren

Die Konzeption der Σ-Objekte basiert auf der Idee, Algebren um einen Zustandsbegriff zu erweitern. Von den Σ-Objekten läßt sich jetzt auch der umgekehrte Weg gehen. Durch eine Projektion läßt sich aus einem Σ-Objekt eine Algebra gewinnen, die einen Zustand des Σ-Objekts beschreibt. Dieser Zustand ist anders gestaltet als der aus Kapitel 3.2.2. Die dortige Interpretation des Zustandsbegriffs soll im weiteren als *Kernzustand*, die Interpretation als Algebra durch den Begriff des *Gesamtzustands* bezeichnet werden.

Definition 3.3.3 *Sei A ein Σ-Objekt. Dann ist die* **Zustandsalgebra** A^{state} *wie folgt definiert:*

- *Trägermengen* A_s *aus A für alle Datensorten S von Σ,*
- *Funktionen* op^A *für alle* $op \in OP$*,*
- *eine statische Funktion STATE als Interpretation einer Operation* $st : ident \rightarrow val$*. Die Sorte* $ident \rightarrow val$ *sei mit state bezeichnet.*

Kommandos sind somit als Funktionen auf Zustandsalgebren, die die Abbildung $STATE$ verändern, realisierbar. Da man Zustandskomponenten auch als dynamische Konstanten ansehen kann, die sich von Zustand zu Zustand ändern, ließen sich Zustandskomponenten auch als Konstante realisieren. In einem späteren Abschnitt werden Σ-Objekte mit Algebren verglichen (siehe Abschnitt 5.4). Dort wird die Eignung der hier gewählten Alternative deutlich.

Definition 3.3.4 *Sei A ein Σ-Objekt. Dann ist die* **Menge der Zustandsalgebren** ZA^A *wie folgt definiert. Die Zustandsalgebra A^{state} ist in ZA^A, falls*

- *die Trägermengen A_s von A^{state} die Trägermengen von A sind,*
- *die Funktionen op^A von A^{state} die Funktionen von A sind,*
- *$STATE \in State$ eine (statische) Abbildung $STATE : IDENT \to VAL$ ist.*

Die Menge der Zustandsalgebren ZA^A eines Σ-Objekts A wird also gebildet, indem, basierend auf allen möglichen Zuständen $STATE$, Zustandsalgebren zu A gebildet werden.

Einen Zustandsbegriff in diesem Sinne interpretieren auch [FJ92, Wie91, Gur93]. Ähnlich ist hier auch der Ansatz MAL [FMR91]. Zustände eines Agenten sind dort die Zustände eines Kripke-Modells (vgl. Einleitung und Kapitel 5.4.2).

3.3.3 Homomorphismen

Ein Homomorphismus ist eine strukturerhaltende Abbildung. In der Anwendung auf Σ-Objekte werden die Elemente eines Σ-Objektes auf die eines anderen Σ-Objektes so abgebildet, daß die Operationen erhalten werden. Während Signaturmorphismen Abbildungen auf syntaktischer Ebene erlauben, ist durch Homomorphismen die Abbildung semantischer Strukturen möglich.

Sei im folgenden op^A die Interpretation der Operation op im Σ-Objekt A.

Definition 3.3.5 *Sei Σ eine Objektsignatur, A und B seien Σ-Objekte. $h : A \to B$ ist ein* **Σ-Homomorphismus**, *falls*

a.) *$h = \{h_s : A_s \to B_s | s \in S \cup \{state\}\}$ eine S-sortierte Menge von Abbildungen ist,*

b.) *h die Operationen erhält, falls also für $op : state \times s_1 \times \ldots \times s_n \to state \times s,\ a_i \in A_{s_i},\ st \in State$ gilt:*

$$[h_{state}(op^A(st, a_1, \ldots, a_n)[1]), h_s(op^A(st, a_1, \ldots, a_n)[2])\,] = op^B(h_{state}(st), h_{s_1}(a_1), \ldots, h_{s_n}(a_n)).$$

Bemerkung 3.3.1 *Bedingung* **b.)** *in Definition 3.3.5 kann abgeschwächt werden:*

- *$h_s(a^A(st, a_1, \ldots, a_n)[2]) = a^B(h_{state}(st), h_{s_1}(a_1), \ldots, h_{s_n}(a_n)\ )[2]$, falls a Attribut ist,*
- *$h_{state}(p^A(st, a_1, \ldots, a_n)[1]) = p^B(h_{state}(st), h_{s_1}(a_1), \ldots, h_{s_n}(a_n)\ )[1]$, falls p Prozedur.*

Die jeweils anderen Fälle brauchen nicht berücksichtigt werden, da sie schon von Attributen bzw. Prozeduren nach Definition erfüllt sind.

Definition 3.3.6 Isomorphismen *sind stets (d.h. für alle* $s \in S \cup \{state\}$*) bijektiv, d.h. alle* h_s *eines* Σ*-Homomorphismus* $h = \{h_s : A_s \to B_s | s \in S \cup \{state\}\}$ *sind bijektiv.*

Definition 3.3.7 *Sei* A' *ein* Σ'*-Objekt und* $\sigma : \Sigma \to \Sigma'$ *ein Signaturmorphismus. Dann wird* $A'|_\sigma$ *das* **σ-Redukt** *von* $A' = (S^{A'}, OP^{A'}, STATE)$ *genannt.* $A'|_\sigma$ *ist das* Σ*-Objekt definiert durch:*

- $(A'|_\sigma)_s := A'_{\sigma(s)}$ *für* $A'_{\sigma(s)} \in S^{A'}$,
- $f^{A'|_\sigma} := \sigma(f)^{A'}$ *für* $f \in opns(\Sigma)$,
- $STATE|_\sigma := STATE$.

Angewandt auf eine Klasse K *von* Σ*-Objekten bedeutet* $K|_\sigma$ *die elementweise Reduktbildung:*

$$K|_\sigma := \{ \ O|_\sigma \mid O \in K \ \}$$

Definition 3.3.8 *Mit* $A|_\Sigma$ *für ein* Σ'*-Objekt* A *wird die Reduktbildung bzgl. einer Subsignatur* $\Sigma \subseteq \Sigma'$ *bezeichnet.* Σ *wird mit dem Signaturmorphismus* $in : \Sigma \to \Sigma'$ *in* Σ' *eingebettet.*

3.4 Auswertung

Im folgenden soll eine dynamische Semantik der Instruktionen vorgestellt werden, die die korrekte Verwendung von Namen voraussetzt und die auf eine Abbildung $STATE$ zugreifen kann, in der alle veränderbaren Komponenten eines Σ-Objekts initial an den undefinierten Wert ω gebunden sind (also $STATE(x) = \omega$ für alle $x \in IDENT$). Der Zustand $STATE$ wird dann von Kommandos durch Substitution verändert.

Generell sind zwei Formen der *Substitution* möglich:

1. $T[x/a]$ oder T^a_x :⇔ im Term T wird jedes freie Vorkommen von x durch a ersetzt[1],
2. $substitute(STATE, x_i \mapsto v_i, 1 \leq i \leq n)$ substituiert in der Abbildung $STATE$ die bisherige Bindung für einen Namen x_i durch eine Bindung von x_i an den Wert v_i für $i = 1,..,n$. Das $1 \leq i \leq n$ wird in der Regel weggelassen. *substitute* liefert das erste Argument in veränderter Form zurück.

Die erste Form beschreibt syntaktische Ersetzung, die zweite beschreibt die Ersetzung in der semantischen Struktur $STATE$. Es soll nur die zweite Form eingesetzt werden, da sie die geeignete Form für die geplanten Parameterübergabemechanismen (vgl. Abschnitt 4.2.2) darstellt. Die erste Form wäre für einen *Call_by_Name*-Mechanismus geeignet.

Definition 3.4.1 *substitute ist wie folgt definiert:*

$$substitute(STATE, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n) := STATE'$$

[1] Diese Form der Substitution auf Termen wird später auch für andere syntaktische Konstruktionen, wie Formeln oder Spezifikationen, benutzt.

$$mit\ STATE'(z) = \begin{cases} STATE(z) & falls\ z \neq x_i,\ i = 1,..,n \\ y_i & falls\ z = x_i\ für\ ein\ i \in \{1..n\} \\ \perp & falls\ z = \perp \end{cases}$$

Die x_i müssen paarweise verschieden sein. Die x_i können in den v_j, $i, j = 1,..,n$ nicht vorkommen, da die Wertebereiche $IDENT$ und VAL disjunkt sind.

$$substitute(STATE, x_i \mapsto v_i, 1 \leq i \leq n)$$

kann auch anstatt

$$substitute(STATE, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$$

benutzt werden.

Definition 3.4.2 *Sei Σ eine Objektsignatur, SP eine Σ-Objektspezifikation, S die Menge der Datensorten, X eine S-sortierte Menge von freien Variablen und $STATE \in State$ ein Zustand. Eine* **Bewertung** *$v(STATE, x)$ von $x \in T(\Sigma, X)_s$ im Σ-Objekt O ist definiert durch*

$$v(STATE, x) = [STATE, STATE(x)]\ für\ x \in X_s\ freie\ Variable$$

mit $STATE(x) \in (A_s \cup \{\omega\})$.

$STATE$ ist als totale Funktion auf $IDENT$ definiert, also für alle auftretenden $x \in X$ definiert. Die Relation zwischen Σ-Termen und Σ-Objekten wird durch *Auswertungen* hergestellt. Auswertungen sind wohldefiniert, wenn das Σ-Objekt für alle verwendeten Sorten nichtleere Trägermengen hat.

Die Bewertung jedes Termes liefert ein Paar aus Zustandskomponente und Wertkomponente. Falls der Term t ein Ausdruck ist, bleibt die erste Komponente erhalten (der Zustand ändert sich nicht). Bei Kommandos wird ein spezieller Wert *'null'* als zweite Komponente zurückgegeben.

Definition 3.4.3 *Eine* **Auswertung** *$v^*(STATE, t)$ eines Σ-Termes $t \in T(\Sigma, X)_s$ im Σ-Objekt O, $s \in (S \cup \{state\})$ ist definiert durch:*

- $v^*(STATE, x) := v(STATE, x)$ *für* $x \in X$
  Die Auswertung von freien Variablen wird auf deren Bewertung zurückgeführt.

- $v^*(STATE, a(t_1, \ldots, t_n)) :=$
  $v^*(STATE, a)(STATE, v^*(STATE, t_1)[2], \ldots, v^*(STATE, t_n)[2])$ *mit* $a \in ATTR$
  wobei $v^*(STATE, a)$ die a im Σ-Objekt O zugeordnete Funktion ist.

  Die Auswertung eines Attributaufrufes liefert ein Wertepaar bestehend aus dem unveränderten Zustand $STATE$ in der ersten Komponente und dem Funktionswert des Attributes in der zweiten Komponente.

- $v^*(STATE, p(t_1, \ldots, t_n)) :=$
  $v^*(STATE, p)(STATE, v^*(STATE, t_1)[2], \ldots, v^*(STATE, t_n)[2])$ *mit* $p \in PROC$
  wobei $v^*(STATE, p)$ die p im Σ-Objekt O zugeordnete Funktion ist.

  Die Auswertung eines Prozeduraufrufes liefert ein Wertepaar, dessen erste Komponente der Folgezustand ist und dessen zweite Komponente auf 'null' gesetzt wird. Prozeduraufrufe sind Kommandos und haben daher keinen Wert im funktionalen Sinne.

- $v^*(STATE, skip) := [STATE, null]$

  skip ist ein Kommando, das keine Zustandsänderung bewirkt.

- $v^*(STATE, asgn(z_i, t)) :=$
  $$[substitute(STATE, z_i \mapsto v^*(STATE, t)[2]), null]$$

  Die Zuweisung ist ein einfaches Kommando, dessen Folgezustand sich durch Substitution der bisherigen Bindung an die linke Seite der Zuweisung durch den Wert der rechten Seite der Zuweisung ergibt. Als Kommando hat die Zuweisung keinen funktionalen Wert, die zweite Komponente ist also $'null'$.

- $v^*(STATE, if(b, c)) :=$

  ```
  if v*(STATE, b)[2] =_bool true then
        v*(STATE, c)
  else if v*(STATE, b)[2] =_bool ⊥_bool then
              [STATE, ω]
        else  [STATE, null]
        end
  end
  ```

  Das bedingte Kommando wird nur dann ausgeführt, wenn die Auswertung der Bedingung den Wert true *ergibt, sonst wird der Zustand unverändert zurückgegeben.* $'if'$ *ist strikt.*

- $v^*(STATE, seq(com1, com2)) :=$
  $$v^*(v^*(STATE, com1)[1], com2)$$

  Das zweite Kommando arbeitet auf dem vom ersten Kommando berechneten Folgezustand.

Mit dieser Definition sind nun die Instruktionen der abstrakten Maschine definiert. Es soll noch die Assoziativität des Sequenzoperators gezeigt werden. Damit sind dann Kommandofolgen getrennt durch *';'* möglich.

Bemerkung 3.4.1 *Der Sequenzoperator ist* assoziativ.

$$v^*(STATE, seq(seq(com1, com2), com3)) = v^*(STATE, seq(com1, seq(com2, com3)))$$

Beweis:

$$\begin{aligned}
&v^*(STATE, seq(seq(com1, com2), com3)) = \\
&\quad v^*(v^*(STATE, seq(com1, com2))[1], com3) = \\
&\quad v^*(v^*(v^*(STATE, com1)[1], com2)[1], com3) = \\
&\quad v^*(v^*(STATE, com1)[1], seq(com2, com3)) = \\
&\quad v^*(STATE, seq(com1, seq(com2, com3)))
\end{aligned}$$

Der Sequenzoperator ist also assoziativ. □

3.5 Operationsbeschreibungen

Jedem Operationssymbol kann eine **Operationsbeschreibung** zugeordnet werden. Diese Operationsbeschreibungen bilden spezielle Gleichungen, die etwa zur Definition von Operationen im Stil einer imperativen Sprache eingesetzt werden können.

Attributbeschreibungen haben die Form

$$a(x_1, \ldots, x_n) \ := \ expr,$$

wobei die x_i und die Zustandsbezeichner frei in $expr$ vorkommen, und $expr$ ein Term der Sorte s ist, falls $a \ : \ state \ \times \ s_1 \ \times \ldots \times s_n \rightarrow state \times s$.

Semantik der Attributbeschreibung: Sei O ein Σ-Objekt. Sei $a(x_1, \ldots, x_n) := expr$, also $a : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$. $v^*(STATE, a) = a^*$ wobei $a^*(STATE, y_1, \ldots, y_n) = v^*(STATE', expr)$ mit $STATE' = substitute(STATE, x_i \mapsto y_i)$ für alle $y_i \in A_{s_i}, i = 1, .., n$. x_i ist formaler Parameter von a, y_i ist der Wert des aktuellen Parameters. a hat als Semantik eine Funktion a^* auf den Trägermengen der Sorten $a^* : State \ \times \ A_{s_1} \times \ldots \times A_{s_n} \rightarrow State \times A_s$ mit $v^*(STATE, a(t_1, \ldots, t_n))[1] = STATE$.

Prozedurbeschreibungen haben die Form

$$p(x_1, \ldots, x_n) \ := \ com,$$

wobei die x_i und die Zustandsbezeichner frei in com vorkommen und com ein Term der Sorte $state$ (also ein Kommando) ist, falls $p \ : \ state \ \times \ s_1 \ \times \ldots \times s_n \rightarrow state \times s$.

Semantik der Prozedurbeschreibung: Sei O ein Σ-Objekt. Sei $p(x_1, \ldots, x_n) := com$, also $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$. $v^*(STATE, p) = p^*$ wobei $p^*(STATE, y_1, \ldots, y_n) = v^*(STATE', com)$ mit $STATE' = substitute(STATE, x_i \mapsto y_i)$ für alle $y_i \in A_{s_i}, i = 1, .., n$. x_i ist formaler Parameter von p, y_i ist der Wert des aktuellen Parameters. p hat als Semantik eine Operation p^* auf den Trägermengen der Sorten $p^* : State \ \times \ A_{s_1} \times \ldots \times A_{s_n} \rightarrow State \times A_s$ mit $v^*(STATE, p(t_1, \ldots, t_n))[2] = null$.

Nachdem nun Signaturen und Beschreibungen für Operationssymbole dieser Signaturen vorgestellt wurden, werden beide Elemente in einem Spezifikationsbegriff zusammengefaßt. Diese Spezifikationen sind modelltheoretisch im Sinne von Z oder VDM.

Definition 3.5.1 *Eine* **operationale Objektspezifikation** $SP \ = <\Sigma, E>$ *ist eine Objektsignatur* Σ *erweitert um eine Menge von Operationsbeschreibungen* E*.*

3.6 Zusammenfassung

In diesem Kapitel wurde ein Berechnungsmodell — die Σ-Objekte — vorgestellt (siehe Abbildung 3.2), das eine operationale Benutzungsschnittstelle zur Auswertung von Ausdrücken und zur Ausführung von Kommandos bereitstellt. Alle Schnittstellenfunktionalitäten sind formal durch mathematische Objekte definiert. Die Schnittstelle unterscheidet Ausdrücke (realisiert durch Attribute ATTR) und Kommandos (realisiert durch Prozeduren PROC und primitive Kommandos $asgn$, if, seq, $skip$). *Ausdrücke* werden wie üblich gebildet, d.h. aus Variablen und Attributaufrufen. Ausdrücke greifen nur lesend auf den Zustand zu. *Kommandos* werden in den Formen Zuweisung, bedingtes Kommando, Kommandosequenz, Prozeduraufruf und

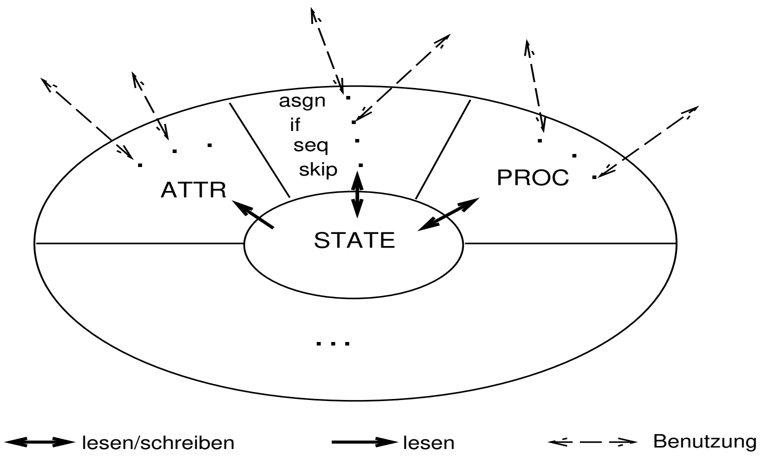


Abbildung 3.2: Abstrakte Maschine mit Instruktionen

leeres Kommando bereitgestellt. Kommandos greifen lesend und schreibend auf den Zustand zu. Die Auswahl der Kommandos orientierte sich an den üblichen primitiven Kommandos sequentieller, deterministischer imperativer Sprachen (vgl. [Hor84]). Σ-Objekte sind das formale Konstrukt, um die dynamische Semantik imperativer Programme bzw. von Spezifikationen imperativer Programme beschreiben zu können.

Σ-Objekte können als abstrakte Maschinen betrachtet werden, auf denen Programme ausgeführt werden können. Programme imperativer Sprachen sind Instruktionen, die die Belegung von Programmvariablen verändern.

Eine aktuelle Belegung aller Programmvariablen wird auch als *Programmzustand* bezeichnet ($STATE$). Um die Veränderung von Programmzuständen, d.h. von Belegungen zu realisieren, wurde hier die Sorte *state* eingeführt. Interpretation eines Elementes der Sorte *state* ist eine Belegung, realisiert durch eine Abbildung $STATE : IDENT \rightarrow VAL$, die Werte an die Bezeichner der Programmvariablen bindet. Diese Modellierung des Programmzustandes durch Belegungen wurde als *Kernzustand* bezeichnet. Ein Programmzustand kann auch als eine Momentaufnahme der gesamten abstrakten Maschine modelliert werden. Dieses wurde als *Gesamtzustand* bezeichnet. Eine solche Momentaufnahme eines Σ-Objekts besteht aus den Belegungen der Variablen (durch die aktuelle Abbildung $STATE$ realisiert), den Wertevorräten der Sorten sowie den Funktionen, die die Attribute und Prozeduren realisieren. Eine solche Struktur bestehend aus Wertemengen und Funktionen ist eine Algebra. Eine Zustandsalgebra wurde definiert.

In der Literatur wird die Modellierung von Zuständen in dieser Weise als *'states_as_algebras'*-Ansatz bezeichnet. Mit diesem Ansatz lassen sich dann auch veränderbare Operationsdefinitionen realisieren. Evolvierende Algebren [Gur93, Gur94] basieren auf dem Konzept dynamischer Operationen, d.h. veränderbarer Operationsdefinitionen. Zustände sind dort Algebren. Darauf sind Kommandos (*updates*) definiert, die veränderte Algebren produzieren. Ähnlich wird in CMSL [Wie91] verfahren. Sogenannte dynamische Objekte werden dort auf Kripke-Modellen realisiert. Veränderungen von Objekten werden durch Ereignisse modelliert. Eine spezielle Sorte *event* wird durch Funktionen auf Kripke-Modellen interpretiert. In Kripke-Modellen sind Zustände (wie es auch hier realisiert ist) endliche Varianten einer initialen

Belegung. Kommandos ändern Belegungen durch Substitution. Kommandos in Form von Prozeduren sind selbst aber nicht in der Struktur realisiert. Ein Vergleich mit Kripke-Modellen erfolgt in Abschnitt 5.4.2.

Die vorliegende Realisierung des Programmzustandes durch den Kernzustand müßte auf die Konstruktion über den Gesamtzustand angepaßt werden, wenn dynamische Operationen betrachtet werden sollen.

Kapitel 4

Erweiterungen des Modells

Als Erweiterungen des Basismodells des Berechnungsmodells der Σ-Objekte werden in diesem Kapitel *prozedurale Abstraktionen* und *Datenabstraktionen* behandelt:

- Operationen sollen nun auch rekursiv definiert werden können. Außerdem wird deren verschachtelte Definition betrachtet (Abschnitt 4.1). In Abschnitt 4.2 werden aufbauend auf 4.1 noch einige besondere Konzepte im Kontext von Operationsvereinbarungen untersucht (lokale Objekte in Prozeduren, verschiedene Parameterübergabemechanismen, Vereinheitlichung von Prozeduren und Attributen).

- Abschnitt 4.3 widmet sich dem Typsystem der Σ-Objekte. Bisher wurden nur primitive Typen wie *Integer* oder *Boolean* betrachtet. Jetzt soll eine Erweiterung des Typsystems um Typkonstruktoren und objektwertige Typen vorgeschlagen werden. Zum Abschluß werden polymorphe Erweiterungen betrachtet.

Abschnitt 4.4 zeigt, wie auf dem bis dahin vorgestellten Berechnungsmodell der Σ-Objekte eine imperative Programmiersprache definiert werden kann.

Das Basismodell ist eine Erweiterung von Algebren um einen Zustandsbegriff. Einige Primitive zur Veränderung des Zustands wurden definiert. Zielsetzung dieses Ansatzes ist es aber auch, die Definition gängiger imperativer Sprachen durch das Berechnungsmodell möglichst leicht zu machen. Dazu müssen die gängigen Konstrukte dieser Sprachen betrachtet werden. Eine übliche Programmiersprache umfaßt eine Vielzahl von Konzepten, die über die primitiven Typen und Kommandos des Basismodells hinausgehen (siehe z.B. [Hor84, Set89]). Programmvariablen und Programmzustand, Ausdrücke sowie die grundlegenden Anweisungen (wie Zuweisung, bedingte Anweisung, iterative Anweisung, Anweisungssequenzen und Sprünge) sind hier im wesentlichen in Kapitel 3 behandelt worden. Auf die Betrachtung von *goto*-Anweisungen wurde hier aus methodischen Gründen verzichtet. Iterative Anweisungen werden durch Rekursion möglich, die in diesem Kapitel eingeführt wird. In der Regel werden elementare, strukturierte und Referenzdatentypen unterschieden. Die elementaren Typen *Integer* oder *Boolean* sind schon betrachtet worden. Als strukturierter Typ wird *Menge* herausgegriffen. Ein Referenzkonzept wird zur Realisierung objektwertiger Zustandskomponenten vorgestellt. Blockstruktur, Sichtbarkeit, verschachtelte Definition von Prozeduren und Implementierung des Laufzeitverhaltens (Displays, Activation Records, ..) sind weitere Aspekte einer Programmiersprache. Weitere Prozedurkonzepte sind Parameterübergabe sowie Überladung und Generizität. Diese Aspekte sollen in diesem Kapitel angesprochen werden. Damit sind dann alle wichtigen programmiersprachlichen Konstrukte ([Hor84] Kapitel 4 bis 7, [Set89] Kapitel 2 bis 5) betrachtet.

Eine Anzahl gängiger Programmiersprachenkonzepte für Datenabstraktion und prozedurale Abstraktion wird nun formal definiert, um eine Semantikdefinition einer Programmiersprache (Programmsemantik) besser zu unterstützen. Wie in Kapitel 5 zu sehen sein wird, können diese Erweiterungen auch in einer Spezifikationslogik für Programme (Programmlogik) eingesetzt werden.

4.1 Rekursion und Verschachtelung

In diesem Abschnitt sollen Rekursion und verschachtelte Operationsdefinitionen das Berechnungsmodell aus Kapitel 3 erweitern. Dazu wird zuerst in Abschnitt 4.1.1 die Grundlage durch Definition eines Laufzeitstacks durch Fixpunktsemantik gelegt. Dann wird Rekursivität (4.1.2) und anschließend Verschachtelung (4.1.3) integriert.

4.1.1 Grundlagen

Die Definition blockstrukturierter Sprachen mit verschachtelbaren Operationen und lokalen Komponenten in Operationen erfordert eine Erweiterung des Zustandskonzeptes. Zur Lösung der Probleme soll auf Techniken des Compilerbaus zurückgegriffen werden. Grundlage des Ansatzes bildet ein Laufzeitstack, der innerhalb des Ansatzes geeignet formalisiert werden soll. Dabei sollen nur die Laufzeiteigenschaften erfaßt werden. Speicherorganisation zur Laufzeit muß nicht betrachtet werden. Es soll hier die dynamische Semantik, also die Semantik der Ausführung von Programmen formal definiert werden. Aspekte des Compilerbaus, wie die syntaktische oder semantische Analyse, die u.a. die Typisierung der Programmobjekte prüft, oder die Laufzeitspeicherorganisation, brauchen deshalb hier nicht berücksichtigt werden. Da der hier präsentierte Formalismus aber auf den Techniken des Compilerbaus basiert, soll die entsprechende Terminologie benutzt werden. Die Semantik des erweiterten Berechnungsmodells basiert also auf formalisierten Compilerbautechniken. Die Techniken des Compilerbaus zur Realisierung der Laufzeitumgebung sind [ASU88] Kapitel 7 entnommen. Die mathematischen Grundlagen sind in Anhang A aufgeführt.

Der Stack

Die Eigenschaften, die in diesem Abschnitt semantisch formal definiert werden sollen, sind *verschachtelte Prozeduren, static scoping, most closely nested rule* sowie *rekursiver Aufruf von Prozeduren*. Die notwendige Basis, solche Eigenschaften modellieren zu können, ist ein Stack. Zur Realisierung der Laufzeitmaschine werden Stacks eingesetzt. Jede Prozedur bildet einen Block. Die Bindungen lokaler Namen eines Block an Werte werden auf dem Stack in Form von Umgebungen, hier durch die Abbildung $STATE$ realisiert, verwaltet. Die notwendigen Funktionen auf dem Stack *push, pop, top* und *Zugriff* (lesend/schreibend) auf lokale und nichtlokale Objekte sollen semantisch formal definiert werden. Geeignete Vorarbeiten durch den Compiler, etwa die Ermittlung von Informationen bzgl. Lokalität von Namen (siehe [ASU88]), seien hier vorausgesetzt.

Ein Laufzeitstack ist notwendig, um in einer Sprache mit *static scoping* und verschachtelten Prozeduren die Verwaltung von *Umgebungen*[1] für die Operationsaufrufe zu realisieren. Das *Tupel* wird die Datenstruktur sein, die den Laufzeitstack formal realisiert. Es soll keine Begrenzung für den Stack vorgegeben werden; ein kartesisches Produkt der Elementzahl n für ein festes n reicht also nicht. Hier wird statt dessen die *freie Halbgruppe* über dem Domain der Stackelemente zugrundegelegt. Um allgemeingültige Ergebnisse zu erhalten, wird zunächst der Domain der Elemente nicht weiter bestimmt. Lediglich die cpo-Eigenschaft wird für ihn vorausgesetzt.

Sei D eine Menge (ein Alphabet). D^+ ist die Menge aller Wörter über D. Die Konkatenation zweier Wörter $w_1 \circ w_2$ ist definiert durch das Hintereinanderschreiben beider Wörter. Dann ist $(D^+, \circ)$ die *freie Halbgruppe über D*. $\circ$ ist assoziativ. $\epsilon \notin D^+$ (der leere Stack wird nicht gebraucht).

Die Menge der möglichen Inhalte pro Stackelement (etwa alle Umgebungen, d.h. Bindungen eines Blockes) bildet das Alphabet. Wörter sind dann mögliche Stacks (ein Wort ist ein n-Tupel über dem Alphabet). Zu zeigen ist, daß die freie Halbgruppe (ebenso wie feste, endliche kartesische Produkte) eine cpo bildet. Dies ist Voraussetzung, um Stetigkeit von Funktionen auf Stacks zeigen und in einem weiteren Schritt die Fixpunktsemantik zur Definition der Rekursion anwenden zu können.

Seien S_1 und S_2 zwei Teilmengen von kartesischen Produkten D^n und D^m. Dann ist $S_1 \circ S_2$ definiert durch $\{(w_1 \circ w_2) | w_1 \in S_1, w_2 \in S_2\}$. Sei $w_1 = (a_1, \ldots, a_n)$ und $w_2 = (b_1, \ldots, b_m)$. Dann ist $w_1 \circ w_2 = (a_1, \ldots, a_n, b_1, \ldots, b_m)$, also ist $w_1 \circ w_2 \in S_1 \times S_2$.

Folgerung 4.1.1 *(Existenz und Wert des Supremums) Sei $(D, \sqsubseteq)$ eine Halbordnung, S_1, S_2 Teilmengen von Wörtern über dem Alphabet D der Länge n bzw. m. Dann existiert $\sqcup(S_1 \circ S_2)$ genau dann, wenn $\sqcup S_1$ und $\sqcup S_2$ existieren, und es ist*

$$\sqcup(S_1 \circ S_2) = (\sqcup S_1) \circ (\sqcup S_2),$$

falls $\sqcup S_1 = (\sqcup pr_1(S_1), \ldots, \sqcup pr_n(S_1))$ und $\sqcup S_2 = (\sqcup pr_1(S_2), \ldots, \sqcup pr_m(S_2))$.

Beweis: Folgt direkt aus den vorangegangenen Überlegungen und Satz A.4.1. □

Der nächste Schritt ist der Nachweis der cpo-Eigenschaft für die freie Halbgruppe $(D^+, \circ)$, also die Existenz des bottom-Elementes und die Existenz des Supremums für jede Kette S in der Menge aller Wörter über D.

Folgerung 4.1.2 *Sei $(D, \sqsubseteq)$ cpo. Dann ist auch $(D^n, \sqsubseteq)$ für jedes $n \in Nat$ cpo.*

Beweis: Beweis folgt direkt aus Satz A.4.3 (Anhang). □

Lemma 4.1.1 *Falls $(D, \sqsubseteq)$ cpo ist, dann ist es auch $(D^+, \sqsubseteq^+)$ mit $D^+ = \bigcup_{i \in Nat} D^i, D^i = D \circ \ldots \circ D \quad (i - mal)$ und $a \sqsubseteq^+ b$ mit $a = a_1..a_n$ und $b = b_1..b_m$ gilt genau dann, wenn $n = m \wedge a_i \sqsubseteq b_i$ für alle i, also: zwei Wörter in D^+ sind geordnet, wenn sie gleich lang sind und in diesem Falle alle Positionen bzgl. $\sqsubseteq$ geordnet sind.*

[1] Es sind *Activation Records* zu verwalten. Da hier aber im wesentlichen Bindungen, also Umgebungen, verwaltet werden, soll auch der Umgebungsbegriff benutzt werden.

Beweis: Es soll gezeigt werden, daß für jede Kette in D^+ das Supremum existiert. Sei $\alpha_1 \sqsubseteq^+ \alpha_2 \sqsubseteq^+ \ldots$ eine Kette in D^+. Somit existiert ein k, so daß $\alpha_i \in D^k$ für alle i. Andernfalls $\alpha_{i_1} \in D^{k_1}$ und $\alpha_{i_2} \in D^{k_2}$ mit $k_1 \neq k_2$, und ohne Einschränkung $i_1 < i_2$. α_{i_1} und α_{i_2} sind dann unvergleichbar nach Definition der Ordnung $\sqsubseteq^+$. Zwei Elemente α_1 und α_2 aus D^+ sind nur dann vergleichbar, wenn sie der gleichen Länge und damit Element des gleichen D^k sind. Für D^k ist die cpo-Eigenschaft (genauer die Existenz des Supremums) oben schon gezeigt. Dies überträgt sich nun einfach auf D^+. $\square$

Also ist jetzt gezeigt, daß sich Tupel im Endlichen beliebig erweitern lassen, ohne daß die cpo-Eigenschaft verlorengeht. Im folgenden wollen wir nun noch die Stetigkeit von Funktionen betrachten. Mit diesen Funktionen wird die Grundlage für die Definition des Stacks gelegt.

Lemma 4.1.2 *'substitute' ist stetige Funktion auf* $(State, \sqsubseteq)$.

Beweis: Sei $sub_{x_i \mapsto y_i} := \lambda STATE.substitute(STATE, x_i \mapsto y_i)$ für $STATE \in State$, d.h. $sub_{x_i \mapsto y_i}(STATE) = substitute(STATE, x_i \mapsto y_i)$. Dann gilt:

$$substitute : state \times (ident \times val)^+ \to state \;\; ist\ stetig$$

gdw.

$$\forall (x_i \mapsto y_i) \in (IDENT \to VAL)^n.sub_{x_i \mapsto y_i} : state \to state \;\; ist\ stetig$$

Mit $(ident \times val)^+$ wird die mehrfache Wiederholbarkeit von $(ident \times val)$ bezeichnet.

Für $st \in State$ mit $st : ident \to val$ ist st strikt. Da $(IDENT, \sqsubseteq)$ und $(VAL, \sqsubseteq)$ flache cpos sind, folgt aus der Striktheit hier die Stetigkeit (Satz A.4.4) direkt. $\square$

Die bisherigen Ergebnisse dieses Abschnitts über die cpo-Eigenschaft von Mengen und die Stetigkeit von Funktionen sollen nun in Form von Datentypen veranschaulicht werden. Diese Datentypen haben keine formale Bedeutung. Tupel sind als freie Halbgruppe über deren Elementen realisiert. Die cpo-Eigenschaft wurde in Lemma 4.1.1 nachgewiesen. Nach den Überlegungen zu Folgerung 4.1.2 lassen sich stetige Operationen auf kartesischen Produkten fester Länge (Tupelkonstruktor, $\circ$, pr_i) auch hier verwenden.

TUPLE
make_tuple : t_elem → tuple
$\circ$*: tuple × tuple → tuple*
$(pr_i)_{i \in Nat}$*: tuple → t_elem*

tuple und *t_elem* sind Bezeichner für die Wertebereiche, auf denen die Funktionen arbeiten. $(pr_i)_{i \in Nat}$ soll eine Familie von Projektionsfunktionen bezeichnen. Die bisherigen Überlegungen zur Behandlung des Zustands sollen nun kurz zusammengefaßt werden. Die zugrundeliegende Datenstruktur ist durch $TUPLE$ gegeben. Sie wird um Operationen zur Zustandsbehandlung erweitert. *tuple* und *t_elem* werden in *state* bzw. *s_elem* umbenannt.

STATE_TYPE
make_binding: ident × val → s_elem
substitute: state × s_elem → state
get_val: state → (ident → val)

Für diesen Datentypen soll für alle $s : state$ gelten, daß $get\_val(s)$ eine Funktion ist. Diese entspricht der Funktion $STATE$ aus den vorangegangenen Abschnitten.

Elementare Stack-Operationen

Zuerst sollen die Operationen des klassischen abstrakten Datentypen *Stack* definiert und deren Stetigkeit nachgewiesen werden. Dazu werden zunächst die Operationen auf Stacks im einzelnen beschrieben. Wie oben angedeutet bilden Tupel die Realisierungsgrundlage des Stacks. Zum Teil wird allerdings $'(\ldots)'$ als Tupelkonstruktor statt *make_tuple* benutzt. Die Stetigkeit der Operationen soll gezeigt werden (nach Definition A.4.5). Eine Funktion ist stetig, wenn sie die Ordnung und die Suprema erhält.

Elementare Stack-Operationen sollen über Projektion pr oder Konkatenation $\circ$ realisiert werden. Zur Stetigkeit von pr, $\circ$ und des Tupelkonstruktors siehe Anhang A.4 oder [LS84]. Im folgenden soll eine Sorte *stack* eingeführt werden, die dem D^+ aus den vorangegangenen Untersuchungen entspricht.

Auch der Datentyp $STACK$ wird als Erweiterung von $TUPLE$ beschrieben. Stacks werden also durch Tupel realisiert. Der Typ der Stackelemente *s_elem* wird hier als generisch beschrieben. Für ihn muß, wie eingangs des Abschnitts erwähnt, cpo-Eigenschaft und Stetigkeit der Operationen gelten.

STACK[s_elem]

empty_stack: $\rightarrow$ *stack*	
top: stack $\rightarrow$ *s_elem*	*mit* $top(s) := \ pr_{\#s}(s)$
pop: stack $\rightarrow$ *stack*	*mit* $pop(s) := \ (pr_1(s), \ldots, pr_{\#s-1}(s))$
push: stack $\times$ *s_elem* $\rightarrow$ *stack*	*mit* $push(s, e) := \ s \circ (e)$
#: stack $\rightarrow$ *int*	*liefert die Anzahl der Elemente auf dem Stack*

Bei Definition der Operationen über Projektion und Konkatenation folgt die Stetigkeit der Stack-Operationen direkt aus der Stetigkeit der Projektion und Konkatenation sowie der Funktionsanwendung (siehe Sätze A.4.7 und A.4.8). # kann über einen externen Zähler realisiert werden, der von *push* und *pop* mitverwaltet wird. So wird das Problem einer rekursiven Definition für # umgangen.

Laufzeitstack-Operationen

Die eben unter $STACK$ zusammengefaßten Operationen $push, pop, top, empty\_stack$ und # realisieren den klassischen abstrakten Datentypen *Stack*. In der Anwendung im Laufzeitsystem eines ausführbaren Programmes sind darauf aufbauende Operationen notwendig, die Variablen lesen, d.h. den Wert einer Bindung in einem der Zustände ermitteln, Variablen schreiben und die Operationsaufrufe behandeln. Auf den elementaren Operationen werden nun anwendungsbezogene Operationen aufgesetzt. Diese sind aber noch unabhängig von der konkreten Realisierung der Stackelemente (d.h. unabhängig von der Abbildung $STATE$ aus Kapitel 3.3.1). Deren Integration erfolgt erst in den Abschnitten 4.1.2 und 4.1.3. Einige Vorarbeiten zur Ausführung sind schon zur Übersetzungzeit des Programmes durch den Compiler durchführbar. So ist bekannt, ob ein Objekt lokal oder global definiert ist. Ein geeigneter Aufruf kann erzeugt werden. Da in dieser Arbeit nur der Laufzeitaspekt definiert werden soll, werden obige Überlegungen vorausgesetzt und im folgenden nicht weiter betrachtet.

Die Anwendungsoperationen auf Stacks sind:

- Lesen von Variablen,

- Schreiben von Variablen,
- Aufruf einer Operation,
- Beenden einer Operation.

Auch für diese Operationen werden wieder die cpo-Eigenschaft zugrundeliegender Mengen und die Stetigkeit betrachtet.

Lesen von Variablen

Ein Compiler kann ermitteln, ob ein zu lesendes Objekt (*right-hand side* Kontext) lokal deklariert ist oder nicht. Die richtige Version der Variablenbewertung kann somit automatisch aufgerufen werden. Die unten stehenden Funktionen $read\_local$ und $read\_non\_local$ verfeinern die in Kapitel 3.3.1 benutzte Bewertung v. Sei x das zu lesende Objekt, $STACK$ der Laufzeitstack. *Displays* realisieren ein Konzept des Zugriffs auf nichtlokale Namen in einer blockstrukturierten Sprache mit verschachtelten Prozeduren. *Displays* werden in Kapitel 7.4 *Access to non-local names* in [ASU88] beschrieben. Displays wurde der Vorzug vor anderen Zugriffstechniken, wie z.B. den ebenfalls in [ASU88] beschriebenen *Access Links* gegeben, da die Realisierung über Displays im formalen Ansatz die einfachere Beschreibung ermöglicht.

Displays werden durch ein Feld d von Zeigern auf Activation Records realisiert. Das Feldelement $d[i]$ zeigt auf den Activation Record in der Aufruftiefe i. Displays werden so verwaltet, daß die $j-1$ ersten Elemente im Feld für eine Prozeduraktivierung p auf Aufruftiefe j auf die jeweils letzte Aktivierung der Prozeduren zeigen, die p syntaktisch umschließen. Um also auf einen nichtlokalen Wert zugreifen zu können, muß nur noch ein Zeiger verfolgt werden. Das Display kann wie der Laufzeitstack auf Tupeln realisiert werden. Stackoperationen können benutzt werden. Im folgenden soll mit d immer das Display bezeichnet werden.

Im folgenden seien die Elemente des Laufzeitstacks $STACK$ jeweils um eine Komponente $saved$ zur Speicherung eines Zeigers auf einen Activation Record des Laufzeitstacks erweitert.

lokal:
$$read\_local(STACK, x) := top(STACK)(x)$$

nichtlokal:
$$read\_non\_local(STACK, x) := pr(d[i])(STACK)(x)$$

d realisiert das Display. d kann als strikte Funktion $Nat \to Nat$ realisiert werden. Parameter von d ist die statische Tiefe der aktuellen Aktivierung. Sie kann zur Übersetzungszeit ermittelt werden. Der Aufruf $d[i]$ liefert den Zeiger auf den Laufzeitstackeintrag, in dem die letzte Aktivierung der Prozedur zu finden ist, in der das gesuchte Objekt x definiert ist. d ist stetig, da $(Nat, \sqsubseteq)$ flache cpo und d strikt ist.

Bemerkung 4.1.1 *Die statische Verschachtelungstiefe i ist von der zugehörigen Variable x funktional abhängig. Da die beiden Wertebereiche $IDENT$ und INT flache cpos sind, ist diese Funktion — Striktheit vorausgesetzt — stetig. Sie wurde in den Ausführungen daher nicht explizit benutzt.*

$pr : nat \rightarrow (tuple \rightarrow state)$ ist definiert durch

$$pr(j) := \begin{cases} pr_j & für\ j \in Nat \backslash \{\perp\} \\ \perp & j = \perp_{Nat} \end{cases}$$

$pr(j)$ liefert die Projektionsfunktion auf die j-te Komponente eines Elementes des Typs *tuple*, also $pr : tuple \rightarrow state$. j bezeichnet die letzte aktuelle Aktivierung der Prozedur, in der ein gesuchtes Objekt zu finden ist. $(Nat, \sqsubseteq)$ ist flache cpo und $[tuple \rightarrow state]$ sei die Menge der Funktionen von *tuple* nach *state*. Mit der kanonischen Ordnung auf Funktionsmengen nach Definition A.4.8 bildet die Menge eine cpo nach Satz A.4.5 und A.4.6. Aus der Flachheit von $(Nat, \sqsubseteq)$ und der Striktheit von pr folgen Monotonie und nach Satz A.4.4(2) Stetigkeit für pr.

Durch Verwendung von Displays konnte auf die Definition einer Routine verzichtet werden, die bei Verwendung von Access Links (siehe [ASU88]) rekursiv den Laufzeitstack nach dem definierenden Kontext durchsucht. Da nur noch ein Zeiger zu verfolgen ist, ist sie in der realisierten Variante nicht rekursiv.

Schreiben von Variablen

Analog zum Lesen kann auch hier ein Compiler benötigte Informationen ermitteln. Die Variable x befindet sich im *left-hand side* Kontext. Zustandsvariablen und andere nichtlokale Variablen werden nicht unterschiedlich behandelt. Schreiben von Variablen bedeutet die Änderung des Wertes, der an den Bezeichner gebunden ist. Schreiben einer lokalen Variable *write_local* führt die Änderung im obersten $STACK$-Element $top(STACK)$ durch. Die Substitution *substitute* ist am Anfang von Kapitel 3.4 beschrieben. Für den nichtlokalen Zugriff *write_non_local* muß zusätzlich der Sichtbarkeitsbereich ermittelt werden, in dem die Variable deklariert ist. Dies geschieht durch Zugriff über das Display. Da der Laufzeitstack als Tupel realisiert ist, kann auf die ermittelte Komponente projiziert werden.

lokal:

$$write\_local(STACK, x, val) := substitute(top(STACK), x \mapsto val)$$

nichtlokal:

$$write\_non\_local(STACK, x, val) := substitute(pr(d[i])(STACK), x \mapsto val)$$

Aus der Stetigkeit der Funktionen *top*, *substitute*, d und der Projektionen sowie der Funktionkomposition folgt die Stetigkeit der Funktionen *write_local* und *write_non_local*.

Aufruf einer Operation

Teilaufgaben sind:

- Zugriff auf die Operationsemantik: Die Verwaltung von Operationssemantiken, definiert durch $v^*(STATE, op)$ für eine Operation op, soll explizit gemacht werden. Dies entspricht dem Vorgehen des Compilers, der zur Übersetzungzeit Operationsdefinitionen in ablauffähigen Code transformiert und dessen Nutzung (Zugriff, Ausführung) zur Laufzeit sicherstellt. Da hier keine Rücksicht auf Speicherverwaltung genommen werden muß, werden Operationssemantiken im Laufzeitstack mitverwaltet.

  Der Compiler ermittelt, wie bei Variablen auch, ob die Operation lokal oder nichtlokal definiert ist, und im letzteren Fall wird über Displays auf den definierenden Kontext zugegriffen.

Sei op die aufzurufende Operation. Die Semantik von op wird über einen parallel zum Laufzeitstack verwalteten Stack $OpSTACK$ für Bindungen von Operationsnamen an ihre Werte ermittelt. Als Wertebereich der Abbildung $OpSTACK$ soll eine flache cpo angenommen werden. Der Wert einer Operation ist die Funktion, die ihre Definition realisiert. Im Kontext einer denotationalen Semantik ist das die Semantik der Operation selbst. Die Semantik für eine lokal definierte Operation erhält man durch $read\_local\_op$. Bei nichtlokal definierten Operationen wird zuerst der definierende Block ermittelt.

lokal:

$$read\_local\_op(OpSTACK, op) := top(OpSTACK)(op)$$

nichtlokal:

$$read\_non\_local\_op(OpSTACK, op) := pr(d[i])(OpSTACK)(op)$$

Da die Funktionen auf der Funktionskomposition aus der Projektion pr, dem Zugriff über Displays d und top gebildet werden, und deren Stetigkeit schon gezeigt wurde, sind nach Satz A.4.7 auch die Funktionen $read\_local\_op$ und $read\_non\_local\_op$ stetig. d realisiert das Display und ist ebenfalls, wie oben schon gezeigt, stetig.

- Neuen Laufzeitstackeintrag für die aufzurufende Operation erzeugen, d.h. Parameter bearbeiten und Display verwalten: Es sind Bindungen für die lokalen Variablen und Operation zu erzeugen. Weiterhin muß der Zeiger # inkrementiert werden.

$$
\begin{array}{l}
new\_activation(st, bindings) := \\
\quad if\ < non\_local >\ then \\
\quad\quad read\_non\_local\_op(OpSTACK, op); \\
\quad\quad STACK(\#STACK+1).saved := d[i]; \\
\quad\quad d[i] := \#STACK+1 \\
\quad else \\
\quad\quad read\_local\_op(OpSTACK, op); \\
\quad\quad STACK(\#STACK+1).saved := \ \#STACK \\
\quad end\ if; \\
\quad push(st, bindings)
\end{array}
$$

Hier wurde eine lesbare, imperative Darstellung der Operation gewählt (statt einer Darstellung über Funktionskomposition). Die Darstellung läßt sich aber leicht in die notwendige funktionale Form übertragen.

Zur Stetigkeit von $new\_activation$: Die Routine setzt sich aus den Elementen if-Operator, Lesen von Operationssemantiken sowie dem Lesen und Schreiben des Laufzeitstacks und des Displays zusammen. Sofern der Stetigkeitsnachweis für die Elemente von $new\_activation$ nicht schon erfolgt ist ($push$, pop, $read$, $write$, $\#, d[i]$), wird die Stetigkeit im folgenden Abschnitt 4.1.2 nachgewiesen. Die dort erzielten Ergebnisse für Stacks können auf Displays übertragen werden.

Beenden einer Operation

Beim Rücksprung nach Operationsausführung muß nur der Stackpointer dekrementiert und das Display aktualisiert werden.

$end\_activation(st) :=$
$d[i] := (pr(d[i])(st)).saved;$
$pop(st)$

Zur Stetigkeit können hier die Aussagen zu *new_activation* übernommen werden.

new_activation und *end_activation* verändern den Laufzeitstack. Sie legen einen neuen Zustand auf dem Stack ab bzw. nehmen das oberste Element wieder herunter. In der vorliegenden Anwendung wird ein neues Element auf dem Laufzeitstack abgelegt, falls eine neue Operation aktiviert wird. Dieses Element wird wieder entfernt, wenn die Aktivierung beendet ist. Im folgenden soll daher angenommen werden, daß *new_activation* die Routine *push* und *end_activation* die Routine *pop* mit ausführt. Außerdem sollen die beiden Funktionen den Operationsstack verwalten, d.h. bei Start einer neuen Aktivierung werden alle lokalen Operationen der neu aktivierten Operation auf dem $OpSTACK$ abgelegt und am Ende wieder entfernt.

Unter der am Anfang des Abschnitts angenommenen Voraussetzung, daß die Elementdomains des Stacks cpos sind, sind alle Operationen stetig.

RUNTIME_STACK
read_local : stack $\times$ *ident* $\rightarrow$ *val*
read_non_local : stack $\times$ *ident* $\rightarrow$ *val*
write_local : stack $\times$ *ident* $\times$ *val* $\rightarrow$ *stack*
write_non_local : stack $\times$ *ident* $\times$ *val* $\rightarrow$ *stack*
read_local_op : opstack $\times$ *ident* $\rightarrow$ *opval*
read_non_local_op : opstack $\times$ *ident* $\rightarrow$ *opval*
new_activation : stack $\times$ *state* $\rightarrow$ *stack*
end_activation : stack $\rightarrow$ *stack*

$RUNTIME\_STACK$ ist eine Erweiterung von $STACK[STATE]$. $STATE$ definiert den Typ der Stackelemente. Es werden sowohl Laufzeitstacks als auch Zustände über Tupel realisiert. Es werden sowohl Datenobjekte (*stack*) als auch prozedurale Objekte (*opstack*) verwaltet.

Die Integration der Stack-Operationen in die Semantik des Berechnungsmodells erfolgt erst in Kapitel 4.1.2 und 4.1.3.

Die Abbildung $STATE$

Funktionen $f : D \rightarrow E$ können grundsätzlich nur dann stetig sein, wenn $(D, \sqsubseteq)$ und $(E, \sqsubseteq)$ cpos sind. Also muß $(State, \sqsubseteq)$ für Funktionen auf Zuständen hier eine cpo bilden, wobei $State$ einen Wertebereich von Funktionen $STATE : IDENT \rightarrow VAL$ repräsentiert.

Lemma 4.1.3 *$(State, \sqsubseteq)$ ist cpo.*

Beweis: Sei $STATE : IDENT \rightarrow VAL$ mit $VAL := \{x | x \in INT \vee x \in BOOL \vee x = null \vee x = \omega\}$ flache cpo. $(State, \sqsubseteq)$ ist cpo, falls VAL cpo ist (nach Satz A.4.6). Nach Satz A.4.2 bildet jede Halbordnung mit $\perp$-Element und nur endlichen Ketten eine cpo. Durch die Endlichkeit des Namensraums $IDENT$ gibt es nur endliche Ketten, da es nur endlich viele eindeutige Abbildungen $STATE \in State$ geben kann. $\square$

Lemma 4.1.4 *$STATE$ beschreibt eine stetige Abbildung.*

Beweis: $STATE : IDENT_\omega \rightarrow VAL$ ist stetig, falls

- $(IDENT, \sqsubseteq)$ und $(VAL, \sqsubseteq)$ cpos sind,
- für alle Ketten $S \in IDENT$ das Supremum $\sqcup STATE(S)$ existiert und es gilt $STATE(\sqcup S) = \sqcup STATE(S)$.

$(IDENT_\omega, \sqsubseteq)$ und $(VAL, \sqsubseteq)$ sind flach, daher sind sie cpos. $IDENT_\omega$ enthält nur endliche Ketten, daher reicht es nach Satz A.4.4(2), die Monotonie $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$ zu zeigen. Da $STATE$ strikt ist, ist $STATE$ hier auch monoton. □

4.1.2 Rekursive Operationen

Vorbereitung

Im vorangegangenen Abschnitt 4.1.1 wurden die Formalisierung eines Laufzeitstack erläutert und Eigenschaften der Abbildung $STATE$ beschrieben. Im folgenden werden die Ergebnisse der Stack-Formalisierung auf das Konzept des Zustandes angewandt. Gleichzeitig wird Rekursion eingeführt. Der Zustand wird nun durch einen Stack von Zustandsabbildungen und nicht mehr durch eine einzelne Abbildung realisiert. Es wird dafür die Bezeichnung $STATE$ für den erweiterten Zustandsbegriff beibehalten.

Bevor mit der Integration von Rekursion begonnen wird, sollen zuerst λ-Terme eingeführt werden. Außerdem wird vorher noch die Konsistenz der bisherigen Semantikdefinition gezeigt. λ-Terme können als Basisdefinition für Prozeduren und Attribute dienen. Zudem werden sie benötigt, um rekursive Funktionen semantisch definieren zu können.

Definition 4.1.1 *Eine* **Basis** *für die λ-Notation ist ein Tripel $B = (S, OP, X)$ disjunkter Symbolmengen.*

- *S ist eine Menge von Basistypen*[2]*.*
- *OP ist eine Menge von Operationssymbolen.*
- *X ist eine Menge von Variablen.*

Jedem Operationssymbol und jeder Variable wird ein Typ zugeordnet.

Definition 4.1.2 *Die Menge der* **Typen über einer Basis** *B wird induktiv definiert:*

- *Jeder Basistyp ist ein Typ.*
- *Falls $\tau_1, \ldots, \tau_n, \tau$ Typen $(n \geq 1)$, dann ist $(\tau_1, \ldots, \tau_n \rightarrow \tau)$ ebenfalls ein Typ.*
- *Nichts sonst ist ein Typ.*

[2] Im folgenden bilden die Sortensymbole die Basistypen.

Definition 4.1.3 *Die Menge der* **λ-Terme** *für jeden Typ wird induktiv definiert:*

- *Jede Variable der Sorte τ ist ein λ-Term vom Typ τ.*
- *Jedes Operationssymbol $op : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$ ist ein λ-Term vom Typ $\tau_1, \ldots, \tau_n \rightarrow \tau$.*
- *Applikation: Falls $t_1, \ldots, t_n$ λ-Terme der Typen $\tau_1, \ldots, \tau_n$, $n \geq 1$ und u ein λ-Term des Typs $\tau_1, \ldots, \tau_n \rightarrow \tau$. Dann ist $u(t_1, \ldots, t_n)$ ein λ-Term des Typs τ.*
- *Abstraktion: Falls $x_1, \ldots, x_n$ paarweise verschiedene Variablen der Typen $\tau_1, \ldots, \tau_n$, $n \geq 1$, und t ein λ-Term des Typs σ, dann ist $\lambda x_1, \ldots, x_n.t$ ein λ-Term des Typs $\tau_1, \ldots, \tau_n \rightarrow \sigma$.*
- *Nichts sonst ist ein λ-Term.*

Mit T soll die Menge der λ-Terme erweitert um ein Element $\perp_T$ bezeichnet werden. Die Ordnung $\sqsubseteq_T$ auf T soll als flache Ordnung auf T definiert sein.

Folgerung 4.1.3 *$(T, \sqsubseteq_T)$ ist flache cpo.*

Beweis: Nach Definition. □

Integration von λ-Termen in den Ansatz

Falls die $x_i : s_i$ verschiedene freie Variablen sind, d.h. $x_i \in X_{s_i}, i = 1, .., n$, und t ein beliebiger Term ist sowie die $t_i : s_i$ Terme einer Datensorte s_i sind, dann ist $\lambda x_1, \ldots, x_n \; . \; t$ eine λ-*Abstraktion* und $\lambda x_1, \ldots, x_n \; . \; t \; (t_1, \ldots, t_n)$ eine *Applikation*.

$$
\begin{aligned}
&v^*(STATE, \lambda x_1, \ldots, x_n \; . \; t \; (t_1, \ldots, t_n)) := v^*(STATE', t) \text{ mit} \\
&\qquad STATE' = substitute(STATE, x_i \mapsto v^*(STATE, t_i))
\end{aligned}
$$

$$
\begin{aligned}
&v^*(STATE, \lambda x_1, \ldots, x_n \; . \; t) := f \\
&\qquad \text{mit } f(STATE, y_1, \ldots, y_n) = v^*(STATE', t) \text{ mit } y_i \in A_{s_i} \text{ falls } x_i : s_i \\
&\qquad\quad \text{es ist } STATE' = substitute(STATE, x_i \mapsto y_i),\; i = 1, .., n \\
&\qquad\quad \text{mit } f : \; State \times A_{s_1} \; \times \ldots \times A_{s_n} \rightarrow State \times A_s
\end{aligned}
$$

Falls t ein Ausdruck der Sorte s ist, so wird

$$pr_2(\lambda x_1, \ldots, x_n \; . \; t \; (t_1, \ldots, t_n))$$

als Term der Sorte s zugeordnet. Falls t ein Kommando ist, so wird

$$pr_1(\lambda x_1, \ldots, x_n \; . \; t \; (t_1, \ldots, t_n))$$

als Term der Sorte *state* zugeordnet.

Semantik einer λ-Abstraktion ist eine Funktion f. f ist definiert durch die Semantik des Termes t. Bei Aufruf von f wird jedes freie Auftreten der formalen Parameter x_i durch aktuelle Parameter ersetzt.

Satz 4.1.1 *Sei v^* eine Auswertung für eine Objektsignatur Σ.*

1. *Für jeden Term t der Sorte $s \in S \cup \{state\}$ gilt:*
   $v^*(STATE, t) \in A_{State} \times A_s$ für alle Zustände $STATE \in State$.
2. *v^* ist stetig.*

Beweis: Es gilt $v^*(STATE, t) = v'(t)(STATE)$ mit $v' : T \to (State \to State \times A_s)$ definiert als $v' : t \mapsto v_t$ für alle $STATE \in State, t \in T$. Dann ist v^* stetig, wenn v' und v_t für alle $t \in T$ stetig sind (siehe Definition A.4.9 und Satz A.4.9).

Teil A: Stetigkeit von v'.

Nachweis der cpo-Eigenschaft der Wertebereiche:

1. $State \to State \times A_s$
   Da $(State, \sqsubseteq)$ und $(A_s, \sqsubseteq)$ cpos sind, sind die Menge der Funktionen und die Menge der stetigen Funktionen über $State \to State \times A_s$ nach den Sätzen A.4.6 und A.4.5 cpos über der kanonischen Ordnung auf Funktionen (Definition A.4.8).
2. $(T, \sqsubseteq)$ ist flache cpo nach Folgerung 4.1.3.

Seien E, F, G cpos und $\phi : E \times F \to G$ eine Abbildung mit

- F ist flach,
- $\phi(e, \perp) = \phi(\perp, f) = \perp$.

Setze für $f \in F$

$$\phi_f := \lambda e.\phi(e, f), \text{ d.h. } \phi_f(e) = \phi(e, f).$$

Dann ist $\phi : E \times F \to G$ genau dann stetig, wenn $\forall f \in F.\ \phi_f : E \to G$ stetig ist. Es reicht also unter der Voraussetzung, daß $(T, \sqsubseteq)$ flache cpo ist, zu zeigen, daß die v_t stetig sind.

Teil B: Stetigkeit der v_t.

Dieser Teil des Beweises erfolgt durch strukturelle Induktion über die Konstrukte.

B.1 Beweis der Basiskonstrukte:

a.1) t ist Variable oder Zustandsbezeichner: $t \equiv x$ mit $x \in Ident$ und $x : s$.
$v^*(STATE, x) := [STATE, read(STATE, x)]$
$\Rightarrow$ 1. erfüllt

read[3] ist stetig (siehe Kapitel 4.1.1). Wie in 3.1 gezeigt wurde, bildet jede Trägermenge eines der zur Verfügung stehenden Datentypen eine cpo. Also ist auch $(A_s, \sqsubseteq)$ cpo. Damit ist nach Satz A.4.3 und den Anmerkungen zu den Laufzeitstack-Operationen v^* stetig, da *read* stetig ist.
$\Rightarrow$ 2. erfüllt

[3] Statt *read* müßte eine der beiden Formen *read_local* oder *read_non_local* aufgerufen werden. Da der Compiler die Auswahl treffen kann, und beide Funktionen stetig sind, wird hier vereinfacht.

a.2) t ist Operation op: Die Bedeutung (Semantik) einer Operation wird zur Übersetzungszeit vom Compiler eindeutig festgelegt. Operationsnamen $op \in ATTR \cup PROC$ werden zustandsunabhängig immer gleich interpretiert. Daher ist $v^*(STATE, op)$ eine konstante Funktion und somit stetig.

B.2 Beweis der zusammengesetzten Konstrukte:

b.) t ist Applikation: $t \equiv a(t_1, \ldots, t_n)$ mit $a : state \times s_1 \times \ldots \times s_n \rightarrow states \times s$

$$\begin{aligned} & v^*(STATE, a(t_1, \ldots, t_n)) := \\ & \qquad v^*(STATE, a)(STATE, v^*(STATE, t_1)[2], \ldots, v^*(STATE, t_n)[2]) \end{aligned}$$

Sei nach Induktionshypothese $v^*(STATE, t_i) \in State \times A_{s_i}$ und $v^*(STATE, a) \in (State \times A_{s_1} \times \ldots \times A_{s_n} \rightarrow State \times A_s)$, der Menge aller Funktionen der angegebenen Signatur wie in Definition A.4.7 festgelegt.
Somit ist $v^*(STATE, a(t_1, \ldots, t_n)) \in State \times A_s$.
$\Rightarrow$ 1. erfüllt

Der Nachweis der Stetigkeit soll über die Zerlegung von v^* in zwei Funktionen g und $apply$ mit $v^* = apply \circ g$ erfolgen[4]. Nach Induktionshypothese sind alle $v^*(STATE, t_i)$ stetig.
Sei $g : State \rightarrow A_{s_1} \times \ldots \times A_{s_n}$ mit
$\quad g(STATE) := (STATE, v^*(STATE, t_1)[2], \ldots, v^*(STATE, t_n)[2])$
Damit ist auch g stetig (da g über den Tupelkonstruktor *make_tuple* realisiert wird).
Nach Induktionshypothese ist $v^*(STATE, a)$ stetig.
Mit $apply : [D \rightarrow E] \times D \rightarrow E$
$\quad apply(f, d) := f(d)$ ist stetig ($\rightarrow$ Currying, siehe [LS84] S. 83, hinter 4.23)
folgt
$\quad v^*(STATE, t) := apply(v^*(STATE, a), g(STATE))$ ist stetig.
$\Rightarrow$ 2. erfüllt

c.) t ist λ-Abstraktion: $t \equiv \lambda x_1, \ldots, x_n.t$ mit $\lambda : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$

$$\begin{aligned} & v^*(STATE, \lambda x_1, \ldots, x_n \,.\, t)(d_1, \ldots, d_n) := \\ & \qquad v^*(STATE', t)(d_1, \ldots, d_n) \text{ mit} \\ & \qquad STATE' = substitute(STATE, x_i \mapsto d_i), d_i \in A_{s_i},\ i = 1, .., n \end{aligned}$$

Somit gilt:

- $v^*(STATE, \lambda x_1, \ldots, x_n \,.\, t) \in State \times A_s$, falls $v^*(STATE', t) \in State \times A_s$,
- $v^*(STATE, \lambda x_1, \ldots, x_n \,.\, t)$ ist stetig, falls $v^*(STATE', t)$ stetig ist,

für alle $STATE' \in State$. Dies gilt wegen der Stetigkeit von *substitute* und nach Induktionshypothese.

d.) t ist Prozeduranwendung: $t \equiv p(x_1, \ldots, x_n)$
Da $v^*(STATE, \lambda x_1, \ldots, x_n \,.\, t) = v^*(STATE, op(x_1, \ldots, x_n) = t)$ gilt: analog b.) (siehe Anfang von 4.1.2 und Kapitel 3.4).

[4] Die Komposition $f \circ g$ ist stetig, wenn f und g stetige Funktionen auf cpos sind

e.) t ist das leere Kommando *skip*: trivial.

f.) t ist Zuweisung: $t \equiv asgn(z_i, t)$

$$v^*(STATE, asgn(z_i,t)) := \\ \quad [write(STATE, z_i, v^*(STATE,t)[2]), null]$$

$\Rightarrow$ 1., 2. erfüllt, da *write* stetig ist (siehe Kapitel 4.1.1) und *null* konstant ist.

g.) t ist bedingtes Kommando : $t \equiv if(b, c)$

$v^*(STATE, if(b,c)\ ) :=$
  `if` $v^*(STATE, b) =_{bool} TRUE$ `then`
    $v^*(STATE, c)$
  `else` `if` $v^*(STATE, b) =_{bool} \bot_{bool}$ `then`
      $[STATE, \omega]$
    `else`
      $[STATE, null]$
    `end`
  `end`

$if : Bool_\bot \times D \times D \rightarrow D$ wobei $(Bool_\bot, \sqsubseteq)$ und $(D, \sqsubseteq)$ cpos sind.

$$if(b, d_1, d_2) := \begin{cases} d_1 & falls\ b = true \\ d_2 & falls\ b = false \\ \bot_D & falls\ b = \bot_{bool} \end{cases}$$

if ist strikt und erhält die Suprema, d.h. if ist stetig. Da ω Bottomelement jedes Domains und $v^*(STATE, c) \in A_{state} \times A_s$:
$\Rightarrow$ 1. erfüllt

v^* ist stetig, da der *if*-Operator $if : Bool_\omega \times [State \times A_s] \times [State \times A_s] \rightarrow [State \times A_s]$ stetig ist.

`if` $v^*(STATE, b) = TRUE$ `then` $v^*(STATE, c)$ `else`
 `if` $v^*(STATE, b) = \omega$ `then` $[STATE, \omega]$ `else` $[STATE, null]$ `end end` $:=$
 $if(v^*(STATE, b), v^*(STATE, c), [STATE, null])$

$\Rightarrow$ 2. erfüllt

h.) t ist Kommandosequenz: $t \equiv seq(com1, com2)$

$$v^*(STATE, seq(com1, com2)) := \\ \quad v^*(v^*(STATE, com1)[1], com2)$$

Nach Induktionshypothese gilt:

$v^*(STATE, com1) \in State \times A_s$ stetig,
$v^*(STATE, com2) \in State \times A_s$ stetig.

$\Rightarrow$ 1. erfüllt

Da $v^*(STATE, com1)[1]$ Element von $State$ (eine cpo) ist:
$\Rightarrow$ 2. erfüllt □

Damit ist für alle Konstrukte, die in Kapitel 3 vorgestellt wurden, die Stetigkeit der Auswertung v^* gezeigt. Für die dort vorgestellten Konstrukte ist jetzt also die Semantik von Mengen und Funktionen auf cpos und stetige Funktionen erweitert worden. Außerdem wurde auch der Zustandsbegriff so erweitert, daß Umgebungen zur Bindung von Werten an Variablen in Blöcken auf einem Stack verwaltet werden können. Rekursion wird nun eingeführt.

Rekursion

Der Nachweis der Stetigkeit ist Voraussetzung für die Anwendbarkeit einer Fixpunktsemantik zur Definition rekursiver Operationen. Die Erweiterung von Operationsdefinitionen auf Rekursivität soll nun erfolgen. Die Behandlung der Rekursion in diesem Abschnitt stützt sich auf [LS84]. Die Grundkonzepte der Fixpunktsemantik werden hier nicht vorgestellt (siehe Anhang A.4).

Es wird zunächst nur direkte Rekursion behandelt. Es werden also Programme der Form

$$op(x_1, \ldots, x_n) = t$$

betrachtet, in denen op innerhalb des Terms t (rekursiv) aufgerufen wird. An der Syntax sind gegenüber der bisherigen Notation keine Veränderungen vorzunehmen.

Die Semantik einer rekursiv definierten Operation (λ-Abstraktion analog) werden wir mit Hilfe der Fixpunktsemantik definieren.

Im folgenden sei $P \equiv op(x_1, \ldots, x_n) = t$ für $op : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$.

$$v^*(STATE, P) := \mu\Phi_{v^*}(STATE, P)$$

mit

$$\Phi_{v^*}(STATE, P) : (State_\omega \times [D_\omega \rightarrow D'_\omega]) \rightarrow (State_\omega \times [D_\omega \rightarrow D'_\omega]),$$
$$D_\omega \equiv State_\omega \times A_{s_1\omega} \times \ldots \times A_{s_n\omega},\ D'_\omega \equiv State_\omega \times A_{s_\omega}$$
$$\Phi_{v^*}(STATE, P) = v^*(STATE, \lambda op.\ \lambda st, x_1, \ldots, x_n\ .\ t).$$

$v^*(STATE, \lambda st, x_1, \ldots, x_n.t) := f_1$ mit $f_1 : State_\omega \times A_{s_1\omega} \times \ldots \times A_{s_n\omega} \rightarrow State_\omega \times A_{s_\omega}$ (also $D_\omega \rightarrow D'_\omega$) nach Definition der λ-Abstraktion. D.h. $v^*(STATE, \lambda op.\lambda st, x_1 \ldots x_n.t) := f_2$ mit $f_2 : (State \times [D_\omega \rightarrow D'_\omega]) \rightarrow (State \times [D_\omega \rightarrow D'_\omega])$.

$\Phi_{v^*}(STATE, P)$ ist das zur rekursiven Operation P gehörende Funktional (vgl. [LS84]), $\mu\Phi_{v^*}(STATE, P)$ bezeichnet dann den kleinsten Fixpunkt von $\Phi_{v^*}(STATE, P)$.

Lemma 4.1.5

1. *$\Phi_{v^*}(STATE, P)$ ist stetig, da $State \times [D_\omega \rightarrow D'_\omega]$ cpo ist.*

2. *Da $\Phi_{v^*}(STATE, P)$ stetig ist, existiert der kleinste Fixpunkt von Φ_{v^*}, also $\mu\Phi_{v^*}(STATE, P)$ mit dem Wert $\mu\Phi_{v^*}(STATE, P) = \sqcup\{(\Phi_{v^*}(STATE, P))^i(\perp) | i \geq 1\}$.*

Beweis:

1. Folgt aus den Sätzen A.4.3 und A.4.6, da $(State, \sqsubseteq)$, $(A_{s_i}, \sqsubseteq)$, $i = 1, .., n$ und $(A_s, \sqsubseteq)$ cpos) und Satz 4.1.1 (dort wird die Stetigkeit für $v^*(STATE, t)$ für λ-Abstraktionen t gezeigt).

2. Anwendung des Fixpunkttheorems A.4.10. □

Damit läßt sich etwa der Operationsaufruf wie folgt definieren (λ-Applikation analog):

$$\begin{aligned}
&v^*(STATE, op(a_1, \ldots, a_n)) := \\
&\qquad v^*(STATE, op)(STATE, v^*(STATE, a_1)[2], \ldots, v^*(STATE, a_n)[2]) \\
&\qquad \text{mit } v^*(STATE, op) := \mu\Phi_{v^*}(STATE, op(x_1, \ldots, x_n) = t) \quad \text{und} \\
&\qquad \Phi_{v^*}(STATE, P) = v^*(STATE, \lambda op.\lambda st, x_1, \ldots, x_n.t).
\end{aligned}$$

Jetzt soll Satz 4.1.1 für rekursive Operationen erweitert werden.

Satz 4.1.2 *Für jede rekursive Operation $P \equiv op(x_1, \ldots, x_n) = t$ gilt:*

1. $v^*(STATE, P) \in A_{state} \times A_s$ *für alle Zustände in State,*

2. $v^*(STATE, P)$ *ist stetig.*

Beweis: Die Konstruktion $v^* = v_t \circ v'$ aus Satz 4.1.1 gilt auch hier. v' ist unverändert.

1. Der Fixpunkt existiert und ist eindeutig, also ist v^* wohldefiniert.

2. $v^*(STATE, P) = \mu\Phi_{v^*}(STATE, P)$, wobei $\Phi_{v^*}(STATE, P)$ stetig nach obigem Lemma ist. Die Stetigkeit des Fixpunktoperators μ ist nach Satz A.4.14 gegeben. Mit Lemma A.4.7 (Stetigkeit der Funktionskomposition) ist auch $\mu\Phi_{v^*}(STATE, P)$ stetig. Daraus folgt die Stetigkeit von v^*. □

Damit ist die Fixpunktsemantik definiert. Rekursive Funktionen sind nun modellierbar.

4.1.3 Integration der Verschachtelung

Bisher gab es nur einen globalen Zustand, der die Bezeichner des Zustands und die Parameter der Operationen verwaltete. Das setzte die globale Eindeutigkeit von Bezeichnern voraus. Im folgenden wird die Bearbeitung lokaler Zustände im Laufzeitstack realisiert, d.h. Zustände, die lokal zu Operationsaktivierungen sind. Ihre Aufgabe ist die Verwaltung lokaler Objekte, wie z.B. der Parameter. Die Ergebnisse aus Abschnitt 4.1.1 werden hierbei benutzt. Veränderungen in der Semantik sind also nur bei Operationsaufrufen (Aufruf) und Operationsdefinitionen (Abstraktion) vorzunehmen.

Aufruf

Sei $op(x_1 : s_1, \ldots, x_n : s_n) = t$.

$$v^*(STATE, op(a_1, \ldots, a_n)) := [end\_activation(v^*(STATE', t)[1]),\ v^*(STATE', t)[2]\ ]$$
$$\text{mit } STATE' = new\_activation(STATE, \{[x_i \mapsto v^*(STATE, a_i)[2]\ ]^n\})$$

$[x_i \mapsto y_i]^n$ erzeugt Bindungen $[x_1 \mapsto y_1], .., [x_n \mapsto y_n]$. Durch den Ausdruck $[x_i \mapsto v^*(STATE, a_i)[2]\ ]^n$ werden Bindungen aktueller Parameterwerte $v^*(STATE, a_i)[2]$ an formale Parameterbezeichner x_i erzeugt. Diese werden von *new_activation* als neuer, lokaler Zustand für die aufgerufene Operation auf dem Laufzeitstack $STATE$ abgelegt. Nachdem der Rumpf t der Operation op ausgeführt wurde, werden von *end_activation* die beim Rücksprung notwendigen Maßnahmen durchgeführt (siehe Abschnitt 4.1.1).

Abstraktion

Sei $v^*(STATE, \lambda x_1, \ldots, x_n.t) := f$ mit $f : State \to State \times A_s$ für $t : s$.

$$f(STATE)[1] := end\_activation(v^*(STATE', t)[1]) \text{ und } f(STATE)[2] := v^*(STATE', t)[2]$$
$$\text{mit } STATE' := new\_activation(STATE, \{[x_i \mapsto y_i]^n\})$$

Obwohl es hier nicht explizit erwähnt wird, werden lokale Operationen mitverwaltet (vgl. Diskussion $OpSTACK$ in Abschnitt 4.1.1). Prozedurdefinitionen in Attributen werden nicht zugelassen.

Da alle benutzten Funktionen sowie die Funktionskomposition stetig sind, ist auch hier Stetigkeit gewährleistet. Somit haben wir also eine Fixpunktsemantik im Sinne denotationaler Semantik für alle bisher definierten Sprachkonstrukte angegeben.

4.2 Sonstige Erweiterungen prozeduraler Abstraktionen

Im Rahmen von sonstigen Erweiterungen der prozeduralen Abstraktionen sollen noch folgende Aspekte behandelt werden:

- lokale Objekte in Prozeduren,
- verschiedene Parameterübergabemechanismen,
- Vereinheitlichung von Prozeduren und Attributen (Seiteneffekte in Ausdrücken),
- indirekte Rekursion.

Damit sind die wichtigsten Prozedurkonzepte abgehandelt (siehe [Hor84] und [ASU88]). Der in Kapitel 4.1 erarbeitete Ansatz wird in den Abschnitten dieses Kapitels nicht fortlaufend weiterentwickelt. Die Abschnitte bauen nicht aufeinander auf, sondern werden sich jeweils auf den Ansatz in Kapitel 4.1 stützen, um die Notation möglichst übersichtlich zu halten, und somit die Umsetzung der Konzepte in den Formalismus klarer herausstellen zu können.

4.2.1 Lokale Objekte in Prozeduren

Die notwendigen Erweiterungen an Syntax und Semantik zur Behandlung lokaler Variablen in Prozeduren sollen nun eingeführt werden. Lokale Variablen in Attributen werden nicht zugelassen. Attribute sollen ihren rein funktionalen Charakter behalten, d.h. nur *rd*-Parameter und keine Möglichkeit zum Aufruf von Prozeduren.

Syntaktisch werden Prozedurvereinbarungen mit lokalen Variablen in der Form

$$p : state \times s_1 \times \ldots \times s_n \rightarrow state \times s \quad (locals\ s_{l_1} \times \ldots \times s_{l_m})$$

angegeben. Die $s_i, i = 1, .., n$ sind die Sorten der Parameter, die $s_{l_j}, j = 1, .., m$ sind die Sorten der lokalen Variablen. Die s_i und die s_{l_j} sind unabhängig voneinander. In den Kommandos eines Prozeduraufrufs ist nun auch eine Zuweisung an lokale Objekte $l_i : s_i$ erlaubt. l_i ist ein Term, der wie ein Zustandsbezeichner behandelt werden kann. Analog sollen auch lokale Objekte über Umgebungen verwaltet werden. Dazu wird beim Aufruf einer Prozedur mit lokalen Variablen vor deren Ausführung eine neue Umgebung auf dem Laufzeitstack angelegt (*push*). Die notwendige Verschattung kann mit diesem Mechanismus realisiert werden. Nach Abarbeitung der Prozedur wird die Umgebung, d.h. die lokalen Objekte, wieder entfernt (*pop*).

Sei $P \equiv p()\ locals\ l_1 : s_{l_1}, \ldots, l_m : s_{l_m}.\ com$. Die Semantik läßt sich für nichtrekursive Prozeduren also wie folgt definieren[5]. Parameter werden hier der Einfachheit halber nicht benutzt, sie würden wie üblich behandelt.

$$v^*(STATE, p()\ ) := [\ end\_activation(v^*(STATE', t)[1]),\ v^*(STATE', t)[2]\ ]$$
$$\text{mit } STATE' = new\_activation(STATE, \{[l_i \mapsto \omega]^m\})$$

Die Semantik läßt sich auch auf rekursive Prozeduren übertragen, nur darf dann der Prozedurrumpf nicht direkt verwendet werden. Statt dessen muß über den Fixpunkt definiert werden.

Die Aussagen der Sätze 4.1.1 und 4.1.2 gelten auch hier. Dies folgt direkt aus den Untersuchungen zu den Funktionalitätserweiterungen von $STATE$, da die Stetigkeit der einzelnen Funktionen schon gezeigt wurde.

4.2.2 Parameterübergabemechanismen

Um verschiedene Parameterübergabeverfahren realisieren zu können, ist die Einführung von Parametermodi unerläßlich. Es sollen *Call_by_Result*, *Call_by_Value/Result*, *Call_by_Reference* und *Call_by_Name* neben der schon behandelten Form *Call_by_Value* betrachtet werden. Allerdings sind die neuen Formen nur für Prozeduren, nicht aber für Attribute relevant.

$$p : state \times \text{ pm } s_1 \times \ldots \times \text{ pm } s_n \rightarrow state \times s \text{ ,wobei pm } \in \{rd,\ wr,\ rw,\ ref\}$$

Parameter, die mit einem der beiden Modi *wr* oder *rw* bezeichnet sind, können wie lokale Objekte behandelt werden. Für die *Call_by_Value/Result*-Implementierung muß, ebenso wie für *Call_by_Value*, statt ω der aktuelle Wert eines Parameters a bei Aufruf an x gebunden

[5] $\{[l \mapsto \omega]\}$ beschreibt eine (hier einelementige) Menge von Bindungen. Es wird der undefinierte Wert ω initial an die lokale Variable l gebunden.

werden ($[x \mapsto v^*(STATE, a)[2]\,]$). Nach Abarbeitung der Prozedur wird die Zuweisung des letzten Wertes des Parameters an die im Aufruf benutzte Variable durchgeführt.

Sei $P \equiv p(rw\ x_1 : s_1, \ldots, rw\ x_n : s_n).\ com$. Die Semantik läßt sich für nichtrekursive Prozeduren also wie folgt definieren:

$$\begin{array}{l} v^*(STATE, p(a_1, \ldots, a_n)\ ) := \\ \quad [\ end\_activation(v^*(v^*(STATE', com)[1], seq(asgn(x_1, a_1), \ldots, asgn(x_n, a_n)))[1]), \\ \quad\ v^*(v^*(STATE', com)[1], seq(asgn(x_1, a_1), \ldots, asgn(x_n, a_n)))[2]\ ] \\ \text{mit } STATE' = new\_activation(STATE, \{[x_i \mapsto v^*(STATE, a_i)[2]]^n\}) \end{array}$$

$STATE'$ ist der um die Bindung der Werte der aktuellen Parameter a_i an die formalen Parameter x_i erweiterte Laufzeitstack. In diesem Zustand wird die Semantik von p im Zustand direkt vor dem Aufruf mit leerer Parameterliste aufgerufen. Im resultierenden Zustand nach Abarbeitung von p wird vor dem *pop*-Aufruf der aktuelle Wert von x_i noch an a_i zugewiesen. Bei einem wr-Parameter wird in der neuen Umgebung $[x_i \mapsto \omega]$ eingetragen, also ω anstelle des Wertes des aktuellen Parameters.

Auch *Call_by_Value* kann auf diese Art realisiert werden. Bei dieser Form der Parameterübergabe entfällt die Zuweisungssequenz nach Ausführung von p.

Bei *Call_by_Reference* werden Referenzen auf das Argument an die Operation übergeben. Ausdrücke als aktuelle Parameter sollen hier nicht erlaubt werden[6]. Aktuelle Referenzparameter müssen jetzt wie nichtlokale Objekte behandelt werden.

Sei $P \equiv p(ref\ x_1 : s_1, \ldots, ref\ x_n : s_n).\ com$ Definition einer Prozedur. Die Semantik eines Aufrufs läßt sich für Variablen $y_1, \ldots, y_n$ wie folgt definieren:

$$v^*(STATE, p(y_1, \ldots, y_n)\ ) := [\ v^*(STATE', com)[1],\ v^*(STATE', com)[2]\ ]$$

mit $STATE' = new\_activiation(STATE, \{[x_i \mapsto STATE(y_i)]^n\})$ und $OID(x_i) := y_i,\ i = 1, .., n$.

$OID : IDENT \rightarrow IDENT$ ist eine Abbildung, die bei Aufruf dem formalen Parameter seinen aktuellen Parameter zuordnet. Da $(IDENT, \sqsubseteq)$ eine flache cpo ist, kann OID als stetige Abbildung definert werden. Bei Aufruf wird dem formalen Parameter x_i der aktuelle Wert von y_i zugeordnet. Bei Veränderung von x_i muß nun die Bindung von y_i geändert werden. Dazu wird die Zuweisung modifiziert. Falls z kein Referenzparameter ist, ist $asgn(z, t)$ wie üblich definiert. Sollte z Referenzparameter sein, so sei definiert:

$$v^*(STATE, asgn(z, t)) := [substitute(STATE,\ OID(z) \mapsto v^*(STATE, t)[2]), null]$$

Dadurch wird der aktuelle Parameter $OID(z)$ geändert. Welche der beiden Formen zu benutzen ist, kann statisch vom Compiler festgestellt werden. Da OID stetig ist, gilt dies auch für die modifizierte Definition von $asgn$.

Call_by_Name wird üblicherweise durch textuelle Substitution realisiert. Da hier nur die dynamische Semantik betrachtet wird, muß *Call_by_Name* also nicht betrachtet werden. [WG84] stellen die *Call_by_Name*-Realisierung von ALGOL60 vor. Dort wird für einen *Call_by_Name*-Parameter eine parameterlose Prozedur bei Aufruf übergeben, die eine Referenz auf das

[6]Sie könnten durch temporäre Variablen behandelt werden (siehe [WG84] Abschnitt 2.5.3).

Argument berechnet. Zugriffe auf den Parameter erfolgen dann über diese Prozedur. Die notwendigen Techniken (lokale Prozeduren, Behandlung von Referenzen) stehen bereit. Da *Call_by_Name* in der ALGOL60-Ausprägung fast keine Anwendung findet, soll auf die formale Umsetzung verzichtet werden.

4.2.3 Vereinheitlichung von Prozeduren und Attributen

Mit der Vereinheitlichung von Prozeduren und Attributen sind zwei Ziele verbunden. Zum einen sollen Seiteneffekte in Ausdrücken beschreibbar sein. Zum anderen geht es um die Vereinheitlichung der Notation. Rein funktionale Attribute könnten zudem beibehalten werden.

Bisher wurden Operationen streng unterschieden in Operationen, die den Zustand verändern (Prozeduren) und Operationen, die den Zustand nicht verändern dürfen, dafür aber einen Wert zurückliefern (Attribute). Anfänglich wurde diese Trennung gefordert, um diese beiden unterschiedlichen Konzepte auch getrennt voneinander einführen zu können. Die Notation wurde allerdings schon so angelegt, daß jetzt die Vereinheitlichung leicht erfolgen kann.

Jede Operationsbeschreibung hat jetzt die Form

$$
\begin{array}{l}
\textit{operation } op(x_1 : s_1, \ldots, x_n : s_n) \textit{ returns } s \\
\qquad \textit{locals } l_1 : s_{l_1}, \ldots, l_n : s_{l_n} \\
\textit{begin} \\
\qquad com \\
\textit{end}
\end{array}
$$

Jede Operation besteht nun aus einer Kommandofolge. Es gibt zusätzlich eine *return*-Anweisung, um die Wertrückgabe zu realisieren:

$$return\ expr$$

Implizit wird als letzte Anweisung in Operationen '*return null*' gesetzt.

Seiteneffekte sind nun überall da möglich, wo Ausdrücke erlaubt sind, d.h. auf der rechten Seite einer Zuweisung, in einer *return*-Anweisung und in der Parameterliste eines Operationsaufrufes.

Es ergeben sich folgende Änderungen in der Semantik der drei genannten Konstrukte.

- $v^*(STATE, asgn(z,t)) := [substitute(v^*(STATE,t)[1], z \mapsto v^*(STATE,t)[2]), null]$

  Die Substitution findet jetzt im Folgezustand der Auswertung von t statt.

- Im Beispielaufruf $op(e_1, e_2)$ sollen e_1 und e_2 Seiteneffekte haben und rd-Parameter von op sein. Es soll keine Reihenfolge vorgegeben werden, in der die beiden Ausdrücke ausgewertet werden. Eine mögliche Semantik ist die Auswertung von links nach rechts. Für zwei Parameter ergäbe sich folgendes:

  $$
  \begin{array}{l}
  v^*(STATE, op(t_1,t_2)) := \\
  \quad v^*(v^*(v^*(STATE,t_1)[1],t_2)[1], op)(v^*(STATE,t_1)[2], v^*(v^*(STATE,t_1)[1],t_2)[2])
  \end{array}
  $$

  t_2 wird im Folgezustand von t_1 ausgewertet. Die Operation op wird im Folgezustand von t_2 ausgeführt.

- $v^*(STATE, return\ t) := [v^*(STATE, t)[1], v^*(STATE, t)[2]]$
  Die Semantik entspricht der Auswertung $v^*(STATE, t)$.

Die Verallgemeinerung von Prozedur- und Attributaufruf wird mit 'Operationsaufruf' bezeichnet.

4.2.4 Indirekte Rekursion

Statt nur einer direkt-rekursiven Operation sollen nun mehrere indirekt-rekursive Operationsdefinitionen betrachten werden.

$$op_i(x_{i_1}, \ldots, x_{i_{m_i}}) = t_i, \quad i = 1, .., n$$

Sei $D^* = [D^1_\omega \rightarrow D_\omega] \times \ldots \times [D^n_\omega \rightarrow D_\omega]$. $D_\omega \equiv State \times A_s$ und $D^j_\omega \equiv State \times A_{s_{j_1}} \times \ldots \times A_{s_{j_m}}, j = 1, .., n$. Dann ist das semantische Funktional $\Phi_{v^*}(STATE, P) : State \times D^* \rightarrow State \times D^*$ (P ist die Menge der rekursiven Operationen $op_1, \ldots, op_n$) komponentenweise durch Projektion definiert:

$$\begin{aligned}
pr_1 \circ \Phi_{v^*}(STATE, P) &= v^*(STATE, \lambda op_1, \ldots, op_n.\lambda x_{1_1}, \ldots, x_{1_{m_1}}.t_1) \\
&\vdots \\
pr_n \circ \Phi_{v^*}(STATE, P) &= v^*(STATE, \lambda op_1, \ldots, op_n.\lambda x_{n_1}, \ldots, x_{n_{m_n}}.t_n)
\end{aligned}$$

Satz A.4.8 macht Aussagen über die Erhaltung der Stetigkeit bei Funktionskomposition mit der Projektion. Die Existenz des kleinsten Fixpunktes ist somit gesichert. Sein Wert ist $\mu\Phi_{v^*}(P) = \sqcup\{(\Phi_{v^*}(P))^i(\perp, \ldots, \perp) | i \geq 1\}$.

4.3 Strukturierte und höhere Datentypen

Einen bedeutenden Teil einer Spezifikations- oder Programmiersprache bildet das *Typsystem*. Im Kontext von Spezifikationssprachen wird auch vom *Sortensystem* gesprochen. Da hier die Implementierungsaspekte im Ansatz relevant sind und auch die im folgenden betrachteten Konzepte im wesentlichen aus dem Bereich der Programmierung stammen, wird der Begriff 'Typsystem' verwendet.

Bisher wurden nur primitive Datentypen wie *Integer* oder *Boolean* betrachtet. In diesem Abschnitt werden als Erweiterungen des Typsystems der Datentypkonstruktor *Menge* und *Objekttypen* vorgestellt. Abschließend werden *polymorphe Typen* diskutiert.

Es sollen im folgenden Mechanismen bereitgestellt werden, die die Definition eines komplexen Typsystems für eine Programmier- oder Spezifikationssprache vereinfachen. Auf Subtypbeziehungen oder Inklusionspolymorphismus wird in diesem Abschnitt nicht eingegangen. Es werden später Subtypbeziehungen auf Spezifikationsebene betrachtet (Abschnitt 10.1).

4.3.1 Typkonstruktoren

In diesem Abschnitt wird kein vollständiger Satz von Typkonstruktoren vorgestellt. Lediglich das Vorgehen bei Definition eines Typkonstruktors soll beispielhaft dargelegt werden. *Mengen* sollen als konstruierter Typ herausgegriffen werden. Mengen eignen sich insbesondere, da sie einen weit verbreiteten Datentyp repräsentieren. Andere strukturierte Datentypen, wie etwa Tupel oder Records, werden nicht explizit behandelt. Tupel werden schon benutzt und sind auch bereits formal behandelt (siehe Abschnitt 4.1.1), allerdings nicht explizit als Datentypkonstruktor verfügbar gemacht worden. Die Typisierung von Funktionen wird hier ebenfalls nicht betrachtet (siehe etwa [CW85, DT88, Car91] zur Typisierung von Funktionen oder etwa [Sto77, LS84] zu typisiertem λ-Kalkül und Domain-Theorie).

Es werden nur homogene Mengen, d.h. Mengen mit Elementen des gleichen Typs bzw. der gleichen Sorte, betrachtet, da wir strenge Typisierung annehmen und (zunächst) keine Polymorphie zulassen wollen. Mit Hilfe eines Datentypkonstruktors sollen Mengentypen auf beliebigen Elementtypen konstruierbar werden. Zuerst wird die notwendige Erweiterung der Syntax beschrieben. Dann werden Mengen im Kontext der Anforderungen einer denotationalen Semantik untersucht.

Syntax

Analog zu den primitiven Datentypen, etwa *Boolean* oder *Integer*, die in Kapitel 3.1 beschrieben sind, werden auch die Mengentypen *set(int)* oder *set(bool)* über Signaturen syntaktisch definiert. Es soll hierzu eine parametrisierte Signatur *set* (eine Art Datentypkonstruktor) eingesetzt werden, die wie eine Makrodefinition zu benutzen ist. Parameter ist der Elementtyp *elem* der Mengen. Durch Aktualisierung des Parameters wird ein konkreter Mengentyp instantiiert. Dieser Makromechanismus ist kein Konzept der Sprache, sondern dient zur Vereinfachung der Notation.

sig set[elem] is
extend Type by
    sorts set , elem
    opns ...
end sig

set und *elem* sind Sortenbezeichner für Mengen bzw. deren Elemente. Die Operationen können die üblichen (leere Menge, Vereinigung, Durchschnitt, usw.) enthalten. *Type* steht für die Spezifikation des bisherigen Typsystems, also die Sortenbezeichner *bool*, *int* und *void*, das über *extend_by* (siehe Abschnitt 3.2.4) in allen Signaturen bereitgestellt wird. *Type* soll nun erweitert werden. *type* legt die Sprache der Typausdrücke für *Type* fest.

type	::=	*base_type* | *constr_type*
base_type	::=	*'bool'* | *'int'* | *'void'*
constr_type	::=	*'set(' type ')'*

set wird damit als Typkonstruktor eingeführt, der auf einen beliebigen Typ anwendbar ist.

Mengen in der denotationalen Semantik

Mengen sollen nun als Domains im Kontext der denotationalen Semantik untersucht werden. Zuerst wird eine geeignete Ordnung auf Mengen definiert. Dann werden Eigenschaften bzgl. der Existenz von Suprema und die cpo-Eigenschaft gezeigt.

Die Ordnung soll die wesentlichen Strukturen in einer Wertemenge festlegen. Die Ordnung $a \sqsubseteq b$ soll beschreiben, daß b präziser, definierter ist als a, oder, daß a eine Approximation an b darstellt[7]. In bezug auf Mengen ist eine Teilmenge dann definierter als eine andere Teilmenge, wenn sie stärker die zur Verfügung stehende Wertemenge (z.B. INT_ω oder $BOOL_\omega$) einschränkt, sie also in der Inklusionsbeziehung zur anderen Teilmenge steht:

$$a \sqsubseteq_{set} b \quad gdw. \quad b \subseteq a$$

Beispiel 4.3.1 *Mit obiger Definition der Ordnung $\sqsubseteq_{set}$ gilt somit:*
$$INT \sqsubseteq_{set} \{1,2,3\} \sqsubseteq_{set} \{1,3\} \sqsubseteq_{set} \{1\}.$$

Die Ordnung ist dann relevant, wenn z.B. strukturerhaltende Funktionen, d.h. Funktionen, die die Inklusionsbeziehung erhalten, definiert werden sollen. Eine Umbenennung ist etwa eine solche strukturerhaltende Funktion.

Die Ordnung in der eben vorgestellten Form ist benutzerspezifisch, d.h. sie sollte in dieser Form in einem Basistyp *Menge* nicht definiert werden. Standarddatentypen sollten so gestaltet werden, daß sie die notwendigen Anforderungen einer Fixpunktsemantik erfüllen. Sie sollten also cpos sein. Da flache cpos reichen, sollten Standarddatentypen auch nur diese Ordnung bereitstellen. Die Ausarbeitung einer ausgefeilteren Ordnung bleibt dem Spezifizierer oder Sprachdesigner überlassen. Die folgenden Ausführungen mögen als Beispiel dafür dienen, wie in einem solchen Fall vorzugehen ist.

Beobachtung 4.3.1 *Sei S eine Teilmenge der Potenzmenge $SET(D)$ über einer Basismenge D und sei $\perp_{set} = \omega$ bottom-Element von $SET(D)$. $\sqcup S$ existiert, falls S bzgl. $\sqsubseteq_{set}$ (wie oben) total geordnet ist. Dann ist $\sqcup S$ der Durchschnitt aller $s \in S$.*

Beweis: S sei bzgl. $\sqsubseteq_{set}$ total geordnet, d.h. alle Elemente von S stehen in der Inklusionsbeziehung. Da Ketten S durch die Teilmengenbeziehung bestimmt sind, ergibt sich das Supremum von S aus dem Durchschnitt aller Elemente von S. □

Beispiel 4.3.2 *(Suprema und Mengen)*
(1) $S = \{\{1\},\{1,2\},\{1,3\}\}$.*Damit ist* $\sqcup S = \{1\}$. *Das Supremum kann also existieren, auch wenn S keine Kette im Sinne von $\sqsubseteq_{set}$ ist.*
(2) $S = \{\{1\},\{2\},\{1,2\}\}$. *Hier ist S nicht total geordnet und das Supremum existiert nicht innerhalb von S (hier ist $\sqcup S = \emptyset$).*

$\emptyset$ ist Element der Potenzmenge, somit ist auch $s = \emptyset$ für $s \in S$ möglich.

Beobachtung 4.3.2 *Falls $(D, \sqsubseteq_d)$ eine cpo ist, dann ist auch $(SET(D), \sqsubseteq_{set})$ eine cpo.*

[7] Dana Scott benutzt dieses Argument als Erklärung für $\sqsubseteq$ (vgl. [Sto77]).

Beweis: Die Existenz des bottom-Elementes $\perp_{set}$ ist durch $\omega \in SET(D)$ gegeben. Für jede Kette S in $SET(D)$ existiert das Supremum $\sqcup S$. Dies ist in Beobachtung 4.3.1 gezeigt. □

Der Mengenkonstruktor kann auf alle definierten Typen (nicht nur Basistypen) angewendet werden, sofern diese cpos bilden.

Das Universum der Werte VAL ist der zur Verfügung stehende Wertevorrat. VAL wird benutzt, um durch die Umgebung $STATE$ die Bindung von Bezeichnern an Werte zu definieren. Bisher war $(VAL, \sqsubseteq_{VAL})$ eine flache cpo für Werte primitiver Typen. Die cpo-Eigenschaft von VAL in der Erweiterung um Mengen ist zu zeigen.

Satz 4.3.1 *$(VAL_{set}, \sqsubseteq_{VAL_{set}})$ ist cpo, falls $VAL_{set} := \{x | x : type \vee x = \omega\}$, wobei type ein Basistyp (int, bool, void) oder ein mit set konstruierter Typ ist, und für die Ordnung gilt:*

$$(a \sqsubseteq_{VAL_{set}} b \ \ gdw. \ \ a \sqsubseteq b \text{ in einem der Domains}) \ \ und \ \ (\forall c \in VAL_{set} \ . \ \omega \sqsubseteq_{VAL_{set}} c)$$

Beweis: $\omega \in VAL_{set}$, d.h. das kleinste Element existiert. Nichtprimitive Ketten S entstehen nur für Mengeninklusion innerhalb eines Domains $SET(D)$ (alle anderen sind Basistypen und daher flach). Die Existenz der Suprema für diese Ketten wurde schon gezeigt. Die einzelnen Domains sind paarweise disjunkt, so daß zwei Domain-verschiedene Elemente nicht bzgl. $\sqsubseteq_{VAL_{set}}$ vergleichbar sind. □

VAL_{set} ist jetzt nicht mehr flach. Es könnte auch eine flache Ordnung auf den Mengen vorgegeben werden. Die Spezifikation einer inklusionsbasierten Ordnung wie der hier vorgestellten oder einer anderen Ordnung bliebe dann dem Spezifizierer überlassen.

4.3.2 Objektwertige Zustandskomponenten

In diesem Kapitel werden objektwertige Zustandskomponenten eingeführt. Hier waren bisher nur einfache Werte als Belegung einer Zustandsvariablen zugelassen. Da kein Importmechanismus vorhanden ist, soll angenommen werden, daß die Einschachtelung von lokalen Objektspezifikationen zusätzlich zu lokal verschachtelten Prozeduren möglich ist, um die Eigenschaften einer objektwertigen Zustandskomponente beschreiben zu können.

Es ergeben sich zwei neue Aspekte. Zum einen können Objekthierarchien aufgebaut werden, d.h. der Zustand eines komplexen Objekts wird auf mehrere (Sub)objekte verteilt. Zum anderen müssen auch Objekte typisiert werden, d.h. ihnen muß ein Sorten- oder Typbezeichner zugeordnet werden, um sie in der Spezifikation der Zustandskomponenten benutzbar zu machen.

Konzeptionelles Modell

Bisher konnten primitive und mit den Erweiterungen aus Abschnitt 4.3.1 auch strukturierte Werte als Komponenten eines Zustandes zugelassen werden; es sollen nun auch Objekte in den Zustandskomponenten erlaubt werden, d.h. an die Zustandsbezeichner gebunden werden. Da jedes Objekt einen eigenen Zustand haben kann, wird bei der Benutzung von Subobjekten der Gesamtzustand eines zusammengesetzten Objektes auch auf die Komponenten verteilt. In diesen soll er verkapselt sein, d.h. nur über die Operationen des Objekts zugreifbar.

Beispiel 4.3.3 *Ein Spezifikationsbeispiel, das dieses Konzept nutzt, ist das folgende:*

```
sig COMPOUND is                    – Hauptobjekt
   sorts       bool, int, LOCAL
   state_def   comp1 : int
               comp2 : LOCAL
   opns        add : state × int × int → state × int
               pred : state × LOCAL → state × bool

   sig LOCAL is                    – Spezifikation des Typs einer Komponente
               sorts       bool, int
               state_def   local_comp : int
               opns        op : state × int → state × int
                           ...
   end sig (* LOCAL *)
end sig (* COMPOUND *)
```

Das spezifizierte Hauptobjekt $COMPOUND$ hat zwei Zustandskomponenten, von denen eine einfachen Typs ist ($comp1 : int$) und die andere objektwertigen Typs ist ($comp2 : LOCAL$). Der Name der Signatur $LOCAL$ wird hier als Sortenbezeichner benutzt. In der Regel würden lokale Spezifikationen über eine Importschnittstelle importiert werden; auf diese Konstruktion soll aber verzichtet werden. Hier wird von der Möglichkeit Gebrauch gemacht, auch Spezifikationen lokal zu definieren.

Das Beispiel ist in Abbildung 4.1 graphisch dargestellt. A ist ein zur Signatur $COMPOUND$ gehörendes Objekt. B ist $LOCAL$-Objekt. B ist somit Element der Trägermenge A_{LOCAL} des $COMPOUND$-Objekts A.

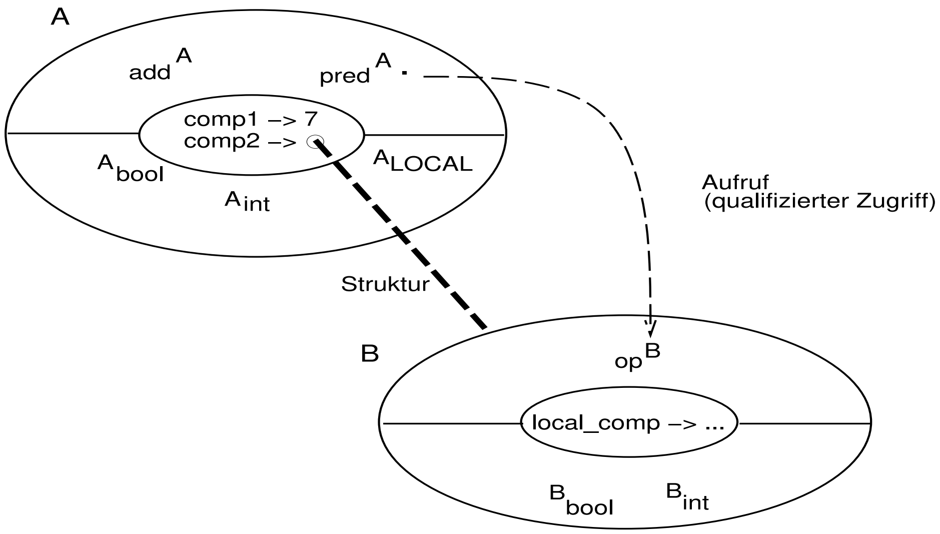


Abbildung 4.1: Zusammengesetztes Objekt

Typisierung

Zustandskomponenten eines Objekts sind typisiert, d.h. ihnen werden Sortennamen zugeordnet. Zur Typisierung einer objektwertigen Zustandskomponente soll deren Objektsignatur herangezogen werden. Typausdrücke lassen sich wie folgt beschreiben[8]:

type	::=	*base_type* | *signature*
base_type	::=	*'bool'* | *'int'* | *'void'*
signature	::=	siehe Abschnitt 3.2

Namen von Objektsignaturen sollen eindeutig sein. Damit ist Namensäquivalenz bezüglich der Typen realisiert. Die typkorrekte Benutzung des Objekts wird durch die definierten Operationen der lokalen Objektsignatur festgelegt. Der Name der Objektsignatur wird als Sortenbezeichner benutzt. Das Typsystem ist eindeutig, d.h. jedem Objekt wird nur ein Typ zugeordnet.

Der Zugriff auf Subobjekte erfolgt ausschließlich über deren Operationen. Syntaktisch wird qualifizierter Zugriff $z_i.op(par_1, \ldots, par_n)$ benutzt. z_i ist der Bezeichner der Komponente mit Deklaration $z_i : \Sigma'$. op ist eine der Operationen, die in der Signatur Σ' definiert sind. Diese Operation sollte n Parameter haben. Objekte sind als Parameter von Operationen zugelassen. Die Benutzung ist dann korrekt, wenn op vom benutzten Objekt zur Verfügung gestellt wird.

Objekte als Werte

Nun soll die Frage geklärt werden, ob Objekte, die Werte einer Zustandskomponente sind, durch Objektbezeichner verwaltet werden sollen (und damit eine Referenzsemantik eingeführt werden kann), oder ob sie als Werte betrachtet werden können.

In Abbildung 4.1 wird durch die Struktur-Beziehung das Subobjekt B direkt an den Zustandsbezeichner $comp2$ gebunden. Der Wert der Zustandskomponente ist also das Objekt selbst. Alternativ könnte an einen Zustandsbezeichner auch ein eindeutiger Objektbezeichner gebunden werden, der mit dem Subobjekt assoziiert ist.

Sei $\Sigma = < S, Z, OP >$ eine Signatur mit $z \in Z$. Zustandsbezeichner z werden bisher (d.h. für primitive Werte) gemäß der Definition

$$v^*(STATE, z) := STATE(z)$$

mit $STATE(z) \in A_s$ für $z : s$ ausgewertet. A_s sei eine Menge atomarer Werte; auf dieser Menge kann eine flache Halbordnung definiert werden, die cpo ist. Die cpo-Eigenschaft erhält sich auch für die Vereinigung aller Trägermengen VAL. Es gibt ein gemeinsames bottom-Element.

Sei nun z eine objektwertige Zustandskomponente, d.h. $z : \Sigma'$ wobei Σ' eine beliebige Objektsignatur ist. Die möglichen Werte, die an z gebunden werden können, sollen nun untersucht werden. Dabei soll zuerst die direkte Bindung von Objekten als Werte an Zustandsbezeichner untersucht werden. Eindeutige Objektbezeichner (Objektreferenzen) werden im nächsten Abschnitt untersucht. Zur Betrachtung einer *Wertsemantik* für Objekte bieten sich zwei Lösungswege an:

[8] Typkonstruktoren wie in Abschnitt 4.3.1 sollen an dieser Stelle nicht betrachtet werden.

1. Projektionen eines Σ-Objekts auf Zustandsalgebren (Definition 3.3.3), die den *Gesamtzustand* eines Objekts beschreiben, bilden die Werte eines Typs. Dieser Ansatz ist vergleichbar mit dem aus der Literatur bekannten *states_as_algebras*-Konzept [PPP94, FJ92, Gur93].

2. Der im ersten Punkt angesprochene Ansatz wird vereinfacht, indem nur auf die veränderlichen Teile der Σ-Objekte, also den *Kernzustand*, projiziert wird. Durch diese Projektionen ergeben sich die Zustandsabbildungen $STATE$ (vgl. Diskussion in den Abschnitten 3.3.2 und 3.6).

Der erste Ansatz ist zu wählen, wenn (z.B. wie bei evolvierenden Algebren) Operationsdefinitionen veränderlich sind. Da dies hier aber keine Grundannahme ist, kann auch der zweite Ansatz betrachtet werden. Dieser soll zuerst untersucht werden.

$$v^*(STATE, z) := STATE(z) \;\; mit \; STATE(z) \in State, \;\; falls \; z \; objektwertig$$

z kann durch eine Zustandsabbildung $STATE \in State$ interpretiert werden, die die aktuellen Belegungen für das durch z bezeichnete Subobjekt angibt (d.h. Belegungen für die lokalen Zustandsvariablen aus Σ'). Es konnte schon gezeigt werden, daß die Menge der Zustandsabbildungen $State$ eine cpo ist (Lemma 4.1.3). Damit ist der Nachweis der cpo-Eigenschaft für die Erweiterung von VAL einfach.

Folgerung 4.3.1

$$VAL_{state} := \bigcup_{s \in S} A_s \cup State$$

$(VAL_{state}, \sqsubseteq_{VAL_{state}})$ ist cpo. Die Ordnung auf VAL_{state} ist weiterhin für zwei Werte a, b definiert durch

$$a \sqsubseteq_{VAL_{state}} b \quad gdw. \quad a \sqsubseteq b \; in \; einem \; der \; Domains.$$

Beweis: Folgt aus der cpo-Eigenschaft beider Operanden der Mengenvereinigung. Die Trägermengen $A_s, s \in S$ sind flach, $State$ ist cpo nach Lemma 4.1.3. □

Auch wenn Zustandsalgebren als Werte eines objektwertigen Typs dienen, kann die cpo-Eigenschaft erhalten werden, wenn die Menge der Zustandsalgebren als flache Halbordnung angenommen wird. Sei A das aktuelle (beliebig gewählte) Σ'-Objekt. ZA^A ist die Menge der Zustandsalgebren von A (vgl. Abschnitt 3.3.2).

$$v^*(STATE, z) := STATE(z) \;\; mit \; STATE(z) \in ZA^A$$

Für die Menge der Zustandsalgebren wird eine flache Halbordnung mit bottom-Element $\perp_{ZA}$ angenommen, die die cpo-Eigenschaft erfüllt. $STATE$ kann hier über eine Projektion aus einer Zustandsalgebra extrahiert werden. Der cpo-Nachweis für die VAL-Erweiterung um Zustandsalgebren soll nun erfolgen.

Folgerung 4.3.2

$$VAL_{ZA} := \bigcup_{s \in S} A_s \cup ZA^A$$

$(VAL_{ZA}, \sqsubseteq_{VAL_{ZA}})$ ist cpo. Die Ordnung sei wie oben definiert.

Beweis: Folgt aus der cpo-Eigenschaft beider Operanden der Mengenvereinigung. □

Objektreferenzen

Der Wert einer objektwertigen Zustandkomponente ist durch die Konstruktion Teil eines Σ-Objekts, unabhängig davon, ob er als Zustandsalgebra oder über die Zustandsabbildung $STATE$ realisiert ist. Der Wert stellt nur eine Momentaufnahme dar, eine Projektion auf den aktuellen Zustand, in dem sich das Subobjekt gerade befindet. Dieses Subobjekt existiert jedoch unabhängig von dem Wert, der in der Zustandskomponente des Superobjekts gerade verwaltet wird. Die Beziehung zwischen der Komponente und dem sich dahinter verbergenden Σ-Objekt kann durch eine Referenz repräsentiert werden. Ein Objektbezeichner-Konzept, welches jedes Σ-Objekt mit einem eindeutigen Objektbezeichner (Objektreferenz) assoziiert, ist dazu notwendig. Durch eine Abbildung — ähnlich $STATE$ — kann der Objektbezeichner immer an den assoziierten Wert (das zugehörige Σ-Objekt) gebunden werden.

Die angesprochenen Objektbezeichner sollen nun kurz formal definiert werden. OID sei eine Menge von Objektbezeichnern, paarweise disjunkt zu allen anderen Wertemengen. OID_ω sei die Erweiterung um ein undefiniertes Element ω. Mit $\omega = \bot_{OID}$ bildet $(OID_\omega, \sqsubseteq_{OID})$ eine flache cpo. Bei Parameterübergabe mit Referenzsemantik werden die Objektbezeichner übergeben.

$$v^*(STATE, z) := STATE(z) \;\; mit \; STATE(z) \in OID,$$

wobei es eine eineindeutige Abbildung

$$OBJ\_REF : OID \rightarrow \{O \mid O \in Obj(\Sigma) \; für \; alle \; \Sigma \in Sign^9\}$$

gibt, die Objektbezeichner mit Objekten assoziiert. OBJ_REF ist stetig, da $(OID_\omega, \sqsubseteq_{OID})$ eine cpo bildet und für die Klasse der Objekte Obj eine flache cpo angenommen werden kann. Somit kann auch die cpo-Eigenschaft für die VAL-Erweiterung gezeigt werden.

Folgerung 4.3.3

$$VAL_{OID} := \{A_s \mid s \; ist \; primitive \; Sorte\} \cup OID$$

$(VAL_{OID}, \sqsubseteq_{VAL_{OID}})$ ist cpo. Die Ordnung sei wie oben definiert.

Beweis: Folgt aus der cpo-Eigenschaft beider Operanden der Mengenvereinigung. □

Eine objektwertige Komponente wird mit dem undefinierten Wert ω (dem bottom-Element der Halbordnung) initialisiert. Eine *create*-Operation oder explizites Initialisieren bei Objekterzeugung ist nicht vorgesehen, um den Ansatz einfach zu halten.

Mit der Konstruktion über Objektbezeichner ist auch die Modellierung von *geteilten Objekten* (englisch *shared objects*) möglich. Ein Objekt kann dann Subobjekt mehrerer anderer Objekte sein, d.h. von diesen gemeinsam benutzt werden. OBJ_REF ist dann nicht mehr eineindeutig. Da wir uns auf sequentielle, deterministische Sprachen beschränken, kann es nicht zu Konsistenz- oder Synchronisationsproblemen kommen.

[9] $Sign$ ist die Klasse aller Signaturen, $Obj(\Sigma)$ ist die Klasse aller Σ-Objekte.

4.3.3 Polymorphie

Einführung

Polymorphie ist ein Konzept, das die Flexibilität und die Ausdrucksfähigkeit von Sprachen erweitert [CW85, DT88]. Polymorphie ist eine Eigenschaft des Typsystems. Unter einem polymorphen Wert oder einer polymorphen Variable versteht man die Eigenschaft des Wertes oder der Variablen, mehrere Typen zu haben. Setzt man diese Definition auf Funktionen um, so sind die Funktionen polymorph, deren Parameter polymorph sind.

Eine Ausprägung der Polymorphie wird durch den Mechanismus der Typparametrisierung erreicht. In einem solchen *parametrischen Polymorphismus* arbeitet eine Funktion gleichförmig auf einer Menge von Typen für die Parameter. Diese Menge ist in der Regel unendlich groß, allerdings haben die einzelnen Typen eine ähnliche Struktur. Die Typparametrisierung muß nicht explizit erfolgen. Zusammen mit dem *Inklusionspolymorphismus* (Polymorphismus auf einem Subtypbegriff) bildet der parametrische Polymorphismus den *universellen Polymorphismus*.

Eine einfachere Form des Polymorphismus wird mit *ad-hoc Polymorphismus* bezeichnet. *Überladung* zählt dazu. Eine überladene Funktion arbeitet auf einer endlichen Menge von Typen, die allerdings verschiedener Struktur sein können. Somit stehen ad-hoc polymorphe Funktionen eigentlich für eine Menge von monomorphen Funktionen. Eine parametrisch polymorphe Funktion benutzt immer den gleichen Code unabhängig vom Typparameter. Überladene Funktionen dürfen verschiedenen Code verwenden.

Ein Konzept, das in seinem Ziel, Code für mehrere Typen wiederzuverwenden, dem Polymorphismus ähnlich ist, ist *Generizität* (siehe [Mey88]). Generische Programmeinheiten, wie Module oder Funktionen, sind typparametrisierte Programmeinheiten. Generizität ist ein Konzept, das in der Regel syntaktisch eingesetzt wird (siehe z.B. *Packages* in Ada). Die Instantiierung der Typparameter erfolgt in der Regel spätestens zur Übersetzungszeit. Die Semantik eines generischen Konstrukts ist Makroexpansion. Damit ist z.B. eine generische Funktion nichts anderes als eine Abkürzung für eine Menge monomorpher Funktionen. In echten polymorphen Systemen wird Code nur einmal für eine generische Funktion generiert.

Im Rest dieses Abschnitts wird nun eine formale Erweiterung des bisherigen Typansatzes vorgestellt, die Generizität und Polymorphismus als zwei Sichten beschreibt. Dabei wird Generizität als statische, modellierungsorientierte Sicht und Polymorphismus als dynamische, ausführungsorientierte Sicht behandelt. Diese beiden Sichten sind zwei alternative Formen einer polymorphen Typsemantik. Bei Definition einer imperativen Sprache kann die geeignetere Form ausgewählt werden.

Typausdrücke und cpo-Eigenschaft

Zur Beschreibung und Definition von polymorphen Typen wird die Sprache der Typausdrücke erweitert.

$$\begin{array}{lll}
type & ::= & base_type \mid constr_type \mid tvar \\
base_type & ::= & \text{'int'} \mid \text{'bool'} \mid \text{'void'} \\
constr_type & ::= & \text{'univ'} \mid \text{'set('}\ type\ \text{')'} \mid \text{'tuple('}\ type\ \text{')'} \\
tvar & ::= & id
\end{array}$$

Mit *Type* sei die Menge aller Typen *type* bezeichnet. Das vorhandene Typsystem der nicht-polymorphen Typen wird um den Typ *univ* erweitert. Die Wertemenge von *univ* ist die Menge aller Werte VAL (VAL wie in den Abschnitten 4.3.1 oder 4.3.2 definiert). *univ* wird als konstruierter Typ definiert, da *constr_type* die polymorphen Typen umfassen soll. Mengen und Tupel können nun homogen oder auch beliebig inhomogen definiert werden. So ist *set*(*int*) homogen und *set*(*univ*) heterogen. Außerdem ist in der Typsprache die Benutzung von Typvariablen möglich. Generische Konstrukte, d.h. Ausdrücke, die Typvariablen enthalten, können *instantiiert* werden. Typvariablen *tvar* lassen sich durch den sie definierenden Typausdruck *texpr* ersetzen. Die Instantiierung kann partiell sein, d.h. *texpr* kann weitere Typvariablen enthalten. Ein nicht weiter instantiierbarer Typ heißt *Grundtyp*. Eine in einem Typausdruck benutzte Typvariable muß nicht durch einen Typausdruck definiert bzw. vollständig instantiiert sein. Ein solcher Ausdruck heißt *polymorph*. Eine Prozedur oder ein Modul, das darauf basiert, heißt *generisch*. Rekursive Typen sollen nicht erlaubt sein, d.h. die Typvariable, die den rekursiven Typen definiert, darf im Typausdruck nicht vorkommen. Die cpo-Eigenschaft wurde schon für alle primitiven Typen, für Mengen und Tupel sowie objektwertige Typen und auch für die Menge aller Werte VAL gezeigt. Also ist die Halbordnung zu jedem (instantiierten) Typausdruck immer eine cpo.

Die einzelnen Konstrukte der Benutzungsschnittstelle der Σ-Objekte sollen nun hinsichtlich der Auswirkungen von Polymorphismus und Generizität auf sie untersucht werden. Für jedes zu behandelnde Sprachkonstrukt (*Sorte, Variable, Operation*) wird zuerst die polymorphe Sicht, dann die generische Sicht definiert. Die polymorphe Sicht orientiert sich an der dynamischen Semantik der Sprache. Sie wird durch Elemente der Σ-Objekte definiert. Die generische Sicht wird auf einer Metaebene definiert. Sie beschreibt Konstruktionen auf syntaktischen Beschreibungen, wie Signaturen und Spezifikationen. Es wird hier auf einer Menge abstrakter Typen argumentiert. Ideen dieser Polymorphiebetrachtung sind [Krä91a] entnommen.

Bezeichne $\mathcal{M}$ im folgenden die Semantikfunktion, die syntaktischen Konstrukten ihre Bedeutung zuweist ($\mathcal{M}_P$ für die polymorphe Sicht und $\mathcal{M}_G$ für die generische Sicht). Stetigkeit und cpo-Eigenschaft sind Forderungen, die für die Wohldefiniertheit der dynamischen Semantik der Sprache erforderlich sind. Dies ist bezüglich der polymorphen Sicht zu betrachten. Hierfür werden die Eigenschaften in den einzelnen Abschnitten gezeigt.

Sorten

Bisher wurden Sorten $s \in S$ durch Trägermengen primitiver Werte interpretiert, d.h. $\mathcal{M}(s) := A_s$. Die nun möglichen Typkonstruktoren müssen integriert werden. T sei im folgenden ein Typkonstruktor, s eine beliebige einfache oder konstruierte Sorte.

Polymorphe Sicht

$\mathcal{M}_P : T(s) \mapsto A_{T(s)},\ mit\ \mathcal{M}_P : Type \to A$, wobei $A := \{A_s \mid s \in S^* \cup \{state\}\ \}$ der Domain der Trägermengen mit $S^* := S \cup \{T(s) \mid T\ Typkonstruktor, s \in S^*\}$ ist. T bezeichnet eine Funktion auf Trägermengen, die die Trägermenge A_s zum Argument s auf eine Trägermenge zusammengesetzter Werte $A_{T(s)}$ abbildet. $\mathcal{P}$ bezeichne im folgenden die Potenzmengenbildung und Π die Bildung des kartesischen Produkts ($\Pi_1^n D := D \times \ldots \times D$, n-mal).

Für die Basistypen muß die Semantikfunktion $\mathcal{M}_P$ definiert werden: $\mathcal{M}_P(int) := \mathcal{Z}, \mathcal{M}_P(bool)$ $:= \{true, false\}$, $\mathcal{M}_P(void) := \{null\}$, $\mathcal{M}_P(univ) := VAL$ und $\mathcal{M}_P(tvar) := VAL$. $\mathcal{M}_P$ wird für beliebige Typausdrücke induktiv definiert.

Seien *set* und *tuple* die zugelassenen Typkonstruktoren. Dann wird folgendes definiert:

$$\mathcal{M}_P(set(s)) := \mathcal{P}(A_s),\ falls\ \mathcal{M}_P(s) = A_s$$

Also wird die Anwendung des Typkonstruktors *set* auf eine Sorte s durch $\mathcal{M}_P$ auf die Potenzmenge der Trägermenge von s abgebildet ($A_s \mapsto \mathcal{P}(A_s)$).

$$\mathcal{M}_P(tuple(s)) := \Pi A_s, \quad falls\ \mathcal{M}_P(s) = A_s$$

Die Anwendung von *tuple* wird auf die freie Halbgruppe der Trägermenge der Elemente abgebildet ($A_s \mapsto \Pi A_s$ mit $\Pi = \bigcup_{n \in Nat} \Pi_1^n$, vgl. Lemma 4.1.1).

Folgerung 4.3.4 *$(\mathcal{P}(A_s), \sqsubseteq)$ und $(\Pi A_s, \sqsubseteq)$ sind cpos, falls $(A_s, \sqsubseteq)$ eine cpo ist. Es gilt also:*

- *Alle Basistypen sind cpos.*
- *Falls die Argumenttypen cpos sind, erhalten die Typkonstruktoren set und tuple die cpo-Eigenschaft.*

Beweis: Siehe Abschnitt 3.1 sowie die Sätze 4.3.1 und 4.3.2 ($(\mathcal{P}(A_s), \sqsubseteq)$ ist cpo) und Satz A.4.3 ($(\Pi A_s, \sqsubseteq)$ ist cpo). □

Die Semantik $\mathcal{M}_P$ der Typkonstruktoren für die polymorphe Sicht basiert also auf Funktionen auf Trägermengen.

Generische Sicht

$\mathcal{M}_G(T(s)) \subseteq Type$, wobei $Type$ die Menge aller Typen ist. $\mathcal{M}_G : Type \to \mathcal{P}(Type)$ bezeichnet eine Funktion auf der Menge der Typen. $\mathcal{M}_G$ soll induktiv definiert werden. Seien die Basistypen wie folgt definiert: $\mathcal{M}_G(int) := \{int\}$, $\mathcal{M}_G(bool) := \{bool\}$, $\mathcal{M}_G(void) := \{void\}$, $\mathcal{M}_G(univ) := \{univ\}$ und $\mathcal{M}_G(tvar) := Type$.

Seien *set* und *tuple* die zugelassenen Typkonstruktoren. Dann wird folgendes induktiv definiert:

$$\mathcal{M}_G(set(s)) := \{set(s') \mid s' \in \mathcal{M}_G(s)\}$$
$$\mathcal{M}_G(tuple(s)) := \{tuple(s') \mid s' \in \mathcal{M}_G(s)\}$$

$Type$ ist abgeschlossen bzgl. *set*- und *tuple*-Anwendungen. $Type$ ist unendlich (aber aufzählbar). Die Semantik $\mathcal{M}_G$ der Typkonstruktoren für die generische Sicht basiert also auf Funktionen auf der Menge der Typen.

Variablen

(Werte-)Variablen id sind in Form von Zustandsbezeichnern oder Parametern von Operationen möglich. Sie werden in der Form $id : texpr$ deklariert. id wird in $STATE$ verwaltet. Falls $x : s$ deklariert wurde, sind Werte aus $\mathcal{M}(s) = A_s$ in der Bindung an x durch $STATE$ erlaubt. Da $STATE$ auf VAL arbeitet, somit ohnehin alle Werte binden kann, sind hier zur Behandlung von Variablen in einem polymorphen Ansatz keine neuen Definitionen vorzunehmen. Eine Variable hat, je nach Sicht, einen polymorphen Typ (beschrieben durch einen parametrisierten Typausdruck) oder eine Menge von Typen (generische Sicht). Also ist:

$$\mathcal{M}(id) := \mathcal{M}(texpr)$$

Die Semantik einer Variablen ist durch die Semantik des sie definierenden Typausdrucks gegeben. Für *Typvariablen*, die neben den üblichen (Werte-)Variablen hier im polymorphen Ansatz auch möglich sind, gilt die gleiche Definition:

$$\mathcal{M}(tvar) := \mathcal{M}(texpr)$$

Gegebenenfalls muß für *texpr* bei impliziter Instantiierung *univ* gesetzt werden.

Operationen

Bisher wurde zu jeder Operation

$$op : state \times s_1 \times \ldots \times s_n \rightarrow state \times s_0$$

eine Funktion

$$op^* : State \times A_{s_1} \times \ldots \times A_{s_n} \rightarrow State \times A_{s_0}$$

als Semantik definiert. Sei jetzt eine der Sorten s_i polymorph, d.h. abhängig von einer Typvariablen t.

Polymorphe Sicht

Für jedes s_i ist eine Trägermenge $\mathcal{M}_P(s_i)$ als Semantik definiert, wobei für t der allgemeinste Typ *univ* gesetzt wird, sofern nichts anderes bekannt ist (d.h. sofern t nicht instantiiert ist). $(\mathcal{M}_P(s_i), \sqsubseteq)$ ist, wie gezeigt, eine cpo. Die Operationsdefinition wird weiterhin durch eine Funktion wie in der ursprünglichen Definition interpretiert. Hier zeigt sich die echte Polymorphie. Es wird nur einmal Code für eine polymorphe Funktion '*erzeugt*'. Der Code ist die Interpretation $v^*(STATE, op)$ für eine Operation *op*. Also gilt:

$$\mathcal{M}_P(op) := v^*(STATE, op)$$

Generische Sicht

Die s_i der Operationsvereinbarung bezeichnen in der generischen Sicht jeweils eine Menge von Typen bzw. Sorten. Da Trägermengen nicht direkt zur Verfügung stehen, kann auch keine Funktion in der semantischen Struktur als Interpretation angegeben werden. Die Interpretationen für Operationen, wie sie in diesem Kapitel angegeben wurden, definieren die dynamische Semantik. Generizität wurde hier auf syntaktischer Ebene, also statisch eingeführt. Interpretation einer Operation auf polymorphen Sorten soll deshalb hier die Menge der Operationen sein, für die Funktionen in der semantischen Struktur angegeben werden können. Dies ist allerdings nur möglich, wenn die polymorphen Sortenausdrücke zu Grundtypen instantiiert sind, nur dann existieren die konkreten Trägermengen. Nicht explizit instantiierte Typvariablen können aber mit *univ* instantiiert werden.

$$\mathcal{M}_G(op) := \{\ op^* \mid op^* = v^*(STATE, op)\ \textit{für beliebige Instantiierungen der Typvariablen } t\ \}$$

Es ist mit allen möglichen Grundtypen zu instantiieren (es werden konkrete Typen gebraucht, um die Semantik $\mathcal{M}_G$ zu bestimmen).

Abschlußbetrachtung Polymorphie

Dieser Polymorphieansatz ist nicht vollständig. So wurden die vorgestellten objektwertigen Typen aus Abschnitt 4.3.2 nicht betrachtet. Da auf ihnen cpos kanonisch definiert werden können, lassen sie sich aber wie die primitiven Typen behandeln.

In der Literatur finden sich polymorphe Ansätze in verschiedenen Sprachklassen. In algebraischen Spezifikationssprachen finden sich unterschiedliche Ausprägungen. In der Regel wird Generizität oder Polymorphie[10] durch parametrisierte Spezifikationen erreicht (siehe [Wir90] oder [EM90]). In der algebraischen Spezifikationssprache *SEGRAS* [Krä91b, Krä91a] wird eine Form von Polymorphie realisiert, deren Ideen der hier vorgestellten generischen Sicht zugrundeliegen. Polymorphe Typen werden dort durch die Menge der Grundtypen interpretiert. In vielen Programmiersprachen sind Polymorphieansätze realisiert. Bedeutende Vertreter sind ML [MTH90] im Bereich der funktionalen Sprachen und Eiffel [Mey92b] für die objektorientierten. In den objektorientierten Sprachen wird allerdings eine andere Form einer immer noch universellen (echten) Polymorphie realisiert. Es liegt dort nicht Typparametrisierung, sondern die Subtypbeziehung zugrunde. [CW85] bezeichnen diese Form als *Inklusionspolymorphie*. Eiffel realisiert eine Mischung aus Generizität für Module bzw. Klassen und Inklusionspolymorphie. Objektorientierung und auf Subtypen basierende Inklusionspolymorphie werden in Kapitel 10 betrachtet. Um ähnliches hier auf Objektebene zu realisieren, könnte der Ansatz der *order-sorted algebras* aufgegriffen werden. Dieser wurde schon vielfach benutzt, um Objektorientierung in algebraischen Spezifikationsansätzen zu beschreiben [Bre91, GM87a, AAZ93, PPP94]. Auch dieser Aspekt wird in Kapitel 10 betrachtet.

4.4 Definition einer imperativen Sprache

In diesem Abschnitt wird gezeigt, wie der Kern einer imperativen Programmiersprache (*Programmieren im Kleinen*) formal definiert werden kann. Ein Modulkonzept (*Programmieren im Großen*) wird erst in einem späteren Kapitel erarbeitet. Die *dynamische Semantik* der Programmiersprache soll im von Neumann'schen Sinne durch eine formale *abstrakte Maschine* im *denotationalen Stil* definiert werden können. Die vorgestellten Σ-Objekte werden dazu herangezogen. Σ-Objekte sind in diesem Sinne ein benutzerfreundlich gestalteter Ansatz einer denotationalen Semantik. Dieser Grundgedanke lag auch der Definition von VDM zugrunde. Action Semantics [Wat91] basieren ebenfalls auf der Idee der besseren Handhabbarkeit einer Semantik durch Bereitstellung primitiver Sprachkonstrukte.

Die Programmiersprache soll alle für die üblichen imperativen Programmiersprachen charakteristischen *Anweisungsformen* enthalten können, also Zuweisung, bedingte Anweisung, Schleifen, Prozeduraufruf und Anweisungssequenz (vgl. [Hor84]). Die Sprache soll nicht in allen Einzelheiten definiert werden. Es werden mögliche Konzepte in Stichworten vorgestellt, die den Umfang und die Eignung des Berechnungsmodells zur Definition einer Programmiersprache veranschaulichen. Der Kern einer Sprache wie ProSet (vgl. Abschnitt 1.4.2) oder einer anderen imperativen Sprache wie *Modula2* oder *Eiffel* läßt sich damit definieren.

[10] Die Begriffe werden hier anders angewandt, da Ausführung kein Bestandteil eines algebraischen Ansatzes ist.

Blöcke und Verschachtelung

Die Sprache kann blockstrukturiert sein (siehe [ASU88]). Jede Operations- oder Objektspezifikation bildet einen Block. Blöcke können ineinander verschachtelt werden. Blockstruktur und Verschachtelung wurden in Abschnitt 4.1 betrachtet.

Sichtbarkeit

Namen eines Blockes sind in eingeschachtelten Blöcken (Operationsdefinitionen) sichtbar. Diese Regel kann nur durch Verschattung gebrochen werden. Eine Deklaration eines Namens x in einem Block verschattet eine sichtbare Variable x aus einem der umgebenden Blöcke. Das lokale x ist dann auch das in allen tiefer verschachtelten Blöcken sichtbare. Es wird die *most closely nested*-Regel realisiert (siehe [ASU88] Kapitel 7.4). Dies gilt für Variablen und Operations- oder Objektdefinitionen. Es gibt nur einen Namensraum. In Abschnitt 4.1 wurden diese Sichtbarkeitsregeln umgesetzt.

Kontrollstrukturen

Operationen können rekursiv sein (Abschnitt 4.1.2). Als Parameterübergabemechanismen sind die Formen *call_by_value, call_by_result, call_by_value/result* und *call_by_reference* zugelassen, bezeichnet durch die Modi *rd, wr, rw* und *ref*. Parametermechanismen sind in Abschnitt 4.2.2 definiert. Durch die Rekursion sind Programme mit expliziten Schleifen semantisch äquivalent beschreibbar.

Die oben angegebenen Anweisungsformen können realisiert werden. Zuweisung, Prozeduraufruf und Anweisungssequenz sind schon als Primitive der Σ-Objekte vorhanden (Abschnitt 3.5). Die einfache Zuweisung $x := t$ einer Programmiersprache entspricht der $asgn$-Konstruktion der Σ-Objekte $(z := t)(st) := asgn(st, z, t)$ für einen aktuellen Programmzustand st. Die Kommandosequenz kann direkt übernommen werden. An den Beispielen von *if-then-else*-Anweisung und *while*-Schleife soll kurz die Definierbarkeit der dynamischen Semantik noch nicht vorhandener Konstrukte einer imperativen Programmiersprache gezeigt werden. Die übliche *if-then-else*-Anweisung läßt sich durch das bedingte Kommando und die Kommandosequenz definieren. Die *if-then-else*-Anweisung $if\_then\_else(b, s_1, s_2)$ sei durch

$$v^*(STATE, if\_then\_else(b, s_1, s_2)) := v^*(STATE, seq(if(st, b, s_1), if(st, \neg b, s_2)))$$

definiert. $STATE$ ist ein beliebiger Zustand eines Σ-Objekts.

Schleifen lassen sich definieren, wenn eine Semantik für rekursive Operationen existiert (dieses ist in Satz 4.1.1 festgehalten). Dies läßt sich auf Sprachebene ausdrücken. Die *while*-Schleife $while\ e\ do\ S\ od$ sei durch

$$v^*(STATE, while\ e\ do\ S\ od) := v^*(STATE, if\_then\_else(e, seq(S, while\ e\ do\ S\ od), skip))$$

definiert. $STATE$ ist ein beliebiger Zustand eines Σ-Objekts. Die Definition der Schleife ist wohldefiniert, da der Fixpunkt zu der rekursiven Definition immer existiert (folgt aus Satz 4.1.2). Die Funktion v^* ist stetig.

Für einfache imperative Konstrukte stellt die abstrakte Maschine der Σ-Objekte also eine Art abstrakter Syntax mit Semantik bereit.

Typisierung

Die Sprache soll statisch getypt sein. Alle Variablen werden mit Typangabe deklariert. Als Datentypen sind primitive Typen wie *Integer* oder *Boolean* zugelassen (siehe Abschnitt 3.1).

Die typkorrekte Verwendung von Namen ist in dieser Sprache statisch überprüfbar. Strukturierte, objektwertige oder polymorphe Datentypen können auf Basis des vorangegangenen Abschnitts 4.3 ebenfalls eingeführt werden.

Objektspezifikationen

Der Begriff des Objektes wurde in Kapitel 1.1 eingeführt. Objekte werden in Objektspezifikationen beschrieben. Objektspezifikationen sind Beschreibungen abstrakter Datentypen mit Zustand. Jegliche Form des Zugriffs auf ein Objekt kann nur über Operationen erfolgen. Es können von außen alle definierten Operationen benutzt werden. Dieses Konzept ist in Abschnitt 4.3.2 definiert.

Das Berechnungsmodell der Σ-Objekte erlaubt sowohl die unterschiedliche Modellierung von Attributen und Prozeduren als auch deren Verschmelzung in einem Operationsbegriff (siehe Abschnitt 4.2.3). Diese Verschmelzung ist in vielen anderen Ansätzen (etwa Kripke-Modellen, CMSL, COLD) nicht möglich.

Wie schon angedeutet wurde, lassen sich die Sprachkonstrukte einer imperativen Sprache hinsichtlich ihrer dynamischen Semantik direkt auf den Konzepten der Σ-Objekte (Kapitel 3) und ihrer Erweiterungen (Kapitel 4) definieren. Diese Einfachheit liegt an der Ausgestaltung des Berechnungsmodells. Dieses liegt in einem Basismodell und einer Reihe von Erweiterungspaketen vor, die es erlauben, bestimmte Aspekte einer Sprache wie Typisierung, Rekursion oder Verschachtelung problemnah semantisch definieren zu können.

4.5 Zusammenfassung

Im Kontext von zustandsbasierten Systemen wurde ein denotational orientiertes Berechnungsmodell zur Modellierung von Objekten präsentiert. Für imperative Operationsbeschreibungen wurden Funktionen auf Zuständen als Semantik in einem denotationalen Stil definiert. Der Zustand eines Objekts wurde durch eine Formalisierung des Konzeptes eines Laufzeitstacks aus dem Compilerbau semantisch definiert. Σ-Objekte sind die Basiskonstrukte eines Software-Systems. Σ-Objekte verkapseln jeweils einen Teil des Gesamtzustands, auf den von außen nur durch Attribute und Prozeduren zugegriffen werden kann. Dies gilt auch bei verschachtelten Objekten. Dieser Ansatz wird *objektbasiert* genannt. Er entspricht damit dem konzeptionellen Modell eines Software-Systems in Π (vgl. [CFGGR91] Kapitel 2.2).

Das Vorgehen dieses Ansatzes besteht darin, eine mehrsortige Algebra um eine Struktur zu erweitern, die die veränderlichen Daten verwaltet. Variabel in einer Algebra sind die aktuellen Werte der freien Variablen (die in einem Objekt veränderbaren Variablen, in der Literatur auch Instanz- oder Objektvariablen genannt). Alles andere ist durch die Interpretation fest vorgegeben. Diese freien Variablen (im vorliegenden Ansatz sind das der Zustandsbezeichner, sowie die lokalen Variablen und die formalen Parameter von Operationen) werden in der oben angesprochenen Zustandsabbildung $STATE$ verwaltet. In ihr werden Bezeichner an Werte gebunden. Die Veränderung von Bindungen wird als Zustandsübergang bezeichnet. Jede Zustandskomponente ist eine *dynamische Konstante*, d.h. in jedem Zustand unveränderlich, aber in Zustandsübergängen änderbar. Die Zustandsabbildungen in den Σ-Objekten werden als Funktionen behandelt. Ein Zustand kann auch als eine Algebra angesehen werden (*states as algebras*). Oben wurde der Zustand durch den veränderlichen Teil eines Σ-Objektes, die Zustandsabbildung $STATE$, definiert. Dieses kann verallgemeinert werden, indem auch die unveränderlichen Teile (die Operationen) hinzugenommen werden. Ein Zustand ist dann eine

Momentaufnahme des ganzen Objekts. Dieser Schritt ist sogar notwendig, wenn Operationsdefinitionen sich zwischen Zuständen ändern können. Der Zustand wird dann durch eine Algebra, die *Zustandsalgebra*, festgelegt. Ein Zustandsübergang wird durch Kommandos spezifiziert. Kommandos modellieren Funktionen auf den Zuständen. Der durch ein Kommando bestimmte Zustand ist eindeutig, da nur deterministische Programme betrachtet werden. Das Zustandskonzept ist hier inhärent. Bei jeder Aktion, die auf einem Σ-Objekt ausgeführt wird (Auswertung eines Attributes, Ausführung einer Prozedur) ist der Zustand sowohl als Parameter als auch als Bestandteil des Ergebnisses enthalten. Da der Zustand immer benutzt werden muß, kann er auch in einer Spezifikation entfallen. Eine entsprechende *implizite Form* der Aktionen wurde definiert.

Die soeben beschriebene Grundidee der Σ-Objekte wurde in einem Basismodell (Kapitel 3) realisiert. Darauf aufbauend sind mehrere Semantikpakete definiert (Kapitel 4), die es erlauben, bestimmte Sprachaspekte von zustandsbasierten Sprachen problemnah definieren zu können. Hierzu gehören die Aspekte Typisierung, Rekursion, Parameterübergabe sowie lokale Objekte und Operationen. Es wurde gezeigt, wie eine imperative Sprache leicht auf den vorgestellten Konzepten definiert werden kann (Abschnitt 4.4).

Eine Besonderheit des Ansatzes liegt in der Verschmelzbarkeit von Attributaspekten (dem funktionalen Effekt) und Prozeduraspekten (dem Umgebungs- oder Seiteneffekt) in einem Operationsbegriff. Die in der Einleitung vorgestellten Ansätze bieten diese Möglichkeit nicht. Während es in der Spezifikation noch sinnvoll ist, Attribute und Prozeduren zu unterscheiden, ist unter Implementierungsgesichtspunkten eine Verschmelzung sinnvoll oder sogar notwendig, wenn etwa nur eine Form einer prozeduralen Abstraktion in einer Programmiersprache unterstützt wird.

Gegenüber anderen zustandsbasierten Spezifikationsansätzen wurden hier einige Vereinfachungen vorgenommen. Dies soll kurz diskutiert werden.

Die primitiven Kommandos, die für Kripke-Modelle vereinbart sind (hierzu gehören *bewachtes Kommando* (*guarded command*) oder *nichtdeterministische Auswahl* (*nondeterministic choice*)) und die auch in der Regel in Spezifikationsansätzen, die auf dynamischer Logik basieren (wie etwa COLD), verfügbar sind, sind hier verändert worden. Es gibt das abstraktere bedingte Kommando anstatt des bewachten Kommandos. Auf Konzepte des Nichtdeterminismus wurde verzichtet. Die in Betracht gezogene Sprachklasse ist die der sequentiellen, deterministischen imperativen Sprachen. Hierauf ist auch die Auswahl der Basiskommandos ausgelegt. Soll die Sprachklasse erweitert werden, so müssen ggf. auch die Basiskommandos erweitert werden.

Mitunter ist es sinnvoll zu spezifizieren, welche einer u.U. großen Anzahl von zugreifbaren Zustandskomponenten von einer Prozedur verändert wird. Dies kann durch eine Vielzahl von Axiomen geschehen. Wird dann aber der Spezifikation eine neue Zustandskomponente hinzugefügt, so sind ggf. viele Änderungen durchzuführen (jede alte Prozedur ändert die neue Komponente nicht). Dieses Problem — mit dem Begriff *Framing Problem* bezeichnet — wird in COLD umgangen, indem in einer Modifikationsklausel zu einer Prozedur explizit die von ihr geänderten Zustandskomponenten aufgelistet werden können. Falls eine neue Zustandskomponente von einer schon bestehenden funktional abhängt, kann dies in COLD durch eine Abhängigkeitsklausel vermerkt werden. Aus Vereinfachungsgründen wurde hier auf solche Spezifikationsmechanismen verzichtet. Solche Konstruktionen fallen auch eher in den Bereich einer Spezifikationssprache als in den eines Berechnungsmodells. Die notwendigen formalen Mechanismen sind jedoch vorhanden.

Teil II

Spezifikationslogik

Kapitel 5

Eine Spezifikationslogik

5.1 Modale Logik zur Spezifikation von Modulen

Die Beschreibung von Objekten in einem komponentenorientierten Ansatz [Goe93] erfordert die Spezifikation der Software-Komponenten auf semantisch hohem Niveau. Komponenten, die Objekte in ihrer Struktur und in ihrem Verhalten beschreiben, sind *Module* und *Operationen*. Besonders charakterisierende Eigenschaften von Komponenten sind der *Effekt* der Operationen und die *Invarianten* der Module (siehe auch [Mey88]). Eine Logik, die die Formulierung entsprechender Eigenschaften von Modulen erlaubt, soll in diesem Kapitel entwickelt werden.

Mit imperativen Sprachen werden zustandsbasierte Systeme beschrieben. Dazu werden von diesen Sprachen Programmvariablen bereitgestellt und Operationen mit Seiteneffekten ermöglicht. Daraus leiten sich auch spezielle Bedürfnisse an einen Spezifikations- und Verifikationsansatz ab, insbesondere gehören dazu Vor- und Nachbedingungen und Invarianten als Spezifikationsmittel. Entsprechende Spezifikationen sind ein adäquaterer, problemnäherer Ansatz zur Beschreibung zustandsbasierter Systeme als algebraische Spezifikationen. Letztere machen einen Zustandsbegriff nicht explizit.

Der hier vorgestellte Ansatz soll die Spezifikation von Datentypen im Sinne algebraischer Spezifikationen und die Spezifikation von zustandsbasierten Systemen vereinen. Es wird in diesem Kapitel eine dynamische Logik über der Struktur der Σ-Objekte in zwei Stufen entwickelt. In der ersten Stufe erfolgt die Definition im Umfang einer üblichen Programmlogik (vgl. Kapitel 2), dann wird die Logik zu einem vollen modalen Umfang erweitert. Das Kapitel schließt mit Vergleichen zu Algebren, Kripke-Modellen, beobachtungsorientierter Spezifikation und temporaler Logik.

Was soll beschrieben werden?

Wie schon in vorangegangenen Kapiteln erläutert, gibt es zwischen Attributen und Prozeduren in ihrer wesentlichen Aufgabe konzeptionelle Unterschiede. Die Vereinheitlichung aus Kapitel 4.2.3 soll hier nicht zugrundegelegt werden. Daher liegt es nahe, die unterschiedlichen Aufgaben unterschiedlich zu spezifizieren. Die Aufgabe eines *Attributes* besteht in der Berechnung eines Wertes. Eine Formel, die die Berechnung spezifiziert, könnte die Berechnung auf andere, grundlegendere Attribute zurückführen.

$$succ(n) = add(n, 1)$$

Dies ist ein Beispiel, welches den Effekt der Nachfolgerfunktion *succ* über die Addition *add* definiert, wenn die Gleichung von links nach rechts gelesen wird.

Die Aufgabe einer *Prozedur* besteht in der Ausführung eines Zustandsübergangs. Eine Formel, die einen Zustandsübergang spezifiziert, könnte den Folgezustand beschreiben, der erreicht wird.

$$s = X \rightarrow [\ p(a_1, \ldots, a_n)\ ]\ s = succ(X)$$

Sei s hier eine Zustandskomponente. Dann wird nach terminierender Ausführung von p die Zustandskomponente s im Folgezustand den Wert $succ(s)$ bezogen auf den Wert von s im Ausgangszustand haben, der mit der Variablen X bezeichnet ist. $[\ p(a_1, \ldots, a_n)\ ]\ s = succ(old(s))$ ist eine abkürzende Schreibweise, wie sie etwa in Eiffel [Mey92b] benutzt wird. Der vordefinierte *old*-Operator ermöglicht den Zugriff auf den Wert des Argumentes vor Ausführung von p.

Der Effekt einer Prozedur wird als Relation zwischen Eingabe und Ausgabe beschrieben. Vorbedingungen dienen dazu, Operationen als partiell zu spezifizieren.

$$s = X \wedge \neg(X = \omega) \rightarrow [\ p(a_1, \ldots, a_n)\ ]\ s = succ(X)$$

Mit $\neg(X = \omega)$ soll verhindert werden, daß p in einem undefinierten Zustand aufgerufen werden kann.

Mit einer Formel

$$\phi \rightarrow [p(\ldots)]\ \psi$$

wird eine *Vor-/Nachbedingungsspezifikation* formuliert. ϕ ist die Vorbedingung. Sie beschreibt die notwendige Belegung der Parametervariablen und legt Eigenschaften der Zustände fest, in denen p aufgerufen werden kann. Die Vorzustände reflektieren die Effekte früherer Operationen. Die Nachbedingung ψ stellt eine Beziehung zwischen Eingabe und Ausgabe sowie zwischen Vorzustand und Nachzustand her, beschreibt also den Effekt von p. In den vorangegangenen Kapiteln wurden Kommandos und Ausdrücke vorgestellt. Mit diesen können Operationen realisiert werden. Nachbedingungen sind Abstraktionen dieser Realisierungen, die nur deren Effekt beschreiben.

Invarianten sind Eigenschaften eines Objekts, die in jedem Zustand gelten, der Anfangszustand ist oder durch Prozeduraufrufe aus diesem erreicht werden kann. In Zwischenzuständen während der Ausführung der Prozeduren muß deren Gültigkeit nicht gewährleistet sein. Die Invarianten können dazu dienen, das Zusammenspiel von Operationen zu beschreiben. Die klassische Stackspezifikation (für einen Stack s und ein Element e) sei ein Beispiel hierfür.

$$pop(push(s, e)) = s$$

Eine andere Anwendung von Invarianten liegt in der Anpassung von Zustandskomponenten an lokale Anforderungen. Zur Einschränkung der Elementanzahl eines Stacks kann folgendes spezifiziert werden:

$$no\_elements \geq 0\ \wedge\ no\_elements \leq MAX$$

wobei MAX eine beliebig gewählte Konstante ist.

Natürlich sind auch Operationsbeschreibungen als invariant in allen Zuständen eines Objekts anzusehen. Die Definition einer Operation darf sich nicht von Zustand zu Zustand ändern.

Wozu wird die Spezifikationslogik gebraucht?

Logische Spezifikationen sind in verschiedenen methodischen Varianten einsetzbar.

Einige Szenarien sollen nun im folgenden kurz vorgestellt werden:

1. Im Rahmen einer *Anforderungsspezifikation* können Eigenschaften zu entwickelnder Komponenten gefordert werden. Die logische Spezifikation beschreibt axiomatisch die Eigenschaften, die dann von Spezialisierungen (noch axiomatisch) oder Implementierungen (in der imperativen Programmiersprache) zu erfüllen sind.
2. Zu einer bestehenden, u.U. imperativen Komponentenspezifikation können wichtige Eigenschaften, wie etwa das beobachtbare Verhalten, in einer logischen Spezifikation zusammengefaßt werden. Damit können etwa Schnittstellen beschrieben werden. Dieser Ansatz wird in der Entwicklung von Komponenten für die *Wiederverwendung* verfolgt [Som92].
3. Außerdem ist der Formalismus notwendig, um die *Korrektheit* einer Implementierung gegenüber ihrer Spezifikation zu beweisen, also um *verifizieren* zu können. Dies ist eine Erweiterung der zuerst beschriebenen Anwendung. Hier wird die Integration von Spezifikations- und Implementierungsformalismus erforderlich.

5.1.1 Modale Logik

Zur Beschreibung von Attributen und Invarianten hat sich für funktionale Ansätze die Gleichungslogik algebraischer Spezifikationen bewährt [Wir90, vHL89, EM85, EM90, EGL89]. Es liegt also nahe, diese Logik erster Stufe um ein Konzept zur Spezifikation von Zustandsübergängen zu erweitern. Eine solche Logik ordnet sich in die *modalen Logiken* ein [KT90, Har84]. Insbesondere kommt hier die *exogene* Form, die *dynamische Logik*, in Frage. Eine modale Logik erlaubt die Formulierung von Aussagen über Programme oder Programmfragmente in einer Formel. Programme sind in exogener Logik explizit. In *endogener* Logik, z.B. temporaler Logik, sind die Programme Teil der Struktur, über der die Logik interpretiert wird. Programme werden in exogener modaler Logik semantisch als Ein-/Ausgaberelation interpretiert. Ein zusammengesetztes Programm ist bestimmt durch die Relationen seiner Bestandteile. Dynamische Logik ist eine Weiterentwicklung der Hoare-Logik (*Partial Correctness Assertions Method*), siehe [Hoa69, Cou90]. Die dynamische Logik wurde schon in den Spezifikationsansätzen [Jon89a], [Wie91], [Hei92a] und [FMR91] eingesetzt. Exogene modale Logiken wie dynamische Logik oder Hoare-Logik sind also (ggf. erweiterte) Programmlogiken im Sinne von Abschnitt 2.2. In diesem Kapitel wird eine dynamische Logik entwickelt.

Das für diesen Ansatz wichtigste Konstrukt einer dynamischen Logik ist der *modale Box-Operator* oder *always-Operator*. Geschrieben wird er $[\ .\ ]\ \phi$. In exogenen Logiken wird er mit einem Programmfragment, z.B. $[\ p(x)\ ]\ \phi$, indiziert. Die Bezeichnung 'Box-Operator' stammt aus der endogenen modalen Logik, wo der Operator $\Box\ \phi$ geschrieben wird. Nun soll kurz die Semantik des Konstrukts erklärt werden. $[\ P\ ]\ \phi$ gilt in einem Zustand, wenn das Programmfragment P so anwendbar ist, daß P terminiert und die Formel ϕ im Folgezustand gilt[1]. $[\ P\ ]\ \phi$ ist eine *modale Formel*. Je nach Mächtigkeit der Logik darf ϕ nur eine Formel einer Prädikatenlogik erster Stufe (Hoare-Logik) oder kann auch modal sein (echt modale Logik).

[1]In einer *weakest-precondition*-Semantik wird die Menge der Zustände, in der die modale Formel gilt, über die schwächste Vorbedingung, die zur terminierenden Ausführung von P mit Folgezustand ϕ nötig ist, definiert (siehe [LS84], [Gri81] oder [Dijk76]).

Der zweite modale Operator der dynamischen Logik ist der *Diamond*-Operator oder *sometimes*-Operator. Er wird $< . > \phi$ geschrieben. Die Bezeichnung stammt ebenfalls aus der endogenen modalen Logik ($\Diamond\phi$). Auch er kann durch Programmfragmente indiziert werden, also z.B. $< p(x) > \phi$. $< P > \phi$ gilt, falls ein Zustand existiert, in dem P terminiert und in dem dann ϕ gilt. Es soll $< P > \phi$ durch $\neg[\ P\ ]\neg\phi$ definiert sein[2].

Dynamische Logik und Hoare-Logik

Die erweiterte Mächtigkeit der dynamischen Logik gegenüber der Hoare-Logik [Hoa69, Cou90] als klassischer Programmlogik oder der Vor-/Nachbedingungsspezifikation der modellbasierten Spezifikationsansätze [Bjø91, Spi87] wird deutlich, wenn auch der *Diamond*-Operator $< . >$. hinzugenommen wird.

spec modal is
state y : int
opns procedure p : state $\times$ *int* $\rightarrow$ *state* $\times$ *void*
$y = z \rightarrow [\ p(z)\ ]\ z = add(4,z)$
procedure q : state $\times$ *int* $\rightarrow$ *state* $\times$ *void*
$y = z \rightarrow [\ p(z)\ ] < q(z) > y = z$

In dieser Spezifikation macht q den Effekt von p rückgängig.

- $< q(z) > y = z$ bedeutet, daß es eine Möglichkeit gibt, q so auszuführen, daß dann $y = z$ gilt.
- $y = z \rightarrow [p(z)] < q(z) > y = z$ bedeutet, daß es für jede Anwendung von p immer eine Anwendung von q gibt, die die Vorbedingung $y = z$ wieder herstellt, also den Effekt von p rückgängig macht.

Aussagen mit modaler Logik

Die benutzten Gleichungen in den Beispielen zur Motivation waren bisher auf Terme von Datensorten beschränkt. Es können auch Gleichungen der Sorte *state* spezifiziert werden. Zulässig wären dann z.B. Aussagen der Form:

1. $asgn(x,n) =_{state} p(a,b)$
   $asgn$ und p bewirken unter der gegebenen Parametrisierung den gleichen Zustandsübergang, falls die Gleichung gilt.
2. $p(a,b) =_{state} seq(asgn(x,y), if(a = b, q(r,x)))$
   p hat bei der gegebenen Parametrisierung den gleichen Zustandseffekt wie die auf der rechten Seite der Gleichung angegebene Anweisungssequenz.
3. $[p(a)]\ (asgn(x,y) =_{state} q(r))$
   Der Effekt von p (die Nachbedingung) wird durch $asgn(x,y) =_{state} q(r)$ definiert. Hierin wird der Effekt von $asgn$ und q als gleich festgelegt. Da sinnvollerweise Kommandos (wie z.B. $asgn$ oder q) immer in gleicher Weise ausgeführt werden sollen, somit also die Gleichung eine Tautologie ist, ist die Aussage äquivalent zu $[p(a)]\ true$.

[2] nichtintuitionistisch

Beispiel 3 macht unter den gegebenen Einschränkungen keinen Sinn. Operationen sollen immer in gleicher Weise ausgeführt werden, unabhängig von der Ausführung von $p(a)$. Beispiel 2 kann durchaus als Beispiel für die Definition von p herangezogen werden. Beispiel 1 macht Sinn, wenn von rechts nach links gelesen würde, d.h. $p(a,b) = asgn(x,n)$ und dann wie bei Beispiel 2 die rechte Seite als Definition für die linke Seite interpretiert wird ($asgn$ ist vordefinierter Term der Sorte $state$).

Die Beispiele 1 und 2 verfolgen einen Ansatz **konstruktiver Axiome**. Für eine zu definierende Prozedur mit formalen Parametern auf der linken Seite wird auf der rechten Seite ein Modell im Sinne der modelltheoretischen Spezifikationsansätze wie VDM oder Z beschrieben (analog zur Attributdefinition über eine Gleichung und deren Ausführung durch Termersetzung). Bei Verwendung von Vorbedingungen erfolgt eine partielle Beschreibung des intendierten Verhaltens. Eine Formel $[p(\ldots)]\ s = t$ definiert den Effekt von p nicht auf einem Modell der Prozedur, sondern durch **operationale Axiome**, also durch die direkte Veränderung auf dem Zustand (dem Seiteneffekt).

5.1.2 Grundlegende Definitionen für eine modale Logik

Bestandteile einer Logik sind *Terme* und *Formeln*. Terme sollen Trägerelemente von semantischen Strukturen, hier also den Σ-Objekten, bezeichnen. Die Beziehung zwischen Termen und Werten, d.h. Trägerelementen in Σ-Objekten, wird durch eine Interpretation, hier die Funktion v^* (siehe Abschnitt 3.4), hergestellt. Sie ordnet Termen zustandsabhängig einen Wert zu. Formeln machen damit Aussagen über Σ-Objekte. Die Relation zwischen Formeln und Σ-Objekten wird durch einen Gültigkeitsbegriff hergestellt.

Eigenschaften von Σ-Objekten, wie der funktionale Effekt von Attributen oder Vor- und Nachbedingungen von Prozeduren, sollen durch Formeln einer modalen Logik ausgedrückt werden. Grundkonstrukt der Formeln ist die Gleichung. Auf ihr können komplexere Formeln mit Hilfe von Konnektoren aufgebaut werden.

Im Unterschied zu den üblichen Logiken erster Stufe enthalten exogene modale Logiken auch *Programmfragmente*. Die verfügbaren Programmfragmente sollen hier die Kommandos der Σ-Objekte sein (vgl. Kapitel 3). Die möglichen Kommandos sind:

Zuweisung	$asgn(z,t)$
Bedingtes Kommando	$if(b,c)$
Kommandosequenz	$seq(c_1,c_2)$
leeres Kommando	$skip$
Prozeduraufruf	$p(t_1,\ldots,t_n)$

Die Sprache der Programmfragmente soll die Menge der Instruktionen der Σ-Objekte sein. Prinzipiell kann dazu der definierte Ansatz im Umfang der Kapitel 3 und 4 herangezogen werden. Der Einfachheit halber sollen aber nur Konstrukte im Umfang von Kapitel 3 benutzt werden. Zudem sei die Fixpunktsemantik angenommen.

Sei $\Sigma = < S \cup \{state\}, Z, (ATTR \cup PROC) >$ eine Signatur. X sei eine S-sortierte Menge freier Variablen. Formeln sind Aussagen über freien Variablen. Den freien Variablen werden Werte durch die Zustandsabbildung $STATE$ zugewiesen. Neben diesen **Programm-** oder **Zustandsvariablen** genannten Variablen werden noch **Spezifikationsvariablen** eingeführt[3].

[3]Wenn lediglich von (freien) Variablen die Rede ist, sind die Programm- oder Zustandsvariablen gemeint.

Diese sind Hilfsvariablen, die nicht in den Programmen auftreten. Sie ähneln gebundenen Variablen, sie werden als implizit allquantifiziert angesehen. Spezifikationsvariablen werden in der Regel mit Großbuchstaben bezeichnet. Der Zweck dieser Spezifikationsvariablen wird später in diesem Abschnitt erläutert.

Definition 5.1.1 *Eine* **Σ-Gleichung** *hat die Form $t =_s t'$ mit $t, t' \in T(\Sigma, X)_s$, $s \in S$. Eine* **Σ_{state}-Gleichung** *hat die Form $t =_{state} t'$ mit $t, t' \in T(\Sigma, X)_{state}$, die t, t' sind Kommandos, also Zuweisung, bedingte Anweisung, Prozeduraufruf oder Anweisungssequenz.*

Die Terme t, t' können freie Programm- und Spezifikationsvariablen enthalten. Die Bewertung von freien Programmvariablen ist zustandsabhängig. Damit wird auch der noch zu definierende Gültigkeitsbegriff zustandsabhängig sein.

Definition 5.1.2 *Die* ***-Hülle** *über einer Menge A ist die kleinste Menge B mit*

1. *$A \subseteq B$*
2. *B ist abgeschlossen bzgl. der Konnektoren $\neg.$, $.\wedge.$, $[P]\,.$ und des Allquantors $\forall.$.*

P ist Kommandoterm der Σ-Objekte. $[P]\,.$ wird auch **modaler Box-Operator** *genannt. Formeln, die den Konnektor $[P]\,.$ nicht enthalten, werden* **nichtmodale Formeln** *genannt.*

Definition 5.1.3 *Die Menge $WFF(\Sigma)$ der* **wohlgeformten Formeln über** *Σ ist die *-Hülle über*

$$G := \{t_1 =_s t_2 \mid s \in S \cup \{state\}, t_i \in T(\Sigma, X)_s\}$$

der Menge der Σ-Gleichungen und der Σ_{state}-Gleichungen.

Lemma 5.1.1 *Sei eine Formelmenge F durch *-Hüllenbildung auf einer Menge von Basisformeln B und der Menge von Konnektoren K induktiv definiert (mit $r \subseteq F^n \times F$ für $r \in K$, wobei $n \geq 1$ und n von r abhängt). Hier besteht K aus den Konnektoren $K = \{\neg., .\wedge., [P]., \forall.\}$ (in Präfix- oder auch Infix-Notation angegeben). Um zu zeigen, daß eine Eigenschaft Prop für alle $\phi \in F$ gilt, reicht es zu zeigen:*

1. *Induktionsbasis: Prop gilt für alle $\phi \in B$*
2. *Induktionsschritt: Falls Prop für $\phi_1, \ldots, \phi_n \in F$ gilt (Induktionshypothese) und für ein $r \in K$ $((\phi_1, \ldots, \phi_n), \phi) \in r$ gilt, dann gilt Prop auch für ϕ.*

Beweis: (Siehe auch Satz 1.6 in [LS84].) Sei $C \subseteq F$ die Menge aller Elemente von F, für die *Prop* gilt. Es ist zu zeigen, daß aus 1. und 2. folgt, daß *Prop* für alle $\phi \in F$ gilt, also $F \subseteq C$. Mit Lemma 5.1.1(1.) wird die Inklusion $B \subseteq C$ beschrieben und 5.1.1(2.) sagt aus, daß, falls $\phi_1, \ldots, \phi_n \in C$ und $((\phi_1, \ldots, \phi_n), \phi) \in r$ für ein $r \in K$, dann $\phi \in C$ ist. C ist die kleinste Menge, die mit der Basis B und der Konstruktormenge K induktiv definiert ist. F ist nach Definition die kleinste Menge nach Hüllenbildung, also folgt $F \subseteq C$. □

Σ-Formeln werden auf Σ-Gleichungen mit Hilfe der Konnektoren $\neg.$, $.\wedge.$, $[P].$ und des Allquantors $\forall$ aufgebaut. Andere Konnektoren können wie üblich gebildet werden:

$$\begin{aligned} \phi \vee \psi &:= \neg(\neg\phi \wedge \neg\psi) \\ \phi \rightarrow \psi &:= \neg\phi \vee \psi \\ \exists x : s.\phi &:= \neg(\forall x : s.\neg\phi) \end{aligned}$$ [4]

[4] nichtintuitionistisch, da Quantifizierung über unendlichen Bereichen

5.2 Einfache modale Logik

In einem ersten Schritt soll eine eingeschränkte modale Logik vorgestellt werden, die etwa den Umfang einer Programmlogik in bezug auf Prozedurspezifikationen hat. Sie erlaubt es also, Prozeduren durch Vor- und Nachbedingungen zu spezifizieren. Weiterhin können Attribute mit den Elementen einer nichtmodalen Prädikatenlogik erster Stufe beschrieben werden. Eine eingeschränkte modale Logik wird in der Regel zur implementierungsnahen Abstraktion von Prozeduren und Attributen eingesetzt. Echt modale Logiken hingegen erlauben auch das abstraktere Spezifizieren der Abhängigkeiten zwischen Operationen. Aus diesen methodischen Gründen erfolgt die Vorstellung der Logik in zwei Stufen.

Das Vorgehen in diesem Abschnitt entspricht dem üblichen Vorgehen bei Definition einer Logik, d.h. Formeln, Gültigkeit und Modellbegriff werden definiert. Die Grundlagen aus Abschnitt 5.1.2 werden vorausgesetzt.

5.2.1 Gleichung und Formel

Grundelemente der Formelsprache sind die Gleichungen. Sie sind in Definition 5.1.1 festgelegt. Der Formelumfang nach Definition 5.1.3 soll hier eingeschränkt werden. Aus methodischen Gründen soll in diesem Abschnitt 5.2 der modale Box-Operator nur auf nichtmodale Formeln angewendet werden. Dies entspricht dem Umfang einer Programmlogik. $WFF(\Sigma)$ ist dann die kleinste Menge, die die folgenden Eigenschaften hat:

- jede Σ-Gleichung und jede Σ_{state}-Gleichung ist in $WFF(\Sigma)$,
- falls $\phi \in WFF(\Sigma)$, dann ist $\neg\phi \in WFF(\Sigma)$,
- falls $\phi, \psi \in WFF(\Sigma)$, dann ist $\phi \wedge \psi \in WFF(\Sigma)$,
- falls P ein Kommandoterm der Σ-Objekte und $\phi \in WFF(\Sigma)$ nichtmodal ist, dann ist $[P]\ \phi \in WFF(\Sigma)$,
- sei $x \in X_s, \phi \in WFF(\Sigma)$, dann ist auch $\forall x : s.\phi \in WFF(\Sigma)$ für $s \in S \cup \{state\}$.

WFF kann für diesen Zweck in einem ersten Schritt durch eine Hüllenbildung analog zu Definition 5.1.2 auf Gleichungen mit den Konnektoren $\neg, \wedge, \forall$ abgeschlossen werden. Auf dieser ersten Basismenge kann dann eine Hülle mit dem modalen Box-Operator gebildet werden, die die Menge $WFF(\Sigma)$ bildet. Durch diese zweistufige Konstruktion bleibt das Induktionslemma 5.1.1 anwendbar.

Dies ist also eine Erweiterung einer Prädikatenlogik erster Stufe. Der vollständige Umfang der dynamischen Logik ist hier noch nicht erreicht, da wir für ϕ in der Konstruktion des modalen Box-Operators $[P]\ \phi$ vorerst eine nichtmodale Formel annehmen wollen. Mit dem Programmfragment P der modalen Formel werden üblicherweise nur Prozeduren — also Abstraktionen von Kommandofolgen — beschrieben, allerdings sind auch andere Kommandos als Kommandoterme P im modalen Box-Operator möglich.

Bemerkung 5.2.1 *Sei $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$ eine Prozedur. Die $a_i, i = 1, .., n$ seien die aktuellen Parameter eines Aufrufs von p. Allquantifizierung über die Parameter wird in der Regel implizit benutzt.*

$$\psi \rightarrow [\ p(a_1, \ldots, a_n)\ ]\ \phi$$

ist eine Allquantifizierung über den Wertebereichen der Parameter von p sowie über den Wertebereichen der Zustandskomponenten, also

$$\forall x_1, \ldots, x_k \;.\; \psi \rightarrow [\ p(a_1, \ldots, a_n)\ ]\ \phi$$

Die Variablen $x_1, \ldots, x_k$ können frei in ϕ und ψ sowie den Termen $a_1, \ldots, a_n$ vorkommen.

Es sei im folgenden die nachstehende Annahme zugrundegelegt. Statt

$$[\ p(a_1, \ldots, a_n)\ ]\ \phi$$

wird

$$x_1 = a_1 \wedge \ldots \wedge x_n = a_n \rightarrow [\ p(x_1, \ldots, x_n)\ ]\ \phi$$

geschrieben. Die $x_1, \ldots, x_n$ seien allquantifizierte Variablen. Wenn mehrere Aufrufe von p, etwa $[\ p(b_1, \ldots, b_n)\ ]\ \phi_b$ und $[\ p(c_1, \ldots, c_n)\ ]\ \phi_c$, spezifiziert werden sollen, so sind die beiden verschiedenen Aufrufe über verschiedene Vorbedingungen zu spezifizieren. Im folgenden wird, falls auf die Parametrisierung nicht explizit Bezug genommen werden soll, als Abkürzung

$$[\ p(\ldots)\ ]\ \phi$$

benutzt. Falls $[P]\ \phi$ benutzt wird, bezeichnet P ein beliebiges Programmfragment.

Definition 5.2.1 *Eine* **zustandsbasierte Spezifikation** *SP ist ein Paar $SP = <\Sigma, E>$ bestehend aus einer Signatur Σ und einer Menge E von wohlgeformten Formeln $E \subseteq WFF(\Sigma)$.*

Die in Abschnitt 5.1.2 angesprochenen *Spezifikationsvariablen* sollen helfen, in einer Nachbedingung auf Werte in einem Vorzustand Bezug nehmen zu können. Eine imperative Formulierung, wie z.B. $z := z * 2$, soll auf logischer Ebene geeignet formuliert werden können. Sei P das Programmfragment, das $z := z * 2$ realisiert.

$$true \rightarrow [\ P\ ]\ z = old(z) * 2$$

ist eine geeignete logische Spezifikation, wenn $old(z)$ den 'alten' Wert von z (d.h. den Wert von z im Vorzustand) bezeichnet. Die Benutzung des alten Wertes soll über Spezifikationsvariablen explizit und formalisierbar gemacht werden.

$$z = X \rightarrow [\ P\ ]\ z = X * 2$$

mit z als Zustandsvariable und X als Spezifikationsvariable ist als Definition für den *old*-Operator anzusehen. Die Aussage $[\ P\ ]\ z = old(z) * 2$ soll weiterhin als Kurzschreibweise erlaubt sein. Zu den Spezifikationsvariablen siehe auch [Fra92] S. 16 und S. 26. Spezifikationsvariablen $X \in SpecVar$[5] werden durch eine Umgebung $env : SpecVar \rightarrow VAL$ an Werte gebunden. Die Interpretation v^* muß diese Bindungen berücksichtigen. Deren Definition in Kapitel 3.4 sei um

$$v^*(STATE, t) := env(t), \quad falls\ t \in SpecVar$$

erweitert.

[5] $IDENT$ und $SpecVar$ seien disjunkt

5.2.2 Gültigkeit

Mit dem Gültigkeitsbegriff wird die Relation zwischen Formeln und Σ-Objekten hergestellt.

Definition 5.2.2 *Sei v eine Bewertung basierend auf $STATE$ und $env : SpecVar \to VAL$ eine Belegung der Spezifikationsvariablen. Für jedes Σ-Objekt O, den Zustand $STATE \in State$ und die Σ-Formel ϕ sei die* **Gültigkeitsrelation**

O **erfüllt** ϕ **im Zustand** $STATE$ *(kurz $O, STATE \models \phi$)*

definiert durch

1. $O, STATE \models t =_s t'$ *gdw.* $v^*(STATE, t) = v^*(STATE, t'), \quad s \in S \cup \{state\}$
2. $O, STATE \models \neg\phi$ *gdw.* $O, STATE \models \phi$ *nicht gilt*
3. $O, STATE \models \phi \wedge \psi$ *gdw.* $O, STATE \models \phi$ *und* $O, STATE \models \psi$
4. $O, STATE \models [\ P\ ]\ \phi$ *gdw.*
   falls $v^*(STATE, P)[1] \neq \perp$ *gilt, dann gilt* $O, v^*(STATE, P)[1] \models \phi$
5. $O, STATE \models \forall x : s.\ \phi$ *gdw. für alle* $a \in A_s$ *gilt* $O,\ {}^a_xSTATE \models \phi$
   mit ${}^a_xSTATE : X \to A$

   $$ {}^a_xSTATE(y) := \begin{cases} STATE(y) & falls\ y \neq x \\ a & falls\ y = x \end{cases} $$

   ${}^a_xSTATE$ setzt den Wert a für die Variable x. Unabhängig von der Wahl von a muß $O,\ {}^a_xSTATE \models \phi$ gelten. Quantifizierungen sind nur über freien Variablen zulässig.

Aus dem modalen Box-Operator $[P]\ \phi$ und einer Implikation $\psi \to \xi$ mit $\xi \equiv [P]\ \phi$ läßt sich eine **Vor-/Nachbedingungsaussage** konstruieren:

$$\psi \to [P]\ \phi$$

wobei ψ die Vorbedingung und ϕ die Nachbedingung genannt wird. Die Hoare-Formel $h \equiv \{\psi\}P\{\phi\}$ wird üblicherweise wie folgt interpretiert [LS84, Fra92]: h gilt genau dann, wenn, falls ψ im aktuellen Zustand gilt und, falls P terminiert, dann ϕ im Folgezustand gilt. Diese Interpretation ist gleichwertig der Interpretation von $\psi \to [P]\ \phi$ (nach Definition der Gültigkeit und des Konnektors $\to$).

Der Gültigkeitsbegriff ist nach Definition 5.2.2 nur für einen gegebenen Zustand $STATE$ anwendbar.

Definition 5.2.3 *Eine Formel ϕ heißt* **invariant im Σ-Objekt** *O, wenn sie in allen Zuständen $STATE \in State$ gilt, also*

$$\forall\ STATE \in State.\ O, STATE \models \phi.$$

Man schreibt dann $O \models \phi$ und spricht 'ϕ gilt'.

Sei K eine Klasse von Σ-Objekten und ϕ eine Formel. $K \models \phi$ gilt genau dann, wenn alle Objekte in K die Formel ϕ erfüllen.

Definition 5.2.4 *Eine Formel heißt* **Basisformel**, *falls sie keine freien Variablen enthält.*

Basisformeln im vorliegenden Ansatz sind alle Formeln, in denen keine Zustandsbezeichner, keine Spezifikationsvariablen und keine formalen Parameter von Operationen auftreten.

Folgerung 5.2.1 *Falls ϕ Basisformel ist und ϕ in einem beliebigen Zustand $STATE$ gilt, folgt die Invarianz von ϕ für alle Σ-Objekte (hinreichende Bedingung für Invarianz).*

Beweis: Invarianz ist gegeben, wenn Unabhängigkeit vom Zustand $STATE$ vorliegt. Alle freien Variablen werden in $STATE$ verwaltet. Da in Basisformeln keine freien Variablen vorkommen, ist $STATE$ bezogen auf diese Formel immer konstant. □

Bemerkung 5.2.2 *Die Umkehrung von Folgerung 5.2.1 gilt nicht, wie das folgende Beispiel zeigt. Σ enthalte nur ein Prozedursymbol p. Sei $\phi \equiv s = 1$ und sei K eine Klasse von Σ-Objekten, in denen $s = 1 \rightarrow [p(x)]\ s = 1$ für die einzige Zustandstransformation p gilt. Dann ist ϕ invariant, aber keine Basisformel, da der Zustandsbezeichner s in ϕ frei vorkommt.*

5.2.3 Erreichbarkeit und Zugänglichkeit

Das *Erzeugungsprinzip* (nach [BW81], Kapitel 3.2.2, S. 207) besagt:

> *Es werden als Rechenstrukturen nur solche Algebren eines gegebenen abstrakten Datentypen betrachtet, für die die definierten Trägermengen lediglich aus Elementen bestehen, die aus den primitiven Trägermengen mit Hilfe der Operationen der Signatur (endlich) erzeugt werden können.*

Die Notwendigkeit der Informatik, endliche Beschreibungen (für Algorithmen) zu finden, kann durch ein Prinzip unterstützt werden, das die konstruktive Beschreibung von Strukturen ermöglicht. Mit dem Erzeugungsprinzip besteht die Möglichkeit, Induktionsverfahren anzuwenden. Das Erzeugungsprinzip wird in den algebraischen Spezifikationen auch mit dem Begriff der *Erreichbarkeit* bezeichnet. Es lassen sich wesentlich mehr Eigenschaften von erreichbaren Modellen ableiten, als wenn die Eigenschaften auch in nichterreichbaren Modellen gelten müssen. Erreichbarkeit wird auch als die *no junk*-Eigenschaft bezeichnet (siehe [Wir90]).

Der Erreichbarkeitsbegriff aus den algebraischen Spezifikationen kann für die Trägermengen der Datensorten einfach adaptiert werden.

Es sei $t^O_{st} := v^*(st, t)[2]$ die eindeutig bestimmte Auswertung eines Σ-Termes t in einem Σ-Objekt O in einem beliebigen Zustand $st \in State$. Falls t ein Grundterm ist, also zustandsunabhängig, wird der Zustand weggelassen, wir schreiben t^O.

Definition 5.2.5 *Ein* **Element** *a einer Trägermenge A_s einer Datensorte s eines Σ-Objekts O heißt* **erreichbar**, *wenn es in einem beliebigen Zustand st durch einen endlichen Σ-Grundterm bezeichnet werden kann, d.h. $a = v^*(st, t)[2]$, bezeichnet mit $a = t^O$. Es muß also gelten:*

$$\forall st : state. \exists t \in T(\Sigma)_s \ . \ a = v^*(st, t)[2]$$

Ein **Σ-Objekt** *O heißt* **erreichbar** *bzgl. der Datensorten, wenn für alle $s \in S \backslash \{state\}$ und $a \in A_s$ ein Σ-Grundterm $t \in T(\Sigma)_s$ mit $a = t^O$ existiert, wenn also alle Trägerelemente erreichbar sind.*

Der Erreichbarkeitsbegriff ist hier auf Σ-Grundterme eingeschränkt worden. Ein Grundterm enthält keine freien Variablen und ist daher unabhängig vom Zustand. Nun soll Erreichbarkeit in bezug auf die Zustandssorte *state* definiert werden.

Definition 5.2.6 *Ein Zustand* $st \in State$ *eines* Σ*-Objekts* O *heißt* **erreichbar**, *wenn alle Elemente der Trägermengen der einzelnen Zustandskomponenten erreichbar sind.* $st : IDENT \to VAL$ *mit* $st(z_i) : s_i, i = 1,..,n$ *heißt erreichbar, falls für jedes* $s_i, a \in A_{s_i}$ *ein* $t \in T(\Sigma)_s$ *mit* $a = t^O$ *und* $z_i \in Z$ *existiert.*

Es soll nun noch ein Zugänglichkeitsbegriff für Zustandsübergänge, die von Prozeduren hervorgerufen werden können, definiert werden. Es sollen die Zustände ausgezeichnet werden, die sich von einem Initialzustand aus als Folge korrekter Zustandstransitionen ergeben.

Definition 5.2.7 *Ein Zustand* $st \in State$ *heißt* **zugänglich**, *wenn er aus dem Initialzustand*[6] *durch (mindestens) eine Folge von Prozeduraufrufen als erste Komponente der Auswertung berechnet werden kann.*

Definition 5.2.8 *Ein* Σ*-Objekt* O *heißt* **zugänglich**, *wenn alle Zustände in State zugänglich sind.*

5.2.4 Modelle und Theorien

Als Modell einer Spezifikation $SP = < \Sigma, E >$ mit der Signatur Σ und der Formelmenge E wird die Klasse von Σ-Objekten bezeichnet, die jeweils alle Formeln in E erfüllen. Die Klasse aller Σ-Objekte wird mit $Obj(\Sigma)$ bezeichnet.

Definition 5.2.9 *Die* **Modelle** *einer Spezifikation* $SP = < \Sigma, E >$ *werden durch die Modellklasse* $Mod(< \Sigma, E >)$ *bezeichnet.*

$$Mod(< \Sigma, E >) := \{O \in Obj(\Sigma) \mid O \models \phi \text{ für alle } \phi \in E\}$$

Definition 5.2.10 *Unter der* **Theorie** *einer Klasse* K *von* Σ*-Objekten versteht man die Menge der Formeln, die von allen Objekten in* K *erfüllt werden.*

$$Th(K) = \{\phi \in WFF(\Sigma) \mid K \models \phi\}$$

Falls K *leer ist, gilt*

$$Th(\emptyset) = WFF(\Sigma)$$

Bemerkung 5.2.3 *Modellbildung, die keine Isomorphieabgeschlossenheit fordert, wird üblicherweise als* 'lose' *bezeichnet. Es wird hier nicht einmal Erreichbarkeit für die Modelle gefordert, d.h. die Modellsemantik ist* 'sehr lose' *[WB89b]. Motivation und Eigenschaften dieser Semantik werden am Ende des Abschnitts diskutiert.*

Lemma 5.2.1 *Sei* $SP = < \Sigma, E >$ *und* $SP' = < \Sigma, E' >$. *Dann gilt:*

$$Mod(SP) \subseteq Mod(SP') \Rightarrow Th(Mod(SP)) \supseteq Th(Mod(SP'))$$

[6] Es soll angenommen werden, daß alle Komponenten mit ω initialisiert sind.

Beweis: Zu zeigen ist

$$Th(Mod(SP')) \subseteq Th(Mod(SP)), d.h.\ f\ddot{u}r\ alle\ \phi \in WFF(\Sigma) : Mod(SP') \models \phi \Rightarrow Mod(SP) \models \phi$$

unter der Voraussetzung

$$Mod(SP) \subseteq Mod(SP'),\ d.h.\ O \in Mod(SP) \Rightarrow O \in Mod(SP').$$

Sei $\phi \in Th(Mod(SP'))$, also $Mod(SP') \models \phi$ genau dann, wenn $\forall O \in Mod(SP') \,.\, O \models \phi$. Daraus folgt $\forall O \in Mod(SP) \,.\, O \models \phi$ genau dann, wenn $Mod(SP) \models \phi$, also $\phi \in Th(Mod(SP))$. □

Folgerung 5.2.2 $E \subseteq Th(Mod(< \Sigma, E >))$

Beweis: Es erfolgt eine Fallunterscheidung bzgl. der Modellklassen:

- $Mod(< \Sigma, E >) = \emptyset : Th(\emptyset) = WFF(\Sigma)$. Es gilt immer $E \subseteq WFF(\Sigma)$.
- $Mod(< \Sigma, E >) \neq \emptyset$: Sei $\phi \in E$ und $O \in Mod(< \Sigma, E >)$. Nach Definition 5.2.9 gilt $O \models \phi$. Damit ist $\phi \in Th(Mod(< \Sigma, E >))$. □

Folgerung 5.2.3 $Mod(< \Sigma, E >) \subseteq Mod(< \Sigma, Th(Mod(< \Sigma, E >)))$

Beweis: Folgt direkt aus vorangegangener Folgerung. □

Eigenschaften von Modellen

Es gibt i.a. mehrere Möglichkeiten, die Modellsemantik einer Spezifikation zu definieren (siehe auch [Wir90] Kap. 3.2, S. 697 und Kap. 6.1, S. 738), etwa durch:

- die Klasse ihrer Modelle,
- die Menge der Isomorphieklassen der Modelle, d.h. Auswahl eines geeigneten Modells (bis auf Isomorphie),
- die Menge ihrer Kongruenzen[7],
- die Theorie ihrer Modellklasse.

Zum Teil werden als Modelle auch nichterreichbare Algebren [WB89a] oder andere Strukturen [ST90b] zugelassen. Hier wurde mit nicht unbedingt erreichbaren Modellen eine möglichst weitgefaßte Modellsemantik gewählt. Dieser Ansatz dient der Spezifikation von Programmen und deren Implementierung (siehe Einleitung). Intendiert ist eine Nähe der Begriffe *Modell* und *Implementierung* einer Spezifikation, etwa durch einen auf Modellklasseninklusion basierenden Implementierungsbegriff. Eine möglichst große Zahl von Programmimplementierungen soll als korrekt bzgl. einer Spezifikation anerkannt werden können — einen sinnvollen Korrektheitsbegriff vorausgesetzt.

[7]Zwischen den Kongruenzen und den Isomorphieklassen besteht eine 1–1 Korrespondenz.

Im *losen Ansatz* wird eine Spezifikation als ein Vertrag angesehen, der durch verschiedene abstrakte Datentypen[8] interpretiert werden kann. Im Kontext loser algebraischer Spezifikationen werden alle erreichbaren Algebren als Modelle anerkannt. Dieser Ansatz eignet sich insbesondere für Programmentwicklung und Programmverifikation wegen seiner Flexibilität bei der Akzeptierung geeigneter Implementierungen. Die Motivation für die Auswahl eines *sehr losen* Ansatzes wird in den Kapiteln 6 und 8 noch deutlich. Dort zeigt sich ein erhöhtes Maß an Flexibilität, falls in bestimmten Situationen auf Erreichbarkeit verzichtet wird.

Der Begriff der *Erfüllbarkeit* einer Spezifikation ist durch eine nichtleere Modellklasse formalisiert. In vielen Ansätzen wird diese Eigenschaft für die Rumpfspezifikation gefordert, falls Schnittstellenkonzepte vorhanden sind (siehe [Wir90]). Da der Nachweis dieser Eigenschaft schwierig ist (nur durch Theorembeweisen zu lösen), soll hier von dieser Forderung Abstand genommen werden. Erfüllbarkeit ist dann von besonderer Bedeutung, wenn Programme aus Spezifikationen synthetisiert werden sollen. Die Existenz von Modellen ist Voraussetzung, um eines davon realisieren zu können.

Modell einer operationalen Spezifikation

In den Kapiteln 3 und 4 wurden durch die Fixpunktsemantik mathematische Objekte den imperativen Konstrukten zugeordnet. Diese mathematischen Objekte lassen sich unter einem Modellbegriff zusammenfassen.

Definition 5.2.11 *Sei Σ eine Signatur und $SP_{impl} = <\Sigma, I>$ eine operationale Spezifikation auf den Konstrukten des Berechnungsmodells. I besteht aus den imperativen Programmfragmenten, die die Operationen in $opns(\Sigma)$ beschreiben. Dann ist das* **Modell einer operationalen Spezifikation** *durch $Mod(SP_{impl}) = \{Obj\}$ definiert, wobei das Σ-Objekt Obj folgendes enthält:*

- *die in Abschnitt 3.1 definierten Mengen als Trägermengen zu den verwendeten Sorten $int, bool$ usw. ,*
- *eine Abbildung $STATE \in State$,*
- *für jedes $op \in OP$ eine Funktion $v^*(STATE, op)$ entsprechend der Implementierung I. Die Funktion ergibt sich durch die Definition der Kommandofolge, die die Operation op imperativ, d.h. mit den Konstrukten der Σ-Objekte, definiert.*

Das Modell setzt sich also aus den denotationalen Semantiken der einzelnen syntaktischen Elemente der Spezifikation zusammen. Das Modell ist ein Σ-Objekt nach Definition. Die Trägermengen sind die cpos, die Funktionen $v^*(STATE, op)$ sind stetig. Die Modellbildung für axiomatische und imperative Spezifikationen kann vereinheitlicht werden. Für die imperativen Spezifikationen ist die Modellbildung *denotational* oder *modelltheoretisch* (im Sinne von VDM oder Z), d.h. die Modellklasse ist einelementig. Wir setzen hier die Fixpunktsemantik aus Kapitel 4 für die imperative Spezifikation voraus. Nur damit sind die Erweiterungen des Berechnungsmodells nach Kapitel 4 möglich.

[8] Als *abstrakter Datentyp* wird in den algebraischen Spezifikationen eine Abstraktion über den konkreten Repräsentationen von Algebren angesehen. Der abstrakte Datentyp einer Signatur Σ ist eine Isomorphieklasse von Algebren, diese Klasse entspricht einer Σ-Kongruenz über der Grundtermalgebra $T(\Sigma)$. Für Details siehe [Wir90] Kap. 2.3 .

Anmerkungen zu anderen Modellbegriffen

Häufig ist es sinnvoll, einen abstrakten Datentypen axiomatisch exakt durch initiale oder terminale Modelle [Wir90] zu definieren. Der *initiale* und auch der *terminale Ansatz* bieten einen Mechanismus, der eine bis auf Isomorphie eindeutige Algebra als Modell festlegt.

Für die Sprache Extended ML ([ST90b] S. 311) werden *konstruktive Modelle* vorgeschlagen. Die Modelle werden aus den formalen Entitäten der Semantik von Standard ML [MTH90] zusammengesetzt. Statt durch eine Kollektion von Trägermengen wird jeder Typ durch eine Menge von Konstruktoren bestimmt. Statt Funktionen wird eine dynamische Umgebung verwaltet, die Operationsnamen an *Closures* bindet. Konstanten werden an SML-Werte gebunden. Diese konkrete Form der Semantik (gegenüber den abstrakteren Algebren) hat den Vorteil, daß die Auswertung eines Ausdrucks im Modell direkt durch die dynamische Semantik der Programmiersprache definiert werden kann.

Die hier vorgeschlagene Semantik ist in diesem Sinne auch konstruktiv, da die Semantik imperativer Spezifikationen — die Σ-Objekte — auch Basis der Modellklassendefinition axiomatischer Spezifikation ist.

5.3 Erweiterte modale Logik

Die Hoare-Logik [Apt81, LS84, Fra92] ist eine *prädikative Logik*[9], d.h. eine Menge von Formeln einer Prädikatenlogik erster Stufe, die bzgl. der Formelbildung über den üblichen Konnektoren ($\neg.$, $. \wedge .$, $\forall.$) abgeschlossen ist, bevor der Hoare-Formelkonstruktor auf der Menge der Formeln erster Stufe definiert wird. Eine echte Erweiterung der Prädikatenlogik erster Stufe läßt sich definieren, wenn der modale Box-Operator gleichwertig zu den anderen benutzt werden kann. Die Menge $WFF(\Sigma)$ ist dann abgeschlossen bzgl. aller Konnektoren (vgl. *-Hülle in Definition 5.1.2). Eine solche Logik wird *imprädikativ* genannt.

Die Erweiterung der in Abschnitt 5.2 vorgestellten (prädikativen) Logik auf einen imprädikativen Umfang soll in diesem Abschnitt erfolgen. Die Grundlagen aus Abschnitt 5.1.2 sollen weiterhin gelten. Am Ende des Abschnitts wird begründet, warum keine formalen Erweiterungen gegenüber 5.1.2 und 5.2 erforderlich sind.

Es soll hier kurz die in Abschnitt 5.2 beschriebene Logik zusammengefaßt werden. Zur Spezifikation von Attributen steht der volle Umfang der oben angesprochenen Prädikatenlogik ohne modale Konstruktoren zur Verfügung. Prozeduren lassen sich bisher über das Verhalten von Attributen im Folgezustand, etwa durch $[p(..)]\ a_1(a_2(..)) = c$ mit den Attributen a_1 und a_2, oder direkt durch den Effekt auf dem Zustand s, etwa durch $[p(..)]\ s = a(..)$ für eine Prozedur p definieren. Das Verhalten kann über Veränderung mittels des *old*-Operators beschrieben werden: $[p(..)]\ s = old(s) + 1$. Beschreibungen, die keine direkten Zustandszugriffe beinhalten, z.B. $p(..) =_{state} seq(q(..), r(..))$ oder $p(..) =_{state} q(..)$ sind ebenfalls möglich, da Prozedurapplikationen in Gleichungen gestattet sind. Gleichungen auf der Zustandssorte *state* erlauben es, Kommandos '*modelltheoretisch*' zu spezifizieren, d.h. $p(..) =_{state} q(..)$ durch Angabe eines Modells q zur Operation p. Das *Modell* wird hier auf der Spezifikationsebene angegeben, nicht auf der semantischen Ebene.

[9]Begriffsbildung nach [Mit90], dort für Typsysteme benutzt.

Erweiterungen

Nun soll der modale Box-Operators $[P]\ \psi$ und somit auch die Vor-/Nachbedingungsspezifikation $\phi \rightarrow [P]\ \psi$ erweitert werden, indem ϕ, ψ nicht mehr nichtmodal sein müssen. Also erfolgt ein Übergang von einer prädikativen zu einer imprädikativen Logik. Der modale Diamond-Operator wird definiert. Es werden nun folgende Erweiterungen betrachtet:

- Für ϕ, ψ in $\phi \rightarrow [P]\ \psi$ werden auch modale Formeln zulassen. Mit der Erweiterung sind nun Verschachtelungen
$$[P][Q]\ \psi$$
formulierbar. [FJ92] führen den Begriff *history* ein. Dieser ist eine Bezeichnung für die Liste der durchlaufenen Zustände. $[P]\ \phi$ bedeutet, daß ϕ im Folgezustand der Ausführung von p interpretiert wird. D.h. $[P][Q]\ \psi$ beschreibt dann zwei Zustandstransitionen auf den Zuständen z_1, z_2 und z_3. P führt eine Transition von z_1 nach z_2, Q eine Transition von z_2 nach z_3 durch, wobei im Zustand z_3 dann ψ gelten soll. Die Liste der durchlaufenen Zustände besteht hier also aus (z_1, z_2, z_3). Eine solche Spezifikation läßt sich durch eine Sequenz von einfacheren Kommandos realisieren (s.u.). In Lemma 7.5.2 wird gezeigt, daß $[P][Q]\ \phi$ zu $[seq(P,Q)]\ \phi$ äquivalent ist. Ebenfalls ist formulierbar:
$$([P]\ \phi) \rightarrow [Q]\ \psi$$
Wenn die Bedingung, daß in allen Zuständen, in denen P ausgeführt wird, nachher ϕ gilt, erfüllt ist, dann soll die partielle Korrektheitsaussage $([P]\ \phi) \rightarrow [Q]\ \psi$ für Q und ψ gelten. Hier wird die Existenz einer Definition für Q von der von P abhängig gemacht. Eine Operation heißt *dynamisch*, wenn ihre Definition (des Verhaltens) zustandsabhängig erfolgen kann; sie heißt *statisch*, wenn sich ihre Definition während der gesamten Lebenszeit des Objekts nicht ändert[10]. Falls durch P eine statische Operation beschrieben wird, ist eine Semantik entweder immer oder garnicht vorhanden. Diese Spezifikation ist nur dann sinnvoll, wenn mit P eine dynamische, d.h. eine veränderbare Operation beschrieben werden soll.

- Der Diamond-Operator $< P >\ \psi$ wird eingeführt. $< P >\ \psi$ besagt, daß P so ausgeführt werden kann (d.h. daß ein Folgezustand gefunden werden kann), daß dann ψ gilt. $< P >\ true$ ist eine einfache Terminierungsaussage.

  Der Diamond-Operator könnte auf der Basismaschine realisiert werden, wenn diese ein *Definiertheitsprädikat* für Programmfragmente bereitstellt. Dies muß auf Operationsaufrufe anwendbar sein. Üblicherweise wird in modalen Logiken [KT90, Har84, Sti91] aber der Diamond-Operator durch Negation des Box-Operators definiert:
$$< P > \phi := \neg([P]\neg\phi)$$
  Die Verschachtelung von Box- und Diamond-Operator ist erlaubt. Deren Anwendung wurde am Beispiel einer Undo-Operation (Abschnitt 5.1.1 oder [FJ92] S. 133) schon erläutert. Es ist allerdings zu bemerken, daß eine solche Verschachtelung selten gebraucht wird.

[10] Die Begriffe *'statische'* und *'dynamische Operationen'* stammen aus [Gur93, Gur94].

Terminierung wird im folgenden nicht detailliert betrachtet.

Gegenüber einer klassischen Programmlogik oder Hoare-Logik [LS84] gibt es mit dem nun vorgestellten Umfang folgende Erweiterungen:

- Diamond-Operator,
- Folgen von modalen Operatoren,
- Verschachtelung von Diamond- und Box-Operator,
- Gleichungen auf der Sorte *state*.

Die in Abschnitt 5.2 vorgestellte Logik reicht für unsere Zwecke aus, um Implementierungs- und Schnittstellenspezifikationen formulieren zu können. Deren Einsatz und technische Unterstützung, z.B. durch ein Beweissystem, ist auf die Verifikation ausgerichtet. Eine abstraktere Logik ist in einem anderen methodischen Kontext einzusetzen. Hier sind Verifizierbarkeit gegenüber imperativen Implementierungen oder gar Ausführbarkeit nicht mehr in gleichem Maße relevant. Für diese dann echt modale Logik ist dann wieder ein Beweissystem zu definieren und zu analysieren. Die echt modale Logik umfaßt die bisherige.

Formeln, Gültigkeit und Modelle

Im vorangegangenen Abschnitt *'Erweiterungen'* wurde die Erweiterung der Spezifikationslogik auf modalen Umfang aus konzeptioneller Sicht behandelt. Die Konzepte sind jetzt noch formal aufzuarbeiten, d.h. Formel-, Gültigkeits und Modellbegriff sind, soweit erforderlich, anzupassen.

Definition 5.1.3 von WFF erlaubt schon beliebige Verschachtelungen des modalen Box-Operators. Da sich der *Diamond*-Operator durch den *Box*-Operator darstellen läßt, braucht er hier nicht weiter betrachtet zu werden. Die Gültigkeitsdefinition 5.2.2 kann unverändert übernommen werden. Auch an der Modellbildung nach Definition 5.2.9 ändert sich nichts.

5.4 Vergleich der Σ-Objekte mit anderen Ansätzen

Zuerst wird der Ansatz zustandsbasierter Spezifikation, insbesondere das Berechnungsmodell der Σ-Objekte mit der *algebraischen Spezifikation*, insbesondere der Modellbildung der algebraischen Spezifikationen, verglichen. Ziel ist der Nachweis, daß mit dem Berechnungsmodell der Σ-Objekte zustandsbasierte Systeme problemnäher modelliert werden können. Dann werden Σ-Objekte mit den Standardstrukturen der modalen Logiken, den *Kripke-Modellen*, verglichen. Es wird gezeigt, daß Σ-Objekte spezielle Kripke-Modelle sind. Anschließend erfolgt ein konzeptioneller Vergleich mit einer sehr mächtigen algebraischen Spezifikationstechnik, dem *beobachtungsorientierten Spezifizieren*. Am Ende des Abschnitts werden temporale Logiken betrachtet.

5.4.1 Algebren

In diesem Abschnitt sollen Unterschiede zwischen Σ-Objekten und Σ-Algebren (nach [Wir90]) herausgearbeitet werden. Basis des Vergleichs sei eine Objektsignatur $\Sigma = < S, Z, OP >$

(nach Definition 3.2.1). Σ-Objekte sind in Abschnitt 3.3 definiert. Unter einer Σ-Algebra zu einer Signatur aus Datensorten S und Operationssymbolen OP versteht man eine Kollektion aus einer S-sortierten Menge von Trägermengen für jede Sorte $s \in S$ und einer Funktion für jedes Operationssymbol $op \in OP$. Zur genauen Definition algebraischer Spezifikationen siehe Anhang A.3.

Wie im folgenden gezeigt wird, können zustandsbasierte Systeme auch algebraisch spezifiziert werden. Um eine Objektsignatur im algebraischen Ansatz darstellen zu können, müssen einige Veränderungen vorgenommen werden. Hierzu werden wir zunächst einige Vorüberlegungen anstellen:

1. Zustandsbezeichner gibt es als solche im algebraischen Ansatz nicht. Zustandsbezeichner sind Variablen, die zustandsabhängig Werte speichern. Da Zustände im algebraischen Ansatz nicht inhärent vorhanden sind, müssen sie explizit modelliert werden. Daher soll eine Menge abstrakter Zustände angenommen werden, in Abhängigkeit derer Zustandsbezeichner interpretiert werden können. Folgende Konstruktion soll dies realisieren:

$$\begin{array}{lll} z & : & \rightarrow ident \qquad \text{für alle } z \in Z \\ ST & : & state \times ident \rightarrow val \end{array}$$

   z stellt einen Zustandsbezeichner dar. z wird als Konstante (null-stellige Operation) realisiert, deren Wert ein Bezeichner ist (Sorte *ident*). *state* soll durch eine Trägermenge abstrakter Zustände interpretiert werden. ST ist eine Abbildung, die jedem Bezeichner zustandsabhängig einen Wert zuweist. ST soll durch die folgenden Eigenschaften charakterisiert sein:

   (a) $ST(\perp_{state}, z_i) = \perp_{val}$, d.h. im undefinierten Zustand $\perp_{state} : state$ ist kein Zustandsbezeichner definiert,

   (b) $\forall st : state, z : ident \; . \; D(ST(st, z)) = true$. D ist ein Definiertheitsprädikat (s.u.).

   $\perp_{val}$ ist eine Konstante der Sorte *val*, die das undefinierte Element repräsentieren soll. Es wird mit der Aussage (b) ausgedrückt, daß ST total sein soll. Mit ST wird die Funktionalität von $STATE$ nachgebildet. $STATE$ ist die Variablenbelegung in den Σ-Objekten. $STATE$ wird durch eine Funktion nachgebildet, die die Veränderbarkeit von $STATE$ realisiert. Es wird durch ST in Abhängigkeit vom aktuellen Zustand den Variablen ein Wert zugewiesen.

2. Die Wertebereichsangabe s einer Operationsvereinbarung $op : s_1 \times \ldots \times s_n \rightarrow s$ im algebraischen Ansatz darf nur eine Komponente umfassen. Es muß also für Operationen eine tupelwertige Sorte gebildet werden, auf die dann durch explizit modellierte Projektionsfunktionen zugegriffen wird. Statt

$$op : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$$

   werden nun

$$op : state \times s_1 \times \ldots \times s_n \rightarrow state\_s$$

   und

$$pr1_s : state\_s \rightarrow state \quad \text{und} \quad pr2_s : state\_s \rightarrow s$$

   benötigt.

Die Projektionen $pr1_s$ und $pr2_s$ sind für jede Datensorte s, die auf der rechten Seite von Operationsvereinbarungen auftritt, zu definieren. $state\_s$ ist für jedes $s \in S$ eine neue Sorte. Projektionsfunktionen $pr1$ und $pr2$ sowie ein Konstruktor $constr$ sollen im folgenden charakterisiert werden. Seien $first, second$ und $pair$ Sorten.

$$\begin{array}{ll} constr & : first \times second \rightarrow pair \\ pr1 & : pair \rightarrow first \\ pr2 & : pair \rightarrow second \end{array}$$

Dies sind die Funktionen, die zusätzlich in den Signaturen der algebraischen Spezifikation gebraucht werden. Sie sollen durch folgende Axiome in den algebraischen Spezifikationen festgelegt sein:

$$\begin{array}{l} pr1(constr(a,b)) = a \\ pr2(constr(a,b)) = b \\ constr(pr1(a), pr2(a)) = a \end{array}$$

Nur so ist die Definition einer Projektion (selektiver Zugriff) möglich. Für diese Spezifikation ist es unerheblich, ob der konstruierte Typ '*pair*' durch kartesische Produkte, Listen oder andere Strukturen dargestellt wird.

3. Prozeduren werden im zustandsbasierten Ansatz durch Funktionen interpretiert, die partiell korrekt bezüglich der Prozedurspezifikation sind. Die partielle Korrektheit ist algebraisch nur darstellbar, wenn im algebraischen Ansatz adäquate Mittel bereitstehen. Dies kann durch partielle Algebren erfolgen [BW82][11]. In dem in [Wir90] Kap. 3.3.2 vorgestellten (und auf [BW82] basierenden) Ansatz wird ein Definiertheitsprädikat D zur Verfügung gestellt. Mit Mitteln totaler Ansätze ist die partielle Korrektheitsaussage, die der Interpretation des Box-Operators zugrundeliegt, nicht darstellbar. Also soll hier ein Definiertheits- bzw. Terminierungsprädikat — letzteres ist für partielle Korrektheit wesentlich — für die unterliegenden partiellen Algebren vorausgesetzt werden. Sei im folgenden D dieses Definiertheitsprädikat. $D(t)$ gilt für einen Term t, falls die Auswertung von t definiert ist. D angewendet auf einen Prozeduraufruf soll ungültig sein, falls die Ausführung der Prozedur nicht terminiert.

   Hier wird eine lose Modellbildung ohne Erreichbarkeit für die zustandsbasierte Spezifikation realisiert. Um die jeweiligen Modellklassenbildungen zu vergleichen, soll eine lose Modellbildung ohne Erreichbarkeitsforderung auch für den algebraischen Ansatz vorausgesetzt werden.

   Sei die Prozedur $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times void$ im zustandsbasierten Ansatz durch das Axiom
$$\phi \rightarrow [p(st, t_1, \ldots, t_n)]\ \psi$$
   spezifiziert. Für die Darstellung im algebraischen Ansatz seien die Konstruktionen aus den ersten beiden Punkten angenommen (ST, Projektionen). Die Spezifikation von p soll nun im algebraischen Ansatz formuliert werden. Dazu muß zuerst das Problem gelöst werden, daß sich eine Zustandsvariable im algebraischen Ansatz nicht direkt auswerten läßt. Sei st der aktuelle Zustand. Dann ist der Wert einer Zustandsvariablen z beschrieben durch $ST(st, z)$. D.h. in jeder Formel ϕ des zustandsbasierten Ansatzes

[11] Alternativ auch stetige Algebren, siehe [Wir90] Kapitel 3.3 oder hier Kapitel 6.1.1.

ist das Auftreten einer Zustandsvariablen durch obige Konstruktion zu ersetzen. Für $\phi \rightarrow [p(st, t_1, \ldots, t_n)]\ \psi$ muß also geschrieben werden:

$$\phi[z_i/ST(st, z_i)] \rightarrow (\ D(p(st, t_1, \ldots, t_n)) \rightarrow \psi[z_i/ST(pr1(p(st, t_1, \ldots, t_n)), z_i)]\ )$$

$\phi[x/y]$ beschreibt die syntaktische Substitution aller freien Vorkommen von x in ϕ durch y. $D(.)$ ist das Prädikat, das Aussagen über die Terminierung des Argumentes macht[12].

Die eben angesprochenen Ideen werden nun in ein formales, konstruktives Vorgehen umgesetzt. Es sei eine Objektspezifikation OS gegeben. Es soll gezeigt werden, daß Modellklassen von OS kleiner sind als Modellklassen einer in bezug auf das Verhalten der Operationen gleichwertigen algebraischen Spezifikation. Dazu ist zuerst eine gleichwertige algebraische Spezifikation $AS = <\overline{\Sigma}, \overline{E}>$ zu $OS = <\Sigma, E>$ zu konstruieren. Dann ist die Vergleichbarkeit auf Modellebene zu gewährleisten.

Die Umsetzung auf der Ebene der Spezifikationen wird durch eine Abbildungsvorschrift $X \mapsto Y$ für ein bestimmtes Sprachkonstrukt angegeben, d.h. X soll durch Y ersetzt werden. Es soll angenommen werden, daß auf Y alle notwendigen Transformationen auch für andere Sprachkonstrukte angewendet werden. Für die Objektspezifikation wird jeweils die explizite Form (etwa $pr_1(p)(st, t_1, \ldots, t_n)$ statt des impliziten $p(t_1, \ldots, t_n)$ für Prozeduraufrufe) benutzt (siehe Diskussion in Abschnitt 3.2.3).

Die Grundlagen algebraischer Spezifikation sind im Anhang A.3 zu finden. Hier soll ein Spezialfall algebraischer Spezifikation benutzt werden: der Ansatz partieller abstrakter Datentypen nach [BW82] (siehe auch Kapitel 3.3.2 *Partial Algebras* in [Wir90]). Das dort vorgestellte Definiertheitsprädikat D soll vorhanden sein (siehe auch Abschnitt 6.1.1). Die Modellbildung soll nach dem losen Ansatz erfolgen. Gültigkeit sei wie folgt definiert (A ist eine Algebra, v eine Belegung):

$$\begin{array}{lll} A, v \models D(t) & \text{gdw.} & v^*(t) \text{ ist definiert} \\ A, v \models t = t' & \text{gdw.} & v^*(t) = v^*(t') \text{ oder } (v^*(t) \textit{ und } v^*(t') \text{ sind undefiniert}) \end{array}$$

Hiermit kann die Gleichheit rekursiver Funktionen beschrieben werden. Nichtterminierende rekursive Funktionen sind undefiniert. Bei der Betrachtung geht es hier um die Terminierungsaussage für die partielle Korrektheit.

Σ-Objekte enthalten (mehr oder weniger explizit) Funktionen, die in der Signatur nicht explizit auftreten: $STATE$ ist die Zustandsabbildung, Projektionen auf *state*- bzw. $s \in S$-Realisierungen, Paarbildung versteckt in den Funktionen sowie die Kommandorealisierungen für $asgn, if, seq$. Diese müssen in der algebraischen Spezifikation explizit nachgebildet werden. Im folgenden wird nun die notwendige Konstruktion für Signaturen, Terme, Interpretationen, Gültigkeit, Axiome und Modellbildung durchgeführt. Das Ergebnis der Untersuchung wird in einem Satz (5.4.1) ausgedrückt.

Signaturen

Aus einer beliebigen Objektsignatur $\Sigma = <S \cup \{state\}, Z, OP>$ ist die zugeordnete algebraische Signatur $\overline{\Sigma} = <\overline{S}, \overline{F}>$ zu konstruieren.

[12] Das Halteproblem für Turingmaschinen ist nicht entscheidbar. Ein Algorithmus, der für jede beliebige Turingmaschine entscheiden kann, ob sie bei gegebener Eingabe terminiert, existiert nicht.

$$
\begin{array}{lll}
\overline{S} & := & S \cup \{state\} \cup \{bool, ident, val\} \cup \{state\_s \mid s \in S\} \\
\overline{F} & := & \quad \{\overline{op} : state \times s_1 \times \ldots \times s_n \rightarrow state\_s \mid op : state \times s_1 \times \ldots \times s_n \rightarrow state \times s \in OP\} \\
 & & \cup \{pair_s : state \times s \rightarrow state\_s \mid s \in S\} \\
 & & \cup \{pr1_s : state\_s \rightarrow state \mid s \in S\} \\
 & & \cup \{pr2_s : state\_s \rightarrow s \mid s \in S\} \\
 & & \cup \{ST : state \times ident \rightarrow val\} \\
 & & \cup \{\overline{z} :\rightarrow ident \mid z \in Z\} \\
 & & \cup \{\overline{asgn} : state \times ident \times val \rightarrow state\} \\
 & & \cup \{\overline{if} : state \times bool \times state \rightarrow state\} \\
 & & \cup \{\overline{seq} : state \times state \times state \rightarrow state\}
\end{array}
$$

Es sind nicht alle Kommandos behandelt (etwa $skip$ fehlt). Es könnte $\overline{F}$ um $\overline{init} :\rightarrow state$ erweitert werden. Durch $\overline{init}$ könnte ein Initialzustand erzeugt werden, der hier durch die Initialisierung aller Zustandsvariablen mit ω erreicht wird. Es soll $\overline{op} = pair_s \circ op$ für $op : state \times s_1 \times \ldots \times s_n \rightarrow state \times s \in OP$ gelten. $asgn, if, seq$ sind Kommandoterme, d.h. sie sind direkt auf der Sorte $state$ definiert, Projektionen für sie werden also nicht benötigt. Ein Element der Sorte $state$ wird im algebraischen Ansatz durch abstrakte Zustände interpretiert. Kommandos sind so realisiert, daß sie die Abstraktion des Folgezustandes liefern (Beispiel $seq : state \times state \times state \rightarrow state$). Der erste Parameter ist der aktuelle Zustand (oder ein Kommando, das ihn liefert); zweiter und dritter Parameter sind die Kommandos, die eine Sequenz bilden sollen. Der Ergebniswert ist der Zustand, der sich nach Ausführung der beiden Kommandos auf dem aktuellen Zustand ergibt.

Terme

Nun werden wir die induktiv definierten Σ-Terme in $\overline{\Sigma}$-Terme umformen. Es erfolgt eine Fallunterscheidung über die Termkonstruktion über Objektsignaturen.

- Zustandsbezeichner z_i (nicht als linke Seite in einem $asgn$): $z_i \mapsto ST(st, z_i)$ wobei $st : state$ freie Variable ist.
- Operationsaufrufe
  - Attributaufrufe $a(st, t_1, \ldots, t_n)$: $pr_2(a)(st, t_1, \ldots, t_n)$ ist ein Term der Sorte s für $a : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$

    $$pr_2(a)(st, t_1, \ldots, t_n) \mapsto pr2_s(\overline{a}(st, t_1, \ldots, t_n))$$

  - Prozeduraufrufe $p(st, t_1, \ldots, t_n)$: $pr_1(p)(st, t_1, \ldots, t_n)$ ist Term der Sorte $state$ für $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$

    $$pr_1(p)(st, t_1, \ldots, t_n) \mapsto pr1_{state}(\overline{p}(st, t_1, \ldots, t_n))$$

- Kommandoterme (außer Prozeduraufruf)
  - Zuweisung $asgn(z, t)$[13]:

    $$asgn(z, t) \mapsto \overline{asgn}(st, z, t)$$

    z wird hier nicht transformiert. st ist freie Variable der Sorte $state$.

[13]Die Behandlung von $asgn$ ist vereinfacht. Eigentlich müßte $\overline{asgn_s} : state \times ident \times s \rightarrow state$ statt $\overline{asgn} : state \times ident \times val \rightarrow state$ definiert werden (für alle $s \in S$). Hier ist die Menge der Sorten durch val verallgemeinert. val muß durch die Vereinigung der Trägermengen aller $s \in S$ interpretiert werden.

- Sequenz $seq(c_1, c_2)$:

$$seq(c_1, c_2) \mapsto \overline{seq}(st, c_1, c_2)$$

st ist freie Variable der Sorte *state*.

- Bedingtes Kommando $if(b, c)$:

$$if(b, c) \mapsto \overline{if}(st, b, c)$$

st ist freie Variable der Sorte *state*.

Interpretation

In Objektspezifikationen werden Variablen durch $STATE$ verwaltet. Es erfolgt eine induktive Erweiterung für Terme, bezeichnet durch v^*. In algebraischen Spezifikationen werden Variablen durch die Belegung $\overline{v}$ verwaltet. Die induktive Erweiterung wird mit $\overline{v^*}$ bezeichnet.

Die Interpretation von Termen soll durch Fallunterscheidung über die Termkonstruktion über Signaturen betrachtet werden.

- Zustandsbezeichner z_i: Sie werden im objektbasierten Ansatz als freie Variable behandelt. Die Belegung mit einem Wert erfolgt über $STATE$. Im algebraischen Ansatz ist z_i ein konstanter Bezeichner. Der Wert wird über den Funktionsaufruf $ST(st, z_i)$ im aktuellen Zustand $st : state$ ermittelt.

- Operationsaufrufe $op(t_1, \ldots, t_n)$: In beiden Ansätzen erfolgt die Interpretation analog. Operationsaufrufe sind jeweils induktiv über Funktionen und die Interpretation der Parameter definiert. Sei $a(t_1, \ldots, t_n)$ ein Attributaufruf (und $\overline{a}(\overline{t_1}, \ldots, \overline{t_n})$ der zugehörige algebraische Term) für $a : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$, O ein Σ-Objekt und A eine $\overline{\Sigma}$-Algebra.

$$\begin{array}{ll} v^*(STATE, a(t_1, \ldots, t_n)) & := a^O(STATE, v^*(STATE, t_1)[2], \ldots, v^*(STATE, t_n)[2]) \\ \overline{v^*}(\overline{a}(st, t_1, \ldots, t_n)) & := \overline{a^O}(\overline{v^*}(st), pr1^A_{s_1}(\overline{v^*}(\overline{t_1})), \ldots, pr2^A_{s_n}(\overline{v^*}(\overline{t_n}))) \end{array}$$

  Analog für Prozeduren.

- Kommandoterme (außer Prozeduraufruf)

  - Zuweisung $asgn(z, t)$: Im Σ-Objekt-basierten Ansatz ist die Zuweisung über Substitution definiert. Im algebraischen Ansatz muß dies axiomatisch spezifiziert werden (s.u. 'Axiome').
  - Sequenz $seq(c_1, c_2)$: Die Sequenz ist im Σ-Objekt-basierten Ansatz semantisch durch Funktionskomposition definiert. Im algebraischen Ansatz muß dies axiomatisch spezifiziert werden.
  - Bedingtes Kommando $if(b, c)$: Im Σ-Objekt-basierten Ansatz ist das bedingte Kommando über ein if-Konstrukt definiert. Im algebraischen Ansatz muß dies axiomatisch spezifiziert werden.

Weitere, fest vorgegebene Interpretationen gibt es für Zustände $STATE \in State$ (d.h. Elemente der Sorte *state*), die Projektionen und die Paarbildungen. Diese werden ebenfalls noch axiomatisch spezifiziert.

Formeln

Die Betrachtung der Formeln erfolgt durch Fallunterscheidung über den Aufbau von $WFF(\Sigma)$. Es ist zu beachten, daß rechts von $'\mapsto'$ die jeweils transformierten Werte stehen.

- Gleichung $t_1 = t_2$:
$$t_1 = t_2 \mapsto \overline{t_1} = \overline{t_2}$$
- Konnektoren $\neg, \wedge, \forall$ in nichtmodaler Formel ϕ:
$$\phi \mapsto \overline{\phi}$$
- Box-Operator $\phi \rightarrow [P]\ \psi$: Auf algebraischer Seite steht das Definiertheitsprädikat D zur Verfügung. P ist ein Kommandoterm, in dem ein $st : state$ explizit auftritt.
$$\phi \rightarrow [P]\ \psi \quad \mapsto \quad \overline{\phi} \rightarrow (D(P) \rightarrow \overline{\psi})$$
wobei $\overline{\phi}$ durch Durchführung aller Transformationen auf ψ gebildet wird. $st : state$ sei die freie Variable, die den Zustand bezeichnet. $\overline{\psi}$ wird durch Durchführung aller Transformationen auf ψ konstruiert, allerdings wird der Folgezustand $pr1_s(P)$ statt des aktuellen Zustandes st bei der Transformation benutzt.

Gültigkeit

Die Gültigkeit wird durch Fallunterscheidung über den Aufbau der Formeln in $WFF(\Sigma)$ betrachtet. Sei O ein Σ-Objekt und A eine $\overline{\Sigma}$-Algebra. Die Bewertung im algebraischen Ansatz sei mit $\overline{v}$, deren Gültigkeit mit $\models_A$ bezeichnet.

- Gleichung $t_1 = t_2$:
$$\begin{array}{lll} O \models t_1 = t_2 & \text{gdw.} & v^*(STATE, t_1) = v^*(STATE, t_2) \text{ für alle } STATE \in State \\ A \models_A \overline{t_1} = \overline{t_2} & \text{gdw.} & \overline{v}^*(\overline{t_1}) = \overline{v}^*(\overline{t_2}) \text{ für alle Bewertungen } \overline{v} : X \rightarrow A \end{array}$$
Wir nehmen nun an, daß die zusätzlich hinzugenommenen Funktionalitäten ($\overline{asgn}$, ...) in $\overline{F}$ so spezifiziert sind, daß ihre Interpretationen die wesentlichen Eigenschaften der entsprechenden Konstrukte in Σ-Objekten erfüllen. Diese Eigenschaften sind weiter unten unter der Überschrift 'Axiome' vollständig spezifiziert. Damit kann die Betrachtung der Elemente der Terme (strukturell induktiv) erfolgen:
  - Zustandsbezeichner:
  $$v^*(STATE, z_i)[2] = \overline{v}^*(ST(st, z_i))$$
  falls st den aktuellen Zustand bezeichnet. Dies gilt nach Konstruktion von $state$ und ST.
  - Operationsaufrufe: Bei analoger Definition der Interpretationen von op und $\overline{op}$ (siehe Anmerkungen zur Interpretation) wird die Gleichheit der Terme über die Projektionen erhalten (mit Einbeziehung der Paarbildung für $\overline{op}$).
  - Kommandos: mit obiger Annahme werden durch $\overline{asgn}$, $\overline{if}$ und $\overline{seq}$ die gleichen neuen Zustände wie von den Kommandos $asgn, if, seq$ produziert (oder präziser die passenden Abstraktionen der Zustände durch die Konstruktion über $ST(st, z_i)$ mit abstraktem Zustand st und Zustandsbezeichner z_i).

- Konnektoren $\neg, \wedge, \forall$: Deren Gültigkeit ist in beiden Ansätzen analog definiert (siehe Formelbehandlung).
- Box-Operator $[P]\ \psi$: Dem $[P]\ \psi$ in modaler Form entspricht $D(P) \rightarrow \overline{\psi}$ in algebraischer Form. Mit $D(P)$ und der Konstruktion des Folgezustandes in $\overline{\psi}$ wird im algebraischen Ansatz genau die partielle Korrektheit umgesetzt.

Axiome

Mit $OS = <\Sigma, E>$ und $AS = <\overline{\Sigma}, \overline{E}>$ ist

$$\overline{E} := \{\overline{\phi} \mid \phi \in E, \phi \mapsto \overline{\phi}\} \cup \{\psi \mid \psi \ spezifiziert\ die\ zusätzlichen\ Operationen\}$$

Zusätzlich zu spezifizieren sind gemäß Konstruktion von $\overline{\Sigma}$:

1. Für Paarbildung und Projektion $pair_s, pr1_s, pr2_s$ für alle $s \in S$:
   $pr1_s(pair_s(a,b)) = a$
   $pr2_s(pair_s(a,b)) = b$
   $pair_s(pr1_s(a), pr2_s(a)) = a$
2. Für Zustandsbezeichner $\overline{z}$ für alle $z \in Z$ und Zustandsabbildung ST:
   $ST(\bot, \overline{z}) = \bot$
   $D(ST(st, \overline{z})) = true$
3. Für die Kommandorealisierungen $\overline{asgn}, \overline{seq}, \overline{if}$:
   $ST(\overline{asgn}(st, z, t), z) = t$
   $\overline{seq}(st, c_1, c_2) = c_2(c_1(st))$
   $b = false \rightarrow \overline{if}(st, b, c) = st$
   $(b = true \wedge (D(c) \rightarrow \phi)) \rightarrow (D(\overline{if}(st, b, c)) \rightarrow \phi)$

Modellbildung

Es werden nichterreichbare Σ-Objekte und nichterreichbare $\overline{\Sigma}$-Algebren betrachtet, um die Modellklassenbildungen vergleichen zu können.

$$Mod(OS) := \{O \in Obj(\Sigma) \mid O \models \phi \ für\ alle\ \phi \in E\ \}$$
$$\overline{Mod}(AS) := \{A \in Alg(\overline{\Sigma}) \mid A \models_A \overline{\phi} \ für\ alle\ \overline{\phi} \in \overline{E}\ \}$$

Definition 5.4.1 *Sei $\Sigma = <S, Z, OP>$ eine Objektsignatur und $\overline{\Sigma} = <\overline{S}, \overline{F}>$ die zugehörige algebraische Signatur. Ein* **OA-Signaturmorphismus** *$\sigma : \Sigma \rightarrow \overline{\Sigma}$ ist ein Tripel von Abbildungen $<\sigma_Z, \sigma_S, \sigma_{OP}>$ mit $\sigma_Z : Z \rightarrow \overline{F}, \sigma_S : S \rightarrow \overline{S}, \sigma_{OP} : OP \rightarrow \overline{F}$, so daß stets gilt: ist $op \in OP$ mit $op : state \times s_1 \ldots \times s_n \rightarrow state \times s$, so ist $\sigma_{OP}(op) : state \times \sigma_S(s_1) \times \ldots \times \sigma_S(s_n) \rightarrow state\_s$. σ_Z bildet die Zustandsbezeichner $z \in Z$ auf Konstanten $z :\rightarrow ident$ ab.*

Folgerung 5.4.1 *Die definierte Abbildung $\Sigma \mapsto \overline{\Sigma}$ ist ein OA-Signaturmorphismus.*

Beweis: Trivial. □

Es wird durch die Abbildung eine Einbettung von Σ in $\overline{\Sigma}$ definiert, allerdings nicht im engeren Sinne von Abschnitt 3.2.4. Es soll nun eine Reduktbildung bzgl. eines OA-Signaturmorphismus definiert werden (vgl. Definitionen 3.3.7 und 3.3.8).

Definition 5.4.2 *Sei $\sigma : \Sigma \to \overline{\Sigma}$ ein OA-Signaturmorphismus und $\overline{Mod}(AS)$ eine Klasse von $\overline{\Sigma}$-Algebren. Dann wird $\overline{Mod}(AS)|_{\Sigma}^{Alg}$ die* **erweiterte Reduktbildung** *genannt.*

$$\overline{Mod}(AS)|_{\Sigma}^{Alg} := \{A|_\sigma \mid A \in \overline{Mod}(AS)\}$$

ST wird bei erweiterter Reduktbildung auf $STATE$ abgebildet. Zustandsbezeichner werden wieder als freie Variablen und nicht mehr als Konstante behandelt.

Satz 5.4.1 *Sei $OS = <\Sigma, E>$ eine Objektspezifikation mit $\Sigma = <S \cup \{state\}, Z, OP>$ und AS die dazu konstruierte algebraische Spezifikation. Sei $\Sigma' = <S \cup \{state\}, Z, OP \cup \{pr1_s, pr2_s, asgn, if, seq \mid s \in S\}>$. Dann gilt:*

$$Mod(OS) \subset \overline{Mod}(AS)|_{\Sigma'}^{Alg}$$

Beweis:

1. Sei $O \in Mod(OS)$. Es ist zu zeigen, daß $O \in \overline{Mod}(AS)|_{\Sigma'}^{Alg}$ ist. Also

   (a) $O \in Alg(AS)|_{\Sigma'}^{Alg}$. O besteht aus:

   - Trägermengen A_s für die Sorten $s \in S$ sowie Trägermengen $State$, $IDENT = \{id_n \mid n \in Nat\}$ und $VAL = \bigcup_{s \in S} A_s$
   - einer Abbildung $STATE : IDENT \to VAL$ aus $State$
   - Funktionen $f : State \times A_{s_1} \times \ldots \times A_{s_n} \to State \times A_s$ für alle $op \in OP$
   - Funktionen (für alle $s \in S$)

     $$\begin{array}{lcl} asgn^O & : & State \times IDENT \times VAL \to State \\ if^O & : & State \times A_{bool} \times State \to State \\ seq^O & : & State \times State \times State \to State \\ pr1_s^O & : & State \times A_s \to State \\ pr2_s^O & : & State \times A_s \to A_s \end{array}$$

   Das Σ-Objekt O wird transformiert zu einer Algebra O', wobei folgende Veränderungen vorgenommen werden (bezeichnet durch $\mapsto$):

   i. $STATE : IDENT \to VAL \quad \mapsto \quad STATE' : State \times IDENT \to VAL$
      $STATE'$ erhält als zusätzlichen Parameter den abstrakten Zustand.

   ii. $f \mapsto f'$ für alle Operationen $f : state \times s_1 \times \ldots \times s_n \to state \times s$ mit
       $f' : State \times A_{s_1} \times \ldots \times A_{s_n} \to State$, falls f Prozedur ist,
       $f' : State \times A_{s_1} \times \ldots \times A_{s_n} \to A_s$, falls f Attribut ist,
       so daß $f' = pair_s \circ f$ gilt.

   O' ist somit als erweitertes Σ'-Redukt einer $\overline{\Sigma}$-Algebra bzgl. eines OA-Signaturmorphismus konstruiert worden (siehe Definition 5.4.2 und Konstruktion $\Sigma \mapsto \overline{\Sigma}$). O' ist also Element von $Alg(AS)|_{\Sigma'}$. O ist daher Element von $Alg(AS)|_{\Sigma'}^{Alg}$.

   (b) $O \in \overline{Mod}(AS)|_{\Sigma'}^{Alg}$. Die Gültigkeit der Axiome von AS ist in O zu zeigen. Dies folgt aus den Umsetzungen der Terme und den Überlegungen zur Gültigkeit (s.o.).

2. Die echte Inklusion gilt.
   Es gibt also Algebren in $\overline{Mod}(AS)|^{Alg}_{\Sigma'}$, die nicht in $Mod(OS)$ sind. Sei $\overline{A} \in \overline{Mod}(AS)|^{Alg}_{\Sigma'}$. Sei $ident$ in $\overline{A}$ durch $\overline{IDENT} = \{\overline{id_n} \mid n \in Nat\}$ interpretiert, so daß die Elemente von $\overline{IDENT}$ paarweise verschieden zu allen Elementen aus $IDENT$ sind (etwa für $\overline{IDENT} \subseteq NAT$). Mit dieser Konstruktion gilt $id_i \neq \overline{id_j}$ für beliebige $i, j \in Nat$. Damit ist ein Modell erzeugt, das sich von der Konstruktion der Σ-Objekte unterscheidet. Die echte Inklusion gilt also. □

Für Projektionen, Paarbildung oder Kommandorealisierungen finden sich Entsprechungen in den Σ-Objekten. Dort findet aber bzgl. dieser Elemente keine Modellklassenbildung statt, sondern nur eine einmalige Realisierung (modelltheoretische Sicht). Durch die axiomatische Spezifikation in AS sind mehrere Modelle möglich. So kann $state$ im algebraischen Ansatz durch beliebige Zustandsabstraktionen interpretiert werden.

Operationen können verschieden interpretiert werden, wenn sie partiell spezifiziert sind, oder wenn sie auf unterschiedlichen Wertebereichen realisiert werden (verschiedene Trägermengen pro Sorte). Aus diesem Überlegungen ergibt sich die echte Mengeninklusion.

Somit läßt sich feststellen, daß die Modellklassen bei Verwendung zustandsbasierter Spezifikation kleiner sind, da für die im algebraischen Ansatz zusätzlich notwendigen Funktionen $Projektion$ und ST durch den eigenschaftsorientierten Ansatz über verschiedenen Trägermengen mehrere Modelle möglich sind. In Σ-Objekten sind diese Funktionen direkt implementiert (modelltheoretisch definiert), somit also nur in einer Realisierung vorhanden.

Σ-Objekte erlauben im Kontext zustandsbasierter Systeme eine problembezogene Interpretation von Spezifikationen. Σ-Objekte sind an einen Zustandsbegriff angepaßt und somit kompakter. Die Modellklassen sind kleiner, und damit spezifischer, da sie gegenüber den Algebren die Zustandsbehandlung '*fest verdrahtet*' haben. Dies stellt eine Optimierung und keine Einschränkung in der Mächtigkeit des Ansatzes dar.

Der Zustand kann im Ansatz zustandsbasierter Spezifikation in den Spezifikationen selbst implizit bleiben. So muß keine Variable der Sorte $state$ explizit benutzt werden. Der aktuelle Zustand ist Parameter aller Operationsaufrufe und Kommandos. Dieser Parameter $state$ ist implizit in den Axiomen allquantifiziert. Da der aktuelle Zustand immer Parameter ist, kann auf seine explizite Benutzung in den Spezifikationen verzichtet werden (siehe explizite und implizite Terme in Abschnitt 3.2.3). Das Zustandskonzept an sich ist aber explizit, da es inhärent im Ansatz ist. Im algebraischen Ansatz müssen Zustände immer explizit gemacht werden, wie am Konstruktionsverfahren dieses Abschnittes zu sehen ist. Attribute werden durch Allquantifizierung über alle Zustände spezifiziert. Für Prozeduren müssen Vorzustände ggf. eingeschränkt werden. Nachzustände werden explizit konstruiert. Es gibt im algebraischen Ansatz kein inhärentes Zustandskonzept, daher muß die Zustandsbehandlung, wenn sie betrachtet werden soll, explizit sein.

5.4.2 Kripke-Modelle

Kripke-Modelle sind die üblichen semantischen Strukturen, um modale Logiken, wie z.B. dynamische oder temporale Logik, zu interpretieren [KT90, Har79, Har84, Sti91]. Ein Kripke-Modell für eine modale Aussagenlogik wird im folgenden definiert. Diese einfache Form dient der Veranschaulichung des Zustandskonzeptes. Dann werden Kripke-Modelle für die Prädikatenlogik definiert. Die Definitionen erfolgen nach [KT90]. Σ-Objekte sollen mit Kripke-

Modellen (nach [KT90]) verglichen werden. Neben einer semantischen Struktur erster Ordnung umfassen Kripke-Modelle Zustände. Diese sind je nach Form der Logik (aussagenlogisch oder prädikatenlogisch) abstrakt oder enthalten Bindungen von Bezeichnern an Werte. Sei zunächst eine dynamische Aussagenlogik wie in [KT90] Kapitel 2 angenommen. Diese Logik besteht aus Programmfragmenten $p, q, \ldots$ und Formeln $\phi, \psi, \ldots$, auf die Operatoren angewendet werden können. Zu den Operatoren gehören logische Konnektoren $(\neg, \vee, \ldots)$, Programmoperatoren (*Sequenz, Iteration, Auswahl*) sowie Mischoperatoren wie der modale Box-Operator.

Definition 5.4.3 *Ein* **Kripke-Modell** *für eine Aussagenlogik ist ein Paar $M = (S^M, I^M)$, wobei $S^M = \{u, v, \ldots\}$ eine Menge abstrakter Zustände und I^M eine Interpretationsfunktion ist.*

Eine Formel ϕ der Logik kann als Teilmenge $\phi^M \subseteq S^M$ und jedes Kommando (Programmfragment) p kann als binäre Relation p^M auf S^M eines Kripke-Modells M interpretiert werden. ϕ^M bezeichnet dann die Menge der Zustände, die ϕ erfüllen und p^M beschreibt die Ein-/Ausgaberelation von p.

Prädikatenlogiken werden über (semantischen) Strukturen interpretiert [Kre91]. Eine Struktur A (einsortig) besteht aus einer Trägermenge sowie Funktionen für alle Funktionssymbole und Relationen für alle Relationssymbole einer Signatur. Diese muß, um ein Kripke-Modell zu sein, um eine **Bewertung** u erweitert werden (sei Var eine Menge von Variablen); es muß also gelten:

$$u(x) \in A \quad für\ x \in Var$$

Durch Induktion kann die Bewertung auf beliebige Terme $t_1, \ldots, t_n$ erweitert werden. f sei ein Funktionssymbol und A eine Struktur.

$$u(f(t_1, \ldots, t_n)) := f^A(u(t_1), \ldots, u(t_n))$$

Mit $u[x \mapsto a]$ wird die Bewertung bezeichnet, die sich bei *Substitution* der Bindung für x ergibt (der Wert a wird an x gebunden).

Definition 5.4.4 *Zwei Bewertungen u, v sind* **endliche Varianten** *voneinander, wenn gilt:*

$$u(x) = v(x) \text{ für alle bis auf endlich viele Variablen } x \in Var$$

Definition 5.4.5 *Sei w eine Bewertung, die Variablen beliebig zuordnet (etwa auf Initialwerte gesetzt). Ein* **Zustand** *ist dann jede endliche Variante von w. Die Menge der Zustände der Struktur A wird mit S^A bezeichnet.*

Damit kann eine Zuweisung $asgn(x, t)$ als primitive Form einer Berechnung wie folgt auf einer Struktur A definiert werden:

$$(\ asgn(x, t)\ )^A := \{(u, u[x \mapsto u(t)]) \mid u \in S^A\}$$

Die definierte Relation *'ist endliche Variante von'* ist eine Äquivalenzrelation auf den Bewertungen. Terminierende Berechnungen können in endlicher Zeit nur endlich viele Zuweisungen ausführen (d.h. nur endlich viele neue Bewertungen produzieren). Eine Berechnung bleibt somit immer innerhalb einer Äquivalenzklasse. Ein Paar (u, v) ist dann Ein-/Ausgaberelation eines Programmes, wenn u endliche Variante von v ist.

Für deterministische Programmfragmente p ist die Semantik p^A einwertig, d.h. eine partielle Funktion auf Zuständen. Die Partialität ergibt sich, da Funktionen nicht terminieren müssen. Nichtterminierung entspricht hier einer nichtdefinierten Ein-/Ausgaberelation.

Definition 5.4.6 *Falls A eine Struktur und u ein Zustand in A ist, dann ist das Paar (A, u) eine* **Interpretation**. *Sei ϕ eine Formel. Man schreibt $A, u \models \phi$ für $u \in \phi^A$.*

Es soll nun eine Definition der Kripke-Modelle für die prädikatenlogische dynamische Logik angegeben werden. Die Definition orientiert sich an [KT90]. Dort wurden allerdings aus Vereinfachungsgründen nur einsortige Kripke-Modelle betrachtet (siehe [KT90] S. 792, Kapitel 1.1).

Definition 5.4.7 *Sei Var eine aufzählbare Menge individueller Variablen. Ein* **Kripke-Modell** *$M = (A, S^A)$ mit der Bewertung u für eine Prädikatenlogik über Sorten S und Funktionssymbolen F ist durch folgende Komponenten charakterisiert:*

1. *eine semantische Struktur A bestehend aus*
   - *Trägermengen A_s für alle Sorten $s \in S$,*
   - *einer n-stelligen Funktion $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ für jedes n-stellige Funktionssymbol $f : s_1 \ldots s_n \to s$ aus F,*
2. *eine Bewertung u über einer S-sortierten Menge von Variablen $Var = (Var_s)_{s \in S}$ mit*
   $$u(x) \in A_s \quad \textit{für jede Variable } x \in Var_s, s \in S;$$
   durch Induktion wird u auf beliebige Terme erweitert
   $$u(f(t_1, \ldots, t_n)) = f^A(u(t_1), \ldots, u(t_n))$$
   falls f ein n-stelliges Funktionssymbol ist.
3. *Sei w eine beliebige initiale Bewertung der Variablen aus Var. Ein Zustand ist dann jede endliche Variante von w. Die Menge der Zustände wird mit Z^A bezeichnet.*

Satz 5.4.2 *Σ-Objekte sind Kripke-Modelle.*

Beweis: Der Beweis erfolgt anhand von Definition 5.4.7 für Kripke-Modelle und Definition 3.3.1 für Σ-Objekte. Sei die geforderte Prädikatenlogik durch Objektsignaturen und sich kanonisch darauf aufbauenden Term- und Formelbegriffen definiert. Sei im folgenden $S^* = S \cup \{state\}$ die Menge der Sorten und F die Menge der Operationssymbole einschließlich der zusätzlichen Funktionen wie $\overline{asgn}$ (siehe Abschnitt 5.4.1). Es wird nun ein Kripke-Modell A aus Elementen eines Σ-Objekts O konstruiert.

1. $Var := IDENT$.
   Da Bezeichner z in Objektsignaturen in der Form $z : s$ angegeben werden müssen, kann $IDENT$ als S-sortiert angenommen werden.

2. $A_s := O_s$ für alle $s \in S^*$ und $f^A := \overline{f^O}$ für alle Operationssymbole.
   Die Abbildung $f \mapsto \overline{f}$ wurde in Abschnitt 5.4.1 beschrieben. Σ-Objekte bestehen aus Trägermengen für alle Sorten und Funktionen für alle Operationssymbole im geforderten Sinne (Definition 5.4.7(1)).

3. $u := STATE$.
   u wird im Σ-Objekt durch $STATE$ realisiert. $STATE$ bildet ebenfalls freie Variablen auf Trägerelemente ab. Die induktiv definierte Erweiterung von u für die Anwendung von Funktionen in Kripke-Modellen wird hier mit v bzw. v^* bezeichnet (siehe Definition 5.4.7(2)). Die Zuweisung wird in beiden Ansätzen über Substitution in den Belegungen ($STATE$ bzw. u) realisiert.

4. $\{u \mid u \text{ ist endliche Variante von } w\} := State$.
   Für beide Ansätze können initiale Belegungen mit undefinierten Werten angenommen werden. w soll die initiale Belegung für Kripke-Modelle bezeichnen.
   Die Menge der möglichen Zustände $State$ im Σ-Objekt wird basierend auf den Abbildungen $STATE$ (siehe Definition 3.3.1) gebildet. In Kripke-Modellen ist die Menge der Zustände ebenfalls durch die möglichen Belegungen definiert (siehe zweiter Spiegelpunkt in Definition 5.4.7). Da terminierende Berechnungen endlich sind, kann es (der Argumentation hinter Definition 5.4.5 folgend) nur endliche Varianten einer initialen Belegung der Variablen geben (Definition 5.4.7(3)).

Somit ist ein Kripke-Modell konstruiert. Alle Elemente der Σ-Objekte sind auch Elemente eines Kripke-Modells. □

Kripke-Modelle sind Paare aus einer semantischen Struktur erster Ordnung und einer Zustandsmenge. Σ-Objekte sind Erweiterungen einer semantischen Struktur erster Ordnung um einen Zustandsbegriff. Sie lassen sich allerdings (wie eben gezeigt) auch in Form von Kripke-Modellen darstellen. Σ-Objekte sind spezielle Kripke-Modelle. Σ-Objekte sind, wie Modelle bei algebraischen Spezifikationen auch, als eine singuläre Struktur (eine abstrakte Maschine) dargestellt. Σ-Objekte sind in ihrem Aufbau semantische Strukturen im Sinne einer Kollektion von Mengen und Funktionen, die durch Zuordnung von Trägermengen zu Sorten und Funktionen zu Operationssymbolen gebildet wird. Σ-Objekte sind spezieller als mehrsortige Kripke-Modelle, da ihre Mengen cpos und die Funktionen stetig sind. Damit lassen sich auf Σ-Objekten rekursive Operationen oder auch partielle Spezifikationen über das bottom-Element realisieren. Im Gegensatz zu Kripke-Modellen erlauben Σ-Objekte auch prozedurale Abstraktionen für die Zustandsübergänge (durch Definition von Prozeduren). Zudem können Attribute (Funktionen in Kripke-Modellen) und Prozeduren verschmolzen werden.

5.4.3 Beobachtungsorientiertes Spezifizieren

Der Ansatz algebraischer Spezifikation wird durch beobachtungsorientiertes Spezifizieren [ST87, Hen89] verallgemeinert. Der Gültigkeitsbegriff der Spezifikationslogik wird so erweitert, daß er sich nur auf ein irgendwie definiertes beobachtbares Verhalten stützt. Dadurch soll von internen Eigenschaften abstrahiert werden. Dies ist insbesondere von Bedeutung, wenn ein Implementierungsbegriff etabliert werden soll, der dann über den internen Eigenschaften abstrahieren kann. Korrekt ist in diesem Sinne eine Implementierung dann, wenn sie ein spezifiziertes beobachtbares Verhalten erfüllt. Der Ansatz geht auf D. Sannella und A. Tarlecki [ST87] zurück. Die hier vorgestellte Form orientiert sich an den Arbeiten von R. Hennicker [Hen89, Hen91c, Hen92].

Der Modellbegriff beobachtungsorientierten Spezifizierens

Es wird ein **Beobachtungsprädikat** *Obs* eingeführt, das wie das Gleichheitsprädikat als Standardprädikat jeder Spezifikation behandelt wird. *Obs* beschreibt beobachtbare Objekte in den Trägermengen. Eine *beobachtungsorientierte Spezifikation* SP ist eine Spezifikation, in der eine Teilmenge von Sorten als beobachtbar ausgezeichnet wird (die Sorten, die beobachtbare Elemente umfassen). Ein *Modell* von SP ist eine Struktur, die alle Axiome von SP bezüglich aller möglichen Beobachtungen erfüllt.

Eine semantische Struktur ist hier eine Algebra. Beobachtungen werden durch beobachtbare Kontexte repräsentiert. Σ sei eine Signatur. Ein **Σ-Kontext** ist ein Σ-Term, der genau eine spezielle Variable z_s für jede Sorte s enthält. Das *Obs*-Prädikat (ein Standardprädikat) kann auf beliebige Terme, die Trägerelemente bezeichnen, angewendet werden. Für jede Trägermenge ergibt sich eine beobachtbare Teilmenge. Spezielle Kontexte werden ausgezeichnet. Ein **beobachtbarer Σ-Kontext** ist ein Σ-Kontext mit beobachtbarer Ergebnissorte. Die speziellen Variablen z_s werden als Metavariablen für Terme bestimmter Sorten gedeutet. Ein Kontext c kann auf einen Term t angewendet werden.

$$c[t] := c[z_s/t] \quad \text{für } t : s$$

Die spezielle Variable z_s soll durch t substituiert werden. Die Anwendung beobachtbarer Kontexte sei am Beispiel eines *Stacks* verdeutlicht: $top(z)$ sei beobachtbarer Kontext mit ausgezeichneter Variable z. Dann muß das Axiom $pop(push(s,e)) = s$ nicht gelten, sondern nur die Anwendung eines beobachtbaren Kontextes auf die Gleichung, also etwa:

$$top(pop(push(s,e))) = top(s)$$

$pop(push(s,e))$ und s substituieren in dieser Gleichung z. Eine **beobachtungsorientierte Spezifikation** SP ist ein Paar $SP = < \Sigma, E >$ bestehend aus einer Signatur Σ und einer Menge E von Hornklauseln der Form

$$\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \phi_{n+1}$$

wobei die $\phi_i, i = 1, \ldots, n+1$ atomare Formeln sind. Eine atomare Formel ist entweder eine Gleichung $t_i = r_i$ oder eine Beobachtung der Form $Obs(t_i)$ mit Termen $t_i, r_i \in T(\Sigma, X)$. X ist eine Menge freier Variablen. Die Hornklauseln in E werden auch die Axiome von SP genannt. Sei $\Sigma = < S, F >$ eine Signatur. Eine **beobachtbare Σ-Algebra** ist ein Paar (A, Obs^A) bestehend aus einer totalen Σ-Algebra A und einer Familie $Obs^A = (Obs_s^A)_{s \in S}$ von Teilmengen $Obs_s^A \subseteq A_s$. Obs^A ist der **beobachtbare Teil** von A. Die Elemente von Obs^A heißen **beobachtbare Objekte**.

Die Gleichheit algebraischer Spezifikationen soll verallgemeinert werden: zwei durch Terme bezeichnete Elemente sind beobachtbar gleich, wenn sie durch zulässige Beobachtungen nicht unterschieden werden können. Die **beobachtbare Gültigkeit** $\models_{OBS}$ ist definiert durch:

$$(A, Obs^A) \models_{OBS} t = t' \quad \text{falls} \quad A \models c[t] = c[t']$$

für alle beobachtbaren Kontexte c. Alle anderen Konnektoren einer Gleichungslogik werden wie üblich definiert. Das *Beobachtungsprädikat* $Obs(t), t \in T(\Sigma, X)$, **gilt** in einer beobachtbaren Σ-Algebra (A, Obs^A) — kurz $(A, Obs^A) \models_{OBS} Obs(t)$ — genau dann, wenn für alle Bewertungen $v : X \rightarrow A$ die Interpretation von t durch v ein beobachtbares Objekt von (A, Obs^A) ist, d.h. $v^*(t) \in Obs^A$. Eine erreichbare beobachtbare Σ-Algebra (A, Obs^A) heißt **Modell** einer beobachtbaren Spezifikation SP genau dann, wenn (A, Obs^A) alle Axiome von SP erfüllt. $Mod_{OBS}(SP)$ ist die Klasse aller Modelle von SP. Die **Modellklasse** von $SP = < \Sigma, Obs, E >$ ist definiert durch (Alg_{OBS} ist die Klasse der beobachtbaren Σ-Algebren):

$$Mod_{OBS}(SP) := \{A \in Alg_{OBS}(\Sigma) \mid A \models_{OBS} \phi \text{ für alle } \phi \in E\}$$

Die **Implementierungsrelation** $\rightsquigarrow$ wird dann wie üblich durch Modellklasseninklusion definiert:

$$SP \rightsquigarrow IMPL \text{ gilt, falls } Mod_{OBS}(IMPL) \subseteq Mod_{OBS}(SP)$$

bei gleicher Signatur von SP und $IMPL$. SP wird durch $IMPL$ implementiert. Implementierungen werden hier erst später betrachtet. Beobachtungsorientierte Spezifikationen werden dann wieder aufgegriffen (Abschnitt 8.2.3).

Vergleich der Konzepte

Für den Vergleich zwischen beobachtungsorientiertem Spezifizieren und dem vorliegenden Ansatz soll die Anwendbarkeit des Beobachtungsprädikates Obs eingeschränkt werden. Es sollen immer (exklusiv) alle oder keine erreichbaren Objekte der Trägermenge einer Sorte beobachtbar sein. Im folgenden wird nur eine Teilmenge der Sorten $OBS \subseteq S$ angegeben, die beobachtbar sein sollen.

$$s \in OBS \;\Leftrightarrow\; Obs(t) \text{ für alle } t : s$$

Dieser einschränkende Ansatz wird auch in [Hen91b] und [Hen92] verfolgt. Dort läßt sich das Obs-Prädikat nur auf Sorten anwenden.

Um den vorliegenden Ansatz mit dem des beobachtungsorientierten Spezifizierens formal vergleichen zu können, müßten Spezifikationen des vorliegenden Ansatzes in eine algebraische Version umgeformt werden. Hierbei geht es um das Explizitmachen versteckter Konstrukte, wie dem Offenlegen der Typen der Zustandskomponenten, die sich hinter der Sammelbezeichnung *state* verbergen, oder dem Einführen expliziter Projektionsfunktionen. Die notwendigen syntaktischen Transformationen könnten sich an dem in Abschnitt 5.4.1 vorgestellten Transformationen orientieren. In diesem Abschnitt sollen jedoch nur die Konzepte verglichen werden.

Die Menge OBS der beobachtbaren Sorten für eine zustandsbasierte Spezifikation soll die Sorten umfassen, die den funktionalen Wert eines Attributes bestimmen:

$$OBS := \{s \in S \mid \exists a \in ATTR \;.\; a : state \times s_1 \times \ldots \times s_n \rightarrow state \times s\}$$

Damit wird deutlich, daß das Verhalten der Prozeduren transparent bleibt, d.h. der zustandsmodifizierende Effekt der Prozeduren kann mit dieser Modellierung nicht beobachtet werden. Beobachtbar sind lediglich Attributanwendungen auf modifizierte Zustände.

Die Idee, verschiedene Funktionalitäten von der Beobachtung auszuschließen, wird im vorliegenden Ansatz durch die Trennung von Attributen und Prozeduren und der Verkapselung des Zustands durch die Sorte *state* realisiert. Operationen auf beobachtbaren Sorten sind die Attribute. Operationen auf nichtbeobachtbaren Sorten sind die Prozeduren. Es liegt somit eine Anwendung der Idee beobachtungsorientierten Spezifizierens vor. Es werden Teile der

Spezifikation der Beobachtung entzogen. Betrachtet man das Problem aus Sicht der beobachtungsorientierten Spezifikation, so läßt sich eine beobachtungsorientierte Spezifikation mit dem vorliegenden Ansatz modellieren, indem Funktionen mit beobachtbarer Ergebnissorte als Attribute und die anderen als Prozeduren realisiert werden.

Vergleich der Benutzung der Ansätze

Nach der konzeptionellen Betrachtung der Unterschiede zwischen algebraischer und zustandsbasierter Spezifikation soll nun kurz auf die unterschiedliche Formulierung von Axiomen in beiden Ansätzen eingegangen werden.

Das Verhalten eines Stacks wird üblicherweise durch das Axiom

$$pop(push(s, e)) = s$$

für einen Stack s und ein beliebiges Element e in algebraischen Ansätzen beschrieben. R. Hennicker schlägt in seinem Ansatz vor, nur die Anwendung von *top* auf den Stack beobachtbar zu machen:

$$top(pop(push(s, e))) = top(s)$$

Im vorliegenden Ansatz würden *push* und *pop*, da sie den Stack verändern, als Prozeduren dargestellt werden. *top* wäre, da der Stack nicht verändert wird, ein Attribut. Die allgemeine algebraische Form läßt sich in dynamischer Logik durch

$$[\ seq(push(e), pop())\ ]\ s = old(s)$$

bzw.

$$s = X \rightarrow [\ seq(push(e), pop())\ ]\ s = X$$

beschreiben. Die beobachtungsorientierte Form läßt sich durch

$$top() = e' \rightarrow [\ seq(push(e), pop())\ ]\ top() = e'$$

spezifizieren, wobei e und e' zwei beliebige Elemente bezeichnen. Verzichtet man auf die Benutzung von Zustandsbezeichnern in den Vor- und Nachbedingungen, dann hat man eine beobachtungsorientierte Spezifikation (sofern nur Attributanwendungen als beobachtbar angesehen werden).

5.4.4 Temporale Logik und TLA

In diesem Kapitel wurde eine dynamische Logik zur Spezifikation zustandsbasierter Systeme vorgestellt. Dynamische Logiken sind exogene modale Logiken, d.h. Programmfragmente werden explizit. Daneben gibt es noch, wie schon in Abschnitt 5.1.1 angesprochen, endogene modale Logiken. Allen modalen, d.h. endogenen und exogenen, Logiken ist der Zustandsbegriff gemein. Damit sind prinzipiell auch endogene modale Logiken geeignet, zustandsbasierte Systeme zu spezifizieren und deren Verifikation zu erlauben. Der bedeutenste Vertreter dieser Richtung, die *temporale Logik* ([Eme90], [KT90] Kapitel 4.4) sowie ein Spezialfall, *TLA* (Temporal Logic of Actions [Lam94]), sollen kurz betrachtet werden.

Wie schon in Abschnitt 5.1.1 beschrieben, werden modale Operatoren in temporaler Logik nicht durch Programme indiziert. Modelle temporaler Logiken sind Folgen von Zuständen oder, bei nichtdeterministischen oder nebenläufigen Programmen, Bäume, deren Knoten Zustände sind. In temporalen Logiken wird

$$\Box\phi \text{ als } '\phi \; gilt \; in \; allen \; zuk\ddot{u}nftigen \; Zust\ddot{a}nden'$$

und

$$\Diamond\phi \text{ als } '\phi \; gilt \; in \; (irgend)einem \; zuk\ddot{u}nftigen \; Zustand'$$

interpretiert.

Hoare-Logiken und deren Erweiterungen zu dynamischer Logik zielen auf die Betrachtung von Programmen ab, deren Effekt durch Nachbedingungen spezifiziert werden kann. Für sogenannte reaktive Systeme, wie etwa Betriebssysteme, kann Terminierung nicht unbedingt vorausgesetzt werden. Es existiert nicht immer ein Folgezustand, der durch eine Nachbedingung spezifiziert werden könnte. Der *sometimes*-Operator $\Diamond$ und der *always*-Operator $\Box$ temporaler Logik sind geeignet, das Verhalten solcher reaktiver Systeme zu beschreiben. Nebenläufige Systeme sind ebenfalls ein breiter Anwendungsbereich für den Einsatz temporaler Logik.

Exogene und endogene modale Logiken sprechen unterschiedliche Sprachklassen an und sind auf die Betrachtung spezieller Probleme dieser Sprachklassen zugeschnitten. Dynamische Logik ist geeignet für die Spezifikation deterministischer, sequentieller imperativer Sprachen. Programme werden in ihrem Verhalten durch die Vor- und Nachbedingungen abstrahiert. In Hinsicht auf die Implementierung von Operationen in einer imperativen Sprache ist diese Ausdrucksmöglichkeit das wesentliche Spezifikationskonstrukt. Temporale Logik erlaubt diese direkte Abstraktion von Programmfragmenten nicht, da die Indizierbarkeit der modalen Operatoren mit Programmfragmenten fehlt. Für die Beschreibung und Verifikation nebenläufiger und reaktiver Systeme sind sogenannte *Lebendigkeits-* und *Sicherheitsaspekte* (*liveness, safety*) von Bedeutung. Diese Aspekte werden wir gleich bei der Betrachtung von TLA erläutern.

Endogene Logiken sind aus den aufgeführten Gründen also nicht geeignet, hier als Spezifikationsformalismus zu dienen. Temporale Logik ist geeignet, wenn weniger das Verhalten einzelner Operationen im Sinne einer Zustandstransition von Interesse ist, als vielmehr *Prozeßmodellierung* betrieben werden soll. Prozesse und ihre Abhängigkeiten können abstrakt und deklarativ spezifiziert werden, während dynamische Logik eine operationalere, auf einfachen Zustandstransitionen basierende Spezifikation ermöglicht.

TLA als pragmatische Ausprägung einer temporalen Logik soll trotzdem aufgegriffen werden, da Lebendigkeit und Sicherheit bei möglichen Erweiterungen der hier vorgestellten Logik von Interesse sein könnten. TLA ist als Logik zur Spezifikation und Verifikation von nebenläufigen Systemen entworfen worden. Gegenüber üblicher temporaler Logik wird in TLA versucht, die im Vergleich zur Logik erster Stufe durch die modalen Operatoren hinzukommende Komplexität in Grenzen zu halten. TLA soll bei der Spezifikation nebenläufiger Algorithmen helfen, wesentliche Eigenschaften wie Lebendigkeit, Sicherheit und Fairness besser formulieren zu können, als es mit nebenläufigen Programmiersprachen der Fall ist. Als Primitive der Logik gibt es in TLA neben Prädikaten auch Aktionen. Aktionen sind Relationen zwischen Zuständen. Eine Aktion A gilt in einer Folge von Zuständen, falls das erste Zustandspaar der Folge der Relation A entspricht.

Sicherheitseigenschaften sollen garantieren, daß bestimmte, unerwünschte Bedingungen nicht eintreten. Dies kann etwa mit Hilfe des *always*-Operators $\Box$ spezifiziert werden:

$$\phi \equiv Init \vee \Box\psi$$

ϕ soll garantieren, daß bei Erfüllung einer Initialbedingung $Init$ dann immer ψ gilt (und damit etwas Unerwünschtes ausgeschlossen wird). Die Formel

$$\phi \equiv \Box\Diamond\psi$$

wird als *'unendlich-oft'*-Formel (*infinitely often*) bezeichnet. $\Box\Diamond\psi$ soll in allen folgenden Zuständen gelten. Damit muß ψ irgendwann in der Zukunft jedes Folgezustandes gelten. ϕ beschreibt eine Lebendigkeitseigenschaft. Die erwünschte Bedingung ψ tritt irgendwann ein (*ψ eventually holds*). Fairness-Bedingungen[14] lassen sich ebenfalls mit den temporalen Operatoren ausdrücken (vgl. [Lam94] Kapitel 5.3).

5.5 Zusammenfassung

In diesem Kapitel wurde eine exogene modale Logik, und damit eine Erweiterung einer Programmlogik, eingeführt. Es handelt sich dabei um eine dynamische Logik. Anpassungen der dynamischen Logik, wie sie in urprünglicher Form von Harel [Har79, Har84] oder in [KT90] vorgestellt wurde, an spezielle Bedürfnisse sind in vielen Spezifikationsansätzen zu finden. Es seien hier CMSL [Wie91], die COLD-Logik MLCM [GRdL94] oder MAL [FMR91] genannt. Mit Einschränkungen sind diese Ansätze auch mit den Vor-/Nachbedingungsspezifikationen von VDM oder Z vergleichbar. Π stellt ebenfalls Konzepte zur zustandsbasierten Spezifikation bereit. Diese sind allerdings rein operational (imperativ). Die Betrachtungen in Abschnitt 5.4.4 haben gezeigt, daß exogene modale Logiken wie die dynamische Logik, die geeignetere Form modaler Logik für den gegebenen Kontext sind. Bei einer Erweiterung der hier vorgestellten Form könnten allerdings Anregungen aus den temporalen Logiken eingehen.

Die Semantik der Logik wurde auf dem in den vorangegangenen Kapiteln vorgestellten Berechnungsmodell der Σ-Objekte definiert. Die Verknüpfung zwischen Spezifikation und den semantischen Strukturen erfolgt durch eine Gültigkeitsrelation. Begriffe wie Erreichbarkeit, Modell oder Theorie wurden eingeführt; ihre Eigenschaften sind untersucht worden. Die Modellbildung erfolgt hier eigenschaftsorientiert (wie auch in algebraisch orientierten Ansätzen sowie in COLD und CMSL). Modelltheoretisch wird hingegen in VDM, Z oder den Interface-Sprachen von Larch verfahren. Dort bezeichnen Spezifikationselemente genau ein mathematisches Modell.

Die Logik wurde in zwei Schritten eingeführt. Im ersten Schritt wurde eine dynamische Logik vorgestellt, die in etwa der Hoare-Logik (einer Programmlogik) entspricht und dazu dienen soll, nahe an möglichen Implementierungen spezifizieren zu können. Die Erweiterung zu einer vollwertigen modalen Logik erfolgte in einem zweiten Schritt, um hier Elemente der Logik hinzuzunehmen, die eine Spezifikation von Abhängigkeiten zwischen Operationen und von Eigenschaften, die Zustandsfolgen betreffen, erlauben. Eine weitere Erweiterung der Logik könnte in Richtung der in MAL realisierten deontischen Logik erfolgen. Aspekte temporaler Logik könnten betrachtet werden.

Die definierte Gültigkeitsrelation realisiert eine partielle Korrektheitsaussage. Aussagen zur Terminierung werden hier vernachlässigt. Durch den Diamond-Operator sind zwar Terminierungsaussagen möglich, dieser wurde aber nur durch Negation des Box-Operators auf syntaktischer Ebene definiert. Eine nähere Betrachtung erfolgt hier nicht. Untersuchungen hierzu sind in [KT90] Kapitel 3.2 und 3.3 zu finden.

[14] Eine Operation wird auch irgendwann ausgeführt, falls die Ausführung möglich ist.

Der Vergleichsteil dieses Kapitels wurde mit einer Betrachtung algebraischer Spezifikation begonnen. Der Vergleich mit Algebren zeigt, daß hier im Kontext zustandsbasierter Systeme Modellklassen kleiner sind. Die Modellbildung ist hier präziser, da problemnah argumentiert werden kann. Es wurde gezeigt, daß Σ-Objekte auch Kripke-Modelle sind, daß sie also den üblichen semantischen Strukturen einer modalen Logik entsprechen. Ergebnisse aus diesem Bereich können somit übernommen werden. CMSL und MAL werden auf Kripke-Modellen definiert. COLD basiert auch *Description Algebras* (mehrsortige Algebren, die insbesondere Konzepte zur Definition von Modularisierungskonzepten umfassen). Es erfolgte außerdem ein Vergleich mit dem Ansatz beobachtungsorientierten Spezifizierens, der gezeigt hat, daß Aspekte des Beobachtens und Verbergens auch hier realisiert sind. Die Betrachtung temporaler Logik bestätigte die Auswahl dynamischer Logik als der geeigneteren Form einer modalen Logik.

[FJ92] führen *shapes* für Axiome ein, d.h. eine Art Klassifikation von Axiomen (Invarianten, Vor-/Nachbedingungen, initialer Zustand). Ähnliches könnte auch hier versucht werden. Dies gehört aber eher in den Bereich der Gestaltung einer Spezifikationssprache als der Gestaltung einer Spezifikationslogik.

Eine interessante Erweiterung läge in der Betrachtung der Verknüpfung von algebraischer Spezifikation und zustandsbasierter Spezifikation. Hier sind algebraische Spezifikationen als die abstrakteren anzusehen, in die dann Zustands- oder Speicherungsaspekte eingebaut werden sollen. Ähnliche Ansätze sind in [Bre91, Lin93] und den Larch-Sprachen [Win87] zu finden. Lin schlägt eine Umsetzung algebraischer Spezifikationen in prozedurale Vor-/Nachbedingungsspezifikationen vor.

Kapitel 6

Spezifikationslogik und Implementierung

6.1 Motivation

Ziel dieses Kapitels ist die Verknüpfung abstrakter Spezifikation und konkreter, imperativer Programmierung. Durch die imperativen Beschreibungselemente der Programmiersprache werden als Semantik *Standardmodelle* vorgegeben. Die Semantik dafür ist in denotationaler Form definiert. Trägermengen für Basisdatentypen (z.B. $Integer, Boolean$) sind cpos, Funktionen darauf sind stetig. Um Spezifikation und Implementierung verknüpfen zu können (etwa um weitere Datentypen axiomatisch spezifizieren zu können), sollen Eigenschaften der Implementierungsmodelle auch in den Spezifikationen berücksichtigt werden. Dazu wird eine Reihe von Konzepten eingeführt (vgl. auch [WB89b]).

- Es kann eine *Ordnung* auf den Trägermengen definiert werden. Aus der Ordnung leitet sich eine Äquivalenzrelation ab, die nicht unbedingt die Identität auf den Trägermengen sein muß.
- *Gleichheit* kann explizit definiert werden.
- Jede Signatur kann um ein Element $\perp$ erweitert werden, welches das *bottom*-Element von cpos darstellen soll. *Striktheit* von Funktionen kann damit gefordert werden. *Partialität* kann modelliert werden.
- Es können *Aussagen über erreichbare Trägerelemente* gemacht werden.

Damit lassen sich im Ansatz verschiedene Konzepte verwirklichen. Bisher wurde eine totale Definition von Operationen angenommen. **Partialität** soll über das *bottom*-Element der Halbordnungen modelliert werden. Der sehr lose Modellbegriff erlaubt nichterreichbare Modelle. Um mit syntaktisch basierten Beweissystemen arbeiten zu können (siehe Gödelsches Vollständigkeitstheorem, [Wir90]) ist die Spezifikation von Eigenschaften der erreichbaren Teile notwendig. In diesen erreichbaren Teilen ist dann **Verifizierbarkeit** gegeben. Durch die Möglichkeit, die Gleichheitsrelation als beliebige Äquivalenzrelation und nicht unbedingt als Identität zu definieren, werden **flexiblere Definitionen neuer Datentypen** auf bestehenden möglich.

Es werden nur flache Halbordnungen und strikte Funktionen spezifizierbar gemacht. Die Existenz der Suprema in den Halbordnungen und die Erhaltung der Suprema durch Funktionen soll hier nicht betrachtet werden. Diese Problematik wird in der Zusammenfassung des Kapitels angesprochen.

Einige der geforderten Konstrukte könnten direkt in den Modellen realisiert werden. So könnten nur erreichbare Modelle zugelassen werden. Auf eine explizite Beschreibung der Ordnung könnte verzichtet werden, wenn die semantischen Strukturen geeignet eingeschränkt würden, etwa dürften Trägermengen dann nur vollständige Halbordnungen sein. Damit würden nur *Standardmodelle* als Modelle akzeptiert. Daß statt dieser Einschränkungen explizite Konstrukte zum Teil notwendig sind, zum Teil die Mächtigkeit des Ansatzes deutlich steigern, wird in den folgenden Abschnitten konzeptionell (6.2 und 6.3) und an einem Beispiel zur axiomatischen Spezifikation eines Datentypen (6.4) erläutert. In diesem Abschnitt 6.1 sollen die neuen Konzepte noch detaillierter motiviert werden.

6.1.1 Partialität

Fehlerfälle sind Situationen, in denen Terme keine Interpretation haben (die Interpretation der Terme ist in diesem Fall *partiell*). Operationen können durch partielle Funktionen modelliert werden. Nichtterminierende Berechnungen (basierend auf rekursiven Definitionen) führen zu partiellen Funktionen. In frühen Stadien der Spezifikation ist es einem Spezifizierer nicht immer zuzumuten, seine Operationen total zu definieren, d.h. inklusive aller Ausnahme- und Fehlerfälle. Partielle Funktionen als Modelle führen dazu, daß die Interpretation v^* geeignet angepaßt werden muß. Eine Operationsanwendung $op(t_1, \ldots, t_n)$ wird durch den undefinierten Wert ω interpretiert, falls einer der Terme $t_i, i = 1, \ldots, n$ undefiniert und keine explizite Behandlung dafür vorgegeben ist. Das heißt die Interpretation von Grundtermen kann zu undefinierten Situationen führen. Durch Ansätze wie *partielle Funktionen* [BW82] oder *stetige Algebren* [Wir90] Kapitel 3.3 kann eine solche Situation explizit beschrieben werden.

Zur Beschreibung nichttotaler Funktionen sollen die beiden wichtigsten Ansätze kurz vorgestellt werden:

- *Partielle Algebren bzw. Objekte.* Es soll durch Formeln ausdrückbar sein, ob ein Term definiert ist. Ansätze wie [BW82] erweitern konventionelle algebraische Spezifikationen wie folgt. Atomare Formeln sind dort Gleichungen oder Definiertheitsformeln $D_s(t)$. Dieser Ansatz ist auch in Abschnitt 5.4.1 beschrieben. D_s ist ein Definiertheitsprädikat. Eine partielle Algebra A erfüllt das Definiertheitsprädikat $D_s(t)$, wenn die Interpretation von t in A definiert ist. Die Spezifikationsansätze [Bre91, FJ92] sowie die Sprache des CIP-Projekts [BBB$^+$85] sind über lose, partielle Semantiken definiert.

- *Stetige Algebren bzw. Objekte.* Ansätze sind in [Möl85] oder [Wir90] Kap. 3.3.3 beschrieben. Die Trägermengen bilden cpos und die Operatoren auf den cpos sind stetig. Klassen stetiger Algebren/Objekte werden durch Axiome beschrieben, die auf Ungleichungen $t \sqsubseteq t'$ aufgebaut sind. Semantisch definiert $\sqsubseteq$ eine Halbordnung auf den Trägermengen, d.h. eine reflexive, antisymmetrische, transitive und mit den Interpretationen der Funktionssymbole kompatible Ordnung. Zu jeder Sorte s gibt es eine Konstante $\perp_s$, die als bottom-Element interpretiert wird: $\forall x : s \,.\, \perp_s \sqsubseteq x$.

6.1.2 Erreichbarkeit und Beobachtbarkeit

Es wird hier ein sehr loser Semantikansatz verfolgt, d.h. nicht alle Trägerelemente der Modelle müssen durch Terme darstellbar, also erreichbar sein[1]. Zur Definition des Modellbegriffs wurde auf die Erreichbarkeit verzichtet, um einen möglichst weitgefaßten Modellbegriff zu erhalten. Es werden alle Objekte als Modelle anerkannt, deren Verhalten in bezug auf die Spezifikation korrekt ist; evtl. vorhandene, nicht erreichbare Trägerelemente werden, da sie keinen Einfluß auf das Verhalten haben, in der Modellbildung nicht explizit ausgeschlossen.

In einem Ansatz, der die Entwicklung von Spezifikationen bis hin zur Implementierung formal unterstützen will, ist es an dieser Stelle wichtig, Modellbildung für axiomatische und imperative Spezifikationen zu integrieren. In Programmiersprachen werden konkrete Modelle für vordefinierte Typen, wie z.B. einer Implementierung ganzer Zahlen mit entsprechenden Funktionen, definiert. Diese bestehen aus cpos und darauf stetigen Funktionen, falls der Ansatz der Kapitel 3 und 4 zugrundeliegt. Ein Software-Entwicklungsansatz, der den Übergang von der Spezifikation zur Implementierung umfaßt, sollte es ermöglichen, die speziellen Modelle vordefinierter imperativer Typen als Modelle axiomatischer Spezifikationen zu akzeptieren. Sinnvoll ist der sehr lose Ansatz, wenn z.B. die üblichen Modelle eines Datentyps $Integer$ (also Funktionen basierend auf der Menge der ganzen Zahlen) auch zur Modellbildung einer Spezifikation der booleschen Werte *Bool* herangezogen werden können. Aus einer *Bool*-Spezifikation sollen die Konstanten $true$ und $false$ sowie die Operation *not* realisiert werden. $true$ läßt sich jetzt etwa durch die ganze Zahl 0 und $false$ durch 1 realisieren. *not* liefert dann für das Argument 0 die 1 und umgekehrt. Alle anderen ganzen Zahlen sind mit $true$, $false$ und *not* nicht erreichbar. Durch Anwendung eines *Beobachtbarkeitsprädikats Obs*

$$Obs_{\{true,false,not\}_{int}}(i)$$

wird ausgesagt, daß nur die durch $true$, $false$ und *not* erreichbaren Elemente der Trägermenge von *int* beobachtbar sind.

Wie sich an diesem Beispiel schon zeigt, ist der Verzicht auf Erreichbarkeit ein Schritt zu einem weitreichenderen Modellbegriff und auch Grundlage eines mächtigen Implementierungsbegriffes, der auf dem Modellbegriff aufsetzt. Auch durch den Verzicht auf Erreichbarkeit ist das geforderte Verhalten der Modelle garantiert. Mit dieser Definition ist jedoch ein Problem verbunden. Es sind Beweise durch strukturelle Induktion auf den Trägermengen auf Basis der syntaktischen Struktur der Termbezeichnungen nicht möglich. Also können syntaktisch basierte Beweissysteme nicht eingesetzt werden. Dem Problem kann begegnet werden, indem das Beobachtbarkeitsprädikat *Obs* eingesetzt wird, mit dem Aussagen über (mit bestimmten Grundtermen) erreichbare Trägerelemente gemacht werden können. Damit kann dann durch

$$Obs_{\{true,false,not\}_{int}}(x) \rightarrow (not(not(x)) = x)$$

spezifiziert werden, daß $not(not(x)) = x$ nur in einer mit den booleschen Operatoren $true$, $false$ und *not* erreichbaren Teilstruktur der Trägermenge von *int* gelten soll. Es kann dann in dieser Teilstruktur verifiziert werden.

Die Wahl der Benennung '*Beobachtbarkeitsprädikat*' deutet schon an, daß hier mehr als nur Erreichbarkeit betrachtet werden soll. Der Begriff der Beobachtbarkeit stammt aus den Ansätzen des beobachtungsorientierten Spezifizierens [ST87, Hen89] (siehe Abschnitt 5.4.3). Über die Beschreibbarkeit von Trägerelementen durch beliebige Terme (wie es Erreichbarkeit fordert)

[1] Fragen zur Erreichbarkeit werden in [Bre91, Wir90, Möl85, FJ92] diskutiert.

hinaus werden dort bestimmte syntaktische Konstruktionen als beobachtbar gekennzeichnet. Eine ähnliche Verallgemeinerung des Erreichbarkeitsbegriffes soll auch hier betrachtet werden.

6.1.3 Das Gleichheitsprädikat

Ein Gleichheitsprädikat ist üblicherweise in fast allen Spezifikationen enthalten. Es nimmt daher eine herausgehobene Stellung ein. In der Regel wird die Gleichheit durch die Identität auf den Trägermengen interpretiert. In einigen Situationen ist es aber auch sinnvoll, Gleichheit als eine beliebige Äquivalenzrelation definieren zu können. Zur Veranschaulichung sei eine Spezifikation von Sequenzen angenommen. Die Spezifikation von Sequenzen soll so erweitert werden, daß sie ein Gleichheitsprädikat im Sinne von Mengen enthält. An diesem Beispiel sieht man besonders deutlich die Notwendigkeit der expliziten Definierbarkeit des Gleichheitsprädikats. Alle Permutationen von Elementen einer Sequenz sind im Sinne der Mengen gleich. Gleichheit sollte also nicht nur als Identität auf den Trägermengen interpretiert werden können. Die Gleichheitsrelation sollte aber auf jeden Fall eine Äquivalenzrelation bilden.

Es kann sinnvoll sein, die Gleichheitsrelation anwendungsspezifisch spezifizieren zu können. Diese Feststellung geht noch über das Beispiel der Gleichbehandlung von Sequenzen hinaus. Hier ist etwa an komplex strukturierte Objekte der Realität zu denken, deren Gleichheit im Sinne der Anwendung evtl. nicht von allen modellierten Eigenschaften abhängt.

6.2 Definition des erweiterten Spezifikationsansatzes

In diesem Abschnitt werden die Spezifikationskonzepte des Kapitels 5 um die am Anfang dieses Kapitels schon informell vorgestellten Konstrukte erweitert. Die betrachteten Prädikate sind teilweise benutzerdefinierbar. Daß für diese immer eine Standardinterpretation gefunden werden kann, die mit den anderen Prädikaten verträglich ist, wird in Abschnitt 6.3 gezeigt. Eine beispielhafte Anwendung ist Inhalt von Abschnitt 6.4. Dieser Abschnitt enthält die notwendigen Definitionen.

Definition 6.2.1 *Eine* **Prädikatsignatur** $\Sigma^P = < S, Z, P, OP >$ *ist eine Objektsignatur* $\Sigma = < S, Z, OP >$ *erweitert um eine Menge von Prädikatensymbolen* P *mit* $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times bool$ *für* $p \in P, s_i \in S \setminus \{state\}, i = 1, \ldots, n$. *Die Prädikate sind — wie für beliebige Operationen üblich — in der kanonischen Form, also auch auf der Sorte state definiert. Da Prädikate als Attribute zustandsunabhängig sind, wird im folgenden der Zustandsparameter weggelassen. Wir fordern*

$$st_1 \neq st_2 \quad \Rightarrow \quad p(st_1, a_1, \ldots, a_n) = p(st_2, a_1, \ldots, a_n)$$

für beliebige Zustände st_1, st_2, *beliebige Argumente* a_i, *und ein Prädikat* $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times bool$. *Prädikate können so in der gewohnten Schreibweise benutzt werden. Da Prädikate Relationen auf Trägermengen definieren, wird auch die Schreibweise* $x \in p$ *für* $p(x) = true$ *verwendet (p sei ein einstelliges Prädikat).*

Prädikate werden als boolesche Operationen definiert. Daher werden sie auch als Elemente von OP behandelt, so daß $P \subseteq OP$ angenommen wird.

Definition 6.2.2 *Sei* $\Sigma = < S, Z, OP >$ *eine Signatur. Die Menge der* **Standardprädikate** *StP für Σ besteht aus:*

- *einem Prädikat $\sqsubseteq_s$: state $\times$ s $\times$ s $\rightarrow$ state $\times$ bool für jede Sorte $s \in S \ \cup \ \{state\}$,*
- *einem Prädikat $Obs_{\Sigma'}$: state $\times$ s $\rightarrow$ state $\times$ bool für jede sinnvolle Subsignatur*[2] *Σ' von Σ.*

Subsignaturen sollen stets sinnvoll sein (wie auch in Kapitel 3.2 gefordert).

An dieser Stelle werden lediglich syntaktische Vereinbarungen getroffen. Wie die mit der Benennung intendierten Eigenschaften durch geeignete Interpretationen in Σ-Objekten realisiert werden können, soll später betrachtet werden.

Bemerkung 6.2.1 *Als Abkürzung soll das Prädikat $\equiv_s$: state $\times$ s $\times$ s $\rightarrow$ state $\times$ bool für jede Datensorte s, definiert durch*

$$a \equiv_s b \ := \ a \sqsubseteq_s b \wedge b \sqsubseteq_s a$$

eingeführt werden.

. $\sqsubseteq_s$. soll anzeigen, daß zwei Elemente einer Sorte vergleichbar sind. $Obs_{\Sigma'}$. soll anzeigen, daß ein Element durch einen Σ'-Grundterm bezeichnet wird. Mit *Obs* (für *observable*) soll ausgesagt werden, daß ein Trägerelement mit bestimmten Termen (der Subsignatur) beschreibbar oder beobachtbar ist. Der Ausdruck '*beobachtbar*' geht auf [ST88] und [Hen89] zurück. . $\equiv_s$. soll die Gleichheit zweier Elemente auf Trägermengen definieren. $\equiv_s$ soll also mit $\sqsubseteq_s$ verträglich sein. Falls keine Mehrdeutigkeiten entstehen, wird der Index s bei $\sqsubseteq$ und $\equiv$ auch weggelassen. $\sqsubseteq_s$ wird auch als **Ordnungsprädikat**, $\equiv_s$ als **Gleichheitsprädikat** und $Obs_{\Sigma'}$ als **Beobachtbarkeitsprädikat** bezeichnet.

$\sqsubseteq$ soll eine Halbordnung beschreiben, $\equiv$ eine Äquivalenzrelation. $\equiv$ muß also nicht unbedingt durch die Identität auf den Trägermengen interpretiert werden. Wie Prädikate so definiert werden können, daß die gewünschten Eigenschaften gelten, wird später untersucht. Das Prädikat $\sqsubseteq$ ist das grundlegendere gegenüber $\equiv$ (siehe Bemerkung 6.2.1). Trotzdem sollen beide Standardprädikate $\equiv$ und $\sqsubseteq$ durch den Spezifizierer definierbar sein. Für das Beobachtbarkeitsprädikat ist eine Definition durch den Spezifizierer nicht sinnvoll. Es wird daher mit einer festen Interpretation versehen.

Da über Erreichbarkeit oder Beobachtbarkeit häufig im Kontext einer Sorte bzw. spezieller Operationssymbole einer Sorte gesprochen wird, sollen drei Formen der Beschreibbarkeit der Signatur Σ' des Prädikats $Obs_{\Sigma'}$ angeboten werden:

- $\Sigma_{<S',Z',OP'>}$: Angabe einer beliebigen Subsignatur.
- Σ_s : Angabe eines Sortenbezeichners s. Es sind dann alle Terme dieser Sorte gemeint.
- $\Sigma_{\{op_1,\ldots,op_n\}_s}$: Angabe von speziellen Operationssymbolen op_i einer Sorte s.

Mit $Obs_{\Sigma'}$ und $\Sigma = \Sigma'$ wird der Erreichbarkeitsbegriff bezüglich Σ modelliert. Im Ansatz von R. Hennicker werden zu beobachtende Konstrukte durch *Kontexte* definiert. Hier sind Subsignaturen der Mechanismus, zu beobachtendes Verhalten festzulegen.

[2] Siehe Definitionen 3.2.4 und 3.2.3.

Definition 6.2.3 *Eine* **Standardsignatur** $\Sigma^{Std} = < S, Z, P, OP >$ *ist eine Prädikatsignatur mit*

- $P = StP$ *(die Standardprädikate),*
- *für jedes* $s \in S \ \cup \ \{state\}$ *existiert eine Konstante* $\perp_s : state \rightarrow state \times s$.

$\perp_s$ soll das bottom-Element in der zugehörigen Trägermenge bezeichnen. Mit $. \sqsubseteq_s .$ kann es als das kleinste Element spezifiziert werden. $\perp_s$ kann auch dazu herangezogen werden, partielle Funktionen zu modellieren. $\perp_s$ repräsentiert dann das undefinierte Element ω.

Definition 6.2.4 *Ein* **geordnetes Σ-Objekt** A^{ord} *besteht aus*

- *Halbordnungen* $(A_s, \sqsubseteq_{A_s})$ *über nichtleeren Trägermengen* A_s *für jedes* $s \in S \setminus \{state\}$,
- *totalen, strikten Funktionen* op^A *für jedes* $op \in OP$ *(und* p^A *für jedes* $p \in P$, *falls* Σ *eine Prädikatsignatur ist),*
- *einer Abbildung* $STATE \in State$.

$(State, \sqsubseteq)$ ist nach Lemma 4.1.3 eine cpo. Ein Σ-Objekt A ist erreichbar bzgl. der Datensorten, wenn für alle $s \in S \setminus \{state\}$ und $a \in A_s$ ein Σ-Grundterm $t \in T(\Sigma)_s$ mit $a = t^A$ existiert, wenn also alle Trägerelemente erreichbar sind (siehe Definition 5.2.5).

Definition 6.2.5 *Sei* A *ein* Σ*-Objekt. Dann heißt* A

1. **flach**, *wenn die Trägermengen* A_s *für alle* $s \in S$ *je ein bottom-Element* $\perp_{A_s}$ *mit* $\perp_{A_s} \sqsubseteq a$ *für alle* $a \in A_s$ *haben, das Interpretation der Standardkonstante* $\perp_s$ *ist.* $b \sqsubseteq a \ \wedge \ a \neq b \Rightarrow b = \perp_{A_s}$, *d.h.* $b \sqsubseteq a$ *für* $b \neq \perp_{A_s}$ *soll nicht erlaubt sein. Ein flaches* Σ*-Objekt* A *heißt* **strikt**, *wenn die Funktionen des Objekts, die Operationssymbole interpretieren, strikt auf den Trägermengen sind.*
2. **identitätsbasiert**, *wenn das Prädikat* $. \equiv .$ *durch die Identitätsrelation auf den Trägermengen interpretiert wird.*
3. **standard**, *wenn es geordnet, erreichbar und identitätsbasiert ist.*

Folgerung 6.2.1 *Ein flaches, striktes* Σ*-Objekt ist geordnet.*

Beweis: Nach Definitionen 6.2.4 und 6.2.5. Alle Trägermengen für die Sorten $s \in S$ sind hier flache cpos durch die Konstruktion in 6.2.5(1). Strikte Σ-Objekte bestehen aus strikten Funktionen. Die Abbildung $STATE$ ist Bestandteil jedes Σ-Objekts. □

Die in Kapitel 3.1 definierten Modelle zu den vordefinierten Datentypen *Integer* oder *Boolean* sind Standard-Σ-Objekte im Sinne von Definition 6.2.5. Mit den Begriff der Standard-Σ-Objekte werden Definitionen für Datentypen charakterisiert, die im Stil der denotationalen Semantik vorgenommen werden. Datentypen in denotationaler Semantik werden durch Angabe von Standardmodellen definiert (vgl. auch modelltheoretische Ansätze wie VDM oder Z). In diesem Ansatz werden die Standardmodelle durch Standard-Σ-Objekte repräsentiert. Nehmen wir nun an, daß für eine Programmiersprache mit einer festen Menge von Datentypen, die durch Standardmodelle definiert sind, weitere Datentypen axiomatisch in einem bottom-up Vorgehen realisiert werden sollen. Dann ist es u.U. notwendig, über Ordnung, Erreichbarkeit und Gleichheit explizit Aussagen machen zu können. Nur so können Standardmodelle für Datentypen auch axiomatisch spezifiziert werden.

Inwiefern die Logik aus Kapitel 5 zu diesem Zweck zu verändern ist, soll nun untersucht werden.

Formeln (Definition 5.1.3) müssen modifiziert werden. Dazu wird eine neue Formelmenge $WFF_P(\Sigma)$ definiert, die die Prädikate berücksichtigt. Falls $p \in P$ ein Prädikat ist und die t_i Σ-Terme sind, dann ist

$$p(t_1, \ldots, t_n) \in WFF_P(\Sigma)$$

Σ-Gleichungen im bisherigen Sinne gibt es nicht mehr. Sie werden durch Prädikate verallgemeinert. Somit ist der Prädikatausdruck die einzige atomare Formel.

Definition 6.2.6 *Die Menge* $WFF_P(\Sigma)$ *der* **wohlgeformten Formeln mit Prädikaten** $Pred = \{\sqsubseteq, \equiv, Obs\}$ **über** Σ *ist die* $*$*-Hülle (siehe Definition 5.1.2) der logischen Konnektoren* $\{\neg, \wedge, \forall, [\,]\}$ *über*

$$\begin{array}{lll} Pred & := & \{ \sqsubseteq (t_1, t_2) \mid t_1, t_2 \in T(\Sigma)_s, s \in S \cup \{state\} \} \\ & \cup & \{ \equiv (t_1, t_2) \mid t_1, t_2 \in T(\Sigma)_s, s \in S \cup \{state\} \} \\ & \cup & \{ Obs(t) \mid t \in T(\Sigma)_s, s \in S \} \end{array}$$

Es sind auch die infix- bzw. postfix-Schreibweisen $t_1 \sqsubseteq t_2$ und $t_1 \equiv t_2$ erlaubt.

Zusätzlich sollen noch zwei häufig benutzte Abkürzungen eingeführt werden.

$$\forall x : s.\ Obs_\Sigma(x) \rightarrow \phi, \quad kurz: \quad \forall x : \Sigma.\ \phi$$

Für alle Elemente gilt ϕ oder sie sind nicht erreichbar, d.h. für alle beobachtbaren Elemente gilt die Formel ϕ.

$$\exists x : s.\ Obs_\Sigma(x) \wedge \phi, \quad kurz: \quad \exists x : \Sigma.\ \phi$$

Es gibt beobachtbare Elemente, für die ϕ gilt.

Auch die **Gültigkeit** (Definition 5.2.2) ist geeignet zu modifizieren. Es wird hier ein neuer Gültigkeitsbegriff eingeführt, der die Prädikate berücksichtigt. Sei O ein Σ-Objekt, $STATE$ ein Zustand und v eine Bewertung.

$$O, STATE \models_P p(t_1, \ldots, t_n) \ \ gdw. \ \ (\ v^*(STATE, t_1)[2], \ldots, v^*(STATE, t_n)[2]\ ) \ \in p^O$$

Die Gültigkeit $\models$ für die nichtprimitiven Konstruktoren $\neg, \wedge, \forall, [\,]$ kann hier völlig analog zu Definition 5.2.2 durchgeführt werden.

6.3 Standardinterpretation

Mit der Namensgebung der Standardprädikate (und auch durch deren Motivation in der Einleitung des Kapitels) verbindet man deren übliche Eigenschaften wie Halbordnungsaxiome oder Äquivalenzrelationsaxiome. Zuerst sollen die erwünschten Eigenschaften, die man von einer Standardsignatur erwartet, formal dargelegt werden (Abschnitte 6.3.2 und 6.3.3).

Dann sollen Standardinterpretationen für eine Standardsignatur angegeben werden, die zeigen, daß geeignete Interpretationen immer existieren (Abschnitt 6.3.4). Dieser Vorschlag kann als Default-Realisierung dienen.

Grundlage der Argumentation in diesem Abschnitt sind beliebige Σ-Objekte, wobei Σ eine Standardsignatur sein soll. Geordnete Σ-Objekte, die etwa Halbordnungen auf den Trägermengen realisieren, werden nicht vorausgesetzt. Stattdessen wird gezeigt, wie diese Halbordnungen spezifiziert werden können.

Zuvor sollen noch einige Notationen eingeführt und Voruntersuchungen durchgeführt werden.

6.3.1 Einführende Überlegungen

Definition 6.3.1 *Sei M eine Menge, $\sim$ eine Äquivalenzrelation. Eine Relation R auf M heißt* **$\sim$-Halbordnung***, falls sie reflexiv und transitiv ist, und falls gilt:*

$$x\ R\ y \wedge y\ R\ x \Leftrightarrow x \sim y \quad \text{für alle } x, y \in M$$

Diese Eigenschaft wird mit **$\sim$-Antisymmetrie** *bezeichnet.*

Lemma 6.3.1 *Sei M eine Menge mit einer Äquivalenzrelation $\sim$ und einer $\sim$-Halbordnung R. Dann definiert $\sqsubseteq_R$ mit*

$$[a]_\sim \sqsubseteq_R [b]_\sim :\Leftrightarrow a\ R\ b$$

eine Halbordnung auf $M/_\sim$, wenn $[a]_\sim$ die zu $a \in M$ gehörende Äquivalenzklasse bezeichnet.

Beweis: Es ist die Wohldefiniertheit von $\sqsubseteq_R$ zu zeigen, d.h.

$$a\ R\ b \wedge a \sim a' \wedge b \sim b' \Rightarrow a'\ R\ b'$$

Dies folgt direkt aus der $\sim$-Antisymmetrie. □

Die übliche Antisymmetrie bei Halbordnungen bezieht sich auf die Identität. Die im Lemma konstruierte Halbordnung $\sqsubseteq_R$ bezieht sich auf die Identität der Äquivalenzklassen.

Definition 6.3.2 *Sei A ein Σ-Objekt, $a \in A_s, s \in S$, und Σ' sei Subsignatur von Σ. Dann sind die Mengen $Obs^A_{\Sigma'_s}$ definiert durch:*

$$a \in Obs^A_{\Sigma'_s} \ \ gdw.\ \ \exists t \in T(\Sigma')_s \ mit\ a = t^A$$

Es ist A^{Obs} das **beobachtbare Σ'-Subobjekt** *von A definiert durch:*

- $A^{Obs}_s := Obs^A_{\Sigma'_s}$ *für* $s \in S \setminus \{state\}$,
- $A^{Obs}_{state} := A_{state}$,
- $op^{Obs}_A := op^A$ *für alle* $op \in OP$,
- $p^{Obs}_A := p^A$ *für alle* $p \in P$,
- $STATE^{Obs} := STATE$.

Es sei auch $Obs^A_{\Sigma'}(a)$ als Schreibweise statt $a \in Obs^A_{\Sigma'}$ zulässig. Mit Obs soll eine S-sortierte Menge von beobachtbaren Teilmengen der Trägermengen für alle Sorten $s \in S$ bezeichnet werden. Mit der obigen Definition wird Beobachtbarkeit in bezug auf die Σ-Objekte betrachtet. Die Anwendung des Beobachtbarkeitsbegriffes, die in der Motivation dieses Kapitels angesprochen wurde, ist die Beobachtbarkeit von Termen einer Subsignatur Σ' in bezug auf Terme der vollständigen Signatur Σ. Von Erreichbarkeit wird gesprochen, wenn (in bezug auf Definition 6.3.2) $\Sigma' = \Sigma$ gilt.

Lemma 6.3.2 *Das beobachtbare Σ'-Subobjekt von A ist ein Σ-Objekt.*

Beweis: Folgt aus der Definition des Σ-Objekts in Kapitel 3.3 sowie der Definition des beobachtbaren Σ-Subobjekts. □

Sei $\equiv$ eine Kongruenzrelation, d.h. eine S-sortierte Menge von Äquivalenzrelationen $(\equiv_s)_{s \in S}$, die jeweils mit den Operationen verträglich sind. Es gilt also, falls $a_i \equiv_{s_i} b_i$ für $a_i, b_i \in A_{s_i}, i = 1,..,n$ und $op : state \times s_1 \times \ldots \times s_n \to state \times s \in OP$, dann für ein Σ-Objekt O:

$$op^O(STATE, a_1, \ldots, a_n)[1] \equiv_{state} op^O(STATE, b_1, \ldots, b_n)[1]$$

und

$$op^O(STATE, a_1, \ldots, a_n)[2] \equiv_s op^O(STATE, b_1, \ldots, b_n)[2]$$

Für $state$ ist $\equiv_{state}$ die Identität. $[.]_s$ bezeichne im folgenden die Äquivalenzklassen bzgl. der Relation $\equiv_s$.

Definition 6.3.3 *Das* **assoziierte Σ-Objekt** *$A|_\equiv$ zu einem Σ-Objekt A sei definiert durch:*

- $(A|_\equiv)_s := A_s|_{\equiv_s}$ *für alle* $s \in S \setminus \{state\}$,
- $(A|_\equiv)_{state} := \{st|_\equiv \mid st \in A_{state}\}$ *wobei* $st|_\equiv(x) := [st(x)]_{state}$ *für alle* $x \in IDENT$ *definiert wird,*
- $op^{A|_\equiv}(STATE|_\equiv, [a_1]_{s_1}, \ldots, [a_n]_{s_n}) := [op^A(STATE, a_1, \ldots, a_n)]_s$ *für alle Operationen* $op \in OP$,
- $p^{A|_\equiv}([a_1]_{s_1}, \ldots, [a_m]_{s_n}) := [p^A(a_1, \ldots, a_m)]_s$ *für alle Prädikate* $p \in P$,
- $STATE|_\equiv(x) := [STATE(x)]_s$ *für alle* $x \in IDENT, x : s$.

Bemerkung 6.3.1 *Da $\equiv$ eine Kongruenzrelation ist, ist $A|_\equiv$ wohldefiniert.*

Hiermit wird eine zu einem Σ-Objekt epimorphe Struktur festgelegt, die das Gleichheitsprädikat $\equiv$ berücksichtigt (s.u.). Es wird Quotientenbildung bzgl. $\equiv$ durchgeführt (siehe [Wir90] S.686). Durch Quotientenbildung werden Äquivalenzklassen für Trägerelemente bzgl. einer Relation gebildet. Aus diesen Klassen wird eine neue Trägermenge zusammengesetzt. $[a]$ bezeichnet die Äquivalenzklasse von a, also alle b, für die $a \equiv b$ gilt.

Lemma 6.3.3 *Das assoziierte Σ-Objekt $A|_\equiv$ ist ein Σ-Objekt.*

Beweis: Folgt aus der Definition des Σ-Objekts in Kapitel 3.3 sowie der Definition des assoziierten Σ-Objekts in Definition 6.3.3. □

Lemma 6.3.4 *Sei A ein Σ-Objekt. Das assoziierte Objekt $A|_{\equiv}$ ist epimorphes Bild von A.*

Beweis: Ein Epimorphismus ist ein surjektiver Σ-Homomorphismus. Sei h ein Σ-Homomorphismus mit $h := (h_s : A_s \rightarrow A_s|_{\equiv_s})_{s \in S \cup \{state\}}$, $h_s : a \rightarrow [a]_s, a \in A_s$ für $s \in S$ und $h_{state} : STATE \rightarrow STATE|_{\equiv}$ (wie oben in Definition 6.3.3 definiert). Nach Konstruktion von $A|_{\equiv}$ ist h ein Σ-Homomorphismus (Definition 3.3.5), d.h. h ist $S \cup \{state\}$-sortiert, erhält die Operationen und ist auch surjektiv, da die Elemente von $A|_{\equiv}$ genau durch die Bilder unter h definiert werden. □

6.3.2 Heuristische Betrachtung der Standardprädikate

Prädikate werden durch boolesche Funktionen auf den Trägermengen interpretiert ($\sqsubseteq, \equiv$ als zweistellige, Obs als einstellige). Sei $p \in P$, dann ist $p^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_{bool}$ [3] für $p : state \times s_1 \times \ldots \times s_n \rightarrow state \times bool$ und ein Σ-Objekt A. Die **Interpretation** der Standardprädikate muß die im folgenden angegebenen Eigenschaften erfüllen. In einem zweischrittigen Vorgehen werden zuerst heuristisch einige vorläufige Eigenschaften beschrieben, die danach in Abschnitt 6.3.3 formal definiert werden.

Sei A ein Σ-Objekt, wobei Σ eine Standard-Signatur ist.

- $Obs_{\Sigma'}$: Sei $s \in S$, Σ' Subsignatur von Σ: Das Prädikat $Obs_{\Sigma'}(t)$ gilt für die Terme $t \in T(\Sigma, X)_s$ einer Sorte s, deren Interpretation auch Interpretation eines Grundtermes der Signatur $\Sigma' \subseteq \Sigma$ ist. Also gilt $Obs_{\Sigma'}(t)$, falls $v^*(STATE, t)[2] \in Obs^A_{\Sigma'}$ für den aktuellen Zustand $STATE$ eines Σ-Objekts A. $Obs_{\Sigma}(t)$ gilt also immer (siehe Definition 6.3.2).

- $\sqsubseteq_s$: Seien t, t' Terme der Sorte s. $\sqsubseteq^{A_s}$ sei die Interpretation von $\sqsubseteq_s$ in A. Die boolesche Funktion $\sqsubseteq^{A_s}$ soll eine Halbordnung auf der Trägermenge A_s definieren. $t \sqsubseteq_s t'$ gilt damit genau dann, wenn $v^*(STATE, t) \sqsubseteq^{A_s} v^*(STATE, t')$ gilt.
  $(v^*(STATE, t_1), v^*(STATE, t_2)) \in \sqsubseteq^{A_s}$ ist als Schreibweise ebenfalls zulässig. $\sqsubseteq_s$ induziert eine Relation $\sqsubseteq^{A_s}$ auf den Trägermengen von A. Die Halbordnungseigenschaften für $\sqsubseteq^{A_s}$, $s \in S$ (reflexiv, antisymmetrisch, transitiv) müssen explizit gefordert werden.

  Die Standardkonstante $\perp_s$ für $s \in S \setminus \{state\}$ soll durch ein geeignetes Element der zugehörigen Trägermenge A_s eines Σ-Objekts A interpretiert werden.

  $$v^*(STATE, \perp_s) := \perp_{A_s} \; mit \; \perp_{A_s} \sqsubseteq^{A_s} a \; für \; alle \; a \in A_s$$

- $\equiv_s$: Seien t, t' Terme derselben Sorte. Für eine Standardinterpretation sollte gelten:

  $$t \equiv_s t' \; gdw. \; v^*(STATE, t) = v^*(STATE, t')$$

  $=$ ist die Identität. Damit ist $\equiv_s$ eine Äquivalenzrelation. Im Fall der Standardinterpretation ist $a = b$ durch $a \sqsubseteq b \wedge b \sqsubseteq a$ definiert. Sonst sollte $\equiv_s$ auf jeden Fall eine Äquivalenzrelation auf A_s induzieren. $\sqsubseteq_s$ ist dann eine $\equiv$-Halbordnung im Sinne von Definition 6.3.1.

[3] *State* als Trägermenge der Sorte *state* tritt hier nicht auf, da die Prädikate zustandsunabhängig definiert sind.

6.3.3 Standardaxiome, -interpretation, -spezifikation

Für Halbordnung, Striktheit sowie Äquivalenz soll nun eine Menge von **Standardaxiomen** E^{Std} für jede Datensorte $s \in S$ angegeben werden, die wir im folgenden beschreiben. Spätestens zur imperativen Implementierung sind dies notwendige Eigenschaften, vorher sind sie wenigstens wünschenswert. Die Standardaxiome garantieren die eben heuristisch beschriebenen Eigenschaften der Standardprädikate.

Definition 6.3.4 *Die* **Standardaxiome** *seien wie folgt festgelegt:*

- Ordnung*:* $\forall x, y, z : s$
  $x \sqsubseteq_s x$ *(reflexiv)*
  $x \sqsubseteq_s y \wedge y \sqsubseteq_s x \Leftrightarrow x \equiv_s y$ *(verträglich mit* $\equiv_s$*)*
  $x \sqsubseteq_s y \wedge y \sqsubseteq_s z \Rightarrow x \sqsubseteq_s z$ *(transitiv)*
  Durch die Verträglichkeit wird die Gleichheit $\equiv_s$ *(nicht die Identität) induziert. Weiterhin muß* $\bot_s$ *als kleinstes Element gefordert werden:*
  $\forall x : s.\ \bot_s \sqsubseteq_s x$ *für alle* $s \in S$

- Striktheit*: Operationen müssen als strikt definiert werden können, d.h. wenn mindestens ein aktueller Parameter* $\bot$ *ist, muß* $\bot$ *als Ergebnis geliefert werden*[4]*.*

  $$f(x_1, \ldots, x_{k-1}, \bot_{s_k}, x_{k+1}, \ldots, x_n) \sqsubseteq \bot_s$$

  $STRICT(OP)$ *sei ein Makro (eine Menge von Axiomen) bestehend aus obiger Formel für alle* $f \in OP, f : s_1 \times \ldots \times s_n \to state \times s,\ s \in S,\ s_k \in (S \cup \{state\})$ *und* $k \in \{1, \ldots, n\}$*.*

- Kongruenzrelation*: Die* $\equiv_s$ *bilden Äquivalenzrelationen auf den Trägermengen* A_s *für alle* $s \in S$*.* $\equiv$ *ist eine Kongruenz*[5]*. Entsprechende Axiome seien angenommen.*

- Beobachtbarkeit*: Das Beobachtbarkeitsprädikat ist nicht benutzerdefinierbar. Es wird durch eine Standardinterpretation realisiert. Daher werden keine Axiome angegeben. Es gilt (wie schon oben beschrieben):* $Obs_{\Sigma'}(t)$ *gilt, falls* $v^*(STATE, t)[2] \in Obs^A_{\Sigma'}$ *für den aktuellen Zustand* $STATE$ *eines* Σ*-Objekts* A*.*

Folgerung 6.3.1 *Falls* $\equiv$ *eine Kongruenz ist, ist* $\sqsubseteq_s$ *für alle* $s \in S$ *eine Halbordnung, die antisymmetrisch bzgl.* $\equiv_s$ *ist.*

Beweis: Trivial. □

Nachdem nun die wünschenswerten Eigenschaften bekannt sind, werden **Standardinterpretationen** für die Elemente einer Standardsignatur vorgestellt, die diese Eigenschaften erfüllen. Die Standardprädikate werden geeignet interpretiert. Falls explizit nichts anderes angegeben ist, wird $\sqsubseteq_s$ als flache Halbordnung und $\equiv_s$ als Identität interpretiert. $Obs_{\Sigma'}$ wird wie angegeben interpretiert. Dies wird später durch den Begriff der Standardspezifikation formalisiert.

Beliebige Trägermengen können immer als bzgl. der Identität geordnet angenommen werden (etwa als triviale Default-Ordnung). Es existieren dann keine zwei Elemente a, b mit $a \neq b$

[4] Es muß mindestens eine nichtstrikte Operation geben, sonst ist $\bot$ immer der kleinste Fixpunkt (in diesem Fall ist die Semantik für die Operation unbrauchbar), siehe [Möl85] Kapitel 7.

[5] Siehe Anmerkungen zur Kongruenzrelation und Strukturverträglichkeit vor Definition 6.3.3.

und $\perp \notin \{a, b\}$ für die gilt $a \sqsubseteq b$ oder $b \sqsubseteq a$. Operationen sind dann monoton. Die triviale Ordnung wird zu einer flachen erweitert, in der $\perp$ das kleinste Element ist.

$$\forall a \in A_s \,.\, \perp_{A_s} \sqsubseteq^{A_s} a$$

Folgerung 6.3.2 *Bei Standardinterpretationen für $\equiv$ und $\sqsubseteq$ (d.h. $\equiv_s$ durch die Gleichheit und $\sqsubseteq_s$ durch die flache Ordnung) gilt für jede Sorte $s \in S$:*

$$a \equiv_s b \quad gdw. \quad a \sqsubseteq_s b \wedge b \sqsubseteq_s a$$

Beweis: Die Standardordnung $\sqsubseteq_s$ induziert die Identität $=_s$, die auch die Definition von $\equiv_s$ ist. □

Definition 6.3.5 *Eine* **Standardspezifikation** $< \Sigma^{Std}, E^{Std} >$ *besteht aus einer Standardsignatur Σ^{Std} und Standardaxiomen E^{Std}.*

6.3.4 Eigenschaften der Standardspezifikation

In diesem Abschnitt wird das Zusammenspiel der Prädikate (benutzerdefiniert oder standardinterpretiert) untersucht.

Folgerung 6.3.3 *Sei A Modell einer Standardspezifikation $< \Sigma^{Std}, E^{Std} >$ und seien t, t_1, t_2 Σ^{Std}-Terme. Dann gilt*[6]*:*

- $A, STATE \models_P t_1 \sqsubseteq t_2$ *gdw.* $v^*(STATE, t_1) \sqsubseteq^A v^*(STATE, t_2)$
- $A, STATE \models_P t_1 \equiv t_2$ *gdw.* $v^*(STATE, t_1) \equiv^A v^*(STATE, t_2)$, *wobei $\equiv^A$ die von $\sqsubseteq^A$ induzierte Kongruenz ist, d.h.*

  $$\begin{aligned} &A, STATE \models_P t_1 \equiv t_1 \quad gdw. \\ &\quad v^*(STATE, t_1) \sqsubseteq^A v^*(STATE, t_2) \ \wedge \ v^*(STATE, t_2) \sqsubseteq^A v^*(STATE, t_1) \end{aligned}$$

- $A, STATE \models_P Obs_{\Sigma'}(t) \quad gdw. \quad \Sigma' \subseteq \Sigma^{Std}$ *und* $v^*(STATE, t)[2] \in Obs^{Q}_{\Sigma'}$

Beweis: Folgt direkt aus der Anwendung der Standardaxiome für die Standardinterpretation. □

Eine neu definierte Kongruenz $\equiv_{neu}$ kann benutzt werden, um eine neue Standardordnung $\sqsubseteq_{neu}$ auf den Äquivalenzklassen einer Trägermenge A_s zu induzieren. Mit $x \equiv_{neu} y$ soll auch $x \sqsubseteq_{neu} y \wedge y \sqsubseteq_{neu} x$ gelten. Die neue Ordnung soll dann flach sein, d.h. $[\perp]_{\equiv_{neu}} \sqsubseteq_{neu} b$ für alle b mit $b := [a]_{\equiv_{neu}}$, $a \in A_s$. Es sei $\perp_{\equiv_{neu}} := [\perp]_{\equiv_{neu}}$. Zwei Äquivalenzklassen bzgl. $\equiv_{neu}$ sind genau dann bzgl. $\sqsubseteq_{neu}$ geordnet, wenn sie

- identisch sind ($[a]_{\equiv_{neu}} \sqsubseteq_{neu} [a]_{\equiv_{neu}}$) oder
- mindestens eine das bottom-Element in der Äquivalenzklassenordnung $\perp_{\equiv_{neu}} := [\perp]_{\equiv_{neu}}$ ist. Dies muß das kleinste Element sein.

[6]Die Definition von $Obs^{A}_{\Sigma'}$ wie in Definition 6.3.2 vorausgesetzt.

Lemma 6.3.5 *Sei $\equiv_s^A$ eine Äquivalenzrelation auf der Trägermenge A_s für alle Datensorten s. Dann ist $(A|_{\equiv_s}, \sqsubseteq)$ eine $\equiv$-Halbordnung über der von $\equiv$ induzierten Ordnung $\sqsubseteq$: $[\perp] \sqsubseteq b$ für alle b mit $b := [a]$ und $a \in A_s$ und $x \sqsubseteq y \wedge y \sqsubseteq x$ folgt aus $x \equiv y$ für Äquivalenzklassen x, y.*

Beweis: $\equiv_s$ wird als Äquivalenzrelation vorausgesetzt. Dann gilt:

1.) Reflexivität: $a \equiv a \quad \Leftrightarrow \quad a \sqsubseteq a \wedge a \sqsubseteq a$.
   $\Rightarrow \sqsubseteq$ ist reflexiv.
2.) Antisymmetrie: $a \equiv b \Leftrightarrow a \sqsubseteq b \wedge b \sqsubseteq a$.
   $\Rightarrow \sqsubseteq$ ist damit antisymmetrisch bzgl. $\equiv$.
3.) Transitivität: $a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$.
   Es gilt $\perp \sqsubseteq x$ und $x \sqsubseteq x$ für alle $x \in A_{s|_\equiv}$ nach Konstruktion der Äquivalenzklassen.
   Es ergeben sich folgende Kombinationsmöglichkeiten:
   $a = b = c = \perp \ \Rightarrow \ \perp \sqsubseteq \perp$
   $a = b = \perp \ \Rightarrow \ \perp \sqsubseteq c$
   $a = \perp \ \Rightarrow \ \perp \sqsubseteq c$
   $\Rightarrow \sqsubseteq$ ist transitiv.

Also ist $(A|_{\equiv_s}, \sqsubseteq)$ Halbordnung. □

Bemerkung 6.3.2 *$=$ bezeichnet im folgenden immer die Identität. $[\perp]_\equiv$ ist immer eine einelementige Äquivalenzklasse. Es gilt $\perp_\equiv := [\perp]_\equiv$ und $\perp \notin [a]$ für $a \neq \perp_{A_s}$, $a \in A_s$ durch das angegebene Konstruktionsverfahren. Also gilt $\perp_\equiv \sqsubseteq_{neu} a$ für alle $a \in A|_{\equiv_{neu}}$ bzw. $\perp_\equiv \sqsubseteq_{neu} [x]$ für alle $x \in A_s$. $\perp$ bleibt von der benutzerspezifizierten Äquivalenzklassenbildung ausgeschlossen.*

Das folgende Lemma setzt die Existenz (und korrekte Definition) folgender Prädikate voraus:

- eine zugrundeliegende Gleichheit (oder Äquivalenz), bezeichnet mit $=$,
- eine Halbordnung, die antisymmetrisch bzgl. $=$ ist, bezeichnet mit $\sqsubseteq$,
- eine Kongruenz, mit deren Hilfe ein assoziiertes Σ-Objekt konstruiert werden kann, bezeichnet mit $\equiv$,
- eine Halbordnung, die antisymmetrisch bzgl. $\equiv$ ist, bezeichnet mit $\sqsubseteq_\equiv$,
- ein Beobachtbarkeitsprädikat *Obs*.

Daß diese Prädikate immer in geeigneter Form existieren können, wurde in diesem Abschnitt, insbesondere in Lemma 6.3.5 gezeigt.

Lemma 6.3.6 *Sei A ein Σ-Objekt, $A|_\equiv$ das zu A assoziierte Σ-Objekt und $\phi \in WFF_P(\Sigma)$. Dann gilt:*

$$A \models_P \phi \ gdw. \ A|_\equiv \models_P \phi$$

Beweis: Der Beweis erfolgt durch Induktion über die syntaktische Struktur der Formeln.

a) atomare Formeln

a.1) Gleichung: Sei ϕ eine Gleichung $x_1 \equiv_s x_2$, wobei x_1, x_2 Variablen der Sorte $s \in S$ sind.

$$\begin{aligned}
&A \models_P x_1 \equiv_s x_2 \;\; gdw. \\
&\quad v^*(STATE, x_1) \equiv_A v^*(STATE, x_2) \;\; gdw. \;\; STATE(x_1) \equiv_{A_s} STATE(x_2) \\
&A|_\equiv \models_P x_1 \equiv_s x_2 \;\; gdw. \\
&\quad v^*(STATE|_\equiv, x_1) \equiv_A v^*(STATE|_\equiv, x_2) \;\; gdw. \;\; [STATE(x_1)]_s = [STATE(x_2)]_s
\end{aligned}$$

Da $STATE|_\equiv(x) = [STATE(x)]_s$ nach Definition gilt, ist damit für den Basisfall der Gleichheitsrelation die Gleichwertigkeit gezeigt. Beliebige Terme über Variablen sind in beiden Termkonstruktionen induktiv definiert, brauchen somit nicht betrachtet werden.

a.2) Ordnung: Sei ϕ eine Ungleichung $x_1 \sqsubseteq_s x_2$. x_1, x_2 sind Variablen der Sorte $s \in S$.

$$\begin{aligned}
&A \models_P x_1 \sqsubseteq_s x_2 \;\; gdw. \\
&\quad v^*(STATE, x_1) \sqsubseteq_A v^*(STATE, x_2) \;\; gdw. \;\; STATE(x_1) \sqsubseteq_{A_s} STATE(x_2) \\
&A|_\equiv \models_P x_1 \sqsubseteq_s x_2 \;\; gdw. \\
&\quad v^*(STATE|_\equiv, x_1) \sqsubseteq_A v^*(STATE|_\equiv, x_2) \;\; gdw. \;\; [STATE(x_1)]_s \sqsubseteq_{\equiv_s} [STATE(x_2)]_s
\end{aligned}$$

wobei $\sqsubseteq_{\equiv_s}$ eine Halbordnung ist, definiert durch $[a]_s \sqsubseteq_{\equiv_s} [b]_s$, $falls\ a \sqsubseteq_{A_s} b$. Also ist $\sqsubseteq_{\equiv_s}$ ist die nach Lemma 6.3.5 konstruierte Ordnung. Damit gilt die Aussage.

a.3) Beobachtbarkeit:

$$\begin{aligned}
&A \models_P Obs_{\Sigma'}(x) \;\; gdw. \\
&\quad v^*(STATE, x)[2] \in Obs^A_{\Sigma'} \\
&A|_\equiv \models_P Obs_{\Sigma'}(x) \;\; gdw. \\
&\quad v^*(STATE|_\equiv, x)[2] \in Obs^{A|_\equiv}_{\Sigma'}
\end{aligned}$$

$a \in Obs^A_{\Sigma'}$ $gdw.$ $[a] \in Obs^{A|_\equiv}_{\Sigma'}$ nach Konstruktion der Äquivalenzklassen und von $Obs_{\Sigma'}$. Dies gilt nach Definition 6.3.3.

Der Wert von Terminterpretationen hängt nur von den Belegungen der freien Variablen ab (siehe Definition der Auswertung in Abschnitt 3.4). Daher wird die Gültigkeit atomarer Formeln über beliebigen Termen nur von den Belegungen der freien Variablen bestimmt. Die Aussagen können also ohne weiteres auf beliebige atomare Formeln übertragen werden.

b) nichtatomare Formeln
Durch die induktive Definition der Formeln gilt die Eigenschaft auch für die nichtatomaren Formeln (siehe Induktionslemma 5.1.1). □

6.4 Definition von Datentypen

Es sollen mit diesem Ansatz, neben den in der Motivation (Abschnitt 6.1) angesprochenen weiteren Problemstellungen, auch benutzerdefinierte Datentypen spezifiziert werden können. Wenn diese Datentypen als realisierte, d.h. ausführbare Spezifikationen von Datentypen (wie die vordefinierten *Integer* oder *Boolean*) verfügbar sein sollen, dann müssen die Modelle der Datentypspezifikation einige Eigenschaften erfüllen. Die Trägermengen müssen cpos und die Funktionen müssen stetig sein. Zur Festlegung der Halbordnung wurde in diesem Kapitel ein Ordnungsprädikat $\sqsubseteq$ bereitgestellt. Neue ausführbare Datentypen können auf den schon bestehenden Datentypen konstruiert werden. Standarddatentypen wie *Integer* oder *Bool* wurden durch Standardmodelle definiert. Daher kann es sinnvoll sein, die Standardmodelle weiterzuverwenden, aber dabei nicht benötigte Elemente dieser Standardmodelle von der Beobachtung auszuschließen. Hierzu kann das Prädikat *Obs* eingesetzt werden.

Als Beispiel wählen wir einen Datentyp *Menge*, der auf Basis eines existierenden Datentyps *Tupel* (*Sequenzen*) spezifiziert wird (vgl. Abschnitte 4.1 und 4.3). Sollen Mengen auf Sequenzen implementiert werden, müssen weitere mengenspezifische Eigenschaften spezifiziert werden. Eine Ordnung auf den permutierten Sequenzen kann festgelegt werden. Es kann wie in Abschnitt 4.3 wieder die Mengeninklusion der Ordnung zugrundegelegt werden. Seien s, t Sequenzen mit $s = s_1 \dots s_n$ und $t = t_1 \dots t_m$. Dann gilt:

$$s \sim t \quad :\Leftrightarrow \quad \begin{cases} 1. & n = m \\ 2. & \exists \pi \in \gamma(\{1,..,n\}).\forall i.\exists j.s_i = t_{\pi(j)} \\ 3. & \exists \rho \in \gamma(\{1,..,n\}).\forall i.\exists j.t_i = s_{\rho(j)} \end{cases}$$

$\gamma(\dots)$ bezeichnet alle Permutationen. Mengen sind dann die Äquivalenzklassen bezüglich $\sim$. Diese Äquivalenzklassenbildung soll die neue Gleichheit auf Mengen definieren.

Als komplexes Beispiel werden Mengen natürlicher Zahlen über Tupeln von ganzen Zahlen spezifiziert. Dazu sind in der Menge der ganzen Zahlen nur die natürlichen als beobachtbar zu kennzeichnen. Es soll hier angenommen werden, daß SET und $TUPLE$ generische Spezifikationen sind, die mit Elementtypspezifikationen, etwa NAT oder INT, instantiiert werden können[7]. Die üblichen Eigenschaften ganzer Zahlen bzw. homogener Tupel seien hier für INT und $TUPLE$ vorausgesetzt.

Beispiel 6.4.1

spec SET(NAT)_by_TUPLE(INT) is
  extend TUPLE(INT) by
  opns
    is_subset: $tuple \times tuple \rightarrow bool$
    $\equiv_{set}$: $tuple \times tuple \rightarrow bool$
    $\sqsubseteq_{set}$: $tuple \times tuple \rightarrow bool$
    set_insert: $tuple \times int \rightarrow tuple$
    set_delete: $tuple \rightarrow tuple$
    empty_set: $\rightarrow tuple$
    set_select: $tuple \rightarrow int$
  axioms $s,t : tuple;\ i : int$
    (1) $Obs_{tuple}(s) \;\Leftrightarrow\; Obs_{\{empty\_set, set\_insert\}_{tuple}}(s)$
    (2) $Obs_{\{zero,succ\}_{int}}(i)$
    (3) $s \equiv_{set} t \;\Leftrightarrow$
      $((\forall i : int.\ \exists j : int.\ i \equiv_{int} set\_select(s) \wedge j \equiv_{int} set\_select(t) \wedge i \equiv_{int} j) \;\wedge$
      $(\ \forall i : int.\ \exists j : int.\ i \equiv_{int} set\_select(t) \wedge j \equiv_{int} set\_select(s) \wedge i \equiv_{int} j))$
    (4) $s \sqsubseteq_{set} t \;\Leftrightarrow\; is\_subset(t,s) = true$
end spec

[7] Da es sich hier um einen funktionalen Datentypen, d.h. eine Spezifikation ohne Zustand handelt, wurde der Sortenbezeichner *state* in den Operationsvereinbarungen nicht benutzt. Alle Operationen sind Attribute.

is_subset testet auf Teilmengenbeziehung. Die oben angegebene Definition auf Sequenzen kann dafür herangezogen werden. *set_insert* fügt ein Element ein, *set_delete* löscht ein Element. Die Operation *empty_set* erzeugt eine leere Menge. *set_select* liefert ein Element der Menge. Auf eine explizite Spezifikation dieser Operationen wurde verzichtet.

Die angegebenen Axiome *(1)* – *(4)* sollen kurz erläutert werden:

1. *set_insert* und *empty_set* sollen *Konstruktoren* sein, d.h. mit ihnen ist die gleiche Menge von Trägerelementen beobachtbar wie mit der gesamten Signatur (vgl. Definitionen 6.3.4 und 6.3.2).

2. Nur die Mengenelemente sind beobachtbar, die mit Grundtermen aus $zero$ und $succ$ erreicht werden können, die also natürliche Zahlen sind; nur für sie müssen die Axiome gelten. Dies ist dann von Bedeutung, wenn auf Standardmodellen eines vordefinierten Datentyps $Integer$ realisiert wird. $Obs_{\{zero,succ\}_{int}}(i)$ gilt, falls $v^*(STATE, i)[2]$ für einen Zustand $STATE$ in der mit $zero$ und $succ$ erreichbaren Teilmenge der Trägermenge A_{int} liegt (vgl. Definition 6.3.4).

3. Zwei Mengen sind gleich, wenn sie aus den gleichen Elementen bestehen. $\equiv_{set}$ ist die neu definierte Gleichheit. Sie orientiert sich an der oben vorgestellten Äquivalenzklassenbildung $\sim$.

4. Zwei Mengen sind geordnet, wenn sie in der Teilmengenbeziehung stehen. $\sqsubseteq_{set}$ ist die neu definierte Ordnung. $\sqsubseteq_{set}$ stützt sich auf $\equiv_{set}$ ab (siehe Abschnitt 6.2). Dies wird noch deutlicher, wenn *is_subset* spezifiziert wird. Dann muß $is\_subset(a, b) = true \land is\_subset(b, a) = true \Rightarrow a \equiv_{set} b$ gelten.

Damit ist veranschaulicht, wie die Aspekte Beobachtbarkeit, Äquivalenz und Ordnung zur axiomatischen Spezifikation von Datentypen eingesetzt werden können, die durch Standardmodelle ausführbar sind (sie sind durch die Fixpunktsemantik der Programmiersprache definiert).

6.5 Zusammenfassung

In diesem Kapitel wurde die Spezifikationslogik aus Kapitel 5 erweitert, indem statt des einen Standardprädikats '*Gleichheit*' nun drei bereitgestellt werden (Ordnung, Äquivalenz und Beobachtbarkeit). Alle drei haben die Aufgabe, die Implementierung von Spezifikationen zu unterstützen.

Modelle ausführbarer Spezifikationen müssen Trägermengen enthalten, die cpos sind. Für *State* als die Menge der Zustände ist das nach Kapitel 3 gegeben. Für Trägermengen der Datensorten muß dies noch explizit gefordert werden können, d.h. eine Ordnung muß spezifizierbar sein. Auf die Spezifikation der Existenz von Suprema oder der Stetigkeit von Funktionen soll allerdings verzichtet werden, da diese Eigenschaften im allgemeinen nicht nachweisbar sind. Verschiedene Autoren [Möl85, GTWW77, WTW78] setzen sich mit der Problematik unendlicher, d.h. nicht darstellbarer Objekte auseinander. Probleme bereiten bei der Spezifikation partieller Funktionen in stetigen Algebren oder Objekten die Behandlung der Suprema. Sie sind in den Trägermengen enthalten (da diese cpos sind), aber sie können nicht immer durch endliche Terme aus $T(\Sigma)$ bezeichnet werden. Die Struktur von Termen entspricht der von Bäumen. B. Möller [Möl85] spricht in diesem Zusammenhang allgemeine Graphen an, die sich durch endliche Terme nicht direkt repräsentieren lassen. Möller benutzt induktiv generierte, stetige Modelle, in die die üblichen Modelle eingebettet werden können, und die die Existenz von Suprema garantieren. Der Ansatz stetiger Algebren wird ebenfalls in [GTWW77, WTW78] behandelt. Die Anforderungen der Existenz der Suprema für Ketten in cpos und der Stetigkeit (d.h. Erhaltung der Suprema) auf cpos sollen mangels geeigneter allgemeiner Nachweismethoden hier nicht spezifizierbar sein. cpo-Eigenschaft und Stetigkeit können nur für den Spezialfall flacher Halbordnungen und strikter Funktionen erfüllt sein.

Um die Mächtigkeit des Ansatzes zu erhöhen, ist es sinnvoll, sich nicht mit der aus einer Halbordnung ableitbaren Gleichheit zufrieden zu geben, sondern zusätzlich die Spezifikation einer Äquivalenz zuzulassen. Es konnte gezeigt werden, daß immer Realisierungen für die Prädikate möglich sind, die die erwarteten Eigenschaften, d.h. die einer Halbordnung oder einer Äquivalenzrelation, erfüllen. Sowohl die Ordnung $\sqsubseteq$ als auch die Äquivalenzrelation $\equiv$ können explizit definiert werden. Es konnte gezeigt werden, daß sowohl aus der Ordnung $\sqsubseteq$ eine Äquivalenz abgeleitet werden kann (nach Definition in Bemerkung 6.2.1) als auch aus $\equiv$ eine Halbordnung $\sqsubseteq$ konstruiert werden kann (nach Lemma 6.3.5).

Modelle können nichterreichbare Elemente enthalten. Daher ist es wichtig, in der Spezifikation von Eigenschaften nur auf erreichbare Elemente Bezug nehmen zu können. Damit kann dann in den erreichbaren Teilstrukturen verifiziert werden.

Die in diesem Kapitel vorgestellten Konzepte können dazu eingesetzt werden, die in Abschnitt 4.3 vorgestellte Erweiterung der vordefinierten Datentypen auf Basis einer Menge primitiver Datentypen auch mit der vorgestellten Spezifikationslogik spezifizieren zu können. Hierzu kann, wie am Beispiel angedeutet, die Beobachtbarkeit eingeschränkt werden. Ebenso sind Ordnungen und angepaßte Gleichheitsprädikate definierbar.

Viele der in der Einleitung vorgestellten Ansätze beinhalten ebenfalls Konzepte, die die Lücke zwischen Spezifikation und Implementierung schließen sollen. Larch stellt als Zwischenebene zwischen einer algebraischen Spezifikationssprache und einer Programmiersprache eine modelltheoretische Interface-Sprache bereit, die im wesentlichen auf Vor-/Nachbedingungsspezifikation beruht. Modelle dieser Schnittstellensprache sind durch die Semantik der Programmiersprache definiert. Die Beziehung zwischen algebraischer und modelltheoretischer Spezifikation wird durch Homomorphismen beschrieben. Ähnlich wird auch von R. Breu und H. Lin verfahren. Der Homomorphismusbegriff wird so angepaßt, daß der Zustandsbegriff der Programmiersprachen darin enthalten ist. Extended ML besteht aus einem algebraischen Spezifikationsansatz, dessen Modellbildung auf der formalen Semantik von Standard ML beruht. Mit dem Übergang von Vor-/Nachbedingungsspezifikation zu imperativen Implementierungen befassen sich Dijkstra, Gries und Morgan [Dijk76, Gri81, Mor88, Mor94]. Im Gegensatz zu Larch oder Extended ML wird die Semantik der axiomatischen Spezifikationen hier nicht direkt durch die Fixpunktsemantik der imperativen Spezifikation definiert. Die in Kapitel 5 benutzte Modellsemantik ist allgemeiner, und damit mächtiger; die Modellbildung kann aber explizit in Richtung Fixpunktsemantik durch die in diesem Kapitel vorgestellten Konzepte eingeschränkt werden.

Kapitel 7

Beweissystem und Verifikation

In den vorangegangenen Kapiteln wurde zum einen ein Berechnungsmodell formal definiert und zum anderen eine Spezifikationslogik entwickelt, mit deren Hilfe Eigenschaften von Programmfragmenten über dem Berechnungsmodell spezifiziert werden können. Da beide Ansätze mit formalen Methoden beschrieben wurden, ergeben sich nun eine Reihe von Möglichkeiten, beide zueinander in Beziehung zu setzen. Dazu sollen im weiteren Formulierungen auf dem Vokabular der Σ-Objekte auch als Programme oder Programmfragmente bezeichnet werden. Die Benutzungsschnittstelle der Σ-Objekte wird als rudimentäre imperative Sprache aufgefaßt.

Synthese bezeichnet die Generierung eines Programmes in einer imperativen Sprache, das eine in einer Spezifikationssprache gegebene Spezifikation erfüllt. Unter *Analyse* versteht man die Konstruktion einer möglichst adäquaten Spezifikation zu einem gegebenen Programm. Unter *Verifikation* ist der Nachweis zu verstehen, daß ein gegebenes Programm eine gegebene Spezifikation erfüllt. Ein weiterer Aspekt ist die Frage der *Äquivalenz* von verschiedenen Programmen bzgl. einer Spezifikation.

Im folgenden soll die Verifikation untersucht werden. Zuerst werden Korrektheitsbegriffe definiert (Abschnitt 7.1). Dann wird in Abschnitt 7.2 ein Beweissystem zur Ermittlung des abstrakten Ein-/Ausgabeverhaltens der imperativen Programme im Umfang des Berechnungsmodells aus Kapitel 3 vorgestellt. In der Literatur wird ein solches Beweissystem *Hoare-Kalkül* genannt [LS84, Fra92, AO94]. Konsistenz und Vollständigkeit dieses Beweissystems werden in Abschnitt 7.3 betrachtet. Abschnitt 7.4 beinhaltet mögliche Erweiterungen zur Verifikation des erweiterten Berechnungsmodells aus den Abschnitten 4.1 und 4.2. Anschließend wird ein Beweissystem für eine vollständige dynamische Logik vorgestellt (Abschnitt 7.5). Am Ende des Kapitels wird die Verifikation eines imperativen Programmes betrachtet.

7.1 Korrektheit

Korrektheit wurde in der Einleitung (Abschnitt 1.3.2) schon als zentrales Qualitätskriterium eingeführt. Ein Korrektheitsbegriff legt fest, wann ein Programm seine Spezifikation erfüllt. Eine Spezifikation mit der modalen Formel

$$\neg(x = 0) \rightarrow [\ p(x)\ ]\ s = div(s, x)$$

soll aussagen, daß die Operation p den Quotienten (div) aus einer Zustandsvariablen s und dem Parameter x bildet, falls x von Null verschieden ist. Dieses Verständnis der Spezifikation setzt implizit die Terminierung von p voraus. Solche Aussagen nennt man Aussagen über *totale Korrektheit*. Wird die Aussage um den Zusatz erweitert, daß p nur dann den Quotienten berechnet, wenn p auch terminiert, spricht man von einer *partiellen Korrektheitsaussage*. Die totale Korrektheit ergibt sich somit als Summe von partieller Korrektheit und Terminierung. Der Nachweis der totalen Korrektheit kann über den getrennten Nachweis von partieller Korrektheit und Terminierung erfolgen. Dies ist formal durch das *Separationslemma* (siehe [Fra92] S. 21) gesichert. Dadurch ist auch der Einsatz unterschiedlicher Beweistechniken für partielle Korrektheit und Terminierung möglich. Im folgenden soll nur die partielle Korrektheit betrachtet werden. Der Begriff wird jetzt formalisiert. Die betrachtete Logik ist vorerst die eingeschränkte Form der dynamischen Logik aus Kapitel 5.2. In Abschnitt 7.5 wird dann auf die erweiterte Logik aus Kapitel 5.3 eingegangen.

7.1.1 Partielle Korrektheit

Definition 7.1.1 *Sei DL die in Kapitel 5.2 vorgestellte dynamische Logik und v die auf dem Zustand konstruierte Bewertung (siehe Abschnitt 3.4). Ein Programmfragment P heißt dann* **partiell korrekt** *bzgl. der Formeln* $\phi, \psi \in WFF_{DL}(\Sigma)$*, falls für ein* Σ*-Objekt O und alle Zustände* $STATE \in State$ *gilt:*

$$\text{falls } O, STATE \models \phi \text{ und } P \text{ terminiert, dann gilt } O, v^*(STATE, P)[1] \models \psi$$

ϕ *heißt Vorbedingung und* ψ *Nachbedingung von P. Es wird auch die Schreibweise* $\phi \to [P]\ \psi$ *für die partielle Korrektheitsaussage benutzt.*

Die Gültigkeitsdefinition aus Kapitel 5.2 setzt also partielle Korrektheit um.

Gegeben sei für dieses Kapitel eine Prädikatenlogik erster Stufe und ein vollständiges und konsistentes Beweissystem für diese Logik (siehe z.B. [Wir90] oder [LS84] Kapitel 2). Die Programmfragmente P in diesem Kapitel können aus den Konstrukten der Σ-Objekte zusammengesetzt werden. Wenn von imperativen Implementierungen oder programmiersprachlichen Realisierungen gesprochen wird, sind diese Programmfragmente gemeint.

In Kapitel 5.1 wurden verschiedene Szenarien zur Nutzung der Logik vorgestellt. Aus den verschiedenen Anwendungsmöglichkeiten soll eine Anwendung herausgegriffen und näher untersucht werden. Es soll eine imperative Implementierung gegen eine axiomatische Spezifikation verifiziert werden. Als Beispiel könnte eine axiomatische Spezifikation lauten:

Beispiel 7.1.1

$$
\begin{array}{lll}
spec\ m & & \\
 & state\_def & s : int \\
 & opns & procedure\ init : state \to state \times void \\
 & & \quad [init()]\ s = 1 \\
 & & procedure\ p : state \to state \times void \\
 & & \quad s > 0 \to [p()]\ s = succ(old(s)) \\
end\ m & &
\end{array}
$$

Bei Expansion des *old*-Operators kann auch für das letzte Axiom

$$s > 0 \wedge s = X \rightarrow [p()]\ s = succ(X)$$

geschrieben werden (X ist eine Spezifikationsvariable, vgl. Abschnitt 5.2.1).

Zu verifizieren ist, ob die Implementierung im folgenden Beispiel partiell korrekt im Sinne eines noch zu definierenden Implementierungsbegriffes ist. Intuitiv ist klar, daß die Implementierung das in der Spezifikation durch Vor- und Nachbedingungen vorgegebene Verhalten realisiert.

Beispiel 7.1.2

```
spec m
        state_def   s : int
        opns        procedure init()
                    begin
                       asgn(s, 1)
                    end init;
                    procedure p()
                    begin
                      if(s > 0, asgn(s, succ(s)))
                    end p;
end m
```

Das Beispiel wird am Ende des Kapitels in Abschnitt 7.6 wieder aufgegriffen und dann verifiziert.

7.1.2 Terminierung

Um Terminierung in einem Verifikationsansatz behandeln zu können, sollte nach [AO94] eine Inferenzregel in das Beweissystem integriert werden, die die Anzahl von Iterationen (von Schleifen oder rekursiven Durchläufen) begrenzt. Eine solche Regel verlangt, daß ein Zähler mit einem positiven Wert geeignet initialisiert wird, der nach jeder Iteration verringert wird, aber nie negativ werden darf. Dadurch kann es nur endlich viele Iterationen geben. Wird der Beweis der Terminierung durchgeführt, so muß der Beweiser selbst die für das vorliegende Programm passende Initialisierung finden.

Nichtterminierung durch undefinierte Situationen, also ein '*Stehenbleiben*' des Programmes, kann hier nicht auftreten, da undefinierte Fälle über den Wert $\perp = \omega$ beschrieben sind. Ausführbare Funktionen sind hier immer strikt, somit kann das Programm auch in undefinierten Situationen immer weiterarbeiten. N. Francez [Fra92] nennt dies *Error Strictness*. Viele Verifikationsansätze definieren einen *Divergenzzustand*, in den ein nicht terminierendes Programm übergeht. Dieser Divergenzzustand kann hier duch das bottom-Element $\perp_{state}$ in *State* modelliert werden. Damit kann die Nichtterminierung eines Programmfragmentes P formal durch $v^*(STATE, P)[1] = \perp_{state}$ beschrieben werden.

Im folgenden soll aber nur die Verifikation der partiellen Korrektheit einer Implementierung gegenüber einer Spezifikation betrachtet werden.

7.1.3 Zum Vorgehen

Bisher sind eine Spezifikationslogik (Kapitel 5, 6) und ein Berechnungsmodell, das semantische Strukturen zur Verfügung stellt (Kapitel 3, 4), verfügbar. In diesem Kapitel wird ein Beweissystem konzipiert. Konsistenz und Vollständigkeit von Beweissystemen sind Eigenschaften, die immer gelten müssen, um dessen Einsetzbarkeit zu garantieren. Die notwendigen Beweise für Konsistenz und Vollständigkeit lassen sich auf zwei Arten erzielen: zum einen '*from scratch*', d.h. ohne Berücksichtigung schon vorhandener, ähnlicher Erkenntnisse wird alles Geforderte gezeigt, und zum anderen durch Abbildung auf Bekanntes. Für eine Reihe von Spezifikationslogiken und zugehörige semantische Strukturen gibt es konsistente und vollständige Beweissysteme, etwa für algebraische Spezifikationslogiken und Algebren [Wir90] oder für modale Logik und Kripke-Modelle [KT90]. Falls sich die gegebene Logik in einen der angesprochenen Ansätze einordnen läßt[1] bzw. durch einen solchen simulierbar ist und die semantischen Strukturen geeignet ähnlich sind (etwa isomorph), dann kann das Beweissystem übertragen werden.

Hier wird ein hybrider Ansatz gewählt. Der Kern des Ansatzes wird '*from scratch*' behandelt. Dies hilft, Eigenschaften des neu entworfenen Berechnungsmodells und der entworfenen Spezifikationslogik zu verstehen. Außerdem ist die Kenntnis anderer Logiken und Semantiken hierzu nicht notwendig. Anders wird bei den Erweiterungen des Prozedurkonzeptes und deren Bearbeitung durch das Beweissystem in Kapitel 7.4 vorgegangen. Die betrachteten Erweiterungen des Berechnungsmodells aus Kapitel 4 sind ebenso in Hoare-Logiken realisiert. Dort werden als Semantiken Modelle angenommen, die in ihrer Struktur den hier benutzten entsprechen. Es soll daher für die Erweiterungen lediglich auf die Literatur verwiesen werden.

7.2 Ein Beweissystem für partielle Korrektheit

Statt des Begriffes *Beweissystem* werden in der Literatur häufig synonym die Begriffe *Inferenzsystem*, *Inferenzkalkül* oder *deduktives System* benutzt [LS84, Fra92, AO91, AO94, Apt81, Kre91, KT90]. Ein Beweissystem für die eingeschränkte Logik aus Kapitel 5.2 soll nun vorgestellt werden.

Zuvor werden wir noch Aussagen über Substitution in Formeln und in den semantischen Strukturen machen. Dies wird dazu dienen, den Effekt von Kommandos auf die Strukturen auch in Formeln explizit machen zu können.

Definition 7.2.1 *ϕ^t_x bezeichnet die gleichzeitige* **Substitution** *aller freien Vorkommen von x durch t in der Formel ϕ.*

Diese Definition der Substitution für Formeln entspricht der Definition der Substitution für Terme in Abschnitt 3.4.

Lemma 7.2.1 *(Substitutionslemma) Sei ϕ eine Formel, x eine Variable einer Datensorte s und t ein Σ-Term der selben Sorte wie x. v sei die auf $STATE$ basierende Bewertung. Dann gilt für jedes Σ-Objekt O und jeden Zustand $STATE \in State$:*

$$O, STATE \models \phi^t_x \quad gdw. \quad O, substitute(STATE, x \mapsto v^*(STATE, t)[2]) \models \phi$$

[1] Wie schon gezeigt werden konnte, sind Σ-Objekte Kripke-Modelle und die vorgestellte Logik ist eine dynamische Logik.

Beweis: Nach Definition 3.4.1 ist $substitute(STATE, x \mapsto y) := STATE'$ mit

$$STATE'(z) = \begin{cases} STATE(z) & falls\ z \neq x, z \neq \bot \\ y & falls\ z = x, z \neq \bot \\ \bot & falls\ z = \bot \end{cases}$$

In $STATE'$ wird der Wert von t (bezeichnet mit y) an x gebunden, sonst bleibt $STATE$ unverändert. Nach Konstruktion der Gültigkeit über der Bewertung v gilt die Aussage. □

Die modale Formel $\phi \to [P]\ \psi$ beschreibt das Ein-/Ausgabeverhalten eines Programmfragmentes P. Um verifizieren zu können, muß daher zunächst ein Beweissystem definiert werden, das das Ein-/Ausgabeverhalten von imperativen Programmfragmenten zu ermitteln gestattet.

Ein *Beweissystem* besteht aus Axiomen und Inferenzregeln (siehe auch [LS84, Fra92, AO91]). Axiome behandeln hier die atomaren Anweisungen *Zuweisung* und *leeres Kommando*; Regeln behandeln im folgenden die zusammengesetzten Anweisungskonstrukte. Die Inferenzregeln sollen für Spezifikations- und Programmiersprache gleichermaßen anwendbar sein. Solche Regeln werden *universell* genannt. Daher muß eine Spezifikationsvariable X (siehe Abschnitt 5.2.1) instantiierbar sein, da sie in der Programmiersprache nicht vorhanden ist. Der Variablen wird dabei ein Wert aus der Trägermenge des zugehörigen Typs zugewiesen.

$$\frac{\phi(X)}{\phi(c)} \qquad [\text{INST}]$$

INST ist eine *Inferenzregel*. $\phi(X)$ ist die *Prämisse* und $\phi(c)$ die *Konklusion*. ϕ ist eine modale Formel mit X als freier Spezifikationsvariablen und einem Grundterm $c : s$, falls $X : s$. Freie Spezifikationsvariablen können zu expliziten (konstanten) Werten instantiiert werden. Die Unterscheidung zwischen Substitution und Instantiierung erfolgt absichtlich. Freie Spezifikationsvariablen X werden mit Werten in einer von den Belegungen der Programmvariablen unabhängigen Bewertung belegt. Sie sollen daher nur durch Konstanten (Grundterme) instantiiert werden können. X kann sowohl in der Vorbedingung als auch in der Nachbedingung auftreten. X soll dabei immer durch den gleichen konstanten Term substituiert werden, damit etwa die Semantik des *old*-Operators definierbar ist. Programmvariablen könnten inzwischen durch den Zustandswechsel ihren Wert verändert haben.

Es werden nun zwei Axiome und vier Inferenzregeln für ein exogenes Beweissystem definiert. Im folgenden sind alle Formeln ϕ, ψ, ξ gemäß der eingeschränkten dynamischen Logik nichtmodal. Das Beweissystem spiegelt dabei die Kompositionalität der Struktur der Programmiersprache wieder. Axiome werden für primitive Anweisungen wie die Zuweisung oder die leere Anweisung definiert. Mit Inferenzregeln können zusammengesetzte Anweisungen betrachtet werden.

Definition 7.2.2 *Ein* **Beweissystem** *für die Logik aus Kapitel 5.2 besteht aus:*

1. *Zuweisungsaxiom*

   $$\phi_x^t \to [asgn(x,t)]\ \phi \qquad [\text{ASGN}]$$

   Jede Eigenschaft ϕ, die vor Ausführung der Zuweisung für t gilt, gilt nachher auch für x.

2. *Leere Anweisung*

$$\phi \rightarrow [skip]\ \phi \qquad [\text{SKIP}]$$

Die leere Anweisung hat keinen Effekt.

3. *Sequenzregel*

$$\frac{\phi \rightarrow [P]\ \psi\ ,\ \ \psi \rightarrow [Q]\ \xi}{\phi \rightarrow [seq(P,Q)]\ \xi} \qquad [\text{SEQUENCE}]$$

Bei geltender Vorbedingung von P gilt, falls die Nachbedingung von P und die Vorbedingung von Q übereinstimmen, die Nachbedingung von Q.

4. *Bedingungs-Regel*

$$\frac{\phi \wedge (b = true) \rightarrow [P]\ \psi\ ,\ \ \phi \wedge \neg(b = true) \rightarrow [skip]\ \psi}{\phi \rightarrow [if(b,P)]\ \psi} \qquad [\text{CONDITIONAL}]$$

Das Kommando P wird nur ausgeführt, falls die Bedingung b wahr ist. Sonst wird nichts (skip) ausgeführt.

5. *Aufruf-Regel*

$$\frac{\phi \rightarrow [com]\ \psi}{\phi \rightarrow [\ p()\ ]\ \psi} \qquad [\text{CALL}]$$

Diese Regel bezieht sich auf parameterlose, nichtrekursive Prozeduren der Form p() = com. Der Rumpf einer Prozedurdefinition kann somit durch den Aufruf der Prozedur p() ersetzt werden. Andere Prozedurformen werden in einem eigenen Abschnitt (7.4) behandelt.

Das System soll noch um eine Regel erweitert werden, die zwar nicht die imperativen Sprachkonstrukte direkt betrifft, die aber hilft, in Beweisen Vor- und Nachbedingungen geeignet abschätzen zu können.

6. *Konsequenz-Regel*

$$\frac{\phi \rightarrow \phi_1,\quad \phi_1 \rightarrow [P]\ \psi_1,\quad \psi_1 \rightarrow \psi}{\phi \rightarrow [P]\ \psi} \qquad [\text{CONS}]$$

Die Regeln ASGN, SEQUENCE und CONDITIONAL basieren auf [LS84] Kapitel 8.1.1. CONS steht für *Consequence* (siehe auch [LS84] Kapitel 8.1.1 oder [Fra92] S. 75). CONS erlaubt es, die Vorbedingung zu verstärken und die Nachbedingung abzuschwächen. Man sagt, daß eine Bedingung ϕ stärker als eine Bedingung ψ ist, wenn sie diese impliziert ($\phi \rightarrow \psi$). Im Gegensatz zu den bisherigen Regeln ist diese unabhängig von der Programmiersprache. Sie hängt nur von der Definition der partiellen Korrektheit ab. Die Anwendbarkeit der CONS-Regel soll kurz illustriert werden. Mit ihr können Implementierungen ausgedrückt werden. Zwischen zwei Prozeduren p und p_1 kann eine Relation $\triangleright$ wie folgt festgelegt werden: Sei p definiert durch $\phi \rightarrow [p(\ldots)]\ \psi$ und p_1 definiert durch $\phi_1 \rightarrow [p_1(\ldots)]\ \psi_1$. Dann gilt $p \triangleright p_1$ genau dann, wenn $\phi \rightarrow \phi_1 \wedge \psi_1 \rightarrow \psi$ gilt. Die Implementierung p_1 ist die mit der schwächeren Vorbedingung und der stärkeren Nachbedingung. Ausgehend von einer 'besseren' Implementierung p_1 kann durch CONS überprüft werden, ob die Spezifikation von p demgegenüber schwächer ist (siehe auch [Mey88] Kap. 11.1, S. 256), ob also p_1 partiell korrekt bzgl. der Spezifikation von p ist. Die Korrektheit von CONS läßt sich leicht anhand der partiellen Korrektheit veranschaulichen. $\phi_1 \rightarrow [p(\ldots)]\ \psi_1$ gilt genau dann, wenn, falls ϕ_1 gilt und p terminiert, dann ψ_1 im Folgezustand gilt. Daraus folgt (wegen $\phi \rightarrow \phi_1$ und $\psi_1 \rightarrow \psi$), daß, falls ϕ gilt und p terminiert, dann ψ im Folgezustand gilt. Dies ist genau die Gültigkeitsdefinition der Konklusion $\phi \rightarrow [p(\ldots)]\ \psi$ von CONS.

Mit Hilfe eines solchen Beweissystems läßt sich zu jeder nichtrekursiven Prozedurimplementierung $p() = com$ und einer initialen oder terminalen Bedingung ein Ein-/Ausgabeverhalten $\phi \rightarrow [com]\ \psi$ ableiten. *com* ist dann partiell korrekt bzgl. ϕ und ψ. Dies wird in Abschnitt 7.6 veranschaulicht.

Bemerkung 7.2.1 *Weitere praktische Regeln sind (neben den im Anhang A.2 aufgeführten prädikatenlogischen Formeln):*

$$\frac{\phi \rightarrow [P]\ \psi_1, \quad \phi \rightarrow [P]\ \psi_2}{\phi \rightarrow [P]\ \psi_1 \wedge \psi_2} \qquad \text{[AND]}$$

$$\frac{\phi_1 \rightarrow [P]\ \psi, \quad \phi_2 \rightarrow [P]\ \psi}{\phi_1 \vee \phi_2 \rightarrow [P]\ \psi} \qquad \text{[OR]}$$

$$\frac{(\phi \wedge \xi) \rightarrow [P]\ \psi, \quad (\phi \wedge \neg\xi) \rightarrow [P]\ \psi}{\phi \rightarrow [P]\ \psi} \qquad \text{[DEMORGAN]}$$

$$\frac{\phi \rightarrow [P]\ \psi}{(\exists x : \phi) \rightarrow [P]\ \psi} \qquad [\exists\text{−EINFÜHRUNG}]$$

x trete weder in P noch in ϕ frei auf.

Diese Regeln sollen im folgenden angenommen werden. Diese Regeln sind [Fra92] Kapitel 4 (AND, OR, DEMORGAN) und [AO94] Anhang B (∃-EINFÜHRUNG) entnommen. Eigenschaften wie Konsistenz und Vollständigkeit sind dort nachzulesen. Konsistenz und Vollständigkeit für die anderen Regeln werden in Abschnitt 7.3 betrachtet.

Die zusätzlichen Regeln werden *Adaptionsregeln* genannt. Sie erlauben eine Korrektheitsformel zu einem Programm an andere Vor- und Nachbedingungen anzupassen, d.h. zu adaptieren, ohne das Programm zu ändern. An den angegebenen Literaturstellen sind weitere Regeln zu finden.

Bemerkung 7.2.2 *Die Erweiterung der Spezifikationslogik aus Kapitel 6 kann durch die Regel*

$$\frac{t_1 =_s t_2, \quad Obs_\Sigma(t_1)}{Obs_\Sigma(t_2)} \qquad \text{[OBS]}$$

für zwei Terme der gleichen Sorte s der Signatur Σ unterstützt werden. Die Prädikate $\equiv$ und $\sqsubseteq$ werden als Äquivalenzrelationen bzw. Halbordnungen axiomatisch spezifiziert. Die entsprechenden Axiome stammen aus einer Prädikatenlogik erster Stufe (vgl. Abschnitt 6.3). Sie brauchen daher nicht betrachtet werden; weitere Betrachtungen der Konzepte aus Kapitel 6 sind nicht notwendig.

Definition 7.2.3 *Sei $E \subseteq WFF(\Sigma), \phi \in WFF(\Sigma)$. Ein* **Beweis** *von ϕ in E ist eine Folge $\phi_1, \ldots, \phi_n$ mit $\phi_n = \phi$, so daß für jedes ϕ_i eine der Aussagen gilt:*

- *ϕ_i ist ein Axiom,*
- *$\phi_i \in E$,*
- *ϕ_i folgt aus einem früheren ϕ_j ($j < i$) durch Anwendung einer Inferenzregel.*

$E \vdash \phi \;:\Leftrightarrow\; \phi$ hat einen Beweis in E. Man spricht: 'E impliziert ϕ' oder 'ϕ ist aus E ableitbar'.

7.3 Konsistenz und Vollständigkeit

Es muß nun sichergestellt werden, daß das definierte Beweissystem (Definition 7.2.2) auch ordnungsgemäß arbeitet, d.h. es ist zu fragen,

1. ob jede abgeleitete Formel auch gilt (*Konsistenz*),
2. ob jede gültige Formel auch ableitbar ist (*Vollständigkeit*).

Hierzu werden Ergebnisse der Hoare-Logik (siehe [LS84] Kap. 8 oder [Fra92, AO94]) herangezogen. Es wird gezeigt, daß die eingeschränkte modale Logik (Abschnitt 5.2) zur Hoare-Logik äquivalent ist. Die Menge der nichtmodalen wohlgeformten Formeln und ihre Gültigkeitsrelation auf Σ-Objekten entspricht einer Prädikatenlogik erster Stufe. Für diese gibt es vollständige und konsistente Beweissysteme (etwa nach [Wir90] oder [LS84]).

Das Problem der Erreichbarkeit wurde in Kapitel 6 schon angesprochen. Voraussetzung für ein vollständiges Beweissystem ist die Darstellbarkeit aller semantischen Elemente im syntaktischen Formalismus. Da für Σ-Objekte als Modelle Erreichbarkeit nicht gefordert ist, ist für beliebige Modelle ein vollständiges Beweissystem nicht zu erhalten. In Kapitel 6 wurde aber ein Beobachtbarkeitsprädikat definiert, mit dem Axiome auf erreichbare Substrukturen bezogen werden können. Die Axiome brauchen daher auch nur in diesen erreichbaren Substrukturen verifiziert werden.

Im folgenden soll zur Vereinfachung angenommen werden, daß alle Σ-Objekte erreichbar oder geeignet mit dem Beobachtbarkeitsprädikat spezifiziert sind. Außerdem soll ein vollständiges und konsistentes Beweissystem für eine Prädikatenlogik erster Stufe angenommen werden.

Definition 7.3.1 *Seien L_1und L_2 zwei Erweiterungen einer Prädikatenlogik L über einer Signatur Σ um weitere logische Konnektoren. Sei eine Menge $WFF_L(\Sigma)$ für die Prädikatenlogik L gegeben. $WFF_{L_1}(\Sigma)$ und $WFF_{L_2}(\Sigma)$ werden analog zu WFF in Abschnitt 5.2.1 gebildet. L_1 und L_2 sind* **äquivalent** *gdw. für alle Σ-Objekte O aller Signaturen Σ gilt:*

$$\forall \phi_1 \in WFF_{L_1}(\Sigma).\ \exists \phi_2 \in WFF_{L_2}(\Sigma).$$
$$\{O \in Obj(\Sigma) \mid O \models \phi_1\} = \{O \in Obj(\Sigma) \mid O \models \phi_2\}$$

und

$$\forall \phi_2 \in WFF_{L_2}(\Sigma).\ \exists \phi_1 \in WFF_{L_1}(\Sigma).$$
$$\{O \in Obj(\Sigma) \mid O \models \phi_2\} = \{O \in Obj(\Sigma) \mid O \models \phi_1\}$$

Zwei Erweiterungen einer Prädikatenlogik sind also gleich, wenn sich eine Formel semantisch analog in der anderen Form darstellen läßt.

[LS84] definieren die Semantik einer Hoare-Formel $\mathcal{I}(\{p\}S\{q\}) : \Gamma \to bool$ als Erweiterung einer Prädikatenlogik wie folgt (dort Definition 8.2, S. 150)[2]:

$$\begin{array}{l} \mathcal{I}(\{p\}S\{q\})(\gamma) = true \text{ gdw.} \\ \qquad \mathcal{I}(p)(\gamma) = true \text{ und} \\ \qquad falls\ \mathcal{M}_{\mathcal{I}}(S)(\gamma)\ definiert,\ dann\ \mathcal{I}(q)(\mathcal{M}_{\mathcal{I}}(S)(\gamma)) = true \end{array}$$

$\mathcal{I}$ ist die Interpretation über einer Basis einer Prädikatenlogik[3], die eine Formel in Abhängigkeit vom aktuellen Zustand interpretiert. Γ ist eine Menge von Zuständen mit $\gamma : V \to D$ und $\gamma \in \Gamma$ (die Zustände γ sind Belegungen von Variablen aus V mit Werten aus D). $\mathcal{M}_{\mathcal{I}}(S) : \Gamma \to \Gamma$ ist die Semantikfunktion für Programme auf den Zuständen, die das Ein/Ausgabeverhalten auf den Zuständen beschreibt. Die Semantik des Programmes S ist eine Zustandstransition. Eine Hoare-Formel $\{p\}S\{q\}$ gilt also genau dann, wenn die Vorbedingung p gilt und, falls die Semantik des Programmes S definiert ist (S also terminiert), im Folgezustand die Nachbedingung q gilt. Die partielle Korrektheit wird so ausgedrückt.

Hier ist für ein Σ-Objekt O definiert (nach Definition 5.2.2):

$$\begin{array}{l} O, STATE \models p \to [S]\ q \text{ gdw.} \\ \qquad O, STATE \models p \text{ und} \\ \qquad falls\ v^*(STATE, S)[1]\ definiert,\ dann\ O, v^*(STATE, S)[1] \models q \end{array}$$

Lemma 7.3.1 *Eine um $\{p\}S\{q\}$ nach [LS84] erweiterte Prädikatenlogik ist äquivalent im Sinne von Definition 7.3.1 zur eingeschränkten modalen Logik (mit $p \to [S]\ q$) nach Kapitel 5.2.*

Beweis: Es sind obige Definition nach [LS84] und Definition 5.2.2 zu vergleichen. Die Gleichwertigkeit des prädikatenlogischen Teils wird angenommen.

$$\mathcal{I}(\phi)(\gamma)\ gdw.\ O, STATE \models \phi$$

gilt für nichtmodale prädikatenlogische Formeln ϕ. Dem Γ aus [LS84] entspricht hier *State*. Damit kann $\gamma = STATE$ gesetzt werden.

Für die Programmfragmente S soll die gleiche Semantik (definiert durch $\mathcal{M}$ oder v^* über einer semantischen Struktur O) angenommen werden. Durch Ausführung von Programmfragmenten werden Belegungen der Variablen in den Zuständen substituiert. $\mathcal{M}_{\mathcal{I}}(S)(\gamma)$ entspricht dem Aufruf $v^*(STATE, S)[1]$ mit $STATE \in State$. Falls das Programmfragment S terminiert ($v^*(STATE, S)[1] \neq \perp_{state}$), ist $\mathcal{M}_{\mathcal{I}}(S)(\gamma)$ definiert.

Die Gültigkeit einer Hoare-Formel $\{p\}S\{q\}$ bzw. $p \to [S]q$ wird also analog definiert. Nach den Vorüberlegungen vor diesem Lemma sind also $\mathcal{I}$ und $\models$ gleichwertig für die neu hinzugenommenen logischen Konnektoren (Hoare-Formel bzw. modaler Box-Operator). Für alle anderen logischen Konnektoren war die Gleichwertigkeit Voraussetzung.

[2] Die Semantik einer Hoare-Formel wird analog in [Fra92] S. 19 und [AO94] Kapitel 3.5 definiert.

[3] Eine Basis ist in [LS84] ein Paar (F, P) aus Operations- und Prädikatensymbolen. Eine Interpretation weist dort jedem Operations- oder Prädikatensymbol eine totale Funktion bzw. ein boolesches Prädikat über einem Domain D (einer Menge von Werten) zu. Die Interpretation ordnet außerdem jeder Formel einen Wahrheitswert zu. Hier wird die Abbildung von Operationssymbolen auf Funktionen durch die Auswertung v^* festgelegt. Die Gültigkeit von Formeln wird hier durch eine Gültigkeitsrelation beschrieben.

Mit der Gleichwertigkeit von $\mathcal{I}$ und $\models$ sowie von $\mathcal{M}$ und v^* ergibt sich dies aus den oben aufgeführten Bedingungen. In beiden Fällen wird die partielle Korrektheit der Formel ausgedrückt. Die Semantikdefinitionen der beiden Formeln sind somit gleichwertig, daher folgt die Äquivalenz sofort. □

Hiermit sichern wir die Verwendbarkeit der Ergebnisse der Hoare-Logik nach [LS84].

Bemerkung 7.3.1 *Zur Konsistenz und Vollständigkeit der Erweiterungsregeln CONS, AND, OR und DEMORGAN siehe [Fra92] Kap. 4.3 und 4.4 und zur ∃-EINFÜHRUNG siehe [AO94] Kap. 3.6. Sie sind für die Arbeit nicht zwingend notwendig (lediglich praktisch), daher sollen sie nicht weiter formal betrachtet werden.*

Konsistenz und Vollständigkeit der OBS-Regel (Bemerkung 7.2.2) ergibt sich aus der Definition der Standardinterpretation (Beobachtbarkeit mit Subsignatur) für Obs in Abschnitt 6.3. Durch die Standardinterpretation folgen die beiden Eigenschaften sofort.

7.3.1 Zur Konsistenz

Das Beweissystem aus Kapitel 7.2 (Definition 7.2.2) wird hier betrachtet. Zuerst soll die Konsistenz gezeigt werden, d.h. daß jede abgeleitete Formel auch gilt.

Satz 7.3.1 *(Konsistenz) Das Beweissystem nach Definition 7.2.2 ist konsistent.*

Beweis: Der Beweis orientiert sich an [LS84] Kapitel 8.2 . Es muß gezeigt werden, daß jedes Axiom tatsächlich gilt und daß, falls die Prämisse einer Inferenzregel gilt, dies auch für die Konklusion gilt.

a.) *Zuweisungs-Axiom* $asgn(x,t)$

Sei O ein Σ-Objekt und v eine Bewertung. Es ist zu zeigen, daß das Axiom $\phi_x^t \rightarrow [asgn(x,t)]\ \phi$ gilt, d.h. $O, STATE \models \phi_x^t \rightarrow [asgn(x,t)]\ \phi$ ist für alle Zustände $STATE$ erfüllt. $STATE \in State$ sei ein beliebiger Zustand, so daß $O, STATE \models \phi_x^t$ erfüllt ist, also die Vorbedingung gilt. $v^*(STATE, asgn(x,t))$ ist total definiert (siehe Kapitel 3.4).

Sei $STATE' = v^*(STATE, asgn(x,t))[1]$. Es ist die Gültigkeit $O, STATE' \models \phi$ zu zeigen. Da Formeln in Zuständen von Objekten interpretiert werden, ist folgende Äquivalenz gültig (nach Substitutionslemma 7.2.1):

$$O, STATE \models \phi_x^t \quad \Leftrightarrow \quad O, substitute(STATE, x \mapsto v^*(STATE,t)[2]) \models \phi$$

Da die Semantik von $asgn(x,t)$ als

$$[substitute(STATE, x \mapsto v^*(STATE,t)[2]), null]$$

definiert ist, folgt

$$O, STATE' \models \phi$$

direkt.

b.) *Leeres Kommando skip*
Es gilt:

$$O \models \phi \rightarrow [skip]\ \phi$$

skip hat keinen Effekt, daher bleibt die Gültigkeit von ϕ in jedem Fall unverändert.

c.) *Sequenz-Regel* $seq(P_1, P_2)$
Es gelte:

$$O \models \phi \rightarrow [P_1]\ \psi, \quad O \models \psi \rightarrow [P_2]\ \xi$$

Es ist die Konklusion

$$O \models \phi \rightarrow [seq(P_1, P_2)]\ \xi$$

zu zeigen. Es soll die Vorbedingung von P_1 in einem beliebigen Zustand $STATE$ eines Σ-Objekts O gelten, also $O, STATE \models \phi$.

1. $v^*(STATE, seq(P_1, P_2))$ ist undefiniert. Dann muß nichts gezeigt werden, da nur bei Terminierung die Nachbedingung gelten muß.
2. Die Sequenz terminiert. Sei $STATE' = v^*(STATE, seq(P_1, P_2))[1]$, dann ist $O, STATE' \models \xi$ zu zeigen. Es ist

$$STATE' = v^*(v^*(STATE, P_1)[1], P_2)[1]$$

nach Definition der Sequenz. Setze $STATE'' = v^*(STATE, P_1)[1]$, damit ist dann $STATE' = v^*(STATE'', P_2)[1]$. Mit $O \models \phi \rightarrow [P_1]\ \psi$ folgt $O, STATE'' \models \psi$. Zusammen mit $O \models \psi \rightarrow [P_2]\ \xi$ gilt dann $O, STATE' \models \xi$.

d.) *Bedingungs-Regel* $if(b, P)$
Es gelte für ein Σ-Objekt O:

$$O \models \phi \wedge (b = true) \rightarrow [P]\ \psi, \quad O \models \phi \wedge \neg(b = true) \rightarrow [skip]\ \psi$$

Es ist die Gültigkeit der Konklusion

$$O \models \phi \rightarrow [if(b, P)]\ \psi$$

zu zeigen. Es gelte $O, STATE \models \phi$ für einen beliebigen Zustand $STATE$.

1. Falls $O, STATE \models b = true$ gilt:
Sei $STATE' = v^*(STATE, P)[1]$. Zu zeigen ist $O, STATE' \models \psi$, falls P terminiert, sonst muß nichts gezeigt werden. Das gilt nach Hypothese.
2. Falls $O, STATE \models \neg(b = true)$ gilt:
Sei $STATE' = v^*(STATE, skip)[1]$. *skip* terminiert auf jeden Fall. Zu zeigen ist also $O, STATE' \models \psi$. Das gilt nach Hypothese.

e.) *Aufruf-Regel* $p()$ nichtrekursiv
Es gelte $O \models \phi \rightarrow [com]\ \psi$. Es ist die Konklusion $O \models \phi \rightarrow [p()]\ \psi$ zu zeigen. Es soll in einem beliebigen Zustand $STATE$ eines Σ-Objekts O die Eigenschaft $O, STATE \models \phi$ gelten. Nach Definition ist:

$$v^*(STATE, p()) := v^*(STATE, p)(STATE) = v^*(STATE, com)(STATE)$$

D.h. auch diese Konklusion gilt. □

7.3.2 Zur Vollständigkeit

Vollständigkeit bedeutet, daß Beweise auch existieren, falls die Gültigkeit einer Formel gegeben ist. Daraus folgt, daß auch alle Aussagen in der Spezifikationssprache ausdrückbar sein müssen.

Ein Problem bei der Betrachtung der Vollständigkeit von Hoare-Logiken ist die Logik der Zusicherungssprache (*assertion language*). Diese wird eingesetzt um Zustände zu beschreiben (Hoare-Formeln hingegen beschreiben Zustandsübergänge). Die Zusicherungssprache ist in der Regel eine Gleichungslogik erster Stufe. Als Frage in bezug auf die Vollständigkeit stellt sich noch, ob die Vollständigkeit eines Hoare-Systems unabhängig von der Zusicherungssprache untersucht werden kann. Cook hat gezeigt, daß, wenn ein vollständiges Beweissystem für die Zusicherungssprache existiert und die Zusicherungssprache eine Expressivitätsbedingung erfüllt, dann jede Hoare-Formel (partielle Korrektheitsaussage) beweisbar ist (nach [Coo74], zitiert nach [Cla79]). Die Vollständigkeit eines Beweissystems einer Gleichungslogik erster Stufe wird in diesem Kapitel vorausgesetzt.

Gödels Unvollständigkeitstheoreme machen Aussagen über die Nichtexistenz von vollständigen Beweissystemen. Wenn die Theorie einer Menge von Formeln in einem Modell gilt und diese Theorie nicht rekursiv aufzählbar ist, dann existiert kein vollständiges Beweissystem[4]. Die Menge der ableitbaren Formeln sollte also (wenn möglich) rekursiv aufzählbar sein[5]. Andernfalls sollte ein geeigneter Vollständigkeitsbegriff gesucht werden. Die Menge der gültigen Axiome in Standardmodellen der *Peano-Arithmetik*[6] ist nicht rekursiv aufzählbar. Ein Beweissystem für eine Logik über diesen Modellen kann nach Gödel nicht vollständig sein (siehe [LS84] Kapitel 8.2, 8.3, [Fra92] Kapitel 4.4, [AO94] Kapitel 3.5, [Apt81] Kapitel 2.7 - 2.9, [Wir90] Kapitel 3.1).

Beispiel 7.3.1 *Die modale Formel* $true \rightarrow [asgn(x,e)]\ x = e$ *bleibt solange unbeweisbar, bis ein Beweissystem existiert, das* $true \Rightarrow e = e$ *beweisen kann. Mit* $e = e$ *als Vorbedingung kann in diesem Fall das Zuweisungsaxiom ASGN angewendet werden, welches* $e = e \rightarrow [asgn(x,e)]\ x = e$ *als korrekt anerkennt. In anderen Situationen müssen auch Vor- und Nachbedingungen durch die CONS-Regel abgeschätzt werden, bevor andere Regeln oder Axiome, wie etwa ASGN eingesetzt werden können.*

Die Gödelschen Unvollständigkeitstheoreme implizieren nach obigen Überlegungen, daß für semantische Strukturen, wie etwa ein Standardmodell für die Peano-Arithmetik, die Menge der wahren Aussagen nicht mehr rekursiv aufzählbar ist. Da Modelle im vorliegenden Ansatz in der Regel Standardmodelle eines Datentyps *Integer* umfassen, greift die Gödelschen Unvollständigkeitstheore auch hier. Es soll hier davon ausgegangen werden, daß für Modelle von Spezifikationen, wie auch für Standardmodelle der Peano-Arithmetik, die wahren Aussagen $true \rightarrow [P]\ \phi$ nicht rekursiv aufzählbar sind. Die Beweisbarkeit einer modalen

[4] Siehe [Bar93] zur Definition der rekursiven Aufzählbarkeit *Def. 6.1* in Abschnitt C.1.6 *Recursive Enumerability* und Abschnitt D.1.2 S. 825 zu den Gödelschen Unvollständigkeitstheoremen. In [Bar93] S. 546 (Abschnitt C.1.7 *Logic and Recursion Theory*) und S. 560 (Abschnitt C.1.10 *Definability and Recursion*) wird festgestellt, daß die Menge der wahren Aussagen für Peano-Arithmetik nicht rekursiv aufzählbar ist.

[5] Zu dieser Problematik siehe auch [Wir90] S.688f, S.694f.

[6] Unter *Peano-Arithmetik* versteht man die Menge aller Formeln, die für die Funktionen $\{0, 1, +, *\}$ und das Prädikat $<$ in der Menge der natürlichen Zahlen Nat_0 mit den üblichen Interpretationen gelten (siehe [LS84] S. 32).

Formel ist nur gegeben, wenn die Einbeziehung der unterliegenden Interpretationen und semantischen Strukturen, in denen gerechnet wird, möglich ist. Ein Beweissystem kann hierfür dann also nicht vollständig sein. Wenn ein Beweissystem nicht vollständig ist, ist die logische Sprache hinsichtlich ihrer Ausdrucksfähigkeit eingeschränkt. Diese Eingeschränktheit wird in den Begriffen der *Expressivität* der Gültigkeitsrelation einer Spezifikationssprache in bezug auf eine Programmiersprache und der *relativen Vollständigkeit* für ein Beweissystem formal erfaßt. Eine Gültigkeitsrelation einer Sprache heißt *expressiv*, wenn für eine Hoare-Formel immer eine Nachbedingung etabliert werden kann, die das Problem des obigen Beispiels löst; oder allgemeiner: sie ist expressiv, wenn Mengen von Zuständen durch Zusicherungen (nichtmodale Formeln) beschreibbar sind. In der Literatur tauchen verschiedene Definitionen der Expressivität auf. Cook's originale Definition [Coo78] basiert auf stärksten Nachbedingungen, Clarke [Cla79] verwendet schwächste Vorbedingungen, de Bakker [dB80] eine Mischung aus beiden. Olderog [Old83] zeigt die Äquivalenz der drei Ansätze. Detaillierte Anmerkungen zu dieser Problematik sind in [Apt81] Kapitel 2.7 bis 2.9 zu finden. Die Vollständigkeit von Beweissystemen im Kontext denotationaler Semantik wird ausführlich in [dB80] behandelt.

Es soll nun der Begriff der *Expressivität* formal eingeführt werden (siehe [LS84] Kap. 6.1). Sei O im folgenden ein Σ-Objekt und $STATE \in State$.

Definition 7.3.2 *Eine Formel ψ ist* **schwächste Vorbedingung** $wp(P,\phi)$ *(weakest precondition) von Programmfragment P und Formel ϕ, wenn für die nichtmodalen Formeln erster Stufe ϕ, ψ gilt:*

$$
\begin{aligned}
&O, STATE \models \psi \ gdw. \\
&\qquad (\ v^*(STATE, P) \ undefiniert \\
&\quad oder \\
&\qquad v^*(STATE, P) \ definiert \ und \ O, v^*(STATE, P)[1] \models \phi \ )
\end{aligned}
$$

Bemerkung 7.3.2 *Diese Definition findet sich in der Literatur (z.B. [LS84, AO94]) unter dem Begriff 'weakest liberal precondition'. Dort wird noch eine 'weakest precondition' definiert, die zur Betrachtung totaler Korrektheit eingesetzt wird.*

Definition 7.3.3 *Eine Formel ϕ ist* **stärkste Nachbedingung** $sp(P,\psi)$ *(strongest postcondition) von Programmfragment P und Formel ψ, wenn für die Formeln erster Stufe ϕ, ψ gilt:*

$$
\begin{aligned}
&O, STATE \models \phi \ gdw. \\
&\qquad es \ existiert \ ein \ STATE' \ mit \ O, STATE' \models \psi \ und \ v^*(STATE', P)[1] = STATE
\end{aligned}
$$

Definition 7.3.4 *Eine Gültigkeitsrelation $\models$ für eine modale Formel $\psi \rightarrow [P]\ \phi$ über einer Prädikatenlogik erster Stufe L heißt* **expressiv**, *falls für jedes Programmfragment P und jede Formel $\psi \in WFF_L(\Sigma)$ eine Formel $\phi \in WFF_L(\Sigma)$ existiert, so daß ϕ die stärkste Nachbedingung $sp(P,\psi)$ von P und ψ ist.*

Hier wird die stärkste Nachbedingung statt der schwächsten Vorbedingung eingesetzt. Eine Definition über schwächste Vorbedingungen wäre ebenfalls möglich. Zur Gleichwertigkeit beider siehe [LS84] Satz 8.16 oder [Old83] Proposition 3.1.

Satz 7.3.2 *(Cook's Satz zur relativen Vollständigkeit) Sei $\models$ eine expressive Gültigkeitsrelation einer Prädikatenlogik L, sei O ein beliebiges Σ-Objekt. Dann gilt für jede modale Formel $h \equiv \phi \rightarrow [P]\ \psi$:*

$$falls\ O \models h,\ dann\ O \vdash h$$

Beweis: Siehe [LS84] Satz 8.11. □

Definition 7.3.5 *Ein Beweissystem wird als* **relativ vollständig** *bezeichnet, wenn es auf einer expressiven Gültigkeitsrelation basiert, d.h. wenn es alle wahren Prädikate in der unterliegenden semantischen Struktur als Axiome benutzen kann. Eine expressive Gültigkeitsrelation (siehe Definition 7.3.4 und Satz 7.3.2) macht dies möglich.*

Formeln einer Prädikatenlogik erster Stufe über Standardmodellen einer Peano-Arithmetik sind expressiv. Die Ähnlichkeit dieser Standardmodelle zu den hier benutzten Modellen wird im folgenden in Beweisen ausgenutzt.

Die Expressivität der Gültigkeitsrelation $\models$ soll nun nachgewiesen werden. Falls das gegeben ist, folgt die (relative) Vollständigkeit sofort (siehe Bemerkung 7.3.5).

Satz 7.3.3 *(Expressivität) Die Gültigkeit $\models$ aus Kapitel 5.2.2 ist expressiv.*

Beweis: Dieser Beweis folgt dem Beweis von Satz 8.16 in [LS84] zur Expressivität der Peano-Arithmetik.

Sei $\psi \in WFF(\Sigma)$, P ein Programmfragment, O ein Σ-Objekt. Zu zeigen ist, daß ein $\phi \in WFF(\Sigma)$ existiert, so daß ϕ stärkste Nachbedingung von P und ψ ist. Konkret muß für alle Zustände $STATE \in State$ gezeigt werden:

$$\begin{array}{l} O, STATE \models \phi\ gdw. \\ \qquad es\ existiert\ STATE'\ mit\ O, STATE' \models \psi\ und\ STATE = v^*(STATE', P)[1] \end{array}$$

$STATE'$ ist der Vorzustand und $STATE$ der Nachzustand, ψ ist damit also die Vorbedingung und ϕ die Nachbedingung des Programmfragmentes P.

Der Beweis erfolgt durch strukturelle Induktion über die Programmkonstrukte.

a.) *Zuweisungs-Axiom* $asgn(x, t)$

Wähle für die Nachbedingung $\phi : \exists c.\ (\psi_x^c \wedge x = t_x^c)$. c sei eine Variable der gleichen Sorte wie x, c trete weder in t noch in p auf.

Es soll hier gezeigt werden, daß die Gültigkeit expressiv ist, also, daß $\phi \equiv \exists c.\ (\psi_x^c \wedge x = t_x^c)$ eine geeignete stärkste Nachbedingung ist. Damit ϕ Nachbedingung ist ($O, STATE \models \phi$), muß für alle Nachzustände $STATE \in State$ einer Zuweisung gelten (siehe auch Einleitung des Beweises):

$$\begin{array}{l} O, STATE \models \phi\ gdw. \\ \qquad es\ existiert\ ein\ Vorzustand\ STATE'\ mit\ O, STATE' \models \psi \\ \qquad und\ STATE = v^*(STATE', asgn(x,t))[1] \end{array}$$

Dies soll nun für obige Wahl von ϕ gezeigt werden.

$O, STATE \models \exists c.\ \psi_x^c \wedge x = t_x^c$
gdw. es existiert $d \in A_s$ für $x : s$ mit
$O, substitute(STATE, c \mapsto d) \models \psi_x^c$ und
$STATE(x) = v^*(substitute(STATE, c \mapsto d), t_x^c)[2]$
gdw. es existiert $d \in A_s$ für $x : s$ mit
$O, substitute(STATE, x \mapsto d) \models \psi$ und
$STATE(x) = v^*(substitute(STATE, x \mapsto d), t)[2]$
gdw. es existiert $STATE'$ mit $STATE'(y) = STATE(y)$ für $y \neq x$ und
$STATE'(y) = v^*(substitute(STATE, x \mapsto d), t)[2]$ für $y = x$,
$O, STATE' \models \psi$ und $STATE(x) = v^*(STATE', t)[2]$
gdw. es existiert $STATE'$ mit
$O, STATE' \models \psi$ und $STATE = v^*(STATE', asgn(x, t))[1]$

b.) *Leeres Kommando* $skip$
Es kann die Vorbedingung als Nachbedingung gewählt werden. Da $skip$ den Zustand nicht verändert, gilt sie dann automatisch.

c.) *Sequenz-Regel* $seq(P_1, P_2)$
Induktionshypothese:

Es existiert ein ϕ, wobei ϕ stärkste Nachbedingung von ψ und P_1 ist.
Es existiert ein ξ, wobei ξ stärkste Nachbedingung von ϕ und P_2 ist.

Wähle ξ als stärkste Nachbedingung von ψ und $seq(P_1, P_2)$. Für alle $STATE \in State$ gilt:

$O, STATE \models \xi$
gdw. es existiert $STATE''$ *mit* $O, STATE'' \models \phi$ und
$STATE = v^*(STATE'', P_2)[1]$
gdw. es existieren $STATE', STATE''$ *mit* $O, STATE' \models \psi$ und
$STATE'' = v^*(STATE', P_1)[1], \quad STATE = v^*(STATE'', P_2)[1]$
gdw. es existiert $STATE'$ *mit* $O, STATE' \models \psi$ und
$STATE = v^*(STATE', seq(P_1, P_2))[1]$

d.) *Bedingungs-Regel* $if(b, P)$
Induktionshypothese:

Es existiert ein ϕ_1, wobei ϕ_1 stärkste Nachbedingung von $\psi \wedge (b = true)$ und P ist.
Es existiert ein ϕ_2, wobei ϕ_2 stärkste Nachbedingung von $\psi \wedge \neg(b = true)$ und $skip$ ist.

Wähle $\phi_1 \vee \phi_2$ als stärkste Nachbedingung von ψ und $if(b, P)$. Für alle $STATE \in State$ gilt:

$O, STATE \models \phi_1 \vee \phi_2$
gdw. es existiert $STATE_1$ *mit* $O, STATE_1 \models \psi \wedge (b = true)$ und
$STATE = v^*(STATE_1, P)[1]$
oder
es existiert $STATE_2$ *mit* $O, STATE_2 \models \psi \wedge \neg(b = true)$ und
$STATE = v^*(STATE_2, skip)[1]$
gdw. es existiert $STATE'$ *mit* $O, STATE' \models \psi$ und
$STATE = v^*(STATE', if(b, P))[1]$

e.) *Aufruf-Regel* $p()$ nichtrekursiv
Induktionshypothese:

Es existiert ein ϕ, wobei ϕ stärkste Nachbedingung von ψ und *com* ist.

Wähle ϕ als stärkste Nachbedingung von ψ und $p()$. Für alle $STATE \in State$ gilt:

$O, STATE \models \phi$
gdw.
es existiert $STATE'$ *mit* $O, STATE' \models \psi$ und $STATE = v^*(STATE', p())[1]$
gdw.
es existiert $STATE'$ *mit* $O, STATE' \models \psi$ und $STATE = v^*(STATE', com)[1]$

Damit ist induktiv die Expressivität für alle Konstrukte gezeigt. □

Nun ist die Expressivität der Gültigkeitsrelation gezeigt. Der Cook'sche Begriff der relativen Vollständigkeit kann nach Satz 7.3.2 und der Bemerkung 7.3.5 nun auf das Beweissystem angewendet werden.

Satz 7.3.4 *(Relative Vollständigkeit) Das Beweissystem nach Definition 7.2.2 ist relativ vollständig.*

Beweis: Folgt direkt aus der Definition der Expressivität 7.3.4 und den Sätzen 7.3.2 und 7.3.3. □

Bemerkung 7.3.3 *Die zusätzlich eingeführten Beweisregeln, die hier im Beweis der Vollständigkeit nicht betrachtet wurden, sind nicht unbedingt notwendig. Die (relative) Vollständigkeit des Beweissystems konnte auch ohne sie gezeigt werden. Das Beweissystem ist damit auch ohne sie mächtig genug, alle geltenden Aussagen ableiten zu können.*

7.4 Erweiterung der Aufrufregel

Die Definition von Beweisregeln für ein mächtiges Prozedurkonzept (wie es in Kapitel 4 vorgestellt wurde) erweist sich als schwierig, da der Abstraktionsmechanismus der Prozeduren mit anderen Konzepten wie Rekursion, Parameterübergabe oder lokalen Objekten kombiniert werden muß. Daher wird den Prozeduren hier ein eigener Abschnitt gewidmet.

Wird ein vollständiges und konsistentes Beweissystem um eine in sich konsistente und vollständige Regel erweitert, so erhalten sich dank der Kompositionalität der syntaktischen Konstrukte und der darauf aufbauenden Regeln Vollständigkeit und Konsistenz. Induktion über die Konstrukte wurde auch schon für die zugrundeliegende Form des Beweissystems eingesetzt (Abschnitte 7.2 und 7.3). Auf dieser Eigenschaft basieren auch die Beweise zu den Sätzen 7.3.1 und 7.3.3. Konsistenz und Vollständigkeit werden in diesem Abschnitt nicht bewiesen. Statt dessen erfolgen Verweise auf die Literatur.

Die Inferenzregel CALL wurde in einfacher Form für parameterlose, nichtrekursive Prozeduren $p() = com$ wie folgt definiert (siehe Abschnitt 7.2):

$$\frac{\phi \rightarrow [com]\ \psi}{\phi \rightarrow [\ p()\ ]\ \psi} \qquad \text{[CALL]}$$

Die Regel CALL soll jetzt bis zum definierten Sprachumfang (siehe Abschnitte 4.1 und 4.2) durch mehrere Regeln schrittweise erweitert werden.

Bevor die Erweiterungen des Beweissystems betrachtet werden, sollen einige kurze Anmerkungen zu notwendigen theoretischen Grundlagen gemacht werden. Auf Basis des Fixpunkttheorems A.4.10 können Eigenschaften des kleinsten Fixpunktes stetiger Funktionen durch Induktion gezeigt werden. Allerdings können nur bestimmte Eigenschaften durch Induktion bewiesen werden (siehe Abschnitt A.4). Diese Eigenschaften lassen sich durch *zulässige* Formeln ausdrücken. Die Beweismethodik wird *Scott'sches Induktionsprinzip* genannt. Die notwendigen Definitionen und Sätze finden sich im Anhang (siehe auch [LS84] Kap. 4.3 *Fixpoints*, S.85ff).

7.4.1 Rekursive, parameterlose Prozeduren

Rekursive Prozeduren rufen sich selbst auf. Hier kann der einfache kompositionale Ansatz nicht verfolgt werden, indem der Prozeduraufruf auf Basis des zuvor bewiesenen Prozedurrumpfes bewiesen werden kann. Die Idee ist nun, Beweise durch Induktion über die rekursiven Aufrufe zu führen. Als Prämisse wird die Annahme benutzt, daß sich $\phi \rightarrow [com]\ \psi$ aus dem inneren, eingeschachtelten rekursiven Aufruf von einer Prozedur p (mit der Spezifikation $\phi \rightarrow [\ p()\ ]\ \psi$) herleiten läßt. Diese Forderung ist eine Transformation des Scott'schen Induktionsprinzips (siehe Satz A.4.13) in den Formalismus des Beweissystems. Siehe hierzu auch Kapitel 3.3 und 3.7 in [Apt81] und [Fra92] S.155ff.

$$\frac{\phi \rightarrow [\ p()\ ]\ \psi \quad \vdash \quad \phi \rightarrow [\ com\ ]\ \psi}{\phi \rightarrow [\ p()\ ]\ \psi} \qquad [\mathrm{CALL-REC}]$$

Mit der Prämisse soll ausgedrückt werden, daß aus dem inneren Aufruf von p der diesen Aufruf umgebende Rumpf *com* abgeleitet werden kann. Dies ist eine Metaregel, die eine Beweisbarkeitsforderung aufstellt.

[LS84] betrachten die Konsistenz und (relative) Vollständigkeit dieser Regel. Die Vollständigkeit wird dort in Satz 8.5 (iv) gezeigt. Zum Beweis der relativen Vollständigkeit ist die Expressivität zu zeigen (siehe Satz 7.3.3 und [LS84] Satz 8.16 (iv)). In den beiden angesprochenen Sätzen in [LS84] werden allerdings keine rekursiven Prozeduren, sondern *while*-Schleifen betrachtet, die aber rekursiv definiert sind. Beide Definitionen sind trotzdem vergleichbar, da sie beide durch Fixpunktsemantik definiert sind, genauer über den kleinsten Fixpunkt des zugehörigen Funktionals (vgl. Abschnitt 4.1.2 und [LS84] Abschnitt 5.4).

7.4.2 Nichtrekursive, parametrisierte Prozeduren

Sei die Prozedur p durch $p(rd\ y, wr\ z) = com$ definiert.

$$\frac{\phi \rightarrow [asgn(y', e); com_y^{y'}; asgn(x, z)]\ \psi}{\phi \rightarrow [\ p(e, x)\ ]\ \psi} \qquad [\mathrm{CALL-PAR}]$$

Diese Regel macht die Parameterübergabemechanismen explizit (vgl. Abschnitt 4.2.2). Die aktuellen *rd*-Parameter werden vor Ausführung des Rumpfes an (temporäre) formale Parameter (hier y') übergeben ($com_y^{y'}$ beschreibt die Substitution aller Vorkommen von y durch y' in *com*). Nach Ausführung des Rumpfes werden die in den formalen Parametern berechneten

Ergebnisse der wr-Parameter an die aktuellen wr-Parameter übergeben (hier von z an x). Diese Regel liefert eine konsistente und relativ vollständige Erweiterung eines Inferenzsystems (siehe [Apt81] Kapitel 6.2.1 und 6.2.2).

7.4.3 Rekursive, parametrisierte Prozeduren

Die Regel für parameterlos-rekursive Prozeduren CALL-REC kann für parametrisierte, rekursive Prozeduren, wie etwa $p(rd\ y, wr\ z) = com$, wobei p in com aufgerufen wird, wie folgt erweitert werden:

$$\frac{\phi(t_i, x_i) \rightarrow [\ p(t_i, x_i)\ ]\ \psi(t_i, x_i) \quad \vdash \quad \phi(t_i, x_i) \rightarrow [asgn(y_i', t_i); com_{y_i}^{y_i'}; asgn(x_i, z_i)]\ \psi(t_i, x_i)}{\phi(t_1, x_1) \rightarrow [\ p(t_1, x_1)\ ]\ \psi(t_1, x_1)} \quad i = 2, .., n$$

[CALL-PAR-REC]

Die Konklusion beschreibt den Aufruf für die äußerste Ebene (Index 1). Für alle inneren Aufrufe von p (Index $i > 1$) muß der Rumpf bewiesen werden. ϕ und ψ können von den Parametern abhängen. Nach J. de Bakker [dB80] ist diese Regel konsistent und relativ vollständig.

7.4.4 Prozeduren mit lokalen Variablen

Die Regel CALL soll um die Behandlung lokaler Variablen erweitert werden. Sei p durch $p(rd\ y, wr\ z) \ \equiv \ local\ v\ begin\ com\ end$ definiert. x sei eine frische, mit ω initialisierte Variable, die in ϕ, ψ, com nicht vorkommt.

$$\frac{\phi \wedge x = \omega \rightarrow [com_v^x]\ \psi}{\phi \rightarrow [\ p(y, z)\ ]\ \psi} \qquad [\mathrm{CALL} - \mathrm{VAR}]$$

Zur Semantikdefinition von lokalen Variablen in Prozeduren benutzt K. Apt einen Mechanismus, der dem hier vorgestellten entspricht (vgl. Kapitel 4.1.1 und [Apt81], [Cla79]). Der Zustand ist dort eine endliche Funktion von der Menge der Variablen in den Domain einer Interpretation (analog zu $STATE : IDENT \rightarrow VAL$). Die Veränderung von Zuständen für neue Blöcke erfolgt durch eine Erweiterungs- und eine Löschoperation, die den *push*- und *pop*-Operationen aus Kapitel 4.1.1 entsprechen. Dies ist keine Erweiterung von CALL-PAR. CALL-VAR ist zwar auch für Parameter definiert, die Behandlung aktueller Parameter — wie in CALL-PAR festgelegt — erfolgt hier nicht. Die lokale Variable wird wie ein vor Aufruf undefinierter rd-Parameter behandelt. Ein um diese Regel erweitertes Beweissystem ist konsistent und vollständig im Cook'schen Sinne (siehe [Apt81]).

7.5 Ein Beweissystem für die vollständige modale Logik

Das in Kapitel 7.2 vorgestellte Beweissystem ist nur für die eingeschränkte modale Logik aus Kapitel 5.2 anwendbar. Der Einsatz eines solchen Beweissystems liegt im wesentlichen in der Verifikation von Implementierungen gegenüber Spezifikationen. Ein Beweissystem für die vollständige modale Logik nach Kapitel 5.3 soll im folgenden angegeben werden. Dieses wird (nach [Coo78]) konsistent, aber nur relativ vollständig sein (siehe [KT90]). Das Beweissystem ist [KT90] entnommen. Dies ist möglich, da das Beweissystem dort für Kripke-Modelle definiert ist und Σ-Objekte in Satz 5.4.2 als Kripke-Modelle nachgewiesen werden konnten. Der Box-Operator wird dort, wie auch in Kapitel 5, als partielle Korrektheitsaussage interpretiert (siehe [KT90] Kapitel 2.1, 3.2, 3.4). Der Diamond-Operator ist die negierte Form des Box-Operators. Die prädikatenlogischen Konnektoren werden wie üblich behandelt. Die Gültigkeitsdefinitionen von [KT90] und Kapitel 5 sind also gleichwertig.

Ein **modales Beweissystem** kann aus den folgenden Axiomen und Inferenzregeln bestehen (nach [KT90]):

Axiome:

1. alle Axiome einer Prädikatenlogik erster Stufe
2. $< P > \phi \ \wedge \ < P > \psi \quad \Rightarrow \quad < P > (\phi \wedge \psi)$
3. $< P > (\phi \ \vee \ \psi) \quad \Leftrightarrow \quad < P > \phi \ \vee \ < P > \psi$
4. $< asgn(x,t) > \phi \quad \Leftrightarrow \quad \phi[x/t]$

Inferenzregeln:

1. Modus Ponens:
$$\frac{\phi, \phi \rightarrow \psi}{\psi}$$
2. Modale Generalisierung:
$$\frac{\phi}{[P]\phi}$$
3. Monotonie von $[P]$:
$$\frac{\phi \rightarrow \psi}{[P]\phi \rightarrow [P]\psi}$$
4. Monotonie von $< P >$:
$$\frac{\phi \rightarrow \psi}{< P > \phi \ \rightarrow < P > \psi}$$
5. Die Hoare'schen Regeln ASGN, SKIP, SEQUENCE, CONDITIONAL, CALL, CONS (siehe Abschnitt 7.2) und ggf. die Prozedurregeln aus Abschnitt 7.4.

Weitere, ableitbare Eigenschaften sind:

- $[P]\phi \vee [P]\psi \Rightarrow [P](\phi \vee \psi)$
- $[P](\phi \wedge \psi) \Leftrightarrow [P]\phi \wedge [P]\psi$
- $[P](\phi \rightarrow \psi) \Rightarrow ([P]\phi \rightarrow [P]\psi)$

Bemerkung 7.5.1 *Das angegebene Beweissystem und seine Erweiterungen sind konsistent und relativ vollständig (nach [KT90] Kapitel 3.4 Theoreme 29,31).*

Im folgenden sollen einige Aussagen über Zusammenhänge zwischen der imperativen Spezifikation von Programmfragmenten und deren logischer Spezifikation gemacht werden.

Lemma 7.5.1 *Es gilt für ein Σ-Objekt O, einen beliebigen Zustand STATE und ein terminierendes Programmfragment P, welches asgn(z, t) als letzten Seiteneffekt auf z enthält:*

$$Aus \quad O, STATE \models [asgn(z,t)]\ true \quad folgt \quad O, STATE \models [P]\ z = t$$

Beweis:

$O, STATE \models [asgn(z,t)]\ true$
gdw.
falls $asgn(z,t)$ terminiert, dann $O, STATE' \models true$ gilt mit

$$STATE'(x) = \begin{cases} STATE(x) & f\ddot{u}r\ z \neq x \\ v^*(STATE, t)[2] & f\ddot{u}r\ z = x \end{cases}$$

⇒
falls P terminiert, dann
$O, v^*(STATE, P)[1] \models z = t$ gilt
gdw.
$O, STATE \models [P]\ z = t$

□

Durch dieses Lemma kann das primitive Kommando *asgn* durch ein geeignetes Programmfragment (etwa eine Prozedur) abstrahiert werden. Dies ist hilfreich, wenn abstrakte Schnittstellen zu einem Programmfragment gebildet werden sollen.

Lemma 7.5.2 *Es gilt für ein Σ-Objekt O und einen beliebigen Zustand STATE:*

$$O, STATE \models [P][Q]\ \psi \quad gdw. \quad O, STATE \models [seq(P,Q)]\ \psi$$

Beweis: Nach Abschnitt 3.4 und Definition 5.2.2 gilt:

$O, STATE \models [P][Q]\ \psi$
gdw.
falls P terminiert, dann gilt $O, v^*(STATE, P)[1] \models [Q]\ \psi$
gdw.
falls P terminiert und dann Q terminiert, dann gilt
$O, v^*(v^*(STATE, P)[1], Q)[1] \models \psi$
gdw.
falls erst P und dann Q terminiert, dann gilt
$O, v^*(STATE, seq(P,Q))[1] \models \psi$
gdw.
$O, STATE \models [seq(P,Q)]\ \psi$

□

Sequenzen lassen sich also durch Verschachtelung des Box-Operators darstellen.

Lemma 7.5.3 *Es gilt für ein Σ-Objekt O, einen beliebigen Zustand STATE und falls eine Formel φ in der imperativen Sprache als boolescher Ausdruck darstellbar ist:*

$$Aus \quad O, STATE \models [if(\phi, P)]\ \psi \quad folgt \quad O, STATE \models \phi \rightarrow [P]\ \psi$$

Beweis: Nach Abschnitt 3.4 und Definition 5.2.2 gilt:

$O, STATE \models [if(\phi, P)]\ \psi$
gdw.
falls $(if(\phi, P))$ terminiert, dann $O, v^*(STATE, if(\phi, P))[1] \models \psi$ gilt
gdw.
falls
ϕ nicht gilt (d.h. $v^*(STATE, \phi)[2] \neq_{bool} true$) und
$O, STATE \models \psi$ gilt ($STATE = v^*(STATE, if(\phi, P))[1]$, falls ϕ nicht gilt)
oder
ϕ gilt und falls P terminiert, dann $O, v^*(STATE, P)[1] \models \psi$ gilt
$\Rightarrow$
falls
ϕ nicht gilt
oder
falls P terminiert, dann $O, v^*(STATE, P)[1] \models \psi$ gilt
gdw.
ϕ nicht gilt oder $O, STATE \models [P]\ \psi$ gilt
gdw.
$O, STATE \models \phi \rightarrow [P]\ \psi$

□

Mit gewissen Einschränkungen läßt sich das bedingte Kommando also axiomatisch darstellen.

Mit den letzten drei Lemmata konnten Eigenschaften gezeigt werden, die Abhängigkeiten zwischen der logischen Sprache und der Sprache des Berechnungsmodells herstellen. Alle drei können dazu genutzt werden, Programmfragmente zu abstrahieren. Für ein Σ-Objekt O und einen beliebigen Zustand $STATE$ gilt:

1. Aus $O, STATE \models [asgn(z,t)]\ true$ folgt $O, STATE \models [P]\ z = t$, falls P terminiert und $asgn(z,t)$ als letzten Seiteneffekt auf z enthält.
2. $O, STATE \models [P][Q]\ \psi$ gilt genau dann, wenn $O, STATE \models [seq(P,Q)]\ \psi$.
3. Aus $O, STATE \models [if(\phi, P)]\ \psi$ folgt $O, STATE \models \phi \rightarrow [P]\ \psi$, falls ϕ als boolescher Ausdruck im Berechnungsmodell darstellbar ist.

7.6 Verifikation imperativer Spezifikationen

Jetzt kann auch formal gezeigt werden, daß das Beispiel aus der Einleitung dieses Kapitels eine korrekte Implementierung beschreibt (Beispiele 7.1.1 und 7.1.2). Zuerst wird für die Kommandos der Implementierung das Ein-/Ausgabeverhalten bestimmt. Da der Korrektheitsnachweis für *init* trivial ist, soll nur p betrachtet werden. Für die Zuweisung ergibt sich durch Anwendung des Axioms ASGN folgendes Ein-/Ausgabeverhalten (es wird die explizite Form mit Spezifikationsvariable X benutzt):

$$s = X \ \rightarrow [asgn(s, succ(s))]\ s = succ(X)$$

$s = succ(X)$ kann nach Satz 7.3.3 (a) für X als Konstante c abgeleitet werden. $s = X$ ergibt sich dann über die Substitution. Darauf aufbauend kann die bedingte Anweisung betrachtet werden (Regel CONDITIONAL):

$$s = X \wedge s > 0 \rightarrow [if(s > 0, asgn(s, succ(s))]\ s = succ(X)$$

$s > 0$ kommt hinzu, wenn die Bedingung der bedingten Anweisung in die Vorbedingung gezogen wird. Damit ergibt sich für die gesamte Implementierung (nach CALL):

$$s = X \wedge s > 0 \rightarrow [impl]\ s = succ(X)$$

Da Vor- und Nachbedingung mit der Spezifikation identisch sind, ist die Implementierung korrekt. Andernfalls könnte mit CONS abgeschätzt werden.

Die Vorbedingung der abgeleiteten Hoare-Formel zu einer Prozedurimplementierung beschreibt die Vorbelegungen der im Prozedurrumpf freien Variablen (Zustandskomponenten und Parameter) soweit bekannt konkret, sonst über Quantifizierung ausgedrückt. Dies ist die schwächste mögliche Form der Vorbedingung. Diese Vorbedingung kann generiert werden. Jede freie Variable wird auf eine beliebige, nicht weiter spezifizierte Konstante gesetzt. Relativ zu dieser Konstante wird dann das Ein-/Ausgabeverhalten abgeleitet.

Wie eben gesehen, können axiomatische Spezifikationen für *Prozeduren* durch die Hoare-Logik abgeleitet werden. Es sollen nun noch *Attribute* betrachtet werden. Sei ein Attribut durch $f(x_1, \ldots, x_n) := t$ imperativ definiert. t ist hierbei ein beliebiger Term der Sorte s, falls $f : state \times s_1 \times \ldots \times s_n \rightarrow state \times s$. t ist Term einer Datensorte, also kein Kommando. Dann läßt sich dieses Attribut direkt in die Gleichung $f(x_1, \ldots, x_n) = t$ umsetzen. Der Unterschied besteht also nur in der Notation (:= bzw. =). In der axiomatischen Form kann noch anhand von Inferenzregeln umgeformt werden.

7.7 Zusammenfassung

Es wurde ein Beweissystem für eine dynamische Logik vorgestellt, das Hoare-Kalküle [Hoa69] umfaßt. Einsatzzweck eines solchen Beweissystems ist die Verifikation einer Operationsimplementierung anhand einer axiomatischen Spezifikation sowie Theoriebildung und Nachweis von Eigenschaften auf rein axiomatischer Ebene. Aus methodischer Sicht ist dies nicht das Hauptziel dieser Arbeit, da Verifikation ein *a posteriori*-Verfahren ist. Ein korrektheitserhaltendes Entwickeln ist so nicht möglich. E. Dijkstra [Dijk76], D. Gries [Gri81] und C. Morgan [Mor88, Mor94] zeigen Möglichkeiten auf, um aus den Techniken des *a posteriori*-Verfahrens der Verifikation einen konstruktiveren Ansatz zur systematischen, korrektheitserhaltenden Programmentwicklung zu erhalten. Notwendig ist die Erarbeitung des Beweissystems aber schon, da der Anspruch dieser Arbeit in der vollständigen formalen Ausleuchtung des Übergangs zwischen axiomatischer und imperativer Beschreibung von Programmen liegt. Ähnliche Ansätze werden mit EML oder VDM verfolgt. Dort gibt es ebenfalls Beweissysteme, die als Basis einer konstruktiven Entwicklungsmethodik zugrundeliegen. Dieser Gedanke soll auch in den nächsten Kapiteln verfolgt werden.

Die Definition des Beweissystems in diesem Kapitel ist von drei Faktoren erheblich erschwert worden:

1. prinzipielle Unvollständigkeit eines Beweissystems für Hoare-Logiken,

2. Rekursion in der Definition von Prozeduren,
3. Prozedurdefinitionen im allgemeinen.

Um zur Bearbeitung der Rekursion die entstehende Zirkularität zu umgehen und wieder einen kompositionalen Ansatz zu erhalten, wurde eine Metaregel eingeführt, die eine Beweisbarkeitsforderung als eine ihrer Prämissen hat. Da Rekursion zu nichtterminierenden Ausführungen führen kann, wäre dies in Terminierungsuntersuchungen zur totalen Korrektheit noch gezielt zu betrachten. Die Realisierung eines mächtigen Prozedurkonzeptes als ein Mechanismus zur Abstraktion ist ein wichtiges, aber in Hinsicht auf die Definition eines passenden Beweissystems erschwerendes Konzept. Diese Komplexität entsteht zum Teil dadurch, daß Prozeduren in Kombination mit anderen Konzepten wie Verschachtelung oder Parameterübergabe realisiert werden müssen. Kanonische Beweisregeln, d.h. solche, die das Prinzip der Kompositionalität unterstützen, sind mitunter komplex. Prozeduren werden aber als Abstraktionsmittel unabhängig von der Ausführung entwickelt und verifiziert. Sie sollten dann in beliebigen Kontexten ohne weitere Verifikation aufgerufen werden können. Nichtkanonische Regeln garantieren letzteres nicht.

Es konnte in diesem Kapitel gezeigt werden, daß ein konsistentes und relativ vollständiges Beweissystem für eine Hoare-Logik über einer imperativen Programmiersprache als auch für eine vollständige dynamische Logik definiert werden kann. Der Mächtigkeit der Programmiersprache sind allerdings Grenzen gesetzt, wie E.M. Clarke in [Cla79] zeigt. Nicht für jede Kombination von Sprachkonstrukten kann ein (relativ) vollständiges Beweissystem existieren. Clarke hat blockstrukturierte Programmiersprachen mit den Eigenschaften *Prozeduren als Parameter*, *Rekursion*, *Static Scoping*, *globale Variablen* und *verschachtelte Prozeduren* untersucht. Ein vollständiges Beweissystem kann erzielt werden, wenn mindestens eine der Eigenschaften ausgeschlossen wird (bei *Static Scoping* kann auf *Dynamic Scoping* ausgewichen werden). Einen weiteren problematischen Bereich bilden Sprachen, die ein Koroutinen-Konzept, *Call_by_Name*-Parameterübergabe oder bestimmte Zeigerkonstrukte anbieten. Hierauf soll aber nicht weiter eingegangen werden. Die Frage nach weiteren geeigneten Einschränkungen als die von Clarke angesprochenen bleibt noch offen. Ein Blick in aktuelle Übersichtsliteratur [Fra92, AO94] zeigt aber, daß hier wahrscheinlich keine wesentlich neuen Erkenntnisse erzielt werden konnten.

Verifikation ist nur mit maschineller Unterstützung praktikabel. Der Einsatz eines *Theorembeweisers*, der nur auf dem Beweissystem einer zugrundeliegenden Logik arbeitet, erfordert vom Benutzer einen inakzeptabel hohen Aufwand an Wissen, Erfahrung und Zeit (siehe Erfahrungen aus dem KORSO-Projekt [BJ94]). Ansätze wie das *taktische Theorembeweisen* [Hei92a] zeigen erste Verbesserungen. Hier werden Beweisstrategien in einer Metasprache implementiert. Zuerst werden abgeleitete Inferenzregeln auf dem gegebenen Kalkül realisiert. Sie stellen erste Abstraktionen dar. Taktiken sind dann Programme der Metasprache, die konkrete logische Bausteine behandeln (etwa bestimmte Sequenzen einer Ableitung). Strategien sind übergeordnete Programme, in denen auch die Benutzerführung und Heuristiken eingearbeitet sind. Ein Beispiel für diesen Ansatz ist die Formalisierung der Programmentwicklungsmethode von D. Gries [Gri81] in dynamischer Logik. Die Realisierung erfolgte für den KIV-Theorembeweiser der Universität Karlsruhe [Hei92b].

Verifikation kann nur Korrektheit gegenüber einer initialen, formalen Spezifikation garantieren. Deren Korrektheit gegenüber informellen Anforderungsbeschreibungen kann nur validiert werden. In sie schleichen sich Fehler ein, etwa durch Mißverständnisse gegenüber dem Auftraggeber oder durch fehlerhafte Umsetzung der vom Spezifizierer intendierten Eigenschaften in den Formalismus. [Fra92] analysiert im Abschnitt 2.4 *Specifications and Intentions* fehlerträchtige Situationen.

Dieser Ansatz beschränkt sich auf partielle Korrektheit. Der Modellbegriff für Prozeduren basiert auf einer partiellen Korrektheitsaussage. Die Betrachtung expliziter Terminierungsaussagen ist daher in diesem Ansatz nicht sinnvoll. Im Kapitel zur Spezifikationslogik wurde schon angesprochen, daß Terminierung im Rahmen einer Erweiterung betrachtet werden könnte. Ansätze hierzu sind in [Fra92, AO94] für Verifikation im Sinne von Hoare und in [KT90] für die dynamische Logik zu finden.

Dieses Kapitel beschäftigte sich mit der Frage, wann eine Implementierung korrekt in bezug auf eine Spezifikation ist. Im Rahmen des Hoare-style Beweissystems wurden nur imperative Implementierungen betrachtet. Es können aber, indem die Regel CONS sowie die Regeln der modalen Logik benutzt werden, in gleicher Weise auch axiomatische Implementierungen behandelt werden. Spezifikationsansätze, die algebraische Spezifikationen mit imperativen oder objektorientierten Programmiersprachen verknüpfen [Bre91, Lin93], müssen zwei unterschiedliche Implementierungsbegriffe verwenden, um Implementierungen sowohl zwischen zwei axiomatischen als auch zwischen einer axiomatischen und einer operationalen Spezifikation beschreiben zu können. Die vereinheitlichte Modellbildung axiomatischer und operationaler Spezifikation auf Basis der Σ-Objekte erlaubt hier auch die Vereinheitlichung des Implementierungsbegriffes.

Erweiterungen von Hoare-style Beweissystemen zur Betrachtung des Verhaltens von Objektsystemen sind auch an anderer Stelle zu finden. Während hier durch den Einsatz dynamischer Logik abstrakt das Verhalten von Programmfragmenten betrachtet werden soll, gibt es stärker prozeßorientierte Spezifikationsansätze (vgl. Abschnitt 5.4.4), in denen Aspekte der Prozeßerzeugung und Prozeßkommunikation im Vordergrund stehen (temporale Logiken oder CSP [Hoa78] und deren Nachfolger). POOL (Parallel Object-Oriented Language, [AR90]) basiert auf CSP. In [AdB94] wird ein Hoare-style Beweissystem für diese Sprache vorgestellt. Näher am vorliegenden Ansatz liegt die Arbeit von G. Antoniou [Ant94]. Dort bildet ein Modulkonzept die Basis, das es gestattet, imperative Implementierungen algebraisch in Schnittstellen zu abstrahieren. Das Modulkonzept basiert auf [EM90]. Zur Verifikation der Implementierung gegenüber der algebraischen Schnittstellenspezifikation wird ebenfalls ein Beweissystem im Hoare'schen Stil definiert. Antoniou stellt in seiner Abschlußbetrachtung fest, daß der Übergang von Hoare-Logik zu dynamischer Logik einige von ihm aufgezeigte Schwächen (zum einen auf der Unvollständigkeit basierend, zum anderen durch die zu starke Fixierung der Hoare-Logik auf Vor- und Nachbedingungen für Programmfragmente bedingt) lösen könnte.

Teil III

Relationen zwischen Spezifikationen

In diesem Teil werden wir verschiedene Formen von Beziehungen zwischen Spezifikationen untersuchen. Diese Relationen sind stark mit dem Prozeß der Entwicklung von Komponenten verknüpft, der in der Methodik zusammengefaßt ist. Grundelemente einer formal gestützten Entwicklungsmethodik könnten die Entwicklungstechniken *Modularisierung*, *Implementierung* und *Verfeinerung* sein.

Wichtig in der Entwicklung von Software ist die Zerlegung des Gesamtsystems durch funktional abgeschlossene, überschaubare Einheiten, die Module. *Modularisierung* soll durch konstruktive Methoden unterstützt werden. Es sollen Operatoren angeboten werden, die die Einhaltung eines Korrektheitsbegriffes bei Modularisierung garantieren. Wiederverwendung kann auf Spezifikationsebene durch *Spezifikationsoperatoren* unterstützt werden, die die Anpassung einer Komponente an einen Anforderungskontext ermöglichen. Zur Modularisierung werden Spezifikationen auf parametrisierte Spezifikationen erweitert. Ein Konstruktor wird angeboten, der die Komposition zweier parametrisierter Spezifikationen über die Aktualisierung eines formalen Imports realisiert. Die zwischen der Importschnittstelle einer parametrisierten Spezifikation und einer zu importierenden Schnittstelle entstehende Beziehung heißt Benutztbeziehung. Für sie kann ein Korrektheitsbegriff definiert werden. *Implementierung im engeren Sinne* ist der Kernbegriff einer Entwicklung, die den Übergang von einer abstrakten, axiomatischen Beschreibung von Programmeigenschaften bis zu ausführbaren, imperativen Programmbeschreibungen charakterisiert. Es ist dabei die Frage zu klären, ob die Spezifikation auf niedrigerem Abstraktionsniveau die Anforderungen der Spezifikation auf höherem Niveau erfüllt. Als *Verfeinerung* bezeichnet man die Weiterentwicklung von Komponenten durch Verschärfen oder Erweitern der Eigenschaften. Die Trennung zwischen Implementierung und Verfeinerung ist nicht eindeutig. Wir werden später Verfeinerung als ein Implementierung umfassendes Konzept vorstellen. Weitere zu betrachtende Relationen sind die *Subtyprelation* und die *Vererbungsrelation* der objektorientierten Programmierung.

Die vorgeschlagenen Techniken wie Modularisierung oder Implementierung lassen sich in zwei Entwicklungsdimensionen klassifizieren. **Vertikale Entwicklung** bezeichnet das Absenken des Abstraktionsniveaus, d.h. Transformation einer abstrakten Spezifikation in eine konkretere Implementierung mit der Absicht, ein System zu realisieren. Hierzu gehört die Technik der Implementierung (i.e.S.) (siehe Kapitel 8). Bei **horizontaler Entwicklung** werden Spezifikationen ohne Ändern des Abstraktionsniveaus bearbeitet, d.h. Komponenten werden einzeln modifiziert oder mehrere Komponenten werden zu komplexeren zusammengesetzt. Hierzu gehört die Modularisierung, die die Komposition von Systemen beschreibt (siehe Kapitel 9), hierzu kann aber auch die Verfeinerung gerechnet werden. Horizontale Entwicklung dient zur Modellierung eines Systems auf einer Abstraktionsebene. Diese Modellierung kann unabhängig von einer Realisierungsabsicht erfolgen. Die Unterscheidung erfolgt also anhand des Kriteriums, ob ein Wechsel des Abstraktionsniveaus erfolgt oder nicht. Eine Verknüpfung zwischen beiden Dimensionen schafft die horizontale Kompositionseigenschaft, die die Verträglichkeit zwischen einem Konstrukt der horizontalen Entwicklung (Komposition) und einem Konstrukt der vertikalen Entwicklung (Implementierungsrelation) herstellt. Die vertikale Kompositionseigenschaft fordert die Transitivität der Implementierungsrelation.

Die *Vererbung* hat als Aspekte Wiederverwendung und Spezialisierung. Spezialisierung bedeutet Modellierung ähnlichen Verhaltens. Subtypen und Vererbung werden im Rahmen eines eigenständigen Kapitels zur objektorientierten Entwicklung (Kapitel 10) betrachtet.

Kapitel 8

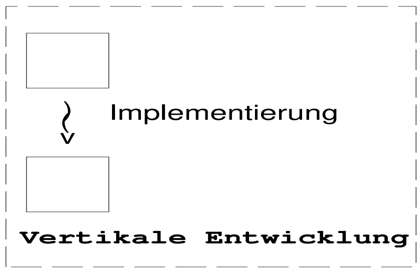
Implementierung und vertikale Entwicklung

Der Implementierungsbegriff wird in der Literatur mit verschiedenen Bedeutungen verwendet und wird auch hier in zwei Formen benutzt.

1. Implementierung bezeichnet häufig eine semantikerhaltende *Verfeinerung* von Spezifikationen. Dies ist *Implementierung im weiteren Sinne*. Formal wird das (in Abschnitt 8.1) durch eine auf Modellklasseninklusion basierende Implementierungsrelation gefaßt. Die partielle Korrektheit wird dabei erhalten. Es wird also nicht notwendigerweise das Abstraktionsniveau gesenkt.

2. *Implementierung im engeren Sinne* beinhaltet auch ein Senken des Abstraktionsniveaus. Konkrete Repräsentationen für abstrakt definierte Datentypen werden ausgewählt. Abstrakt spezifizierte Operationen werden realisiert, d.h. Algorithmen werden ausgewählt. Damit ist Implementierung i.e.S. eine Technik des vertikalen Entwickelns.

In diesem Kapitel wird es zuerst um die Implementierung i.w.S. gehen, was zu einer Definition einer Implementierungsrelation führt (siehe Abbildung rechts), und dann wird vertikale Entwicklung durch Implementierung i.e.S. betrachtet.



Die Begriffsverwirrung könnte umgangen werden, wenn die in Abschnitt 8.1 als Implementierungsrelation bezeichnete Relation $\rightsquigarrow$ umbenannt würde (sie realisiert Implementierung i.w.S.). Dies wäre aber nicht mehr konform zur üblichen Begriffsbildung in algebraischen Spezifikationsansätzen. Daher wird der Begriff der Implementierungsrelation beibehalten. Die Gültigkeit, und somit auch die Modellklassenbildung, basiert auf partieller Korrektheit. Auf Modellklasseninklusion basierend, setzt die Implementierungsrelation also partielle Korrektheit zwischen zwei Spezifikationen um. Partielle Korrektheit wird in diesem Ansatz durch Modellbildung und durch das Beweissystem unterstützt (siehe Kapitel 5 und 7). Partielle Korrektheit ist daher fundamental und somit auch geeignet, Basis weiterer Definitionen für horizontale und vertikale Entwicklung zu sein. Eine Variante der Implementierungsrelation

$\rightsquigarrow$, die für einen Spezifizierer handhabbarer ist, wird ebenfalls angeboten. Beide Relationen werden zunächst anhand des Korrektheitsbegriffes motiviert (Abschnitt 8.1). Sie werden dann verglichen (Abschnitt 8.2). Abschnitt 8.3 befaßt sich mit der vertikalen Entwicklung. In Abschnitt 8.4 wird inkrementelle Entwicklung vorgestellt.

Die Implementierungsrelation $\rightsquigarrow$ soll hier als fundamentale Relation dienen. Sie soll sowohl in die vertikale als auch in die horizontale Entwicklung eingehen.

8.1 Korrekte Implementierungen

Es soll nun das Beweissystem aus Kapitel 7 eingesetzt werden, um das Ein-/Ausgabeverhalten eines Programmfragmentes zu betrachten. Dazu werden zunächst einige Korrektheitsbegriffe entwickelt. Es soll jetzt mit Hilfe des definierten Beweissystems folgende Frage beantwortet werden:

> Ist die vorliegende Implementierung $\phi_{P_2} \rightarrow [P_2]\ \psi_{P_2}$ eine geeignete Implementierung der Spezifikation $\phi_{P_1} \rightarrow [P_1]\ \psi_{P_1}$?

Dies soll durch eine *Implementierungsrelation* formal erfaßt werden. Eine auf Modellklasseninklusion basierende Implementierungsrelation würde partielle Korrektheit beschreiben. Dies geht aus den Anmerkungen zur Modellklassenbildung in Abschnitt 5.2.4 hervor. Modelle sind partiell korrekt bzgl. ihrer Spezifikation. Jetzt soll partielle Korrektheit auch auf Spezifikationsebene zwischen zwei Spezifikationen durch eine Relation erfaßt werden können. Modellklasseninklusion ist daher eine geeignete Semantik einer Implementierungsrelation. Der hier einzusetzende Implementierungsbegriff soll aber auch möglichst konstruktiv gestaltet werden (konstruktiver als der auf Modellklasseninklusion basierende es ist). Es ist also eine zweite Relation zu definieren. Dies schließt sich an die Abschlußbemerkungen aus Abschnitt 7.7 an, in denen das Fehlen von Konstruktivität in der Technik der Verifikation angemerkt wurde. Der am Anfang des Kapitels eingeführte Begriff der partiellen Korrektheit, der auf semantischen Eigenschaften basiert, soll hier um einen syntaktisch basierten, also besser handhabbaren und somit konstruktiveren Korrektheitsbegriff ergänzt werden.

8.1.1 Korrektheit von Prozedurimplementierungen

Ein Korrektheitsbegriff zwischen zwei Spezifikationen soll nun schrittweise entwickelt werden. Zuerst sollen dazu Prozeduren betrachtet werden.

Definition 8.1.1 *Seien p_1, p_2 Prozeduren mit $\phi_{p_1} \rightarrow [p_1(\ldots)]\ \psi_{p_1}$ und $\phi_{p_2} \rightarrow [p_2(\ldots)]\ \psi_{p_2}$ als Spezifikationen der Prozeduren. Die Relation $p_1 \rhd^{op} p_2$* **gilt***, falls*

$$\phi_{p_1} \rightarrow \phi_{p_2} \wedge \psi_{p_2} \rightarrow \psi_{p_1}$$

Bemerkung 8.1.1 *Es sollen hier für Definition 8.1.1* terminierende Prozeduren *angenommen werden, so daß durch Ausführung einer Prozedur die Nachbedingung immer etabliert werden kann.*

Die Definition 8.1.1 kann auch auf beliebige Programmfragmente übertragen werden. Eine Objektspezifikation soll später als korrekt implementiert gelten, wenn alle Prozeduren korrekt implementiert werden.

Eine Bedingung ist *stärker* als eine andere, wenn sie diese impliziert, sonst heißt sie *schwächer*. In der Implikation $x \geq 5 \rightarrow x \geq 0$ ist $x \geq 5$ stärker. Soll eine Vorbedingung schwächer als $x \geq 5$ sein, so muß sie von $x \geq 5$ impliziert werden. [Mey88] wendet diese Begriffe auf Vor- und Nachbedingungen an, um die Redefinitionen von Operationen bei Spezialisierungen zu definieren. Redefinierte Operationen sollen 'besser' als die originalen Definitionen sein. Die Implementierung muß die schwächere Vorbedingung haben, d.h. weniger eingeschränkt sein (auf mehr Eingabedaten arbeiten). Sie muß außerdem die stärkere Nachbedingung haben, also mehr oder präzisere Ergebnisse liefern. Zusammenfassend muß die Implementierung also 'genauso gut oder besser' sein als es die Spezifikation fordert, um akzeptiert zu werden. Die oben in Definition 8.1.1 angegebene Korrektheit für Prozeduren realisiert unter der gegebenen Einschränkung der Terminierung die Redefinitionsregel von Eiffel [Mey92b].

Statt der Korrektheit einer Implementierung über die Relation $\triangleright^{op}$ ist auch eine andere Definition möglich, etwa wie schon angesprochen über Modellklasseninklusion. Siehe hierzu auch [Mey88] S.126f, der dort die Semantik des Hoare'schen Tripels $\{P\}A\{Q\}$ wie folgt definiert:

> Die Ausführung von A, falls sie in einem P erfüllenden Zustand gestartet wird, endet in einem Zustand, der Q erfüllt.

Dies ist eine totale Korrektheitsaussage, d.h. partielle Korrektheitsaussage und Terminierungsaussage sind zusammengefaßt. Somit kann die Etablierung der Nachbedingungen immer vorausgesetzt werden. Wir wollen für die folgende Definition auf die Terminierungsforderung verzichten.

Definition 8.1.2 *Seien p_1, p_2 Prozeduren zu einer Signatur Σ mit $\phi_{p_1} \rightarrow [p_1(\ldots)]\ \psi_{p_1}$ und $\phi_{p_2} \rightarrow [p_2(\ldots)]\ \psi_{p_2}$. Die Relation $p_1 \leadsto^{op} p_2$* **gilt***, falls jede Funktion, die p_2 in einem Modell zur Signatur Σ, das $\phi_{p_2} \rightarrow [p_2(\ldots)]\ \psi_{p_2}$ erfüllt, interpretiert, auch Interpretation von p_1 in einem Modell zur Signatur Σ, das $\phi_{p_1} \rightarrow [p_1(\ldots)]\ \psi_{p_1}$ erfüllt, ist.*

Falls $p_1 \leadsto^{op} p_2$ gilt, ist p_2 partiell korrekt bezüglich der Spezifikation von p_1.

8.1.2 Korrektheit von Spezifikationsimplementierungen

Wir werden nun die prozedurbezogenen Definitionen aus Abschnitt 8.1.1 auf Spezifikationen erweitern.

Definition 8.1.3 *Seien sp_1 und sp_2 Spezifikationen mit Signatur $\Sigma = < Z, S, OP >$. Die Relation $sp_1 \triangleright sp_2$* **gilt***, falls $\forall op \in OP : op_{sp_1} \triangleright^{op} op_{sp_2}$ gilt. op_{sp_1} und op_{sp_2} deuten die unterschiedliche Spezifikation der Operation op in sp_1 und sp_2 an.*

Die Relation $\triangleright$ wird wegen ihrer Nähe zur Vererbung in Eiffel auch als *Redefinitionsregel* bezeichnet.

Falls Invarianten vorhanden sind, wird jede Nachbedingung einer Prozedur um die Invarianten erweitert. An dieser Stelle unterscheidet sich die Definition von [Mey88]. Dort wird jede Vor- und Nachbedingung um die Invarianten erweitert. Auf die Erweiterung der Vorbedingungen kann verzichtet werden, wenn jede Prozedur einen Folgezustand erreicht (beschrieben durch die Nachbedingung), in dem die Invariante gilt. Es muß sichergestellt sein, daß jede Prozedur nur in einem Zustand aufgerufen werden kann, in dem die Invariante gilt. Wenn der initiale Zustand die Invariante erfüllt, ist diese Bedingung erfüllt, da nur durch Prozeduren Seiteneffekte auf den Zuständen auftreten. Daher soll für die folgenden Überlegungen angenommen werden, daß eine Objekterzeugungsprozedur *create* existiert, die

$$true \rightarrow [create()]\ inv$$

erfüllt, also die Invariante *inv* etabliert. Die Attributspezifikationen sind zustandsabhängig, d.h. durch nichtmodale Formeln beschrieben. Sie können also als Invariante behandelt werden.

Bemerkung 8.1.2 *Invarianten (und somit auch Attributspezifikationen) müssen in einer Relation $\triangleright$ verschärft werden. Gelte $sp_1 \triangleright sp_2$. Dann muß gelten:*

$$inv_{sp_2} \rightarrow inv_{sp_1}$$

Dies steht im Einklang mit der Anforderung, daß die Nachbedingung einer Prozedurimplementierung ψ_{sp_2} die Nachbedingung der Spezifikation ψ_{sp_1} implizieren muß.

Ein Implementierungsbegriff zwischen Spezifikationen kann auch, wie es häufig in algebraischen Spezifikationen [Wir90, Hen89] getan wird, über Modellklasseninklusion definiert werden. Die Implementierung wird über die Relation $\leadsto$ ausgedrückt.

Definition 8.1.4 *Seien sp_1, sp_2 zwei Spezifikationen.*

$$sp_1 \leadsto sp_2\ ,\ \ falls\ \ sig(sp_2) = sig(sp_1)\ und\ Mod(sp_2) \subseteq Mod(sp_1)$$

Modellklasseninklusion ist das ausschlaggebende Kriterium. Die Gleichheit der Signaturen wird vorausgesetzt, da hier nur *Eigenschaftsverfeinerung* betrachtet werden soll. *Signaturverfeinerung* kann etwa durch Signaturmorphismen erreicht werden.

8.2 Vergleich von $\triangleright$ und $\leadsto$

Wie in den vorangegangen Abschnitten beschrieben, erfolgt bei einer Verschärfung von Axiomen zwischen zwei Spezifikationen eine Verkleinerung der Modellklasse. In diesem Abschnitt soll die Abhängigkeit zwischen der Relation $\triangleright$ und der auf Modellklasseninklusion basierenden Relation $\leadsto$ präzisiert werden.

8.2.1 Vorbemerkungen

Bevor der Zusammenhang zwischen $\leadsto$ und $\triangleright$ untersucht wird, sollen einige Vereinfachungen vorgenommen und Grundlagen erläutert werden. Es wird im folgenden ein konsistentes und vollständiges Beweissystem für eine Prädikatenlogik erster Stufe vorausgesetzt (siehe Anhang A.2 und Kapitel 7).

Folgende Vereinfachungen sollen angenommen werden:

- Es gibt explizit keine Invarianten *inv*. Diese werden den Prozedurdefinitionen zugeordnet, d.h. sie werden einer Prozedurdefinition $\phi \rightarrow [P]\ \psi$ in der Form $\phi \rightarrow [P]\ \psi \wedge inv$ hinzugefügt. Daß *inv* nicht auch der Vorbedingung ϕ hinzugefügt werden muß, wurde in Abschnitt 8.1.2 erläutert.
- Es gibt nur Prozeduren. Attribute werden durch nichtmodale Formeln spezifiziert, da sie keine Zustandsveränderung bewirken. Attribute arbeiten in jedem Zustand gleich, daher können Attributdefinitionen als zustandsunabhängige Invarianten aufgefaßt und den Prozedurdefinitionen zugeordnet werden.
- Jede Prozedur wird durch nur eine Formel definiert (daß dies immer möglich ist, wird unten in Lemma 8.2.1 gezeigt).

Diese Vereinfachungen stellen somit keine Einschränkung der Mächtigkeit des Ansatzes dar. Mehrere modale Formeln können in speziellen Situationen bei Vorhandensein eines Beweissystems nach den folgenden Regeln zusammengefaßt werden:

$$\frac{\phi \rightarrow [P]\ \psi_1, \quad \phi \rightarrow [P]\ \psi_2}{\phi \rightarrow [P]\ \psi_1 \wedge \psi_2} \qquad \text{[AND]}$$

$$\frac{\phi_1 \rightarrow [P]\ \psi, \quad \phi_2 \rightarrow [P]\ \psi}{\phi_1 \vee \phi_2 \rightarrow [P]\ \psi} \qquad \text{[OR]}$$

$$\frac{(\phi \wedge \xi) \rightarrow [P]\ \psi, \quad (\phi \wedge \neg\xi) \rightarrow [P]\ \psi}{\phi \rightarrow [P]\ \psi} \qquad \text{[DEMORGAN]}$$

Siehe auch Kapitel 7 und Anhang A.2, dort sind noch weitere Regeln angegeben. Diese Regeln reichen aber nicht für beliebige Situationen.

Lemma 8.2.1 *Jede Menge von modalen Formeln $\{h_i \mid h_i \equiv \phi_i \rightarrow [P]\ \psi_i,\ i = 1, .., n\}$ kann in eine Formel $\phi \rightarrow [P]\ \psi$ umgeformt werden, die genau dann gilt, wenn die modalen Formeln h_i gelten.*

Beweis: Der Beweis wird induktiv geführt. Zuerst werden zwei Formeln h_1 und h_2 verknüpft. Seien $h_1 \equiv \phi_1 \rightarrow [P]\ \psi_1$ und $h_2 \equiv \phi_2 \rightarrow [P]\ \psi_2$ zwei modale Formeln. h_1 und h_2 sollen gelten. D.h. es gelten (nach Definition der Gültigkeit für die modale Formel in Definition 5.2.2):

Falls ϕ_1 gilt und P terminiert, dann gilt ψ_1 im Folgezustand
und
falls ϕ_2 gilt und P terminiert, dann gilt ψ_2 im Folgezustand.
D.h.
Falls ϕ_1 oder ϕ_2 gilt und P terminiert, dann gilt
falls ϕ_1 im Vorzustand galt, gilt ψ_1 im Folgezustand und
falls ϕ_2 im Vorzustand galt, gilt ψ_2 im Folgezustand.
Dies ist (wiederum nach Definition 5.2.2)
$\phi \rightarrow [P]\ \psi$ mit $\phi = \phi_1 \vee \phi_2$, $\psi = (\phi_1' \rightarrow \psi_1) \wedge (\phi_2' \rightarrow \psi_2)$ und $\phi_i' := \phi_i[x/old(x)]$,
wobei $\phi_i[x/old(x)]$ die Substitution aller freien Variablen x durch $old(x)$ in ϕ_i beschreibt.

Die ϕ_i legen die Vorbedingungen fest, unter denen das Programmfragment P korrekt arbeitet. Dies ist durch die Disjunktion realisiert. Mit der Nachbedingungskonstruktion ψ wird für jede geltende Vorbedingung die Etablierung der zugehörigen Nachbedingung sichergestellt. Dazu muß in den ϕ_i durch old auf die Belegungen des Vorzustands zurückgegriffen werden. Aus der Definition der Gültigkeit 5.2.2 für den Box-Operator folgt die Äquivalenz.

Aus einer Menge $\{h_i \mid i = 1, \ldots, n\}$ von modalen Formeln werden zwei zusammengefaßt; das Ergebnis kann mit der nächsten Formel zusammengefaßt werden. $\square$

Das Ergebnis aus Lemma 8.2.1 kann als Inferenzregel festgehalten werden:

$$\frac{\phi_1 \rightarrow [P]\ \psi_1, \quad \phi_2 \rightarrow [P]\ \psi_2}{\phi_1 \vee \phi_2 \rightarrow [P]\ (\phi_1[x/old(x)] \rightarrow \psi_1) \wedge (\phi_2[x/old(x)] \rightarrow \psi_2)} \quad \text{[MODAL MERGE]}$$

8.2.2 Vergleich der Relationen

Nach den Vorbemerkungen über Prädikatenlogik und Beweissysteme soll nun das Verhältnis der modellklassenbasierten Implementierungsrelation $\rightsquigarrow$ und der implikationsbasierten Relation $\triangleright$ untersucht werden. Es gelten die Einschränkungen vom Anfang des Abschnitts 8.2, d.h. Attribute werden als Invarianten behandelt und mit diesen den Prozedurspezifikationen zugeordnet. Jede Prozedur wird durch nur eine modale Formel spezifiziert. Dies ist keine Einschränkung hinsichtlich der Mächtigkeit, sondern führt lediglich zu einer kompakteren Notation (siehe Lemma 8.2.1).

Satz 8.2.1 *Sei $\Sigma = < S, Z, OP >$ eine Signatur und seien sp_1 und sp_2 Σ-Spezifikationen (unter den oben genannten Einschränkungen). Dann gilt:*

$$sp_1 \triangleright sp_2 \Rightarrow sp_1 \rightsquigarrow sp_2$$

Beweis: Modelle bestehen aus Trägermengen, Funktionen und der Abbildung $STATE$. Es kann gezeigt werden, daß die Modellklasseninklusion hier nur von den Funktionen, die Prozeduren interpretieren, abhängt. Damit kann die Aussage stellvertretend für eine Prozedur gezeigt werden.

Zuvor sollen noch einige Definitionen angegeben werden. Sei eine Prozedur p in sp_1 durch $\phi_1 \rightarrow [p(\ldots)]\ \psi_1$ und in sp_2 durch $\phi_2 \rightarrow [p(\ldots)]\ \psi_2$ spezifiziert. Mit $Mod_{sp}^{op}(p)$ seien die Modelle einer Prozedur p beschrieben, d.h. die Funktionen, die in Modellen einer Spezifikation sp die Prozedur p interpretieren können. Eine entsprechende Gültigkeitsrelation $\models_{op}$ wird ebenfalls definiert. Sei $O \in Mod(sp), p^* \in Mod_{sp}^{op}(p)$ und $p \in opns(sig(sp))$.

$p^* \models_{op} \phi$, *falls für alle Objekte $O \in Mod(sp)$, die p durch p^* interpretieren, gilt $O \models \phi$*

Also:

$$p^* \in Mod_{sp}^{op}(p) \ \ gdw. \ \ p^* \models_{op} \phi \rightarrow [p(\ldots)]\ \psi$$

Trägermengen, die Zustandsabbildung $STATE$ und Funktionen, die Attribute interpretieren, müssen bei der Modellbildung hier nicht betrachtet werden:

- Der Zustand $STATE$ ist unabhängig von konkreten Signaturen, da er als totale Abbildung auf dem Argumentbereich $IDENT$ definiert ist. $STATE$ ist in dieser Form also in allen Σ-Objekten aller Signaturen enthalten.

- Da die Modellsemantik sehr lose ist, können alle Trägermengen zu den Sorten aus S beliebige nichterreichbare Trägerelemente enthalten. Die einzige Anforderung, die an Trägermengen zu stellen ist, ist die Interpretierbarkeit der Formeln ϕ und ψ einer Prozedurspezifikation $\phi \rightarrow [p(\ldots)]\ \psi$, d.h. die Σ-Terme in den Formeln müssen durch Trägerelemente interpretierbar sein. Dies ist aber für die Spezifikationen sp_1 und sp_2 in gleicher Weise gegeben, da sie dieselbe Signatur haben, also dieselben Terme gebildet werden können.
- Eigenschaften der Attribute sind in den Invarianten enthalten und somit über die Spezifikationen der Prozeduren mit betrachtet (siehe Abschnitt 8.1.2 und insbesondere Bemerkung 8.1.2).

Statt für Spezifikationen und Σ-Objekte kann nun nach obigen Überlegungen anhand der Einschränkungen für Operationen argumentiert werden, da nun folgendes gilt:

$$Mod^{op}_{sp_1}(p) \subseteq Mod^{op}_{sp_2}(p) \Rightarrow Mod(sp_1) \subseteq Mod(sp_2)$$

Die Modellklasseninklusion für Prozeduren soll nun gezeigt werden. Es gilt nach Definition der Modellklassen für Prozeduren:

$$Mod^{op}_{sp_2}(p) \subseteq Mod^{op}_{sp_1}(p)$$

gdw.

$$p^* \in Mod^{op}_{sp_2}(p) \quad \Rightarrow \quad p^* \in Mod^{op}_{sp_1}(p)$$

gdw.

$$p^* \models_{op} \phi_2 \rightarrow [p(\ldots)]\ \psi_2 \quad \Rightarrow \quad p^* \models_{op} \phi_1 \rightarrow [p(\ldots)]\ \psi_1$$

Die Vorbedingung $sp_1 \rhd sp_2$ gilt, d.h. $\phi_1 \rightarrow \phi_2 \wedge \psi_2 \rightarrow \psi_1$ gilt für die Prozedur p. Es kann nun mit den Ergebnissen aus Kapitel 7 die Inferenzregel CONS eingesetzt werden.

$$\frac{\phi_1 \rightarrow \phi_2, \quad \phi_2 \rightarrow [p(\ldots)]\ \psi_2, \quad \psi_2 \rightarrow \psi_1}{\phi_1 \rightarrow [p(\ldots)]\ \psi_1} \qquad \text{[CONS]}$$

Mit $p^* \in Mod^{op}_{sp_2}(p)$, also $p^* \models_{op} \phi_2 \rightarrow [p(\ldots)]\ \psi_2$, folgt nach CONS-Regel und Voraussetzung die Aussage $p^* \models_{op} \phi_1 \rightarrow [p(\ldots)]\ \psi_1$. Damit ist Modellklasseninklusion $Mod^{op}_{sp_2}(p) \subseteq Mod^{op}_{sp_1}(p)$ gewährleistet. Dies gilt für alle Prozeduren $p \in OP$. Damit ist die Modellklasseninklusion für den Operationsanteil OP gezeigt (Attributspezifikationen wurden den Prozeduren als Invariante zugeordnet). Die weiteren Bestandteile von Σ-Objekten verhalten sich — wie oben gezeigt — neutral. Damit gilt die Modellklasseninklusion $Mod(sp_2) \subseteq Mod(sp_1)$, also $sp_1 \rightsquigarrow sp_2$, nach obigen Überlegungen. □

Daß die Umkehrung der Implikation aus dem Satz 8.2.1 nicht gilt, kann anhand eines Beispiels verdeutlicht werden. Sei eine Prozedur p in sp_1 durch

$$x < 3 \wedge x > 5 \rightarrow [p(\ldots)]\ z = 1$$

und in sp_2 durch

$$x < 4 \wedge x > 5 \rightarrow [p(\ldots)]\ z = 0$$

spezifiziert. Jede beliebige Funktion (mit korrekten Argument- und Wertebereichen) zu einer Prozedur erfüllt die Spezifikation in sp_1 und sp_2 im Sinne des Modellklassenbegriffs, da die

Vorbedingungen nie eintreten können und somit nach Definition der Gültigkeit keine weiteren Anforderungen an das Modell der Prozedur gestellt werden. Sei p^* ein solches Modell. Dann gilt:

$$p^* \in Mod^{op}_{sp_2}(p) \Rightarrow p^* \in Mod^{op}_{sp_1}(p)$$

Somit gilt $sp_1 \rightsquigarrow sp_2$. $sp_1 \rhd sp_2$ gilt hingegen nicht, da aus $z = 0$ nicht $z = 1$ folgt (dies wäre notwendige Bedingung für die $\rhd^{op}$-Relation zusammen mit der Terminierungsforderung). Also läßt sich folgende Beobachtung notieren.

Beobachtung 8.2.1 *Sei $\Sigma = < S, Z, OP >$ eine Signatur und seien sp_1 und sp_2 Σ-Spezifikationen (unter den genannten Einschränkungen).*

$$sp_1 \rightsquigarrow sp_2 \Rightarrow sp_1 \rhd sp_2 \textit{ gilt } \textbf{nicht} \textit{ für beliebige } sp_1, sp_2$$

Bemerkung 8.2.1 *Solche Beispiele lassen sich auch konstruieren, wenn auf widersprüchliche Aussagen verzichtet wird. So entsteht die gleiche Situation, wenn in Vorbedingungen nichtzugängliche Zustände (siehe Definition 5.2.7) spezifiziert sind. Auch dann läßt sich nie ein Zustand erreichen, in dem eine bestimmte Vorbedingung gilt.*

Die bisher gemachten Aussagen lassen sich wie folgt zusammenfassen:

$$sp_1 \rightsquigarrow sp_2 \quad \Rightarrow \quad sp_1 \rhd sp_2$$

kann *nicht* etabliert werden, falls

- die Modelle Funktionen enthalten, die unter Umständen nicht terminieren (siehe Abschnitt 8.1.1),
- die Formeln, die Zustände beschreiben, nicht erfüllbar sind (siehe obiges Gegenbeispiel),
- Vorbedingungen nicht zugängliche Zustände beschreiben (siehe Bemerkung 8.2.1).

Damit wird deutlich, daß $\rhd$ bis auf Sonderfälle den Eigenschaften von $\rightsquigarrow$ nahekommt. Da aus methodischer Sicht die Implikation $sp_1 \rightsquigarrow sp_2 \quad \Rightarrow \quad sp_1 \rhd sp_2$ nicht von Bedeutung ist, soll es hier bei der Angabe von Heuristiken bleiben. Wichtig ist, daß bei Benutzung von $\rhd$ die Modellklasseninklusion gesichert ist. Die Modellklasseninklusion ist für Spezifizierer kaum nachweisbar, daher wird sie auch kaum Ausgangspunkt der Betrachtung sein.

8.2.3 Implementierung und beobachtungsorientierte Spezifikation

In diesem Abschnitt soll die Idee der beobachtungsorientierten Spezifikation wie sie im Abschnitt 5.4.3 beschrieben wurde, nochmals aufgegriffen und im Kontext der Implementierungsrelation betrachtet werden. Die Implementierungsrelation beobachtungsorientierten Spezifizierens ist ebenfalls auf Modellklasseninklusion definiert [Hen91c]. Der Modellbegriff umfaßt dort verhaltens-, d.h. beobachtungsäquivalente Modelle, so daß Implementierungen auch nur in ihrem beobachtbaren Verhalten mit der Spezifikation übereinstimmen. Durch die hier vorliegende Definition von Modellbildung und Implementierungsrelation wird der gleiche Effekt erzielt. Die Repräsentation des Zustands wird durch $STATE$ und die Trägermengen modelliert. Da diese transparent bei der Modellklasseninklusion sind (gezeigt im Beweis zu Satz

8.2.1), ist das Kriterium der Implementierungsrelation das Verhalten der auf dem Zustand arbeitenden Operationen.

Auch wenn strukturelle Aspekte der Realisierung des Zustands hier transparent bleiben, so werden doch die Eigenschaften des Zustands, sofern sie durch Invarianten spezifiziert werden, berücksichtigt. Dies geschieht, da Invarianten, also Eigenschaften des Zustands und der Attribute, den Prozedurspezifikationen zugeordnet werden. In erster Linie wird durch die Vererbungsrelation hier wie auch bei R. Hennicker eine *Verhaltensverfeinerung* beschrieben. *Strukturverfeinerung* kann hier ebenfalls ausgedrückt werden, indem strukturelle Eigenschaften der Zustände axiomatisch spezifiziert werden.

8.3 Vertikale Entwicklung

In diesem Abschnitt wird vertikale Entwicklung (Implementierung im engeren Sinne) durch die Implementierungsrelation definiert. Im folgenden Abschnitt 8.4 wird der Begriff der vertikalen Entwicklung zu einem Verfeinerungsbegriff erweitert, um inkrementelle Entwicklung vorstellen zu können. Wie schon angesprochen, geht es in der *vertikalen Entwicklung* um das Senken des Abstraktionsniveaus. Dies wurde in den vorangegangenen Kapiteln mit Implementierung im engeren Sinne bezeichnet. Als Relation, die die Implementierung im weiteren Sinne beschreibt, ist in Definition 8.1.4 die Implementierungsrelation $\rightsquigarrow$ eingeführt worden. Da es sich bei Implementierung i.e.S. um einen Spezialfall handelt, soll $\rightsquigarrow$ als Operationalisierung für die vertikale Entwicklung herangezogen werden. Eine Modifikation von $\rightsquigarrow$ für diesen Zweck ist allerdings nicht sinnvoll. Um das Senken des Abstraktionsniveaus präziser erfassen zu können, könnten syntaktische Konstruktionen eingeführt werden, die etwa die Abbildung einer abstrakten Sorte auf eine konkretere Repräsentation beschreiben. Da aber die Identität als Abbildung an dieser Stelle nicht ausgeschlossen werden sollte, soll auf syntaktische Restriktionen verzichtet werden. Die Relation $\rightsquigarrow$ basiert auf Modellklassen und abstrahiert somit von syntaktischen Bezeichnungen. Daher modelliert sie die Anforderungen der vertikalen Entwicklung am besten.

Die Transitivität der Relation(en) der vertikalen Entwicklung wird als *vertikale Kompositionseigenschaft* bezeichnet. Sie soll nun gezeigt werden.

Satz 8.3.1 *(Vertikale Komposition für $\rightsquigarrow$) Die Relation $\rightsquigarrow$ ist transitiv.*

Beweis: Für zwei Spezifikationen sp_1, sp_2 gilt $sp_1 \rightsquigarrow sp_2$, falls $sig(sp_1) = sig(sp_2) \wedge Mod(sp_2) \subseteq Mod(sp_1)$. Damit folgt die Transitivität von $\rightsquigarrow$ direkt, da $=$ und $\subseteq$ transitiv sind. $\square$

Der Ansatz, Komponenten über die Relation $\triangleright$ zu entwickeln, ist gegenüber $\rightsquigarrow$ konstruktiver. Er ist kompositional, da er von der Implementierung der Subkonstrukte (Prozeduren) abhängt und diese durch ein Beweissystem bearbeitbar sind. Das Beweissystem, insbesondere die Regel CONS, gibt Anleitung zum korrektheitserhaltenden Weiterentwickeln von Komponenten (vgl. Kapitel 7). Es soll für $\triangleright^{op}$ und $\triangleright$ in diesem Abschnitt noch gezeigt werden, daß diese Relationen ebenfalls transitiv sind, also auch die *vertikale Kompositionseigenschaft* erfüllen.

Satz 8.3.2 *(Vertikale Komposition für $\triangleright^{op}, \triangleright$) Die Relationen $\triangleright^{op}$ und $\triangleright$ sind transitiv.*

Beweis: Sei $\Sigma = < S, Z, OP >$ die Signatur der Spezifikationen.

1. *Operationen* ($\rhd^{op}$): Es ist für $op_1, op_2, op_3 \in OP$ zu zeigen:

$$op_1 \rhd^{op} op_2 \ \wedge \ op_2 \rhd^{op} op_3 \ \Rightarrow \ op_1 \rhd^{op} op_3$$

mit den Spezifikationen $\phi_1 \rightarrow [op_1(\ldots)]\ \psi_1$, $\phi_2 \rightarrow [op_2(\ldots)]\ \psi_2$ und $\phi_3 \rightarrow [op_3(\ldots)]\ \psi_3$. Gelten die Prämissen $op_1 \rhd^{op} op_2$ (d.h. $\phi_1 \rightarrow \phi_2$ und $\psi_2 \rightarrow \psi_1$) und $op_2 \rhd^{op} op_3$ (d.h. $\phi_2 \rightarrow \phi_3$ und $\psi_3 \rightarrow \psi_2$). Zu zeigen ist die Konklusion $op_1 \rhd^{op} op_3$ (d.h. $\phi_1 \rightarrow \phi_3$ und $\psi_3 \rightarrow \psi_1$). Dies gilt wegen der Transitivität der Implikation $\rightarrow$.

2. *Spezifikationen* ($\rhd$): Es gelte für die Spezifikationen sp_1, sp_2, sp_3:

$$sig(sp_1) = sig(sp_2) = sig(sp_3)\ und\ sp_1 \rhd sp_2\ und\ sp_2 \rhd sp_3$$

Da $sp_1 \rhd sp_3$ genau dann gilt, wenn $\forall op \in OP_\Sigma.\ op_{sp_1} \rhd^{op} op_{sp_3}$ gilt, folgt die Transitivität aus Teil 1 des Beweises.

□

8.4 Inkrementelle Entwicklung

Inkrementelle Entwicklung von Spezifikationen soll nun vorgestellt werden. Obwohl das Konzept in weiterem Rahmen einsetzbar ist, als nur für vertikale Entwicklung und Implementierung, soll es doch hier vorgestellt werden. Vertikale Entwicklung im weiteren Sinne wie auch inkrementelle Entwicklung beinhaltet das Treffen von Entwurfsentscheidungen, die bisher noch offen waren. Es soll an dieser Stelle keine Modularisierung, d.h. Zerlegung in kleinere Spezifikationen, betrachtet werden.

Zur Bearbeitung einer Spezifikation lassen sich dann die Aktivitäten Dekomposition, Implementierung und Verfeinerung unterscheiden. *Dekomposition* bezeichnet das Zerlegen von Operationen. Dies kann *daten-* und *funktionsorientiert* erfolgen. Bei der datenorientierten Zerlegung ergibt sich die Zerlegung der Operationen aus der Zerlegung der Parameterdaten oder anderer nichtlokaler Daten einer Operation. Die Schnittstellen ändern sich. Die funktionale Zerlegung bedeutet eine Zerlegung in zwei oder mehr weniger umfangreiche Operationen, die die Aufgabe der ursprünglichen Operation durch Hintereinanderschaltung oder nebenläufige Ausführung erfüllen. Die Schnittstelle der Ausgangsoperation bleibt erhalten. Funktionsorientierte Dekomposition kann durch Spezifikation auf dem Gleichheitsprädikat $\equiv_{state}$ unterstützt werden, etwa durch $p(a,b) \equiv_{state} seq(p_1(a,b), p_2(a,b))$ als Zerlegung der Operation p in zwei hintereinandergeschaltete Operationen p_1 und p_2. Datenorientierte Dekomposition soll an dieser Stelle nicht betrachtet werden; hierzu sei auf die Datentypbetrachtungen in Abschnitt 4.3 verwiesen. In diesem Kapitel stehen Verhaltenseigenschaften und Verhaltensverfeinerung im Vordergrund.

Implementierung (i.e.S.) bezeichnet die Realisierung gegebener Eigenschaften auf einem semantisch niedrigeren Niveau, um weniger abstrakt zu werden oder sogar Ausführbarkeit zu erreichen. Die Eigenschaften dürfen in bezug auf die Schnittstelle auf keinen Fall schlechter werden, wobei wir unter der intuitiven Bezeichnung 'schlechter' das Verstoßen gegen Spezifikationen meinen. 'Bessere' Implementierungen sind Verfeinerungen, die die Semantik der

Spezifikation erhalten und zusätzliche Eigenschaften fordern. Mit dem Begriff der *Verfeinerung* ist also das Präzisieren von Spezifikationen gemeint. Hierzu gehört auch das Hinzunehmen neuer Funktionalitäten, realisiert durch neue Operationen. Die Präzisierung schon bestehender Operationsspezifikationen kann auf zwei Arten erfolgen:

1. *Mögliche Aufrufsituationen erweitern:* Es entfallen dann Vorbedingungen oder diese werden abgeschwächt, d.h. die Spezifikation wird weniger partiell.

2. *Leistungen verbessern:* Nachbedingungen, die zusätzliche Leistungen beschreiben, kommen hinzu oder bestehende werden verstärkt.

Die zwei Präzisierungsarten können anhand einer Spezifikationsschablone für eine Prozedur erläutert werden:

$$\phi_1 \wedge \ldots \wedge \phi_n \rightarrow [p(\ldots)]\ \psi_1 \wedge \ldots \wedge \psi_m$$

Die ϕ_i beschreiben die notwendigen Aufrufbedingungen (die alle erfüllt sein müssen) und die ψ_j die Effekte, die durch Anwendung von p eintreten (beispielsweise pro Parameter oder globaler Variable ein ψ_j, das den Effekt auf diese Variable beschreibt). '$\mapsto$' beschreibt im folgenden die Ersetzung der linken Seite von $\phi \mapsto \psi$ durch die rechte. Diese Notation bezieht sich auf eine Weiterentwicklung einer Spezifikation für eine Prozedur p von $\phi \rightarrow [p(\ldots)]\ \psi$ zu $\phi' \rightarrow [p(\ldots)]\ \psi'$. Es wird hierfür $\phi \mapsto \phi'$ und $\psi \mapsto \psi'$ geschrieben. Im folgenden werden Präzisierungsschemata (a) – (f) zur Verfeinerung einer Prozedur p angegeben. Die Bedingungen an eine Verfeinerung von p sind so formuliert, daß sie die Relation $\rhd^{op}$ erfüllen. Es muß dazu $\phi \rightarrow \phi'$ und $\psi' \rightarrow \psi$ gelten. Diese beiden Implikationen sollen von den Transformationsschemata $. \mapsto .$ garantiert werden.

1. Behandlung von Vorbedingungen:

   (a) $\phi_1 \wedge \ldots \wedge \phi_n \quad \mapsto \quad \phi_1 \wedge \ldots \wedge \phi_{n-1}$ ist als Implikation eine Tautologie.

   (b) $\phi_1 \wedge \ldots \wedge \phi_{i-1} \wedge \phi_i \wedge \phi_{i+1} \wedge \ldots \wedge \phi_n \quad \mapsto \quad \phi_1 \wedge \ldots \wedge \phi_i' \wedge \ldots \wedge \phi_n \quad$ mit $\phi_i \rightarrow \phi_i'$.

2. Behandlung von Nachbedingungen:

   (c) $\psi_1 \wedge \ldots \wedge \psi_m \quad \mapsto \quad \psi_1 \wedge \ldots \wedge \psi_m \wedge \psi_{m+1}$ ist als Implikation von rechts nach links gelesen eine Tautologie.

   (d) $\psi_1 \wedge \ldots \wedge \psi_{j-1} \wedge \psi_j \wedge \psi_{j+1} \ldots \wedge \psi_m \quad \mapsto \quad \psi_1 \wedge \ldots \wedge \psi_j' \wedge \ldots \wedge \psi_m \quad$ mit $\psi_j' \rightarrow \psi_j$.

Disjunktionen können ebenfalls benutzt werden. Insbesondere zur Formulierung von Vorbedingungen kann es sinnvoll sein, mehrere Bedingungen anzugeben, von denen mindestens eine erfüllt sein muß:

$$\phi_1 \vee \ldots \vee \phi_n \rightarrow [p(\ldots)]\ \psi$$

Die ϕ_i und ψ können dabei beliebige Formeln (etwa Konjunktionen) sein. Es gilt für die Vorbedingungsbehandlung:

(e) $\phi_1 \vee \ldots \vee \phi_n \quad \mapsto \quad \phi_1 \vee \ldots \vee \phi_n \vee \phi_{n+1}$ ist als Implikation eine Tautologie.

(f) $\phi_1 \vee \ldots \vee \phi_i \vee \ldots \vee \phi_n \quad \mapsto \quad \phi_1 \vee \ldots \vee \phi_i' \vee \ldots \vee \phi_n \quad$ mit $\phi_i \rightarrow \phi_i'$.

Damit sind Grundlagen zur Verfügung gestellt, die sinnvoll sind, um konstruktiv die oben aufgelisteten Arten der Präzisierung einer Operationsspezifikation durchführen zu können. Bei Anwendung einer der durch $\phi \mapsto \phi'$ beschriebenen Transformationen erhält sich die Relation $\rhd^{op}$ zwischen Ausgangsspezifikation und der Verfeinerung.

Folgerung 8.4.1 *Operationsspezifikationen für eine Prozedur p, die nach den obigen Schemata* **(a)** *bis* **(f)** *verfeinert werden, erfüllen die Relation* $\phi \rightarrow [p(\ldots)]\ \psi \rhd^{op} \phi' \rightarrow [p(\ldots)]\ \psi'$.

Beweis: Trivial, da die Transformationsschemata als Implikationen gelesen wie oben beschrieben die $\rhd$-Bedingungen $\phi \rightarrow \phi'$ und $\psi' \rightarrow \psi$ erfüllen. □

Die Schemata sollen am Beispiel der Stack-Operation *push* verdeutlicht werden. *push* in der Form

$$not(full) \rightarrow [push(e)]\ top() = e$$

soll weniger partiell werden. Nach Schema (a) kann $[push(e)]\ top() = e$ erzeugt werden, mit Schema (b) ist $true \rightarrow [push(e)]\ top() = e$ möglich. (d) erlaubt es, $[push(e)]\ top() = e \wedge (full \rightarrow top() = old(top()))$ zu erzeugen. Eine Sonderfallbehandlung für volle Stacks wird mit letzterem hinzugenommen.

Die definierten Schemata sollen dazu dienen, den konstruktiven Korrektheitsbegriff, wie er durch die Relation $\rhd$ eingeführt wurde, gegenüber $\rhd$ noch handhabbarer für den Spezifizierer zu machen. Diese Schemata liefern eine konkrete Vorgabe zur Entwicklung entlang der Relation $\rhd$. Die Verfeinerung soll hier allerdings nicht durch ein explizites syntaktisches Konstrukt unterstützt werden. Die notwendigen logischen Abschätzungen muß der Spezifizierer leisten. Die Forderung der Relation $\phi_1 \rightarrow [P]\ \psi_1 \;\rhd\; \phi_1 \rightarrow [P]\ \psi_2$ bzgl. der Vorbedingungen lautet $\phi_1 \rightarrow \phi_2$. Dies bedeutet, daß ϕ_1 stärker (im Sinne von stärker einschränkend) ist. Sinnvoll ist hier die Anwendung der syntaktischen Ableitbarkeit $\{\phi_1\} \vdash \phi_2$ mit Hilfe eines geeigneten Beweissystems (siehe Kapitel 7). Nur dann ist eine weitgehende konstruktive Unterstützung für den Entwickler möglich. In einem vollständigen und konsistenten Beweissystem gilt $E \vdash \phi$ genau dann, wenn $Mod(\Sigma, E) \models \phi$ (mit einer Signatur Σ, einer Formelmenge $E \subseteq WFF(\Sigma)$ und einer Formel $\phi \in WFF(\Sigma)$).

Lemma 8.4.1 *Aus* $\{\phi_1\} \vdash \phi_2$ *folgt* $\phi_1 \rightarrow \phi_2$.

Beweis: Für die Ableitung $\{\phi_1\} \vdash \phi_2$ gilt $\{\phi_1\} \vdash \phi_2$ genau dann, wenn $Mod(<\Sigma, \{\phi_1\}>) \models \phi_2$. Daraus folgt $\forall O \in Mod(<\Sigma, \{\phi_1\}>).\ O \models \phi_1 \;\Rightarrow\; \forall O \in Mod(<\Sigma, \{\phi_1\}>).\ O \models \phi_2$. Das wiederum ist nach Definition von $\phi_1 \rightarrow \phi_2$ durch 'falls ϕ_1 gilt, muß ϕ_2 gelten': $\forall O \in Mod(<\Sigma, \{\phi_1\}>).\ O \models \phi_1 \rightarrow \phi_2$. □

Mit dem Lemma ist gezeigt, daß, um eine geeignete Implikation für die Relation $\rhd$ zu finden, Inferenzregeln einer Prädikatenlogik benutzt werden können. Einige Regeln (etwa Konjunktions- oder Disjunktionsregel) sind im Anhang zu finden. Es können auch modale Formeln ϕ_1, ϕ_2 betrachtet werden. Dazu ist das in Abschnitt 7.5 vorgestellte Beweissystem notwendig. Es sollte einem Spezifizierer auf jeden Fall die Möglichkeit gegeben werden, syntaktisch zu argumentieren (d.h. über die Ableitung $\vdash$, nicht über die Implikation $\rightarrow$). Die Unterstützung des Spezifizierers auf Basis der Syntax ist eine wesentliche Intention der Relation $\rhd^{op}$.

Es soll jetzt auf die Hinzunahme neuer Funktionalitäten eingegangen werden. Falls eine Operation p neu hinzugenommen wird, so kann sie in der Ausgangsspezifikation als durch $false \rightarrow [p(\ldots)]\ true$ spezifiziert angenommen werden.

Folgerung 8.4.2 *Jede beliebig spezifizierte Prozedur $\phi \to [p(\ldots)]\ \psi$ erfüllt die Relation $\rhd^{op}$ bezüglich $false \to [p(\ldots)]\ true$.*

Beweis: $false \to \phi$ und $\psi \to true$ sind Tautologien. □

Bedingung einer Verfeinerung ist, daß die Operationen verbessert werden, d.h. eine schwächere Vorbedingung und eine stärkere Nachbedingung realisiert wird. Eine Verschlechterung in den Leistungen sollte nicht vorkommen. Falls dieses im Rahmen der Entwicklung notwendig ist, sollte ein Rückschritt oder Zyklus im Modell der Entwicklungsmethodik durchlaufen werden. Dies soll hier aber nicht betrachtet werden. *Evolution* ist das Schlagwort, das im allgemeinen die Weiterentwicklung von Komponenten bzw. Spezifikationen beschreibt, das auch Korrektheitsverstöße umfaßt.

8.5 Zusammenfassung

Um aus Spezifikationen korrekte Programme entwickeln zu können, wird ein Implementierungsbegriff benötigt, der in einer Variante so abgewandelt wurde, daß Implementierungsbeziehungen formal sichergestellt werden können, ohne vollständig auf semantischer Ebene argumentieren zu müssen. Die Implementierungsrelation $\rightsquigarrow$ auf Basis der Modellklasseninklusion ist ein Ansatz zur Modellierung der vertikalen Entwicklung, mit dem deren Anforderungen präzise modelliert werden konnten. Implementierungen im Rahmen dynamischer Logik werden auch von Wand in [Wan82] untersucht. Einige Ansätze (etwa [BMPW86, Par90]) realisieren Implementierungen im weiteren Sinne durch eine fest vorgegebene Folge von Anwendungen verschiedener Spezifikationsoperatoren. Diese sind etwa unter der Bezeichnung *Synthesize-Restrict-Identify*-Implementierung bekannt (siehe auch [Wir90, Meh95]). Mit einem solchen Ansatz werden zwar die einzelnen Schritte deutlicher, die bei der vertikalen Entwicklung durchzuführen sind (Umbenennen, Ausblenden nicht benötigter Funktionalitäten, Bilden von Äquivalenzen, ...); die Implementierungsrelation aber ist ein einfaches und mächtiges Konstrukt. Weitere Unterstützung könnte hier durch eine geeignete Methodik erfolgen. Die Implementierungsrelation wurde hier als Mittel zur *Eigenschaftsverfeinerung* definiert, d.h. nur Eigenschaften können präzisiert werden, die Schnittstelle bleibt aber gleich. Eine *Schnittstellenverfeinerung* kann durch Signaturmorphismen erreicht werden.

Die Korrektheit einer Implementierung $\rightsquigarrow$ läßt sich nur a *posteriori* durch Verifikation der partiellen Korrektheit garantieren. Daher wurde der konstruktive Vererbungsbegriff $\rhd$ hinzugenommen, der als Spezialfall der Relation $\rightsquigarrow$ nachgewiesen werden konnte. Die Transitivität von $\rhd$ und $\rightsquigarrow$ wurde nachgewiesen. $\rhd$ macht totale Korrektheitsaussagen, während $\rightsquigarrow$ partielle Korrektheit betrachtet. Dies ist einer der Gründe, warum keine Gleichwertigkeit zwischen beiden Begriffen besteht. Der andere (wesentlichere) ist die Grundannahme in der Logik, daß bei Nichterfüllung der Vorbedingung ϕ einer Implikation (etwa von $\phi \to [P]\ \psi$) die Implikation unabhängig von der Gültigkeit der Nachbedingung gilt. Dies führt zu einer größeren Mächtigkeit von $\rightsquigarrow$ durch die Modellklassenbildung. Auf Basis des Beweissystems konnten Entwicklungsschemata für Operationen angegeben werden, die ein konstruktives, korrektheitserhaltendes Weiterentwickeln erlauben. Der Spezifizierer hat dann nur noch elementare prädikatenlogische Abschätzungen zu liefern. Dieses wurde unter dem Stichwort *inkrementelle Enwicklung* vorgestellt.

Somit konnten Techniken der Verifikation in eine Methodik zur vertikalen Entwicklung integriert und so die Kritik am Verfahren der Verifikation deutlich abgeschwächt werden.

Kapitel 9

Horizontale Entwicklung

Die horizontale Entwicklung ist mit der Verknüpfung von Komponenten, d.h. der Komposition von Komponenten durch Modularisierung und Spezifikationsoperatoren, befaßt. Diese bearbeiten Spezifikationen auf *horizontaler Ebene*. Die Implementierung ist hingegen ein Konstrukt der *vertikalen Entwicklung*. Sie wird in der Regel nicht durch im Sinne eines Korrektheitsbegriffes korrekte Operatoren unterstützt. Horizontale Entwicklung soll in dieser Hinsicht stärker durch Konstrukte unterstützt werden.

Im vorangegangenen Kapitel wurden unter dem Begriff *Verfeinerung* Techniken vorgestellt, die zur Weiterentwicklung einer einzelnen Komponente dienen. Die Zerlegung von Komponenten in kleinere Einheiten wurde dabei ausgeschlossen. Diese Zerlegung (bzw. die Zusammensetzbarkeit als deren Umkehrung) von Komponenten ist Kernaktion der *Modularisierung*. Modularisierung erlaubt die separate, d.h. parallele und unabhängige Realisierung von Komponenten und dient so auch der Verständlichkeit. Ein Parametrisierungskonzept über einem formalen Import, der Anforderungen an aktuelle Parameter, d.h. zu importierende Konstrukte, zu beschreiben gestattet, ist die Grundlage eines Komponenten- oder Modulkonzeptes. Der Import bildet somit einen eigenständigen Spezifikationsteil. Dessen Beziehung zum Rumpf der Spezifikation soll durch einen geeigneten Korrektheitsbegriff beschrieben werden. Um das Geheimnisprinzip zu gewährleisten, wird durch einen Exportmechanismus eine Schnittstellenbeschreibung über einem Spezifikationsrumpf ermöglicht.

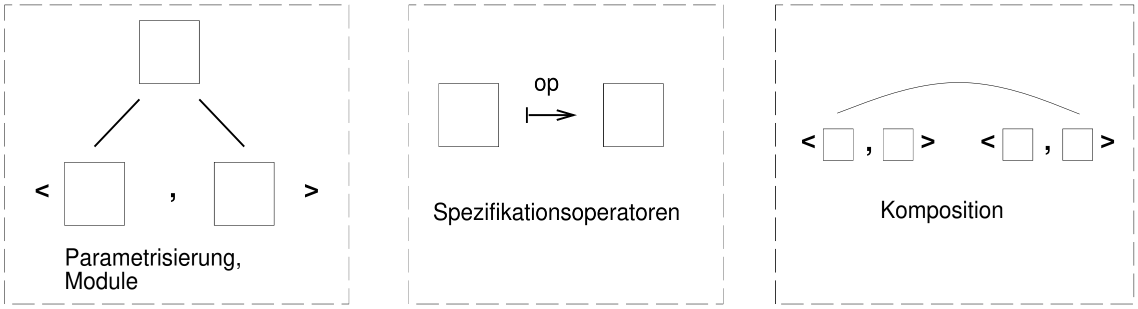


Abbildung 9.1: Horizontale Entwicklung

Die Zusammensetzung einzelner Komponenten zu einer umfangreicheren Komponente wird durch einen Kompositionsmechanismus unterstützt, der den formalen Import einer parametrisierten Komponente durch den Export einer anderen aktualisiert. Die Parametrisierbarkeit von Modulen ermöglicht deren *Wiederverwendung* in verschiedenen Kontexten. Die Anpassung, d.h. Integration einer wiederverwendbaren Komponente in einen Applikationskontext, wird durch geeignete Operatoren unterstützt.

Die Auswahl der Operatoren und Kompositionskonstrukte (Abbildung 9.1) orientiert sich an den Arbeiten von R. Hennicker [Hen91a, Hen92, HN92], die formal auf der Kategorientheorie nach [EM85, EM90] basieren. Hier liegt der Spezifikations- und Modellbildungsansatz der vorangehenden Kapitel zugrunde. Die Theorie der darauf aufsetzenden Spezifikationsoperatoren stammt im wesentlichen aus [Wir90]. Auf diesen Grundlagen wird der Ansatz über ein Kompositionskonstrukt zu einem komponentenorientierten Modulkonzept erweitert.

Im Rahmen einer Entwurfsmethodik, die Modularisierung und Wiederverwendung unterstützt, sei angenommen, daß ein zu entwickelndes System mit Hilfe der Dekomposition in einzelne Bausteine zerlegt ist. Jeder einzelne dieser Bausteine kann entweder von Hand realisiert oder durch Wiederverwendung eines geeigneten Bausteins implementiert werden. Falls auf der Ebene der Spezifikationen ein korrekt modularisiertes System vorliegt, soll die Korrektheit auch auf der Ebene der Implementierungen der jeweiligen Komponente erhalten bleiben. Dies wird die *horizontale Kompositionseigenschaft* genannt. Sie soll kurz veranschaulicht werden. Seien M_1 und M_2 zwei Komponenten. $M_1(M_2)$ soll ausdrücken, daß M_2 als aktueller Parameter an M_1 übergeben wird. Die Komposition $M_1(M_2)$ ist korrekt, wenn M_2 eine den Anforderungen eines formalen Imports von M_1 genügende Komponente ist. Die horizontale Kompositionseigenschaft fordert, daß, falls $M_1 \rightsquigarrow I_1$ und $M_2 \rightsquigarrow I_2$ Implementierungen sind, auch $I_1(I_2)$ eine korrekte Komposition ergibt.

In Abschnitt 9.1 werden Grundlagen beschrieben und Spezifikationsoperatoren für einfache Spezifikationen definiert. Parametrisierte Spezifikationen werden in Abschnitt 9.2 eingeführt. Zusammen mit einer Exportschnittstelle wird in Abschnitt 9.3 daraus ein Modulkonstrukt entwickelt. Ähnliche Untersuchungen sind für den rein algebraischen Bereich in [Wir90, EM85, EM90] zu finden. Im Kontext objektorientierter Relationen gibt es Untersuchungen etwa von [PPP94, AAZ93, CL94]. Relationen zwischen Spezifikationen im Kontext der objektorientierten Entwicklung werden wir in Kapitel 10 betrachten.

9.1 Einfache Spezifikationsausdrücke

Es soll eine Sprache der *Spezifikationsausdrücke* entwickelt werden, in der Spezifikationen durch Anwendung von Spezifikationsoperatoren auf primitivere Spezifikationen beschrieben werden können. Die Sprache wird durch eine *Modellsemantik* (siehe [Wir90]) definiert. Um von der konkreten syntaktischen Beschreibung in einer Spezifikation (Signatur und Axiome) abstrahieren zu können, wird eine Darstellung für Spezifikationen eingeführt, die die Signatur und die Modellklasse der Spezifikation verknüpft. Einige geeignete Operatoren werden definiert. Von der Definition der Operatoren wird die Einhaltung von Eigenschaften auf der Modellsemantik gefordert (Striktheit, Monotonie). Spezifikationsoperatoren, die strikte und monotone Funktionen auf der Modellsemantik definieren, werden Spezifikationsfunktionen genannt. Einige Spezifikationsoperatoren (*rename*, *export*, *restrict*, *enrich*) werden definiert und als Spezifikationsfunktionen nachgewiesen. Mit Hilfe dieser Eigenschaften kann gezeigt werden, daß die Operatoren die horizontale Kompositionseigenschaft besitzen.

9.1.1 Grundlagen für Spezifikationsausdrücke

Zuerst sollen Spezifikationsausdrücke induktiv definiert werden.

Definition 9.1.1 *Sei eine Menge Spec_Op von Spezifikationsoperatoren gegeben. Ein* **Spezifikationsausdruck** *ist*

- *jede Spezifikation $SP = <\Sigma, E>$ nach Kapitel 5 Definition 5.2.1,*
- *falls SP ein Spezifikationsausdruck und OP ein Spezifikationsoperator ist, dann ist auch die Anwendung des Spezifikationsoperators auf SP ein Spezifikationsausdruck.*

$Spec\_Expr$ bezeichne die Gesamtheit aller Spezifikationsausdrücke.

Diese Sprache wird semantisch definiert, indem jedem Spezifikationsausdruck die Klasse seiner Modelle und jedem Spezifikationsoperator eine Funktion auf Modellklassen zugeordnet wird. Diese Form wird *Modellsemantik* genannt [Wir90]. Ein anderer Ansatz ist die Definition über eine *Präsentationsemantik*. Hier wird jedem Spezifikationsausdruck eine Normalform, die sogenannte Präsentation zugewiesen (siehe [Wir86]). Nach [Wir90] eignet sich die Präsentationssemantik besser zur formalen Manipulation von Spezifikationsausdrücken, während die Modellsemantik die abstrakte, mathematische Semantik einer Spezifikation bestimmt. Die Modellsemantik kann direkt auf der in Kapitel 5 vorgestellten Modellbildung aufsetzen, sie wird daher ausgewählt. Präsentationssemantik wird somit nicht weiter betrachtet.

$Sign$ ist die Klasse aller Signaturen. $\perp_{Sign}$ ist die leere Signatur. Die Klasse $Spec$ umfaßt alle möglichen Modellsemantiken von Spezifikationen über $Sign$:

$$Spec := \{ <\Sigma, C> \mid \Sigma \in Sign,\ C \subseteq Obj(\Sigma)\}$$

$Spec_\perp$ ist die Erweiterung von $Spec$ um das bottom-Element $\perp_{Spec} = <\perp_{Sign}, Obj(\perp_{Sign})>$:

$$Spec_\perp := Spec \cup \{\perp_{Spec}\}$$

$sp_{|\Sigma}$ beschreibt die Restriktion der Spezifikation $sp = <\Sigma', E>$ auf die Σ-relevanten Anteile. Es muß $\Sigma \subseteq \Sigma'$ gelten.

$$sp_{|\Sigma} := <\Sigma, \{\phi \in E \mid \phi \in WFF(\Sigma)\}>$$

Auf $Spec_\perp$ werden zwei Projektionsfunktionen sig und Mod angeboten:

$$sig : Spec_\perp \rightarrow Sign \ \ mit \ \ sig(<\Sigma, C>) := \Sigma$$

und

$$Mod : Spec_\perp \rightarrow \{C \subseteq Obj(\Sigma) \mid \Sigma \in Sign\} \ \ mit \ \ Mod(<\Sigma, C>) := C$$

Mod ist somit auf Spezifikationen (nach Definition 5.2.9) und auf Elemente aus $Spec_\perp$ anwendbar.

Sei auf $Spec_\perp$ die Relation $\sqsubseteq_{Spec}$ wie folgt definiert:

$$(1) \quad <\Sigma, C> \sqsubseteq_{Spec} <\Sigma', C'> \ \ gdw. \ \ \Sigma = \Sigma' \ und \ C' \subseteq C$$

und

$$(2) \quad \perp_{Spec} \sqsubseteq_{Spec} sp \ \ für \ alle \ sp \ in \ Spec_\perp$$

Die Relation $\sqsubseteq_{Spec}$ basiert also auf der Modellklasseninklusion.

Folgerung 9.1.1 *$(Spec_\perp, \sqsubseteq_{Spec})$ ist eine Halbordnung.*

Beweis: Reflexivität, Antisymmetrie und Transitivität der Relationen $=$ und $\subseteq$ übertragen sich. $\square$

Folgerung 9.1.2 *Seien SP und SP' zwei Spezifikationen gleicher Signatur Σ. Dann gilt:*

$$< \Sigma, Mod(SP) > \sqsubseteq_{Spec} < \Sigma, Mod(SP') > \quad gdw. \;\; SP \rightsquigarrow SP'$$

Beweis: Folgt direkt aus den Definitionen von $\sqsubseteq_{Spec}$ und $\rightsquigarrow$. In beiden Relationen kommt hier die Modellklassenbildung Mod nach Definition 5.2.9 zur Anwendung. $\square$

Die Definition der Halbordnung $(Spec_\perp, \sqsubseteq_{Spec})$ abstrahiert über der axiomatischen Beschreibung der Modelle und der Bildung von Relationen durch $\rightsquigarrow$. Da für dieses Kapitel die Modellklassensemantik und die Implementierungsrelation aus den vorangegangenen Kapiteln als unveränderlich angesehen werden können, wird in der Regel die konkretere Notation über $\rightsquigarrow$ bzw. $\subseteq$ der Ordnung $\sqsubseteq_{Spec}$ bevorzugt. Dies erleichtert die Verständlichkeit der Beweise.

Satz 9.1.1 *$(Spec_\perp, \sqsubseteq_{Spec})$ ist eine cpo.*

Beweis:

1. $\perp_{Spec} \in Spec_\perp$ ist kleinstes Element (wegen (2) in der Definition von $\sqsubseteq_{Spec}$).

2. $(Spec_\perp, \sqsubseteq_{Spec})$ ist eine Halbordnung nach Folgerung 9.1.1. Die Relation $sp_1 \sqsubseteq_{Spec} sp_2$ gilt für zwei Elemente aus $Spec_\perp$, falls sie die gleiche Signatur haben und die sp_2 zugeordnete Menge von Σ-Objekten eine Teilmenge der sp_1 zugeordneten Menge von Σ-Objekten ist. Ketten sind somit durch die Teilmengenbeziehung auf Σ-Objektmengen bestimmt. Supremum einer Kette ist daher immer der Durchschnitt aller Σ-Objektmengen der Kette. $\square$

9.1.2 Spezifikationsoperatoren

Spezifikationsoperatoren werden durch Spezifikationsfunktionen realisiert. Diese Spezifikationsfunktionen $f : Spec_\perp \rightarrow Spec_\perp$ sollen die Struktur der Modellklassen erhalten; sie sollen also strikt und monoton sein[1]. Damit ist gewährleistet, daß die Spezifikationsoperatoren die Semantik (Modellklasseninklusion) erhalten.

Definition 9.1.2 *Eine Abbildung $f : Spec_\perp \rightarrow Spec_\perp$ heißt* **Spezifikationsfunktion**, *wenn gilt:*

1. *f ist strikt, also: $f(\perp_{Spec}) = \perp_{Spec}$,*

2. *f ist monoton, also: $sp \sqsubseteq_{Spec} sp'$ impliziert $f(sp) \sqsubseteq_{Spec} f(sp')$ für alle $sp, sp' \in Spec_\perp$.*

[1] Damit sind sie dann auch stetig, da die Suprema, wie in Satz 9.1.1 gezeigt, Elemente der Ketten sind.

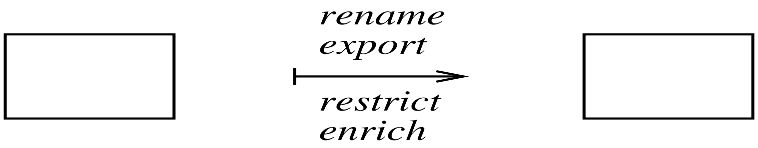


Abbildung 9.2: Spezifikationsoperatoren

Einige Notationen sollen nun eingeführt werden. Die Semantikfunktionen $\mathcal{M}$ und $\mathcal{M}^{op}$ bilden Spezifikationsausdrücke bzw. Spezifikationsoperatoren auf ihre Semantik ab.

$$\mathcal{M} : Spec\_Expr \rightarrow Spec_{\perp}$$

$$\mathcal{M}^{op} : Spec\_Op \rightarrow (Spec_{\perp} \rightarrow Spec_{\perp})$$

Ein Spezifikationsausdruck wird auf seine Modellsemantik abgebildet. Spezifikationsoperatoren werden auf Abbildungen auf $Spec_{\perp}$ abgebildet. Einer Spezifikation SP aus $Spec\_Expr$ wird durch $\mathcal{M}(SP)$ ein Element sp zugeordnet, wobei sp das zugehörige Bild in $Spec_{\perp}$ mit $sp = < sig(SP), Mod(SP) >$ ist. Die Semantik der Anwendung eines Spezifikationsoperators SP_OP auf einen Spezifikationsausdruck SP_EXPR läßt sich durch die Funktionsanwendung $\mathcal{M}^{op}(SP\_OP)$ $(\mathcal{M}(SP\_EXPR))$ beschreiben. Dieser Ausdruck liefert ein Element aus $Spec_{\perp}$.

Im folgenden kann für Spezifikationsausdrücke SP die Anwendung der Semantikfunktion $\mathcal{M}$ weggelassen werden, wenn keine freien Variablen (s.u.) in SP verwendet werden. Um Verwirrungen zu vermeiden, werden in diesem Kapitel Spezifikationsausdrücke mit Großbuchstaben $(SP_1, SP_2, \ldots)$ und Elemente aus $Spec_{\perp}$ mit Kleinbuchstaben $(sp_1, sp_2, \ldots)$ bezeichnet.

Die Monotonie der Spezifikationsfunktionen ist von Bedeutung für die Erfüllung der *horizontalen Kompositionseigenschaft.* Diese fordert, daß die Spezifikationsoperatoren die Implementierungsrelation, d.h. die Modellklasseninklusion, erhalten sollen.

Es wurden bisher Grundlagen einer Sprache auf Spezifikationen definiert. Mit dieser Sprache können Spezifikationsausdrücke formuliert werden. Syntaktisch stehen Spezifikationen und Spezifikationsoperatoren zur Verfügung. Spezifikationsausdrücke werden durch Modellsemantiken interpretiert. Spezifikationsfunktionen sind Funktionen auf Modellsemantiken (Modellklassen). An diese Funktionen wird die Stetigkeitsforderung gestellt. Im folgenden werden vier Spezifikationsoperatoren (*Umbenennung, Export, Restriktion* und *Erweiterung*) als Elemente von $Spec\_Op$ definiert, die eine Spezifikation in eine andere transformieren (siehe Abbildung 9.2). Für jeden Operator wird die Semantik durch $\mathcal{M}^{op}$ definiert. Daß dadurch Spezifikationsfunktionen definiert werden, wird jeweils nachgewiesen.

Definition 9.1.3 *Die* **Umbenennung** *rename SP by* ρ *ist ein Spezifikationoperator, falls* ρ *ein bijektiver Signaturmorphismus ist.* ρ *bildet Zustandsbezeichner, Sorten- und Operationsnamen ab.* $sig(rename\ SP\ by\ \rho) := \Sigma$, *falls* $\rho(sig(SP)) = \Sigma$, *und* $Mod(rename\ SP\ by\ \rho) := \{O|_{\rho} \in Obj(\Sigma) \mid O \in Mod(SP)\}$.

Die Modelle der Umbenennung sind die Redukte, die über Signaturmorphismen gebildet werden (vgl. Definitionen 3.3.7 und 3.3.8).

Folgerung 9.1.3 *$Mod(SP) = Mod(rename\ SP\ by\ \rho)$.*

Beweis: Es werden nur Umbenennungen durchgeführt. Die Strukturen bleiben unverändert, da ρ ein bijektiver Spezifikationsmorphismus ist. □

Lemma 9.1.1 *rename definiert eine Spezifikationsfunktion, d.h. $\mathcal{M}^{op}(rename\ .\ by\ \rho) : Spec_\perp \to Spec_\perp$ ist eine Spezifikationsfunktion.*

Beweis: Nach Folgerung 9.1.3 bleibt die Modellklasse unverändert. Mit ρ als bijektivem Signaturmorphismus folgen also Striktheit und Monotonie sofort. □

Definition 9.1.4 *Der* **Export** *$export\ \Sigma\ from\ SP$ ist ein Spezifikationsoperator, falls Σ eine Subsignatur von $sig(SP)$ ist. Dann gilt: $sig(export\ \Sigma\ from\ SP) := \Sigma$ und $Mod(export\ \Sigma\ from\ \ SP) := \{O|_\Sigma \in Obj(\Sigma) |\ O \in Mod(SP)\}$.*

Es handelt sich hier um keine Exportschnittstelle eines Modulkonzeptes im Sinne einer Verkapselung der internen Realisierung. Der Export ist eine rein syntaktische Konstruktion, die zum Ausblenden von Signaturelementen dient. Semantische Beschreibungen des Exports sind nicht möglich.

Lemma 9.1.2 *export definiert eine Spezifikationsfunktion.*

Beweis: Es sind Striktheit und Monotonie nachzuweisen.

1. Striktheit: Eine echte Subsignatur von $sig(\perp_{Spec})$ existiert nicht, d.h. es ist
$$export\ sig(\perp_{Spec})\ from\ \perp_{Spec} = < sig(\perp_{Spec}), Obj(\perp_{Spec}) > \ .$$
2. Monotonie: Sei $sp \sqsubseteq_{Spec} sp'$, d.h. $Mod(sp') \subseteq Mod(sp)$. Aus
$$O \in Mod(sp') \ \Rightarrow\ O \in Mod(sp)$$
folgt mit der Definition des Exports
$$O|_\Sigma \in Mod(export\ \Sigma\ from\ sp') \ \Rightarrow\ O|_\Sigma \in Mod(export\ \Sigma\ from\ sp)\ .$$
D.h. die Inklusion
$$Mod(export\ \Sigma\ from\ sp') \ \subseteq\ Mod(export\ \Sigma\ from\ sp)$$
gilt. Damit gilt auch
$$export\ \Sigma\ from\ sp \ \ \sqsubseteq_{Spec}\ \ export\ \Sigma\ from\ sp'\ .$$

□

Definition 9.1.5 *Eine eingeschränkte Form des Exports soll als* **Restriktion** *definiert werden:*
$$restrict\ SP\ by\ \Sigma := export\ \Sigma\ from\ SP$$
mit $state\_comp(\Sigma) = state\_comp(sig(SP))$.

Die Zustandskomponenten bleiben gleich, nur die Funktionalität wird eingeschränkt.

Lemma 9.1.3 *restrict definiert eine Spezifikationsfunktion.*

Beweis: Trivial, da *restrict* ein Spezialfall von *export* ist. □

Definition 9.1.6 *Die* **Erweiterung** $enrich\ SP\ by\ S', Z', OP', E'$ *ist ein Spezifikationsoperator, falls* SP *eine Spezifikation,* S' *eine Menge von Sorten,* Z' *eine Menge von Zustandsbezeichnern,* OP' *eine Menge von Operationssymbolen und* $E' \in WFF(sig(enrich\ SP\ by\ S', Z', OP', E'))$ *ist.*

Es gilt

$sig(enrich\ SP\ by\ S', Z', OP', E') := sig(SP) \cup sig(< S', Z', OP' >)$

und

$Mod(enrich\ SP\ by\ S', Z', OP', E') :=$
$\{A \in Obj(sig(enrich\ SP\ by\ S', Z', OP', E')) \mid A|_{sig(SP)} \in Mod(SP) \wedge A \models e \text{ für alle } e \in E'\}.$

Lemma 9.1.4 *enrich definiert eine Spezifikationsfunktion.*

Beweis:

1. Striktheit: $SP = \perp_{Spec}$, d.h. $Mod(\perp_{Spec}) = Obj(\perp_{sign} \cup sig(< S', Z', OP' >))$. Da die Signaturvereinigung $\cup$ strikt ist (siehe Definition 3.2.5), ist $enrich\ SP\ by\ S', Z', OP', E'$ $= < \perp_{Sign}, Obj(\perp_{Sign}) > = \perp_{spec}$.

2. Monotonie: analog *export* in Lemma 9.1.2.

□

Die horizontale Kompositionseigenschaft fordert, daß die Spezifikationsoperatoren mit der Implementierungsrelation verträglich sind. Später werden für Spezifikationsausdrücke, etwa für die Komposition, explizite Korrektheitsbegriffe eingeführt. Im Augenblick soll angenommen werden, daß die Korrektheit immer erfüllt ist. Korrektheitsbedingungen sind in den drei Operatoren implizit enthalten. So muß etwa ρ bei der Umbenennung ein bijektiver Spezifikationsmorphismus sein oder Σ muß bei Export oder Restriktion eine Subsignatur der Signatur der zugrundeliegenden Spezifikation sein. Der folgende Satz wird in Abbildung 9.3 veranschaulicht.

Satz 9.1.2 *rename, export, restrict, enrich erfüllen die horizontale Kompositionseigenschaft. D.h. es gilt für korrekte Spezifikationen* SP *und* $IMPL$*:*

1. *Aus*

   $$rename\ SP\ by\ \rho \text{ ist korrekt und } SP \rightsquigarrow IMPL$$

   folgt

   $$rename\ SP\ by\ \rho \rightsquigarrow rename\ IMPL\ by\ \rho \text{ und } rename\ IMPL\ by\ \rho \text{ ist korrekt.}$$

2. *Aus*

$$export\ \Sigma\ from\ SP \text{ ist korrekt und } SP \rightsquigarrow IMPL$$

folgt

$$export\ \Sigma\ from\ SP \rightsquigarrow export\ \Sigma\ from\ IMPL \text{ und } export\ \Sigma\ from\ IMPL \text{ ist korrekt.}$$

3. *Aus*

$$restrict\ SP\ by\ \Sigma \text{ ist korrekt und } SP \rightsquigarrow IMPL$$

folgt

$$restrict\ SP\ by\ \Sigma \rightsquigarrow restrict\ IMPL\ by\ \Sigma \text{ und } restrict\ IMPL\ by\ \Sigma \text{ ist korrekt.}$$

4. *Aus*

$$enrich\ SP\ by\ S', Z', OP', E' \text{ ist korrekt und } SP \rightsquigarrow IMPL$$

folgt

$$enrich\ SP\ by\ S', Z', OP', E' \rightsquigarrow enrich\ IMPL\ by\ S', Z', OP', E'$$

und

$$enrich\ IMPL\ by\ S', Z', OP', E' \text{ ist korrekt.}$$

Beweis: Folgt, da $rename, export, restrict, enrich$ durch Spezifikationsfunktionen definiert sind. Die Funktionen sind monoton und erhalten damit die Ordnung, die durch $\rightsquigarrow$ bestimmt ist. Sowohl die Ordnung $\sqsubseteq_{Spec}$ als auch $\rightsquigarrow$ basieren auf der Modellklasseninklusion (siehe auch Folgerung 9.1.2). Für die Spezifikationsoperatoren gibt es keinen expliziten Korrektheitsbegriff, der noch zu erfüllen wäre. Signaturen brauchen nicht betrachtet werden, da $sig(SP) = sig(IMPL)$ gilt, falls $SP \rightsquigarrow IMPL$. □

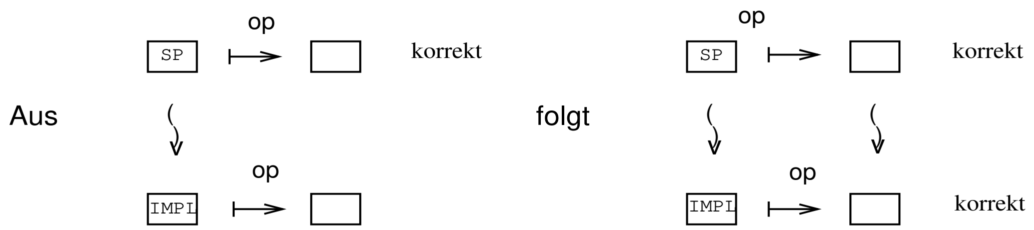


Abbildung 9.3: Spezifikationsoperatoren erhalten die Implementierung

Hier ist nur ein Basissatz von Spezifikationsoperatoren vorgestellt worden. Die Auswahl orientiert sich an [HN92]. Von dort wurde auch die Zuordnung von expliziten Korrektheitsbedingungen für Spezifikationsoperatoren übernommen. In Kapitel 7 wurde ein Korrektheitsbegriff für Implementierungen eingeführt. Der hier benutzte Korrektheitsbegriff bezieht sich auf Spezifikationsausdrücke. Eine Verwechslung ist somit nicht möglich. Die Spezifikationsoperatoren sind hier durch die Theorie strukturierter algebraischer Spezifikationen aus [Wir90] Kapitel 6 auf Basis der Modellsemantik aus Kapitel 5 definiert. [HN92] setzt auf Kategorientheorie nach [EM85, EM90] auf. Die Operatoren sind dort im Kontext eines Wiederverwendungsansatzes definiert. Mit *enrich* kann ein formaler Import zu einem Rumpf erweitert werden. Mit *export/restrict* und *rename* können Spezifikationen an die Bedingungen eines formalen Imports einer parametrisierten Spezifikation bei dessen Aktualisierung angepaßt werden. *export* und *restrict* können ebenfalls benutzt werden, um syntaktische Exportschnittstellen zu konstruieren. Parametrisierung von Spezifikationen, deren Aktualisierung (Komposition) und Exportschnittstellen werden in den nächsten Abschnitten betrachtet. Weitere Operatoren, die an den vorliegenden Ansatz angepaßt werden können, finden sich in den Standardwerken zur algebraischen Spezifikation [EM85, EM90, Wir90].

9.2 Parametrisierte Spezifikationen

Das Kernkonzept der horizontalen Entwicklung ist die Modularisierung. Mit ihrer Hilfe wird ein komplexes System in mehrere, unabhängig voneinander entwickelbare Teilsysteme zerlegt. Jede Komponente wird dazu mit einer Importschnittstelle versehen. Nur eine formale Importschnittstelle garantiert, daß Komponenten unabhängig von ihrem Benutzungskontext entwickelt werden können. Dies war eine der Grundforderungen des in Kapitel 1.4.2 vorgestellten Komponentenmodells. Zwei Komponenten werden zu einer zusammengesetzt, indem der Import der einen Komponente mit Leistungen der anderen aktualisiert wird.

In diesem Abschnitt werden parametrisierte Spezifikationen eingeführt. Sie werden über λ-Abstraktion definiert und zusätzlich mit einem Korrektheitsbegriff versehen. Das Konstrukt der Komposition erlaubt die Zusammensetzung von parametrisierten Spezifikationen. Sie wird ebenfalls über λ-Abstraktion definiert. Damit kann die Komposition auch als Spezifikationsfunktion nachgewiesen werden. Ein geeigneter Korrektheitsbegriff wird auch für die Komposition definiert. Dieser stellt eine Verschärfung der Definition über λ-Ausdrücke dar, die die Benutzung der Relationen $\rightsquigarrow$ und $\triangleright$ erlaubt. Es wird gezeigt, daß korrekte Kompositionen die Implementierungsrelation erhalten, d.h. daß die horizontale Kompositionseigenschaft gilt. Im Anschluß daran erfolgen einige Überlegungen zur Methodik des Imports und zu einer Benutztbeziehung zwischen einer Spezifikation mit formalem Import und deren Aktualisierung.

9.2.1 Parametrisierung

Definition 9.2.1 *Eine* **parametrisierte Spezifikation** *PS ist ein Paar $< SP_I, SP_R >$ aus Spezifikationsausdrücken SP_I und SP_R. SP_I wird* Importspezifikation *und SP_R* Rumpfspezifikation *genannt.*

Mögliche Alternativen der Semantikdefinition für parametrisierte Spezifikationen sind (nach [Wir90], Kapitel 7):

1. Die Definition erfolgt über λ-Abstraktion. Insbesondere kann zur flexibleren Behandlung der Aktualisierungen von Parametern das $\lambda\pi$-Kalkül eingesetzt werden (siehe auch [Feij89]).
2. Einsatz der Kategorientheorie. SP_R kann als *Erweiterung* — einer geeigneten Spezifikationsoperation — von SP_I definiert werden. Dieser Ansatz wird in der Literatur [EM85, TWW82] auch *Pushout-Approach* genannt. Pushouts werden etwa in [EM90] oder [HN92] zur Definition von Parametrisierungs- oder Modulkonstrukten eingesetzt. Da hier die Kategorientheorie als formale Grundlage eingeführt werden müßte, soll dem schon benutzten λ-Kalkül der Vorzug gegeben werden.

Der erste Punkt (λ-Abstraktion) soll jetzt näher untersucht werden. Eine parametrisierte Spezifikation PS wird als λ-Ausdruck formuliert:

$$PS \equiv \lambda X : SP_I .\ SP_R$$

SP_I ist eine Parameterrestriktion (die Importspezifikation) und SP_R ein Spezifikationsausdruck (die Rumpfspezifikation). X ist freie Variable. Sie kann in SP_R benutzt werden. X darf nicht in SP_I benutzt werden. Sie wird mit dem aktuellen Parameter belegt werden.

Semantisch soll der parametrisierte Spezifikationsausdruck $< SP_I, SP_R >$ als Funktion von $sig(SP_I)$-Modellen in $sig(SP_R)$-Modelle definiert werden. $env \in Env$ ist eine Umgebung, in der die freien Variablen $X \in Var$ verwaltet werden, also $env : Var \rightarrow Spec_\perp$ (siehe [Wir90] S.740). Durch env werden die Aktualisierungen des formalen Imports an die Variablen gebunden. Env ist die Menge aller Umgebungen. $\mathcal{M}^{par}$ soll die Semantik einer λ-Applikation beschreiben. Es soll $\mathcal{M}^{par}$ analog zu $\mathcal{M}^{op}$, der Semantik der Spezifikationsoperatoren, vereinbart werden.

$$\mathcal{M}^{par} : Par\_Spec\_Expr \rightarrow (Env \rightarrow (Spec_\perp \rightarrow Spec_\perp))$$

Par_Spec_Expr bezeichnet einen parametrisierten Spezifikationsausdruck. Gegenüber der Semantik der Spezifikationsoperatoren muß hier noch die Umgebung Env berücksichtigt werden. Es soll nach der Definition gezeigt werden, daß parametrisierte Spezifikationen ebenso wie Spezifikationsoperatoren Spezifikationsfunktionen definieren.

Definition 9.2.2 *Sei $\mathcal{M}$ als Semantik für nichtparametrisierte Spezifikationen vorgegeben. Darauf wird $\mathcal{M}^{par}(PS) : Env \rightarrow Spec_\perp \rightarrow Spec_\perp$ als Semantik einer parametrisierten Spezifikation $PS \equiv \lambda X : SP_I .\ SP_R$ definiert, wobei SP_I, SP_R Spezifikationsausdrücke sind*[2]. *Es sei definiert:*

$$\mathcal{M}^{par}(\lambda X : SP_I .\ SP_R)(env)(sp) := \begin{cases} \mathcal{M}(SP_R)(env[X \mapsto sp]) & falls\ sp \neq \perp_{Spec}\ und \\ & \mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} sp \\ \perp_{Spec} & sonst \end{cases}$$

$env[X \mapsto sp]$ beschreibt die Bindung von X an den aktuellen Wert sp in der Umgebung $env \in Env$.

$\mathcal{M}^{par}$ ist strikt. $sp \in Spec_\perp$ muß die Importspezifikation SP_I erfüllen, ausgedrückt in der Modellsemantik durch $\mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} sp$, d.h. unter Berücksichtigung der Umgebung env soll sp eine Implementierung der Parameterrestriktion SP_I sein. Die Semantik $\mathcal{M}(SP)$ einer einfachen Spezifikation SP ist deren zugeordnetes Element $< \Sigma, C >$ aus $Spec_\perp$.

Folgerung 9.2.1 *Die Definition der parametrisierten Spezifikation entspricht für den Fall, daß X die einzige freie Variable ist, der Implementierung von SP_I durch sp (d.h. $Mod(sp) \subseteq Mod(SP_I)$).*

Beweis: Folgt aus der Definition von $\sqsubseteq_{spec}$ über die Umkehrung der Mengeninklusion für Modellklassen. env ist nicht zu berücksichtigen, da nach Voraussetzung X die einzige zu bindende Variable ist. $\square$

---

[2] $\mathcal{M}^{par}$ arbeitet auf Elementen von $Spec_\perp$. Bezeichner aus Großbuchstaben sind weiterhin Spezifikationsausdrücke, die durch $\mathcal{M}$ auf das zugehörige Element in $Spec_\perp$ abgebildet werden.

Später wird in Abschnitt 9.2.2 diese Formalisierung der Parametrisierung durch ein Kompositionskonstrukt noch verallgemeinert (eine syntaktische Anpassung des aktuellen Parameters an den formalen durch einen Spezifikationsmorphismus wird dann zugelassen).

Definition 9.2.3 *Sei $PS \equiv \lambda X : SP_I.\ SP_R$ eine parametrisierte Spezifikation. Dann ist $(\lambda X : SP_I.\ SP_R)(SP)$ ein Spezifikationsausdruck, falls SP ein Spezifikationsausdruck ist.*

Satz 9.2.1 *$PS \equiv \lambda X : SP_I.\ SP_R$ sei eine parametrisierte Spezifikation, wobei X nicht in SP_I auftritt. Dann ist $\mathcal{M}^{par}(PS)(env) : Spec_\perp \to Spec_\perp$ eine Spezifikationsfunktion für jedes $env \in Env$.*

Beweis: Siehe auch Fact 7.1.1 (1) in [Wir90] Kapitel 7. Es sind Striktheit und Monotonie zu zeigen.

1. Striktheit: nach Definition.
2. Monotonie: Sei $sp \sqsubseteq_{Spec} sp'$. Dann ist

$$\mathcal{M}^{par}(\lambda X : SP_I.\ SP_R)(env)(sp) \sqsubseteq_{Spec} \mathcal{M}^{par}(\lambda X : SP_I.\ SP_R)(env)(sp'),$$

   da

   (a) $sp = \perp_{Spec}$: $\perp_{Spec} \sqsubseteq_{Spec} sp'$ nach Definition von $\sqsubseteq_{Spec}$, also $sp' = \perp_{Spec} \Rightarrow sp = \perp_{Spec} \Rightarrow sp \sqsubseteq_{Spec} sp'$.

   (b) $sp \neq \perp_{Spec}$: $\mathcal{M}(SP_R)(env[X \mapsto sp]) \sqsubseteq_{Spec} \mathcal{M}(SP_R)(env[X \mapsto sp'])$ wird durch Induktion über die Struktur von SP_R gezeigt. Sei SP_R primitiv. SP_R ist also nicht von X abhängig, also gilt $\mathcal{M}(SP_R) \sqsubseteq_{Spec} \mathcal{M}(SP_R)$. Mit $\mathcal{M}(SP_R)(env[X \mapsto sp]) \sqsubseteq_{Spec} \mathcal{M}(SP_R)(env[X \mapsto sp'])$ folgt $\mathcal{M}^{par}(\lambda X : SP_I.\ SP_R)(env)(sp) \sqsubseteq_{Spec} \mathcal{M}^{par}(\lambda X : SP_I.\ SP_R)(env)(sp')$ nach Definition. □

Bemerkung 9.2.1 *Statt $\lambda X : SP_I.\ SP_R$ soll auch $< SP_I, SP_R >$ geschrieben werden, wenn etwa auf die Aktualisierung von X noch nicht bezug genommen werden soll.*

Parametrisierte Spezifikationen wurden durch λ-Ausdrücke beschrieben, um sie wie *rename*, *export*, *restrict* und *enrich* auch als Spezifikationsoperatoren behandeln zu können. Da parametrisierte Spezifikationen auch Spezifikationsfunktionen definieren, sind Striktheit und Monotonie, also die Erhaltung der semantischen Strukturen, gegeben.

Definition 9.2.4 *Sei SP eine Spezifikation mit Signatur Σ' und sei ρ ein Signaturmorphismus $\rho : \Sigma \to \Sigma'$. Dann wird durch $SP|_\rho$ eine Spezifikation mit der Semantik $< \Sigma, Mod(SP)|_\rho > \in Spec_\perp$ definiert.*

Diese Notation ermöglicht es, auch auf Spezifikationsebene die Reduktbildung bezeichnen zu können (etwa in Ausdrücken, in denen $\rightsquigarrow$ benutzt wird). Diese Konstruktion wird in der Literatur auch als *forgetful mapping* bezeichnet (siehe [FJ92], Kapitel 8.8).

Parametrisierte Spezifikationen können verschachtelt werden. Die dabei benutzten Variablen aus Var werden in einer Umgebung env verwaltet. Diese Umgebung kann aber nur für Elemente aus $Spec_\perp$ benutzt werden. Jetzt soll eine Notation eingeführt werden, die dies aus der

Ebene der Spezifikationsausdrücke möglich macht. Sei $SP \in Spec\_Expr$ ein beliebiger Spezifikationsausdruck, der freie Variablen $X_1, \ldots, X_n \in Var$ enthält und $env : Var \to Spec_\perp$ eine Umgebung.

$$SP^{env} := SP[X_1/SP_1, \ldots, X_n/SP_n]$$

Alle freien Vorkommen der X_i werden ersetzt, wobei SP_i die Variable X_i syntaktisch ersetzt, falls $env(X_i) = sp_i$ mit $sig(SP_i) = sig(sp_i)$ und $Mod(SP_i) = Mod(sp_i)$. Diese Konstruktion ist notwendig, um die Implementierungsrelation auf Spezifikationen mit freien Variablen anwenden zu können.

Der Spezifikationsoperator PS (*parametrisierte Spezifikation*) wird nun mit einem Korrektheitsbegriff versehen, der weitere (semantische) Eigenschaften festlegt.

Definition 9.2.5 *Sei $env \in Env$ eine Umgebung zur Belegung der freien Variablen aus Var. Eine parametrisierte Spezifikation $SP = < SP_I, SP_R >$ heißt* **korrekt**, *falls*

$$sig(SP_I^{env}) \subseteq sig(SP_R^{env}) \quad \wedge \quad SP_R^{env}|_{sig(SP_I^{env})} \rightsquigarrow SP_I^{env} \quad .$$

Also ist Korrektheit gegeben, falls $Mod(SP_I^{env}) \subseteq Mod(SP_R^{env})|_{sig(SP_I^{env})}$. Jedes Modell des Imports kann zu einem Modell des Rumpfes erweitert werden. Der Rumpf erhält somit den Import. Diese Form der Definition einer parametrisierten Spezifikation ist auch in [Hen91a, Hen92, HN92] oder [EM85] Kapitel 7 zu finden. Die Signaturinklusion erlaubt die Reduktbildung. Der angewendete Signaturmorphismus ist eine Einbettung. Der Rumpf ist also syntaktisch eine Erweiterung des Importes.

Es soll hier nochmals darauf hingewiesen sein, daß in einer parametrisierten Spezifikation $< SP_I, SP_R >$ mit der Semantik $\lambda X : SP_I.SP_R$ die Variable X in SP_R, nicht aber in SP_I verwendet werden darf. In SP_I dürfen aber, wie in SP_R auch, die freien Variablen umgebender parametrisierter Spezifikationen benutzt werden. Die Definition der parametrisierten Spezifikation wurde hier gegenüber [Wir90] leicht eingeschränkt. Dort ist eine Verwendung von X in der zugehörigen Importspezifikation SP_I gestattet. Da das Abhängigmachen der Parameterrestriktion vom aktuellen Parameter nicht zur Verständlichkeit beiträgt, haben wir hierauf verzichtet.

Parametrisierte Spezifikationen wurden neben der Definition über λ-Abstraktion (die sich auf die Applikation, also die Aktualisierung des formalen Imports bezog) auch mit einem Korrektheitsbegriff versehen, der auf der Implementierungsrelation basiert. Der formale Parameter X vom Typ SP_I soll im Rumpf SP_R Verwendung finden. Die Beziehung, die zwischen Parameterrestriktion und Rumpf in einer korrekten parametrisierten Spezifikation entsteht, soll nun benannt werden.

Definition 9.2.6 *SP_I ist* **Import von** *SP_R, falls $< SP_I, SP_R >$ eine korrekte parametrisierte Spezifikation ist.*

Folgerung 9.2.2 *Der Import impliziert die Implementierungsrelation, falls die Signaturen gleich sind.*

Beweis: Folgt direkt aus den Definitionen des Imports und der Korrektheit für parametrisierte Spezifikationen. □

Damit ist gezeigt, daß die Korrektheitsbedingung sinnvoll gewählt ist. Jetzt soll noch veranschaulicht werden, wie eine parametrisierte Spezifikation durch den Spezifikationsoperator $enrich$ unterstützt werden kann.

Sei eine Spezifikation

$$\begin{array}{lll} \textit{spec ELEM is} & & \\ & \textit{sorts} & \textit{elem, bool} \\ & \textit{opns} & eq : elem \times elem \rightarrow bool \\ & \textit{axioms} & \ldots \\ \textit{end spec} & & \end{array}$$

gegeben. Dann kann eine parametrisierte Spezifikation, hier am Beispiel der parametrisierten Mengen

$$ParSet \equiv < ELEM, SET_{ELEM} >$$

mit Parameterrestriktion $ELEM$ und formaler Parametervariablen X, wie folgt beschrieben werden:

$$\begin{array}{lll} \textit{spec } SET_{ELEM} \textit{ is} & & \\ & \textit{enrich X by} & \\ & \textit{sorts} & \textit{set} \\ & \textit{opns} & empty : \rightarrow set \\ & & insert : set \times elem \rightarrow set \\ & \textit{axioms} & \ldots \\ \textit{end spec} & & \end{array}$$

Dies läßt sich durch einen λ-Ausdruck beschreiben:

$$ParSet \equiv \lambda X : ELEM \;.\; SET_{ELEM}$$

Dieser Ausdruck könnte etwa mit

$$ParSet(NAT)$$

aufgerufen werden, um eine Spezifikation von Mengen natürlicher Zahlen zu erhalten. Die Sorte *elem* der Importspezifikation wird durch *nat* aktualisiert.

Bemerkung 9.2.2 *Die Konstruktion SP^{env} aus SP muß überall da angewendet werden, wo Signaturbildung, Modellbildung oder die Relationen $\leadsto$ und $\rhd$ auf Ebene der Spezifikationsausdrücke angewendet werden sollen. Im folgenden nehmen wir an, daß dies implizit geschieht. Wir werden dann das 'env' in SP^{env} zur Vereinfachung der Notation weglassen. Die Existenz einer Umgebung $env \in Env$ sei angenommen.*

9.2.2 Komposition

Nachdem parametrisierte Spezifikationen definiert sind, soll nun die Komposition eingeführt werden. Die Komposition beschreibt die Aktualisierung des formalen Imports einer parametrisierten Spezifikation. Grundlagen hierfür sind durch die Definition der Parametrisierung über λ-Abstraktion nach [Wir90] schon gelegt worden. Ein Kompositionskonstrukt in der unten angegebenen Form wird von [Wir90] nicht angeboten. Entsprechendes ist aber in [EM85, EM90] und in einer Anwendung davon in [Hen91a, Hen92] zu finden. Die Ideen der dort zugrundeliegenden Definition über Pushouts werden hier Anwendung finden.

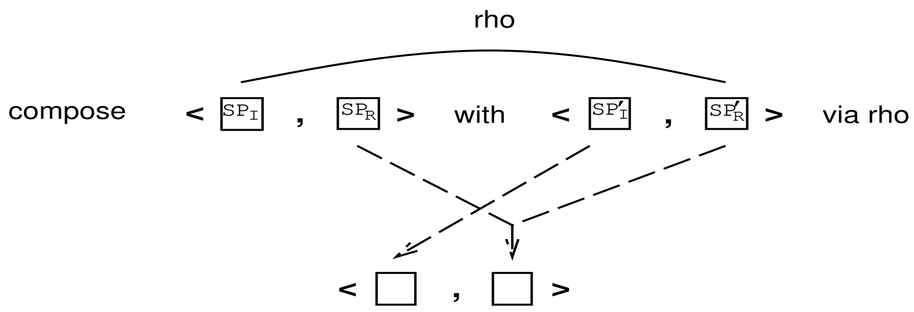


Abbildung 9.4: Komposition von parametrisierten Spezifikationen

Definition 9.2.7 *Seien $SP = < SP_I, SP_R >$ und $SP' = < SP'_I, SP'_R >$ parametrisierte Spezifikationen bestehend aus Import- und Rumpfspezifikation.*

$$compose < SP_I, SP_R > with < SP'_I, SP'_R > via\ \rho$$

ist eine **Komposition**, *falls ρ ein Signaturmorphismus $\rho : sig(SP_I) \rightarrow sig(SP'_R)$ ist. SP importiert von SP'. Alle Importe aus SP_I müssen erfüllt sein, es müssen aber nicht alle Exporte aus SP'_R benutzt werden. Der Signaturmorphismus ρ muß diese Anpassung leisten.*

$$compose\ \ < SP_I, SP_R >\ \ with\ \ < SP'_I, SP'_R >\ \ via\ \rho\ \ := \ < SP'_I, (\lambda X : SP_I.SP_R)(SP'_R|_\rho) >$$

Die Definition ist kompositional, d.h. SP' kann ein beliebiger Spezifikationsausdruck sein.

In einem Kompositionsausdruck $compose\ SP\ with\ SP'\ via\ \rho$ ist die erste angegebene Spezifikation (hier SP) die importierende (siehe Abbildung 9.4). SP_I ist die formale Parameterrestriktion, SP'_R ist aktueller Parameter (nach syntaktischer Anpassung durch ρ). Durch Expansion des λ-Ausdrucks in der Definition ergibt sich die folgende Eigenschaft:

$$compose\ SP\ with\ SP'\ via\ \rho \ \ = \ < SP'_I, \mathcal{M}(SP_R)(env[X \mapsto SP'_R|_\rho]) >$$

falls $\mathcal{M}(SP'_R|_\rho) \neq \perp_{Spec}$ und $\mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} \mathcal{M}(SP'_R|_\rho)(env)$. Import der zusammengesetzten Spezifikation ist der Import der Aktualisierung. Der Rumpf der zusammengesetzten Spezifikation ergibt sich aus den Rümpfen der beiden Ausgangsspezifikationen.

Für die Komposition soll ein Korrektheitsbegriff definiert werden, der die Forderung der Parameterrestriktion aus der λ-Definition explizit macht. Es wird hier zusätzlich noch die Möglichkeit der syntaktischen Anpassung zugelassen.

Definition 9.2.8 *Die Komposition*

$$compose < SP_I, SP_R > with < SP'_I, SP'_R > via\ \rho$$

heißt **korrekt**, *falls*

$$SP_I \rightsquigarrow SP'_R|_\rho.^3$$

[3] Vgl. Bemerkung 9.2.2.

Der aktuelle Import $SP'_R|_\rho$ muß den formalen Import SP_I übererfüllen, d.h. er muß genauso gut oder besser sein. Der aktuelle Import ist also ggf. eingeschränkter. Daher wurde die modellbasierte Implementierungsrelation $Mod(SP'_R|_\rho) \subseteq Mod(SP_I)$ als Definition herangezogen. Hier liegt als Annahme zugrunde, daß die Importspezifikation Anforderungen stellt, die an anderer Stelle durch andere Spezifikationen detaillierter beschrieben sind. Hier kann ein Vergleich zur Korrektheitsdefinition der parametrisierten Spezifikation gezogen werden. Dort wird ebenfalls eine Implementierungsrelation $\rightsquigarrow$, ausgedrückt durch Modellklasseninklusion, zur Definition herangezogen.

In einer einfachen Komposition — X ist hier die einzige freie Variable in der parametrisierten Spezifikation $< SP_I, SP_R >$ — gilt folgende Vereinfachung für die Definition der Komposition:

$$compose < SP_I, SP_R > with < SP'_I, SP'_R > via\ \rho \quad = \quad < SP'_I, SP_R[X/SP'_R|_\rho] >$$

Die zwei verschiedenen semantischen Bedingungen der Aktualisierung in der Komposition sollen im folgenden untersucht werden. Dies sind $\mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} \mathcal{M}(SP'_R)(env)$ in den Definitionen 9.2.2 und 9.2.7 und $SP_I \rightsquigarrow SP'_R|_\rho$ in der Korrekheit (Definition 9.2.8), falls eine Spezifikation $< SP_I, SP_R >$ mit dem Rumpf SP'_R einer Spezifikation $sp \in Spec_\perp$ aktualisiert wird. Damit wird gezeigt, daß die gewählte Korrektheitsbedingung sinnvoll und anwendbar ist.

Lemma 9.2.1 *Aus der Korrektheit einer Komposition (Definition 9.2.8)*

$$compose < SP_I, SP_R > with < SP'_I, SP'_R > via\ \rho$$

folgt die Erfüllung der Parameterrestriktion der λ-Abstraktion (Definitionen 9.2.2 und 9.2.7).

Beweis: $SP_I \rightsquigarrow SP'_R|_\rho$ als Korrektheitsbedingung ist durch Modellklasseninklusion definiert. Für Kompositionen ist eine syntaktische Anpassung des aktuellen Parameters an den formalen durch den Signaturmorphismus ρ möglich, die in der Definition der Parameterrestriktion $\mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} \mathcal{M}(SP'_R)(env)$ nicht enthalten ist. Erst nach syntaktischer Anpassung muß die Modellklasseninklusion gelten (siehe Definition 9.2.2 und der Definition von $\sqsubseteq_{Spec}$ über Inklusion von Modellklassen). Daher gilt auch die Aktualisierung über λ-Abstraktion, die auf den Modellklassen basiert, da sie auf $\sqsubseteq_{Spec}$ definiert ist. □

Lemma 9.2.2 *Seien $< SP_I, SP_R >$ und $< SP'_I, SP'_R >$ korrekte parametrisierte Spezifikationen und gelte*

$$SP_I \rhd SP'_R|_\rho.$$

Dann ist

$$compose\ < SP_I, SP_R >\ with\ < SP'_I, SP'_R >\ via\ \rho$$

eine korrekte Komposition.

Beweis: Folgt direkt aus Definition 9.2.8 und Satz 8.2.1. □

Dieses Lemma erlaubt dem Spezifizierer, korrekte Kompositionen zu erstellen, indem er den gegenüber $\rightsquigarrow$ handhabbareren Mechanismus $\rhd$ einsetzt.

Satz 9.2.2 *Falls* $< SP_I, SP_R >$ *und* $< SP'_I, SP'_R >$ *korrekte parametrisierte Spezifikationen sind, und auch*

$$compose < SP_I, SP_R > with < SP'_I, SP'_R > via\ \rho$$

eine korrekte Komposition mit $sig(SP'_R) \subseteq sig((\lambda X : SP_I.\ SP_R)(SP'_R|_\rho))$ *ist, so wird durch den Kompositionsausdruck auch eine korrekte parametrisierte Spezifikation definiert.*

Beweis: Nach Definition 9.2.5 ist die parametrisierte Spezifikation

$$compose < SP_I, SP_R > with < SP'_I, SP'_R > via\ \rho$$

korrekt, falls gilt (siehe auch Definition 9.2.7 für die Komposition und 9.2.5 für die Korrektheit parametrisierter Spezifikationen)

$$Mod(SP'_I) \subseteq Mod((\lambda X : SP_I.\ SP_R)(SP'_R|_\rho))|_{sig(SP'_I)}\ ,$$

also falls

$$Mod(SP'_I) \subseteq Mod(SP_R(env(X \mapsto SP'_R|_\rho)))|_{sig(SP'_I)}$$

gilt. $Mod(SP'_I) \subseteq Mod(SP'_R)|_{sig(SP'_I)}$ gilt, da $< SP'_I, SP'_R >$ als korrekte parametrisierte Spezifikation vorausgesetzt wurde. $Mod(SP'_R) \subseteq Mod((\lambda X : SP_I.\ SP_R)(SP'_R|_\rho))|_{sig(SP'_R)}$ gilt, da nach Definition des λ-Ausdrucks und der vorausgesetzten Korrektheit der Komposition SP'_R durch die Umgebung env syntaktisch in $(\lambda X : SP_I.\ SP_R)(SP'_R|_\rho)$ eingebettet ist, also jedes Modell von SP'_R zu einem Modell von $(\lambda X : SP_I.\ SP_R)(SP'_R|_\rho)$ erweitert werden kann. Die Forderung nach Signaturinklusion $sig(SP'_R) \subseteq sig((\lambda X : SP_I.\ SP_R)(SP'_R|_\rho))$ ermöglicht dies. Zusammen mit $sig(SP'_I) \subseteq sig(SP'_R)$ und $Mod(SP'_I) \subseteq Mod(SP'_R)|_{sig(SP'_I)}$ gilt die geforderte Modellklasseninklusion. □

In [Wir90] wird zwar parametrisierte Spezifikation über λ-Ausdrücke vorgestellt, deren Umsetzung in eine Komposition in der hier präsentierten Form fehlt aber. Das hier vorgestellte Kompositionskonstrukt ähnelt dem in [EM90] Section 3A bzw. [Hen91a, Hen92] vorgestellten. Dort werden Pushouts zur Definition benutzt. In [Hen91a, Hen92] wird ein ähnlicher Korrektheitsbegriff für parametrisierte Spezifikationen benutzt, der ebenfalls die hier benutzte Implementierung fordert. Die Pushout-Konstruktion garantiert dort die Konstruktion einer korrekten parametrisierten Spezifikation, da sie so gestaltet ist, daß sie die hier explizit geforderte Signaturinklusion $sig(SP'_R) \subseteq sig((\lambda X : SP_I.\ SP_R)(SP'_R|_\rho))$ durch Konstruktion sicherstellt (vgl. Abschnitt 2 (Pushout-Definition), Definition 4.2 und Proposition 4.4 in [Hen92]). Eine Einschränkung auf die in Satz 9.2.2 angegebene Signaturinklusion führt zum gleichen Ziel, macht die notwendige syntaktische Einschränkung an dieser Stelle aber explizit. Auf diese Einschränkung schon bei Definition der Komposition haben wir verzichtet, um bei Bedarf hier mehr Flexibilität in der Anpassung eines aktuellen an einen formalen Parameter zu gestatten. Mit obiger Zusatzforderung entsprechen sich die Kompositionsdefinitionen von Hennicker und dieses Ansatzes.

9.2.3 Implementierung von parametrisierten Spezifikationen

Die Implementierungsrelation $\rightsquigarrow$ ist bisher nur für primitive Spezifikationen definiert, sie soll nun auf parametrisierte Spezifikationen erweitert werden.

Definition 9.2.9 *Seien* $SP = \; < SP_I, SP_R >$ *und* $IMPL = \; < IMPL_I, IMPL_R >$ *korrekte parametrisierte Spezifikationen.* $< IMPL_I, IMPL_R >$ *ist eine* **Implementierung** *von* $< SP_I, SP_R >$ *oder*

$$< SP_I, SP_R > \;\rightsquigarrow_{par}\; < IMPL_I, IMPL_R >$$

falls gilt:

1. $sig(IMPL_R) = sig(SP_R)$ *und* $Mod(IMPL_R) \subseteq Mod(SP_R)$*, d.h.* $SP_R \rightsquigarrow IMPL_R$*,*
2. $sig(IMPL_I) = sig(SP_I)$ *und* $Mod(SP_I) \subseteq Mod(IMPL_I)$*, d.h.* $IMPL_I \rightsquigarrow SP_I$*.*

Dies läßt sich wie folgt veranschaulichen (gleiche Signaturen vorausgesetzt):

$$\begin{array}{ccc} < SP_I, SP_R > & & < SP_I \;\;,\;\; SP_R > \\ \wr\!\!\downarrow^{par} & \text{falls} & \wr\!\!\uparrow \qquad \wr\!\!\downarrow \\ < IMPL_I, IMPL_R > & & < IMPL_I, IMPL_R > \end{array}$$

Falls keine Mehrdeutigkeiten entstehen, wird auch $\rightsquigarrow$ statt $\rightsquigarrow_{par}$ für parametrisierte Spezifikationen benutzt. $\triangleright_{par}$ könnte analog zu $\rightsquigarrow_{par}$ definiert werden.

Die zweite Bedingung aus Definition 9.2.9 sagt aus, daß die implementierte Spezifikation $IMPL$ eine weniger restriktive Importschnittstelle als die abstrakte Spezifikation haben muß. Auf der anderen Seite müssen die Leistungen der Implementierung in bezug auf die Rumpfspezifikation besser sein als die von SP. Hier ergibt sich also eine Analogie zur Definition der Operationsimplementierung $\triangleright^{op}$, bei der die Vorbedingung der Implementierung ebenfalls weniger restriktiv sein muß und die Implementierung (der Effekt) mehr leisten muß.

9.2.4 Horizontale Kompositionseigenschaft

Der folgende Satz formuliert das zentrale Ergebnis dieses Abschnitts, also die horizontale Kompositionseigenschaft für parametrisierte Spezifikationen.

Satz 9.2.3 *(Horizontale Komposition für parametrisierte Spezifikationen)*
Seien $< SP_I^1, SP_R^1 >$, $< IMPL_I^1, IMPL_R^1 >$, $< SP_I^2, SP_R^2 >$ *und* $< IMPL_I^2, IMPL_R^2 >$ *korrekte Spezifikationen, sei* $\rho : sig(SP_I^1) \rightarrow sig(SP_R^2)$ *ein Signaturmorphismus, gelte*

$$< SP_I^1, SP_R^1 > \;\rightsquigarrow_{par}\; < IMPL_I^1, IMPL_R^1 > \quad \textit{und} \quad < SP_I^2, SP_R^2 > \;\rightsquigarrow_{par}\; < IMPL_I^2, IMPL_R^2 >$$

und sei die Komposition

$$\textit{compose} < SP_I^1, SP_R^1 > \textit{with} < SP_I^2, SP_R^2 > \textit{via } \rho$$

korrekt.

Dann gilt:

a) $\rho' : sig(IMPL_I^1) \rightarrow sig(IMPL_R^2)$ *mit* $\rho' := \rho$ *ist ein Signaturmorphismus.*

b) *Die Komposition*

$$\textit{compose} < IMPL_I^1, IMPL_R^1 > \textit{with} < IMPL_I^2, IMPL_R^2 > \textit{via } \rho'$$

ist korrekt.

c) *Die Kompositionen stehen in der Implementierungsrelation:*

$$(compose < SP_I^1, SP_R^1 > with < SP_I^2, SP_R^2 > via\ \rho) \qquad \rightsquigarrow_{par}$$
$$(compose < IMPL_I^1, IMPL_R^1 > with < IMPL_I^2, IMPL_R^2 > via\ \rho')$$

Beweis: Dieser Beweis orientiert sich zum Teil an *Proposition 4.3* aus [Hen91a].

a) Da die Implementierungsrelationen der Voraussetzung gelten, gilt $sig(SP_I^1) = sig(IMPL_I^1)$ und $sig(SP_R^2) = sig(IMPL_R^2)$. Damit ist ρ' ein Signaturmorphismus, falls ρ einer ist.

b) Es soll gezeigt werden, daß die Komposition korrekt ist. Es muß somit

$$IMPL_I^1 \rightsquigarrow IMPL_R^2|_{\rho'}$$

gelten. Nach Voraussetzung gelten

$$SP_I^1 \rightsquigarrow SP_R^2|_\rho,\ IMPL_I^1 \rightsquigarrow SP_I^1,\ SP_R^2 \rightsquigarrow IMPL_R^2$$

Wegen der Transitivität von $\rightsquigarrow$ und $\rho = \rho'$ (und damit $SP_R^2|_{\rho'} \rightsquigarrow IMPL_R^2|_{\rho'}$) gilt die Behauptung.

c) Die Implementierungsrelation $\rightsquigarrow_{par}$ gilt nach Definition 9.2.7 und 9.2.9, falls:

1. $sig((\lambda X : SP_I^1.\ SP_R^1)(SP_R^2|_\rho)) = sig((\lambda X : IMPL_I^1.\ IMPL_R^1)(IMPL_R^2|_{\rho'}))$ und
2. $Mod((\lambda X : IMPL_I^1.\ IMPL_R^1)(IMPL_R^2|_{\rho'})) \subseteq Mod((\lambda X : SP_I^1.\ SP_R^1)(SP_R^2|_\rho))$,
3. $sig(IMPL_I^2) = sig(SP_I^2)$ und
4. $Mod(SP_I^2) \subseteq Mod(IMPL_I^2)$.

Die Gültigkeit der vier Forderungen kann wie folgt gezeigt werden:

1. Erfüllt, da aus

$$< SP_I^1, SP_R^1 > \rightsquigarrow_{par} < IMPL_I^1, IMPL_R^1 >$$

die Signaturgleichheit

$$sig(SP_R^1) = sig(IMPL_R^1)$$

folgt (es gilt $sig((\lambda X : SP_I.\ SP_R)(SP_R'|_\rho)) = sig(SP_R)$ nach Definition).

2. Die Inklusion

$$Mod((\lambda X : IMPL_I^1.\ IMPL_R^1)(IMPL_R^2|_{\rho'})) \subseteq Mod((\lambda X : SP_I^1.\ SP_R^1)(SP_R^2|_\rho))$$

bzw.

$$Mod(IMPL_R^1(env[X \mapsto IMPL_R^2|_{\rho'}])) \subseteq Mod(SP_R^1(env[X \mapsto SP_R^2|_\rho]))$$

gilt, da wegen der vorausgesetzten Implementierungsrelationen $\rightsquigarrow^{par}$

$$Mod(IMPL_R^1) \subseteq Mod(SP_R^1) \text{ und } Mod(IMPL_R^2) \subseteq Mod(SP_R^2)$$

gelten (siehe Definition 9.2.2) und der λ-Ausdruck eine Spezifikationsfunktion definiert, also monoton bzgl. $\subseteq$ ist (nach Satz 9.2.1).

3. Analog 1.
4. Gilt nach Voraussetzung.

□

Die horizontale Kompositionseigenschaft (Ausschnitt Satz 9.2.3.b)) kann wie folgt veranschaulicht werden:

Gelte

$$\begin{array}{c} < SP_I^1, SP_R^1 > \\ \Big\downarrow^{par} \\ < IMPL_I^1, IMPL_R^1 > \end{array} \quad \text{und} \quad \begin{array}{c} < SP_I^2, SP_R^2 > \\ \Big\downarrow^{par} \\ < IMPL_I^2, IMPL_R^2 > \end{array}$$

und sei

$$compose \ < SP_I^1, SP_R^1 > \ with \ < SP_I^2, SP_R^2 > \ via \ \rho$$

korrekt. Dann ist

$$compose \ < IMPL_I^1, IMPL_R^1 > \ with \ < IMPL_I^2, IMPL_R^2 > \ via \ \rho$$

korrekt.

Wenn die *horizontale Kompositionseigenschaft* gilt, können separat entwickelte Implementierungen problemlos zu global korrekten Systemen zusammengesetzt werden. Die Spezifikationen

$$IMPL^1 = < IMPL_I^1, IMPL_R^1 > \quad \text{und} \quad IMPL^2 = < IMPL_I^2, IMPL_R^2 >$$

werden als Implementierungen zu den Spezifikationen SP^1 und SP^2 unabhängig voneinander entwickelt. Falls die abstrakteren Spezifikationen SP^1 und SP^2 korrekt zusammengesetzt werden können, ist dies automatisch auch für deren Implementierungen möglich.

9.2.5 Anmerkungen zur Methodik

Es gibt zwei unterschiedliche Verwendungsmöglichkeiten für importierte Spezifikationen im Rumpf einer parametrisierten Spezifikation:

- Die Importe sind grundlegende Datentypen oder Spezifikationen eines Subobjekts (Subkomponente). In beiden Fällen ist nur die Funktionalität des Imports, nicht aber dessen Realisierung von Bedeutung. Hier sollte eine Exportschnittstelle[4] importiert werden, falls ein solches Schnittstellenkonzept in der gegebenen Spezifikationssprache realisiert ist. Dies ist die klassische Benutztbeziehung.

- Die importierte Spezifikation soll direkt weiterentwickelt werden, etwa durch Redefinition von Operationen oder Hinzunahme weiterer Elemente. Bei zustandsbasierten Typen ist dann auch die interne Realisierung des Zustands von Bedeutung, da weitere Operationen direkt auf diesen zugreifen können. Hier sollte eine Rumpfspezifikation importiert werden. Diese Form der Anwendung ist auch durch ein Vererbungskonstrukt darstellbar.

[4] Diese wird später in Abschnitt 9.3.1 noch formal definiert.

Es gibt nur eine Form einer Schnittstelle für parametrisierte Spezifikationen. Die formale Definition der Importrelation erlaubt beide Fälle, es können beide Verwendungsmöglichkeiten modelliert werden. Die zwei Fälle sollen an schematisierten Beispielen erläutert werden.

Beispiel 9.2.1 Vererbung

$Nat \equiv < Int, Nat\_by\_Int >$ mit
  spec Nat_by_Int is
    enrich X by
    sorts
    state_def
    opns
    axioms $\forall i : int.\ i > 0$
  end spec

Int ist der formale Parameter (die Importspezifikation) und soll Eigenschaften ganzer Zahlen fordern. Hier wird ein Abkömmling von Int erzeugt. Nat ist eine Spezialisierung auf natürliche Zahlen, da die Modifikation durch Wertebereichseinschränkung hier über eine Invariante erreicht wird. Dies ist ein Vorgehen, wie es auch in einer Vererbungsbeziehung ausgedrückt wird. Es wird die importierte Spezifikation redefiniert oder verfeinert (vgl. Kapitel 10).

Mit dem Begriff der Vererbung ist ebenfalls der Aspekt der Ersetzbarkeit von Objekten durch Spezialisierungen umfaßt. In Abschnitt 4.3.3 wurde dieser Aspekt unter dem Stichwort *Polymorphie* betrachtet. Ein Modell einer Spezifikation kann immer durch ein Modell einer davon erbenden, also einer spezialisierenden Spezifikation ersetzt werden. Dieser Aspekt wird in Kapitel 10 betrachtet. Dort wird gezeigt, daß die Ersetzbarkeit von Objekten zwischen formalem und aktuellem Import besteht.

Beispiel 9.2.2 Benutzung

$Complex \equiv < Int, Complex\_by\_Int >$ mit
  spec Complex_by_Int is
    enrich X by
    sorts cmplx
    state_def
    opns make_cmplx $: int \times int \rightarrow cmplx$
    axioms
  end spec

Int ist wie im Vererbungsbeispiel eine Parameterrestriktion. Hier wird durch Int ein unterstützender Typ importiert, auf dem ein neuer konstruiert werden soll (etwa $cmplx = int \times int$ *als Repräsentation komplexer Zahlen durch ein Paar ganzer Zahlen). Der Import realisiert hier eine der in Complex verwendeten Sorten. Int wird benutzt.*

In beiden Fällen ist die formale Korrektheitsbedingung der parametrisierten Spezifikation mächtig genug, diese Situationen modellieren zu können. Die Kombination aus Implementierungsrelation und Reduktbildung ermöglicht dies. Mit dem Vererbungsbeispiel soll andeutet werden, daß solche Aspekte auch über die Komposition dargestellt werden können. Bei Vorhandensein einer Vererbungsrelation sollte dies natürlich sinnvollerweise über diese Relation explizit ausgedrückt werden. Da die Benutzung auf jeden Fall eine sinnvolle Beziehung ist, soll sie noch als Relation explizit gemacht werden.

Definition 9.2.10 *Sei compose* $< SP_I, SP_R >$ *with* $< SP_I', SP_R' >$ *via* ρ *eine korrekte Komposition. Man sagt dann* SP_R **benutzt** SP_R'.

Bemerkung 9.2.3 *Die beiden Beispiele betrachten nur funktionale Datentypen, d.h. Spezifikationen, die zustandslos sind. Die Anmerkungen sind aber auch auf objektwertige Spezifikationen (in Rumpf und Import) übertragbar. Eine Erweiterung eines zustandsbasierten Imports ist möglich, ebenso wie die unterstützende Nutzung eines zustandsbasierten Imports etwa als Sorte einer Zustandskomponente (zur Bildung von Objekthierarchien).*

9.3 Module

Exportschnittstellen für Module unterstützen das Geheimnisprinzip, d.h. Implementierungsdetails sollen für den Benutzer eines Moduls unsichtbar bleiben. Er hat nur über die exportierten Operationen Zugriff auf ein Objekt. Dadurch kann eine konsistenzerhaltende Benutzung des Objekts gewährleistet werden.

Zur Realisierung eines Exportschnittstellenkonzeptes in einem formalen Spezifikationsansatz gibt es zwei Alternativen:

1. *Syntaktischer Export*: Es wird eine Auswahl von Operationssignaturen (Namen und Parametersorten) exportiert.
2. *Semantischer Export*: Es werden neben den Operationssignaturen auch semantische Informationen über die Operationen (und ggf. auch über Invarianten) bereitgestellt. Die semantische Beschreibbarkeit von Schnittstellen war eines der Kernkonzepte des in Kapitel 1.4.2 vorgestellten Komponentenmodells.

Für beide Formen muß ein geeigneter *Korrektheitsbegriff*, etwa analog zur parametrisierten Spezifikation, entwickelt werden:

1. *Syntaktischer Export*: Korrektheit kann hier über einen geeigneten Signaturmorphismus definiert werden. Subsignaturbildung ist hier eine geeignete Basis der Definition.
2. *Semantischer Export*: Hier ist der Implementierungsbegriff anwendbar. Die Exportschnittstelle ist eine gegenüber dem Rumpf abstraktere Beschreibung, die ggf. auf semantisch höherem Niveau beschreibt, auf jeden Fall aber über Implementierungsdetails abstrahiert. Der Rumpf sollte die Anforderungen der Schnittstelle erfüllen. Eine Exportschnittstelle kann also auch als Anforderungsdefinition an eine Rumpfspezifikation gesehen werden. Jedes Modell des Rumpfes wäre dann Modell der Schnittstelle (nach geeigneter Anpassung der Signaturen). Der Rumpf würde somit eine Implementierung der Exportschnittstelle sein.

Der syntaktische Export kann durch den schon beschriebenen Spezifikationsoperator *export* (oder in der eingeschränkteren Form des *restrict*) realisiert werden (siehe Abschnitt 9.1). Daher wird im folgenden hier der semantische Export untersucht, der im Sinne des Komponentenmodells der sinnvollere ist. Die semantischen Grundlagen der Abschnitte 9.1 und 9.2 werden hier angewendet, um ein Modulkonstrukt im Sinne des Komponentenmodells nach [Goe93] zu realisieren. Ein Modulkonstrukt mit formalem Import und semantisch beschreibbarer Exportschnittstelle ist in [EM90] beschrieben. In *Section 3A* wird dort eine Modulkomposition definiert. In [Hen91a, Hen92] wird dieses Konstrukt nicht bereitgestellt. In [EM90] werden noch weitere Operationen auf Modulen definiert. Da wir hier Komposition als das zentrale Konstrukt in einem komponentenorientierten Ansatz identifiziert haben, soll nur dieses betrachtet werden.

Als wesentliche Einschränkung in der Formulierbarkeit von Aussagen im Export gegenüber dem Rumpf ist es notwendig, keine Zustandsbezeichner zu verwenden, da diese zum zu verbergenden Teil einer Spezifikation gehören. Operationen können in der Exportspezifikation also nur auf den Eigenschaften grundlegenderer Operationen spezifiziert werden, aber nicht mehr direkt über ihren Effekt auf den Zustand.

Es wird nur *eine* Spezifikation exportiert. Die vordefinierten Typen ($int, bool, ..$) können zur Spezifikation der Schnittstelle benutzt werden, ohne daß sie explizit importiert werden müssen. Hier unterscheidet sich der Ansatz von Sprachen wie ACT TWO [EM90] oder Π [GS94]. In diesen Sprachen werden mehrere Spezifikationen (im Sinne von Typen) explizit exportiert. Vordefinierte Typen gibt es in diesen Sprachen (mit Ausnahme des Typs *bool*) nicht.

9.3.1 Exportschnittstellen und Module

Es soll in diesem Abschnitt beschrieben werden, wie aus einer Operationsspezifikation im Rumpf eine geeignete abstraktere Exportspezifikation konstruiert werden kann. Die Operationsspezifikation kann dabei sowohl imperativ als auch axiomatisch vorliegen. Sei beispielsweise

$$z = X \rightarrow [p(\ldots)]\ z = e(X)$$

eine Rumpfspezifikation für eine Operation p, in der z eine Zustandskomponente, X eine Spezifikationsvariable und e einen beliebigen Ausdruck (Σ-Term) auf X bezeichnet. Dann ist eine geeignete Abstraktion über dem direkten Zugriff auf z durch das Attribut

$$read\_z() = X \rightarrow [p(\ldots)]\ read\_z() = e(X)$$

gegeben. *read_z* ist dabei ein Attribut, welches direkt die Zustandskomponente z ausliest (spezifizierbar durch die Invariante $read\_z() = z$). Die direkte Benutzung einer Zustandskomponente in einer Schnittstellenspezifikation ist nicht mehr möglich. Ein Attribut muß statt dessen benutzt werden. Dieses kann durchaus auch berechnend sein, d.h. das Attribut kann auf dem Wert von z noch Berechnungen durchführen. *read_z* muß exportiert werden.

Bemerkung 9.3.1 *Die Verwendung von z auch in der Schnittstellenspezifikation ist formal definierbar, soll aber aus methodischen Gründen nicht erlaubt werden. Eine Schnittstelle soll keinerlei Realisierungsdetails offenbaren.*

Definition 9.3.1 *Eine* **Exportschnittstelle** *ist ein Paar* $< SP_R, SP_E >$ *mit einer Rumpfspezifikation* SP_R. SP_E *wird Exportspezifikation genannt.* SP_R *und* SP_E *müssen Spezifikationsausdrücke sein.*

Wie schon in Abschnitt 9.2.1 erläutert, können in einer Rumpfspezifikation SP_R freie Variablen von umgebenden parametrisierten Spezifikationen benutzt werden, die zur Anwendung etwa der Implementierungsrelation durch geeignete Spezifikationsausdrücke ersetzt werden müssen. Da der Export auf dem Rumpf konstruiert wird, müssen auch für den Export ggf. die für den Rumpf erforderlichen Ersetzungen durchgeführt werden. Diese Ersetzungen sollen in Anwendung von Bemerkung 9.2.2 als implizit durchgeführt angenommen werden.

Definition 9.3.2 *Die Exportschnittstelle* $< SP_R, SP_E >$ *heißt* **korrekt**, *falls*

1. $SP_E \rightsquigarrow SP_R|_{sig(SP_E)}$ *und*
2. $state\_comp(sig(SP_E)) = \emptyset$

Die Signatur der Exportspezifikation muß Subsignatur der Rumpfspezifikation sein ($sig(SP_E) \subseteq sig(SP_R)$).

Definition 9.3.3 SP_E *heißt* **Export von** SP_R, *falls* $< SP_R, SP_E >$ *eine korrekte Exportschnittstelle ist.*

Die Exportrelation impliziert somit nach Definition 9.3.2 die Implementierungsrelation.

Definition 9.3.4 *Ein* **Modul** *ist ein Tripel* $< SP_I, SP_R, SP_E >$ *aus Import- und Exportspezifikation* SP_I *und* SP_E *und einer Rumpfspezifikation* SP_R. SP_I, SP_R, SP_E *müssen Spezifikationsausdrücke sein.*

Definition 9.3.5 *Ein Modul* $< SP_I, SP_R, SP_E >$ *heißt* **korrekt**, *falls*

1. $< SP_I, SP_R >$ *eine korrekte parametrisierte Spezifikation ist und*
2. $< SP_R, SP_E >$ *eine korrekte Exportschnittstelle ist.*

Definition 9.3.6 *Seien* $SP = < SP_I, SP_R, SP_E >$ *und* $SP' = < SP'_I, SP'_R, SP'_E >$ *korrekte Module. Die* **Komposition** *von Modulen sei durch*

$$compose\ < SP_I, SP_R, SP_E >\ with\ < SP'_I, SP'_R, SP'_E >\ via\ \rho\ :=$$

$$< SP'_I, (\lambda X : SP_I.\ SP_R)((SP'_R|_{sig(SP'_E)})|_\rho), SP_E >$$

definiert, falls $\rho : sig(SP_I) \to sig(SP'_E)$ *ein Signaturmorphismus ist.*

Für die Komposition von Modulen (siehe auch Abbildung 9.5) gilt $sig(SP'_E) \subseteq sig(SP'_R)$. Mit $\rho : sig(SP_I) \to sig(SP'_E)$ ist $(SP'_R|_{sig(SP'_E)})|_\rho$ eine korrekte Reduktbildung. $SP'_R|_{sig(SP'_E)}$ reduziert SP'_R auf die exportierten Elemente, die dann mit ρ an die Anforderungen des formalen Imports SP'_I angepaßt werden. Import des zusammengesetzten Moduls ist der Import des

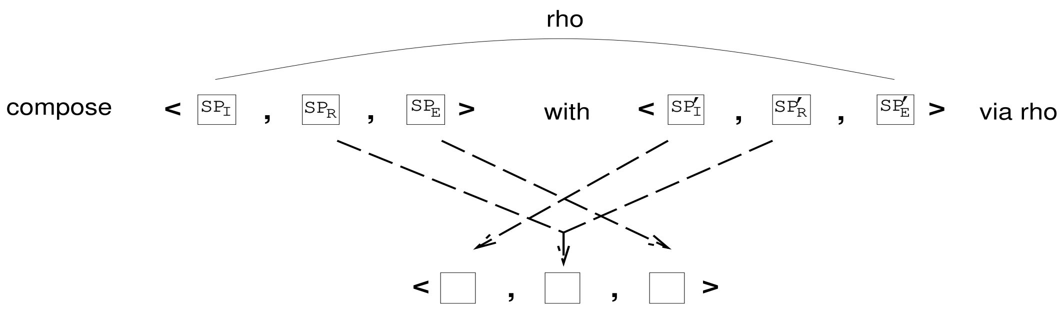


Abbildung 9.5: Komposition von Modulen

Aktualisierungsmoduls. Export des zusammengesetzten Moduls ist der Export des aktualisierenden Moduls. Der Rumpf wird aus den Rümpfen beider Ausgangsmodule zusammengesetzt.

Da der Rumpf eines korrekten Moduls (hier SP'_R) eine Implementierung des Exports ist, also auf keinen Fall schlechtere Eigenschaften hat, kann er somit problemlos zur Definition des Rumpfes der Komposition herangezogen werden.

Definition 9.3.7 *Die Komposition*

$$compose \;\; < SP_I, SP_R, SP_E > \;\; with \;\; < SP'_I, SP'_R, SP'_E > \;\; via \; \rho$$

heißt **korrekt***, falls*

$$SP_I \rightsquigarrow SP'_E|_{sig(SP_I)} \;.$$

Der Satz 9.2.2 (Erhaltung der Korrektheit bei Komposition) kann auch auf Module übertragen werden.

Satz 9.3.1 *Falls* $< SP_I, SP_R, SP_E >$ *und* $< SP'_I, SP'_R, SP'_E >$ *korrekte Module sind und*

$$compose \;\; < SP_I, SP_R, SP_E > \;\; with \;\; < SP'_I, SP'_R, SP'_E > \;\; via \; \rho$$

eine korrekte Komposition mit $sig(SP'_R) \subseteq sig((\lambda X : SP_I.\; SP_R)((SP'_R|_{sig(SP'_E)})|_\rho))$ *ist, so wird durch den Kompositionsausdruck Modul definiert.*

Beweis: Nach Definition 9.3.5 ist zu zeigen:

1. $< SP'_I, (\lambda X : SP_I.\; SP_R)((SP'_R|_{sig(SP'_E)})|_\rho) >$ ist eine korrekte parametrisierte Spezifikation und
2. $< (\lambda X : SP_I.\; SP_R)((SP'_R|_{sig(SP'_E)})|_\rho), SP_E >$ ist eine korrekte Exportschnittstelle.

Zu 1. ist zu zeigen:

$$Mod(SP'_I) \subseteq Mod((\lambda X : SP_I.\; SP_R)((SP'_R|_{sig(SP'_E)})|_\rho))|_{sig(SP'_I)}$$

Mit $\rho : sig(SP_I) \to sig(SP'_E)$ und $sig(SP'_E) \subseteq sig(SP'_R)$ ist die Reduktbildung $(SP'_R|_{sig(SP'_E)})|_\rho$ möglich. Damit kann analog Satz 9.2.2 argumentiert werden.

Zu 2. ist zu zeigen:

a) $Mod((\lambda X : SP_I.\ SP_R)((SP'_R|_{sig(SP'_E)})|_\rho))|_{sig(SP_E)} \subseteq Mod(SP_E)$
b) $state\_comp(sig(SP_E)) = \emptyset$

Es gilt $sig(SP_E) \subseteq sig(SP_R)$ nach Voraussetzung. Damit ist auch

$$sig(SP_E) \subseteq sig(SP_R) \subseteq sig((\lambda X : SP_I.\ SP_R)((SP'_R|_{sig(SP'_E)})|_\rho),$$

da nach Konstruktion SP_R syntaktisch Bestandteil des λ-Ausdrucks ist. Mit der Voraussetzung

$$Mod(SP_R)|_{sig(SP_E)} \subseteq Mod(SP_E)$$

folgt die Forderung 2.a).

2.b) gilt nach Voraussetzung. □

In [EM90] Section 3A Definition 3.1[5] wird die *Komposition* zweier Module

$$M_1 = < IMP_1, BODY_1, EXP_1 > \quad \text{und} \quad M_2 = < IMP_2, BODY_2, EXP_2 >$$

als ein Modul

$$M_3 = < IMP_2, PO(BODY_1, BODY_2), EXP_1 >$$

definiert, wobei der Export EXP_2 von M_2 den Import IMP_1 von M_1 unter Zuhilfenahme eines geeigneten Morphismus aktualisieren soll. $PO(BODY_1, BODY_2)$ ist die Pushout-Konstruktion auf den Rümpfen $BODY_1$ und $BODY_2$. Damit entsprechen sich die Kompositionsdefinitionen nach [EM90] und Definition 9.3.6 (vgl. Anmerkungen zu Pushouts in Abschnitt 9.2.2).

9.3.2 Implementierung von Modulen

Eine Erweiterung der Relation $\rightsquigarrow$ für einfache (bzw. $\rightsquigarrow^{par}$ für parametrisierte) Spezifikationen auf Module soll nun vorgestellt werden.

Definition 9.3.8 *Die* **Implementierung** *von Modulen*

$$< SP_I, SP_R, SP_E > \rightsquigarrow_{mod} < SP'_I, SP'_R, SP'_E >$$

gilt, falls

$$sig(SP_I) = sig(SP'_I) \ \wedge\ sig(SP_R) = sig(SP'_R) \ \wedge\ sig(SP_E) = sig(SP'_E) \ \wedge$$

$$SP'_I \rightsquigarrow SP_I \ \wedge\ SP_R \rightsquigarrow SP'_R \ \wedge\ SP_E \rightsquigarrow SP'_E$$

$\rhd_{mod}$ könnte analog zu $\rightsquigarrow_{mod}$ realisiert werden.

[5] Module in [EM90] enthalten noch einen Teil PAR (die Parameterspezifikation). Dieser wird hier nicht betrachtet.

9.3.3 Horizontale Kompositionseigenschaft

Es soll nun gezeigt werden, daß Module die Implementierungsrelation erhalten, d.h. daß auch für sie die horizontale Kompositionseigenschaft gilt.

Satz 9.3.2 *(Horizontale Kompositionseigenschaft für Module) Seien* $< SP_I^1, SP_R^1, SP_E^1 >$, $< IMPL_I^1, IMPL_R^1, IMPL_E^1 >$, $< SP_I^2, SP_R^2, SP_E^2 >$ *und* $< IMPL_I^2, IMPL_R^2, IMPL_E^2 >$ *korrekte Module, sei* $\rho : sig(SP_I^1) \to sig(SP_E^2)$ *ein Signaturmorphismus, gelte*

$$< SP_I^1, SP_R^1, SP_E^1 > \rightsquigarrow_{mod} < IMPL_I^1, IMPL_R^1, IMPL_E^1 >$$

und

$$< SP_I^2, SP_R^2, SP_E^2 > \rightsquigarrow_{mod} < IMPL_I^2, IMPL_R^2, IMPL_E^2 >$$

und sei

$$compose < SP_I^1, SP_R^1, SP_E^1 > with < SP_I^2, SP_R^2, SP_E^2 > via\ \rho$$

eine korrekte Komposition.

Dann gilt:

a) $\rho' : sig(IMPL_I^1) \to sig(IMPL_E^2)$ *mit* $\rho' := \rho$ *ist ein Signaturmorphismus,*

b)

$$compose < IMPL_I^1, IMPL_R^1, IMPL_E^1 > with < IMPL_I^2, IMPL_R^2, IMPL_E^2 > via\ \rho'$$

ist eine korrekte Komposition,

c)

$$(compose < SP_I^1, SP_R^1, SP_E^1 > with < SP_I^2, SP_R^2, SP_E^2 > via\ \rho) \quad \rightsquigarrow_{mod}$$
$$(compose < IMPL_I^1, IMPL_R^1, IMPL_E^1 > with < IMPL_I^2, IMPL_R^2, IMPL_E^2 > via\ \rho')$$

Beweis: (Vgl. auch Beweis zu Satz 9.2.3)

a) Mit den vorausgesetzten Signaturgleichheiten trivial.

b) Zu zeigen ist $IMPL_I^1 \rightsquigarrow IMPL_E^2|_\rho$. Folgt aus den geltenden Implementierungen.

c) Zu zeigen ist nach Definition 9.3.8:

1. $sig((\lambda X : SP_I^1.\ SP_R^1)((SP_R^2|_{sig(SP_E^2)})|_\rho)) =$
   $sig((\lambda X : IMPL_I^1.\ IMPL_R^1)((IMPL_R^2|_{sig(IMPL_E^2)})|_{\rho'}))$ und
2. $Mod((\lambda X : IMPL_I^1.\ IMPL_R^1)((IMPL_R^2|_{sig(IMPL_E^2)})|_{\rho'})) \subseteq$
   $Mod((\lambda X : SP_I^1.\ SP_R^1)((SP_R^2|_{sig(SP_E^2)})|_\rho))$,
3. $sig(IMPL_I^2) = sig(SP_I^2)$ und
4. $Mod(SP_I^2) \subseteq Mod(IMPL_I^2)$,
5. $sig(IMPL_E^1) = sig(SP_E^1)$ und
6. $Mod(IMPL_E^1) \subseteq Mod(SP_E^1)$.

Die Punkte 1. bis 4. können analog Satz 9.2.3 gezeigt werden. 5. und 6. folgen nach Voraussetzung. □

9.4 Abschließende Bemerkungen zu Spezifikationsausdrücken

Die Sprache der Spezifikationsausdrücke basiert auf dem Spezifikationsansatz zustandsbasierter Systeme. Die Zuhilfenahme der vollständigen Halbordnung $(Spec_\perp, \sqsubseteq_{Spec})$ ermöglicht eine Argumentation auf einer Ebene, die unabhängig von der konkreten Spezifikation in Form von Axiomen ist und die statt dessen auf den Modellsemantiken arbeitet.

In diesem Kapitel sind Ideen umgesetzt und formalisiert, die die Basis vieler Spezifikationsansätze bilden [Wir90, Bre91, KST94, FJ92, GS94]. Hierzu gehört die Gewährleistung der horizontalen Kompositionseigenschaft. Die Komposition ist das wichtigste Konstrukt der hier definierten Sprache der Spezifikationsausdrücke. Die Auswahl der Spezifikationsoperatoren orientiert sich an R. Hennicker und F. Nickl [HN92], die diese als geeignet für einen Wiederverwendungsansatz vorschlagen. Spezifikationsausdrücke, die die Weiterentwicklung von Spezifikationen in horizontaler und vertikaler Richtung erlauben, sind essentiell für eine Spezifikationssprache. Daher sind entsprechende Konzepte in fast allen Ansätzen zur modularen Spezifikation enthalten. In VDM gibt es auf einem Beweissystem basierende Techniken, Z bietet ein Schemakalkül. Ein mächtiges Kompositionskonzept, welches getrennte Entwicklung erlaubt, ist in Π zu finden. Diese Konzepte sind auch hier umgesetzt. Formaler Import ist auch in COLD realisiert; die dort realisierte Definition über das $\lambda\pi$-Kalkül wurde hier übernommen.

Besonderen Wert wurde hier auf die Umsetzung des Komponentenmodells gelegt. Die Unterstützung semantischer Schnittstellen wurde realisiert. Da die Sprache Π auch die Anforderungen des Komponentenmodells erfüllt, sind hier Ähnlichkeiten festzustellen. Die wichtigsten Konstrukte von Π sind die CEMs, sie entsprechen (unter Vernachlässigung der Sichten) den Modulen. In CEMs gibt es zusätzlich noch *Common Parameters*, eine Abkürzung für die Schnittmenge aus Import und Export. Die Komposition ist ähnlich definiert. In beiden Ansätzen ist die Aktualisierung eines formalen Imports durch Umbenennung und Zuordnung einzelner Elemente von Komponenten möglich. Flexible und mächtige Kompositionskonstrukte, wie die beiden eben angesprochenen, sind eher selten in bestehenden Ansätzen zu finden (vgl. Abschnitt 1.5).

Unter dem Stichwort *specification matching* sind in der Literatur Ansätze zu finden, zwei Spezifikationen zu vergleichen. Es kann damit bestimmt werden, ob eine Spezifikation durch eine andere ersetzt werden kann. Dieses *Matching* kann ebenso im Retrieval bei Wiederverwendung und bei Benutzung von Komponentenbibliotheken benutzt werden, wie zur Feststellung, ob eine Spezifikation ein verhaltensäquivalenter Subtyp einer anderen Spezifikation ist. Matching wurde hier durch die Implementierungsrelation $\rightsquigarrow$ und durch $\triangleright$ formalisiert. Diese Relationen finden hier zur Definition partieller Korrektheit (Kapitel 8) ebenso Anwendung, wie zur Definition intramodularer Relationen (Import, Export) und intermodularer Relationen (Aktualisierung in der Komposition). An diesen Stellen ist zur Überprüfung der Korrektheit der Relationen ganz besonders eine Unterstützung durch Werkzeuge in einer Entwicklungsumgebung wichtig.

Amy Moormann Zaremski und Jeannette Wing zeigen in [ZW95] einen Ansatz, der Werkzeugunterstützung für Matching realisiert. Das dort benutzte Konzept ist das gleiche, das auch hier der Relation $\triangleright$ zugrundeliegt. Operationen werden durch Vor- und Nachbedingungen spezifiziert. Module werden als Kollektionen von Operationen angesehen. Die hier definierte Form $\triangleright$ wird dort als *Plug-in Match* bezeichnet. In [ZW95] werden noch weitere Matching-Formen definiert, die sich hier durch Modellklassengleichheit (*Exact Pre/Post Match*) oder

Abschwächung der Vorbedingungen (*Plug-in Post Match, Weak Post Match*) darstellen lassen. Der Ansatz von Moormann Zaremski und Wing ist im Kontext von Larch angesiedelt. Der Larch-Theorembeweiser LP (*Larch Prover*) kann zur Verifikation der Matching-Bedingungen eingesetzt werden. Die zu verifizierenden Bedingungen sind Formeln einer Prädikatenlogik erster Stufe. Eingaben für den Larch Prover können dort automatisch generiert werden, der Theorembeweiser wird dann interaktiv benutzt. Mit den dortigen Arbeiten ist gezeigt, daß mit dem gegenwärtigen Stand der Werkzeugtechnik eine Unterstützung der Relation $\triangleright$ in einer Entwicklungsumgebung möglich ist. Das Ziel der Konstruktivität in der Definition von Relationen ist also an dieser Stelle erreicht.

Das Matching im Kontext der Wiederverwendung wird auch von J. Cramer detailliert in seiner Dissertation beschrieben [Cra94]. Dort liegen ebenfalls formale Spezifikationstechniken zugrunde, die auf der algebraisch orientierten Typsicht von Π beruhen (vgl. Abschnitt 1.5.1). Es geht dort, wie auch in [ZW95], darum, das Matching in verschiedenen Graden von exakter Übereinstimmung bis zu ähnlichen Spezifikationen beschreiben zu können.

Die allgemeine Zielstellung eines möglichst konstruktiven Ansatzes hat also hier Anwendung gefunden. Alle Spezifikationsausdrücke und -operatoren sind syntaktisch auf Signaturmorphismen definiert. Semantische Einschränkungen werden durch Korrektheitsbegriffe formuliert. Dies ermöglicht eine weitgehende Unterstützung des Ansatzes durch Werkzeuge — eine unabdingbare Forderung für einen praktischen Einsatz einer Spezifikationssprache. Die Unterstützung der Verifikation der Korrektheit durch Werkzeuge ist theoretisch und technologisch schwierig. Daher wurden Mechanismen entworfen, korrektheitserhaltendes Entwickeln erlauben.

Weitere zusammenfassende Betrachtungen zu Relationen zwischen Spezifikationen sind am Ende von Kapitel 10 zu finden.

Kapitel 10

Objektorientierte Entwicklung

Eine Möglichkeit, die Wiederverwendung von Modulen zu unterstützen, besteht nach B. Meyer [Mey88, Mey92b, Mey92a] darin, *Abkömmlinge* eines Moduls zu erzeugen, die die Leistungen des Ausgangsmoduls weiter nutzen und lediglich einige Anpassungen (Erweiterungen oder Redefinitionen) vornehmen. Für Meyer ist die so zwischen zwei Modulen entstehende *Vererbungsrelation*, neben einer *Benutztbeziehung* (dort *Clientship* genannt), die wichtigste Relation zur Strukturierung von Software-Systemen. Ein Modul kann auch Leistungen mehrerer anderer Module erben. Dies wird als *Mehrfachvererbung* bezeichnet. Die Benutztbeziehung zwischen zwei Komponenten entsteht in dieser Arbeit, wenn mit der Komposition zwei Spezifikationen verknüpft werden (vgl. Kapitel 9). In [Mey88] wird benannter, d.h. nichtformaler Import vorgeschlagen. Vererbung und Benutzung sind Kernkonzepte der *objektorientierten Entwicklung*. Weitere Schlüsselkonzepte sind *Objekte*, *Klassendefinition*, *Abstraktion*, *Datenkapselung* und *Polymorphie*. Bis auf die Vererbung sind diese Konzepte in den vorangegangen Kapiteln betrachtet worden. In diesem Kapitel soll dargelegt werden, daß alle gängigen Konzepte der objektorientierten Entwicklung im vorliegenden Ansatz realisiert sind und somit eine Sprache, die dem objektorientierten Paradigma unterliegt, mit den hier vorgestellten Mitteln definiert werden kann.

Objektsysteme lassen sich in die folgenden drei Kategorien klassifizieren (nach [Weg90]):

- *objektbasiert*: reine zustandsbasierte Objekte.
- *klassenbasiert*: durch Klassen verkapselbare Objekte.
- *objektorientiert*: durch Vererbung in Beziehung setzbare Klassen.

Der vorliegende Ansatz ist im Kern objektbasiert. In Kapitel 9.3 wurde durch eine explizite Exportschnittstelle ein Verkapselungsaspekt hinzugenommen. Die Vererbungsbeziehung soll in diesem Kapitel untersucht werden. Mit der klassischen Vererbungsbeziehung zwischen Spezifikationen können zwei Aspekte beschrieben werden:

- *Wiederverwendung von Code*: Ein Implementierer ist daran interessiert, Spezifikationen in Form von Code zu übernehmen und syntaktisch zu modifizieren oder zu erweitern. Dieser Aspekt ist aber eher pragmatisch und nicht auf logischer Ebene zu sehen. Er wird hier nicht weiter untersucht.

- *Spezialisierung*: Für die Instanzebene sollen Kompatibilitätsregeln spezifiziert werden. Damit wird festgelegt, wann ein Objekt einer gegebenen Klasse durch ein Objekt einer spezialisierenden Klasse ersetzt werden kann. Diese Kompatibilität ist mit dem Verhalten von Objekten verknüpft.

Der Begriff der Spezialisierung ist mit dem des Subtypbegriffs verwandt. Subtypen werden in Abschnitt 10.1 untersucht. Eine Subtypbeziehung wird in der Regel direkt auf der Semantik der Sprache definiert. Eine Spezialisierungsbeziehung sollte explizit modellierbar sein, d.h. explizit durch den Spezifizierer vereinbar. Die Spezialisierung sollte somit die Subtypbeziehung implizieren. Einige Autoren (R. Breu [Bre91], Cook et.al. [CHC90]) schlagen auch eine Gleichwertigkeit beider Begriffe vor. Die Spezialisierung stellt Anforderungen an die Laufzeitkompatibilität von Objekten. In Abschnitt 10.2 wird dann Vererbung betrachtet. Das Kapitel schließt mit einer Zusammenfassung und abschließenden Betrachtung des gesamten Teils III dieser Arbeit.

10.1 Subtypbeziehung

In Kapitel 9 wurden Spezifikationsausdrücke und Relationen zwischen Spezifikationen betrachtet, wie sie für algebraische Spezifikationssprachen ebenfalls üblich sind. In diesem Abschnitt wird eine weitere Relation, die *Subtypbeziehung*, untersucht. Sie ist für die Gestaltung des Typsystems (insbesondere der Programmiersprache) von großer Bedeutung (siehe [CW85, DT88]).

Die Subtypbetrachtung soll sich also sowohl auf axiomatische als auch auf imperative Spezifikationen beziehen. Die Subtypbeziehung zwischen Spezifikationen sollte dann gelten, wenn ein Modell der einen Spezifikation durch Modelle der anderen Spezifikation in jedem Kontext ersetzt werden kann, ohne daß gegen geforderte Eigenschaften verstoßen wird. Dies unterscheidet sie von einer Spezialisierungsbeziehung, die die Ersetzbarkeit auf bestimmte, von Spezifizierer festgelegte Bedingungen einschränkt (siehe [Cas95]). Die Subtypbeziehung ist notwendig, um den Begriff der *Inklusionspolymorphie* (siehe [CW85]) formal erfassen zu können. Subtypen können wie Implementierungen als Verfeinerungen einer Ausgangsspezifikation in vertikaler Entwicklungsrichtung (falls Typ- und Spezifikationsbegriff gleichgesetzt werden) angesehen werden. Außerdem kann die Vererbungsbeziehung über die Subtypbeziehung definiert werden. Dieser Aspekt soll im anschließenden Abschnitt 10.2 untersucht werden.

In diesem Abschnitt werden Typ- und Spezifikationsbegriff gleichgesetzt. Ein geeigneter Subtypbegriff soll nun vorgestellt werden.

Modellklasseninklusion erfüllt die Forderung, daß ein Subtyp überall da einsetzbar sein muß, wo ein Supertyp gefordert ist. Bei Definition über die Implementierungsrelation ist jedes Modell eines Subtyps auch Modell eines Supertyps, es kann also immer anstelle eines solchen benutzt werden. Erweiterungen der Syntax (im Rahmen einer Einbettung der Signaturen) sind dabei möglich. Es ergibt sich für die Subtyprelation folgende Definition:

Definition 10.1.1 *Zwei Typen t_1 und t_2 (also zwei Spezifikationen t_1 und t_2) mit $sig(t_2) \subseteq sig(t_1)$ stehen in der* **Subtypbeziehung** $<$ *:*

$$t_1 < t_2 \quad gdw. \quad t_2 \rightsquigarrow t_1|_{sig(t_2)}$$

Die Reduktbildung basiert auf der Subsignaturbildung $sig(t_2) \subseteq sig(t_1)$ durch die Einbettung (vgl. Abschnitt 3.2.4). Zwei Spezifikationen, die in einer Implementierungsrelation stehen, stehen damit automatisch in der Subtypbeziehung.

Folgerung 10.1.1 *Seien t_1 und t_2 Spezifikationen mit $sig(t_1) = sig(t_2)$.*
Aus $t_1 \rightsquigarrow t_2$ folgt $t_2 < t_1$.

Beweis: Folgt direkt aus den Definitionen von $<$ und $\rightsquigarrow$. □

Diese Eigenschaft läßt sich fortsetzen, da die Implementierungsrelation logische Folge der Relation $\triangleright$ ist.

Folgerung 10.1.2 *Seien t_1 und t_2 Spezifikationen mit $sig(t_1) = sig(t_2)$.*
Aus $t_1 \triangleright t_2$ folgt $t_2 < t_1$.

Beweis: Folgt direkt aus obiger Folgerung und Satz 8.2.1. □

Bemerkung 10.1.1 *Ein Typ kann Subtyp mehrerer Supertypen sein.*

Der bisherige Ansatz nach Kapitel 9 modelliert eingeschränkten Polymorphismus (Subtyp- oder Inklusionspolymorphismus), da bei der Aktualisierung in der Komposition Freiheit bei der Auswahl eines geeigneten aktuellen Parameters im Rahmen der Implementierungsrelation gegeben ist. Da nicht nur die Freiheit innerhalb der Implementierungsrelation vorliegt, sondern auch mit den Signaturmorphismen eine Teilmengenbeziehung der Signaturen beschrieben werden kann, liegt bei der Komposition sogar eine Subtypbeziehung zwischen formalem und aktuellem Parameter vor (vgl. Abschnitt 9.2.5).

Folgerung 10.1.3 *Sei compose $< SP_I, SP_R >$ with $< SP'_I, SP'_R >$ via ρ eine korrekte Komposition. Dann gilt: $SP'_R < SP_I$.*

Beweis: Folgt aus der Definition der Komposition und der Subtypbeziehung. □

Ein vergleichbarer Ansatz zur Definition eines Inklusionspolymorphismus sind *order-sorted algebras* [GM87a, GM87b, GW90]. Sie basieren auf einer Subtypbeziehung auf den Datensorten. Ziel ihrer Einführung war die Integration von Vererbung in die algebraische Spezifikation. Mehrfachvererbung ist mit diesem Ansatz realisierbar. Mit dieser Ordnungbeziehung kann auch eine *bottom*-Sorte definiert werden, die zur Modellierung partieller Anwendungen herangezogen werden kann. Die folgende Definition erfolgt nach [GM87a]. Eine *order-sorted algebra* ist eine Algebra, deren Trägermengen bzgl. einer Ordnung $\leq$ eine Halbordnung bilden, und deren Operatoren auf den Trägermengen bzgl. $\leq$ monoton sind. Die Subsortenbeziehung wird semantisch als Teilmengenbeziehung auf den Trägermengen der Algebra interpretiert. $s \leq t$ für zwei Sorten s und t impliziert $A_s \subseteq A_t$ für die zugehörigen Trägermengen. Die Relation $\leq$ kann zur Definition einer Subtyp- oder Vererbungbeziehung herangezogen werden (vgl. Anmerkungen zum Ansatz von R. Breu in Abschnitt 10.2.6).

Auf weitere Ansätze zur Darstellung einer Subtypbeziehung wird im nächsten Abschnitt (10.2.6) eingegangen.

10.2 Objektorientierte Relationen

Zuerst wird hier die Einfachvererbung betrachtet, dann folgen Betrachtungen zur Erweiterung auf Module und ein Abschnitt zur Mehrfachvererbung. Daran anschließend werden noch einige besondere Vererbungkonzepte der objektorientierten Entwicklung vorgestellt.

10.2.1 Einfachvererbung

Wenn eine Spezifikation A von einer Spezifikation B *erbt*, kann sie alle von B zur Verfügung gestellten Dienste (Zustandskomponenten, Attribute und Prozeduren) nutzen, d.h. auch anderen diese Leistungen zur Verfügung stellen sowie neue Dienste hinzufügen und vorhandene — mit bestimmten Einschränkungen — redefinieren. Die Redefinition ist Einschränkungen unterworfen, falls die Prozeduren axiomatisch spezifiziert sind. Eine Prozedur p kann durch eine Prozedur q redefiniert werden, wenn q eine schwächere Vorbedingung *pre* (also $pre_p \rightarrow pre_q$) und eine stärkere Nachbedingung *post* (also $post_q \rightarrow post_p$) hat. Die Invarianten sind dabei zu berücksichtigen. Diese sind aber nichts anderes als syntaktische Abkürzungen, die in die Prozedurspezifikationen integriert werden können (siehe Abschnitt 8.1.2).

Definition 10.2.1 *Eine Spezifikation SP* **erbt von** *SP_0, falls*

- $SP_0 \rhd SP|_{sig(SP_0)}$ *und*
- $sig(SP_0) \subseteq sig(SP)$

Es dürfen bei Vererbung Operationen (Prozeduren und Attribute) und Zustandskomponenten hinzugenommen werden. Eine Operation wird durch die $\rhd^{op}$-Relation redefiniert (vgl. Abschnitt 8.1.1). Im Vergleich zur Relation $\rhd$ wurde hier eine Erweiterung der Signatur zugelassen. Damit entspricht die Vererbung der Eiffel-Definition [Mey92b].

Lemma 10.2.1 *Die Relation 'erbt von' ist transitiv, d.h. aus SP_1 erbt von SP_0 und SP_2 erbt von SP_1 folgt SP_2 erbt von SP_0 .*

Beweis: Es gelte SP_1 *erbt von* SP_0, somit gilt nach Definition $SP_0 \rhd SP_1|_{sig(SP_0)} \wedge sig(SP_0) \subseteq sig(SP_1)$ und SP_2 *erbt von* SP_1, also $SP_1 \rhd SP_2|_{sig(SP_1)} \wedge sig(SP_1) \subseteq sig(SP_2)$. Da $\rhd$ und $\subseteq$ transitiv sind (siehe Satz 8.3.2) und $SP_2|_{sig(SP_0)}$ eine korrekte Reduktbildung beschreibt ($sig(SP_0) \subseteq sig(SP_1) \subseteq sig(SP_2)$), ist damit SP_2 *erbt von* SP_0 korrekt, d.h. die Relation ist transitiv. □

Im folgenden sei vorausgesetzt, daß sich Axiome eindeutig den Prozeduren zuordnen lassen. Um dies zu erreichen, können die im Kapitel 7 beschriebenen Inferenzregeln wie AND, OR, DEMORGAN oder Lemma 8.2.1 benutzt werden. Daß immer nur ein Axiom pro Prozedur notwendig ist, wurde ebenfalls in Lemma 8.2.1 gezeigt. Es soll nun ein Spezifikationsoperator *inherit* definiert werden, der die Vererbungsrelation *erbt von* erhält. Relationen gelten unabhängig von der Benutzerspezifikation. Eine Vererbungsbeziehung zwischen zwei Spezifikationen soll aber in der Regel nur zwischen bestimmten, vom Benutzer explizit angegebenen Spezifikationen gelten. Daher wird ein Spezifikationsoperator definiert, mit dem die *erbt von*-Beziehung durch den Benutzer definiert werden kann.

Definition 10.2.2 *Sei* $sp = < sig(SP), Mod(SP) >$ *durch* $SP = < S, Z, OP, E >$ *spezifiziert. Es wird angenommen, daß jede Prozedur durch nur ein Axiom spezifiziert ist und die Invarianten den Prozeduren zugeordnet sind (siehe Abschnitt 8.2). Der Spezifikationsoperator* $inherit$ *mit*

$$inherit\ from\ SP\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 >$$

ist definiert durch:

1. $sp = \bot_{spec}$:
   $inherit\ from\ SP\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 > \ := \ \bot_{Spec}$
2. $sp \neq \bot_{spec}$:
   $sig(inherit\ from\ SP\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 >) \ :=$
   $< S \cup S_1, Z \cup Z_1, OP \cup OP_1 >$

   und

   $Mod(inherit\ from\ SP\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 >) :=$
   $\{O \in Obj(< S \cup S_1, Z \cup Z, OP \cup OP_1 >) \mid O \models E_1 \cup E_2 \cup (E \setminus E|_{OP_2}\ )\}$

$E|_{OP_2}$ *bezeichnet die Axiome aus* E*, die die Operationen in* OP_2 *betreffen, also die Axiome, die eine Prozedur* p *in der Form* $\phi \rightarrow [p(\ldots)]\ \psi$ *spezifizieren.*

Definition 10.2.3 *inherit heißt* **korrekt**, *falls*

- $SP \triangleright (inherit\ from\ SP\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 >)|_{sig(SP)}$
- $OP_1 \cap opns(sig(SP)) = \emptyset, OP_2 \subseteq opns(sig(SP))$

Redefinierte Dienste (OP_2) müssen schon vorhanden sein, Erweiterungen (OP_1) dürfen noch nicht vorhanden sein. Damit sind die Mengen der redefinierten Dienste und der Erweiterungen disjunkt ($OP_1 \cap OP_2 = \emptyset$).

Folgerung 10.2.1 *Falls ein durch inherit konstruierter Spezifikationsausdruck korrekt ist, erbt die neu konstruierte Spezifikation von* SP *(nach Definition 10.2.1).*

Beweis: Erster Spiegelpunkt in Definition 10.2.3 und Definition 10.2.1. □

Satz 10.2.1 *Es sei ein relativ vollständiges und konsistentes Beweissystem für die Spezifikationslogik angenommen. Dann definiert inherit eine Spezifikationsfunktion.*

Beweis: Sei sp_i das zum Spezifikationsausdruck SP_i gehörende Element aus $Spec_\bot$ für $i = 1, 2$. Seien als Abkürzungen definiert:

$$INH_1 \equiv inherit\ from\ SP_1\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 >$$
$$INH_2 \equiv inherit\ from\ SP_2\ extend\ < S_1, Z_1, OP_1, E_1 >\ redefine\ < OP_2, E_2 >$$

Es sind Striktheit und Monotonie zu zeigen.

- Striktheit: nach Definition.

- Monotonie: Zu zeigen ist (mit $sig(SP_1) = sig(SP_2)$):

  $$sp_1 \sqsubseteq_{Spec} sp_2 \quad \Rightarrow \quad < sig(INH_1), Mod(INH_1) > \sqsubseteq_{Spec} < sig(INH_2), Mod(INH_2) >$$

  Nach Voraussetzung gilt $Mod(SP_2) \subseteq Mod(SP_1)$, d.h. $E_{SP_2} \vdash E_{SP_1}$ gilt, da ein (relativ) vollständiges und konsistentes Beweissystem angenommen wurde (siehe Kapitel 7). Nach Konstruktion von *inherit* (Definition der Modellklasse) gilt damit

  $$Mod(INH_2) \subseteq Mod(INH_1)$$

  und somit auch

  $$INH_1 \sqsubseteq_{Spec} INH_2.$$

□

Der *inherit*-Operator etabliert die Relation *erbt von*. Es ist noch zu zeigen, daß die Vererbungsrelation die Subtypbeziehung impliziert.

Lemma 10.2.2 *Gelte SP_1 erbt von SP_0 . Dann folgt $SP_1 < SP_0$.*

Beweis: Folgt aus den Definitionen von *erbt von* und $<$ sowie Satz 8.2.1. □

10.2.2 Vererbung für parametrisierte Spezifikationen und Module

Die Definition der Vererbung für parametrisierte Spezifikationen oder Module soll nicht formal erfolgen. Die Konzeption eines Vererbungsbegriffes für parametrisierte Spezifikationen oder Module kann anhand von Ko- oder Kontravarianzdefinitionen erfolgen. Cardelli definiert in [Car84] eine *kontravariante* Subtypbeziehung $\leq$ für Funktionstypen wie folgt:

$$Es\ gilt\ S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2\ ,\ falls\ T_1 \leq S_1\ und\ S_2 \leq T_2.$$

Kontravarianz wird genutzt, um Subtypbeziehungen zu definieren (siehe [Cas95]). Eine Subtypbeziehung charakterisiert, welche Mengen von Funktionen eine gegebene Funktion in jeder beliebigen Situation ersetzen können. *Kovarianz* ist ein anderes Konzept, das die Spezialisierungsbeziehung zu formulieren gestattet. Hier wird charakterisiert, welcher neue Code alten Code in bestimmten Situationen ersetzen kann. Für Kovarianz muß gelten:

$$Es\ gilt\ S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2\ ,\ falls\ S_1 \leq T_1\ und\ S_2 \leq T_2.$$

Nach [Cas95] führt Kovarianz zu Typisierungsproblemen, ist aber einsichtiger.

Parametrisierung ist hier der Mechanismus zur Definition des formalen Imports. Es liegt also nahe, die Ideen der Ko- oder Kontravarianz hier anzuwenden. Die Implementierungsrelation für parametrisierte Spezifikationen (Definition 9.2.9) realisiert nach den obigen Begriffsdefinitionen also Kontravarianz, wenn eine Spezifikation $< SP_I, SP_R >$ als funktionaler Typ $SP_I \rightarrow SP_R$ aufgefaßt wird. Die Vererbung für parametrisierte Spezifikationen kann in gleicher Weise definiert werden.

Der Export ist eine Abstraktion des Rumpfes. Er sollte im Rahmen einer Vererbungsdefinition für Module wie der Rumpf behandelt werden.

Zu Vor- und Nachteilen von ko- bzw. kontravarianten Definitionen siehe [Cas95]. Ko- und Kontravarianz sind die üblichen Vorgehensweisen, Vererbung kompositional von einfachen Konstruktionen auf parametrisierte zu übertragen. Bei einer kontravarianten Definition könnte die horizontale Kompositionseigenschaft in bezug auf die Vererbungsrelation nachgewiesen werden. Der Beweis verliefe dann unter Anwendung von Satz 8.2.1 analog zur horizontalen Komposition in bezug auf die Implementierungsrelation (Sätze 9.2.3 und 9.3.2), es müssen dabei lediglich die für die Vererbung möglichen syntaktischen Erweiterungen betrachtet werden. Bei kovarianter Definition ist der Nachweis der Bedingung *b)* der Sätze 9.2.3 und 9.3.2 nur durch eine starke Restriktion der Vererbung erreichbar. Diese Problematik wird in [Cas95] unter dem Stichwort *Typisierung* betrachtet.

10.2.3 Mehrfachvererbung

Namenskonflikte erschweren die Anwendung von Mehrfachvererbung. Namenskonflikte (englisch *Name Clashes*) sollen hier deshalb Durch den *rename*-Operator können ggf. Probleme bezüglich der Operationsnamen umgangen werden. Ein Vereinigungsoperator $\bigcup$ auf Spezifikationen, der Signaturen und Axiome vereinigt, ist dann problemlos einsetzbar.

Definition 10.2.4 *Eine Spezifikation SP* **erbt mehrfach von** *den Spezifikationen SP_1, $\ldots$, SP_n, falls*

- *alle Namen von Operationssymbolen und Sortensymbolen aller $SP_i, i = 1, \ldots, n$ jeweils paarweise verschieden sind (also keine Namenskonflikte entstehen),*
- $(\bigcup_{i=1}^{n} SP_i) \triangleright SP|_{sig(\bigcup_{i=1}^{n} SP_i)}$,
- $sig(\bigcup_{i=1}^{n} SP_i) \subseteq sig(SP)$.

Lemma 10.2.3 *Die Relation 'erbt mehrfach von' ist transitiv.*

Beweis: Analog der Transitivität der Relation *erbt von* (Lemma 10.2.1), da $SP_0 = \bigcup_{i=1}^{n} SP_i$ gesetzt werden kann. $\square$

Die bei mehrfacher Vererbung auftretenden Probleme *wiederholtes Erben*, d.h. mehrfaches Erben vom selben Vorfahren (auch hier ist die Bedingung verletzt, daß bei Mehrfacherbung die Namen verschieden sein müssen), und *Namenskonflikte* (vgl. [Mey88]) können mit Umbenennung und Export behandelt werden.

10.2.4 Besondere Konzepte der Vererbung

Hier sollen Konzepte wie *Redefinition* oder *verzögerte* bzw. *virtuelle Klassen* vorgestellt werden. Die Konzepte stammen in der hier zugrundegelegten Form aus der objektorientierten *Programmierung* und sind in Sprachen wie Eiffel oder C++ realisiert.

Redefinition: Die Redefinition, wie sie in Eiffel realisiert ist, ist als Konzept in ihrer *bedingten* (d.h. durch semantische Bedingungen eingeschränkten) Form schon durch die Relation $\triangleright$ vorgestellt und formalisiert worden. In vielen objektorientierten Sprachen ist sie allerdings nur in der *unbedingten* Form (d.h. nur signaturbasiert ohne Berücksichtigung semantischer Eigenschaften) realisiert.

Verzögerte Klassen: Mit diesem Begriff (englisch *deferred classes*) werden von B. Meyer Klassen bezeichnet, in denen einzelne Operationen nicht ausformuliert sind. Diese Operationen werden dann *abstrakt* genannt. Lediglich Name und Parameterschnittstelle werden angegeben. Mangels Implementierungen sind solche Klassen nicht ausführbar. Mit diesem Mechanismus kann eine explizite Implementierung auf späteren Vererbungsebenen erzwungen werden. *Verzögerte Klassen* können also als Spezifikationen oder partielle Implementierungen angesehen werden. Sinnvoll ist ein solcher Ansatz dann, wenn das zu implementierende Verhalten einer Operation etwa durch Vor- und Nachbedingungen spezifiziert wird (siehe Eiffel [Mey92a]). Verzögerte Klassen werden auch als *abstrakte Klassen* bezeichnet.

Die *virtuellen Klassen* von C++ sind ein Spezialfall der *verzögerten Klassen*, in denen keinerlei Implementierung (d.h. kein Rumpf der Operationen) vorhanden ist.

Die hier vorgestellten Konzepte lassen sich durch das Vorhandensein der Spezifikationslogik und der Relationen zwischen Spezifikationen leicht im vorliegenden Ansatz darstellen.

10.2.5 Benutztbeziehung

Als *Clientship* bezeichnet B. Meyer eine Beziehung zwischen einem Modul A und anderen, die von A benutzt werden. Durch das Konzept des formalen Imports entsteht die *Benutztbeziehung* im Sinne von B. Meyer zwischen dem Rumpf einer parametrisierten Spezifikation und der Aktualisierung des formalen Parameters. Im Abschnitt 9.2.5 und in Definition 9.2.10 wird dieser Aspekt betrachtet.

10.2.6 Vergleichbare Ansätze

In [Bre91] werden zwei unterschiedliche Aspekte der Vererbung identifiziert. Der eine — auch mit dem Begriff *Vererbung* gekennzeichnet — basiert auf dem semantischen Konzept der *Modellverfeinerung*, d.h. Modelle eines Nachfahren sind Erweiterungen der Modelle des Vorfahrens. Die *Subtypbeziehung* (es wird auch von Subklassen gesprochen) basiert auf *Datenverfeinerung*. Hier liegt eine Teilmengenbeziehung der Trägermengen zugrunde, modelliert durch ordnungssortierte Algebren. Hierdurch wird Inklusionspolymorphismus realisiert. Diese Sicht der Vererbung basiert auf den Ideen von Cook et.al. [CHC90]. Die Vererbung gilt in [Bre91] (sp erbt von sp'), falls $\sigma : sig(sp') \to sig(sp)$ ein Signaturmorphismus ist und $Mod(sp)|_\sigma \subseteq Mod(sp')$ gilt. sp ist Subklasse von sp' genau dann, wenn $sig(sp') \subseteq sig(sp), sorts(sp) \leq sorts(sp')$ und $Mod(sp)|_{sig(sp')} \subseteq Mod(sp')$. $\leq$ ist die Ordnung auf Sorten. $s \leq t$ für zwei Sorten s, t impliziert $A_s \subseteq A_t$ für die zugehörigen Trägermengen. Die Definitionen sind Definition 4.1 in [Bre91] entnommen. Wie leicht zu sehen ist, lassen sich beide Relationen (bis auf das Ordnungskriterium $\leq$) durch die Implementierungsrelation darstellen. Eine Teilmengenbeziehung zwischen Trägermengen wird hier nicht unterstützt. Statt dessen wurde die Subtypbeziehung hier über $\leadsto$, und damit über Modellverfeinerung, definiert. Damit ist die Idee der Ersetzbarkeit von Supertypobjekten durch Subtypobjekte realisiert. Betrachtet man im vorliegenden Ansatz objektwertige Sorten, deren Trägermengen also Mengen von Σ-Objekten sind, und nutzt die Modellklasseninklusion, so ergeben sich Parallelen zum Ansatz ordnungssortierter Algebren. Die Ordnung ist hier $\sqsubseteq_{spec}$. Echte Werte, wie z.B. Wahrheitswerte oder natürliche Zahlen, würden dann als degenerierte Objekte, d.h. Objekte ohne veränderbaren Zustand, angesehen.

Parisi-Presicce und A. Pierantonio [PPP94] beschäftigen sich, wie Breu [Bre91] auch, mit der Vererbungsrelation im Kontext algebraischer Spezifikationen. [PPP94] stellen Ähnlichkeiten zwischen der Hierarchie der Spezifikationen, die sich aus der Komposition ('benutzt') ergibt, und der Vererbungshierarchie (siehe dort Abschnitt 5 und Conclusions) fest. Eine Klasse C, die eine Importspezifikation einer parametrisierten Klasse C' erfüllt, kann als Nachfahre in einer Vererbungsbeziehung von C' angesehen werden. C ist dann eine Spezialisierung von C'. Dies entspricht der Korrektheit der Komposition hier; die Spezialisierung ist hier durch die Implementierungsrelation ausgedrückt.

Castagna [Cas95] untersucht Subtypen und Spezialisierungen, die durch die Konzepte Kontravarianz (Subtyp) und Kovarianz (Spezialisierung) ausgedrückt werden können.

Es sollen nun noch Vererbungsrelationen in verschiedenen objektorientierten Programmiersprachen (Eiffel nach [Mey92b], Smalltalk nach [Gol85], Beta nach [MMPN93]) betrachtet werden. Es wird jeweils gezeigt, daß sich die dort vorhandenen Konzepte mit den hier erarbeiteten Mitteln modellieren lassen.

Wie schon angedeutet, ist die Vererbungsdefinition von *Eiffel* direkt durch die Relation $\triangleright$ formalisiert. Eiffel erlaubt Erweiterungen, außerdem dürfen Operationen unter den genannten Einschränkungen redefiniert werden. Eiffel erlaubt Mehrfachvererbung.

Smalltalk realisiert Vererbung in der Tradition von Simula. Es ist also nur Einfachvererbung gestattet. Nachfolger können gegenüber dem Vorgänger neue Variablen und neue Operationen enthalten. Operationen können redefiniert werden. Da es keine axiomatische Spezifikation der Operationen gibt, ist die Redefinition nicht weiter eingeschränkt. Nimmt man nun an, daß die Operationen axiomatisch so spezifiziert sind, daß jedes Modell (sofern es den syntaktischen Vorgaben entspricht, also korrekt typisiert ist) geeignet ist, dann läßt sich die Smalltalk-Vererbung durch die Implementierungrelation darstellen. Eine Vererbungrelation, die auch eine explizit spezifizierte Verschlechterung bei der Redefinition zuläßt, läßt sich ebenfalls modellieren, auch wenn sich jetzt die Implementierung nicht direkt einsetzen läßt. Durch Anwenden des *export*-Operators wird zuerst die zu redefinierende Operation ausgeblendet. Dann kann sie über den *inherit*-Operator als Neudefinition wieder hinzugenommen werden.

Beta realisiert Vererbung ebenfalls in der Simula-Tradition, verallgemeinert den Ansatz aber. Das Grundkonstrukt von Beta sind Muster (*pattern*). Eine Submusterhierarchie kann modelliert werden. Die Menge der Objekte eines Submusters ist Teilmenge der Objekte des Supermusters. Diese Teilmengenbildung wird durch Hinzufügen von weiteren Eigenschaften erreicht. Dazu gehören neue Attribute oder Eigenschaften der Attribute. Eine besondere Stellung nimmt die Spezialisierung von Aktionen (in Beta Prozedurmuster genannt) ein. Ein sogenanntes *inner*-Konstrukt wird dazu bereitgestellt. Ein Prozedurmuster spezifiziert eine partiell geordnete Sequenz von Teilaktionen, welche spezialisiert werden kann. Die Spezialisierung beruht auf dem Vererben von imperativen Code-Fragmenten. Dies läßt sich ebenfalls mit den vorliegenden Mitteln modellieren. Sei $\phi \rightarrow [P]\ \psi$ die Spezifikation des Superprozedurmusters P und $\phi' \rightarrow [P']\ \psi'$ die Spezifikation des Subprozedurmusters P'. P und P' sind jeweils Programmfragmente, die aus Teilfragmenten $P \equiv P_1, \ldots, P_n$ und $P' \equiv P'_1, \ldots, P'_m$ durch die Programmkonstruktoren *bedingtes Kommando* oder *Kommandosequenz* auf primitiven Kommandos zu der geforderten, partiell geordneten Kommandosequenz zusammengesetzt sind. In der Folge der P_i kann das Konstrukt *inner* (mehrfach) benutzt werden. Bei Ausführung eines Submusters wird die Supermusterfolge ausgeführt, wobei anstelle des *inner* jeweils die Folge der Programmfragmente des Submusters eingesetzt wird. Modelle von P bzw. P' berücksichtigen deren interne Struktur und die Spezifikation der P_i und P'_j ($\phi_{P_i} \rightarrow [P_i]\ \psi_{P_i}$ und

$\phi_{P'_j} \rightarrow [P'_j]\ \psi_{P'_j}$). *inner* kann so spezifiziert werden ($false \rightarrow [inner]\ true$), daß jedes Programmfragment $\phi' \rightarrow [P']\ \psi'$ ein korrektes Subprozedurmuster ist. Die P'_j des Submusters ersetzen also das *inner*-Konstrukt (bestimmte P_i des Supermusters). Es können dadurch Abhängigkeiten beschrieben werden, die sich nicht unbedingt durch Modellklasseninklusion beschreiben lassen, etwa wenn durch *inner* im Subprozedurmuster P' die Nachbedingung ψ' so verändert wird, daß nicht mehr $\psi' \rightarrow \psi$ gilt. Es muß dann das oben schon vorgestellte Vorgehen des Ausblendens und Neudefinierens gewählt werden.

10.3 Zusammenfassung und Abschlußbemerkungen

Der Bereich der Relationen zwischen Spezifikationen (Kapitel 8, 9 und 10) soll mit einer systematischen Zusammenfassung der bisher betrachteten Relationen zwischen Spezifikationen beendet werden.

Folgende Relationen $SP_1\ \mathcal{R}\ SP_2$ sind für einfache Spezifikationen SP_1, SP_2 definiert worden:

Name	Syntaktische Bedingung	Semantische Bedingung
$\rightsquigarrow$	$sig(SP_1) = sig(SP_2)$	$Mod(SP_2) \subseteq Mod(SP_1)$
$\triangleright$	$sig(SP_1) = sig(SP_2)$	$\forall op \in OP\ .\ op_{SP_1} \triangleright^{op} op_{SP_2}$
Import von	$sig(SP_1) \subseteq sig(SP_2)$	$Mod(SP_1) \subseteq Mod(SP_2)\vert_{sig(SP_1)}$
Export von	$sig(SP_1) \subseteq sig(SP_2)$ $state\_comp(sig(SP_1)) = \emptyset$	$Mod(SP_1) \subseteq Mod(SP_2)\vert_{sig(SP_1)}$
$<$	$sig(SP_2) \subseteq sig(SP_1)$	$Mod(SP_1)\vert_{sig(SP_2)} \subseteq Mod(SP_2)$
erbt von	$sig(SP_2) \subseteq sig(SP_1)$	$SP_2 \triangleright SP_1\vert_{sig(SP_2)}$

Auf den Modellsemantiken $< \Sigma, C >$ und $< \Sigma', C' >$ mit Σ, Σ' Signaturen und C, C' Modellklassen ist folgende Relation definiert:

Name	Syntaktische Bedingung	Semantische Bedingung
$\sqsubseteq_{Spec}$	$\Sigma = \Sigma'$	$C' \subseteq C$

Für parametrisierte Spezifikationen $SP_1 = < SP_I^1, SP_R^1 >$ und $SP_2 = < SP_I^2, SP_R^2 >$ sind folgende Relationen definiert worden:

Name	Syntaktische Bedingung	Semantische Bedingung
$\rightsquigarrow^{par}$	$sig(SP_R^1) = sig(SP_R^2)$ $sig(SP_I^1) = sig(SP_I^2)$	$SP_R^1 \rightsquigarrow SP_R^2$ $SP_I^2 \rightsquigarrow SP_I^1$
benutzt	ρ ist Signaturmorphismus	$SP_I^1 \rightsquigarrow SP_R^2\vert_\rho$

Definierte Relation für Module $SP_1 = < SP_I^1, SP_R^1, SP_E^1 >$, $SP_2 = < SP_I^2, SP_R^2, SP_E^2 >$ ist:

Name	Syntaktische Bedingung	Semantische Bedingung
$\rightsquigarrow^{mod}$	$sig(SP_R^1) = sig(SP_R^2)$ $sig(SP_E^1) = sig(SP_E^2)$ $sig(SP_I^1) = sig(SP_I^2)$	$SP_R^1 \rightsquigarrow SP_R^2$ $SP_E^1 \rightsquigarrow SP_E^2$ $SP_I^2 \rightsquigarrow SP_I^1$

Um eine der Relationen zwischen zwei Spezifikationen aufzustellen, ist in der Regel zuerst in jeder Benutzung einer Relationsform eine Anpassung von Signaturen durchzuführen. Dies ist ein rein syntaktisches Vorgehen und daher problemlos. Die Spezifikationsoperatoren sind durch Signaturmorphismen definiert, etwa *rename* basiert auf einem bijektiven Signaturmorphismus, *export* definiert eine Einbettung. Sie erlauben also die in den Relationsdefinitionen geforderte Reduktbildung auszuführen. Nach Anpassung der Signaturen reduzieren sich die meisten Relationen auf die Implementierungrelation $\rightsquigarrow$. Diese ist zwar eine semantisch basierte Konstruktion, kann aber in der Regel durch die konstruktivere Relation $\rhd$ ersetzt werden.

Lemma 10.3.1 *Seien SP_1 und SP_2 einfache Spezifikationen. Dann gilt:*

(1)	$SP_1 < SP_2$	$\Leftrightarrow$	SP_1 *Import von* SP_2
(2)	$SP_1 \rhd SP_2$	$\Rightarrow$	$SP_1 \rightsquigarrow SP_2$
	$SP_1 \rightsquigarrow SP_2$	$\Rightarrow$	$SP_1 < SP_2$
	SP_1 *Export von* SP_2	$\Rightarrow$	$SP_1 < SP_2$
	SP_1 *Export von* SP_2	$\Rightarrow$	SP_1 *Import von* SP_2
	$SP_1 \rhd SP_2$	$\Rightarrow$	SP_1 *erbt von* SP_2

Beweis: Siehe Tabelle für einfache Spezifikationen. □

Folgerung 10.3.1 *Seien SP_1 und SP_2 Spezifikationen mit $sig(SP_1) = sig(SP_2)$. Dann gilt:*

(1)	$SP_1 \rightsquigarrow SP_2$	$\Leftrightarrow$	$SP_1 < SP_2$
	$SP_1 < SP_2$	$\Leftrightarrow$	SP_1 *Import von* SP_2
	SP_1 *Import von* SP_2	$\Leftrightarrow$	SP_1 *Export von* SP_2
	SP_1 *erbt von* SP_2	$\Leftrightarrow$	$SP_1 \rhd SP_2$
(2)	$SP_1 \rhd SP_2$	$\Rightarrow$	$SP_1 \rightsquigarrow SP_2$

Beweis: Siehe Tabelle für einfache Spezifikationen. □

Folgerung 10.3.1 zeigt, daß nach syntaktischer Anpassung von Signaturen die wichtigsten Relationen äquivalent zur Implementierungsrelation sind (dies gilt auch für die Beziehung zwischen aktuellem und formalem Parameter einer Komposition). Die Implementierungsrelation wiederum läßt sich durch die syntaktisch orientierte Relation $\rhd$ approximieren. Damit sind zur Handhabung des Ansatzes im wesentlichen syntaxbasierte Mechanismen erforderlich.

Die hier vorgestellten Spezifikationsmechanismen und -relationen umfassen die gängigen Konzepte in existierenden imperativen und objektorientierten Programmier- und Spezifikationssprachen (Eiffel, Beta, Smalltalk, OS (nach R. Breu)). Es wurde hier, wie auch in [Bre91], versucht, Relationen der objektorientierten Entwicklung in einen Ansatz formaler, axiomatischer Spezifikation zu integrieren.

Es wurde in diesem Kapitel ein Ansatz realisiert, der unter der Bezeichnung *Modul* eine formale Importschnittstelle mit einer expliziten Exportschnittstelle über einer Rumpfspezifikation realisiert. Dies waren Anforderungen des Komponentenmodells. Diese Mechanismen erlauben verschiedene methodische Vorgehensweisen. Ein *bottom-up*-Vorgehen zur Konstruktion umfassender Bausteine aus einfacheren durch Aktualisierung eines formalen Imports einer Komponente durch eine Exportspezifikation einer anderen ist ebenso möglich, wie ein *top-down*-Vorgehen durch Verfeinerung und Implementierung. Insbesondere durch den formalen

Imports wird die unabhängige Entwicklung einzelner Komponenten erlaubt. Erst bei der Komposition werden Kontextbezüge hergestellt.

Als Semantik für die Spezifikationsausdrücke wurde hier eine funktionale, d.h. auf typisiertem λ-Kalkül basierende Semantik gewählt. Morphismen zwischen Spezifikationsteilen (Rumpf, Import, Export) basieren auf Modellklasseninklusion (formalisiert durch die Implementierungsrelation $\rightsquigarrow$). $\rightsquigarrow$ ist die zentrale Relation. Bei gleicher Signatur zweier Spezifikationen SP_1, SP_2 gilt für fast alle Relationen $\mathcal{R}$ (siehe Folgerung 10.3.1)

$$SP_1 \ \mathcal{R} \ SP_2 \ \Rightarrow \ SP_1 \rightsquigarrow SP_2$$

falls $\mathcal{R}$ hier eine der Relationen $<$ (Subtypbeziehung), $\triangleright$ (Vererbungsrelation), Importrelation oder Exportrelation ist. Bei der Subtypbeziehung ist die Umkehrung $\mathcal{R}^{-1}$ zu nehmen (sie entspricht in der definierten Form dem üblichen Sprachgebrauch). Alle wichtigen Korrektheitsbedingungen (etwa für Komposition oder parametrisierte Spezifikation) sind durch die Implementierungsrelation (und ggf. eine syntaktische Anpassung) definiert worden. $\rightsquigarrow$ ist die mächtigste der betrachteten Relationen. Von allen untersuchten gilt die Implementierungsrelation zwischen den meisten Spezifikationen. Auch im Vergleich zu anderen Ansätzen aus der Literatur ist die Implementierungrelation hier mächtig angelegt, da die Relation auf Modellklasseninklusion basiert – ein intuitiv einsichtiger Ansatz im Gegensatz zu Implementierungsbegriffen, die auf der Anwendung einer Folge von Spezifikationsoperatoren basieren [Wir90, Par90][1]. Modelle sind hier nach dem sehr losen Ansatz gebildet, d.h. die Modellklassen müssen weder isomorphieabgeschlossen noch vollständig erreichbar sein. Es werden alle Objekte als Modelle akzeptiert, die ein geeignetes Verhalten realisieren. Der Modellbegriff ist daher im weitestmöglichen (sinnvollen) Umfang definiert. Wesentliches Kriterium bei der Definition war die Verhaltensgleichheit. Unterschiedliche interne Strukturierungen sollten keine Rolle spielen. Da die Implementierungsrelation allerdings nur auf rein semantischen Kriterien (der Modellklasseninklusion) beruht, ist der Ansatz durch verschiedene, methodisch motivierte Relationen erweitert worden, die konstruktiver, d.h. stärker syntaktisch basiert, und somit für den Spezifizierer beherrschbarer sind.

[1] Bei diesen ergibt sich zusätzlich das Problem, daß die horizontale Kompositionseigenschaft nicht (immer) gilt.

Teil IV

Abschluß

Kapitel 11

Zusammenfassung und zukünftige Arbeiten

11.1 Zusammenfassung

Der in dieser Arbeit vorgestellt Ansatz stellt die formalen Grundlagen bereit, die notwendig sind, um einen integrierten Sprachansatz aus imperativer Programmiersprache und zustandsbasierter, komponentenorientierter Spezifikationssprache definieren zu können. Es wurde dabei ein besonderes Augenmerk auf die Anwendbarkeit des Ansatzes gelegt. Es sollen also mit Hilfe dieses Ansatzes reale Programmiersprachen betrachtet werden können, die mehr als nur 'Spielzeug'-Charakter haben. Durch die Betrachtung verschiedenster Programmiersprachenkonzepte ist dies möglich. Eine Unterstützbarkeit durch Werkzeuge in einer Software-Entwicklungsumgebung war eine Leitlinie beim Entwurf des Ansatzes.

Ihre Hauptmotivation bezieht diese Arbeit aus der Notwendigkeit formaler Methoden in der Software-Entwicklung. Insbesondere in die am weistesten verbreiteten imperativen Sprachen haben sie kaum Eingang gefunden. Unter dem Gesichtspunkt der Korrektheit von Software ist dies aber von essentieller Bedeutung.

In dieser Arbeit wurden alle wesentlichen Konstrukte bereitgestellt, die notwendig sind, einen integrierten, praxisrelevanten Sprachansatz definieren zu können. Im Bereich der imperativen Konstrukte sind alle gängigen Elemente imperativer Sprachen behandelt. Die Auswahl dynamischer Logik zur Spezifikation fußt auf den positiven Erfahrungen, die mit Invarianten und Vor- und Nachbedingungen zur Spezifikation zustandsbasierter Datentypen (wie etwa in Eiffel) gemacht wurden und auch auf langjährigen Erfahrungen aus dem Einsatz der Hoare-Logik und Sprachen wie VDM und Z zur Spezifikation und Verifikation von Programmen. Die dynamische Logik bietet hier sogar noch weitere Möglichkeiten. Die für den Bereich der horizontalen und vertikalen Entwicklung ausgewählten Konstrukte haben sich bereits in einer Vielzahl von algebraischen Spezifikationssprachen bewährt. Sie wurden zudem in ein Komponentenkonzept eingebettet, daß die Software-Entwicklung in Gruppen und in größeren Projekten unterstützt.

Im Ansatz sind die Forderungen des Komponentenmodells CM realisiert. Es sind also nicht nur mächtige Beschreibungsmittel für Komponenten selbst, sondern auch Konzepte zur Unterstützung methodischer Belange, wie Modularisierung oder Verfeinerung, mit in die Arbeit eingeflossen.

Im Kern des Ansatzes steht das Berechnungsmodell der Σ-Objekte. Vereinfacht gesprochen sind Σ-Objekte Algebren, die um ein Zustandskonzept erweitert wurden. Ein Zustand kann durch Kommandos, die zusätzlich zu Ausdrücken bereit stehen, verändert werden. Kommandos und Ausdrücke bilden die Benutzungsschnittstelle der Σ-Objekte. Diese sind formal durch Mengen und Funktionen auf diesen Mengen definiert. Σ-Objekte realisieren damit eine formal definierte abstrakte Maschine. Σ-Objekte sind in verschiedenen Richtungen erweitert worden. Die formale Definition ist auf Fixpunktsemantik erweitert worden, so daß auch rekursive Operationsdefinitionen zugelassen sind. Zur weiteren Unterstützung prozeduraler Abstraktionen wurden deren verschachtelte Definierbarkeit, verschiedene Parameterübergabemechanismen und lokale Objekte in Operationen betrachtet. Erweiterungen des Typsystems auf Σ-Objekt-Ebene wurden realisiert. Hierzu gehören Datentypkonstruktoren, objektwertige Typen und polymorphe Typen.

Σ-Objekte sind so gestaltet, daß auf ihnen sowohl imperative Sprachen semantisch definiert (sie nutzen den Charakter der abstrakten, formal definierten Maschine) als auch Interpretationen zustandsbasierter Logiken festgelegt werden können (letztere betrachten Σ-Objekte als semantische Strukturen). Damit wird eine konstruktive Modellbildung erlaubt, d.h. die Modellbildung der logischen Spezifikationssprache basiert auf der Semantik der imperativen Sprache. Hier wurde eine dynamische Logik vorgestellt. Sie erweitert eine Prädikatenlogik um Konnektoren zur Spezifikation von Zustandsübergängen. Als Modelle dieser Übergänge werden bzgl. der Spezifikation partiell korrekte Funktionen akzeptiert. Modellbildung erfolgt hier nach einem sehr losen Ansatz, d.h. daß Erreichbarkeit von Trägerelementen nicht gefordert wird und Modellklassen nicht isomorphieabgeschlossen sein müssen. Es wird so eine verhaltensorientierte Modellbildung erreicht, in der Strukturaspekte transparent sind. Um die Mächtigkeit des Ansatzes weiter zu erhöhen, sind auf Spezifikationsebene Konstrukte eingeführt worden, die spezielle Eigenschaften der Σ-Objekte sichtbar machen. Drei Standardprädikate werden hierzu eingesetzt. Ein Ordnungsprädikat erlaubt es, Aussagen über Halbordnungen auf Trägermengen zu machen. Das Gleichheitsprädikat wurde von einer festen Interpretation als Identität gelöst und kann nun frei als Äquivalenzrelation definiert werden. Mit Hilfe eines Beobachtungsprädikates können Aussagen über erreichbare Teilstrukturen gemacht werden. Zur Logik gehört ein konsistentes und relativ vollständiges Beweissystem. Dieses unterstützt die Ziele einer Programmlogik, die partielle Korrektheit von Implementierungen gegenüber Spezifikationen nachweisbar macht.

Den letzten Teil der Arbeit bildet die Übertragung der aus der Literatur bekannten Ansätze horizontaler und vertikaler Entwicklung. Im Zentrum steht hier eine Implementierungsrelation zwischen zwei Spezifikationen, die über Modellklasseninklusion definiert ist. Sie kann beliebige semantische Verfeinerungen von Spezifikationen fassen und ist daher fundamental. Da die Implementierungsrelation rein auf semantischen Kriterien beruht, wurde eine Spezialisierung dieser Relation definiert, die stärker als die Implementierungsrelation selbst auf syntaktischen und damit für den Spezifizierer überprüfbaren Kriterien beruht.

Vertikale Entwicklung bedeutet Senken des Abstraktionsniveaus einer Spezifikation. Dies kann direkt mit der Implementierungsrelation modelliert werden. Horizontale Entwicklung setzt im wesentlichen das Konzept modularer Entwicklung um. Es werden Parametrisierungs- und Schnittstellen-Konstrukte definiert, auf denen Kompositionen basieren. Zur weiteren Unterstützung wurden einige Spezifikationsoperatoren angeboten. Wichtige Eigenschaften wie die horizontale und die vertikale Kompositionseigenschaft konnten nachgewiesen werden. Beziehungen zwischen Spezifikationen konnten in der Regel auf die Implementierungsrelation zurückgeführt werden. Als Erweiterung der Betrachtung wurden eine Reihe weiterer Relationen zwischen Spezifikationen untersucht, die für Aspekte wie Typisierung oder Objektorientierung von Bedeutung sind.

Die Anwendbarkeit der vorgestellten Ergebnisse war von besonderer Bedeutung bei der Gestaltung dieser Arbeit. Die Anwendbarkeit des Ansatzes zur Definition von Spezifikations- und Implementierungssprachen wurde immer im Auge behalten. Die Kritik der Unstrukturiertheit und der für Anwender zu starken Formalität an denotationaler Semantik zur Definition imperativer Sprachen wurde berücksichtigt und führte zur Definition einer erweiterbaren, modularen, formal definierten abstrakten Maschine, den Σ-Objekten. Der Semantikansatz der Σ-Objekte ist so gestaltet worden, daß die Definition imperativer Sprachen durch Bereitstellung einer Vielzahl von Sprachprimitiven gegenüber der Benutzung klassischer Semantikansätze deutlich erleichtert wird. Die Konzepte für Spezifikationssprachen orientieren sich vielfach direkt an bewährten Konzepten, etwa an algebraischen Spezifikationssprachen, die hier an den zustandsbasierten Kontext angepaßt wurden. Spezifikationsansätze sind ohne Unterstützung durch Werkzeuge nur wenig für den praktischen Einsatz geeignet. Dieses wurde beim Entwurf der Spezifikationslogik und der Konzepte zur horizontalen und vertikalen Entwicklung berücksichtigt. Syntaktische Kriterien lassen sich leicht durch Werkzeuge unterstützen. Die Implementierungsrelation wurde durch die konstruktivere Relation $\triangleright$ unterstützt. Hierzu wurden noch eine Reihe von Verfeinerungsschemata angegeben. Die Kritik am wenig konstruktiven Vorgehen der Verifikation wurde damit aufgegriffen und in ein mehr transformationelles, korrektheitserhaltendes Vorgehen umgesetzt. Die Konzepte der horizontalen Entwicklung sind durch Signaturmorphismen und die Implementierungsrelation (die wiederum durch $\triangleright$ approximiert werden kann) definiert.

Fazit

Die Kernidee dieser Arbeit war, die *formale Spezifikation zustandsbasierter, modularer Systeme* zu ermöglichen. Zum einen sind viele der zu realisierenden Software-Systeme zustandsbasiert. Zum anderen basieren imperative oder objektorientierte Sprachen auf einem Zustandskonzept. Geeignete Abstraktionen hierfür sind also sinnvoll. Aus dieser Idee ergibt sich sofort die Anforderung, ein geeignetes *Berechnungsmodell* zur Definition eines solchen Ansatzes — Spezifikations- und Programmiersprachen integrierend — verfügbar zu machen. Daß Σ-Objekte hier ein adäquates, problemnahes Konzept darstellen, konnte gezeigt werden. Das Komponentenmodell CM zeigt, daß auch die Methodik und der Prozeß der Software-Entwicklung in die Sprachgestaltung einfließen sollte. Dieser Anforderung ist Rechnung getragen, indem ein modulares *Komponentenkonzept* und eine Vielzahl mächtiger *Relationen zwischen Spezifikationen* erarbeitet wurde.

11.2 Zukünftige Arbeiten

Zukünftige Arbeiten betreffen zum einen Erweiterungen des Ansatzes selbst, etwa für die Σ-Objekte oder die Ebene der Logik, und zum anderen die Umsetzung in methodische Kontexte und bestehende Spezifikationssprachen. Einige Ideen sollen hier kurz angerissen werden. Das Berechnungsmodell könnte um Konzepte zur Ausnahmebehandlung erweitert werden. Die dynamische Logik ist zu einer deontischen Logik erweiterbar. Eine sich anbietende Umsetzung ist im Kontext der Sprache Π gegeben. Aus methodischer Sicht scheint es sinnvoll, sich mit komponentenorientierter Wiederverwendung zu beschäftigen.

11.2.1 Ausnahmebehandlung und Zusicherungen

Zusicherungen [Mey92a, Ros95] sind formale Bedingungen, die an das Verhalten eines Software-Systems gestellt werden. Im Kontext von Programmiersprachen geschieht dies üblicherweise durch Annotation des Quelltextes. Zusicherungen können eingesetzt werden, um statisch die Implementierung zu verifizieren, oder um dynamisch Software-Fehler zu erkennen (und zu behandeln), dies im wesentlichen im Rahmen des Testens und *Debugging*.

In Eiffel können Zusicherungen zur Erkennung von Laufzeitfehlern sogar in Produktionsversionen eingesetzt werden. Dann sollten Zusicherungen allerdings mit einem Ausnahmebehandlungskonzept gekoppelt werden. Die Annotationen in solchen Systemen werden durch einen Präprozessor in Anweisungen zur Laufzeitüberprüfung transformiert, die an geeigneten Stellen der Ausführung aufgerufen werden. Die Verläßlichkeit von Software-Systemen kann nur dann garantiert werden, wenn auf unerwartet auftretende Ereignisse wie Systemfehler, die Verstöße gegen Zusicherungen darstellen, noch innerhalb eines Programmes reagiert werden kann.

Zusicherungskonzepte treten in drei Formen auf:

1. *Erweiterung einer Programmiersprache:* Ein Beispiel ist ANNA, eine Sprache, die die Annotation von Ada-Programmen ermöglicht [LvH85].
2. *In Programmiersprachen integriert:* Ein Beispiel ist hier Eiffel (Vor- und Nachbedingungen, Invarianten).
3. *Als vollständige formale Spezifikationssprache:* In VDM oder Z gibt es Vor- und Nachbedingungen, die Ausprägungen von Zusicherungen sind.

Im dritten Fall sind Zusicherungen Teil der Spezifikation eines abstrakten Datentyps, der durch die imperative Beschreibung implementiert wird. Zusicherungen sind also Teil der axiomatischen, semantischen Beschreibung, können aber auch in Laufzeitüberprüfungen genutzt werden.

11.2.2 Deontische Logik

Deontische Logik [Åqv84] ist eine modale Logik, die die Spezifikation von Lebenszyklen von Objekten zu spezifizieren erlaubt. Deontische Logik wird auch als Logik der Zwänge und Erlaubnisse bezeichnet (engl. *logic of obligation and permission*). Sie läßt sich als Erweiterung der dynamischen Logik entwickeln. Statt von Objekten spricht man hier auch häufig von Agenten [Wie91, FMR91] oder Aktoren [WM91].

Deontische Logik ist mit temporaler Logik verwandt. In beiden Ansätzen wird abstrakt das Zusammenspiel von Operationen spezifiziert. Insbesondere sind hier Veränderungen auf Folgen von Zuständen von Bedeutung. Die deontische Logik MAL [FMR91] beschäftigt sich mit Lebendigkeits- und Sicherheitsaspekten (siehe auch Einleitung 1.5.2 und Abschnitt 5.4.4).

11.2.3 Π-Kontext

Π basiert, wie auch der vorliegende Ansatz, auf den Ideen des Komponentenmodells CM. Daher bietet sich eine Verknüpfung beider an. Der Vergleich der Vorstellung von Π und insbesondere der Abschnitte zur horizontalen und vertikalen Entwicklung zeigt die Nähe beider Ansätze. Es wurden hier Schnittstellen-, Strukturierungs- und Kompositionskonzepte wie in Π erarbeitet. Daher ist der vorliegende Ansatz prädestiniert, eine zustandsbasierte Sicht für Π zu realisieren.

Zur Umsetzung des bisherigen Ansatzes in den Π-Kontext bietet es sich an, die imperative Sicht zu einer zustandsbasierten Sicht zu erweitern. Die imperative Sicht könnte um axiomatische Spezifikationen in den Schnittstellen erweitert werden. Im Rumpf würde weiterhin imperativ spezifiziert. Ein einfacher modaler Ansatz könnte hier in den Schnittstellensektionen realisiert werden. Die imperativen Konstrukte von Π lassen sich bis auf den Parallelitätsanteil einfach auf Σ-Objekten definieren. Die Verknüpfung zwischen Rumpf und Schnittstellen ist damit geschaffen. Das Parallelitätskonstrukt ist noch zu untersuchen, da sich der vorliegende Ansatz auf sequentielle, deterministische Programme beschränkt.

Als Erweiterung bietet es sich an, eine vollständig axiomatisch spezifizierbare, zustandsbasierte Sicht zuzulassen.

11.2.4 Wiederverwendung

Wiederverwendung verfolgt im wesentlichen zwei Ziele. Zum einen sollen die Kosten der Software-Entwicklung reduziert werden, indem schon vorhandene Software-Produkte wiederverwendet werden. Zum anderen soll die Vertrauenswürdigkeit der erstellten Software verbessert werden.

Wiederverwendung ist an zwei Stellen in der Software-Entwicklung relevant:

Entwicklung mit Wiederverwendung: Im Rahmen der Entwicklung neuer Software wird versucht, in einer Kollektion schon realisierter Komponenten geeignete für die gegebenen Anforderungen zu finden und diese zu integrieren.

Entwicklung zur Wiederverwendung: Wiederverwendbare Bausteine können auf zwei Arten entstehen. Zum einen können entwickelte Einzelkomponenten auf ihre Wiederverwendbarkeit geprüft werden. Grundlegend ist aber die Entwicklung von Wiederverwendungsbibliotheken für allgemeine und anwendungsspezifische Komponenten. In diese Bibliotheken sind die zuerst angesprochenen Einzelkomponenten zu integrieren.

Zwei Techniken, die eng miteinander gekoppelt sind, sind die Modularisierung und die Wiederverwendung von Bausteinen. Jede Zerlegung in Teilsysteme sollte so erfolgen, daß, wenn möglich, neu entstehende Teilsysteme durch schon existierende Komponenten realisiert werden. Nicht jedes Fragment einer Programmier- oder Spezifikationssprache ist geeignet, Objekt der Wiederverwendung zu sein. Die Methodik legt fest (oder leitet an), wie Anforderungen formuliert und gruppiert werden. Dabei spielt die inhaltliche Geschlossenheit der Anforderungskollektionen eine wichtige Rolle beim Verständnis und bei der Strukturierung des Systems. Einen Kriterienkatalog zur Evaluierung von Fragmenten hat M. Goedicke in [Goe93] im Rahmen des Komponentenmodells *CM* bereitgestellt. Je besser ein Fragment anhand des Modells bewertet wird, um so mehr ist es als Einheit der Wiederverwendung geeignet. Gängige Komponenten, die dieses Kriterium erfüllen, sind Operationen und Module. Detailliert werden Aspekte der Wiederverwendung im Rahmen formaler komponentenorientierter Entwicklung in [Cra94] untersucht. Wiederverwendung im Kontext formaler Entwicklung wird auch in [CDG94] betrachtet.

11.2.5 Weitere Bereiche für Erweiterungen

Weitere Ansätze für zukünftige Arbeiten liegen in den Bereichen:

Erweiterung der Sprachklasse. Eine Erweiterung von sequentiellen, deterministischen zu parallelen, nichtdeterministischen Sprachen kann durchgeführt werden. Viele imperative Sprachen sind um solche Konzepte erweitert. Um nicht nur den Kern dieser Sprachen definieren zu können, ist eine Erweiterung der Σ-Objekte sinnvoll. Zum einen können die primitiven Konstrukte zur Behandlung von Nichtdeterminismus erweitert werden. Bei konkurrierender Benutzung von Ressourcen sind Überlegungen zur Synchronisation anzustellen.

Formale Entwicklung und Prototyping. Die Vorzüge formaler Entwicklung kommen nur dann zum Tragen, wenn die Anforderungen klar und formal formulierbar sind. Prototyping ist insbesondere dann von Nutzen, wenn diese Klarheit nicht gegeben ist [BKKZ92, FGH+93]. Methodische Überlegungen zu einer Integration beider Ansätze könnten angestellt werden.

Modulkonzept für ProSet. Ausgehend von den Überlegungen zur methodischen Integration von Prototyping und formaler Entwicklung kann das Modulkonzept der Sprache ProSet überarbeitet werden. Das Prototyping von Algorithmen kombiniert mit einer formalen Spezifikation von Schnittstellen könnte Ausgangspunkt der Entwicklung eines Modulkonzeptes sein. Erfahrungen im Einsatz von ProSet habe gezeigt, daß, neben der notwendigen Flexibilität zum Prototyping, insbesondere im Bereich von Modulen und Prozeduren eine formale Beschreibung der Schnittstellen wünschenswert ist. Die in der Einleitung (Abschnitt 1.4.3) angegebene und an ProSet angelehnte Zielvorstellung könnte formal definiert werden.

Teil V

Anhang

Anhang A

Mathematische Grundlagen

A.1 Mengen, Relationen, Funktionen

Notationen aus der Mengentheorie sind:

$\emptyset$	:	die leere Menge
$S \subseteq T$	:	S ist Teilmenge von T
$\mathcal{P}(S)$	:	bezeichnet die Potenzmenge von S, d.h. die Menge aller Teilmengen von S
$S_1 \times \ldots \times S_n$	:	Das kartesische Produkt der Mengen $S_1, \ldots, S_n$ $S_1 \times \ldots \times S_n = \{(s_1, \ldots, s_n) \mid s_i \in S_i \ \textit{für}\ i = 1, .., n\}$ auch zum Teil $\Pi_1^n s_i$ geschrieben
$\mathcal{Z} = \{\ldots, -1, 0, 1, \ldots\}$	:	die Menge der ganzen Zahlen
$\mathcal{N} = \{1, 2, 3, \ldots\}$	:	die Menge der natürlichen Zahlen
$\mathcal{N}_0 = \{0, 1, 2, \ldots\}$	:	$\mathcal{N} \cup \{0\}$

Seien S und T Mengen. Eine **Relation** (zwischen S und T) ist eine Teilmenge $R \subseteq S \times T$.

Sei $R \subseteq S \times T$ eine Relation. Für jede Teilmenge $A \subseteq S$ ist das **Bild** von A (unter der Relation R) definiert durch:

$$R(A) = \{t \in T \mid (s,t) \in R \ \textit{für mindestens ein}\ s \in A\}$$

Für jede Teilmenge $B \subseteq T$ ist das **Urbild** von B (unter der Relation R) definiert durch:

$$R^{-1}(A) = \{s \in S \mid (s,t) \in R \ \textit{für mindestens ein}\ t \in B\}$$

Eine Relation $R \subseteq S \times T$ heißt:

- **total**, falls $R^{-1}(T) = S$
- **partielle Funktion**, falls für jedes $s \in S$ höchstens ein $t \in T$ mit $(s,t) \in R$ existiert
- **totale Funktion**, falls R total ist und R eine partielle Funktion ist.

Totale Funktionen werden auch mit $f : S \to T$ bezeichnet.

$s \in S$ ist **Fixpunkt** einer (totalen) Funktion $f : S \to S$, falls $f(s) = s$.

Seien $f : S \to T$ und $g : T \to V$ totale Funktionen. Die **Komposition** von f und g wird mit $g \circ f$ bezeichnet und durch $g \circ f(s) = g(f(s))$ für alle $s \in S$ definiert.

Sei f eine totale Funktion. Dann wird die i-te Iteration von f, bezeichnet mit f^i, definiert durch ($i \in \mathcal{N}_0$):

$$f^0 = f$$
$$f^{i+1} = f \circ f^i$$

A.2 Prädikatenlogik

Die folgenden Definitionen orientieren sich an [LS84] Kapitel 2. Sei V eine Menge von Variablen.

Definition A.2.1 *Eine* **Basis** *für eine Prädikatenlogik ist ein Paar $B = (F, P)$ von Symbolmengen, wobei F und P Mengen von Funktions- bzw. Prädikatensymbolen sind. Jedem Funktions- oder Prädikatensymbol wird eine Stelligkeit zugeordnet.*

Definition A.2.2 *Die Menge T_B aller* **Terme** *einer Prädikatenlogik über einer Basis $B = (F, P)$ wird induktiv definiert:*

- *Jede Variable aus V ist ein Term. Jedes nullstellige Funktionssymbol (Konstante) ist ein Term.*
- *Falls $t_1, \ldots, t_n$ Terme und $f \in F$ ein n-stelliges Funktionssymbol, dann ist $f(t_1, \ldots, t_n)$ ein Term.*

Definition A.2.3 *(Syntax) Die Menge WFF_B aller* **wohlgeformten Formeln** *einer Prädikatenlogik über einer Basis $B = (F, P)$ wird induktiv definiert:*

- *Die Wahrheitswerte $true$ und $false$ sind Formeln. Jede Konstante aus P ist eine Formel. Falls t_1 und t_2 Terme sind, ist $t_1 = t_2$ eine Formel. Falls $t_1, \ldots, t_n$ Terme und $p \in P$ ein n-stelliges Prädikatensymbol, dann ist $p(t_1, \ldots, t_n)$ eine Formel.*
- *Falls ϕ eine Formel ist, dann ist auch $\neg\phi$ eine Formel. Falls ϕ eine Formel und x eine Variable ist, dann sind $\forall x.\phi$ und $\exists x.\phi$ Formeln. Falls ϕ und ψ Formeln sind, dann sind auch $\phi \wedge \psi$ und $\phi \vee \psi$ Formeln.*

Definition A.2.4 *(Interpretation) Sei $B = (F, P)$ eine Basis für eine Prädikatenlogik. Eine* **Interpretation** *von B ist ein Paar $I = (D, I_0)$, wobei D eine nichtleere Menge (Domain) und I_0 eine Abbildung ist, definiert durch folgende Zuweisungen:*

1. *zu jeder Konstante $c \in F$ ein Element $I_0(c) \in D$,*
2. *zu jedem Funktionssymbol $f \in F$ der Stelligkeit $n \geq 1$ eine totale Funktion $I_0(f) : D^n \to D$,*
3. *zu jeder Konstante $a \in P$ ein Element $I_0(a) \in Bool$,*

4. zu jedem Prädikatensymbol $p \in P$ der Stelligkeit $n \geq 1$ ein Prädikat $I_0(p) : D^n \rightarrow Bool$.

Eine totale Funktion $\gamma : V \rightarrow D$, die Variablen in den Domain D einer Interpretation abbildet, heißt **Zuweisung**. Die Menge aller Zuweisungen einer Interpretation wird mit Γ bezeichnet.

Definition A.2.5 *(Semantik) Sei $I = (D, I_0)$ eine Interpretation für eine Basis $B = (F, P)$. Für jeden Term t wird eine Funktion $I(t) : \Gamma \rightarrow D$ definiert und für jede Formel ϕ eine Funktion $I(\phi) : \Gamma \rightarrow Bool$. Diese Funktionen werden induktiv definiert:*

- Terme*: Falls $c \in F$ eine Konstante ist, dann ist $I(c)(\gamma) = I_0(c)$ für alle Zuweisungen $\gamma \in \Gamma$. Falls $x \in V$ eine Variable ist, dann ist $I(x)(\gamma) = \gamma(x)$ für alle $\gamma \in \Gamma$. Falls die $t_1, \ldots, t_n$ Terme sind und $f \in F$ ein n-stelliges Funktionssymbol ist, dann ist $I(f(t_1, \ldots, t_n))(\gamma) = I_0(f)(I(t_1)(\gamma), \ldots, I(t_n)(\gamma))$ für alle $\gamma \in \Gamma$.*
- Formeln*: $I(true)(\gamma) =$ `true` für alle $\gamma \in \Gamma$. $I(false)(\gamma) =$ `false` für alle $\gamma \in \Gamma$. Falls $a \in P$ eine Konstante ist, dann ist $I(a)(\gamma) = I_0(a)$ für alle $\gamma \in \Gamma$. Falls t_1 und t_2 Terme sind, dann ist*

$$I(t_1 = t_2)(\gamma) = \begin{cases} \texttt{true} & \textit{falls } I(t_1)(\gamma) = I(t_2)(\gamma) \\ \texttt{false} & \textit{sonst} \end{cases}$$

für alle $\gamma \in \Gamma$. Falls die $t_1, \ldots, t_n$ Terme sind und $p \in P$ ein n-stelliges Prädikatensymbol ist, dann ist $I(p(t_1, \ldots, t_n))(\gamma) = I_0(p)(I(t_1)(\gamma), \ldots, I(t_n)(\gamma))$ für alle $\gamma \in \Gamma$. Falls $\phi \in WFF_B$ eine Formel ist, dann ist

$$I(\neg\phi)(\gamma) = \begin{cases} \texttt{true} & \textit{falls } I(\phi)(\gamma) = \texttt{false} \\ \texttt{false} & \textit{sonst} \end{cases}$$

für alle $\gamma \in \Gamma$. Analog für die Konnektoren $\vee$ und $\wedge$. Falls $\phi \in WFF_B$ und $x \in V$, dann ist

$$I(\forall x.\phi)(\gamma) = \begin{cases} \texttt{true} & \textit{falls für alle } d \in D \ I(\phi)(\gamma[x/d]) = \texttt{true} \\ \texttt{false} & \textit{sonst} \end{cases}$$

für alle $\gamma \in \Gamma$.

Ableitungsregeln für eine Prädikatenlogik erster Stufe (nach [Kre91] Kapitel 8) sollen im folgenden angegeben werden.

Sei E eine Menge von Formeln und seien ϕ, ϕ' Formeln.

Konjunktion

$$\phi \wedge \phi' \vdash \phi \ \ bzw. \ \ \phi \wedge \phi' \vdash \phi'$$

$$\phi, \phi' \vdash \phi \wedge \phi'$$

Implikation

$$\vdash \phi \vee (\phi \rightarrow \phi')$$

$$\phi' \vdash \phi \Rightarrow \phi'$$

Disjunktion

$$\phi \vdash \phi \vee \phi'$$

$$\phi' \vdash \phi \vee \phi'$$

Äquivalenz

$$\phi, \phi' \vdash \phi \Leftrightarrow \phi' \vdash \phi \vee \phi' \vee (\phi \Leftrightarrow \phi')$$

$$\phi \Leftrightarrow \phi', \phi \vdash \phi'$$

$$\phi \Leftrightarrow \phi', \phi' \vdash \phi$$

Quantoren

$$\frac{E, \phi \vdash \phi'}{E, \forall x.\phi \vdash \phi'} \qquad \frac{E \vdash \phi \vee \phi'}{E \vdash (\forall x.\phi) \vee \phi'} \ (*)$$

$$\frac{E, \phi \vdash \phi'}{E, \exists x.\phi \vdash \phi'} \ (*) \qquad \frac{E \vdash \phi \vee \phi'}{E \vdash (\exists x.\phi) \vee \phi'}$$

$(*)$ nur falls x in E, ϕ' nicht frei vorkommt.

A.3 Algebraische Spezifikation

Diese Zusammenfassung orientiert sich an [Wir90].

Signaturen und Spezifikationen

Eine **Signatur** $\Sigma =< S, F >$ besteht aus einer Menge von **Sorten** S und einer Menge F von **Funktionssymbolen**. Jedem Element f aus F ist ein Typ $type(f)$ mit $type : F \rightarrow S^* \times S$ zugeordnet. Falls $type(f) = (s_1 \ldots s_n, s)$, schreiben wir $f : s_1 \times \ldots \times s_n \rightarrow s$.

Gegeben seien eine Signatur Σ und eine Familie $X = (X_s)_{s \in S}$ von Variablen. Für jede Sorte $s \in S$ ist die Menge der **Σ-Terme der Sorte** s $T(\Sigma, X)_s$ die kleinste Menge, die folgendes enthält:

1. jedes $x \in X_s$ und jedes nullstellige Operationssymbol $f \in F$ mit $type(f) = (s)$ und
2. jedes $f(t_1, \ldots, t_n)$, wobei $f : s_1 \times \ldots \times s_n \rightarrow s$ ein Operationssymbol aus F und jedes t_i $(i = 1, .., n)$ ein Term der Sorte s_i in $T(\Sigma, X)_{s_i}$ ist.

Die Menge $T(\Sigma, X)$ der **Terme mit Variablen** ist die Vereinigung aller Mengen $T(\Sigma, X)_s$. Die Menge $T(\Sigma)$ aller **Terme** (ohne Variablen) ist definiert durch $T(\Sigma) = T(\Sigma, \emptyset)$.

Eine **Gleichungsspezifikation**, oder kurz **Spezifikation**, $SP = < S, F, E >$ besteht aus einer Signatur $\Sigma =< S, F >$ und einer Menge E von Formeln.
$\forall x \in X \ . \ t_1 = t_2$ ist eine **Gleichung**, wobei X eine Menge von **Variablen** ist und t_1, t_2 Terme mit Variablen aus X über Σ der gleichen Sorte sind. **Formeln** werden in der Menge $WFF(\Sigma)$ zusammengefaßt. $WFF(\Sigma)$ ist die kleinste Menge, die die folgenden Eigenschaften hat:

1. jede Gleichung ist in $WFF(\Sigma)$,
2. falls $\phi, \psi \in WFF(\Sigma)$, dann sind auch $\neg\phi, \phi \wedge \psi \in WFF(\Sigma)$,
3. falls $x \in X_s$ und $\phi \in WFF(\Sigma)$, dann ist $\forall x : s.\phi \in WFF(\Sigma)$.

Ein **Signaturmorphismus** $h : \Sigma_1 \rightarrow \Sigma_2$ mit $\Sigma_i = < S_i, F_i >$ für $i = 1, 2$ ist ein Paar $h = (h_S, h_F)$ von Funktionen $h_S : S_1 \rightarrow S_2$ und $h_F : F_1 \rightarrow F_2$, so daß für jedes $f : s_1 \times \ldots \times s_n \rightarrow s$ in F und $n \geq 0$ gilt $h_F(f) : h_S(s_1) \times \ldots \times h_S(s_n) \rightarrow h_S(s)$ in F_2.

Algebren und Homomorphismen

Sei eine Signatur $\Sigma = < S, F >$ gegeben. Eine **Σ-Algebra** A besteht aus einer Familie $(A_s)_{s \in S}$ von **Trägermengen** A_s und einer Familie $F_A = (f_A)_{f \in F}$ von **Operationen** f_A :

$A_{s_1} \times \ldots \times A_{s_n} \to A_s$ für $f : s_1 \times \ldots \times s_n \to s$ und $n \geq 0$, wobei im Fall $n = 0$ $f_A \in A_s$ eine **Konstante** genannt wird.

Eine Abbildung $v : X \to A$ heißt **Bewertung** von X in A. Eine **Interpretation** eines Termes t in A (in bezug auf v) ist eine Abbildung $v^* : T(\Sigma, X) \to A$, die wie folgt definiert wird:

1. $v^*(x) := v(x)$ für alle $x \in X$,
2. $v^*(f(t_1, \ldots, t_n)) := f^A(v^*(t_1), \ldots, v^*(t_n))$ für jedes $f : s_1 \times \ldots \times s_n \to s \in F$ und $t_i \in T(\Sigma, X)_{s_i}, i = 1, .., n$.

Sei A eine Σ-Algebra und v eine Belegung. Zu einer Spezifikation $SP = < \Sigma, E >$ ist ein **Modell** A eine Σ-Algebra, die alle Formeln $e \in E$ erfüllt ($A \models e$). Eine Gleichung $t_1 = t_2$ **gilt** in einer Σ-Algebra, d.h. $A \models t_1 = t_2$, wenn für alle Belegungen $v_A : X \to A$ gilt $v_A^*(t_1) = v_A^*(t_2)$ für die erweiterte Belegung v_A^* von v_A. Die Formel $\neg\phi$ gilt in A bzgl. v ($A, v \models \neg\phi$), falls ϕ nicht gilt. Die Formel $\phi \wedge \psi$ gilt in A unter v, falls ϕ gilt und ψ gilt. $\forall x : s.\phi$ gilt in A bzgl. v, falls ϕ in A bzgl. aller Belegungen $v_x : X \to A$ mit $v_x(y) = v(y)$ für $x \neq y$ gilt.

Seien eine Signatur $\Sigma = < S, F >$ und A und B zwei Σ-Algebren. Ein **Σ-Homomorphismus** $h : A \to B$ ist eine Familie $h = (h_s)_{s \in S}$ von Funktionen $h_s : A_s \to B_s$, so daß für jede Konstante $f :\to s$ in F und $s \in S$ gilt $h_s(f_A) = f_B$ und für jedes Operationssymbol $f : s_1 \times \ldots \times s_n \to s$ für alle $a_i \in A_{s_i}$ und $i = 1, .., n$

$$h_s(f_A(a_1, \ldots, a_n)) = f_B(h_{s_1}(a_1), \ldots h_{s_n}(a_n))$$

Ein Σ-Homomorphismus $h : A \to B$ heißt **isomorph**, wenn alle Funktionen $h_s : A_s \to B_s$ für $h = (h_s)_{s \in S}$ bijektiv sind.

Termalgebren, Initialität

Die **Termalgebra** $T_{SIG}(X)$ **mit Variablen** hat die Trägermengen $T(\Sigma, X)_s$, die Konstanten $f_T = f$ für jedes Konstantensymbol in F und die Operationen $f_T : T(\Sigma, X)_{s_1} \times \ldots \times T(\Sigma, X)_{s_n} \to T(\Sigma, X)_s$ definiert durch $f_T(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ für jedes $f : s_1 \times \ldots \times s_n \to s$ in F und $t_i \in T(\Sigma)_{s_i}$ für $i = 1, .., n$. Für $X = \emptyset$ erhält man die **Termalgebra** T_{SIG} (ohne Variablen).

Eine Σ-Algebra heißt **erreichbar**, falls gilt: zu jedem $s \in S, a \in A_s$ existiert $t \in T(\Sigma)_s$ mit $a = t^A$ (die eindeutig bestimmte Interpretation von t in A). Sei $Alg(\Sigma)$ die Klasse aller Σ-Algebren und $Gen(\Sigma)$ die Klasse aller erreichbaren Σ-Algebren. $Gen(\Sigma, E)$ ist dann die Klasse aller erreichbaren Σ-Algebren, die die Formeln in E erfüllen. Eine Σ-Algebra A heißt **initial**, gdw. es zu jeder Σ-Algebra B genau einen Σ-Homomorphismus $h : A \to B$ gibt.

Modellsemantiken für $SP = < \Sigma, E >$:

$Mod(SP) := Gen(\Sigma, E)$	*Lose Semantik*
$Mod(SP) := \{A \in Gen(\Sigma, E) \mid A \text{ ist initial in } Gen(\Sigma, E)\}$	*Initiale Semantik*

A.4 Denotationale Semantik

Inhalte dieses Abschnitts sind [LS84] entnommen.

Halbordnung, Suprema, cpo

Definition A.4.1 *(Halbordnung) Sei D eine Menge und $\sqsubseteq$ eine Relation auf D. Dann heißt das Paar $(D, \sqsubseteq)$ eine* **Halbordnung**, *falls die Relation $\sqsubseteq$ reflexiv, antisymmetrisch und transitiv ist.*

Definition A.4.2 *(Kleinste obere Schranke) Sei $(D, \sqsubseteq)$ eine Halbordnung und S eine (möglicherweise leere) Teilmenge von D. Ein Element $u \in D$ heißt* **obere Schranke** *von S (in D), falls für alle $d \in S$ $d \sqsubseteq u$ gilt; u heißt* **kleinste obere Schranke** *von S (in D), falls u das kleinste Element in der Menge aller oberen Schranken von S in D ist.*

Bemerkung A.4.1 *Die kleinste obere Schranke wird auch* **Supremum** *genannt und für eine Menge S mit $\sqcup S$ bezeichnet.*

Bemerkung A.4.2 *Für ein kartesisches Produkt $D_1 \times \ldots \times D_n$, $n \geq 1$ und $i = 1, \ldots, n$ bezeichnet pr_i die i-te Projektionsfunktion $pr_i : D_1 \times \ldots \times D_n \to D_i$ definiert durch:*

$$pr_i(d_1, \ldots, d_n) = d_i$$

Sei S eine Teilmenge des kartesischen Produkts $D_1 \times \ldots \times D_n$. Dann ist $pr_i(S)$ definiert durch:

$$pr_i(S) = \{pr_i(s) | s \in S\}$$

Definition A.4.3 *Die Ordnung $\sqsubseteq$ auf dem kartesischen Produkt $D_1 \times \ldots \times D_n$ sei wie folgt definiert:*

$$(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n) \text{ gdw. } a_i \sqsubseteq b_i \text{ für alle } i = 1, \ldots, n$$

Satz A.4.1 *(Kartesisches Produkt und Supremum) Seien $(D_1, \sqsubseteq), \ldots, (D_n, \sqsubseteq)$ Halbordnungen und sei S eine Teilmenge des kartesischen Produkts $D_1 \times \ldots \times D_n$, $n \geq 1$. Dann existiert $\sqcup S$ gdw. $\sqcup pr_i(S)$ existiert für $i = 1, \ldots, n$. Weiterhin hat das Supremum, falls es existiert, den Wert*

$$\sqcup S = (\sqcup pr_1(S), \ldots, \sqcup pr_n(S))$$

Bemerkung A.4.3 *Sei $(D, \sqsubseteq)$ eine Halbordnung. Eine nichtleere Teilmenge S von D heißt* **Kette** *in D, falls $d \sqsubseteq d'$ oder $d' \sqsubseteq d$ (oder beides) gilt für alle Elemente $d, d' \in S$. Man kann auch sagen, S ist eine Kette, falls die Ordnungsrelation $\sqsubseteq$ auf S total ist.*

Definition A.4.4 *(Vollständige Halbordnung) Eine Halbordnung $(D, \sqsubseteq)$ ist eine* **vollständige Halbordnung** *oder* **cpo** *(für 'complete partial order'), falls die beiden folgenden Bedingungen gelten:*

1. *Die Menge D hat ein kleinstes Element. Dieses Element wird mit $\perp_D$ oder $\perp$ bezeichnet.*

2. *Für jede Kette S in D existiert das Supremum $\sqcup S$.*

Satz A.4.2 *Jede Halbordnung, die ein kleinstes Element hat und nur endliche Ketten enthält, ist eine cpo.*

Satz A.4.3 *Falls* $(D_1, \sqsubseteq), \ldots, (D_n, \sqsubseteq)$ *cpos sind, ist auch* $(D_1 \times \ldots \times D_n, \sqsubseteq)$ *cpo.*

Stetigkeit, Sub-cpo, Funktionen

Definition A.4.5 *(Stetigkeit) Seien* $(D, \sqsubseteq)$ *und* $(E, \sqsubseteq)$ *cpos. Eine Funktion* $f : D \to E$ *heißt* **stetig**, *falls für jede Kette* S *in* D *das Supremum* $\sqcup f(S)$ *existiert und es gilt* $f(\sqcup S) = \sqcup f(S)$.

Satz A.4.4 *Seien* $(D, \sqsubseteq)$ *und* $(E, \sqsubseteq)$ *cpos und* $f : D \to E$ *eine Funktion.*

1. *Die Funktion* f *ist stetig, gdw.* f *monoton ist und für jede Kette* S *in* D *gilt* $f(\sqcup S) \sqsubseteq \sqcup f(S)$.
2. *Falls* D *nur endliche Ketten enthält, ist* f *stetig gdw.* f *monoton ist.*

Definition A.4.6 *Sei* $(D, \sqsubseteq)$ *eine cpo und* E *eine Teilmenge von* D. E *heißt* **Sub-cpo** *von* D, *falls*

1. *die Halbordnung* $(E, \sqsubseteq)$ *eine cpo ist, und*
2. $\sqcup_E S = \sqcup_D S$, *für jede Kette* S *in* E.

Definition A.4.7 *Seien* D *und* E *Wertebereiche. Dann ist* $(D \to E)$ *die Menge aller totalen Funktionen von* D *nach* E.

Definition A.4.8 *Seien* $f, g \in (D \to E)$.

$$f \sqsubseteq g \ \text{gdw.}\ f(d) \sqsubseteq g(d) \quad \forall d \in D$$

Satz A.4.5 *Falls* D *eine Menge und* $(E, \sqsubseteq)$ *eine cpo, dann ist* $((D \to E), \sqsubseteq)$ *eine cpo.*

Satz A.4.6 *Seien* $(D, \sqsubseteq)$ *und* $(E, \sqsubseteq)$ *cpos. Die Menge* $[D \to E]$ *aller stetigen Funktionen von* D *nach* E *ist eine Sub-cpo von der Halbordnung auf der Menge der totalen Funktionen von* D *nach* E $((D \to E), \sqsubseteq)$.

Satz A.4.7 *Seien* $(D, \sqsubseteq), (E, \sqsubseteq)$ *und* $(F, \sqsubseteq)$ *cpos und* $g : D \to E$ *und* $f : E \to F$ *stetige Funktionen. Dann ist ihre Komposition* $f \circ g : D \to F$ *ebenfalls stetig.*

Satz A.4.8 *Seien* $(D, \sqsubseteq), (E_i, \sqsubseteq), i = 1, \ldots, n$ *cpos mit* $n \geq 1$. *Sei* $f : D \to E_1 \times \ldots \times E_n$ *eine Funktion. Dann ist* f *stetig gdw.* $pr_i \circ f$ *stetig für* $i = 1, \ldots, n$ *(*pr_i *ist die* i*-te Projektionsfunktion).*

Definition A.4.9 *Seien* $D_1, \ldots, D_n$, $n \geq 2$ *und* E *beliebige Mengen und sei* $f : D_1 \times \ldots \times D_n \to E$ *eine Funktion. Dann ist die* **Curry-Funktion** *von* f, *bezeichnet mit* f^c, *eine Funktion von* $D_1 \times \ldots \times D_{n-1}$ *in die Menge* $(D_n \to E)$, *definiert durch*

$$f^c(d_1, \ldots, d_{n-1})(d_n) = f(d_1, \ldots, d_n)$$

Satz A.4.9 *Seien* $(D_1, \sqsubseteq), \ldots, (D_n, \sqsubseteq)$ *und* $(E, \sqsubseteq)$ *cpos. Und sei* $f : D_1 \times \ldots \times D_n \to E$ *eine Funktion und* f^c *die Curry-Funktion zu* f*. Dann ist* f *genau dann eine stetige Funktion über* $D_1 \times \ldots \times D_n \to E$*, wenn* f^c *stetige Funktion über* $D_1 \times \ldots \times D_{n-1} \to (D_n \to E)$ *ist.*

Fixpunkte

Bemerkung A.4.4 *Ein* **Fixpunkt** *einer (totalen) Funktion* $f : S \to S$ *ist ein Element* $s \in S$*, für das* $f(s) = s$ *gilt. Der* **kleinste Fixpunkt** *von* f *ist das kleinste Element in der Menge der Fixpunkte von* f*.*

Satz A.4.10 *(Fixpunkttheorem) Sei* $(D, \sqsubseteq)$ *eine cpo und* $f : D \to D$ *eine stetige Funktion. Dann hat* f *den kleinsten Fixpunkt* μf *mit*

$$\mu f = \sqcup\{f^i(\bot) | i \in \mathcal{N}_0\}$$

(wobei $\bot$ *das kleinste Element der cpo* $(D, \sqsubseteq)$ *ist und* f^i *die* i*-te Iteration der Funktion* f*).*

Satz A.4.11 *Sei* $(D, \sqsubseteq)$ *eine cpo. Dann ist das Funktional* $\mu : [D \to D] \to D$*, das eine Funktion* f *auf deren Fixpunkt* μf *abbildet, stetig.*

Definition A.4.10 *Sei* $(D, \sqsubseteq)$ *eine cpo und* $\phi : D \to Bool$ *ein Prädikat. Das Prädikat* ϕ *heißt* **zulässig***, falls für jede Kette* S *in* D *die folgende Bedingung gilt:*

Falls $\phi(d) = true$ *für alle* $d \in S$*, dann* $\phi(\sqcup S) = true$

Bemerkung A.4.5 *Eine Formel* ϕ *gilt in einem* Σ*-Objekt* O *unter der Bewertung* v*, d.h.* $O \models \phi$*, gdw.* $v^*(STATE, \phi(..))[2] = true$ *für alle* $STATE \in State$*.*

Satz A.4.12 *Seien* $(D, \sqsubseteq)$*,* $(E, \sqsubseteq)$ *cpos. Seien* $g_i, h_i \in [D \to E]$ *für* $i = 1, ..n$*. Dann ist das Prädikat* $\phi(d) = true$ *gdw.* $g_i(d) \sqsubseteq h_i(d)$ *für alle* $i = 1, .., n$ *zulässig.*

Korollar A.4.1 *Seien* $(D, \sqsubseteq)$*,* $(E, \sqsubseteq)$ *cpos. Seien* $g, h \in [D \to E]$*. Dann ist das Prädikat* $\phi(d) = true$ *gdw.* $g(d) = h(d)$ *zulässig.*

Satz A.4.13 *(Scott'sches Induktionsprinzip oder Fixpunktinduktionsprinzip) Sei* $(D, \sqsubseteq)$ *eine cpo,* $f \in [D \to D]$ *eine stetige Funktion und* $\phi : D \to Bool$ *ein zulässiges Prädikat. Falls*

- *(Induktionsanfang)* $\phi(\bot) = true$*, und*
- *(Induktionsschritt)* $\phi(d) = true$ impliziert $\phi(f(d)) = true$ *für jedes* $d \in D$

dann $\phi(\mu f) = true$*.*

Bemerkung A.4.6 *Das Satz A.4.13 erlaubt das Zeigen von Eigenschaften für* $f^i(\bot)$ *durch Induktion über* i*. Die zulässigen Prädikate garantieren, daß der Sprung von der Kette* $\{f^i(\bot) | i \in \mathcal{N}_0\}$ *zu ihrem Supremum* μf *möglich ist.*

Satz A.4.14 *(Stetigkeit des Fixpunktoperators) Sei* $(D, \sqsubseteq)$ *eine cpo. Dann ist das Funktional* $\mu : [D \to D] \to D$*, das eine Funktion* $f \in [D \to D]$ *auf ihren kleinsten Fixpunkt* μf *abbildet, stetig.* μ *wird Fixpunktoperator genannt.*

Literaturverzeichnis

[AAZ93] D. Ancona, E. Astesiano, und E. Zucca. Towards a Classification of Inheritance Relations. In U. Lipeck und G. Koschorrek, Hrsg., *IS-CORE'93 Workshop, Informatik-Berichte 01/93*, Seiten 90–113. Universität Hannover, 1993.

[AdB94] P. America und F. de Boer. Reasoning about Dynamically Evolving Process Structures. *Formal Aspects of Computing*, 6:269–316, 1994.

[Ame90] P. America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J.W. de Bakker, W.P. Roever, und G. Rozenberg, Hrsg., *Foundations of Object-Oriented Languages, REX School/Workshop, LNCS 489*, Seiten 60–90. Springer-Verlag, 1990.

[Ant94] G. Antoniou. The Verification of Modules. *Formal Aspects of Computing*, 6:223–244, 1994.

[AO91] K.R. Apt und E.-R. Olderog. Introducing Program Verification. In E.J. Neuhold und M. Paul, Hrsg., *Formal Description of Programming Concepts*, Seiten 363–430. Springer-Verlag, 1991.

[AO94] K.R. Apt und E.-R. Olderog. *Programmverifikation*. Springer-Verlag, 1994.

[Apt81] K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oktober 1981.

[Åqv84] L. Åqvist. Deontic Logic. In D. Gabbay und F. Guenthner, Hrsg., *Handbook of Philosophical Logic Vol. II: Extensions of Classical Logic*, Seiten 605–714. D. Reidel Publishing, 1984.

[AR90] P. America und J. Rutten. A Layered Semantics for a Parallel Object-Oriented Programming Language . In J.W. de Bakker, W.P. Roever, und G. Rozenberg, Hrsg., *Foundations of Object-Oriented Languages, REX School/Workshop, LNCS 489*, Seiten 91–123. Springer-Verlag, 1990.

[AS88] K. Alber und W. Struckmann. *Einführung in die Semantik von Programmiersprachen*. BI Wissenschaftsverlag, 1988.

[Ast91] E. Astesiano. Inductive and Operational Semantics. In E.J. Neuhold und M. Paul, Hrsg., *Formal Description of Programming Concepts*, Seiten 51–136. Springer-Verlag, 1991.

[ASU88] A.V. Aho, R. Sethi, und J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[Bar93] K.J. Barwise, Hrsg. *Handbook of Mathematical Logic – Studies in Logic and the Foundations of Mathematics, Vol. 90.* North-Holland, 1993. 8. Auflage.

[BBB+85] 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, und H. Wössner, Hrsg. *The Munich Project CIP, Vol.1: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, 1985.

[BF91] Jan A. Bergstra und L.M.G. Feijs, Hrsg. *Algebraic Methods II: Theory, Tools and Applications.* LNCS 490. Springer-Verlag, 1991.

[BJ94] M. Broy und S. Jähnichen, Hrsg. *Korrekte Software durch formale Methoden – Abschlußbericht des BMFT-Verbundprojekts KORSO.* Bundesministerium für Forschung und Technologie, 1994.

[Bjø91] Dines Bjørner. Specification and Transformation – Methodology Aspects of the Vienna Development Method. In E.J. Neuhold und M. Paul, Hrsg., *Formal Description of Programming Concepts*, Seiten 137–258. Springer-Verlag, 1991.

[BKKZ92] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, und Heinz Züllighoven. *Prototyping: An Approach to Evolutionary System Development.* Springer-Verlag, 1992.

[BMPW86] M. Broy, B. Möller, P. Pepper, und M. Wirsing. Algebraic Implementations Preserve Program Correctness. *Science of Computer Programming*, 7:35–53, 1986.

[Bre91] Ruth Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* LNCS 562, Springer-Verlag, 1991.

[BW81] F.L. Bauer und H. Wössner. *Algorithmische Sprache und Programmentwicklung.* Springer-Verlag, 1981.

[BW82] M. Broy und M. Wirsing. Partial Abstract Types. *Acta Informatica*, 18:47–64, 1982.

[BWP87] M. Broy, M. Wirsing, und P. Pepper. On the Algebraic Definition of Programming Languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54–99, Januar 1987.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, und G. Plotkin, Hrsg., *Proc. International Symposium on Semantics of Data Types, LNCS 173*, Seiten 51–67. Springer-Verlag, 1984.

[Car91] L. Cardelli. Typeful Programming. In E.J. Neuhold und M. Paul, Hrsg., *Formal Description of Programming Concepts*, Seiten 431–507. Springer-Verlag, 1991.

[Cas95] G. Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, Mai 1995.

[CDG94] J. Cramer, E.-E. Doberkat, und M. Goedicke. Some Formal Methods Supporting Software Reuse. In W. Schäfer, R. Prieto-Diaz, und M. Matsumoto, Hrsg., *Software Reusability*, Seiten 79–112. Ellis Horwood, 1994.

[CFGGR91] J. Cramer, W. Fey, M. Goedicke, und M. Große-Rhode. Towards a Formally Based Component Description Language – a Foundation for Reuse. *Structured Programming*, 12(12):91–110, 1991.

[CHC90] W.R. Cook, W.L. Hill, und P.S Canning. Inheritance is not Subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, Seiten 125–135. ACM Press, 1990.

[CL94] Y. Cheon und G.T. Leavens. The Larch/Smalltalk Interface Specification Language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, 1994.

[Cla79] E.M. Clarke. Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems. *Journal of the ACM*, 26(1):129–147, Januar 1979.

[Coo74] S.A. Cook. Axiomatic and interpretive semantics for an Algol fragment. Technical Report 79, University of Toronto, Computer Science Department, 1974.

[Coo78] S.A. Cook. Soundness and Completeness of an Axiom System for Program Verification. *SIAM Journal on Computers*, 7:70–90, 1978.

[Cou90] P. Cousot. Methods and Logics for Proving Programs. In J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Vol. B*, Seiten 841–996. Elsevier Science Publishers, 1990.

[Cra94] J. Cramer. Interconnecting and Reusing Component Specifications. Dissertation, Universität Dortmund, 1994.

[CW85] L. Cardelli und P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism . *ACM Computing Surveys*, 17(4):471–522, Dezember 1985.

[dB80] J.W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall, 1980.

[DFG+92a] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, und C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, Hrsg., *Proc. Third International Workshop on Rapid System Prototyping*, Seiten 235–248, Research Triangle Park, NC, Juni 1992. IEEE Computer Society Press.

[DFG+92b] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, und C. Pahl. ProSet — Prototyping with Sets, Language Definition. Software-Engineering Memo 15, Universität GH Essen, März 1992.

[Dil91] A. Diller. *Z — An Introduction to Formal Methods.* John Wiley and Sons, 1991.

[Dijk76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DT88] S. Danforth und C. Tomlinson. Type Theories and Object–Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, März 1988.

[EGL89] H.J. Ehrich, M. Gogolla, und U. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* B.G. Teubner, 1989.

[EM85] H. Ehrig und B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[EM90] H. Ehrig und B. Mahr. *Fundamentals of Algebraic Specification 2: Modules and Constraints, EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Vol. B*, Seiten 995–1072. Elsevier Science Publishers, 1990.

[Eve95] M. Everz. Mengentransformationen für die mengen-orientierte Prototyping-Sprache ProSet (Diplomarbeit). SWT-Memo 79, Universität Dortmund, April 1995.

[Feij89] L.M.G. Feijs. The Calculus $\lambda\pi$. In *Algebraic Methods: Theory, Tools and Applications*, Seiten 307–328. Springer-Verlag, 1989.

[Feij94] L.M.G. Feijs. An Overview of the Development of COLD. In D.J. Andrews, J.F. Groote, und C.A. Middelburg, Hrsg., *1st International Workshop on Semantics of Specification Languages, Utrecht, 1993*, Seiten 15–22. Springer-Verlag, 1994.

[Fen93] Dieter Fensel. Über den Sinn formaler Spezifikationssprachen. Bericht 293, Universität Karlsruhe, Februar 1993.

[FGH+93] W. Franke, U. Gutenbeil, W. Hasselbring, C. Pahl, H.-G. Sobottka, und B. Sucrow. Prototyping mit Mengen — der ProSet-Ansatz. In H. Züllighoven, W. Altmann, und E.-E. Doberkat, Hrsg., *Requirements Engineering 93: Prototyping*, Seiten 165–174. B.G. Teubner, Stuttgart, 1993.

[Fis92] R. Fischbach. Programming by Contract – Erfüllt Eiffel das Ideal? In H.-J. Hoffmann, Hrsg., *Eiffel — Fachtagung German Chapter of the ACM, Darmstadt*, Seiten 55–68. Springer-Verlag, Mai 1992.

[FJ92] L.M.G Feijs und H.B.M Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992.

[FM90] J. Fiadero und T. Maibaum. Describing, Structuring and Implementing Objects. In J.W. de Bakker, W.P. Roever, und G. Rozenberg, Hrsg., *Foundations of Object-Oriented Languages, REX School/Workshop, LNCS 489*, Seiten 274–310. Springer-Verlag, 1990.

[FM95] J. Fiadero und T. Maibaum. Interconnecting Formalisms: Supporting Modularity, Reuse, and Incrementality. In Gail E. Kaiser, Hrsg., *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, Seiten 72–80. ACM Software Engineering Notes 20 (4), Oktober 1995.

[FMR91] J. Fiadero, T. Maibaum, und M. Ryan. Sharing Actions and Attributes in Modal Action Logic. In T. Ito und A.R. Meyer, Hrsg., *Proceedings Conference on Theoretical Aspects of Computer Software TACS'91, Sendai, Japan*. LNCS 526, Springer-Verlag, September 1991.

[Fra92] Nissim Francez. *Program Verification.* Addison Wesley, 1992.

[FSMS91] J. Fiadero, C. Sernadas, T. Maibaum, und G. Saake. Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In R.A. Meersman, W. Kent, und S. Khosla, Hrsg., *Object-Oriented Databases: Analysis, Design and Construction (DS-4)*, Seiten 243–284. North Holland, 1991.

[Gau91] M.-C. Gaudel. Advantages and Limits of Formal Approaches for Ultra-High Dependency. In *Proceedings sixth International Workshop on Software Specification and Design*, Seiten 237–241, Oktober 1991.

[GH93] J.V. Guttag und J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, und Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[GM87a] J. A. Goguen und J. Meseguer. Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems. In *Proceedings Logic in Computer Science*, Seiten 18–29, 1987.

[GM87b] J.A. Goguen und J. Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In B. Shriver und P. Wegener, Hrsg., *Research Directions in Object-Oriented Programming*, Seiten 417–478. MIT Press, 1987.

[Goe93] M. Goedicke. On the Structure of Software Description Languages: A Component Oriented View. Habilitationsschrift, Forschungsbericht 473, Universität Dortmund, Fachbereich Informatik, 1993.

[Gol85] A. Goldberg. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1985.

[GRdL94] R. Groenboom und G.R. Renardel de Lavalette. Reasoning about Dynamic Features in Specification Languages – A Modal View on Creation and Modification. In D.J. Andrews, J.F. Groote, und C.A. Middelburg, Hrsg., *1st International Workshop on Semantics of Specification Languages, Utrecht, 1993*, Seiten 340–356. Springer-Verlag, 1994.

[GRdL95] R. Groenboom und G.R Renardel de Lavalette. A Formalization of Evolving Algebra. In S. Fisher und M. Trautwein, Hrsg., *Proceedings Accolade '95*, Seiten 17 – 28, Amsterdam, 1995. Dutch Graduate School in Logic.

[Gri81] D. Gries. *The Science of Programming.* Springer Verlag, 1981.

[GS94] M. Goedicke und H. Schumann. Component-Oriented Software Development with Π. Bericht 01-94, Universität GH Essen, 1994.

[GSC91] M. Goedicke, H. Schumann, und J. Cramer. On the Specification of Software Components. In *Proceedings sixth International Workshop on Software Specification and Design*, Seiten 166–174, Oktober 1991.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, und J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[Gur93] Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In G. Rozenberg und A. Salomaa, Hrsg., *Current Trends in Theoretical Computer Science*, Seiten 266–292. World Scientific, 1993.

[Gur94] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, Hrsg., *Specification and Validation Methods*. Oxford University Press, 1994.

[GW90] J.A. Goguen und D. Wolfram. On Types and FOOPS. In *Proceedings IFIP TC2 Working Conference on Database Semantics, Object–Oriented Databases, Windermere*, Juli 1990.

[Har79] David Harel. *First-Order Dynamic Logic*. Springer-Verlag, 1979.

[Har84] David Harel. Dynamic Logic. In D. Gabbay und F. Guenthner, Hrsg., *Handbook of Philosophical Logic Vol. II: Extensions of Classical Logic*, Seiten 497–604. D. Reidel Publishing, 1984.

[Has94] W. Hasselbring. Formale Spezifikation und Prototyping im Sprachentwurf. *Informatik — Forschung und Entwicklung*, 9(3):132–140, 1994.

[Hei92a] M. Heisel. *Formale Programmentwicklung mit dynamischer Logik*. Deutscher Universitäts Verlag, 1992.

[Hei92b] M. Heisel. Formalizing and Implementing Gries' Program Development Method in Dynamic Logic. *Science of Computer Programming*, 18:107–137, 1992.

[Hen89] R. Hennicker. Implementation of Parameterized Observational Specifications. In J. Diaz und F. Orejas, Hrsg., *TAPSOFT'89 Proceedings of the International Joint Conference on Theory and Practise of Software Development, Barcelona, Spain*, Seiten I:290–305. Springer-Verlag, 1989.

[Hen91a] R. Hennicker. Consistent Configuration of Modular Algebraic Implementations. Bericht MIP - 9102, Fakultät für Mathematik und Informatik, Universität Passau, 1991.

[Hen91b] R. Hennicker. Context Induction: A Proof for Behavioural Abstractions and Algebraic Implementations. *Formal Aspects of Computing*, 3:326–345, 1991.

[Hen91c] R. Hennicker. Observational Implementation of Algebraic Specifications. *Acta Informatica*, 28:187–230, 1991.

[Hen92] R. Hennicker. Behavioural Specification and Implementation of Modular Systems. Bericht MIP - 9203, Fakultät für Mathematik und Informatik, Universität Passau, 1992.

[Heu91] Andreas Heuer. Konzepte objektorientierter Datenmodelle. In K.-U. Witt und G. Vossen, Hrsg., *Entwicklungstendenzen bei Datenbank-Systemen*. Oldenbourg, 1991.

[HN92] R. Hennicker und F. Nickl. A Behavioural Algebraic Framework for Modular System Design with Reuse. Bericht LMU - 9206, Informatik, Ludwig-Maximilians-Universität München, 1992.

[Hoa69] C.A.R Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

[Hoa78] C.A.R Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21:666–677, 1978.

[Hor84] E. Horowitz. *Fundamentals of Programming Languages.* Computer Science Press, 1984.

[Jon89a] H.B.M. Jonkers. An Introduction to COLD-K. In *Algebraic Methods: Theory, Tools and Applications*, Seiten 139–206. Springer-Verlag, 1989.

[Jon89b] H.B.M. Jonkers. Description Algebra. In *Algebraic Methods: Theory, Tools and Applications*, Seiten 283–306. Springer-Verlag, 1989.

[Jon90] Cliff B. Jones. *Systematic Software Development with VDM.* Prentice Hall, 1990.

[Krä91a] B. Krämer. A Sort of Parametric Polymorphism for Algebraic Specifications. *Journal of Systems and Software*, 15:33–42, 1991.

[Krä91b] B. Krämer. Introducing the GRASPIN Specification Language SEGRAS. *Journal of Systems and Software*, 15:17–32, 1991.

[Krä92] B. Krämer. Formale Spezifikationstechniken – Stand von Methoden und Anwendungsumgebungen. *Informatik Forschung und Entwicklung*, 7:62–72, 1992.

[KRdL89] C.P.J. Koymans und G.R. Renardel de Lavalette. The Logic MPL_ω. In *Algebraic Methods: Theory, Tools and Applications*, Seiten 247–282. Springer-Verlag, 1989.

[Kre91] H.-J. Kreowski. *Logische Grundlagen der Informatik.* R. Oldenbourg Verlag, 1991.

[KST94] S. Kahrs, D. Sannella, und A. Tarlecki. The Semantics of Extended ML: A Gentle Introduction. In D.J. Andrews, J.F. Groote, und C.A. Middelburg, Hrsg., *1st International Workshop on Semantics of Specification Languages, Utrecht, 1993*, Seiten 186–215. Springer-Verlag, 1994.

[KT90] Dexter Kozen und Jerzy Tiuryn. Logics of Programs. In J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Vol. B*, Seiten 789–840. Elsevier Science Publishers, 1990.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, Mai 1994.

[Lin93] H. Lin. Procedural Implementation of Algebraic Specifications. *ACM Transactions on Programming Languages and Systems*, 15(5):876–895, November 1993.

[LK93] U. Lipeck und G. Koschorrek, Hrsg. *IS-CORE'93 Workshop, Informatik-Berichte 01/93.* Bericht Universität Hannover, 1993.

[LS84] J. Loeckx und K. Sieber. *The Foundations of Program Verification.* Wiley-Teubner, 1984.

[Lud93] J. Ludewig. Sprachen für das Software-Engineering. *Informatik-Spektrum*, 16:286–294, Dezember 1993.

[LvH85] D.C. Luckham und F.W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2:9–22, März 1985.

[Meh95] Michael Mehlich. Implementation of Combinator Specifications: Notions and Proving Techniques. Dissertation, Institut für Informatik, Ludwig-Maximilians-Universität München, 1995.

[Mey85] Bertrand Meyer. On Formalism in Specification. *IEEE Software*, 2(1):6–26, 1985.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mey92a] Bertrand Meyer. Applying Design by Contract. *Computer*, Seiten 40–51, Oktober 1992.

[Mey92b] Bertrand Meyer. *Eiffel: the Language.* Prentice Hall, 1992.

[Mit90] J.C. Mitchell. Type Systems for Programming Languages. In J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Vol. B*, Seiten 365–458. Elsevier Science Publishers, 1990.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, und Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[Möl85] B. Möller. On the Algebraic Spezification of Infinite Objects – Ordered and Continuous Models of Algebraic Types. *Acta Informatica*, 22:537–578, 1985.

[Mor88] C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):402–419, Juli 1988.

[Mor94] C. Morgan. *Programming from Specification – Second Edition.* Addison-Wesley, 1994.

[Mos89] P.D. Mosses. Unified Algebras and Modules. In *Proc. 16th ACM Symp. on Principles of Programming Languages, Austin, Texas*, Seiten 329–343, 1989.

[Mos90] P.D. Mosses. Denotational Semantics. In J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Vol. B*, Seiten 575–631. Elsevier Science Publishers, 1990.

[Mos91] P.D. Mosses. Denotational Semantics. In E.J. Neuhold und M. Paul, Hrsg., *Formal Description of Programming Concepts*, Seiten 1–50. Springer-Verlag, 1991.

[MTH90] R. Milner, M. Tofte, und R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NP91] E.J. Neuhold und M. Paul, Hrsg. *Formal Description of Programming Concepts.* Springer-Verlag, 1991.

[NS89] M. Norris und S. Stockmann. Industrializing formal methods for telecommunications. In C. Ghezzi und J. McDermid, Hrsg., *Proceedings European Conference on Software Engineering ESEC'89*, Seiten 159–175. Springer-Verlag, 1989.

[Old83] E.-R. Olderog. On the Notion of Expressiveness and the Rule of Adaption. *Theoretical Computer Science*, 24:337–347, 1983.

[Pah94] C. Pahl. Erweiterung einer imperativen Programmiersprache zu einer komponenten-orientierten Software-Beschreibungssprache. In F. Simon, Hrsg., *Workshop Deklarative Programmierung und Spezifikation, Bad Honnef, Mai 1994*, Seiten 90–93. Universität Kiel, Institut für Informatik und praktische Mathematik, Bericht 9412, 1994.

[Par90] H. Partsch. *Specification and Transformation of Programs.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Plo79] Gordon D. Plotkin. Dijkstra's predicate transformers and Smyth's powerdomains. In D. Bjørner, Hrsg., *Abstract Software Specifications, LNCS 86*, Seiten 527–553. Springer-Verlag, 1979.

[PPP94] F. Parisi-Presicce und A. Pierantonio. An Algebraic Theory of Class Specification. *ACM Transactions on Software Engineering and Methodology*, 3(2):166–199, April 1994.

[Pro95] Projektgruppe 240. Scotland Yard – Evaluation und Entwurf von Werkzeugen für eine Entwicklungsumgebung zum Software Prototyping. Abschlußbericht, Universität Dortmund, Februar 1995.

[PST91] B. Potter, J. Sinclair, und D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[RdL94a] G.R. Renardel de Lavalette. From Implicit via Inductive to Explicit Definitions. In D.J. Andrews, J.F. Groote, und C.A. Middelburg, Hrsg., *1st International Workshop on Semantics of Specification Languages, Utrecht, 1993*, Seiten 304–314. Springer-Verlag, 1994.

[RdL94b] G.R. Renardel de Lavalette. The Static Part of the Design Language COLD. In D.J. Andrews, J.F. Groote, und C.A. Middelburg, Hrsg., *1st International Workshop on Semantics of Specification Languages, Utrecht, 1993*, Seiten 51–82. Springer-Verlag, 1994.

[Ros95] David S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Januar 1995.

[SBC92] S. Stepney, R. Barden, und D. Cooper, Hrsg. *Object Orientation in Z.* Springer-Verlag, 1992.

[Sch95] A. Schönegge. Extended Dynamic Logic. Technischer Bericht, Universität Karlsruhe, 1995.

[Set89] R. Sethi. *Programming Languages – Concepts and Constructs.* Addison Wesley, 1989.

[SJS91] Gunter Saake, Ralf Jungclaus, und Cristina Sernadas. Abstract Data Type Semantics for Many-Sorted Object Query Algebras. In B. Thalheim, J. Demetrovics, und H.-D. Gerhardt, Hrsg., *Proc. Symposium Mathematical Fundamentals of Database and Knowledge Systems, Rostock, Germany, 1991*, Seiten 291–307. Springer-Verlag, 1991.

[Som92] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1992. 4th edition.

[Spi87] J.M. Spivey. *The Z Notation.* Prentice Hall, 1987.

[SST90] D. Sannella, S. Sokolowski, und A. Tarlecki. Towards Formal Development of Programs from Algebraic Specifications. Bericht 06/90, Informatik, Universität Bremen, 1990.

[ST87] D. Sannella und A. Tarlecki. On Observational Equivalence and Algebraic Specification. *Journal of Computer Systems Sciences*, Seiten 150–178, 1987.

[ST88] D. Sannella und A. Tarlecki. Specifications in arbitrary institutions. *Information and Computation*, 76:165–210, 1988.

[ST90a] D. Sannella und A. Tarlecki. A Kernel Specification Formalism with Higher-Order Parameterization. In H. Ehrig, K.P. Jantke, F. Orejas, und H. Reichel, Hrsg., *Recent Trends in Data Type Specification, 7th Workshop on Specification of Abstract Data Types*, Seiten 274–296. Springer-Verlag, 1990.

[ST90b] D. Sannella und A. Tarlecki. Extended ML: Past, present and future. In H. Ehrig, K.P. Jantke, F. Orejas, und H. Reichel, Hrsg., *Recent Trends in Data Type Specification, 7th Workshop on Specification of Abstract Data Types*, Seiten 297–322. Springer-Verlag, 1990.

[Sti91] C. Stirling. Modal and Temporal Logics. In S. Abramski, Hrsg., *Handbook of Logic in Computer Science*, Seiten 477–563. Reidel, 1991.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Ten91] R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[TWW82] J.W. Thatcher, E.G. Wagner, und J.B. Wright. Data Type Specification and the Power of Specification Techniques. *ACM Transactions on Programming Languages and Systems*, 4:711–773, 1982.

[vdL94] F.J. van der Linden. Formal Methods: from object-based to object-oriented. *ACM SIGPLAN Notices*, 29(7):29–38, Juli 1994.

[vHL89] Ivo van Horebeek und Johan Levi. *Algebraic Specifications in Software Engineering.* Springer-Verlag, 1989.

[Wan82] M. Wand. Specifications, Models, and Implementations of Data Abstractions. *Theoretical Computer Science*, 20:3–32, 1982.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, 1991.

[WB89a] M. Wirsing und J.A. Bergstra, Hrsg. *Algebraic Methods: Theory, Tools and Applications*. LNCS 394. Springer-Verlag, 1989.

[WB89b] M. Wirsing und M. Broy. A Modular Framework for Specification and Implementation. In J. Diaz und F. Orejas, Hrsg., *TAPSOFT'89 Proceedings of the International Joint Conference on Theory and Practise of Software Development, Barcelona, Spain*, Seiten I:42–73. Springer-Verlag, 1989.

[Weg90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM OOPS Messenger*, Seiten 8–87, 1990.

[WG84] W.M. Waite und G. Goos. *Compiler Construction*. Springer-Verlag, 1984.

[Wie91] R. Wieringa. A formalization of objects using equational dynamic logic. In R.A. Meersman, W. Kent, und S. Khosla, Hrsg., *Object-Oriented Databases: Analysis, Design and Construction (DS-4)*, Seiten 431–452. North Holland, 1991.

[Win87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, Januar 1987.

[Wir86] M. Wirsing. Structured Algebraic Specifications: a Kernel Language. *Theoretical Computer Science*, 43:123–250, 1986.

[Wir90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Vol. B*, Seiten 675–788. Elsevier Science Publishers, 1990.

[Wir95] M. Wirsing. Algebraic Specification Languages. In E. Astesiano, G. Reggio, und A. Tarlecki, Hrsg., *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types, 1994*, Seiten 81–115. Springer-Verlag, 1995.

[WM91] R. Wieringa und J.J. Meyer. Actor-Oriented Specification of Deontic Integrity Constraints. In B. Thalheim, J. Demetrovics, und H.-D. Gerhardt, Hrsg., *Proc. Symposium Mathematical Fundamentals of Database and Knowledge Systems, Rostock, Germany, 1991*, Seiten 89–103. Springer-Verlag, 1991.

[WPP+83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, und M. Broy. On Hierarchies of Abstract Data Types. *Acta Informatica*, 20:1–33, 1983.

[WTW78] E.G. Wagner, J.W. Thatcher, und J.B. Wright. Programming languages as mathematical objects. In J. Winkowski, Hrsg., *Mathematical Foundations of Computer Science*, Seiten 84–101. Springer-Verlag, 1978.

[ZW95] A. Moormann Zaremski und J.M. Wing. Specification Matching of Software Components. In Gail E. Kaiser, Hrsg., *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, Seiten 6–17. ACM Software Engineering Notes 20(4), Oktober 1995.

Index