Methodik funktionaler Systementwicklung

Claus Dendorfer

2. Juli 1995

Zusammenfassung

Die vorliegende Arbeit befaßt sich mit der Methodik der funktionalen Systementwicklung. Sie hat das Ziel, einen Beitrag zur Konsolidierung und Weiterentwicklung der formalen Entwurfsmethodik FOCUS zu leisten.

Dazu wird zunächst eine Variante von FOCUS eingeführt, die auf der durchgehenden Verwendung von Agenten mit benannten Kanälen basiert. Die Eigenschaften dieses Formalismus, insbesondere im Hinblick auf Spezifikationen von Agenten höherer Ordnung, werden untersucht.

Dann wird eine formale Beschreibung der einzelnen Stufen (Spurspezifikation, funktionale Spezifikation, Implementierung) einer vollständigen Systementwicklung gegeben. Verfeinerungsbegriffe innerhalb und zwischen diesen Stufen werden dargestellt.

Die Methodik der Entwicklung von einem funktional spezifizierten Agenten hin zum ausführbaren Programm bildet einen Schwerpunkt der Arbeit. Ein zustandsorientierter Formalismus für die einzelnen Entwurfsschritte wird definiert und untersucht. Es zeigt sich, daß durch die Verwendung aufeinander abgestimmter Spezifikationsschemata die Systementwicklung in einer Folge von kleinen und handhabbaren Schritten durchgeführt werden kann.

Zur Behandlung spezieller Problemfälle, beispielsweise bei der Modellierung von Betriebssystemen, werden in der Arbeit weiterhin Techniken zur Zeitmodellierung und zur Spezifikation zeitbehafteter Agenten angegeben. Schließlich wird ein auf zustandsorientierten Spezifikationstechniken basierender Formalismus zur Darstellung von dynamischen Agentennetzen definiert und untersucht.

Danksagung

Besonders danke ich Herrn Prof. Dr. Manfred Broy, der diese Arbeit in allen Stufen ihrer Entstehung wohlwollend begleitet und mehrere Vorversionen durchgesehen hat. Herr Prof. Tobias Nipkow hat sich freundlicherweise bereit erklärt, die Arbeit zu begutachten.

Rainer Weber, Ketil Stølen, Bernhard Schätz und Bernhard Rumpe haben Vorversionen dieser Arbeit gelesen und kommentiert, wofür ich ihnen herzlich danke.

Nicht zuletzt gebührt mein Dank allen ehemaligen Kolleginnen und Kollegen für interessante und wertvolle Gespräche und Diskussionen. Ich möchte die Zeit nicht missen, die ich an der Technischen Universität München verbringen durfte.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit dem ersten, anfänglich provokativen Auftauchen des Begriffs *software engineering* 1967 [NR69] ist die Notwendigkeit deutlich geworden, den Prozeß der industriellen Programmentwicklung ingenieurmäßiger, also systematischer und weniger fehleranfällig zu gestalten. Dazu dient insbesondere die Verwendung *formaler Methoden.*

Elemente formaler Methoden sind Beschreibungsformalismen, Vorgehensmodelle, Verfeinerungsregeln, Beweiskalküle und Validierungsmethoden. Typischerweise wird bei der formalen Programmentwicklung zunächst eine Anforderungsspezifikation erstellt und hinsichtlich der informellen Vorstellungen über das erwünschte Systemverhalten validiert.

Die formale Entwurfsmethode soll das Erstellen eines diese Anforderungsspezifikation erfüllenden lauffähigen Programms unterstützen. Die auf dieses Ziel gerichtete Entwicklung geschieht in einer Reihe von Schritten, in denen jeweils Entwurfsentscheidungen getroffen werden. Die Korrektheit dieser Schritte folgt entweder aus der Anwendung von vorgegebenen Verfeinerungsregeln, oder sie muß unter Verwendung von Beweisregeln gezeigt werden. Neben dem lauffähigen Programm wird auch die Folge der einzelnen Entwicklungsschritte als wichtiges Ergebnis betrachtet. Diese Geschichte des Entwicklungsvorgangs dient zur Dokumentation und erleichtert die Programmwartung und die Anpassung an neue Anforderungen.

Es existiert eine Vielzahl von Methoden, die sich in ihrem konzeptuellen Aufbau, ihrer Herkunft und ihrem Formalisierungsgrad unterscheiden. Grob lassen sich zwei Hauptlinien angeben:

- Methoden und Vorgehensmodelle aus dem industrienahen Umfeld zur Beschreibung von Systemen in verschiedenen Abstraktions- und Entwicklungsebenen. Die verwendeten Beschreibungsmittel sind oft graphischer Natur und an betriebswirtschaftliche Organisationsdiagramme (Datenflußdiagramme), Beschreibungsmittel

für Datenbanken (*Entity/Relationship*-Diagramme) und Flußdiagramme (Programmablaufpläne) angelehnt. Beispiele für solche Methoden sind *JSD* [Jac83], *SSADM* [DCC92] und *OMT* [RBP+91].

- Formalismen aus dem universitären Bereich mit Wurzeln im Bereich der Semantik von Programmiersprachen und der mathematischen Logik. Ausgehend von frühen Ansätzen von Hoare [Hoa69] und Dijkstra [Gri81] sind hier zum Beispiel der *Refinement Calculus* [Mor90], die verschiedenen Formalismen und Techniken der algebraischen Spezifikation [Wir90] und Formalismen der temporalen Logik [MP91] zur Modellierung verteilter Systeme zu nennen. Einige dieser Formalismen und deren methodische Verwendung sind in [San88] beschrieben.

Seit einiger Zeit werden Anstrengungen unternommen, die Vorteile der Methoden beider Richtungen zu kombinieren. Es wird zunehmend Wert darauf gelegt, mathematisch exakt fundierte Formalismen in der Praxis einsetzbar zu gestalten. Dies ist zum Beispiel das erklärte Ziel der Methoden *Z* [Spi87] und *VDM* [Jon90]. Einen Überblick über den praxisorientierten Einsatz verschiedener Methoden gibt [Win90]. Erfahrungen beim Einsatz von *Z* sind in [Hal90] geschildert. Die Beiträge in [Inf93] behandeln den jüngeren Stand der Bemühungen. In [Huß94] wird eine Kombination des pragmatischen Formalismus *SSADM* [DCC92] mit der formalen algebraischen Spezifikationssprache SPECTRUM [BFG+93] dargestellt.

Gegenüber einer *ad hoc* Programmentwicklung führt die Verwendung formaler Methoden zu einem erheblichen Mehraufwand in den frühen Entwicklungsphasen. Dieser Mehraufwand wird jedoch dadurch kompensiert, daß Fehler in der Entwicklung vermieden oder zumindest früher entdeckt werden, denn die Behebung eines Fehlers ist um so kostspieliger, je später dieser erkannt wird. Überdies ist ein *Beweis* der Korrektheit eines Programmes bezüglich einer Anforderungsspezifikation nur mittels formaler Methoden möglich.

Besonders vorteilhaft ist die Verwendung formaler Methoden bei sicherheitskritischen Anwendungen, für die hohe Anforderungen an die Fehlerfreiheit gestellt werden, und bei Systemen, die schwierig zu entwickeln sind, weil sie eine oder mehrere der folgenden Komplikationen aufweisen (siehe [MB87, Kapitel 5] mit weiteren Indizien für schwierig zu entwickelnde Systeme):

- logische oder physische Verteilung,
- Nebenläufigkeit,
- Verwendung gemeinsamer Ressourcen,
- intensiver Einsatz des Systems in unterschiedlichen Konfigurationen und einer Vielzahl von Betriebszuständen, oder
- zeitkritische Systeme, Echtzeitanforderungen.

Oft treten diese Problempunkte gehäuft auf. Betriebssysteme und Telekommunikationseinrichtungen sind Beispiele für Systeme, bei denen alle oben genannten Punkte

zutreffen. Weil bei diesen Systemen auch die Zuverlässigkeitsanforderungen besonders hoch sind, bietet sich der Einsatz formaler Methoden hier besonders an. Zur Modellierung solcher verteilter, oft zeitkritischer Systeme sind zum Beispiel vorgeschlagen oder eingesetzt worden:

- Protokollbeschreibungssprachen wie *LOTOS, Estelle* und *SDL* [Hog89],
- Prozeßalgebren wie *CSP* [Hoa85] oder *CCS* [Mil80] mit Weiterentwicklungen wie *Timed CSP* [RR86] oder *ACP* [BK84] mit zahlreichen Erweiterungen,
- die Sprache und Methodik *UNITY* [CM88],
- die Lamportsche Transitions-Axiom-Methode [Lam83, Lam89],
- die Erweiterungen des Hoare-Kalküls von Owicki und Gries [OG76],
- Petri-Netze [Rei82],
- temporale Logiken [MP91].

Als Beitrag zu diesem Gebiet wurde am Lehrstuhl Professor Broy der Technischen Universität München die Methodik FOCUS entwickelt. FOCUS vereinigt mehrere mathematisch fundierte Formalismen (Spurformalismus, funktionale Netzwerke, Implementierungssprachen) in einem methodischen Rahmen des schrittweisen Systementwurfs. Eine Einführung in FOCUS findet sich in [BDD+92a], und [BDD+92b] enthält einen Überblick über bisher durchgeführte Fallstudien. Eine kurze beispielhafte Systementwicklung wird in [DW93] und [DDW93] beschrieben.

Die vorliegende Arbeit baut auf FOCUS auf. Ausgangspunkt ist das Vorhaben, Beschreibungsmittel und -techniken für die Spezifikation von Betriebssystemstrukturen bereitzustellen. Betriebssysteme sind ein wichtiges Beispiel für komplexe Softwaresysteme und, wie oben erläutert, ein Gebiet, auf dem der Einsatz formaler Methoden besonders notwendig und vielversprechend ist. Wegen der Vielzahl der zu behandelnden Erscheinungen sind Betriebssysteme auch ein Prüfstein für eine formale Entwurfsmethode. Zum Beispiel treten klassische Probleme wie die Behandlung des nicht-strikten fairen Mischens oder das Verteilen von Nachrichten an einen sich dynamisch ändernden Empfängerkreis auf, wenn Betriebssysteme mit funktionalen Techniken entwickelt werden [Sto86, JS89, Tur90].

Die bei der Modellierung von Betriebssystemen auftretenden Probleme sind jedoch nicht grundsätzlich anders als die Schwierigkeiten, die bei allgemeinen komplexen, nebenläufigen Systemen auftauchen. Daher werden Betriebssystemstrukturen hier nur beispielhaft verwendet. Die in dieser Arbeit vorgestellten Techniken sind nicht auf den speziellen Anwendungsfall der Betriebssysteme beschränkt.

Beim Einsatz von FOCUS in Fallstudien stellte sich heraus, daß ein Bedarf zur Konsolidierung, Modifikation und Erweiterung sowohl auf dem Gebiet der Beschreibung von Agenten und Netzen als auch auf dem Gebiet des methodischen Einsatzes der Techniken in einer Systementwicklung besteht. Zum Beispiel wird in [BDD+92a] die gesamte

Entwurfsmethodik nur informell erläutert und insbesondere der Entwurfsprozeß auf der funktionalen Ebene nicht näher konkretisiert. Um diese Lücken auszufüllen, gibt Kapitel 4 eine exakte Spezifikation des Entwicklungsprozesses, und Kapitel 5 beschreibt eine methodisch aufeinander abgestimmte Folge von Schemata zur zustandsorientierten Entwicklung funktionaler Agenten. Diese beiden Kapitel bilden den Schwerpunkt der vorliegenden Arbeit.

Insgesamt versucht die vorliegende Arbeit, Beschreibungsmittel und -methoden zu entwickeln, die die Spezifikation und Programmentwicklung für komplexe Systeme erleichtern. Es sollen Erweiterungen und spezielle Ausprägungen von FOCUS angegeben werden, die den Formalismus und die Entwurfsmethodik sowohl von der theoretischen Seite her weiter fundieren als auch die praktische Anwendbarkeit verbessern.

In Kapitel 2 werden einige grundlegende Begriffe und Operatoren definiert.

Das Kapitel 3 enthält eine Beschreibung der Formalismen der Spurspezifikation und der funktionalen Spezifikation. Wir verwenden dabei eine Variante von FOCUS, die auf benannten Kanälen basiert. Die Grundidee dazu findet sich schon in [BD92], sie wird aber hier erweitert und konsequent eingesetzt. Durch die Verwendung von Kanalnamen ergibt sich eine natürliche Verbindung zwischen Nachrichten und Aktionen. Agenten werden als stromverarbeitende Funktionen mit benannten Kanälen definiert. Ein universeller Kompositionsoperator zur Verbindung gleichnamiger Kanäle wird eingeführt. Die Verwendung von Agenten höherer Ordnung wird motiviert. Neu ist eine Untersuchung über die Begriffe Sicherheit und Lebendigkeit bei Agenten sowie bei Nachrichten höherer Ordnung.

Kapitel 4 gibt eine formale Definition der verwendeten Entwurfsmethodik und ihrer Schritte. Die einzelnen Spezifikationsebenen werden formalisiert und somit die bei einer informellen Beschreibung wie [BDD+92a] immer auftretenden Problem- und Zweifelsfälle eindeutig gelöst. Ein Modell für den komponentenweisen Aufbau eines Systems (die *Systemstruktur*) wird neu eingeführt. Ein Verfeinerungsbegriff innerhalb und zwischen den Spezifikationsebenen wird formal definiert, und Beispiele für syntaktische Verfeinerungsregeln werden gegeben. Das Kapitel enthält den Beginn einer Systementwicklung als Beispiel.

Das Kapitel 5 befaßt sich mit zustandsorientierten Spezifikationstechniken, wie sie insbesondere in implementierungsnahen Entwurfsphasen verwendet werden. Wir beschreiben eine Folge von Automatenbegriffen, die sich zur schrittweisen Verfeinerung von Spezifikationen hin zu ausführbaren Programmen verwenden lassen. Verfeinerungsregeln für den Übergang zwischen den verschiedenen Automatenbegriffen und innerhalb eines Formalismus werden gegeben. Einige Probleme bei der Implementierung werden aufgezeigt. Ein Agent des im vorhergehenden Kapitel 4 beispielhaft entwickelten Systems wird in den verschiedenen Automatenbegriffen definiert. Ein Teil der in diesem Kapitel angegebenen Automatenbegriffe wurde in vereinfachter Form schon in der Fall-

studie [DW92a] verwendet. Die jetzigen Definitionen und Sätze und deren Anwendung sind neu.

Die Modellierung und methodische Verwendung von Zeit wird in Kapitel 6 behandelt. Dazu werden zunächst verschiedene in der Literatur zu findende Zeitbegriffe verglichen. Eine einfache Möglichkeit zur Zeitmodellierung wird genauer vorgestellt und der Kombinationsoperator aus Kapitel 3 entsprechend angepaßt. Möglichkeiten zur Spezifikation zeitbehafteter Bausteine werden beschrieben. Ein an die Definitionen von Kapitel 5 angelehnter neuer Automatenbegriff zur Spezifikation zeitbehafteter Agenten wird angegeben. Zwei weitere Agenten des Beispiels aus Kapitel 4 werden in einem zeitbehafteten Formalismus spezifiziert.

Im Kapitel 7 befassen wir uns mit der Modellierung allgemeiner dynamischer Netzwerke. Dazu wird zunächst ein einfaches funktionales Modell entwickelt, das auf den Zustandsautomatenbegriffen von Kapitel 5 basiert, wobei als Zustand eines dynamischen Netzes dessen augenblickliche Agentenstruktur betrachtet wird. Drei praxistypische Netzwerke werden als Beispiele spezifiziert. Möglichkeiten zur Erweiterung dieser Modellierung werden aufgezeigt. Verfeinerungsregeln für den Übergang von funktionalen Agentendefinitionen zu dynamischen Netzen und für die Entwicklung innerhalb der Ebene dynamischer Netze werden angegeben. Die Definitionen und Ergebnisse dieses Kapitels sind neu.

Schließlich faßt das Kapitel 8 die Arbeit zusammen und gibt Hinweise auf weiterführende Fragestellungen.

Kapitel 2

Grundlagen

In diesem Kapitel finden sich grundlegende Definitionen. Einige häufig bei Spezifikationen verwendete Bereiche, wie zum Beispiel die Bereiche der benannten Tupel oder der Ströme, werden als Sonderfälle von Funktionenbereichen definiert. Ein Satz von Basisfunktionen auf Strömen wird eingeführt.

2.1 Funktionen

Eine Funktion f zwischen den Mengen X und Y ist ein Tripel, dessen Komponenten die Mengen X und Y sowie eine ebenfalls mit f bezeichnete Teilmenge $f \subseteq X \times Y$ sind, wobei f die Eigenschaft $(x, y) \in f \wedge (x, y') \in f \Rightarrow y = y'$ für alle $x \in X$, $y, y' \in Y$ hat. Wir schreiben $\bullet f$ für X und $f\bullet$ für Y. Zwei Funktionen f und g sind genau dann gleich, wenn die Mengen f und g gleich sind und außerdem $\bullet f = \bullet g$ und $f\bullet = g\bullet$ gilt.

Die Anwendung einer Funktion $f \in X \to Y$ auf ein Argument $x \in X$ schreiben wir $f.x$ oder auch f_x. Die Subskriptschreibweise hat eine höhere Bindungsstärke als alle anderen Operatoren. Die Punktschreibweise ist links-assoziativ und hat die zweithöchste Bindungsstärke.

Zur Projektion der Komponenten eines Paares verwenden wir die beiden Funktionen $fst \in (X \times Y) \to X$ und $snd \in (X \times Y) \to Y$, die durch $fst.(x, y) = x$ beziehungsweise $snd.(x, y) = y$ definiert sind.

Prädikate auf einer Menge X betrachten wir wahlweise als Teilmengen von X oder als Funktionen aus $X \to \mathbb{B}$. Je nachdem, welche Schreibweise im Einzelfall günstiger ist, schreiben wir $P \subseteq X$ gleichbedeutend mit $P \in X \to \mathbb{B}$ und $P.x$ gleichbedeutend mit $x \in P$.

Als Grundfunktion verwenden wir die (eindeutige) Abbildung zwischen zwei einelementigen Mengen:

$$\begin{aligned} x \mapsto y \quad &\in \{x\} \to \{y\} \\ (x \mapsto y).x \; &= \; y \end{aligned}$$

Gegeben seien eine nicht-leere Namensmenge N und eine mit $n \in N$ indizierte Familie von Funktionen $f_n \in X_n \to Y_n$. Diese Funktionen können zu einer Funktion *vereinigt* werden, falls $f_n.x = f_m.x$ für alle $x \in X_n \cap X_m$ und $n, m \in N$ gilt. Das Ergebnis ist eine Funktion, die sich für jedes $x \in X_n$ so verhält wie f_n:

$$\begin{aligned} &\textstyle\bigcup_{n\in N} f_n \;\in\; \bigcup_{n\in N} X_n \to \bigcup_{n\in N} Y_n \\ &\forall\, x \in \textstyle\bigcup_{n\in N} X_n,\; n \in N : x \in X_n \Rightarrow (\bigcup_{n\in N} f_n).x = f_n.x \end{aligned}$$

Statt $\bigcup_{n\in N} f_n$ schreiben wir oft $\bigcup F$ mit $F = \{f_n : n \in N\}$. Außerdem schreiben wir auch $f \cup g$ statt $\bigcup\{f, g\}$.

Funktionen lassen sich in einer Tabellenschreibweise oft besonders übersichtlich definieren. In der einfachsten Form für Funktionen mit endlicher Definitionsmenge werden dazu die einzelnen Argumente mit den Ergebnissen der Funktionsapplikation tabellarisch in Beziehung gesetzt; siehe dazu Beispiel 4.1. Für kompliziertere Definitionen können Prädikate über den Funktionsargumenten mit Ausdrücken in Beziehung gesetzt werden, in denen gegebenenfalls die Funktionsargumente als freie Bezeichner vorkommen. Diese lokalen Bezeichner für die Funktionsargumente werden in der Kopfleiste der Tabelle eingeführt. Derartige Tabellen sind in den Bildern 5.2, 5.3 und 7.5 gezeigt. Weitere Anmerkungen zu erweiterten Tabellennotationen finden sich in [PW89].

Eine Funktion $f \in X \to Y$ kann auf eine Teilmenge ihrer ursprünglichen Definitionsmenge eingeschränkt werden. Für beliebige Mengen X' definieren wir:

$$\begin{aligned} f|_{X'} \quad &\in X \cap X' \to Y \\ (f|_{X'}).x \; &= f.x \quad \text{für} \quad x \in X \cap X' \end{aligned}$$

Die punktweise Veränderung der Werte einer Funktion $f \in X \to Y$ bezeichnen wir mit $f[x \mapsto y]$ für $x \in X$ und $y \in Y$:

$$\begin{aligned} f[x \mapsto y] \; &\in X \to Y \\ f[x \mapsto y] \; &= (x \mapsto y) \cup f|_{X\setminus\{x\}} \end{aligned}$$

Schließlich vereinbaren wir noch die Umbenennung der Eingabewerte einer Funktion $f \in X \to Y$. Von der oben definierten punktweisen Veränderung von Funktionswerten unterscheidet sich diese Operation erstens durch die veränderte Funktionalität und zweitens dadurch, daß in jedem Fall eine Anwendung von f erfolgt. Seien $x \in X$ und $x' \notin X$ gegeben. Dann definieren wir:

$$\begin{aligned} [x'/x]f \; &\in (X\setminus\{x\}) \cup \{x'\} \to Y \\ [x'/x]f \; &= (x' \mapsto f.x) \cup f|_{X\setminus\{x\}} \end{aligned}$$

2.2 Bereiche

Wir definieren den Begriff *Bereich* gemäß [Bro86, S. 12f], [GS90, S. 636] und [Plo83]:

Definition 2.1 (vollständige partielle Ordnung): Eine Menge X mit partieller Ordnung $\sqsubseteq$ heißt *Bereich,* wenn gilt:

- X enthält ein kleinstes Element $\bot$.
- Die kleinsten oberen Schranken aller gerichteten Teilmengen $Y \subseteq X$ sind in X enthalten.

Hierbei heißt eine Teilmenge Y *gerichtet,* wenn sie nicht leer ist und für alle $x, y \in Y$ ein $z \in Y$ existiert mit $x \sqsubseteq z$ und $y \sqsubseteq z$. Die kleinste obere Schranke einer gerichteten Teilmenge $Y \subseteq X$ bezeichnen wir mit $\bigsqcup Y$. □

In der Literatur finden sich auch Definitionen, in denen die Vollständigkeit von X bezüglich der kleinsten oberen Schranken aller abzählbar-unendlichen Ketten (total geordneten Teilmengen) verlangt wird [Mos90, S. 589f]. Laut [LS87, S. 88] ist dies eine im wesentlichen äquivalente Definition. Wenn man die Vollständigkeit von X bezüglich *aller* nicht-leeren Teilmengen fordert, erhält man das Konzept eines Halbverbandes.

Zusätzlich lassen sich weitere Endlichkeits- und Abschlußeigenschaften fordern, um der intuitiven Vorstellung von berechenbaren Elementen gerecht zu werden. Ein Element x eines Bereichs X nennen wir *endlich,* wenn für jede gerichtete Teilmenge Y mit $x \sqsubseteq \bigsqcup Y$ auch $x \sqsubseteq y$ für ein $y \in Y$ gilt. Die Menge der endlichen Elemente von X bezeichnen wir mit $\mathbb{E}.X$. Ein Bereich X heißt *algebraisch,* wenn für jedes Element $x \in X$ eine gerichtete Menge $Y \subseteq \mathbb{E}.X$ aus endlichen Elementen mit $x = \bigsqcup Y$ existiert.

Bereiche sind beispielsweise die um ein unendliches Element erweiterte Menge der natürlichen Zahlen $\mathbb{N} \cup \{\infty\}$ mit der kleiner-gleich-Ordnung $\leq$ oder die Potenzmenge $\wp\mathbb{N}$ der Menge $\mathbb{N}$ mit der Ordnung der Mengeninklusion $\subseteq$. Diese Bereiche sind auch algebraisch. Ein Beispiel für einen nicht-algebraischen Bereich findet sich in [Plo83]:

Beispiel 2.1: Die Menge $\mathbb{N} \cup \{\bot, ?, \infty\}$ mit der durch $\bot \sqsubseteq 0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty$ und $\bot \sqsubseteq ? \sqsubseteq \infty$ gegebenen Ordnung (bei der also ? mit keinem der Elemente aus $\mathbb{N}$ vergleichbar ist) ist ein nicht-algebraischer Bereich, denn das nicht-endliche Element ? läßt sich nicht durch endliche Elemente approximieren. □

Ob man algebraische Bereiche fordert, macht in dieser Arbeit nur für Abschnitt 3.8 einen Unterschied.

Eine Menge X mit $\bot \notin X$ und partieller Ordnung $\sqsubseteq$ kann um das Element $\bot$ als kleinstes Element erweitert werden vermöge:

$$\begin{aligned} X^{\bot} &= X \cup \bot \\ x \sqsubseteq^{\bot} y &= (x = \bot \vee x \sqsubseteq y) \end{aligned}$$

Die Menge $X^\perp$ mit Ordnung $\sqsubseteq^\perp$ erfüllt im allgemeinen nicht die für einen Bereich geforderte Vollständigkeitseigenschaft. Für jede Menge X ist jedoch $X^\perp$ mit Ordnung $=^\perp$ ein Bereich, der sogenannte *flache Bereich.*

Gegeben seien eine Namensmenge N und eine Familie von Mengen X_n mit Ordnungen $\sqsubseteq_n$ für $n \in N$. Daraus läßt sich die Menge der *benannten Tupel* mit der punktweisen Ordnung $\sqsubseteq^\otimes$ bilden. Wir bezeichnen diese Menge mit $\otimes_{n\in N}\, n\!:\!X_n$. Für ein benanntes Tupel $u \in \otimes_{n\in N}\, n : X_n$ ist $\bullet u$ die Menge N der Komponentennamen und $u.n$ der Wert der Komponente mit Namen n. Die Veränderung des Wertes einer Komponente erfolgt über die in Abschnitt 2.1 definierten Operatoren zur punktweisen Änderung einer Funktion. Benannte Tupel sind als Funktionen definiert vermöge:

$$\begin{aligned} \otimes_{n\in N}\ n\!:\!X_n &= \{u \in N \to (\textstyle\bigcup_{n\in N} X_n) : \forall\, n \in N : u.n \in X_n\} \\ u \sqsubseteq^\otimes v &= \forall\, n \in N : u.n \sqsubseteq_n v.n \end{aligned}$$

Wir schreiben einfacher auch $n_1 : X_{n_1} \otimes\ n_2 : X_{n_2}$ statt $\otimes_{n\in\{n_1,n_2\}}\, n : X_n$ und $\otimes\, n_1 : X_{n_1}$ statt $\otimes_{n\in\{n_1\}}\, n\!:\!X_n$.

Die Menge $\otimes_{n\in N}\, n : X_n$ mit Ordnung $\sqsubseteq^\otimes$ ist ein Bereich, wenn für alle $n \in N$ die Menge X_n mit der Ordnung $\sqsubseteq_n$ ein Bereich ist. In diesem Fall und für $|N| < \infty$ gilt bezüglich der endlichen Objekte:

$$\mathbb{E}.(\otimes_{n\in N}\, n\!:\!X_n) = \{u \in (\otimes_{n\in N}\, n\!:\!X_n) : \forall\, n \in N : u.n \in \mathbb{E}.X_n\}$$

Im Einklang mit der in dieser Arbeit generell verwendeten funktionalen Sichtweise führen wir Ströme als Funktionen von natürlichen Zahlen auf Stromelemente ein. Aus einer Menge X mit Ordnung $\sqsubseteq$ läßt sich die Menge X^ω der *endlich und unendlich langen Sequenzen (Ströme)* mit der punktweisen Ordnung $\sqsubseteq^\omega$ bilden vermöge:

$$\begin{aligned} X^\omega &= \{s \in \mathbb{N} \to X^\perp : \forall\, n,\, m \in \mathbb{N} : (n < m \wedge s.n = \perp) \Rightarrow s.m = \perp\} \\ s \sqsubseteq^\omega t &= \forall\, n \in \mathbb{N} : s.n \sqsubseteq^\perp t.n \end{aligned}$$

Die Menge X^ω mit $\sqsubseteq^\omega$ ist ein Bereich, wenn $X^\perp$ mit $\sqsubseteq^\perp$ ein Bereich ist. Als *Strombereich* bezeichnen wir jeden Bereich der Form X^ω. Wir schreiben $Msg.(X^\omega)$, um die Grundmenge X der in den Strömen zulässigen Nachrichten zu bezeichnen. Diese Schreibweise ist wohldefiniert; es gilt $Msg.S = \bigcup_{s\in S}(s\bullet)\backslash\{\perp\}$ für jeden Strombereich S.

Eine Teilmenge eines Strombereichs X^ω ist die Menge der *endlich langen Sequenzen* X^* definiert durch $X^* = \{s \in X^\omega : \exists\, n \in \mathbb{N} : s.n = \perp\}$. Eine endliche Sequenz s mit $n+1$ Elementen schreiben wir gewöhnlich $\langle s.0,\, ...,\, s.n\rangle$. Die leere Sequenz wird durch $\langle\rangle$ bezeichnet.

Die Menge der endlichen Objekte von X^ω ist die Menge der endlich langen Sequenzen, in denen außerdem keine unendlichen Elemente auftauchen:

$$\mathbb{E}.X^\omega = \{s \in X^* : \forall\, n \in \mathbb{N} : s.n \neq \perp \Rightarrow s.n \in \mathbb{E}.X\}$$

Neben der punktweisen Ordnung $\sqsubseteq^\omega$ definieren wir auf Strommengen X^ω noch die Präfixordnung $\leq^\omega$ vermöge:

$$s \leq^\omega t = \forall\, n \in \mathbb{N} : (s.n = \bot \vee s.n = t.n)$$

Offensichtlich fallen $\sqsubseteq^\omega$ und $\leq^\omega$ genau dann zusammen, wenn die Ordnung $\sqsubseteq$ auf den definierten Objekten die Gleichheit $=$ ist.

Im folgenden seien A und B Bereiche mit den Ordnungen $\sqsubseteq_A$ und $\sqsubseteq_B$. Wir definieren zunächst die Begriffe Monotonie und Stetigkeit:

Definition 2.2 (Monotonie, Stetigkeit): Eine Funktion $f \in A \to B$ heißt *monoton,* wenn für alle a, $a' \in A$ mit $a \sqsubseteq_A a'$ gilt, daß $f.a \sqsubseteq_B f.a'$.

Eine monotone Funktion $f \in A \to B$ heißt *stetig,* wenn für alle gerichteten Teilmengen A' von A gilt, daß $f.(\bigsqcup A') = \bigsqcup\{f.a : a \in A'\}$. □

Eine stetige Funktion ist monoton. Die Umkehrung gilt im allgemeinen nicht. Die Menge der stetigen Funktionen zwischen zwei Bereichen A und B bezeichnen wir mit $A \twoheadrightarrow B$. Zusammen mit der punktweisen Ordnung $\sqsubseteq^{\twoheadrightarrow}$ ergibt sich wieder ein Bereich:

$$\begin{aligned} A \twoheadrightarrow B &= \{f \in A \to B : f \text{ ist stetig}\} \\ f \sqsubseteq^{\twoheadrightarrow} g &= \forall\, a \in A : f.a \sqsubseteq_B g.a \end{aligned}$$

Die beiden Pfeiloperatoren $\to$ und $\twoheadrightarrow$ sind rechts-assoziativ. Wenn keine Verwechslungsgefahr besteht, bezeichnen wir die Ordnung eines Bereiches im folgenden stets mit dem Symbol $\sqsubseteq$, das kleinste Element eines Bereiches mit $\bot$ und die Präfixordnung auf Strombereichen mit $\leq$.

2.3 Funktionen auf Strömen

Wir führen folgende Funktionen auf Strömen ein:

- Ein Element x vor einen Strom s gesetzt. Das Ergebnis von $\bot \,\&\, s$ ist der leere Strom:

$$\begin{aligned} \_\&\_ \quad &\in X^\bot \to X^\omega \twoheadrightarrow X^\omega \\ (x \,\&\, s).n &= x \qquad\quad \text{wenn} \quad x = \bot \vee n = 0 \\ &= s.(n-1) \quad \text{sonst} \end{aligned}$$

- Zwei Ströme hintereinandergehängt:

$$\begin{aligned} \_\circ\_ \quad &\in X^\omega \to X^\omega \twoheadrightarrow X^\omega \\ (s \circ t).n &= s.n \qquad\quad \text{wenn} \quad n < \#s \\ &= t.(n - \#s) \quad \text{sonst} \end{aligned}$$

- Der Strom, der aus dem m-fach wiederholten Element x besteht:

$$\begin{array}{lll} \_^{-} & \in X \to \mathbb{N} \cup \{\infty\} \twoheadrightarrow X^{\omega} & \\ (x^{m}).n & = x \quad \text{wenn} \quad n < m & \\ & = \bot \quad \text{sonst} & \end{array}$$

- Das erste Element eines Stromes, oder $\bot$, falls der Strom leer ist:

$$\begin{array}{lll} ft & \in X^{\omega} \twoheadrightarrow X^{\bot} & \\ ft.s & = s.0 & \end{array}$$

- Der Rest eines Stromes ohne das erste Element, oder $\langle\rangle$, falls der Strom leer ist:

$$\begin{array}{lll} rt & \in X^{\omega} \twoheadrightarrow X^{\omega} & \\ rt.s.n & = s.(n+1) & \end{array}$$

- Die Länge eines Stromes:

$$\begin{array}{lll} \#\_ & \in X^{\omega} \to \mathbb{N} \cup \{\infty\} & \\ \#s & = 0 & \text{wenn} \quad s = \langle\rangle \\ & = \infty & \text{wenn} \quad \forall\, n \in \mathbb{N} : s.n \neq \bot \\ & = 1 + max\{n \in \mathbb{N} : s.n \neq \bot\} & \text{sonst} \end{array}$$

- Die elementweise Anwendung einer Funktion auf einen Strom:

$$\begin{array}{lll} \_*\_ & \in (X \to X) \to X^{\omega} \to X^{\omega} & \\ (f * x).n & = f.(x.n) \quad \text{wenn} \quad x.n \neq \bot & \\ & = \bot \qquad \text{sonst} & \end{array}$$

- Der Strom, der durch das Filtern eines Stroms entsteht:

$$\begin{array}{lll} \_\textcircled{c}\_ & \in \wp X \to X^{\omega} \to X^{\omega} & \\ Y\,\textcircled{c}(x \,\&\, s) & = x \,\&\, (Y\,\textcircled{c}s) \quad \text{wenn} \quad x \in Y \cup \{\bot\} & \\ & = Y\,\textcircled{c}s \qquad \text{sonst} & \end{array}$$

Kapitel 3

Spezifikationen

Dieses Kapitel führt für das Erstellen von Spezifikationen wesentliche Begriffe und Operationen ein. Es werden benannte *Kanäle* zur Verbindung der einzelnen *Agenten* und Operationen zur *Agentenkomposition* über Kanalnamen definiert. Die *Spezifikation* von Agenten erfolgt über *Spuren* von Aktionen oder über Prädikate auf Funktionen. Anwendungen für Agenten auf Objekten *höherer Ordnung* werden vorgestellt. Formalisierungen der Begriffe *Sicherheit* und *Lebendigkeit* für funktionale Agenten, auch für Agenten höherer Ordnung, werden gegeben.

3.1 Kanäle und Aktionen

Einen *Kanal* kann man sich als eine benannte Leitung vorstellen, über die Nachrichten aus einer vorgegebenen Menge fließen. Als *Kanalbündel* bezeichnen wir eine Familie von Kanälen, denen jeweils ein Name und eine Menge möglicher Nachrichten zugeordnet sind. Für ein Kanalbündel ist ein *Kanalzustand* eine Kommunikationsgeschichte, also eine Zuordnung von den Kanalnamen des Bündels zu den Strömen der bislang auf diesen Kanälen übertragenen Nachrichten. Die Menge aller für ein Kanalbündel möglichen Kanalzustände nennen wir einen *Kanaltyp.*

Definition 3.1 (Kanaltyp, Kanalzustand): Sei N eine höchstens abzählbar unendliche Menge von Kanalnamen. Der Kanaltyp eines Kanalbündels mit $|N|$ Kanälen, die mit den Elementen aus N benannt sind und denen jeweils eine Nachrichtenmenge M_n (für $n \in N$) zugeordnet ist, ist die Menge $\otimes_{n \in N} n : (M_n)^\omega$. Ein Kanalzustand ist jedes Element eines Kanaltyps. □

Bei einem Kanaltyp S bezeichnet also $\bullet s$ für jedes $s \in S$ die Menge der Kanalnamen. Dafür schreiben wir auch $\bullet S$; es gilt $\bullet S = \bigcup_{s \in S} \bullet s$. Den Bereich der auf einem Kanal $c \in \bullet S$ möglichen Nachrichtenströme bezeichnen wir mit $Str_S.c$; hierfür gilt die

Beziehung $Str_S.c = \{s.c : s \in S\}$. Offensichtlich ist (für die Funktion *Msg* aus Abschnitt 2.2) $Msg.(Str_S.c)$ die Menge der möglichen Nachrichten auf dem Kanal c, also die Menge M_n in Definition 3.1 für $n = c$.

Für jeden Kanaltyp S bezeichnen wir mit S^{fin} die im folgenden definierte Menge derjenigen Kanalzustände, in denen nur endlich lange Ströme auftreten. Die Menge S^{fin} ist kein Bereich:

$$S^{fin} = \{s \in S : \forall\, c \in \bullet S : \#s.c < \infty\}$$

Die Präfixordnung $\leq$ auf Strömen wird auf Kanalzustände $s, s' \in S$ erweitert vermöge der Definition:

$$s \leq s' = \forall\, c \in \bullet S : s.c \leq s'.c$$

Zur Umbenennung von Kanalnamen dient der in Abschnitt 2.1 eingeführte Operator $[\_/\_]\_$. Gegeben seien der Kanalzustand $s \in S$ und $c \in \bullet S$, $c' \notin \bullet S$. Dann ist $[c'/c]s$ der Kanalzustand, der aus s durch Umbenennung des Kanals c in c' hervorgeht.

Wir unterscheiden Nachrichten von *Aktionen.* Eine Aktion ist ein Paar bestehend aus einem Kanalnamen c und einer Nachricht m, die auf diesem Kanal übertragen werden kann, also in der dem Kanal c durch den Kanaltyp zugeordneten Nachrichtenmenge enthalten ist. Man kann sich eine Aktion als Beobachtung des Nachrichtenaustausches vorstellen; genauer gesagt als Beobachtung, daß die Nachricht m auf dem Kanal c für den Empfänger verfügbar wird. Diese Beobachtung ist sowohl vom Senden als auch vom tatsächlichen Empfang der Nachricht zu unterscheiden, die beide nicht explizit modelliert werden.

Definition 3.2 (Aktionen): Gegeben sei ein Kanaltyp S. Dann definieren wir die Menge von Aktionen von S vermöge:

$$Act.S = \bigcup_{c \in \bullet S} \{(c, m) : m \in Msg.(Str_S.c)\} \qquad \square$$

Für alle Kanaltypen S definieren wir folgende Operationen auf Kanalzuständen und Aktionen:

- Den leeren Kanalzustand ε_S (oder auch ε, wenn S klar ist):

$$\begin{array}{lll} \varepsilon_S & \in S & \\ \varepsilon_S.c & \in \langle\rangle & \text{für } \ c \in \bullet S \end{array}$$

- Die (minimale) Länge der Ströme eines Kanalzustands:

$$\begin{array}{lll} \#\_ & \in S \to \mathbb{N} \cup \{\infty\} & \\ \#s & = \infty & \text{wenn } \ \bullet S = \varnothing \\ & = min\{\#s.c : c \in \bullet S\} & \text{sonst} \end{array}$$

- Umwandlung einer Aktion in den entsprechenden Kanalzustand:

$$\begin{aligned} \bar{\_} &\in Act.S \to S \\ \bar{a}.c &= \langle snd.a \rangle \quad \text{wenn} \quad fst.a = c \\ &= \langle\rangle \qquad\quad \text{sonst} \end{aligned}$$

- Zwei Kanalzustände gleichen Kanaltyps hintereinandergehängt:

$$\begin{aligned} &\_ \circ \_ \in S \to S \twoheadrightarrow S \\ &s \circ t = \bigcup_{c \in \bullet S} (c \mapsto s.c \circ t.c) \end{aligned}$$

- Eine Filter-Operation, die aus einer Aktionsspur die auf einem Kanal übertragenen Nachrichten aus einer gegebenen Nachrichtenmenge extrahiert:

$$\begin{aligned} \_ \textbf{ on } \_ \textbf{ in } \_ &\in \wp Messages \to \bullet S \to (Act.S)^\omega \to Messages^\omega \\ &\quad \text{für} \quad Messages = \bigcup_{c \in \bullet S} Msg.(Str_S.c) \\ M \textbf{ on } c \textbf{ in } t &= snd * (\{(c, m) : m \in M\} \copyright t) \end{aligned}$$

3.2 Agenten

Agenten sind die Grundbausteine der Systemmodellierung. Formal ist ein Agent eine stetige Funktion vom Kanaltyp der Eingabe auf den der Ausgabe. Dies entspricht einer stromverarbeitenden Funktion mit benannten Ein- und Ausgabekanälen. Wir verwenden in dieser Arbeit durchgehend benannte Kanäle, weil Kanalnamen für die Spezifikation der einzelnen Agenten hilfreich sind und darüberhinaus die Verbindungsstruktur von Agentennetzen, wie in Abschnitt 3.3 gezeigt werden wird, bequem über die Kanalnamen definiert werden kann.

Definition 3.3 (Agenten): Gegeben seien zwei Kanaltypen IS und OS. Ein *Agent* ist eine stetige Funktion $f \in IS \twoheadrightarrow OS$, wobei IS den Kanaltyp der Eingabe und OS den der Ausgabe bezeichnet. □

Für die Menge derjenigen Agenten von IS nach OS, die stets nur endlich lange Ausgaben liefern, schreiben wir $IS \twoheadrightarrow OS^{fin}$ als Abkürzung für die Menge $\{f \in IS \twoheadrightarrow OS : \forall\, x \in IS : f.x \in OS^{fin}\}$.

Gegeben sei ein Agent $f \in IS \twoheadrightarrow OS$ erster Ordnung, also ein Agent, auf dessen Ein- und Ausgabekanälen die Approximationsordnung $\sqsubseteq$ mit der Präfixordnung $\leq$ zusammenfällt. Eine Eingabe $s \in IS$ wirkt auf f in zweierlei Hinsicht: Erstens wird eine Ausgabe $f.s \in OS$ erzeugt, und zweitens verändert sich im allgemeinen die Reaktion des Agenten auf weitere Eingaben. Man kann dies so auffassen, daß nach der Verarbeitung der Eingabe s aus dem Agenten f ein neuer Agent entstanden ist. Diesen durch die folgende Gleichung definierten Agenten $f \ll s$ nennen wir die *Resumption von f nach*

dem Einlesen von s. Die Ausgabe dieses Agenten ist für jede Eingabe $x \in IS$ gleich der zusätzlichen Ausgabe des Agenten f für die zusätzliche Eingabe x:

$$\begin{aligned} &f \ll s \ \in \ IS \twoheadrightarrow OS \\ &f.s \circ (f \ll s).x \ = \ f.(s \circ x) \end{aligned}$$

Der Resumptionsoperator $\ll$ ist nur für Agenten erster Ordnung wohldefiniert. Betrachten wir zum Beispiel den Agenten $f \in \otimes\, i : \mathbb{N}^\omega \twoheadrightarrow \otimes\, o : (\mathbb{N}^\omega)^\omega$ mit $f.(i \mapsto x) = o \mapsto \langle x \rangle$. Es seien $s = (i \mapsto \langle\rangle)$ und $z = (i \mapsto \langle 1 \rangle)$. Für den Agenten f gilt $f.s.o = \langle\langle\rangle\rangle$ und $f.(s \circ z).o = \langle\langle 1 \rangle\rangle$, also gibt es kein $f \ll s$ mit $f.s \circ (f \ll s).z = f.(s \circ z)$.

Im folgenden geben wir Operationen zum Umbenennen von Kanälen eines Agenten sowie zum Einschränken des Kanalnamensraumes an. Zum Umbenennen benötigen wir je eine Operation für die Ein- beziehungsweise Ausgabekanäle. Diese Unterscheidung ist notwendig, weil Agenten gleichnamige Ein- und Ausgabekanäle aufweisen können.

Gegeben seien ein Agent $f \in IS \twoheadrightarrow OS$, ein $i \in \bullet IS$ und ein $i' \notin \bullet IS$. Wir verwenden die (überladene) Notation $[i'/i]f$ für den Agenten, der sich aus f ergibt, wenn man den Eingabekanal i in i' umbenennt:

$$\begin{aligned} &[i'/i]f \quad \ \in \{[i'/i]s : s \in IS\} \twoheadrightarrow OS \\ &([i'/i]f).x \ = \ f.([i/i']x) \end{aligned}$$

Gegeben seien ein Agent $f \in IS \twoheadrightarrow OS$, ein $o \in \bullet OS$ und ein $o' \notin \bullet OS$. Wir verwenden die Notation $f[o'/o]$ für den Agenten, der sich ergibt, wenn man den Ausgabekanal o in o' umbenennt:

$$\begin{aligned} &f[o'/o] \quad \ \in IS \twoheadrightarrow \{[o'/o]s : s \in OS\} \\ &(f[o'/o]).x \ = \ [o'/o](f.x) \end{aligned}$$

Schließlich betrachten wir die Restriktion eines Agenten auf eine Teilmenge der Ein- und Ausgabekanäle. Seien $f \in IS \twoheadrightarrow OS$, $I \subseteq \bullet IS$ und $O \subseteq \bullet OS$. Weiterhin seien die Ausgaben des Agenten auf den Kanälen O unabhängig von den Eingaben auf den Kanälen $\bullet IS \backslash I$, also für alle $x, y \in IS$ mit $x|_I = y|_I$ gelte $(f.x)|_O = (f.y)|_O$. Dann definieren wir:

$$\begin{aligned} &f|^I_O \quad \ \in \{s|_I : s \in IS\} \twoheadrightarrow \{s|_O : s \in OS\} \\ &(f|^I_O).x \ = \ (f.x')|_O \quad \text{wobei} \ \ x' \in IS \ \ \text{so daß} \ \ x'|_I = x \end{aligned}$$

3.3 Agentenkomposition

Für die Beschreibung der Verbindungsstruktur eines Netzwerkes gibt es zwei grundlegende Möglichkeiten. Das Netz kann aus einzelnen Agenten entweder über eine Anzahl die Verbindungsstruktur festlegender Kompositionsoperatoren zusammengesetzt werden, oder durch einen Operator, der gleichbenannte Kanäle verbindet.

Im ersten Fall können als Kompositionsoperatoren zum Beispiel parallele und sequentielle Komposition sowie ein Rückkopplungsoperator verwendet werden, wie in [Bro86] dargestellt. Weiter werden noch einige Grundfunktionen zur Leitungsführung benötigt. Insbesondere einfache, regelmäßige Strukturen lassen sich damit elegant darstellen. Außerdem kann man einen Beweiskalkül angeben, dessen Regeln die Struktur der Kompositionsoperatoren widerspiegeln, so daß der Beweis einer Eigenschaft eines Netzes in seiner Struktur entsprechend der Netzstruktur aufgebaut werden kann.

Die Verbindung von Agenten über Kanalnamen hat dagegen den Vorteil, daß sie bei komplex aufgebauten Netzen zu übersichtlicheren Ergebnissen führt. Ein oftmals schwer verständlicher Ausdruck zur Angabe der Netzstruktur entfällt. Die Komplexität der Definition der Netzstruktur hängt nur von der Anzahl der Agenten ab und nicht davon, ob die Netzstruktur zu den Verbindungsoperatoren paßt oder nicht. Es lassen sich Agenten mit (potentiell) unendlich vielen Ein- und Ausgängen definieren und miteinander verbinden.

Der in der folgenden Definition 3.4 vereinbarte universelle Kombinationsoperator für Agenten verbindet gleichnamige Kanäle miteinander. Dazu wird zunächst die Funktion $\hat{\|}_{n\in N} f_n$ eingeführt, die formal die Ein- und Ausgabekanäle aller Agenten f_n aufweist. Von außen kommende Nachrichten auf einem Eingabekanal, der intern an einen Ausgang gleichen Namens angeschlossen ist, werden ignoriert. Dies kommt in der Definition des Wertes x' zum Ausdruck, der die von außen zugänglichen Eingabekanäle (mit Namen aus $I \setminus O$) und die internen rückgekoppelten Kanäle (mit Namen aus $I \cap O$) zusammenfaßt.

Definition 3.4 (Kombination von Agenten): Gegeben seien eine höchstens abzählbar unendliche Namensmenge N und Agenten $f_n \in IS_n \twoheadrightarrow OS_n$ für $n \in N$. Für alle $n, m \in N$ müssen die Agenten *kombinierbar* sein, das heißt folgende Eigenschaften aufweisen:

- disjunkte Ausgabekanäle: $\bullet OS_n \cap \bullet OS_m = \varnothing$
- kompatible Eingabekanäle: $\forall\, c \in \bullet IS_n \cap \bullet IS_m : Str_{IS_n}.c = Str_{IS_m}.c$
- kompatible Ein- und Ausgabekanäle: $\forall\, c \in \bullet IS_n \cap \bullet OS_m : Str_{IS_n}.c = Str_{OS_m}.c$

Abkürzend schreiben wir I für $\bigcup_{n\in N} \bullet IS_n$ und O für $\bigcup_{n\in N} \bullet OS_n$. Wir definieren $\hat{\|}_{n\in N} f_n$ als die kleinste Funktion, die die folgende Gleichung erfüllt:

$$\hat{\|}_{n\in N} f_n \in \otimes_{c\in I}\, c{:}\bigcup_{n\in\{m\in N:\, c\in\bullet IS_m\}} Str_{IS_n}.c \twoheadrightarrow \otimes_{c\in O}\, c{:}\bigcup_{n\in\{m\in N:\, c\in\bullet OS_m\}} Str_{OS_n}.c$$
$$(\hat{\|}_{n\in N} f_n).x \;=\; \bigcup_{n\in N} f_n.(x'|_{\bullet IS_n}) \quad \text{wobei} \quad x' = x|_{I\setminus O} \cup ((\hat{\|}_{n\in N} f_n).x)|_{I\cap O}$$

Die Kombination $\|_{n\in N} f_n$ ergibt sich, indem man intern angeschlossene Eingabekanäle abkapselt, vermöge:

$$\|_{n\in N} f_n \;=\; (\hat{\|}_{n\in N} f_n)|^{I\setminus O}_{O\setminus I} \qquad \square$$

Die Existenz einer die Bedingungen von Definition 3.4 erfüllenden Funktion $\hat{\|}_{n\in N} f_n$ folgt, wie allgemein bei rekursiven Definitionen, aus dem Satz von Knaster/Tarski [Tar55]. Zur Anwendung dieses Satzes betrachten wir die folgende Hilfsfunktion h:

$$h.f.x = \bigcup_{n\in N} f_n.(x'|_{\bullet IS_n}) \quad \text{wobei} \quad x' = x|_{I\setminus O} \cup (f.x)|_{I\cap O}$$

Die Funktion h ist monoton, weshalb sie nach dem genannten Satz einen kleinsten Fixpunkt hat. Dieser ist die Funktion $\hat{\|}_{n\in N} f_n$ von Definition 3.4.

Wiederum verwenden wir die Schreibweisen $\|\{f_n : n \in N\}$ statt $\|_{n\in N} f_n$ sowie $f\|g$ statt $\|\{f, g\}$ und $\|f$ statt $\|\{f\}$.

Die Abschottung von Eingangskanälen, die intern mit einem Ausgang verbunden sind, nach außen verhindert, daß zwei Ausgangskanäle zusammengeschaltet werden können.

Gegeben seien zwei Agenten $f \in IS \twoheadrightarrow OS$ und $g \in IS' \twoheadrightarrow OS'$, jeweils mit disjunkten Ein- und Ausgabekanälen (also $\bullet IS \cap \bullet OS = \varnothing$ und $\bullet IS' \cap \bullet OS' = \varnothing$). Für solche Agenten gilt übrigens $\|f = f$ beziehungsweise $\|g = g$. Offensichtlich erhält man die klassischen Verbindungsoperatoren der sequentiellen und parallelen Komposition sowie die Rückkopplung als Spezialfälle des Operators $\|$. Die sequentielle Komposition der Agenten f und g ergibt sich als $f\|g$, wenn $OS = IS'$ und $\bullet IS \cap \bullet OS' = \varnothing$ gilt. Die parallele Komposition ist $f\|g$ für $\bullet IS \cap (\bullet IS' \cup \bullet OS') = \varnothing$ und $\bullet OS \cap (\bullet IS' \cup \bullet OS') = \varnothing$. Die Rückkopplung eines Kanals c erhält man für einen Agenten $h \in IS \twoheadrightarrow OS$ mit $\bullet IS \cap \bullet OS = \{c\}$ durch $\|h$.

Der Operator $\|$ ist idempotent, wenn man von der notwendigen Bildung der einelementigen Ergebnismenge absieht:

Satz 3.1: Es gilt $\|\{\|F\} = \|F$ für kombinierbare Agentenmengen F.

Beweis: Offensichtlich ist $\|F$ ein Agent mit disjunkten Ein- und Ausgabekanälen. Für einelementige Mengen solcher Agenten folgt aus Definition 3.4 $\|\{\|F\} = \hat{\|}\{\|F\} = \hat{\|}F = \|F$. □

3.4 Spuren von Agenten

Spuren sind Sequenzen von Aktionen. Eine Spur repräsentiert die Beobachtung eines Ablaufs. Auch einem Agenten läßt sich eine Spurmenge zuordnen, nämlich die Menge der Beobachtungen von Abläufen des Agenten für alle Eingaben und alle zeitlichen Verzahnungen der Ein- und Ausgaben auf verschiedenen Kanälen. Um die Spuren eines Agenten formal zu definieren, benötigen wir zunächst eine Operation, welche aus einer Aktionsspur einen Kanalzustand erzeugt, der dem durch die Spur beschriebenen Ablauf entspricht. Gegeben seien ein Kanaltyp S' und eine Aktionsspur $t \in (Act.S')^\omega$.

Weiterhin sei ein Kanaltyp S gegeben, der sich zu S' erweitern läßt (das heißt, daß $s|_{\bullet S} \in S$ für alle $s \in S'$ gilt). Wir definieren $S \circledS t$ als denjenigen Kanalzustand, der den Kanaltyp S hat und eine t entsprechende Kanalbelegung repräsentiert:

$$\begin{array}{lcl} S \circledS t & \in & S \\ (S \circledS t).c & = & snd * (\{(c, m) : m \in Msg.(Str_S.c)\} \circledC t) \end{array}$$

In der folgenden Definition 3.5 drückt das Konjunktionsglied $OS \circledS t = f.(IS \circledS t)$ aus, daß Spuren immer vollständige Abläufe darstellen. Ausgaben dürfen zwar beliebig, aber nicht unendlich lange verzögert werden. Das Konjunktionsglied $\forall\, s \leq t : OS \circledS s \leq f.(IS \circledS s)$ besagt, daß Ausgaben in einer Spur erst nach den zur Erzeugung der Ausgabe notwendigen Eingaben auftreten dürfen. Weil Spuren eine globale Sicht modellieren, müssen die Namen der Ein- und Ausgabekanäle disjunkt sein.

Definition 3.5 (Spuren eines Agenten): Gegeben sei ein Agent $f \in IS \twoheadrightarrow OS$, für den $\bullet IS \cap \bullet OS = \varnothing$ gilt. Wir definieren die von f erzeugten Spuren $Traces.f$ vermöge:

$$\begin{array}{lcl} Traces.f & = & \{t \in (Act.IS \cup Act.OS)^{\omega} \ : \\ & & \quad OS \circledS t = f.(IS \circledS t) \ \wedge \\ & & \quad \forall\, s \in (Act.IS \cup Act.OS)^{*} : s \leq t \ \Rightarrow \ OS \circledS s \leq f.(IS \circledS s)\} \end{array} \qquad \square$$

3.5 Agentenspezifikationen

Funktionale Agentenspezifikationen sind Prädikate auf Agenten, also auf stromverarbeitenden Funktionen. Im Gegensatz zu den sonst oft verwendeten relationalen Spezifikationen, bei denen eine Relation zwischen den Ein- und Ausgaben angegeben wird, vermeidet diese Modellierung Probleme bei Netzwerken mit Rückkopplung [Bro88a]. Überdies ist es für Beweise in der Praxis vorteilhaft, daß jede Funktion einem *deterministischen* Agenten entspricht. Eine Agentenspezifikation hat also die Form $F \in (IS \twoheadrightarrow OS) \to \mathbb{B}$ beziehungsweise, damit gleichbedeutend, $F \subseteq IS \twoheadrightarrow OS$.

Die in Abschnitt 3.2 eingeführten Operationen auf Agenten wenden wir auch elementweise auf Agentenspezifikationen an vermöge:

$$\begin{array}{ll} [i'/i]F & = \{[i'/i]f : f \in F\} \\ F[o'/o] & = \{f[o'/o] : f \in F\} \\ F|_O^I & = \{f|_O^I : f \in F\} \\ \|_{n \in N}\ F_n & = \{\|_{n \in N} f_n : (\forall\, n \in N : f_n \in F_n)\} \end{array}$$

Ebenso erweitern wir den in Abschnitt 3.4 definierten Operator *Traces* auf Agentenspezifikationen vermöge:

$$Traces.F = \bigcup_{f \in F} Traces.f$$

3.6 Agenten auf Objekten höherer Ordnung

Die bisher in diesem Kapitel angegebenen Definitionen erlauben es, Agenten auf Strömen höherer Ordnung zu spezifizieren. In diesem Abschnitt soll geklärt werden, in welchen Fällen eine solche Möglichkeit vorteilhaft ist.

Agenten, also laufende Prozesse, werden in FOCUS generell durch Funktionen modelliert. Wenn wir beispielsweise das Betriebssystem *UNIX* betrachten, ist ein einfaches Programm, das nicht auf das Dateisystem oder andere Systemkomponenten zugreift oder diese ändert, ein Element der Menge *Program*:

$$Program = \otimes\, stdin : Char^{\omega} \twoheadrightarrow stdout : Char^{\omega} \otimes\ stderr : Char^{\omega}$$

Es liegt nahe, nicht nur gerade laufende, sondern alle *lauffähigen* Programme durch Funktionen zu modellieren. Wir ordnen also den nebenwirkungsfreien *UNIX*-Programmen generell den Typ *Program* zu, unabhängig davon, ob sie gerade ausgeführt werden oder im Rechner in Form von (Binär-)Dateien vorliegen.

Wenn lauffähige Programme als Funktionen dargestellt werden, sind natürlich Programme, die andere Programme erzeugen oder verarbeiten, Funktionen höherer Ordnung. Ein typisches Beispiel sind Compiler, die Programmtexte einlesen und Programme generieren. Ein Compiler wird zum Beispiel als Element der Menge *Compiler* modelliert:

$$Compiler = \otimes\, in : (Char^{\omega})^{\omega} \twoheadrightarrow \otimes\, out : Program^{\omega}$$

Andere Anwendungen für Funktionen höherer Ordnung finden sich bei der Modellierung von Betriebssystemkomponenten, insbesondere solchen Komponenten, die die Abarbeitung von Anwenderprogrammen steuern.

Auch Teile der Benutzeroberfläche werden sinnvollerweise mit Funktionen höherer Ordnung modelliert. Wenn wir eine typische *UNIX-Shell* betrachten, dient der sogenannte *pipe*-Operator $\_|\_$ dazu, einzelne Programme hintereinanderzuschalten. Eine formale Modellierung dieses Operators ist durch die im folgenden definierte Funktion gegeben:

$$\begin{aligned} \_|\_ \quad &\in Program \times Program \to Program \\ (f|g).s &= (f.s)[stdout \mapsto \langle\rangle] \circ g.(stdin \mapsto f.s.stdout) \end{aligned}$$

Funktionen höherer Ordnung sind auch nötig, wenn als Daten nicht Programme, sondern andere komplexe Objekte, wie zum Beispiel Ein- und Ausgabeströme, auftreten. Eine *UNIX*-Gerätedatei wie */dev/tty* kann als Agent modelliert werden, der einen Befehlsstrom erhält und als Reaktion auf die darin enthaltenen Befehle einen Strom von Antworten liefert, der also zum Beispiel ein Element der Menge *Device* ist:

$$Device = \otimes\, cmd : Char^{\omega} \twoheadrightarrow \otimes\, rply : Char^{\omega}$$

Wenn solche Gerätedateien nicht nur an ein Programm "fest angeschlossen", sondern als Objekte verarbeitet werden sollen, benötigt man wiederum Funktionen höherer Ordnung. Beispielsweise hat in einer *UNIX*-Shell der Operator $>$ die Wirkung, die Ausgabe eines Programmes in eine Datei zu lenken. Formal läßt sich dieser Operator durch die im folgenden definierte Funktion modellieren:

$$\begin{aligned} \_ > \_ \ &\in \ \mathit{Program} \to \mathit{Device} \to \mathit{Program} \times \mathit{Device} \\ f > d \ &= \ (f', d') \quad \text{wobei} \quad f'.x = (f.x)[\mathit{stdout} \mapsto \langle\rangle] \\ &\qquad\qquad\qquad\quad d'.x = d.((\mathit{cmd} \mapsto f.x.\mathit{stdout}) \circ x) \end{aligned}$$

Eine wichtige Frage bei der Modellierung mit Funktionen höherer Ordnung ist die der angemessenen Approximationsordnung auf den einzelnen Objekten. Wir haben zum Beispiel auf Strömen schon die punktweise Ordnung $\sqsubseteq$ und die Präfixordnung $\leq$ eingeführt. Für Ströme über den definierten Elementen flacher Bereiche fallen diese Ordnungen zusammen.

Für Ströme über nicht-flachen Bereichen unterscheiden sich dagegen die Ordnungen. Welche Ordnung hier sinnvollerweise verwendet wird, hängt von den zu modellierenden Gegebenheiten ab. Bei informationstheoretischer Betrachtungsweise sollen zwei Ströme dann vergleichbar sein, wenn der eine Strom mehr Informationen als der andere über *denselben* Berechnungsablauf wiedergibt. Entsprechen die Ströme dagegen *verschiedenen* Berechnungen, dann sollen sie nicht vergleichbar sein.

Demzufolge ist die Präfixordnung $\leq$ angemessen, wenn Objekte höherer Ordnung als Modellierung für in der Implementierung endliche Objekte verwendet werden, die nicht schrittweise approximiert werden. Betrachten wir zum Beispiel einen Agenten des oben definierten Typs *Compiler*, der einen Strom von Programmtexten einliest und die entsprechenden Programme generiert. Hier ist für die Eingabeströme die Präfixordnung $\leq$ angebracht, denn natürlich sollen zwei Eingabeströme nur dann in einer Approximationsbeziehung stehen, wenn alle definierten Elemente (Programmtexte) gleich sind. Die bloße Verlängerung eines Programmtextes führt zu einem anderen, unvergleichbaren Programm.

Operationell kann man sich vorstellen, daß bei der Kommunikation von Datenelementen, für die die Präfixordnung $\leq$ angemessen ist, jedes Datenelement vollständig übertragen wird und schon gesendete Datenelemente nicht mehr nachträglich verändert werden können.

Die punktweise Ordnung $\sqsubseteq$ wird dagegen verwendet, wenn Objekte höherer Ordnung solche Daten oder Funktionen repräsentieren, die in der Implementierung nur schrittweise angenähert oder übertragen werden können. Gegeben sei zum Beispiel ein Agent, der einen Strom von Eingabeströmen verarbeitet, von denen jeder ein physisch vorhandenes Eingabegerät modelliert. Wenn die Eingabe x' aus der Eingabe x durch Verlängerung eines solchen Teilstromes hervorgeht, so heißt dies, daß mehr Informa-

tionen über den Berechnungsablauf vorhanden sind, also daß x' durch x approximiert wird. Ähnlich verhält es sich, wenn ein aus Funktionen bestehender Strom als Modellierung dafür benutzt wird, daß in der Implementierung Zeiger auf diese Funktionen übertragen werden und einzelne Funktionsaufrufe nur über einen *remote procedure call* möglich sind.

Daher ist die punktweise Ordnung $\sqsubseteq$ eine angemessene Modellierung für eine Kommunikation, bei der die einzelnen Objekte nicht "in einem Stück" übertragen, sondern nur schrittweise approximiert werden.

Die obigen Überlegungen bezüglich einer geeigneten Approximationsordnung betreffen Fragen der Monotonie- und Stetigkeitsanforderungen für Agenten, die durch stromverarbeitende Funktionen modelliert werden. Die Modellierungstechniken in anderen Formalismen werden davon nicht unbedingt betroffen. So wird zum Beispiel in der Definition 3.5 der Operation *Traces* im Konjunktionsglied $\forall\, s \leq t : OS\circledS s \leq f.(IS\circledS s)$ ausschließlich die Präfixordnung $\leq$ verwendet, um partielle Berechnungsabschnitte zu charakterisieren, die als Teilstück eines vollständigen Berechnungsablaufs auftreten können. Eine erweiterte Definition, etwa der Form $\forall\, s \sqsubseteq t : OS\circledS s \sqsubseteq f.(IS\circledS s)$, ist nicht erforderlich. Ein erster Grund dafür ist, daß in einer durch *Traces* bestimmten Spurmenge sowieso alle möglichen Eingabesequenzen auftreten. Zweitens beschreibt jede Spur nur *einen* Berechnungsablauf, so daß eventuell durch Monotonie und Stetigkeit geforderte Zusammenhänge zwischen verschiedenen Berechnungen keine Rolle spielen.

3.7 Sicherheit und Lebendigkeit

In [Lam77] wurden die Begriffe *Sicherheit* und *Lebendigkeit* mit der informellen Beschreibung geprägt, eine Sicherheitseigenschaft fordere, daß unerwünschtes Verhalten während der Programmausführung nicht auftreten dürfe, und eine Lebendigkeitseigenschaft fordere, daß gewünschtes Verhalten (irgendwann) auftreten müsse. Eine etwas präzisere Charakterisierung ist, daß eine Sicherheitseigenschaft durch endliche Beobachtungen des Systemverhaltens widerlegt werden kann, während Lebendigkeitseigenschaften nur durch unendliche Beobachtungen widerlegbar sind.

Für die Unterteilung von Spezifikationen in Sicherheits- und Lebendigkeitseigenschaften gibt es eine Reihe von Gründen:

- Sicherheits- und Lebendigkeitseigenschaften werden technisch unterschiedlich behandelt, weil entweder unterschiedliche Spezifikationsformalismen oder zumindest speziell für eine Klasse von Eigenschaften angepaßte Notationen und Konventionen eingesetzt werden (zum Beispiel Tabellennotation eines Transitionssystems

für Sicherheitseigenschaften und temporallogische Formeln für Lebendigkeitseigenschaften).

- Die Beweistechniken für Sicherheits- und Lebendigkeitseigenschaften sind unterschiedlich. So können Sicherheitseigenschaften oft durch Induktion über die Länge der Eingaben oder der Aktionsspuren bewiesen werden, während für Lebendigkeitseigenschaften Beweistechniken wie zum Beispiel *proof lattices* [OL92] erforderlich sind.
- Methodisch ist die Unterscheidung von Sicherheit und Lebendigkeit relevant, da während der Programmentwicklung der Sicherheitsanteil der Spezifikation schrittweise erhöht und der Lebendigkeitsanteil entsprechend verringert wird, bis nur noch eine triviale Lebendigkeitseigenschaft übrigbleibt (nämlich die, daß die Berechnung stetig fortschreitet).

Eine formale Definition von Sicherheit und Lebendigkeit bei Zustandsspuren findet sich in [Sch87]. Äquivalente Definitionen, auf Spuren von Aktionen übertragen, sind in [DW89] gegeben. [Kin93] enthält einen Literaturüberblick über weitere Formalisierungen.

Eine oft diskutierte Frage im Zusammenhang mit Sicherheit und Lebendigkeit ist, ob gefordert werden soll, daß sich alle sicheren Teilabläufe zu sicheren *und* lebendigen Abläufen verlängern lassen (*machine closedness*, *feasibility*). Auf diese (auch methodische) Fragestellung wollen wir hier nicht näher eingehen. Es sollen vielmehr die Zusammenhänge zwischen den Begriffen Sicherheit und Lebendigkeit auf der Spurebene und bei funktionalen Spezifikationen untersucht werden.

Als Ausgangspunkt unserer Überlegungen dient die Definition in [DW89, Abschnitt 3.1]. Danach ist ein Spurprädikat $P \subseteq X^\omega$ eine Sicherheitseigenschaft, wenn gilt:

$$\forall\, t \in X^\omega : P.t \Leftrightarrow (\forall\, s \in X^* : s \leq t \Rightarrow P.s)$$

Die beiden Implikationsrichtungen entsprechen den beiden grundlegenden Kennzeichen von Sicherheitseigenschaften: durch die Richtung $P.t \Rightarrow (\forall\, s \in X^* : s \leq t \Rightarrow P.s)$ wird ausgedrückt, daß Sicherheitseigenschaften nach unten (hinsichtlich der Präfixbildung) abgeschlossen sind, und durch die Richtung $P.t \Leftarrow (\forall\, s \in X^* : s \leq t \Rightarrow P.s)$, daß sie hinsichtlich der Grenzwertbildung (nach oben) abgeschlossen sind.

Ein Spurprädikat $P \subseteq X^\omega$ heißt Lebendigkeitseigenschaft, wenn alle endlichen Spuren zu einer lebendigen Spur verlängerbar sind, also wenn gilt:

$$\forall\, s \in X^* : \exists\, t \in X^\omega : P.(s \circ t)$$

In [Sch87] ist gezeigt, daß sich zu jedem Prädikat $P \subseteq X^\omega$ eine Sicherheitseigenschaft S und eine Lebendigkeitseigenschaft L mit $P = S \wedge L$ finden lassen.

Analog zu den obigen Definitionen lassen sich die Begriffe Sicherheit und Lebendigkeit für funktionale Agentenspezifikationen dadurch charakterisieren, daß eine Sicher-

heitseigenschaft durch ihre Abgeschlossenheit nach unten und hinsichtlich der Grenzwertbildung gekennzeichnet ist, und eine Lebendigkeitseigenschaft dadurch, daß alle "endlichen" Agenten zu lebendigen Agenten erweiterbar sind.

Wir geben im folgenden formale Definitionen für Sicherheit und Lebendigkeit bei Agentenspezifikationen, die eine eventuelle Ordnung der auf den Kanälen gesendeten Nachrichten nicht berücksichtigen. Dabei verwenden wir eine auf Agenten erweiterte Präfixordnung $\leq$, die für $f, g \in IS \nrightarrow OS$ definiert ist vermöge:

$$f \leq g \;=\; \forall\, x \in IS : f.x \leq g.x$$

Endliche Agenten sind in diesem Zusammenhang solche Agenten, die für alle Eingaben stets endlich lange Ausgabeströme liefern. Die entsprechende Notation $IS \nrightarrow OS^{fin}$ wurde in Abschnitt 3.2 definiert:

Definition 3.6 (Sicherheit und Lebendigkeit bei Agentenspezifikationen):
$F \subseteq IS \nrightarrow OS$ ist eine *Sicherheitseigenschaft* genau dann, wenn gilt:

$$\forall\, f \in IS \nrightarrow OS : F.f \Leftrightarrow (\forall\, g \in IS \nrightarrow OS^{fin} : g \leq f \Rightarrow F.g)$$

$F \subseteq IS \nrightarrow OS$ ist eine *Lebendigkeitseigenschaft* genau dann, wenn gilt:

$$\forall\, f \in IS \nrightarrow OS^{fin} : \exists\, g \in IS \nrightarrow OS : f \leq g \wedge F.g$$ □

Auch hier können beliebige Agentenspezifikationen in Sicherheits- und Lebendigkeitseigenschaften nach der obigen Definition aufgespalten werden:

Satz 3.2: Zu jeder Agentenspezifikation $F \subseteq IS \nrightarrow OS$ existieren eine Sicherheitseigenschaft $S \subseteq IS \nrightarrow OS$ und eine Lebendigkeitseigenschaft $L \subseteq IS \nrightarrow OS$, so daß $F = S \wedge L$ gilt.

Beweis: Nach Anmerkung 3.1 in Verbindung mit der Beobachtung, daß Strombereiche über flachen Nachrichtenbereichen immer algebraisch sind, ist Satz 3.2 ein Spezialfall von Satz 3.5 (hierbei wird $|\bullet IS| < \infty$ angenommen). □

Es stellt sich die Frage, inwieweit die gerade definierten Sicherheits- und Lebendigkeitsbegriffe der Klassifizierung entsprechen, die man auf der Spurebene durch Anwendung des *Traces*-Operators auf Agentenspezifikationen erhält. Zu dieser Untersuchung definieren wir für alle $s \in (Act.IS \cup Act.OS)^{\omega}$ die Funktion $g_s \in IS \nrightarrow OS$, die jeweils die Ausgabenachrichten des maximalen Präfixes von s ausgibt, das bezüglich seiner Eingabenachrichten nicht den Eingaben im Argument von g_s widerspricht:

$$\begin{aligned} g_s.x \;=\; OS\circledS s' \quad & \text{für dasjenige } s' \in (Act.IS \cup Act.OS)^{\omega} \text{ mit} \\ & s' \leq s \;\wedge\; IS\circledS s' \leq x \;\wedge \\ & \neg\exists a \in Act.IS \cup Act.OS : s' \circ \bar{a} \leq s \wedge IS\circledS(s' \circ \bar{a}) \leq x \end{aligned}$$

Offensichtlich haben die Funktionen g_s folgende Eigenschaften:

$$g_s \in IS \twoheadrightarrow OS^{fin} \text{ für } \#s < \infty \quad (1)$$
$$Traces.g_s.s \quad (2)$$
$$\forall f \in IS \twoheadrightarrow OS,\ t \in (Act.IS \cup Act.OS)^{\omega} : (s \leq t \wedge Traces.f.t) \Rightarrow g_s \leq f \quad (3)$$

Satz 3.3: Falls $F \subseteq IS \twoheadrightarrow OS$ eine Sicherheitseigenschaft ist, dann ist auch $Traces.F$ eine Sicherheitseigenschaft.

Beweis: Gegeben sei ein $F \subseteq IS \twoheadrightarrow OS$ mit $\forall f \in IS \twoheadrightarrow OS : F.f \Leftrightarrow (\forall g \in IS \twoheadrightarrow OS^{fin} : g \leq f \Rightarrow F.g)$ $(*)$. Wir untersuchen die beiden Implikationsrichtungen der zu zeigenden Behauptung $\forall t \in (Act.IS \cup Act.OS)^{\omega} : Traces.F.t \Leftrightarrow (\forall s \in (Act.IS \cup Act.OS)^{*} : s \leq t \Rightarrow Traces.F.s)$ getrennt.

"$\Rightarrow$" (Abgeschlossenheit nach unten): Es gelte $Traces.F.t$, also existiert ein $f \in F$ mit $Traces.f.t$. Sei $s \in (Act.IS \cup Act.OS)^{*}$ mit $s \leq t$ beliebig aber fest. Wegen der Eigenschaften (1), (2) und (3) von g_s erhalten wir nach Richtung "$\Rightarrow$" von $(*)$ $F.g_s$ und somit $Traces.F.s$.

"$\Leftarrow$" (Abgeschlossenheit hinsichtlich der kleinsten oberen Schranken): Gegeben sei ein $t \in (Act.IS \cup Act.OS)^{\omega}$ mit $\forall s \in (Act.IS \cup Act.OS)^{*} : s \leq t \Rightarrow Traces.F.s$ $(**)$. Es sei s_i für $i \in \mathbb{N}$ jeweils das Präfix von t mit $\#s_i = i$. Nach $(**)$ gibt es zu jedem s_i ein $\hat{g}_{s_i} \in F$ mit $Traces.\hat{g}_{s_i}.s_i$. Nach Eigenschaft (3) von g_{s_i} gilt dann $g_{s_i} \leq \hat{g}_{s_i}$ und nach Richtung "$\Rightarrow$" von $(*)$ $F.g_{s_i}$. Da für alle $h \in IS \twoheadrightarrow OS^{fin}$ mit $h \leq g_t$ auch ein g_{s_i} mit $h \leq g_{s_i}$ existiert, gilt $F.h$ und nach Richtung "$\Leftarrow$" von $(*)$ auch $F.g_t$. Da nach Eigenschaft (2) von g_t auch $Traces.g_t.t$ gilt, erhält man insgesamt $Traces.F.t$. □

Satz 3.4: Falls $F \subseteq IS \twoheadrightarrow OS$ eine Lebendigkeitseigenschaft ist, dann ist auch $Traces.F$ eine Lebendigkeitseigenschaft.

Beweis: Gegeben sei ein $f \subseteq IS \twoheadrightarrow OS$ mit $\forall f \in IS \twoheadrightarrow OS^{fin} : \exists g \in IS \twoheadrightarrow OS : f \leq g \wedge F.g$ $(*)$. Weiter sei ein $s \in (Act.IS \cup Act.OS)^{*}$ beliebig aber fest gegeben. Für die oben definierte Funktion $g_s \in IS \twoheadrightarrow OS^{fin}$ existiert nach $(*)$ ein $g'_s \in IS \twoheadrightarrow OS$ mit $g_s \leq g'_s$ und $F.g'_s$. Dieses g'_s erzeugt mit der Eingabe ISⓢs eine Spur, die eine Verlängerung von s ist, so daß $\exists t \in (Act.IS \cup Act.OS)^{\omega} : Traces.g'_s.(s \circ t)$ und damit auch $Traces.F.(s \circ t)$ gilt. □

Die Umkehrungen von Satz 3.3 und Satz 3.4 gelten allerdings nicht. Es folgt also für $F \subseteq IS \twoheadrightarrow OS$, wobei $Traces.F$ eine Sicherheitseigenschaft beziehungsweise eine Lebendigkeitseigenschaft ist, nicht notwendigerweise, daß auch F eine Sicherheitseigenschaft beziehungsweise eine Lebendigkeitseigenschaft ist. Dies liegt daran, daß beim Bilden der Spurmenge einer Spezifikation F Verwischungseffekte auftreten können, die dadurch entstehen, daß die Spurmengen der einzelnen Funktionen $f \in F$ vereinigt werden. Die Zuordnung einer Spur zu der sie erzeugenden Funktion ist dann nicht mehr

erkennbar. Diesen Effekt zeigen die Gegenbeispiele 3.1 für Sicherheitseigenschaften und 3.2 für Lebendigkeitseigenschaften:

Gegenbeispiel 3.1: Es sei $F = \{f \in IS \nrightarrow OS : \exists\, x \in IS : \#f.x < \#x\}$. Offensichtlich gilt $Traces.F = (Act.IS \cup Act.OS)^\omega$, so daß $Traces.F$ eine Sicherheitseigenschaft ist. F ist jedoch keine Sicherheitseigenschaft, denn für jedes $f' \in IS \nrightarrow OS$ mit $\forall\, x \in IS : \#f'.x = \#x$ liegen alle $g \in IS \nrightarrow OS^{fin}$ mit $g \leq f'$ in F (wegen der Endlichkeit von g), aber $F.f'$ gilt nicht. □

Gegenbeispiel 3.2: Es sei F die Menge der Funktionen, die jeweils ein konstantes Ergebnis approximieren, also $F = \{f \in IS \nrightarrow OS : \exists\, y \in OS : \forall\, x \in IS : f.x \leq y\}$. Offensichtlich gilt $Traces.F = (Act.IS \cup Act.OS)^\omega$, so daß $Traces.F$ eine Lebendigkeitseigenschaft ist. F ist jedoch beispielsweise für $IS = in : \mathbb{N}^\omega$ und $OS = out : \mathbb{N}^\omega$ keine Lebendigkeitseigenschaft, denn für $f' \in IS \nrightarrow OS^{fin}$ mit $f'.(in \mapsto c).out = \langle ft.c \rangle$ existiert kein $g \in IS \nrightarrow OS$ mit $f' \leq g$ und $F.g$. □

3.8 Sicherheit und Lebendigkeit bei Agenten höherer Ordnung

In diesem Abschnitt soll untersucht werden, unter welchen Bedingungen die bisher beschriebene Definition von Sicherheit und Lebendigkeit auch bei den in unserem Formalismus verwendeten Agenten höherer Ordnung adäquat ist.

Wie schon in Abschnitt 3.6 kommt es auch hier darauf an, wie wir ein Datenelement höherer Ordnung interpretieren. Wenn wir annehmen, daß alle Stromelemente höherer Ordnung (zum Beispiel unendliche Ströme oder Funktionen) nur auf der Spezifikationsebene verwendet werden, um ein in der Implementierung endliches Objekt (zum Beispiel eine zyklische Zeigerstruktur oder eine Berechnungsregel) zu repräsentieren, dann ist die Definition 3.6 angemessen.

Wenn man dagegen annimmt, daß Stromelemente höherer Ordnung nur schrittweise approximiert werden und nicht in "kompakter" Form dargestellt und übertragen werden können oder sollen, dann entspricht Definition 3.6 unserer am Anfang von Abschnitt 3.7 beschriebenen Intuition nicht.

Betrachten wir zum Beispiel die folgende Agentenspezifikation F für ein beliebiges IS:

$$\begin{aligned} F &\subseteq IS \nrightarrow \otimes\, out : (\mathbb{N}^\omega)^\omega \\ F.f &= \forall\, x \in IS, n \in \mathbb{N} : n < \#f.x \Rightarrow \#f.x.out.n = \infty \end{aligned}$$

Das Prädikat F ist nach Definition 3.6 eine Sicherheitseigenschaft, obwohl es bezüglich der "inneren" Stromelemente der Ausgabe eine Lebendigkeitseigenschaft ausdrückt,

wenn sich die einzelnen $f.x.out.n$ nur schrittweise berechnen lassen. Daher modifizieren wir die Definition von Sicherheit und Lebendigkeit wie folgt:

Definition 3.7 (Sicherheit und Lebendigkeit bei Agenten höherer Ordnung):
$F \subseteq IS \twoheadrightarrow OS$ ist eine *HO-Sicherheitseigenschaft* genau dann, wenn gilt:

$$\forall\, f \in IS \twoheadrightarrow OS : F.f \Leftrightarrow (\forall\, g \in \mathbb{E}.(IS \twoheadrightarrow OS) : g \sqsubseteq f \Rightarrow F.g)$$

$F \subseteq IS \twoheadrightarrow OS$ ist eine *HO-Lebendigkeitseigenschaft* genau dann, wenn gilt:

$$\forall\, f \in \mathbb{E}.(IS \twoheadrightarrow OS) : \exists\, g \in IS \twoheadrightarrow OS : f \sqsubseteq g \wedge F.g$$ □

Anmerkung 3.1: Falls IS und OS Kanaltypen mit Nachrichten aus flachen Bereichen und endlich vielen Kanalnamen sind, fallen die Begriffe Sicherheit und HO-Sicherheit beziehungsweise Lebendigkeit und HO-Lebendigkeit zusammen, da die Ordnungen $\sqsubseteq$ und $\leq$ zusammenfallen und $\mathbb{E}.(IS \twoheadrightarrow OS) = IS \twoheadrightarrow OS^{fin}$ gilt. □

Die Aufspaltbarkeit beliebiger Agentenspezifikationen in eine HO-Sicherheits- und eine HO-Lebendigkeitseigenschaft gilt nur für algebraische Bereiche, wie das folgende Beispiel zeigt:

Gegenbeispiel 3.3: Es sei X der in Beispiel 2.1 gegebene nicht-algebraische Bereich, IS beliebig und $OS = \otimes\, out : X$. Dann kann die Agentenspezifikation $F \subseteq IS \twoheadrightarrow OS$ mit $F.f = \forall\, x \in IS : f.x.out = \langle ? \rangle$ nicht in eine HO-Sicherheitseigenschaft S und eine HO-Lebendigkeitseigenschaft L mit $F = S \wedge L$ aufgespalten werden. □

Für algebraische Bereiche sind dagegen beliebige Spezifikationen aufspaltbar:

Satz 3.5: Zu jeder Agentenspezifikation $F \subseteq IS \twoheadrightarrow OS$ existieren eine HO-Sicherheitseigenschaft $S \subseteq IS \twoheadrightarrow OS$ und eine HO-Lebendigkeitseigenschaft $L \subseteq IS \twoheadrightarrow OS$, so daß $F = S \wedge L$ gilt.

Beweis: Die Beweisidee stammt aus [Sch87]. Gegeben sei eine Agentenspezifikation $F \subseteq IS \twoheadrightarrow OS$. Wir definieren:

$$\begin{aligned} M.f &= \forall\, g \in \mathbb{E}.(IS \twoheadrightarrow OS) : g \sqsubseteq f \Rightarrow (\exists\, h \in IS \twoheadrightarrow OS : g \sqsubseteq h \wedge F.h) \\ S.f &= F.f \vee M.f \\ L.f &= F.f \vee \neg M.f \end{aligned}$$

Offensichtlich gilt $F = S \wedge L$. Zu zeigen ist noch, daß S eine HO-Sicherheitseigenschaft ist und L eine HO-Lebendigkeitseigenschaft.

a) S ist HO-Sicherheitseigenschaft:
Beweis analog zu [DW89, Seite 10]. Da wir vorausgesetzt haben, daß Spezifikationen nur algebraische Bereiche verwenden, ist auch in dem hier gegebenen Rahmen jedes Bereichselement die kleinste obere Schranke seiner endlichen Approximationen.

b) L ist HO-Lebendigkeitseigenschaft:
Es gilt $L.f = F.f \vee (\exists\, g \in \mathbb{E}.(IS \twoheadrightarrow OS) : g \sqsubseteq f \wedge \neg(\exists\, h \in IS \twoheadrightarrow OS : g \sqsubseteq h \wedge F.h))$. Zu zeigen ist, daß für jedes $f' \in \mathbb{E}.(IS \twoheadrightarrow OS)$ ein $f \in IS \twoheadrightarrow OS$ mit $f' \sqsubseteq f$ existiert, für das $L.f$ gilt. Wählen wir ein beliebiges endliches $f' \in \mathbb{E}.(IS \twoheadrightarrow OS)$. Wenn dazu ein $f \in IS \twoheadrightarrow OS$ mit $f' \sqsubseteq f$ existiert, für das $F.f$ gilt, ist das erste Disjunktionsglied von L erfüllt. Wenn kein solches f existiert, für das $F.f$ gilt, ist das zweite Disjunktionsglied von L mit $g = f'$ erfüllt. □

Als Ergebnis der Abschnitte 3.7 und 3.8 halten wir fest, daß sich die ursprünglich auf Spuren definierten Begriffe von Sicherheit und Lebendigkeit auch auf Agenten und Agenten höherer Ordnung ausdehnen lassen. Die intuitive Bedeutung und die wesentlichen Eigenschaften dieser Begriffe ändern sich dadurch nicht.

Kapitel 4

Methodik des Systementwurfs

Der im vorhergehenden Kapitel 3 beschriebene "Werkzeugsatz" von grundlegenden Begriffen und Operatoren muß geeignet eingesetzt werden. Dazu wird hier als formaler Rahmen eine Entwurfsmethodik vorgeschlagen, die sich an der am Lehrstuhl Broy der Technischen Universität München entwickelten Methodik FOCUS [BDD+92a] orientiert. Es wird eine formale Definition des Entwurfsprozesses und der verschiedenen Spezifikationsarten (globale Spezifikation, Komponentenspezifikation) gegeben, wie sie für FOCUS bisher nicht existiert. Verfeinerungsbegriffe sowohl innerhalb der Spezifikationsebenen als auch für den Ebenenübergang werden angegeben. Unterschiede zu FOCUS ergeben sich insbesondere dadurch, daß die hier dargestellte Methodik konsequent Agenten mit benannten Kanälen und Agentenverbindungen über Kanalnamen verwendet. Weiterhin wird durch die Verwendung sogenannter Systemstrukturen zur Modellierung der Agenten- und Verbindungsstruktur eines Systems eine klare Trennung der Spezifikation des Aufbaus eines Systems von der Spezifikation der einzelnen Agenten geschaffen, während in FOCUS keine klaren Regeln dafür bestehen, welche Verbindungsstruktur eines Systems einer gegebenen Spezifikation entspricht.

Der im folgenden beschriebene Entwurfsprozeß ist linear strukturiert, das heißt, er spiegelt nur die "geradlinige" Systementwicklung ohne Irrwege oder Sackgassen wider. Das ist insofern sinnvoll, als eine formale Definition des Entwurfsprozesses nur den Sinn hat, die Korrektheit des fertigen Programms (bezüglich der Anforderungsspezifikation) sicherzustellen. Dazu genügt es, wenn der schließlich "erfolgreiche" Entwicklungsweg aus einer Folge von korrekten Entwicklungsschritten besteht. Für den praktischen Einsatz benötigt man natürlich ein Werkzeug, das unterschiedliche Versionen der Spezifikation und das Wiederaufsetzen an einem früheren Punkt der Systementwicklung unterstützt, wenn sich eine Entwurfsentscheidung als nicht angemessen erweist. Diese pragmatisch überaus wichtigen Funktionen müssen jedoch in der hier gegebenen formalen Definition des Entwurfsprozesses nicht behandelt werden, da sie keinen Einfluß auf das endgültige Programm haben.

1. globale Spezifikation

   Lokalisierung der Anforderungen;
   Aufspaltung in Komponenten, die
   (zum Beispiel räumlich) getrennt
   implementiert werden sollen.

2. Komponentenspezifikation
   (Spurformalismus)

   Transformation in andere
   Entwurfsformalismen

3. Komponentenspezifikation
   (andere Formalismen)

   Verfeinerungsschritte hin
   zu implementierungs-
   nahen Spezifikationen

4. Implementierung

   weitere Verfeinerungs-
   schritte, zum Beispiel von
   deklarativen zu imperativen
   Programmiersprachen

Bild 4.1: Überblick über die Entwurfsmethodik

Bild 4.1 gibt einen Überblick über die gesamte Methode. Wir unterteilen den Entwurfsprozeß in drei Ebenen:

- Entwicklung auf der Systemebene
- Entwicklung auf der Komponentenebene
- Effizienzsteigerung auf der Implementierungsebene

Zunächst wird eine globale Spezifikation erstellt. Dazu ist der Formalismus der Spurspezifikation angemessen, weil Spuren ein intuitiv besonders eingängiges Beschreibungsmittel für globale Spezifikationen sind. Dies gilt insbesondere für die frühen Entwicklungsstufen neuer Systeme, bei denen die Anforderungen an das Gesamtsystem und die Aufteilung des Systems in Komponenten erst festgelegt und formalisiert werden müssen. Handelt es sich dagegen um eine Entwicklung, bei der die geforderte Struktur des Systems und die Eigenschaften der Komponenten schon von Anfang an feststehen, dann kann es sinnvoll sein, die Stufe der globalen Systementwicklung im Spurformalismus zu überspringen und gleich mit einer komponentenorientierten Spezifikation, beispielsweise im funktionalen Formalismus, zu beginnen.

Wenn die globale Spezifikation vorliegt, werden die globalen Anforderungen durch schrittweise Verfeinerung lokalisiert. Das heißt, für jede einzelne Systemkomponente werden Spezifikationen angegeben, so daß das Gesamtsystem die globalen Anforderungen erfüllt. In dieser Entwicklungsphase können Komponenten aufgespalten werden, um eine weitere Verteilung des Gesamtsystems festzulegen. Ebenso kann man die Spezifikation aufspalten, um die weitere Entwicklung für die neu gebildeten Teilspezifikationen parallel vorzunehmen. Bis zu diesem Punkt findet der Entwicklungsprozeß auf der *Systemebene* statt.

Sind lokale Anforderungen für jede Systemkomponente gefunden, dann beginnt der Entwicklungsprozeß auf der *Komponentenebene.* Hier geht es darum, aus der anfänglich noch sehr "impliziten" lokalen Spezifikation schrittweise eine "explizite", das heißt implementierungsnahe Fassung zu entwickeln. Auch dieser Entwicklungsvorgang kann prinzipiell in verschiedenen Formalismen ablaufen, wobei aber hier nur auf den Formalismus der *funktionalen Spezifikation* näher eingegangen wird. Hat die Entwicklung auf der Systemebene im Spurformalismus stattgefunden, dann muß als erster Schritt der Übergang von der lokalen Spurspezifikation in eine entsprechende funktionale Spezifikation bewerkstelligt werden. Die Entwicklung auf der funktionalen Ebene führt dann zum Beispiel auf Spezifikationen, die Programmen in einer funktionalen Programmiersprache ähneln. Eine besondere Ausprägung der Spezifikation auf der funktionalen Ebene ist die *zustandsorientierte* funktionale Spezifikation, die in Kapitel 5 ausführlich beschrieben wird.

Die Umwandlung einer funktionalen Spezifikation in eine geeignete Programmiersprache markiert den Übergang zur *Implementierungsebene.* Auch hier kann man verschiedene Sprachen in den hier beschriebenen Entwurfsprozeß einfügen. Wurde für die Kom-

ponentenspezifikation ein funktionaler Formalismus gewählt, dann bietet sich als Implementierungssprache natürlich eine funktionale Sprache wie PL, ML oder HASKELL an. Wurden zustandsorientierte Techniken verwendet, dann ist auch die Umsetzung in parallele imperative Sprachen relativ leicht möglich. Innerhalb der Implementierungsebene können weitere Verfeinerungsschritte durchgeführt werden.

4.1 Systemstrukturen

Eine *Systemstruktur* repräsentiert die äußere Form eines verteilten Systems. Da Systemstrukturen in allen Entwicklungsebenen gleichermaßen vorkommen, werden sie hier zunächst, zusammen mit einigen Operationen darauf, eingeführt.

Die intuitive Vorstellung ist, daß eine Systemstruktur einen gewissen Entwicklungsstand eines aus mehreren Komponenten und unidirektionalen Verbindungskanälen zusammengesetzten verteilten Systems darstellt. Eine Systemstruktur kann man sich also als einen gerichteten Graph vorstellen, dessen Knoten die Systemkomponenten und dessen Kanten die Kanäle repräsentieren. Ein Kanal verbindet einen Ausgang mit einem oder mehreren Eingängen. Außerdem gibt es Verbindungen zur Umgebung, also Kanäle, die nur an Eingänge oder nur an einen Ausgang angeschlossen sind.

Wir modellieren eine Systemstruktur durch drei Funktionen In, Out und Cha. Dabei gibt In zu jeder Komponente die Menge der Eingänge an und Out die Menge der Ausgänge. Ein- und Ausgänge mit gleichem Namen betrachten wir als miteinander verbunden. Die Funktion Cha gibt zu jedem Namen eines Ein- oder Ausgangs die Menge der Nachrichten an, die auf dem entsprechenden Kanal ausgetauscht werden können.

Definition 4.1 (Systemstruktur): Eine *Systemstruktur* ist ein Tripel $(In,\ Out,\ Cha)$ mit folgenden Eigenschaften:

- In und Out sind Funktionen mit $\bullet In = \bullet Out$.
- Für alle $z \in \bullet In$ ist $In.z$ eine endliche Menge.
- Für alle $z \in \bullet Out$ ist $Out.z$ eine endliche Menge, und alle Mengen $Out.z$ sind paarweise disjunkt.
- Cha ist eine Funktion mit $\bullet Cha = (\bigcup_{z \in \bullet In}\ In.z) \cup (\bigcup_{z \in \bullet Out}\ Out.z)$, und $Cha.c$ ist für jedes $c \in \bullet Cha$ eine Menge (von Nachrichten). □

Gegeben sei eine Systemstruktur $S = (In,\ Out,\ Cha)$. Für die Menge der Komponenten schreiben wir auch $Comp.S$ (also $Comp.S = \bullet In = \bullet Out$). Es sei $I = (\bigcup_{z \in \bullet In}\ In.z)$ und $O = (\bigcup_{z \in \bullet Out}\ Out.z)$. Die Menge der Namen der Eingabekanäle von der Umgebung zum System bezeichnen wir mit $Inputs.S = I \setminus O$, die der Ausgabekanäle an die Umgebung mit $Outputs.S = O \setminus I$ und die der internen Kanäle mit $Internals.S = I \cap O$.

Das heißt, daß ein Kanal c genau dann ein interner Kanal ist, wenn in der Systemstruktur (mindestens ein) Eingang und ein Ausgang mit Namen c existieren. Ein interner Kanal ist von außen nicht verfügbar, insbesondere kann er auch kein Ausgabekanal der Systemstruktur sein.

Zur späteren Schreiberleichterung führen wir für Systemstrukturen $S = (In, Out, Cha)$ und Komponentennamen $z \in Comp.S$ die folgenden Notationen ein:

- $CompStruct_S.z$ ist die der Komponente z entsprechende Systemstruktur:
  $CompStruct_S.z \;=\; (In|_{\{z\}},\; Out|_{\{z\}},\; Cha|_{In.z \cup Out.z})$
- $InType_S.z$ ist der Kanaltyp der Eingabekanäle der Komponente z:
  $InType_S.z \;=\; \otimes_{c \in In.z}\, c : (Cha.c)^{\omega}$
- $OutType_S.z$ ist der Kanaltyp der Ausgabekanäle der Komponente z:
  $OutType_S.z \;=\; \otimes_{c \in Out.z}\, c : (Cha.c)^{\omega}$
- $InAct_S.z$ ist die Menge der bei der Komponente z möglichen Eingabeaktion:
  $InAct_S.z \;=\; Act.(InType_S.z)$
- $OutAct_S.z$ ist die Menge der bei der Komponente z möglichen Ausgabeaktion:
  $OutAct_S.z \;=\; Act.(OutType_S.z)$
- $Act_S.z$ ist die Menge der bei z möglichen Ein- und Ausgabeaktion:
  $Act_S.z \;=\; InAct_S.z \cup OutAct_S.z$
- Act_S ist die Menge der in S möglichen Ein- und Ausgabeaktion:
  $Act_S \;=\; \bigcup_{z \in Comp.S} Act_S.z$

Beispiel 4.1: Wir betrachten eine Prozessoreinheit *PSys* mit zwei Rechenkernen *Proc1* und *Proc2* sowie einem Verwaltungsagenten *Sched*. Der Verwaltungsagent erhält von der Umgebung Aufträge über den Kanal *SCmd* und liefert Rückmeldungen über *SRply*. Die Rechenkerne erhalten Aufträge über den gemeinsamen Kanal *PCmd* und liefern Rückmeldungen über die Kanäle *P1Rply* und *P2Rply*. Die zweifache Ausführung der Rechenkerne dient hier also nicht einer Steigerung der Rechenleistung, sondern der Ausfallsicherheit. Bild 4.2 zeigt diese Prozessoreinheit:

Bild 4.2: Eine einfache Systemstruktur

Um die Systemstruktur $PSysStruct = (PSysIn, PSysOut, PSysCha)$ der Prozessoreinheit $PSys$ formal zu modellieren, definieren wir die Abbildungen $PSysIn$ und $PSysOut$ in Tabellenschreibweise:

$PSysIn$	
$Sched$	$\{SCmd,\ P1Rply,\ P2Rply\}$
$Proc1$	$\{PCmd\}$
$Proc2$	$\{PCmd\}$

$PSysOut$	
$Sched$	$\{SRply,\ PCmd\}$
$Proc1$	$\{P1Rply\}$
$Proc2$	$\{P2Rply\}$

Zur Definition der Abbildung $PSysCha$ legen wir zunächst die Mengen Cmd und $Rply$ der auf den Kanälen fließenden Nachrichten fest. Dabei seien die Menge Id der Identifikatoren und der Bereich $Data$ der Daten gegeben. Der Einfachheit halber wählen wir den Funktionenbereich $Prog = Data \twoheadrightarrow Data$ als Bereich der zulässigen Programme.

Die Befehls- und Antwortmengen sind definiert als:

$$\begin{aligned} Cmd_i &= \{Task(i,p,d) : p \in Prog,\ d \in Data\} \\ Rply_i &= \{Error(i)\} \cup \{Result(i,d) : d \in Data\} \\ Cmd &= \bigcup_{i \in Id} Cmd_i \\ Rply &= \bigcup_{i \in Id} Rply_i \end{aligned}$$

Damit kann nun die Abbildung $PSysCha$ in Tabellenschreibweise angegeben werden:

$PSysCha$	
$SCmd,\ PCmd$	Cmd
$SRply,\ P1Rply,\ P2Rply$	$Rply$

Mit den obigen Definitionen erhalten wir:

$$\begin{aligned} Inputs.PSysStruct &= \{SCmd\} \\ Outputs.PSysStruct &= \{SRply\} \\ Internals.PSysStruct &= \{PCmd, P1Rply, P2Rply\} \end{aligned}$$ □

Als Operation auf Systemstrukturen betrachten wir zunächst die *Kombination.* Die Kombination verbindet mehrere Systemstrukturen, falls diese bestimmte Eigenschaften aufweisen:

Definition 4.2 (Kombinierbarkeit und Kombination von Systemstrukturen):
Gegeben sei ein endliche, nicht-leere Namensmenge N und eine Familie von Systemstrukturen $S_n = (In_n,\ Out_n,\ Cha_n)$ für $n \in N$. Diese Systemstrukturen heißen *kombinierbar,* wenn für alle $n, m \in N$ mit $n \neq m$ gilt:

- disjunkte Komponenten: $Comp.S_n \cap Comp.S_m = \varnothing$
- disjunkte Ausgabekanäle: $Outputs.S_n \cap Outputs.S_m = \varnothing$
- eindeutige interne Kanäle: $Internals.S_n \cap \bullet Cha_m = \varnothing$
- kompatible Kanäle: $Cha_n|_{\bullet Cha_n \cap \bullet Cha_m} = Cha_m|_{\bullet Cha_n \cap \bullet Cha_m}$

Wir bezeichnen die *Kombination* einer Familie kombinierbarer Systemstrukturen $S_n = (In_n,\ Out_n,\ Cha_n)$ mit $\|_{n\in N}\, S_n$ und definieren:

$$\|_{n\in N}\, S_n = (\bigcup_{n\in N} In_n, \bigcup_{n\in N} Out_n, \bigcup_{n\in N} Cha_n) \qquad \square$$

Statt $\|_{n\in N}\, S_n$ schreiben wir auch $\|\Sigma$ mit $\Sigma = \{S_n : n \in N\}$. Offensichtlich dient der Kombinationsoperator insofern der Modularisierung, als die in der sich ergebenden Systemstruktur internen Kanäle von außen nicht mehr zugänglich sind. Daher ist es auch nicht ausreichend, den Operator $\|$ nur binär zu verwenden, weil man sonst keine Verbindung eines Ausgangs mit den Eingängen mehrerer Systemstrukturen herstellen könnte.

Die Bedingungen für die Kombinierbarkeit von Systemstrukturen lassen sich für beliebige Mengen von Systemstrukturen durch geeignete Umbenennung erfüllen. In der Praxis spielen Namenskonflikte interner Kanäle keine Rolle, auch wenn sie formell ausgeschlossen werden müssen, weil die Entwicklung immer nur in Richtung einer zunehmenden Aufspaltung und nicht in Richtung einer Zusammenlegung erfolgt. Daher ist es beispielsweise mit einem geeigneten Entwicklungswerkzeug problemlos möglich, internen Kanälen einer Systemstruktur eine (für den Benutzer verborgene) eindeutige Kennzeichnung hinzuzufügen. Wenn wir im folgenden von der Kombination von Systemstrukturen sprechen, so setzen wir deren Kombinierbarkeit stillschweigend voraus.

Anmerkung 4.1: Für alle (kombinierbaren) Systemstrukturen S gilt $\|\{S\} = S$. $\square$

Im Hinblick auf die hier zu beschreibende Entwurfsmethodik legt eine Systemstruktur natürlich nicht das endgültige Aussehen des zu erstellenden Systems fest. Vielmehr werden nur gewisse Mindestanforderungen beschrieben, die während des ganzen Entwicklungsprozesses erhalten bleiben sollen. Insbesondere sind dies die *geforderte Mindestverteilung* und die *erlaubte maximale Kommunikationsstruktur.*

In Beispiel 4.1 wird etwa gefordert, daß das zu entwerfende System mindestens zwei getrennte Rechenkerne aufweisen soll, die parallel die gleichen Aufträge verarbeiten. Die geforderte Trennung dient hier der Ausfallsicherheit. Natürlich darf diese Trennung nicht durch eine direkte Kommunikationsverbindung zwischen den Rechenkernen aufgehoben werden, denn sonst könnte man einen der Prozessoren einfach durch einen Agenten implementieren, der die Ergebnisse des anderen Rechenkerns dupliziert.

Neben Sicherheitsüberlegungen kann eine Trennung von Agenten auch zur Leistungssteigerung durch parallele Verarbeitung oder zur besseren konzeptuellen Gliederung eines Systems dienen. In Telekommunikationssystemen ist die physische Trennung von Sender und Empfänger gerade *die* grundlegende Anforderung. Auch hier ist es offensichtlich, daß die in der Anforderungsspezifikation festgelegte Verbindungsstruktur (die zum Beispiel über unzuverlässige Kanalagenten verläuft) nicht durch eine später eingefügte direkte Verbindung "ergänzt" werden darf.

Es soll nun ein *Verfeinerungsbegriff* auf Systemstrukturen definiert werden, der den genannten Anforderungen entspricht. Das heißt, daß durch einen Verfeinerungsschritt keine ursprünglich getrennten Komponenten zusammengelegt und keine neuen Kanäle zwischen bestehenden Komponenten eingefügt werden dürfen. Formal definieren wir einen Verfeinerungsschritt auf Systemstrukturen als das Ersetzen einer Komponente z einer ursprünglichen Systemstruktur S durch ein ganzes Subsystem S'. Die Ein- und Ausgabekanäle der ersetzten Komponente entsprechen den Ein- und Ausgabekanälen des neuen Subsystems.

Definition 4.3 (Verfeinerung von Systemstrukturen): Es seien zwei Systemstrukturen $S = (In, Out, Cha)$ und $S' = (In', Out', Cha')$ und ein $z \in Comp.S$ gegeben. Die Menge Σ von Systemstrukturen sei definiert vermöge:

$$\Sigma = \{CompStruct_S.z' : z' \in Comp.S \backslash \{z\}\} \cup \{S'\}$$

Falls Σ kombinierbar ist und außerdem $In.z = Inputs.S'$ und $Out.z = Outputs.S'$ gilt, sagen wir, daß die Systemstruktur $\|\Sigma$ durch einen *Verfeinerungsschritt* aus S hervorgeht.

Die transitive Hülle der die einzelnen Verfeinerungsschritte enthaltenden Relation bezeichnen wir als *Verfeinerungsrelation* $\rightsquigarrow$. □

Die Verfeinerungsrelation $\rightsquigarrow$ auf Systemstrukturen mit mindestens einer Komponente ist offensichtlich reflexiv. Eine Folge einzelner Verfeinerungsschritte, die in dem Sinne aneinander anschließen, daß eine im jeweils vorhergehenden Schritt neu hinzugefügte Komponente weiter verfeinert wird, läßt sich zu einem einzigen Verfeinerungsschritt zusammenfassen. Die Hüllenbildung in der obigen Definition 4.3 ist aber trotzdem erforderlich, weil sich Verfeinerungsschritte, die von zwei getrennten Komponenten ausgehen, nicht so zusammenfassen lassen.

Beispiel 4.2: In der Systemstruktur *PSysStruct* soll der Agent *Sched* durch das in Bild 4.3 gezeigte Netz verfeinert werden:

Bild 4.3: Verfeinerung des Agenten *Sched*

Es sei $SchedSys = (SchedIn, SchedOut, SchedCha)$ die Struktur mit den in Bild 4.3 veranschaulichten Zuordnungsfunktionen *SchedIn* und *SchedOut* sowie der Abbildung

SchedCha, die definiert ist durch:

$$SchedCha = PSysCha \cup (MtoF \mapsto Cmd) \cup (FtoT \mapsto Cmd) \cup (TtoM \mapsto \{Error(i) : i \in Id\})$$

Mit *PSysStruct'* bezeichnen wir die Struktur, die sich ergibt, wenn *Sched* in *PSysStruct* durch *SchedSys* ersetzt wird, also:

$$PSysStruct' = \| \{ SchedSys,\ CompStruct_{PSysStruct}.Proc1,\ CompStruct_{PSysStruct}.Proc2 \}$$

Gemäß Definition 4.3 gilt $PSysStruct \rightsquigarrow PSysStruct'$, weil *PSysStruct'* durch einen Verfeinerungsschritt aus *PSysStruct* hervorgeht. □

Der in Definition 4.3 gegebene Verfeinerungsbegriff für Systemstrukturen läßt sich als horizontale Verfeinerung oder, in der Terminologie von [Bro94b], als Verfeinerung der Verteilung/Architektur eines Systems beschreiben. Dies ist eine Verfeinerung des internen Aufbaus des Systems, bei der die Ein- und Ausgabekanäle unverändert bleiben (*glass box refinement*).

Dieses Verfeinerungskonzept erlaubt es auch, eine neue Komponente in einen internen oder einen Ausgabekanal einzufügen, indem man diejenige Komponente verfeinert, an deren Ausgang der Kanal angeschlossen ist. Bei Eingabekanälen des Gesamtsystems, die an mehr als eine Komponente angeschlossen sind, ist eine gemeinsame Verfeinerung durch Einfügen einer neuen Komponente nicht möglich. Dies ist methodisch sinnvoll, da nicht sichergestellt ist, daß der Kanal vor seiner Verzweigungsstelle (physisch oder konzeptuell) verfügbar ist. Wenn man ermöglichen möchte, eine Komponente in einen Eingabekanal schon vor dessen Verzweigung einzufügen, so muß dies explizit in der Systemstruktur definiert sein.

Der Verfeinerungsbegriff nach Definition 4.3 erlaubt eine modulare Verfeinerung von Systemstrukturen:

Satz 4.1 (Modulare Verfeinerung von Systemstrukturen): Gegeben sei eine Menge von kombinierbaren Systemstrukturen $\{S_n : n \in N\}$, ein $m \in N$ und eine Systemstruktur S mit $S_m \rightsquigarrow S$, so daß $\{S_n : n \in N \backslash \{m\}\} \cup \{S\}$ kombinierbar ist. Dann gilt:

$$\|\{S_n : n \in N\} \rightsquigarrow \|(\{S_n : n \in N \backslash \{m\}\} \cup \{S\})$$

Beweis: Es genügt zu zeigen, daß die in Satz 4.1 angegebene Eigenschaft für aus jeweils einem Schritt bestehende Verfeinerungen gilt.

Aus $S_m \rightsquigarrow S$ folgt, daß sich S durch das Ersetzen einer Komponente z in S_m durch eine Systemstruktur S' ergibt. Die gleiche Ersetzung kann in der Systemstruktur $\|\{S_n : n \in N\}$ durchgeführt werden, wodurch sich die Systemstruktur $\|(\{S_n : n \in N \backslash \{m\}\} \cup \{S\})$ ergibt. □

4.2 Spurspezifikationen

In diesem Abschnitt wird der Formalismus der Spurspezifikationen genauer beschrieben. Eine Spurspezifikation besteht aus einer Systemstruktur und der Menge der zulässigen globalen Systemabläufe, wobei jeder solche Ablauf als *Spur,* das heißt als Sequenz von Aktionen modelliert wird.

Die Systemspezifikation durch Spuren stellt eine voll abstrakte und kompositionale Semantik für Netzwerke asynchron kommunizierender Agenten dar. Dies wurde in [Jon89] für Netze von I/O-Automaten gezeigt, und [Web92] enthält das entsprechende Ergebnis für Netzwerke von stromverarbeitenden Funktionen. Spurspezifikationen werden außerdem auch auf dem Gebiet der Prozeßalgebren [Hoa85] und für die Hardwaremodellierung [Dil89] eingesetzt.

Definition 4.4 (Spurspezifikation): Eine Spurspezifikation ist ein Paar (S, P), wobei S eine Systemstruktur $S = (In,\ Out,\ Cha)$ und $P \subseteq Acts^{\omega}$ ein Spurprädikat ist. □

Intuitiv beschreibt eine Spurspezifikation (S, P) solche Systeme, die die Struktur S haben und für deren Abläufe P gilt. Das Spurprädikat P kann beliebig sein. Es muß jedoch alle zulässigen Systemabläufe enthalten, da in späteren Verfeinerungsschritten die Vielfalt der Abläufe nur eingeschränkt, nicht aber erweitert werden kann. Eine Spezifikation, die "zu wenige" Spuren umfaßt, kann nicht implementiert werden. Was "zu wenig" genau heißt, hängt von den in den folgenden Entwicklungsschritten verwendeten Spezifikations- und Implementierungsformalismen ab. Beispielsweise muß, wenn eine Spurspezifikation zu einer funktionalen Spezifikation nach Abschnitt 4.3 verfeinert werden soll, in der Spurspezifikation für *jede* Sequenz von Eingabeaktionen eine entsprechende Spur vorhanden sein. Eine Klassifikation von Spurmengen hinsichtlich ihrer Realisierbarkeit, also ihrer Implementierbarkeit in verschiedenen Formalismen, findet sich in [BDD+91].

Bei der Spezifikation ist also darauf zu achten, daß die Spurmenge nicht so weit eingeschränkt wird, daß das System nicht mehr implementiert werden kann. Die Spurmenge darf jedoch "zu groß" sein; es dürfen also Spuren vorhanden sein, die sich nicht implementieren lassen. Dies sind zum Beispiel "prophetische" Spuren, bei denen Antworten schon vor den dafür erforderlichen Eingabedaten erscheinen. Dadurch vereinfachen sich häufig Spezifikationen, weil Aussagen über den gesamten Systemablauf ohne Rücksicht auf Kausalitätsbeziehungen gemacht werden. Das folgende Beispiel 4.3 zeigt eine Spurspezifikation der Prozessoreinheit *PSys* aus Beispiel 4.1. In den Prädikaten *SchedTrace* und *ProcTrace* sind jeweils "prophetische" Anteile enthalten.

Beispiel 4.3: Zur Spezifikation eines Spurprädikats $PSysTrace \subseteq (Act_{PSysStruct})^{\omega}$ definieren wir zunächst durch ein Hilfsprädikat *SchedTrace* die zulässigen Sequenzen von Rechenaufträgen an das System und Antworten vom System. Für jeden Identifikator

darf höchstens ein Rechenauftrag existieren. Das System liefert für jeden Rechenauftrag genau eine Antwort, und zwar entweder eine Fehlermeldung oder das korrekte Resultat:

$$
\begin{array}{l}
SchedTrace.t \;=\; \\
\quad \forall\, i \in Id : (Cmd_i \textbf{ on } SCmd \textbf{ in } t = \langle\rangle \;\wedge\; Rply_i \textbf{ on } SRply \textbf{ in } t = \langle\rangle) \vee \\
\qquad\quad (\exists\, p \in Prog,\, d \in Data : \\
\qquad\qquad Cmd_i \textbf{ on } SCmd \textbf{ in } t = \langle Task(i,p,d)\rangle \wedge \\
\qquad\qquad Rply_i \textbf{ on } SRply \textbf{ in } t \in \{\langle Result(i,\, p.d)\rangle, \langle Error(i)\rangle\})
\end{array}
$$

Die Anforderungen an die Kommunikation mit den einzelnen Rechenkernen sind weniger streng. Rechenaufträge können unbeantwortet bleiben. Für jede Antwort muß allerdings ein dazu passender Rechenauftrag existieren. Das heißt auch, daß eine Ergebnisantwort das korrekte Resultat enthalten muß:

$$
\begin{array}{l}
ProcTrace_{cc,rc}.t \;=\; \\
\quad \forall\, i \in Id,\, r \in Rply_i : \#(\{r\} \textbf{ on } rc \textbf{ in } t) \geq 1 \;\Rightarrow \\
\qquad (\exists\, p \in Prog,\, d \in Data \,:\, \#(\{Task(i,p,d)\} \textbf{ on } cc \textbf{ in } t) \geq 1 \wedge \\
\qquad\qquad\qquad\qquad r \in \{Result(i,p.d),\, Error(i)\})
\end{array}
$$

Das Gesamtsystem wird durch das Spurprädikat *PSysTrace* beschrieben:

$$
\begin{array}{l}
PSysTrace.t \;=\; SchedTrace.t \wedge \\
\qquad\qquad ProcTrace_{PCmd,P1Rply}.t \wedge ProcTrace_{PCmd,P2Rply}.t
\end{array}
$$

Insgesamt erhalten wir das Paar (*PSysStruct*, *PSysTrace*) als Spurspezifikation der Prozessoreinheit *PSys*. □

Die obige Spezifikation läßt gemäß dem Spurprädikat *SchedTrace* eine Implementierung zu, die ausschließlich Fehlermeldungen liefert. Hier stellt sich die grundsätzliche Frage, ob dies nicht durch weitere Anforderungen ausgeschlossen werden sollte. Zum Beispiel könnte man in der Art einer Fortschrittseigenschaft fordern, daß für unendlich viele Rechenaufträge auch unendlich viele erfolgreiche Antworten gegeben werden müssen. Es ist allerdings fraglich, ob eine solche zusätzliche Forderung tatsächlich praktischen Wert hätte, denn sie würde immer noch Implementierungen zulassen, die erfolgreiche Antworten nur nach jeweils unbrauchbar vielen Fehlschlägen oder nach unvertretbar langer Zeit liefern. Eine für die Praxis brauchbare Zusatzbedingung müßte genaue Anforderungen an die Leistungsfähigkeit des Gesamtsystems stellen, zum Beispiel durchschnittliche Antwortzeiten und Fehlerraten festlegen. Eine solche detaillierte Spezifikation wäre aber äußerst komplex.

Daher halten wir es für angemessen, wie in Beispiel 4.3 nur Anforderungen an die Korrektheit des Gesamtsystems zu stellen. Dies können Sicherheits- oder Lebendigkeitseigenschaften sein. Darüber hinausgehende Leistungsdaten werden auf der Ebene der Anforderungsspezifikation nicht formalisiert. Natürlich schließt dies nicht aus, daß

die weitere Systementwicklung mit dem Ziel einer maximalen Leistungsfähigkeit erfolgt. So kann zum Beispiel für zeitkritische Agenten in späteren Entwicklungsstufen ein zeitbehafteter Formalismus verwendet werden (siehe Beispiel 6.1).

Der Vorgang der Systementwicklung besteht aus einer Folge von *Verfeinerungsschritten,* die in verschiedenen Formalismen durchgeführt werden. Der Spurformalismus ist der Ausgangspunkt der Entwicklung und dient überdies als Korrektheitsmaßstab für die Verfeinerungsbegriffe in den anderen Formalismen. Wir geben nämlich zu jedem später verwendeten Formalismus (Mengen von Spurspezifikationen, funktionale Spezifikationen und Mengen davon) eine semantische Abbildung zu Spurspezifikationen an und zeigen, daß die Verfeinerungsbegriffe dieser Formalismen mit dem nun zu definierenden Verfeinerungsbegriff auf Spurspezifikationen verträglich sind.

Intuitiv sollen als Verfeinerungsschritte auf Spurspezifikationen das Verstärken der Systemeigenschaften und das Verfeinern der Systemstruktur erlaubt sein. Nachdem eine Verfeinerung der Systemstruktur das Einführen interner Kanäle ermöglicht, müssen auch neue Aktionen eingeführt werden können. Dann soll gelten, daß die Spuren des verfeinerten Systems eine Teilmenge der Spuren des ursprünglichen Systems sind, wenn man die neuen Aktionen herausfiltert. Dieser Verfeinerungsbegriff ist in der Terminologie von [Bro94b] sowohl eine Verfeinerung des internen Aufbaus als auch der Eigenschaften eines Systems, also eine Kombination aus horizontaler und vertikaler Verfeinerung.

Definition 4.5 (Verfeinerung von Spurspezifikationen):
Die Verfeinerungsrelation $\rightsquigarrow$ auf Spurspezifikationen ist für Spurspezifikationen (S, P) und (S', P') definiert durch:

$$(S,P) \rightsquigarrow (S',P') \;=\; S \rightsquigarrow S' \wedge (\forall\, t \in {Act_{S'}}^{\omega} : P.(Act_S \copyright t) \Leftarrow P'.t) \qquad \square$$

Anmerkung 4.2: Die Verfeinerungsrelation $\rightsquigarrow$ auf Spurspezifikationen, deren Systemstruktur mindestens eine Komponente aufweist, ist reflexiv und transitiv. $\square$

Das Einführen neuer Aktionen ist bei dem gerade definierten Verfeinerungsbegriff nur auf internen Kanälen möglich, die durch eine Verfeinerung der Systemstruktur neu entstanden sind. Da die Ein- und Ausgabekanäle eines Systems durch die Verfeinerung der Systemstruktur nicht verändert (oder ergänzt) werden können, sind auch keine neuen Ein- oder Ausgabeaktionen möglich.

Beispiel 4.4: Die Spurspezifikation $(PSysStruct, PSysTrace)$ von Beispiel 4.3 soll zunächst an die Systemstruktur $PSysStruct'$ von Beispiel 4.2 angepaßt werden. Wir definieren das Prädikat $PSysTrace'$:

$$\begin{aligned} PSysTrace' &\subseteq {Act_{PSysStruct'}}^{\omega} \\ PSysTrace'.t &= PSysTrace.(Act_{PSysStruct} \copyright t) \end{aligned}$$

Offensichtlich gilt $(PSysStruct, PSysTrace) \rightsquigarrow (PSysStruct', PSysTrace')$. Man beachte, daß $PSysTrace'$ keine Einschränkungen bezüglich solcher Aktionen enthält, die in $PSysStruct'$, nicht jedoch in $PSysStruct$ auftreten können.

In einem nächsten Verfeinerungsschritt soll das Spurprädikat $PSysTrace'$ sinnvoll eingeschränkt werden. Dieses Prädikat setzt in der verfeinerten Systemstruktur $PSysStruct'$ Aktionen in Beziehung, die verschiedenen Agenten zugeordnet sind. Dies sind beispielsweise im Prädikat $SchedTrace$ die Befehle auf Kanal $SCmd$, der ein Eingabekanal des Agenten $Mixer$ ist, und die Antworten auf Kanal $SRply$, einem Ausgabekanal des Agenten $Feeder$. Zur Implementierung einer Komponente wird aber im Endeffekt eine sogenannte *lokale* Beschreibung benötigt. Das ist eine Beschreibung, in der die geforderten Eigenschaften in Prädikate gruppiert sind, die jeweils nur die Aktionen eines Teils des Systems (im Extremfall nur einer Komponente) betreffen. Um eine solche Beschreibung zu erhalten, ist es erforderlich, die Spurspezifikation so zu verstärken, daß auch die auf den internen Kanälen übertragenen Nachrichten hinreichend genau bestimmt sind. Dazu geben wir im folgenden Spurprädikate an, die die Aktionen der Komponenten $Feeder$, $Timer$ und $Mixer$ zueinander in Beziehung setzen.

Der Agent $Feeder$ verarbeitet die vom Agenten $Mixer$ gelieferten Kommandos und Antworten. Auf dem Kanal $MtoF$ eingehende Kommandos werden mindestens einmal an den Kanal $FtoT$ weitergegeben. Unter der Voraussetzung, daß für jedes ausgegebene Kommando mindestens eine Antwort eingeht, soll schließlich genau eine dieser Antworten an den Kanal $SRply$ geliefert werden:

$$
\begin{array}{l}
FeederTrace.t \;=\; \forall\, i \in Id : \\
\quad (Cmd_i \textbf{ on } MtoF \textbf{ in } t = \langle\rangle \;\Rightarrow \\
\qquad (Cmd_i \textbf{ on } FtoT \textbf{ in } t = \langle\rangle \;\wedge\; Rply_i \textbf{ on } SRply \textbf{ in } t = \langle\rangle)) \wedge \\
\quad (\#(Cmd_i \textbf{ on } MtoF \textbf{ in } t) = 1 \;\Rightarrow \\
\qquad (\exists\, n \in \mathbb{N} : \#(\{ft.(Cmd_i \textbf{ on } MtoF \textbf{ in } t)\}) \textbf{ on } FtoT \textbf{ in } t) = n + 1 \;\wedge \\
\qquad\qquad (\#(Rply_i \textbf{ on } MtoF \textbf{ in } t) > n \;\Rightarrow \\
\qquad\qquad\quad (\exists\, r \in Rply_i : Rply_i \textbf{ on } SRply \textbf{ in } t = \langle r\rangle \;\wedge \\
\qquad\qquad\qquad\qquad \#(\{r\}) \textbf{ on } MtoF \textbf{ in } t) \geq 1))))
\end{array}
$$

Der Agent $Timer$ gibt jedes Kommando mit Identifikator i an den Ausgang $PCmd$ weiter. Außerdem wird eine Nachricht $Error(i)$ (mit zeitlicher Verzögerung, die jedoch in der in diesem Kapitel dargestellten Ausprägung des Spurformalismus nicht modelliert werden kann) an den Ausgang $TtoM$ gegeben. Diese Nachricht signalisiert einen Zeitfehler. Die Verzögerung, mit der sie erzeugt wird, beeinflußt die Leistung des Prozessorsystems, nicht jedoch dessen Korrektheit, da das Prozessorsystem für *jede* Verzögerungszeit zulässige Antworten liefern muß:

$$
\begin{array}{l}
TimerTrace.t \;= \\
\quad Cmd \textbf{ on } FtoT \textbf{ in } t = Cmd \textbf{ on } PCmd \textbf{ in } t \;\wedge \\
\quad \forall\, i \in Id : \#(Cmd_i \textbf{ on } FtoT \textbf{ in } t) = \#(\{Error(i)\}) \textbf{ on } TtoM \textbf{ in } t)
\end{array}
$$

Der Agent *Mixer* gibt eine nicht-strikte faire Mischung der eingehenden Nachrichten ohne Berücksichtigung der Reihenfolge aus. Dabei ist das erste Konjunktionsglied der folgenden Gleichung eine Anforderung an die Umgebung. Der Agent *Mixer* kann also nur in solchen Umgebungen implementiert werden, die diese Anforderung erfüllen:

$$\begin{aligned} &MixerTrace.t = \\ &\quad \forall\, i \in Id : \#(Cmd_i \textbf{ on } SCmd \textbf{ in } t) \leq 1 \wedge \\ &\quad \forall\, c \in Cmd : \#(\{c\} \textbf{ on } MtoF \textbf{ in } t) = \#(\{c\} \textbf{ on } SCmd \textbf{ in } t) \wedge \\ &\quad \forall\, r \in Rply : \#(\{r\} \textbf{ on } MtoF \textbf{ in } t) = \#(\{r\} \textbf{ on } P1Rply \textbf{ in } t) + \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \#(\{r\} \textbf{ on } P2Rply \textbf{ in } t) + \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \#(\{r\} \textbf{ on } TtoM \textbf{ in } t) \end{aligned}$$

Als Spurprädikat für das verfeinerte Gesamtsystem erhalten wir:

$$\begin{aligned} PSysTrace''.t = {} & FeederTrace.t \wedge TimerTrace.t \wedge MixerTrace.t \wedge \\ & ProcTrace_{PCmd,P1Rply}.t \wedge ProcTrace_{PCmd,P2Rply}.t \end{aligned}$$

Insgesamt ergibt sich $(PSysStruct', PSysTrace') \rightsquigarrow (PSysStruct', PSysTrace'')$ und wegen der Transitivität der Verfeinerungsrelation auch $(PSysStruct, PSysTrace) \rightsquigarrow (PSysStruct', PSysTrace'')$. □

Die bisher eingeführten Begriffe betrafen jeweils *eine* Spezifikation. Eine wichtige Eigenschaft jeder Entwurfsmethodik ist aber die Modularität, also die Möglichkeit, die einzelnen Teile eines Systems getrennt voneinander zu entwickeln. Um eine modulare Systementwicklung im Spurformalismus durchführen zu können und um den Übergang in den funktionalen Formalismus zu ermöglichen, geben wir im folgenden Operationen zur Kombination und Aufspaltung von Spurspezifikationen an. Als Hilfsfunktion definieren wir zunächst einen Kombinationsoperator, mit dem Mengen von Spurspezifikationen auf einzelne Spurspezifikationen abgebildet werden:

Definition 4.6 (Kombinierbarkeit und Kombination von Spurspezifikationen):
Gegeben sei eine endliche, nicht-leere Menge von kombinierbaren Spurspezifikationen $\{(S_n, P_n) : n \in N\}$, wobei die Spurspezifikationen *kombinierbar* heißen, wenn deren Systemstrukturen $S_n = (In_n, Out_n, Cha_n)$ kombinierbar sind. Es bezeichne $S = (In, Out, Cha)$ die Kombination $\|_{n \in N}\, S_n$ der Systemstrukturen. Weiterhin sei das Prädikat $P \subseteq {Act_S}^{\omega}$ definiert durch:

$$P.t = \forall\, n \in N : P_n.(Act_{S_n} \textcircled{c} t)$$

Dann ist die *Kombination* $\|_{n \in N}\, (S_n, P_n)$ (auch geschrieben als $\|\{(S_n, P_n) : n \in N\}$) die Spurspezifikation (S, P). □

Anmerkung 4.3: Für alle Spurspezifikationen (S, P) gilt $\|\{(S, P)\} = (S, P)$. □

Durch den Kombinationsoperator wird die Semantik einer kombinierbaren Menge Σ von Spurspezifikationen definiert vermöge $[\![\, \Sigma \,]\!]_{tra} = \|\Sigma$. Wir erweitern die Verfei-

nerungsrelation auch auf Mengen von Spurspezifikationen durch die Definition, daß $\Sigma \rightsquigarrow \Sigma'$ genau für $[\![\Sigma]\!]_{tra} \rightsquigarrow [\![\Sigma']\!]_{tra}$ gelten soll. Wegen Anmerkung 4.3 läßt sich der Verfeinerungsbegriff problemlos auch auf den Übergang von einer Spurspezifikation auf die entsprechende einelementige Menge vermöge $(S, P) \rightsquigarrow \{(S, P)\}$ erweitern.

Es ist also als Verfeinerungsschritt möglich, eine einzelne Spurspezifikation in mehrere Spurspezifikationen aufzuspalten, die sich jeweils auf einen Teil des ursprünglichen Systems beziehen. Solche *lokalen* Spurspezifikationen können natürlich nur über diejenigen Aktionen Aussagen machen, die Komponenten des jeweiligen Teilsystems betreffen. Im Extremfall kann eine Spurspezifikation in Teilsysteme aufgespalten werden, die jeweils nur *eine* Komponente aufweisen. Um eine lokale Spurspezifikation zu erhalten, sind dabei im allgemeinen Entwurfsentscheidungen notwendig (siehe Beispiel 4.4 sowie Abschnitt 4.4). Wenn sich die einzelnen lokalen Spezifikationen über den Kombinationsoperator wieder zur ursprünglichen Spezifikation (oder einer Verfeinerung davon) zusammenfügen lassen, so stellt die Aufspaltung einen Verfeinerungsschritt dar.

Beispiel 4.5: Die in Beispiel 4.4 angegebenen Spurprädikate sind bereits lokal zu den einzelnen Komponenten des zu entwickelnden Systems. Daher kann die Spurspezifikation $(PSysStruct', PSysTrace'')$ aufgespalten werden in die Menge Σ mit:

$$\begin{aligned}\Sigma = \{\ & (CompStruct_{PSysStruct'}.Feeder,\ FeederTrace),\\ & (CompStruct_{PSysStruct'}.Timer,\ TimerTrace),\\ & (CompStruct_{PSysStruct'}.Mixer,\ MixerTrace),\\ & (CompStruct_{PSysStruct}.Proc1,\ ProcTrace_{PCmd,P1Rply}),\\ & (CompStruct_{PSysStruct}.Proc2,\ ProcTrace_{PCmd,P2Rply})\ \}\end{aligned}$$

Es gilt $\|\Sigma = (PSysStruct', PSysTrace'')$ und damit in Verbindung mit dem Ergebnis von Beispiel 4.4 insgesamt $(PSysStruct, PSysTrace) \rightsquigarrow \Sigma$.

Zur Vereinfachung haben wir in diesem Beispiel und in den Beispielen 4.3 und 4.4 eine explizite Typangabe bei den je eine Komponente beschreibenden Spurprädikaten weggelassen. Der genaue Typ ergibt sich jeweils aus dem Kontext. □

Der folgende Satz zeigt, daß die Verfeinerung einer einzelnen Komponente mit dem Verfeinerungsoperator auf Spezifikationsmengen verträglich ist. Damit ist eine modulare Systementwicklung möglich, denn die einzelnen lokalen Spezifikationen können unabhängig voneinander weiterentwickelt werden. Wie schon bei Systemstrukturen müssen zwar formal Namenskonflikte ausgeschlossen werden, da aber tatsächlich die Entwicklung nur zu immer weiteren Aufspaltungen verläuft, spielen diese in der praktischen Anwendung keine Rolle.

Satz 4.2 (Modulare Verfeinerung von Spurspezifikationen): Gegeben sei eine Menge von kombinierbaren Spurspezifikationen $\{(S_n, P_n) : n \in N\}$, ein $m \in N$ und eine

Spurspezifikation (S, P), so daß $\{(S_n, P_n) : n \in N \backslash \{m\}\} \cup \{(S, P)\}$ kombinierbar ist und $(S_m, P_m) \rightsquigarrow (S, P)$ gilt. Dann gilt auch:

$$\{(S_n, P_n) : n \in N\} \rightsquigarrow \{(S_n, P_n) : n \in N \backslash \{m\}\} \cup \{(S, P)\}$$

Beweis: Es genügt, unter den Voraussetzungen des Satzes 4.2 die folgenden Eigenschaften zu zeigen:

(1) $\|\{S_n : n \in N\} \rightsquigarrow \|(\{S_n : n \in N \backslash \{m\}\} \cup \{S\})$

(2) $\forall\, t \in (Act_S \cup (\bigcup_{n \in N \backslash \{m\}} Act_{S_n}))^\omega : Q.((\bigcup_{n \in N} Act_{S_n}) \copyright t) \Leftarrow R.t$
wobei $Q.t = \forall\, n \in N : P_n.(Act_{S_n} \copyright t)$
$R.t = P.(Act_S \copyright t) \wedge (\forall\, n \in N \backslash \{m\} : P_n.(Act_{S_n} \copyright t))$

Eigenschaft (1) folgt aus Satz 4.1. Eigenschaft (2) erhält man durch Einsetzen der Definitionen 4.5 und 4.6. □

4.3 Funktionale Spezifikationen

In diesem Abschnitt wird der funktionale Formalismus als zweite Stufe der Entwurfsmethodik dargestellt. Eine funktionale Spezifikation besteht aus einer Systemstruktur und der Spezifikation eines Agenten, dessen Ein- und Ausgabekanäle mit denen der Systemstruktur übereinstimmen. In jeder funktionalen Spezifikation wird also nur das Ein-/Ausgabeverhalten *eines* Agenten beschrieben, der, falls es sich um eine Systemstruktur mit mehreren Komponenten handelt, eine innere Struktur haben kann. Die Aufspaltung in ein dieser Struktur entsprechendes Netz von Agenten geschieht dann, ähnlich wie auf der Spurebene, über eine Menge von funktionalen Spezifikationen.

Es ist ein klassisches Ergebnis, daß stromverarbeitende Funktionen sich zur kompositionalen Modellierung von Netzen kommunizierender Agenten eignen [Kah74, KM77]. Wie schon in Abschnitt 3.5 erläutert, gilt dies auch für die hier verwendeten Spezifikationen durch Prädikate über solchen Funktionen.

Definition 4.7 (Funktionale Spezifikationen): Eine funktionale Spezifikation ist ein Paar (S, F), wobei $S = (In, Out, Cha)$ eine Systemstruktur ist und F eine Agentenspezifikation nach Abschnitt 3.5, für die gilt:

$$F \subseteq \otimes_{c \in Inputs.S} c : (Cha.c)^\omega \twoheadrightarrow \otimes_{c \in Outputs.S} c : (Cha.c)^\omega \qquad \square$$

Intuitiv beschreibt eine funktionale Spezifikation Systeme mit der Struktur S, deren funktionales Verhalten das einer stromverarbeitenden Funktion aus F ist. Auch hier charakterisiert die Systemstruktur die zumindest geforderten Verteilungseigenschaften, so daß eine Implementierung nur korrekt hinsichtlich einer funktionalen Spezifikation ist, wenn sie die geforderten funktionalen *und* die Verteilungseigenschaften aufweist.

Eine funktionale Spezifikation (S, F) beschreibt nur das Ein-/Ausgabeverhalten des Systems S. Sie macht also, im Gegensatz zu einer Spurspezifikation, keine Aussagen über die Kommunikation auf internen Kanälen. In der im folgenden gegebenen Spursemantik für funktionale Spezifikationen drückt sich dies dadurch aus, daß die Spuren beliebige interne Aktionen enthalten dürfen. Nur die Ein- und Ausgabeaktionen müssen den durch das Prädikat F beschriebenen Abläufen entsprechen.

Definition 4.8 (Spursemantik einer funktionalen Spezifikation): Die Spursemantik einer funktionalen Spezifikation (S, F) mit $F \subseteq IS \nrightarrow OS$ ist:

$$[\![(S, F)]\!]_{tra} = (S, \{t \in Acts^{\omega} : (Act.IS \cup Act.OS) \copyright t \in Traces.F\}) \qquad \square$$

Wir definieren die Verfeinerung zwischen Spurspezifikation und funktionaler Spezifikation durch direkten Rückgriff auf die Spursemantik:

Definition 4.9 (Verfeinerung durch Ebenenübergang):
Eine Spurspezifikation (S, P) wird durch eine funktionale Spezifikation (S, F) verfeinert, wenn gilt: $(S, P) \rightsquigarrow [\![(S, F)]\!]_{tra}$. $\square$

Wegen der in Definition 4.8 gegebenen Spursemantik ist bei Systemen mit mehreren Komponenten ein Ebenenübergang von der Spurspezifikation zu einer funktionalen Spezifikation nur möglich, wenn die Spurspezifikation keinerlei Einschränkungen hinsichtlich der internen Aktionen enthält. Andernfalls muß man, entsprechend der FOCUS-Methodik, die Spuranforderungen zunächst lokalisieren und dann das Gesamtsystem in Teilsysteme aufspalten. Zumindest nach einer Aufspaltung in einzelne Komponenten kann der Ebenenübergang erfolgen, weil ein aus nur einer Komponente bestehendes Teilsystem keine internen Kanäle enthält.

Es ist oftmals unpraktisch, die beim Ebenenübergang notwendigen Beweise direkt auf die Definition 4.9 abgestützt zu führen. Ein etwas einfacheres Beweisprinzip (die Verwendung von Spuren in sogenannter Normalform) findet sich in [BDD+92a, Seite 42f]; dieses Prinzip wird zum Beispiel in [Ded90] angewendet.

Bei der Systementwicklung in der Praxis wird man für den Ebenenübergang bevorzugt syntaktische Transformationsregeln wie die folgenden benutzen, die eine erweiterte Fassung der in [DW92a] vorgestellten sind:

Satz 4.3: In einer Spurspezifikation (S, P) sei $Comp.S = \{z\}$, $IS = InType_S.z$ und $OS = OutType_S.z$. P habe die Form $P.t = Q.(IS \circledS t).(OS \circledS t)$ für ein Prädikat $Q \in IS \to OS \to \mathbb{B}$. Dann wird (S, P) verfeinert durch (S, F), wobei $F \subseteq IS \nrightarrow OS$ definiert ist durch:

$$F.f = \forall\, x \in IS : Q.x.(f.x)$$

Beweis: Nach Definition 4.9 ist $(S,P) \rightsquigarrow [\![(S,F)]\!]_{tra}$ zu zeigen, was nach den Definitionen 4.8 und 4.5, und weil S keine internen Kanäle aufweist, mit $P \Leftarrow Traces.F$ gleichbedeutend ist.

Gegeben sei ein $t \in (Act.IS \cup Act.OS)^\omega$ mit $Traces.F.t$. Dann existiert nach Definition 3.5 ein $f \in F$ mit $OS ⓢ t = f.(IS ⓢ t)$ $(*)$. Für dieses f gilt unter den Voraussetzungen von Satz 4.3 $Q.x.(f.x)$ für alle $x \in IS$. Für $x = IS ⓢ t$ ergibt sich $Q.(IS ⓢ t).(f.(IS ⓢ t))$ und nach $(*)$ $Q.(IS ⓢ t).(OS ⓢ t)$. Aus der in Satz 4.3 geforderten Struktur von P erhält man schließlich $P.t$. □

Satz 4.4: In einer Spurspezifikation (S,P) sei $Comp.S = \{z\}$, $IS = InType_S.z$ und $OS = OutType_S.z$. P habe die Form $P.t = \forall\, s \in (Act.IS \cup Act.OS)^* : s \leq t \Rightarrow Q.(IS ⓢ s).(OS ⓢ s)$ für ein Prädikat $Q \in IS \to OS \to \mathbb{B}$. Dann wird (S,P) verfeinert durch (S,F), wobei $F \subseteq IS \twoheadrightarrow OS$ definiert ist durch:

$$F.f = \forall\, x \in IS^{fin}, y \in OS^{fin} : y \leq f.x \Rightarrow Q.x.y$$

Beweis: Gegeben sei ein beliebiges $t \in (Act.IS \cup Act.OS)^\omega$ mit $Traces.F.t$. Zu zeigen ist, wie schon in Satz 4.3, $P.t$.

Nach Definition 3.5 existiert ein $f \in F$ mit $OS ⓢ s \leq f.(IS ⓢ s)$ für alle $s \in (Act.IS \cup Act.OS)^*$. Setzt man $x = IS ⓢ s$ und $y = OS ⓢ s$, so gilt $x \in IS^{fin}$, $y \in OS^{fin}$ und $y \leq f.x$. Unter den Voraussetzungen von Satz 4.4 erhält man $Q.x.y$ für alle $s \in (Act.IS \cup Act.OS)^*$ mit $s \leq t$. Nach der in Satz 4.4 geforderten Struktur von P folgt damit $P.t$. □

Beispiel 4.6: Wir schreiben abkürzend $FeederStruct$ für $CompStruct_{PSysStruct'}.Feeder$. Durch die Anwendung von Satz 4.3 auf eine geeignet umgeformte Fassung der Spurspezifikation $(FeederStruct, FeederTrace)$ erhält man das folgende Prädikat $FeederFun$, für das $(FeederStruct, FeederTrace) \rightsquigarrow (FeederStruct, FeederFun)$ gilt:

$$\begin{array}{l}
FeederFun \subseteq InType_{PSysStruct'}.Feeder \twoheadrightarrow OutType_{PSysStruct'}.Feeder \\
FeederFun.f \;= \\
\quad \forall\, x \in InType_{PSysStruct'}.Feeder,\; i \in Id : \\
\qquad (Cmd_i ⓒ x.MtoF = \langle\rangle \;\Rightarrow \\
\qquad\quad (Cmd_i ⓒ f.x.FtoT = \langle\rangle \;\wedge\; Rply_i ⓒ f.x.SRply = \langle\rangle)) \wedge \\
\qquad (\#Cmd_i ⓒ x.MtoF = 1 \;\Rightarrow \\
\qquad\quad (\exists\, n \in \mathbb{N} : \#\{ft.(Cmd_i ⓒ x.MtoF)\} ⓒ f.x.FtoT = n+1 \wedge \\
\qquad\qquad\quad (\#Rply_i ⓒ x.MtoF > n \;\Rightarrow \\
\qquad\qquad\qquad (\exists\, r \in Rply_i : Rply_i ⓒ f.x.SRply = \langle r\rangle \wedge \\
\qquad\qquad\qquad\qquad \#\{r\} ⓒ x.MtoF \geq 1))))
\end{array}$$ □

Der Verfeinerungsbegriff innerhalb des Formalismus der funktionalen Spezifikation ist analog zu dem entsprechenden Begriff bei Spurspezifikationen definiert als eine Verfei-

nerung der Systemstruktur bei gleichzeitiger Einschränkung der Verhaltensmöglichkeiten des Agenten. Auch hier ist also sowohl eine Verfeinerung des internen Aufbaus als auch der Eigenschaften des Systems möglich.

Definition 4.10 (Verfeinerung von funktionalen Spezifikationen):
Die Verfeinerungsrelation $\rightsquigarrow$ auf funktionalen Spezifikationen ist für funktionale Spezifikationen (S, F) und (S', F') definiert durch:

$$(S, F) \rightsquigarrow (S', F') \;=\; S \rightsquigarrow S' \wedge F \Leftarrow F' \qquad \square$$

Diese Verfeinerungsrelation ist wohldefiniert, weil sich durch das Verfeinern einer Systemstruktur die Ein- und Ausgabekanäle des Systems nicht ändern. Nach Satz 4.1 und wegen der entsprechenden Eigenschaften der Implikation ist die Relation $\rightsquigarrow$ auf funktionalen Spezifikationen reflexiv und transitiv.

Beispiel 4.7: Für das in Beispiel 5.1 definierte Prädikat *FeederFun'* gilt $(\mathit{FeederStruct}, \mathit{FeederFun}) \rightsquigarrow (\mathit{FeederStruct}, \mathit{FeederFun}')$. $\square$

Satz 4.5 zeigt, daß der Verfeinerungsbegriff auf der funktionalen Ebene verträglich mit dem auf der Spurebene ist:

Satz 4.5: Für alle funktionalen Spezifikationen (S, F) und (S', F') gilt:

$$(S, F) \rightsquigarrow (S', F') \;\Rightarrow\; [\![(S, F)]\!]_{tra} \rightsquigarrow [\![(S', F')]\!]_{tra}$$

Beweis: Gegeben seien funktionale Spezifikationen (S, F) und (S', F') mit $(S, F) \rightsquigarrow (S', F')$, was nach Definition 4.10 mit $S \rightsquigarrow S' \wedge F \Leftarrow F'$ gleichbedeutend ist. Wegen $S \rightsquigarrow S'$ haben die Systemstrukturen S und S' identische Ein- und Ausgabekanäle; wir bezeichnen den Eingabekanaltyp von S beziehungsweise S' mit IS und den Ausgabekanaltyp von S beziehungsweise S' mit OS.

Aus $F \Leftarrow F'$ folgt $\mathit{Traces}.F \supseteq \mathit{Traces}.F'$. Somit ergibt sich für alle $t' \in {\mathit{Act}_{S'}}^{\omega}$ mit $(\mathit{Act}.IS \cup \mathit{Act}.OS) \copyright t' \in \mathit{Traces}.F'$:

$$\mathit{Act}_S \copyright t' \in \{t \in {\mathit{Act}_S}^{\omega} : (\mathit{Act}.IS \cup \mathit{Act}.OS) \copyright t \in \mathit{Traces}.F\}$$

Daraus erhalten wir nach Definition 4.5 der Verfeinerungsrelation auf Spurspezifikationen und wegen $S \rightsquigarrow S'$:

$$\begin{aligned} &(S,\ \{t \in {\mathit{Act}_S}^{\omega} : (\mathit{Act}.IS \cup \mathit{Act}.OS) \copyright t \in \mathit{Traces}.F\}) \\ &\rightsquigarrow\ (S',\ \{t \in {\mathit{Act}'_S}^{\omega} : (\mathit{Act}.IS \cup \mathit{Act}.OS) \copyright t \in \mathit{Traces}.F'\}) \end{aligned}$$

Schließlich folgt nach Definition 4.8 die gewünschte Aussage:

$$[\![(S, F)]\!]_{tra} \rightsquigarrow [\![(S', F')]\!]_{tra} \qquad \square$$

Die Umkehrung von Satz 4.5 gilt nicht, da es $F \neq F'$ mit $Traces.F = Traces.F'$ gibt. Gegeben sei beispielsweise die Agentenspezifikation F aus Gegenbeispiel 3.2:

$$F = \{f \in IS \twoheadrightarrow OS : \exists\, y \in OS : \forall\, x \in IS : f.x \leq y\}$$

Für dieses F und $F' = IS \twoheadrightarrow OS$ gilt offensichtlich $Traces.F = Traces.F' = (Act.IS \cup Act.OS)^{\omega}$, aber $F \Leftarrow F'$ ist nicht erfüllt.

Wie bereits erwähnt, beschreibt bei einer funktionalen Spezifikation (S, F) das Prädikat F nur die Kommunikation auf den Ein- und Ausgabekanälen des Systems. Dies schließt jedoch nicht aus, daß die Struktur des Prädikats F dem inneren Aufbau des Systems angepaßt ist. Beispielsweise kann das Prädikat F für $S = (In, Out, Cha)$ die folgende Form haben, wobei für jede Komponente $z \in Comp.S$ ein Hilfsprädikat F_z vorgesehen ist, das die Ein- und Ausgaben dieser Komponente zueinander in Beziehung setzt:

$$\begin{aligned} F.f \;=\; & \forall\, x \in \otimes_{c \in Inputs.S}\, c : (Cha.c)^{\omega} : \\ & \exists\, y \in \otimes_{c \in Outputs.S}\, c : (Cha.c)^{\omega},\; w \in \otimes_{c \in Internals.S}\, c : (Cha.c)^{\omega} : \\ & f.x = y \;\wedge\; (\bigwedge_{z \in Comp.S} F_z.((x \cup w)|_{In.z}).((y \cup w)|_{Out.z})) \end{aligned}$$

Auch auf der funktionalen Ebene ist es wünschenswert, eine einzelne Spezifikation in mehrere Spezifikationen aufspalten zu können. Dazu wird im folgenden zunächst ein Kombinationsoperator für funktionale Spezifikationen definiert. Die Aufspaltung einer funktionalen Spezifikation ist ein Verfeinerungsschritt, wenn sich die erhaltenen Spezifikationen wieder zu der ursprünglichen (oder einer Verfeinerung davon) zusammensetzen lassen. Methodisch dient eine Aufspaltung der Modularisierung. Außerdem lassen sich nur durch eine Aufspaltung die Eigenschaften einzelner Agenten eines ursprünglich größeren Systems definieren, weil dabei interne Kanäle zu Ein- oder Ausgabekanälen werden. Der Aufspaltungsschritt ist unproblematisch möglich, wenn das Prädikat F der ursprünglichen funktionalen Spezifikation die oben angegebene Struktur hat.

Definition 4.11 (Kombination von funktionalen Spezifikationen):
Gegeben sei eine endliche, nicht-leere Menge von kombinierbaren funktionalen Spezifikationen $\{(S_n, F_n) : n \in N\}$, wobei die Spezifikationen *kombinierbar* heißen, wenn deren Systemstrukturen $S_n = (In_n, Out_n, Cha_n)$ kombinierbar sind. Dann ist die *Kombination* $\|_{n \in N}\,(S_n, F_n)$ (auch geschrieben als $\|\{(S_n, F_n) : n \in N\}$) die funktionale Spezifikation $(\|_{n \in N}\, S_n, \|_{n \in N}\, F_n)$. □

In der obigen Definition ist der erste Verbindungsoperator der auf Systemstrukturen, der zweite der auf Agentenspezifikationen. Die Wohldefiniertheit der Definition beruht darauf, daß die Kombinierbarkeit der Systemstrukturen S_n auch die Verbindbarkeit der Agentenspezifikationen F_n impliziert.

Anmerkung 4.4: Für alle funktionalen Spezifikationen (S, F) gilt $\|\{(S, F)\} = (S, F)$. □

Die semantische Abbildung einer kombinierbaren Menge Σ von funktionalen Spezifikationen zu einer einzigen funktionalen Spezifikation wird über den Kombinationsoperator definiert vermöge $[\![\Sigma]\!]_{fun} = \|\Sigma$. Auch der Verfeinerungsbegriff wird durch $\Sigma \rightsquigarrow \Sigma'$ genau für $[\![\Sigma]\!]_{fun} \rightsquigarrow [\![\Sigma']\!]_{fun}$ entsprechend erweitert. Die Verfeinerung ist wegen Satz 4.5 verträglich mit dem Verfeinerungsbegriff auf Spurspezifikationen. Auch den Übergang zwischen einer funktionalen Spezifikation (S, F) und der einelementigen Menge $\{(S, F)\}$ bezeichnen wir als Verfeinerung, was nach Anmerkung 4.4 mit der semantischen Abbildung von Spezifikationsmengen verträglich ist.

Im folgenden Satz 4.6 wird schließlich noch gezeigt, daß auch im funktionalen Formalismus eine modulare Verfeinerung einzelner Elemente aus einer Menge von Spezifikationen möglich ist:

Satz 4.6 (Modulare Verfeinerung von funktionalen Spezifikationen): Gegeben sei eine Menge von kombinierbaren funktionalen Spezifikationen $\{(S_n, F_n) : n \in N\}$, ein $m \in N$ und eine funktionale Spezifikation (S, F), so daß $\{(S_n, F_n) : n \in N \backslash \{m\}\} \cup \{(S, F)\}$ kombinierbar ist und $(S_m, F_m) \rightsquigarrow (S, F)$ gilt. Dann gilt auch:

$$\{(S_n, F_n) : n \in N\} \rightsquigarrow \{(S_n, F_n) : n \in N \backslash \{m\}\} \cup \{(S, F)\}$$

Beweis: Der Satz folgt aus Satz 4.1 sowie durch Einsetzen der Definitionen 4.10 und 4.11. □

4.4 Der Entwicklungsprozeß

In diesem Abschnitt geben wir einen Überblick über eine Systementwicklung, in der die beiden Formalismen der Spurspezifikation und funktionalen Spezifikation angewendet werden. Eine solche vollständige Systementwicklung ist in der Praxis eher untypisch. Oft liegt zum Beispiel die Aufteilung der Anforderungen auf die einzelnen Komponenten schon sehr früh fest. Dann kann man die Entwicklung auf der Spurebene weglassen und gleich mit der funktionalen Spezifikation beginnen. Ebenso kann man die funktionale Ebene weglassen, wenn eine Implementierungssprache gewählt wird, für die eine Spursemantik existiert.

Der Startpunkt einer Systementwicklung, also Nummer 1 in Bild 4.1, ist die Erarbeitung einer *globalen Anforderungsspezifikation* im Spurformalismus. Eine globale Anforderungsspezifikation beschreibt die zulässigen Abläufe und gewisse strukturelle Eigenschaften (die mindestens geforderte Verteilung und die höchstens erlaubten Kommunikationsverbindungen) des Gesamtsystems.

Die Charakterisierung der zulässigen Abläufe des Systems kann auch implizite Anforderungen an die Umgebung enthalten. In Beispiel 4.3 ist dies die Forderung, daß

höchstens ein Befehl für jeden Befehlsidentifikator auftreten darf. Das System kann dann nur in solchen Umgebungen realisiert werden, die die Anforderungen erfüllen. Hinsichtlich der Eigenschaften des Systems wird oftmals nur die Kommunikation an den Ein- und Ausgabekanälen eingeschränkt. Es kann aber sinnvoll sein, Eigenschaften auch für interne Kanäle zu fordern. Eine solche Forderung könnte zum Beispiel die Beschränkung der Kommunikationsrate für interne Kanäle sein.

Natürlich ist es im allgemeinen ein schwieriger Prozeß, aus einer anfangs noch ungenauen Vorstellung vom Verhalten eines Systems eine globale Anforderungsspezifikation zu erstellen. Auf diesen Vorgang des *requirements engineering* wird hier nicht näher eingegangen. Es kann dabei notwendig sein, die Spezifikation zu *validieren,* das heißt, ihr Übereinstimmen mit den Vorstellungen des Auftraggebers beispielsweise durch das Durchspielen von Testfällen zu überprüfen.

Ist die globale Anforderungsspezifikation erstellt, dann müssen die Anforderungen *lokalisiert,* das heißt einzelnen Teilen des Systems zugeordnet werden. Das Ziel ist es, eine lokale Anforderungsspezifikation zu erhalten. Das ist eine die ursprüngliche Spezifikation verfeinernde Menge von Spurspezifikationen, die jeweils nur eine Aussage über ein Teilsystem machen. Eine lokale Anforderungsspezifikation ist also eine Kollektion getrennter Spezifikationen von Teilsystemen, im Extremfall der einzelnen Komponenten. Im Spurformalismus entspricht sie der Nummer 2 in Bild 4.1. Die Spezifikation kann dann aufgespalten werden, und die weitere Systementwicklung erfolgt modular.

Um eine lokale Anforderungsspezifikation zu erhalten, sind im allgemeinen Entwurfsentscheidungen erforderlich, da sich die globalen Anforderungen in unterschiedlicher Weise auf die einzelnen Komponenten aufteilen lassen. Insbesondere ist es notwendig, die zwischen den Komponenten (also auf internen Kanälen des Gesamtsystems) ausgetauschten Informationen zu spezifizieren. Für Sicherheitseigenschaften existiert dabei eine kanonische Aufspaltung [Web92].

Als Verfeinerungsschritte zur Entwicklung der lokalen Anforderungsspezifikation sind möglich:

- Verfeinern der Systemstruktur (erstes Konjunktionsglied von Definition 4.5),
- Verstärken der Spurprädikate (zweites Konjunktionsglied von Definition 4.5),
- Übergang zu einer einelementigen Menge von Spurspezifikationen (Definition von $[\![\_]\!]_{tra}$ im Anschluß an Anmerkung 4.3),
- Aufspaltung eines Elements aus einer Menge von Spurspezifikationen (Definition von $[\![\_]\!]_{tra}$ im Anschluß an Anmerkung 4.3) und
- modulare Verfeinerung eines Elements aus einer Menge von Spurspezifikationen (Satz 4.2).

Die Verfeinerung einer Spurspezifikation durch Verfeinern der Systemstruktur und gegebenenfalls durch weitere Aufspaltung innerhalb einer Menge von Spezifikatio-

nen kann im weiteren Entwicklungsprozeß nicht mehr rückgängig gemacht werden. Sie ist dann angebracht, wenn klar ist, daß die verfeinerte Struktur auch in dieser Weise implementiert werden soll, zum Beispiel, wenn sie aus Sicherheitsgründen auf verschiedene Baugruppen verteilt werden soll. Ein anderer Grund für eine Aufspaltung kann sein, daß einzelne Komponenten in unterschiedlichen Formalismen weiterentwickelt werden sollen, beispielsweise teils in einer auf Aktionsspuren basierenden Hardware-Beschreibungssprache und teils in einem funktionalen Entwurfsformalismus. Die Verwendung unterschiedlicher Formalismen ist auch sinnvoll, um bestimmte Problemfälle zu vermeiden, etwa die Behandlung des nicht-strikten fairen Mischens in einem funktionalen Formalismus.

Für einige Komponenten ist mit der lokalen Anforderungsspezifikation der Entwicklungsprozeß beendet. Das ist zum Beispiel dann der Fall, wenn die Komponente direkt von der Spurspezifikation aus implementiert wird, insbesondere bei Hardware-Elementen. Ebenso kann es sein, daß einzelne Komponenten überhaupt nicht implementiert werden sollen, weil ihre Funktionen zum Beispiel durch menschliche Interaktion ausgeführt werden.

Für diejenigen Komponenten, die weiter entwickelt werden sollen, erfolgt als nächster Schritt der Entwurfsmethodik die Umsetzung der einzelnen Spurspezifikationen in funktionale Spezifikationen. Dieser Umsetzungsschritt ist der durch die Verwendung zweier Formalismen verursachte zusätzliche Arbeitsaufwand, den man mittels schematischer Umsetzungsregeln (wie den in den Sätzen 4.3 und 4.4 angegebenen) zu verringern sucht. Das Ergebnis des Ebenenübergangs entspricht Nummer 3 in Bild 4.1.

Wenn eine Systementwicklung nur im funktionalen Formalismus durchgeführt werden soll, dann ist hier der Einstiegspunkt für eine globale Anforderungsspezifikation, die genau der oben geschilderten globalen Spur-Anforderungsspezifikation entspricht, nur daß der funktionale Formalismus verwendet wird.

Auch auf der funktionalen Ebene finden wieder Verfeinerungsschritte statt:

- Verfeinern der Systemstruktur (erstes Konjunktionsglied von Definition 4.10),
- Verstärken der funktionalen Prädikate (zweites Konjunktionsglied von Definition 4.10),
- Übergang zu einer einelementigen Menge von Spezifikationen (Definition von $[\![\_]\!]_{fun}$ im Anschluß an Anmerkung 4.4),
- Aufspaltung eines Elements aus einer Menge von Spurspezifikationen (Definition von $[\![\_]\!]_{fun}$ im Anschluß an Anmerkung 4.4) und
- modulare Verfeinerung eines Elements aus einer Menge von funktionalen Spezifikationen (Satz 4.6)

Wie schon auf der Spurebene erfolgt das Verstärken der funktionalen Prädikate prinzipiell nach den allgemeinen Regeln der Prädikatenlogik. Während aber bei der An-

forderungsspezifikation auf der Spurebene die dadurch erreichte maximale Flexibilität erwünscht ist, ist es für die Agentenentwicklung auf der funktionalen Ebene eher sinnvoll, spezielle Spezifikationsstile bereitzustellen, die eine systematische Entwicklung der Spezifikation hin zur angestrebten Implementierungssprache unterstützen. Ein solcher Spezifikationsstil, der auf dem Konzept von Zustandsmaschinen basiert, wird im folgenden Kapitel 5 dargestellt.

Das Ziel der Verfeinerungsschritte auf der funktionalen Ebene ist eine implementierbare Spezifikation. Wir legen diesen Begriff hier nicht formal fest. Intuitiv stellen wir uns unter einer implementierbaren Spezifikation eine Menge von funktionalen Spezifikationen vor, deren Systemstrukturen jeweils nur eine Komponente enthalten und aus deren Agentenspezifikationen sich durch einfache syntaktische Transformationen Programme in der Implementierungssprache gewinnen lassen.

Die Ebene der Implementierung ist in Nummer 4 von Bild 4.1 dargestellt. Für die verwendeten Implementierungssprachen soll die Verfeinerungsrelation $\rightsquigarrow$ von einer Spurspezifikation oder funktionalen Spezifikation hin zu Programmen in diesen Sprachen definiert sein. Außerdem soll die durch die Betriebssystemumgebung realisierte Kommunikation zwischen Prozessen, die Agenten implementieren, mit dem in der Modellierung verwendeten asynchronen Ansatz verträglich sein. Eine Erweiterung der Entwurfsmethodik auf Systeme mit synchroner Kommunikation wird in [Sch95] vorgestellt.

Als Implementierungssprachen kommen zum Beispiel die Sprachen *AL* und *PL* in Frage. Diese Sprachen sind in [Ded92] definiert und ihre Semantiken als Mengen von stromverarbeitenden Funktionen angegeben. Wegen ihrer Nähe zu funktionalen Spezifikationen sind auch funktionale Programmiersprachen wie *Haskell* oder *ML* geeignet. Insbesondere für funktionale Spezifikationen, die in einem zustandsorientierten Spezifikationsstil wie dem von Kapitel 5 erstellt sind, bietet sich eine Implementierung in imperativen Sprachen mit entsprechenden Kommunikationsprimitiven an. Diese Möglichkeit ist in Abschnitt 5.6 genauer beschrieben.

Auch innerhalb der Ebene der Implementierung sind Verfeinerungsschritte möglich, zum Beispiel zur Effizienzsteigerung. Einige Transformationsregeln für den Übergang von *AL* zu *PL* sind in [Ded92] angegeben.

Kapitel 5

Methodik des Agentenentwurfs

Während wir in Kapitel 4 einen Rahmen für den gesamten Entwurfsprozeß angegeben haben, beschäftigen wir uns nun genauer mit der Entwicklungsmethodik auf der Ebene des funktionalen Spezifikationsformalismus. Eine Entwicklung nach der hier vorgeschlagenen Methodik besteht aus einer Folge von funktionalen Spezifikationen, die sich auf je ein Prädikatenschema abstützen. Die einzelnen Schemata sind aufeinander abgestimmt. Intuitiv beschreiben sie unterschiedliche Ausprägungen von Zustandsmaschinen (Automaten).

Automatenorientierte Formalismen zur Beschreibung reaktiver und verteilter Systeme sind schon in großer Zahl vorgeschlagen worden. Transitionssysteme über ungegliederte Aktionenmengen (siehe [Web92, Abschnitt 3.4] mit weiteren Nachweisen) können zur Spezifikation im Spurformalismus verwendet werden. Die bekannten Prozeßalgebren wie CSP [Hoa85], CCS [Mil80] oder ACP [BK84] sind ebenfalls als Transitionssysteme interpretierbar, wenn man Prozeßterme als Repräsentanten für den Zustand betrachtet. Der automatenorientierte *StateChart*-Formalismus [Har87] ist insbesondere im Hinblick auf die praktischen Modularisierungsanforderungen beim Systementwurf weiterentwickelt.

Mehr in Richtung unseres Agentenbegriffs gehen Ansätze, bei denen zwischen Ein- und Ausgabeaktionen unterschieden wird, wobei in jedem Zustand alle Eingabeaktionen möglich (*enabled*) sein müssen. Einen solchen Ansatz stellen zum Beispiel *I/O-Automaten* [Jon89] dar. Nahe am klassischen Begriff von Automaten mit Ausgabe ist die bei der Protokollspezifikation verwendete Sprache *Estelle* [Est88] angesiedelt, wobei hier der Zustandsraum in einen endlichen Programmablaufszustand und einen Speicher gegliedert ist. Auch Prozesse der Sprache *SDL* [SDL89] lassen sich als Zustandsmaschinen interpretieren, die eine Warteschlange für ankommende Nachrichten aufweisen.

Der hier vorgestellte Ansatz kommt den bekannten endlichen Automaten mit Ausgabe (Mealy-/Moore-Automaten) nahe. Die Ausgabe wird schrittweise erzeugt, wobei jede

- Allgemeine funktionale Spezifikation (Abschnitt 4.3)

  Mathematische Umformungen, um die geforderten Eigenschaften in explizite Bedingungen für endliche Eingaben umzuwandeln

- Eingeschränkte funktionale Spezifikation (Abschnitt 5.1)

  Umformung, syntaktische Transformation

- Spezifikation nach dem Schema *Strat* mit implizitem Zustandsraum (Abschnitt 5.2)

  Syntaktische Umformung

- Spezifikation nach dem Schema *RStrat* mit implizitem Zustandsraum (Abschnitt 5.3)

  Entwicklung eines expliziten Zustandsraumes,
  Beweis der Verfeinerungseigenschaft

- Spezifikation nach dem Schema *EStrat* mit explizitem Zustandsraum (Abschnitt 5.4)

  Verfeinerung des expliziten Zustandsraumes,
  Beweis der Verfeinerungseigenschaft

- Spezifikation nach dem Schema *EStrat* mit verfeinertem expliziten Zustandsraum (Abschnitt 5.5)

  Transformation in geeignete Programmiersprache

- Lauffähiges Programm (Abschnitt 5.6)

Bild 5.1: Überblick über die Verfeinerungsschritte mit zustandsorientierten Techniken

Eingabe in Abhängigkeit vom bisherigen Zustand die Ausgabe fortsetzt und einen Zustandsübergang bewirkt. Anders als die bekannten Formalismen sehen wir Automaten jedoch nicht als eigenständiges Konzept, sondern als Beschreibungsmittel für stromverarbeitende Funktionen. Daher lassen wir beliebige (auch unendliche) Zustandsräume und mehrere Ein- und Ausgabekanäle zu. Die durch Automaten beschriebenen Funktionen müssen die Monotonie- und Stetigkeitsforderung für Agenten erfüllen.

Als wesentliche Neuerung wird innerhalb des zustandsorientierten Formalismus eine Entwurfsmethodik angegeben, die über mehrere vorgegebene Spezifikationsschemata durch schrittweise Verfeinerung zur Implementierung führt. Das Ziel ist, über theoretische Erkenntnisse hinaus eine praktisch verwendbare Vorgehensweise zu liefern. Eine Fallstudie, in der unter anderem auch zustandsorientierte Techniken für eine Systementwicklung verwendet werden, findet sich in [DW92a] beziehungsweise [DW92b]. Die dort angedeuteten Lösungen sind hier näher untersucht, erweitert und methodisch ausgebaut. Ein ausführliches Beispiel für zustandsorientierte Spezifikation im Zusammenhang mit stromverarbeitenden Funktionen, bei dem aber keine Entwicklungsschritte stattfinden, ist in [Den91] enthalten.

Bild 5.1 gibt einen Überblick über den in diesem Kapitel dargestellten methodischen Entwicklungsvorgang. Innerhalb der FOCUS-Methodik ist die zustandsorientierte Spezifikation in der Ebene der funktionalen Spezifikation und im Übergangsbereich zur Implementierung angesiedelt.

In diesem Zusammenhang stellt sich die Frage nach den Vor- und Nachteilen von zustandsorientierten gegenüber anderen Spezifikationstechniken. Verglichen mit eigenschaftsorientierten Spezifikationen sind zustandsorientierte in der Regel weniger abstrakt, da sie die schrittweise Berechnung des Ergebnisses auf einer (wenn auch nur gedachten) Maschine widerspiegeln. Diese geringere Abstraktheit wird oft als Nachteil angesehen. Da aber die FOCUS-Methodik das ganze Spektrum von abstrakter Anforderungsbeschreibung bis zu Programmen umfaßt, ist es notwendig, auch für die implementierungsnäheren Ebenen einen geeigneten Spezifikationsstil zur Verfügung zu stellen. Dafür sind zustandsorientierte Spezifikationen geeignet.

Zustandsorientierte Spezifikationen sind relativ leicht zu lesen und zu verstehen. Wenn die Bedeutung der Zustandskomponenten intuitiv klar ist, kann man die Wirkung der einzelnen Eingaben unabhängig voneinander erfassen, während bei eigenschaftsorientierten Spezifikationen komplexe Prädikate "in einem Stück" verstanden werden müssen. Wegen ihrer intuitiven Eingängigkeit können zustandsorientierte Spezifikationen sogar ein geeigneter *Einstiegs*punkt für Systementwicklungen sein, bei denen die Agentenstruktur und die genauen Aufgaben der einzelnen Agenten schon von Anfang an feststehen.

In der FOCUS-Methodik werden zustandsorientierte Spezifikationen für die implementierungsnahen Ebenen des Entwurfsprozesses eingesetzt, weil sie dafür besonders geeig-

net sind. Für die abstrakteren Ebenen dienen, wie in Kapitel 4 beschrieben, die Formalismen der Spurspezifikation und der allgemeinen funktionalen Spezifikation. Die Idee, mehrere Formalismen zu kombinieren und die einzelnen Entwicklungsschritte jeweils im für die Problemstellung günstigsten Formalismus auszuführen, ist ein generelles Kennzeichen von FOCUS. Natürlich müssen die verwendeten Formalismen so aufeinander abgestimmt sein, daß durch den Übergang nicht zuviel Mehraufwand entsteht.

Im Ergebnis können durch die Kombination mehrerer Formalismen die Stärken jedes einzelnen Formalismus genutzt und dessen Schwächen durch Verlagerung auf andere Ebenen vermieden werden. Zum Beispiel ist die Beschreibung von komplexen Eigenschaften für unendliche Eingaben in einem rein zustandsorientierten Formalismus schwierig. Im Spurformalismus und auf der Ebene der funktionalen Spezifikation mit beliebigen Prädikaten bestehen dagegen keine methodischen oder spezifikationstechnischen Unterschiede zwischen Eigenschaften für endliche und für unendliche Eingaben. Daher schlagen wir vor, Eigenschaften für unendliche Eingaben schon auf den genannten abstrakteren Ebenen durch prädikatenlogische Umformungen so umzuwandeln, daß die weitere Entwicklung dann im zustandsorientierten Formalismus problemlos erfolgen kann.

In den folgenden Abschnitten gehen wir von einem Automatenbegriff aus, bei dem eine Vergrößerung der Ein- und Ausgabeströme im Sinne der Approximationsordnung auch eine Verlängerung im Sinne der Präfixordnung darstellt. Wir setzen also in diesem Kapitel und bei den Automatenbegriffen *TStrat* und *NStrat* der Kapitel 6 und 7 durch Automaten zu modellierende Agenten erster Ordnung voraus.

5.1 Eingeschränkte funktionale Spezifikationen

Wie in Bild 5.1 gezeigt, geht die Entwicklung auf der funktionalen Ebene von einer allgemeinen funktionalen Spezifikation (S, F) aus, deren Agentenspezifikation F eine Agentenmenge durch eine beliebige prädikatenlogische Formel beschreibt. Dabei ist es möglich, die für eine Verlängerung einer endlichen Eingabe erfolgende zusätzliche Ausgabe sehr indirekt zu charakterisieren. Wegen ihrer Stetigkeit nähern Agenten nämlich ein für unendliche Eingaben festgelegtes Ergebnis in endlichen Berechnungen schrittweise an. Somit wird durch Eigenschaften, die nur für unendliche Eingaben gefordert werden, indirekt auch das endliche Verhalten eines Agenten festgelegt.

Wir unterscheiden also bei der Spezifikation eines Agenten zwischen Eigenschaften, die unmittelbar für endliche Eingaben gefordert werden, und Eigenschaften für unendliche Eingaben, die sich nur indirekt über die Stetigkeit des Agenten auf endliche Berechnungen auswirken. Diese Charakterisierung bezieht sich, im Unterschied zu den in Abschnitt 3.7 behandelten Begriffen Sicherheit und Lebendigkeit, auf die *Eingaben*

eines Agenten, während die Begriffe Sicherheit und Lebendigkeit nach Definition 3.6 die *Ausgaben* betreffen.

In einem automatenorientierten Formalismus wird die schrittweise Berechnung des Ergebnisses modelliert. Jeder Berechnungsschritt geht von einer endlichen bisherigen Eingabe aus. Die funktionale Spezifikation, die den Ausgangspunkt der in diesem Kapitel beschriebenen Systementwicklung darstellt, enthält aber im allgemeinen Eigenschaften für unendliche Eingaben. Solche Eigenschaften werden in der ersten in Bild 5.1 gezeigten Entwicklungsstufe durch geeignete prädikatenlogische Umformungen in direkte Beschreibungen für endliche Eingaben umgewandelt.

Ein typischer Entwicklungsschritt auf dieser Stufe ist es, eine Eigenschaft für unendliche Eingaben durch die Angabe einer endlichen Schranke zu ersetzen, deren Über- oder Unterschreitung schon für jede endliche Teilberechnung festgestellt werden kann. Betrachten wir beispielsweise eine Spezifikation $F \subseteq IS \nrightarrow OS$ eines Agenten, der für jede unendlich lange Eingabe auch eine unendlich lange Ausgabe liefern soll:

$$F.f = \forall\, x \in IS : \#x = \infty \Rightarrow \#f.x = \infty$$

Als sinnvolle Konkretisierung kann beispielsweise gefordert werden, daß die Länge der Ausgabe stets mindestens der Länge der (endlichen) Eingabe entsprechen soll, wie in der folgenden Spezifikation F' angegeben. Dies stellt eine echte Entwurfsentscheidung dar, weil die Implementierungsmöglichkeiten eingeschränkt werden:

$$F'.f = \forall\, x \in IS^{fin} : \#x \leq \#f.x$$

Um sich nicht unnötig früh auf Details der Implementierung festlegen zu müssen, kann es sinnvoll sein, eine Menge von Werten anzugeben, aus denen die Schranke ausgewählt wird. Gegeben sei zum Beispiel die Spezifikation $G \subseteq IS \nrightarrow OS$ eines Agenten, der beliebige, aber nicht unendlich lange Ausgaben liefert:

$$G.g = \forall\, x \in IS : \#g.x < \infty$$

Als mögliche Konkretisierung dieser Eigenschaft kann gefordert werden, daß eine beliebig große, aber endliche Schranke für die Länge der Ausgabe existiert:

$$G'.g = \exists\, n \in \mathbb{N} : \forall\, x \in IS^{fin} : \#g.x \leq n$$

Obwohl hier die Menge der Implementierungen nicht eingeschränkt wurde, stellt diese Verfeinerung einen zielgerichteten Entwicklungsschritt dar, denn für jede anfängliche Auswahl einer Maximallänge n kann eine Überschreitung dieser Länge schon während des Berechnungsablaufs festgestellt werden.

Die gerade angegebene Agentenspezifikation G' ist ein Beispiel für die allgemeine Technik, nichtdeterministische Entscheidungen in sogenannte *Orakel* zu codieren. Für jede Auswahl des Orakels ergibt sich ein Agent, der die funktionale Spezifikation erfüllt.

Sobald der Wert des Orakels gewählt ist, stehen intern alle Entscheidungen fest, auch solche, die extern erst im Verlauf der Berechnung beobachtet werden können. Dadurch werden Eigenschaften für unendliche Eingaben umgewandelt in Eigenschaften, deren Verletzung auch für endliche Eingaben festgestellt werden kann, wenn der ausgewählte Wert des Orakels bekannt ist. Agentenspezifikationen, die Orakel verwenden, eignen sich besonders gut zur Umsetzung in den zustandsorientierten Formalismus, wie in Satz 5.3 gezeigt werden wird.

Gegeben sei eine Agentenspezifikation $F \subseteq IS \nrightarrow OS$, die mittels eines Orakels zu einer eingeschränkten funktionalen Spezifikation verfeinert werden soll. Dazu werden eine nicht-leere Menge *Oracle* sowie ein Prädikat $G \in Oracle \to IS \to OS \to \mathbb{B}$ angegeben, so daß das im folgenden definierte Prädikat $F' \subseteq IS \nrightarrow OS$ eine Verfeinerung von F ist, also $F \Leftarrow F'$ gilt:

$$F'.f \;=\; \exists\, z \in Oracle : \forall\, x \in IS^{fin} : G.z.x.(f.x)$$

Diese Form der eingeschränkten funktionalen Spezifikation eines Agenten mittels eines Orakels wird im folgenden Beispiel 5.1 veranschaulicht:

Beispiel 5.1: Bei dem in Beispiel 4.6 durch das Prädikat *FeederFun* spezifizierten Agent *Feeder* ist die durch die lokale Variable n festgelegte Zahl von Wiederholungen eines Rechenauftrags an den Kanal $FtoT$ beliebig groß, aber endlich. Um diese Eigenschaft implementierungsnäher zu formulieren, wird für jeden Bezeichner i eine endliche Schranke $z.i$ für die zulässige Anzahl der Wiederholungen eingeführt. Die Agentenspezifikation *FeederFun'* nach dem oben angegebenen Schema ist eine Verfeinerung von *FeederFun*:

$$FeederFun'.f \;=\; \exists\, z \in Id \to \mathbb{N} : \forall\, x \in (InType_{PSysStruct'}.Feeder)^{fin} : G.z.x.(f.x)$$

$$\begin{aligned}
&G.z.x.s \;=\; \forall\, i \in Id :\\
&\quad (Cmd_i \copyright x.MtoF = \langle\rangle \;\Rightarrow\; (Cmd_i \copyright s.FtoT = \langle\rangle \wedge Rply_i \copyright s.SRply = \langle\rangle)) \wedge\\
&\quad (\#Cmd_i \copyright x.MtoF = 1 \;\Rightarrow\\
&\qquad (\exists\, n \in \mathbb{N} : n \le z.i \wedge \#\{ft.(Cmd_i \copyright x.MtoF)\} \copyright s.FtoT = n+1 \wedge\\
&\qquad\quad (\#Rply_i \copyright x.MtoF > n \;\Rightarrow\\
&\qquad\qquad (\exists\, r \in Rply_i : Rply_i \copyright s.SRply = \langle r\rangle \wedge \#\{r\} \copyright x.MtoF \ge 1))))
\end{aligned}$$ □

5.2 Implizite Zustände — Schema *Strat*

In diesem Abschnitt soll eine grundlegende Form von zustandsorientierten Spezifikationen betrachtet werden, nämlich eine, bei der die Zustände genau die Sequenzen der bisherigen Eingaben sind. Wir nennen einen solchen Zustandsraum *implizit,* weil er schon durch den Kanaltyp der Eingabe eines Agenten kanonisch festgelegt ist, also sich implizit aus diesem Kanaltyp ergibt. Es müssen noch keine Entwurfsentscheidungen

über einen für die spätere Implementierung zu konstruierenden expliziten Zustandsraum getroffen werden. Daher kommen zustandsorientierte Spezifikationen mit implizitem Zustandsraum den eingeschränkten funktionalen Spezifikationen besonders nahe. Sie sind in der hier angegebenen Entwurfsmethodik (siehe Bild 5.1) der erste Schritt im zustandsorientierten Formalismus.

Es stellt sich die Frage, welche Form der implizite Zustandsraum bei Agenten mit mehreren Eingabekanälen haben soll. Die beiden grundlegenden Möglichkeiten sind, als Zustand entweder den aktuellen Kanalzustand oder, wie schon auf der Spurebene, die Sequenz der bisherigen Eingabe*aktionen* zu verwenden, also die total geordneten Eingaben auf allen Kanälen. Die erstgenannte Möglichkeit kommt unserem Agentenbegriff näher, denn auch ein Agent ist eine Funktion auf Kanalzuständen und nicht auf Aktionssequenzen. Wie in Satz 5.1 gezeigt werden wird, ist der erste Zustandsbegriff hinreichend ausdrucksstark. Wir beschäftigen uns daher in diesem Kapitel mit diesem Zustandsbegriff.

Eine Spezifikation nach dem in der folgenden Definition 5.1 angegebenen Schema $Strat_H$ beschreibt einen Automaten mit der Menge IS der Eingabekanalzustände (Eingabekanalbelegungen) als Zustandsraum. Jede deterministische Verhaltensweise des Automaten wird durch eine Strategie $h \in H$ festgelegt. Zu einem Zustand s und einer neuen Eingabeaktion a (also einer aktuellen Eingabekanalbelegung $s \circ \bar{a}$, wobei $\bar{a}$ die der Aktion a entsprechende Kanalbelegung ist) gibt die Strategie h die Kanalbelegung $h.s.a$ an, um die sich die Ausgabekanalbelegung des Agenten verlängert:

Definition 5.1 (Spezifikationsschema mit implizitem Zustand): Gegeben seien Kanaltypen IS und OS. Die Menge $STRAT_{IS,OS}$ von *Strategien* ist definiert vermöge:

$$STRAT_{IS,OS} = IS \to Act.IS \to OS$$

Eine Spezifikation mit implizitem Zustand ist ein Prädikat nach dem Schema $Strat_H$ für ein Hilfsprädikat $H \subseteq STRAT_{IS,OS}$:

$$\begin{aligned} Strat_H &\subseteq IS \twoheadrightarrow OS \\ Strat_H.f &= \exists\, h \in H : \forall\, s \in IS^{fin},\, a \in Act.IS : \\ &\qquad f.\varepsilon = \varepsilon \;\wedge\; f.(s \circ \bar{a}) = f.s \circ h.s.a \end{aligned}$$ □

Beispiel 5.2: Die in Beispiel 5.1 angegebene Agentenspezifikation $FeederFun'$ kann durch die zustandsorientierte Spezifikation $FeederFun'' = Strat_H$ verfeinert werden. Die Strategienmenge H ist im folgenden definiert. Jedes h_z ist durch die in Bild 5.2 gezeigte Tabellennotation (die Schreibweise ist in Abschnitt 2.1 erläutert) gegeben:

$$\begin{aligned} H &\subseteq STRAT_{InType.PSysStruct'.Feeder,\, OutType.PSysStruct'.Feeder} \\ H &= \{h_z : z \in Id \to \mathbb{N}\} \end{aligned}$$

Neben der Umwandlung in eine Spezifikation nach dem Schema *Strat* ist hierbei auch der Verfeinerungsschritt vorgenommen worden, daß ein Kommando nicht beliebig oft

bis maximal zur Obergrenze $z.i$ an die Prozessoren gesendet wird, sondern genau so oft, bis entweder die Obergrenze $z.i$ erreicht oder eine erfolgreiche Rückantwort eingegangen ist. Das Hilfsprädikat $pending_{i,(z.i)}.s$ drückt aus, daß im Zustand s ein Rechenauftrag mit Identifikator i noch zu bearbeiten ist.

Die Verfeinerungseigenschaft der Spezifikation *FeederFun''* gegenüber *FeederFun'* folgt aus dem in diesem Abschnitt weiter unten angegebenen Satz 5.3. □

Eine Zustandsmaschine nach dem Schema *Strat* ähnelt dem Mealy-Automaten, weil Zustand und letzte Eingabe den Folgezustand und die Ausgabe bestimmen. Die dem Moore-Automaten entsprechende Alternative, eine Strategie der Form $H \subseteq IS \to OS$ zu verwenden und in der Definition von $Strat_H$ den Ausdruck $h.s.a$ durch $h.(s \circ \bar{a})$ zu ersetzen, ist dagegen nicht allgemein verwendbar. Zum Beispiel ließe sich damit ein Agent nicht ausdrücken, der auf zwei Eingängen eintreffende Nachrichten streng abwechselnd (deterministisch) auf einen Ausgang weiterleitet.

Wie schon in Abschnitt 5.1 erläutert, enthält das Schema *Strat* nur für endliche Eingaben explizite Bedingungen für die Ausgabe. Das Verhalten eines durch *Strat* definierten Agenten für unendliche Eingaben folgt aus der immer geforderten Stetigkeit von Agenten. Dies erleichtert die Spezifikation von Strategien, denn es muß nicht auf die Stetigkeitsforderung Rücksicht genommen werden.

Jeder durch das Schema *Strat* beschriebene Agent reagiert auf leere Eingaben mit einer leeren Ausgabe. Dies ist eine vereinfachende Einschränkung, die wir in diesem Kapitel treffen, um den Formalismus nicht unnötig zu überladen. Sonst müßte die erste, nicht von außen angestoßene Ausgabe getrennt angegeben werden. Wenn man von dieser Einschränkung absieht und die Stetigkeit von Agenten berücksichtigt, ist das Schema *Strat* für Agentenspezifikationen universell verwendbar, wie der folgende Satz 5.1 zeigt:

Satz 5.1: Zu jeder Agentenspezifikation $F \subseteq IS \twoheadrightarrow OS$ mit $f.\varepsilon = \varepsilon$ für alle $f \in F$ existiert ein $H \subseteq STRAT_{IS,OS}$ mit $F = Strat_H$.

Beweis: Es genügt zu zeigen, daß zu jedem $f \in IS \twoheadrightarrow OS$ mit $f.\varepsilon = \varepsilon$ ein $h \in STRAT_{IS,OS}$ existiert, für das die folgende Bedingung erfüllt ist:

$$\forall\, s \in IS^{fin},\ a \in Act.IS : f.(s \circ \bar{a}) = f.s \circ h.s.a$$

Zu gegebenem f legt diese Bedingung für jedes $s \in IS^{fin}$, $a \in Act.IS$ genau einen Wert $h.s.a$ fest, denn für Nachrichten erster Ordnung gilt wegen der Monotonie von f, daß $f.s \sqsubseteq f.(s \circ \bar{a})$, also $f.(s \circ \bar{a}) = f.s \circ z$ für genau ein $z \in OS$, also $h.s.a = z$. □

Die obige Definition 5.1 ist also hinreichend mächtig. Dagegen gilt die Umkehrung von Satz 5.1 (also, daß es für jede Strategie einen entsprechenden Agenten gibt) nicht, denn eine Spezifikation nach dem Schema *Strat* kann unerfüllbar sein, wenn für jede Strategie $h \in H$ die Reihenfolge des Eintreffens von Nachrichten auf mehreren Eingabekanälen

einen Unterschied bezüglich der Ausgaben machen würde, wenn also zum Beispiel für zwei Aktionen a und b auf verschiedenen Kanälen gilt: $h.\varepsilon.a \circ h.\bar{a}.b \neq h.\varepsilon.b \circ h.\bar{b}.a$. Für zeitbehaftete Agenten nach dem in Kapitel 6 in Definition 6.3 angegebenen Schema *TStrat* gilt diese Einschränkung nicht.

Im Rahmen der Systementwicklung soll natürlich sichergestellt werden, daß eine Spezifikation erfüllbar ist. Eine Beweisbedingung dafür gibt der folgende Satz 5.2:

Satz 5.2: Gegeben sei eine Strategienmenge $H \subseteq STRAT_{IS,OS}$. Eine hinreichende Bedingung für $Strat_H \neq \varnothing$ ist es, wenn gilt:

$$\exists\, h \in H : \forall\, s \in IS^{fin},\ a, b \in Act.IS\ :\\ s \circ \bar{a} \circ \bar{b} = s \circ \bar{b} \circ \bar{a}\ \Rightarrow\ h.s.a \circ h.(s \circ \bar{a}).b = h.s.b \circ h.(s \circ \bar{b}).a$$

Beweis: Gegeben sei ein $h \in H$, das die im Satz angegebene Eigenschaft hat. Zu zeigen ist $Strat_{\{h\}} \subseteq Strat_H \neq \varnothing$. Dazu betrachten wir die Funktion g:

$$\begin{array}{lcl} g & \in & (Act.IS)^\omega \twoheadrightarrow OS \\ g.\langle\rangle & = & \varepsilon \\ g.(t \circ \langle a \rangle) & = & g.t \circ h.(IS \textcircled{s} t).a \end{array}$$

Durch Induktion über die Länge von t zeigen wir im folgenden, daß für jedes $s \in IS^{fin}$ die Menge $G.s = \{g.t : t \in (Act.IS)^\omega \wedge IS \textcircled{s} t = s\}$ einelementig ist, so daß die durch $\{f.s\} = G.s$ für $s \in IS^{fin}$ gegebene Funktion $f \in IS \twoheadrightarrow OS$ wohldefiniert ist; das Verhalten von f für Eingaben aus $IS \backslash IS^{fin}$ folgt aus der Stetigkeit. Offensichtlich erfüllt dieses f die Bedingungen des Schemas $Strat_{\{h\}}$, womit $Strat_H \neq \varnothing$ gezeigt ist.

Für $\#t \leq 1$ ist die Behauptung trivialerweise erfüllt. Für den Schritt von $\#t = n$ auf $n+1$ seien beliebige $s \in IS^{fin}$, $t, t' \in (Act.IS)^\omega$ und $a, b \in Act.IS$ mit $\#t = \#t' = n$ und $IS \textcircled{s} (t \circ \langle a \rangle) = IS \textcircled{s} (t' \circ \langle b \rangle) = s$ (1) gegeben. Zu zeigen ist $g.(t \circ \langle a \rangle) = g.(t' \circ \langle b \rangle)$. Für $a = b$ ist dies nach der Induktionsannahme und der Definition von g klar. Der Fall $a \neq b \wedge fst.a = fst.b$ ist nach (1) nicht möglich, also nehmen wir $fst.a \neq fst.b$ an. Daher und nach (1) lassen sich t und t' aufspalten in $t = u \circ \langle b \rangle \circ v$ und $t' = u' \circ \langle a \rangle \circ v'$ mit $IS \textcircled{s} (u \circ v) = IS \textcircled{s} (u' \circ v')$ (2), wobei in v keine Aktion des Kanals $fst.b$ auftaucht und in v' keine Aktion des Kanals $fst.a$. Daher gilt auch $IS \textcircled{s} (u \circ \langle b \rangle \circ v) = IS \textcircled{s} (u \circ v \circ \langle b \rangle)$ (3) und dementsprechend $IS \textcircled{s} (u' \circ \langle a \rangle \circ v') = IS \textcircled{s} (u' \circ v' \circ \langle a \rangle)$ (4).

Insgesamt erhalten wir:

$$\begin{array}{lcl} g.(t \circ \langle a \rangle) & = & g.(u \circ \langle b \rangle \circ v) \circ h.(IS \textcircled{s} (u \circ \langle b \rangle \circ v)).a \\ & & \qquad \text{[Zerlegung von } t\text{; Definition von } g\text{]} \\ & = & g.(u \circ v \circ \langle b \rangle) \circ h.(IS \textcircled{s} (u \circ v \circ \langle b \rangle)).a \\ & & \qquad \text{[(3) und Induktionsannahme]} \\ & = & g.(u \circ v) \circ h.(IS \textcircled{s} (u \circ v)).b \circ h.(IS \textcircled{s} (u \circ v \circ \langle b \rangle)).a \\ & & \qquad \text{[Definition von } g\text{]} \end{array}$$

$$
\begin{aligned}
&= g.(u \circ v) \circ h.(IS\textcircled{s}(u \circ v)).a \circ h.(IS\textcircled{s}(u \circ v \circ \langle a \rangle)).b \\
&\qquad\qquad [\text{Voraussetzung von Satz 5.2}] \\
&= g.(u' \circ v') \circ h.(IS\textcircled{s}(u' \circ v')).a \circ h.(IS\textcircled{s}(u' \circ v' \circ \langle a \rangle)).b \\
&\qquad\qquad [(2) \text{ und Induktionsannahme}] \\
&= g.(u' \circ v' \circ \langle a \rangle) \circ h.(IS\textcircled{s}(u' \circ v' \circ \langle a \rangle)).b \\
&\qquad\qquad [\text{Definition von } g] \\
&= g.(u' \circ \langle a \rangle \circ v') \circ h.(IS\textcircled{s}(u' \circ \langle a \rangle \circ v')).b \\
&\qquad\qquad [(4) \text{ und Induktionsannahme}] \\
&= g.(t' \circ \langle b \rangle) \qquad [\text{Zusammensetzen von } t\text{; Definition von } g] \qquad \square
\end{aligned}
$$

Die Bedingung von Satz 5.2 ist für diejenigen Fälle stärker als notwendig, für die durch die bisherigen Eingaben schon eine unendliche Ausgabe erzeugt wurde.

Automaten nach dem Schema *Strat* sind im allgemeinen nichtdeterministisch. Zur Definition eines nichtdeterministischen Automaten gibt es die zwei grundsätzlichen Möglichkeiten, entweder eine Zustandsübergangs*relation* zu verwenden, oder, wie hier, eine Menge von Zustandsübergangs*funktionen.*

Betrachten wir zunächst die erste Möglichkeit. Die Verwendung einer Zustandsübergangsrelation (oder einer äquivalenten mengenwertigen Funktion) ist in der Automatentheorie üblich. Die operationelle Vorstellung ist, daß während des Berechnungsvorganges in jedem Schritt eine Auswahl unter den zulässigen Transitionen stattfindet. Dabei ergibt sich aber das Problem, daß bei der Vereinigung von Berechnungsabläufen Verwischungseffekte auftreten. So würde man zum Beispiel den in Abschnitt 5.1 angegebenen Agenten G, der für jede Eingabe eine beliebige, aber endlich lange Ausgabe liefert, durch eine Zustandsübergangsrelation beschreiben, bei der in jedem Berechnungsschritt zumindest die Möglichkeit einer weiteren Ausgabe besteht. Ein solcher Agent könnte dann aber auch unendliche Ausgaben erzeugen.

Daher ist es bei Automaten, die Zustandsübergangsrelationen verwenden, erforderlich, zusätzliche Eigenschaften der nichtdeterministischen Auswahl der Zustandsübergänge anzugeben. In gebräuchlichen Formalismen finden sich zu diesem Zweck vorgegebene Akzeptanzbedingungen (Büchi-Automaten), fest vorgegebene Fairnesseigenschaften (*UNITY*), Fairnesseigenschaften, die über Fairnessmengen in eingeschränkter Weise spezifiziert werden können (I/O-Automaten), oder die getrennte Spezifikation von Lebendigkeitseigenschaften mit temporaler Logik (Transitions-Axiom-Methode).

Bei Automaten, die mit einer Menge von Zustandsübergangsfunktionen (einem Prädikat über Strategien) spezifiziert sind, tritt dieses Problem nicht auf. Alle nichtdeterministischen Entscheidungen während der Berechnung werden auf die Auswahl einer Strategie konzentriert. Jede Strategie beschreibt einen (deterministischen) Agenten, dessen Verhalten alle geforderten Eigenschaften aufweist. Im obigen Beispiel würde jede Strategie etwa einem Agenten mit einer fest begrenzten Ausgabelänge entsprechen. Ein solcher Automatenbegriff steht in Einklang mit der in FOCUS generell eingesetzten

Technik, Agenten durch ein Prädikat über Funktionen (und nicht durch eine Relation) zu spezifizieren. Dies hat insbesondere auch beweistechnische Vorteile.

Insgesamt halten wir den hier gewählten Automatenbegriff für vorteilhaft, weil Zusatzbedingungen der oben genannten Art nicht benötigt werden. Zu dem eventuellen Einwand, daß eine anfängliche Auswahl aus einer möglicherweise unendlichen Strategienmenge nicht implementierbar sei, ist zu sagen, daß dies auch nicht erforderlich ist. Die Auswahl kann in der Realität sehr wohl schrittweise während der Berechnung erfolgen. Sie muß nur konsistent mit der Modellierung sein. Die Darstellung der Berechnung durch eine Zustandsübergangsrelation mit schrittweiser Auswahl scheint zwar implementierungsnäher, ist es aber in der Praxis nicht, weil die schematischen Zusatzbedingungen (zum Beispiel die Fairness von *UNITY*) zu stark sind, um sich effizient implementieren zu lassen.

Jede Strategie beschreibt das gesamte Verhalten des Automaten. Dies unterscheidet eine Strategie von einem Orakel (siehe Abschnitt 5.1), denn Orakel werden wie Daten behandelt, also etwa wie ein zusätzlicher Eingabewert, den der Agent erhält. Formal sind natürlich beide Formulierungen äquivalent: Einerseits können Strategienmengen über Orakel parametrisiert werden (zum Beispiel die Strategien h_z in Beispiel 5.1 für $z \in Id \to \mathbb{N}$), und andererseits kann eine Orakelmenge aus Strategien bestehen.

Um den in Bild 5.1 dargestellten Verfeinerungsschritt von einer eingeschränkten funktionalen Spezifikation F nach Abschnitt 5.1 zu einer zustandsorientierten Spezifikation zu bewerkstelligen, muß eine Strategie H gefunden werden, so daß $Strat_H$ die Spezifikation F impliziert. In einfachen Fällen geschieht das, indem F schrittweise in eine syntaktische Form gebracht wird, die sich mit dem Schema *Strat* zur Deckung bringen läßt. Normalerweise ist es jedoch bequemer, eine Strategie H anzugeben und die Verfeinerungseigenschaft über den folgenden Satz zu zeigen:

Satz 5.3: Gegeben sei eine funktionale Spezifikation $F \subseteq IS \twoheadrightarrow OS$, die für eine nichtleere Menge *Oracle* definiert ist durch:

$$F.f \;=\; \exists\, z \in Oracle : \forall\, s \in IS^{fin} : G.z.s.(f.s)$$

Weiter sei eine Strategie $H \subseteq STRAT_{IS,OS}$ gegeben. Es gilt $F \Leftarrow Strat_H$, wenn sich zeigen läßt:

$$\begin{array}{ll} \forall\, h \in H : \exists\, z \in Oracle : & \\ \quad G.z.\varepsilon.\varepsilon \wedge & (1) \\ \quad \forall\, s \in IS^{fin},\, r \in IS,\, a \in Act.IS : G.z.s.r \Rightarrow G.z.(s \circ \bar{a}).(r \circ h.s.a) & (2) \end{array}$$

Beweis: Gegeben sei eine Funktion $f \in IS \twoheadrightarrow OS$ mit $Strat_H.f$, also $f.\varepsilon = \varepsilon$ (3) und $f.(s \circ \bar{a}) = f.s \circ h.s.a$ (4) für ein $h \in H$ und alle $s \in IS^{fin}$, $a \in Act.IS$. Zu zeigen ist, daß für ein die Bedingungen (1) und (2) erfüllendes $z \in Oracle$ auch $f \in F$ gilt, also $G.z.s.(f.s)$ für alle $s \in IS^{fin}$. Wir zeigen dies durch strukturelle Induktion über s.

Für $s = \varepsilon$ gilt nach (1) $G.z.\varepsilon.\varepsilon$ und weiter nach (3) $G.z.\varepsilon.(f.\varepsilon)$. Für den Schritt von s auf $s \circ \bar{a}$ nehmen wir $G.z.s.(f.s)$ an. Nach (2) gilt dann, wenn wir $r = f.s$ setzen, $G.z.(s \circ \bar{a}).(f.s \circ h.s.a)$ und nach (4) $G.z.(s \circ \bar{a}).(f.(s \circ \bar{a}))$. □

Beispiel 5.3: Durch Anwendung des Satzes 5.3 läßt sich zeigen, daß das Prädikat *FeederFun′* von Beispiel 5.1 durch das Prädikat *FeederFun″* von Beispiel 5.2 verfeinert wird. □

5.3 Implizite Zustände — Schema *RStrat*

Das Schema *Strat* ist die einfachste Formulierung einer zustandsorientierten Spezifikation. Hier fallen Zustand und bisherige Eingaben zusammen. In den folgenden Abschnitten wird der Zustand als explizites, von den Eingaben getrenntes Hilfsobjekt weitergebildet. Als ersten Schritt dazu geben wir das Schema *RStrat* an, in dem Zustand und Eingaben getrennt sind. Die Form von *RStrat* ist an rekursive Definitionen in funktionalen Programmiersprachen angelehnt. Der ursprüngliche implizite Zustandsraum wird in einer Spezifikation nach dem Schema *RStrat* noch beibehalten.

Definition 5.2 (Spezifikationsschema RStrat): Ein Prädikat nach dem Schema $RStrat_H$ ist für ein Hilfsprädikat $H \subseteq STRAT_{IS,OS}$ definiert vermöge:

$$
\begin{aligned}
RStrat_H \quad &\subseteq IS \twoheadrightarrow OS \\
RStrat_H.f \; &= \exists\, h \in H,\, g \in IS \to IS \twoheadrightarrow OS : \forall\, s \in IS^{fin},\, x \in IS,\, a \in Act.IS : \\
&\qquad f = g_\varepsilon \wedge \\
&\qquad g_s.\varepsilon = \varepsilon \;\wedge\; g_s.(\bar{a} \circ x) = h.s.a \circ g_{(s \circ \bar{a})}.x
\end{aligned}
$$
□

Die Umformung von *Strat* nach *RStrat* ist ein trivialer Entwicklungsschritt, denn beide Prädikatenschemata sind äquivalent, wie der folgende Satz zeigt:

Satz 5.4: Für alle $H \subseteq STRAT_{IS,OS}$ gilt $Strat_H = RStrat_H$.

Beweis: Wir zeigen zunächst die folgende Behauptung:

Behauptung 5.1: Gegeben sei ein $g \in IS \to IS \twoheadrightarrow OS$, so daß $g_s.\varepsilon = \varepsilon$ (1) und $g_s.(\bar{a} \circ x) = h.s.a \circ g_{(s \circ \bar{a})}.x$ (2) für alle $s \in IS^{fin}$, $x \in IS$, $a \in Act.IS$ gilt. Dann gilt auch $g_\varepsilon.s \circ h.s.a = g_\varepsilon.(s \circ \bar{a})$ für alle $s \in IS^{fin}$, $a \in Act.IS$.

Beweis von Behauptung 5.1: Durch strukturelle Induktion über s zeigen wir die allgemeinere Behauptung $g_u.s \circ h.(u \circ s).a = g_u.(s \circ \bar{a})$ für alle $u, s \in IS^{fin}$, $a \in Act.IS$. Behauptung 5.1 ist der Spezialfall $u = \varepsilon$.

Für den Induktionsbeginn $s = \varepsilon$ erhalten wir durch zweimalige Anwendung von (1) die Gleichung $g_u.\varepsilon \circ h.(u \circ \varepsilon).a = h.u.a \circ g_{(u \circ \bar{a})}.\varepsilon$, und daraus mit (2) das gewünschte Ergebnis $g_u.\varepsilon \circ h.(u \circ \varepsilon).a = g_u.(\varepsilon \circ \bar{a})$.

Für den Induktionsschritt von s auf $\bar{b} \circ s$ gilt:

$$\begin{aligned}
& g_u.(\bar{b} \circ s) \circ h.(u \circ \bar{b} \circ s).a \\
&= h.u.b \circ g_{(u \circ \bar{b})}.s \circ h.(u \circ \bar{b} \circ s).a && \text{[nach (2)]} \\
&= h.u.b \circ g_{(u \circ \bar{b})}.(s \circ \bar{a}) && \text{[Induktionsannahme]} \\
&= g_u.(\bar{b} \circ s \circ \bar{a}) && \text{[nach (2)]}
\end{aligned}$$

□

Zum Beweis von Satz 5.4 genügt es zu zeigen, daß $Strat_{\{h\}} = RStrat_{\{h\}}$ für alle $h \in STRAT_{IS,OS}$ gilt. Gegeben sei ein beliebiges $h \in STRAT_{IS,OS}$. Wir führen den Beweis für beide Implikationsrichtungen getrennt.

"$Strat_{\{h\}} \Leftarrow RStrat_{\{h\}}$": Für jedes $f \in RStrat_{\{h\}}$ existiert nach Definition 5.2 ein $g \in IS \to IS \twoheadrightarrow OS$, das die Bedingungen der Behauptung 5.1 erfüllt und für das $f = g_\varepsilon$ gilt. Nach der Behauptung 5.1 gilt $g_\varepsilon.s \circ h.s.a = g_\varepsilon.(s \circ \bar{a})$ für alle $s \in IS^{fin}$, $a \in Act.IS$ und wegen $f = g_\varepsilon$ erhält man $f.s \circ h.s.a = f.(s \circ \bar{a})$, also $f \in Strat_{\{h\}}$.

"$Strat_{\{h\}} \Rightarrow RStrat_{\{h\}}$": Gegeben sei ein $f \in Strat_{\{h\}}$. Zu der Strategie h existiert offensichtlich ein $g \in IS \to IS \twoheadrightarrow OS$, das die Bedingungen (1) und (2) von Behauptung 5.1 (und somit das zweite und dritte Konjunktionsglied von Definition 5.2) erfüllt.

Eigenschaft (1) von g_ε entspricht dem ersten Konjunktionsglied von Definition 5.1 des Schemas *Strat*. Aus Behauptung 5.1 folgt das zweite Konjunktionsglied. Damit gilt $g_\varepsilon \in Strat_{\{h\}}$.

Weil durch das Schema *Strat* für jede Strategie höchstens eine stetige Funktion beschrieben wird, gilt damit $f = g_\varepsilon$ (also das erste Konjunktionsglied von Definition 5.2) und schließlich $f \in RStrat_{\{h\}}$. □

Beispiel 5.4: Die Agentenspezifikation *FeederFun''* ist nach Satz 5.4 äquivalent zu dem Prädikat *FeederFun'''*, das für die in Beispiel 5.2 angegebene Strategienmenge H vermöge $FeederFun''' = RStrat_H$ definiert ist. □

Im bisherigen Verlauf der Entwicklung wurde aus einer allgemeinen funktionalen Spezifikation zunächst eine zustandsorientierte Spezifikation nach dem Schema *Strat* beziehungsweise *RStrat* abgeleitet, wobei der Zustandsraum implizit festgelegt war. Der folgende Abschnitt beschreibt die Verwendung explizit angegebener Zustandsräume.

5.4 Explizite Zustandsräume — Schema *EStrat*

Spezifikationen, die einen impliziten Zustandsraum verwenden, lassen sich nicht direkt effizient implementieren, denn der implizite Zustand enthält die *vollständige* Information über die bisherigen Eingaben. Diese Information müßte gespeichert und in jedem Berechnungsschritt erneut ausgewertet werden. Um eine vernünftige Implementierung

zu ermöglichen, ist es notwendig, als Entwurfsentscheidung einen Zustandsraum explizit anzugeben. Wir bezeichnen dies als eine Spezifikation mit *explizitem Zustandsraum.* Dieser soll folgende Eigenschaften aufweisen:

- Jeder Zustand enthält genügend Informationen zur Bestimmung des zukünftigen Systemverhaltens.
- Der Zustandsraum ist klein genug für eine effiziente Implementierung.
- Die Komponenten der einzelnen Zustände entsprechen Datenstrukturen, die in der Implementierungssprache verfügbar sind.
- Die durch die Eingaben bewirkten Zustandsübergänge sind effizient implementierbar.
- Die Bestimmung der Ausgaben ist effizient implementierbar.

Es kann keine generelle Regel zur Konstruktion eines solchen expliziten Zustandsraumes gegeben werden, der den fünf oben angegebenen Kriterien optimal entspricht. Insbesondere würde ein hinsichtlich der Größe *minimaler* Zustandsraum, wie er von der Automatentheorie her bekannt ist, im allgemeinen die dritte bis fünfte der oben angegebenen Eigenschaften nicht besitzen.

In der Praxis wird man von der Überlegung ausgehen, welche Informationen über den bisherigen Berechnungsablauf zur Bestimmung des zukünftigen Systemverhaltens notwendig sind. Diese Informationen wird man dann durch Datenstrukturen darstellen, die an die geplante Implementierungssprache angepaßt sind. Oft geben Hilfsprädikate, die in der Spezifikation mit implizitem Zustandsraum verwendet werden, einen Anhaltspunkt für die darzustellenden Informationen, so etwa das Prädikat *pending* aus Beispiel 5.2.

Zur Aufschreibung eines expliziten Zustandsraumes bieten sich Techniken des *Software Engineering* an, wie etwa die *Schemas* der Spezifikationssprache *Z* [Spi87]. Für die einfachen Beispielsfälle der vorliegenden Arbeit reicht jedoch die übliche mathematische Mengenschreibweise aus.

Eine Spezifikation nach dem im folgenden angegebenen Schema $EStrat_{H,I}$ beschreibt einen Automaten mit Zustandsraum ST. Die Zustandsübergangsfunktion $h \in H$ gibt zu jedem Zustand s und jeder Eingabeaktion a die erzeugten Ausgaben $h.s.a.out$ sowie den Folgezustand $h.s.a.next$ an. Die Menge der zulässigen Anfangszustände wird mit I bezeichnet.

Definition 5.3 (Spezifikationsschema mit explizitem Zustandsraum):
Gegeben seien Kanaltypen IS und OS sowie ein Zustandsraum ST. Die Menge $ESTRAT_{ST,IS,OS}$ von Strategien ist definiert vermöge:

$$ESTRAT_{ST,IS,OS} = ST \to Act.IS \to (out : OS \otimes next : ST)$$

Eine Spezifikation mit explizitem Zustand ist ein Prädikat nach dem Schema $EStrat_{H,I}$ für eine Menge $I \subseteq ST$ von Anfangszuständen und ein Hilfsprädikat $H \subseteq ESTRAT_{ST,IS,OS}$:

$$\begin{aligned} EStrat_{H,I} &\subseteq IS \twoheadrightarrow OS \\ EStrat_{H,I}.f &= \exists\, h \in H,\ i \in I,\ g \in ST \to IS \twoheadrightarrow OS : \\ &\qquad \forall\, s \in ST,\ x \in IS,\ a \in Act.IS : \\ &\qquad\quad f = g_i \wedge \\ &\qquad\quad g_s.\varepsilon = \varepsilon \ \wedge\ g_s.(\bar{a} \circ x) = h.s.a.out \circ g_{h.s.a.next}.x \end{aligned}$$ □

Beispiel 5.5: Das Prädikat $FeederFun'''$ von Beispiel 5.4 wird durch eine Spezifikation $FeederFun'''' = EStrat_{H',I'}$ verfeinert. Jeder Zustand aus ST ist eine Abbildung von einem Identifikator entweder auf die leere Sequenz oder auf die Sequenz, die als erstes Element einen noch zu bearbeitenden Rechenauftrag und als weitere Elemente die bisher für diesen Auftrag erhaltenen Antworten aufweist. Formal sind der Zustandsraum ST, die Menge der Anfangszustände I' und die Strategienmenge H' gegeben durch:

$$\begin{aligned} ST &= Id \to (Cmd \cup Rply)^{\omega} \\ I' &= \{s \in ST : \forall\, i \in Id : s.i = \langle\rangle\} \\ H' &\subseteq ESTRAT_{ST,\, InType.PSysStruct'.Feeder,\, OutType.PSysStruct'.Feeder} \\ H' &= \{h'_z : z \in Id \to \mathbb{N}\} \end{aligned}$$

Jedes h'_z ist dabei durch die in Bild 5.3 gezeigte Tabellennotation definiert. □

Bei einer Spezifikation nach dem Schema $EStrat_{H,I}$ wird das Verhalten des Agenten von der Auswahl einer Strategie $h \in H$ und der Auswahl eines Anfangszustandes $i \in I$ bestimmt. Wie der folgende Satz 5.5 zeigt, können diese beiden Auswahlschritte auf eine einzige Entscheidung konzentriert werden. Natürlich braucht die Auswahl nicht notwendigerweise als nichtdeterministische Entscheidung während des Programmablaufs erfolgen. Sie ist auch in Form von Entwurfsentscheidungen während der Programmentwicklung möglich. Die Verhaltensmöglichkeiten des Agenten können dadurch bis zu einem deterministischen Programm eingeschränkt werden.

Satz 5.5: Für alle Zustandsmengen ST, Mengen $H \subseteq ESTRAT_{ST,IS,OS}$ von Strategien und Mengen $I \subseteq ST$ von Anfangszuständen gibt es eine Zustandsmenge ST_1, ein $h_1 \in ESTRAT_{ST_1,IS,OS}$ und ein $I_1 \subseteq ST$ mit $EStrat_{H,I} = EStrat_{\{h_1\},I_1}$. Weiter existieren eine Zustandsmenge ST_2, ein $H_2 \subseteq ESTRAT_{ST_2,IS,OS}$ und ein $i_2 \in ST$ mit $EStrat_{H,I} = EStrat_{H_2,\{i_2\}}$.

Beweis: Der erste Teil des Satzes 5.5 ist bei gegebenen ST, H und I offensichtlich erfüllt für:

$$\begin{aligned} ST_1 &= \{(s,h) : s \in ST,\ h \in H\} \\ I_1 &= \{(i,h) : i \in I,\ h \in H\} \\ h_1 &\quad \text{mit}\ \ h_1.(s,h).a = (out \mapsto h.s.a.out) \cup (next \mapsto (h.s.a.next, h)) \end{aligned}$$

Der zweite Teil des Satzes ist erfüllt für:

$$\begin{aligned} ST_2 &= ST \cup \{s'\} \text{ mit } s' \notin ST \\ i_2 &= s' \\ H_2 &= \{h'_{h,i} : h \in H,\ i \in I\} \text{ mit } h'_{h,i}.s.a = h.i.a \text{ für } s = s' \\ &\qquad\qquad\qquad\qquad\qquad\qquad = h.s.a \text{ sonst} \end{aligned}$$

□

Bei der im obigen Beweis gezeigten Konstruktion vergrößert sich allerdings der Zustandsraum ST, und zwar im ersten Fall um die Strategie $h \in H$ als zusätzliche Zustandskomponente und im zweiten Fall um einen neuen Startzustand s'.

Das Prädikatenschema *RStrat* aus Abschnitt 5.3 ist nur ein Spezialfall des Schemas *EStrat*, wie sich durch Einsetzen der Definition von H' der folgenden Anmerkung 5.1 in das Schema *EStrat* ergibt:

Anmerkung 5.1: Für Kanaltypen IS und OS und eine Menge $H \subseteq STRAT_{IS,OS}$ von Strategien sei $H' \subseteq ESTRAT_{IS,IS,OS}$ gegeben durch:

$$\begin{aligned} H'.h' = \exists\, h \in H : \forall\, s \in IS^{fin},\ a \in Act.IS : \\ h'.s.a = (out \mapsto h.s.a) \cup (next \mapsto s \circ \bar{a}) \end{aligned}$$

Dann gilt $RStrat_H = EStrat_{H',\{\varepsilon\}}$. □

Diese Aussage ist natürlich für die Systementwicklung wenig brauchbar, weil der implizite Zustand ja weiterentwickelt werden soll, also $ST = IS$ gerade nicht gelten soll. Um zu zeigen, daß eine Spezifikation nach dem Schema *EStrat* eine Spezifikation nach dem Schema *RStrat* impliziert, kann man eine Repräsentationsfunktion φ von impliziten zu expliziten Zuständen angeben und zeigen, daß die beiden Automaten bezüglich der Zustandspaare s und $\varphi.s$ äquivalent sind.

Dieses Beweisprinzip wird durch den folgenden Satz 5.6 ausgedrückt. Der Satz ist ein deutlich einfacherer Spezialfall des Satzes 5.8, denn über die Repräsentationsfunktion φ kann nur eine Verfeinerungseigenschaft gezeigt werden, bei der der Zustandsraum gröber wird, also mehrere implizite Zustände einem expliziten Zustand entsprechen. Methodisch gesehen reicht dies für den Ebenenübergang zwischen impliziten und expliziten Zuständen aus, denn der implizite Zustandsraum IS ist auf der hier betrachteten Abstraktionsebene der feinste sinnvolle Zustandsraum.

Satz 5.6: Gegeben seien ein Zustandsraum ST und Agentenspezifikationen $RStrat_H$ und $EStrat_{H',I'}$. Für alle $h' \in H'$ und $i' \in I'$ existiere ein $h \in H$ und eine Funktion $\varphi \in IS \to ST$, so daß gilt:

$$\begin{aligned} &\varphi.\varepsilon \in I' \wedge \\ &\forall\, x \in IS^{fin},\ a \in Act.IS : h'.(\varphi.x).a = (out \mapsto h.x.a) \cup (next \mapsto \varphi.(x \circ \bar{a})) \end{aligned}$$

Dann gilt $RStrat_H \Leftarrow EStrat_{H',I'}$.

Beweis: Gegeben seien ST, $RStrat_H$ und ein $f \in EStrat_{H',I'}$. Nach Definition 5.3 gibt es ein $h' \in H'$ und $i' \in I'$ mit $f \in EStrat_{\{h'\},\{i'\}}$, und nach der Voraussetzung des Satzes 5.6 existieren dann ein $h \in H$ und $\varphi \in IS \to ST$ mit $\varphi.\varepsilon = i'$ (1) und $\forall\, x \in IS^{fin},\ a \in Act.IS : h'.(\varphi.x).a = (out \mapsto h.x.a) \cup (next \mapsto \varphi.(x \circ \bar{a}))$ (2).

Zu beweisen ist $RStrat_H.f$. Wir zeigen dazu in einem ersten Schritt $EStrat_{\{\hat{h}\},\{\varepsilon\}}.f$ für ein geeignet definiertes $\hat{h}$ unter Verwendung des Satzes 5.8. In einem zweiten Schritt wird dann $RStrat_{\{h\}}.f$ unter Verwendung der Anmerkung 5.1 gezeigt.

Schritt 1: Es sei $\hat{h} \in ESTRAT_{IS,IS,OS}$ definiert durch $\hat{h}.s.a = (out \mapsto h.s.a) \cup (next \mapsto s \circ \bar{a})$ und $\sim\ \subseteq IS \times ST$ durch $s \sim s' = (\varphi.s = s')$. Dann gilt nach Satz 5.8 mit $EStrat_{\{h'\},\{i'\}}.f$ auch $EStrat_{\{\hat{h}\},\{\varepsilon\}}.f$, denn es ist $\varepsilon \sim i'$ (nach (1) und der Definition von $\sim$) und für $s \sim s'$ gilt:

$$\begin{aligned} \hat{h}.s.a.out &= h.s.a && \text{[Definition von } \hat{h}\text{]} \\ &= h'.(\varphi.s).a.out && \text{[nach (2)]} \\ &= h'.s'.a.out && \text{[Definition von } \sim \text{ für } s \sim s'\text{]} \end{aligned}$$

Ebenso gilt $\hat{h}.s.a.next \sim h'.s'.a.next$ wegen:

$$\begin{aligned} \varphi.(\hat{h}.s.a.next) &= \varphi.(s \circ \hat{a}) && \text{[Definition von } \hat{h}\text{]} \\ &= h'.(\varphi.s).a.next && \text{[nach (2)]} \\ &= h'.s'.a.next && \text{[Definition von } \sim \text{ für } s \sim s'\text{]} \end{aligned}$$

Schritt 2: Da die Definition von $\hat{h}$ offensichtlich den Voraussetzungen von Anmerkung 5.1 entspricht, gilt $RStrat_{\{h\}}.f$ und damit auch $RStrat_H.f$. □

Beispiel 5.6: Wir betrachten die Definition des Prädikats *FeederFun''''* in Beispiel 5.5. Dort wurde die intuitive Bedeutung eines Zustands $s \in ST$ bereits erläutert. Für jedes $h'_z \in H'$ bildet die im folgenden definierte Funktion φ_z einen impliziten Zustand $s \in InType.PSysStruct'.Feeder$ auf einen expliziten Zustand $\varphi_z.s$ ab, der eine Funktion von Identifikatoren i auf Befehls- und Antwortsequenzen ist:

$$\begin{aligned} \varphi_z.s.i &= ft.(Cmd_i \copyright s.MtoF)\,\&\,(\{Error(i)\} \copyright s.MtoF) && \text{wenn } pending_{i,(z.i)}.s \\ &= \langle\rangle && \text{sonst} \end{aligned}$$

Unter Verwendung des Satzes 5.6 läßt sich damit zeigen, daß die Verfeinerungsbeziehung $FeederFun'''' \Leftarrow FeederFun'''$ gilt. □

Wie oben erwähnt, sind explizite Zustandsräume im allgemeinen nicht minimal in dem Sinne, daß sich alle Zustände beobachtbar (also hinsichtlich des Ein-/Ausgabeverhaltens) unterscheiden. Es existieren also äquivalente Zustände. Überdies kann es wünschenswert sein, einen Zustandsraum zu definieren, der "überflüssige", also durch keine Eingabe erreichbare Zustände enthält. Das gilt insbesondere dann, wenn bei einem Zustandsraum mit mehreren Komponenten nur einige spezielle Fälle ausgeschlossen sind. Zum Beispiel hat es sich in der Fallstudie der Beschreibung eines

Postsystems [Den91] als praktisch erwiesen, jedem Postfach zwei unabhängige Warteschlangen für sendende beziehungsweise empfangende Prozesse zuzuordnen, obwohl nur Zustände erreichbar sind, in denen mindestens eine dieser Warteschlangen leer ist. Diese Warteschlangen sind auch inaktiven Postfächern zugeordnet, so daß zwei Zustände äquivalent sind, die sich nur hinsichtlich des Inhalts der Warteschlangen von inaktiven Postfächern unterscheiden.

Bevor wir auf die methodische Verwendung der Begriffe Erreichbarkeit und Äquivalenz eingehen, legen wir sie in den folgenden Definitionen 5.4 und 5.5 formal fest. Offensichtlich ist die Relation $\sim$ eine Äquivalenzrelation auf H.

Definition 5.4 (Äquivalente Zustände): Für eine Strategie $H \subseteq ESTRAT_{ST,IS,OS}$ sind zwei Zustände $s, s' \in ST$ *äquivalent,* (geschrieben $s \sim s'$), wenn $EStrat_{H,\{s\}} = EStrat_{H,\{s'\}}$ gilt. □

Definition 5.5 (Erreichbare Zustände): Für eine Strategie $H \subseteq ESTRAT_{ST,IS,OS}$ und ein Prädikat $S \subseteq ST$ sind die von S aus *erreichbaren Zustände* $REACH_{H,S} \subseteq ST$ definiert durch $REACH_{H,S} = \bigcup_{h \in H} RCH_{h,S}$, wobei $RCH_{h,S}$ das stärkste Prädikat ist, das die folgenden Bedingungen erfüllt:

$$\begin{aligned} &RCH_{h,S} \supseteq S \\ &RCH_{h,S} \supseteq \{h.s.a.next : s \in S,\ a \in Act.IS\} \\ &RCH_{h,S} \supseteq RCH_{h,RCH_{h,S}} \end{aligned}$$

□

Beispiel 5.7: Für die Mengen ST, H' und I' aus Beispiel 5.5 und $s, s' \in ST$ erhält man die folgende Äquivalenzrelation $\sim$ und Menge $REACH_{H',I'}$ erreichbarer Zustände:

$$s \sim s' = \forall\, i \in Id : \#s.i = \#s'.i \ \wedge\ (s.i \neq \langle\rangle \Rightarrow (ft.(s.i) = ft.(s'.i)))$$

$$\begin{aligned} REACH_{H',I'} = \{s \in ST : \forall\, i \in Id : s.i \neq \langle\rangle \ \Rightarrow \\ (ft.(s.i) \in Cmd_i \ \wedge\ rt.(s.i) \in \{Error(i)\}^*)\} \end{aligned}$$

□

Eine alternative Charakterisierung der erreichbaren Zustände als diejenigen Zustände, die sich in endlich vielen Schritten von einem Initialzustand erreichen lassen, ergibt sich durch den folgenden Satz 5.7:

Satz 5.7: Für eine Strategie $H \subseteq ESTRAT_{ST,IS,OS}$ und ein Prädikat $S \subseteq ST$ gilt:

$$REACH_{H,S} = \{step_h.s.t : h \in H,\ s \in S,\ t \in (Act.IS)^*\}$$

Hierbei ist die Funktion *step* definiert durch:

$$\begin{aligned} step &\in ESTRAT_{ST,IS,OS} \to ST \to (Act.IS)^\omega \to ST \\ step_h.s.\langle\rangle &= s \\ step_h.s.(a \,\&\, t) &= step_h.(h.s.a.next).t \end{aligned}$$

Beweis: Es sei $STP_{h,S} = \{step_h.s.t : s \in S,\ t \in (Act.IS)^*\}$. Zum Beweis des Satzes genügt es, $STP_{h,S} = RCH_{h,S}$ zu zeigen. Wir führen den Beweis getrennt für beide Inklusionsrichtungen.

Zum Beweis der Richtung $STP_{h,S} \supseteq RCH_{h,S}$ zeigen wir, daß $STP_{h,S}$ die drei in Definition 5.5 geforderten Eigenschaften hat. Offensichtlich gilt $STP_{h,S} \supseteq \{step_h.s.\langle\rangle : s \in S\} = S$ und $STP_{h,S} \supseteq \{step_h.s.\langle a\rangle : s \in S,\ a \in Act.IS\} = \{h.s.a.next : s \in S,\ a \in Act.IS\}$. Außerdem ist die dritte Eigenschaft erfüllt, denn es gilt:

$$
\begin{array}{ll}
STP_{h,S} & \\
= \{step_h.s.t : s \in S,\ t \in (Act.IS)^*\} & \text{[Definition von } STP\text{]} \\
= \{step_h.s.(t \circ t') : s \in S,\ t, t' \in (Act.IS)^*\} & \text{[Aufspaltung von } t\text{]} \\
= \{step_h.(step_h.s.t).t' : s \in S,\ t, t' \in (Act.IS)^*\} & \text{[Eigenschaft von } step\text{]} \\
= STP_{h,STP_{h,S}} & \text{[Definition von } STP\text{]}
\end{array}
$$

Zum Beweis der Richtung $STP_{h,S} \subseteq RCH_{h,S}$ zeigen wir durch Induktion über $t \in (Act.IS)^*$, daß jeweils $\{step_h.s.t : s \in S\} \subseteq RCH_{h,S}$ gilt.

Für $t = \langle\rangle$ folgt dies mit $\{step_h.s.\langle\rangle : s \in S\} = S \subseteq RCH_{h,S}$ aus der ersten Eigenschaft in Definition 5.5 und für $\#t = 1$ mit $\{step_h.s.\langle a\rangle : s \in S,\ a \in Act.IS\} = \{h.s.a.next : s \in S,\ a \in Act.IS\} \subseteq RCH_{h,S}$ aus der zweiten Eigenschaft.

Für den Schritt von t auf $t \circ \langle a\rangle$ für $\#t \geq 1$ erhalten wir:

$$
\begin{array}{ll}
\{step_h.s.(t \circ \langle a\rangle) : s \in S\} & \\
= \{step_h.(step_h.s.t).\langle a\rangle : s \in S\} & \text{[Eigenschaft von } step\text{]} \\
= \{h.s'.a.next : s' \in \{step_h.s.t : s \in S\}\} & \text{[Definition von } step\text{]} \\
\subseteq \{h.s'.a.next : s' \in RCH_{h,S}\} & \text{[Induktionsannahme]} \\
\subseteq RCH_{h,RCH_{h,S}} & \text{[Definition von } RCH\text{]} \\
\subseteq RCH_{h,S} & \text{[Definition von } RCH\text{]}
\end{array}
$$

□

Nicht-erreichbare und äquivalente Zustände brauchen nicht unbedingt gesondert betrachtet zu werden, das heißt, man kann eine Systementwicklung auch durchführen, wenn man alle Zustände als erreichbar und nicht-äquivalent ansieht. Allerdings wird die Verfeinerung von expliziten Zustandsräumen erleichtert, wenn wir nur die erreichbaren Zustände berücksichtigen.

Wenn eine zustandsorientierte Spezifikation als Ausgangspunkt einer Entwicklung dient (also nicht als Verfeinerung einer abstrakteren Spezifikation bewiesen wird), ist es sinnvoll, eine formale Beschreibung der nicht-erreichbaren und äquivalenten Zustände als redundante Information zur Konsistenzüberprüfung anzugeben. Eine solche Spezifikation für einen Agenten $F \subseteq IS \twoheadrightarrow OS$ besteht aus der Angabe einer Zustandsmenge ST, einer Strategie $H \subseteq ESTRAT_{ST,IS,OS}$, einer Menge initialer Zustände $I \subseteq ST$, einem Prädikat $REACH'$ und einer Äquivalenzrelation $\sim'$.

Um zu verifizieren, ob die Spezifikation die erwarteten und durch das Prädikat $REACH'$ sowie die Relation $\sim'$ angegebenen Eigenschaften aufweist, ist zum Beispiel $REACH' = REACH_{H,I}$ und $\sim = \sim'$ zu zeigen. Natürlich kann man diese Beweisverpflichtung auch abschwächen und beispielsweise nur verlangen, daß bei der Spezifikation keine erreichbaren Zustände übersehen wurden, also daß $REACH' \supseteq REACH_{H,I}$ gilt.

5.5 Verfeinerung von Zustandsräumen

Auch für Agentenspezifikationen nach dem Schema *EStrat* ist die Verfeinerung als Implikation (oder, gleichbedeutend, als Mengeninklusion) definiert. Es ist wünschenswert, an den Formalismus der expliziten zustandsorientierten Spezifikation angepaßte Beweistechniken zu verwenden, mit denen die Implementierungsbeziehung einfacher gezeigt werden kann. Diese Beweistechniken sollen es erlauben, den konkreten Zustandsraum zu verändern. Das kann nötig sein, um die Spezifikation den in der späteren Implementierung verwendeten Datenstrukturen anzunähern.

In praktisch vorkommenden Entwicklungen tritt bei der Verfeinerung sowohl der Fall auf, daß ein ursprünglicher Zustand durch mehrere neue Zustände ersetzt werden soll (wenn zum Beispiel implementierungsnahe Informationen zusätzlich verwaltet werden müssen), als auch der umgekehrte Fall (wenn etwa mehrere äquivalente Zustände durch einen einzigen Zustand ersetzt werden). Da also der Zustandsraum bei der Entwicklung sowohl vergröbert als auch verfeinert werden kann, verwenden wir im folgenden Satz ein Prädikat $\sim$, das einander entsprechende Zustände in Beziehung setzt:

Satz 5.8 (Verfeinerung von expliziten Zustandsräumen):
Gegeben seien Zustandsräume ST und ST', Mengen $H \subseteq ESTRAT_{ST,IS,OS}$ und $H' \subseteq ESTRAT_{ST',IS,OS}$ von Strategien sowie Mengen $I \subseteq ST$ und $I' \subseteq ST'$ von Anfangszuständen. Für alle $h' \in H'$ und $i' \in I'$ gebe es eine Strategie $h \in H$, einen Anfangszustand $i \in I$ und ein Prädikat $\sim \subseteq ST \times ST'$, so daß gilt:

$$
\begin{array}{l}
i \sim i' \wedge \\
\forall\, s \in ST,\, s' \in ST',\, a \in Act.IS : \\
\quad s \sim s' \;\Rightarrow\; (h.s.a.out = h'.s'.a.out \wedge h.s.a.next \sim h'.s'.a.next)
\end{array}
$$

Dann gilt $EStrat_{H,I} \Leftarrow EStrat_{H',I'}$.

Beweis: Wir zeigen zunächst die folgende Behauptung 5.2:

Behauptung 5.2: Gegeben sei ein $h \in ESTRAT_{ST,IS,OS}$ und ein $h' \in ESTRAT_{ST',IS,OS}$. Zu h sei $g \in ST \to IS \to OS$ definiert durch $g_s.\varepsilon = \varepsilon \wedge g_s.(\bar{a} \circ x) = h.s.a.out \circ g_{h.s.a.next}.x$, und zu h' sei $g' \in ST' \to IS \to OS$ definiert durch $g'_s.\varepsilon = \varepsilon \wedge g'_s.(\bar{a} \circ x) = h'.s.a.out \circ g'_{h'.s.a.next}.x$. Weiter sei eine Relation $\sim \subseteq ST \times ST'$ gegeben, so daß für

alle $s \in ST$, $s' \in ST'$ und $a \in Act.IS$ mit $s \sim s'$ gilt $h.s.a.out = h'.s'.a.out$ (1) und $h.s.a.next \sim h'.s'.a.next$ (2).

Dann ist $g_s = g'_{s'}$ für alle $s \in ST$, $s' \in ST'$ mit $s \sim s'$.

Beweis von Behauptung 5.2: Wir zeigen $\forall\, s \in ST,\ s' \in ST' : s \sim s' \Rightarrow (g_s.x = g'_{s'}.x)$ durch strukturelle Induktion über den Aufbau von $x \in IS$.

Für $x = \varepsilon$ gilt $g_s.\varepsilon = \varepsilon = g'_{s'}.\varepsilon$ für alle s und s' nach der Definition von g und g'.

Für den Schritt von x auf $\bar{a} \circ x$ seien beliebige $s \in ST$, $s' \in ST'$ mit $s \sim s'$ gegeben. Nach der Definition von g gilt $g_s.(\bar{a} \circ x) = h.s.a.out \circ g_{h.s.a.next}.x$, und nach der Definition von g' gilt $g'_{s'}.(\bar{a} \circ x) = h'.s'.a.out \circ g'_{h'.s'.a.next}.x$. Nun ist aber wegen (1) $h.s.a.out = h'.s'.a.out$, und nach der Induktionsvoraussetzung gilt $g_{h.s.a.next}.x = g'_{h'.s'.a.next}.x$, denn $h.s.a.next$ und $h'.s'.a.next$ stehen wegen (2) in der Relation $\sim$. Damit ist $g_s.(\bar{a} \circ x) = g'_{s'}.(\bar{a} \circ x)$ gezeigt. □

Zum Beweis von Satz 5.8 sei ein $f \in EStrat_{H',I'}$ gegeben. Zu zeigen ist, daß unter den Voraussetzungen des Satzes auch $f \in EStrat_{H,I}$ gilt. Nach der Definition von *EStrat* gibt es ein $h' \in H'$ und ein $i' \in I'$, so daß für ein (durch h' eindeutig festgelegtes) g' auch $g'_{i'} = f$ gilt. Nach der Voraussetzung des Satzes 5.8 existieren zu diesem h' und i' auch $h \in H$, $i \in I$ und $\sim \subseteq ST \times ST'$, so daß $i \sim i'$ (1) und $s \sim s' \Rightarrow (h.s.a.out = h'.s'.a.out \wedge h.s.a.next \sim h'.s'.a.next)$ (2) gelten. Wegen (2) ist die obige Behauptung 5.2 anwendbar, und aus dieser folgt wegen (1), daß $g_i = g'_{i'}$ gilt. Insgesamt erhalten wir $f = g'_{i'} = g_i$, also $f \in EStrat_{\{h\},\{i\}}$ und somit auch $f \in EStrat_{H,I}$. □

Eine interessante Folgerung ist, daß unter den in Satz 5.8 aufgeführten Bedingungen zu jedem erreichbaren Zustand in ST' auch ein entsprechender Zustand in ST existiert:

Folgerung 5.1: Unter den Bedingungen von Satz 5.8 gilt:

$$\forall\, s' \in REACH_{H',I'} : \exists\, s \in REACH_{H,I} : s \sim s'$$

Beweis von Folgerung 5.1: Gegeben sei ein beliebiges $s' \in REACH_{H',I'}$. Es gibt dann ein $h' \in H'$ und $i' \in I'$ mit $s' \in REACH_{\{h'\},\{i'\}}$, und nach den Voraussetzungen des Satzes 5.8 gibt es auch $h \in H$, $i \in I$ und $\sim \subseteq ST \times ST'$, so daß $i \sim i'$ (1) und $s \sim s' \Rightarrow (h.s.a.out = h'.s'.a.out \wedge h.s.a.next \sim h'.s'.a.next)$ (2) gelten. Außerdem läßt sich für alle $t \in (Act.IS)^*$ durch Induktion über t unter Verwendung von (1) und (2) zeigen, daß $step_h.i.t \sim step_{h'}.i'.t$ $(*)$ gilt. Da nach Satz 5.7 $s' = step_{h'}.i'.t$ für ein $t \in (Act.IS)^*$ gelten muß, folgt mit $(*)$ auch $step_h.i.t \sim s'$ und daraus wieder mit Satz 5.7 $s \sim s'$ für ein $s \in REACH_{\{h\},\{i\}} \subseteq REACH_{H,I}$. □

Die in Satz 5.8 angegebene Beweisregel dient zum Nachweis der Verfeinerungseigenschaft von Agentenspezifikationen nach dem Schema *EStrat*. Leider ist die Beweisregel nicht vollständig, weil es Paare von Agentenspezifikationen gibt, zwischen denen die als Implikation definierte Verfeinerungseigenschaft besteht, die aber die Voraussetzungen

von Satz 5.8 nicht erfüllen. Der Grund dafür ist, daß nicht-implementierbare Strategien existieren, die sich nicht auf die Menge der durch eine Spezifikationen nach dem Schema *EStrat* beschriebenen Agenten auswirken, aber die Anwendbarkeit von Satz 5.8 zerstören. Dies wird im folgenden Gegenbeispiel 5.1 gezeigt:

Gegenbeispiel 5.1: Gegeben seien der Eingabekanaltyp $IS = i_1 : M^\omega \otimes i_2 : M^\omega$, der Ausgabekanaltyp $OS = \otimes\, o : M^\omega$, die Zustandsmenge $ST = \{s\}$ und die im folgenden definierten Strategien $h_\varepsilon, h_{mg} \in ESTRAT_{ST,IS,OS}$.

Die Strategie h_ε beschreibt einen Agenten, der niemals Ausgaben erzeugt. Sie ist für $a \in Act.IS$ definiert vermöge $h_\varepsilon.s.a.out.o = \langle\rangle$ und $h_\varepsilon.s.a.next = s$. Offensichtlich bezeichnet $EStrat_{\{h_\varepsilon\},\{s\}}$ genau eine stromverarbeitende Funktion.

Die Strategie h_{mg} beschreibt das nicht-strikte faire Mischen. Sie ist für $c \in \{i_1, i_2\}$ und $m \in M$ definiert vermöge $h_{mg}.s.(c,m).out.o = \langle m\rangle$ und $h_{mg}.s.(c,m).next = s$. Weil das nicht-strikte faire Mischen durch keine (monotone und stetige) stromverarbeitende Funktion darstellbar ist, gilt $EStrat_{\{h_{mg}\},\{s\}} = \varnothing$.

Das Verhalten einer Strategienmenge ergibt sich durch die Vereinigung der Verhaltensweisen der einzelnen Strategien. Somit gilt $EStrat_{\{h_{mg}\},\{s\}} = EStrat_{\{h_\varepsilon,h_{mg}\},\{s\}}$. Die Agentendefinition $EStrat_{\{h_\varepsilon,h_{mg}\},\{s\}}$ ist also eine Verfeinerung von $EStrat_{\{h_{mg}\},\{s\}}$. Trotzdem ist Satz 5.8 nicht anwendbar, denn im Zustand s definiert die Strategie h_ε andere Ausgabewerte als die Strategie h_{mg}. □

Die Technik, Verfeinerungsbeziehungen in sequentiellen Programmen durch die Angabe von Abstraktions- oder Repräsentationsfunktionen (wie die Funktion φ in Abschnitt 5.4) oder durch eine Relation auf Zuständen zu zeigen, geht auf frühe Arbeiten über die Korrektheit von Programmen zurück. Sie wurde insbesondere auf dem Gebiet der abstrakten Datentypen und der algebraischen Spezifikationen weiterentwickelt. Ein umfassender Überblick über dort vorgeschlagene Implementierungsbegriffe findet sich in [Wir90].

Auch bei modellorientierten Spezifikationstechniken wird die Technik der Zustandsverfeinerung häufig eingesetzt; beispielsweise verwendet *Z* [Spi87] ein *abstraction schema,* um abstrakte und konkrete Zustände zueinander in Beziehung zu setzen. Zu demselben Zweck dient im *Refinement Calculus* [Mor90] eine sogenannte *coupling invariant.*

Weiter bestehen Zusammenhänge zwischen unserer Beweistechnik und den *refinement mappings* aus [AL88]. Dort werden Zustandssequenzen durch eine Abbildung miteinander in Beziehung gesetzt. Es wird gezeigt, daß die Abbildung unter bestimmten Voraussetzungen die Vorgeschichte eines Zustandes nicht zu berücksichtigen braucht. Die Zustandssequenzen in [AL88] stellen die eigentliche Spezifikation dar, sie sind also nicht, wie in unserem Ansatz, lediglich ein Hilfsmittel zur Beschreibung von Funktionen. Dabei ergeben sich insbesondere dann Schwierigkeiten, wenn zwei Systeme mit

unterschiedlicher Geschwindigkeit laufen. Diese Probleme treten bei der hier vorgestellten automatenorientierten Spezifikationstechnik nicht auf, denn jede Eingabe bewirkt genau einen Zustandsübergang.

Die hier vorgeschlagene Relation $\sim$ erinnert an Äquivalenzbegriffe in Prozeßalgebren wie etwa die *Bisimulation* [GW90]. Es sei *Proc* eine Menge von Prozessen (beispielsweise in *CCS*) und *Act* die Menge von Aktionen, die diese Prozesse ausführen können. Für $p, p' \in Proc$ und $a \in Act$ schreiben wir $p \xrightarrow{a} p'$, um auszudrücken, daß p mit der Aktion a in p' übergehen kann. Zwei Prozesse p und q heißen *bisimular,* wenn eine Relation $R \subseteq Proc \times Proc$ auf Prozessen existiert, für die gilt:

$$
\begin{array}{l}
p\,R\,q\ \wedge \\
\forall\, x, x', y \in Proc,\ a \in Act : x\,R\,y \wedge x \xrightarrow{a} x' \ \Rightarrow\ \exists\, y' \in Proc : x'\,R\,y' \wedge y \xrightarrow{a} y'\ \wedge \\
\forall\, x, y, y' \in Proc,\ a \in Act : x\,R\,y \wedge y \xrightarrow{a} y' \ \Rightarrow\ \exists\, x' \in Proc : x'\,R\,y' \wedge x \xrightarrow{a} x'
\end{array}
$$

Diese Definition ähnelt den in Satz 5.8 gegebenen Anforderungen an die Relation $\sim$. Die vorhandenen Unterschiede lassen sich durch die unterschiedliche Struktur von *CCS*-Prozessen einerseits und unseren Automaten andererseits begründen.

Als ersten Unterschied zwischen *CCS*-Prozessen und unseren Automaten stellen wir fest, daß ein *CCS*-Prozeß eine Aktion entweder ganz ablehnen oder mit ihr in einen von mehreren Folgeprozessen übergehen kann. Daher sind in der obigen Formulierung die Existenzquantifikation des Folgeprozesses und die getrennte Angabe zweier Implikationsrichtungen notwendig. In unserem Automatenbegriff ist dagegen bei gegebenen Strategien h und h' für alle Zustände s und s' und alle Aktionen a jeweils genau ein Folgezustand $h.s.a.next$ beziehungsweise $h'.s'.a.next$ festgelegt, so daß sich die gewünschte Eigenschaft durch die eine Implikation von Satz 5.8 ausdrücken läßt.

Als zweiter Unterschied sind bei unserem Ansatz Ein- und Ausgabe getrennt (wodurch eine asynchrone Kommunikation möglich wird), während bei der synchronen Kommunikation der Prozeßalgebren diese Unterscheidung nicht erforderlich ist. Unsere Aktionen sind reine Eingabeaktionen, und daher ist es in Satz 5.8 erforderlich, als zusätzliche Bedingung zu verlangen, daß jeweils die Ausgaben $h.s.a.out$ und $h'.s'.a.out$ übereinstimmen.

Die hier verwendete Relation $\sim$ unterscheidet sich also formal von einer Bisimulation im Sinne von [GW90], beruht aber auf ähnlichen Grundideen. Methodisch gibt es dagegen erhebliche Unterschiede, denn während die Bisimulationsäquivalenz in Prozeßalgebren ein eigenständiges Konzept ist, stellt für uns die Relation $\sim$ lediglich ein Hilfsmittel zum Nachweis einer über das extensionale Verhalten zweier Agenten definierten Verfeinerungsbeziehung dar.

5.6 Implementierung

Die Entwicklung auf der Ebene der Spezifikationen mit explizit angegebenem Zustandsraum hat zum Ziel, durch schrittweise Modifikation von Zustandsraum und Strategie die folgenden Anforderungen im Hinblick auf eine zu verwendende Implementierungssprache zu erfüllen:

- Der Zustandsraum läßt sich effizient auf Datenstrukturen der Implementierungssprache abbilden.
- Bei Verwendung einer deterministischen Implementierungssprache beschreibt die Strategie H genau eine Übergangs- und Ausgabefunktion.
- Die Definition der Strategie H ist so konkret, daß sie sich in effiziente Algorithmen der Implementierungssprache umsetzen läßt.

Eine solche explizite zustandsorientierte Spezifikation stellt in der hier vorgestellten Entwurfsmethodik den Endpunkt der Entwicklung dar. Sie ist nach der Umsetzung in die Implementierungssprache effizient ausführbar.

Bei Agenten mit mehreren Eingabekanälen muß darauf geachtet werden, daß die Implementierung einer Strategienmenge H mit Anfangszuständen I nur solche Verhalten zeigt, die mit einem Element aus $EStrat_{H,I}$ verträglich sind. Dies heißt, daß als Basis für die Implementierung nur eine Strategie $h \in H$ und ein Anfangszustand $i \in I$ verwendet werden dürfen, für die $EStrat_{\{h\},\{i\}} \neq \varnothing$ gilt. Dies ist etwa für die in Gegenbeispiel 5.1 angegebene Spezifikation $EStrat_{\{h_{mg}\},\{s\}}$ des nicht-strikten fairen Mischens nicht der Fall. Auch wenn die Strategie h_{mg} schematisch in ein lauffähiges Programm umgesetzt werden könnte, ist ein solches Programm natürlich keine zulässige Implementierung des (im bisher vorgestellten Formalismus nicht existierenden) Agenten $EStrat_{\{h_{mg}\},\{s\}}$.

Wenn also eine Implementierung durch schematische Umsetzung einer Strategie (und eines Anfangszustandes) erhalten werden soll, ist es im allgemeinen erforderlich, die Verträglichkeit dieser Strategie mit unserer funktionalen Agentensicht zu beweisen. Diese Schwierigkeit tritt nicht auf bei Agenten mit nur einem Eingabekanal und bei den in Kapitel 6 behandelten gezeiteten Agenten.

Als Zielsprachen für die Implementierung bieten sich neben den *general purpose languages* insbesondere Sprachen an, die auf dem Konzept erweiterter Zustandsmaschinen beruhen, wie zum Beispiel die Protokollbeschreibungssprache SDL [SDL89]. Eine prototypische Sprache für verteilte Systeme ist in [BDS93] beschrieben. Die Implementierung dieser Sprache verwendet C für die Definition von Agenten und das Postsystem des *Multiprocessor Multitasking Kernel* MMK [BKM+91] für die Kommunikationskanäle.

Bei der Implementierung von stetigen stromverarbeitenden Funktionen müssen geeignete Kommunikationsprimitive gewählt werden. Auf jeden Fall muß das Senden von

Nachrichten nicht-blockierend sein, was sich durch geeignetes Scheduling in Verbindung mit Nachrichtenpuffern sicherstellen läßt.

Hinsichtlich des Empfangs von Nachrichten reicht es nicht aus, als Kommunikationsprimitiv nur das blockierende Warten auf die nächste Nachricht eines Eingabekanals zuzulassen. Damit ließe sich nur eine Teilmenge der stetigen Funktionen implementieren, nämlich die sogenannten *sequentiellen* Funktionen. Nicht implementieren ließe sich etwa ein Agent mit zwei Eingängen und zwei Ausgängen, der die am ersten Eingang ankommenden Nachrichten an den ersten Ausgang und die am zweiten Eingang ankommenden Nachrichten an den zweiten Ausgang weitergibt.

Um alle Agenten implementieren zu können, muß man eine Möglichkeit vorsehen, Eingabekanäle nicht-blockierend abzufragen ("polling"). Eine derartige Kommunikation ist jedoch in gewisser Hinsicht zu mächtig, denn sie erlaubt es, Agenten zu implementieren, die sich nicht angemessen mit stetigen stromverarbeitenden Funktionen modellieren lassen, zum Beispiel das nicht-strikte faire Mischen. Als Lösung bietet sich an, den Formalismus zu erweitern und gezeitete stromverarbeitende Funktionen zu verwenden, die im folgenden Kapitel 6 behandelt werden.

Kapitel 6

Zeitmodellierung

Bisher haben wir nur Spezifikationen betrachtet, bei denen die während des Programmablaufs vergehende Zeit nicht berücksichtigt wurde. Diese Zeitabstraktion dient der notationellen und semantischen Vereinfachung. Sie ermöglicht einfachere und leichter handhabbare Spezifikationen für nicht zeitkritische Systeme.

Eine vollständige Zeitabstraktion ist jedoch nicht immer adäquat. Offensichtlich müssen Zeiteigenschaften zumindest dann explizit modelliert werden, wenn Zeitanforderungen Bestandteil der Funktionalität eines Systems sind, wie beispielsweise bei zeitkritischen Programmen oder Echtzeitanwendungen. Auch wenn keine strengen Zeitanforderungen bestehen, kann eine vollständige Zeitabstraktion modellierungstechnische Schwierigkeiten verursachen. Bei der FOCUS zugrundeliegenden kanalgebundenen Kommunikation geht durch die Zeitabstraktion die zeitliche Vergleichbarkeit von Nachrichten auf unterschiedlichen Kanälen verloren. Ein klassisches Beispiel für dadurch entstehende Probleme ist die Tatsache, daß sich ein nicht-strikter fairer Mischagent nicht durch monotone stromverarbeitende Funktionen beschreiben läßt.

Die in der Literatur zur Zeitmodellierung vorgeschlagenen Formalismen basieren auf einem breiten Spektrum unterschiedlicher Abstraktionsebenen. Sie enthalten unterschiedlich viel Information über den Zeitablauf und sind daher für unterschiedliche Anwendungsfälle geeignet. In diesem Kapitel geben wir zunächst einen Überblick über eine Reihe von Zeitmodellierungen. Dann definieren wir einen einfachen Zeitbegriff. Schließlich geben wir Methoden zur Spezifikation gezeiteter Agenten an.

6.1 Systematik verschiedener Zeitbegriffe

Wir gehen in diesem Abschnitt von einem sehr generellen Zeitkonzept aus und zeigen, wie sich daraus durch schrittweise Verfeinerung konkretere Zeitbegriffe herleiten lassen.

Als Grundannahmen legen wir fest, daß es einen oder mehrere Zeitmaßstäbe gibt, auf denen sich Zeitpunkte eindeutig als reelle Zahlen kennzeichnen lassen. Weiter nehmen wir an, daß das Gesamtsystem durch eine Menge kombinierbarer Systemstrukturen repräsentiert wird. Jeder dieser Systemstrukturen muß sich eindeutig ein Zeitmaßstab zuordnen lassen, das heißt, zumindest muß der Zeitbegriff innerhalb der einzelnen Systemstrukturen einheitlich sein.

Wir befassen uns mit *Beobachtungen,* wobei uns nur die Zeitpunkte interessieren, zu denen Nachrichten auf Kanäle gelegt beziehungsweise von diesen gelesen werden. Eine *vollständige Beobachtung* des Gesamtsystems wird durch eine Abbildung modelliert, die für jeden Zeitpunkt und jeden "Anschluß" eines Agenten (Verbindung von Agent und Kanal) die Menge der Nachrichten angibt, die zu dieser Zeit den Anschluß passiert.

Eine explizite Modellierung von Zeitintervallen halten wir nicht für erforderlich. Zeitintervalle können entweder eine Zeitdauer oder zulässige Toleranzen oder Meßungenauigkeiten widerspiegeln. Diese beiden Bedeutungen können jedoch auch ohne eine Verwendung expliziter Zeitintervalle modelliert werden. So reicht es zur Modellierung dauerhafter Vorgänge aus, zwei Aktionen zu verwenden, die Beginn und Ende des Vorgangs kennzeichnen. Mengen von vollständigen Beobachtungen können für nicht genau bestimmte Zeitpunkte stehen.

Wir untersuchen im folgenden, welche Verfeinerungen dieses generellen Zeitmodells sinnvoll sind und zu welchen Modellierungen sie führen.

Oben sind wir davon ausgegangen, daß in einer vollständigen Beobachtung sowohl das Senden als auch der Empfang einer Nachricht modelliert werden. Als Konsistenzbedingung fordern wir zumindest, daß Nachrichten nicht früher ankommen, als sie abgeschickt wurden. Es können aber auch, je nach den zu modellierenden Gegebenheiten, weitere Zeiteigenschaften der Datenübertragung angegeben werden. So ist zum Beispiel eine verzögerungsfreie (synchrone) Übertragung angemessen, wenn bei der Hardwaremodellierung ein Spannungspegel auf einer elektrischen Leitung dargestellt wird [Fuc94]. Eine beliebig verzögernde (asynchrone) Übertragung trifft die Verhältnisse in vielen verteilten Rechnerarchitekturen am besten, während zur Modellierung von räumlich sehr weit verteilten Netzen sogar Kanäle denkbar sind, welche die Reihenfolge der einzelnen Nachrichten vertauschen können.

Eine grundlegende Unterscheidung ist die zwischen globaler und lokaler Zeit. Ein lokaler Zeitbegriff geht von der Überlegung aus, daß es nicht sinnvoll ist, bei einem räumlich verteilten System verschiedene Aktionen alle auf *einer* Zeitskala anzuordnen. Als Gründe dafür werden meßtechnische Probleme oder die Schwierigkeit der Uhrensynchronisation angeführt. Tatsächlich ist der lokale Zeitbegriff näher an der Implementierung, weil auch die Verzögerung bei der Kommunikation zu einzelnen lokalen Zeitinseln führt.

Das Hauptargument für die Verwendung eines globalen Zeitbegriffs ist, daß dieser eine Spezifikation vereinfachen kann und daher eine sinnvolle Abstraktion darstellt, wenn die feinere Modellierung der lokalen Zeit nicht notwendig ist. Der eingangs beschriebene Zeitbegriff erlaubt beliebige Mischformen, weil er jeder Systemstruktur einen eigenen (lokalen) Zeitmaßstab zuordnet, der aber für alle Agenten innerhalb dieser Systemstruktur (global) gilt. Als Konsistenzbedingung fordern wir zumindest, daß es zwischen den Zeitskalen der einzelnen Systemstrukturen einen monotonen Zusammenhang gibt. Typischerweise werden noch weitere Synchronisationseigenschaften der lokalen Uhren bekannt sein.

Wir unterscheiden zwischen kontinuierlicher und diskreter Zeit. Kontinuierliche Zeit wird modelliert, indem man Zeitpunkte aus $\mathbb{R}$ zuläßt, oder, für praktische Anwendungen damit gleichbedeutend, aus $\mathbb{Q}$. Ein kontinuierlicher Zeitbegriff findet sich in Arbeiten über *Timed CSP* [RR86] oder ACP_ρ [BB90]. Um der operationellen Intuition gerecht zu werden, wird oft gefordert, daß nicht unendlich viele Aktionen in endlicher Zeit stattfinden dürfen.

Demgegenüber verwenden diskrete Zeitbegriffe nur ganzzahlige Vielfache einer festgelegten Zeiteinheit. Der Hauptvorteil diskreter Zeitbegriffe ist die einfachere Handhabbarkeit für Spezifikationen und Beweise, etwa durch Induktion. Ein diskreter Zeitbegriff mit festen Zeitpunkten eignet sich zur Modellierung von Hardwaresignalen, bei denen der Zustand einer Leitung in festen Abständen abgetastet wird. Beim Schicken von Nachrichten über einen Kanal ist es dagegen nicht sinnvoll, zu fordern, daß jede Nachricht zu einem "ganzzahligen" Zeitpunkt geschickt wird. Vielmehr braucht nur beschrieben zu werden, daß die Nachricht in einem bestimmten *Zeitintervall* gesendet wird. Dann ist es eine vereinfachende Modellierung, ganzzahlige Zeitpunkte als Repräsentanten für die entsprechenden Intervalle zu verwenden.

Ein diskreter Zeitbegriff mit ganzzahligen Zeitpunkten kann durch einen mengenwertigen Strom modelliert werden, bei dem die einzelnen Mengen die zu jeweils einem Zeitpunkt auf einem Kanal ausgetauschten Nachrichten enthalten. Bei einem solchen diskreten Zeitbegriff muß als Modellierungsentscheidung festgelegt werden, ob die Anzahl der in einem Zeitintervall übertragenen Nachrichten begrenzt sein soll. Die Beschränkung der maximalen Kommunikationsrate auf eine Nachricht pro Zeiteinheit ermöglicht eine weitere Vereinfachung der Modellierung, denn ein derartig begrenzter Strom ist offensichtlich isomorph zu einem Strom, der an jeder Stelle entweder das in diesem Zeitintervall gesendete Element enthält, oder ein spezielles Zeichen $\checkmark$, wenn keine "echte" Nachricht versendet wurde. Mit dieser Zeitmodellierung werden wir uns im folgenden Abschnitt 6.2 noch genauer beschäftigen.

Verwandt mit diesem Zeitbegriff sind Ansätze, in denen keine exakten Zeitangaben auftreten, sondern nur eine besondere Nachricht, die für das Ablaufen eines unbestimmten, endlichen Zeitintervalls steht (zum Beispiel ein *Hiaton* in [Par82]).

6.2 Grundlagen eines einfachen Zeitbegriffs

In diesem Abschnitt soll der oben angedeutete diskrete, globale Zeitbegriff mit bestimmten Zeitintervallen und beschränkter Kommunikationsrate als spezifikationstechnisch besonders einfache Zeitmodellierung genauer beschrieben werden. Wir gehen von einem Grundzeitintervall ("Systemtakt") aus, das als fest angenommen wird, auch wenn es nicht konkret in irgendeiner Einheit (zum Beispiel Mikrosekunden) angegeben werden muß. In jedem solchen Zeitintervall kann auf jedem Kanal genau eine Nachricht gesendet werden. Die Tatsache, daß auf einem Kanal keine "echte" Nachricht übertragen wird, stellen wir durch die Pseudo-Nachricht ✓ ("Tick") dar.

Gegeben seien eine Menge X mit der Ordnung $\sqsubseteq$ und ein zusätzliches Element $\checkmark \notin X$. Wir erweitern die Ordnung $\sqsubseteq$ auch auf $X \cup \{\checkmark\}$, wobei ✓ mit keinem Element aus X vergleichbar sein soll. Entsprechend der Definition in Abschnitt 2.2 bezeichnet $(X \cup \{\checkmark\})^\omega$ die Menge der Ströme mit Elementen aus $X \cup \{\checkmark\}$, und $\sqsubseteq^\omega$ bezeichnet die punktweise, auf $\sqsubseteq$ gestützte Ordnung auf dieser Menge. Die Menge X^τ der *gezeiteten Ströme* mit der Ordnung $\sqsubseteq^\tau$ ist definiert vermöge:

$$\begin{aligned} X^\tau &= (X \cup \{\checkmark\})^\omega \\ s \sqsubseteq^\tau t &= s \sqsubseteq^\omega t \ \wedge \ (\forall\, i \in \mathbb{N} : i + 1 < \#s \Rightarrow s.i = t.i) \end{aligned}$$

In der gewählten Approximationsordnung $\sqsubseteq^\tau$ spiegelt sich die Tatsache wider, daß die Vergangenheit nicht verändert werden kann. Ein gezeiteter Strom s approximiert einen gezeiteten Strom t genau dann, wenn t entweder eine Verlängerung von s ist (also die Zeit fortgeschritten ist), oder wenn die Ströme gleich lang sind und das letzte Element von s das letzte Element von t approximiert. Die Menge X^τ mit der Ordnung $\sqsubseteq^\tau$ ist ein Bereich, wenn die Menge $X^\perp$ (gemäß der Definition in Abschnitt 2.2) mit der Ordnung $\sqsubseteq^\perp$ ein Bereich ist.

Analog zu Definition 3.1 bezeichnen wir als *gezeitete Kanaltypen* Mengen der Form $\otimes_{n \in N}\, n : (M_n)^\tau$ für eine Menge N von Kanalnamen, denen jeweils eine Nachrichtenmenge M_n (für $n \in N$) zugeordnet ist. Ein gezeiteter Kanalzustand ist jedes Element eines gezeiteten Kanaltyps.

Wir führen nun noch zwei Hilfsoperationen ein. Für einen nicht-gezeiteten Kanaltyp S bezeichnen wir mit S^τ den entsprechenden gezeiteten Kanaltyp:

$$S^\tau = \otimes_{c \in \bullet S}\, c : (Msg.(Str_S.c))^\tau$$

Aus einem gezeiteten Kanalzustand $s \in S^\tau$ werden durch die im folgenden definierte Funktion *untick* die Zeitmarkierungen entfernt:

$$\begin{aligned} untick &\in S^\tau \to S \\ untick.s.c &= Msg.(Str_S.c) \copyright s.c \end{aligned}$$

Einen Nachrichtenstrom der Länge i interpretieren wir dahingehend, daß damit die auf dem entsprechenden Kanal gesendeten oder noch zu sendenden Nachrichten bis zum Zeitpunkt i feststehen. Das heißt nicht notwendigerweise, daß die Ein- und Ausgaben eines Agenten alle stets die gleiche Länge haben müssen, denn es können ja zum Beispiel die Ausgaben bis zum Zeitpunkt $i+1$, die Eingaben aber nur bis zum Zeitpunkt i feststehen. Das ist etwa bei verzögernden Agenten der Fall, weil sich hier die Eingaben erst (ein oder mehrere Zeitintervalle) später auf die Ausgabe auswirken. Als Mindestanforderung für gezeitete Agenten verlangen wir aber, daß kein Ausgabestrom für eine kürzere Zeit festgelegt ist als der kürzeste Eingabestrom. Wenn nämlich ein Agent die Ausgabe für den Zeitpunkt i erst zum Zeitpunkt $i+1$ bestimmen müßte, dann müßte der Agent die Vergangenheit ändern oder in die Zukunft sehen können. Wir erhalten somit die folgende Definition:

Definition 6.1 (Gezeitete Agenten): Gegeben seien zwei gezeitete Kanaltypen IS und OS. Ein *gezeiteter Agent* ist eine stetige Funktion $f \in IS \twoheadrightarrow OS$, wobei für alle $s \in IS$ die *Zeitfortschrittsanforderung* $\#s \leq \#f.s$ gilt. □

Analog zur Definition in Abschnitt 3.5 sind gezeitete Agentenspezifikationen Prädikate $P \subseteq IS \twoheadrightarrow OS$, wobei IS und OS gezeitete Kanaltypen sind und $P.f$ nur für gezeitete Agenten f gilt.

Beispiel 6.1: Der in Beispiel 4.4 durch das Spurprädikat *TimerTrace* definierte Agent *Timer* gibt für jeden am Eingang *FtoT* ankommenden Rechenauftrag mit Identifikator i nach einer bestimmten Zeit eine Zeitfehler-Nachricht $Error(i)$ auf den Ausgang *TtoM*. Die Wahl der Zeitschranke beeinflußt zwar nicht die Korrektheit, wohl aber die Leistung des Gesamtsystems, denn eine Zeitfehler-Nachricht soll natürlich erst dann erzeugt werden, wenn die bei einwandfreier Funktion der Prozessoren zu erwartende Rechenzeit überschritten ist. Daher ist es sinnvoll, den Agenten *Timer* in einem gezeiteten Formalismus weiterzuentwickeln.

Die folgende Spezifikation *TimerFun* führt eine Hilfsfunktion h ein, die für jeden Zeitpunkt k, zu dem ein Rechenauftrag eintrifft, den Zeitpunkt $h.k$ angibt, zu dem die entsprechende Zeitfehler-Nachricht ausgegeben wird. Im Prädikat *TimerFun* wird nur gefordert, daß zwischen Rechenauftrag und Zeitfehler-Nachricht eine Verzögerung von mindestens einer Zeiteinheit liegen soll. Diese relativ schwache Forderung müßte in einer weiteren Systementwicklung natürlich noch verfeinert und konkretisiert werden.

$$
\begin{aligned}
TimerFun \quad &\subseteq (InType_{PSysStruct'}.Timer)^{\tau} \twoheadrightarrow (OutType_{PSysStruct'}.Timer)^{\tau} \\
TimerFun.f = \; &\exists\, h \in \mathbb{N} \to \mathbb{N} : \forall\, x \in (InType_{PSysStruct'}.Timer)^{\tau},\, i \in Id,\, j,k \in \mathbb{N} : \\
&\qquad f.x.PCmd = x.FtoT \;\wedge \\
&\qquad h.j > j \;\wedge \\
&\qquad j \neq k \;\Rightarrow\; h.j \neq h.k \;\wedge \\
&\qquad x.FtoT.k \in Cmd_i \;\Leftrightarrow\; f.x.TtoM.(h.k) = Error(i)
\end{aligned}
$$

□

Bei gezeiteten Agenten lassen sich nach [BD92] zwei Unterklassen auszeichnen, nämlich die der *synchronen* und der *verzögernden* Agenten. Synchrone Agenten modellieren Baugruppen oder Programme, die im betrachteten Zeitmaßstab verzögerungsfrei sind, bei denen also die Ausgaben synchron zu den Eingaben erfolgen. Für synchrone Agenten $f \in IS \twoheadrightarrow OS$ gilt, daß die Länge jedes Ausgabestroms gleich der minimalen Länge der Eingabeströme ist:

$$\forall\, s \in IS,\ c \in \bullet OS : \#f.s.c = \#s$$

Bei verzögernden Agenten $f \in IS \twoheadrightarrow OS$ muß dagegen jede Ausgabe um mindestens ein Element länger sein als die kürzeste Eingabe. Das heißt, daß die Ausgaben bis zum Zeitpunkt $i+1$ nur durch die Eingaben bis zum Zeitpunkt i bestimmt werden, oder anders ausgedrückt, daß der Agent um mindestens eine Zeiteinheit verzögert:

$$\forall\, s \in IS,\ c \in \bullet OS : \#f.s.c > \#s$$

Zwei gezeitete Agenten, die sich für jede Eingabe nur bezüglich der Länge der Ausgabeströme unterscheiden, sind in dem Sinne äquivalent, daß zu jeder Zeit die Ausgabeströme zumindest bis zu der durch die Zeitfortschrittseigenschaft geforderten Länge übereinstimmen.

Die grundlegende Zeitfortschrittsanforderung von Definition 6.1 sowie die beiden gerade eingeführten Zeitfortschrittseigenschaften für synchrone und verzögernde Agenten sind Lebendigkeitseigenschaften im Sinne von Definition 3.6. Da durch jede dieser Zeitfortschrittseigenschaften die Mindestlänge der Ausgabe festgelegt ist, läßt sich ein gezeiteter Agent durch eine Sicherheitseigenschaft hinreichend genau, nämlich für diese Mindestlänge, spezifizieren.

Im folgenden wird die Abgeschlossenheit der verschiedenen Agentenklassen gegenüber der Kombination durch den Verbindungsoperator $\|$ untersucht. Wie in Abschnitt 3.3 erläutert worden ist, lassen sich drei Grundformen der Komposition auszeichnen, nämlich die parallele und sequentielle Komposition und die Rückkopplung. In der folgenden Tabelle 6.1 geben wir zunächst die Abgeschlossenheitseigenschaften gezeiteter, synchroner und verzögernder Agenten gegenüber diesen drei Grundformen an. Der Operator $\|^{\tau}$ in der letzten Zeile der Tabelle wird in Definition 6.2 eingeführt.

Abgeschlossenheits-eigenschaften	gezeitete Agenten	synchrone Agenten	verzögernde Agenten
sequentielle Komposition	gezeitet	synchron	verzögernd
parallele Komposition	gezeitet	gezeitet	verzögernd
direkte Rückkopplung	nicht gezeitet	nicht gezeitet	verzögernd
Kombination über $\|$	nicht gezeitet	nicht gezeitet	verzögernd
Kombination über $\|^{\tau}$	gezeitet	gezeitet	verzögernd

Tabelle 6.1: Eigenschaften unterschiedlicher Verbindungsoperatoren

Wie man aus Tabelle 6.1 ersieht, können die Abgeschlossenheitseigenschaften des allgemeinen Verbindungsoperators $\|$ nicht "besser" sein als die der jeweils "schlechtesten" Grundfunktion, da sich durch den Verbindungsoperator $\|$ alle Grundfunktionen und alle Netzstrukturen darstellen lassen.

Verzögernde Agenten haben insbesondere den Vorteil, daß sie gegenüber der Agentenkombination abgeschlossen sind. Eine Möglichkeit wäre es also, nur verzögernde Agenten zu verwenden. In der Realität weist jede Software- oder Hardware-Komponente eine gewisse Verzögerung auf, so daß die Verwendung verzögernder Agenten bei genügend feiner Zeitmodellierung den realen Gegebenheiten entspricht.

Oft möchte man jedoch gezielt eine gröbere Zeitmodellierung verwenden, in der die reale Verzögerung eines Agenten gerade nicht zum Ausdruck kommen soll. Für allgemeine gezeitete oder synchrone Agenten ist insbesondere die Rückkopplung von Kanälen problematisch, denn ein gezeiteter Agent erfüllt unter direkter Rückkopplung im allgemeinen die Zeitfortschrittsanforderung nicht mehr.

Ein Lösungsansatz für das Problem der Rückkopplung gezeiteter Agenten ist ein zweistufiger Zeitbegriff, in dem zwischen Makro- und Mikro-Zeitebenen unterschieden wird. Die Agenten arbeiten auf der Mikro-Zeitebene verzögernd, erscheinen aber synchron, wenn sie auf der Makro-Zeitebene betrachtet werden. Eine solche Zeitmodellierung findet sich zum Beispiel in [Bro93b] oder [Fuc94] für die Hardwarebeschreibung. In [Den90] ist ein Ansatz zur Erweiterung dieses zweistufigen Zeitbegriffs auf einen Zeitbegriff mit beliebig vielen hierarchischen Zeitebenen angegeben.

Als weiterer Vorschlag zur Lösung des genannten Problems wurde in [BD92] ein gezeiteter Rückkopplungsoperator angegeben, der als erstes Element in jeden rückgekoppelten Kanal einen Zeittick einfügt. Ein derartiger Rückkopplungsoperator erhält die Klassen gezeiteter, synchroner und verzögernder Agenten. Während dies ein theoretisch befriedigendes Ergebnis ist, ist es schwer zu motivieren, warum eine Verzögerung ausschließlich in rückgekoppelte Leitungen eingefügt werden soll, nicht jedoch in Leitungen, die beispielsweise zwei Agenten sequentiell miteinander verbinden. Dies gilt um so mehr, als sich eine sequentielle Komposition auch durch eine parallele Komposition mit geeigneter Rückkopplung darstellen läßt. Obwohl diese beiden Verbindungsstrukturen in der intuitiven Anschauung äquivalent sind, würden sich mit dem gezeiteten Rückkopplungsoperator unterschiedliche Zeiteigenschaften ergeben. Die Unterscheidung zwischen Rückkopplungskanälen und sonstigen Verbindungen entspricht also nicht den tatsächlichen Verhältnissen, weder bei der Modellierung von Hardware- noch bei der von Softwareagenten.

Aus diesen Gründen schlagen wir vor, einen Kombinationsoperator für gezeitete Agenten zu verwenden, der eine Verzögerung in Form eines Zeitticks in *jeden* internen Kanal einfügt. Ein solcher Operator $\|^{\tau}$ ist in der folgenden Definition 6.2 angegeben.

Definition 6.2 (Verzögernde Kombination von gezeiteten Agenten): Gegeben sei eine endliche, nicht-leere Namensmenge N und gezeitete Agenten $f_n \in IS_n \twoheadrightarrow OS_n$ für $n \in N$, die nach Definition 3.4 kombinierbar sind. Abkürzend schreiben wir I für $\bigcup_{n\in N} \bullet IS_n$ und O für $\bigcup_{n\in N} \bullet OS_n$. Wir definieren $\hat{\|}^{\tau}_{n\in N} f_n$ als die kleinste Funktion, die die folgende Gleichung erfüllt:

$$\hat{\|}^{\tau}_{n\in N} f_n \in \otimes_{c\in I}\, c : \bigcup_{n\in\{m\in N : c\in\bullet IS_m\}} Str_{IS_n}.c \twoheadrightarrow \otimes_{c\in O}\, c : \bigcup_{n\in\{m\in N : c\in\bullet OS_m\}} Str_{OS_n}.c$$

$$(\hat{\|}^{\tau}_{n\in N} f_n).x \;=\; \bigcup_{n\in N} f_n.(x'|_{\bullet IS_n})$$

$$\text{wobei}\quad x'.c = x.c \qquad \text{für}\quad c \in I \setminus O$$

$$x'.c = \checkmark \,\&\, (\hat{\|}^{\tau}_{n\in N} f_n).x.c \qquad \text{für}\quad c \in I \cap O$$

Die Kombination $\|^{\tau}_{n\in N} f_n$ ergibt sich vermöge:

$$\|^{\tau}_{n\in N} f_n \;=\; (\hat{\|}^{\tau}_{n\in N} f_n)|^{I\setminus O}_{O\setminus I} \qquad \square$$

Wie aus der letzten Zeile von Tabelle 6.1 ersichtlich ist, erhält die Kombination über den Operator $\|^{\tau}$ die allgemeine Zeitfortschrittseigenschaft und die Verzögerungseigenschaft. Die Kombination synchroner Agenten ist aber im allgemeinen nicht synchron, zum Beispiel bei der sequentiellen Hintereinanderschaltung zweier synchroner Agenten. Der Operator $\|^{\tau}$ ist also hauptsächlich zur Verbindung allgemeiner gezeiteter oder verzögernder Agenten geeignet.

Bei der Spezifikation größerer Systeme kann es wünschenswert sein, Teilspezifikationen mit unterschiedlichen Zeitbegriffen oder im Extremfall gezeitete und ungezeitete Spezifikationen zu verbinden. Die Verbindung von gezeiteten Agenten, denen verschieden granulare Zeitbegriffe zugrunde liegen (also Zeitbegriffe, bei denen jeweils ein Nachrichtenelement unterschiedlich lange Zeitspannen repräsentiert), ist in [BD92] ausführlich beschrieben.

6.3 Spezifikation gezeiteter Agenten

Bei gezeiteten Agentendefinitionen unterscheiden wir in Anlehnung an [Bro93a] und [BS94] drei Klassen, die hier informell charakterisiert werden:

- Zeitunabhängige Agenten sind solche, bei denen durch eine Zeitabstraktion kein zusätzlicher Nichtdeterminismus entsteht. Ein zeitunabhängiger Agent kann ohne Informationsverlust hinsichtlich der Nachrichten, die keine Zeitticks sind, als stromverarbeitende Funktion in einem ungezeiteten Formalismus spezifiziert und dann schematisch in einen gezeiteten Agenten umgesetzt werden.
- Schwach zeitabhängig sind nichtdeterministische Agenten, deren Nichtdeterminismus auf eingeschränkte Weise von den Zeiteigenschaften der Eingabe abhängt. Bei einer Zeitabstraktion ergeben sich aus schwach zeitabhängigen Agenten keine

(stetigen) stromverarbeitenden Funktionen. Es ist möglich, solche Agenten mit Formalismen zu spezifizieren, die zwischen gezeiteten und ungezeiteten Formalismen angesiedelt sind.

- Zeitabhängige Agenten sind solche, deren Ausgaben streng von den Zeiteigenschaften der Eingaben abhängen, so daß eine sinnvolle Zeitabstraktion oder eine schematische Umsetzung einer ungezeiteten Spezifikation nicht möglich ist.

Zeitunabhängige Agenten können im gewohnten ungezeiteten Formalismus spezifiziert werden. Die schematische Umsetzung eines ungezeiteten Agenten $f \in IS \nrightarrow OS$ in einen gezeiteten Agenten $g \in IS^\tau \nrightarrow OS^\tau$ erfolgt im einfachsten Fall vermöge der im folgenden angegebenen Beziehung, die noch durch die geforderte Zeitfortschrittseigenschaft verstärkt werden muß:

$$\forall\, s \in IS^\tau : untick.(g.s) = f.(untick.s)$$

Der in den Beispielen 4.6 und 5.1 bis 5.5 angegebene Agent *Feeder* kann auf diese schematische Weise in eine gezeitete Spezifikation umgesetzt werden.

Für schwach zeitabhängige Agenten betrachten wir das Beispiel des nicht-strikten fairen Mischens, wie es etwa bei dem Agenten *Mixer* aus Beispiel 4.4 auftritt. Es ist ein altbekanntes Problem, daß das in der Praxis unabdingbare nicht-strikte faire Mischen sich weder durch eine stetige Funktion noch durch eine beliebig nichtdeterministische Auswahl aus einer Menge stetiger Funktionen modellieren läßt [Bro88a]. Durch Relationen oder mengenwertige Funktionen ließe sich ein nicht-strikter fairer Mischagent zwar darstellen; diese Formalismen sind jedoch nicht kompositional [BA81].

Diese Schwierigkeiten entstehen durch die völlige Abstraktion von der Zeit. Tatsächlich ist die Definition eines Mischagenten unproblematisch, wenn man einen gezeiteten Formalismus verwendet. Das heißt nicht, daß die Zeitticks explizit in der Spezifikation auftauchen müssen. Es genügt vielmehr, gezeitete Funktionen als semantische Basis zu verwenden. Schwach zeitabhängige Agenten können dann einfach als ungezeitete Relationen spezifiziert und schematisch in gezeitete Agenten umgesetzt werden, wobei man unmittelbar das Verhalten nur für unendliche Eingaben beschreibt. Diese im folgenden Beispiel 6.2 veranschaulichte Technik ist aus [BS94] bekannt.

Beispiel 6.2: Der in Beispiel 4.4 definierte Agent *Mixer* ist nur in Umgebungen implementierbar, die das erste Konjunktionsglied des Prädikats *MixerTrace* erfüllen. Wenn wir eine solche Umgebung annehmen, läßt sich der Agent *Mixer* durch das Prädikat *MixerFun* als schwach zeitabhängige Spezifikation angeben:

$$\begin{aligned} MixerFun' &\subseteq (InType_{PSysStruct'}.Mixer)^\tau \nrightarrow (OutType_{PSysStruct'}.Mixer)^\tau \\ MixerFun'.f &= \forall\, x \in (InType_{PSysStruct'}.Mixer)^\tau : \\ &\qquad \#x = \infty \;\Rightarrow\; MF.(untick.x).(untick.(f.x)) \end{aligned}$$

$$\begin{aligned} MF.x.y = \forall\, c \in Cmd : \#\{c\}\copyright y.MtoF = \#\{c\}\copyright x.SCmd \wedge \\ \forall\, r \in Rply : \#\{r\}\copyright y.MtoF = \#\{r\}\copyright x.SCmd + \\ \#\{r\}\copyright x.P1Rply + \\ \#\{r\}\copyright x.P2Rply \end{aligned}$$ □

Häufig wird behauptet, daß die Verwendung eines voll gezeiteten Formalismus ein unnötig mächtiges Instrument darstellt, wenn es nur darum geht, schwach zeitabhängige Agenten zu beschreiben. Ein etwas schwächerer Zeitbegriff sind die in [Par82] eingeführten *Hiatons*. Formal besteht ein hiatonisierter Strom (ebenso wie unser gezeiteter Strom) aus Nachrichtenelementen und einem Sondersymbol, dem Hiaton. Im Gegensatz zu unserer Modellierung bezeichnet ein Hiaton aber einen undefinierten, endlich langen Zeitraum. Daher ist die in Definition 6.1 gegebene Zeitfortschrittsanforderung zu stark; es genügt vielmehr, zu fordern, daß ein hiatonisierter Agent für unendlich lange Eingaben auch unendlich lange Ausgaben erzeugen muß. Hiatonisierte Agenten sind also etwas abstrakter als unsere gezeiteten Agenten, aber die realen Spezifikationen unterscheiden sich in ihrer Komplexität kaum.

Als weiterer Formalismus zur Spezifikation schwach zeitabhängiger Agenten wurden in [Bro93a] die sogenannten *input choice specifications* angegeben. Diese Beschreibungstechnik geht von der Idee aus, die Auswahl einer (deterministischen) stromverarbeitenden Funktion aus einer Agentenspezifikation abhängig von der Eingabe vorzunehmen. Die ausgewählte Funktion muß das gewünschte Ergebnis vollständig erzeugen. Die restlichen Funktionen der Agentenspezifikation dürfen keine Ausgaben liefern, die diesem Ergebnis widersprechen, also hinsichtlich dieses Ergebnisses nicht sicher sind.

Schließlich ist zu bemerken, daß Agenten wie das nicht-strikte faire Mischen problemlos durch eine Spurspezifikation beschrieben werden können. Dies ist nicht überraschend, da eine Spur eine Sequentialisierung *aller* Aktionen eines Systems darstellt. Um Monotonieprobleme zu vermeiden, reicht es dagegen schon aus, nur die Eingaben auf den Kanälen *eines* Agenten in eine zeitliche Beziehung zueinander zu setzen. Im Rahmen der hier dargestellten Entwurfsmethodik besteht daher die Möglichkeit, für schwach zeitabhängige Agenten die funktionale Beschreibung ganz zu umgehen, indem man direkt von der Spur- zur Implementierungsebene springt. Davon sollte natürlich nur Gebrauch gemacht werden, wenn dies im konkreten Fall auch methodisch sinnvoll ist.

Für die dritte oben aufgeführte Gruppe, also für gezeitete Agenten, die weder zeitunabhängig noch schwach zeitabhängig sind, müssen die geforderten Zeiteigenschaften explizit in der Spezifikation angegeben werden, wie etwa in Beispiel 6.1. In Anlehnung an die in Kapitel 5 gezeigten Techniken ist es möglich, einen solchen Agenten zustandsorientiert zu beschreiben.

In der folgenden Definition 6.3 wird als Zustandsraum der implizite Zustand verwendet, also die Kanalbelegung der bisherigen Eingaben. In einem Zustand s sei t eine zusätzliche "Zeitscheibe". Dies ist eine Kanalbelegung, die für jeden Kanal genau eine

Nachricht oder einen Zeittick enthält. Es gebe eine Sequentialisierung von t in die Aktionssequenz u, wobei die nicht nur einen Zeittick als Nachricht aufweisenden Aktionen von u die Sequenz $\langle a, b, c, \ldots \rangle$ bilden. Dann werden die zusätzlichen Ausgaben $h.s.\bar{a} \circ h.(s \circ \bar{a}).b \circ h.(s \circ \bar{a} \circ \bar{b}).c \circ \cdots$ erzeugt (Hilfsfunktion *scan*). Auf jedem sonst leeren Ausgabekanal wird ein Zeittick ausgegeben (Hilfsfunktion *addtime*).

Definition 6.3 (Zustandsorientierte Spezifikation gezeiteter Agenten):
Gegeben seien ungezeitete Kanaltypen IS und OS sowie ein Hilfsprädikat $H \subseteq STRAT_{IS,OS}$ wie in Definition 5.1. Wir setzen $\bullet IS < \infty$ voraus. Eine zustandsorientierte Spezifikation eines gezeiteten Agenten ist ein Prädikat nach dem Schema $TStrat_H$:

$$
\begin{aligned}
TStrat_H &\subseteq IS^{\tau} \nrightarrow OS^{\tau} \\
TStrat_H.f &= \exists\, h \in H : \forall\, s, t \in (IS^{\tau})^{fin},\ c \in \bullet IS : \#t.c = 1 \ \Rightarrow \\
&\qquad \exists\, u \in (Act.(IS^{\tau}))^{*} : t = IS^{\tau} \textcircled{s} u \ \wedge \\
&\qquad\quad f.\varepsilon = \varepsilon \ \wedge\ f.(s \circ t) = f.s \circ addtime.(scan.h.s.(Act.IS \textcircled{c} u)) \\
&\qquad \text{wobei} \quad scan.h.s.\langle\rangle = \varepsilon \\
&\qquad\qquad\quad scan.h.s.(a \,\&\, u) = h.s.a \circ scan.h.(s \circ \bar{a}).u \\
&\qquad\qquad\quad addtime.r.o = \langle\checkmark\rangle \ \text{für} \ o \in \bullet OS,\ r.o = \langle\rangle \\
&\qquad\qquad\qquad\quad\ = r.o \ \text{für} \ o \in \bullet OS,\ r.o \neq \langle\rangle
\end{aligned}
$$

□

Eine Spezifikation nach dem Schema $TStrat_H$ beschreibt im allgemeinen auch für einelementige Strategienmengen H mehrere (potentiell unendlich viele) gezeitete stromverarbeitende Funktionen, weil das Verhalten einer solchen Funktion für jede Zeitscheibe t nur einer der möglichen Sequentialisierungen von t entsprechen muß.

Beispiel 6.3: Die schwach zeitabhängige Agentenspezifikation *MixerFun* von Beispiel 6.2 erlaubt einen Mischagenten, der Nachrichten beliebig verzögert. Aus den schon in den Anmerkungen in Beispiel 6.1 angegebenen Gründen beeinflußt die Verzögerung des Mischagenten die Leistung des Gesamtsystems. Das folgende Prädikat *MixerFun'* beschreibt daher einen Mischagenten, der zumindest dann nicht verzögert, wenn in jeder Zeitscheibe höchstens eine echte Nachricht eingeht. Es gilt $MixerFun \Leftarrow MixerFun'$.

$$
\begin{aligned}
MixerFun' &\subseteq (InType_{PSysStruct'}.Mixer)^{\tau} \nrightarrow (OutType_{PSysStruct'}.Mixer)^{\tau} \\
MixerFun' &= TStrat_{\{h\}} \quad \text{wobei} \quad h.s.(c, m).MtoF = \langle m \rangle
\end{aligned}
$$

□

Eine Erfüllbarkeitsbedingung wie die von Satz 5.2 ist bei Definitionen nach dem Schema *TStrat* nicht notwendig, wie die folgende Anmerkung zeigt:

Anmerkung 6.1: Für jede Strategienspezifikation $H \subseteq STRAT_{IS,OS}$ gilt $TStrat_H \neq \varnothing$.
□

Kapitel 7

Netzwerke

Eine wesentliche Aufgabe von Betriebssystemen ist die Prozeßverwaltung, also das Starten, Beenden und koordinierte Ausführen einer Vielzahl von Prozessen. Bei der Modellierung eines Betriebssystems und der unter seiner Steuerung ausgeführten Prozesse in der vorliegenden Methodik wird sinnvollerweise jeder Prozeß durch einen Agenten dargestellt. Die Gesamtheit der kommunizierenden Prozesse entspricht einem Agentennetz. Offensichtlich reicht es jedoch zur Beschreibung von sich hinsichtlich ihrer Anzahl und ihrer Kommunikationsverbindungen dynamisch verändernden Prozessen nicht aus, lediglich eine statische Agentenstruktur zu betrachten. Wir beschäftigen uns daher in diesem Kapitel mit dynamischen Netzwerken.

Die Modellierung dynamischer Netze ist ein gegenwärtig aktuelles Forschungsgebiet. Neben der hier verfolgten Zielsetzung der Darstellung von Betriebssystemstrukturen werden dynamische Netze zum Beispiel auch zur Beschreibung von sich verändernden Hardware-Konfigurationen ausfallsicherer Systeme oder von objektorientierten Programmstrukturen benötigt.

7.1 Grundlegende Überlegungen

In dem in dieser Arbeit vorgestellten Formalismus mit benannten Kanälen lassen sich relativ einfach Netzwerke ausdrücken, bei denen die Prozessoren und der maximale Kanalraum statisch gegeben sind, aber während der Berechnung nur jeweils ein Teil davon aktiviert ist. Dazu erhält jeder Agent als Namen für die Ein- und Ausgabekanäle nicht nur die im Moment "tatsächlich" verwendeten Kanäle, sondern den gesamten für das (Teil-)System zur Verfügung stehenden Kanalnamensraum. Hinsichtlich der Ausgabekanäle muß entweder durch geeignete Nebenbedingungen sichergestellt werden, daß keine Schreibkonflikte auf gleichnamigen Kanälen auftreten, oder es wird festgelegt,

daß alle auf solchen Kanälen erfolgenden Ausgaben fair gemischt und somit zu einem einzigen Strom zusammengefügt werden.

Dieser Ansatz ist in [GSB95] ausgearbeitet. Grundlage ist ein gezeiteter Formalismus mit durch Zeitticks getrennten Zeiteinheiten. Die als *point-to-point composition* bezeichnete Verbindung entspricht unserem Operator $\|$. Eine weitere Verbindungsart, die *many-to-many composition,* setzt implizite faire Mischagenten ein. Um eine sich dynamisch verändernde Kanalstruktur zu modellieren, können Kanalnamen über schon bestehende Kanäle ausgetauscht werden. Eine Reihe von Nebenbedingungen stellt sicher, daß jeder Agent nur auf Kanäle (schreibend oder lesend) zugreift, deren Namen in seinem augenblicklich verfügbaren Kanalnamensraum vorhanden sind.

Im Modell nach [GSB95] ist die Verbindungsstruktur zwischen den Agenten veränderbar. Die Agenten selbst sind dagegen statisch gegeben. Die Prozeßverwaltung in einem realen Rechner läßt sich nur über zusätzliche Konventionen modellieren, beispielsweise über die Vereinbarung, daß ein Agent erst dann als im Speicher befindlich betrachtet wird, wenn er auf einem speziellen Kanal ein Startsymbol erhält. Überdies ist bei dieser Darstellung das grundlegende Konzept von FOCUS, die Ein- und Ausgabeanschlüsse eines Agenten als dessen "syntaktisches Interface" zu betrachten, kaum mehr erkennbar. Die Darstellung führt vielmehr zu einem Modell, in dem jeder Agent potentiell mit jedem anderen verbunden ist und der Nachrichtenaustausch über Adressen erfolgt. Dies entspricht dem *sort office* bei der funktionalen Implementierung von Betriebssystemen [Tur90] oder auch der in der Sprache *SDL* [SDL89] verwendeten Kommunikation.

In [Gro94] wird vorgeschlagen, Nachrichten höherer Ordnung zur Modellierung dynamischer Netze einzusetzen. So wird beispielsweise die Erweiterung eines Agenten um einen Eingabekanal dadurch dargestellt, daß ein diesem Eingabekanal entsprechender Strom als ein Datenelement über einen schon bestehenden Kanal übertragen wird. Dies entspricht der bereits in Abschnitt 3.6 in Zusammenhang mit der punktweisen Ordnung $\sqsubseteq$ diskutierten Idee. Die Erweiterung eines Agenten um einen Ausgabekanal wird modelliert, indem der erweiterte Agent über einen Kanal des ursprünglichen Agenten übertragen wird und dann in der Art einer *continuation* den ursprünglichen Agenten ersetzt. Dynamische Netze in dieser Modellierung weisen zwar das gewünschte funktionale Verhalten auf, die genaue Netzstruktur wird aber nicht explizit dargestellt.

Ein axiomatischer Ansatz zur Beschreibung dynamischer Netze findet sich in [Bro94a]. Ein aus einem Ein- und einem Ausgabekanal eines Agenten bestehendes Paar kann hier als eigenständiger Kanal "herausgezogen" werden, wenn der ursprüngliche Agent die Nachrichten auf dem Eingabekanal unverändert an den Ausgabekanal weitergibt.

Dynamische Netze im Kontext von Prozeßalgebren (*mobile processes*) werden im *π-Kalkül* [MPW89] behandelt. Hier können Kanalnamen zwischen Prozessen ausgetauscht werden, wodurch sich die Kommunikationsstruktur verändert. In [Tho89] ist eine Erweiterung von *CCS* mit veränderlicher Prozeßstruktur vorgestellt.

7.2 Ein einfaches Modell für dynamische Netze

Wir beschreiben in diesem Abschnitt eine abstrakte funktionale Modellierung dynamischer Netzwerke. Die Modellierung enthält genug Informationen, um daraus sowohl das statische funktionale Verhalten (die berechnete stromverarbeitende Funktion) als auch das dynamische Verhalten (die sich für eine Eingabe ergebende Folge von Netzstrukturen) eines dynamischen Netzwerks ableiten zu können. Von ihrer Grundidee her ist die Modellierung an die in Kapitel 5 vorgestellte zustandsorientierte Spezifikationstechnik angelehnt.

Ein sich dynamisch veränderndes System wird durch eine Strategie eines Zustandsautomaten dargestellt. Im einfachsten Fall entspricht jeder Zustand des Automaten einem Agentennetz. Als "Netz" bezeichnen wir in diesem Zusammenhang eine Repräsentation der zu einem Zeitpunkt existierenden Agenten und der sie verbindenden Kanalstruktur. Jede Eingabeaktion des Automaten kann zunächst einen Zustandsübergang, also eine Netztransformation, hervorrufen. Die in der Eingabeaktion enthaltene Nachricht wird dann dem gegebenenfalls durch die Transformation veränderten Agentennetz zugeleitet. Die in den einzelnen Schritten erzeugten Ausgaben ergeben zusammen die Gesamtausgabe des Systems, und die Folge der Netztransformationen spiegelt die Veränderungen der Prozeßkonfiguration während der Verarbeitung des Eingabestromes wider.

Ein dynamisches Netz hat in der hier vorgestellten Modellierung "nach außen" einen festen Kanaltyp, also statisch festgelegte Ein- und Ausgabekanäle. Die "innere", sich verändernde Agentenstruktur braucht jedoch nicht genau diese Ein- und Ausgabekanäle aufweisen, sondern kann weniger oder mehr Kanäle haben. Der erste Fall kann zum Beispiel auftreten, wenn ein Wartezustand dargestellt werden soll, bei dem das volle Agentennetz erst nach Eingang einer bestimmten Nachricht aufgebaut wird. Der zweite Fall kann während des Aufbaus eines dynamischen Netzes auftreten, wenn Ein- oder Ausgabekanäle von Teilstrukturen existieren, die erst beim weiteren Netzaufbau an andere Agenten angeschlossen und somit zu internen Kanälen werden. Als Konsistenzbedingung fordern wir allerdings, daß alle nicht-internen Kanäle, deren Namen mit Ein- und Ausgabekanälen des Gesamtnetzes übereinstimmen, mit diesen kompatibel sind, also die gleichen Nachrichtenmengen übertragen können.

Die Verbindungsstruktur innerhalb eines dynamischen Netzes wird über Kanalnamen definiert. Dies wurde schon in Abschnitt 3.3 damit begründet, daß eine Modellierung mit speziellen Kompositionsoperatoren (etwa sequentieller und paralleler Komposition sowie Rückkopplung) zusätzliche Agenten zur "Leitungsführung" benötigen würde. Solche Agenten sind im Zusammenhang mit dynamischen Netzen besonders störend, weil sie explizit als Bestandteile des Netzes auftauchen würden, was offensichtlich den realen Verhältnissen nicht entspricht.

Wir formalisieren nun die bisher informell beschriebenen Konzepte von Agentennetzen und Strategien zur Modellierung dynamischer Netzwerke. Gegeben seien zwei Kanaltypen S und S'. Wir schreiben S **cpt** S', um auszudrücken, daß S mit S' kompatibel ist. Dies ist dann der Fall, wenn alle gleichbenannten Kanäle in S und S' auch hinsichtlich der auf ihnen übertragbaren Nachrichten übereinstimmen:

$$S \textbf{ cpt } S' = \forall\, c \in \bullet S \cap \bullet S' : Str_S.c = Str_{S'}.c$$

Es sei AGT das Universum aller Agenten im Sinne von Definition 3.3, also aller stetigen Funktionen zwischen zwei Kanaltypen. Mit $AGT_{IS,OS}$ für Kanaltypen IS und OS bezeichnen wir die Menge derjenigen Agenten, deren Eingabekanäle mit IS und deren Ausgabekanäle mit OS kompatibel sind:

$$AGT_{IS,OS} = \{f \in AGT : \bullet f \textbf{ cpt } IS \ \wedge \ f\bullet \textbf{ cpt } OS\}$$

Die Charakterisierung $f \in AGT_{IS,OS}$ ist offensichtlich weniger streng als $f \in IT \twoheadrightarrow OT$, weil die erste Eigenschaft aus der zweiten folgt, die Umkehrung jedoch nicht gilt.

Ein Agentennetz modellieren wir als leere Sequenz, oder als Sequenz von Agenten, die kombinierbar im Sinne von Definition 3.4 sind und deren Kombination Ein- und Ausgabekanäle aufweist, die mit den (festgelegten) Ein- und Ausgabekanälen des Gesamtsystems kompatibel sind. Die Menge solcher Netze bezeichnen wir mit $NET_{IS,OS}$, wie in der folgenden Gleichung angegeben:

$$NET_{IS,OS} = \{\langle\rangle\} \cup \{F \in AGT^{\omega} : \{F.n : n \in \mathbb{N},\, n < \#F\} \text{ ist kombinierbar} \wedge \|_{n\in\mathbb{N}, n<\#F}\ F.n \in AGT_{IS,OS}\}$$

Die Darstellung eines Netzes als Agentensequenz (und nicht etwa als Agentenmenge) hat den Vorteil, daß die in dieser Arbeit in Kapitel 2 eingeführten Funktionen auf Sequenzen auch zur Spezifikation dynamischer Netze verwendet werden können. Beispielsweise ergibt sich gemäß der Definition in Abschnitt 2.2 das $(n+1)$-te Element einer Sequenz F durch die Funktionsapplikation $F.n$. Die einzelnen Agenten, also die Elemente der Sequenz, können mit den in den bisherigen Kapiteln vorgestellten Mitteln definiert und modifiziert werden; so kann die Umbenennung von Kanälen etwa durch die in Abschnitt 3.2 angegebenen Funktionen erfolgen.

In der hier angegebenen Technik der dynamischen Modellierung wird das System durch einen Zustandsautomaten beschrieben. Die intuitive Bedeutung der Berechnungsschritte dieses Automaten ist in diesem Abschnitt bereits eingangs dargestellt worden. Im einfachsten, hier zunächst betrachteten Fall entspricht jeder Zustand des Automaten einem Netz. Demgemäß ist die Strategie ein Element der im folgenden definierten Menge $NSTRAT_{IS,OS}$, also eine Abbildung, die zu einem Netz und einer Eingabeaktion das gegebenenfalls veränderte Netz liefert, das zur Verarbeitung der Eingabeaktion dienen soll:

$$NSTRAT_{IS,OS} = NET_{IS,OS} \to Act.IS \to NET_{IS,OS}$$

Die Beschreibung eines dynamischen Netzes mit nach außen sichtbarem Eingabekanaltyp *IS* und Ausgabekanaltyp *OS* ist ein Paar aus einer Menge von Anfangszuständen $I \subseteq NET_{IS,OS}$ und einer Strategienmenge $H \subseteq NSTRAT_{IS,OS}$. Vor der Formalisierung des funktionalen Verhaltens eines so beschriebenen Netzes in Definition 7.1 soll in den folgenden drei Beispielen gezeigt werden, daß sich der Formalismus zur Modellierung praxistypischer Anwendungsfälle eignet.

Beispiel 7.1 (Umsetzung eines programmiersprachlichen Konstrukts):
Als erstes Anwendungsbeispiel betrachten wir das await-Konstrukt der in [BDS93] angegebenen Programmiersprache *HOPSA*. Eine Agentendefinition der Form $g(x) =$ await x then $f(x)$ bedeutet hier, daß beim Programmablauf ein Agent f erst dann im Speicher angelegt wird, wenn das erste Datenelement auf dem Kanal x ankommt. Nachdem dies geschehen ist, verarbeitet der Agent f als von nun an statisches Netz die ankommenden Eingaben. Wenn wir $f \in IS \twoheadrightarrow OS$ annehmen, entspricht die obige *HOPSA*-Definition der dynamischen Modellierung mit jeweils einelementigen Mengen von Anfangszuständen $I = \{\langle\rangle\}$ und Strategien $H = \{h\}$, wobei h definiert ist vermöge:

$$\begin{aligned} h &\in NSTRAT_{IS,OS} \\ h.F.a &= \langle f \rangle \quad \text{wenn} \quad F = \langle\rangle \\ &= F \quad \text{sonst} \end{aligned}$$

□

Beispiel 7.2 (Sortiernetzwerk): Ein weiteres Beispiel ist die dynamische Modellierung eines Sortiernetzwerks, wie es aus [Bro88b] bekannt ist. Das Netz besteht aus einer sich dynamisch verändernden Reihe von Sortierzellen mit je zwei Ein- und zwei Ausgängen. Jede Zelle speichert ein Datenelement. Ein neu ankommendes Datenelement wird so lange von Zelle zu Zelle weitergereicht, bis es kleiner als ein gespeichertes Element ist. Die entsprechende Zelle vertauscht die beiden Elemente und leitet das ursprünglich gespeicherte weiter. Dieses Verfahren wird bis zu einer das Sortiernetz abschließenden leeren Zelle fortgesetzt, die das sie erreichende kleinste Element speichert.

Parallel zum Einsortieren eines neuen Datenelementes gibt das Netz eine sortierte Folge aller bisher eingelesenen Daten aus, die mit dem Sonderzeichen ↲ abgeschlossen wird. Dazu gibt jede Zelle, wenn sie ein Element von ihrem Eingang i zum Ausgang o' weiterreicht, das (gegebenenfalls vertauschte) gespeicherte Element an den Ausgang o aus und leitet dann Daten, die am Eingang i' erscheinen, ebenfalls an den Ausgang o weiter, so lange, bis das Abschlußzeichen ↲ am Eingang i' erkannt wird. Dieses Zeichen ist von der letzten, ursprünglich leeren Zelle des Netzes erzeugt worden, nachdem sie das am Eingang i eingetroffene Datenelement gespeichert und auf den Ausgang o gegeben hat.

Wir definieren nun den Agenten *cell* durch drei sich rekursiv aufeinander abstützende Funktionen, die jeweils zur Verbindung mit dem vorhergehenden Agenten die Ein- und Ausgänge i und o und zur Verbindung mit dem folgenden Agenten die Ein- und

Ausgänge i' und o' aufweisen. Die Funktion *cell* entspricht der das Sortiernetzwerk abschließenden Zelle, in der noch kein Datenelement gespeichert ist. Die Funktionen $store_m$ und $copy_m$ entsprechen Zellen, die das Element m gespeichert haben. In den folgenden Definitionen sei $\mathbb{N}_{\hookleftarrow} = \mathbb{N} \cup \{\hookleftarrow\}$. Für die hier nicht ausdrücklich angegebenen Fälle erzeugen die Funktionen keine Ausgaben:

$$
\begin{aligned}
&cell \;\in\; (i{:}\mathbb{N}_{\hookleftarrow}{}^{\omega} \otimes\; i'{:}\mathbb{N}_{\hookleftarrow}{}^{\omega}) \twoheadrightarrow (o{:}\mathbb{N}_{\hookleftarrow}{}^{\omega} \otimes\; o'{:}\mathbb{N}_{\hookleftarrow}{}^{\omega})\\
&cell.(\varepsilon[i \mapsto \langle n\rangle] \circ x) \;=\; \varepsilon[o \mapsto \langle n, \hookleftarrow\rangle] \circ store_n.x \quad \text{für} \quad n \in \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
&store \;\in\; \mathbb{N} \to (i{:}\mathbb{N}_{\hookleftarrow}{}^{\omega} \otimes\; i'{:}\mathbb{N}_{\hookleftarrow}{}^{\omega}) \twoheadrightarrow (o{:}\mathbb{N}_{\hookleftarrow}{}^{\omega} \otimes\; o'{:}\mathbb{N}_{\hookleftarrow}{}^{\omega})\\
&store_m.(\varepsilon[i \mapsto \langle n\rangle] \circ x)\\
&\quad = ((o \mapsto \langle \min(n,m)\rangle) \cup (o' \mapsto \langle \max(n,m)\rangle)) \circ copy_{\min(n,m)}.x \quad \text{für} \quad n \in \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
&copy \;\in\; \mathbb{N} \to (i{:}\mathbb{N}_{\hookleftarrow}{}^{\omega} \otimes\; i'{:}\mathbb{N}_{\hookleftarrow}{}^{\omega}) \twoheadrightarrow (o{:}\mathbb{N}_{\hookleftarrow}{}^{\omega} \otimes\; o'{:}\mathbb{N}_{\hookleftarrow}{}^{\omega})\\
&copy_m.(\varepsilon[i' \mapsto \langle \hookleftarrow\rangle] \circ x) \;=\; \varepsilon[o \mapsto \langle \hookleftarrow\rangle] \circ store_m.x\\
&copy_m.(\varepsilon[i' \mapsto \langle n\rangle] \circ x) \;=\; \varepsilon[o \mapsto \langle n\rangle] \circ copy_m.x \quad \text{für} \quad n \in \mathbb{N}
\end{aligned}
$$

Das dynamische Sortiernetzwerk ist mit der Umgebung durch einen Eingang u_0 und einen Ausgang v_0 verbunden. Es weist im Initialzustand keine Sortierzellen auf. Für jedes auf dem Kanal u_0 neu ankommende Zeichen wird dem Netzwerk eine Zelle *cell* hinzugefügt. Außerdem ist es möglich, das gesamte Sortiernetzwerk zu löschen, indem eine Nachricht $\hookleftarrow$ auf dem Kanal u_0 gesendet wird.

Für die Formalisierung nehmen wir an, daß die Kanalnamen u_i und v_i für jedes $i \in \mathbb{N}$ zur Verfügung stehen. Die Ein- und Ausgabekanaltypen des Sortiernetzwerkes sind $IS = \otimes\, u_0 : \mathbb{N}_{\hookleftarrow}{}^{\omega}$ und $OS = \otimes\, v_0 : \mathbb{N}_{\hookleftarrow}{}^{\omega}$. Dann entspricht das Sortiernetzwerk der dynamischen Modellierung mit jeweils einelementigen Mengen von Anfangszuständen $I = \{\langle\rangle\}$ und Strategien $H = \{h'\}$, wobei h' definiert ist vermöge:

$$
\begin{aligned}
&h' \;\in\; NSTRAT_{IS,OS}\\
&h'.F.(u_0, \hookleftarrow) \;=\; \langle\rangle\\
&h'.F.(u_0, n) \;=\; ([u_{\#F}/i][v_{\#F+1}/i']cell[v_{\#F}/o][u_{\#F+1}/o'])\,\&\,F \quad \text{für} \quad n \in \mathbb{N} \qquad \square
\end{aligned}
$$

Beispiel 7.3 (Betriebssystem): Als drittes Beispiel betrachten wir einen an das Beispiel in Abschnitt 3.6 angelehnten Ausschnitt aus dem *UNIX*-Betriebssystem. Das dynamische Netzwerk erhält einen Strom von Programmen, Daten und Steuerzeichen auf dem Kanal *stdin*. Dieser Strom hat die folgende schematische Form für Programme $pr_1, pr_1', pr_2, pr_2', pr_3, \ldots$ und Daten $d_1, d_2, \ldots$:

$$\langle pr_1, pr_2, pr_3, \ldots, \hookleftarrow, d_1, d_2, \ldots, \mathbf{eof}, pr_1', pr_2', \ldots\rangle$$

Die Eingabe an das Netz besteht also abwechselnd aus Kommando- und Datenzeilen, die durch die Zeichen $\hookleftarrow$ beziehungsweise **eof** abgeschlossen sind. Die in einer Kommandozeile enthaltenen Programme sollen in Form einer *UNIX-pipe* hintereinandergeschaltet werden, es soll also jeweils der Standardausgang eines Programmes

Bild 7.1: Netzzustand nach dem Einlesen von $\langle pr_1, \ldots \rangle$

Bild 7.2: Netzzustand nach dem Einlesen von $\langle pr_1, pr_2, \ldots \rangle$

Bild 7.3: Netzzustand nach dem Einlesen von $\langle pr_1, pr_2, pr_3, \ldots \rangle$

Bild 7.4: Netzzustand nach dem Einlesen von $\langle pr_1, pr_2, pr_3, \lrcorner, \ldots \rangle$

mit dem Standardeingang des folgenden Programmes verbunden werden. Die an den Fehlerausgängen der Programme erscheinenden Ausgaben sollen gemischt auf einem gemeinsamen Fehlerkanal erscheinen. So soll etwa für die Befehlszeile $\langle pr_1, pr_2, pr_3, \downarrow, \ldots \rangle$ das in den Bildern 7.1 bis 7.4 gezeigte Netz schrittweise aufgebaut werden, wobei der Agent mg_3 einen Mischknoten mit drei Eingängen darstellt. Das in Bild 7.4 gezeigte Netz bearbeitet die auf die Befehlszeile folgende Datenzeile und wird beim Einlesen des diese Zeile beendenden **eof** wieder gelöscht.

Dieses informell beschriebene Verhalten wird durch die dynamische Modellierung mit einer einelementigen Menge $I = \{\langle\rangle\}$ von Startzuständen und einer Strategienmenge $H = \{h\}$ formalisiert, wobei h durch die Tabellennotation von Bild 7.5 gegeben ist. Wir nehmen dabei an, daß die Kanalnamen $link_i$ und $stderr_i$ für jedes $i \in \mathbb{N}$ zur Verfügung stehen.

Der Agent mg_n mischt die an den Eingängen $stderr_1, \ldots, stderr_n$ ankommenden Fehlermeldungen zeilenweise und gibt sie an den Ausgang $stderr$ weiter. Um Monotonieprobleme zu umgehen, nehmen wir an, daß die einzelnen Programme für jedes eingehende Datenelement an ihren Ausgängen $stdout_i$ und $stderr_i$ jeweils genau eine Ausgabe (gegebenenfalls eine leere Pseudo-Nachricht) erzeugen. Natürlich kann statt eines Mischagenten mit n Eingängen auch ein dynamisches Mischnetz vorgesehen werden, dessen Grundbausteine Agenten mit je zwei Eingängen sind. □

Wir führen nun einige Hilfsoperationen ein, die in der unten angegebenen Definition 7.1 benötigt werden. Gegeben seien zwei Kanaltypen S und S', die kompatibel sind, für die also S **cpt** S' gilt. Die Funktion adj_S wandelt einen Kanalzustand des Typs S' in den entsprechenden Kanalzustand des Typs S um. Kanäle, die in S', nicht aber in S vorhanden sind, werden entfernt. Kanäle, die in S, nicht aber in S' vorhanden sind, erhalten den leeren Strom $\langle\rangle$ als Belegung:

$$\begin{aligned} adj_S &\in S' \twoheadrightarrow S \\ adj_S.x &= x|_{\bullet S \cap \bullet S'} \cup \varepsilon|_{\bullet S \setminus \bullet S'} \end{aligned}$$

Weiter benötigen wir einen Resumptionsoperator, der gegenüber dem in Abschnitt 3.2 definierten auf kombinierbare Agentensequenzen erweitert ist. Gegeben sei eine Agentensequenz $F \in NET_{IS,OS}$, so daß die Agenten $F.n$ für $n \in \mathbb{N}$ mit $n < \#F$ kombinierbar sind und jeweils Nachrichten erster Ordnung verarbeiten. Wir schreiben von nun an vereinfachend $\|F$ statt $\|_{n \in \mathbb{N}, n < \#F}\ F.n$.

Es gelte $\|F \in IT \twoheadrightarrow OT$. Für jedes $s \in IT$ ist $F \ll s$ diejenige Agentensequenz, die sich ergibt, nachdem das Netz F die Eingaben s verarbeitet und die Ausgaben $(\|F).s$ erzeugt hat. Um diese Sequenz formal zu definieren, wird in der folgenden Gleichung für jeden Agenten $F.n$ aus F die Resumption von $F.n$ nach einer Eingabe gebildet, die sich zusammensetzt aus der in s enthaltenen Belegung der von außen zugänglichen Eingabekanäle von $F.n$ und der in einem Wert v enthaltenen Belegung der internen

rückgekoppelten Eingabekanäle von $F.n$. Der Kanalzustand v ist unter Verwendung des Operators $\hat{\|}$ aus Definition 3.4 bestimmt. Der Wert $\bullet\bullet f$ bezeichnet die Menge der Eingabekanäle eines Agenten f:

$$\begin{array}{l} \#(F \ll s) = \#F \ \wedge \\ \forall\, n \in \mathbb{N} : n < \#F \ \Rightarrow (F \ll s).n = F.n \ll (s|_{\bullet\bullet F.n \setminus \bullet OT} \cup v|_{\bullet\bullet F.n}) \\ \qquad\qquad\qquad\qquad \text{wobei} \ \ v = (\hat{\|}F).(adj_{IT}.s) \ \ \text{für} \ \ \hat{\|}F \in IT \twoheadrightarrow OT \end{array}$$

Der Operator $\ll$ hat die intuitiv erwartete Eigenschaft, die Resumption einer Agentensequenz zu berechnen, wie im folgenden Beispiel 7.4 sowie in der Anmerkung 7.1 zum Ausdruck kommt:

Beispiel 7.4: Gegeben sei das aus den Agenten *store* und *cell* von Beispiel 7.2 gebildete Sortiernetzwerk $F = \langle([v/i']store_5[u/o']), ([u/i]cell[v/o])\rangle$. Dann gilt:

$$F \ll (i \mapsto \langle 7\rangle) \ = \ \langle([v/i']store_7[u/o']), ([u/i]store_5[v/o])\rangle \qquad \square$$

Anmerkung 7.1: Für jede kombinierbare Agentensequenz $F \in NET_{IS,OS}$ mit $\|F \in IT \twoheadrightarrow OT$ gilt:

$$\begin{array}{l} \forall\, s \in IT : \|(F \ll s) \ = \ (\|F) \ll s \\ \forall\, s, s' \in IT : (F \ll s) \ll s' \ = \ F \ll (s \circ s') \end{array} \qquad \square$$

Wie schon oben in diesem Abschnitt beschrieben, ist die intuitive Vorstellung vom funktionalen Verhalten eines dynamischen Netzes, daß jede Eingabeaktion zunächst gemäß der das Netz modellierenden Strategie eine Netztransformation hervorrufen kann und dann dem gegebenenfalls veränderten Netz zugeführt wird. Die Ausgaben dieser einzelnen Berechnungsschritte ergeben zusammen die Ausgabe des Gesamtsystems. Diese Vorstellung spiegelt sich in der folgenden Definition 7.1 wider:

Definition 7.1 (Funktionales Verhalten dynamischer Netze): Gegeben seien Kanaltypen IS und OS, ein Prädikat $H \subseteq NSTRAT_{IS,OS}$ sowie eine Menge $I \subseteq NET_{IS,OS}$ von Anfangszuständen. Das funktionale Verhalten des durch H und I modellierten dynamischen Netzes ist durch die Funktion $NStrat_{H,I}$ gegeben:

$$\begin{array}{l} NStrat_{H,I} \ \subseteq \ IS \twoheadrightarrow OS \\ NStrat_{H,I}.f \\ \quad = \exists\, h \in H,\ i \in I,\ g \in NET_{IS,OS} \to IS \twoheadrightarrow OS : \\ \qquad \forall\, F \in NET_{IS,OS},\ x \in IS,\ a \in Act.IS : \\ \qquad\quad f = g_i \ \wedge \\ \qquad\quad g_F.\varepsilon = \varepsilon_{OS} \ \wedge \\ \qquad\quad g_F.(\bar{a} \circ x) = adj_{OS}.((\|h.F.a).(adj_{IT}.\bar{a})) \circ g.(h.F.a \ll adj_{IT}.\bar{a}).x \\ \qquad\quad \text{für} \ \ \|h.F.a \in IT \twoheadrightarrow OT \end{array} \qquad \square$$

Ähnlich wie in Abschnitt 5.6 dargelegt, muß auch hier bei der Implementierung einer Strategie sichergestellt werden, daß die Implementierung nur solche Verhalten zeigt, die einem zulässigen funktionalen Verhalten eines Agenten entsprechen. Dies heißt insbesondere, daß vor der Implementierung im allgemeinen $NStrat_{H,I} \neq \emptyset$ gezeigt werden muß.

Eine Strategie in dem hier vorgestellten Formalismus erlaubt beliebige Netztransformationen in einem Schritt, also beliebige Übergänge von F zu $h.F.a$ beim Einlesen einer Aktion a. Abhängig von den in der Zielsprache bereitgestellten Prozeßverwaltungsmechanismen ist oft eine Beschränkung der zulässigen Netztransformationen angemessen. So wurden in den Beispielen dieses Kapitels nur Operatoren verwendet, die einem Netz einen weiteren Agenten hinzufügen, einzelne Kanäle eines Agenten umbenennen oder das gesamte Netz löschen.

Bei der hier vorgeschlagenen dynamischen Modellierung ist typischerweise jeweils nur ein kleiner Teil des Zustandsraumes erreichbar. Daher ist es für die praktische Anwendbarkeit des Formalismus wichtig, Techniken bereitzustellen, die eine bequeme Spezifikation einer Strategie nur für den erreichbaren Teil des Zustandsraumes erlauben.

7.3 Erweiterte dynamische Netzmodellierungen

Der Formalismus aus Abschnitt 7.2 ist auf eine möglichst einfache Darstellung dynamischer Netze bei hinreichender Ausdrucksstärke zugeschnitten. In diesem Abschnitt geben wir einige erweiterte Strategienbegriffe an, die sich für die Modellierung von Systemen eignen, die entweder komplexer aufgebaut sind oder ein komplexeres dynamisches Verhalten zeigen.

Zunächst bietet es sich an, den Zustandsraum durch zusätzliche Komponenten zu erweitern. Dies ist zur Darstellung von Systemen erforderlich, deren Reaktion auf eine Eingabeaktion nicht nur von der augenblicklichen Netzstruktur, sondern auch von der Vorgeschichte abhängig ist. Ein weiterer Grund für die Verwendung eines erweiterten Zustandsraumes ist die Beobachtung, daß in realen Systemen der Netzzustand typischerweise nicht zur Steuerung des Programmablaufes herangezogen wird.

Aus diesen Gründen ist es oft eine realitätsnähere Modellierung, einen expliziten Zustandsraum zu verwenden, der eine Netzzustandskomponente und mindestens eine Programmzustandskomponente aufweist. Der Netzzustand beeinflußt dabei den Folgezustand nur insofern, als auf ihn gegebenenfalls eine schematische Netztransformation angewendet wird, beispielsweise das Anhängen eines weiteren Agenten. Der Programmzustand steuert, ob und welche Netztransformation in einem Berechnungsschritt stattfindet.

Eine Strategie für einen in diesem Sinne erweiterten Zustandsraum ST mit einer Zustandskomponente aus $NET_{IS,OS}$ ist ein Element aus der Menge $NSTRAT'_{ST,IS,OS}$:

$$NSTRAT'_{ST,IS,OS} = ST \to Act.IS \to ST$$

Weiter kann die dem Netz zugeleitete Eingabe durch die Strategie bestimmt werden. Dadurch wird es möglich, Eingaben an den Prozeßwechselmechanismus von Eingaben an die Agentennetze zu unterscheiden. Die intuitive Vorstellung ist, daß eine Strategie $h \in NSTRAT''_{IS,OS}$ Systeme beschreibt, die bei einer Eingabeaktion a von einem Netzzustand F in einen Netzzustand $h.F.a.net$ übergehen, dem die Eingabe $h.F.a.in$ zugeführt wird:

$$NSTRAT''_{IS,OS} = NET_{IS,OS} \to Act.IS \to (in : IS \otimes net : NET_{IS,OS})$$

In den Beispielen 7.2 und 7.3 haben wir als Netztransformation auch das vollständige Löschen eines Netzes in einem Schritt eingesetzt. Wenn ein System dargestellt werden soll, in dem derartige "Sprungstellen" nicht auftreten, ist es erforderlich, daß eine einzige Eingabeaktion eine Folge von Transformationsschritten anstoßen kann. Zur Spezifikation solcher Systeme dient der erweiterte Strategienbegriff $NSTRAT'''_{IS,OS}$:

$$NSTRAT'''_{IS,OS} = NET_{IS,OS} \to Act.IS \to (NET_{IS,OS})^{\omega}$$

Eine Einschränkung unseres einfachen Strategienbegriffes ist, daß ein Transformationsschritt immer nur stattfindet, *bevor* die Eingabeaktion verarbeitet wird. So ist es etwa nicht möglich, ein dynamisches Netzwerk ähnlich dem aus Beispiel 7.2 zu definieren, bei dem *nach* Erhalt eines Sonderzeichens zunächst die sortierte Sequenz der gespeicherten Zahlen ausgegeben und dann das Netz gelöscht wird. Zur Beschreibung derartiger Systeme muß ein Strategienbegriff gewählt werden, der einen Transformationsschritt *vor* und einen Transformationsschritt *nach* der Verarbeitung einer Eingabeaktion zuläßt, beispielsweise ein Strategienbegriff nach dem Schema $NSTRAT''''_{IS,OS}$:

$$NSTRAT''''_{IS,OS} = NET_{IS,OS} \to Act.IS \to (pre : NET_{IS,OS} \otimes post : NET_{IS,OS})$$

Schließlich ist es in manchen Fällen wünschenswert, hierarchische Netzstrukturen zu definieren. Diese Technik wurde in [Bro88b] zur Beschreibung eines Sortiernetzes verwendet. Sie vermeidet Namenskonflikte von Kanälen und kann die Struktur eines Netzes besonders gut zum Ausdruck bringen. Ein hierarchisches Netz ist ein Element aus NET', also entweder ein Agent oder eine Sequenz von (wiederum hierarchischen) Netzen. Ein hierarchisches Netz $F \in NET'$ ist dann mit den Kanaltypen IS und OS kompatibel (geschrieben $F \in NET'_{IS,OS}$), wenn seine Kombination (über einen hier nicht definierten Operator $\|$) mit diesen Kanaltypen kompatibel ist. Eine Strategie auf mit IS und OS kompatiblen hierarchischen Netzen ist ein Element aus $NSTRAT'''''_{IS,OS}$:

$$NET' = AGT \cup (NET')^{\omega}$$
$$NET'_{IS,OS} = \{F \in NET' : \| F \in AGT_{IS,OS}\}$$
$$NSTRAT'''''_{IS,OS} = NET'_{IS,OS} \to Act.IS \to NET'_{IS,OS}$$

7.4 Methodische Einordnung

Methodisch ist die Verwendung einer dynamischen Netzmodellierung ein relativ implementierungsnahes Konzept. Im Rahmen der in dieser Arbeit vorgeschlagenen Entwurfsmethodik ist diese Modellierung zwischen funktionalen Spezifikationen und konkreten Programmiersprachen angesiedelt, wie Bild 7.6 veranschaulicht:

- funktionale Agentenspezifikation

  Transformation nach Satz 7.1 oder Satz 7.2

- abstrakte Modellierung

  Verfeinerung nach Satz 7.3

- verfeinerte abstrakte Modellierung

  weitere Umformungen

- konkrete Programmiersprache

Bild 7.6: Einordnung der abstrakten Modellierung

Wegen seiner Einordnung zwischen funktionaler Spezifikation und konkreter Programmiersprache eignet sich der Formalismus dynamischer Netze als Zwischenschritt bei der systematischen Programmentwicklung. Außerdem kann der Formalismus als Mittel zur semantischen Darstellung programmiersprachlicher Konstrukte dienen, insbesondere solcher, durch die Prozesse erzeugt, beeinflußt oder gelöscht werden. Dies wurde in Beispiel 7.1 gezeigt.

Wir befassen uns nun mit der Frage nach einer angemessenen methodischen Vorgehensweise für den Programmentwurf. Zunächst gehen wir davon aus, daß als Ergebnis der in den bisherigen Kapiteln beschriebenen Entwicklungsschritte eine funktionale Agentenspezifikation in Form einer kombinierbaren Agentensequenz $G \in NET_{IS,OS}$ vorhanden ist. Wegen der engen Verzahnung des funktionalen Formalismus mit der dynamischen Modellierung ist der Ebenenübergang unproblematisch, wie der folgende Satz 7.1 zeigt:

Satz 7.1 (Verfeinerung von statischem Netz zu Schema NStrat):
Gegeben sei eine kombinierbare Agentensequenz $G \in NET_{IS,OS}$ mit $\|G \in IS \twoheadrightarrow OS$ und $(\|G).\varepsilon = \varepsilon$. Die Strategie $h' \in NSTRAT_{IS,OS}$ sei definiert vermöge: $h'.F.a = F$ für alle $F \in NET_{IS,OS}$ und $a \in Act.IS$. Dann gilt:

$$\{\|G\} \Leftarrow NStrat_{\{h'\},\{G\}}$$

Beweis: Der Beweis stützt sich auf den unten angegebenen Satz 7.2. Wegen der in Anmerkung 7.1 angegebenen Eigenschaften des Resumptionsoperators ist $\{\|G\}$ äquivalent

zu einer zustandsorientierten Spezifikation $EStrat_{\{h\},\{\varepsilon\}}$ mit Zustandsmenge $ST = IS$ und Strategie $h \in ESTRAT_{ST,IS,OS}$, die definiert ist vermöge:

$$\begin{aligned} h.s.a.out &= (\|(G \ll s)).\bar{a} \\ h.s.a.next &= s \circ \bar{a} \end{aligned}$$

Die Relation $\sim \subseteq ST \times NET_{IS,OS}$ sei definiert durch $s \sim F = (F = G \ll s)$ für alle $s \in ST$ und $F \in NET_{IS,OS}$. Gegeben seien beliebige $s \in ST$ und $F \in NET_{IS,OS}$ mit $s \sim F$. Es ist zu bemerken, daß die Ein- und Ausgabekanaltypen von $\|G$, $\|(G \ll s)$, $\|F$ und $\|h'.F.a$ übereinstimmen. Wir bezeichnen sie mit IT beziehungsweise OS $(*)$. Dann gilt für die Anfangszustände ε und G zunächst $\varepsilon \sim G \ll \varepsilon$ nach der Definition von $\sim$ und weiter $G \ll \varepsilon = G$ wegen $(\|G).\varepsilon = \varepsilon$, also insgesamt $\varepsilon \sim G$. Für die Strategien h und h' gilt:

$$\begin{aligned} h.s.a.out &= (\|(G \ll s)).\bar{a} && \text{[Definition von } h\text{]} \\ &= (\|F).\bar{a} && \text{[wegen } s \sim F\text{]} \\ &= (\|h'.F.a).\bar{a} && \text{[Definition von } h'\text{]} \\ &= adj_{OS}.((\|h'.F.a).(adj_{IT}.\bar{a})) && \text{[wegen } (*)\text{]} \end{aligned}$$

$$\begin{aligned} h.s.a.next &= s \circ \bar{a} && \text{[Definition von } h\text{]} \\ &\sim G \ll (s \circ \bar{a}) && \text{[Definition von } \sim\text{]} \\ &= (G \ll s) \ll \bar{a} && \text{[Anmerkung 7.1]} \\ &= F \ll \bar{a} && \text{[wegen } s \sim F\text{]} \\ &= h'.F.a \ll \bar{a} && \text{[Definition von } h'\text{]} \\ &= h'.F.a \ll adj_{IT}.\bar{a} && \text{[wegen } (*)\text{]} \end{aligned}$$

Die Bedingungen von Satz 7.2 sind damit erfüllt. Insgesamt erhalten wir $\{\|G\} = EStrat_{\{h\},\{\varepsilon\}} \Leftarrow NStrat_{\{h'\},\{G\}}$. □

Die Aussage von Satz 7.1 läßt sich sogar verstärken zu $\{\|G\} = NStrat_{\{h'\},\{G\}}$, denn für die im Satz definierte Strategie h' ist $|NStrat_{\{h'\},\{G\}}| = 1$. Für den in diesem Abschnitt behandelten Übergang zwischen einer allgemeinen Agentenspezifikation und einer dynamischen Modellierung reicht die Aussage des Satzes 7.1 jedoch aus.

Die durch Anwendung von Satz 7.1 entstandene Spezifikation nach dem Schema *NStrat* ist zwar formal eine dynamische Modellierung, sie ist aber nicht im eigentlichen Sinne "dynamisch", da die Struktur des dargestellten Netzes für alle Eingaben unverändert bleibt. In der weiteren Entwicklung wird daher die Strategie h dahingehend modifiziert, daß "Sprungstellen" zu neuen Netzstrukturen eingefügt werden. Natürlich muß dann gezeigt werden, daß das über *EStrat* definierte funktionale Verhalten des modifizierten Netzes eine Verfeinerung des ursprünglichen Verhaltens ist. Dies kann beispielsweise über den unten angegebenen Satz 7.3 geschehen.

Bevor wir jedoch auf die Verfeinerung innerhalb des Formalismus dynamischer Modellierungen eingehen, zeigen wir eine alternative Möglichkeit des Ebenenüberganges von

funktionalen Agenten zu dynamischen Modellierungen. Hier ist eine zustandsorientierte Spezifikation nach dem Schema *EStrat* der Ausgangspunkt. Der folgende Satz 7.2 beruht auf den in Abschnitt 5.5 dargestellten Techniken:

Satz 7.2 (Verfeinerung von Schema EStrat zu Schema NStrat): Gegeben seien Kanaltypen IS und OS sowie ein Zustandsraum ST. Weiterhin seien Mengen $H \subseteq ESTRAT_{ST,IS,OS}$ und $H' \subseteq NSTRAT_{IS,OS}$ von Strategien sowie Mengen $I \subseteq ST$ und $I' \subseteq NET_{IS,OS}$ von Anfangszuständen gegeben. Für alle $h' \in H'$ und $i' \in I'$ gebe es eine Strategie $h \in H$, einen Anfangszustand $i \in I$ und ein Prädikat $\sim \subseteq ST \times NET_{IS,OS}$, so daß gilt:

$$
\begin{array}{l}
i \sim i' \wedge \\
\forall\, s \in ST,\ F \in NET_{IS,OS},\ a \in Act.IS : \\
\quad s \sim F \ \Rightarrow\ h.s.a.out = adj_{OS}.((\|h'.F.a).(adj_{IT}.\bar{a})) \wedge \\
\qquad\qquad h.s.a.next \sim h'.F.a \ll adj_{IT}.\bar{a} \ \text{ für } \ \|h'.F.a \in IT \twoheadrightarrow OT
\end{array}
$$

Dann gilt $EStrat_{H,I} \Leftarrow NStrat_{H',I'}$.

Beweis: Der Beweis erfolgt analog zu dem Beweis von Satz 5.8, basierend auf der Beobachtung, daß eine Spezifikation $NStrat_{\{h'\},\{i'\}}$ äquivalent ist zu einer Spezifikation $EStrat_{\{\hat{h}\},\{\hat{\imath}\}}$ mit $\widehat{ST} = NET_{IS,OS}$, $\hat{\imath} = i'$ und $\hat{h} \in ESTRAT_{\widehat{ST},IS,OS}$ definiert vermöge:

$$
\begin{array}{l}
\hat{h}.s.a.out \ = \ adj_{OS}.((\|h'.s.a).(adj_{IT}.\bar{a})) \\
\hat{h}.s.a.next \ = \ h'.s.a \ll adj_{IT}.\bar{a} \ \text{ für } \ \|h'.s.a \in IT \twoheadrightarrow OT
\end{array}
$$

□

Beispiel 7.5: Gegeben sei ein durch das Schema *EStrat* definierter Sortieragent, dessen expliziter Zustand die sortierte Sequenz der bisher eingelesenen Daten ist, dessen Zustandsraum also durch $ST = \mathbb{N}^*$ gegeben ist. Um zu zeigen, daß das Sortiernetzwerk $NStrat_{\{h'\},\{\langle\rangle\}}$ aus Beispiel 7.2 diesen Agenten implementiert, können die Beweisbedingungen von Satz 7.2 für die folgende Relation $\sim \subseteq ST \times NET_{IS,OS}$ gezeigt werden:

$$
\begin{array}{l}
s \sim F \ = \ (\#s = \#F) \wedge \\
\qquad\qquad \forall\, n \in \mathbb{N} : n < \#s \ \Rightarrow\ F.n = [u_n/i][v_{n+1}/i']store_{s.n}[v_n/o][u_{n+1}/o']
\end{array}
$$

□

Im Gegensatz zu Satz 7.1, bei dem der Ebenenübergang nur zu einer sehr eingeschränkten dynamischen Modellierung möglich war, erlaubt Satz 7.2 den Übergang zu Modellierungen, die eine echte dynamische Veränderung der Netzstruktur aufweisen. Das diesem Satz zugrundeliegende Prinzip läßt sich jedoch nicht nur für den Ebenenübergang einsetzen, sondern auch für Verfeinerungsschritte, die innerhalb der Ebene der dynamischen Modellierung stattfinden. Dies zeigt der dieses Kapitel abschließende Satz 7.3:

Satz 7.3 (Verfeinerung von dynamischen Modellierungen): Gegeben seien Kanaltypen IS und OS und Mengen $H \subseteq NSTRAT_{IS,OS}$ und $H' \subseteq NSTRAT_{IS,OS}$ von Strategien sowie Mengen $I \subseteq NET_{IS,OS}$ und $I' \subseteq NET_{IS,OS}$ von Anfangszuständen. Für alle

$h' \in H'$ und $i' \in I'$ gebe es eine Strategie $h \in H$, einen Anfangszustand $i \in I$ und ein Prädikat $\sim \subseteq NET_{IS,OS} \times NET_{IS,OS}$, so daß gilt:

$$
\begin{array}{l}
i \sim i' \wedge \\
\forall\, F, F' \in NET_{IS,OS},\ a \in Act.IS : \\
\quad F \sim F' \Rightarrow\ adj_{OS}.((\|h.F.a).(adj_{IT}.\bar{a})) = adj_{OS}.((\|h'.F'.a).(adj_{IT'}.\bar{a})) \wedge \\
\qquad\qquad h.F.a \ll adj_{IT}.\bar{a} \sim h'.F'.a \ll adj_{IT'}.\bar{a} \\
\qquad\qquad \text{für}\ \ \|h.F.a \in IT \twoheadrightarrow OT\ \ \text{und}\ \ \|h'.F'.a \in IT' \twoheadrightarrow OT'
\end{array}
$$

Dann gilt $NStrat_{H,I} \Leftarrow NStrat_{H',I'}$.

Beweis: Der Beweis erfolgt über die im Beweis von Satz 7.2 dargestellte Umwandlung einer Spezifikation nach dem Schema *NStrat* in eine Spezifikation nach dem Schema *EStrat* analog zu dem Beweis von Satz 5.8. □

Kapitel 8

Zusammenfassung und Ausblick

In der vorliegenden Arbeit haben wir Methoden und Techniken der funktionalen Systemspezifikation dargestellt. Wir hatten uns dabei zum Ziel gesetzt, einen Beitrag zur Konsolidierung und Weiterentwicklung der formalen Entwurfsmethodik FOCUS zu leisten.

Dazu haben wir zunächst in den Kapiteln 2 und 3 eine Variante von FOCUS eingeführt, die auf der durchgehenden Verwendung von Agenten mit benannten Kanälen basiert und Agenten höherer Ordnung erlaubt. Wir haben Operatoren zur Spezifikation von Agenten und Agentennetzwerken angegeben und motiviert, warum Agenten höherer Ordnung in einer Reihe von Fällen, insbesondere bei der Modellierung von Betriebssystemstrukturen, hilfreich sind. Als theoretischen Aspekt haben wir die Begriffe Sicherheit und Lebendigkeit für Agentenspezifikationen und Agentenspezifikationen höherer Ordnung definiert und ihre Beziehung zu den bekannten Definitionen von Sicherheit und Lebendigkeit für Spurspezifikationen untersucht.

Für die Systementwicklung in FOCUS gab es bisher nur eine informelle Beschreibung. Zur Formalisierung haben wir in Kapitel 4 zunächst den Begriff der Systemstruktur eingeführt und definiert, was unter einer Spurspezifikation und einer funktionalen Spezifikation zu verstehen ist. Um den Entwurfsprozeß formal zu beschreiben, haben wir einen Verfeinerungsbegriff angegeben, der die Entwicklung auf der Spurebene, den Übergang zur funktionalen Ebene und die Entwicklung auf der funktionalen Ebene umfaßt. Auf jeder der beiden genannten Ebenen sind als Entwicklungsschritte eine Verfeinerung der Systemstruktur, eine Verfeinerung des Systemverhaltens, ein Aufspalten der Spezifikation und eine modulare Verfeinerung von Teilspezifikationen möglich. Wir haben syntaktische Regeln angegeben, die den Übergang zwischen den Ebenen der Spurspezifikation und der funktionalen Spezifikation erleichtern. Als Beispiel haben wir ein einfaches verteiltes Prozessorsystem in den genannten Formalismen spezifiziert und verfeinert.

Die Entwicklung von einem funktional spezifizierten Agenten zum ausführbaren Programm haben wir in Kapitel 5 untersucht. Die dort vorgestellte Methodik ist eine Erweiterung von FOCUS. Um die Entwicklung auf der funktionalen Ebene in eine Folge von kleinen und leicht handhabbaren Schritten zu unterteilen, haben wir die zustandsorientierten Spezifikationsschemata *Strat*, *RStrat* und *EStrat* definiert. Wir haben die Eigenschaften von Spezifikationen nach diesen Schemata untersucht und Verfeinerungsregeln für den Übergang zwischen den Schemata angegeben. Für das Schema *EStrat*, das einen explizit definierten Zustandsraum voraussetzt, haben wir Hinweise zur Konstruktion eines geeigneten Zustandsraumes und zur Validierung einer Spezifikation gegeben. Außerdem haben wir uns mit der Verfeinerung von expliziten Zustandsräumen beschäftigt. Ein Agent des Prozessorsystems von Kapitel 4 ist in den unterschiedlichen zustandsorientierten Formalismen als Beispiel spezifiziert worden.

In Kapitel 6 haben wir zunächst einige bekannte Zeitbegriffe zueinander in Beziehung gesetzt und dann einen einfachen Zeitbegriff vorgeschlagen und dessen Eigenschaften untersucht. Wir haben Techniken zur Spezifikation gezeiteter Agenten verglichen und das Schema *TStrat* angegeben, das auf den zustandsorientierten Spezifikationstechniken von Kapitel 5 beruht. Ein weiterer Agent des Prozessorsystems von Kapitel 4 wurde spezifiziert.

Schließlich haben wir in Kapitel 7 gezeigt, wie sich die zustandsorientierten Konzepte von Kapitel 5 auch zur Modellierung dynamischer Netzwerke einsetzen lassen. Wir haben ein einfaches Modell für solche Netze eingeführt und drei beispielhafte dynamische Systeme modelliert. Das funktionale Verhalten eines dynamischen Netzes wurde durch das Schema *NStrat* definiert. Wir haben eine Reihe von Erweiterungen des einfachen Netzformalismus vorgeschlagen. Um die dynamische Netzmodellierung in die allgemeine Entwurfsmethodik der Arbeit einzugliedern, haben wir Regeln zum Übergang von statischen zustandsorientierten Spezifikationen zu dynamischen Netzen und zur Verfeinerung innerhalb des dynamischen Netzformalismus angegeben.

Als besonders wichtig und lohnend für weiterführende Arbeiten erachten wir die praktische Umsetzung der in dieser Arbeit aufgezeigten Techniken. So ist es im Hinblick auf Kapitel 4 wünschenswert, den dort formalisierten Entwurfsprozeß mit Werkzeugunterstützung durchführen zu können. Ein Anfang wäre die Erstellung eines Werkzeugs zur Verwaltung der im Entwicklungsprozeß auftretenden Spezifikationen und Spezifikationsmengen. Für Systemstrukturen sollte ein graphischer Editor zur Verfügung stehen.

Für die Beschreibung von Agenten in einem zustandsorientierten Formalismus sollten die Möglichkeiten der Tabellennotation, die in dieser Arbeit nur beispielhaft in den Bildern 5.2, 5.3 und 7.5 dargestellt wurden, methodisch weiter ausgebaut werden. Möglichkeiten zur schematischen Modellierung häufig vorkommender Entwicklungsschritte, zum Beispiel das Einführen von Fehlerzuständen, sollten untersucht werden.

Auch hier ist eine Werkzeugunterstützung möglich und sinnvoll. Der erste Schritt dazu ist die Festlegung einer Tabellennotationssprache und ihrer Semantik.

Die in Kapitel 5 beschriebenen Entwicklungsschritte sind auf eine Implementierung hin gerichtet. Hier ist es erforderlich, eine geeignete Zielsprache zur Verfügung zu stellen. Die grundlegenden Anforderungen an eine solche Sprache sind in Abschnitt 5.6 dargelegt. Um eine effiziente Programmierung verteilter oder nebenläufiger Systeme zu ermöglichen, sollte eine derartige Sprache auch zur Definition dynamischer Agentennetze geeignet sein.

Die Techniken zur Modellierung dynamischer Netze von Kapitel 7 sollten an weiteren Beispielen erprobt werden. Insbesondere sind die in Abschnitt 7.3 dargestellten Erweiterungen zu untersuchen. Damit die angegebenen Techniken praktisch eingesetzt werden können, müssen Verfeinerungsregeln von dynamischen Modellierungen zu entsprechenden Konstrukten einer Implementierungssprache angegeben werden.

Insgesamt scheinen auf dem Gebiet der theoretischen Grundlagen des dargestellten Formalismus nur noch wenige wichtige Fragen offen. Auch die methodische Anwendung ist für kleine Fallstudien bekannt. Größere Fallstudien, insbesondere solche, in denen ein auf einem realen Rechner lauffähiges Programm entwickelt wird, sind bis jetzt nicht durchgeführt worden. Es ist zu erwarten, daß sich bei solchen Fallstudien auch ein Bedarf zur Fortbildung der Methodik zeigen wird. Auf den Gebieten der Implementierung und Werkzeugunterstützung existieren bisher nur Ansätze. Hier müßten noch größere Anstrengungen unternommen werden, um für die in dieser Arbeit dargestellten Techniken den Schritt in die Praxis zu ermöglichen.

Literaturverzeichnis

[AL88] M. Abadi, L. Lamport. The existence of refinement mappings. In: *3rd Symp. on Logic in Computer Science*, Seiten 165–175. IEEE, 1988.

[BA81] J.D. Brock, W.B. Ackermann. Scenarios: A model of nondeterministic computation. In: J. Diaz and I. Ramos, Hrsg., *Foundations of Programming Concepts*, Seiten 252–259. Springer Verlag, 1981. LNCS 107.

[BB90] J.C.M. Baeten, J.A. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing*, 1990.

[BD90] M. Broy, C. Dendorfer. Functional modelling of operating system structures by timed higher order stream processing functions. SFB-Bericht 342/22/90 A, Technische Universität München, November 1990. Vorversion von [BD92].

[BD92] M. Broy, C. Dendorfer. Modelling operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2(1):1–21, Januar 1992. Überarbeitete Fassung von [BD90].

[BDD+91] M. Broy, F. Dederichs, C. Dendorfer, R. Weber. Characterizing the behaviour of reactive systems by trace sets. SFB-Bericht 342/2/91 A, Technische Universität München, Februar 1991.

[BDD+92a] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber. The design of distributed systems — an introduction to FOCUS. SFB-Bericht 342/2/92 A, Technische Universität München, Januar 1992.

[BDD+92b] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber. Summary of case studies in FOCUS — a design method for distributed systems. SFB-Bericht 342/3/92 A, Technische Universität München, Januar 1992.

[BDS93] M. Broy, C. Dendorfer, K. Stølen. HOPSA — a high-level programming language for parallel computations. In: P.P. Spies, Hrsg., *Proc. Euro-ARCH*, Seiten 636–646. Springer Verlag, 1993.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen. The requirement and design specification language SPECTRUM — an informal introduction. Bericht TUM-I9311 (Teil I) und TUM-I9312 (Teil II), Technische Universität München, Mai 1993.

[BK84] J.A. Bergstra, J.W. Klop. Algebra of communicating processes. Report CS-R8421, Centre for Mathematics and Computer Science, Amsterdam, 1984.

[BKM+91] T. Bemmerl, C. Kasperbauer, M. Mairandres, B. Ries. Programming tools for distributed multiprocessor computing environments. SFB-Bericht 342/31/91 A, Technische Universität München, Oktober 1991.

[Bro86] M. Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, (45):1–61, 1986.

[Bro88a] M. Broy. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[Bro88b] M. Broy. Views of queues. *Science of Computer Programming*, 11:65–86, 1988.

[Bro93a] M. Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology*, 2(1):1–46, Januar 1993.

[Bro93b] M. Broy. *Informatik — Eine grundlegende Einführung in vier Teilen*, Teil II. Springer Verlag, 1993.

[Bro94a] M. Broy. Equations for describing dynamic nets of communicating systems. *Proc. 10th ADT/COMPASS Workshop*, 1994.

[Bro94b] M. Broy. (Inter-)action refinement: The easy way. 1994. Manuskript.

[BS94] M. Broy, K. Stølen. Specification and refinement of finite dataflow networks — a relational approach. 1994. Manuskript.

[CM88] K.M. Chandy, J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[DCC92] E. Downs, P. Clare, I. Coe. *Structured Systems Analysis and Design Method — Application and Context.* Prentice Hall, 2. Auflage, 1992.

[DDW93] F. Dederichs, C. Dendorfer, R. Weber. FOCUS: A formal design method for distributed systems. In: A. Bode, M. Dal Cin, Hrsg., *Parallel Computer Architectures*, Seiten 190–202. Springer Verlag, 1993. LNCS 732.

[Ded90] F. Dederichs. System and environment: The philosophers revisited. Bericht TUM-I9040, Technische Universität München, Oktober 1990.

[Ded92] F. Dederichs. Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen. SFB-Bericht 342/17/92 A, Technische Universität München, August 1992. Dissertation.

[Den90] C. Dendorfer. Ein hierarchischer Zeitbegriff für stromverarbeitende Funktionen, Juni 1990. Manuskript.

[Den91] C. Dendorfer. Funktionale Modellierung eines Postsystems. SFB-Bericht 342/28/91 A, Technische Universität München, Oktober 1991.

[Dil89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.

[DW89] F. Dederichs, R. Weber. Safety and liveness from a methodological point of view. Bericht MIP-8918, Universität Passau, 1989.

[DW92a] C. Dendorfer, R. Weber. Development and implementation of a communication protocol — an exercise in FOCUS. SFB-Bericht 342/4/92 A, Technische Universität München, März 1992. Erweiterte Fassung von [DW92b].

[DW92b] C. Dendorfer, R. Weber. From service specification to protocol entity implementation — an exercise in formal protocol development. In: R.J. Linn and M.Ü. Uyar, Hrsg., *Protocol Specification, Testing and Verification, XII*, Seiten 163–177. IFIP, North-Holland, 1992. Kurzfassung von [DW92a].

[DW93] C. Dendorfer, R. Weber. An informal introduction to the design method FOCUS. In: Hartmut König, Hrsg., *Formale Methoden für verteilte Systeme*, Seiten 9–20. K. G. Saur Verlag, 1993.

[Est88] *Estelle — A Formal Description Technique Based on an Extended State Transition Model*. 1988. ISO/IS 8807.

[Fuc94] M. Fuchs. Technologieabhängigkeit von Spezifikationen digitaler Hardware. SFB-Bericht 342/14/94 A, Technische Universität München, Juli 1994. Dissertation.

[Gri81] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer Verlag, 1981.

[Gro94] R. Grosu. An object-oriented method for concurrent modular specification, 1994. Dissertation.

[GS90] C.A. Gunther, D.S. Scott. Semantic domains. In: Jan van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Volume B*, Kapitel 12, Seiten 635–674. Elsevier Science Publishers, 1990.

[GSB95] R. Grosu, K. Stølen, M. Broy. A denotational model for mobile dataflow networks. 1995. Manuskript.

[GW90] R. van Glabbeek, P. Weijland. Branching time and abstraction in bisimulation semantics. SFB-Bericht 342/29/90 A, Technische Universität München, Dezember 1990.

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, Seiten 11–19, September 1990.

[Har87] D. Harel. StateCharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oktober 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hog89] D. Hogrefe. *Estelle, LOTOS und SDL: Standard-Spezifikationssprachen für verteilte Systeme*. Springer Verlag, 1989.

[Huß94] H. Hußmann. Formal foundations for SSADM, Juni 1994. Habilitationsschrift.

[Inf93] *Informatik Spektrum*, Band 16, Heft 5, Sonderheft "25 Jahre Software-Engineering". W. Brauer, Hrsg., Springer Verlag, Oktober 1993.

[Jac83] M.A. Jackson. *System Development*. Prentice Hall, 1983.

[Jon89] B. Jonsson. A fully abstract trace model for dataflow networks. In: *Proc. 16th Symposium on Principles of Programming Languages*, Seiten 155–165. ACM, 1989.

[Jon90] C. Jones. *Systematic Software Development using VDM*. Prentice Hall, 2. Auflage, 1990.

[JS89] S.B. Jones, A.F. Sinclair. Functional programming and operating systems. *The Computer Journal*, 32(2):162–174, 1989.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In: J.L. Rosenfeld, Hrsg., *Information Processing 74*, Seiten 471–475. IFIP, North Holland Publishing Company, 1974.

[Kin93] E. Kindler. Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick. SFB-Bericht 342/2/93 B, Technische Universität München, Dezember 1993.

[KM77] G. Kahn, D.B. MacQueen. Coroutines and networks of parallel processes. In: B. Gilchrist, Hrsg., *Information Processing 77*, Seiten 993–998. IFIP, North Holland Publishing Company, 1977.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2:125–143, März 1977.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, Seiten 190–222, April 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[LS87] J. Loeckx, K. Sieber. *The Foundations of Program Verification.* Wiley-Teubner series in computer science. 2. Auflage, 1987.

[MB87] A. Macro, J. Buxton. *The Craft of Software Engineering.* International Computer Science Series. Addison-Wesley, 1987.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer Verlag, 1980. LNCS 92.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[Mos90] P.D. Mosses. Denotational semantics. In: J. van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Volume B*, Kapitel 11, Seiten 577–631. Elsevier Science Publishers, 1990.

[MP91] Z. Manna, A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, Band 1 und 2. Springer Verlag, 1991.

[MPW89] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. LFCS Report 89-85 (Teil I) und 89-86 (Teil II), University of Edinburgh, 1989.

[NR69] P. Naur, B. Randell, Hrsg. *Software Engineering.* NATO Science Committee, Januar 1969.

[OG76] S. Owicki, D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[OL92] S. Owicki, L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, Juli 1992.

[Par82] D. Park. The "fairness" problem and nondeterministic computing networks. In: *Proceedings of the 4th Advanced Course on Theoretical Computer Science*. Mathematisch Centre Amsterdam, 1982.

[Plo83] G. Plotkin. Domains. Technical report, University of Edinburgh, 1983. Vorlesungsskript.

[PW89] D.L. Parnas, Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Queen's University at Kingston, Canada, Oktober 1989.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rei82] W. Reisig. *Petrinetze. Eine Einführung*. Springer Verlag, 1982.

[RR86] G.M. Reed, A.W. Roscoe. A timed model for communicating sequential processes. In: L. Kott, Hrsg., *Automata, Languages, and Programming*, Seiten 314–323. Springer Verlag, 1986. LNCS 226.

[San88] D. Sannella. A survey of formal software development methods. LFCS Report 88-56, University of Edinburgh, 1988.

[Sch87] F.B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Cornell University, Oktober 1987.

[Sch95] B. Schätz. Verschiedene Formalismen zur Spezifikation verteilter Systeme. Technische Universität München, 1995. Dissertation (Vorversion).

[SDL89] CCITT. *Functional Specification and Description Language (SDL)*. Blue Book Volume X. International Telecommunication Union, 1989.

[Spi87] J.M. Spivey. *The Z Notation — A Reference Manual*. Prentice Hall, 1987.

[Sto86] W. Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6:291–311, 1986.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tho89] B. Thomsen. A calculus of higher order communicating systems. In: *Proc. 16th Symposium on Principles of Programming Languages*, Seiten 143–154. ACM, 1989.

[Tur90] D. Turner. An approach to functional operating systems. In: D. Turner, Hrsg., *Research Topics in Functional Programming*, Seiten 199–217. Addison-Wesley, 1990.

[Web92] R. Weber. Eine Methodik für die formale Anforderungsspezifikation verteilter Systeme. SFB-Bericht 342/13/92 A, Technische Universität München, März 1992. Dissertation.

[Win90] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, Seiten 8–24, September 1990.

[Wir90] M. Wirsing. Algebraic specification. In: Jan van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science, Volume B*, Kapitel 13, Seiten 675–788. Elsevier Science Publishers, 1990.

Namens- und Symbolverzeichnis